

BROWN BELT



CODE NINJAS®

ROBOMANIA	3
FIND THE EXIT	21
CLOUD HOP	36
JUNGLE ESCAPE	49
NINJA RUN	66
EVIL FORTRESS OF DOCTOR WORM	79
CYBERFU - PART 1	104
SHAPE JAM	121
LABYRINTH	157
CYBERFU - PART 2	172
AMAZING NINJA WORLDS - PART 1	191
WORLD OF COLOR	218
AMAZING NINJA WORLDS - PART 2	236
AMAZING NINJA WORLDS - PART 3	267
SCAVENGER HUNT DELUXE	283
FOOD FRENZY PART 1	307
FOOD FRENZY PART 2	342

Activity 1

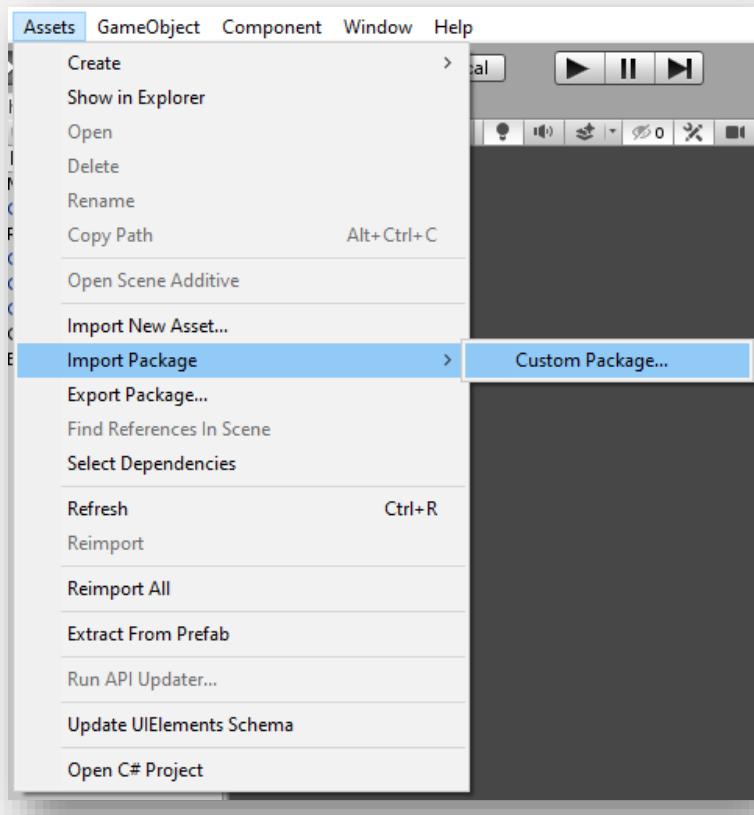
Robomania

Objective: The ninja will be able to write conditional statements to program a non-playable character's movement using the **rigidbody** component and the component's built-in functions and properties in a futuristic robot game.

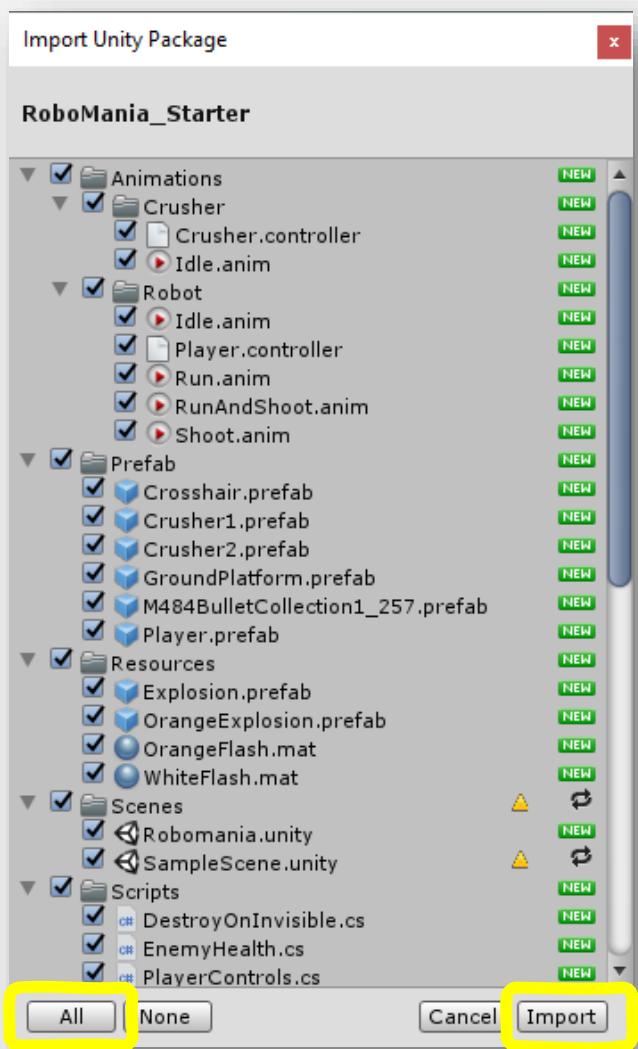
Our hero RoboMan must overcome two evil crushers to save the day!



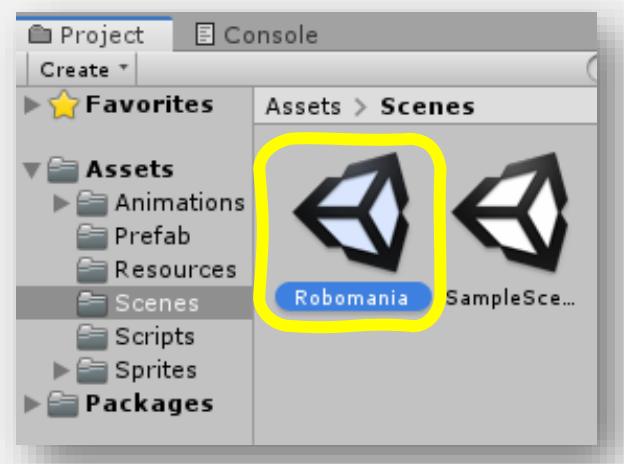
-
- 1** Start a new **Unity** Project and name it **YOUR INITIALS – Robomania**.
Select the **3D core** and place the project in the correct folder.
Click the **Create** button.
-
- 2** After Unity loads your new project, **import** the Robomania Starter Unity Package, named **Robomania_Ninja.unitypackage** by clicking on **Assets > Import Package > Custom Package**.



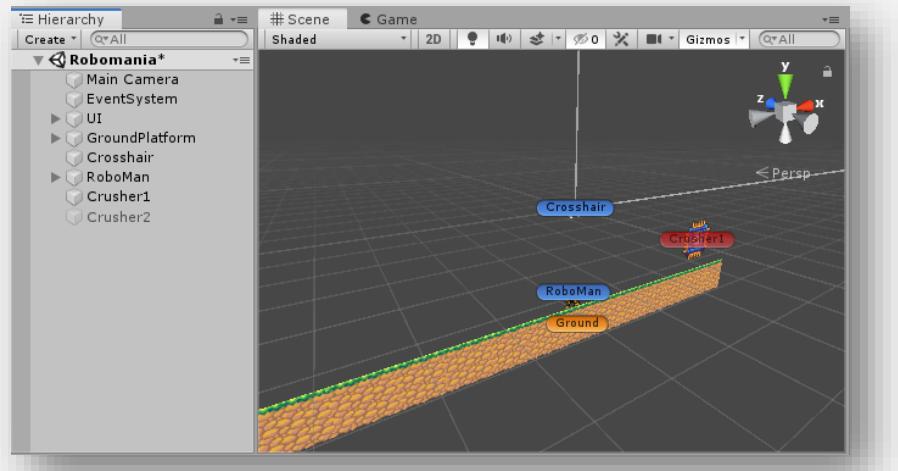
3 On the **Import Unity Package** window, click **All** and then **Import**.



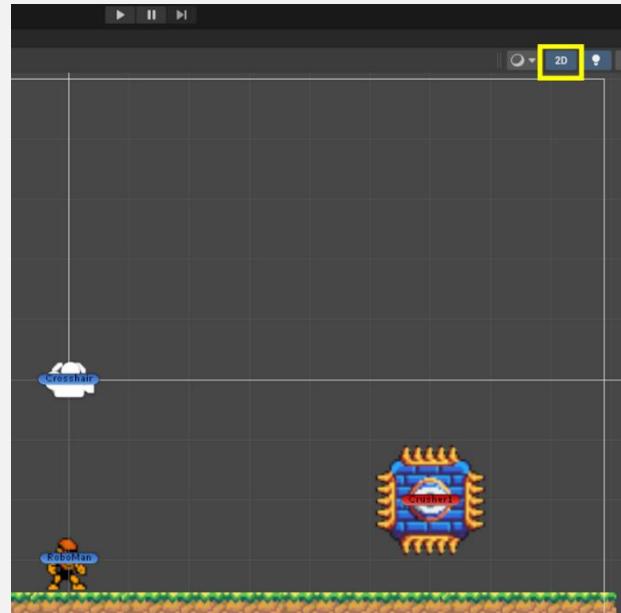
4 After it is done importing, double-click the **Robomania** scene. It will be located in the **Project** tab within the **Assets > Scenes** folder.



- 5** Opening **Robomania** will load the **Scene** with all our game objects. Make sure that you can see all the objects and the assets in the scene tab.



- 6** If the sprites look funny and out of place, click the **2D** button at the top of the Scene tab to align the view to the game.

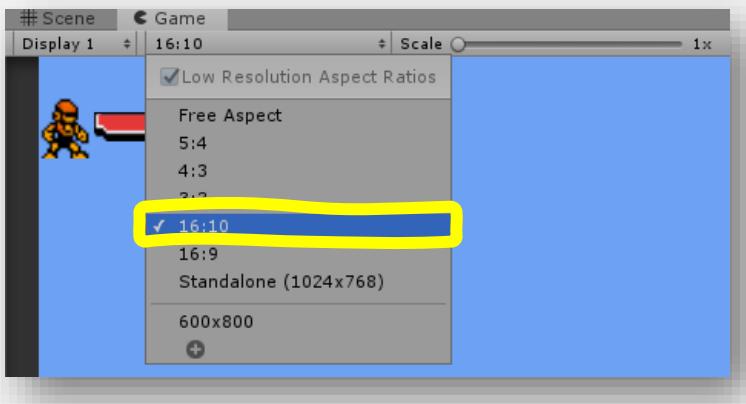


- 7** Click Unity's **play** button and do your first playtest of Robomania. You can move RoboMan with the **WASD** keys or the arrow keys, make RoboMan jump with the spacebar, and fire toward the cursor by clicking the left mouse button.

Try to destroy the crusher! Did you notice how the crusher just floats there? What happens when you try to use your ProtonBlaster to defeat the crusher? What happened when you run RoboMan into a crusher?

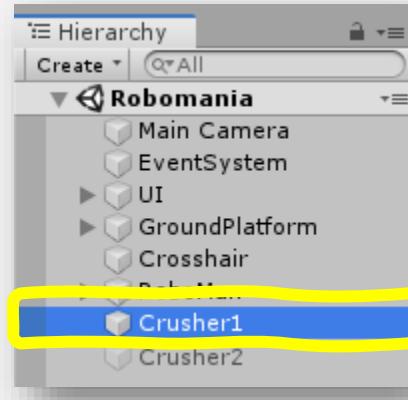
Right now, the crusher doesn't know how to do anything but blink!

- 8** If you tried to play and you couldn't see everything in the Game tab, set your resolution to **16:10** and make sure your **Scale** is at **1x**.



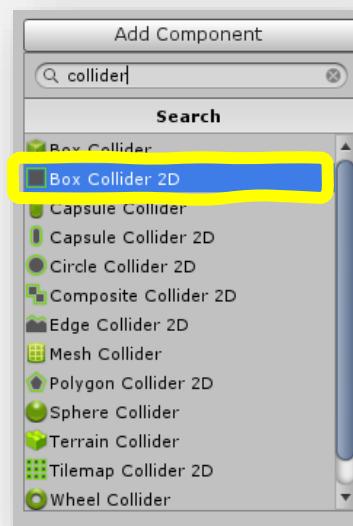
- 9** How do we get the crusher to respond to the proton blasts and **RoboMan**? Think back to other games you made where game objects needed to **collide** with other game objects.

We need to add a **collider** to the crusher for it to interact with other game objects! Click on **Crusher1** in the **Hierarchy** or the scene to open it in the Inspector.

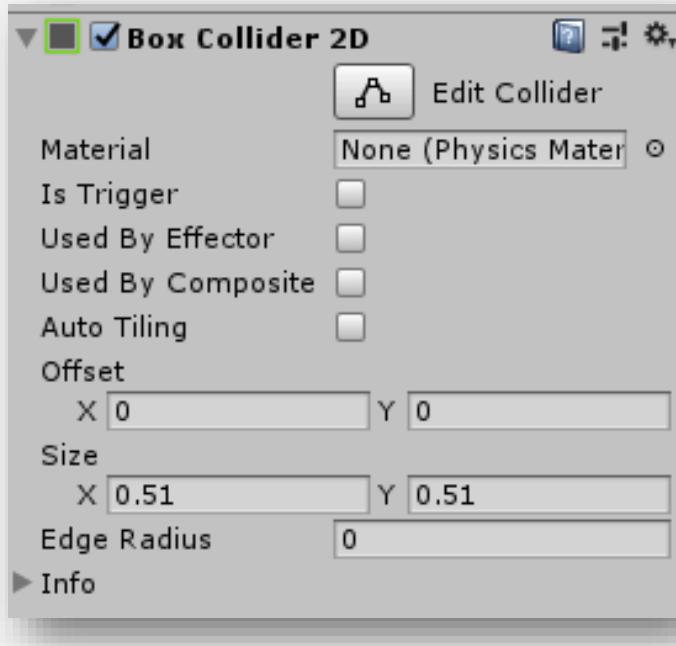


- 10** Click the **Add Component** button.

Search for "**collider**" and select **Box Collider 2D**.



- 11** The **Crusher1** game object has a **Box Collider 2D** component. We can leave the default settings as they are.



- 12** Play the game and see what adding a **collider** changed. Based on the changes, talk to your **Code Sensei** about the purpose of a **collider** and how it changes the way **game objects** interact with each other.

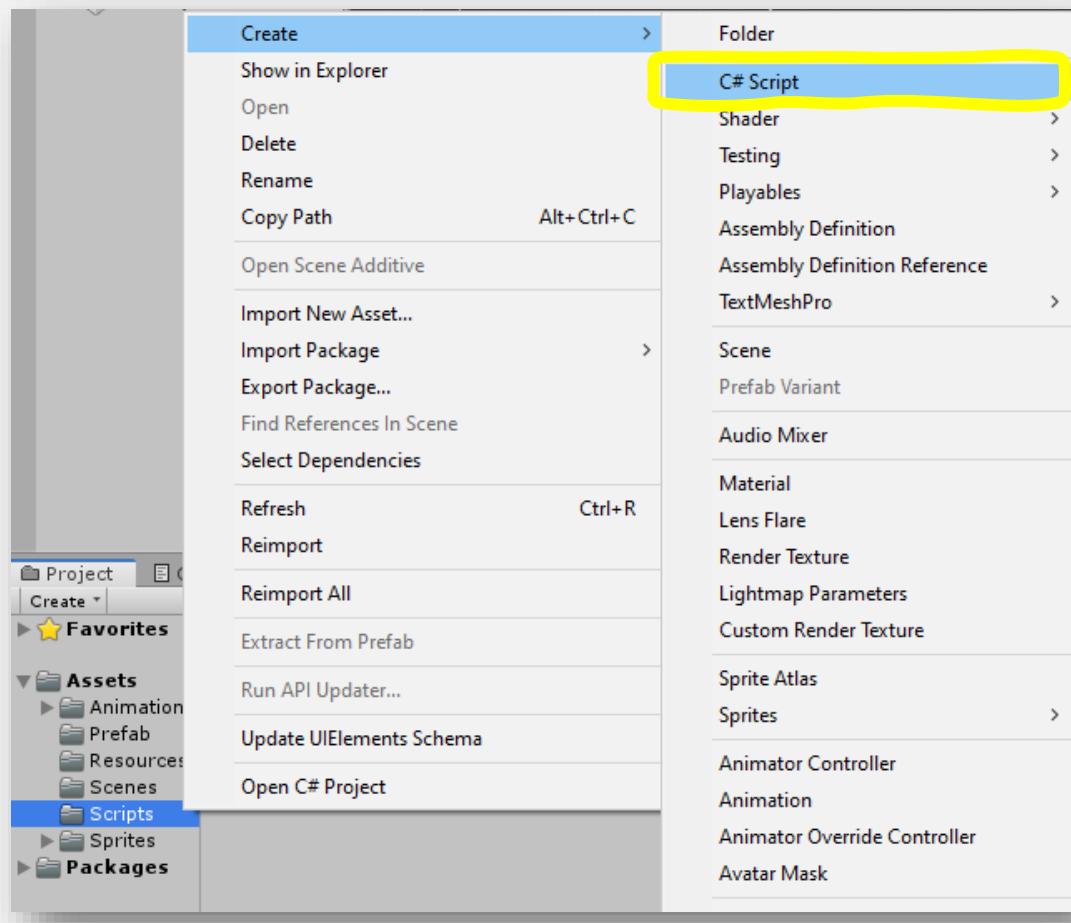
How can we make the game more challenging?

What if we made the crusher move?

What type of **component** do we need to add in order to **script** out an enemy's movement?

13 If we want to control how the crusher moves, we need to create a new script and provide it instructions on what to do.

Create a new script by opening the **Project** tab, right-clicking on the **Scripts** folder then selecting **Create > C# Script**.



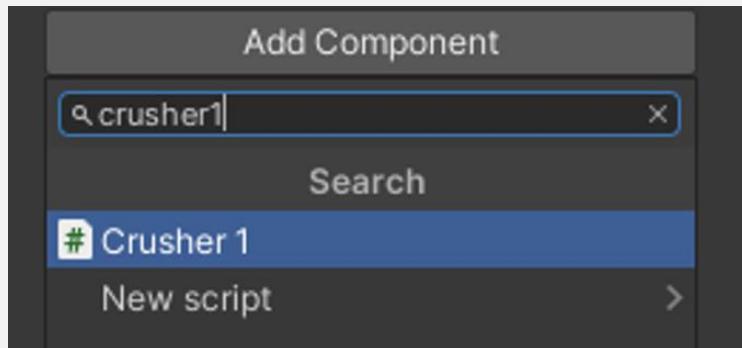
14 Name the script **Crusher1**.



15 In the **Hierarchy**, click on **Crusher1** to open it in the **Inspector**.

16 Click the  button.

Search for "Crusher1" to find the **Crusher1** script you just made.



17 Double-click the script to open it.

18 We do not need the `Start` or `Update` functions, so delete both of those so you only have **three using statements** and the `public class` line in the file.

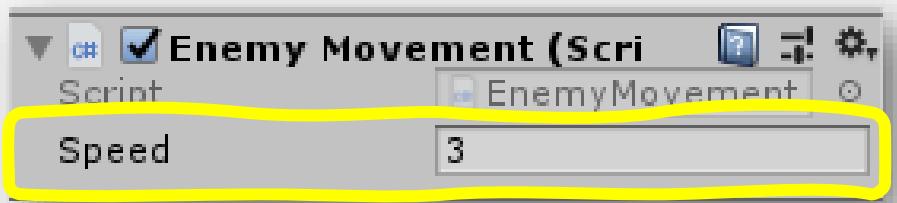
```
[-]using System.Collections;
 [-]using System.Collections.Generic;
 [-]using UnityEngine;

[-]public class Crusher1 : MonoBehaviour
 [ { ... } ]
```

- 19** We want to move the **crusher** at a constant speed, so on the line after the open curly brace type `public float speed;` to create a **variable** that we can change in the **Inspector** tab.

```
public class Crusher1 : MonoBehaviour
{
    public float speed;
```

- 20** In the **Inspector**, set the value of **Speed** to 3. You can change this later, but `3` is a good starting value.



- 21** After the `public float speed;` line, create a `FixedUpdate` function by typing `private void FixedUpdate() { }` making sure there is an empty line between the curly brackets.

```
private void FixedUpdate()
{
}
```

Why are we using `FixedUpdate`? `FixedUpdate` always runs the same amount of times every second (usually 50 times per second). `Regular Update` runs as fast as the computer can update, which sometimes will be 50, but it could be more if the computer is faster, or less if the computer is slower.

Anytime you are changing a game object's **position** or **physics**, you should be using Unity's `FixedUpdate` function to make sure that physics is the same on any computer that plays the game.

- 22** In the `FixedUpdate` function, we need to calculate the crusher's new position. We want the **crusher** to move left and right, but *not* up and down. Declare a new float variable named `newXPosition` and set it equal to the crusher's current x position and add the speed times the amount of time that has passed.

Type `float newXPosition = transform.position.x + speed * Time.fixedDeltaTime;`

```
private void FixedUpdate()
{
    float ... newXPosition = transform.position.x + speed * Time.fixedDeltaTime;
}
```

- 23** Since we do not want to move the **crusher** up and down, type `float newYPosition = transform.position.y;`

This line of code will make sure the crusher's new **y position** is equal to its old **y position**.

```
private void FixedUpdate()
{
    float newXPosition = transform.position.x + speed * Time.fixedDeltaTime;
    float ... newYPosition = transform.position.y;
}
```

- 24** We need to combine these two new **position** values into one **vector**. On the next line, type `Vector2 newPosition = new Vector2(newXPosition, newYPosition);` to declare a new `Vector2` variable named `newPosition`.

```
private void FixedUpdate()
{
    float newXPosition = transform.position.x + speed * Time.fixedDeltaTime
    float ... newYPosition = transform.position.y;
    Vector2 ... newPosition = new Vector2(newXPosition, newYPosition);
}
```

25 Save your script and play the game.

What happened? Nothing?

We calculated everything correctly, but what did we miss? Did we ever use our `newPosition` **vector**?

26 We need to set the crusher's position to be equal to the `newPosition` variable we just created. On the next line, type `transform.position = newPosition;`

```
private void FixedUpdate()
{
    float newXPosition = transform.position.x + speed * Time.fixedDeltaTime;
    float newYPosition = transform.position.y;
    Vector2 newPosition = new Vector2(newXPosition, newYPosition);
    transform.position = newPosition;
}
```

27 Save your script and play your game. Is the **crusher** moving? Is the **crusher** moving how you want it to move?

With your **Code Sensei**, discuss two different possible solutions on how to make your **crusher** move to the left toward **RoboMan**.

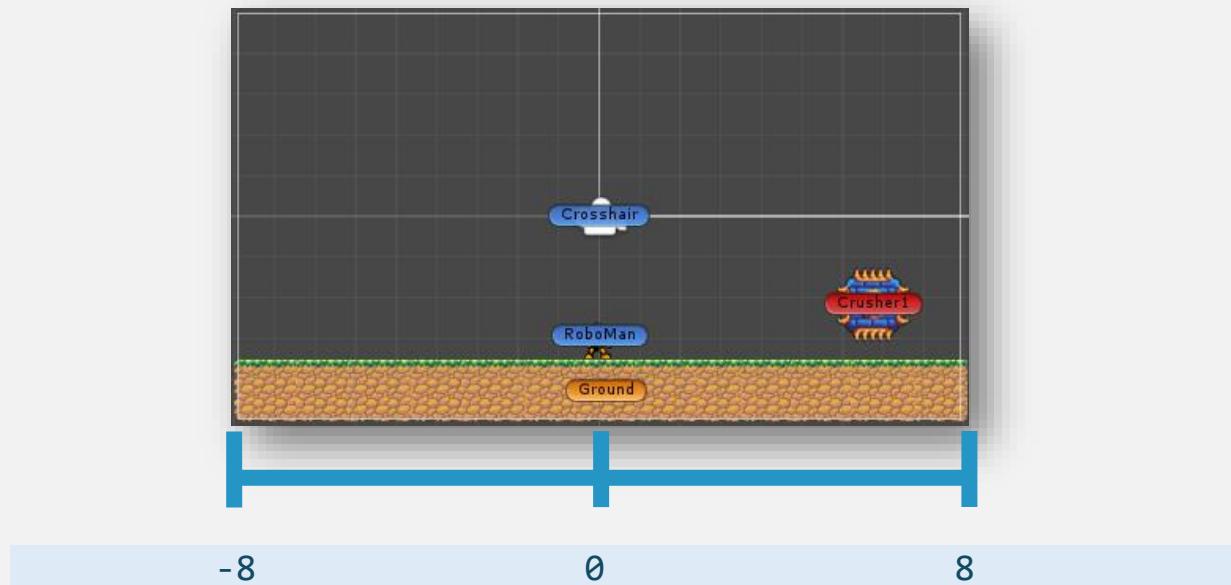
28 Since the **crusher** is moving to the right and we want it to move to the left, we will fix it by setting the **Speed** in the **Inspector** to be equal to `-3` instead of `3`.

29 Play your game and see if the problem is solved.

Now is a good time to think about what parts of your game you still need to change.

What happens when the crusher reaches the edge of the screen?
What happens to **RoboMan** when he hits the crusher?

- 30** Let's first make sure that the **crusher** doesn't fly off the screen! The **scene** is set up so the **camera** at the **center** of the screen has an **x coordinate** of **0**, the far left is at an **x coordinate** of **-8**, and the far right is at an **x coordinate** of **8**.



- 31** We need to make sure that if the crusher reaches the edge of the screen that it bounces back in the opposite direction. With your **Code Sensei**, discuss how we check a value and execute different pieces of code based on the result.

- 32** We need to first check to see if the crusher hits the left side of the scene. At the start of the **FixedUpdate** function before the **float newXPosition** line, add an **if statement** that checks to see if the position of the crusher is less than or equal to **-8**. Type **if (transform.position.x <= -8) { }** making sure to put an empty line between the curly brackets.

```
private void FixedUpdate()
{
    if (transform.position.x <= -8)
    {
        ...
    }

    float newXPosition = transform.position.x
    float newYPosition = transform.position.y
    Vector2 newPosition = new Vector2(newXPos
...
```

33 Whenever the **crusher** hits the left side of the scene, the code inside this if statement will execute. What do we need to do to the **speed** of the **crusher** to get it to move in the opposite **direction**?

We need to change a **negative speed** to a **positive speed**! We can flip the sign of the **speed** by simply **multiplying** the **speed** by **-1**.

Type `speed = speed * -1;` inside the **if statement** to **flip** the sign when the **crusher** reaches the edge of the scene.

```
if (transform.position.x <= -8)
{
    speed *= -1;
}
```

34 Play your game and let the **crusher** bounce across the scene.

Is everything working like you expected? Did your crusher bounce off the left side of the scene? What about the right side?

35 We need to also **flip** the sign of the **speed** if the **crusher** reaches the right side of the scene! Since we are already flipping the speed in one if statement, lets use an **or** condition to also flip it we reach the right side!

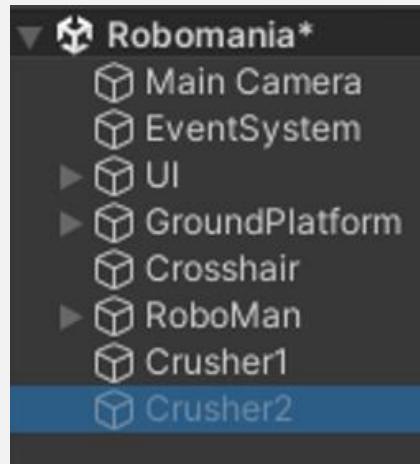
Type `|| transform.position.x >= 8` inside the condition for the if statement.

```
if (transform.position.x <= -8 || transform.position.x >= 8)
{
    speed *= -1;
}
```

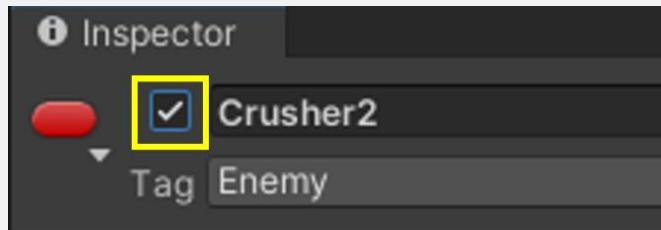
36 Play your game. Is the crusher behaving how you expect it to? It should be bouncing off the walls and changing direction!

That was a lot of work! Let's take a look at how Unity has some tools to save us some time, using the powerful Rigidbodies!

37 In the **Hierarchy**, click on **Crusher2** to show it in the **Inspector**.



38 In the **Inspector**, enable **Crusher2** by checking the box to the left of its name.

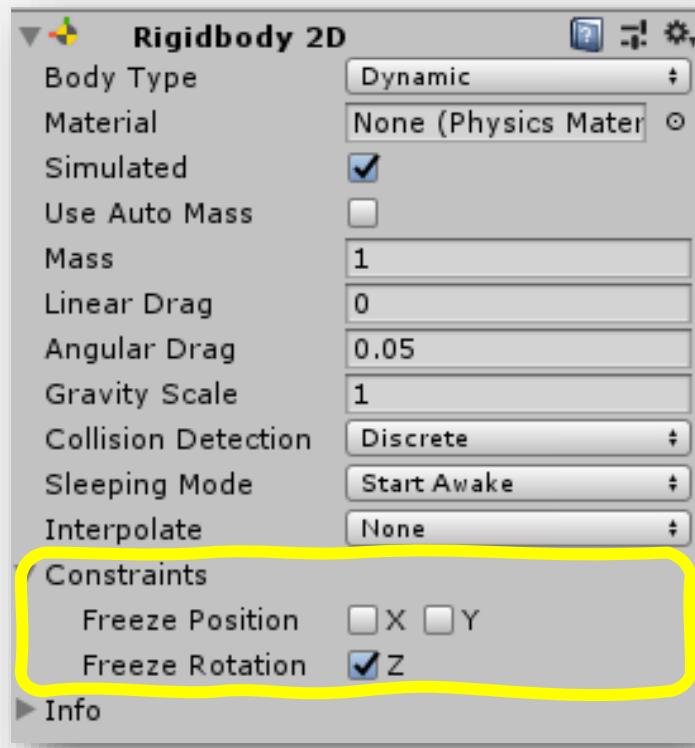


39 This **crusher** is now in the game, but it does not have a **BoxCollider2D** yet. Click **Add Component** to add in the component.

40 We also want to add in the Rigidbody component. Click on **Add Component** and search for "rigid" and add a **Rigidbody 2D component** to **Crusher2**.

41 The **Crusher2** game object now has a **Rigidbody 2D** component. The only change we need to make is to **freeze rotation** on the **z axis**.

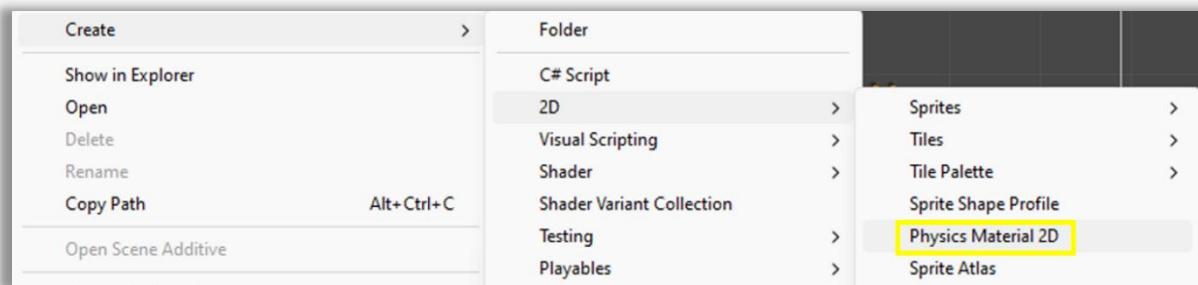
Open the **Constraints** dropdown and click the **Z checkbox** next to **Freeze Rotation**.



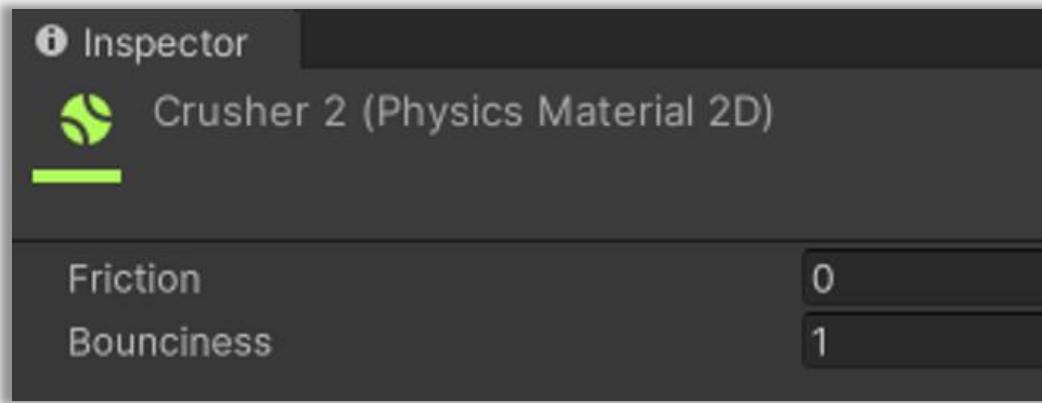
42 Play your game and experiment! How does Crusher2 behave? Does it bounce around like Crusher1?

43 In order to tell Unity how a rigidbody object should behave, we need to create a physics material. This has some properties to determine how bouncy an object is, and how much friction applies to it.

44 Right click in the Assets folder and create a new physics material by clicking in **Create > 2D > Physics Material 2D**. Name this material "Crusher2"



- 45** Select the physics material. In the Inspector, set the friction to 0 and bounciness to 1.



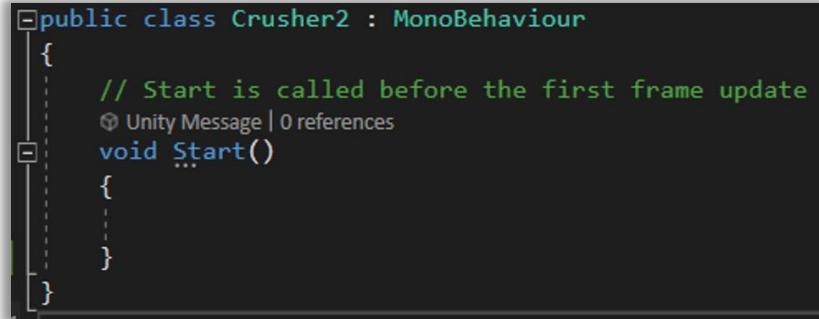
- 46** Drag the physics material onto Crusher2 in the Hierarchy.

- 47** Play your game! Do you notice the difference between the two crushers?

- 48** Let's give Crusher2 a bit of a push. Create a script called "**Crusher2**" in the scripts folder.



- 49** Open the script and remove the **Update** method.



- 50** Just like Crusher1, create a speed **float** variable, but set this one at 5.

```
public float speed = 5f;
```

- 51** In the `Start` method, get the `rigidbody` component using `GetComponent`, then we add force using the `AddForce` `rigidbody` method. We will add force to the right with a mode of `Impulse`. `Impulse` means the force is applied right away instead of over time.

```
public class Crusher2 : MonoBehaviour
{
    public float speed = 5f;
    // Start is called before the first frame update
    void Start()
    {
        GetComponent<Rigidbody2D>().AddForce(Vector2.right * speed, ForceMode2D.Impulse);
    }
}
```

- 52** In Unity, add the `Crusher2` script to the GameObject.

- 53** Play your game. `Crusher2` should be bouncing all around the screen!

Tell your sensei about when you think we might manually move objects like `Crusher1`, and when we might use `rigidbody` like `Crusher2`?

Prove Yourself

Use the Force!

Convert Crusher1 to use Rigidbody physics as well. You will have to:

- Add Rigidbody2D component
- Freeze the Z rotation
- Add the physics material with 0 friction and 1 bounce
- Modify Crusher1's script to apply force to the left at the start of the game
- Raise Crusher1 up in the game off the ground more so it has room to bounce

Activity 2

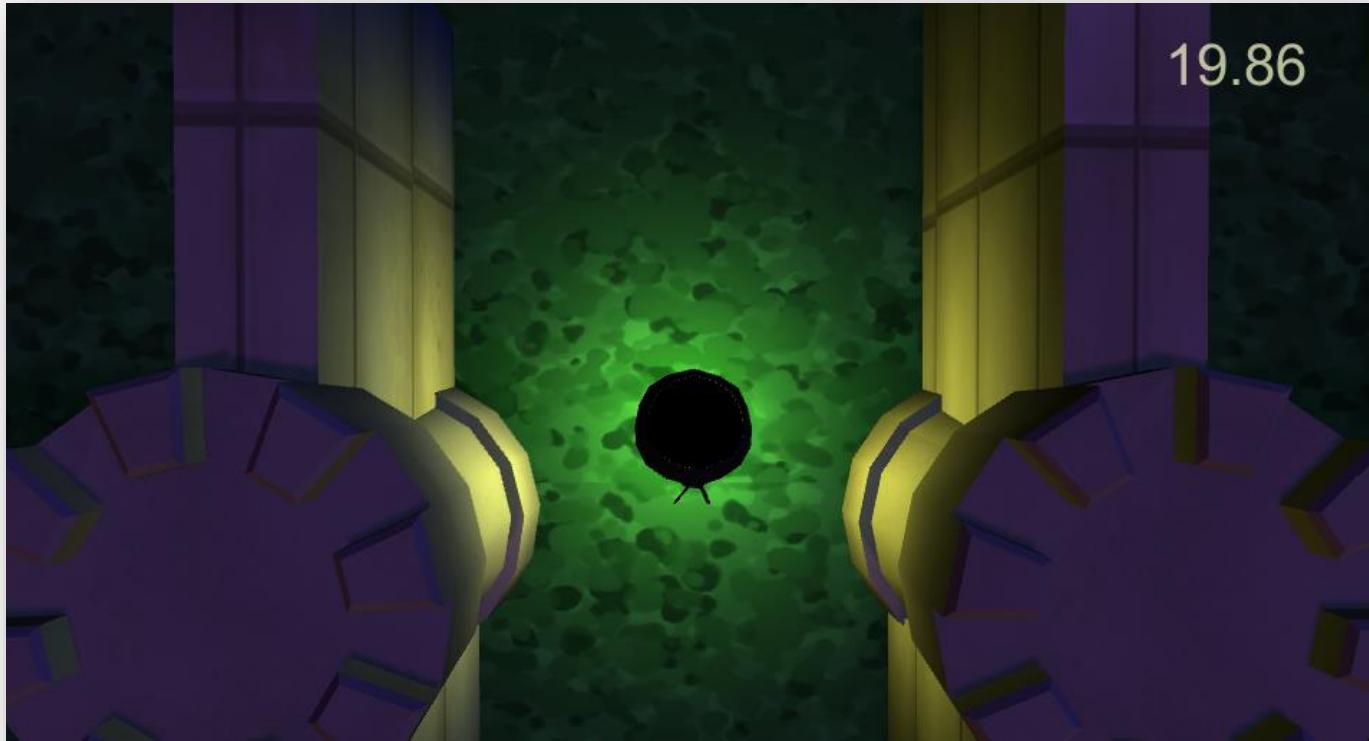
Find the Exit

In this game, you will be able to add movement to a character by *translating* (or moving) the object at a fixed rate of time.

Your mission: Escape the maze in the least amount of time possible. If you press **Play**, you'll discover you are trapped! The player character, Codey, can't move yet because there is no movement code.

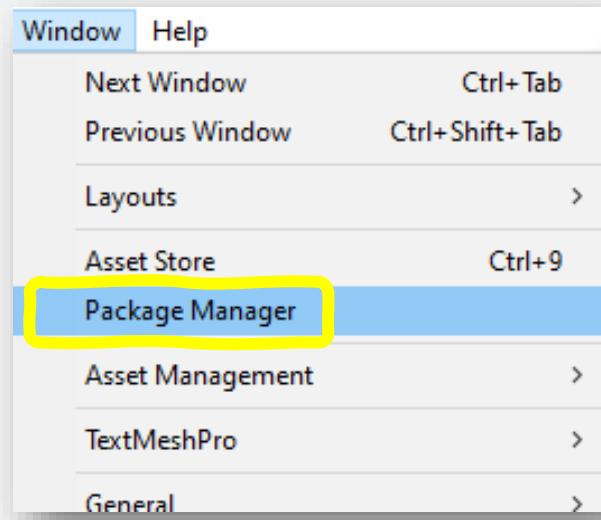
Code your player character's movement. This is a key step in making your game because it lets the player character, Codey, move around the game, advance through challenges and reach their goal!

Now that you know your mission, let's get moving!

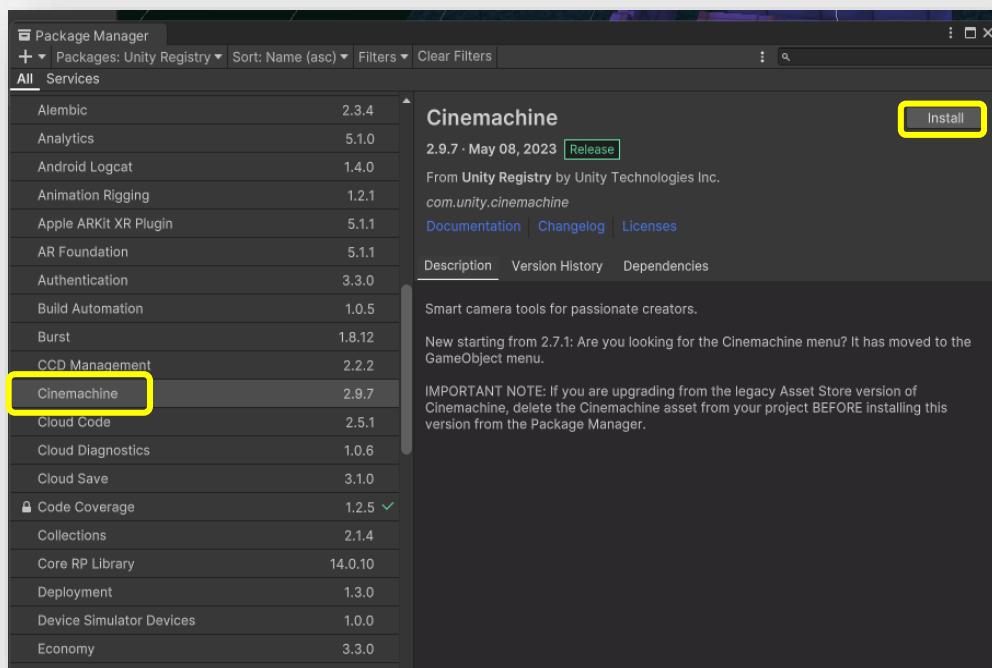


-
- 1** Start a new Unity Project and name it **YOUR INITIALS - Find the Exit**.
Select **3D core**.
- 2** We will be using Cinemachine to control our camera. This will help us create awesome camera shots and angles for our game. Go ahead and open **Window > Package Manager**.

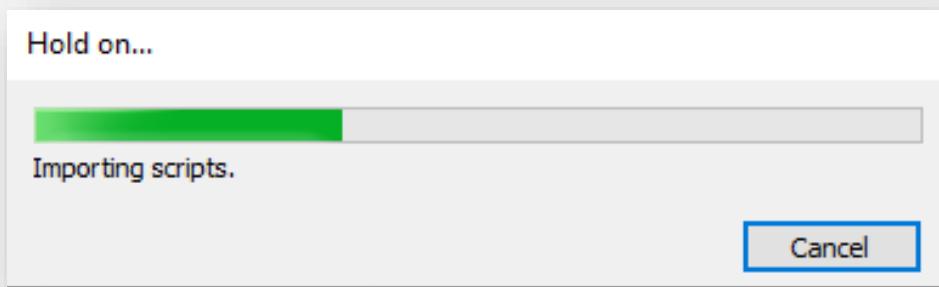
Be patient if it doesn't open right away. It might take a minute to load.



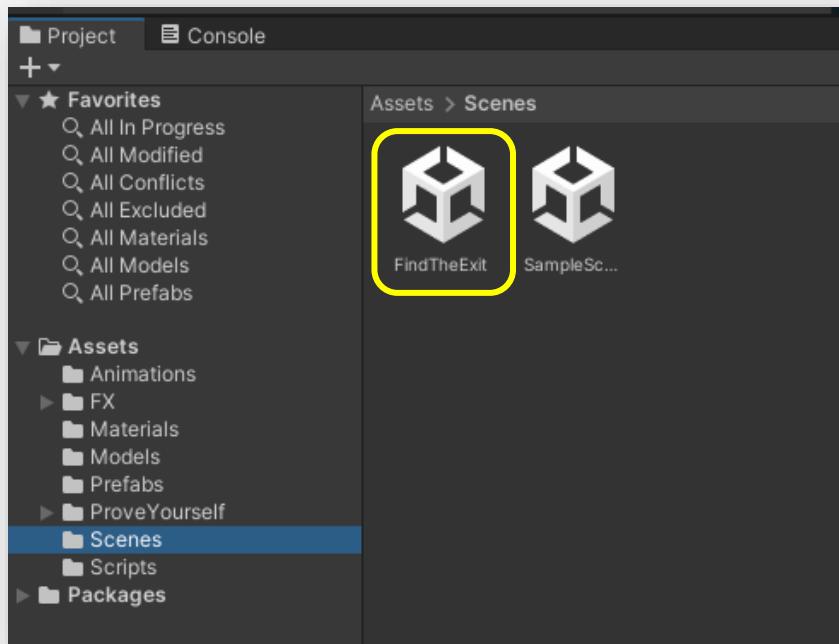
-
- 3** Switch to the Unity Registry, find **Cinemachine** and click **Install** in the top right of the window.



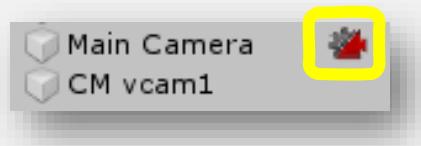
- 4** We've created a starter pack to give you a head start! To use it, import the A1 – **Find the Exit.unitypackage** by going to **Assets > Import Package > Custom Package> All > Import**.



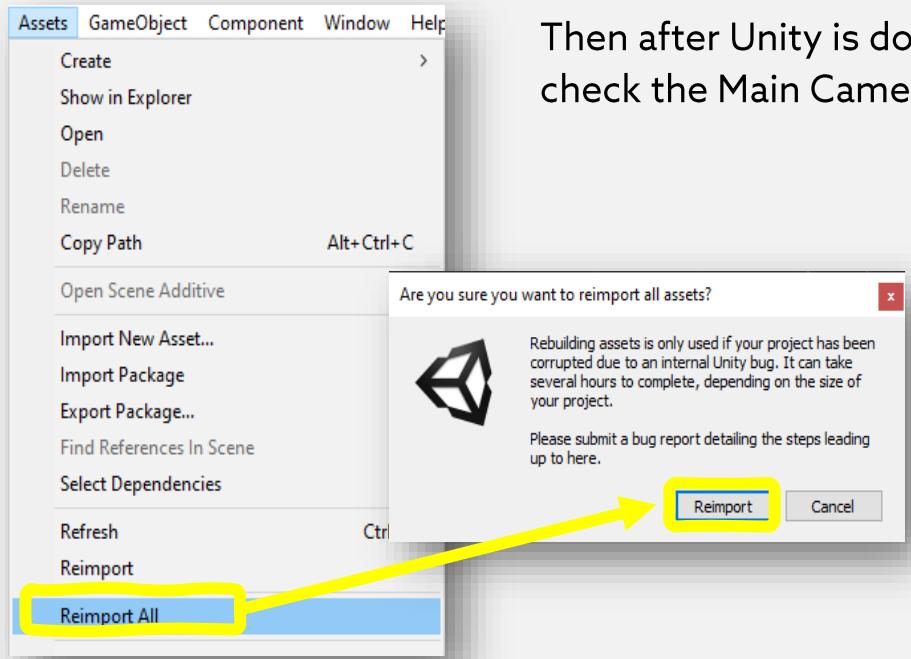
- 5** To open the starter package, double-click on the **FindTheExit** scene. You can find this in the **Project** tab under **Assets > Scenes > FindTheExit**.



- 6** We should check that Cinemachine is working like it is meant to.
In the **Hierarchy** tab, you should see a list of all the game objects in the scene. If you see a little grey and red icon attached to the Main Camera (shown in the picture), Cinemachine is working!



If you do not see it, go into the **Assets** tab and press **Reimport All**.



Then after Unity is done importing, check the Main Camera again.

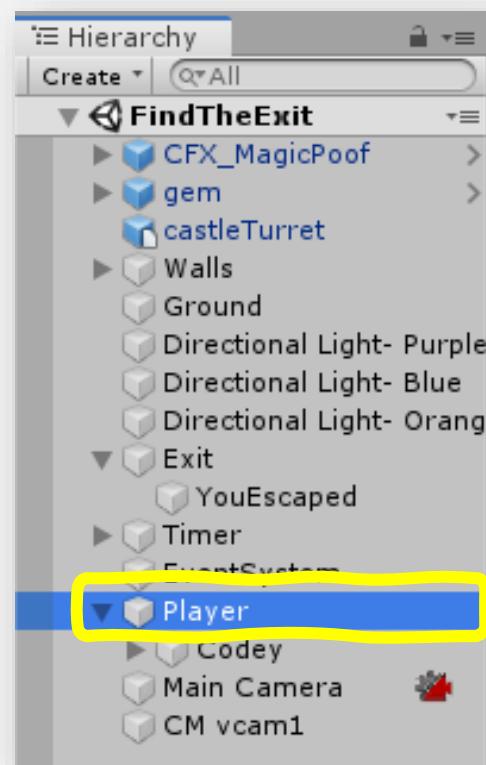


Codey will be the main character in several Brown Belt games! You will learn how to make him move, jump, and interact with the world around him.

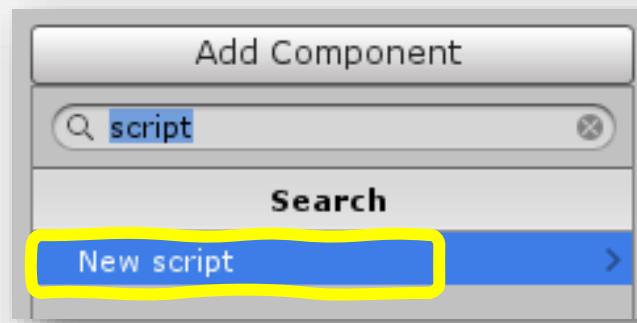
- 7** Now you can start creating the script to give Codey the ability to move in the game!

To get started, go to the **Hierarchy** and click the **Player** game object to open it in the **Inspector** tab.

The **Inspector** is where you can see all of the components and properties of a single game object in your scene.



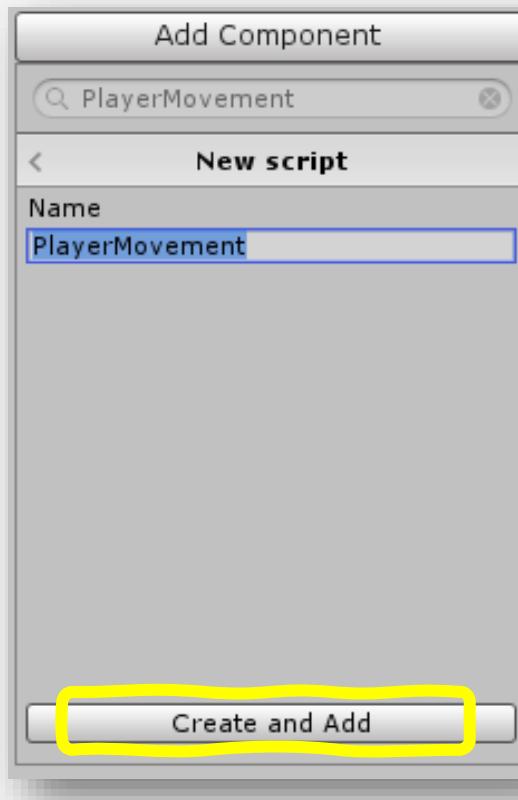
- 8** In the Inspector, click **Add Component** and search for "script". Click on **New Script**.



- 9** Name the script “*PlayerMovement*” and press **Enter** or click **Create and Add**.

In the **Project** tab, move the new **PlayerMovement** script to the **Scripts** folder to keep things organized.

Staying organized is important so you can find what you need when you need it.



- 10** After you have saved and moved the script to the correct folder, double-click on the **PlayerMovement** script in the **Inspector** to open the file in Visual Studio. This is where you’ll add code to get Codey in motion!



- 11** First, we need to set a speed for Codey. On a new line before `void Start()`, type `public float speed = 5;` to initialize Codey’s speed. A `public` variable can be changed in the Inspector. We want the value of speed to be a `float` because that means we can have decimal values and not just whole numbers.

```
public class PlayerMovement : MonoBehaviour
{
    public float speed = 5;
    void Start()
    {
    }
}
```

12 In the `Update` function, we want to add one line of code to move our ninja, Codey.

To do this, let's try using Unity's `Translate` function on Codey's `transform`. Unity's `Translate` function will let us change an object's position. In the `void Update()` function, type `transform.Translate(Vector3.forward * speed * Time.deltaTime);` to tell Codey to move forward at our set speed of 5 at a constant rate of time.

```
void Update()
{
    transform.Translate(Vector3.forward * speed * Time.deltaTime);
}
```

All we do when we *Translate* an object in Unity or in the real world is move it from one location to another.

Vector 3

A `Vector3` is a special object that contains three numbers, one for each of the directions in 3D space. We are using `Vector3.forward` to ask Unity to give us a vector that points forward.

Time.deltaTime

`Time.deltaTime` is a fancy way of asking Unity to give the value for how much time has passed since the last time the game loop updated. We use it to make sure any movements and other calculations are not done too slowly or too quickly. We want them to be done exactly right each time!

13 Save your code and press Play.

Codey should translate forward at a speed of 5. But what about at higher speeds? Try changing your speed variable and see what happens at high speeds.



Something seems wrong! Codey can move through walls at high speeds! This is because `transform.Translate` doesn't take into account any walls if the distance it is moving is on the other side of the wall. Instead, let's try using Rigidbody and Velocity!

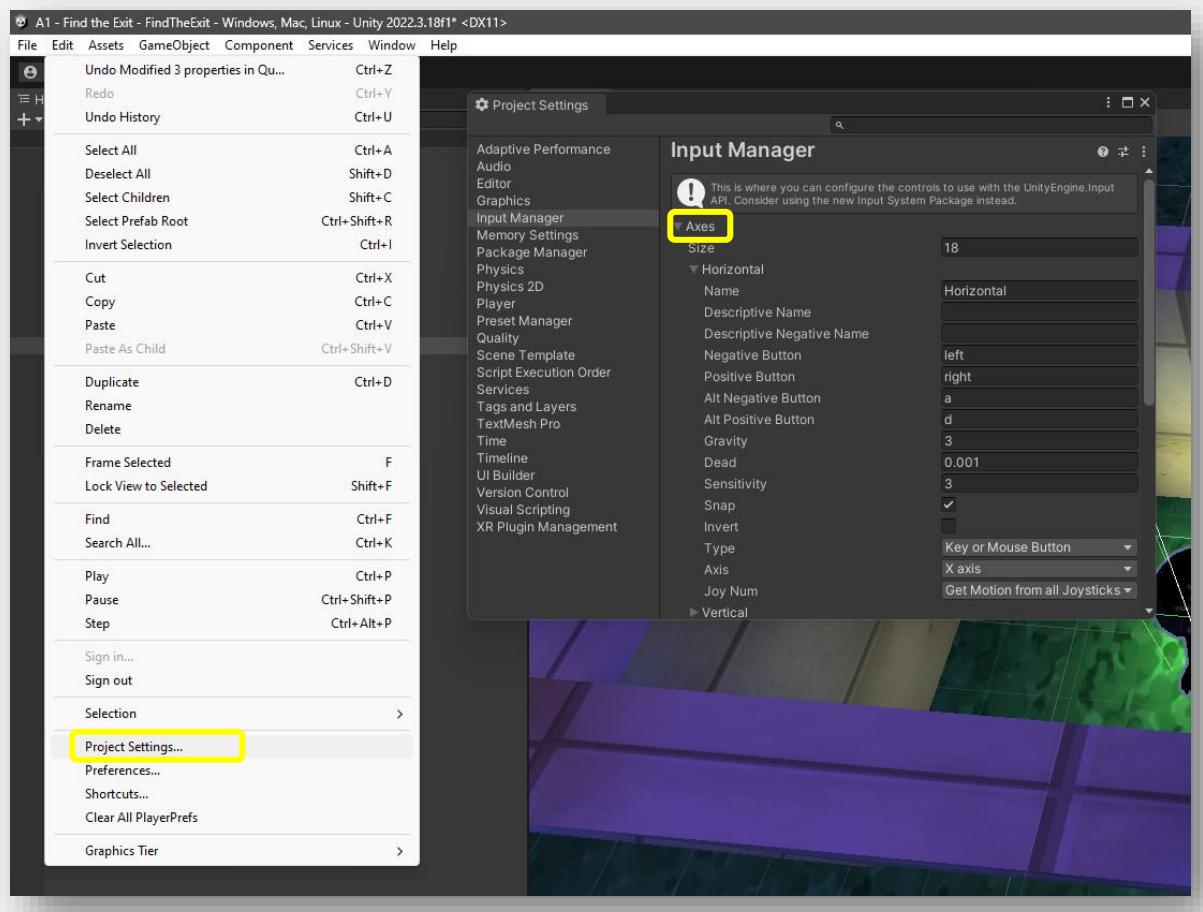
14 Comment out your previous line so that it will not run.

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
```

15 Get the Rigidbody component from Codey, and then we will set the velocity to the forward direction at the speed we set. Try out different speeds now and see the difference! Next, let's get Codey moving with arrow keys!

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
    GetComponent<Rigidbody>().velocity = Vector3.forward * speed;
```

16 Let's first see how Unity interprets user input. Go to **Edit > Project Settings > Input Manager**. Expand the **Axes** settings by clicking on the drop-down arrow next to it. This **Project Settings** panel will show a lot of behind-the-scenes Unity settings. For Horizontal, notice the **right** arrow and **d** are listed as *Positive Button*, meaning it will output a **1**. While the **left** arrow and **a** are listed as *Negative Button*, meaning it will output a **-1**. Close the **Project Settings** window.



- 17** Go back to your **PlayerMovement** script. After the `GetComponent< Rigidbody >()` line, log the value of the horizontal axis by typing `Debug.Log(Input.GetAxisRaw("Horizontal"));` in the **Update** function.

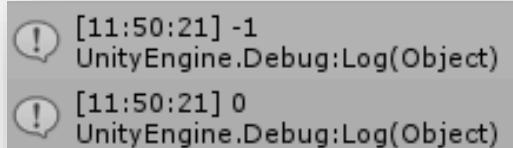
```
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
    GetComponent< Rigidbody >().velocity = Vector3.forward * speed;
    Debug.Log(Input.GetAxisRaw("Horizontal"));
}
```

On every frame, Unity will receive the user input, and get the value for the horizontal axis. With Debug, we are then asking Unity to log that value into the console.

- 18** Before you play the game, what number do you think will be in the log when you press the `a` key? The `d` key?

What about the left and right arrows? What will happen if you don't press any keys? Write your answers down somewhere so you can check to see if you were right!

- 19** Play your game and look at the console. It will display a `0` if there is no player input. Now press the left arrow or `a`; the console should have a new log saying `-1`. Next, test out the right arrow and `d`. Did you get it right?



```
[11:50:21] -1
UnityEngine.Debug:Log(Object)
[11:50:21] 0
UnityEngine.Debug:Log(Object)
```

Before moving on to the next step, think about what would happen if you wrote `Vertical` in your script instead of `Horizontal`? Test out your guess! Once you are done, you can delete the `Debug.Log` line from the **Update** function.

20 We now understand how to read player input to get `-1` or `1` depending on which key the user is pressing, but how can we connect this to the movement of Codey?

Remember, we are using `Vector3.Forward` to move our character. This will make Codey always move forward and ignore the user's input. Before we change the `Rigidbody`'s `velocity`, let's make a new `Vector3` based on user inputs!

21 Above the `GetComponent< Rigidbody >()` line, declare a new `float` variable named `horizontal` to store the value of the user input's horizontal axis by typing:

```
float horizontal = Input.GetAxisRaw("Horizontal");
```

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
    float horizontal = Input.GetAxisRaw("Horizontal");
    GetComponent< Rigidbody >().velocity = Vector3.forward * speed;
}
```

22 On the next line after `float horizontal`, write a similar piece of code but for the `Vertical` axis. Don't forget to check your work!

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
    float horizontal = Input.GetAxisRaw("Horizontal");
    float vertical = Input.GetAxisRaw("Vertical");
    GetComponent< Rigidbody >().velocity = Vector3.forward * speed;
}
```

23 Play your game and see what happens. Codey still always moves in a straight line! This is because while we are getting the user input, we aren't using it to move Codey!

- 24** On the next line, create a new `Vector3` destination with three parameters of `horizontal`, `0`, and `vertical` by typing `Vector3 destination = new Vector3(horizontal, 0, vertical);`.

This will create a new vector that will change based on the user input! Our `horizontal` variable is in the `x` value spot to move the character left and right, and the `vertical` variable is in the `z` value spot to move the character forward and backward in the game world. The `y`-axis would be used for moving the character up and down above the game world, like for jumping or falling. If you want to see what this vector looks like, you can `Debug.Log` it to the console.

```
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
    float horizontal = Input.GetAxisRaw("Horizontal");
    float vertical = Input.GetAxisRaw("Vertical");
    Vector3 destination = new Vector3(horizontal, 0, vertical);
    GetComponent<Rigidbody>().velocity = Vector3.forward * speed;
}
```

Parameter

A *parameter*, or *argument*, is a value or variable that we give to a function by placing it inside parentheses.

- 25** Codey will still ignore user input because we haven't told Codey to move based on the destination vector we just made. In the final line of `Update`, change `Vector3.forward` to `destination`. Save and test out your game!

```
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
    float horizontal = Input.GetAxisRaw("Horizontal");
    float vertical = Input.GetAxisRaw("Vertical");
    Vector3 destination = new Vector3(horizontal, 0, vertical);
    GetComponent<Rigidbody>().velocity = destination * speed;
}
```

26 In the `Update` function, change `Input.GetAxisRaw` to `Input.GetAxis`.
Now play the game.

Did you notice a change in how Codey moves? If you didn't, that's okay! `Input.GetAxisRaw` is like an on/off switch. Either the user is pressing a key (value `1`) or the user is not pressing a key (value `0`). `Input.GetAxis` will behave slightly differently.

If a user is not pressing a key, the value will always be `0`. As soon as a user presses a key, the value will start slowly increasing from `0` to `1`.

This will mean that Codey will slowly speed up until the `Input.GetAxis` value reaches `1`.

While it might not make a big difference for all games and player movement code, sometimes this very small difference can make a big impact on the enjoyment of someone playing your game!

This is your game, you decide which way you like better:

`Input.GetAxisRaw` or `Input.GetAxis`!

```
void Update()
{
    //transform.Translate(Vector3.forward * speed * Time.deltaTime);
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");
    Vector3 destination = new Vector3(horizontal, 0, vertical);
    GetComponent<Rigidbody>().velocity = destination * speed;
}
```

27 Now that you have Codey responding to the user's input, you can change the `destination` vector!

You can try switching the places of `horizontal` and `vertical`, making one variable negative, changing the `y` value to be `5` instead of `0`, or anything else you want to try.

By changing these values and playing the game to see what happens, you can gain a better understanding of what `Input` and Rigidbody's velocity are doing to your ninja.

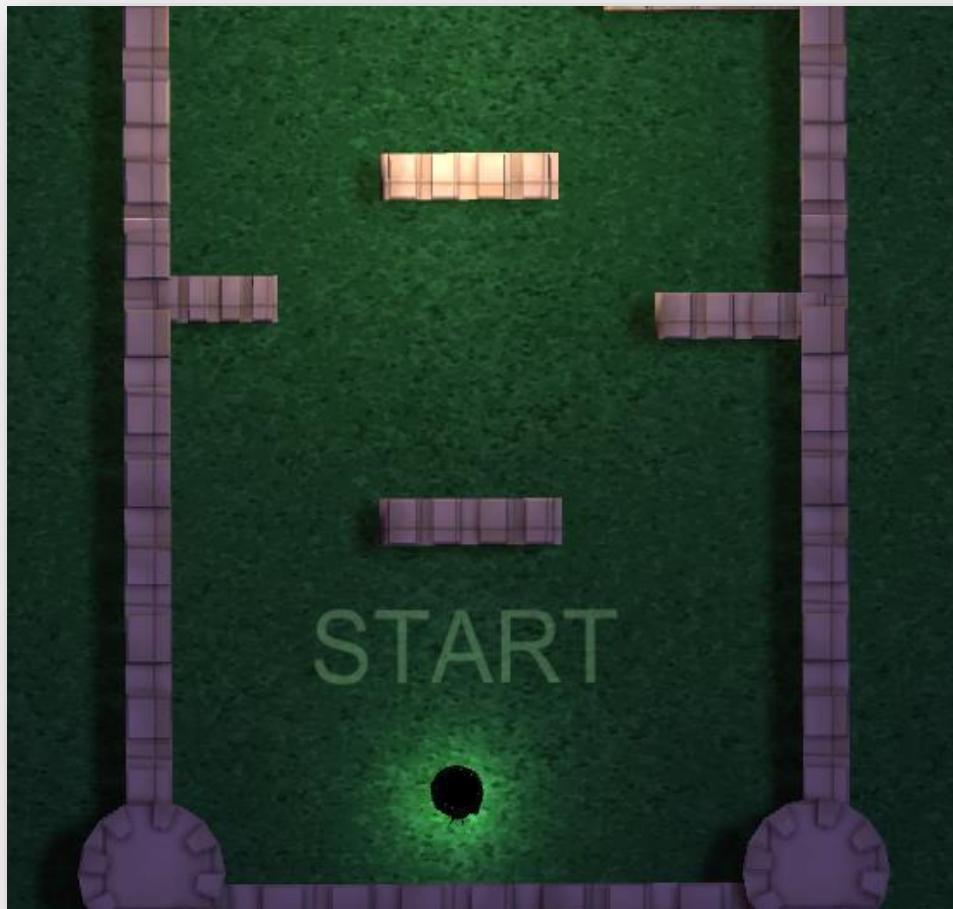
Prove Yourself

Get Started

- Look in your **Projects** tab under the **Prove Yourself** folder and double-click the scene named PY to open it.
- All the assets are ready, you just need to create the player movement script. Be careful though - if Codey hits a wall, he gets sent back to start!

Task

In this Prove Yourself, Find the Exit becomes Reach the Exit as you challenge your understanding of `Rigidbody` and `Input.GetAxisRaw()` or `Input.GetAxis()` to create a different type of player movement. Use your knowledge of how these functions work separately and then together to create a player movement in which Codey constantly moves forward and the user can only move him left and right.

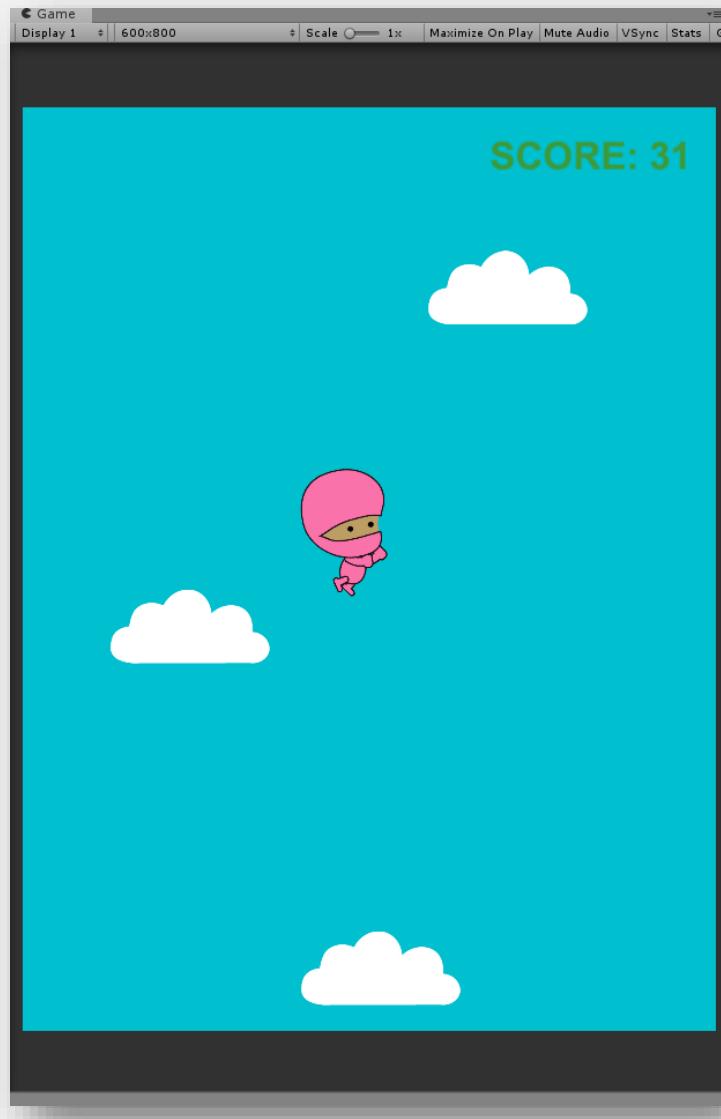


Activity 3

Cloud Hop

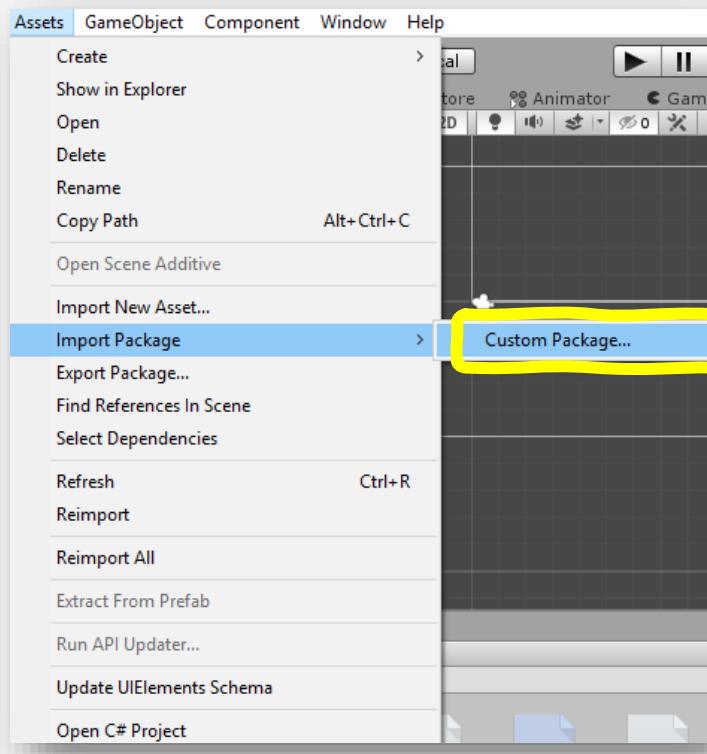
You will be able to make a player jump using `Input.GetButton` to check for input, `AddForce` to simulate the jump, and `velocity` to check if the player is grounded. You'll also learn more about using the `rigidbody` component.

In Cloud Hop, the sky's the limit! Jump from cloud to cloud to see how far you can go. Each cloud earns you a point but be careful - falling off restarts the game!

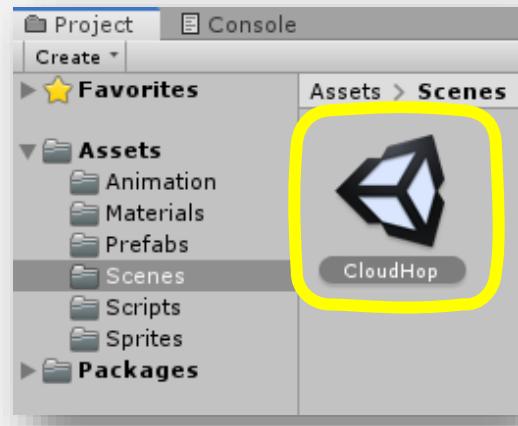


1 Start a new Unity Project and name it **YOUR INITIALS - Cloud Hop**. Be sure to select **2D core**.

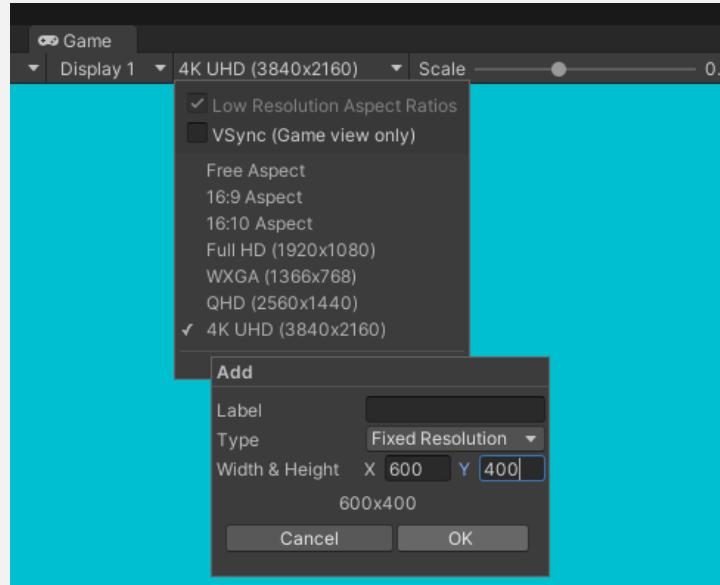
2 We've created a starter pack to give you a head start! To use it, import the **A2 - CloudHop.unitypackage** by going to **Assets > Import Package > Custom Package> All > Import**.



3 To open the starter package, double-click on the **CloudHop** scene. You can find this in the **Project** tab under **Assets > Scenes**.

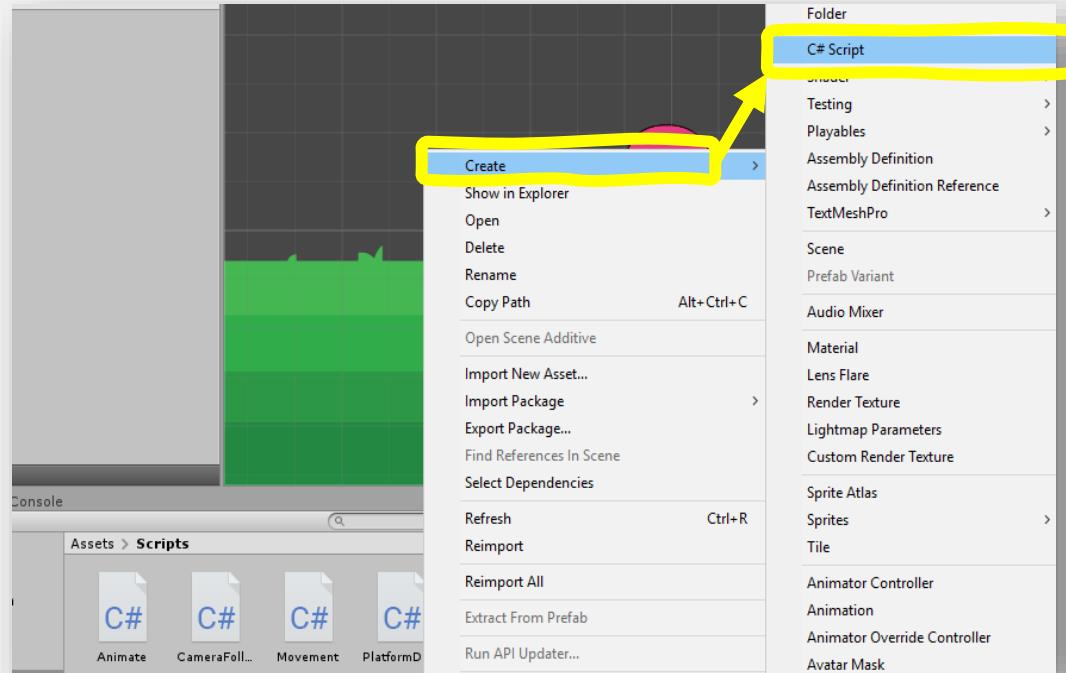


- 4** Go to the **Game** tab and change the Game Resolution to 600x800.
After you change the Game Resolution, you'll want to select the **Scene** tab for the next steps.

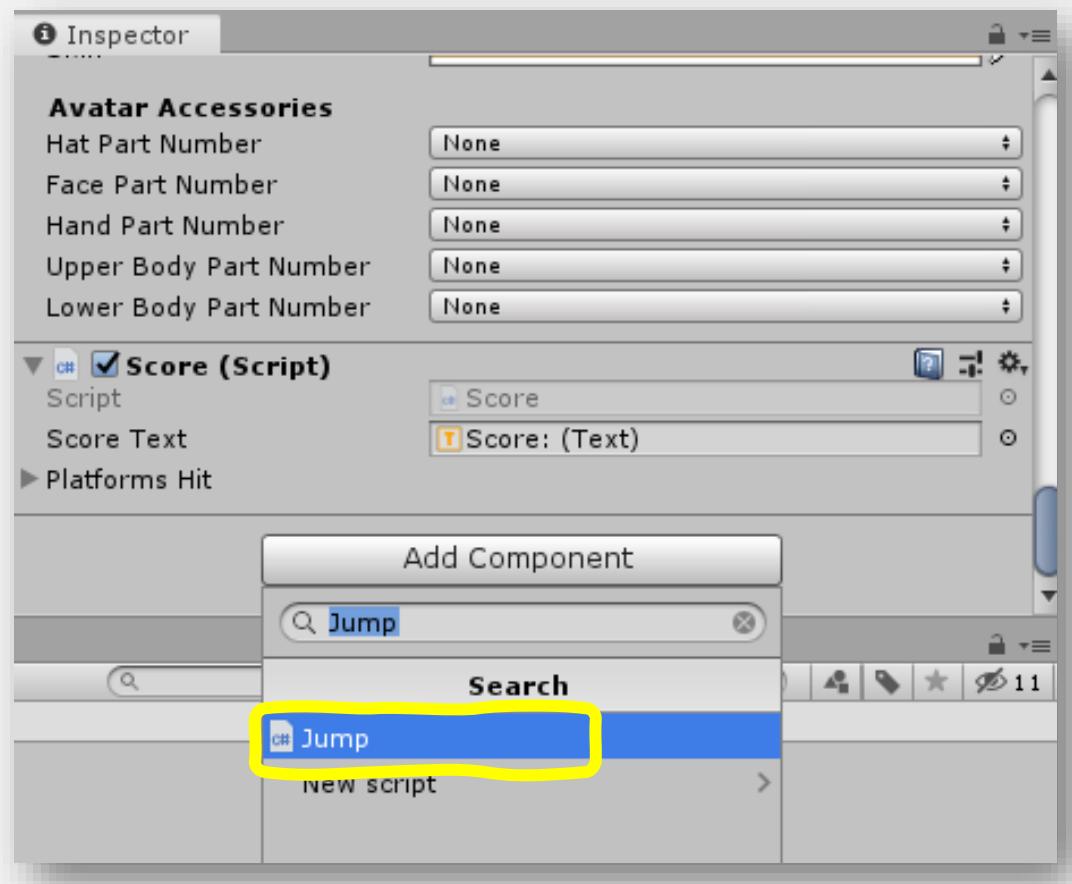


Note: If you do not have the "600x800" option in the drop-down menu, click on the plus sign at the bottom and enter 600 for width and 800 for height.

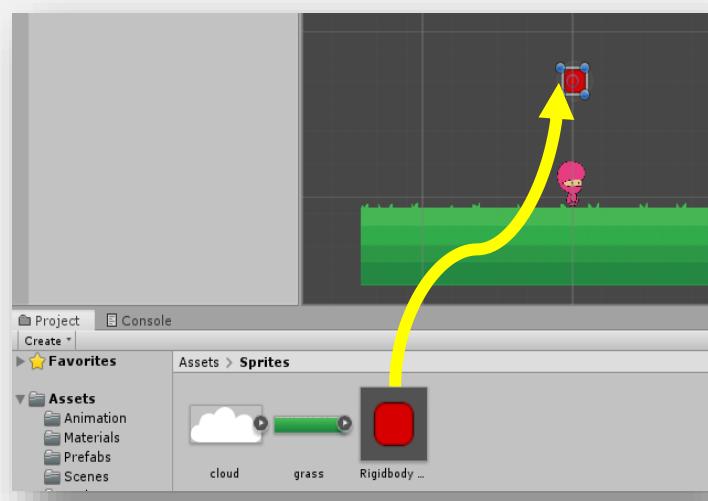
- 5** Create a new C# Script by right-clicking in your **Scripts** folder, then hover over **Create** and click **C# Script**. Name this script *Jump*.



- 6** To add the Jump script to the player character, click on **Player** in the Hierarchy. Then, in the Inspector, scroll to the bottom and click **Add Component**. Search and click on **Jump**.



- 7** Remember that adding a **rigidbody** component means the object will be affected by gravity. Let's see what this looks like. Go to the **Sprites** folder and drag **Rigidbody Test** into the scene.



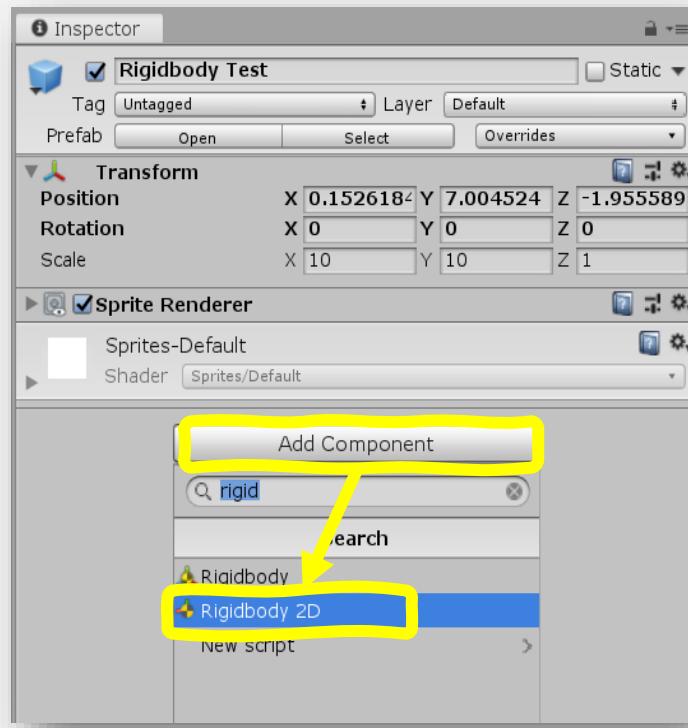
- 8** Press **Play**. See how the object just stays in place, not doing anything?

Go ahead and exit play mode.



- 9** Now try adding a **rigidbody** to the sprite to see what changes! To do this, click on the **Rigidbody Test** game object in the **Hierarchy**. In the **Inspector**, scroll to the bottom and click on the **Add Component** button, search for *Rigidbody2D*.

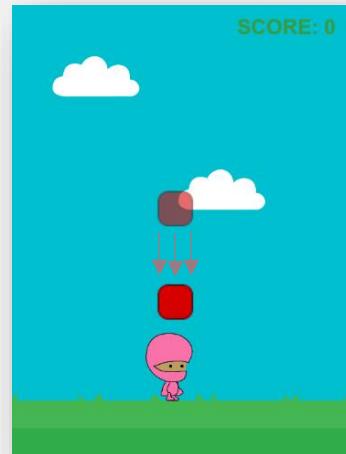
Remember, because this is a 2D game, we use Rigidbody 2D. A 3D game would just use a Rigidbody.



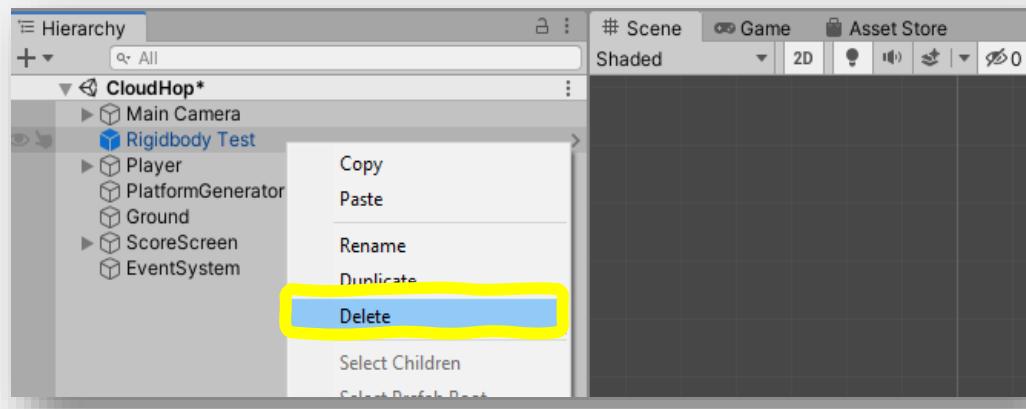
10 Play the game to see what happens when an object has a **rigidbody**.

It falls!

Remember, it is affected by physics, including forces like gravity.



11 Delete **Rigidbody Test** from the **Hierarchy**.



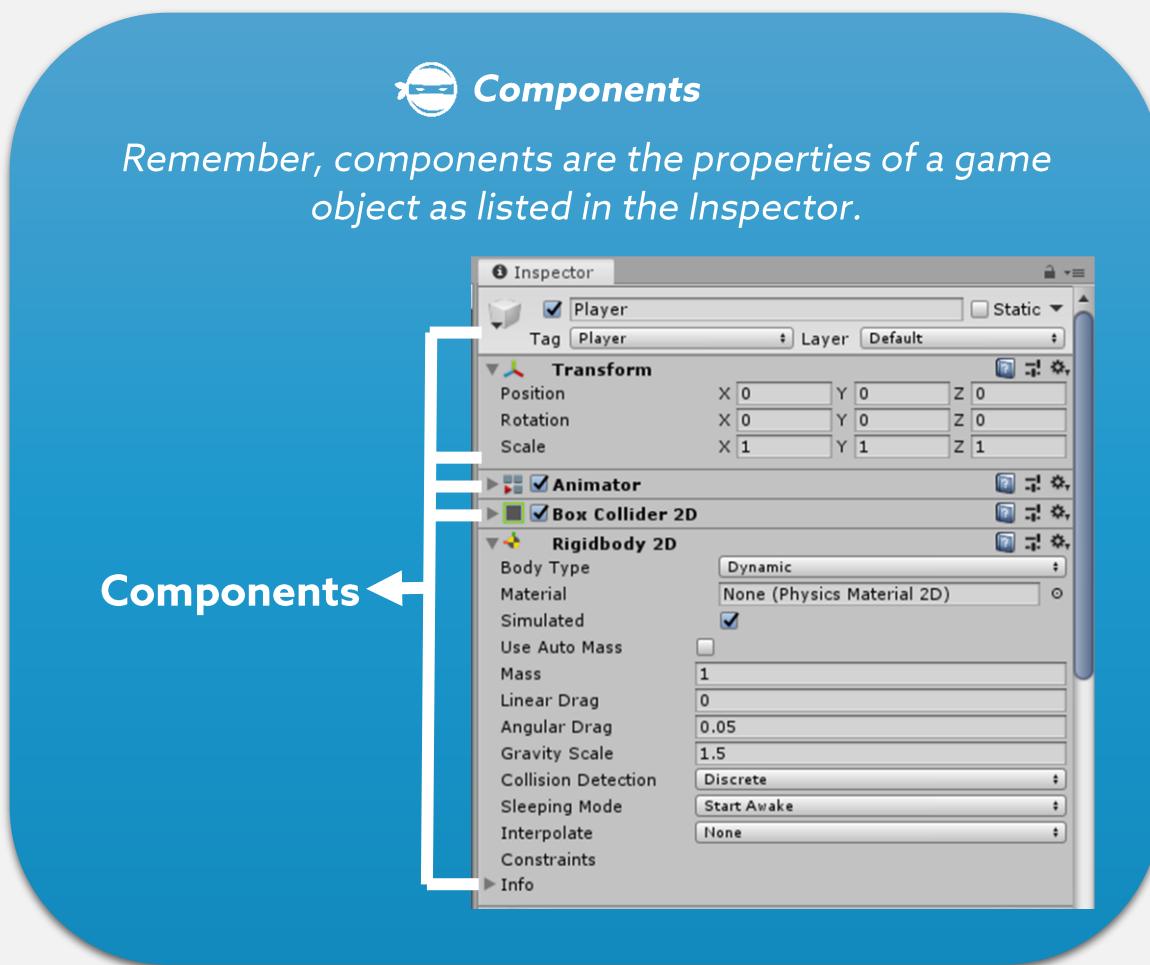
12 Let's use **rigidbodies** in the Cloud Hop game to make the player fall. You'll want to open your **Jump** script.

13 First, declare `private Rigidbody2D rb`. We do this so when we use the name "rb," Unity will know we are referencing a Rigidbody2D.

```
public class Jump : MonoBehaviour
{
    private Rigidbody2D rb;

    // Start is called before the first frame update
    void Start()
    {
```

- 14** Notice we did not declare `Rigidbody2D rigidbody` as a public variable. Previously, we would use a public variable and then drag the object into the slot in the Inspector. Instead, we will use a new function called `GetComponent<>()`.



We can assign and reference a component in our code using `GetComponent<ComponentName>()`. Get the `Rigidbody` component by adding the following line to your `Jump` script's `Start()` function:

```
public class Jump : MonoBehaviour
{
    private Rigidbody2D rb;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
    }
}
```

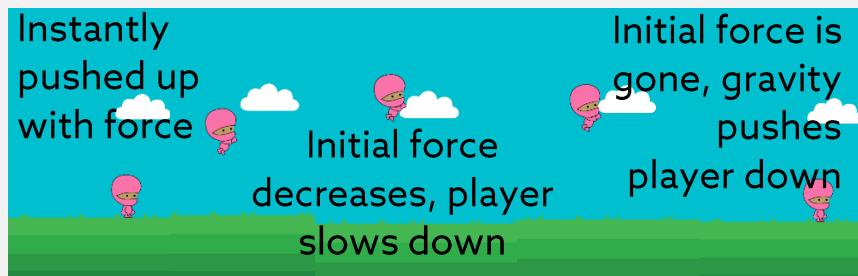
Rigidbodies

Why are *rigidbodies* particularly useful for setting up a jump? Not only will gravity do the work of pulling our character back to the ground, but it lets us use functions specific only to rigidbodies that will do the work to push our character up in the air.

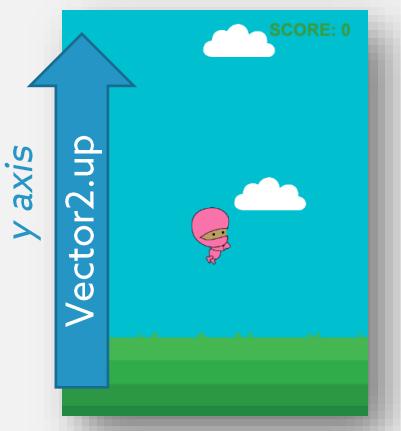
- 15** We will use `rb.AddForce()`. Adding a force creates movement similar to pushing something, it is pushed away at a set force or speed and then slows down over time.

`AddForce` takes the parameters: (float x, float y, float z, ForceMode)

ForceMode2D.Impulse tells Unity to apply the force instantly.



Instead of telling Unity the (x, y, z) coordinates of the player, we are going to use `Vector2.up` which is a shorter way of saying (0, 1). In other words, `Vector2.up` only changes the y direction, which is up.



Input.GetAxisRaw()

Remember, `Input.GetAxisRaw()` receives keyboard input for the arrow keys or WASD and returns either a 1, 0, or -1.

- 16** To adjust how high the player jumps, multiply it by a **jumpForce**. In your Jump script, declare `float jumpForce = 15`:

```
private Rigidbody2D rb;
private float jumpForce = 15;

// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
```

- 17** Let's see how the `rb.AddForce` function works by adding the following line of code in your `Start()` function. This will push the character up at a force of 15 as soon as the game is started:

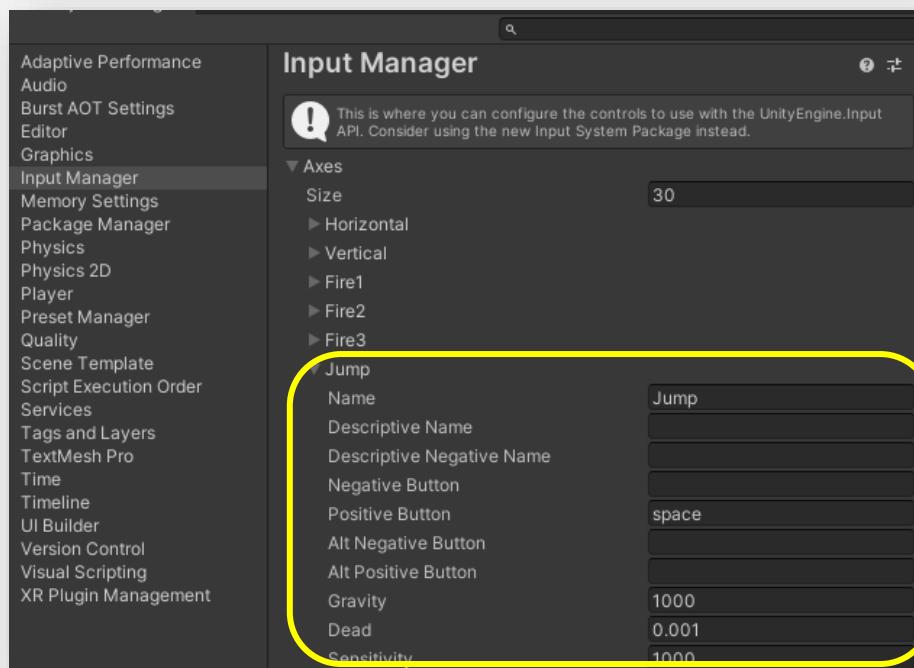
```
// Start is called before the first frame update
Unity Message | 0 references
void Start()
{
    rb = GetComponent<Rigidbody2D>();

    rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
}
```

- 18** Press **Play** and test it out! In your test, your player should jump, but only at the start.

19 Let's change the code so the player jumps when you press the space bar. We will use an Input function called `Input.GetButtonDown()`.

This returns true if the user is pressing whichever key you include in the parameters. Using `Input.GetButtonDown("Jump")` checks if spacebar was just pressed down.



Note: There are some other types of Input methods, such as `GetButtonUp`, which returns if the player released a button, or `GetButton`, which tells you if the player is holding down the button.

20 In Jump script, delete the `rb.AddForce()` function in `Start()`.

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();

    rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
}
```

- 21** Next, in `Update()`, add a conditional statement. Use `Input.GetButtonDown("Jump")` as the condition and `rigidbody.AddForce()` as the consequence like this:

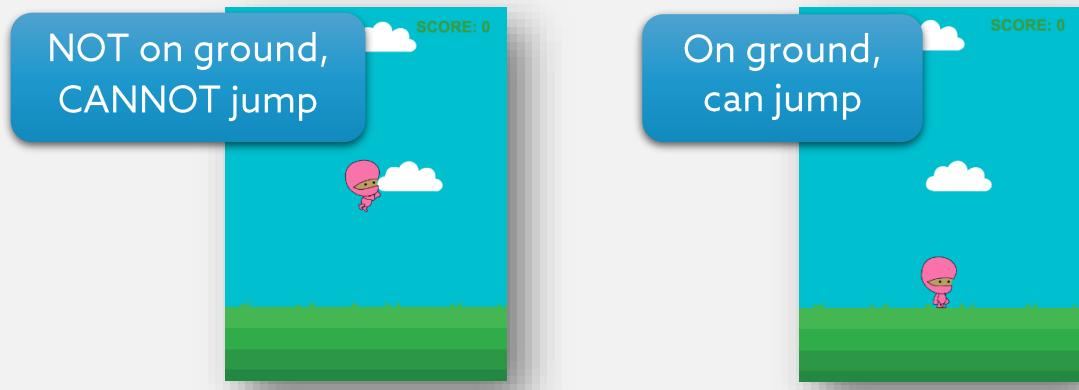
```
void Update()
{
    if (Input.GetButtonDown("Jump"))
    {
        rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
    }
}
```

- 22** Press **play** to test out your code! The player should now jump when you press the spacebar.

- 23** There is a problem with the jump; did you find it? Our code says if the spacebar is pressed, add an upward force. What happens if we just keep pressing space? Try repeatedly pressing spacebar. The player just keeps jumping mid-air, which makes the game too easy! Let's fix this.

- 24** First, we need a way to check if the player is on a surface (either the grass or cloud platforms).

If they are not on a surface, they are not allowed to jump again.



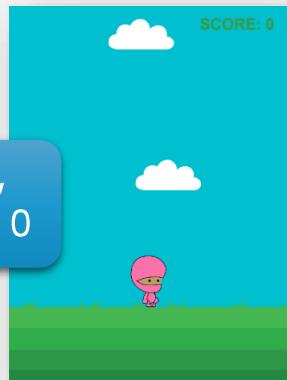
25 To check if the player is in the air, we will use `rb.velocity`.

Velocity

The velocity of the character is the rate at which it changes position.

When the player is staying still, their velocity is 0. In other words, **if** player velocity is 0, **then** the player is standing still on a surface.

on ground,
Y Velocity = 0



26 Jumping only deals with the vertical, or y axis. Therefore, we can use `rb.velocity.y` to check if the player is jumping or not.

Let's add an if/else statement that checks if the player is jumping or not to our if statement. Add this line of code in your Jump's `Update()` function:

```
void Update()
{
    if (Input.GetButtonDown("Jump") && rb.velocity.y == 0)
    {
        rb.AddForce(Vector2.up * jumpForce, ForceMode2D.Impulse);
    }
}
```

27 Great job, ninja! Play your game to test it out. You should be able to jump from cloud to cloud. But the player should only be able to jump from a surface and not while in mid-air.

Prove Yourself

Get Started

- Look in your Projects tab under the PY folder. Double-click on the **CantStopTheRain** scene in the Scenes folder.
- Make sure the Game Resolution is 600x800.

Task

Your challenge is to code the jump for the fire player in the scene Can't Stop the Rain. You'll first have to add in a Rigidbody 2D component (TIP: Make sure to turn check the rotation constraints on the rigidbody). Then, try and create a script to add force the player. Just like Cloud Hop, Can't Stop the Rain is a 2D platformer. The goal is to reach the final platform where there is safety from the rain. Be careful, the rain damages the fire's health! You'll be able to regain health by collecting the logs. Feel free to challenge your interface skills and rearrange the platforms to change the level of difficulty!

Activity 4

Jungle Escape

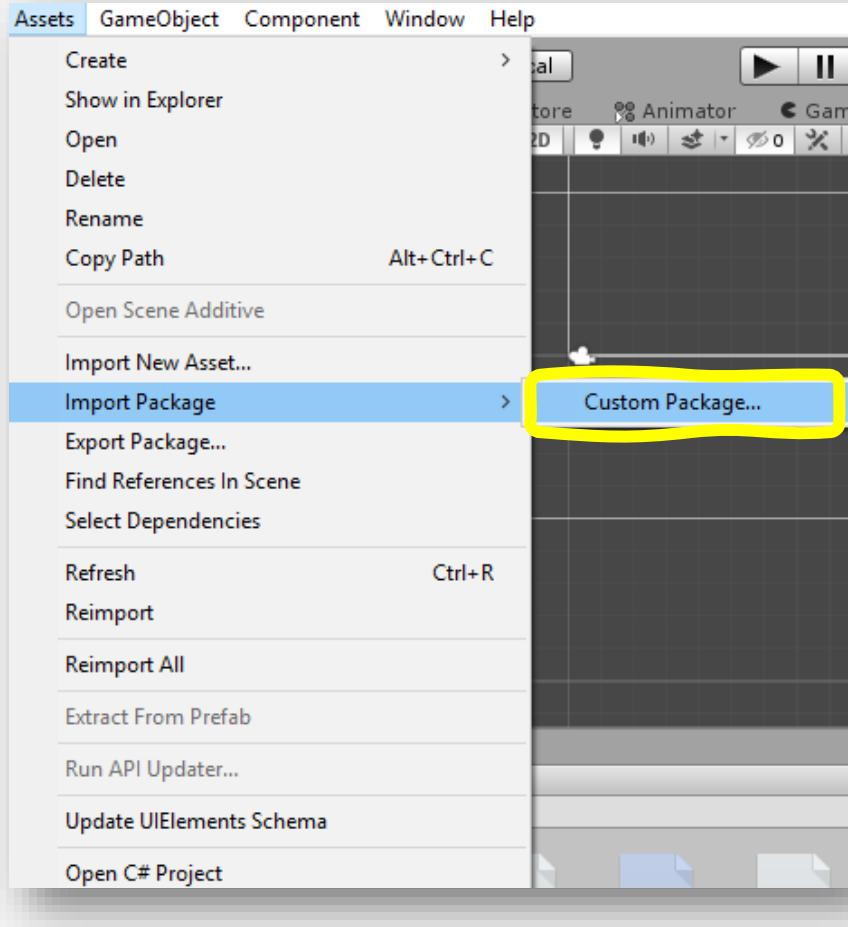
Through this game, you will learn about **raycasts** and use it as another way to check if the player is grounded. You'll also make jumping more realistic by adjusting gravity as the player falls. Lastly, you'll learn about the **Animator Controller**, including how to control the Animator's parameters in your code.

Today's mission: Navigate the wooden boards toward the glowing portal that will teleport you to your escape: the beach! Be careful though – some boards may disappear on you and others will try to trick you!

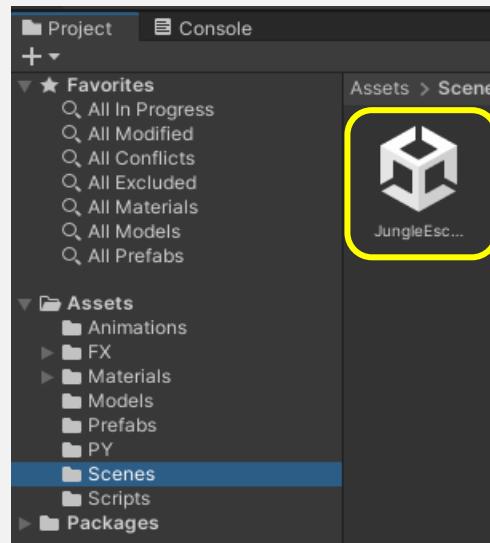
If you press play, you will notice that Codey's animations do not play. Can you figure out the second problem with Codey?



-
- 1** Start a new Unity Project and name it *YOUR INITIALS - JungleEscape*.
Select **3D core**.
- 2** Import the Jungle Escape- starter Unity Package by going to
Assets > Import Package > Custom Package > All > Import



-
- 3** Double-click on the **JungleEscape** scene. You can find this in the
Project tab under **Assets > Scenes**.



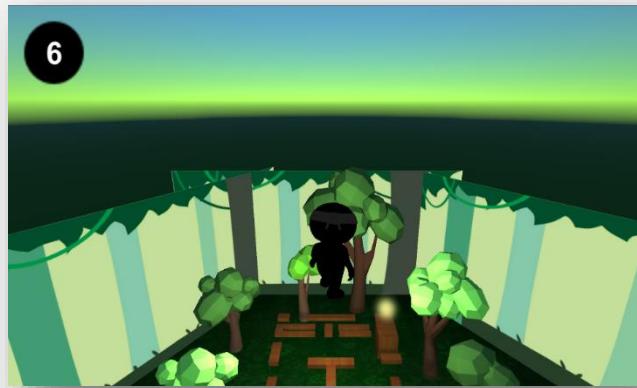
4 Go to the **Game** tab and change the Game Resolution to 16:9.

5 In your **Project** tab, open the C# Script named **Jump**. Notice the code adding force is already there. This is the same code we used in Cloud Hop. Do you see anything missing?

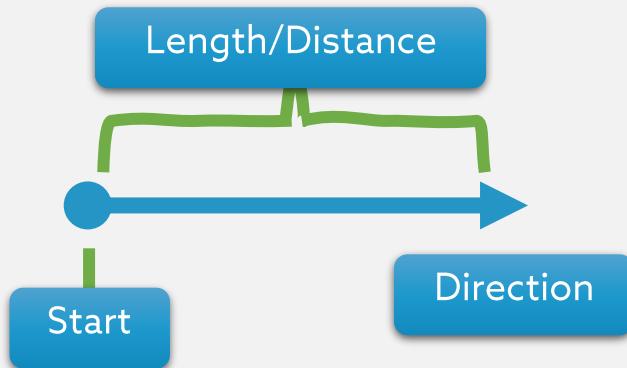
```
7  Unity Script (2 asset references) | 0 references
8  public class Jump : MonoBehaviour
9  {
10     Rigidbody rb;
11
12     float jumpForce = 5.7f;
13
14     void Start()
15     {
16         rb = GetComponent<Rigidbody>();
17     }
18
19     void Update()
20     {
21         if(Input.GetButtonDown("Jump")){
22             rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
23         }
24     }
}
```

6 There is no grounded condition yet! What happens when there is no grounded check? This should help you figure out what, besides the animation, is wrong with the game. Press play and test out your theory!

Did you repeatedly press spacebar? The second problem is that the player can infinitely jump, without ever touching the ground!



- 7** Cloud Hop taught you how to check if the player is grounded using the player's vertical velocity. In Jungle Escape, you will learn another way to check if the player is grounded. We will be using a raycast with the function `Physics.Raycast()`.
- 8** Like the name suggests, a **raycast** casts a ray. What does that mean? It creates a line with a specific start point, length, and direction.

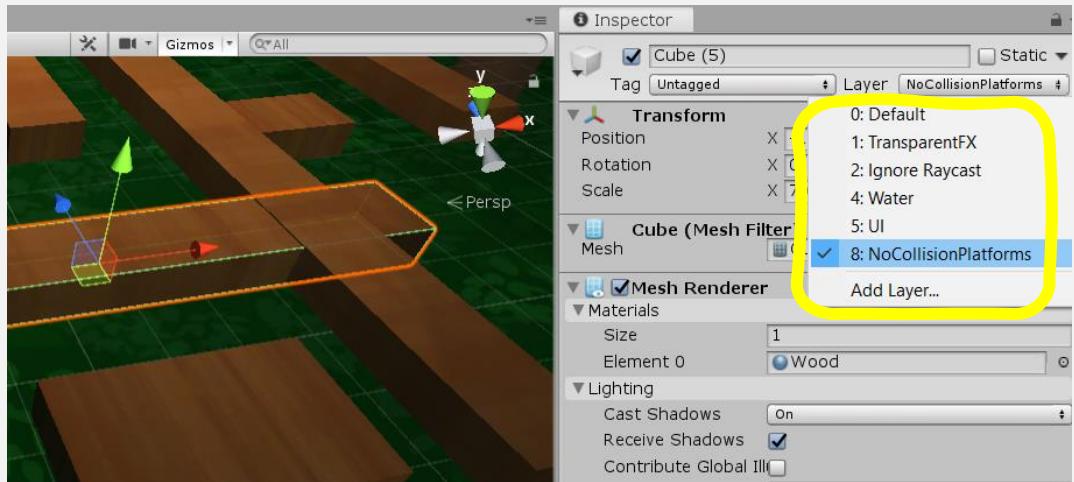


- 9** The main parameters for `Physics.Raycast()` are as follows:
- ```
Physics.Raycast(Vector3 origin, Vector3 direction, float distance, int layer)
```

### 💡 **What is a Layer?**

*Objects are put into different layers so that you can select which functions affect it.*

*If you include the layer parameter in a raycast, then the raycast only returns true if it hits an object in that layer.*



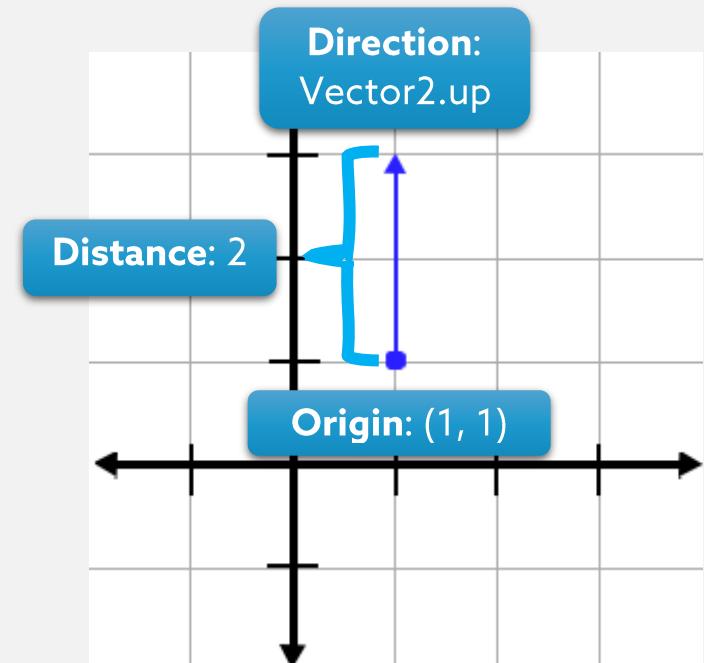
In this picture, the selected platform is in layer 8, which we've named **NoCollisionPlatforms**. To reference a layer, you must use a very specific format. We cannot use the number 8.

Instead, format it like this: `1 << layer`. For a **raycast** only on layer 8, it would read: **Physics.Raycast(origin, direction, distance, 1 << 8)**. If you want the raycast to look at all layers, simply leave out the layer parameter.

You can also create a **layermask variable** to make it a bit easier to read.

## EXAMPLE:

What do you think the function looks like for the ray below?  
Assume it looks at all layers.

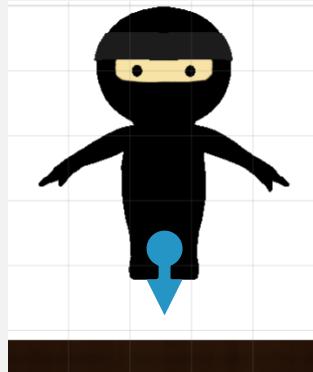


The function for this ray is:

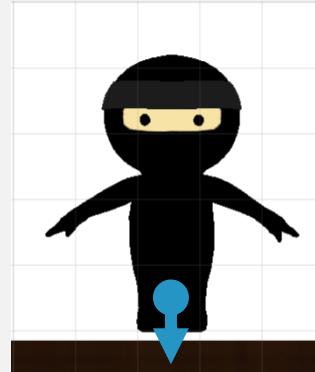
`Physics.Raycast(new Vector2 (1, 1), Vector2.up, 2).`

- 10** How does using a ray help check if the player is grounded?  
`Physics.Raycast()` returns true when it touches another collider! So, if we cast a ray down from our player for a specific distance, then we know the player is grounded when the raycast is true.

Ray NOT touching ground  
`Physics.Raycast() = false`



Ray touching ground  
`Physics.Raycast() = true`



- 11** How do we get the origin to follow our player? A new Vector is just an unchanging location; we need to get the constantly changing location of our player. We can make the vector origin change using `transform.position`, which always gets the position of the GameObject that the script is attached to. In this case, it's our player.

Now you'll get to see it all come together in our Jungle Escape game.

- 12** Let's get coding! In your **Project** tab under the **Scripts** folder, open the **Jump** script. First declare a `public bool isGrounded`. This will serve the same purpose as the `canJump` bool used in Cloud Hop.

```
public class Jump : MonoBehaviour
{
 Rigidbody rb;

 float jumpForce = 5.7f;

 public bool isGrounded;

 void Start()
 {
```

`isGrounded` will be assigned to the `Physics.Raycast()` so we can easily access the ray's return. Start by adding the following line to your `Update()`:

```
void Update()
{
 isGrounded = Physics.Raycast();

 if(Input.GetButtonDown("Jump")){
 rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
 }
}
```

- 13** Now using what you've learned about `Physics.Raycast()` create a function in `Update` that casts a ray from the player for a short distance. Remember the three parameters: (origin, direction, distance).

Add the parameters as shown here:

```
void Update()
{
 isGrounded = Physics.Raycast(transform.position, Vector3.down, .15f);

 if(Input.GetButtonDown("Jump")){
 rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
 }
}
```

**14** Before we press play, Unity has a handy function that lets us see the ray we just created: `Debug.DrawRay()`!

```
Unity Message | 0 references
void Update()
{
 isGrounded = Physics.Raycast(transform.position, Vector3.down, .15f);
 Debug.DrawRay(transform.position, Vector3.down * .15f, Color.red);

 if(Input.GetButtonDown("Jump")){
 rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
 }
}
```

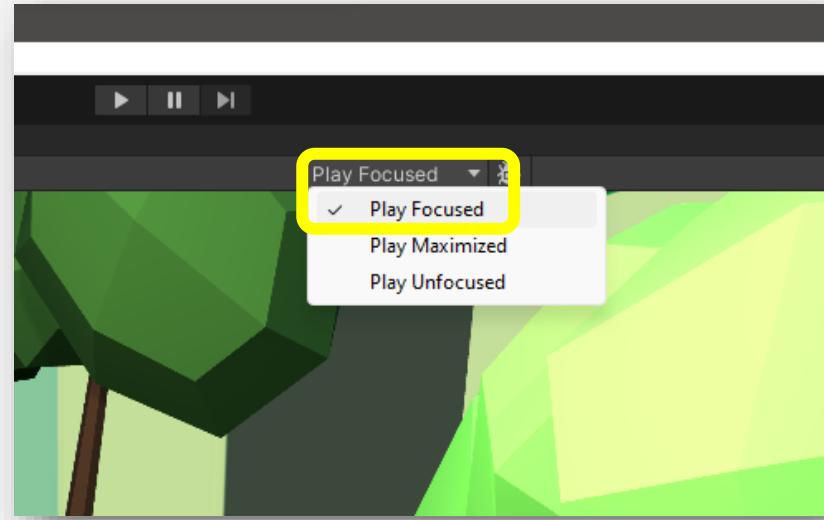
**15** Save your script.

**16** Let's see what the **raycast** looks like!

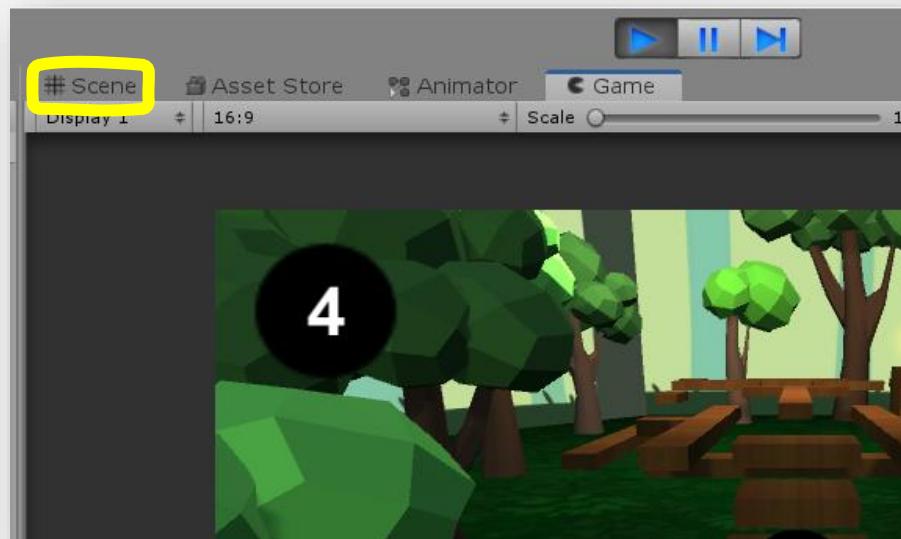
Note: `isGrounded` is not yet included in the condition for jump; this means you can still infinitely jump.

Press **play**.

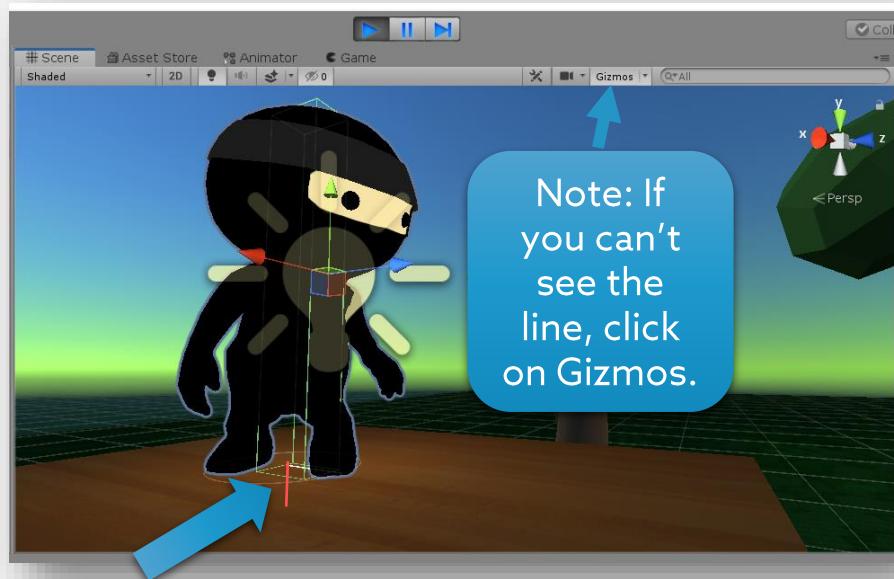
**17** If your game is full screen, click on **Play Focused** in the game window (circled below):



Next, click the **play** button to exit play mode. Then press **play** again. Click on the **Scene** tab.



- 18** Zoom in and rotate around Codey to see the **raycast** you made. Remember in the `Debug.DrawRay`, you included `Color.red`? Can you find the **raycast**?



*Note: The ray goes through the wooden platform, so it may be hard to see. If you are having trouble finding it, press space bar to get Codey to start his jump, then pause the game. Press the skip button until you can see the ray.*

## 19 Awesome job, ninja! All the pieces are coming together!

Let's do a quick review. We have two separate functions: an add force and a physics raycast stored in `isGrounded`. What is the final step we need to finally get rid of the continuous jump?

Stop and think about what we need to do before you move on to the next step.

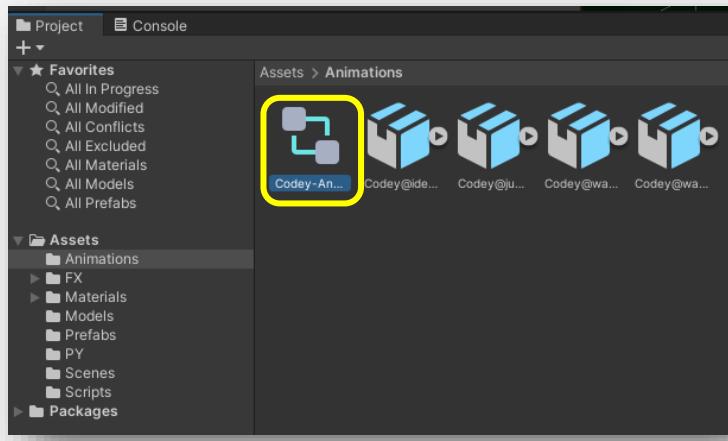
## 20 That's right! Add `isGrounded` to the add force's `if` condition:

```
Unity Message | 0 references
void Update()
{
 isGrounded = Physics.Raycast(transform.position, Vector3.down, .15f);
 Debug.DrawRay(transform.position, Vector3.down * .15f, Color.red);

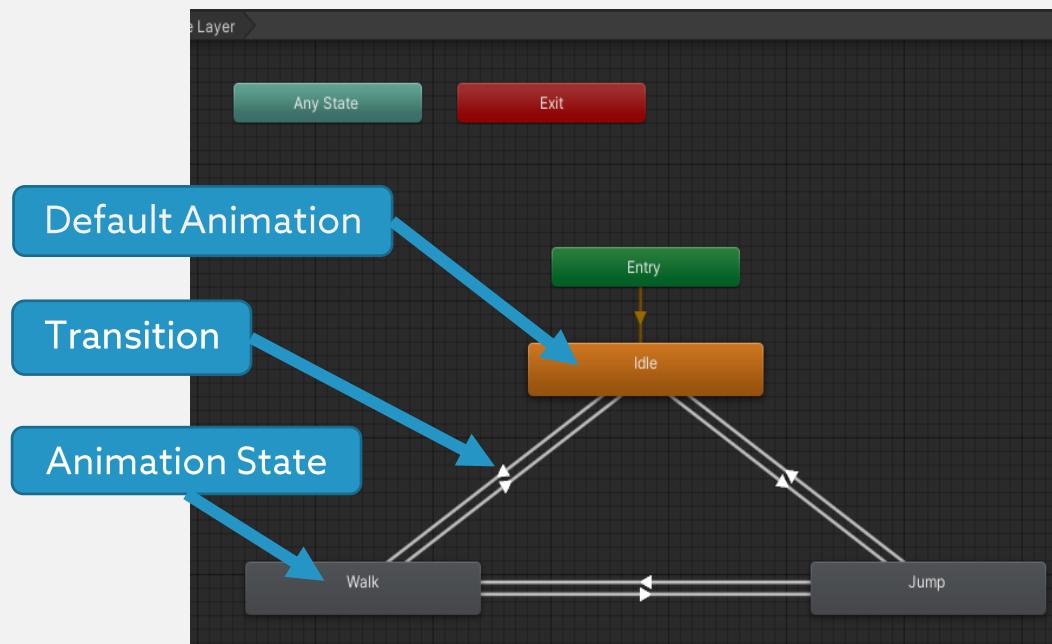
 if(Input.GetButtonDown("Jump") && isGrounded){
 rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
 }
}
```

## 21 Save your script and test out your jump!

## 22 Now that you've mastered movement code, you must now master animation. If you recall from your lessons in Purple Belt, animations are controlled in the animator controller. In your **Project** tab under the **Animations** folder, you will find the **animator controller** for Codey. Double-click on the controller to open it.

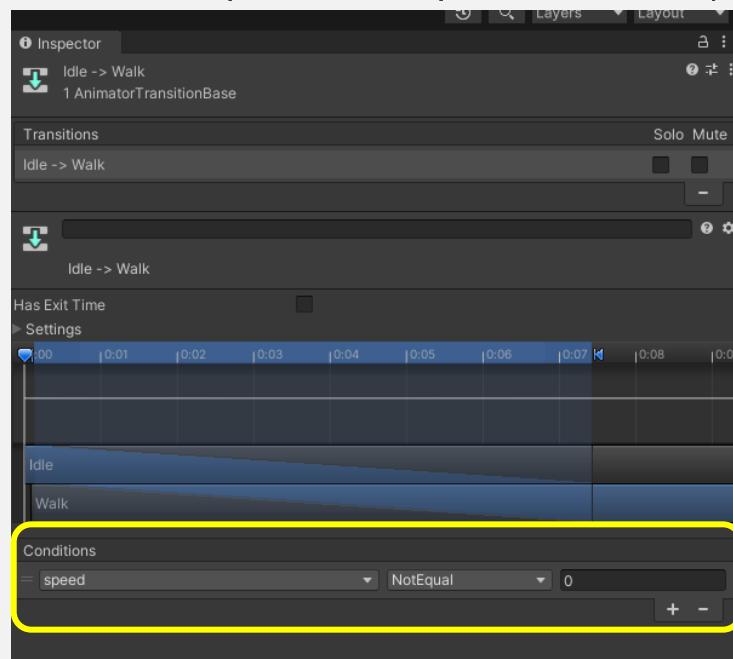


**23** Let's review the animator controller interface. Each rectangle is an **animation state**, so in this case Codey can either be idling, walking, walking backwards, or jumping. The arrows connecting the states are **transitions**.



**24** An arrow means Codey can transition from that animation to the other if it meets the conditions. Click on the **transition** arrow from Idle to Walk and look in the **Inspector**.

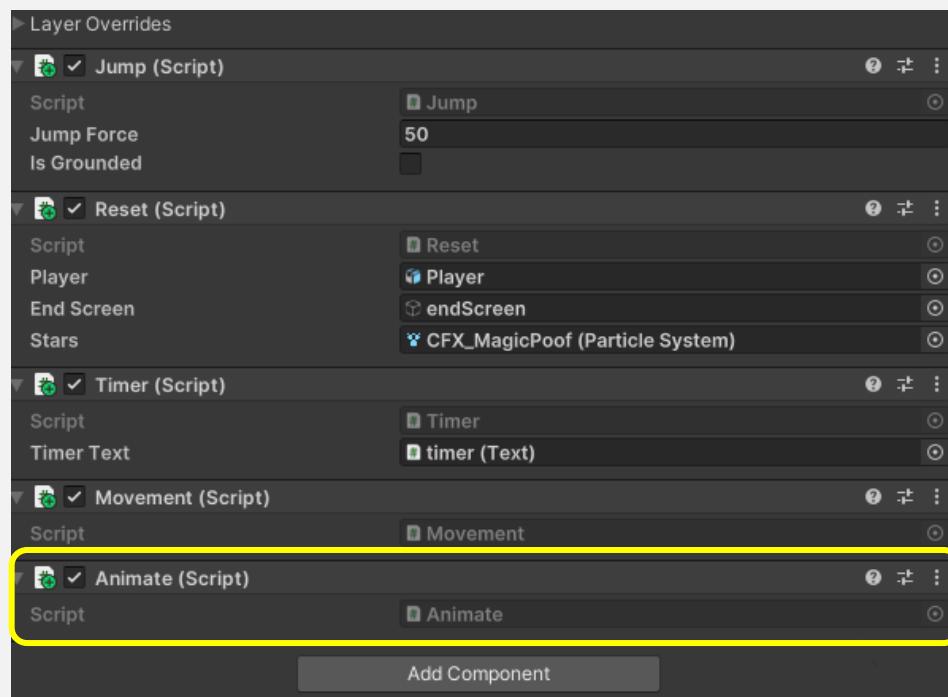
**25** Under **Conditions**, speed must be greater than 0 for Codey to change from idle to walk. How can you tell Unity what Code's speed is?



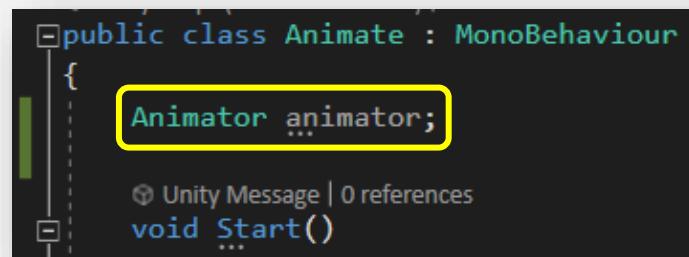
**26** Let's get coding! The condition speed is a whole number, or integer (int for short!). Therefore, we must set up if statements that will say when each animation condition is true or false. First, let's open up the Animate script.

**27** Click on the **Player** in the **Hierarchy**.

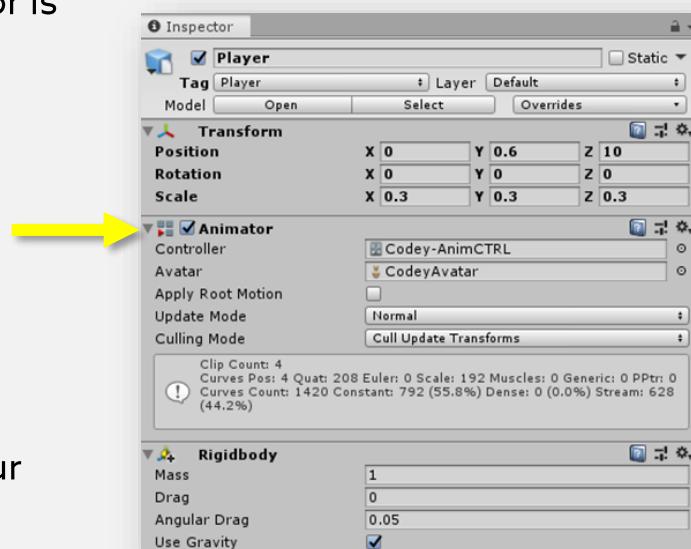
In the **Inspector**, double click on the **Animate** script to open it up.



**28** Declare `Animator animator`. This will let Unity know that whenever we use the name Animator, we are referencing the type Animator (for Animator Controller).



**29** Remember that the Animator is a component. How did we access a game object's component before, like the rigidbody? We use `GetComponent<>()`.

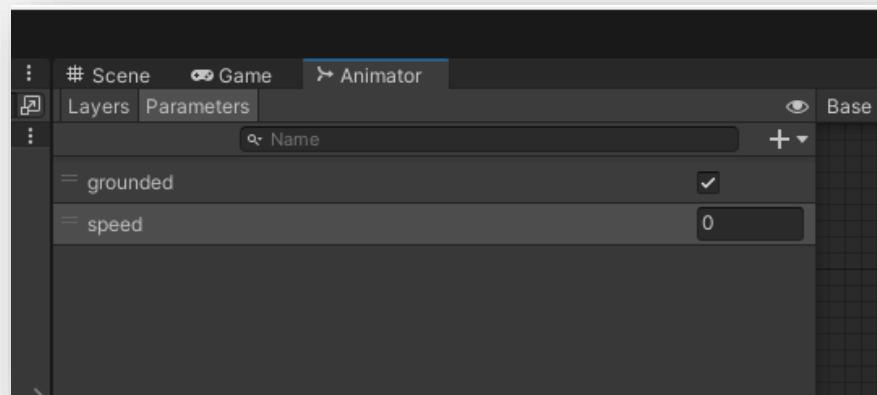


Add the following line to your `Start()` function:

```
void Start()
{
 animator = GetComponent<Animator>();
}
```

**30** Now that we have the Animator Controller in our script, let's think about how to organize our Animate script! We need it to control when Codey stays in an animation or transitions between animations. Remember, in the **Animator Controller**, Codey needed the speed condition.

Before we think about the speed, let's first look at the other parameter, grounded.



**31** Notice that we included “grounded” as a factor. For jump, the user input is the spacebar, but the player must also be grounded. If we just use `Input.GetButton("Jump")` as the condition for the jump animation, what happens when Codey is in the air and the spacebar is pressed? Codey does not jump again, but his jump animation will play again.

**32** We need to use `isGrounded` as a condition for the jump animation, but the `isGrounded` variable was made in the Jump script. How do we access a variable that was made and defined in another script?

**33** Just like other types, you can declare a script. This makes all declarations and functions from that script recognized in the current script. To declare a script, the type is the name of the script, then the name you would like to call it in the current script.

For example, to declare the Jump script, write the following line:

```
public class Animate : MonoBehaviour
{
 Animator animator;
 Jump jump;
```

**34** Assign the Jump component to the variable using `GetComponent<>()`

```
void Start()
{
 animator = GetComponent<Animator>();
 jump = GetComponent<Jump>();
}
```

**35** To use variables or functions from Jump all we need to do is include the script name with a period, like so: `Script.Variable` or `Script.Function()`.

**36** Great! Now that we can access `isGrounded`, let's set the animation bool to be the same. In the **Animate** script under `Update()`, add the following statement:

```
void Update()
{
 animator.SetBool("grounded", jump.isGrounded);
}
```

 **Remember**

An exclamation mark "!" in front of a variable means NOT.  
So `!Jump.isGrounded` means `isGrounded` equals false.

**37** Now let's get the speed parameter. Just like how we created a reference to the Jump script, we are going to get a reference to the movement script. Insert the code as shown below.

```
public class Animate : MonoBehaviour
{
 Animator animator;
 Jump jump;
 Movement movement;

 void Start()
 {
 animator = GetComponent<Animator>();
 jump = GetComponent<Jump>();
 movement = GetComponent<Movement>();
 }
}
```

---

**38** Now using `SetFloat`, get the speed from the movement script and set the animation parameter to match.

```
void Update()
{
 animator.SetBool("grounded", jump.isGrounded);
 animator.SetFloat("speed", movement.speed);
}
```

---

**39** Save your script. Your game is ready to go! Press **play** and test it out.

# Prove Yourself

## Get Started

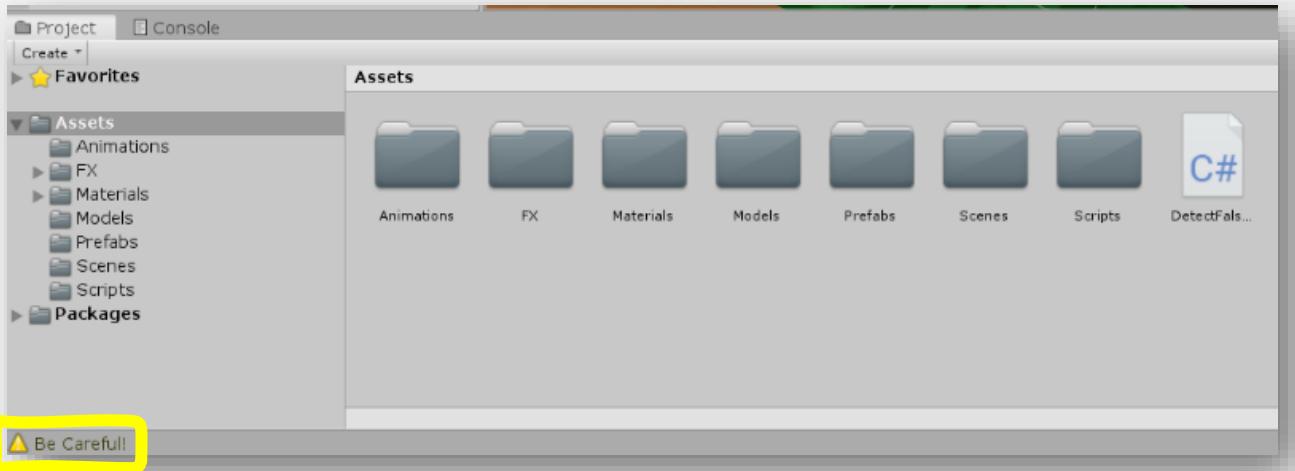
- Look in your Projects tab under the PY folder. Double-click on the **Jungle Escape PY** scene.
- Make sure the Game Resolution is 16:9.

## Task

In Jungle Escape, it was up to you to test the platforms and find your way. Now, you can create a guide using a raycast to log if the next platform is safe. If the ray hits a platform with no collider, it should give a warning telling you to avoid that platform!

In the **Project** tab under the **PY** folder, open the **DetectFalsePlatforms** script. First, create a bool called *hit* and set it equal to your `Physics.Raycast()`. The ray's origin needs to be Codey's position and it should face in the direction `transform.forward`. `Transform.forward` is like `Vector3.forward`, except that it updates to face the direction Codey is facing. The distance needs to be such that it is long enough to reach the next platform, but not too long where it is detecting platforms further away.

You can use `Debug.DrawRay()` to help find the right distance. We placed all the platforms with no colliders on layer 8, so make sure to include the layer parameter `1 << 8`. Alternatively, create a public variable of type `LayerMask`, and set it to only be layer 8 in the inspector. Once the raycast is made, have it `Debug.LogWarning("Be careful!")` if it is true, else `Debug.Log("All clear!")`.



## Activity 5

# Ninja Run

You will be able to create a pickup item with a tag and trigger collision which will be utilized to code a scoring system and control particle effects.

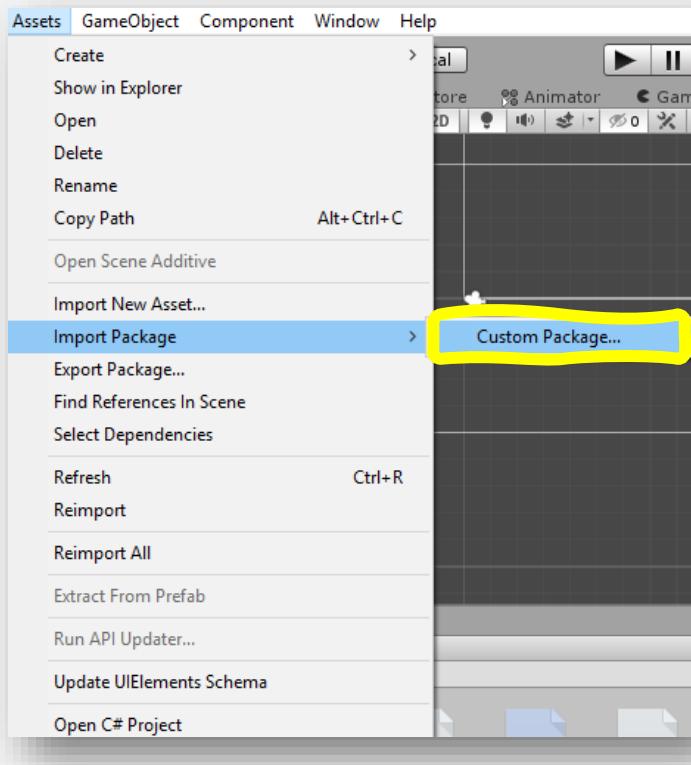
Your mission: With three lives, it is your goal to navigate the endless castle walls to pick up as many coins as possible. But beware – the further you get, the faster you go! Press play to test out the game.

After playing, you may have noticed that Codey can't pick up any coins! Alright ninja, we've got work to do. We need to code our pickups to work, complete with score and particle effects.

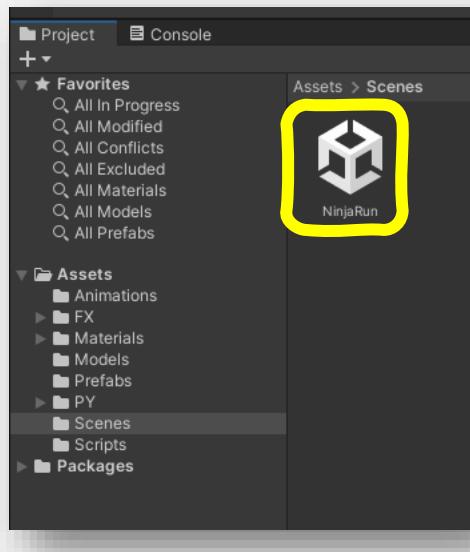


**1** Start a new Unity Project and name it *YOUR INITIALS – Ninja Run*.  
Select **3D core**.

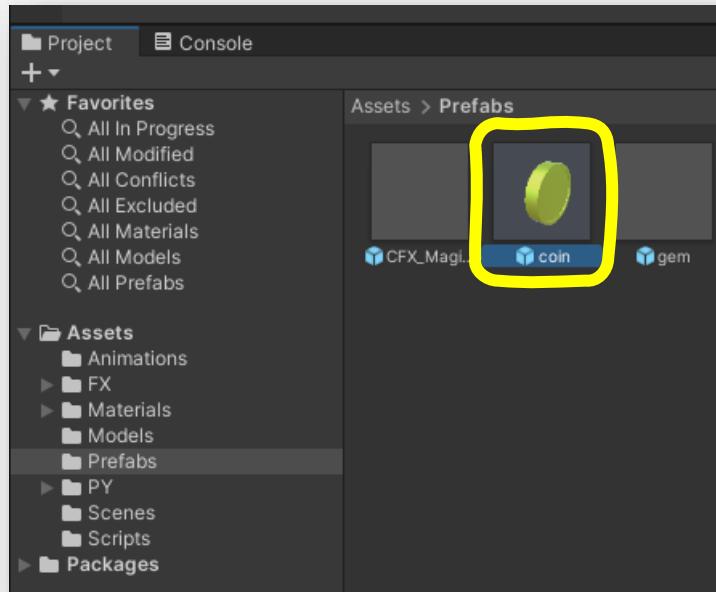
**2** Import the Ninja Run starter Unity Package by going to  
**Assets > Import Package > Custom Package > All > Import**.



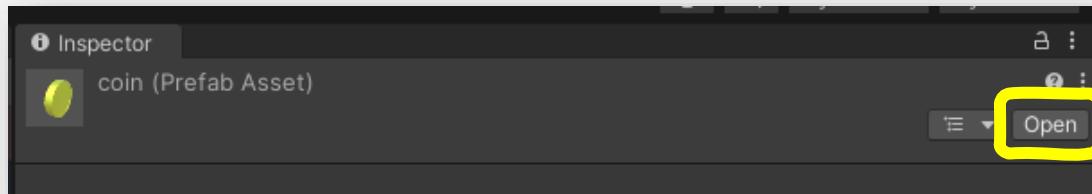
**3** Double-click on the **NinjaRun** scene. You can find this in the **Project** tab under **Assets > Scenes**.



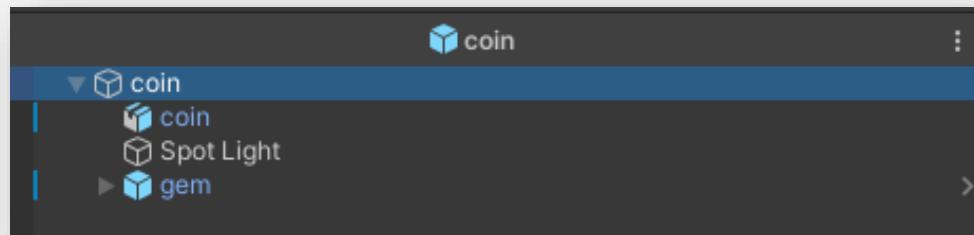
- 4** Let's start by creating our pickup item! Under your **Project** tab in the **Prefabs** folder, click on the **coin** prefab.



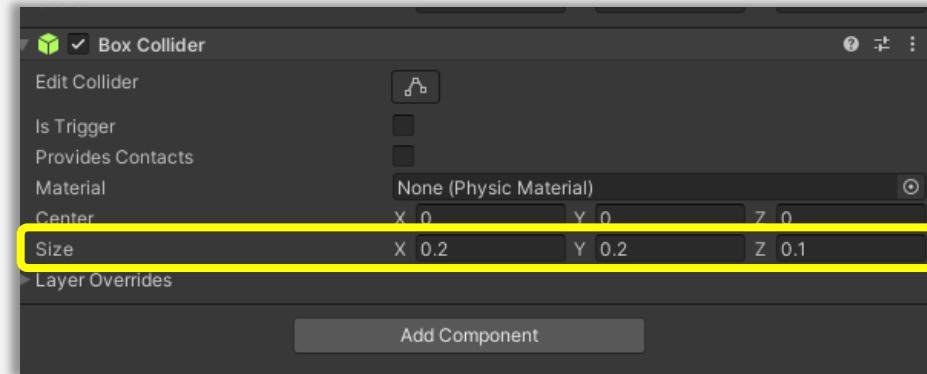
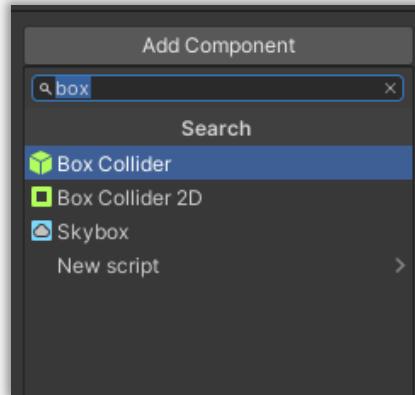
In the **Inspector**, click on the **Open Prefab** button. Here we can get a closer look and make any edits we need.



- 5** Notice that there is no collider component. Remember, a collider defines the space of the coin; without it, Codey can't interact with the coin. Make sure to select the *parent* of the coin prefab.



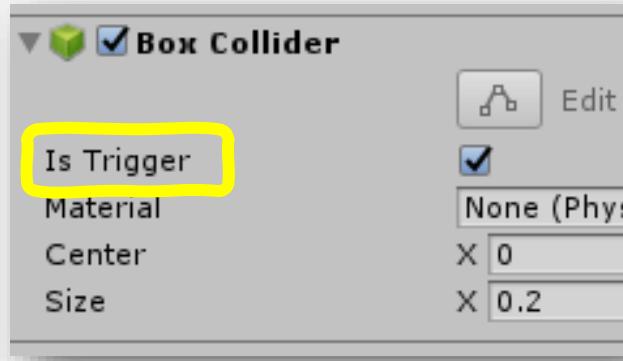
- 6** With the parent selected, add a **Box Collider** component and shape it to match the size of the coin. We used: (0.2, 0.2, 0.1)



- 7** Press **play**. Now when Codey hits a coin, he gets stuck! The colliders are doing their job and bumping into each other. We need to know when Codey hits the coin without the physical collision.

- 8** To fix this, we will make the collider into a trigger. A trigger will set off when hit by another collider, but without the physical collision. This is exactly what we need!

To make a collider a trigger, check the box for **Is Trigger** in the **Box Collider** component within the **Inspector**.



**9** Great job! The trigger is set up in the scene. Now, how can we use this in our code? Open the **Pickups** script – you can find it in the **Scripts** folder or on **Codey** which is a *child* object of **Player** in the **Hierarchy**. Delete the `Update()` function and replace it with:

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



### **OnTriggerEnter()**

We are using `OnTriggerEnter()`. This means that the function is called when the object enters the trigger.

There are also functions such as `OnTriggerExit()`, called when object exits trigger, and `OnTriggerStay()`, called while the object is inside the trigger.

**10** In your `OnTriggerEnter` function, add `Debug.Log("Trigger Enter")`.

**11** Save your work and go to your Unity window. Press **play**.

Check the console to see when the `OnTriggerEnter` function gets called.

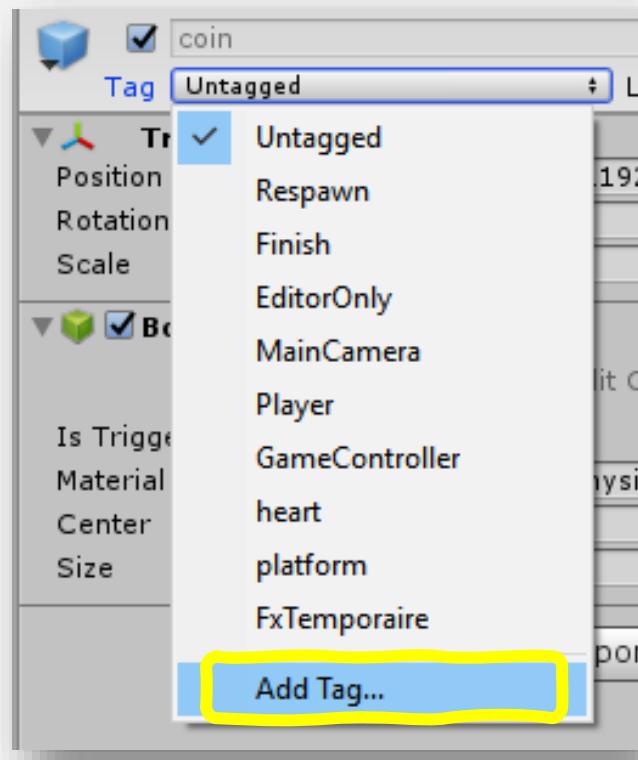
**12** The coins are the only objects in the scene with a trigger, so this works perfectly.

However, what would happen if there was an obstacle with a trigger? How would Unity know the difference?

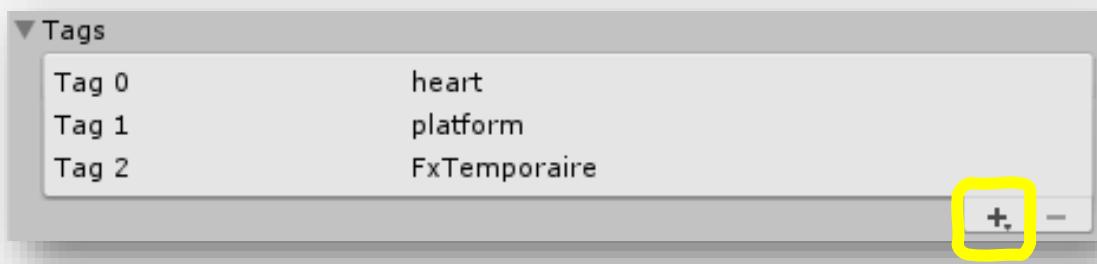
We can add a tag to objects and check the tag in the script! **Tags** are like a label that lets you easily find multiple objects with one name (or tag) in your code.

**13** In your **Project** tab under **Prefabs**, click on the **coin**.

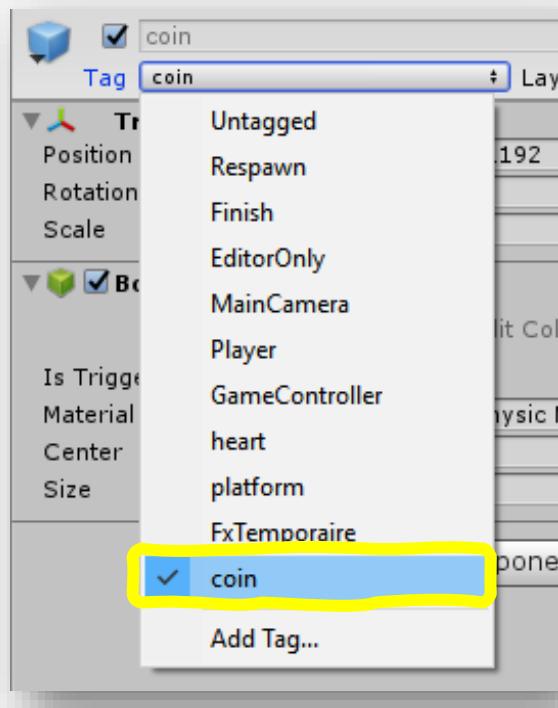
In the **Inspector** under the game object name, there is a drop-down for your tag. Right now, it is Untagged. Click on the drop-down and select **Add Tag**:



**14** Click the **+**, type *coin*, and click **Enter**.



**15** Click on the coin again in your **Prefabs** folder. Now when you click on the drop-down for the tag, you can select **coin**.



**16** Now we will use `CompareTag()` to check the tag. Open your **Pickups** scripts, in `OnTriggerEnter()`, delete the `Debug.Log()` and add this conditional statement:

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("coin"))
 {
 }
}
```

**17** Now Unity knows when Codey touches a coin, but the coin still stays in the scene!

This can be confusing because it looks like Codey didn't pick up the coin at all.

- 18** To remove the coin from the scene when Codey touches it, we will use the `Destroy()` function. To do this, open the **Pickups** script and add:

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("coin"))
 {
 Destroy(other.gameObject);
 }
}
```



### *Disappearing Game Objects*

*What if you wanted to destroy a game object not involved in the collision? For instance, if you wanted a door to disappear when the player picks up a key, the door is not involved in the collision between player and key. You would have to declare game object door and then use `Destroy(door)`.*

- 18** Remember the coins are the source of points. The User Interface (or UI for short) for the score is already in the scene. However, it doesn't update when you pick up a coin; the score just stays at 0.

- 19** In **Pickups**, declare `public int score` at the top.

```
public class Pickups : MonoBehaviour
{
 public int score;
 public Text scoreText;

 void Start()
 {
```

- 20** Right now, the score is staying at 0, but we want it to go up when we hit a coin. We'll need to include `score++` like this:

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("coin"))
 {
 score++;
 Destroy(other.gameObject);
 }
}
```

*Remember `++` adds 1 to the variable.*

- 21** Press **play**. The text score is still not updating!

Can you guess why?

The score variable is not yet connected to the text object.

In the meantime, you can test if the score is working properly by adding:

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("coin"))
 {
 score++;
 Debug.Log(score);
 Destroy(other.gameObject);
 }
}
```

Now the score will be output to the console.

**22** Scripting UI will be explained further in a later section; for now, let's go over sending the score integer to the score text object.

The score text object is already declared in your code as `scoreText`. We will use the `ToString()` function to convert the `score` integer to a string. This way it can be displayed as text.

We will assign the `scoreText.text` value to the stringified version of `score`:

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("coin"))
 {
 score++;
 scoreText.text = score.ToString();
 Destroy(other.gameObject);
 }
}
```

**23** Great Job! We just need to add some final details to really emphasize the pickup and add some fun. A great way to do this is with a particle system. Remember making them in Purple Belt?

Well now you are going to learn how to control it in your code!

**24** In `Pickups`, declare `public ParticleSystem Pickup;`:

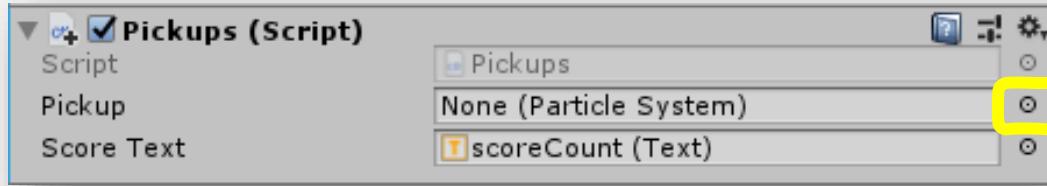
```
public class Pickups : MonoBehaviour
{
 public int score;
 public Text scoreText;

 public ParticleSystem Pickup;

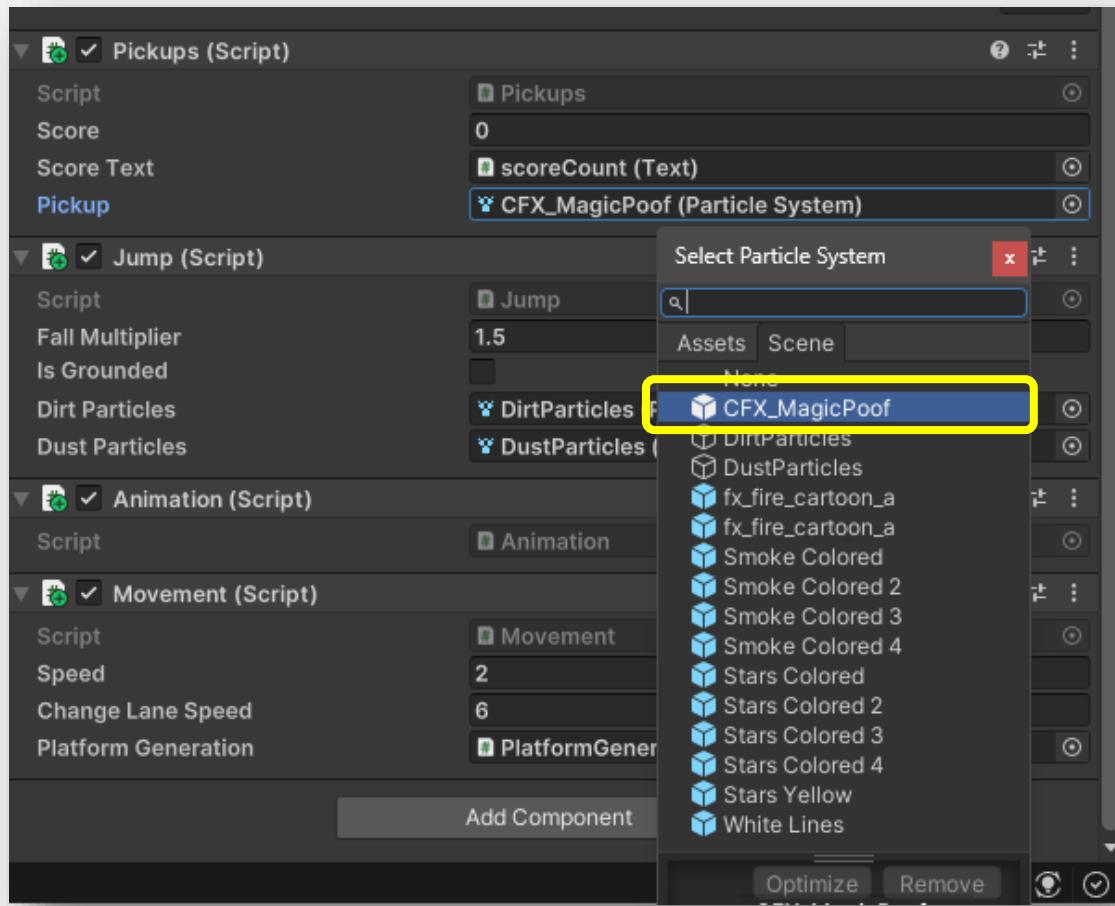
 void Start()
 {
```

## 25 Save the code and switch over to your Unity window.

With **Codey** selected, look in the **Inspector** for the **Pickup** script component and click on the circle to the right of the **Pickup** slot.



In the pop-up window, select **CFX\_MagicPoof**.



**26** Remember this emphasizes that a coin's been picked up, so it should only play when the trigger event is called. We will use the functions `ParticleSystem.Start()` and `ParticleSystem.Stop()` which tells Unity when to play or stop a particle system.

Back in our **Pickup** script, under `Start()`, add:

```
void Start()
{
 Pickup.Stop();
}
```

In `OnTriggerEnter()`, add:

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("coin"))
 {
 score++;
 scoreText.text = score.ToString();
 Destroy(other.gameObject);

 Pickup.Play();
 }
}
```

Remember to **save**.

Yay! You did it, ninja! Codey is ready to go.

*Before moving on to the Prove Yourself, challenge yourself to see how many coins you can collect.*

# Prove Yourself

## Get Started

- Look in your Projects tab under the PY folder. Double-click on the **FindTheExitPY** scene in the **Scenes** folder.

## Task

In a twist of events, the exit in Find the Exit is now blocked by a door! For this Prove Yourself, you need to set up a coin pickup item that the player must collect to destroy the door and escape. The coin is already in the scene. Use your knowledge of triggers, tags, and the `Destroy()` function to have the coin and door destroyed whenever the player touches the coin.

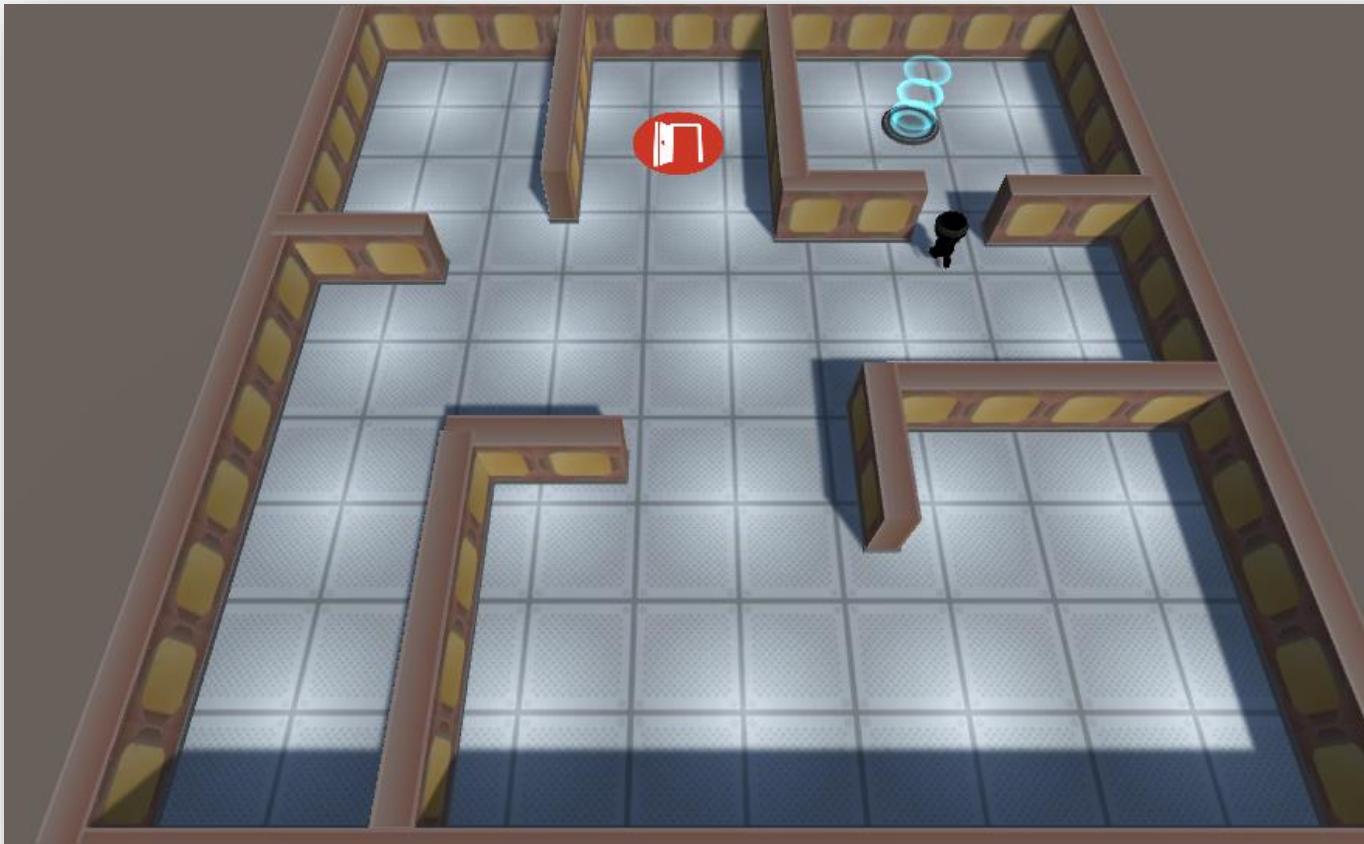


## Activity 6

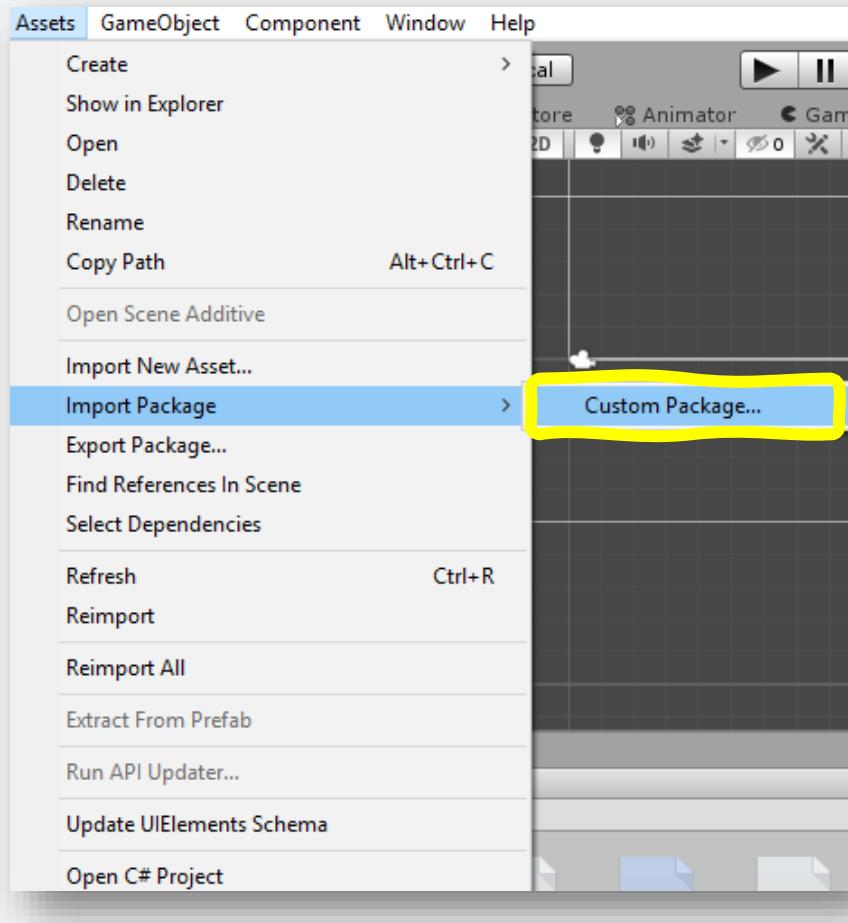
# Evil Fortress of Doctor Worm

Welcome to the Evil Fortress of Doctor Worm! When you playtest the game, you can see that you can move the character. However, the doors and switches aren't working yet.

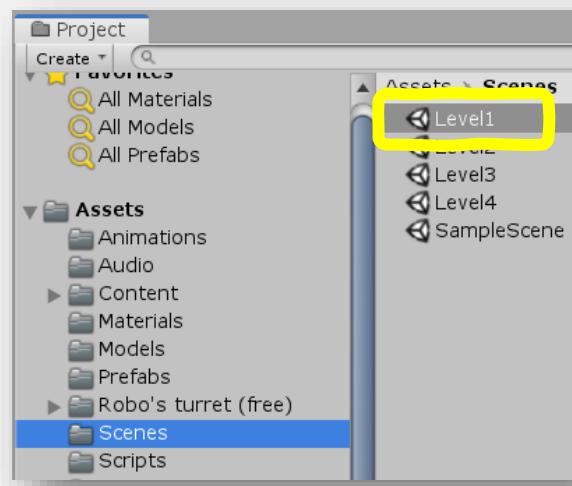
The mission: Codey is trapped in the evil fortress, and it's up to you to set him free! You'll need to create scripts to make doors in the fortress and add triggers so Codey can open them to escape.



- 
- 1** Start a new Unity Project and name it *YOUR INITIALS - Doctor Worm*.  
Select **3D core**.
- 2** Import the **DrWormFortress\_Ninja.unitypackage** file by going to  
**Assets > Import Package > Custom Package > All > Import**.



- 
- 3** Double-click on the **Level1** scene. You can find this in the **Project** tab under **Assets > Scenes**.

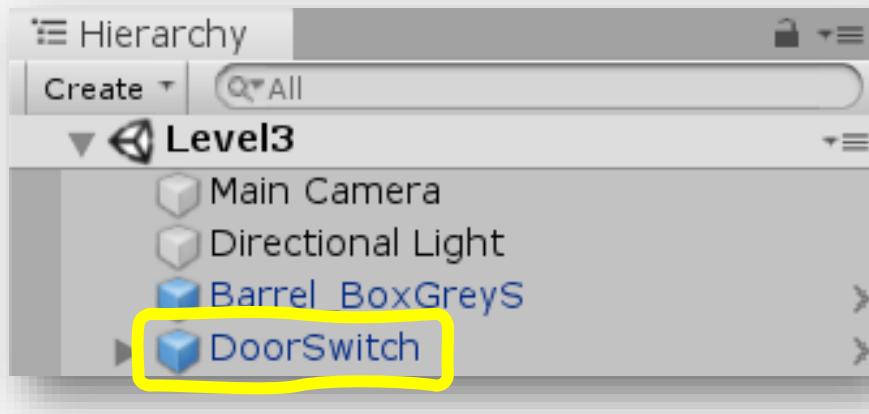


## 4 Playtest the game.

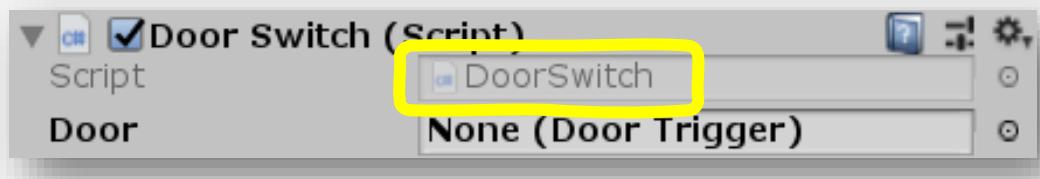
When you play, you'll notice the trigger to open the door doesn't work yet. You'll fix that in the next steps.

## 5 Let's connect the trigger to the door.

First, select the **DoorSwitch** game object in the Hierarchy.



## 6 In the Inspector, go to the **DoorSwitch (Script)** component and open the script.



## 7 The script has three functions and three variables.

The functions are designed to make the door switch change its color depending on whether the player has triggered the switch or not.

- 8** We are using `OnTriggerStay` for when the player is on the switch and doesn't leave. `OnTriggerStay` continues to run as long as something is colliding with it. The door switch will change its color from red to green.

```
//When any object enters the switch and stays, the switch icon is Green and the Red part is hidden.
Unity Message | – references
public void OnTriggerEnter(Collider other)
{
 StopAllCoroutines();
 switchIcon = this.transform.Find("green").gameObject;
 switchIcon.GetComponent<SpriteRenderer>().enabled = true;
 switchIcon = this.transform.Find("red").gameObject;
 switchIcon.GetComponent<SpriteRenderer>().enabled = false;

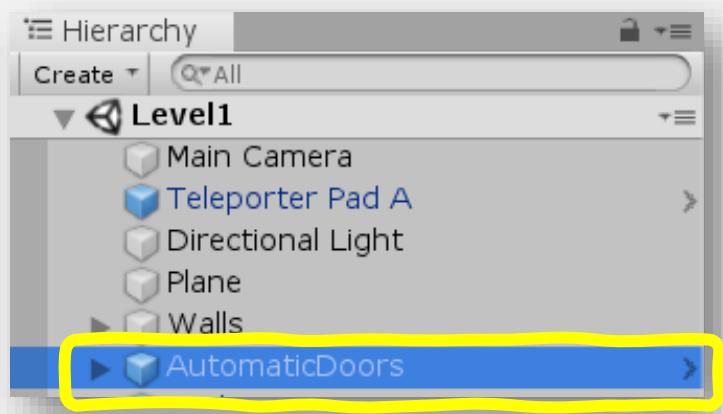
 //Have the door check if all of the switches are green
 door.SwitchCheck();
}
```

- 9** Return to the Unity Editor. With the **DoorSwitch** game object still selected, go to the **Door Switch (Script)** component in the **Inspector**.

- 10** On the **Door Switch (Script)** object, click the **Door** property's tiny circle to open the Selector window. Find and attach the "AutomaticDoors" object.



- 11** In the **Hierarchy**, select the **AutomaticDoors** game object.



**12** In the **Inspector**, find the **DoorTrigger** script.



**13** There is a variable called **Buttons** that is used for the door switches. Click on the triangle to the left of **Buttons** to see how many switches are in the scene.

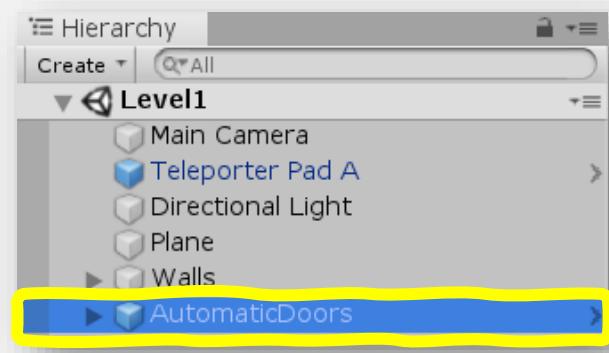


There is only one switch in the first part of Doctor Worm.

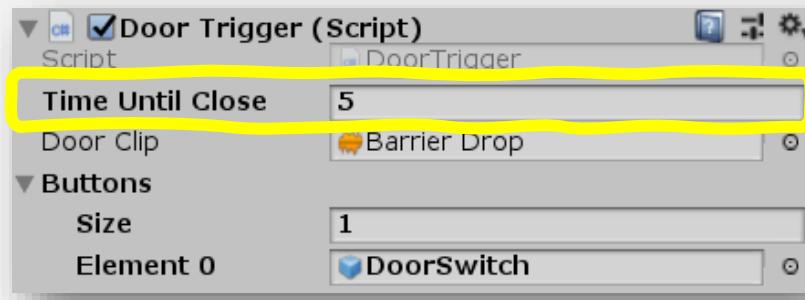
**14** **Playtest** your game by clicking the **play** button. Does the switch work? What happens when the player steps off the switch?

**15** Right now, the door closes 1 second after the player leaves the switch. That's not enough time to let the player leave the fortress!

- 16** Select the **AutomaticDoors** in the **Hierarchy**.



- 17** In the **Inspector**, find the **Door Trigger (Script)** component and change the value of **Time Until Close** from 1 to 5. This will leave the door open for 5 seconds giving Codey more time to escape.



- 18** **Playtest** the game. Now the player can make it to the exit!



*Congratulations, you have finished part 1 of the Evil Fortress of Doctor Worm! Continue your lesson with part 2 on the next page.*

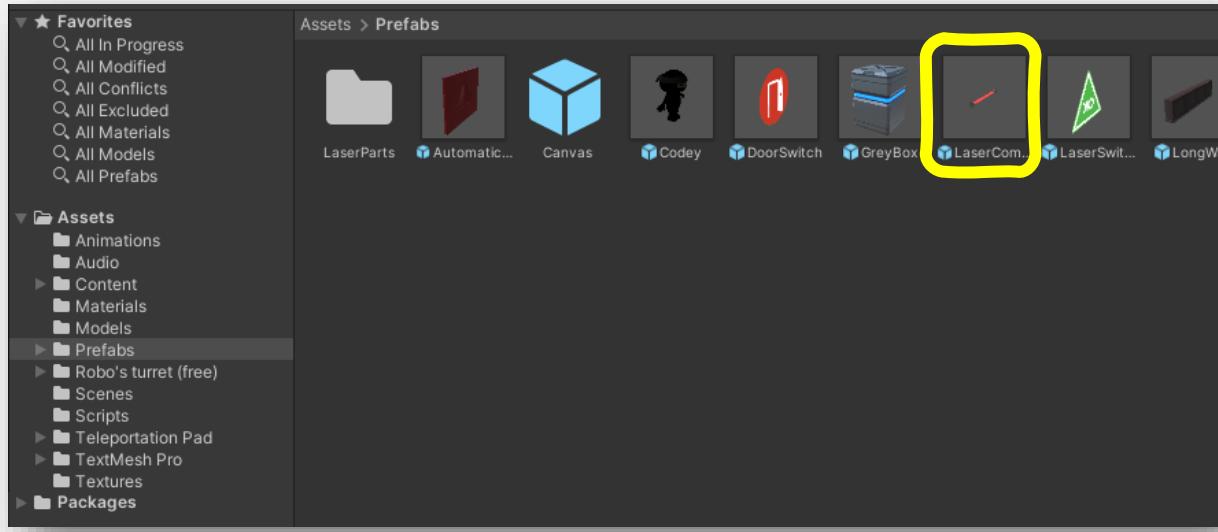
**19** Open the Unity Project that you used for part 1 of the Evil Fortress of Doctor Worm.

**20** In the **Project** tab within the **Assets > Scenes** folder, open **Level 2**.

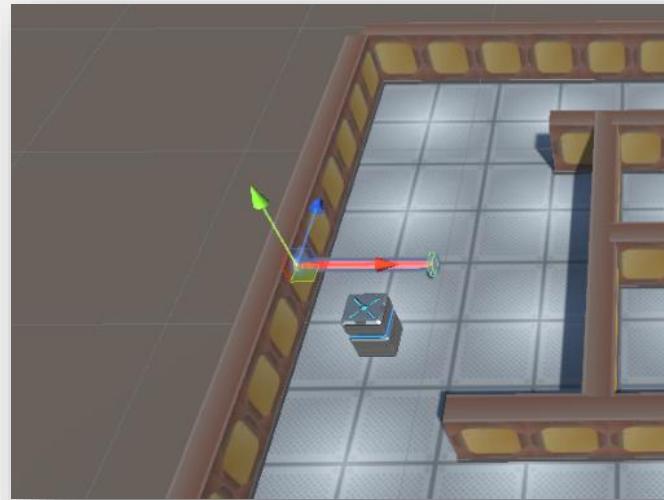


**21** This maze looks a bit empty! Let's add some challenge for the player by adding in a laser.

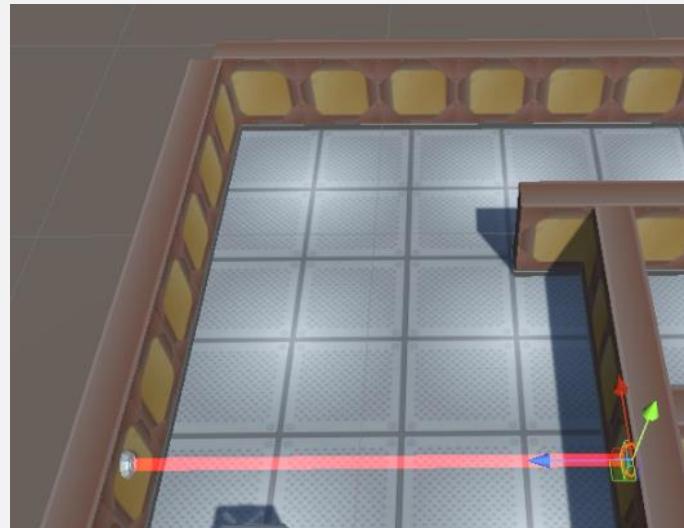
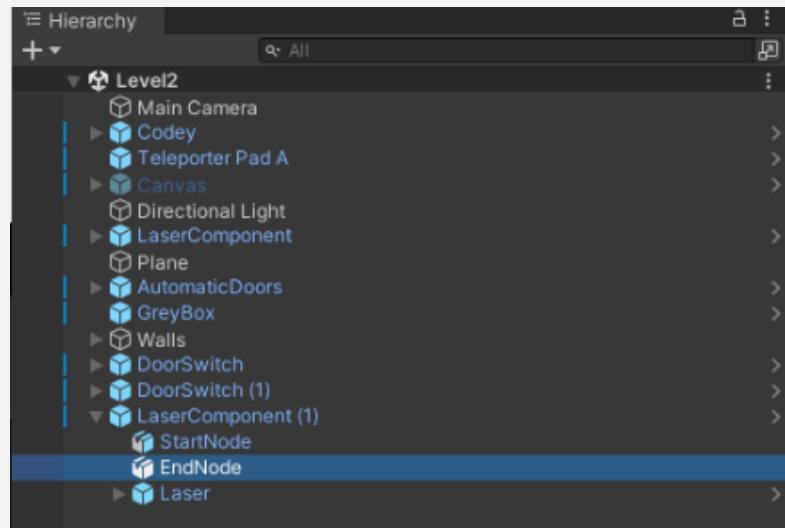
**22** In the **Project** tab, in the **Assets > Prefabs** folder, drag the **LaserComponent** object into the scene.



**23** Adjust the laser using the transform tool so it is against the left wall.



**24** Expand the prefab by clicking on the arrow next to it in the hierarchy, and then select the **EndNode**. Move the EndNode to the right wall.

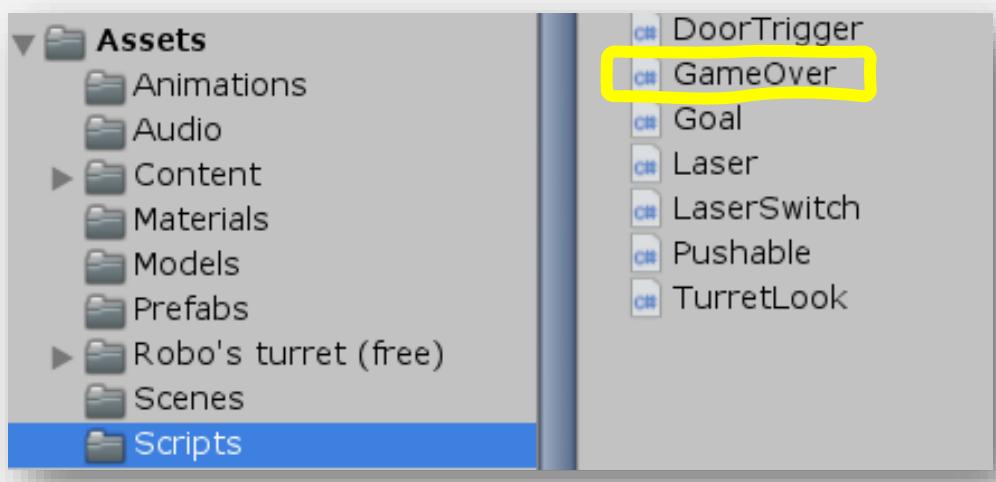


**25** Try adjusting the other laser in the scene to go all the way to the other wall!

**26** **Playtest** the game. What happens when Codey touches a laser?

**27** An evil mad scientist like Doctor Worm needs a little more drama than just resetting the scene. We can add an **animation** that plays whenever Codey touches the laser.

In the **Project** tab under **Assets > Script**, open the **GameOver** script.



**28** In the **Game Over** script, scroll down to the **PlayerHit** function.

Inside the function, there is a conditional `if (playerTouchedLaser)` that controls what happens when the player touches the laser.

```
public void PlayerHit()
{
 //Audio for hitting the player
 teleporterAudio.PlayOneShot(laserHit, 1.0f);
 StopAllAudio();

 if (playerTouchedLaser)
 {
 RestartLevel();
 }
}
```

**29** We don't want to restart the level immediately; we want Codey to react to touching the laser first. Above the `RestartLevel();` line, type `animator.SetTrigger("PlayerHit");` to tell Codey's animator to play the correct animation.

```
public void PlayerHit()
{
 //Audio for hitting the player
 teleporterAudio.PlayOneShot(laserHit, 1.0f);
 StopAllAudio();

 if (playerTouchedLaser)
 {
 animator.SetTrigger("PlayerHit");
 RestartLevel();
 }
}
```

**30** **Playtest** your game. What happens when Codey touches the laser now? Did anything change?

**31** We are asking Codey to play his hit animation, but we aren't giving him time to do it!

We are still immediately restarting the scene.

- 
- 32** In Unity, when you want to delay a function from running, use the **Invoke** function.

Change the `RestartLevel();` line to `Invoke("RestartLevel", 5);` to ask Unity to run the `RestartLevel` function 5 seconds after Codey touches the laser.

```
public void PlayerHit()
{
 //Audio for hitting the player
 teleporterAudio.PlayOneShot(laserHit, 1.0f);
 StopAllAudio();

 if (playerTouchedLaser)
 {
 animator.SetTrigger("PlayerHit");
 Invoke("RestartLevel", 5);
 }
}
```

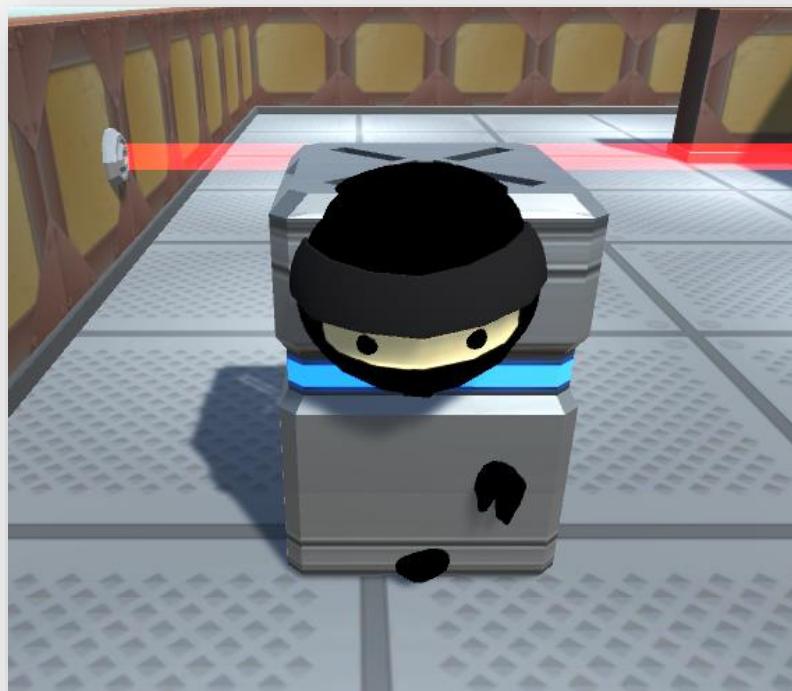
- 
- 33** **Playtest** your game.

Codey should now properly react when he touches the laser.

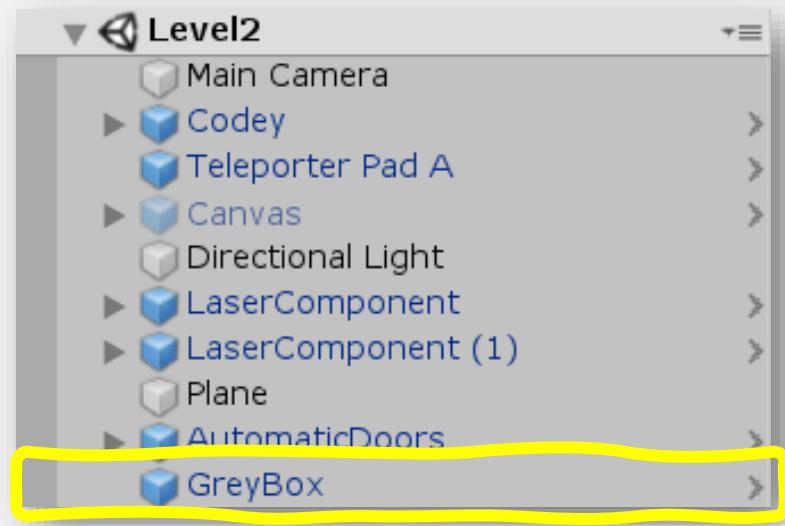
What happens if Codey tries to push the box in front of the laser?

---

**34** We need to code the boxes so Codey can push the boxes without running through them!

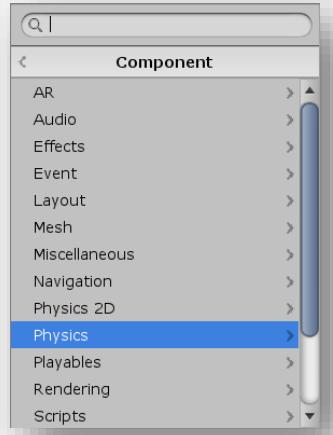


**35** Select the **GreyBox** game object in the **Hierarchy**.

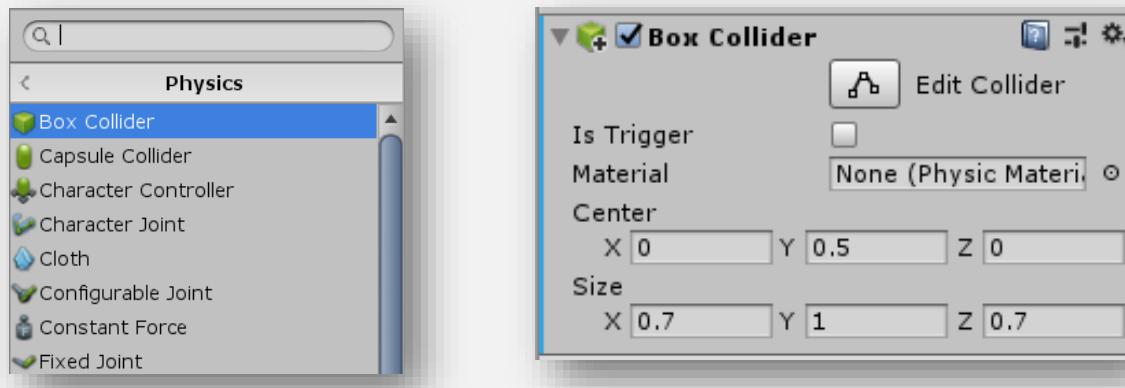


**36** In the **Inspector**, click on the  button.

**37** Click on **Physics**.



**38** Click on **Box Collider** to add a Box Collider component to the **GreyBox**.



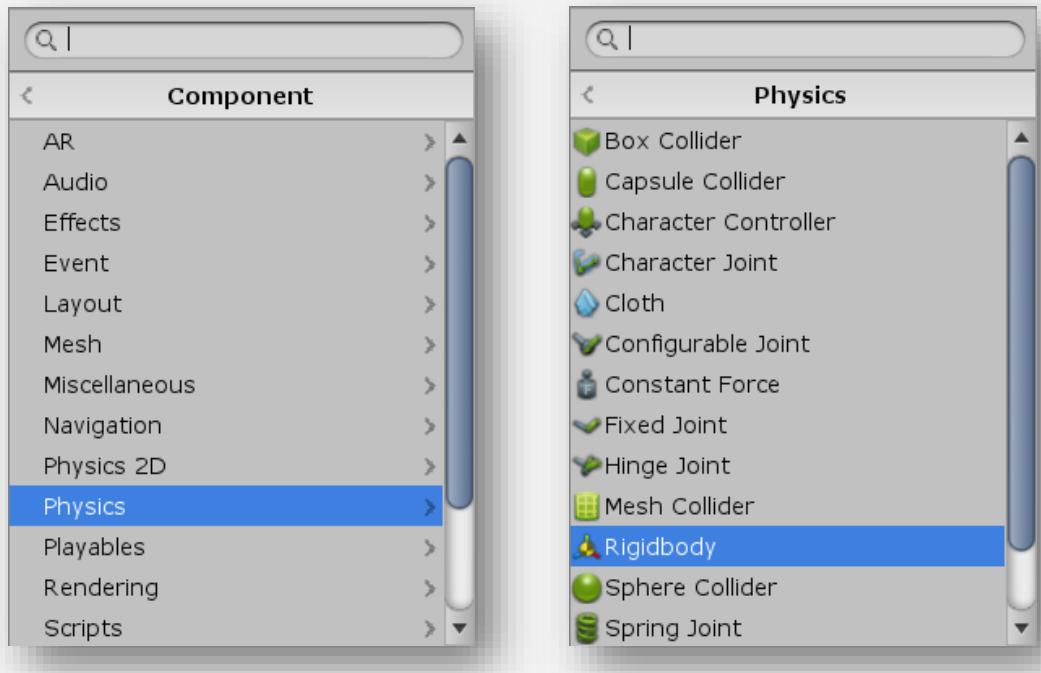
**39** **Playtest** your game and try pushing the box into the laser. Codey collides with the box, but it doesn't move!



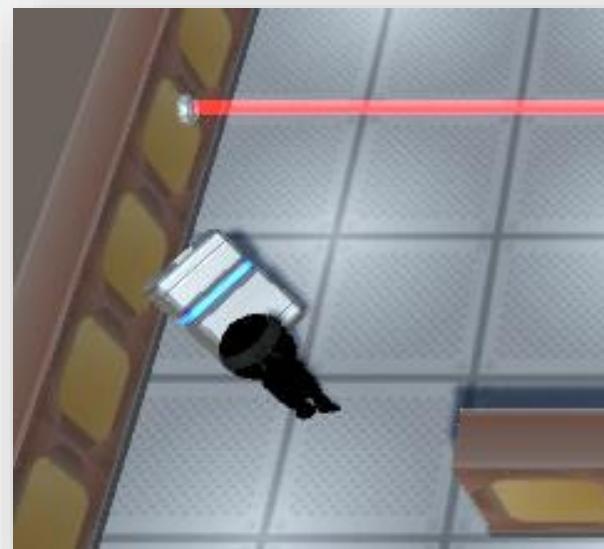
**40** While we were able to make Codey collide with the box, he can't push it yet. This is because we need to add one more component to the box to make it respond to physics!

**41** With the **GreyBox** selected in the **Hierarchy**, click on the **Add Component** button in the **Inspector**.

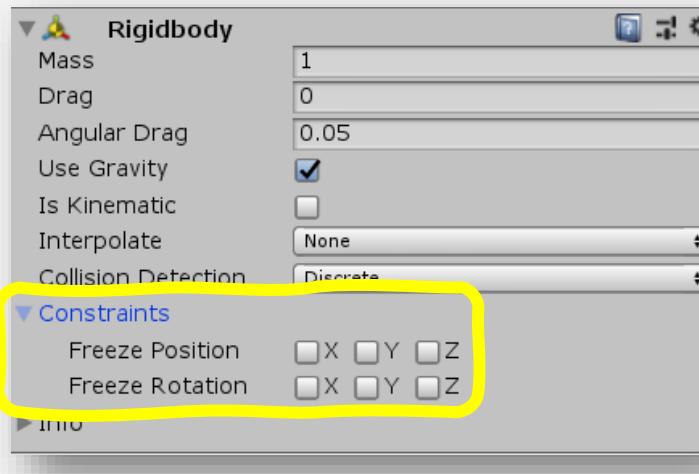
**42** Click on **Physics** and then **Rigidbody** to add a Rigidbody component.



**43** **Playtest** your game. What happens when Codey pushes the box? He can tip it over now!



**44** On the new **Rigidbody** component, click on the triangle next to **Constraints** to expand the properties.



**45** Check all three of the **Freeze Rotation** boxes to prevent the box from tipping over.



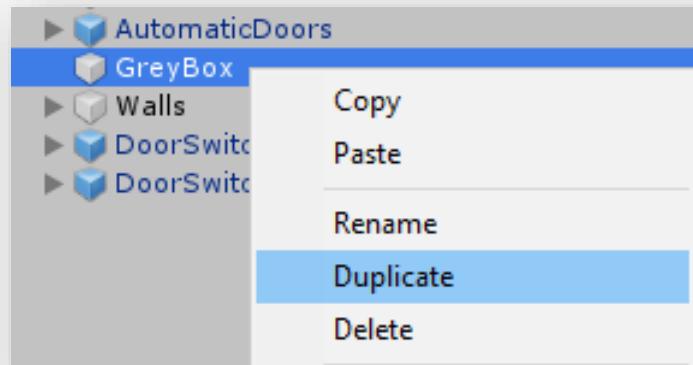
**46** Playtest the game.

Codey can now get by the first laser!

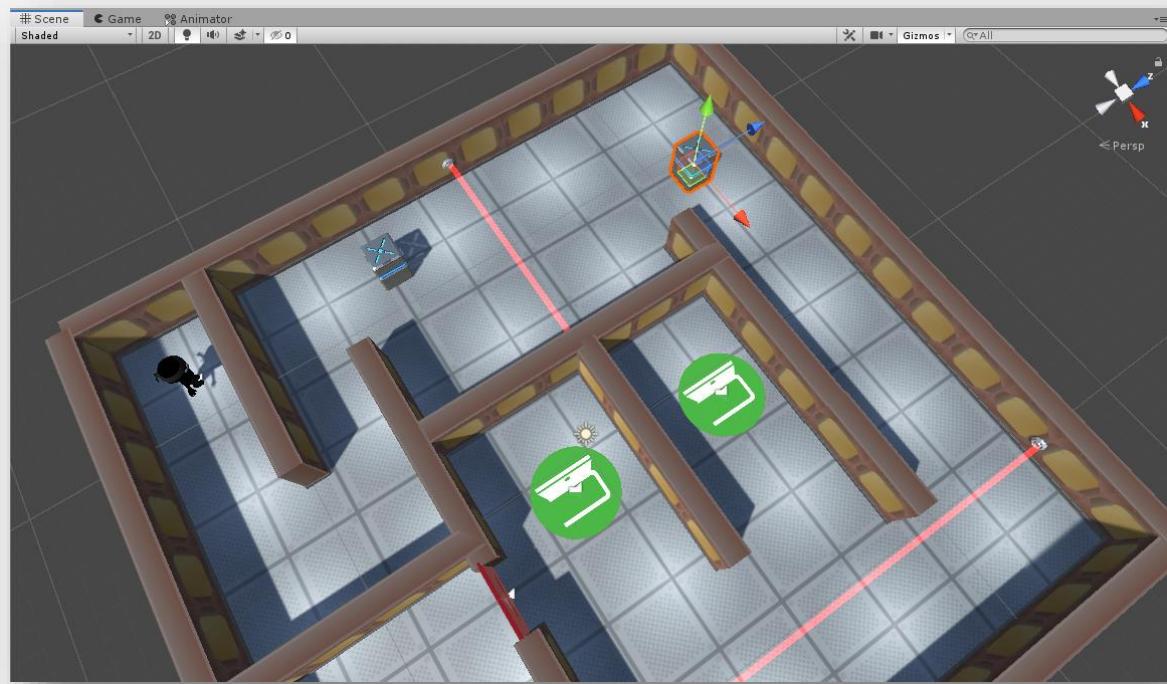
What about the second laser?

**47** We need to make a second **GreyBox** for Codey to push around!

**48** Right-click the **GreyBox** game object in the **Hierarchy** and select **Duplicate**.

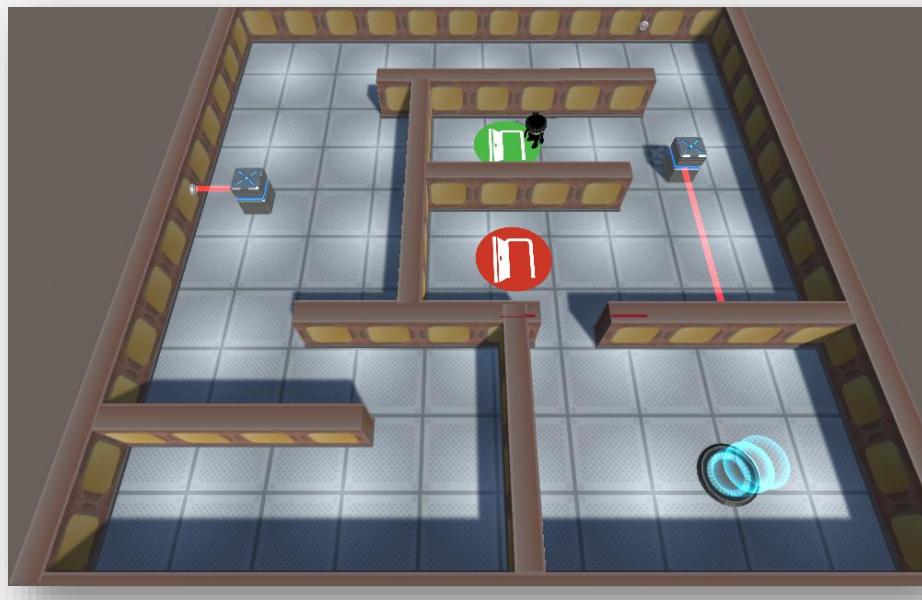


**49** Move the new box somewhere above the first laser in the Scene by dragging the blue and red arrows.

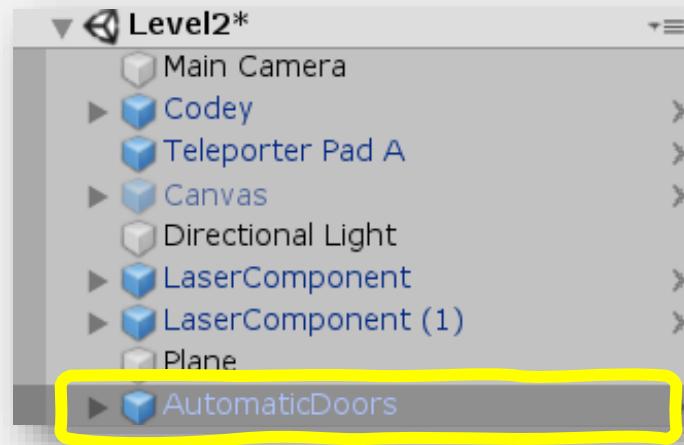


**50** **Playtest** the game. Codey can now get by both sets of lasers! Can he open the door yet?

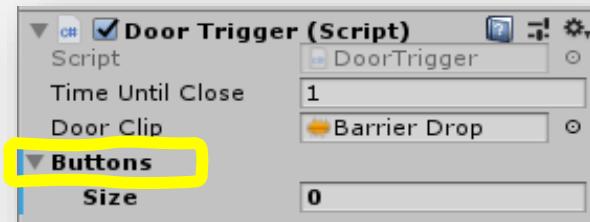
**51** Codey can open the door by standing on just one of the switches. Doctor Worm needs to fix his fortress!



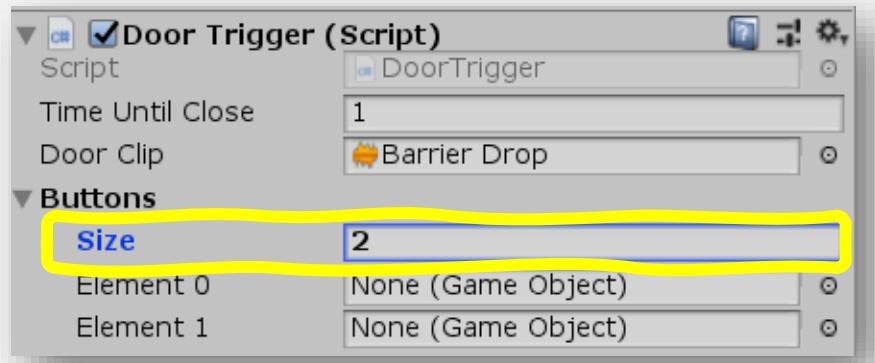
**52** Select the **AutomaticDoors** object in the **Hierarchy**.



**53** Find the **Door Trigger (Script)** and expand the **Buttons** property.



**54** Change the value of **Size** from 0 to 2.



**55** Attach the **DoorSwitch** object to the **Element 0** property and the **DoorSwitch (1)** object to the **Element 1** property.

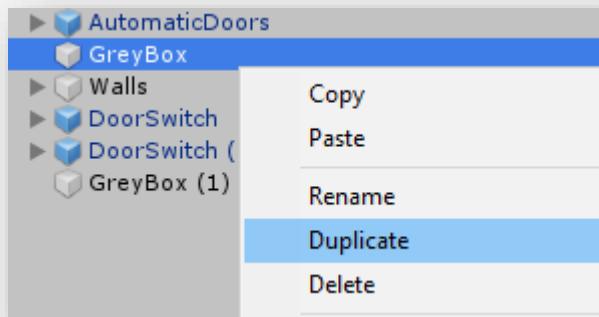
Either drag the objects from the **Hierarchy** to the property in the **Inspector** or click on the little circle to open the **Game Object Selector**.



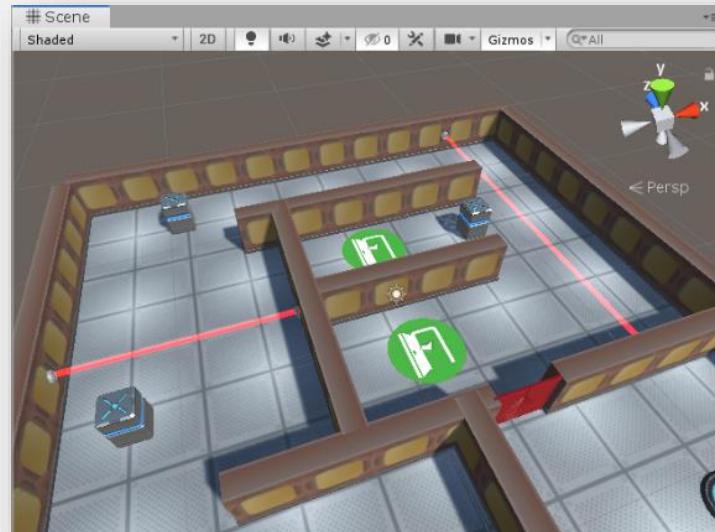
**56** Playtest the game. The switches are now properly wired to the door and Codey can't escape!



- 57** How can Codey get out?! We can duplicate the GreyBox one more time. Right-click the **GreyBox** object in the **Hierarchy** and select **Duplicate**.



- 58** Move the new **GreyBox** somewhere near the two switches using the blue and red arrows.

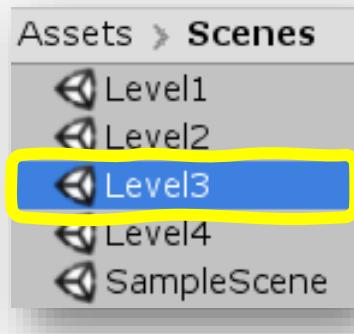


- 59** Playtest the game. Codey can successfully escape!

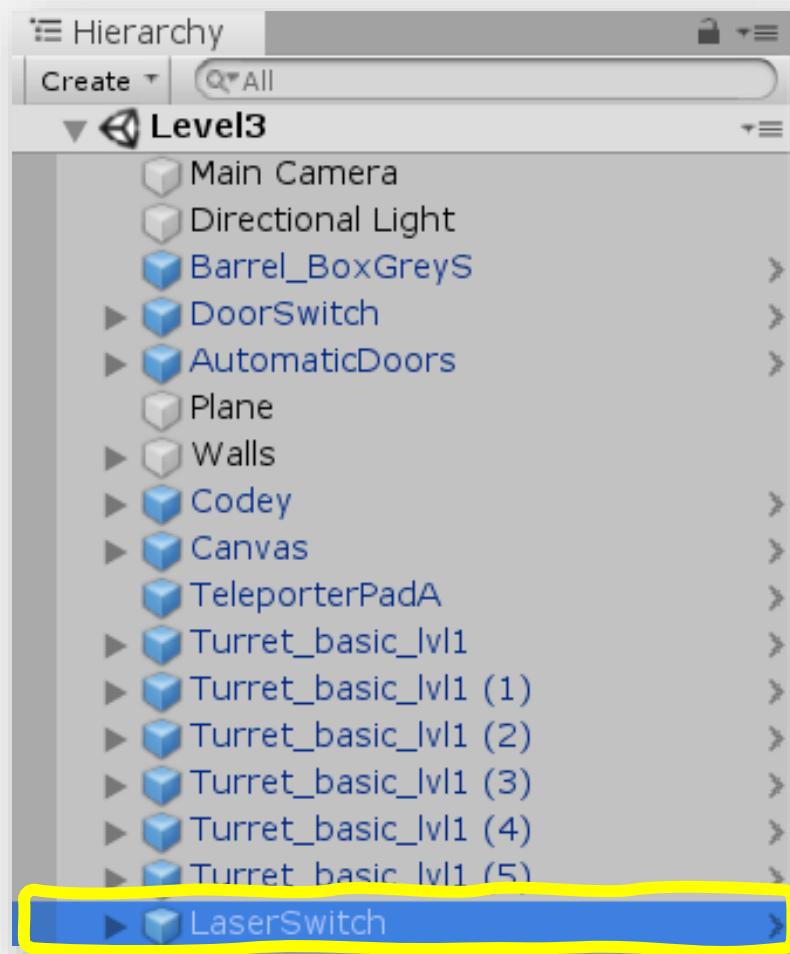
*Codey's adventure continues in the Evil Fortress of Doctor Worm part 3! Begin part 3 on the next page.*



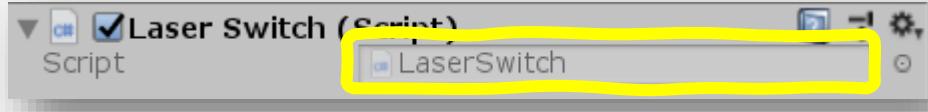
**60** Open your Fortress of Doctor Worm Unity project and load the Level 3 scene.



**61** Let's get the laser power switch working first. Select the **LaserSwitch** in the **Hierarchy**.



- 62** In the **Inspector**, find the **Laser Switch (Script)** component and open the attached script in Visual Studio.



- 63** We need to add a Boolean to the LaserSwitch script that we can use to inform the laser objects to turn off when Codey steps on the switch.

After `private AudioSource playBeep;` type `public bool lasersAreOff = false;` to create a Boolean with a value of false.

```
private GameObject switchIcon;
private AudioSource playBeep;

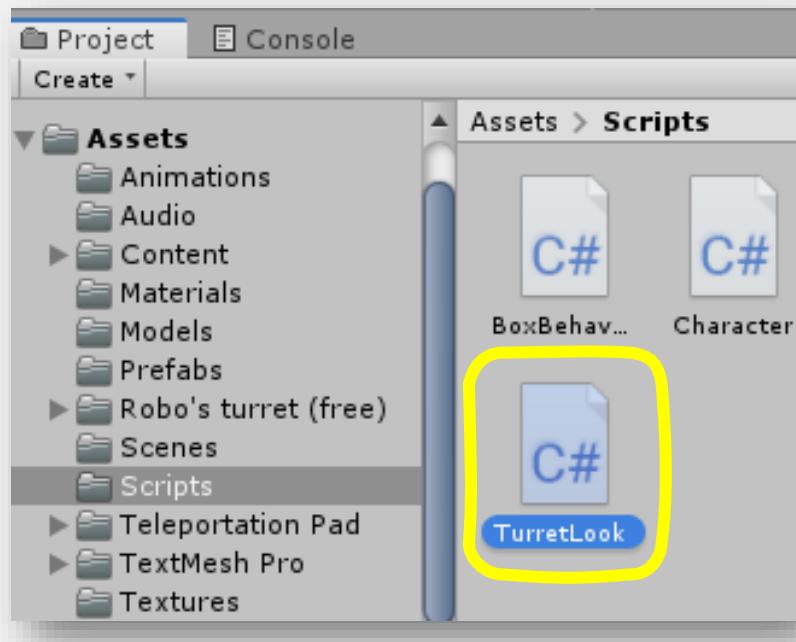
public bool lasersAreOff = false;
```

- 64** We want to turn the lasers off when Codey or the box enters the switch. In the `OnTriggerEnter` function, add `lasersAreOff = true;` after the `playBeep.Play();` line.

```
public void OnTriggerEnter(Collider other)
{
 playBeep.Play();
 lasersAreOff = true;
}
```

**Save** your script.

- 65** Each laser turret has a script called **TurretLook**. To open it, find it in the **Assets > Scripts** folder and double-click.



- 67** In the script's `Update` function, create an `if` statement that checks to see if the `laserSwitch`'s `lasersAreOff` Boolean is true by typing `if (laserSwitch.lasersAreOff) { }` making sure to leave an empty line in between the curly brackets.

```
void Update()
{
 if (laserSwitch.lasersAreOff)
 {
 transform.LookAt(player);
 }
}
```

- 
- 68** Code added within this `if` statement will execute if the switch has turned the lasers off.

We need to disable the lasers for Codey, so type `laser.SetActive(false);` inside the curly brackets.

```
void Update()
{
 if (laserSwitch.lasersAreOff)
 {
 laser.SetActive(false);
 }
 transform.LookAt(player);
}
```

- 
- 69** Play your game. The lasers turned off, but the turrets still follow Codey!



**70** We can make sure the turrets are shut down for good with one simple line of code. After the `laser.SetActive(false);` line, type `return;` This exits the Update function early and prevents the `transform.LookAt(player);` code from running.

```
void Update()
{
 if (laserSwitch.lasersAreOff)
 {
 laser.SetActive(false);
 return;
 }
 transform.LookAt(player);
}
```

**71** **Playtest** your game. The turrets are now completely disabled!



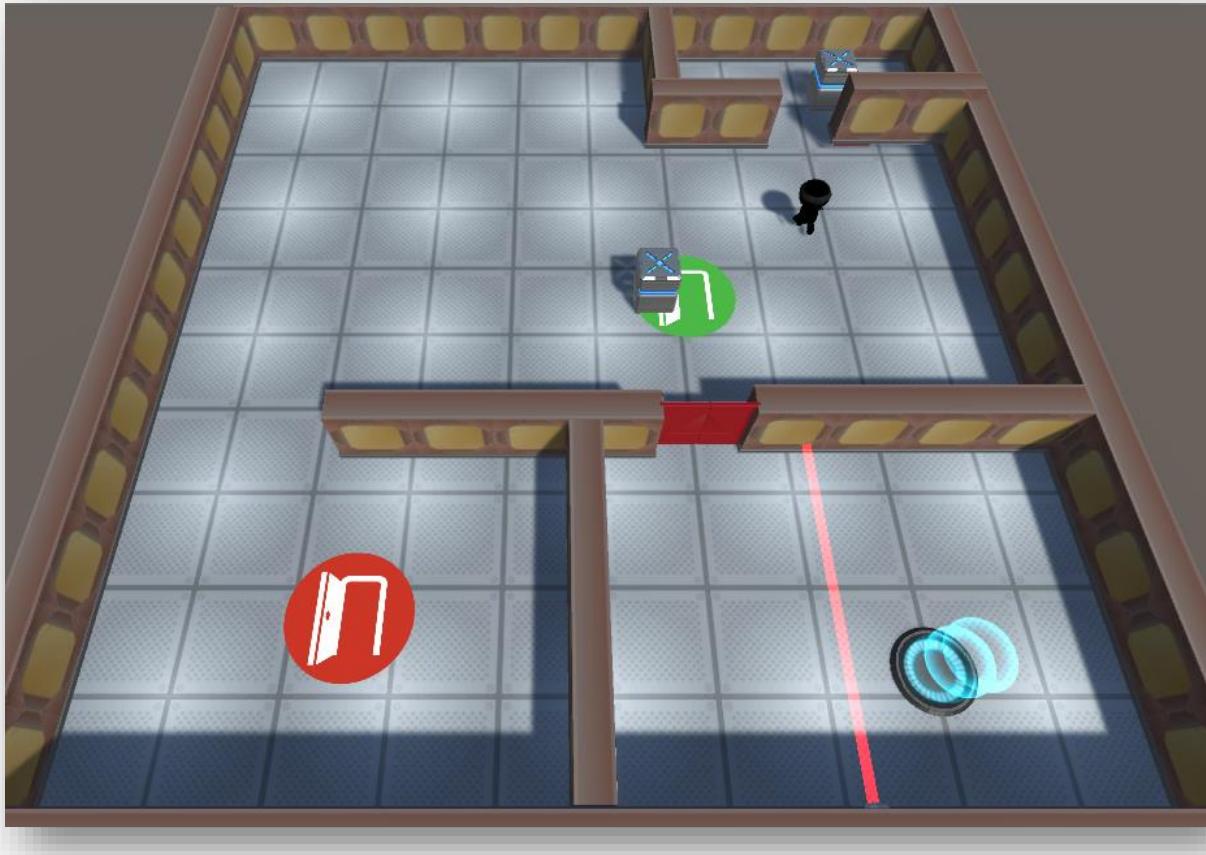
# Prove Yourself

## Get Started

- Open the Level 4 scene
- Initially the scene just has Codey, a door and an exit.

## Task

Evil Fortress of Doctor Worm has a fourth scene, but it's empty. Use what you've learned to set up a new room with another puzzle for the player to solve. Make sure to use the boxes, switches and lasers located in the Prefabs folder. You might need to add scripts and other components from parts 1, 2, and 3 to the objects.

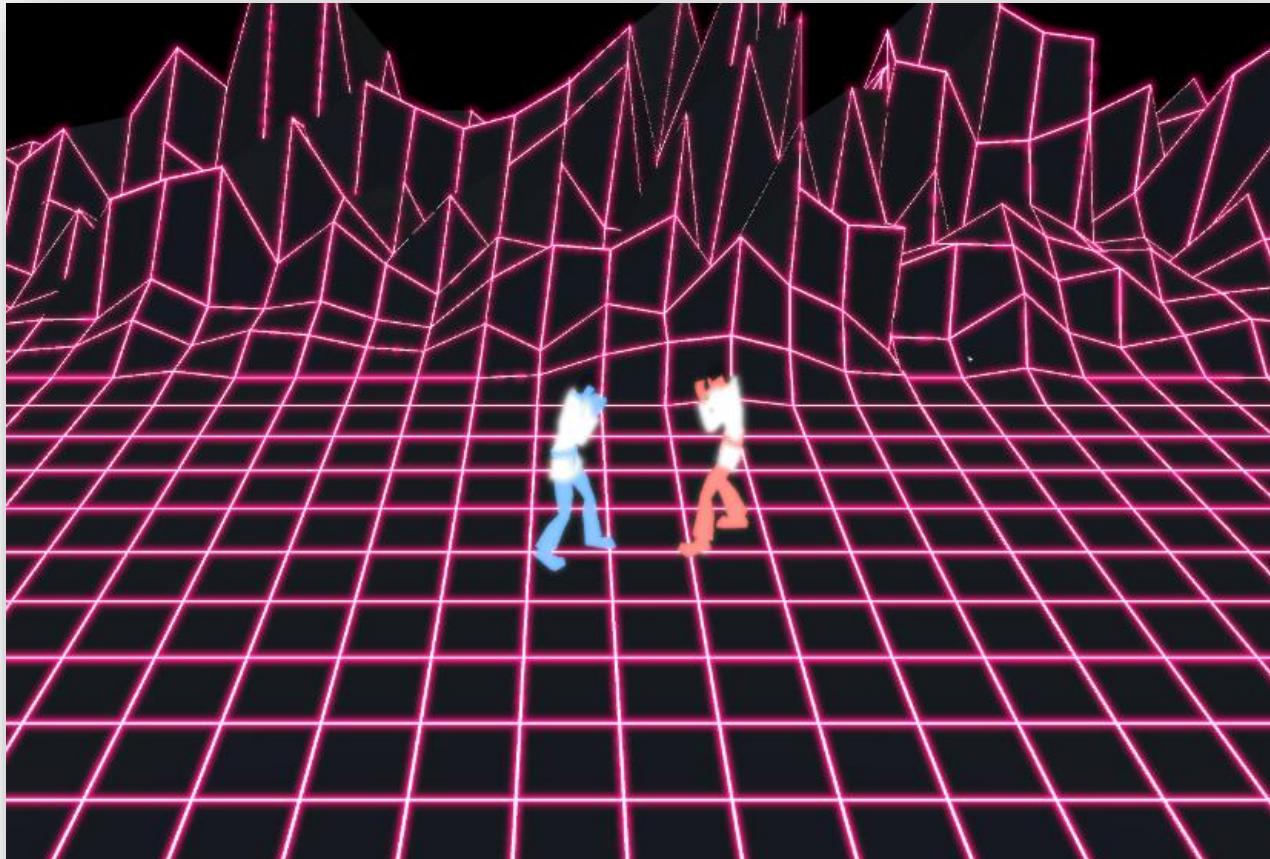


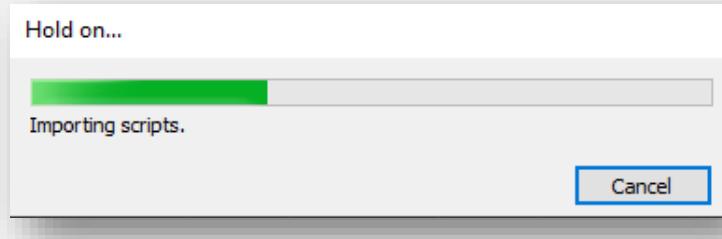
## Activity 7

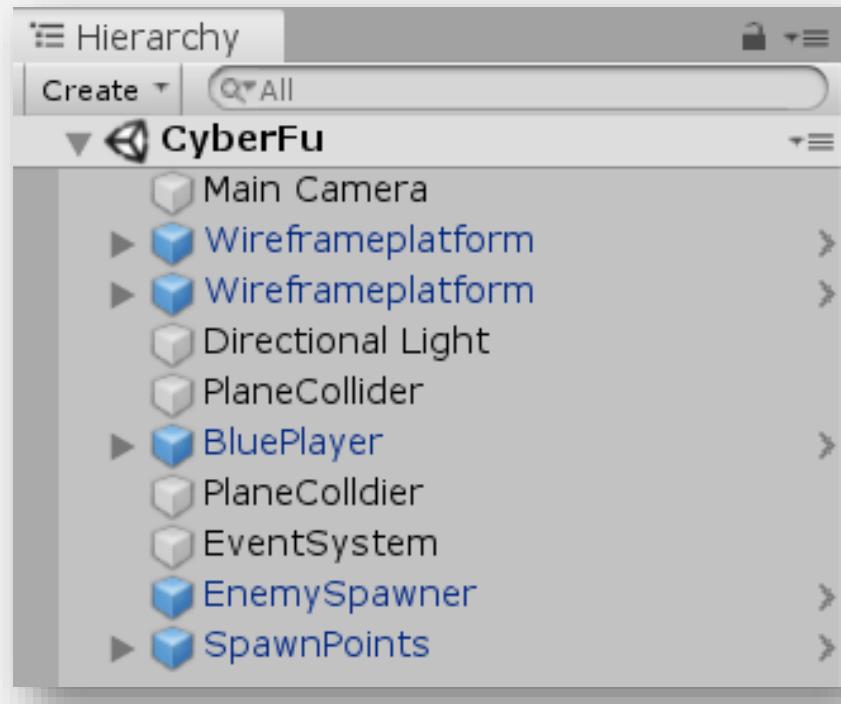
# CyberFu - Part 1

If you press **Play**, the enemy objects that are spawned in the game aren't moving! The enemies can't move or follow the player yet because there is no movement code.

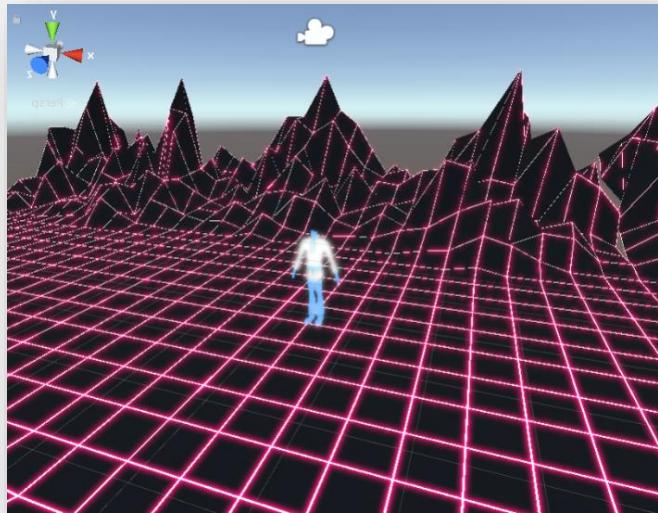
New mission: Code your enemy to follow and chase the player. This is a key step in making your game because it will cause the enemies to move towards the player by calculating how far each enemy is!



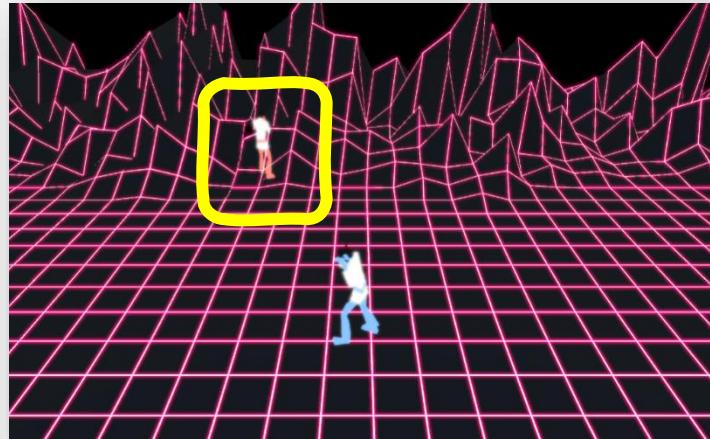
- 
- 1** Start a new Unity Project and name it **YOUR INITIALS - CyberFu**.  
Select **3D core**.
  - 2** Import the **CyberFu-Part1** starter Unity Package by going to **Assets > Import Package > Custom Package > All > Import**.
- 
- A screenshot of a Unity import progress dialog titled "Hold on...". It shows a green progress bar partially filled, with the text "Importing scripts." below it. A blue "Cancel" button is at the bottom right.
- 
- 3** After it is done importing, double click the **CyberFu** scene. Located in the **Project** tab in the **Scenes** folder inside of the **Assets** folder.
  - 4** When all the assets have been imported into the project, open the scene "**CyberFu**" and you will see that the game is already set up with a couple of objects shown in the **hierarchy** and **scene**.



- 5** Before we start configuring our **enemy**, start the game to see what happens when we **play** the game as is.



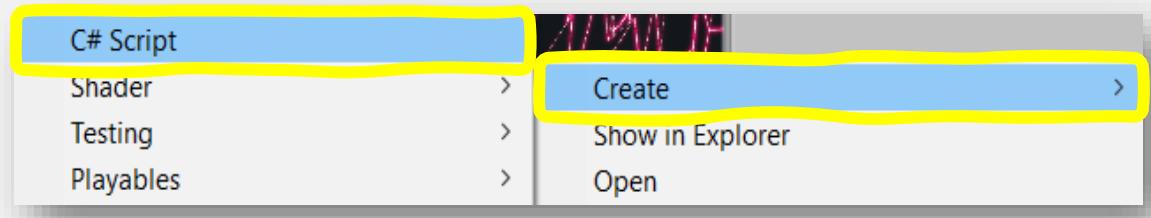
- 6** As you can see, when the game starts **enemies** will **spawn** at one of three **spawn points**. All the **enemies** are the same **prefab** we will use throughout the lesson. When the **enemy** is **spawned** into the game, it's not facing nor following the player.



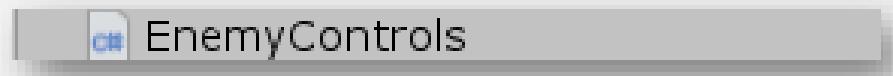
- 7** We are going to have our **enemies** follow and attack the **player**, so let's go ahead and start controlling our **enemies** to do so.

- 8** First, go into your **Scripts** folder.

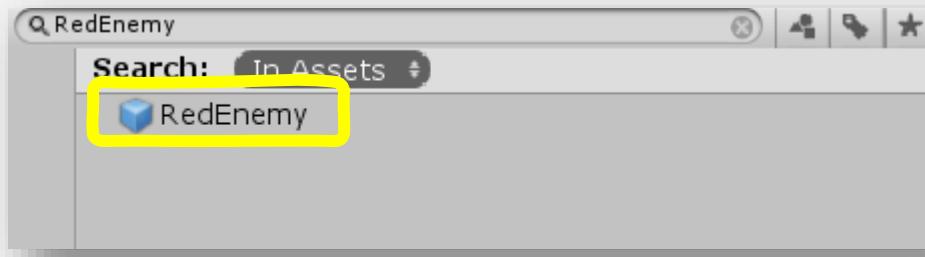
**9** Next, right-click an empty space. Go to “Create” and select “C# Script”.



**10** Rename it to “EnemyControls”.



**11** Next, let's find our **enemy**. Instead of searching through many folders to find the **enemy prefab**, let's use the **search bar** in the **project**. Type “**RedEnemy**” and select the **prefab** model.



**12** Now, let's add the “**Enemy Controls**” **script** to the **enemy prefab**. To do that, in the “**Inspector Tab**”, scroll down and click on “**Add Component**”.



**13** In the **search bar**, type “**EnemyControls**” and select the **component** from the **drop down** **menu**.



**14** The **enemy controls script** is now part of the **enemy prefab**.



**15** We are going to get started with **scripting** our **enemy** with **public variables** first.

These **variables** will be inserted between the first bracket and the **void Start** function.

**16** The **first variable** will be for setting the **speed** of the **enemy**:

```
//the set speed of the enemy
public float speed = 2f;
```

**17** The **second variable** will be for the **set attacking distance** for the **enemy**:

```
//the set attacking distance for the enemy
public float attackingDistance = 0.6f;
```

**18** This is what the code should look like for you without comments.

```
public class EnemyControls : MonoBehaviour
{
 public float speed = 2f;
 public float attackingDistance = 0.6f;

 // Start is called before the first frame update
 void Start()
 {
 }
}
```

A screenshot of the Unity Editor's code editor showing the C# script for "EnemyControls". The script starts with "public class EnemyControls : MonoBehaviour". Inside the class, there are two public float variables: "speed" and "attackingDistance", both initialized to their respective values. Below these variables is a "Start" method. The entire code block is shown in a dark-themed code editor.

**19**

Now, let's add **private variables** to the **enemies**. We are going to start with the **components**.

**20**

The **first variable** will be for the **Animator component** for the **enemy**. The **animator component** controls the **animations** for the **enemy**.

```
//the animator component controls the animations for the enemy
private Animator animatorEnemy;
```

**21**

The **second variable** will be for the **Rigidbody** for the **enemy**. The **rigidbody component** controls the **physics** for the **enemy**.

```
//the rigidbody component controls the physics for the enemy
private Rigidbody rigidbodyEnemy;
```

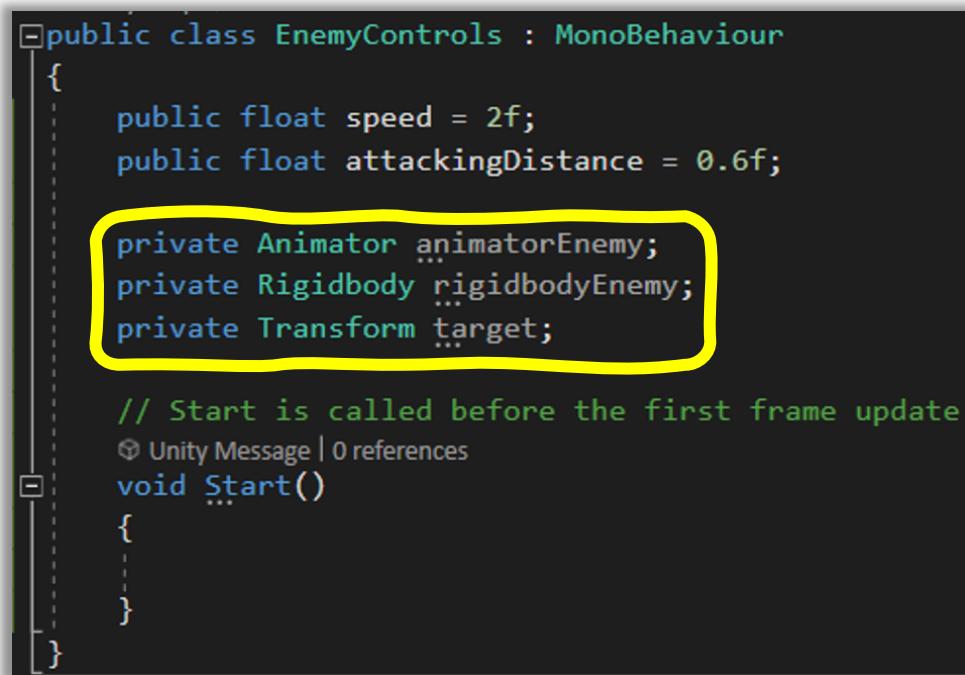
**22**

The **third variable** will be for position, rotation, and size of the object. This variable allows the enemy to figure out who it is targeting.

```
//this variable will help the enemy target any object with the
variable
private Transform target;
```

**23**

This is what your **script** should look like so far without comments.



```
public class EnemyControls : MonoBehaviour
{
 public float speed = 2f;
 public float attackingDistance = 0.6f;

 private Animator animatorEnemy;
 private Rigidbody rigidbodyEnemy;
 private Transform target;

 // Start is called before the first frame update
 void Start()
 {
 }
}
```

**24** Now, we will use a **boolean** to track if the enemy is following or not. A **boolean** is a type of variable that can be one of two values, either **true** or **false**.

**25** Declare the variable above Start.

```
//If the enemy is following the target or not
private bool isFollowingTarget;
```

**26** What your **script** should look like without comments:

```
public float speed = 2f;
public float attackingDistance = 0.6f;

private Animator animatorEnemy;
private Rigidbody rigidbodyEnemy;
private Transform target;

private bool isFollowingTarget;
```

**27** The last two **variables** we will add will be for **attacking** the **target**.

**28** The **first variable** will be for **current value** of what the current time is for **attacking** the **target**.

```
//current value of what the current time is for attacking
private float currentAttackingTime = 0f;
```

**29** The **second variable** will be the **set value** of the maximum time of **attacking**.

```
//the set value of the maximum time of attacking
private float maxAttackingTime = 2;
```

**30** Here is what your script should look like without comments:

```
public float speed = 2f;
public float attackingDistance = 0.6f;

private Animator animatorEnemy;
private Rigidbody rigidbodyEnemy;
private Transform target;

private bool isFollowingTarget;

private float currentAttackingTime = 0f;
private float maxAttackingTime = 2f;
```

**31** Add **three variables** into the **Start** function.

The first two variables will be for the components of the **enemy** and the third will be the **gameObject** that will be the **target** for the **enemy** to **follow** and **attack**.

The first **variable** we will add is getting the **component** for the **animator**.

```
//getting the component for the animator
animatorEnemy = GetComponent<Animator>();
```

**32** The **second variable** is getting the **component** for the **rigidbody**.

```
//getting the component for the rigidbody
rigidbodyEnemy = GetComponent<Rigidbody>();
```

## 33 The third variable is the **target**.

This is different from the other variables because we will set the target for the enemy to **follow** and **attack** to the player. We'll find the player using the Player tag that is only assigned to the player gameobject.

We do "**target = ...**" that way we get the position where the game object with the tag "**Player**" is located.

```
//set the target for the enemy to follow and attack to the
player by tag

target = GameObject.FindGameObjectWithTag("Player").transform;
```

## 34 Here is what the function should look like without comments.

```
Unity Message | 0 references
void Start()
{
 animatorEnemy = GetComponent<Animator>();
 rigidbodyEnemy = GetComponent<Rigidbody>();
 target = GameObject.FindGameObjectWithTag("Player").transform;
}
```

## 35 So, now that we have set up our enemy to where the **target** is **player**. Let's get our enemy to follow the player.

## 36 In the Update method, let's make the enemy face the player. We can do this by using the LookAt method. This method rotates a transform to face any position. In this case, we want the enemy transform to rotate towards the player's position. Add this line inside the **Update** method.

```
// Update is called once per frame
Unity Message | 0 references
void Update()
{
 transform.LookAt(target.position);
}
```

**37** Save your script and return to Unity. Press play. The enemies should now be rotating towards the player. But they are still standing still....

**38** In the Update method, we need to determine if the enemy should be following or attacking. We can determine this based on the distance that the player is from the enemy. We can tell the distance between two positions using **Vector3.Distance()** method. Add this line in Update.

```
isFollowingTarget = Vector3.Distance(transform.position,
target.position) >= attackingDistance;
```

**39** Now use an if statement to check if the enemy is following the player. Add the if statement below the previous line in Update.

```
if (isFollowingTarget)
{
}
}
```

**40** Inside the if statement, we are going to use the rigidbody's velocity to move the enemy towards the player. Since the enemy is already facing the player, we can use transform.forward as the direction for them to move. Add this line inside the if statement.

```
if(isFollowingTarget)
{
 rigidbodyEnemy.velocity = transform.forward * speed;
}
```

**41** Here is what your script looks like without comments:

```
void Update()
{
 transform.LookAt(target.position);

 isFollowingTarget = Vector3.Distance(transform.position, target.position) >= attackingDistance;

 if (isFollowingTarget)
 {
 rigidbodyEnemy.velocity = transform.forward * speed;
 }
}
```

**42** Before we **playest** the game, let's see if the enemy is going to attack the player by checking the **boolean "isFollowingTarget"**.

To see `isFollowingTarget` while playing, let's expose the variable to the inspector. Normally, a private variable is hidden and doesn't show up in the inspector. We can make this variable show up in the inspector by using `SerializeField`. Place this above the declaration for `isFollowingTarget`.

```
[SerializeField]
private bool isFollowingTarget;
```

**43** **Save** the script and **play** the game to see if it works.

**44** Our enemy should now look at **the player, and move towards them until they get to attack range**.



**45** Even though we got our enemy to **move** and **follow** the player, there are still some problems. The enemy does not stop walking when they are close to the player, nor do they attack. Let's update the script.

- 46** First thing we are going to add is an **else statement** to where the enemy's position is **less than or equal to** the **attacking distance** to attack the player. So, below the **if statement**, add:

```
//If enemy's distance is close to attack the player

else
{
}
}
```

- 47** Since Attacking will have a lot of code, it's a good idea to separate this into its own method. Create a `void Attack()` method at the bottom of our script.

- 48** Next, inside the **Attack method** we are going freeze the enemy's **rigidbody** to where it cannot move from its spot. We will use **Vector3.zero** and have all of the **values** of all velocity in each direction 0.

```
//Keeps the enemy from moving within its place

rigidbodyEnemy.velocity = Vector3.zero;
```

- 49** Your Attack method should look like this so far.

```
void Attack()
{
 rigidbodyEnemy.velocity = Vector3.zero;
}
```

- 50** Add a call to the attack method in the else section in the Update method.

```
else
{
 Attack();
}
```

**51** Save your script and return to Unity. The enemy should stop as soon as it gets in attacking range of the player.

**52** To have the enemy attack the player, we will use some attack animations. Before we do that, let's use the animation for walking that is already created for us.

Walking is a boolean, so we can re-use the `isFollowingTarget` boolean to determine if the walk animation should be playing.

**53** Add this line below the `else` statement in `Update`.

```
else
{
 Attack();
}

animatorEnemy.SetBool("Walk", isFollowingTarget);
```

**54** Save the script and check in Unity. The enemies should have a walking animation until they get in attack range.

**55** Now let's have the enemies attack! There are already some attack animation created for our player, we just have to play one when they get in range, and the attack cooldown is ready.

**56** In our `Attack` method, have the **current attacking time** add to the **value of delta time**. Delta time is the difference in time between frames, so this is like adding the number of milliseconds every time.

```
//the current attacking time add to the value of delta time
currentAttackingTime += Time.deltaTime;
```

- 
- 57** Add an **if statement** to where if the **current attacking time** is greater than the **max value** of attacking time .

The reason why we need to add this is because if the **current attacking time** equals to **2**, we want to have the enemy do something.

```
//if the current attack time is greater than the max attack time
if (currentAttackingTime > maxAttackingTime)
{
}
}
```

- 
- 58** Inside the **if statement**, set the value of the **current attack time** to **0**. Every time the **current attack time** is set **2**, it will reset to the value to **current attacking time** to **0**.

```
//set the current attacking time to 0
currentAttackingTime = 0f;
```

- 
- 59** Let's add an **attack** as well. This **attack animation** will only trigger once every time the **current attacking time** equals to **2**. We only want the enemy to attack once **every 2 seconds**, so instead of using **SetBool** for a boolean, which will continue to play the animation over and over, we will use **SetTrigger** to execute the animation only once.

```
//trigger the enemy's attack animation
animatorEnemy.SetTrigger("Attack1");
```

- 60** Here is what the script should look like:

```
void Attack()
{
 rigidbodyEnemy.velocity = Vector3.zero;

 currentAttackingTime += Time.deltaTime;

 if (currentAttackingTime > maxAttackingTime)
 {
 currentAttackingTime = 0f;
 animatorEnemy.SetTrigger("Attack1");
 }
}
```

- 61** Save the script and play the game to test it.

- 62** There are more **attack animations** for the enemy and we want the enemy to use them. Let's go back to the **enemy control script** and update it.

- 63** All of the attacks are named "Attack" with a number at the end. For example, "Attack1", "Attack2", "Attack3", etc. Therefore, we can get a random number to determine what attack we will use. To do this, we will use Unity's built in **Random** method.

- 64** We will use **Random.Range** to get the random number. The first parameter is the minimum number inclusive, and the second is the maximum number exclusive. Inclusive means that it **can** be the random number, and exclusive means it **cannot** be the random number. Therefore, if we want a random number to either be 1, 2, or 3, then we need to set the minimum to 1 (inclusive) and the maximum to 4 (exclusive).

- 65** Go back to the Attack method. Above the "SetTrigger" line, create a variable to store the random number. Add the line below.

```
int rand = Random.Range(1, 4);
```

- 66** Now, in the “SetTrigger” line, we can replace the “Attack1” string with “Attack” string plus the random number we generated. Change the line so it reads as follows.

```
AnimatorEnemy.SetTrigger("Attack" + rand);
```

- 67** Your script should look like this.

```
void Attack()
{
 rigidbodyEnemy.velocity = Vector3.zero;

 currentAttackingTime += Time.deltaTime;

 if (currentAttackingTime > maxAttackingTime)
 {
 currentAttackingTime = 0f;
 int rand = Random.Range(1, 4);
 animatorEnemy.SetTrigger("Attack" + rand);
 }
}
```

- 68** Save your script. Return to Unity and playtest your game! The enemies should throw out different attacks.

- 69** Congratulations, you have complete part 1 of the lesson for CyberFu.

In part 2, you will learn how to add **health** and **damage** to both the enemy and the player.

# Prove Yourself

## Add More Attacks

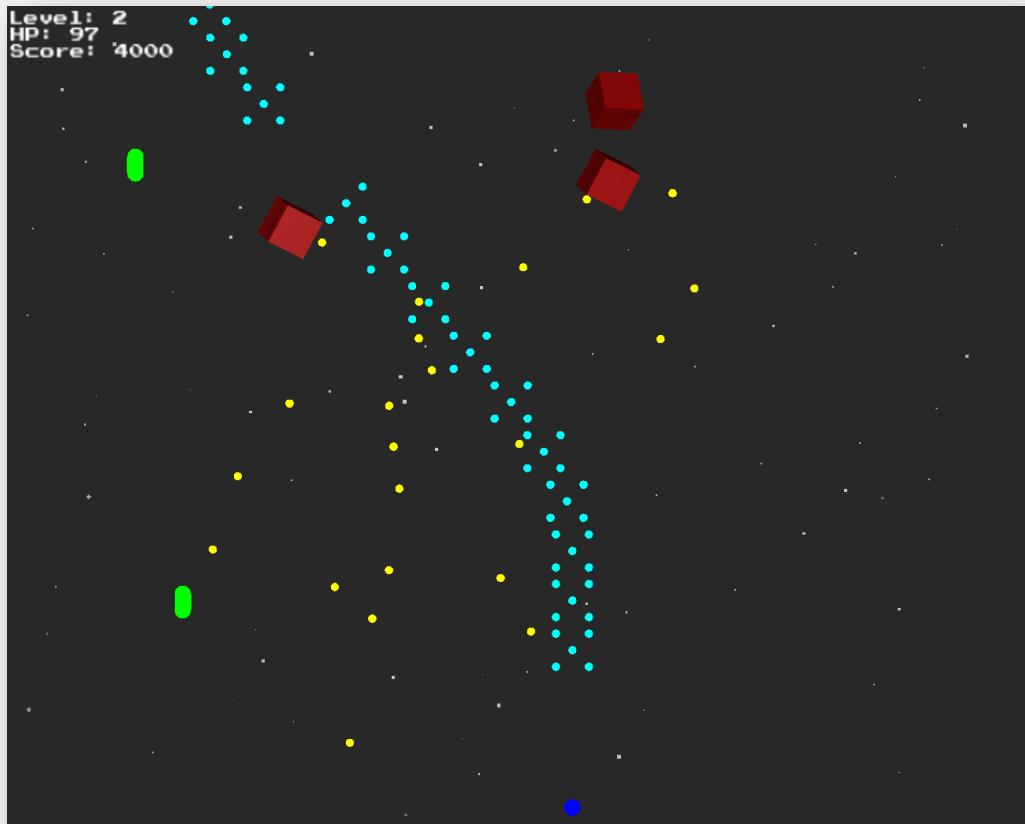
There are more than 3 attacks for the enemy already made. Use your understanding of Random.Range to figure out how to add the other 3 attacks for the attack function.

## Activity 8

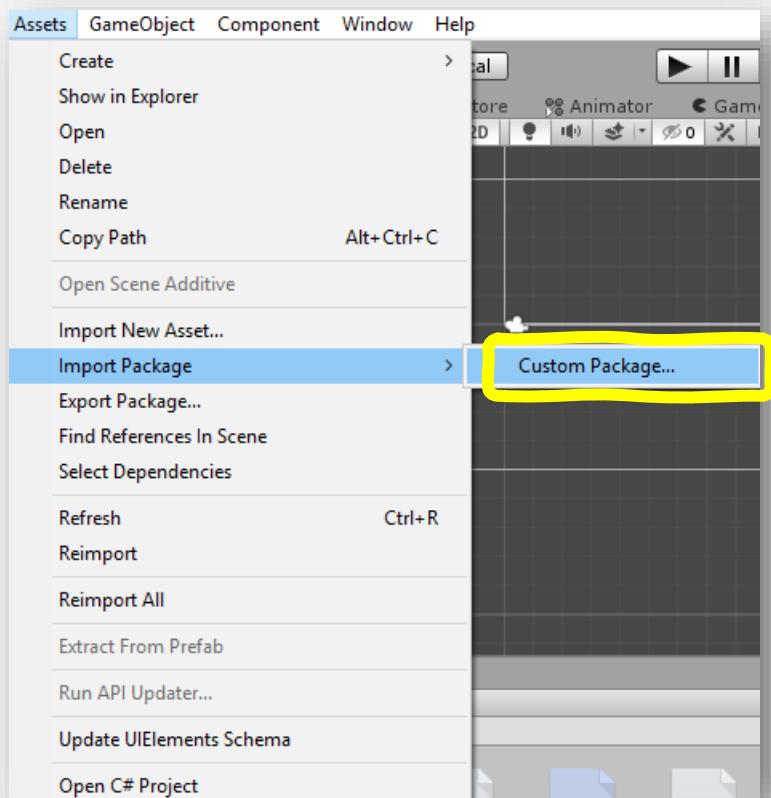
# Shape Jam

Your mission: A nearly complete game is given to you, but the most essential functions are missing - those that create new objects in the scene. You will add the code that creates these new objects and elements within the game.

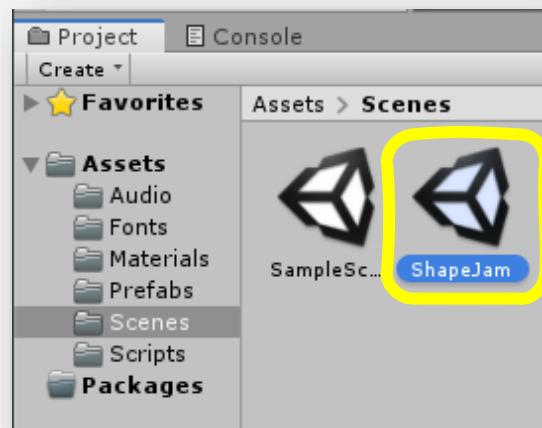
In the end, the player will be creating projectiles. Enemies and powerups will be created on timers, and these enemies will constantly fire hazardous objects at the player. You will learn a few ways to use code to create these new objects using the `Instantiate()` function.



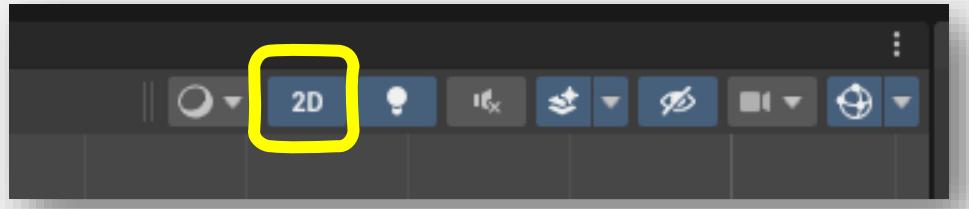
- 
- 1** Start a new Unity Project and name it **YOUR INITIALS - Shape Jam**.  
Select **3D core**.
- 2** We've created a starter pack to give you a head start! To use it, import the **ShapeJam\_Ninja.unitypackage** by going to **Assets > Import Package > Custom Package> All > Import**.



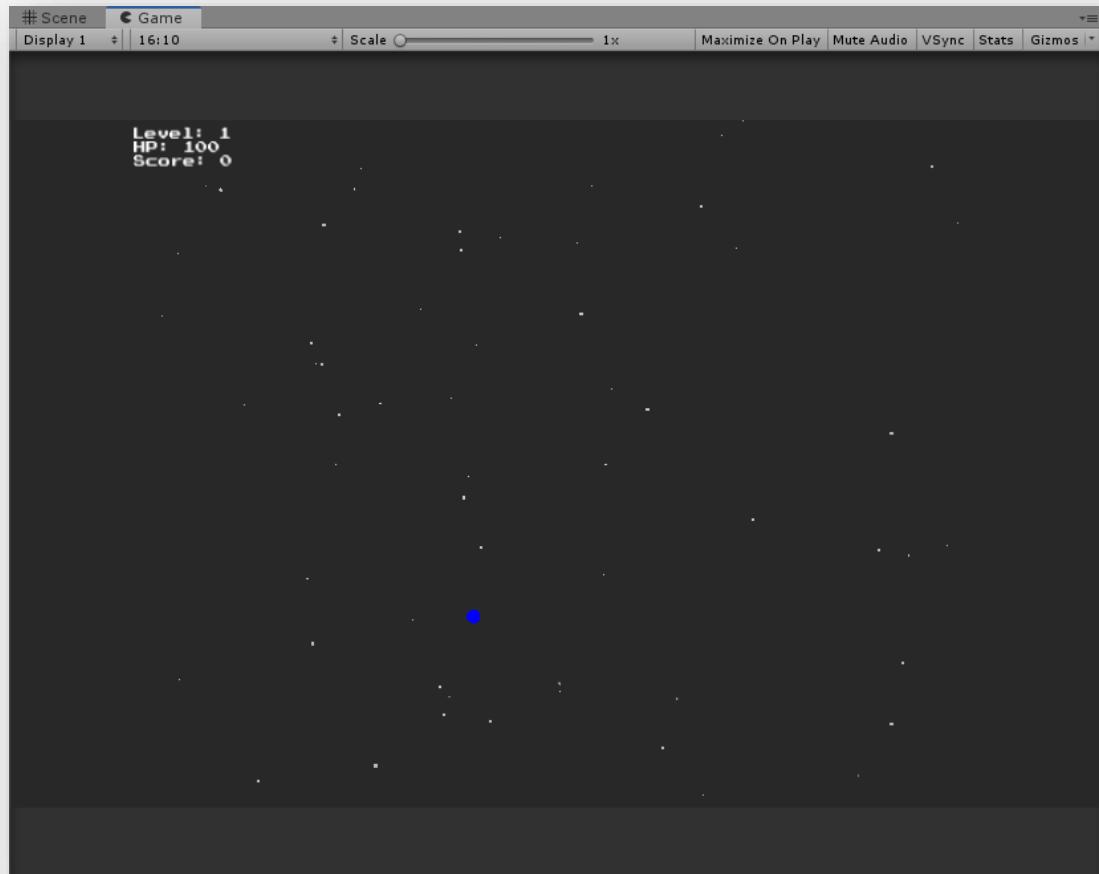
- 3** To open the starter package, double click on the **"ShapeJam"** Scene. You can find this in the **Project** tab under **Assets > Scenes**.



**4** Click the **2D** button at the top of the scene to align the camera properly.

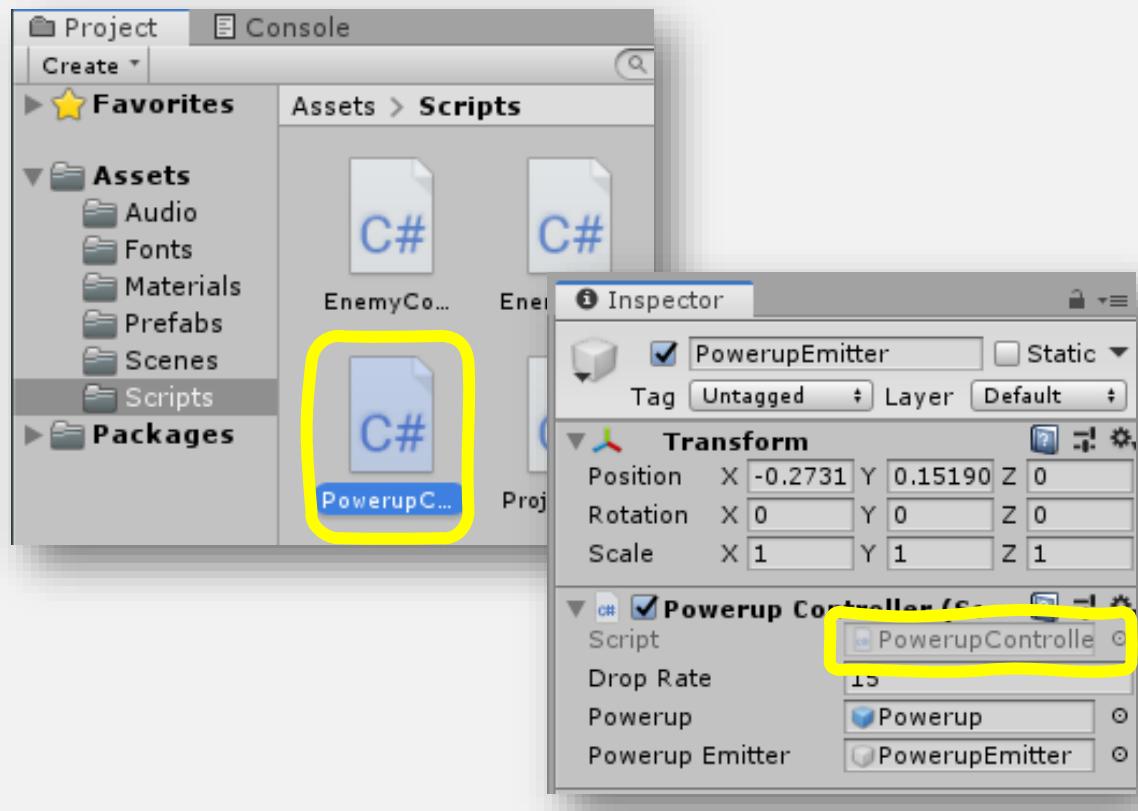


**5** Before getting to work, **play** your game and see what happens!



What is already coded for you? You can control the blue dot, but you cannot fire your blaster. Right now, there are no enemies!

- 6** The first thing we want to do is add a powerup for the player to collect. We need to edit the **PowerupController** script. We can find this in the scripts folder or attached to the **PowerupEmitter** object in the **Hierarchy**.



Open the **PowerupController** script in Visual Studio by double clicking on it.

- 7** At the top of the class, there are different private and public variables that help the script run.

Some of those variables are initialized in the **Start** function. We need to write the code that will **instantiate**, or **create**, the power up objects for the scene.

Since we need to react to the game as it runs, we will write this code in the **Update** function.

- 8** Find the `Update` function and look at the code that is already present. It might seem like a lot, but there are a lot of lines of comments!

```
void Update()
{
 // Null check for gameOver state
 if (playerController.gameOver)
 {
 return;
 }

 // At the interval decided by dropRate,
 // powerupPosition is set to a random x from -6 to 6
 // above the game the powerupEmitter itself is moved to that position,
 // and the nextDrop is set
 // a powerup is instantiated at the emitter's position and rotation

 if (Time.time > nextDrop)
 {
 float randomX = Random.Range(-6.0f, 6.0f);
 Vector3 powerupPosition = new Vector3(randomX, 6.0f, 0.0f);
 nextDrop = Time.time + dropRate;

 /***** Add your code below *****/
 }
}
```

Look at the first `if` statement. What do you think that statement is checking for? What happens if the statement evaluates to true? What happens if the statement evaluates to false? What does `return;` do?

This `if` statement checks to see if the game is over.

If the game is over, then `playerController.gameOver` will have the value of true and the code inside the `if` statement will be executed.

The code inside the statement is just `return;` which will tell Unity to stop running the function and to ignore the rest of the code in the function.

If the game is not over, then `playerController.gameOver` will have the value of false and the code inside the statement, `return;`, will not be run. Then Unity will continue executing the rest of the `Update` function.

- 9** Look at the second `if` statement. `Time.time` gets the number of seconds since the game started. The statement here is checking to see if it is time to drop a powerup to the player by seeing if the current time of `Time.time` is past the `nextDrop` time.

```
if (Time.time > nextDrop)
{
 float randomX = Random.Range(-6.0f, 6.0f);
 Vector3 powerupPosition = new Vector3(randomX, 6.0f, 0.0f);
 nextDrop = Time.time + dropRate;

 /***** Add your code below ****/
 | **** Add your code below ****|
 ***** Add your code below *****/

 /***** Add your code above ****/
 | **** Add your code above ****|
 ***** Add your code above *****/
}
```

- 10** If the condition is met, we first get a random value between -6 and 6 to make sure the powerup spawns inside the view of the camera.

```
float randomX = Random.Range(-6.0f, 6.0f);
```

- 11** We then use that `randomX` value to create a `new Vector3` to describe the powerup's position.

```
Vector3 powerupPosition = new Vector3(randomX, 6.0f, 0.0f);
```

- 12** By changing the value of `nextDrop` we can control when the next powerup spawns, so we set it equal to the current time and add the `dropRate`.

```
nextDrop = Time.time + dropRate;
```

**13** You might have noticed that even though we are setting a position and deciding when the next powerup should spawn, no powerups are spawning! This is because we never **create** a powerup!

We create objects in Unity by using the **Instantiate** function. The **Instantiate** function takes three parameters.

- The first parameter is the **game object** we want to create inside the scene.
- The second parameter is the **position** of where we want the object to be created.
- The third and final parameter is the **rotation** of what direction we want our object to be facing when it is created.

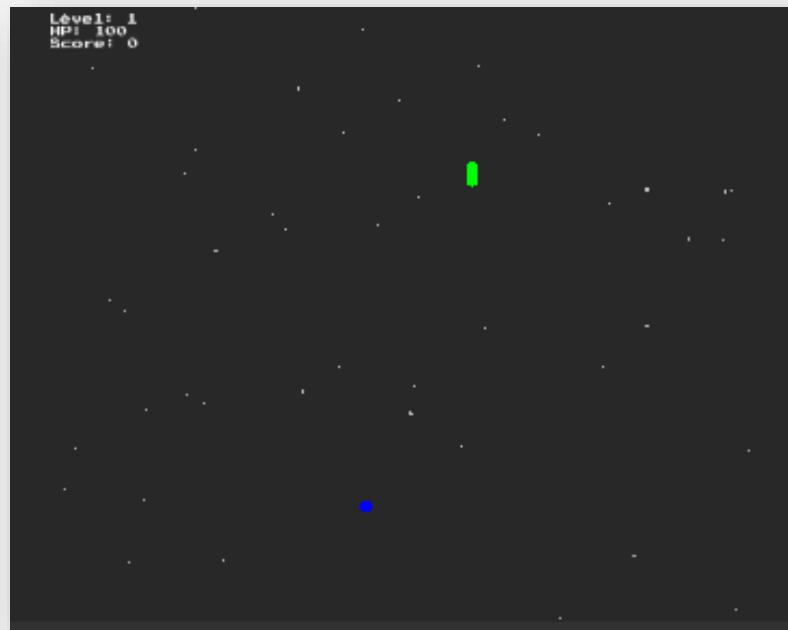
**14** On a line in-between the two big comments telling you where to write your code, **Instantiate** a new powerup object at the **PowerupEmitter's** location and direction by typing **Instantiate(powerup, powerupPosition, transform.rotation);**.

```
/*******************\
| **** Add your code below ****|
********************/
Instantiate(powerup, powerupPosition, transform.rotation);

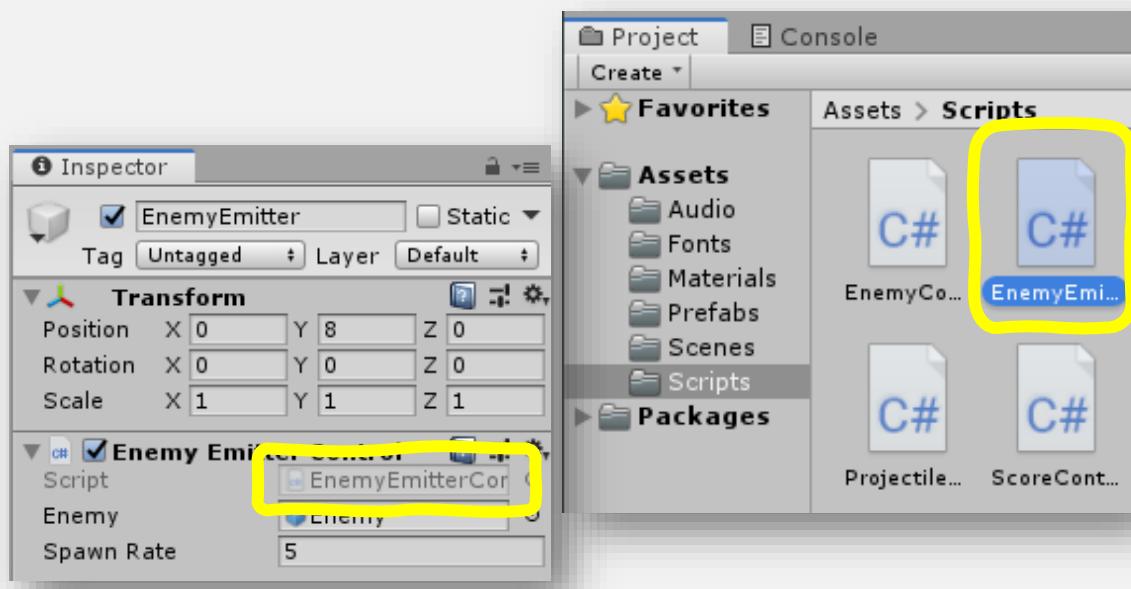
/*******************\
| **** Add your code above ****|
********************/
```

Remember that Unity understands that the **transform** will refer to the **PowerupEmitter** because the script is attached to it.

**15** Play your game and wait to see what happens. After 15 seconds, a green powerup will fall from above the screen! What happens when you try to catch it?



**16** Now that we have powerups working with the `Instantiate` function, we can try something a little more complex when we create enemies. We need to edit the **EnemyEmitterController** script. We can find this in the scripts folder or attached to the **EnemyEmitter** object in the **Hierarchy**.



**17** Open the **EnemyEmitterController** script in Visual Studio by double clicking on it.

**18** At the top of the **class**, there are different **private** and **public variables** that help the **script** run. Some of those **variables** are initialized in the **Start** function.

We need to write the code that will “instantiate” (create) the enemy objects in the scene.

Since we need to react to the game as it runs, we will write this code in the **Update** function.

**19** Find the **Update** function and look at the code that is already present. There’s less code than the last **Update** function that we looked at!

```
void Update()
{
 // Null check
 if (playerController.gameOver)
 {
 return;
 }
 // Usual time delay code
 if (Time.time > nextSpawn)
 {
 nextSpawn = Time.time + spawnRate;
 // Spawns a number of Enemy objects equivalent to the currentLevel
 // These spawn at a random x-value above the stage

 /******
 | **** Add your code below ****|
 *****/

 /******
 | **** Add your code above ****|
 *****/
 }
}
```

**20** Look at the first `if` statement. Have we seen this code before? Yes! We saw it in the **PowerupController's** `Update` function. Recall that it checks to see if the game is over.

If it is over, it ignores the code below the `return;` statement.

If the game is not over, it will run all the code in the function.

**21** Look at the next `if` statement. This is very similar to the second `if` statement in the **PowerupController's** `Update` function.

We want to check to see if it is time to spawn an enemy.

If it is, then we want to execute our spawn code that lives inside of the `if` statement.

```
if (Time.time > nextSpawn)
{
 nextSpawn = Time.time + spawnRate;
 // Spawns a number of Enemy objects equivalent to the currentLevel
 // These spawn at a random x-value above the stage

 /******
 |**** Add your code below ****|

```

**22** There is only one line of provided code in this `if` statement.

This line of code sets the value of `nextSpawn` to be equal to the current time plus the `spawnRate` of enemies.

This controls when the next enemy will spawn!

**23** To make sure the player has a challenge, we want to spawn enemies based on the current level the player is on.

For example, the game will start spawning one enemy at a time.

Once the player picks up a green powerup and gets to level 2, we want the game to spawn two enemies at the same time.

We can do this by using a `for` loop that starts at `0` and loops up to the current level.

Type `for (int i = 0; i < playerController.currentLevel; i++)`  
`{ }` making sure to leave an empty space between the brackets.

```
/*
| **** Add your code below ****|

```

```
for (int i = 0; i < playerController.currentLevel; i++)
{
}

/*
| **** Add your code above ****|

```

**24** Inside the loop, we need to select a random x position for the enemy and then use that to create a position vector.

Have you seen code like that already? Yes, in the **PowerupController!** We can use that exact same code with a few name changes.

**25** Inside the loop, create a `float` variable named `randomX` and set it equal to a **random** number between `-6` and `6` by typing `float randomX = Random.Range(-6.0f, 6.0f);`

We are putting an “f” at the end of the two numbers to tell Unity that we want our random values to not be just whole numbers or integers like `-2` or `4` and instead numbers that are **floats** like `-2.41571623` and `4.000245`. This will give a lot more variation to where the enemies spawn.

```
for (int i = 0; i < playerController.currentLevel; i++)
{
 float randomX = Random.Range(-6.0f, 6.0f);
}
```

**26** Next, we want to create a new vector called `enemyPosition`.

We want the x position to be the random number we just created, the y position to be `6` so it starts above the screen, and the z position to be `0` because we want our game to behave like a 2D game.

Type `Vector3 enemyPosition = new Vector3(randomX, 6, 0);` after the `float randomX` line.

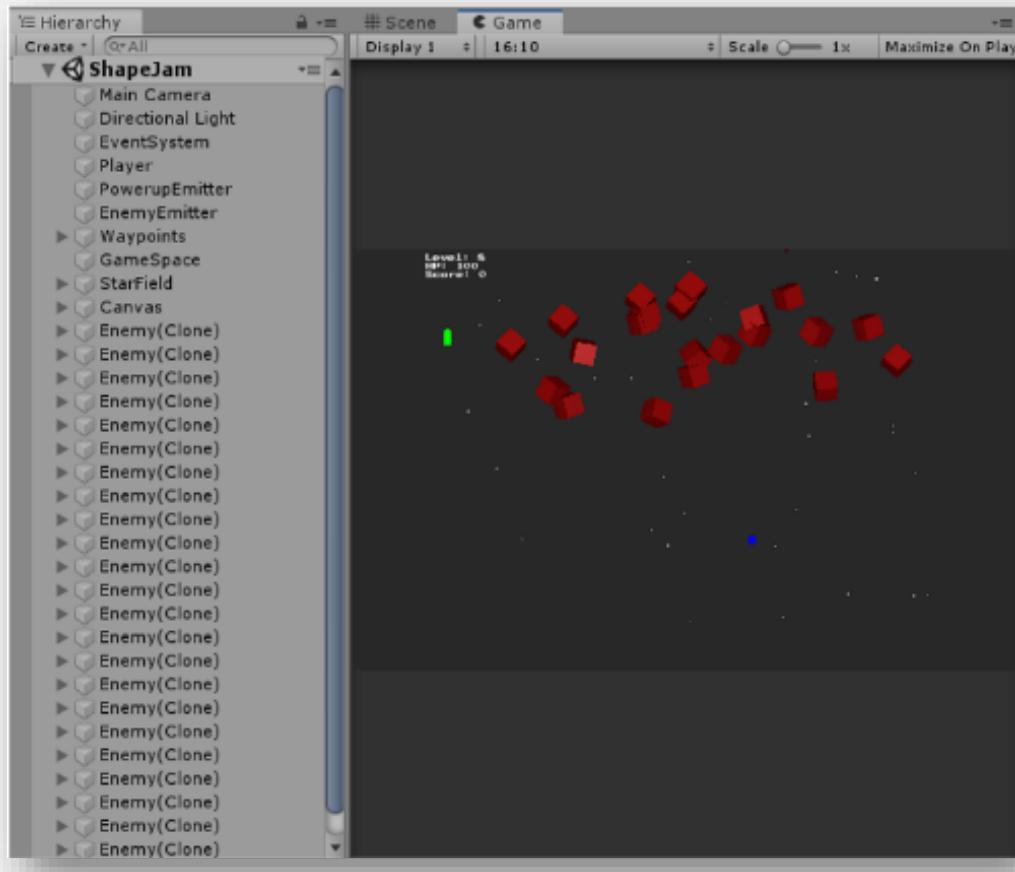
```
for (int i = 0; i < playerController.currentLevel; i++)
{
 float randomX = Random.Range(-6.0f, 6.0f);
 Vector3 enemyPosition = new Vector3(randomX, 6, 0);
}
```

**27** Now that we have the enemy emitter in the new position, we can create our enemy. This is just like how we created the powerup.

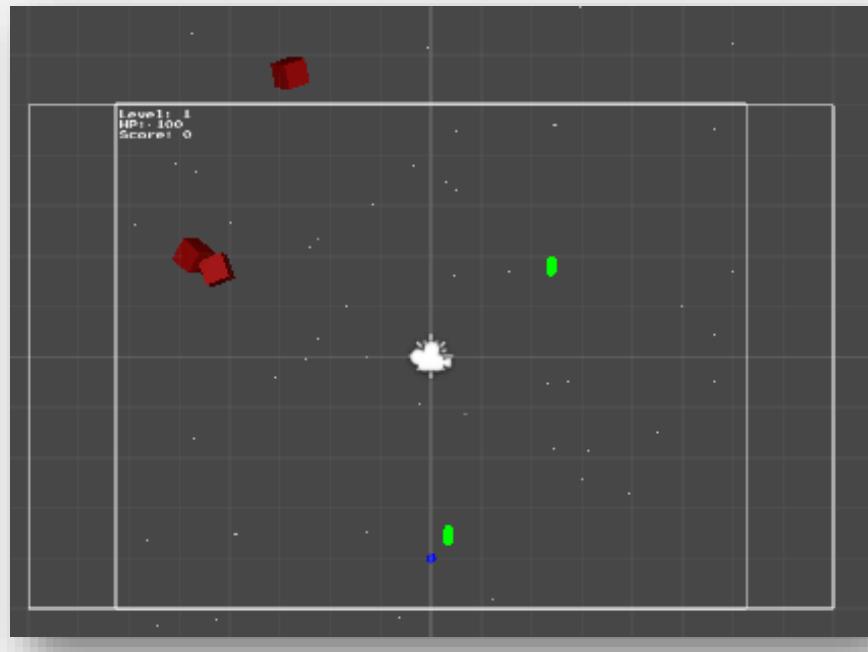
We want to **Instantiate** the **Enemy** game object and set its **position** and **rotation** based on the enemy emitter.

Type `Instantiate(Enemy, enemyPosition, transform.rotation);` after the `enemyPosition` line.

**28** Save your code and play your game. Pay attention to the **current level** and **how many enemies** are spawned. Every time you get a green powerup, the enemy emitter will start to create more enemies based on the player's level! Without a way to fight back, things can get crazy really fast! What happens if the player collides with an enemy?



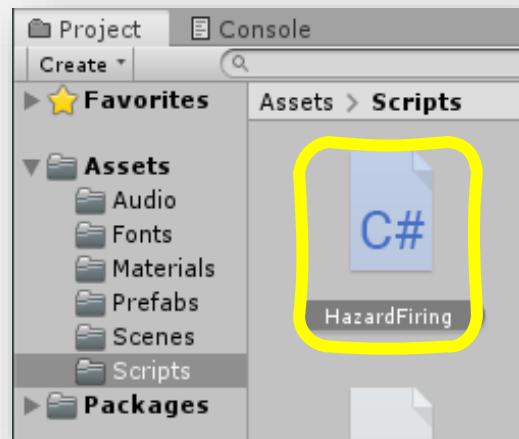
**29** While the game is running, click on the **# Scene** tab. You'll be able to see outside of the player's view and watch the enemies and powerups spawn!



**30** During your playtest, you should have noticed that nothing happens when the player touches an enemy.

That means that there is no challenge because the player can never lose! We can fix this by editing the **HazardFiring** script.

We can find this in the **scripts** folder. Double click it to open it in Visual Studio.



**31** At the top of the class, there are different private and public variables that help the script run.

Some of those variables are initialized in the Start function. We need to write the code that will instantiate or create enemy hazards in the scene.

Since we need to react to the game as it runs, we will write this code in the Update function.

**32** Find the Update function and look at the code that is already present. It might seem like a lot, but there are a lot of lines of comments!

```
void Update()
{
 // Null check for GameOver
 if (playerController.gameOver)
 {
 return;
 }

 // Target is set to the Player's position,
 // and a vector is made from the current position to the target
 // At the rate decided by nextFire, and the rotation dictated by the hazardSpawn,
 // a clone is instantiated
 // which will start with normalized velocity in the direction of the target (player)
 target = player.transform.position;
 HazardMoveDirection = target - transform.position;

 if (Time.time > nextFire)
 {
 nextFire = Time.time + fireRate;

 /***** Add your code below ****/
 | **** Add your code below ****|
 **** Add your code below ****/

 /***** Add your code above ****/
 | **** Add your code above ****|
 **** Add your code above ****/
 }
}
```

**33** Look at the first if statement. This is the third time we have seen this exact code! Before moving on, explain what it does to your Code Sensei.

```
if (playerController.gameOver)
{
 return;
}
```

- 34** There are two lines of code between the two if statements. The first statement sets up the target position for the hazard by finding out the current position of the player.

```
target = player.transform.position;
```

- 35** The second statement calculates the direction that the hazard needs to move in by taking the difference of the target and the enemy's position.

```
HazardMoveDirection = target - transform.position;
```

- 36** Look at the second if statement. This is very similar to the if statements in the `PowerupController` and `EnemyEmitterController` Update functions that we looked at before. Before moving on, explain what it does to your Code Sensei.

```
if (Time.time > nextFire)
{
 nextFire = Time.time + fireRate;

 /***** Add your code below ****/
 | **** Add your code below ****|
 ***** Add your code below *****/

 /***** Add your code above ****/
 | **** Add your code above ****|
 ***** Add your code above *****/
}
```

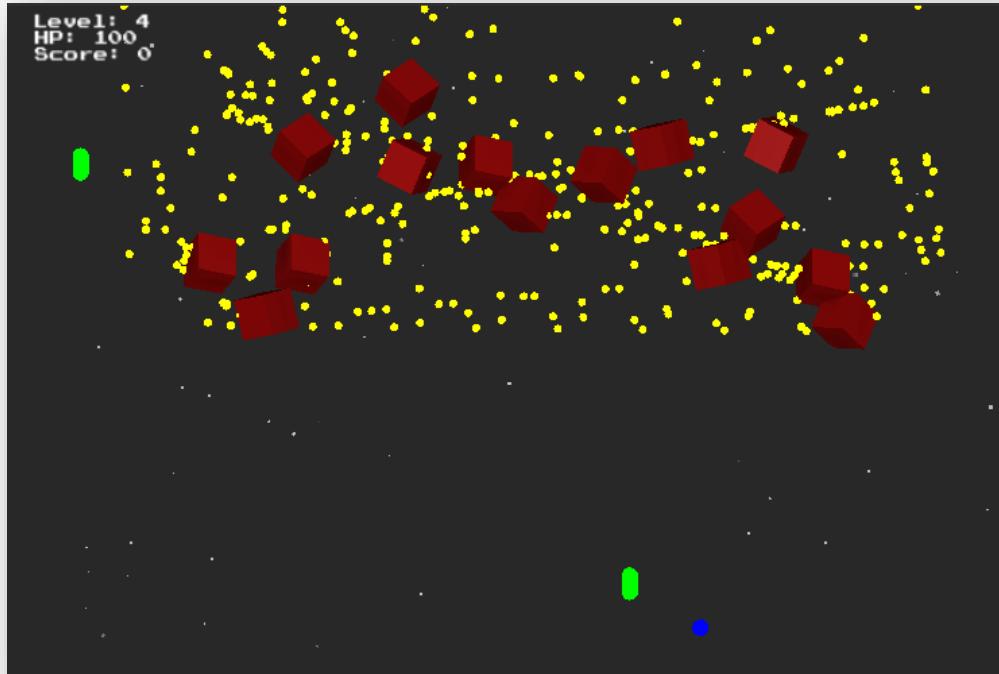
- 37** The first thing we want to do is create a clone of the hazard object. We do this by using the Instantiate function and providing it the original game object, the position where we want our new cloned object, and the direction we want our object to be facing. Type `GameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);` to create a clone of the hazard.

```
if (Time.time > nextFire)
{
 nextFire = Time.time + fireRate;

 /***** Add your code below ****\
 | **** Add your code below ****|
 ***** Add your code above *****/
 GameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);

 /***** Add your code above ****\
 | **** Add your code above ****|
 ***** Add your code above *****/
}
```

- 38** Play your game! Why are the hazards frozen in the air? We instantiated them, but did we ever tell them what to do after they were created?



- 
- 39** We need to tell the hazards to move! We can do that by getting our new hazardClone's rigid body and giving it a velocity.

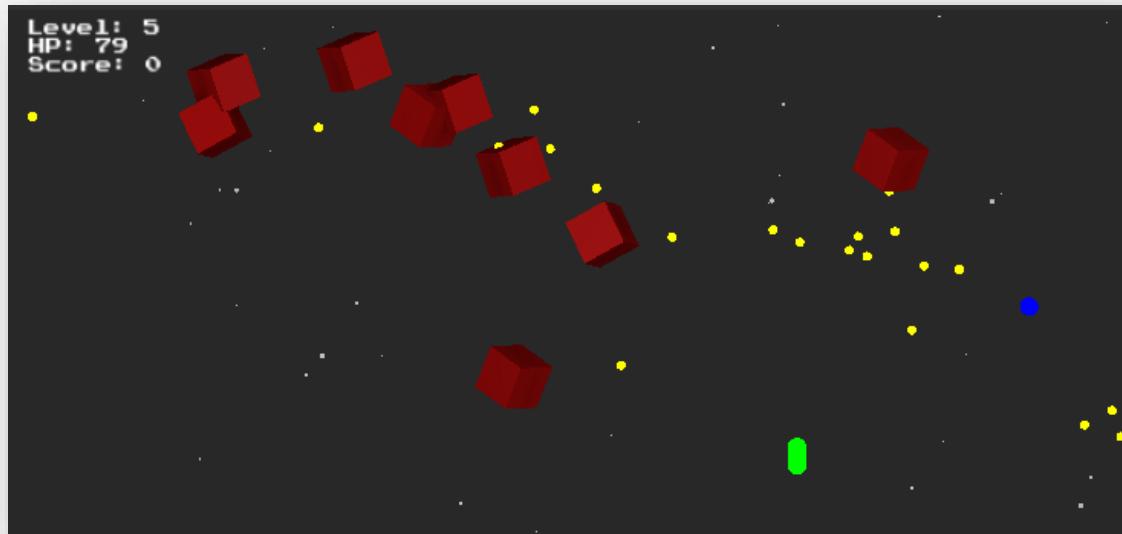
First, we need get the Rigidbody component that is attached to the hazardClone by using Unity's `GetComponent<Rigidbody>()` function on our hazardClone and storing it in a variable named hazardRigidbody.

```
*****\n| **** Add your code below ****|\n*****\n\nGameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);\n\nRigidbody hazardRigidbody = hazardClone.GetComponent<Rigidbody>();\n\n*****\n| **** Add your code above ****|\n*****\n
```

- 
- 40** Next, we need to apply a velocity to that rigid body. On the next line, type `hazardRigidbody.velocity = HazardMoveDirection;` to set the hazard's velocity to the vector that represents the direction between the enemy and the player that we talked about in the previous step.

```
*****\n| **** Add your code below ****|\n*****\n\nGameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);\n\nRigidbody hazardRigidbody = hazardClone.GetComponent<Rigidbody>();\nhazardRigidbody.velocity = HazardMoveDirection;\n\n*****\n| **** Add your code above ****|\n*****\n
```

**41** Play your game! The hazards are all moving now! What can you tell about the behavior of the hazards? Where do they start and what's their destination? Do they all travel at the same speed? What is the difference between a hazard that starts far away from the player and a hazard that starts really close to the player?

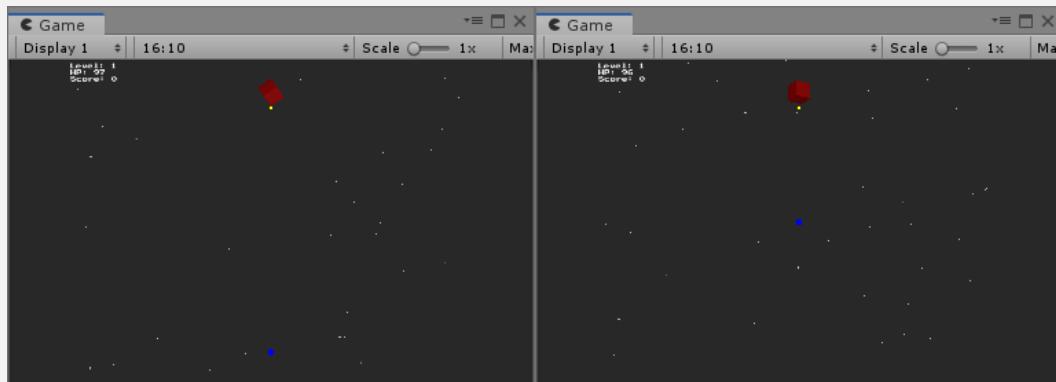


## 42 Why are the hazards traveling at different speeds based on their distance from the player?

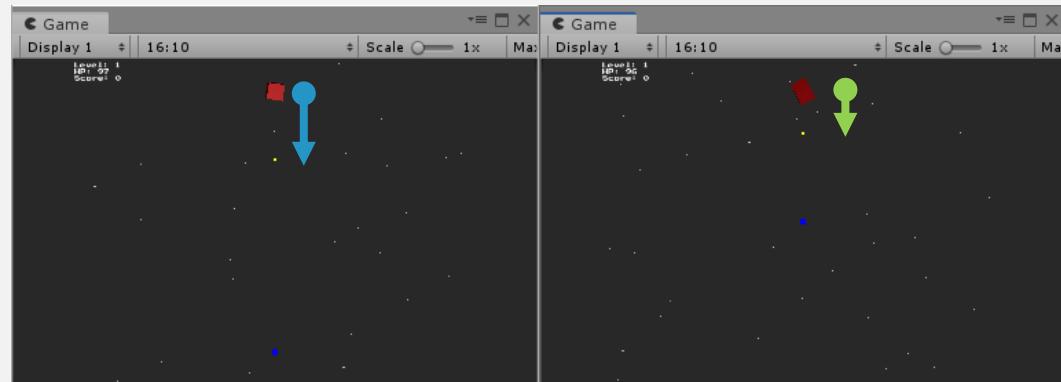
We are setting the velocity equal to a vector that describes the distance and direction between the enemy and the player.

The greater the distance the greater the velocity.

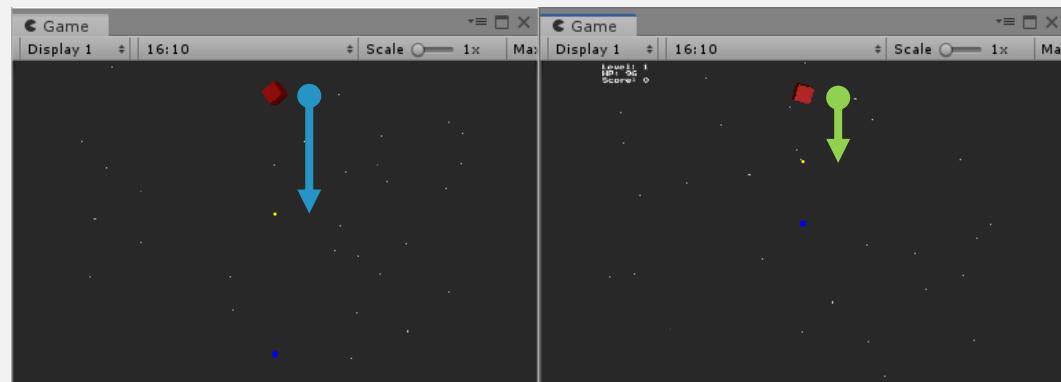
0 Seconds, frame 0



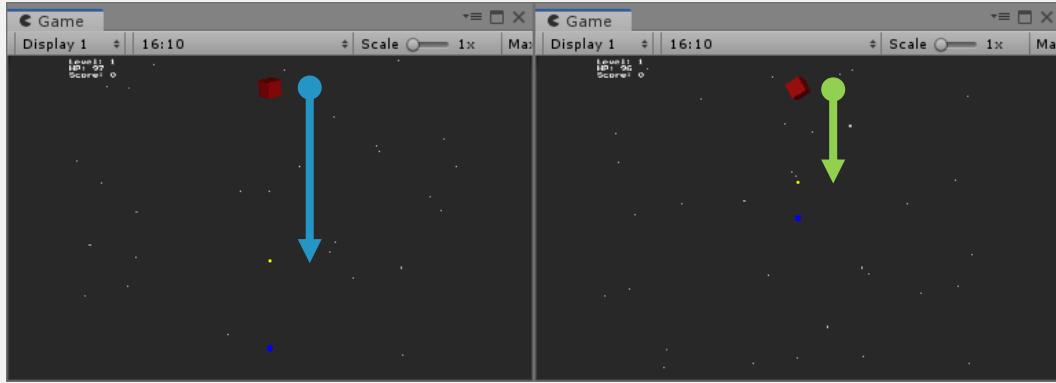
0.3 Seconds, frame 10



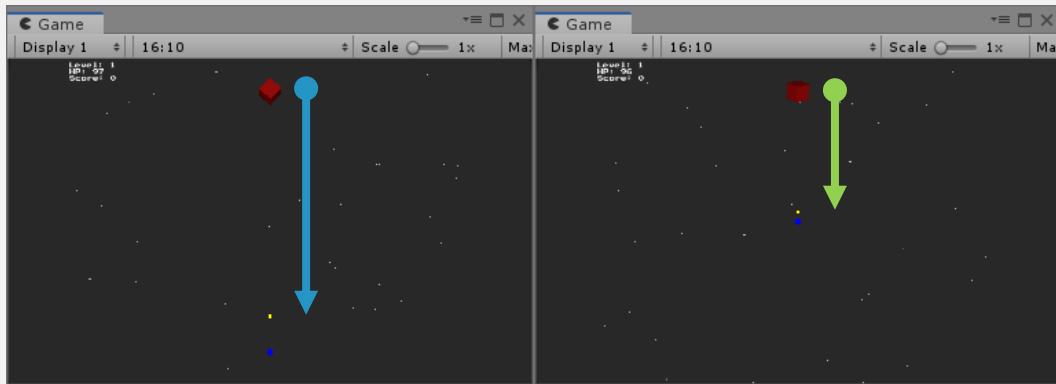
.6 seconds, frame 20



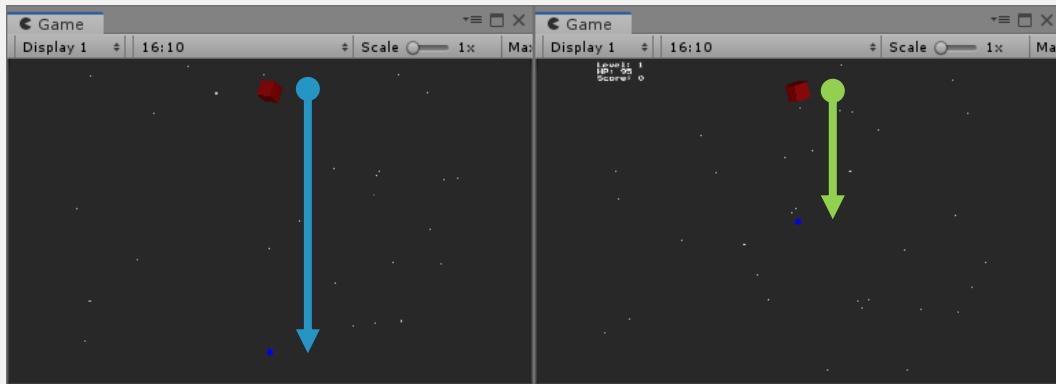
1 second, frame 30



1.3 seconds, frame 40



1.6 seconds, frame 50



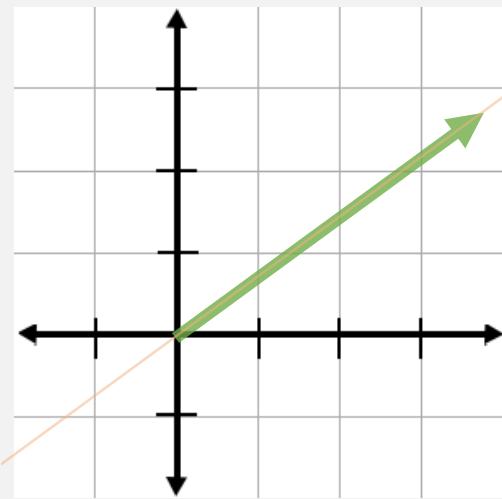
- 43** We want to make sure that our hazards travel towards the player, but we do not want the speed to change based on how close or far the player is.

We can accomplish this by normalizing our HazardMoveDirection variable to make sure we only use the direction in our calculations and not the distance.

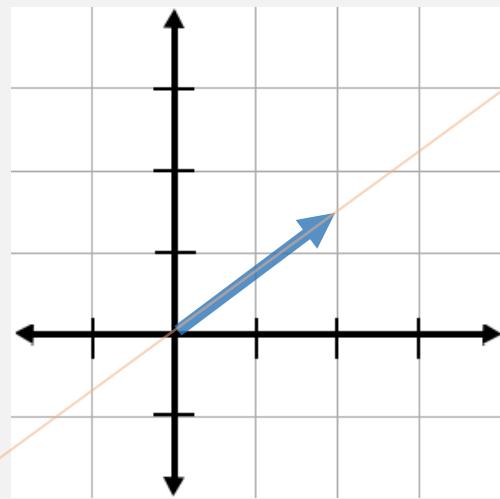
**44** Every vector has **magnitude** (or size) and a **direction**. When we normalize a vector, we are keeping the direction, but we are shrinking it or growing it to make its magnitude exactly equal to 1.

For this first example, we have a green vector that is really big! When we normalize it, we shrink the size down to exactly 1 but keep it pointing in the same direction along the orange line.

Original

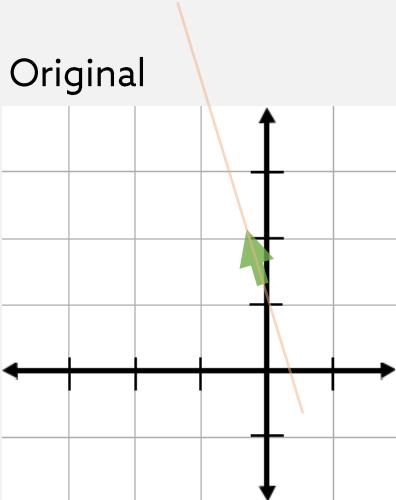


Normalized

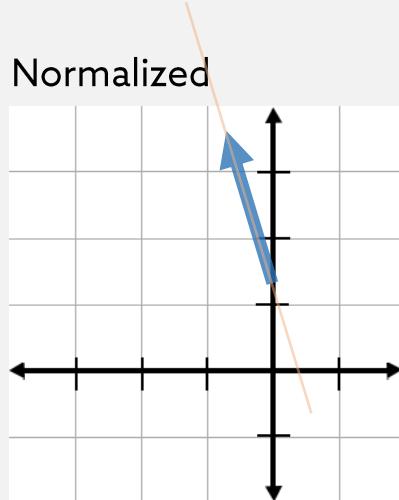


For this next example, we have a green vector that is really small! When we normalize it, we increase the size up to exactly 1 but keep it pointing in the same direction along the orange line.

Original



Normalized



The blue arrow in the first example and the blue arrow in the second example are facing different directions, but they are both the exact same length! They each have a magnitude of 1.

Before moving on, explain what happens when we normalize a vector to your **Code Sensei** and then to one other **Ninja** in your Dojo!

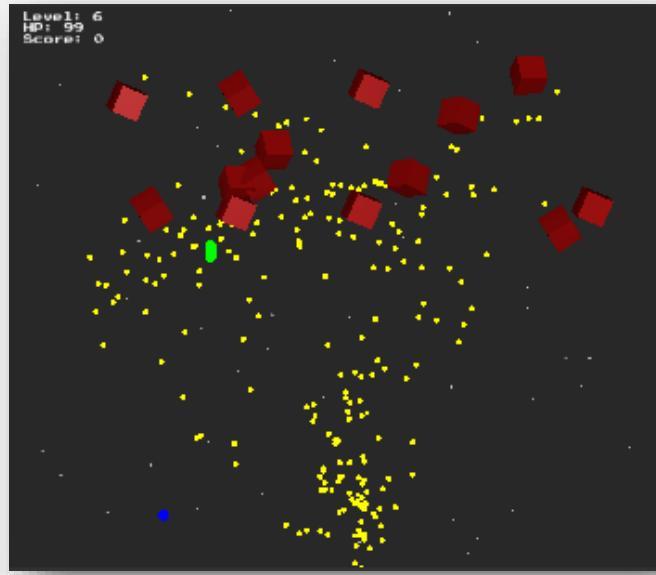
- 45** The math it takes to normalize a vector might seem complicated, but Unity already knows how to do it! Change the last line we typed from `hazardrigidBody.velocity = HazardMoveDirection;` to `hazardrigidBody.velocity = HazardMoveDirection.normalized;` to have Unity normalize the `HazardMoveDirection` before setting the hazard's velocity.

```
/*
| **** Add your code below ****|
**** Add your code above ****/
GameObject hazardClone = Instantiate(hazard, transform.position, transform.rotation);

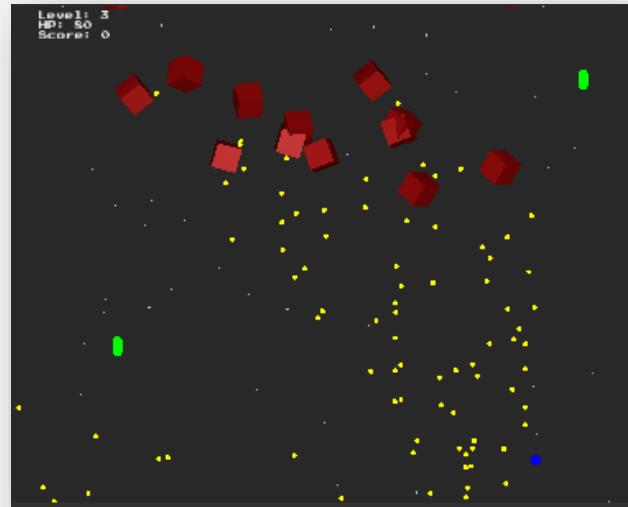
Rigidbody hazardrigidBody = hazardClone.GetComponent< Rigidbody>();
hazardrigidBody.velocity = HazardMoveDirection.normalized;

/*
| **** Add your code above ****|
**** Add your code below ****/
```

**46** Save your script and play your game! What has changed since before you normalized the direction vector? All the hazards are now moving at the same exact speed no matter where the player is, but now they are moving very slowly!



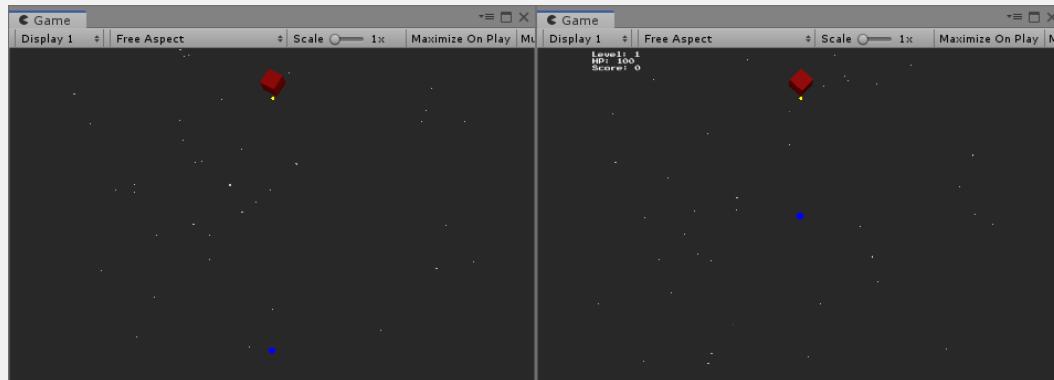
**47** The **HazardFiring** script has a **public** variable named **speed**. Change the last line of code we typed to include the **speed** variable by changing it from **hazardrigidBody.velocity = HazardMoveDirection.normalized;** to **hazardrigidBody.velocity = HazardMoveDirection.normalized \* speed;** to multiply the normalized direction vector by the speed variable. Play your game and see that the hazards are traveling faster!



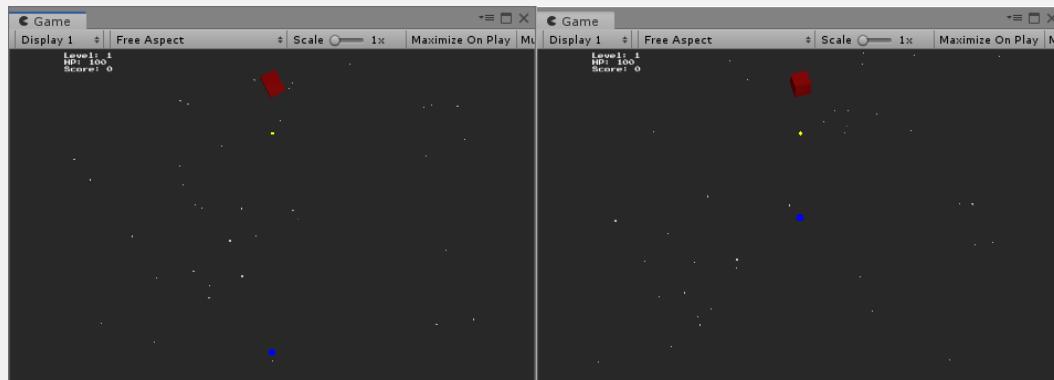
48

Let's double check to see if the player's **position** has an impact on the **speed** of the hazard now that we have **normalized** the **HazardMoveDirection** vector.

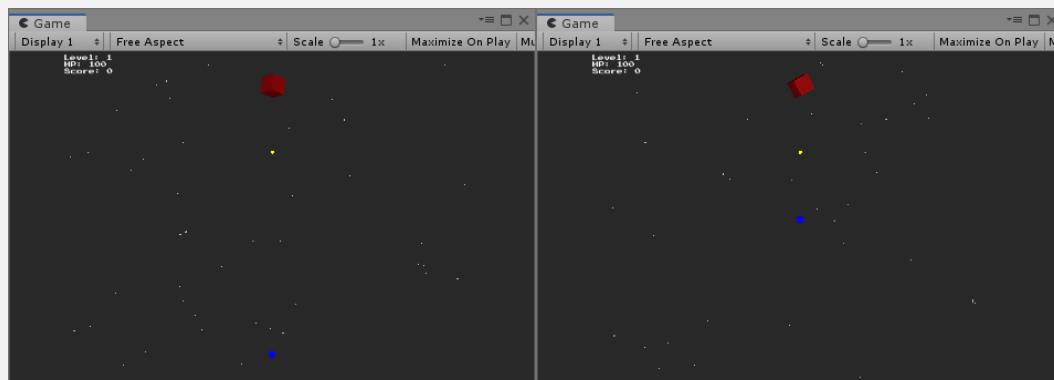
0 seconds, frame 0



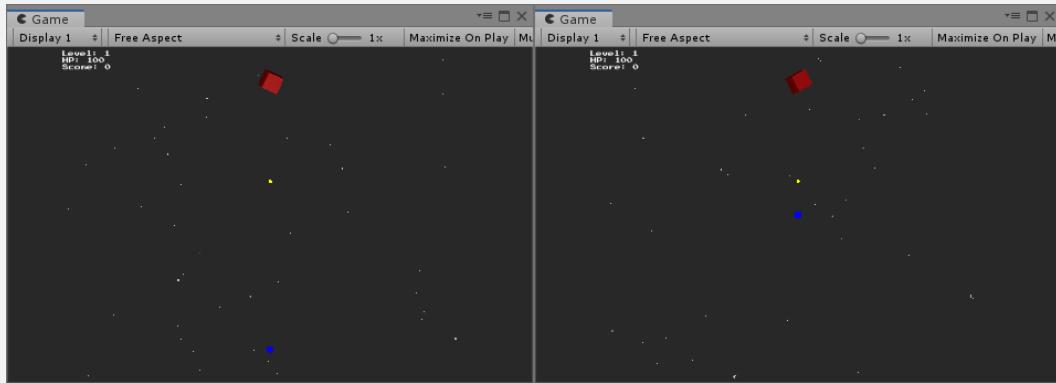
.6 seconds, frame 20



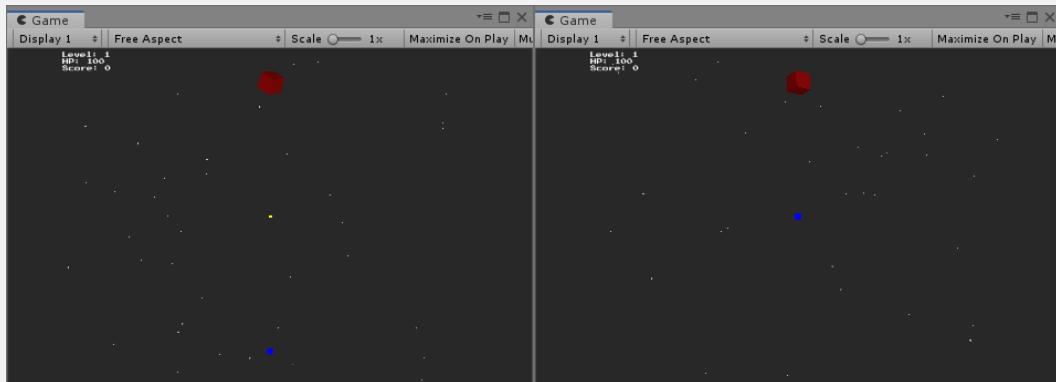
1.3 seconds, frame 40



2 seconds, frame 60



2.6 seconds, frame 80



Compare the hazard's location in both images (left and right). Although the player's position is closer in the images on the right, what do you notice about the hazard's location on the screen?

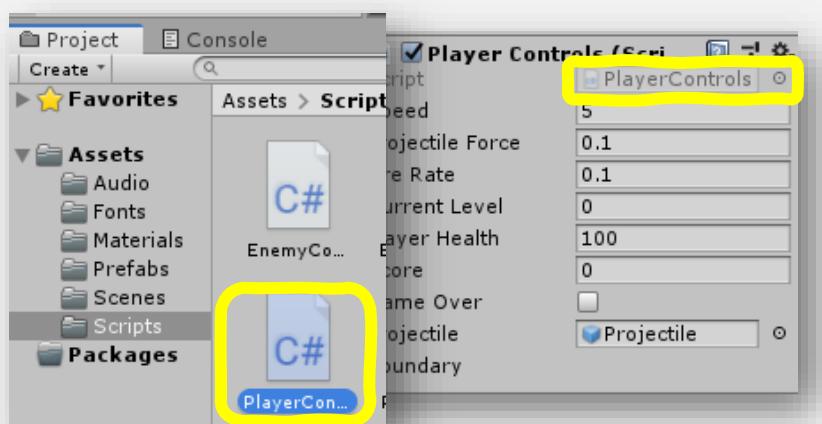
The hazard is traveling at the same speed no matter where the player is because we normalized the HazardMoveDirection vector!

## 49 Play your game and think about what is still missing.

You can play it to see how long you last, but we can still add in the ability for the player to fire hazards back at the enemies!

Next, let's add player bullets!

**50** We can give the player the ability to fire hazards by editing the **PlayerControls** script. We can find this in the scripts folder or attached to the **Player** object in the **Inspector**. Double click it to open it in Visual Studio.



**51** At the top of the **PlayerControls** class, there are different **private** and **public** variables that help the script run. Some of those variables are initialized in the Start function. We need to write the code that will instantiate the player hazards in the scene. Since we need to react to the game as it runs, we will write this code in the **Update** function.

```
void Update()
{
 if (gameOver)
 {
 return;
 }
 // While the game is running, if the player presses space or z, and you can fire
 // Potentially 5 clones of the projectile object will be made
 // If the currentLevel is 1 (or greater) a projectile is created directly at the player
 // If the currentLevel is 3 (or greater) additional projectiles are added to the left and right
 // If the currentLevel is 5 (or greater) additional projectiles are added again to the left and right
 if ((Input.GetKey("space") || Input.GetKey("z")) && (Time.time > nextFire))
 {
 nextFire = Time.time + fireRate;

 /***** Add your code below ****/

 }

 /***** Add your code above ****/

}

// If the playerHealth is reduced to 0, the gameOver bool is set to true, having various effects across multiple scripts
if (playerHealth <= 0) {
 gameOver = true;
}
```

**52** Find the Update function and look at the code that is already present. It might seem like a lot, but there are a lot of lines of comments!

**53** Look at the first if statement. This isn't the exact same code that we looked at in the other three scripts.

In the other scripts it says if (playerController.gameOver) while in this script it just says if (gameOver).

Why do you think we do not need to use playerController in this script?

It's because this script is the playerController variable we used previously!

```
if (playerController.gameOver)
{
 return;
}
```

```
if (gameOver)
{
 return;
}
```

```
public int playerHealth;
public int score;
public bool gameOver = false;
```

The gameOver variable belongs to the script we are working on!

**54** Look at the last if statement. This checks to see if the player's health is less than or equal to zero and sets the gameOver variable to true. Setting gameOver to true will stop the enemy and powerup emitters, the enemies will not fire, and the player will not be able to move.

```
if (playerHealth <= 0) {
 gameOver = true;
}
```

**55** Finally, look at the middle if statement. The conditional is checking two things. First, it checks to see if the user has pressed the space or z keys. If one of those keys is pressed, then it checks to see if the player is allowed to fire by comparing the current time to the next time the player is allowed to fire. The one line of provided code inside the statement sets up when the player can fire next.

```
if ((Input.GetKey("space") || Input.GetKey("z")) && (Time.time > nextFire))
{
 nextFire = Time.time + fireRate;

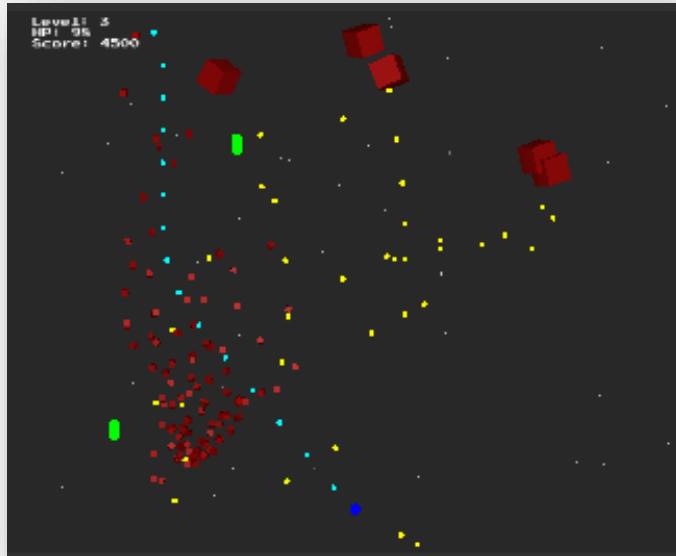
 /***** Add your code below *****
 |
 ***** Add your code above *****
 /
}
```

**56** If the player is pressing the correct key and can fire, we want to create a new projectile object at the player's location with the player's direction. Type Instantiate(projectile, transform.position, transform.rotation);.

```
if ((Input.GetKey("space") || Input.GetKey("z")) && (Time.time > nextFire))
{
 nextFire = Time.time + fireRate;

 /***** Add your code below *****
 |
 ***** Add your code above *****
 /
 Instantiate(projectile, transform.position, transform.rotation);
}
```

- 57** Play your game and try to fire! Everything works! We did not set a velocity for the new projectile because the projectile understands that it needs to go up no matter what. You can open the ProjectileScript file to see exactly how that is coded!



- 58** As the player levels up, more and more enemies appear, but the player doesn't get more powerful. We should reward the player for leveling up. After the Instantiate line you just wrote, create an if statement that checks to see if the currentLevel is greater than or equal to 3 by typing if (currentLevel >= 3) { } making sure to leave a blank line between the curly brackets.

```
/*
| **** Add your code below ****|

```

```
Instantiate(projectile, transform.position, transform.rotation);
if (currentLevel >= 3)
{
 |
}
```

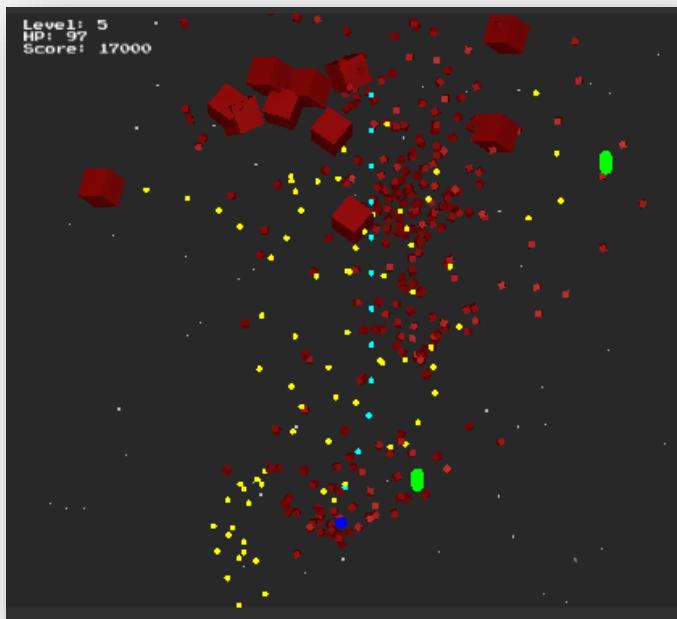
```
/*
| **** Add your code above ****|

```

- 59** We can reward the player by adding additional projectiles! If they get to level three, they should have three projectiles. Add two more Instantiate statements inside of the if statement.

```
*****\n| Add your code below |\n*****\n\nInstantiate(projectile, transform.position, transform.rotation);\nif (currentLevel >= 3)\n{\n Instantiate(projectile, transform.position, transform.rotation);\n Instantiate(projectile, transform.position, transform.rotation);\n}\n\n*****\n| Add your code above |\n*****
```

- 60** Play your game, get to level 3, and see what happens! It's hard to tell, but even though it looks like the player is only firing one projectile, there are actually three that have the same exact position in the game scene.



- 61** We can fix this by changing the x positions of our two new projectiles we are spawning.

Above the two Instantiate lines, create a new Vector3 variable named rightOffset and set it equal to a new Vector3 with an x value of 0.2f, a y value of 0, and a z value of 0.

```
if (currentLevel >= 3)
{
 Vector3 rightOffset = new Vector3(0.2f, 0, 0);
 Instantiate(projectile, transform.position, transform.rotation);
 Instantiate(projectile, transform.position, transform.rotation);
}
```

- 62** After our rightOffset variable, create a second new Vector3 variable named leftOffset and set it equal to a new Vector3 with an x value of -0.2f, a y value of 0, and a z value of 0.

```
if (currentLevel >= 3)
{
 Vector3 rightOffset = new Vector3(0.2f, 0, 0);
 Vector3 leftOffset = new Vector3(-0.2f, 0, 0);
 Instantiate(projectile, transform.position, transform.rotation);
 Instantiate(projectile, transform.position, transform.rotation);
}
```

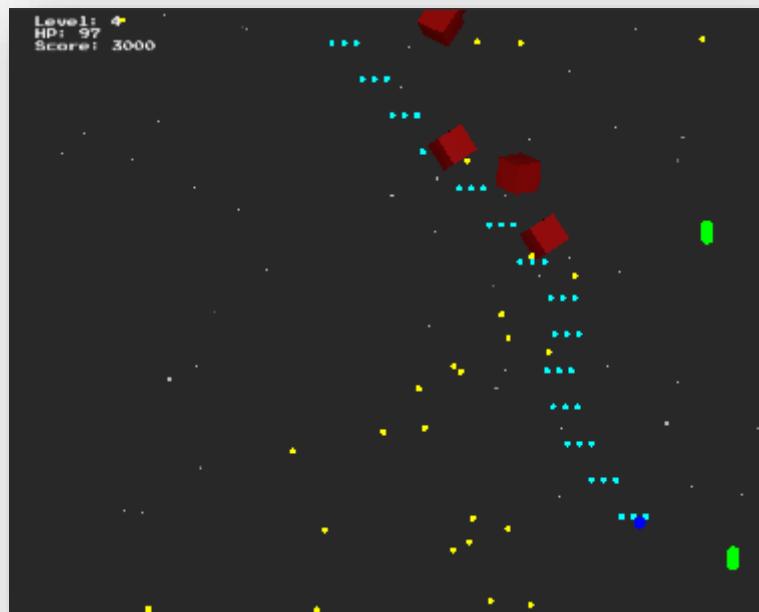
- 63** We now need to add the right and left offsets to the position of our new projectiles. When adding vectors, each number gets added individually. So a Vector3(1, 2, 3) plus a Vector3(4, 5, 6) would equal Vector3(1 + 4, 2 + 5, 3 + 6) or Vector3(5, 7, 9). In the first Instantiate, change the second argument to be transform.position + rightOffset.

```
if (currentLevel >= 3)
{
 Vector3 rightOffset = new Vector3(0.2f, 0, 0);
 Vector3 leftOffset = new Vector3(-0.2f, 0, 0);
 Instantiate(projectile, transform.position + rightOffset, transform.rotation);
 Instantiate(projectile, transform.position, transform.rotation);
}
```

- 64** In the second Instantiate, change the second argument to be transform.position + leftOffset.

```
if (currentLevel >= 3)
{
 Vector3 rightOffset = new Vector3(0.2f, 0, 0);
 Vector3 leftOffset = new Vector3(-0.2f, 0, 0);
 Instantiate(projectile, transform.position + rightOffset, transform.rotation);
 Instantiate(projectile, transform.position + leftOffset, transform.rotation);
}
```

- 65** Play your game and see how adding offsets changed the extra projectiles in level 3 and above.



# Prove Yourself

## Task

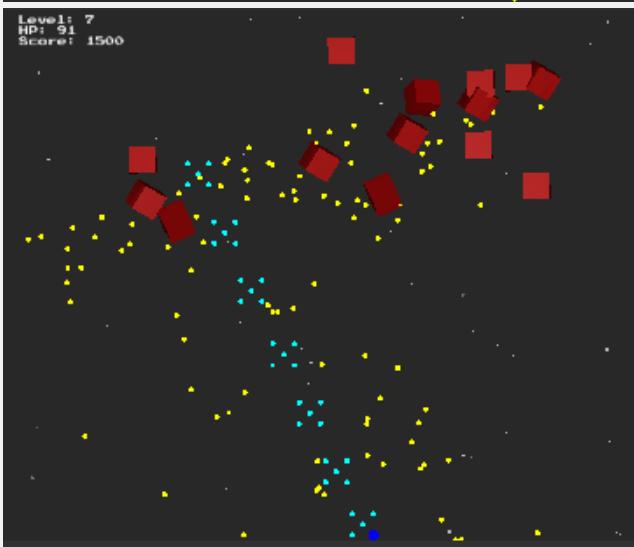
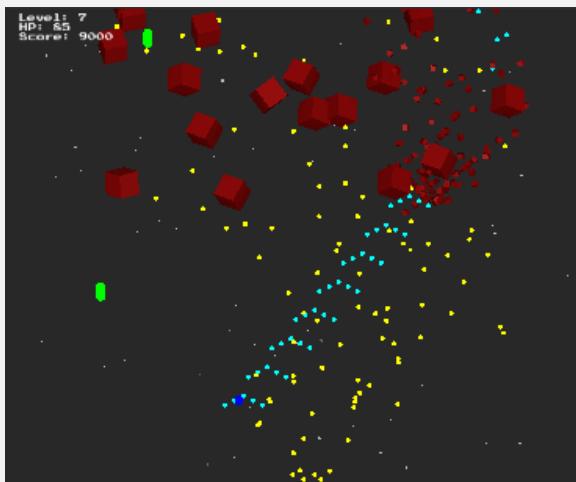
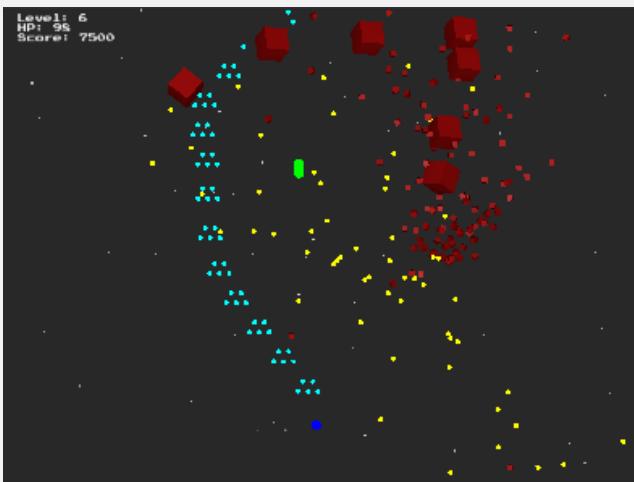
Using similar logic for giving the player more projectiles at level 3, create a new `if` statement that checks to see if the player is at level 5 or above.

Inside the `if` statement, instantiate at least two more projectiles.

Create two more offset `Vector3` variables to give these two new projectiles different positions than the other three.

How many cool patterns can you create?

Check out these examples of patterns:



The player gets upgrades for leveling up, but the enemies never get any tougher.

In the **EnemyEmitterController** script, add a new **public** game object variable.

Come up with a name and assign it the **GiantEnemy** Asset in the **Inspector** by clicking on the little circle to open the **Game Object** selector.

In the **Update** function create your very own conditional statement that will determine when this new **GiantEnemy** will spawn. You can place this inside or after the **for** loop.

```
*****\n|**** Add your code below ****|\n*****\n\nfor (int i = 0; i < playerController.currentLevel; i++)\n{\n float randomX = Random.Range(-6.0f, 6.0f);\n Vector3 enemyPosition = new Vector3(randomX, 6, 0);\n transform.position = enemyPosition;\n Instantiate(Enemy, transform.position, transform.rotation);\n\n if ()\n {\n }\n}\n\n*****\n|**** Add your code above ****|\n*****\n\n*****\n|**** Add your code below ****|\n*****\n\nfor (int i = 0; i < playerController.currentLevel; i++)\n{\n float randomX = Random.Range(-6.0f, 6.0f);\n Vector3 enemyPosition = new Vector3(randomX, 6, 0);\n transform.position = enemyPosition;\n Instantiate(Enemy, transform.position, transform.rotation);\n}\nif ()\n{\n}\n\n*****\n|**** Add your code above ****|\n*****
```

*Inside for loop*

*After for loop*

Use the other lines of code for inspiration, but make sure to **Instantiate** with your **variable**, a **position**, and a **location**. You can also use a **modulo** (%) operator. This operator gives the remainder of the number divided. For example,  $5 \% 2$  would give a result of 1, since  $5 / 2$  has a remainder of 1. You can use modulo to spawn in a boss every couple of levels. For example, to spawn in a boss every three levels, it would look something like “`level % 3 == 0`”, since `level % 3` will be zero with no remainder at level 3, level 6, level 9, and so on.

## Activity 9

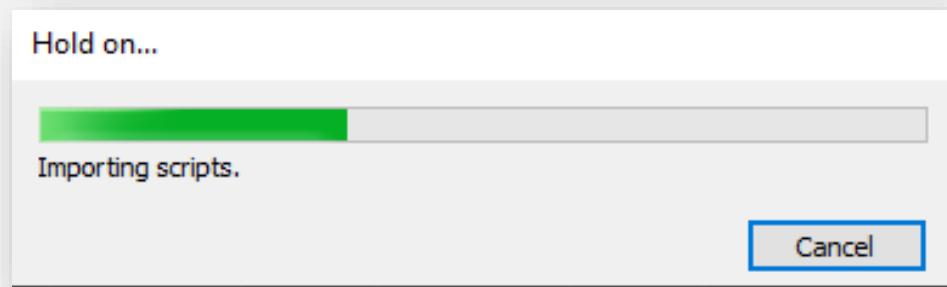
# Labyrinth

A familiar maze awaits you. There's just one trick – you can no longer control Codey the Ninja. You could probably tell Codey what to do programmatically, by writing code telling Codey how many steps to take in which direction until the exit is found, but is there a better way? Unity has a built-in system known as the **NavMesh** that you can use instead. What does this **NavMesh** do?

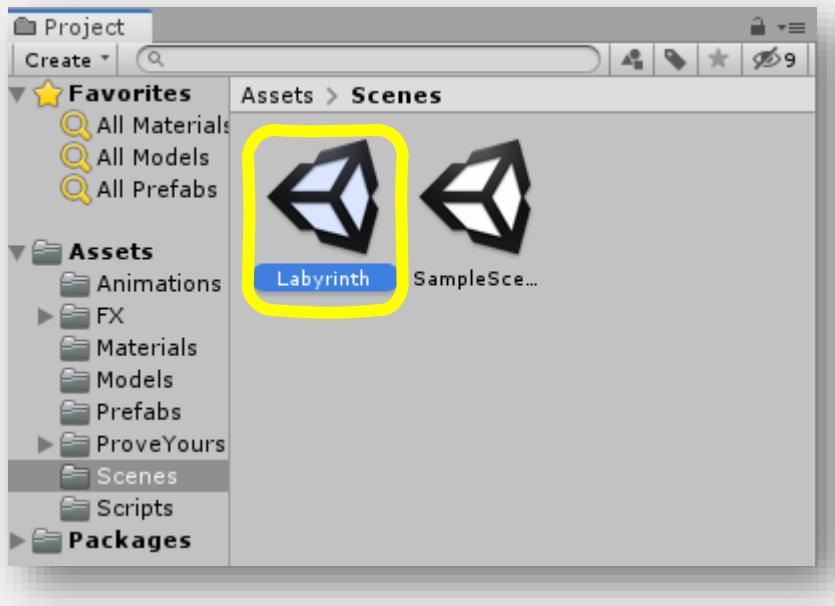
The **NavMesh** is an AI navigation system that comes with Unity that is very powerful, but the essentials are very basic. It tells a target object, called an agent, where it can and cannot move. It then uses this information to figure out the shortest path to its destination. You will help Codey escape the labyrinth from Find the Exit using this NavMesh system. You will learn how to apply the NavMesh, which decides what is and isn't a walkable terrain, to a scene, and how to interact with the NavMeshAgent component by having it control elements of Codey's automatic behavior.



- 
- 1 Start a new Unity Project and name it *YOUR INITIALS - Labyrinth*. Select **3D core** then **Create**.
  - 2 We've created a starter pack to give you a head start! To use it, import the **Labyrinth\_Ninja.unitypackage** by going to **Assets > Import Package > Custom Package > All > Import**.



- 
- 3 To open the starter package, double-click on the **Labyrinth** scene. You can find this in the **Project** tab under **Assets > Scene > Labyrinth**.



- 
- 4** This will load our **Hierarchy** with game objects alongside what looks like the original starting version of Find the Exit.



- 
- 5** Play your game and see what happens.

The maze is set up, and the walls and effects work, but the Player has no movement code.



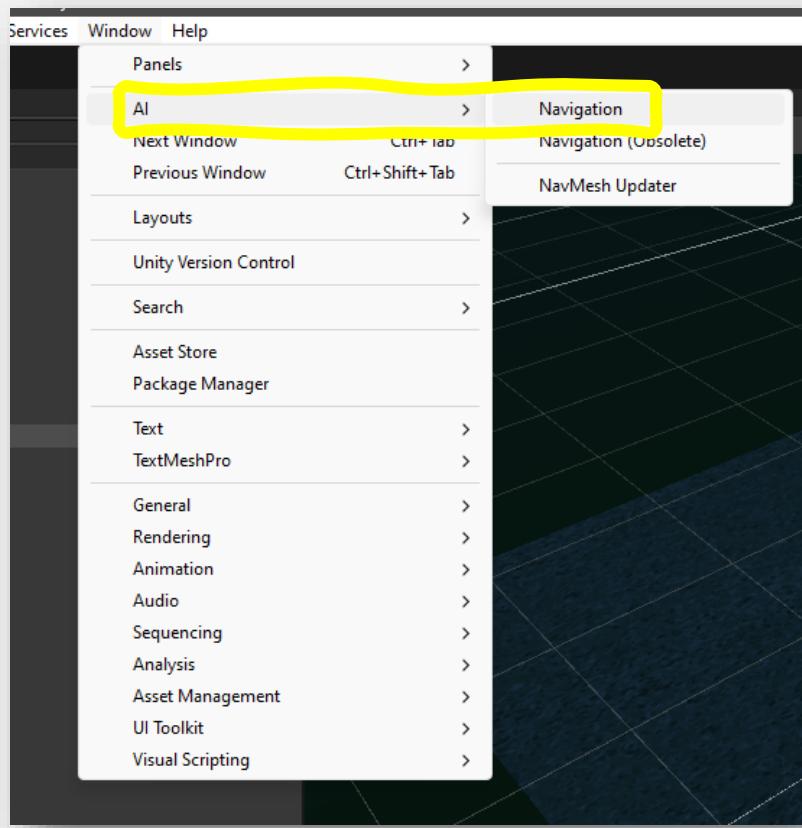
**6** In order to navigate Codey to the exit, the game will need **three pieces** of information provided by you.

- What parts of the scene are “**walkable**”; where should Codey be allowed to move?
- What parts of the scene are “**non-walkable**”; where Codey will have to move around or over?
- Who is the “**agent**”, the object trying to navigate the scene?

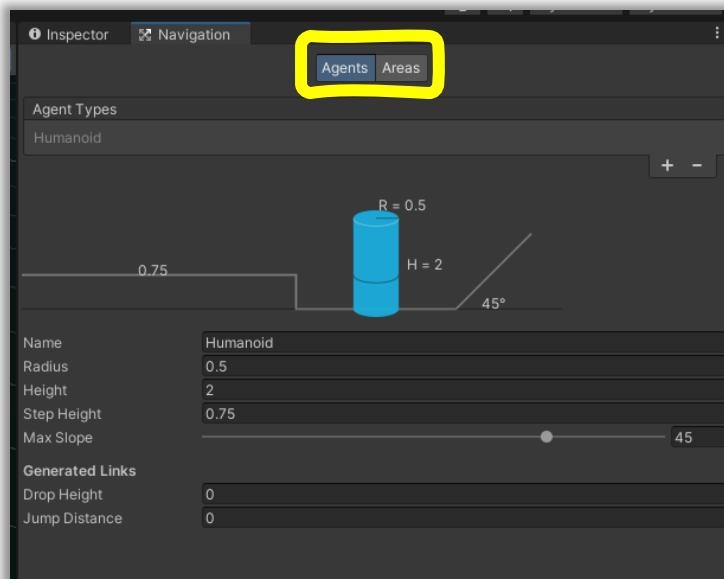
Below you can see all the answers to the questions from above.



- 7** You will need to add a window next to your Inspector window that shows navigation information. To do this, go to the menu, and find your way to **Window > AI > Navigation**. If it is not here, make sure to install the “AI Navigation” package from the Unity Registry.

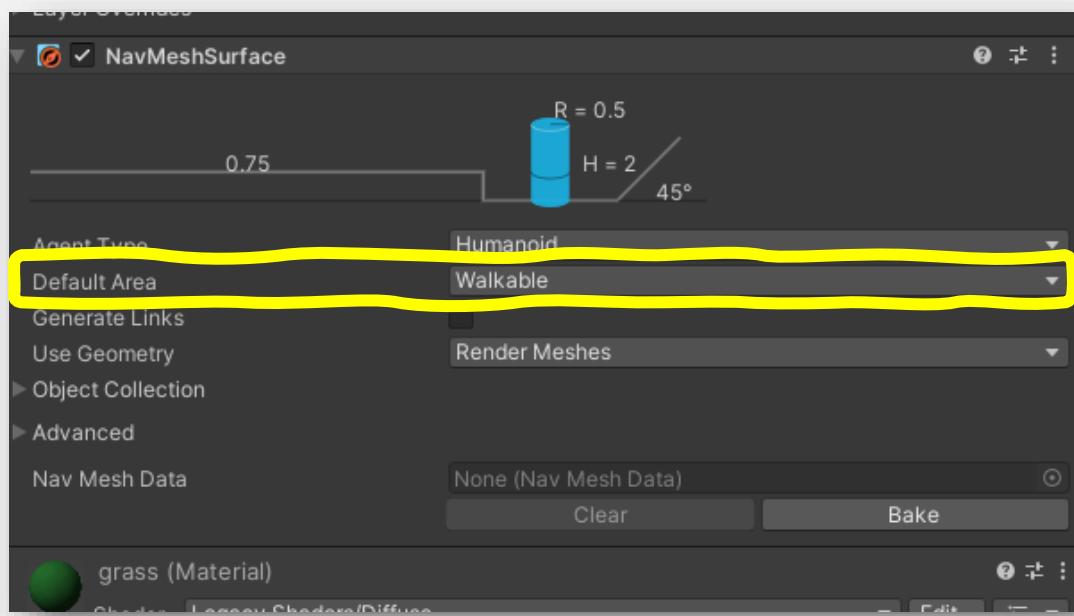


A new window will be added to your Unity interface. You'll see two different sub-menus in the Navigation panel, **Agents** and **Areas**.

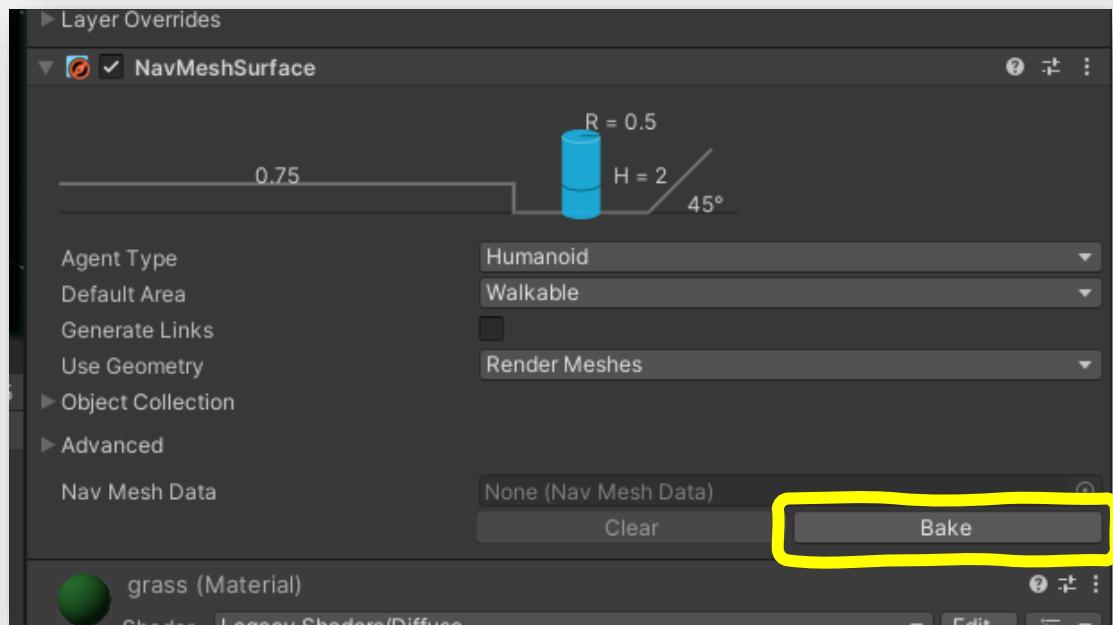


- 8** In your Hierarchy select the Ground Object. Add the **NavMeshSurface** component.

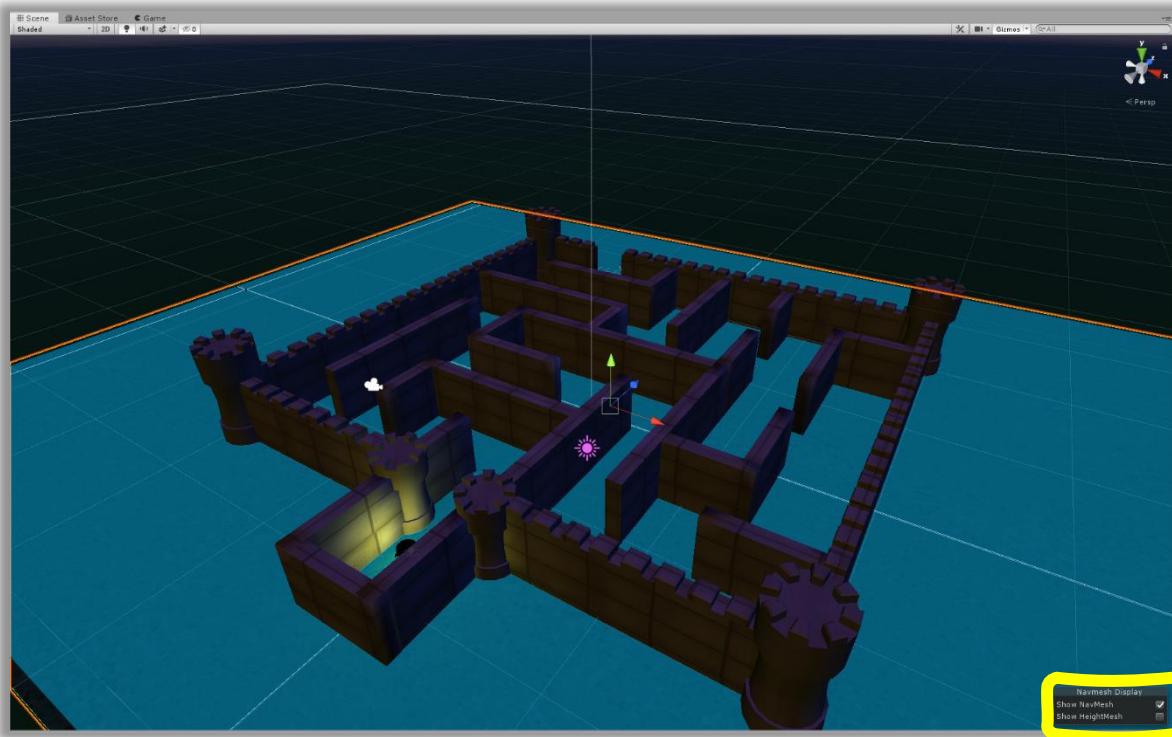
By default the **Navigation Area** is set to **Walkable**, leave that as it is.



- 9** At the bottom of the component menu click on **Bake**.



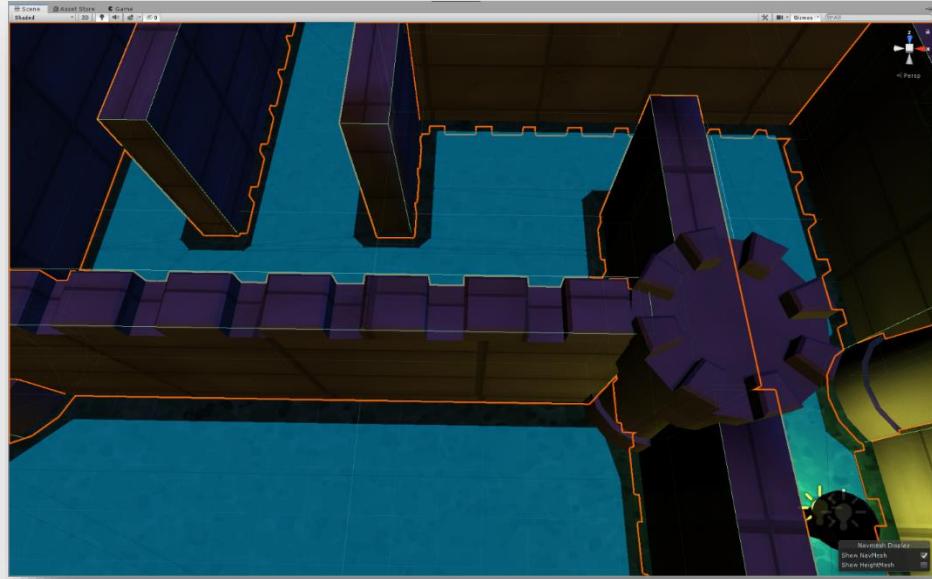
- 10** After Unity has completed “baking” the Ground will turn blue. This means that the NavMesh has been built unto the scene.



At the bottom right, you'll see a menu to turn this blue highlighted area for the NavMesh on or off. See what it does when you uncheck it.

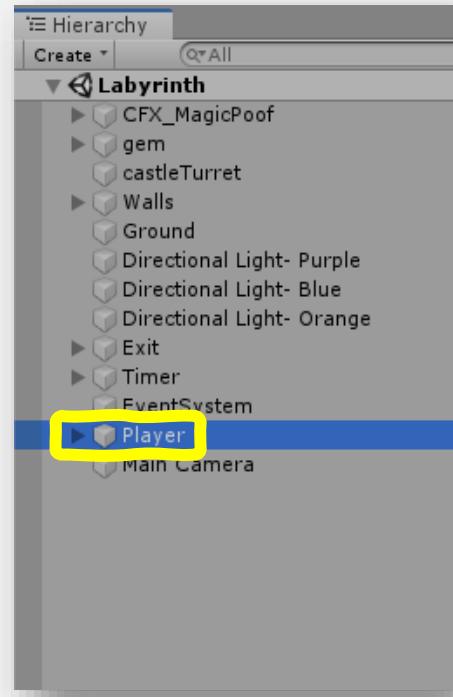


- 11** Look at your **Scene**, notice the gaps where the walls are in the ground NavMesh.

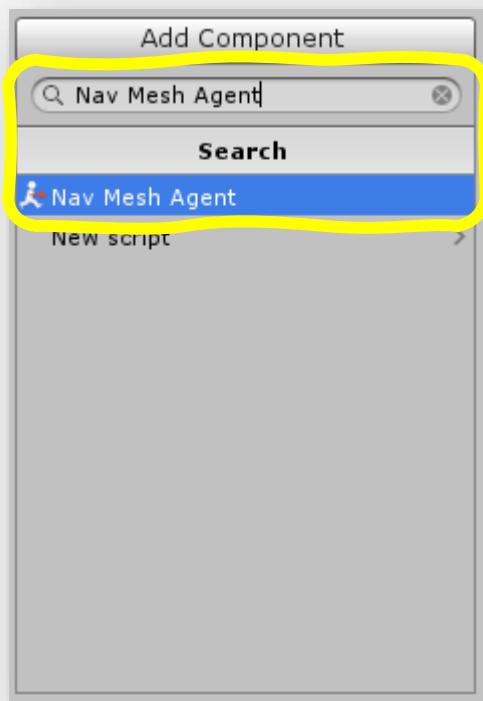


- 12** The scene now has the information where NavMeshAgents can and cannot walk. All we have left to do is assign an object as an agent. Can you guess which object will be our agent?

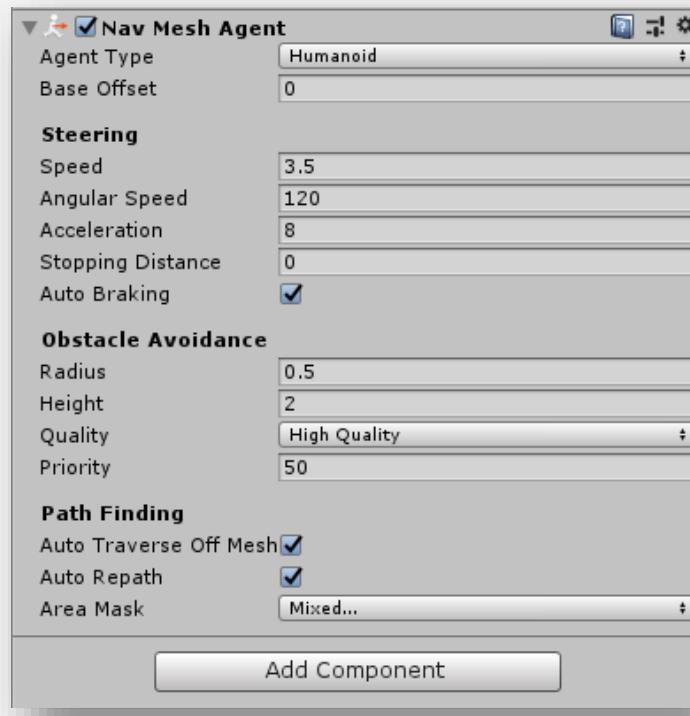
It's going to be our **Player** object!  
Select the Player in the Hierarchy.



- 13** In the previous two parts the NavMesh was not a component, but the **NavMeshAgent** is. In the **Inspector** we want to click **Add Component** and type **NavMeshAgent**.

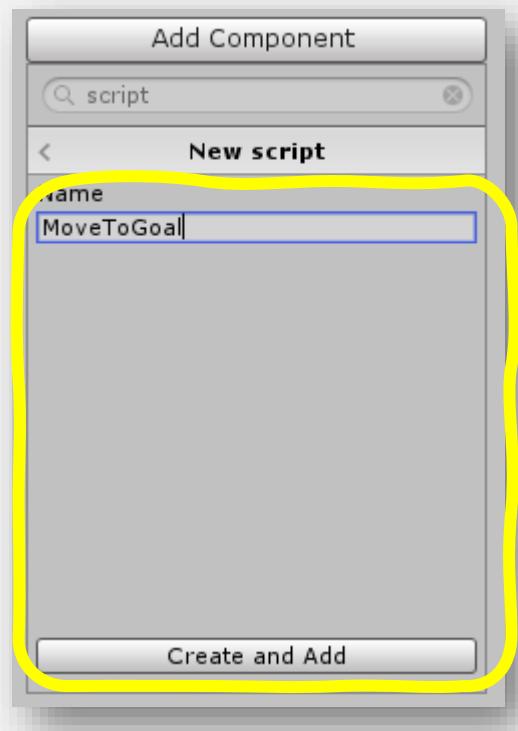


- 14** For now the properties won't be changed.



- 15** While the NavMeshAgent does a lot for us, we still need to provide our Player with more information. With the Player still selected in the **Hierarchy**, go to **Inspector** and click **Add Component**. Type in **New Script** and call it **MoveToGoal**, because that is exactly what it's going to tell the Player to do!

Open up the **MoveToGoal** script by double clicking on it.



- 16** At the top of every script, we've been telling Unity what we need from its library of functions and capabilities. We need to add `UnityEngine.AI;`

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.AI;
5
```

**17** We are going to need to initialize three variables. Type `public Transform goal;` as this will help the **Player** know it's target destination. Secondly type `private Animator animator;` so we can give Codey their running animation. Lastly type `private NavMeshAgent agent;` this way we set the agent itself.

```
6 public class MoveToGoal : MonoBehaviour
7 {
8 public Transform goal;
9 private Animator animator;
10 private NavMeshAgent agent;
11 }
```

In the screenshot above the `Start()` and `Update()` functions have been deleted, but will be added later. You can decided to keep them or delete them.

**18** Let's work on our `Start()` function. We are going to assign the animator as `animator = GetComponentInChildren<Animator>();` so we can get the the Animator component from Codey.

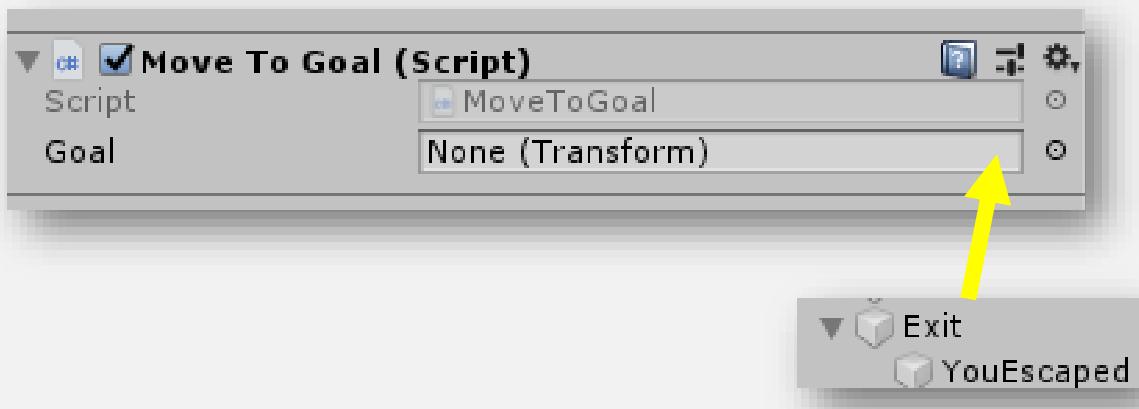
Next, type `agent = GetComponent<NavMeshAgent>();` as we want to store the `NavMeshAgent` components to `agent`.

Right after the code you just wrote type, `agent.destination = goal.position;` this way our agent will know our goal's position.

```
12 private void Start()
13 {
14 animator = GetComponentInChildren<Animator>();
15 agent = GetComponent<NavMeshAgent>();
16 agent.destination = goal.position;
17 }
```

Wait a minute, we have not assigned our goal so how will the agent know the goal position? **Save** your script and return to Unity.

- 19** Since we set up our goal as a public variable, we can assign its value in the **Inspector**. Make sure you are still in the **Player object** and find the **MoveToGoal** script. Next to Goal, drag the **Exit** game object.



The agent variable will now know what the goal position is and where it has to move to!

- 20** Play your game and see what happens. Codey is able to find his way to the Exit with no help or input!

- 21** Here we learn about the first and most important NavMeshAgent built-in properties **destination**.

Setting the destination will tell the NavMeshAgent where it needs to go.

This is why we set destination to our goal variable, **Exit**.

You can notice that Codey doesn't go through walls, this is because earlier we **Baked** the **Walkable** and **Non-Walkable** surfaces of the scene and it did all the work for us.

- 
- 22** So far Codey will just float to the goal with no animations. Let's fix that!

In the original Find the Exit, the **player input** would trigger his animations. Since the player doesn't give the game input anymore, we need a new condition to trigger animations.

We will be doing this by another important property of the NavMeshAgent, the Boolean property **hasPath**.

As long as the NavMeshAgent has a **path** to their set destination this will return **true**.

- 
- 23** When was the last time that you used the **Animator**? We simply need to use **SetBool** based on whether or not **hasPath** is returning **true** or **false**.

If the NavMeshAgent has a path to it's destination the "**isRunning**" animation is set to **true**. Otherwise, if the NavMeshAgent does not have a path to it's destination, the "**isRunning**" animation is set to **false**.

Type the following in the **Update()** function:

```
19 private void Update()
20 {
21 if (agent.hasPath)
22 {
23 animator.SetBool("isRunning", true);
24 }
25 else
26 {
27 animator.SetBool("isRunning", false);
28 }
29 }
30 }
```

---

**24** Save your script and playtest your game. Codey will now run properly all the way to the exit without any input!

If everything works properly let's submit your game before moving on.

---

**25** Now that you know the basics of NavMesh, let's get a little creative with the next Prove Yourself activity.

# Prove Yourself

## Task

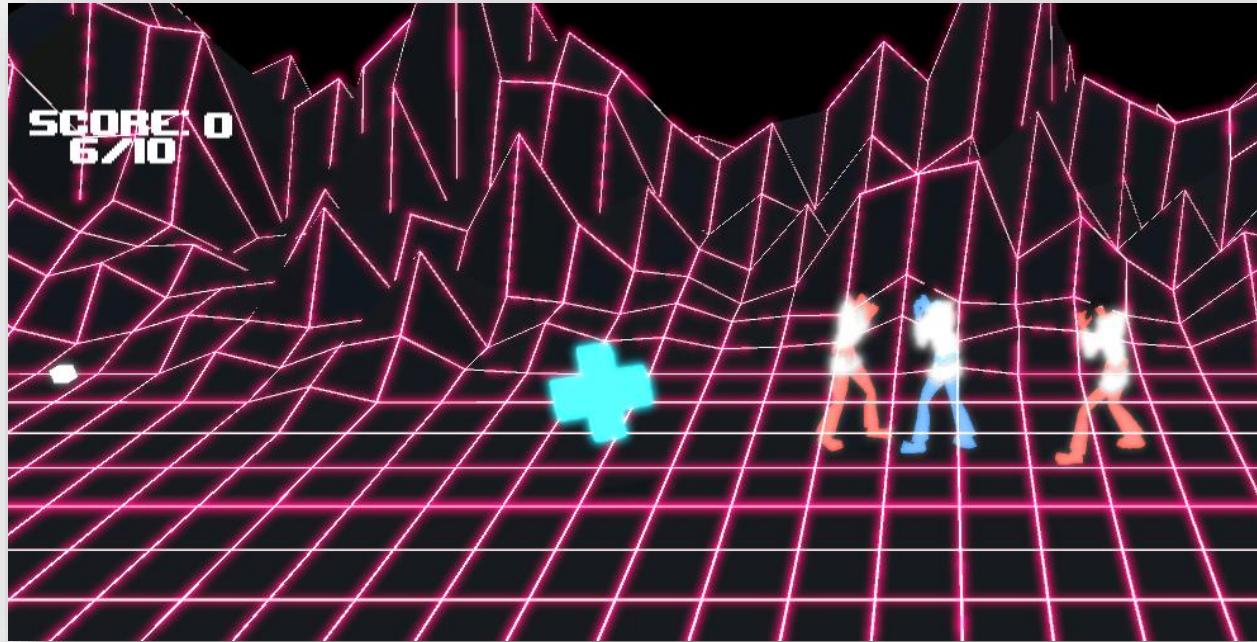
- Can you figure out which properties to edit in the Player's NavMeshAgent component to solve the maze in less than 30 seconds?  
*Hint: the properties you need are contained within the Steering section.*
- Create any new 3D game object and apply the item pickup code that you learned in earlier activities to this new object. This is set as the NavMeshAgent's first destination. After it is grabbed, it's removed using the `Destroy()` function and then a second destination is set – the original exit.
- Try opening up the PY screen in the folder labeled **PY**. Use your knowledge of NavMesh to help Codey Automatically get to the exit. You must set up the NavMesh surfaces, the Agent, and you can re-use the script from earlier, but you just have to set a new destination.



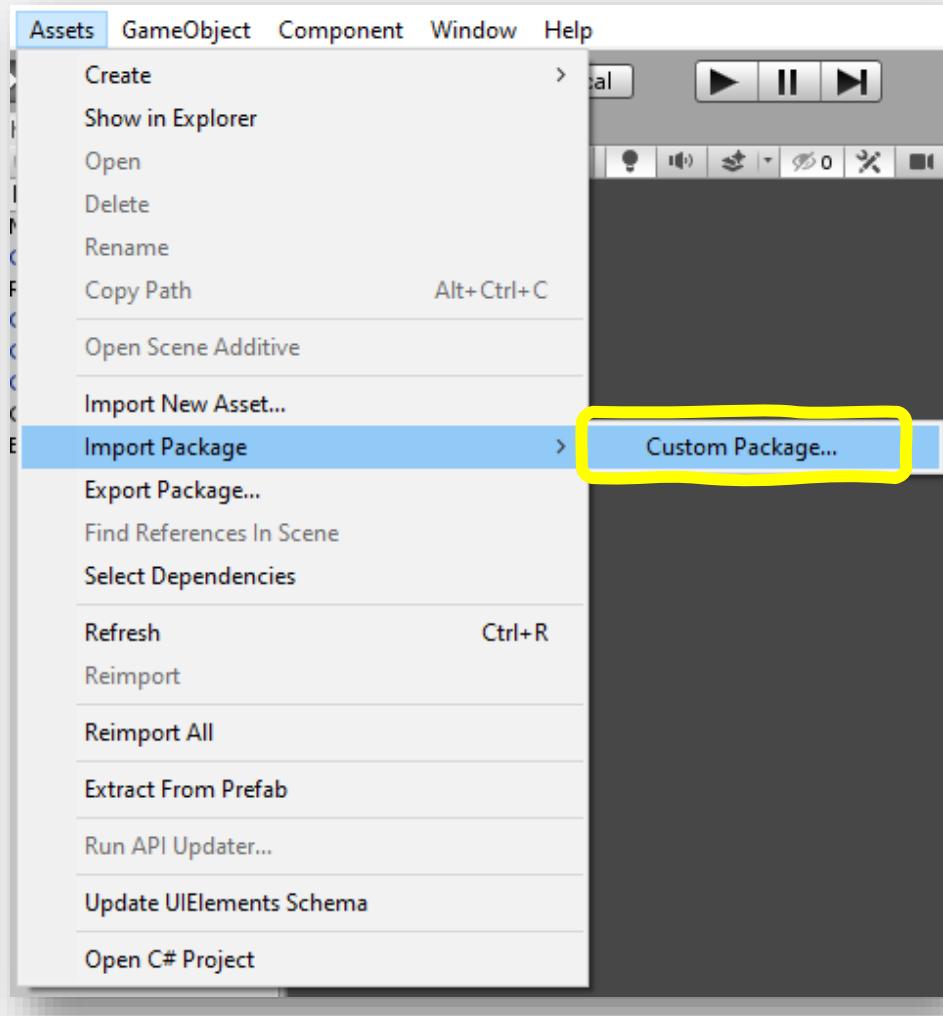
## Activity 10

# CyberFu - Part 2

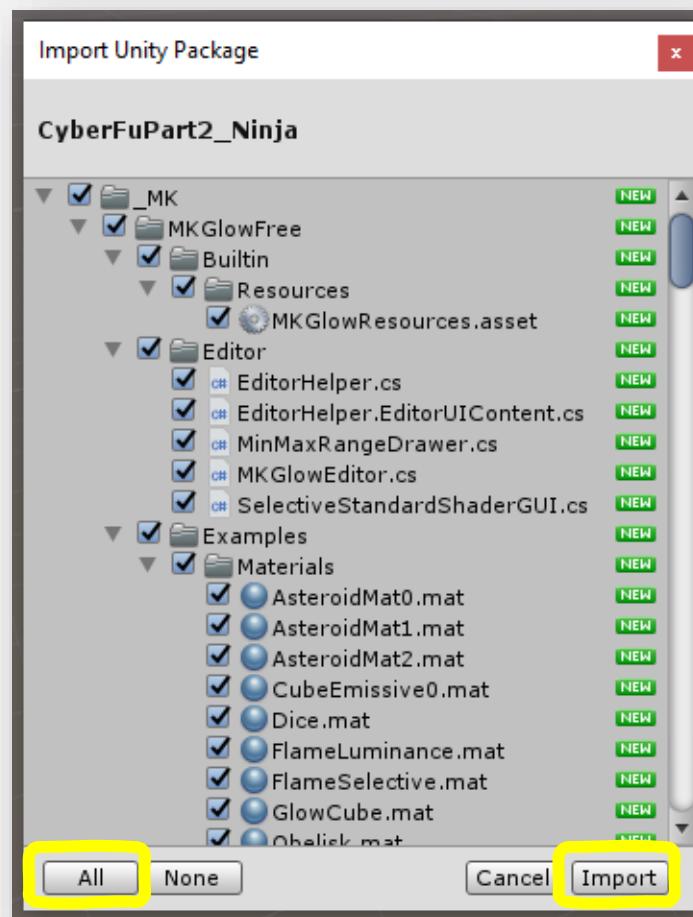
In this lesson, you will add health to both the player and the enemy, then program what happens when they run out of health.



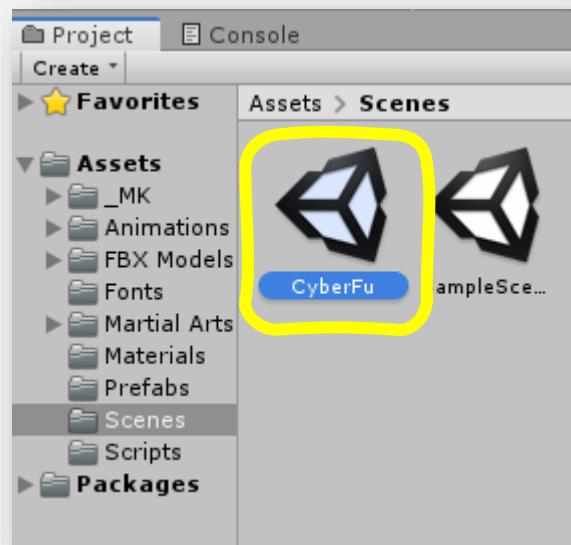
- 
- 1** Start a new **Unity** Project and name it **YOUR INITIALS - CyberFu Part 2**. Select the **3D core** and place the project it in the correct folder. Click the “**Create**” button.
- 2** After **Unity** loads your new project, **import** the CyberFu Part 2 Ninja Unity Package by clicking on **Assets** -> **Import Package** -> **Custom Package**.



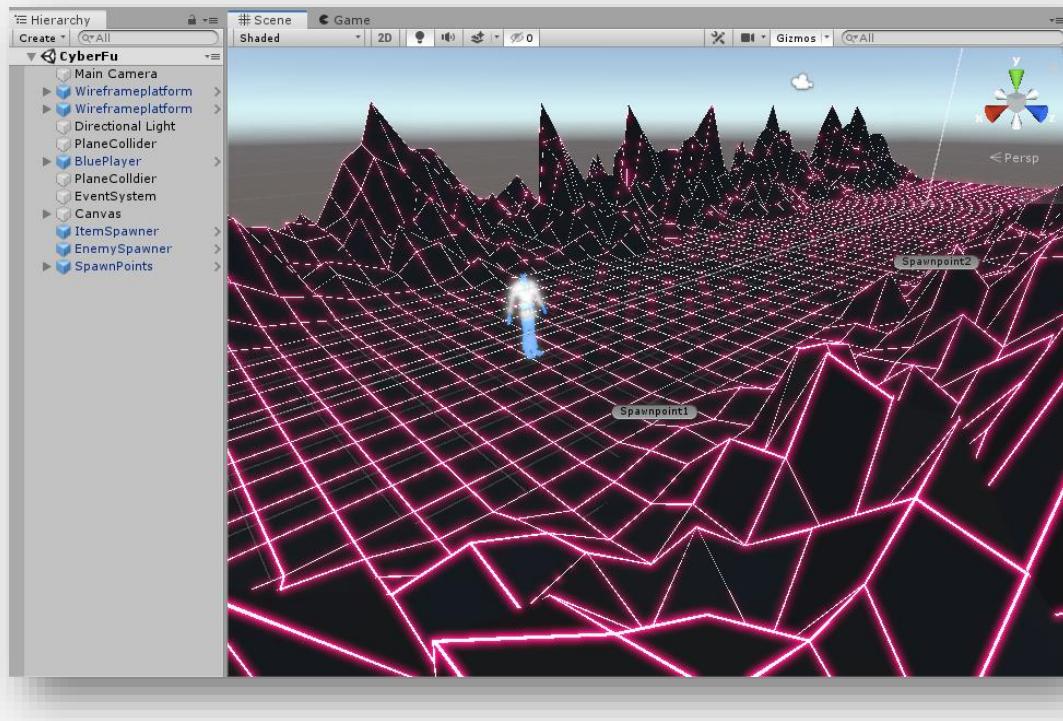
**3** On the Import Unity Package window, click “All” and then “Import”.



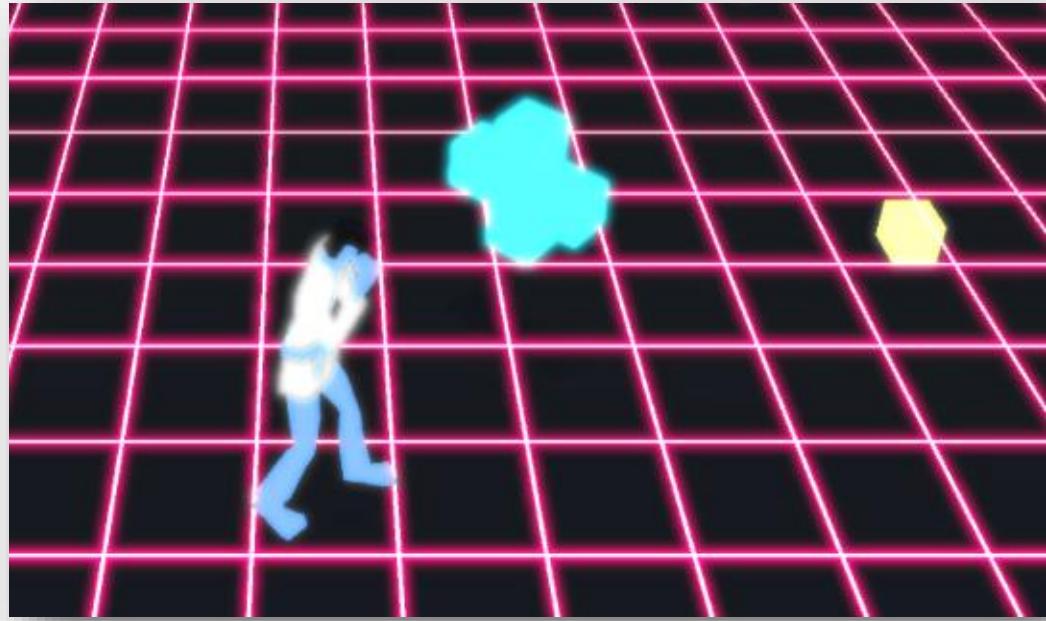
**4** After it is done importing, double click the **CyberFu** scene located in Project -> Scenes -> Assets.



- 5** This will load the **scene** with all our game objects! Make sure can see all the objects and the assets in the **Hierarchy**.

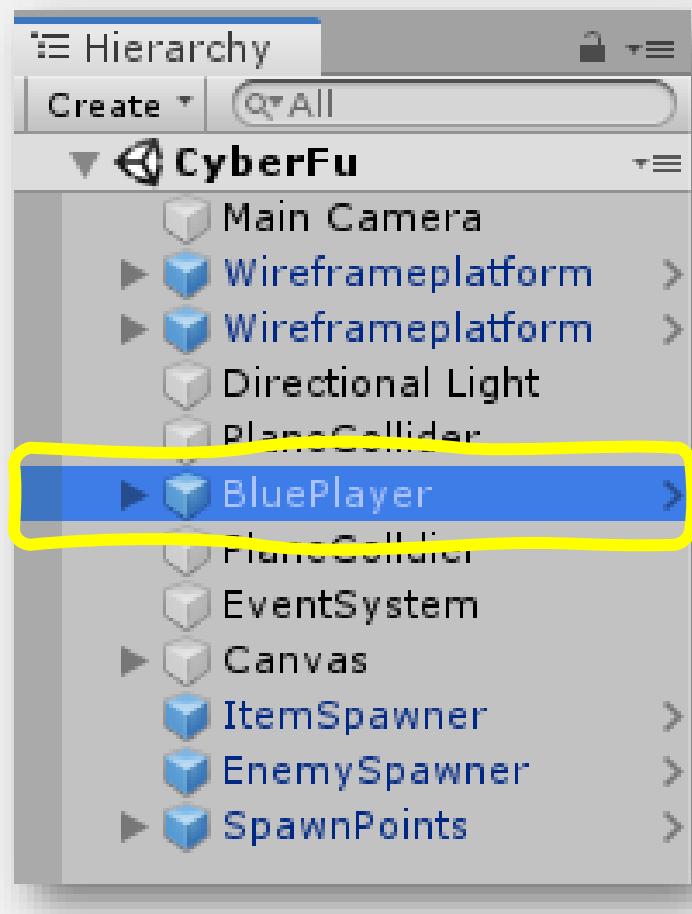


- 6** Play your game. What happens **touches** a blue or yellow **pickup**? What happens if an **enemy touches** a **pickup**?

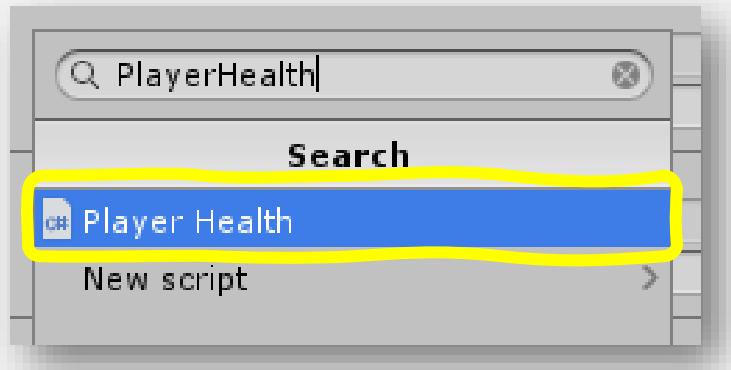


- 7** One of the **assets** that we **imported** was the **PlayerHealth** script. We need to add this to the **BluePlayer** game object.

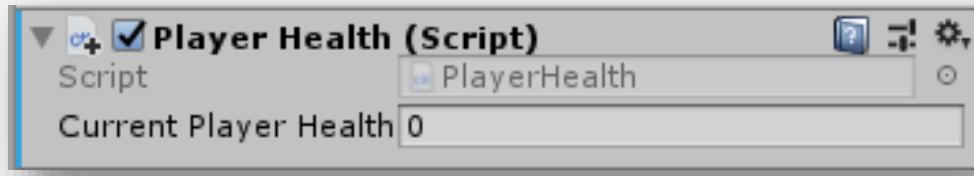
In the **Hierarchy**, click on **BluePlayer**.



- 8** Click the **Add Component** button. Search for "**PlayerHealth**" and select the **Player Health** result.



- 9** The **BluePlayer** game object now has a **Player Health (Script)** component.



- 10** Double click the **script** to open it in **Visual Studio**.

- 11** We need to create **public variables** to define the **properties** of the player's health.

There is already a variable to store the health of the player. Its initial value is set to **10**.

```
public class PlayerHealth : MonoBehaviour
{
 public int currentPlayerHealth = 10;
```

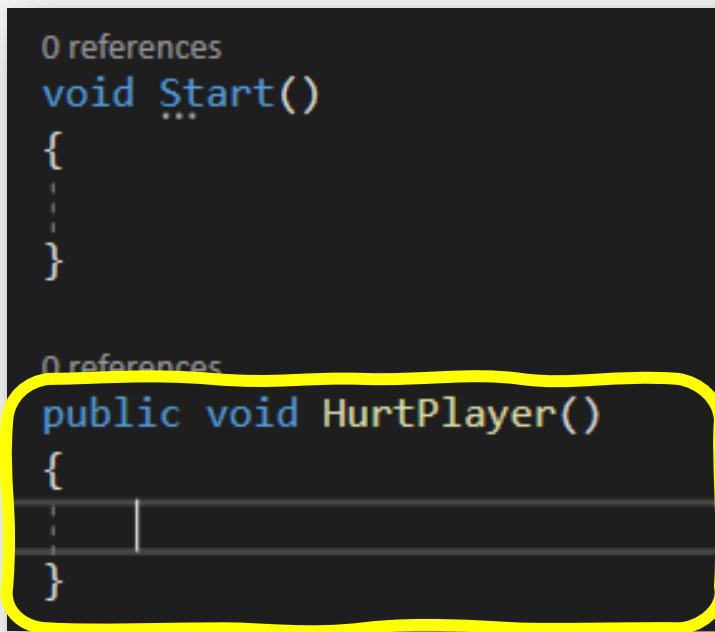
- 12** After the `public int currentPlayerHealth;` line, type `public int enemyDamage = 2;` to set how much damage each enemy hit causes.

```
public class PlayerHealth : MonoBehaviour
{
 public int currentPlayerHealth = 10;
 public int enemyDamage = 2;
```

You can adjust this value to make the game easier or harder!

**13** Now that we have the **variables** that store the player's health and the enemy's damage, we need to create a **function** that will help us change the **values** of the **variables**.

After the **Start** function, create a new **function** called **HurtPlayer** by typing `public void HurtPlayer() { }` making sure to leave an empty line between the curly brackets.

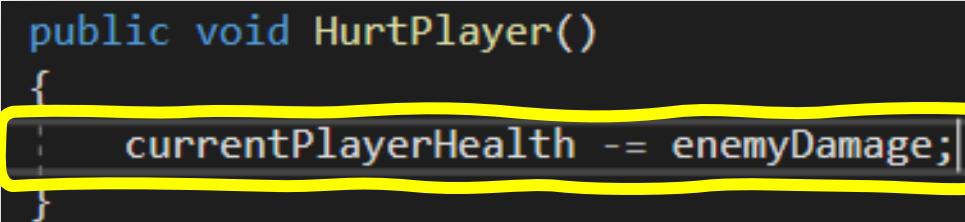


```
0 references
void Start()
{
}

0 references
public void HurtPlayer()
{
}
```

**14** Every time we call the **HurtPlayer** function we want to **subtract** the **enemy's damage** from the **player's current health**.

Type `currentPlayerHealth -= enemyDamage;` inside of the **body** of the function.



```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
}
```

**15** Play your game. The player's health never goes down, no matter how many enemies are attacking! Talk with a **Code Sensei** about why the player is not responding to enemy attacks.



**16** Our player knows how to respond to an enemy attack, it just doesn't know when it should!

After the **HurtPlayer** function, start typing "ontrigger" then **Visual Studio** might suggest a list of functions. You can select "**OnTriggerEnter**" and have **Visual Studio** auto-complete the function for you.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
}

ontrigger|
① OnParticleTrigger
② OnTriggerEnter OnTriggerEnter is called when the Collider other enters the trigger
③ OnTriggerEntered
④ OnTriggerExit
⑤ OnTriggerExit2D
⑥ OnTriggerStay
⑦ OnTriggerStay2D
```

- 17** Alternatively, you can type the entire function out yourself. Type `private void OnTriggerEnter(Collider other) { }` making sure to leave an empty line between the two curly brackets.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
}

0 references
private void ...OnTriggerEnter(Collider other)
{
 |
}
```

Unity will run the code inside of the **OnTriggerEnter** function **every time** the player **collides** with a **trigger collider**.

- 18** We want to run our **HurtPlayer** function **if** the player collides with another trigger with the special “**HitCollider**” **tag**. Inside the **OnTriggerEnter** function, check to see if the **other** object has a **tag equal to “HitCollider”** by typing `if (other.CompareTag("HitCollider")) { }` making sure to leave a line in between the curly brackets.

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("HitCollider"))
 {
 |
 }
}
```

- 19** Inside of our **if** statement, we want to run our **HurtPlayer** function. Type `HurtPlayer();` to have the player run the function **every time** the enemy attacks and **collides** with the player.

```
private void OnTriggerEnter(Collider other)
{
 if (other.CompareTag("HitCollider"))
 {
 HurtPlayer();
 }
}
```

- 20** Play your game. The player's health now correctly responds when the enemies attack.



Talk to a **Code Sensei** and **explain** how the player knows if the enemy's attack was successful or not.

- 21** We can add an animation for when the player gets hit. We need to control when the player **animates**, so we need to get a reference to its **Animator** component.

After the `public int enemyDamage = 2;` line, type `private Animator playerAnimator;` to set up the animator variable that only this script knows about.

```
public class PlayerHealth : MonoBehaviour
{
 public int currentPlayerHealth = 10;
 public int enemyDamage = 2;

 private Animator playerAnimator;
```

- 22** In the Start function after the `currentPlayerHealth = 10;` line, type `playerAnimator = GetComponent<Animator>();` to find the **BluePlayer's Animator** component and store it in the `playerAnimator` variable.

```
void Start()
{
 playerAnimator = GetComponent<Animator>();
}
```

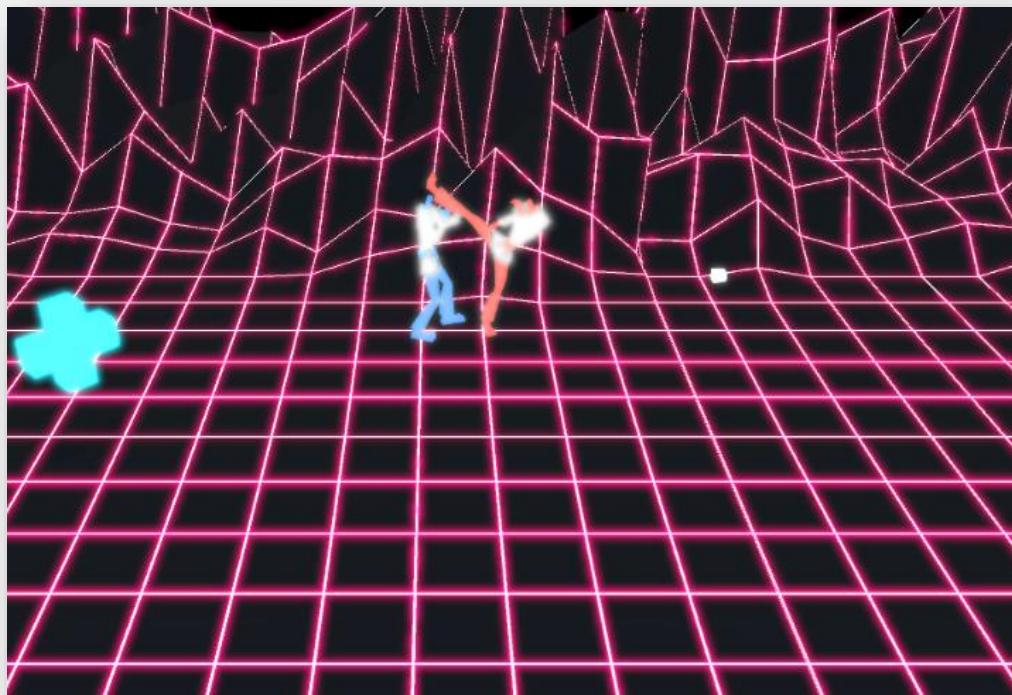
- 23** Now that we can use the player's **animator**, we need to tell it to play the "Hit" animation whenever the player receives an enemy's attack.

Talk with a **Code Sensei** about **where** this code should go. It should be wherever the player **responds** to being hit by an enemy.

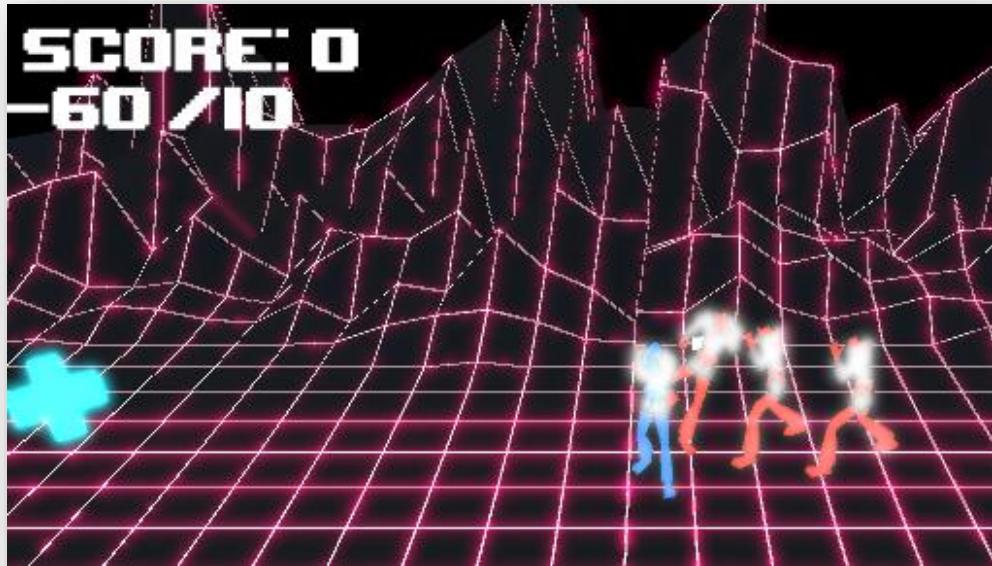
- 24** Inside the **HurtPlayer** function and after the `currentPlayerHealth -= enemyDamage;` line, type `playerAnimator.SetTrigger("Hit");` to tell the player's **animator** to **play** the "**Hit**" animation.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
 playerAnimator.SetTrigger("Hit");
}
```

- 25** Play your game. The **player** is now **animating every time** he gets **hit**!



- 26** In your **playtests**, did you ever notice what happened when the player's **health** reached **zero**? The game keeps playing and the **value** of **health** becomes **negative**!



- 27** We need to code what happens when the player runs out of health. At the end of the **HurtPlayer** function, we need to check to see if the **value** of the player's health **is less than or equal to zero** and then disable the player.

After the `playerAnimator.SetTrigger("Hit");` line, type `if (currentPlayerHealth <= 0) { }` making sure to leave an empty line between the curly brackets.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
 playerAnimator.SetTrigger("Hit");

 if (currentPlayerHealth <= 0)
 {
 }
}
```

- 28** We want to **disable** the player if his life is **zero or less**, so inside the if statement type `gameObject.SetActive(false);`

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
 playerAnimator.SetTrigger("Hit");

 if (currentPlayerHealth <= 0)
 {
 gameObject.SetActive(false);
 }
}
```

- 29** Play your game. What happens now when the player is defeated?



- 30** We want to give the player another chance by **reloading the scene** if they lose. Create a new function after the **HurtPlayer** function called **ReloadScene** by typing `private void ReloadScene() { }` making sure to leave a blank line between the curly brackets.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
 playerAnimator.SetTrigger("Hit");

 if (currentPlayerHealth <= 0)
 {
 gameObject.SetActive(false);
 }
}

private void ReloadScene()
{
}
```

- 31** In this function we want to ask Unity's **SceneManager** to reload the CyberFu scene by typing `SceneManager.LoadScene("CyberFu");` inside the body of the function.

```
private void ReloadScene()
{
 SceneManager.LoadScene("CyberFu");
}
```

- 32** We need to run this **function** when the player runs out of health. In the `HurtPlayer` function after the `gameObject.SetActive(false);` line, type `ReloadScene();` to call the **function** and reload the **scene**.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
 playerAnimator.SetTrigger("Hit");

 if (currentPlayerHealth <= 0)
 {
 gameObject.SetActive(false);
 ReloadScene();
 }
}
```

- 33** Play your game and test what happens when the player runs out of health. The game instantly reloads!

- 34** We can use the special Unity **function** named `Invoke` to **delay** running our `ReloadScene` function.

Delete the `ReloadScene();` line and replace it with `Invoke("ReloadScene", 5);` to ask Unity to call the `ReloadScene` function 5 five seconds after the **player** runs out of **health**.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
 playerAnimator.SetTrigger("Hit");

 if (currentPlayerHealth <= 0)
 {
 gameObject.SetActive(false);
 Invoke("ReloadScene", 5);
 }
}
```

## 35 Play your game.

The game should now properly reload when the player is defeated.

## 36

We can make the player's defeat a little more dramatic! After the

`public int enemyDamage = 2;` line, type `public PlayerExplosionParticles particles;` to create a **variable** for the explosion script.

```
public int currentPlayerHealth = 10;
public int enemyDamage = 2;

public PlayerExplosionParticles particles;

private Animator playerAnimator;
```

## 37

In the Start function type `particles = GetComponent<PlayerExplosionParticles>();` to get the player's **PlayerExplosionParticles component** and store it in the **particles variable**.

```
void Start()
{
 playerAnimator = GetComponent<Animator>();
 particles = GetComponent<PlayerExplosionParticles>();
}
```

- 38** In the **HurtPlayer's if statement**, remove the `gameObject.SetActive(false);` line and replace it with `particles.Explode();`.

```
public void HurtPlayer()
{
 currentPlayerHealth -= enemyDamage;
 playerAnimator.SetTrigger("Hit");

 if (currentPlayerHealth <= 0)
 {
 particles.Explode();
 Invoke("ReloadScene", 5);
 }
}
```

- 39** Play your game. Now when the player runs out of **health**, there's an explosion of particles!

# Prove Yourself

## Task

For this **Prove Yourself**, you must change the initial **value** of the player's **health** and the enemy's **damage**. You need to make sure that your **UI** reflects the changes that you make. You might need to move or change the **HitPoints** object located in the **Canvas** object.

## Activity 11

# Amazing Ninja Worlds - Part 1

Super Ninja World is a side-scrolling adventure where the player must activate and find a teleportation pad to advance to the next level. There are a few obstacles that get in the way.

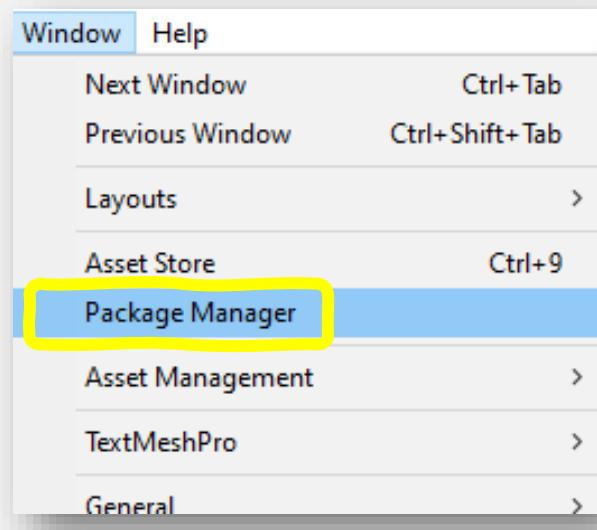
In this activity, you will start building the game by setting up a health system for the player.



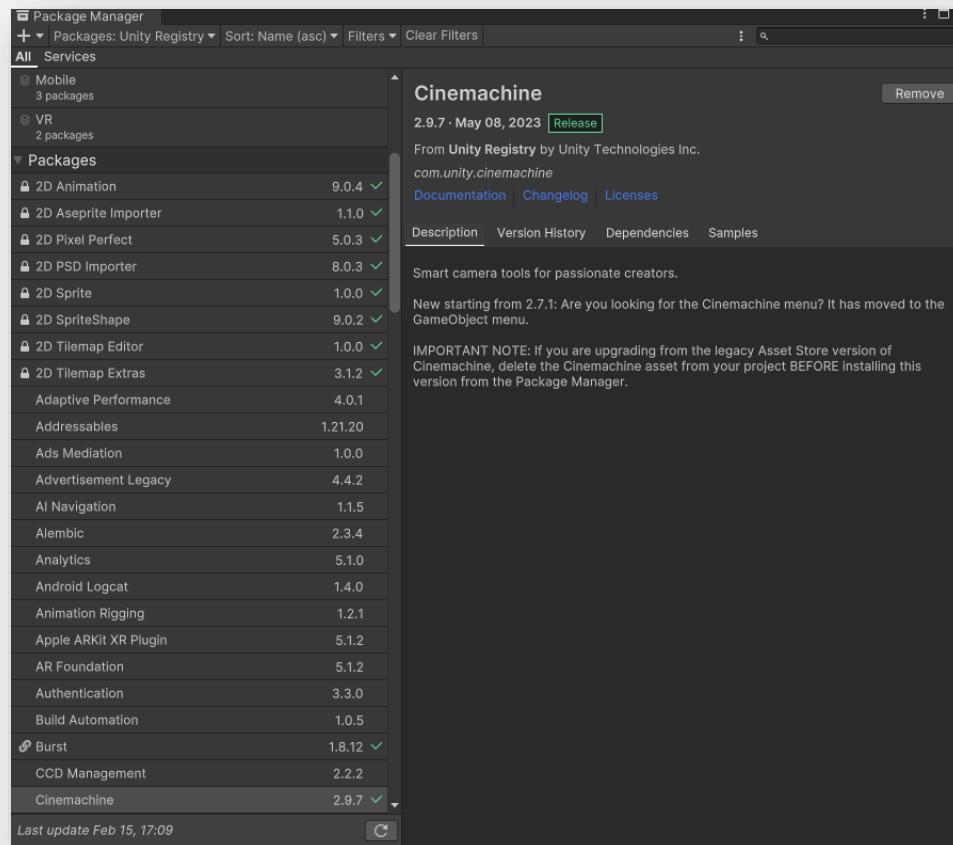
**1** Start a new Unity Project and name it **YOUR INITIALS -Amazing Ninja Worlds**. Select **3D core**.

**2** We will be using Cinemachine to control our camera. This will help us create awesome camera shots and angles for our game.  
Go ahead and open **Window > Package Manager**.

*Be patient if it doesn't open right away. It might take a minute to load.*



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

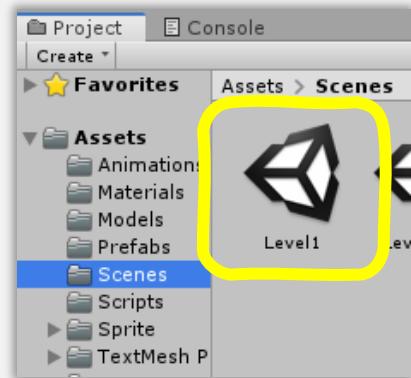


Remember to switch to Unity Registry in the top left.

**4** We've created a starter pack to give you a head start! To use it, import the **NinjaWorld1\_Ninja.unitypackage** by going to **Assets > Import Package > Custom Package > All > Import**.

**5** To open the starter package, double-click on the **Level1** scene.

You can find this in the **Project** tab under **Assets > Scenes > Level1**.

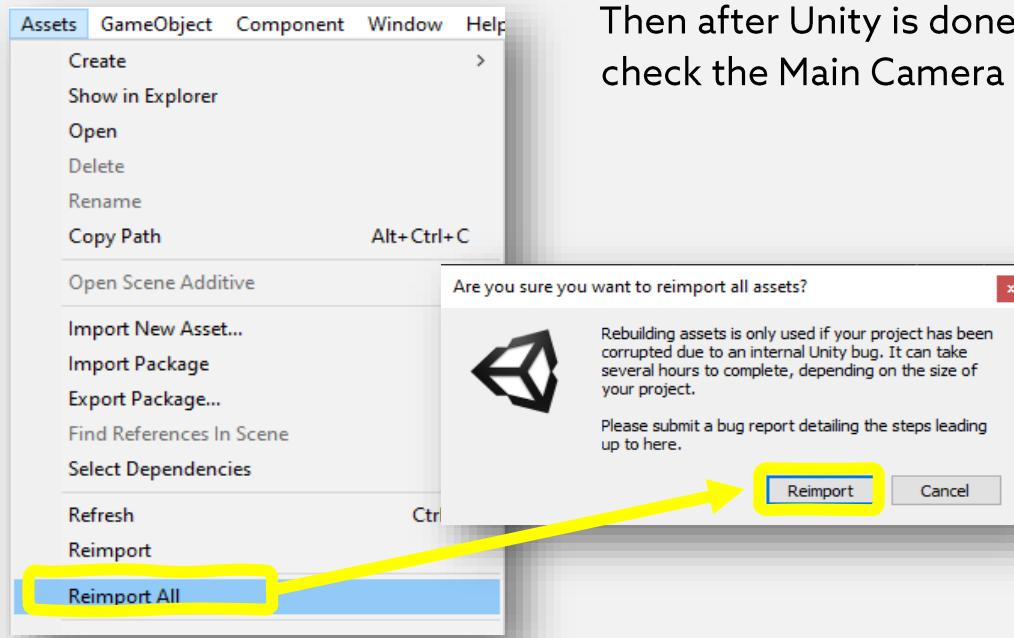


**6** We should check that Cinemachine is working like it is meant to.

In the **Hierarchy** tab, you should see a list of all the game objects in the scene. If you see a little grey and red icon attached to the Main Camera (shown in the picture), Cinemachine is working!

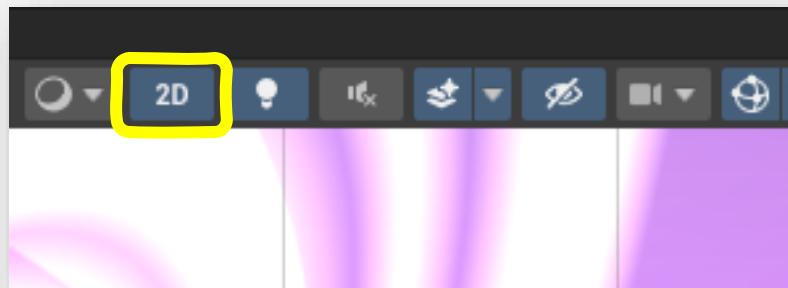


If you do not see it, go into the **Assets** tab and press **Reimport All**.

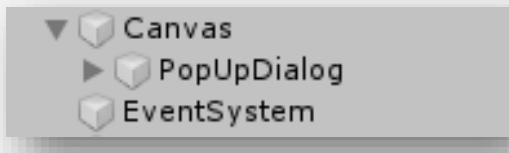


Then after Unity is done importing, check the Main Camera again.

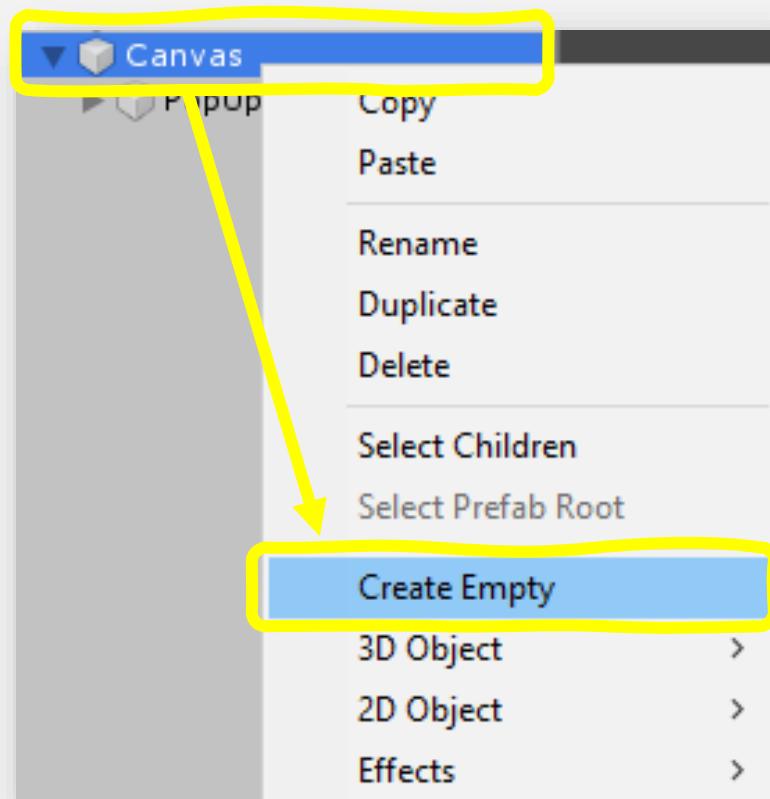
- 7** Make sure you click the 2D button on the bar on top of the Scene tab. This will align your camera properly.



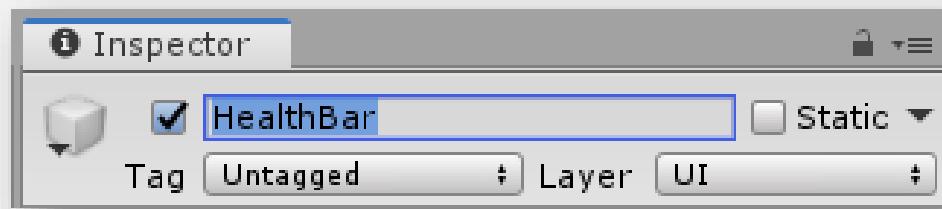
- 8** Play the game. Did you get hurt and reset? If the player touches the vines, they get sent back to the beginning. We want to add a health system in which the player has only three lives before they get a game over.
- 9** In the Level 1 Scene, find the game object named "Canvas".



- 10** Right click the "Canvas" game object and click "Create Empty".



- 11** Rename the empty object you just created from "GameObject" to "HealthBar". To do this, left click "GameObject" to select it, then change the name at the top of the inspector tab.



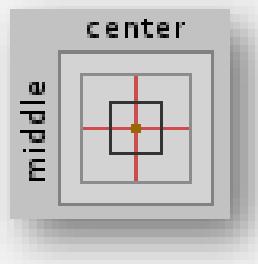
- 12** Double click on the Canvas Component to Zoom out to see the whole UI canvas.

- 13** With your new “HealthBar” object selected, look at the “Rect Transform” component in the Inspector. We need to move the health bar to the top left of its parent “Canvas” element.

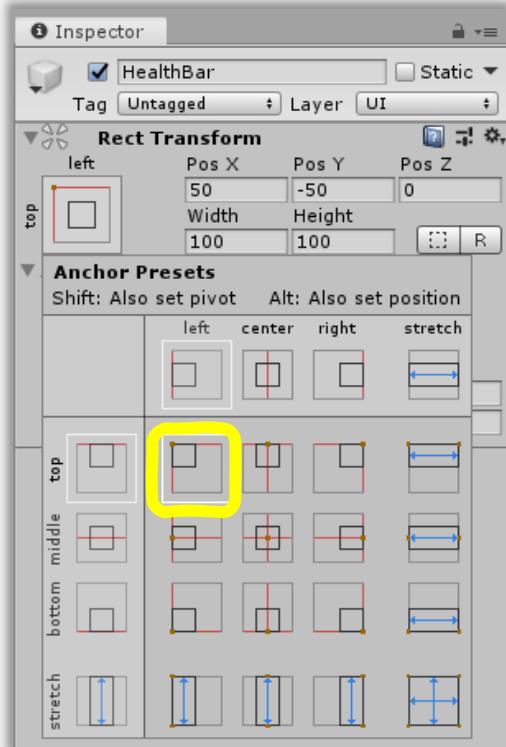
We could manually type in Pos X, Pos Y, and Pos Z values, but Unity can do it for us using Anchoring! Anchoring allows UI elements to generally stay in the same spot at different screen sizes. When we set an anchor to the top left for example, the UI element will try to keep the same distance from the top left corner, even on different screen sizes.

- 14** Click the box with the red and gray lines to open a new dialogue called “Anchor Presets”.

This will let you easily anchor or pin the HealthBar object to the top corner of the Canvas.

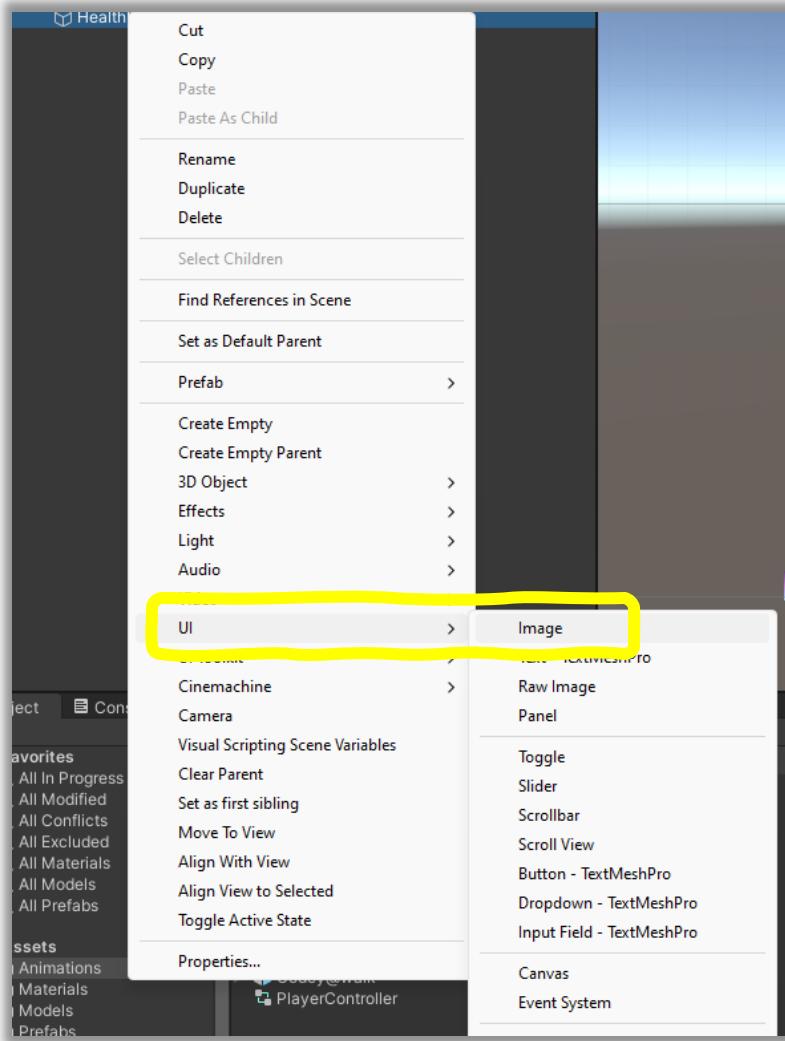


Since we want to set the position, hold the **Alt key** while you click the square circled in the image.



**15** We will use the HealthBar object to hold the images we need to make in our game UI.

Right click on the HealthBar object in the Scene Hierarchy, go to UI and then click on Image.



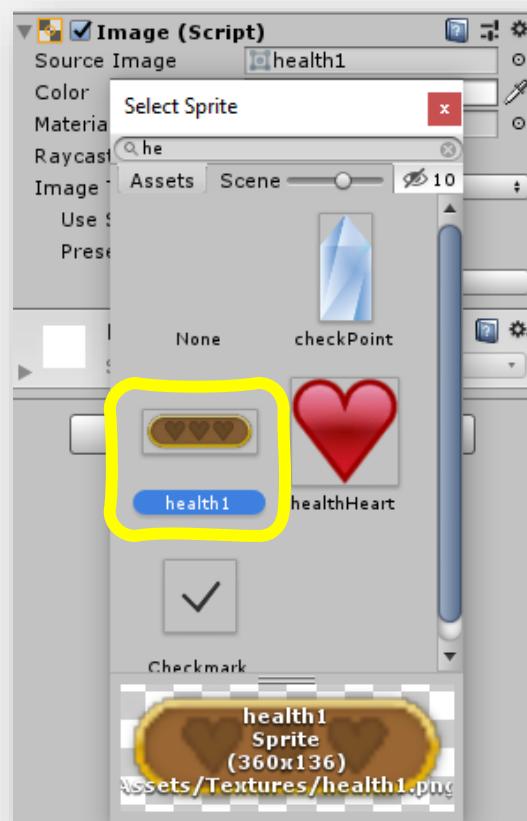
**16** Rename this image to "BG" to identify it as the background.

To do this, left click "Image" to select it, then change the name at the top of the inspector tab.

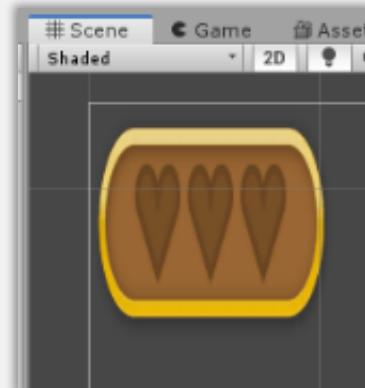
**17** In the Inspector tab, we need to set the Source Image property to our health bar background asset.

Click the tiny circle to open up the sprite selector.

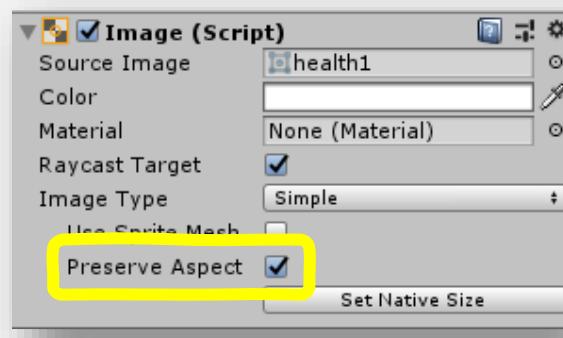
- 18** Search for “health1”, click the “health1” image, and close out the sprite selector menu.



- 19** Does your image look a little squished? If so, it's easy to fix!

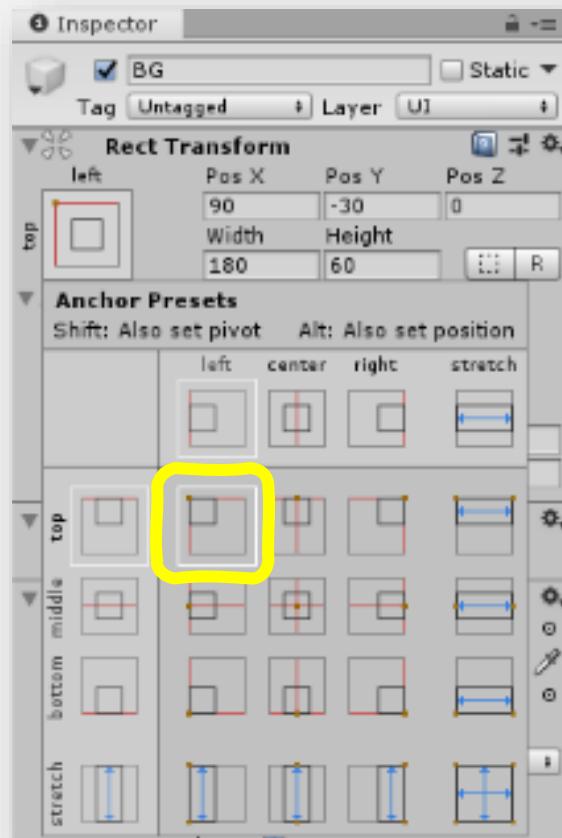


- 20** Select the BG game object. On the Image (Script) component, enable “Preserve Aspect”. This will let us drag the corner of the image to change the size without worrying about squishing or stretching it.



**21** A Width of 180 and a Height of 60 look pretty good, but you can make it as big or as small as you want!

**22** We also need move it to the top left corner of its parent. We did that earlier with the HealthBar game object! Click the box with the red and gray lines to open a new dialogue called "Anchor Presets". Since we want to set the position, hold the Alt key while you click the square circled in the image.



**23** Now that we have the background, we need to place the three hearts. Right click the HealthBar object in the Hierarchy and add an Image by finding it in the UI menu.

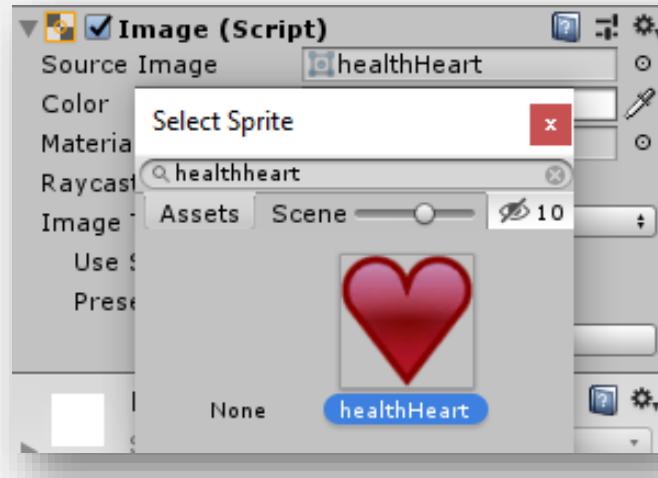


**24** Click this new game object named "Image" and rename it from "Image" to "Heart1" by using the box at the top of the Inspector tab.

**25** With the Heart1 game object open it in the inspector, find the Image (Script) component. Open the Source Image's Sprite Selector by clicking on the little circle .



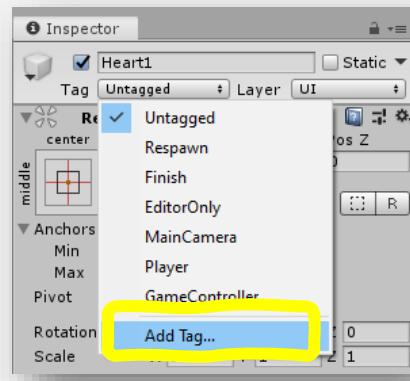
**26** Search for and select the image named "healthHeart".



**27** Close the Select Sprite dialog.

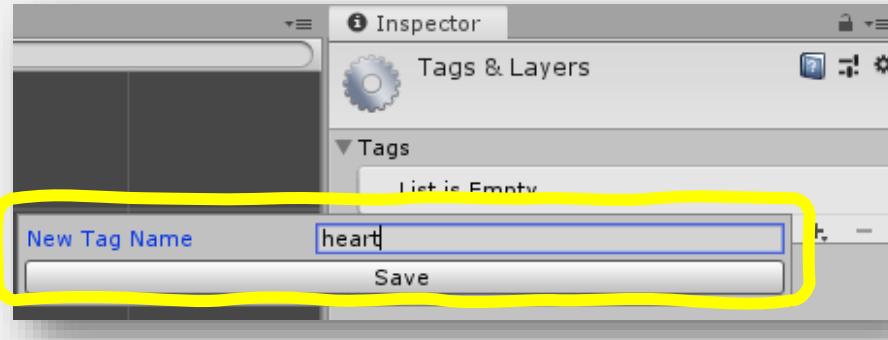
**28** Since we are using these images to track our player's health, we want to be able to easily find them. Unity's tagging system is the perfect solution.

**29** In the inspector, under the name of the game object is a dropdown to change the object's Tag. We want to create our own, so click on the "Add Tag..." option.

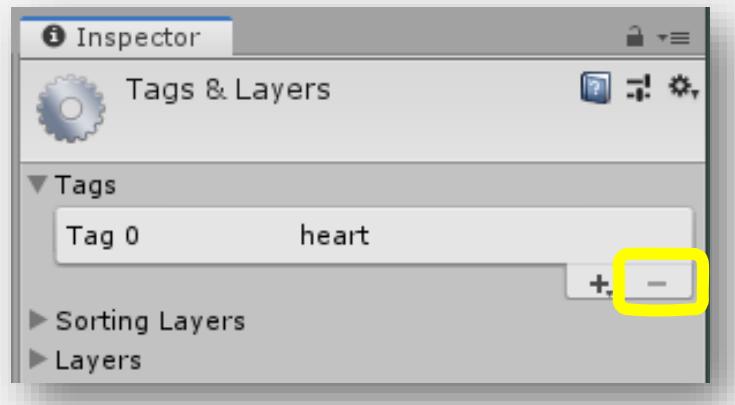


**30** This changes the Inspector to the Tags & Layers options.

**31** Click the plus sign and type “heart” into the box that pops up. Click the Save button.

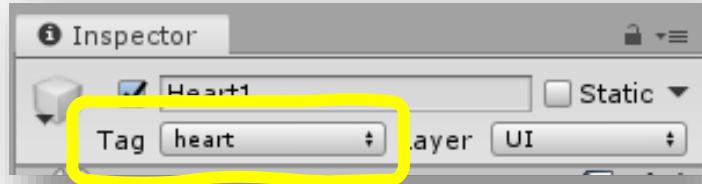


**32** You should now see your new tag in the Tags table. If you did something wrong, you could click on the tag name and then the “-” button to delete it and try again.

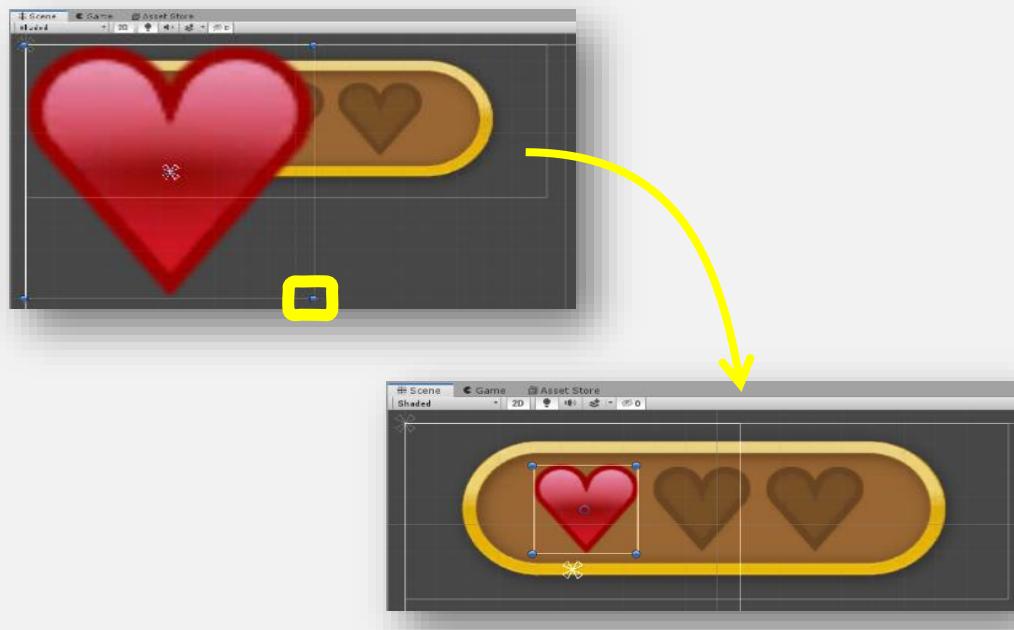


**33** Click on the Heart1 game object in the Hierarchy. We now need to set Heart1's tag to the new “heart” tag we just created.

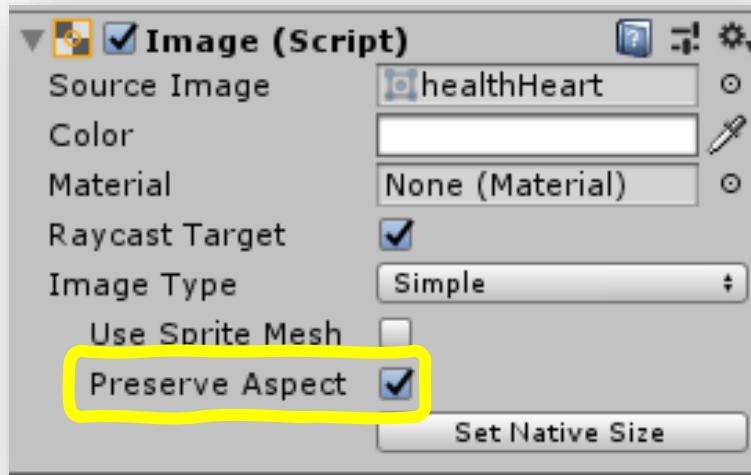
Click on the Tag dropdown and select “heart” from the list.



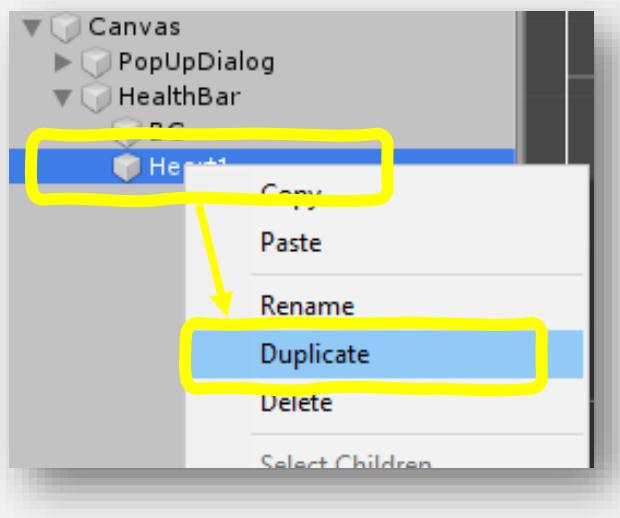
**34** In the scene, click on the red heart. Using the blue circles in the corners, change the size and align it with the first dark heart slot.



**35** You can enable Preserve Aspect in the Heart1's Image (Script) component to help you get the proportions correct.



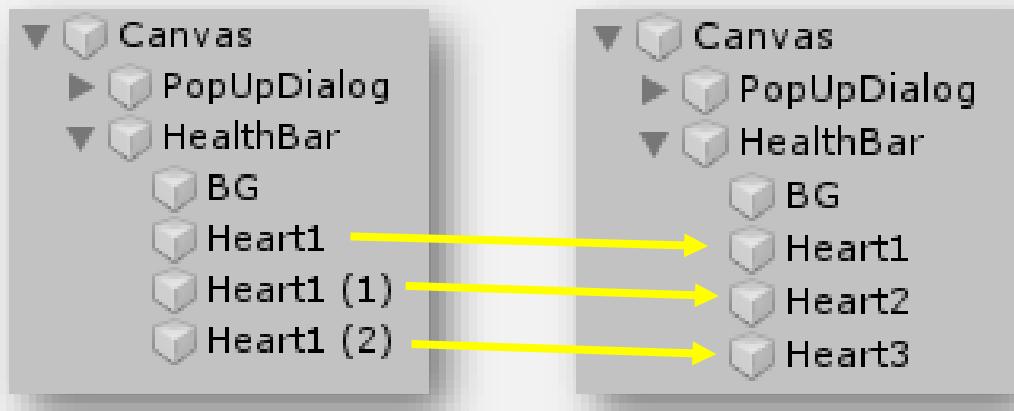
**36** We need to place two more hearts. Instead of repeating the process of adding an Image, setting a source image, tagging, and resizing, we can just duplicate Heart1!



**37** In the Hierarchy, right click on the Heart1 game object and click Duplicate.

**38** Right click Heart1 a second time and click Duplicate again.

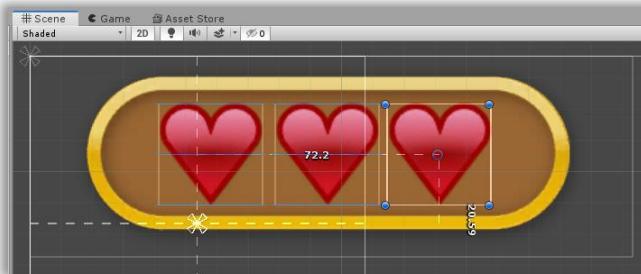
**39** You should now have Heart1, Heart1 (1), and Heart1 (2).



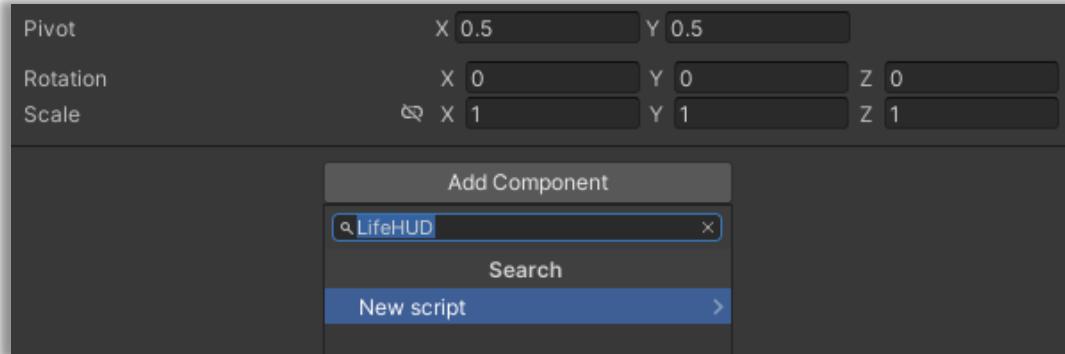
Rename these new game objects from "Heart1 (1)" and "Heart1 (2)" to "Heart2" and "Heart3".

- 40** If you look in the scene, it doesn't look like anything changed! There are three hearts in the scene, but it looks like just one because we duplicated Heart1. This means that all three hearts have the same exact x and y positions.
- 41** Click on Heart2 in the Hierarchy. Click inside the highlighted box in the scene and drag it to the right and place it in the second slot of the background.
- 42** Click on Heart3 in the Hierarchy. Click inside the highlighted box in the scene and drag it to the right and place it in the third slot of the background.
- 43** You might notice that Unity helped you place the hearts in the right place! Unity was able to understand that you wanted to keep the images in a straight line.

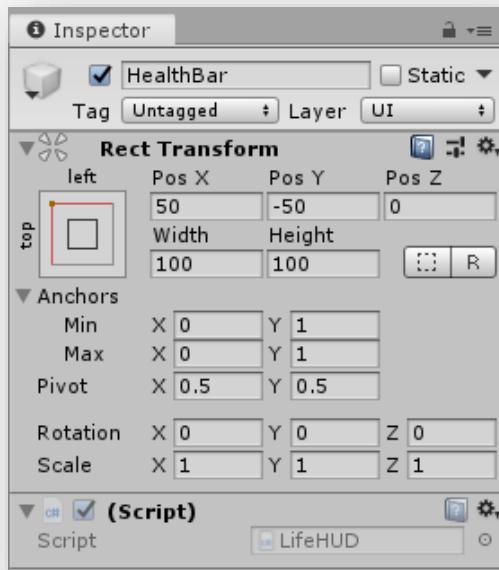
We now have the UI for our health system all set up! Next, we need to write a script to program it!



- 44** Select the HealthBar object in the Hierarchy to open it in the Inspector. Below the Rect Transform, there should be an Add Component button. Search for "LifeHUD" and click the "New script" button.



- 45** Double click on LifeHUD in the (Script) component. This will open up the code file in Visual Studio.



- 46** We need to set up 3 variables at the start of the class: one that points to the hearts in our HealthBar, one that points to the GameManager, and one that states how many lives the player has.

These three variables will go inbetween public class `LifeHUD : MonoBehaviour {` and `void Start()`.

```
public class LifeHUD : MonoBehaviour
{
 // Start is called before the first frame
 void Start()
```

**47** The first line to add is

```
private GameObject[] hearts;
```

This will create a private variable named hearts.

This means that no script or game object other than LifeHUD will even know that hearts exist!

The GameObject[] in the middle tells Unity that we want hearts to be an array of GameObjects.

```
public class LifeHUD : MonoBehaviour
{
 private GameObject[] hearts;

 // Start is called before the first frame update
 void Start()
}
```

*Remember that an array is just a way to store a group of similar objects together.*

**48** The second line to add is

```
private int lives = 3;
```

This will create a private variable named lives.

```
public class LifeHUD : MonoBehaviour
{
 private GameObject[] hearts;
 private int lives = 3;

 // Start is called before the first frame update
 void Start()
```

*Remember that this means that no script or game object other than LifeHUD will know that the lives variable exists!*

The int is telling Unity that we want lives to be an integer (a fancy way of saying a whole number). Finally, we set the value of lives to equal 3. This means that every time we start the game, Codey will have exactly 3 lives.

**49** The final line to add is

```
public GameObject background;
```

This will create a public variable that is a GameObject named background. By making this variable public, we can use the Inspector to connect the variable to an existing game object in our scene.

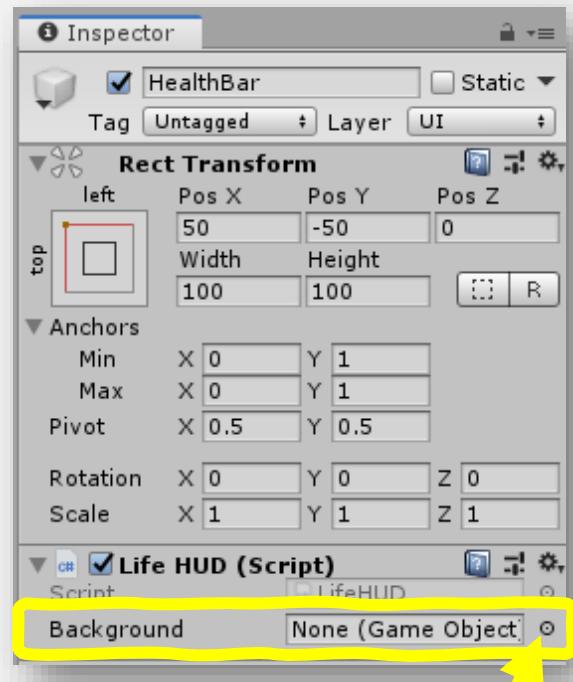
```
public class LifeHUD : MonoBehaviour
{
 private GameObject[] hearts;
 private int lives = 3;
 public GameObject background;

 // Start is called before the first frame update
 void Start()
```

**50** The game object that we need to connect to our LifeHUD script is simply called “background”.

**51** Go back to Unity and click on the HealthBar game object in the Hierarchy.

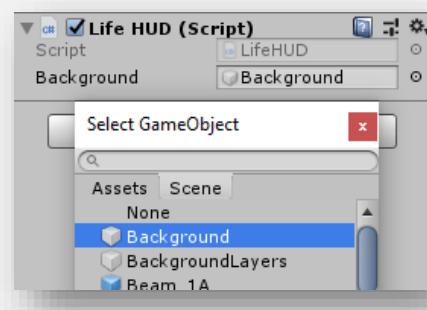
**52** Now, in the Inspector look at the Life HUD (Script) component. There should be a new property called Background.



Click the little circle to open the Game Object selector window.

**53** This window will display all the GameObjects in the scene. We want to find the one named “Background”. Click on it and close the selector window.

Our LifeHUD script is now connected to our Background game object. We will use this to tell the game to show the game over screen.



**54** We need to add a new function that will run our code when the entire game starts. The Start function will run when the object first loads into the scene.

**55** Once the everything in the game is loaded and the HUD is ready to start, we want to find all of the hearts we set up in an earlier step with the line `private GameObject[] hearts;`.

We do this by asking the HealthBar GameObject to find all of the objects that have the “heart” tag. We set that up in an earlier step!

**56** In the body of the Start function, type

`hearts = GameObject.FindGameObjectsWithTag("heart");`

This will set the hearts variable defined earlier in the code to be equal to all of the GameObjects tagged with “heart” that Unity can find.

Because we set up the tags, we know that the value of hearts will be an array of the three red hearts on our HealthBar! This function gets the hearts in the order that they are in the scene from top to bottom, so make sure your hearts are in the right order!

```
public void Start()
{
 hearts = GameObject.FindGameObjectsWithTag("heart");
}
```

**57** If your game won’t start, look in the console for an error. Do you see something like “Cannot implicitly convert type” next to a red stop sign? This message means that it is trying to assign the wrong type to a variable.

If yes, then you are probably using `FindGameObjectWithTag` instead, which only returns one game object and not an array. Go back to the line of code we just typed and make sure the function you are using is `FindGameObjectsWithTag` and not `FindGameObjectWithTag`.

We know we have more than one heart, so we need to find GameObjects, not just one GameObject.

**58**

Now that we have all of the pieces set up, we need to build out a function for the LifeHUD.

The function we need to make is one that runs whenever the player gets hurt.

While the functions can go in any order, we will place this new function after Start and before Update.

In between those two functions type

```
public void HurtPlayer() {
}
```

Making sure to leave a blank line between the brackets. We decided to name this function “HurtPlayer” because it describes what the function does.

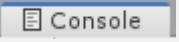
This function will be responsible for subtracting lives from the player, removing hearts from the HUD, and triggering the Game Over screen.

We want this function to be public because it needs to be called, or run, outside of this script, and the void indicates that it does not return a value.

**59**

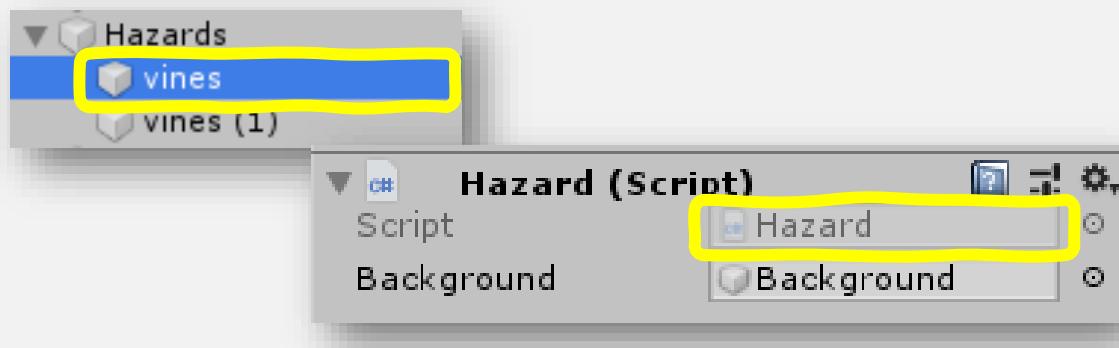
Before we code those three functions, add `Debug.Log("Ouch!");` to let us know when the function is run.

```
public void HurtPlayer()
{
 Debug.Log("Ouch!");
}
```

- 60** In Unity, open the Console tab  and play the game. Try to hurt your Ninja by running into the vines.

The console is empty even after running into the vine a hundred times! This is because we never run the HurtPlayer function!

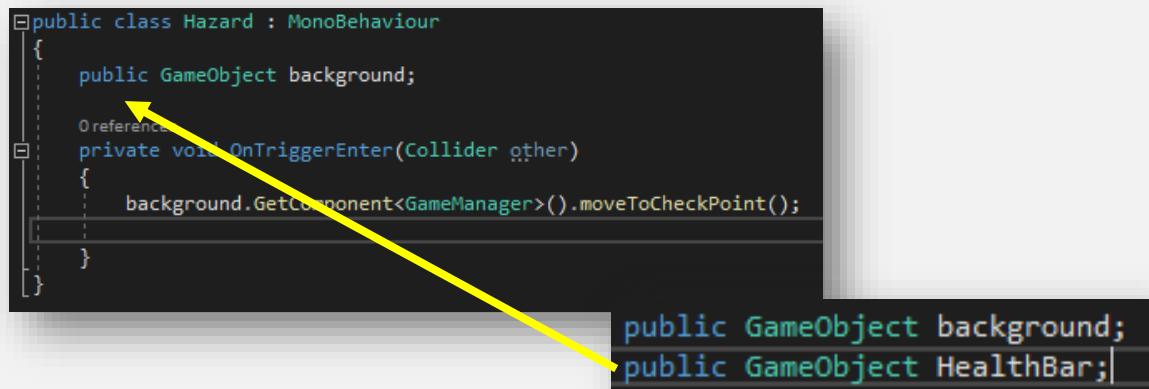
- 61** What game object should hurt the player? The vines! Find the vines in the Hierarchy and look for the Hazard script in the Inspector and open it.



- 62** Right now, this script uses the background to move the player to a checkpoint whenever the player touches a vine.

Before we can run the PlayerHurt function, we must get a reference to where the PlayerHurt function is defined- the HealthBar object. After `public GameObject background;` type

```
public GameObject HealthBar;.
```



```
public class Hazard : MonoBehaviour
{
 public GameObject background;
 private void OnTriggerEnter(Collider other)
 {
 background.GetComponent<GameManager>().moveToCheckPoint();
 }
}
```

The screenshot shows the Unity code editor with the `Hazard` script. The `background` field is highlighted with a yellow box. A yellow arrow points from this field in the code to its declaration in the Inspector below. Another yellow arrow points from the newly added `HealthBar` field in the code to its declaration in the Inspector.

- 63** Save the script and go back to Unity. In the Inspector, you should see a new Health Bar property on Hazard (Script). Click the little white circle  to open the GameObject selector, search for “HealthBar”, select the HealthBar result, and close the Selector window.



- 64** Since we have two vines in the scene, we need to repeat the process. Select “vines (1)” to open this game object in the inspector.

Click the little white circle  to open the GameObject selector, search for “HealthBar”, select the HealthBar result, and close the Selector window.

- 65** Now that we have the HealthBar game object connected to the Hazard script, we can call our HurtPlayer function.

Since we want to call this only when the player touches vines, we need to write our line of code inside of the OnTriggerEnter function.

```
private void OnTriggerEnter(Collider other)
{
 background.GetComponent<GameManager>().moveToCheckPoint();
}
```

Remember that OnTriggerEnter is a special function that will execute code whenever the Ninja collides with the vines because the vines’ collider has the IsTrigger property checked.

**66** On the line after

```
background.GetComponent<GameManager>().moveToCheckPoint();
type
HealthBar.GetComponent<LifeHUD>().HurtPlayer();.
```

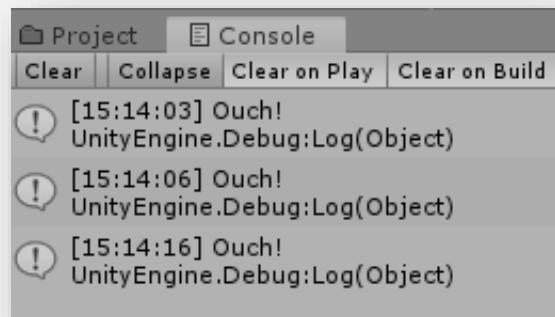
Since the HurtPlayer function lives inside the LifeHUD script which is attached to the HealthBar game object, we have to chain a few commands together.

First we ask the HealthBar variable for the LifeHUD script component by using GetComponent<LifeHUD>(). We then need to ask that script to run the HurtPlayer function with HurtPlayer().

```
private void OnTriggerEnter(Collider other)
{
 background.GetComponent<GameManager>().moveToCheckPoint();
 HealthBar.GetComponent<LifeHUD>().HurtPlayer();
}
```

**67** Play the game and run into the vines. Does something show up in the console now? Make sure to check both sets of vines.

You should see the “Ouch!” message appear after the Ninja runs into the vines. This means that the vines are connected to the HurtPlayer function and we can continue adding functionality!



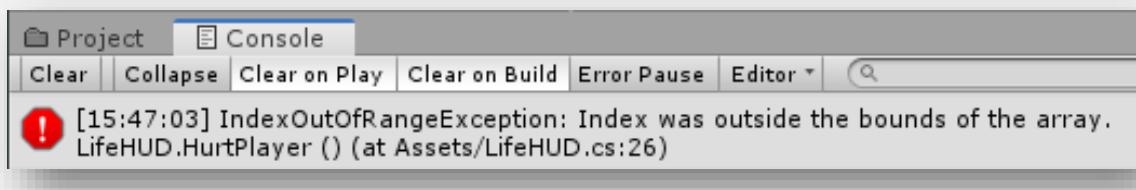
- 68** Open up the LifeHUD script. Inside the HurtPlayer function, after Debug.Log("Ouch!"); add `lives -= 1;` to subtract a life each time the player gets hurt. Remember that the -= operator reduces the value of the variable by one.

```
public void HurtPlayer()
{
 Debug.Log("Ouch!");
 lives -= 1;
}
```

- 69** Now that we are tracking when we lose lives, we need to update the hearts in the HealthBar user interface. After each time we remove a life, we need to deactivate the appropriate heart in the UI. We do this by using the hearts array. On the line after `lives -= 1` type `hearts[lives].SetActive(false);` to disable the appropriate heart in the UI.

```
public void HurtPlayer()
{
 Debug.Log("Ouch!");
 lives -= 1;
 hearts[lives].SetActive(false);
}
```

- 70**



Play the game and run the Ninja into the vines. It works! What happens when you run into the vines when you are out of hearts? The game should stop and show a game over screen, but that doesn't happen. Instead the Ninja gets sent to the checkpoint and we get an error in the console since the number of lives is less than 0! That's okay, we can fix it!

- 71** We need to catch when the player has run out of lives. On the line after hearts[lives].SetActive(false); type

```
if (lives == 0) {
}
}
```

Make sure to put a blank line inbetween the brackets.

```
public void HurtPlayer()
{
 Debug.Log("Ouch!");
 lives -= 1;
 hearts[lives].SetActive(false);
 if (lives == 0)
 {
 |
 }
}
```

- 72** The game over screen should be shown only if the player has exactly 0 lives left, so inside the if statement type

```
background.GetComponent<GameManager>().GameOver(); .
```

```
if (lives == 0)
{
 background.GetComponent<GameManager>().GameOver();
}
```

Since the GameOver function lives inside the GameManager script which is attached to the background game object, we have to chain a few commands together.

First we ask the background variable for the GameManager script component by using GetComponent<GameManager>().

We then need to ask that script to run the GameOver function with GameOver().

- 73** Play the game and run the Ninja into the vines until there are no more lives left. The Game Over screen pops up!

# Prove Yourself

## Get Started

- Create a heal function on LifeHUD.
- Add a reference to the healthbar on gem script (pickupkey)
- In PickUpKey, add `HealthBar.GetComponent<LifeHUD>().HealPlayer();` to `OnTriggerEnter`

```
public void HealPlayer()
{
 Debug.Log("Yay!");
 if (lives < 3)
 {
 hearts[lives].SetActive(true);
 lives += 1;
 }
}
```

Many games give the player an opportunity to heal themselves. In this game, the Ninja must collect gems around the world to unlock the secret teleporter to the next level. What if picking up a gem gave the Ninja back a missing heart?

## Task

In this Prove Yourself Activity, you must

1. Create a HealPlayer function in the LifeHUD script that activates a heart and adds a life if the Ninja is not at full health already.
2. Find the script that runs code when the Ninja touches a gem
3. Add the HealthBar game object to the script from step 2
4. Call the HealPlayer function when the Ninja collides with the gem

As you complete this challenge, ask yourself at each step “Have I done something like this in this game already?”

**Step 1** is just like creating the HurtPlayer function in reverse.

**Step 2** is just like finding the script on the vines.

**Step 3** is just like adding the HealthBar object to the Hazard script.

**Step 4** is just like calling the HurtPlayer function in the Hazard script’s OnTriggerEnter function.

If you are getting stuck on the HealPlayer function, remember that it needs to do the opposite of the HurtPlayer function. Instead of subtracting a life, setting the heart to be inactive, and checking if there are no lives, the HealPlayer function should check if the player is missing a life, set the heart to be active, and add a life.

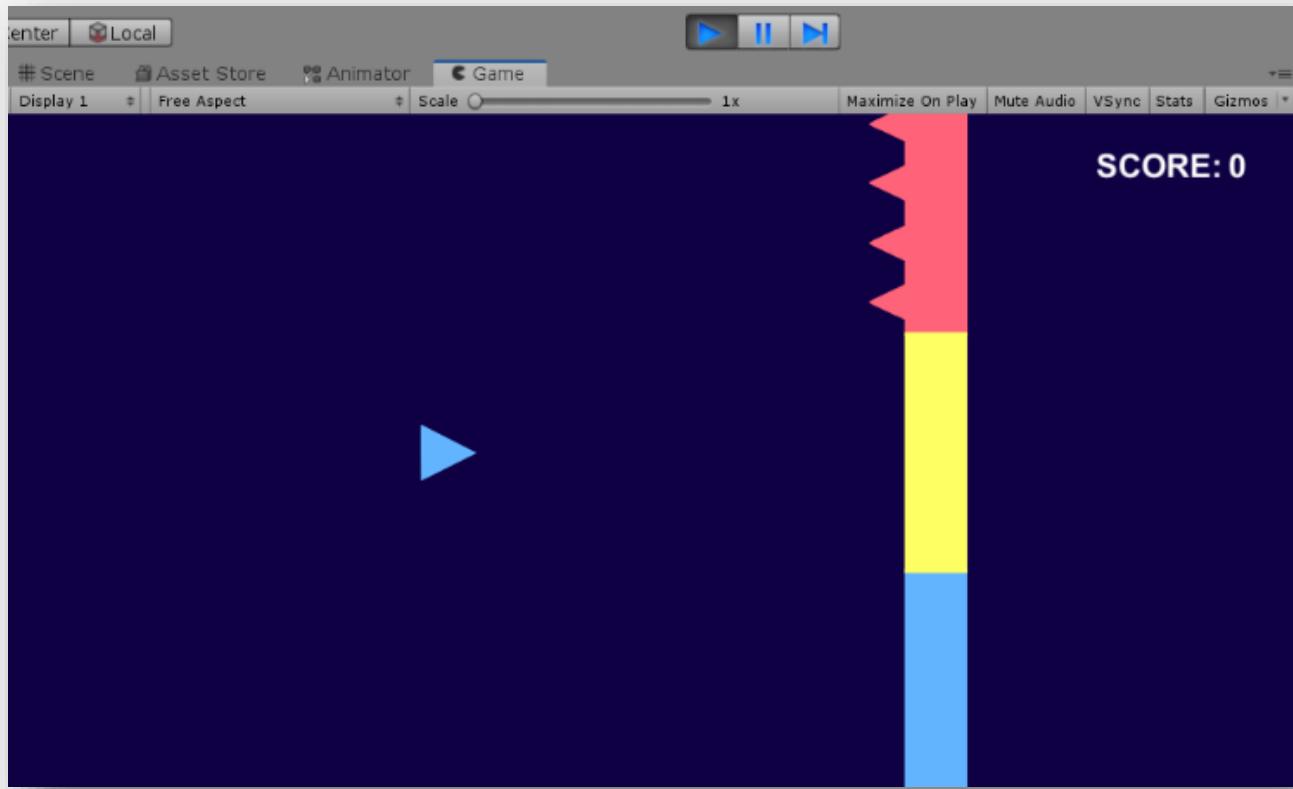
For an extra challenge, let the Ninja gain more than 3 lives. What do you need to do in the code to let this happen? Do you need to add more heart game objects? What impact does this have on the HealthBar UI in the game? Does our existing HealthBar UI support more than 3 hearts? What solutions can you come up with?

## Activity 12

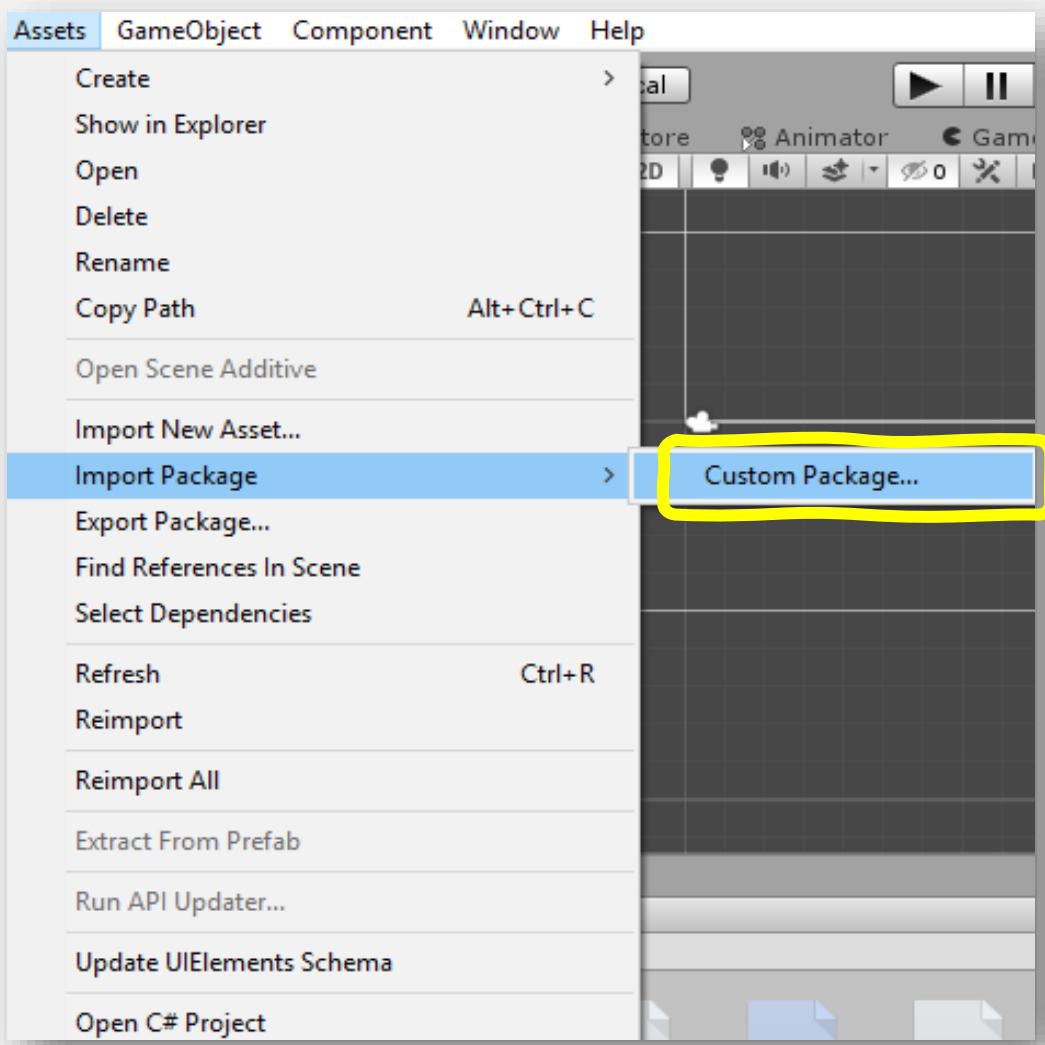
# World of Color

You will create game over mechanics by simulating player death using `gameObject.SetActive`. Then, a delayed level restart will be made with `Invoke` and `SceneManager`. Particle systems will be reviewed as you use critical thinking to implement their functions for a new situation.

World of Color...or is it shape? Stay alert because the rules change just as much as the player! If the player is a square, match the player color to the obstacle. If the player is a triangle, match the player shape to the obstacle.

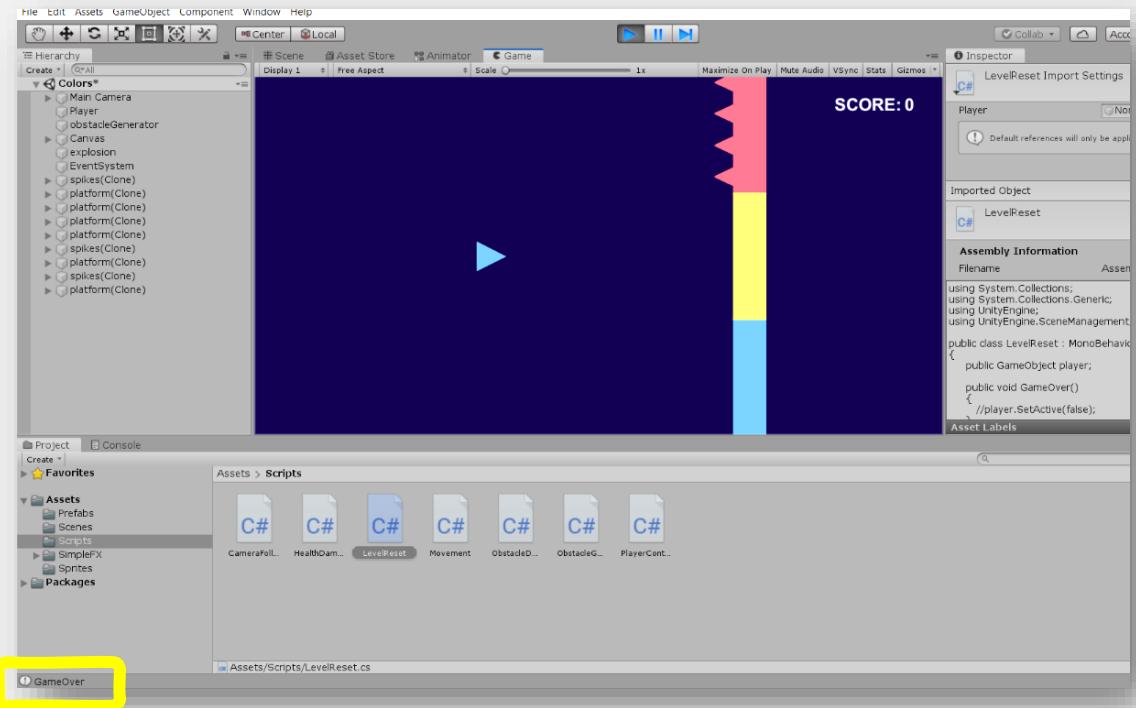


- 1** Start a new Unity Project and name it *YOUR INITIALS - World of Color*.  
Select **2D core**.
- 2** We've created a starter pack to give you a head start! To use it, import the *WorldofColor.unitypackage* by going to **Assets > Import Package > Custom Package > All > Import**.

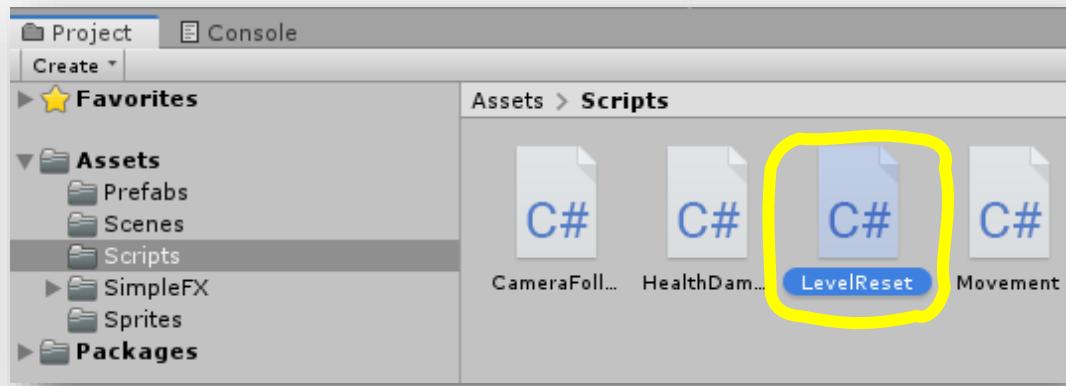


**3** Press play to test. There's no game over! The **console** tells you when game over should happen, but there's no player death or level reset.

Without a clear challenge and goal, the game is confusing and not fun, so let's set up a game over.



**4** In the **Project** window, under the **scripts** folder, open the **LevelReset** script.



- 5** Here is what your LevelReset script looks like right now. There is already a new function included called `public void GameOver()`.

```
public class LevelReset : MonoBehaviour
{
 public void GameOver()
 {
 }
}
```

### Public Void

*There are two references to `public void GameOver()`. Like variables, functions can be public. Instead of being made visible in the inspector, public voids are made visible to **other scripts**. Let's see how this is used:*

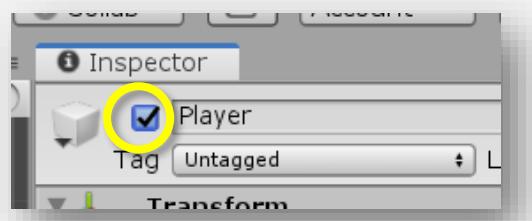
```
if(!other.CompareTag("obstacle")){
 if(other.tag != gameObject.tag){
 LevelReset.GameOver();
```

- 6** What should happen for game over when the player hits the wrong obstacle? There must be a clear visual indicator that the player did something wrong and can no longer play. Let's make the player disappear using `gameObject.SetActive()`.

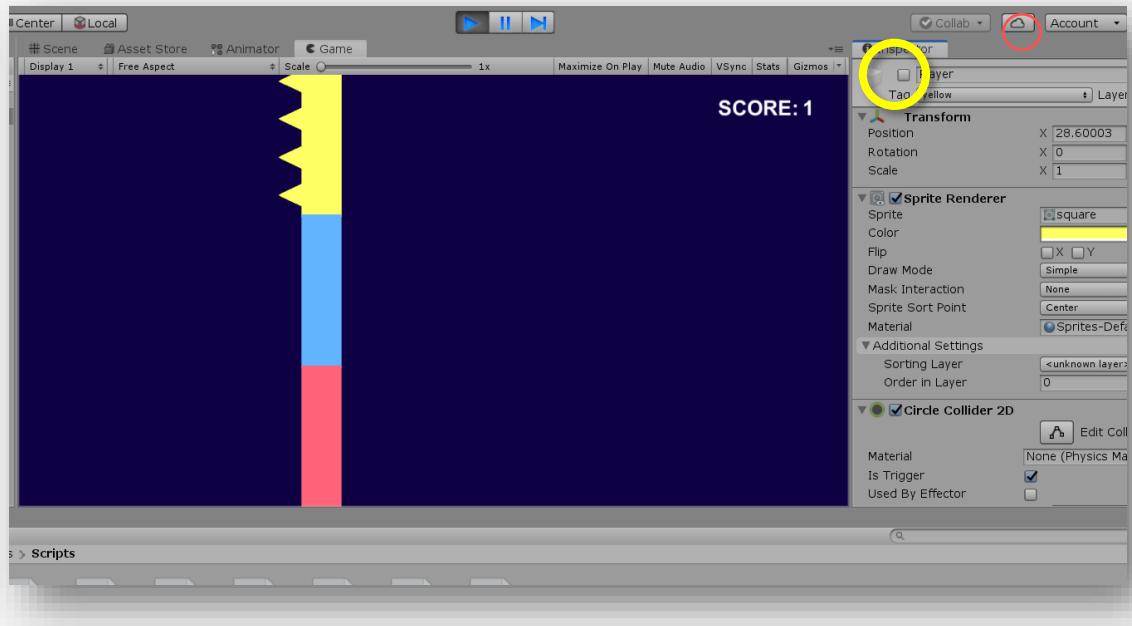
- 7** In `public void GameOver()`, set player active **false**. Since the script is on the player, we can use `gameObject` to get a reference to the player.

```
2 REFERENCES
public void GameOver()
{
 gameObject.SetActive(false);
}
```

- 8** Save and close the script. In the Unity window, select the **Player** and make sure the **Inspector** is scrolled to the top. Notice the box next to the game object name.



Keep an eye on that and press play! When you hit the wrong obstacle, what happens to the player?

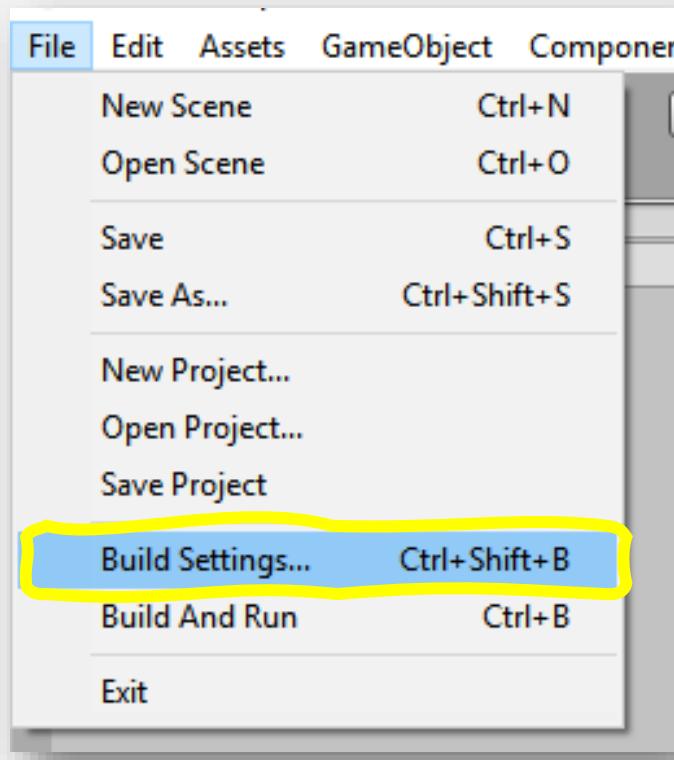


In the **Inspector**, the box becomes unchecked. This turns off everything attached to the player, including the sprite renderer (which makes the player visible), colliders, and all scripts.

- 9** Great job! Game over is now clear, but what if we want the game to automatically restart after player death? Everything related to restarting a level or changing between levels is handled with Unity's **Scene Management**.

*Remember that scenes represent the levels.*

**10** Let's get our scenes setup before we start coding. Click on the **File** tab, then **Build Settings**.



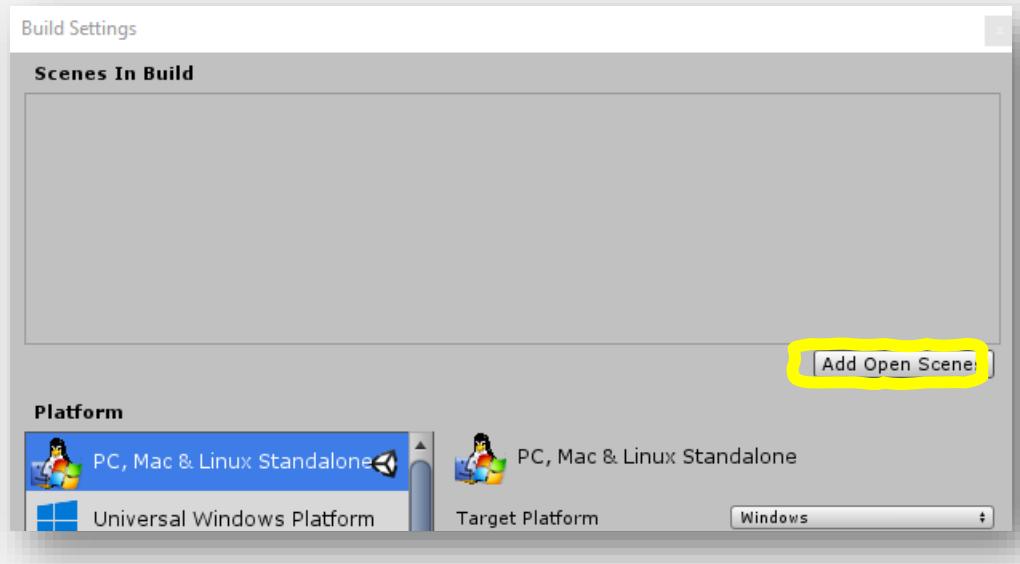
### ***Building***

*Building a game is the final step. It creates an independent executable app, meaning it can be played and shared outside of Unity. The Build Settings define the settings for the game build, such as the scenes to include and the platform for your game- Windows, Mac, Linux, etc.*

**11** A new window will open. Focus on the top part of the window, titled **Scenes in Build**.

This tells Unity which scenes to include in the game build. Right now, the list is empty. If there are any scenes in the build, select them and press the delete key to clear them out.

To add the current scene, click on the **Add Open Scenes** button.



**12** The name of the scene- Colors- will appear.

Notice on the right, there is the number 0.

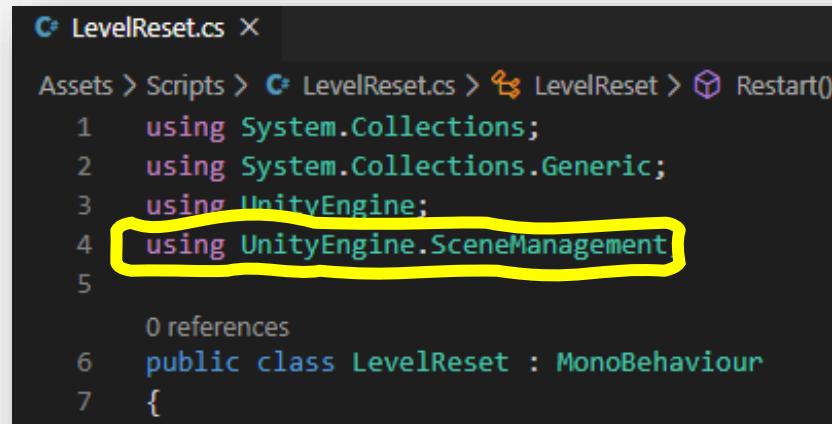
This is the number of your scene.



This is the only scene in our game, but if we add another scene, can you guess what its number would be?

That's right: 1! Keep in mind, whatever scene is labeled 0 is the scene that is loaded when the game starts once built.

- 13** With the scene set up in the build settings, we can now use it in our code! Open the **LevelReset** script. Add the Unity namespace **SceneManagement** to the top of your code. This stores functions related to handling scenes.

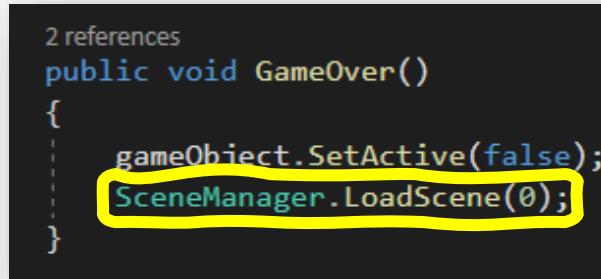


```
C# LevelReset.cs X

Assets > Scripts > C# LevelReset.cs > LevelReset > Restart()

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.SceneManagement
5
6 0 references
7 public class LevelReset : MonoBehaviour
8 {
```

- 14** To load or reload a scene, we will use **SceneManager.LoadScene()**. This looks at the build settings and reloads the scene with the number you give it. Remember, Unity starts counting scenes at 0 not 1. The Colors scene is scene 0, so we want the line to read:



```
2 references
public void GameOver()
{
 gameObject.SetActive(false);
 SceneManager.LoadScene(0);
}
```

- 15** Save your script and press play.

- 16** You did it, your level restarts on player death! But it restarts exactly on player death. This seems too fast for the person playing to realize it was game over and to get ready for the restart. **Invoke()** will solve this.

Invoke takes two parameters: ("functionName", time). It waits the set time before calling the function.

- 17** We need a new function to place the reload in. First delete the previously included `SceneManager.LoadScene()`. After `public void GameOver()`, add a function called `Reload()`:

```
2 references
public void GameOver()
{
 gameObject.SetActive(false);
}

0 references
void ... Reload()
{
}
```

- 18** In void `Reload()`, add `SceneManager.LoadScene(0)`:

```
void Reload()
{
 SceneManager.LoadScene(0);
}
```

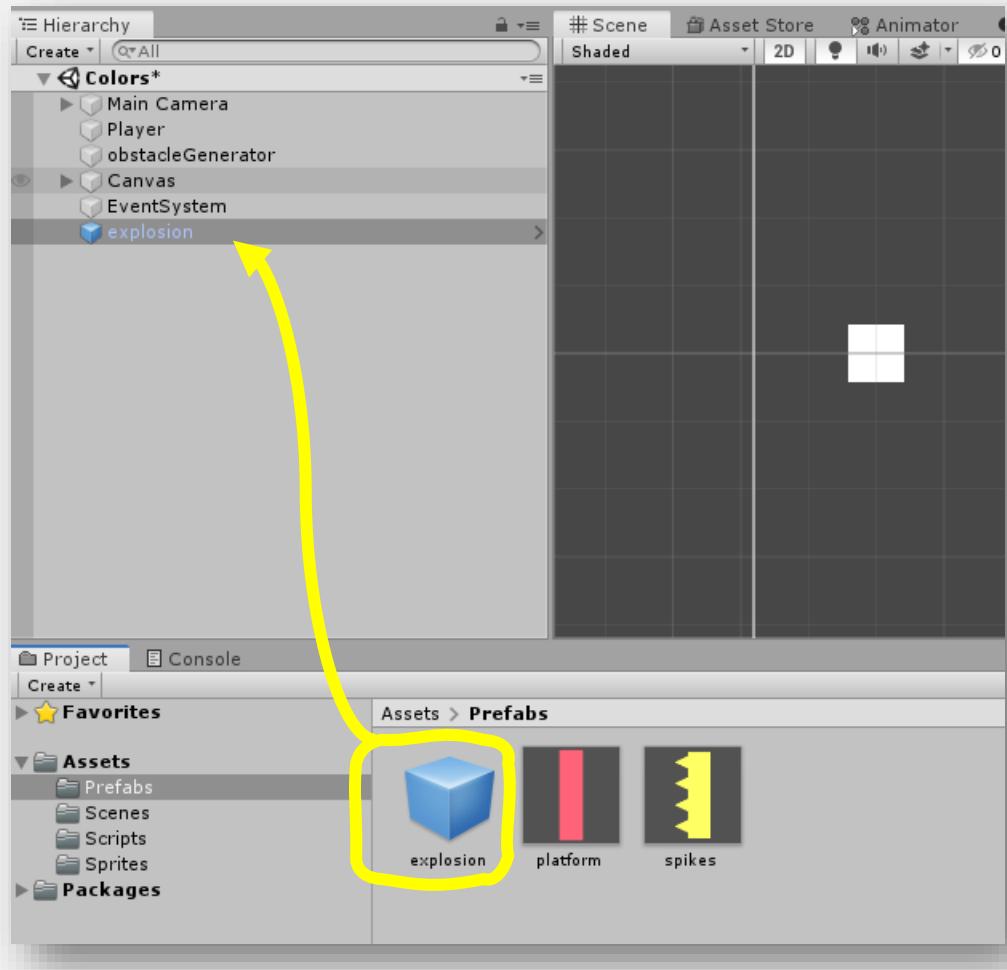
- 19** Remember `public void GameOver()` is called when the player hits an obstacle. To start the invoke timer at player death, include it in `GameOver()`. We will have the Invoke wait two frames:

```
public void GameOver()
{
 gameObject.SetActive(false);
 Invoke("Reload", 2);
}

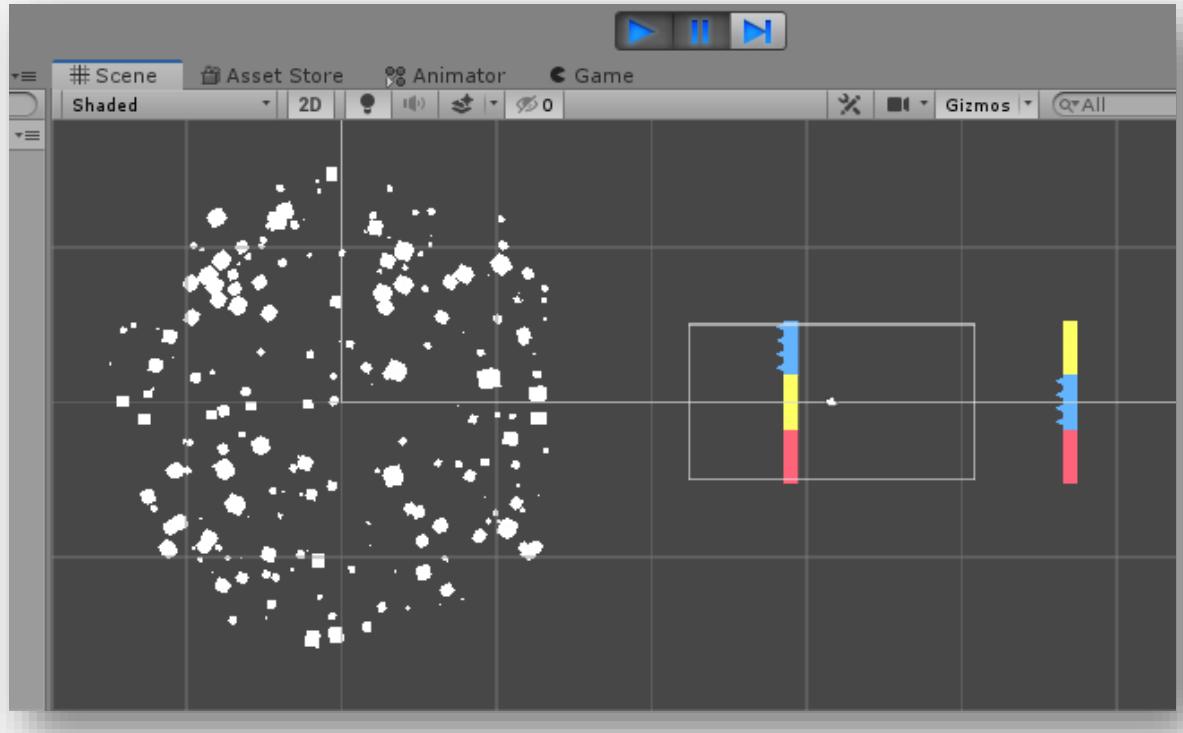
0 references
void Reload()
{
 SceneManager.LoadScene(0);
}
```

- 20** Press **play** and test out the game.

**21** Amazing job! But can we make it even better? Right now, the player just disappears when it hits an obstacle; what if we add a particle system? Particle systems are a great way to catch people's attention and adds that extra element of fun and exaggeration. In your **Project** window, under the **Prefab** folder, drag the **explosion** prefab into the **Hierarchy**.

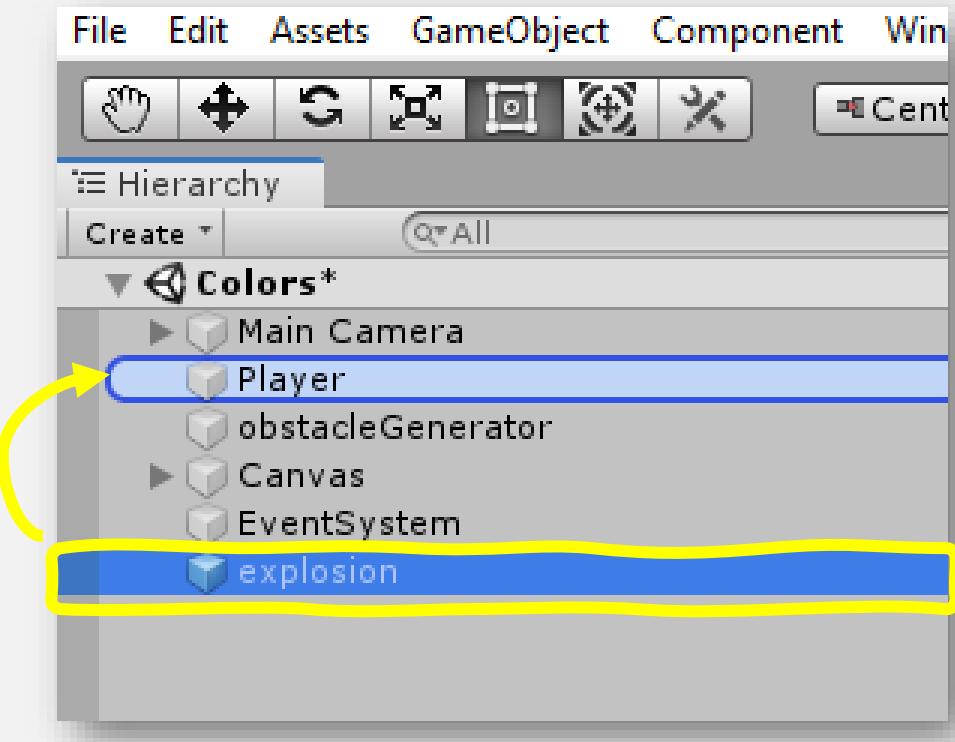


**22** Let's think, the explosion needs to play on player death, but the player is constantly moving forward. As is, our explosion will just stay in the same position. The picture below shows this:

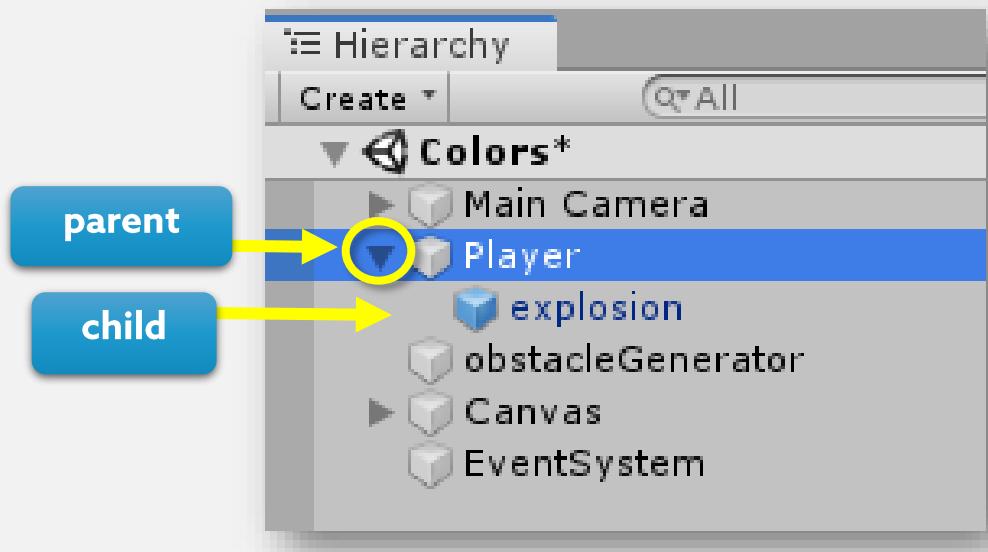


*How did we test this? When you press play, make sure you switch to the **Scene** tab. Then, scroll out so you can see the full scene. Wait until the player hits the wrong obstacle. If need be, you can press Pause to pause the game, and then the Skip to move forward one frame.*

**23** How do we get the explosion to follow the player? Remember, the **Hierarchy** is laid out like a family tree- with parent and child objects. Child objects follow the parent. Click and drag the **explosion** on top of **Player**, let go when there is a blue box around Player, like so:

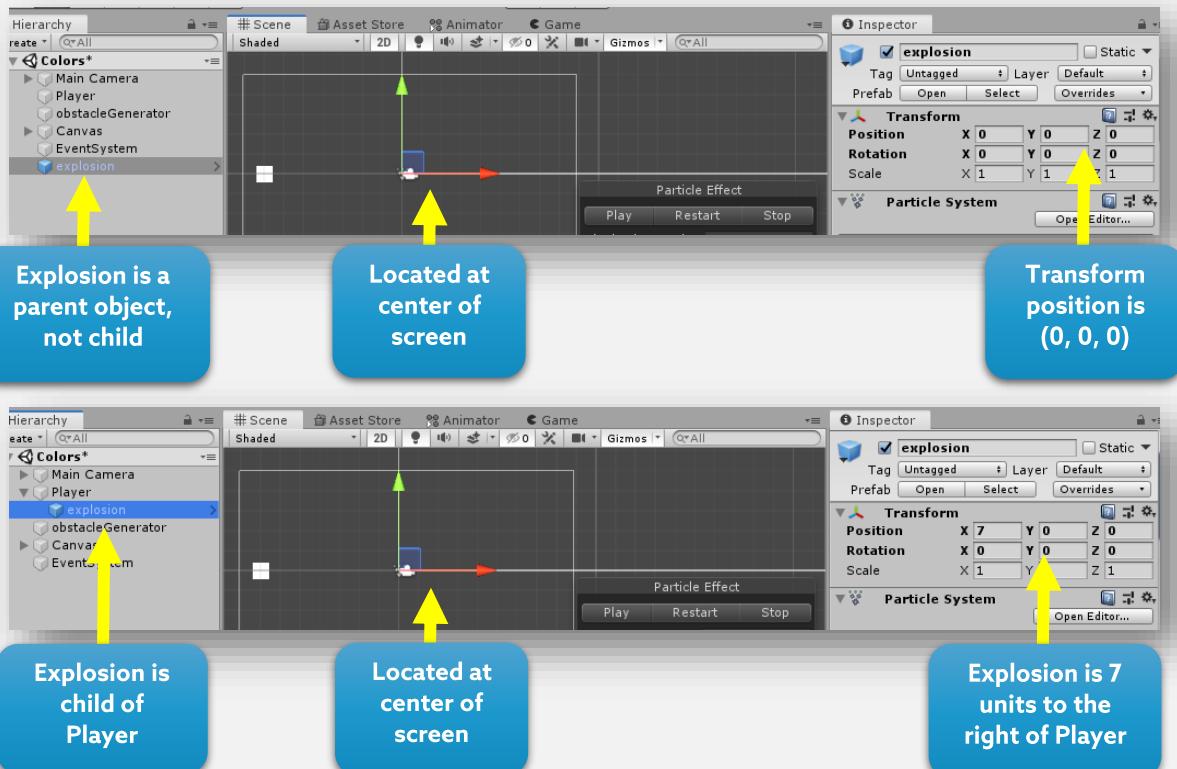


A grey triangle will appear to the left of **Player**. If you click on it, you will see Player's hierarchy, with the explosion as a child object.



**24** Now the explosion will move forward with the player. We still need to adjust one more thing! Click on explosion and look in the **Inspector** at its **Transform**. When you place an object in the scene, its transforms are based on the screen, with 0 being the center. When an object becomes a child, its transforms are based on the parent, so 0 becomes the center of the parent.

The pictures below illustrate how an object's transforms change when it becomes a child:



**25** We need the **explosion** to be at the center of **Player**, so make sure its **Transforms** are as follows:



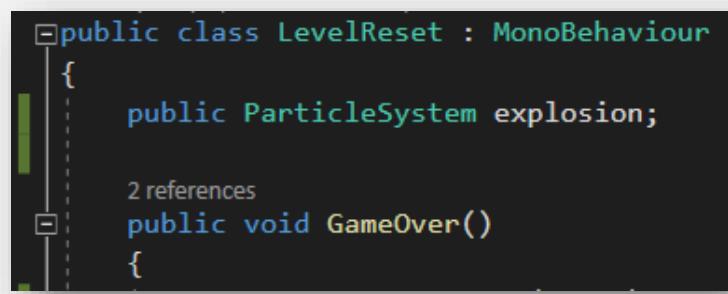
**26** Press play. The explosion is now at the center of Player, but still only plays once right at the start.

Remember `ParticleSystem.Stop()` and `.Play()`?

Let's use those to stop the explosion at Start and play it at Game Over.

Open your **LevelReset** script.

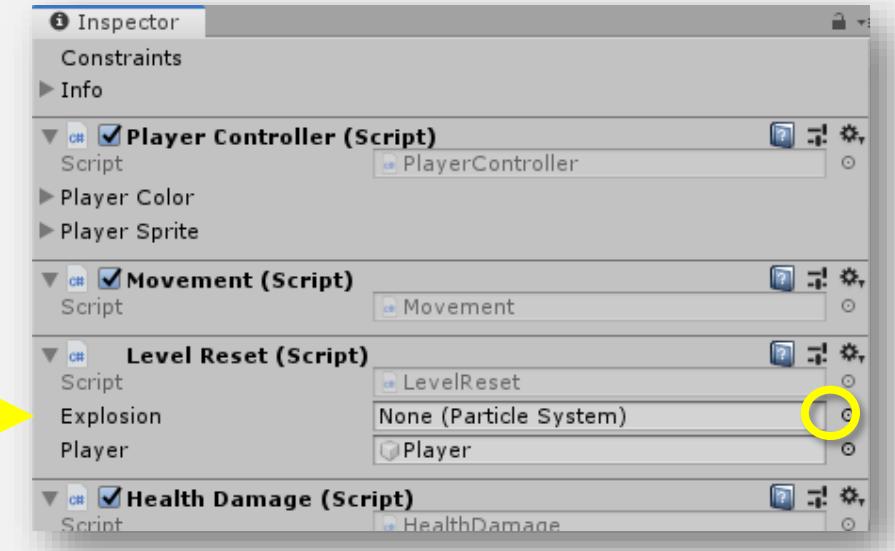
## 27 Declare the particle system explosion:



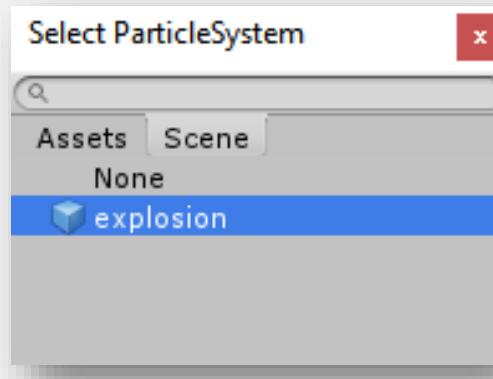
```
public class LevelReset : MonoBehaviour
{
 public ParticleSystem explosion;

 2 references
 public void GameOver()
 {
```

28 Save your script. Switch back to the Unity window. In the **Hierarchy**, click on **Player**. In the **Inspector**, scroll down to the **Level Reset** script component. Click on the circle next to the empty **Explosion** slot:



29 In the pop-up window, select **explosion**.



- 30** Open the **LevelReset** script. We did not want the particle to play at the start of the game. So which function do we need to add back into our script? **void Start()**:

```
Unity Message | 0 references
private void Start()
{
|
|
|}
}
```

- 31** In **void Start()**, stop the particle system from playing:

```
void Start()
{
|
|
|}
explosion.Stop();
```

- 32** Where should we put **explosion.Play()**? Think about when we want it to play. Add it to your code:

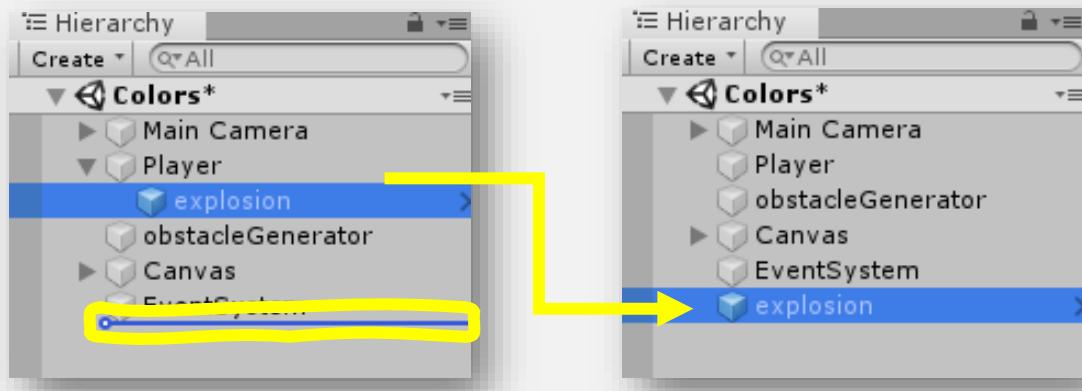
```
2 references
public void GameOver()
{
|
|
|}
gameObject.SetActive(false);
Invoke("Reload", 2);
explosion.Play();
}
```

- 33** Press play to test that the explosion only plays on player death.

**34** It's not playing the explosion! Our code makes sense, but it's not working. Brainstorm some reasons why. I'll give you a hint: [SetActive\(\)](#).

Explosion was made a child of Player so it would follow its transforms, which it does! But remember adding `player.SetActive(false)`? The child follows the parent: if the parent is set to false, so is the child. This means the particle system is set to play, but at the same time deactivated. To fix this, we must code the player follow.

**35** Unparent explosion from Player. To do this, click and drag on **explosion**. Do not release it on top of another object (if you do, hit ctrl and z on your keyboard to undo). Release when there is a blue line, like so:



**36** Open the **LevelReset** script. Add a line below the `Invoke` line. This will set the position of the explosion to the player before we play it.

```
2 references
public void GameOver()
{
 gameObject.SetActive(false);
 Invoke("Reload", 2);
 explosion.transform.position = transform.position;
 explosion.Play();
}
```

---

**37** Save your script. Press play and test out your game.

---

**38** Great problem solving, Ninja! Your game over code has made the game more fun and understandable! Test out your hard work. Can you keep up with the constantly changing World of Color?

---

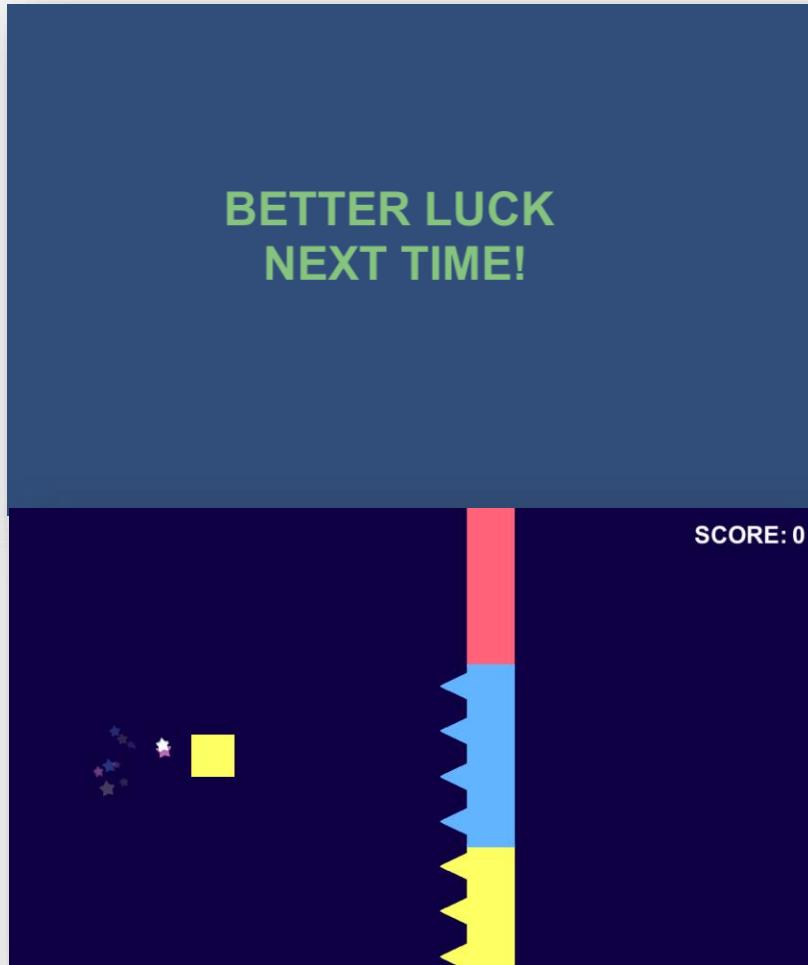
# Prove Yourself

## Get Started

- If not already open, open your Unity Project YOUR INITIALS – World of Color.
- Continue working in the Colors scene. All the assets you need are in the **Project** window under the **ProveYourself** folder.

## Task

Add the **trail particle system** to the player, so it plays when the player is alive and disappears at player death. Next, create a new scene called **Game Over** and place the **GameOverScreen** prefab in the scene. Instead of level reset, have the game wait briefly before switching to the Game Over scene at player death.



## Activity 13

# Amazing Ninja Worlds - Part 2

By the end of the game, you will be able to track and respond to player actions by creating a checkpoint and pickup system and changing game scenes.

When the player is hurt, they get sent back to a checkpoint which also happens to be where they started the game. A checkpoint is a great way to save the player's progress so that if they are hurt after a difficult portion of the game, they get sent back to after that part instead of having to go back all the way to the beginning.

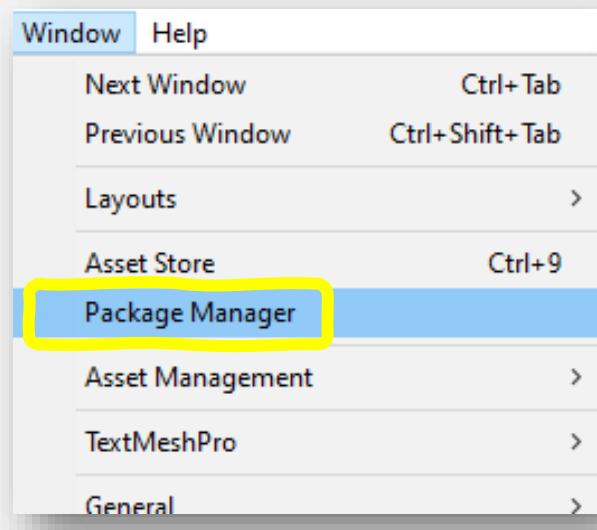
After the player gets past the vines near the beginning of the level, there's a second set of vines. If the player gets hurt there, they'll have to deal with both sets again and again. Let's change that.



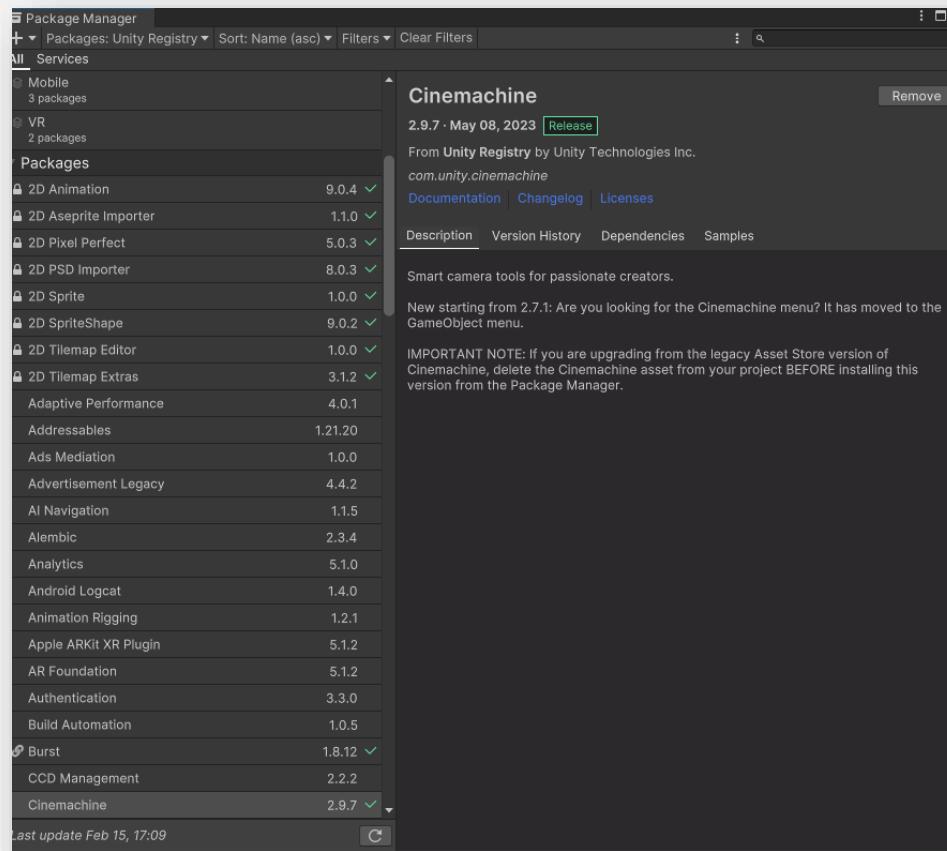
**1** Start a new Unity Project and name it **YOUR INITIALS -Amazing Ninja Worlds 2**. Select **3D core**.

**2** We will be using Cinemachine to control our camera. This will help us create awesome camera shots and angles for our game.  
Go ahead and open **Window > Package Manager**.

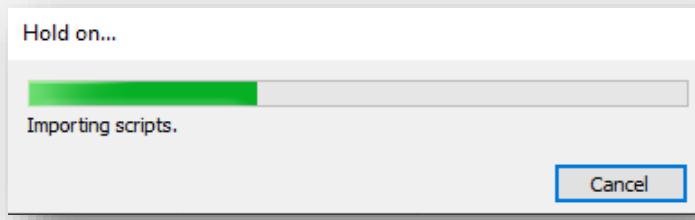
*Be patient if it doesn't open right away. It might take a minute to load.*



**3** Find **Cinemachine** in the Unity Registry and click **Install** in the top right of the window.

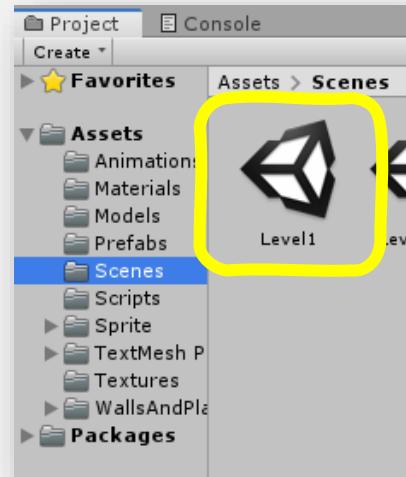


- 4** We've created a starter pack to give you a head start! To use it, import the **NinjaWorld2\_Starter.unitypackage** by going to **Assets > Import Package > Custom Package > All > Import**.



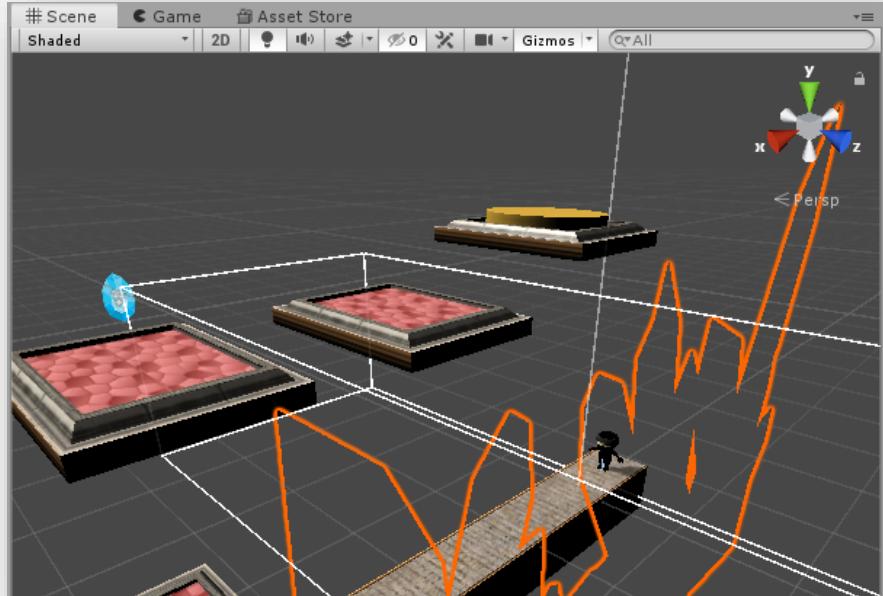
- 5** To open the starter package, double-click on the **Level1** scene.

You can find this in the **Project** tab under **Assets > Scenes > Level1**.



This will load the scene with all our game objects!

Make sure you can see all the objects and the assets in the scene tab.



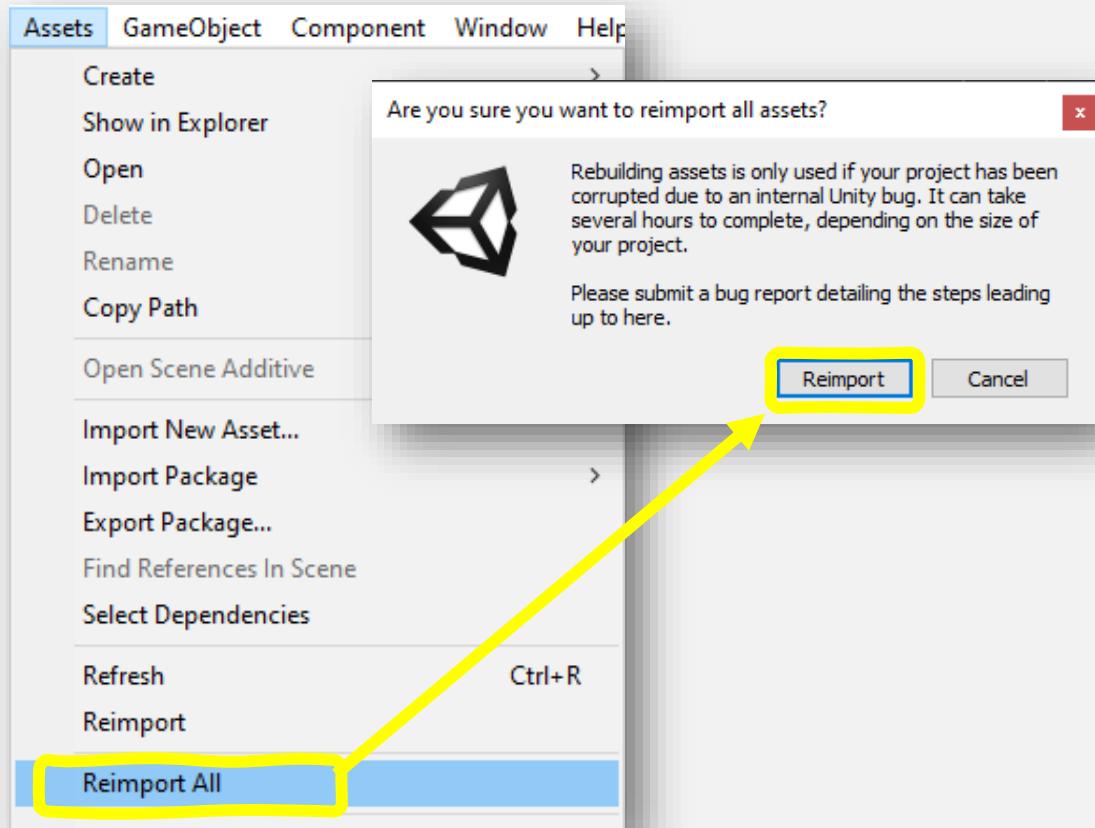
- 6** We should check that Cinemachine is working like it is meant to.  
In the **Hierarchy** tab, you should see a list of all the game objects in the scene.

If you see a little grey and red icon attached to the Main Camera (shown in the picture), Cinemachine is working!



If you do not see it, go into the **Assets** tab and press **Reimport All**.

Then after Unity is done importing, check the Main Camera again.



- 7** Make sure you click the 2D button on the bar on top of the Scene tab. This will align your camera properly.



- 8** In games, a checkpoint is a position in the world that the player is warped to if they fail a difficult challenge.

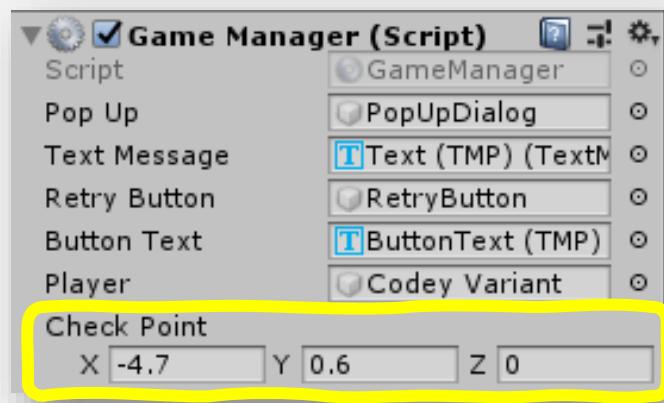
Have you ever played a difficult game where if you lost you got sent back to the start of the level? That can be very frustrating! Part of game design is always thinking about the player.

- 9** How could we program a checkpoint system? What data would we need to store? How could we move Codey to the checkpoint? Discuss some possible ideas with your Code Sensei before moving on.

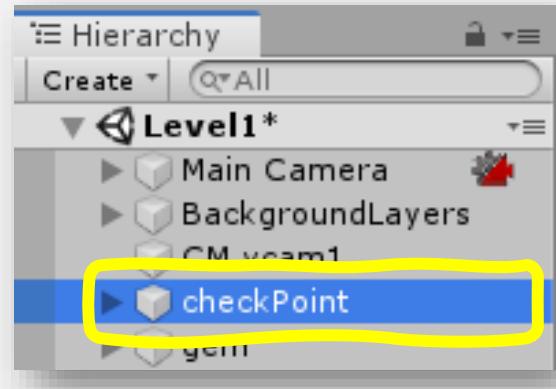
- 10** In the Hierarchy, find the Background game object that is located inside of the BackgroundLayers game object.



- 11** In the Inspector, find the Game Manager (Script) component. The Ninja World game is set up so this object keeps track of the player's current checkpoint. Right now, the checkpoint is initialized to be Codey's starting position.



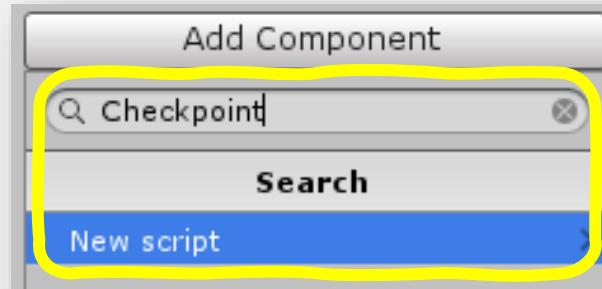
- 12** Now that we know how Ninja World stores checkpoints, we need to figure out how to update them! Find the existing checkPoint game object in the Hierarchy and click on it to open it in the Inspector.



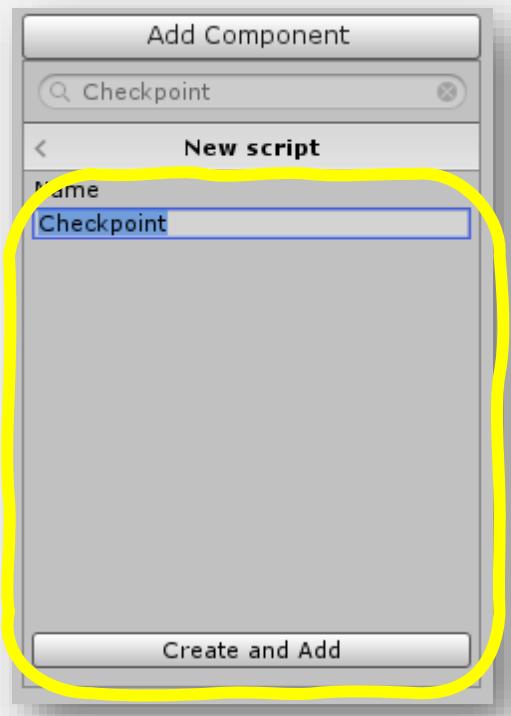
- 13** Click the Add Component button.

Add Component

- 14** Type "Checkpoint" and select "New Script".



- 15** Click "Create and Add" to create and attach a new script named "Checkpoint".



- 16** Double click on the Script Checkpoint box in the Inspector to open it in Visual Studio.



- 17** We do not need the Start or Update functions so delete them both.

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

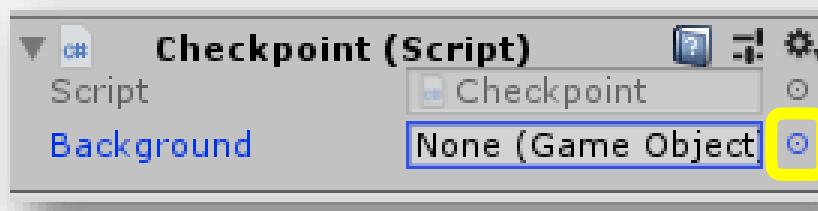
public class Checkpoint : MonoBehaviour
{
}
```

A screenshot of a Visual Studio code editor window. It displays a C# script named 'Checkpoint.cs'. The code is as follows:  
`using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class Checkpoint : MonoBehaviour  
{  
}`

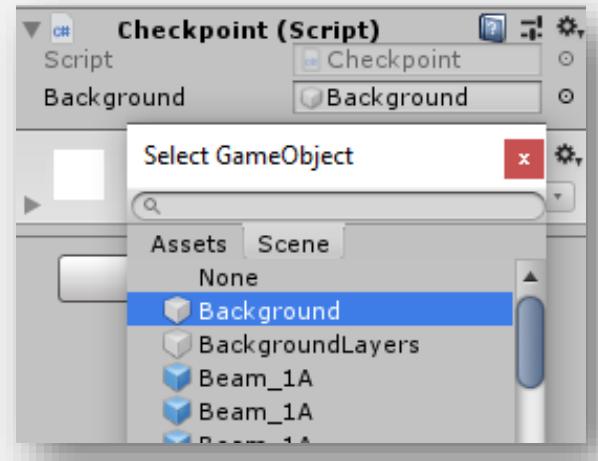
- 18** Based on our earlier investigation, we know that the background object needs to know about Codey's checkpoint. Add public GameObject background; inside the two curly brackets.

```
public class Checkpoint : MonoBehaviour
{
 public GameObject background;
}
```

- 19** Save your script and go back to Unity. Click the little circle next to Background None (Game Object) to open the GameObject selector.

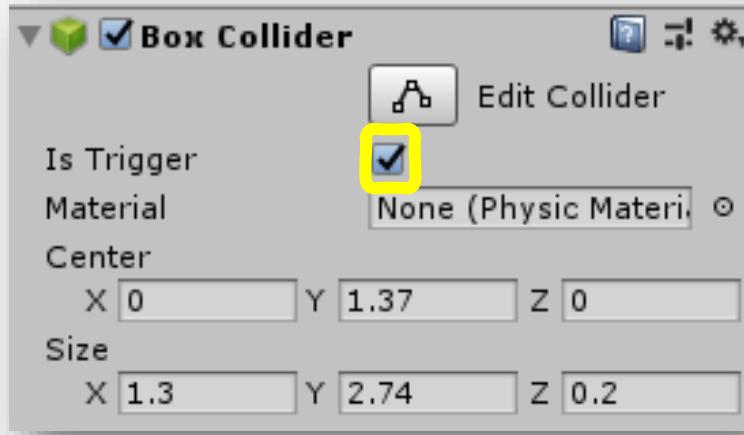


- 20** Find "Background" and click it. Close out of the Select GameObject window. We can now reference and use the background in our Checkpoint script!



**21** While in the Inspector, look at the checkpoint's Box Collider component.

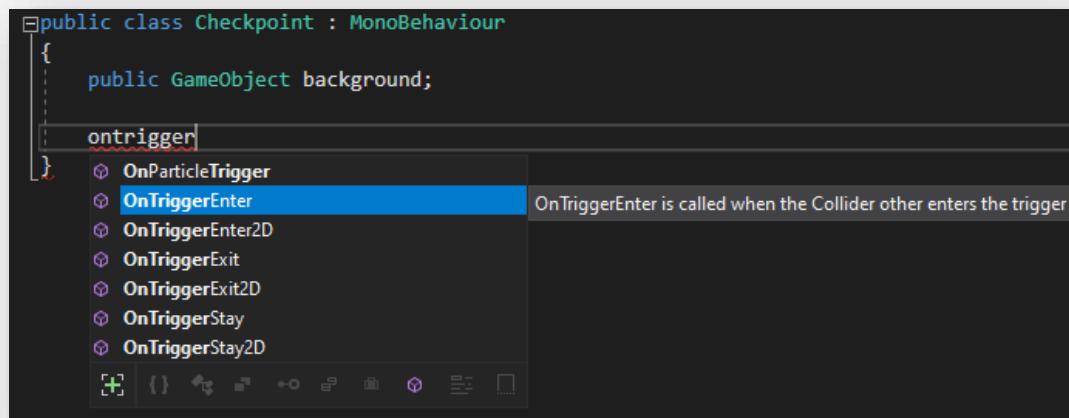
We have Is Trigger enabled. This means that we can check to see if Codey collides with the checkpoint object, but the object will not interact with Codey's movement.



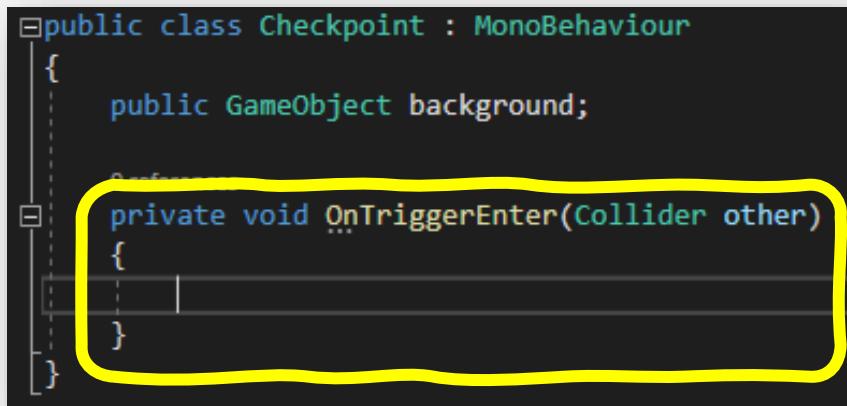
**22** Open the Checkpoint script in Visual Studio.

After the `public GameObject background;` line, start typing "ontrigg" and Visual Studio should pop up a box of suggestions for you.

Double click on OnTriggerEnter to have the function automatically created for you!



- 23** If it didn't work, type `private void OnTriggerEnter(Collider other) { }`, making sure to leave an empty line between the curly brackets.



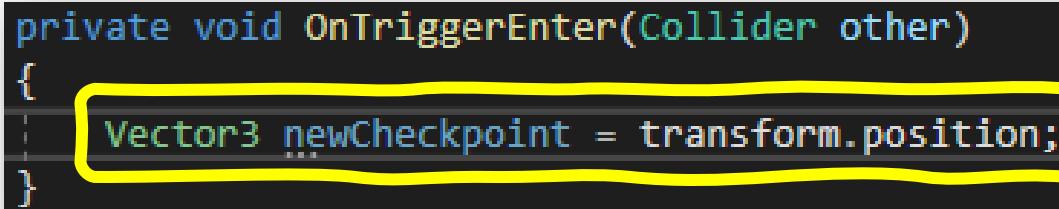
```
public class Checkpoint : MonoBehaviour
{
 public GameObject background;

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

- 24** What do we need to do when Codey runs into the checkpoint? We need to grab the position of the checkpoint object and ask the background to update its checkpoint variable to the new position.

Inside the `OnTriggerEnter` function create a new `Vector3` variable named `newCheckpoint` and set it equal to the `checkpoint` game object's position by typing

```
Vector3 newCheckpoint = transform.position;
```



```
private void OnTriggerEnter(Collider other)
{
 Vector3 newCheckpoint = transform.position;
}
```

- 25** In order to send this new checkpoint to the background, we need to find the background's `GameManager` component set its `checkpoint` variable to be equal to our new checkpoint.

After the `newCheckpoint` line, type

```
background.GetComponent<GameManager>().checkPoint =
newCheckpoint; .
```

**26** Play your game and test out your new checkpoint!

- What happens when you touch the checkpoint and run into vines?
- What happens when you run out of lives?
- What happens if you jump over the checkpoint and touch the vines?

**27** In Unity, open the game's scene by clicking on the Scene tab.



**28** Click on the checkpoint crystal in the scene to select it.



**29** Place the crystal where you think there should be a checkpoint. This is up to you! It can be anywhere!

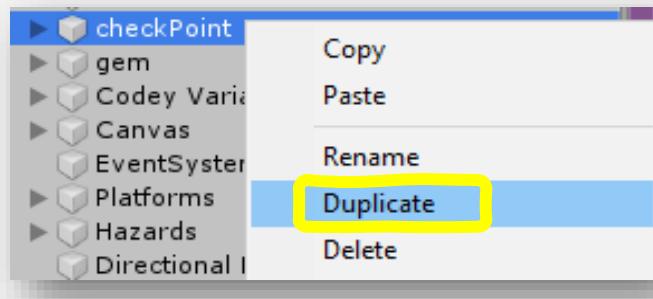
What happens when you place it in the air? On a platform? In vines?



**30** Playtest your game and settle on a good place for your checkpoint!

**31** What if you want to add a second checkpoint somewhere else in the level? All you need to do is duplicate our existing checkpoint!

**32** Find the checkPoint game object in the Hierarchy. Right click it and select Duplicate.



**33** You should now have a new game object called checkPoint (1) in your scene.

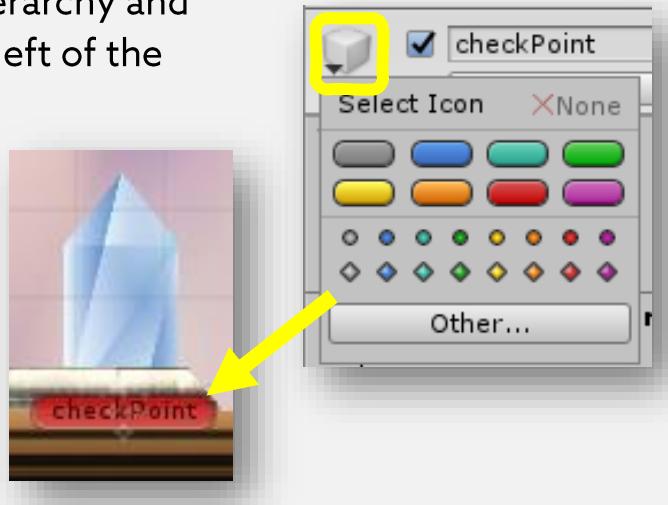


- 34** This object is a perfect copy of the original checkpoint, down to the position in the scene.

But how do we know which is which?

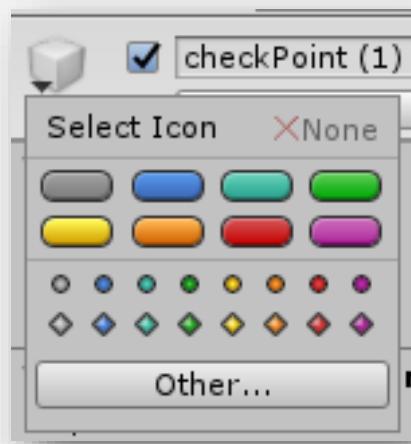
Click on checkPoint in the Hierarchy and look at the Inspector. To the left of the name is a gray cube.

Click on the cube to open the Select Icon window. Select one of the ovals and see what changes in the scene!

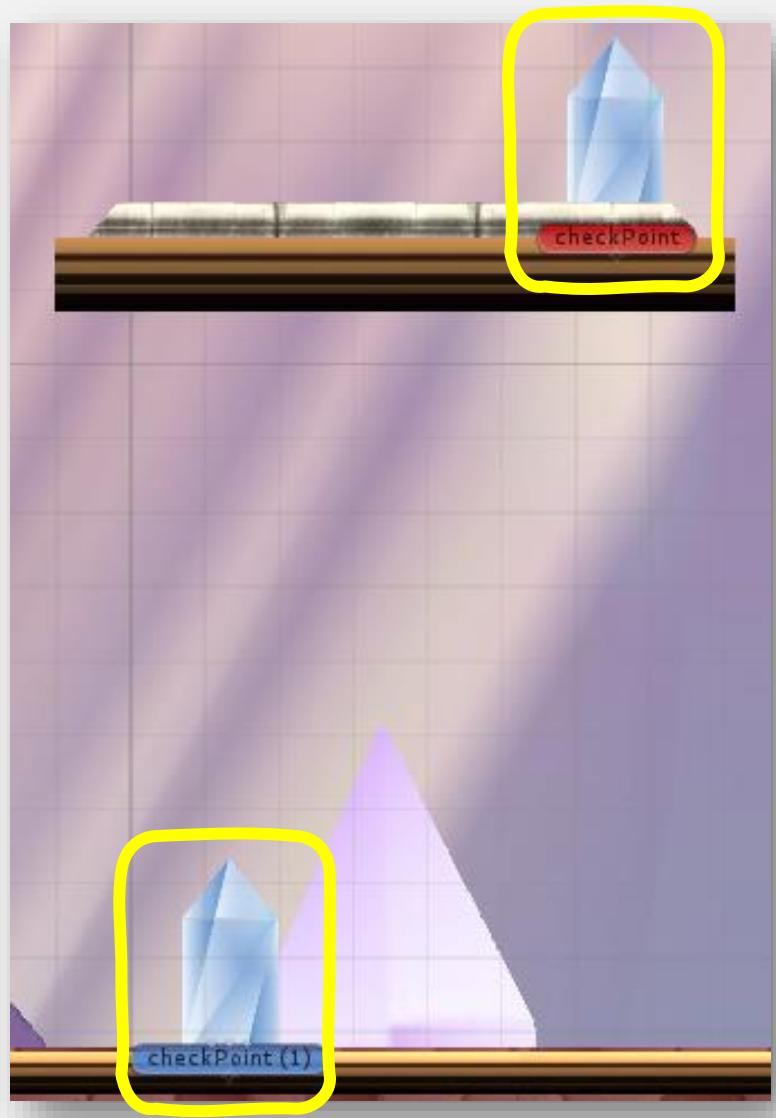


- 35** Click on checkpoint (1) in the Hierarchy and look at the Inspector. Click on the gray cube to open the Select Icon window. Select a different color than you did for the first checkPoint object.

- 36** Click on the new checkPoint (1) object and drag it to somewhere else in the scene.



**37** Now that we can tell them apart, move checkPoint (1) somewhere else in the scene.



**38** Playtest your game.

Try to play each possible outcome with checkpoints!

- Skip the first checkpoint and run into the vines.
- Skip the second checkpoint and run into the vines.
- Touch both checkpoints and run into the vines.
- Touch the first, then the second, and then the first again before running into the vines.

**39** The goal of this level is to use the teleporter at the right side of the scene. We need to add and code a key to unlock the teleporter for Codey!

**40** Find the blue gem on a platform in the top center of the scene. What happens now when Codey touches the gem? Nothing!

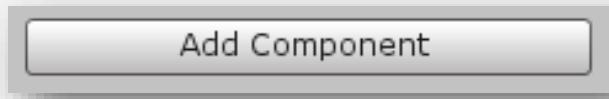


**41** Like the checkpoint, the gem has a box collider set as a trigger. We just need to create the script and code it!

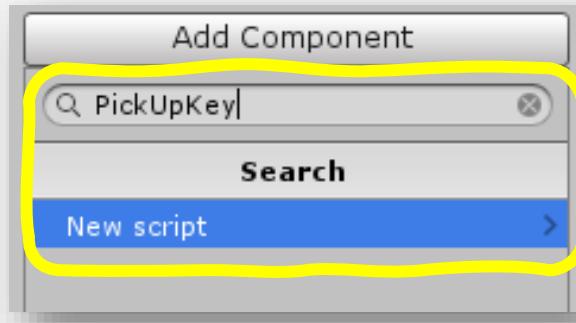
**42** Click on the gem game object in the Hierarchy.



**43** In the Inspector, click on Add Component.



**44** In the box, type "PickUpKey" and click on the New Script option.



**45** Make sure the name is correct and click Create and Add to create the script and add it to the gem component.



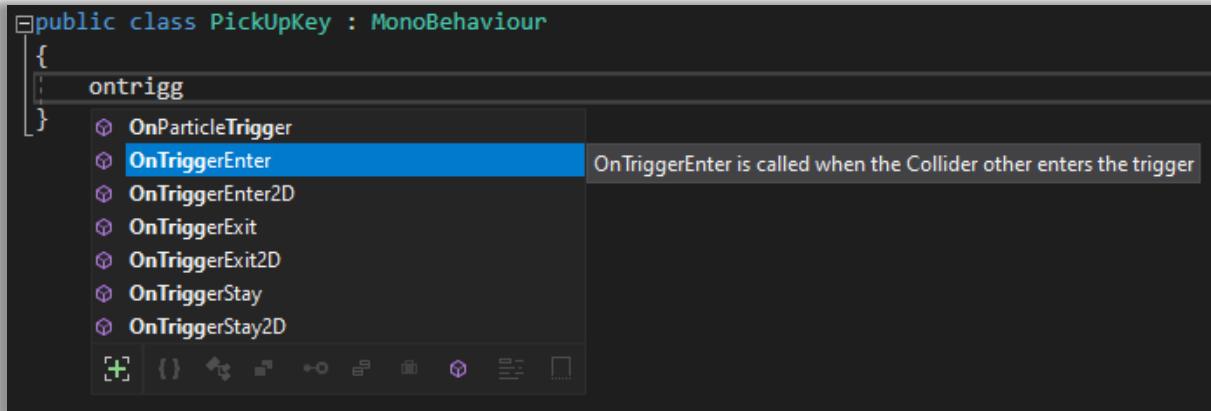
**46** Double click on the Script PickUpKey box in the Inspector to open it in Visual Studio.



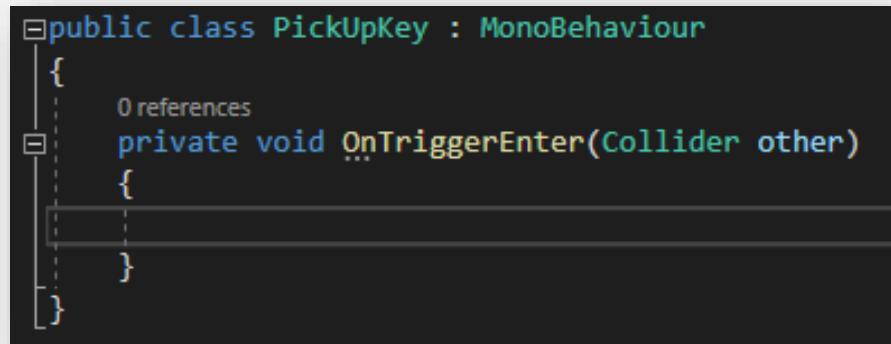
**47** We do not need the Start or Update functions so delete them both.

```
public class PickUpKey : MonoBehaviour
{
}
```

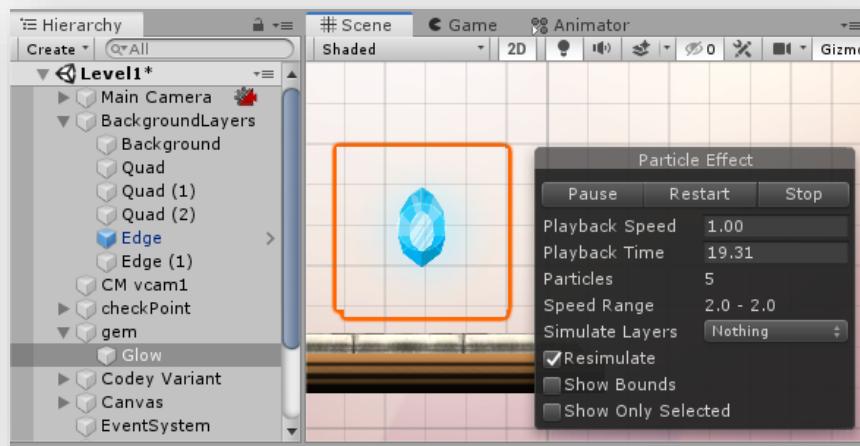
**48** We want to detect when Codey touches our gem, so start typing “ontrigg” and Visual Studio should pop up a box of suggestions for you. Double click on OnTriggerEnter to have the function automatically created for you!



**49** If it didn't work, type `private void OnTriggerEnter(Collider other) { }`, making sure to leave an empty line between the curly brackets.



**50** When Codey touches the gem, we need to disable the gem and the attached particle system. Think back to the Colors game you made earlier in this belt. Do you remember how you disabled an object and its particle system?



**51** If we disable the parent gem object, we will at the same time disable the child Glow particle system! Inside the `OnTriggerEnter` function, type `gameObject.SetActive(false);` to grab the gem's game object and disable it.

```
private void OnTriggerEnter(Collider other)
{
 gameObject.SetActive(false);
}
```

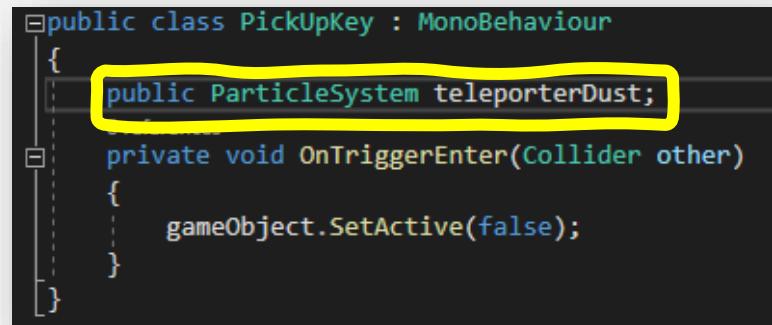
**52** Save your script and play your game. What happens when Codey touches the gem now? The gem sprite and the glow from the attached particle system should disappear!

Now that you have the gem, try to find a teleporter to leave the level. Nothing happens!

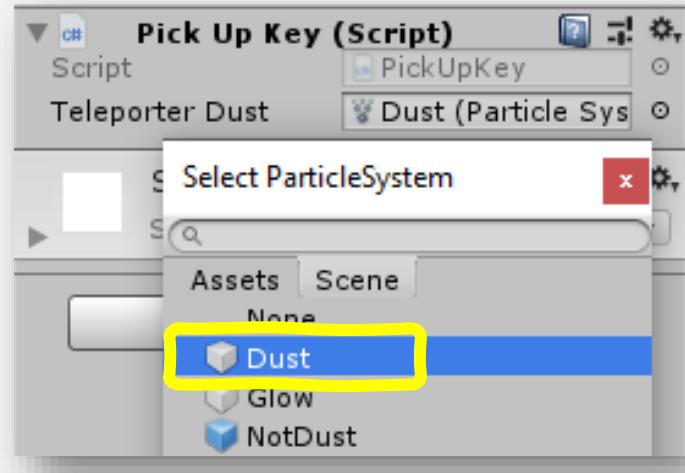
We just need to create a script and write a few more lines of code to get our teleporter working!



- 53** We can use a particle system to communicate to the player that the teleporter is active. We need to enable the teleporter's particles after Codey touches the gem. We first need to create a public variable for the teleporter's particle system inside the PickUpKey script. Before the OnTriggerEnter function, type `public ParticleSystem teleporterDust;` to create a variable we can set in the inspector.



- 54** In the gem's Inspector, find the Pick Up Key (Script) component. Click the little circle next to Teleporter Dust None (ParticleSystem) to open the Select ParticleSystem menu. Find and click on Dust to attach it to the PickUpKey script as the `teleporterDust` variable.

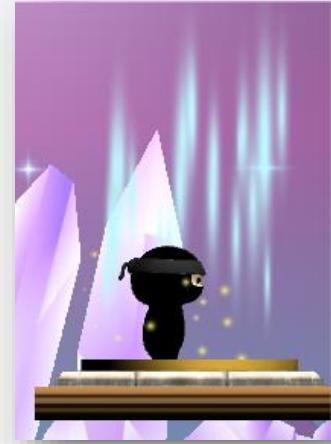


- 55** Back in the PickUpKey script, we need to tell the teleporterDust to play after Codey picks up the gem. After the `gameObject.SetActive(false);` line, add `teleporterDust.Play();` to tell the particle system to play!

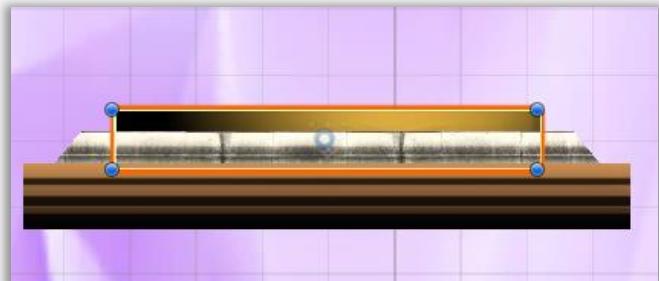
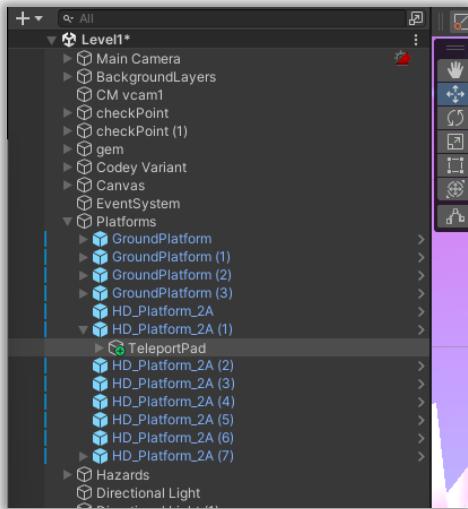
```
private void OnTriggerEnter(Collider other)
{
 gameObject.SetActive(false);
 teleporterDust.Play();
}
```

- 56** Playtest your game. Look at the teleporter on the right of the scene before and after you pick up the gem.

The teleporter still doesn't send Codey anywhere, but the player can now see when the teleporter is active thanks to the particle system!



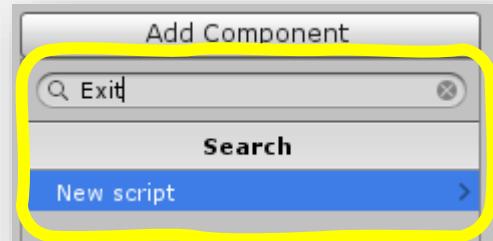
- 57** In the scene, click on the teleporter to select it in the Hierarchy and the Inspector. Alternatively, you can find it directly in the Hierarchy in **Platforms -> HD\_Platform\_2A (1) -> TeleportPad**



- 58** In the Inspector, click on Add Component.



- 59** Type "Exit" and select "New Script".



- 60** Make sure the script name is "Exit" and click the Create and Add button to create the script and attach it to the teleport object.



- 61** In the inspector, double click the grey box that says Script Exit to open the script in Visual Studio.



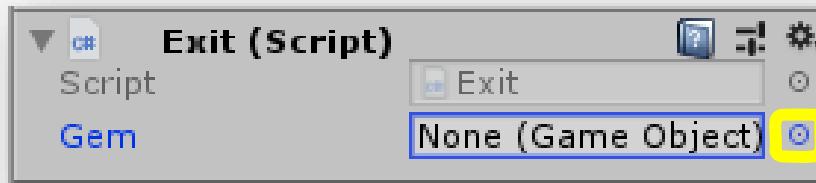
- 62** We do not need the Start or Update functions, so we can delete them both.

```
public class Exit : MonoBehaviour
{
}
```

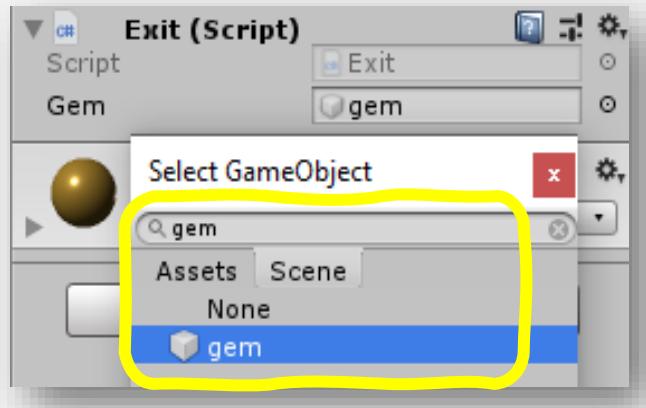
- 63** The teleporter needs to know if Codey has picked up the gem, so we need to create a public reference to the gem so we can connect it to the teleporter through the Inspector. Inside the two curly brackets, type `public GameObject gem;`

```
public class Exit : MonoBehaviour
{
 public GameObject gem;
}
```

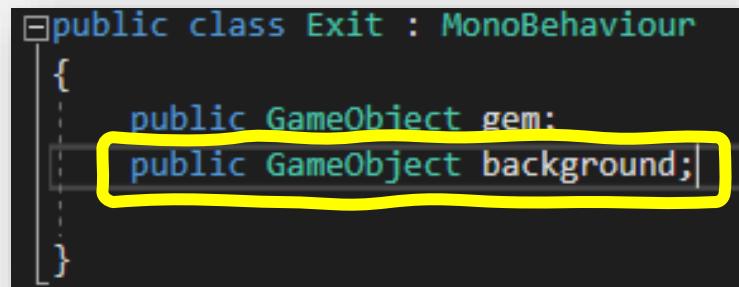
- 64** In the Inspector, click the little circle next to Gem None (Game Object) to open the GameObject selector window.



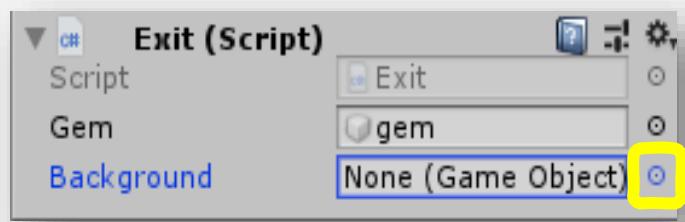
- 65** In the window, search for and select “gem” to attach it to the gem variable inside of our Exit script.



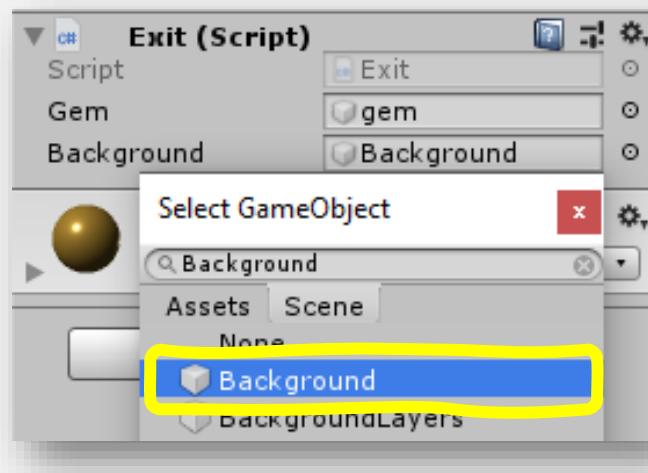
- 66** The teleporter also needs to know about the TeleportOpen function. This function will teleport Codey to another scene! The TeleportOpen function lives inside the GameManager script. The GameManager script is attached to the scene's background object. After the public GameObject gem; line type `public GameObject background;` to create a public variable we can control in the Inspector.



- 67** In the Inspector, click the little circle next to Background None (Game Object) to open the GameObject selector window.



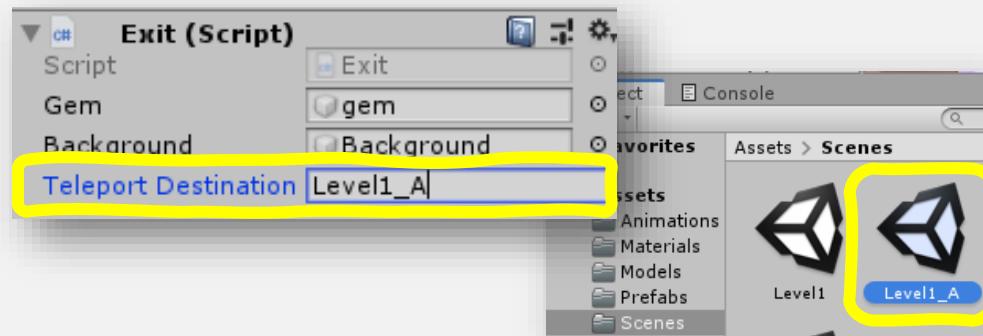
- 68** In the window, search for and select “Background” to attach it to the gem variable inside of our Exit script.



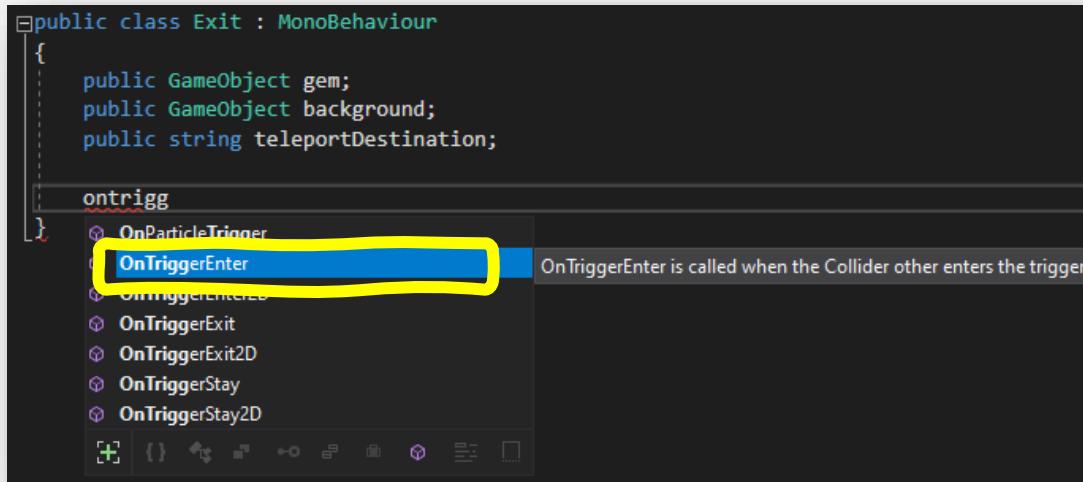
- 69** The final thing that the teleporter needs to know about is where to send Codey. To do this we will create a public string object called `teleportDestination` so we can set it in the Inspector. After the `public GameObject background;` line type `public string teleportDestination;`

```
public class Exit : MonoBehaviour
{
 public GameObject gem;
 public GameObject background;
 public string teleportDestination;
}
```

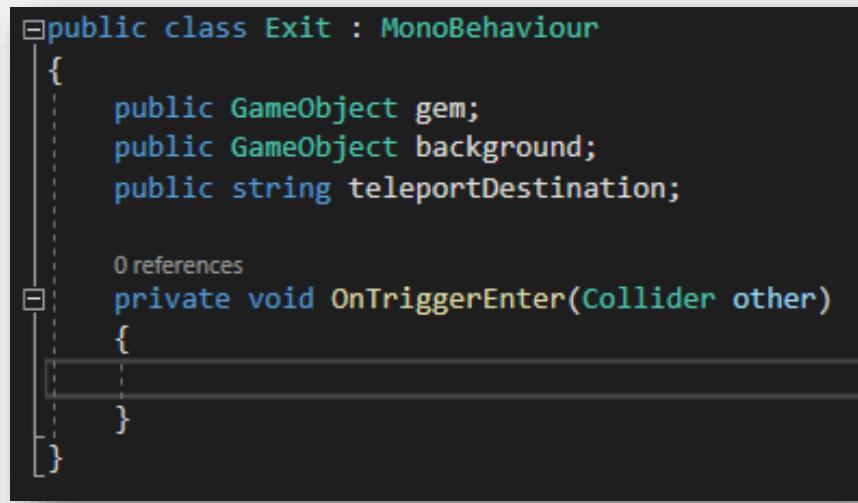
- 70** In the Inspector, give Teleport Destination a value of `Level1_A`. Make sure you type it in exactly to match the name of the scene in the game.



- 71** Now that we have our variables set up, we want to detect when Codey is touching our teleporter, so after the public string teleportDestination; line start typing “ontrigg” and Visual Studio should pop up a box of suggestions for you. Double click on OnTriggerEnter to have the function automatically created for you!



- 72** If it didn't work, type `private void OnTriggerEnter(Collider other) { }`, making sure to leave an empty line between the curly brackets.



**73** When Codey enters the teleporter, we want him to teleport to next level based on our teleportDestination variable. We can do that by asking the background's GameManager component to run the TeleportOpen function with our teleportDestination string. Basically, we are asking the teleporter to send Codey to another one of our scenes. Inside the OnTriggerEnter function type

```
background.GetComponent<GameManager>().TeleportOpen(telepor
tDestination);
```

```
private void OnTriggerEnter(Collider other)
{
 background.GetComponent<GameManager>().TeleportOpen(teleportDestination);
}
```

**74** Play the game and collect the gem and go to the teleporter. Did something happen when you collided with the teleporter before you picked up the gem?



We need to create the logic that requires Codey to have touched the gem before enabling the teleporter!

- 75** In the OnTriggerEnter function, create an if statement by typing if () {} and put the background.GetComponent line of code inside of the if statement's curly brackets.

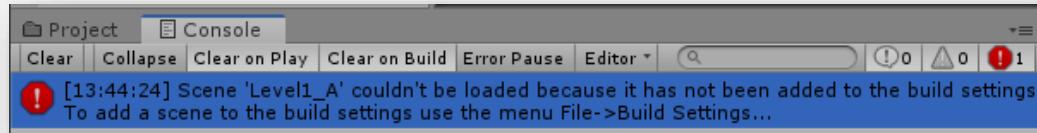
```
private void OnTriggerEnter(Collider other)
{
 if ()
 {
 background.GetComponent<GameManager>().TeleportOpen(teleportDestination);
 }
}
```

- 76** Inside of the parentheses, we need to see if the gem is disabled before teleporting Codey. Type `gem.activeInHierarchy == false` inside the parentheses to run the body of the if statement only if the gem is not active in the Hierarchy.

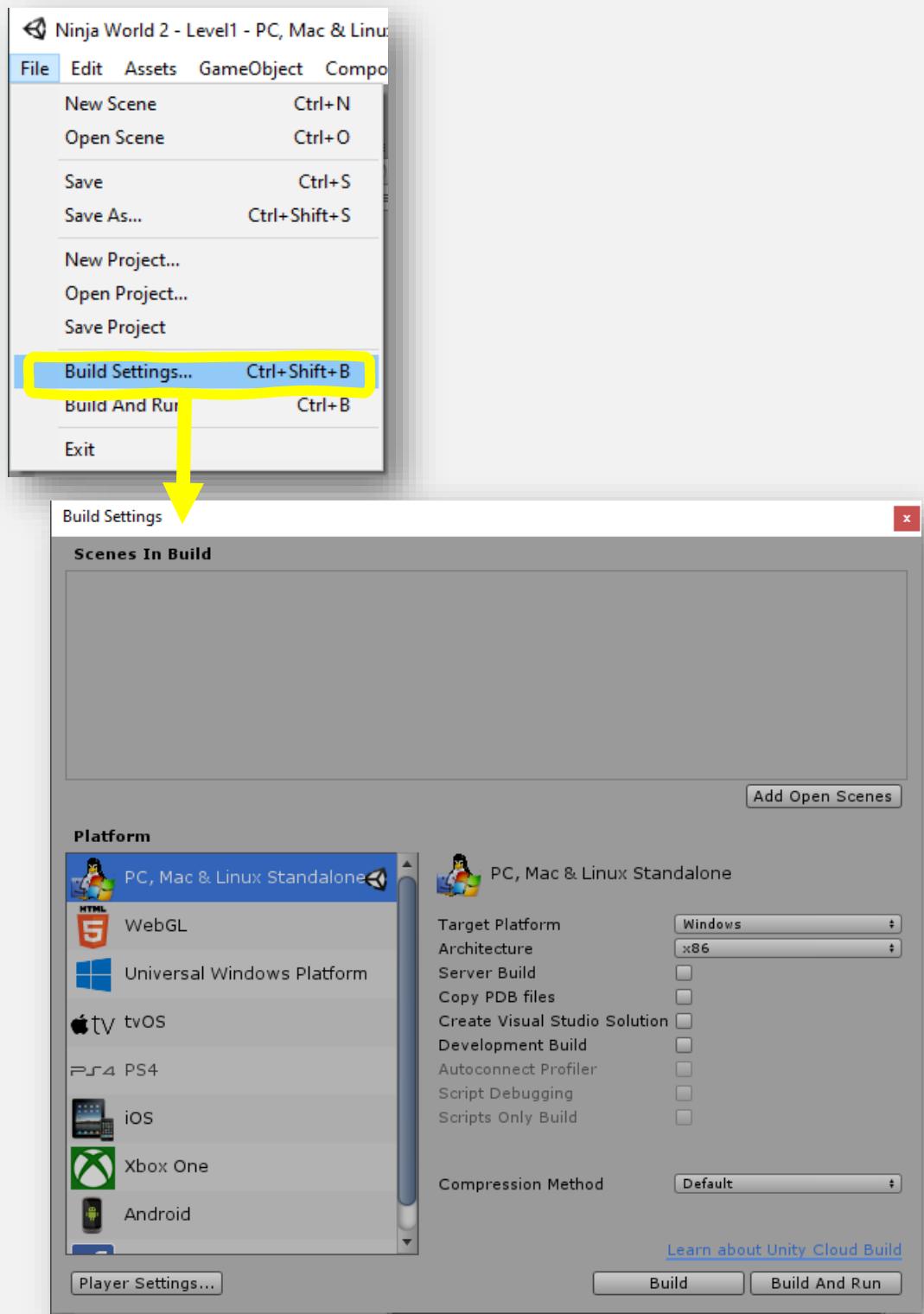
```
private void OnTriggerEnter(Collider other)
{
 if (gem.activeInHierarchy == false)
 {
 background.GetComponent<GameManager>().TeleportOpen(teleportDestination);
 }
}
```

- 77** Now play your game, making sure to touch the teleporter before you collect the gem. Nothing should happen until after you collect the gem and then touch the teleporter.

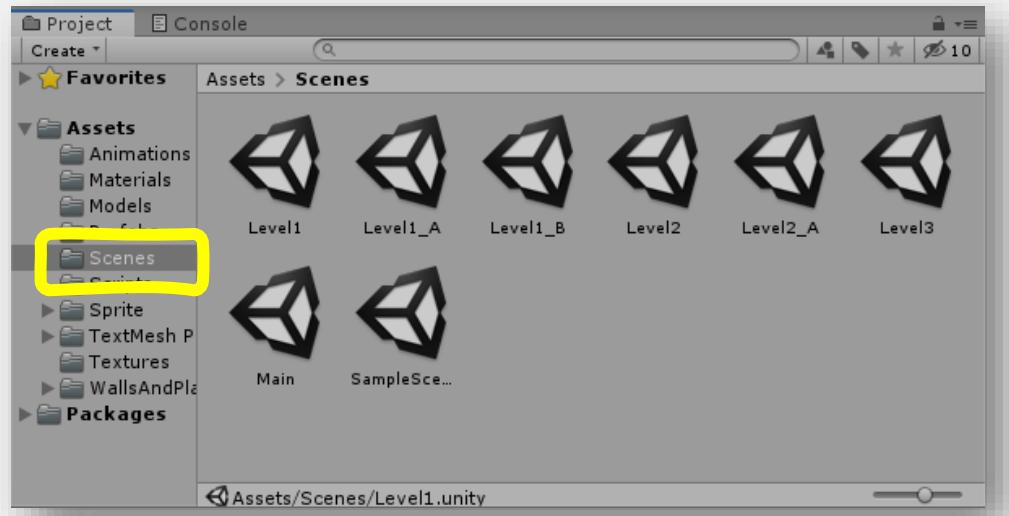
Once you get the gem and touch the active teleporter, what happens? You should get a Good Job screen with a Click to Continue button. What happens when you click that button? Did you get this error? Unity can't find the Level1\_A scene because it is not loaded in the Build Settings menu! We can easily fix that!



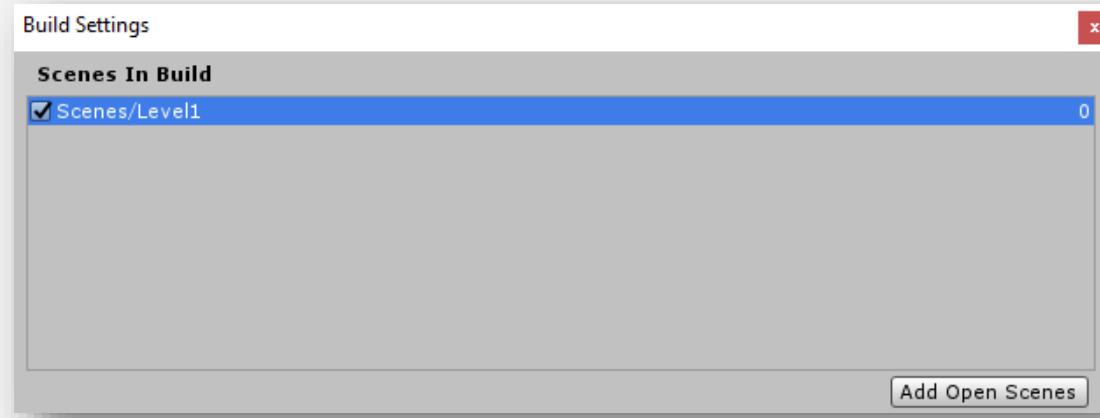
**78** In Unity, go to File -> Build Settings to open the Build Settings window.



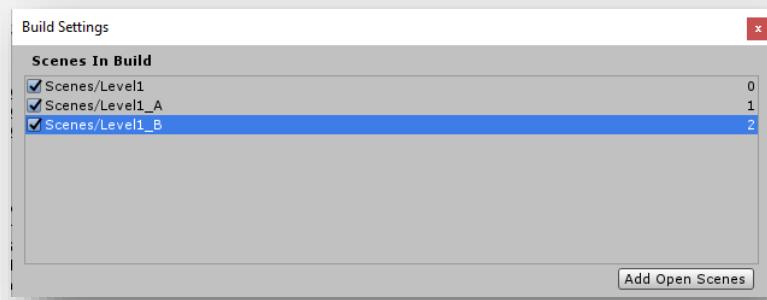
**79** Also open the Scenes folder located in the Project tab in the Assets -> Scenes folder.



**80** Click, hold, and drag the Level1 scene to the box under Scenes in Build.



**81** Repeat this for the Level1\_A and Level1\_B scenes.



**82** Adding these scenes to the Scenes in Build setting will tell Unity to load all three scenes whenever you play the game. It can now find the teleporter's destination of Level1\_A!

**83** Play your game and see what happens when Codey enters an activated teleporter!



# Prove Yourself

## Task

Using what you learned in the lesson try to complete this new level! Take note of what features work and what features don't work because it's your job to fix it.

Attach the PickUpKey script to the gem and the Exit script to the teleporter.

Make sure you edit the Teleport Destination variable to point to Level1\_B.

Get creative! Add in new checkpoints and teleporters that send Codey to different scenes!

## Activity 14

# Amazing Ninja Worlds - Part 3

In this final part of building your Super Ninja World platform game, you'll add a start screen and a menu. The start screen and menu will make it feel like a completed game!

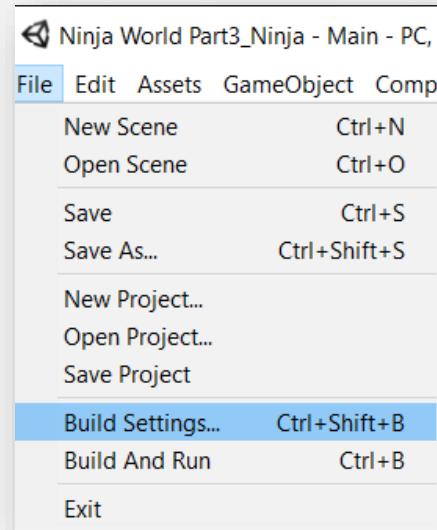


- 1** You should continue with the project that you had completed in the previous activity.
- 2** Now that you're comfortable moving the player around the environment and between scenes, we can make this into a complete game by adding a start screen and a menu.
- 3** Remember using the Build Settings in World of Color? We are now going to use it to manage even more scenes.

Click on the **File** tab and click on **Build Settings**.

*Reminder: The main role of the Build Settings is to export your project so that others can play it.*

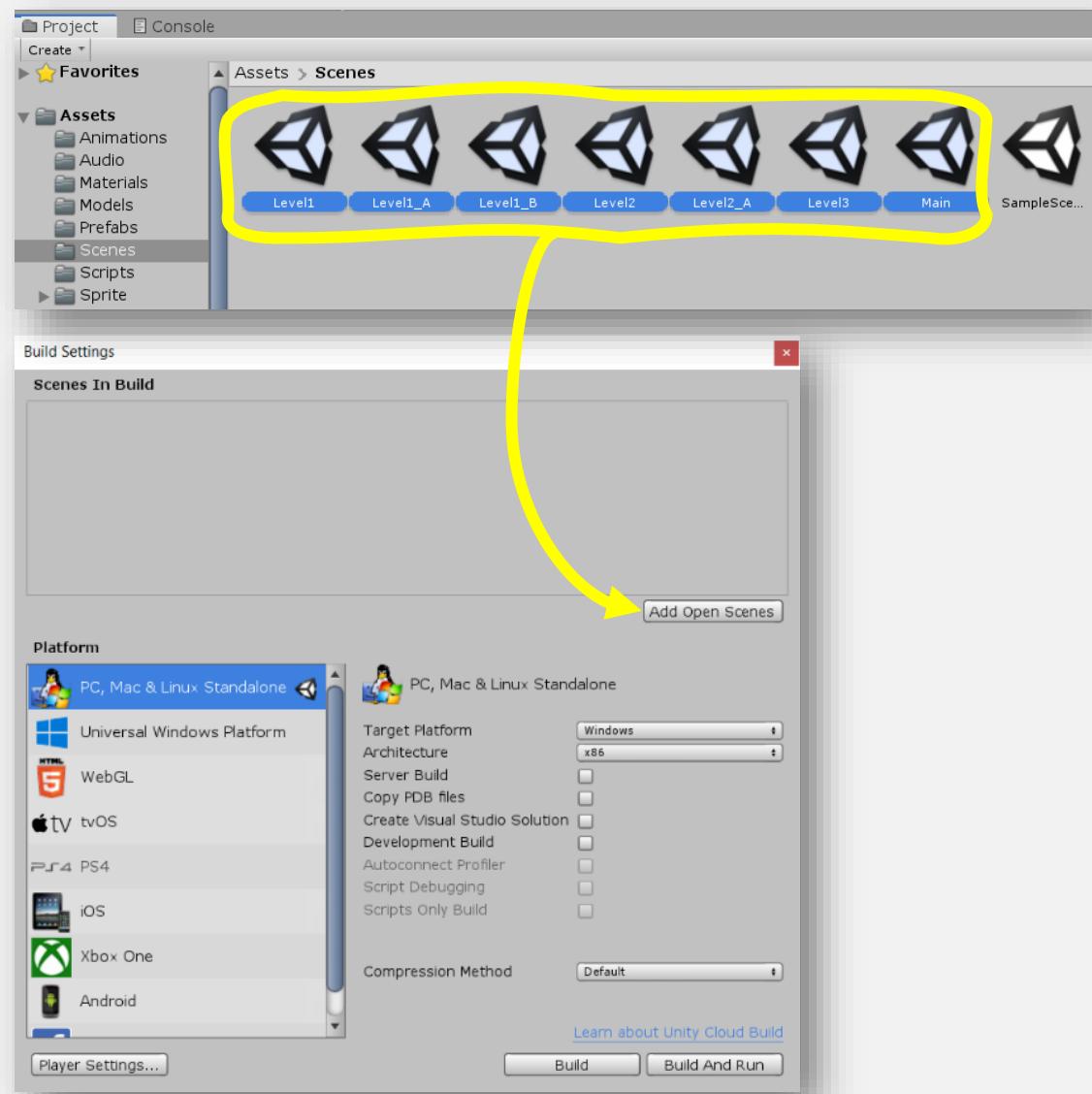
When you have multiple scenes like in this game, all scenes need to be in the Scenes in Build section of the Build Settings. If not, Unity won't be able to load any of the scenes by name.



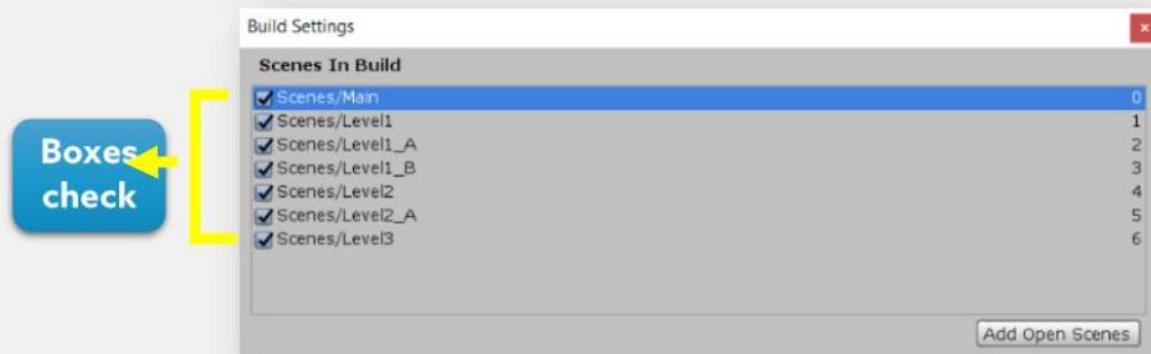
#### **Pro Tip:** Use the Scene's Name

*This game has more than five scenes and it's possible that there will be more. If scenes get added (or removed), their order might change. What was once scene number 1 might get moved to 3 or anywhere else. By referring to the scenes by name, you'll always get the scene you want.*

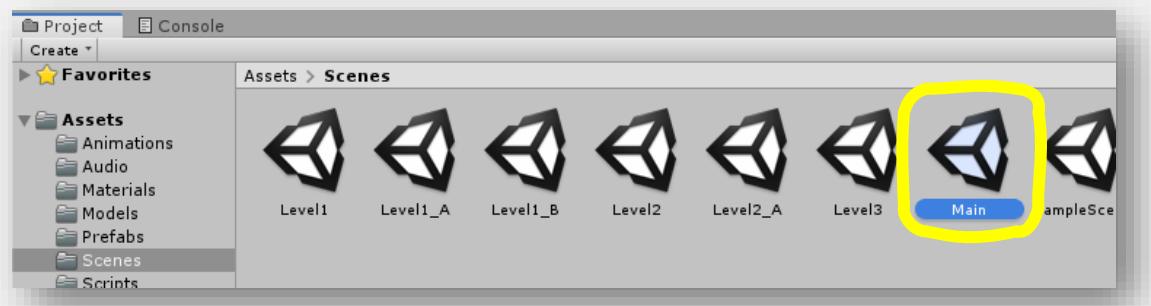
- 4** With **Build Settings** still open, select all the scenes in the **Project** window (except for SampleScene) and drag them into the **Scenes in Build** box in Build Settings.



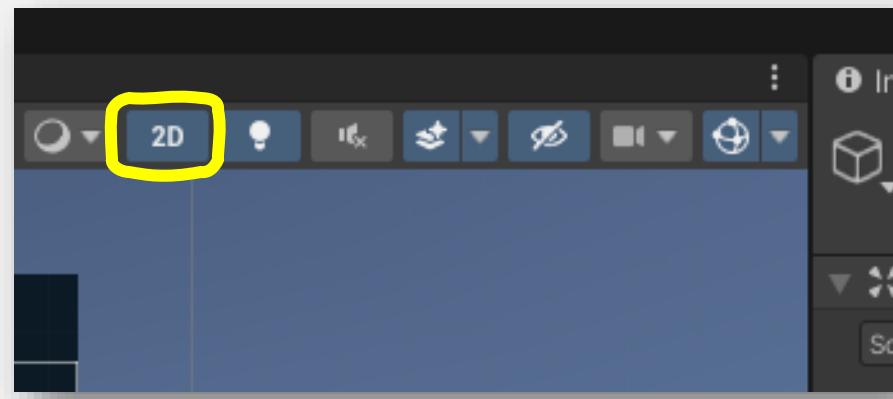
- 5** Make sure that all their boxes are checked. Highlight the **Main** scene and drag it to the top so that it is scene 0. Whatever scene is at the top of the list gets loaded first when the game is started after the game is built.



- 6** Now we can set up the main menu where the player can pick which scene they want to play. Close the Build Settings menu and load the **Main** scene by selecting it from the **Scenes** folder in the **Project** window.



Since the menu is 2D, make sure the **Scene** window is in **2D** mode.

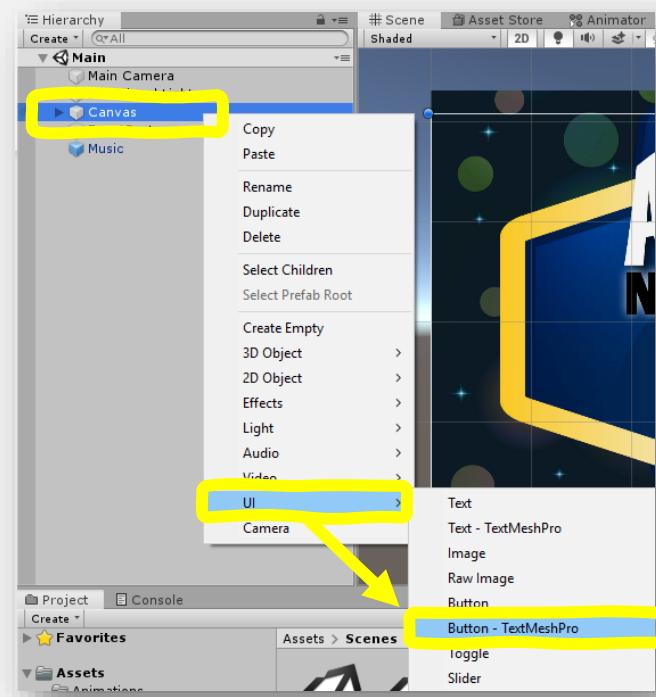


**7** The menu canvas object has been set up with an image, but there are no interactive buttons. We need to add buttons that send the player to the right scenes.

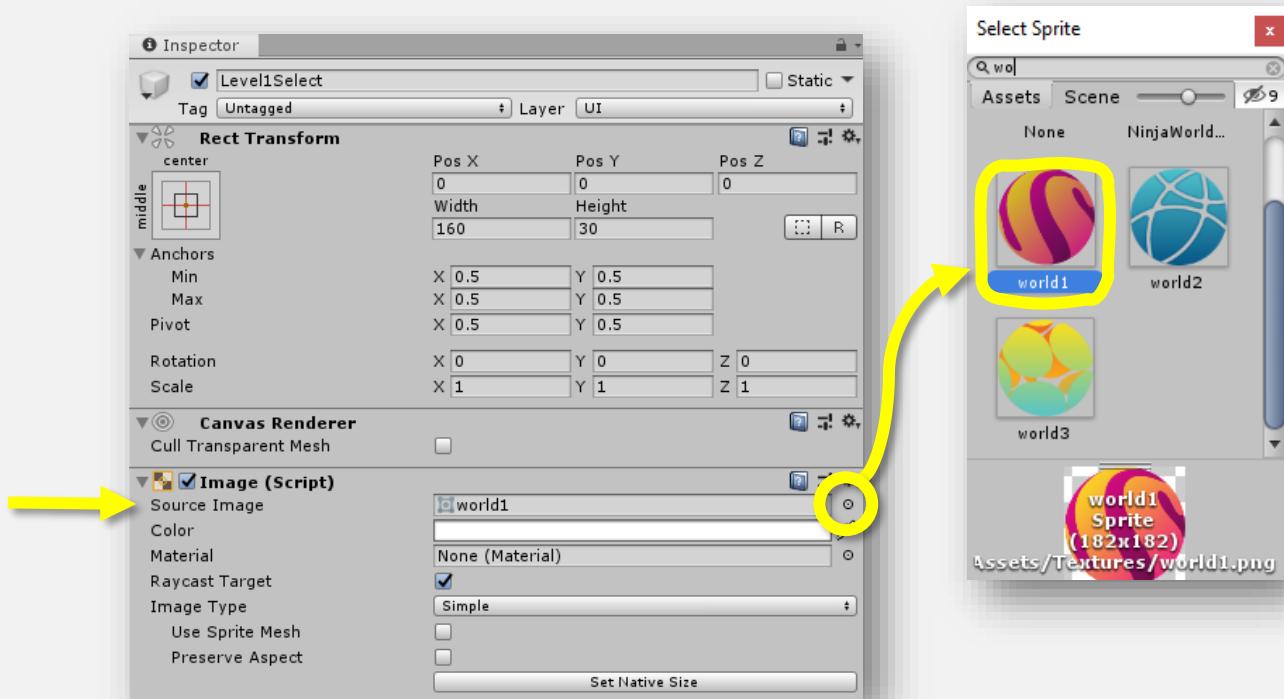
With the **Canvas** selected, add a new GameObject from **UI/Button - Text Mesh Pro**.

We're using **Text Mesh Pro** instead of an ordinary button since we want to include some special text with the button.

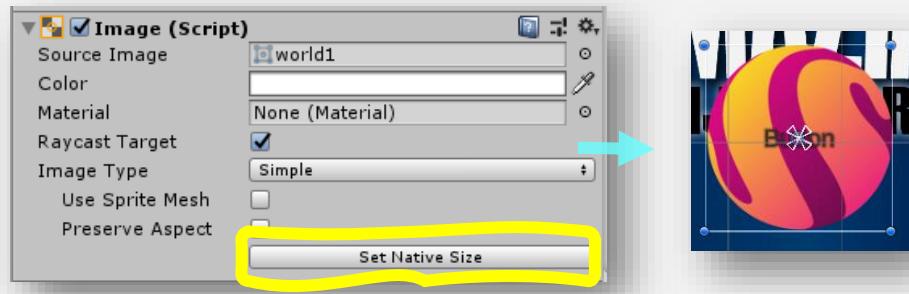
Change the name of the object to "**Level1Select**".



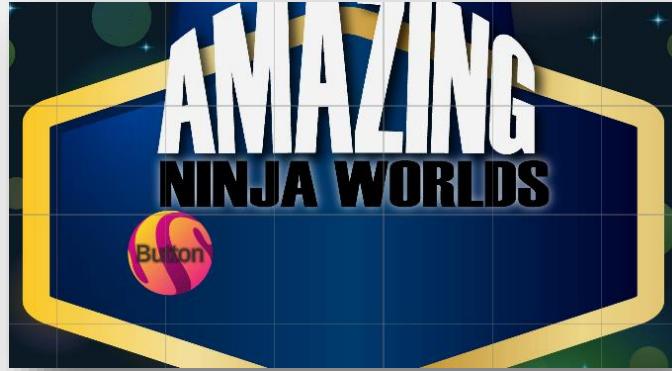
**8** In the **Inspector** for the **Level1Select** button, change the **Source Image** in the **Image** component to "world1."



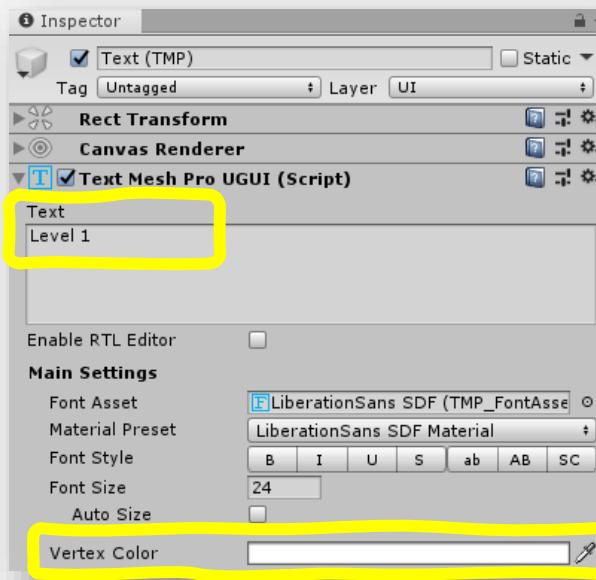
**9** You may notice that the button looks stretched. It looks like an oval rather than a circle. This means that it has the wrong aspect ratio. To fix this, in the **Inspector**, click **Set Native Size**.



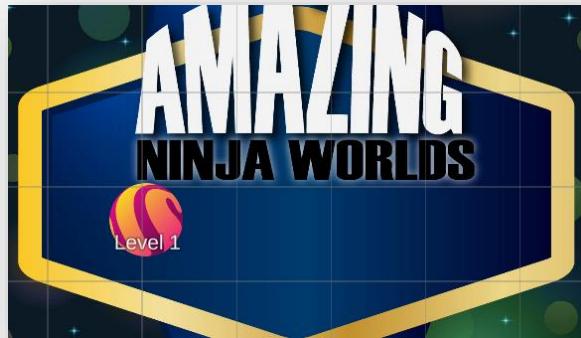
**10** Resize and move the button object so that it fits well under the game title. There will be three buttons total, so move this button to the left side of the menu as shown.



**11** The button comes with a text object attached. Select it. In the **Inspector**, change the **Text** to "Level 1" and change the **Vertex Color** to white.



- 12** Move the text object so that it is in the lower area of the button as shown:



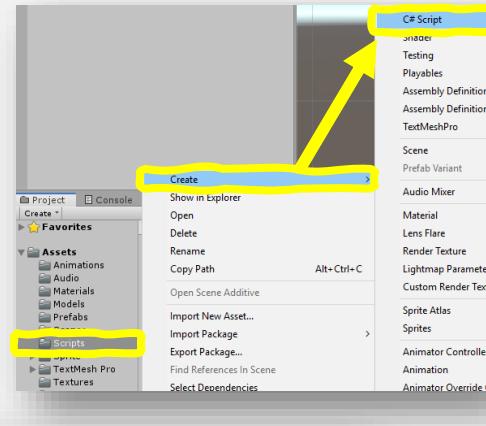
### Master Your Skill

Text Mesh Pro (or "TMP") has lots of options to modify the look of the text. You can add things like a drop shadow and so on. Feel free to take a moment to experiment with the

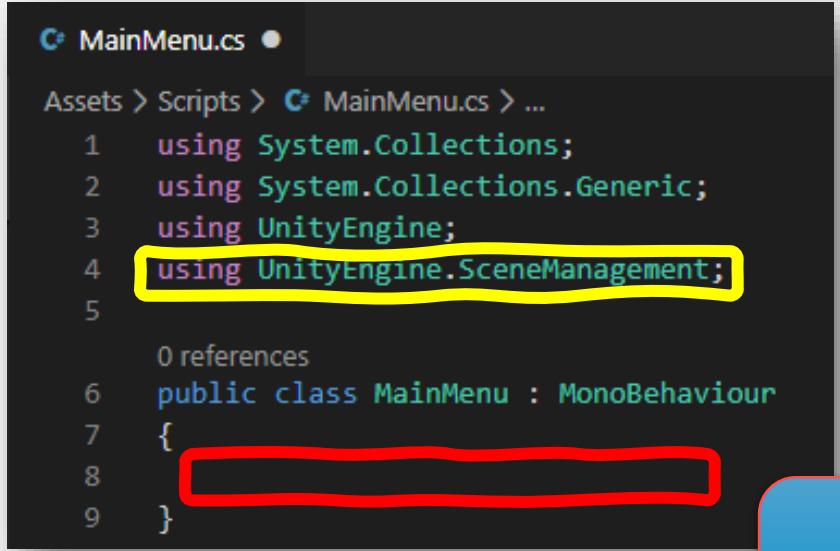
...

- 13** We need a script to control what happens when the user clicks on the button. Create a **C# script** in the **Scripts** folder and name it **"MainMenu"**. Drag the script component onto the **Level1Select** game object.

**IMPORTANT** – As you might have guessed, we will be duplicating the **Level1Select** button for **Level2** and **Level3**, but do not do so until instructed to do it. Complete **Level1Select** button first in order to save yourself some time and frustration.



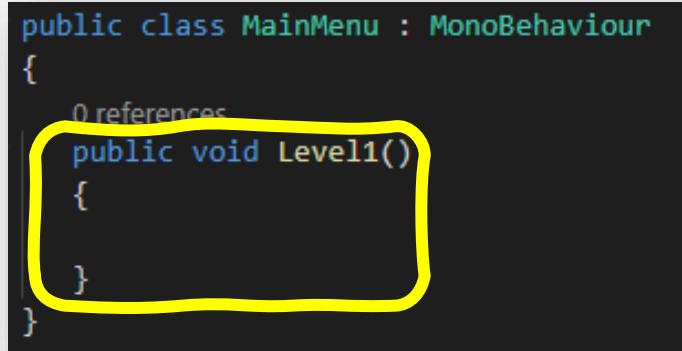
**14** Open the **MainMenu** script. We won't be using the **Start** or **Update** functions, so delete those. This script will not need any variables, but you will need to add the scene management directive at the top:  
`UnityEngine.SceneManagement`



```
C# MainMenu.cs •
Assets > Scripts > C# MainMenu.cs > ...
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.SceneManagement;
5
0 references
6 public class MainMenu : MonoBehaviour
7 {
8 }
9 }
```

**Deleted:**  
`void Start()`  
`void Update()`

**15** Create a new function as follows:



```
public class MainMenu : MonoBehaviour
{
 0 references
 public void Level1()
 {
 }
}
```

**16** Inside the function, we will use the **SceneManager** to change scenes:

```
public void Level1()
{
 SceneManager.LoadScene("Level1");
}
```

### Careful!

Check your spelling.  
Make sure it matches  
the scene name exactly.  
As always, case  
matters!

**LoadScene** does exactly what it says – it replaces the current scene with the scene you've chosen. When **LoadScene** is used, the new scene behaves exactly like it would when you start it the first time.

**17** When we load a new game- a new level 1- we need to think about what happens to the other variables, such as the lives left.

#### How Does Ninja World Handle Lives Left?

The game uses a built-in Unity class called *PlayerPrefs* to track the number of lives left. The lives are saved as "LIVES\_LEFT" in *PlayerPrefs*.

We use this so the number of lives left can be carried over between levels. For example, if you have two lives left at the end of level 1, you will start level 2 with two lives.

When the player starts a new game, we want to make sure that the player has the maximum lives, not the lives left. To do this, use *PlayerPrefs* to delete the "LIVES\_LEFT" key so that the game starts fresh with the maximum 3 lives. Add the following line:

```
public void Level1()
{
 SceneManager.LoadScene("Level1");
 PlayerPrefs.DeleteKey("LIVES_LEFT");
}
```

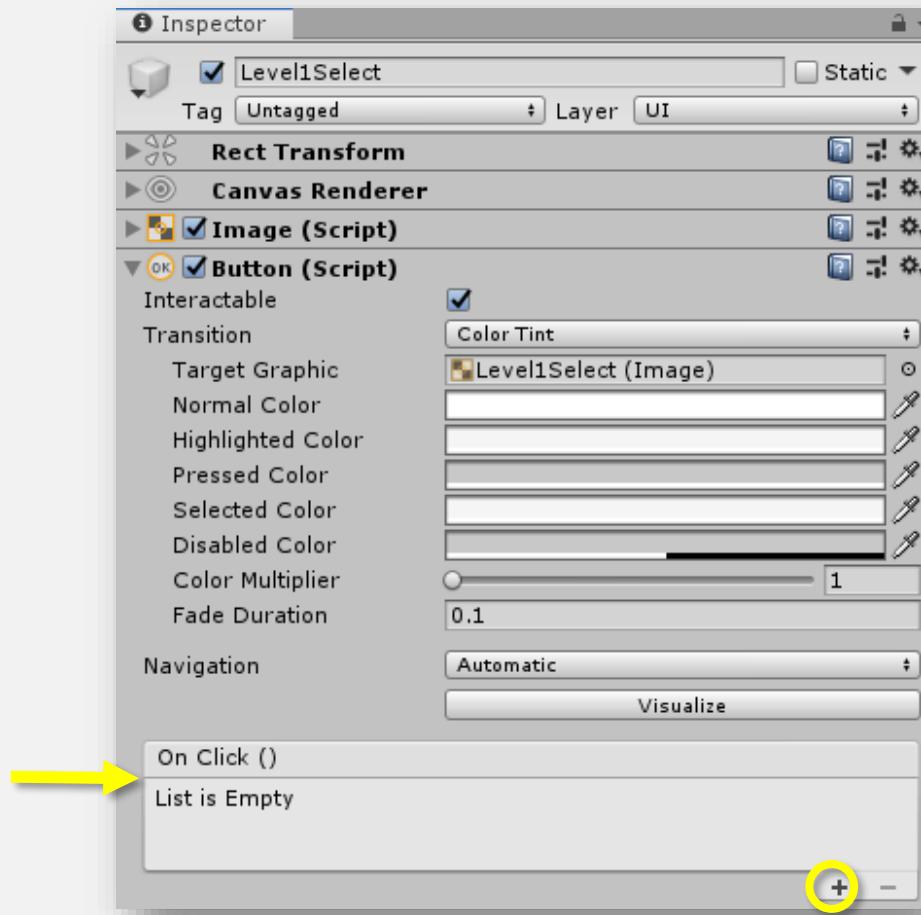
**DeleteKey()** removes the variable completely so that there is no stored value for "LIVES\_LEFT" when beginning the game.

## 18 Save your script.

19 Back in Unity, make sure the **Level1Select** object is selected. We will now set up the button component, so the button actually works.

Scroll down in the **Inspector** to the **Button** component. At the bottom of the component is an empty list for events to run when “**On Click ()**” happens. Click on the **+** symbol to add an event to the list.

The first selector is “**Runtime Only**”. Since we only want this to work when the game is actually being run, leave this as it is.



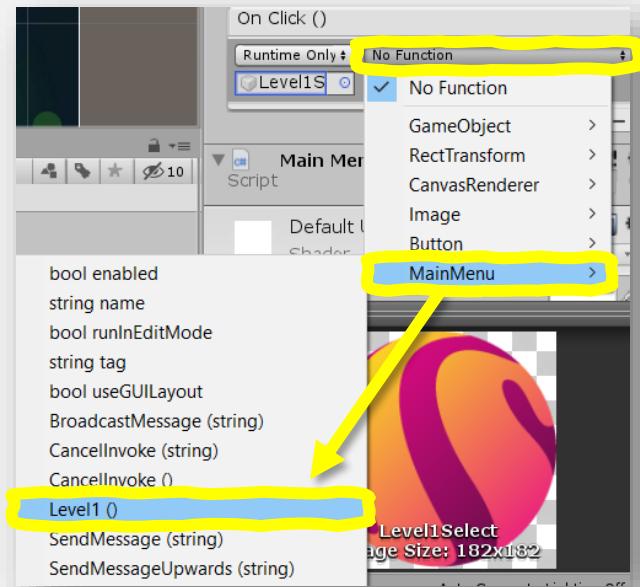
**20** Beneath the **Runtime Only** drop-down, is a slot for the object. Drag the **Level1Select** object from the **Hierarchy** into this **box**.



*Why does Unity insist on referring to an object that the component is already attached to? Because sometimes you may want to have a button affect an unrelated object – so that functionality is there.*

**21** Finally, click on the selector that says, “**No Function**.” In the drop down, select **MainMenu** (the script you just wrote) and select **Level1()** from the options.

This tells Unity: when click on the Level1Select button, run the Level1() function in the MainMenu script.



---

**22** Play the scene. Test the button to see if Level1 is loaded on click.

Is something missing? How do we get back to the main menu? We'll address that in a minute, but first, let's set up the buttons for the remaining levels.

Stop your game.

---

**23** Instead of creating 3 methods, one for each level, since they will each be doing the same thing we can use a method with a parameter!

**24** Open the **MainMenu** script. Change the method from Level1 to LevelChange.

```
public void LevelChange()
{
 SceneManager.LoadScene("Level1");
 PlayerPrefs.DeleteKey("LIVES_LEFT");
}
```

---

**25** Inside the parentheses, define a string parameter called "level".

```
public void LevelChange(string level)
{
 SceneManager.LoadScene("Level1");
 PlayerPrefs.DeleteKey("LIVES_LEFT");
}
```

---

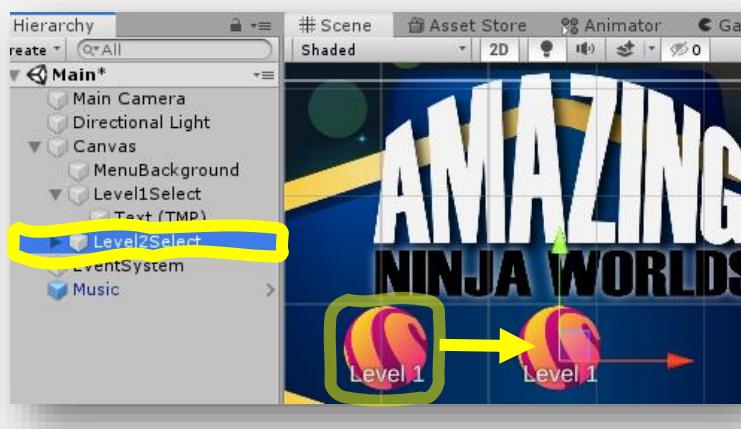
**26** Replace the "Level1" string inside the LoadScene method with our new parameter.

```
public void LevelChange(string level)
{
 SceneManager.LoadScene(level);
 PlayerPrefs.DeleteKey("LIVES_LEFT");
}
```

**27** Back in Unity, select **Level1Select** and change the method to our new **LevelChange** method. In the box below, type in the name of the scene we are going to switch to ("Level1").

**28** Now for the other buttons. Select **Level1Select** and duplicate it using **ctrl+D**. Rename this button "**Level2Select**".

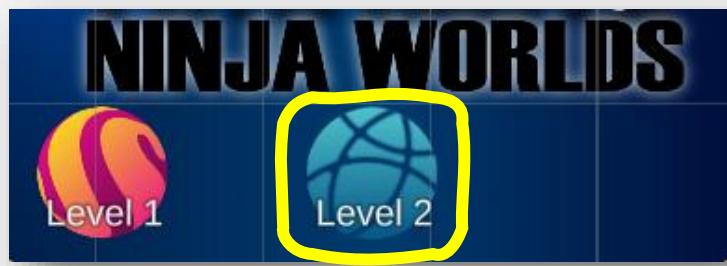
Move it to the right so it is in the middle under the title.



**29** We will go through the same steps to update the image for level 2's button's as we did for level 1. In the **Inspector**, in the **Image** component, change the following items:

- 1)** Change the **Source Image** to "world2".
- 2)** If the image looks a little strange, click on "**Select Native Size.**"
- 3)** Change **Text(TMIP)**'s **Text** component to "**Level2.**"

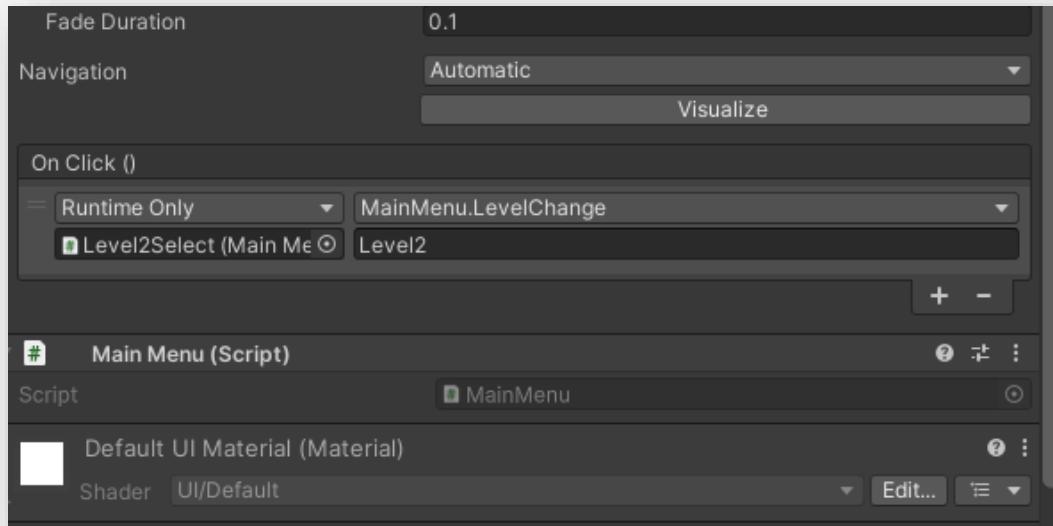
Your Level2Select button should look like this:



**30** For the **Button** component, make the following changes:

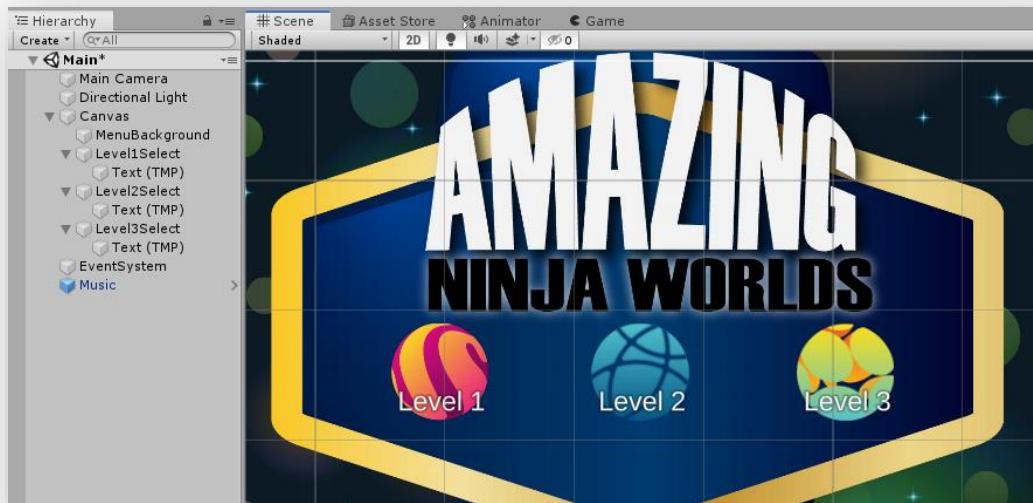
- 1) Change the object in the **On Click ()** list to **Level2Select**.
- 2) Change **No Function** to **MainMenu/LevelChange()**.
- 3) Type in “Level2” as the parameter.

Your Level2Select OnClick () should look as follows:



**31** Repeat the steps to make a **Level3Select** button, changing the number 2 to 3 in all the same places.

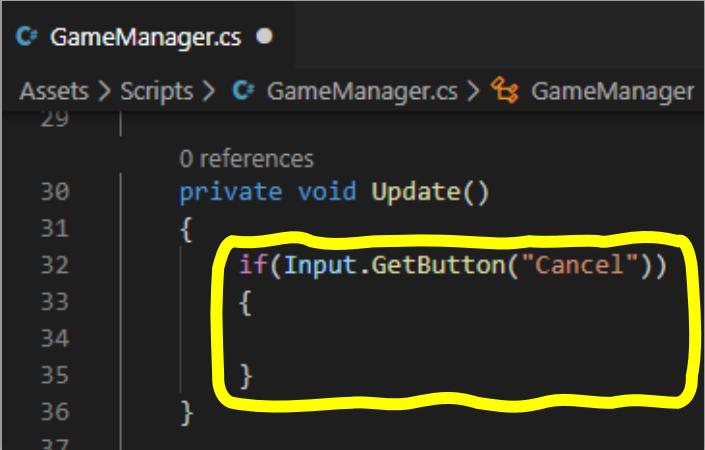
Your Main scene should look as follows:



**32** Play the game and see what pressing the buttons does.

**33** Now all we need is some way to get back to the main menu from anywhere in the game. There is one script that is in every level: **GameManager**. Find it in your **Project** window to open it.

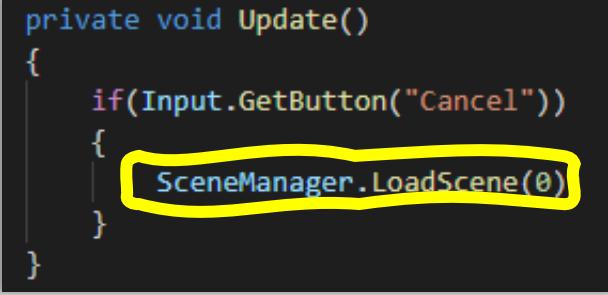
**34** In the **private void Update()** function, add a conditional to check if the cancel button is being pressed:



```
C# GameManager.cs
Assets > Scripts > C# GameManager.cs > GameManager
29
30 0 references
31 private void Update()
32 {
33 if(Input.GetButton("Cancel"))
34 {
35 }
36 }
37
```

In Edit > Project Settings, the cancel button is linked to the esc key.

**35** Inside the conditional, we'll use the SceneManager to load the main menu:



```
private void Update()
{
 if(Input.GetButton("Cancel"))
 {
 SceneManager.LoadScene(0)
 }
}
```

You can also use the scene's name, "Main," but loading 0 will always take you to the opening scene because it's the first scene in the build.

Save and close your script.

**36** Play the game. Make sure you can load the right level with the right button and that you can press esc to go back to the main menu at any time.

# Prove Yourself

## Get Started

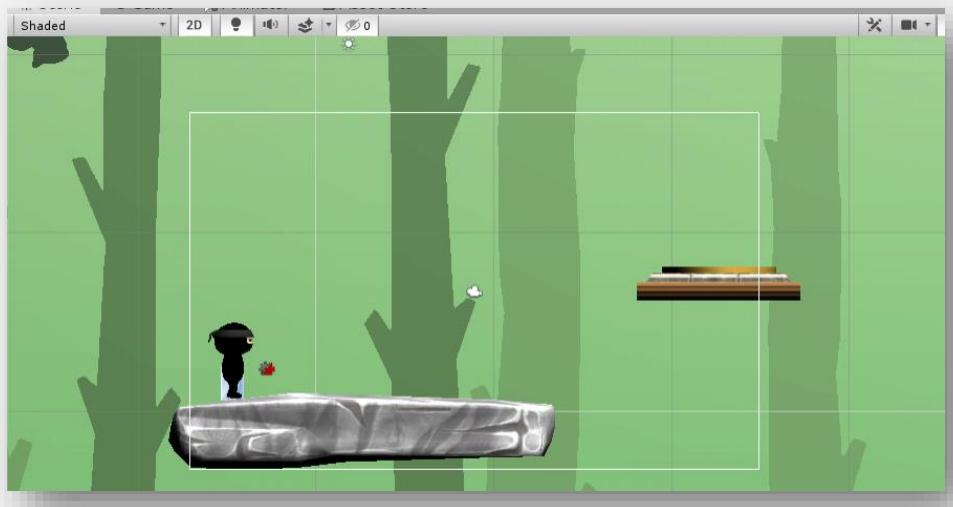
- Continue using the Unity project file used for the lesson.

## Task

Build a new world! After setting up the levels and the menu, all the assets have been included for you to make an adventure on a third planet.

Some things to look out for when building this level:

1. Make sure all the platforms are in the same position on Z. Toggle between 2D and 3D to double check if any platforms don't line up.
2. The actual size of the level is determined by the GameScene quad in the Background Layers Game Object. If you want to make a larger level, make sure to resize the GameScene quad to contain everything.
3. There are two objects, Edge and Edge (1) that keep the player from walking off the sides of the level. Locate these and move them if you change the size of the GameScene quad.
4. You can always use ctrl-D to duplicate a scene. This is an easy way to create a mini-level (like Level1\_A) or to revisit an old level with new parameters (like Level1\_B).
5. Don't feel that you have to stop at 3 levels. Add as many levels as you like!



## Activity 15

# Scavenger Hunt Deluxe

While not every type of game may need it, especially Arcade-style games that are trying to imitate the classic simplicity of games from the '80's and earlier, many modern games have dialogue systems.

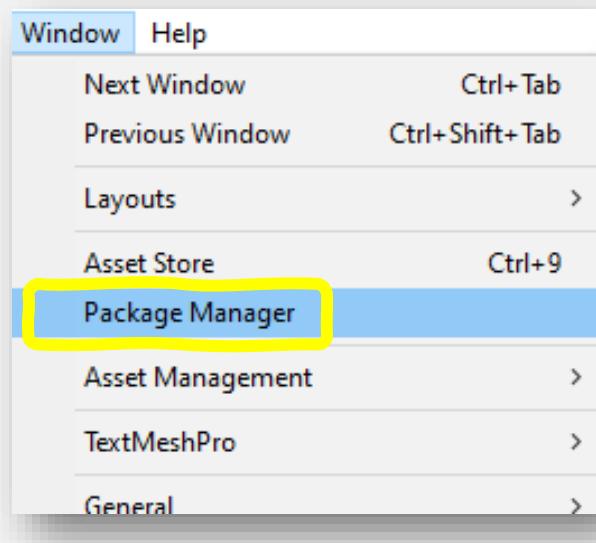
Your mission: Design a simple dialogue system for the player to be able to communicate with a character in the Scavenger Hunt environment. This character will have some situational awareness of what the player has done to trigger different responses in a dialogue tree.

Your avatar will be able to go around the scene, collecting coins. Playtest the game, do you notice anything different? In this version of the game, you need to find the right coin. But which coin? There's a non-player character (NPC) in the scene, but he's not talking. Let's fix that.

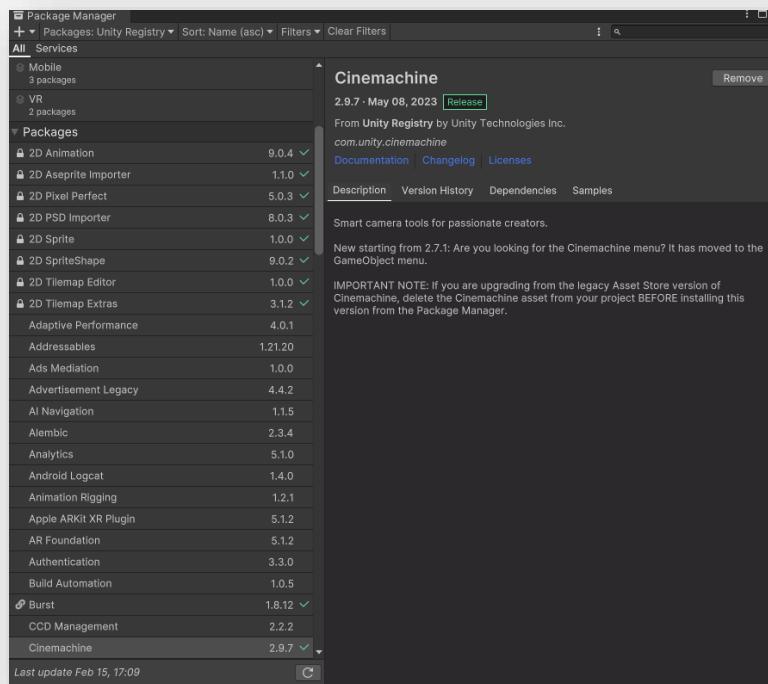


- 
- 1 Start a new Unity Project and name it **YOUR INITIALS - Scavenger Hunt Deluxe**. Select **2D core**.
  - 2 We will be using Cinemachine to control our camera. This will help us create awesome camera shots and angles for our game.  
Go ahead and open **Window > Package Manager**.

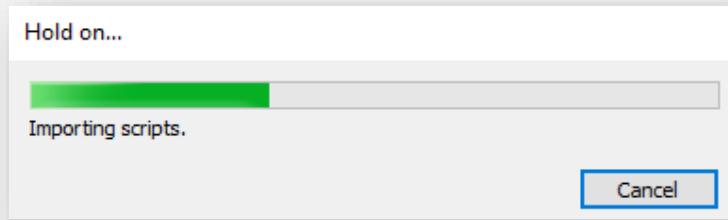
*Be patient if it doesn't open right away. It might take a minute to load.*



- 3 Find **Cinemachine** in the Unity registry and click **Install** in the top right of the window.

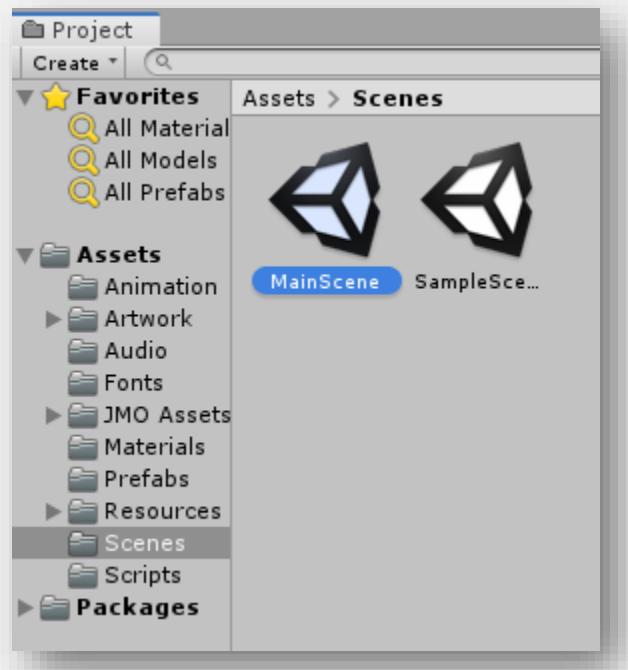


- 4** We've created a starter pack to give you a head start! To use it, import the **ScavengerHunt\_Ninja.unitypackage** by going to **Assets > Import Package > Custom Package > All > Import**.

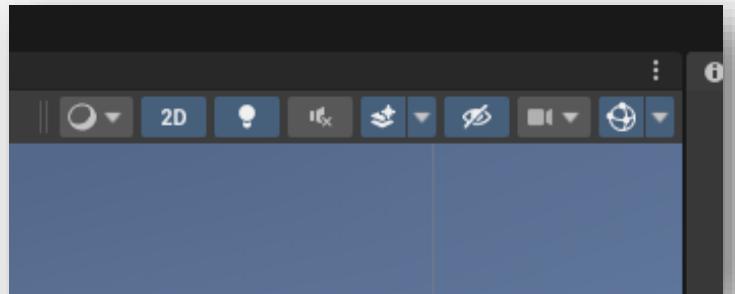


- 5** To open the starter package, double-click on the scene. You can find this in the **Project** tab under **Assets > Scenes > MainScene**.

If your camera does not look right in the Game view, click the **Assets** tab then select **Reimport All**.



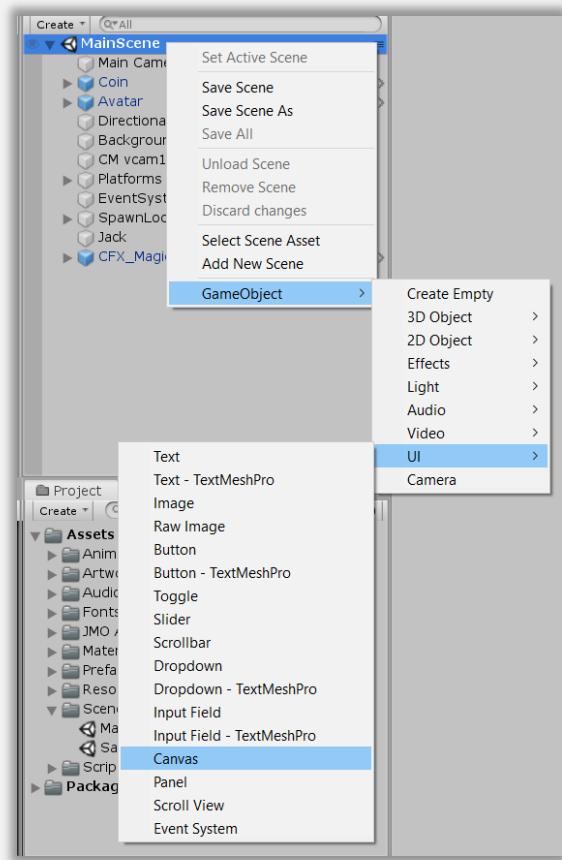
- 6** Make sure that the game has been stopped. Since we're working on the interface, we won't need full 3D. Make sure that the 2D box in the Scene window is active.



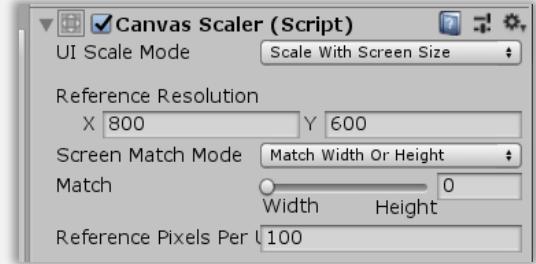
**7** To help the jack-o'-lantern find its voice, we are creating a *dialogue* system, which will facilitate *dialogue* between two characters.

The spelling of dialogue and dialog might be confusing – dialogue is used specifically in computer science to refer to a window presenting information.

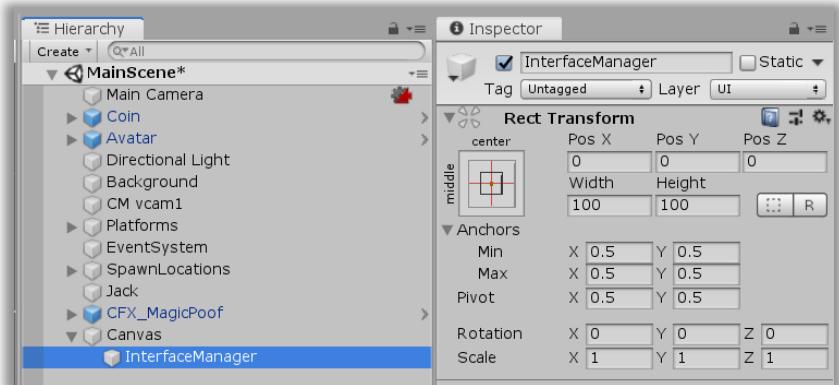
This will be part of the User Interface, so start by adding a **Canvas** object to the **Hierarchy** from the **GameObject** > **UI** menu.



**8** Select the **Canvas** **GameObject**. In the **Inspector**, find **Canvas Scaler** and change the **UI Scale Mode** to “**Scale With Screen Size**”. This will help make sure that the canvas appears the same on all screens, no matter the resolution.



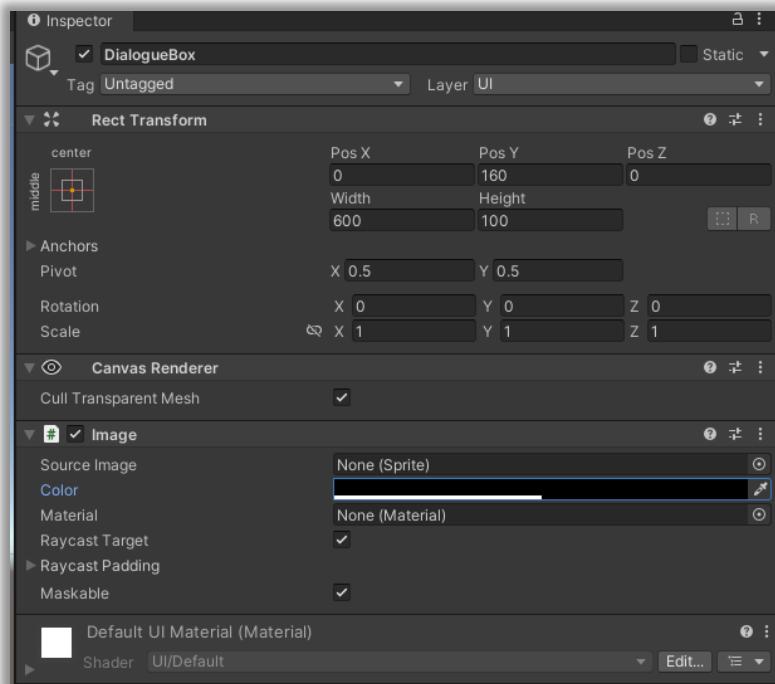
**9** Inside the **Canvas**, add an **Empty** **GameObject** and rename it as “**InterfaceManager**”.



**10**

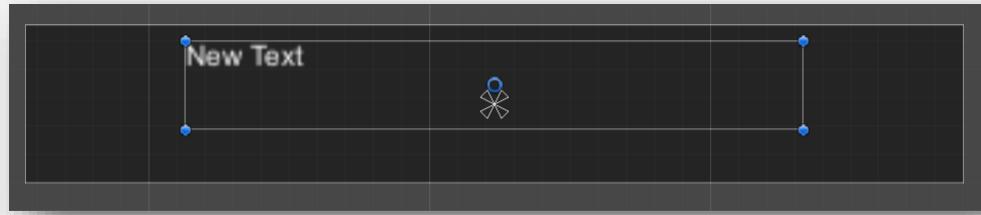
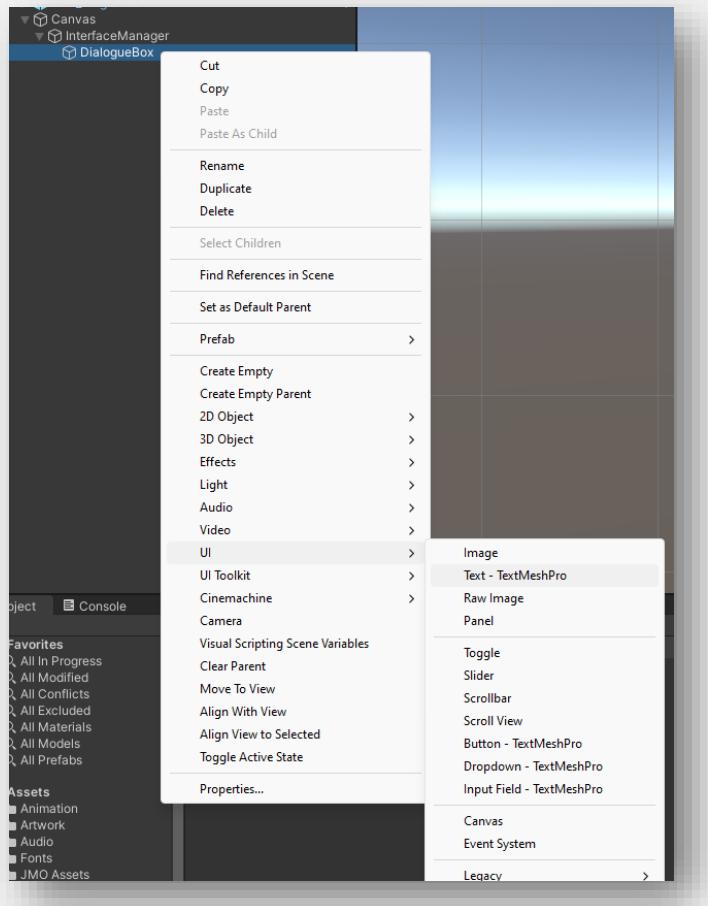
For our box, add an **UI > Image** object to the **InterfaceManager** that you just created. Rename it as “**DialogueBox**”. Change the width and height and move it to the top center of the Canvas. We used the dimensions of 600 units wide and 100 units high in our game.

We don't need to have an actual image for this box, we just want to change the color to transparent black.  
Inside the **Image** component in the **Inspector**, click on **color**. Select black and change the **alpha** to 50%.

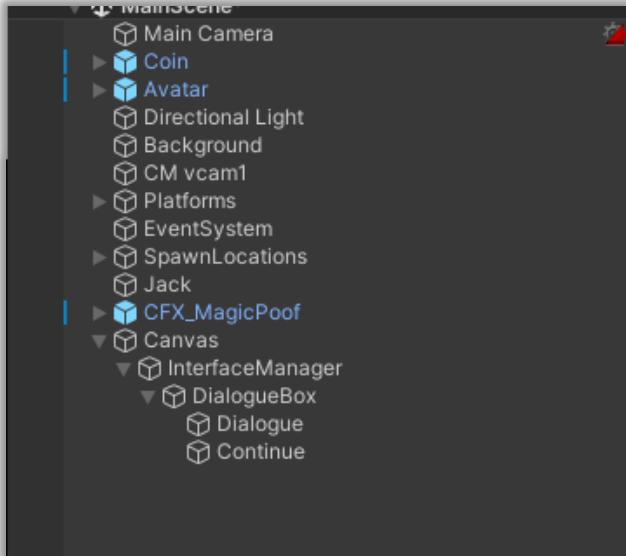


# 11

The dialogue itself will be text. Inside the **DialogueBox**, add a **UI > Text - TextMeshPro** object. Rename it as “**Dialogue**”. Center it inside the **DialogueBox** and leave some space on both ends for images. We used dimensions of 395 units wide and 56 units high. In the **Text** component in the **Inspector**, make sure that the **font** is Arial. Set the **font size** to 18. Make the **color** white. We’ll be using a script to change the text.

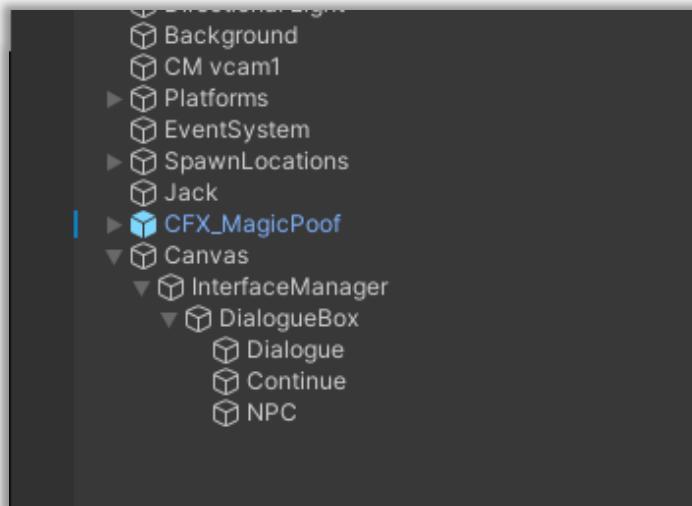


**12** Add another **Text** object to the dialogue box. Change the name to “**Continue**” and move it to the lower-right of the **DialogueBox**. In the **Text** component, change the text to “**Press Enter to Continue**” and change the **font size** to 14 and make sure the **color** is white. Resize the rectangle for this object to fit the text.



**13** In this game, Jack is the only one speaking. But if we added another character or wanted to have the player say something back to Jack, it's helpful to have a reminder of who is speaking.

Select the **DialogueBox** and add a **UI > Image** object. Change the name of the **Image** object to “**NPC**”. Select “**JackHead0**” (from **Assets > Artwork > Sprites**) for the **Source Image** and adjust it so that it fits to the left of the Dialogue text in the DialogueBox.

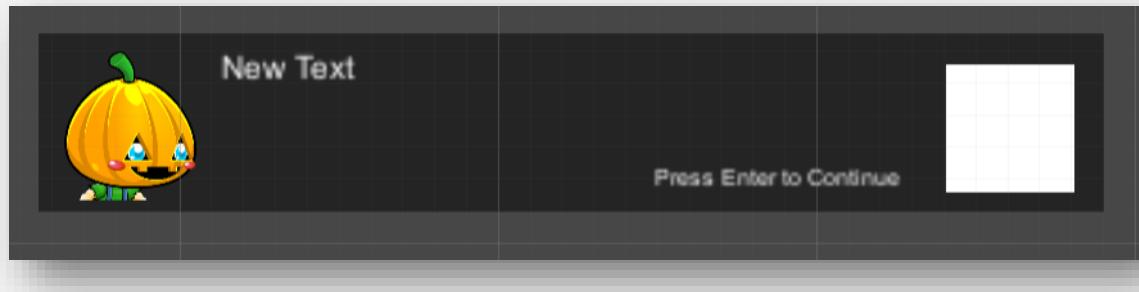


**14**

Remember, Jack is going to ask for you to find a specific coin. Add a **UI > Image** object to the dialogue box and name it, “**Item**”.

Make the image a bit smaller (ours is 70 units wide by 70 units high) and place it on the right side of the dialogue box, in the opposite location of Jack’s head.

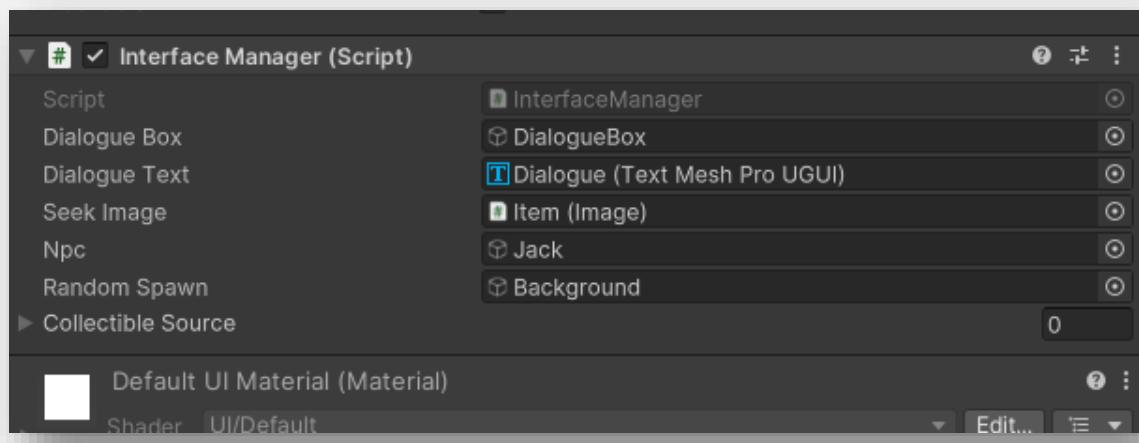
For now, do not change any of the other Item Image properties.



**15**

Save your project. Select the **InterfaceManager GameObject** and add the **InterfaceManager** script to it. The script has a few slots to fill. Fill the slots with these:

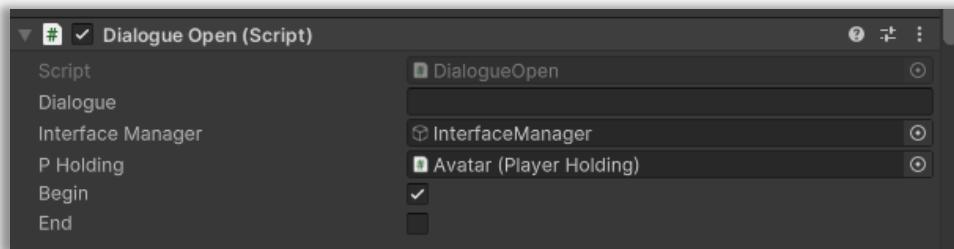
- Dialogue Box: DialogueBox
- Dialogue Text: Dialogue
- Seek Image: Item
- NPC: Jack
- Random Spawn: Background



**16**

In the Hierarchy, select the character, "Jack". If Necessary, use the "F" key to focus on him in the scene. You may notice that Jack has a box collider that extends past the sides of the object. This is so that Jack being triggered occurs before the avatar reaches him.

Jack has a script attached called "Dialogue Open". There is a slot in the **Dialogue Open** component for the **InterfaceManager GameObject**. Link **InterfaceManager** object to the slot for it in the **Dialogue Open** component, then open the **DialogueOpen** script. All we do when we *Translate* an object in Unity or in the real world is move it from one location to another.



**17**

This script serves two functions. Its main purpose is to control what Jack says about the coin you're looking for. Also, when the game begins, it randomly decides what that coin is.

```
public class DialogueOpen : MonoBehaviour
{
 public string dialogue;
 public GameObject interfaceManager;
 public PlayerHolding pHolding;
 public bool begin = true;
 public bool end = false;
 private string[] collectibles;
 private int clue;

 private AudioSource greeting;

 // Start is called before the first frame update
 void Start()
 {
 greeting = GetComponent<AudioSource>();
 collectibles = new string[] { "film", "balloons", "life saver", "bull's eye", "pipe", "key", "fish", "birdhouse", "red airhorn", "magic hat" };
 createClue();
 }

 public void createClue()
 {
 clue = Random.Range(0, 9);
 }

 private void OnTriggerEnter2D(Collider2D other)
 {
 if (!begin && pHolding.Verify())
 {
 checkClue();
 }
 greeting.Play(0);
 }

 private void checkClue()
 {
 if (pHolding.holdValue == clue)
 {
 end = true;
 }
 else
 }
```

**18**

Look at the **Start** function. It sets up the  **AudioSource** for Jack's greeting sound and the names for all ten coins in a single array. Once that is done, it calls the **createClue** function.

The **createClue** function simply picks a random number between 0 and 9 and assigns it to the **clue** variable. After that, nothing happens.

What we need to do is have Jack now tell the player what coin they're looking for. To do that, we'll call another function. At the end of the **createClue** function, add this:

```
searchDialogue();
```

```
public void createClue()
{
 clue = Random.Range(0, 9);
 searchDialogue();
}
```

**19**

The next step is to create the **searchDialogue** function. After the **createClue** function, add this:

```
public void searchDialogue()
{
 dialogue = "Hi! Can you help me find my " +
collectibles[clue] + "?";
}
```

```
1 reference
public void searchDialogue()
{
 dialogue = "Hi! Can you help me find my " + collectibles[clue] + "?";
}
```

What this does is create a string for the **dialogue** variable that includes the name of the random clue from the **collectibles** array.

The next step is to send this information to the **InterfaceManager**.

**20**

We don't want to send the dialogue to the interface manager all the time, only when the player has moved next to Jack. We already have a trigger set up to play a sound when the player gets there as well as a conditional that will be used later to check if the player has the right coin:

We'll use this function to send information to the **InterfaceManager**. Add this line:

```
interfaceManager.GetComponent<InterfaceManager>().ShowBox(dialogue, clue);
```

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D other)
{
 if (!begin && pHolding.Verify())
 {
 checkClue();
 }
 greeting.Play(0);
 interfaceManager.GetComponent<InterfaceManager>().ShowBox(dialogue, clue);
}
```

This code finds the **InterfaceManager** component of the **interfaceManager GameObject** and runs the **ShowBox** function in that script component, sending it the values of dialogue and the clue.

But to make it work, there are a couple of other things we need to do.

**Save** the script and go back to Unity.

**21**

Play the game. The dialogue box still doesn't do anything. Also, we only want to see the box when Jack has something to say to us. These are handled in the **InterfaceManager** script.

**Stop** the game.

**22**

Select and open the **InterfaceManager** script.

When the game starts, we want to hide the **dialogueBox**.

In Unity, we can deactivate the object with the command of `SetActive(false);`.

In the Start function, add this:

```
dialogueBox.SetActive(false);
```

```
void Start()
{
 dialogueBox.SetActive(false);
}
```

Now the dialogue box is hidden until we need it. Can you figure out how to show it?

**23**

Remember, in the **DialogueOpen** script, we called the **ShowBox** function in the **InterfaceManager** script when the player gets close to Jack. In the **InterfaceManager** script, the **ShowBox** function is empty. This is where we can show the dialogue box by making **SetActive** true:

```
dialogueBox.SetActive(true);
```

```
public void ShowBox(string dialogue, int item)
{
 dialogueBox.SetActive(true);

 if (npc.GetComponent<DialogueOpen>().begin)
 {
 scatterCoins();
 }
}
```

After adding the code, save the script and give it a try. The dialogue box appears when you get close to Jack, and then it just stays there. We need to make it go away now!

**24**

Our dialogue box has the words “Press Enter to Continue”. In the **Input Project** settings, Unity is using the enter key for Submit. So, we just need to check for the submit input and use it to hide the box.

Find the **Update** function in the **InterfaceManager** script. Add this:

```
if (Input.GetButton("Submit"))
{
 dialogueBox.SetActive(false);
}
```

```
void Update()
{
 if (Input.GetButton("Submit"))
 {
 dialogueBox.SetActive(false);
 }
}
```

**25**

Play the game again. Now you can control when the dialogue box is seen and not seen.

Stop the game. We don’t need to check for the Input when the box isn’t active. So, modify the conditional this way:

```
if (Input.GetButton("Submit") &&
dialogueBox.activeInHierarchy)
```

```
void Update()
{
 if (Input.GetButton("Submit") && dialogueBox.activeInHierarchy)
 {
 dialogueBox.SetActive(false);
 }
}
```

**26**

We also want to change the text and picture in the dialogue box using the information sent over from the DialogueOpen script. To update the text, we simply need to change the text value of the **dialogueText** object.

Add this to the ShowBox function:

```
dialogueText.text = dialogue;
```

```
public void ShowBox(string dialogue, int item)
{
 dialogueBox.SetActive(true);
 dialogueText.text = dialogue;

 if (npc.GetComponent<DialogueOpen>().begin)
 {
 scatterCoins();
 }
}
```

Remember the dialogue string is sent over from **DialogueOpen**.

**27**

Updating the image with a new sprite is a little bit trickier.

We need to identify the **sprite** component of the **seekImage** object (our item) and update it with the sprite that matches the number selected in **DialogueOpen**:

```
seekImage.GetComponent<Image>().sprite =
collectibleSource[item];
```

```
1 reference
public void ShowBox(string dialogue, int item)
{
 dialogueBox.SetActive(true);
 dialogueText.text = dialogue;
 seekImage.GetComponent<Image>().sprite = collectibleSource[item];

 if (npc.GetComponent<DialogueOpen>().begin)
 {
 scatterCoins();
 }
}
```

**28**

One of the advantages of using the scripts to change the text in the Dialogue Box is that we can customize what Jack says depending on what is going on in the game. After the player has spoken to Jack for the first time, a function is run to scatter the coins around the scene:

```
scatterCoins();
```

After that runs, it would be a good idea to have him say something else.

In the **DialogueOpen** script's **OnTriggerEnter2D** function is a conditional to see if:

- (a) The **begin** variable (which lets us know if the game has gone through the beginning steps) is false, and:
- (b) Has the player collected anything?

If so, then we will move to the **checkClue** function which looks for a match between what the player is holding and the actual clue.

```
private void OnTriggerEnter2D(Collider2D other)
{
 if (!begin && pHolding.Verify())
 {
 checkClue();
 }
}
```

If so, then the end variable is true, allowing us to end the game.

If there isn't a match, then we know that the player must keep looking.

```
private void checkClue()
{
 if (pHolding.holdValue == clue)
 {
 end = true;
 }
}
```

**29**

In the **checkClue** function, we can change the dialogue depending on whether the player has found the right clue or not.

If there is a match, we can let the player know by adding this:

```
dialogue = "You found my " + collectibles[clue] + "!\nHooray!";
```

Else, we will display this dialogue:

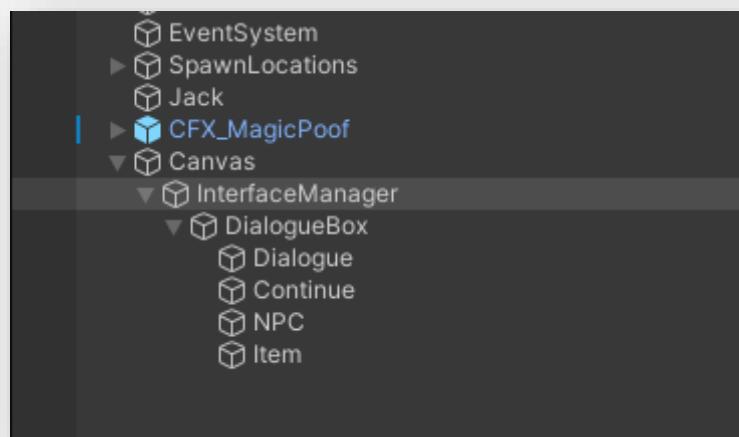
```
dialogue = "No, that's not my " + collectibles[clue] + ".";
```

```
private void checkClue()
{
 if (pHolding.holdValue == clue)
 {
 dialogue = "You found my " + collectibles[clue] + "!\nHooray!";
 end = true;
 }
 else
 {
 dialogue = "No, that's not my " + collectibles[clue] + ".";
 }
}
```

**30**

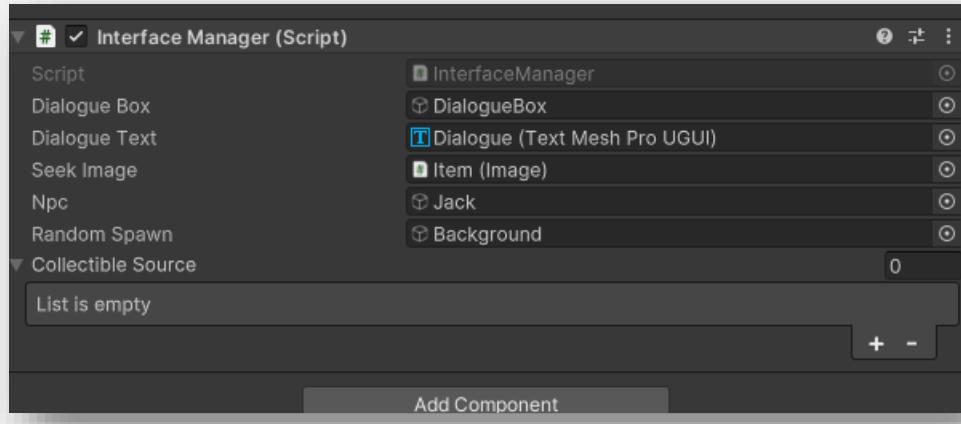
**Save** your script and return to Unity.

We need to tell Jack what items it can ask for. Select the **InterfaceManager** in the Hierarchy.

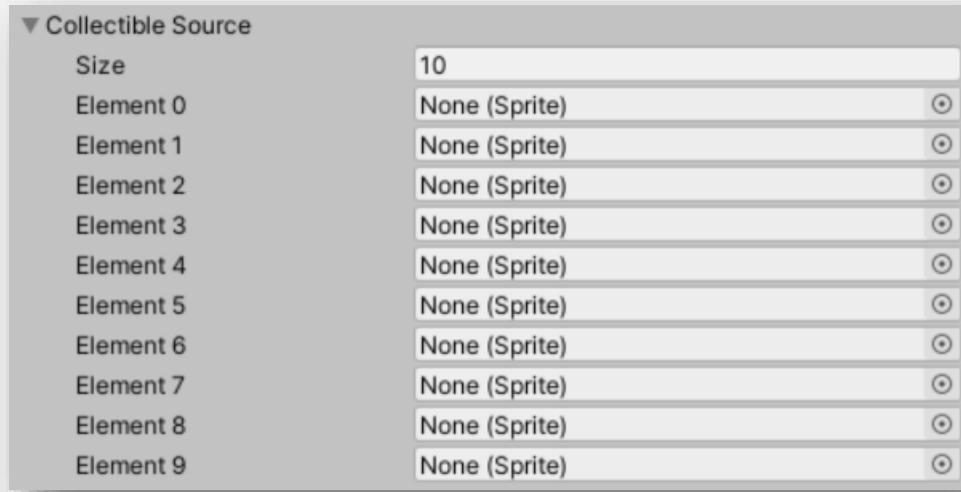


**31**

In the **Inspector** find the **Interface Manager (Script)** and expand the **Collectible Source** found at the bottom on the component by clicking on the arrow.

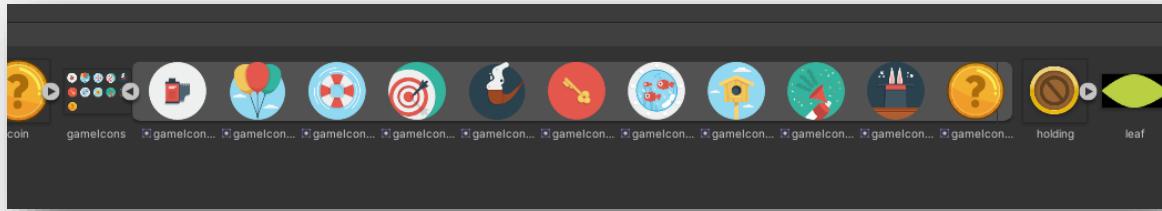


You should see a **Size**, change the size to **10** because we have 10 items that Jack can ask for.

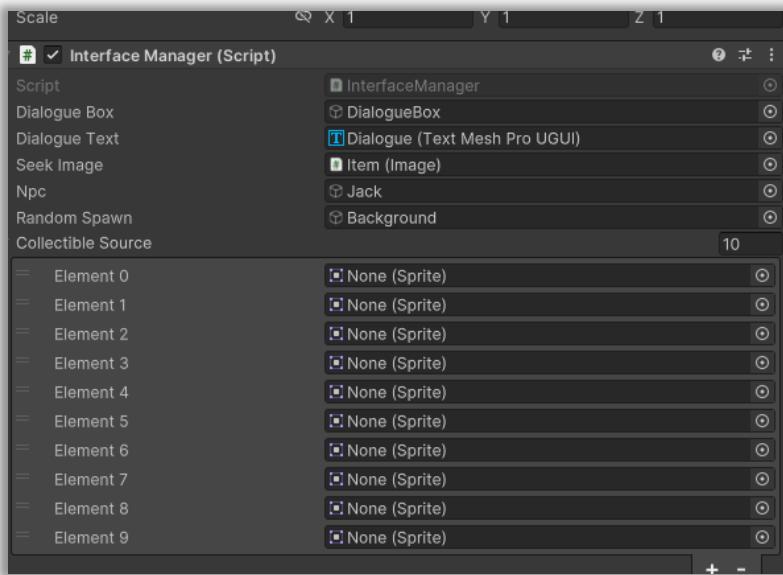


**32**

We need to add our items into each of these elements. Go into the **Artwork folder** in the Projects tab. Then expand the **gamelicons** by clicking on the arrow.

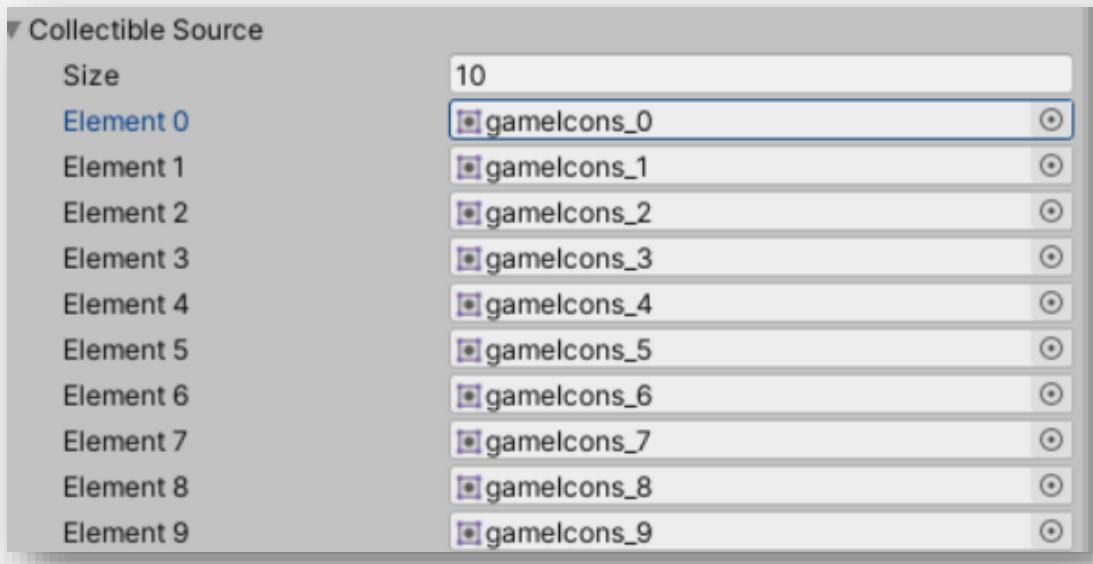


Drag each game icon into an Element slot.



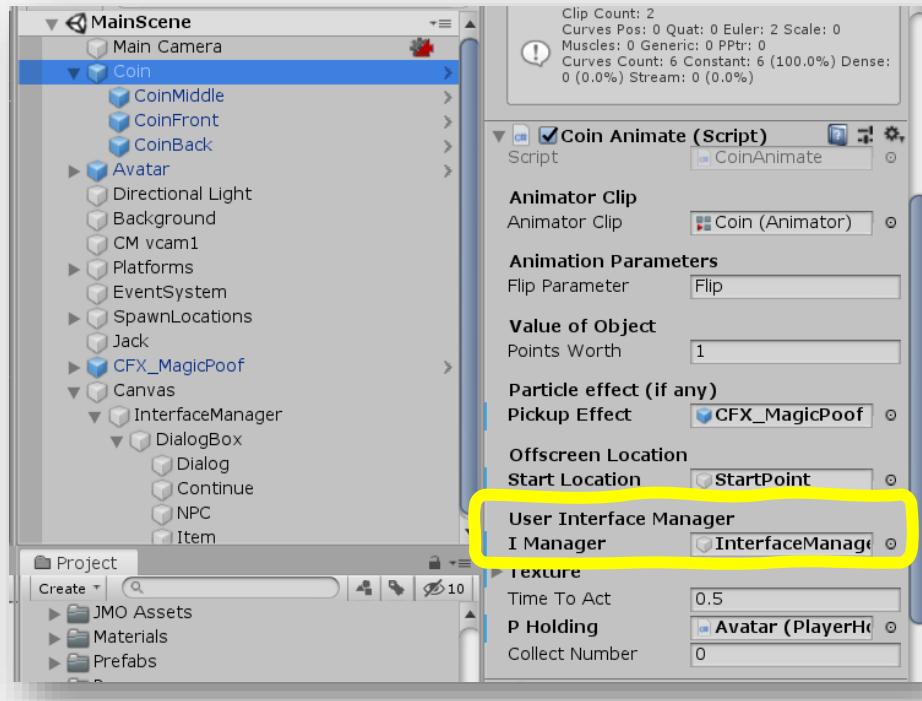
**33**

Your element components should look something similar to this:



**34** To get the coin collection to work, we'll have to make sure the coin can communicate with the **InterfaceManager**. Select the **Coin GameObject** in the hierarchy and find the component for **Coin Animate** in the **Inspector**.

Make sure that the slot for **User Interface Manager (I Manager)** is linked to the **Interface Manager Object**.



**35** Go ahead and **play** the game.

You should be able to get Jack to tell you what to look for and he will tell you if you're right or wrong.

The only problem is that we don't know which coin the player has collected.

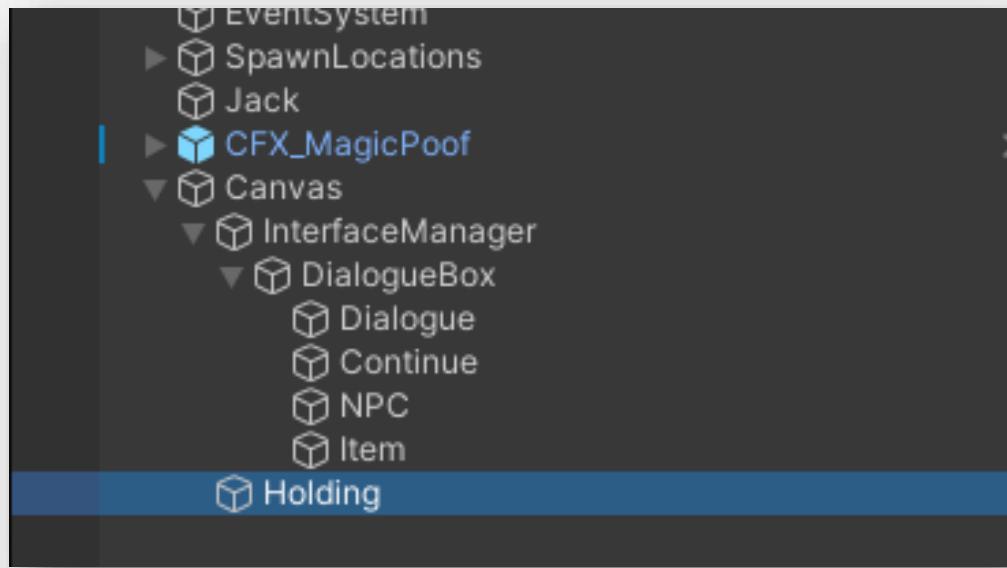
**36**

When playing the game, the player can only hold one coin at a time, so when another coin is picked up, they drop the current coin.

This makes it difficult to know what the player is carrying.

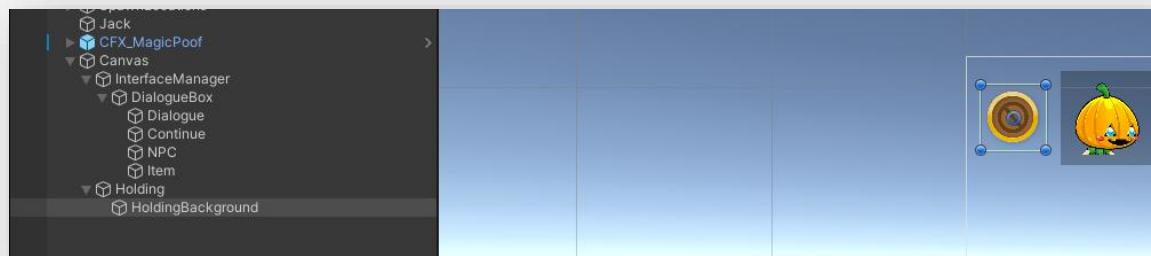
For that, we should add an inventory to the User Interface.

In Unity, find the **Canvas** object in the Hierarchy. Add an **empty GameObject** to the **Canvas** (not the InterfaceManager), and name it "**Holding**".



**37**

Add **UI > Image** to the **Holding** object and name it "**HoldingBackground**". In the **Image** component in the Inspector, change the **Source Image** to the **holding** sprite. Move it to the upper-left corner.



## 38 Add an **empty GameObject** to **Holding** and name it "**HoldingItem**".

Add an **UI > Image** object to the newly made **HoldingItem** object. Adjust it so that it fits in the center of the **HoldingBackground**. Ours is 42 units high, and 42 units wide.



## 39 Once more, open the **InterfaceManager** script. We need variables for this new image as well as to tell if there's anything there at all. Add these variables:

```
public Image collectible;
public GameObject showSprite;
```

```
Unity Script | 2 references
public class InterfaceManager : MonoBehaviour
{
 public GameObject dialogueBox;
 public TMP_Text dialogueText;
 public Image seekImage;
 public GameObject npc;
 public GameObject randomSpawn;

 public Image collectible;
 public GameObject showSprite;

 [SerializeField]
 private Sprite[] collectibleSource;

 // Start is called before the first frame update
 Unity Message | 0 references
```

**40**

When the game starts, we want the inventory to be empty. Add this to the `Start()` function.

```
showSprite.SetActive(false);
```

```
void Start()
{
 dialogueBox.SetActive(false);
 showSprite.SetActive(false);
}
```

**41**

Insert this into the `CollectibleUpdate` function to update what the player is holding:

```
showSprite.SetActive(true);
collectible.GetComponent<Image>().sprite =
collectibleSource[item];

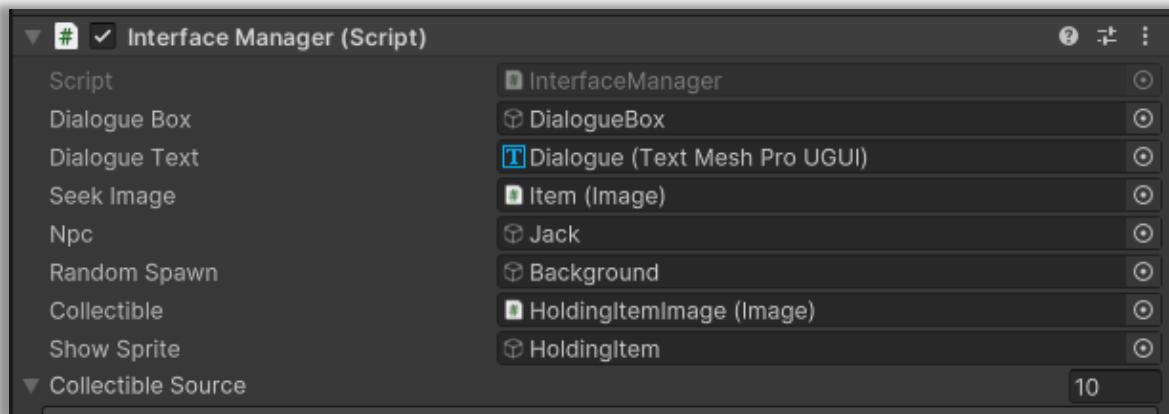
public void CollectibleUpdate(int item)
{
 showSprite.SetActive(true);
 collectible.GetComponent<Image>().sprite = collectibleSource[item];
}
```

**42**

Back in Unity, select the **InterfaceManager** and in the **InterfaceManager** script component, link these elements:

Collectible: HoldingItemImage

Show Sprite: HoldingItem



**43**

Click the **play** button and now give the game a try! What happens if you want to play again, though?

**Stop** the game.

**44**

In the **InterfaceManager** script, in the **Update** function, inside the conditional that checks for the “**Submit**” input, add in:

```
if (npc.GetComponent<DialogueOpen>().end)
{
 SceneManager.LoadScene(0);
}
```

```
void Update()
{
 if (Input.GetButton("Submit") && dialogueBox.activeInHierarchy)
 {
 dialogueBox.SetActive(false);

 if (npc.GetComponent<DialogueOpen>().end)
 {
 SceneManager.LoadScene(0);
 }
 }
}
```

If Jack tells the player that they have the right coin, and the player presses the Input “Submit” button, then the game starts over. Make sure to add the MainScene to the scenes in build settings.

# Prove Yourself

## Get Started

- You will be using the Scavenger Hunt Deluxe game that you just completed. Make sure you've completed the regular activity first.

## Task

In this Prove Yourself, you will have to create unique dialogue between the player and Jack for each item. **If the collectible is a specific item, then the dialogue used is unique to that item.** Try using a **switch** statement!



## Activity 16

# Food Frenzy Part 1

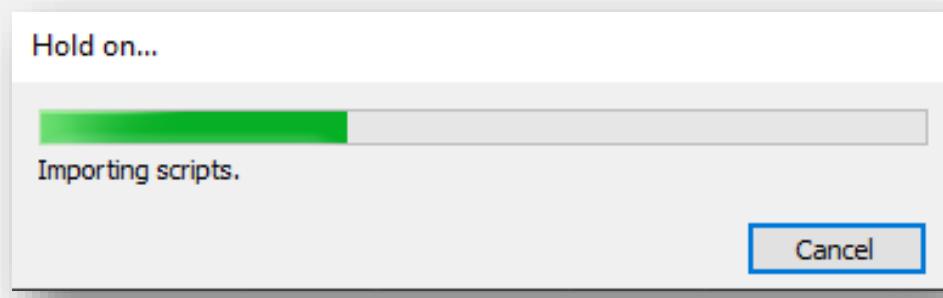
Well done! You've made it to the last game in Brown Belt. You've learned some pretty complicated game building skills, and you're finally on the last challenge to earn your belt.

Have you ever heard of a "Match-3" game? The objective is to match three or more of the objects in the game in a row. Doing so will remove the matched objects and give you points based on how many objects were matched. It's a simple type of game, but our focus is going to be on the interface that gives players information about the game state and what they can do.

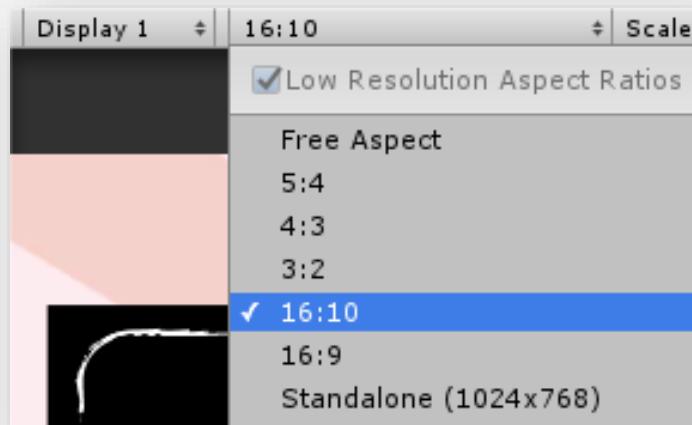
Objectives: Build an interface that guides the players through a game with multiple levels, game over conditions and states, and even animated interface elements.



- 
- 1 Start a new Unity Project and name it *YOUR INITIALS - Food Frenzy*. Select **3D core**.
  - 2 We've created a starter pack to give you a head start! To use it, import the **FoodFrenzy\_Ninja.unitypackage** file by going to **Assets > Import Package > Custom Package**, clicking **All**, and then **Import**.



Set the aspect ratio to 16:10.



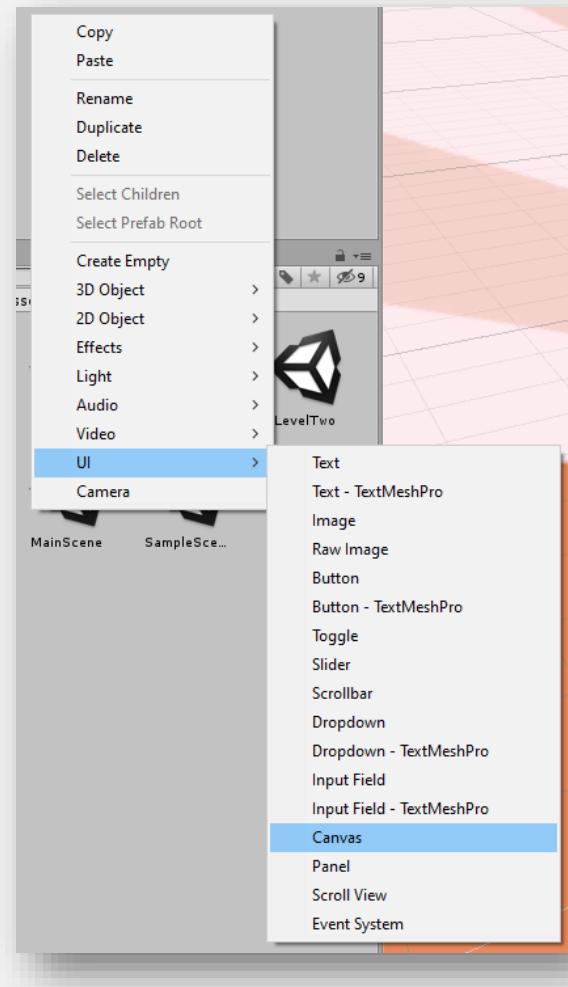
**3** Select the **LevelOne** scene. **Play** this scene.

*Play Instructions: Make a match by dragging icons from one cell in the grid to another. You can only drag an icon if the new position makes a row or column of three.*

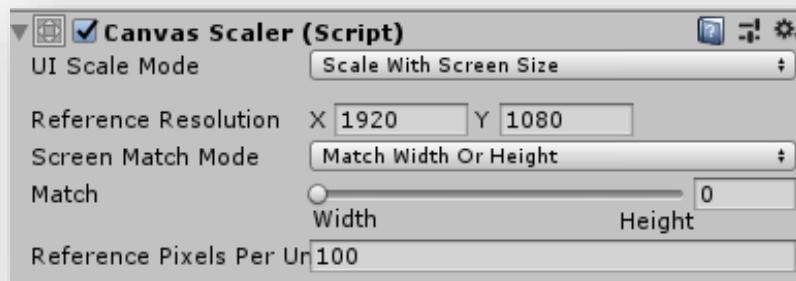


Most of the game is complete. All that's left to do is the User Interface (UI) where we update the player's score and keep track of the goals for the game.

- 4** The **User Interface (UI)** uses a **GameObject** called the **Canvas**. In the **Hierarchy**, add a new **GameObject** by right-clicking a blank space on the hierarchy, selecting the **UI** submenu, then **Canvas** from there.



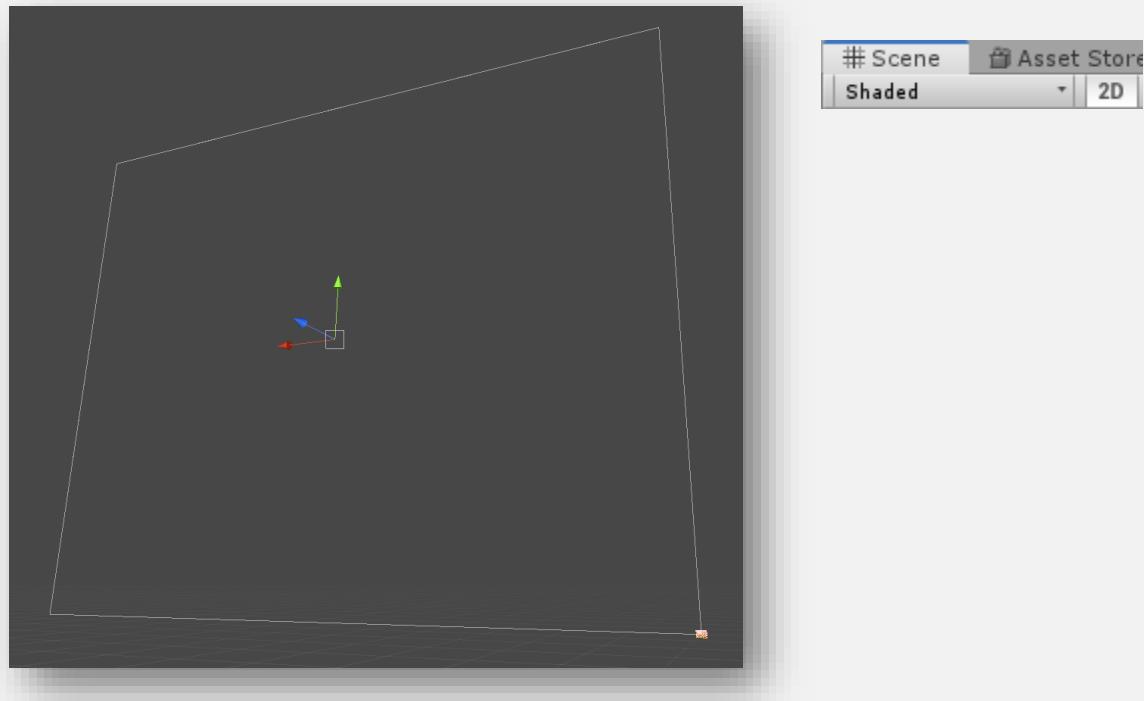
- 5** With the **Canvas** selected, find the **Canvas Scaler** component in the **Inspector**. Set the **UI Scale Mode** to "Scale With Screen Size". The **Reference Resolution** should be 1920 x 1080.



- 6** In the **Scene** window, the **Canvas** is much larger than the game itself. This is normal. You can check how things look by switching to the **Game** view at any time.

Remember, you can only edit in the **Scene** view and not the **Game** view. Since we are building a 2D game, don't forget to **change the view to 2D** when you're dragging and editing parts in the **Scene**.

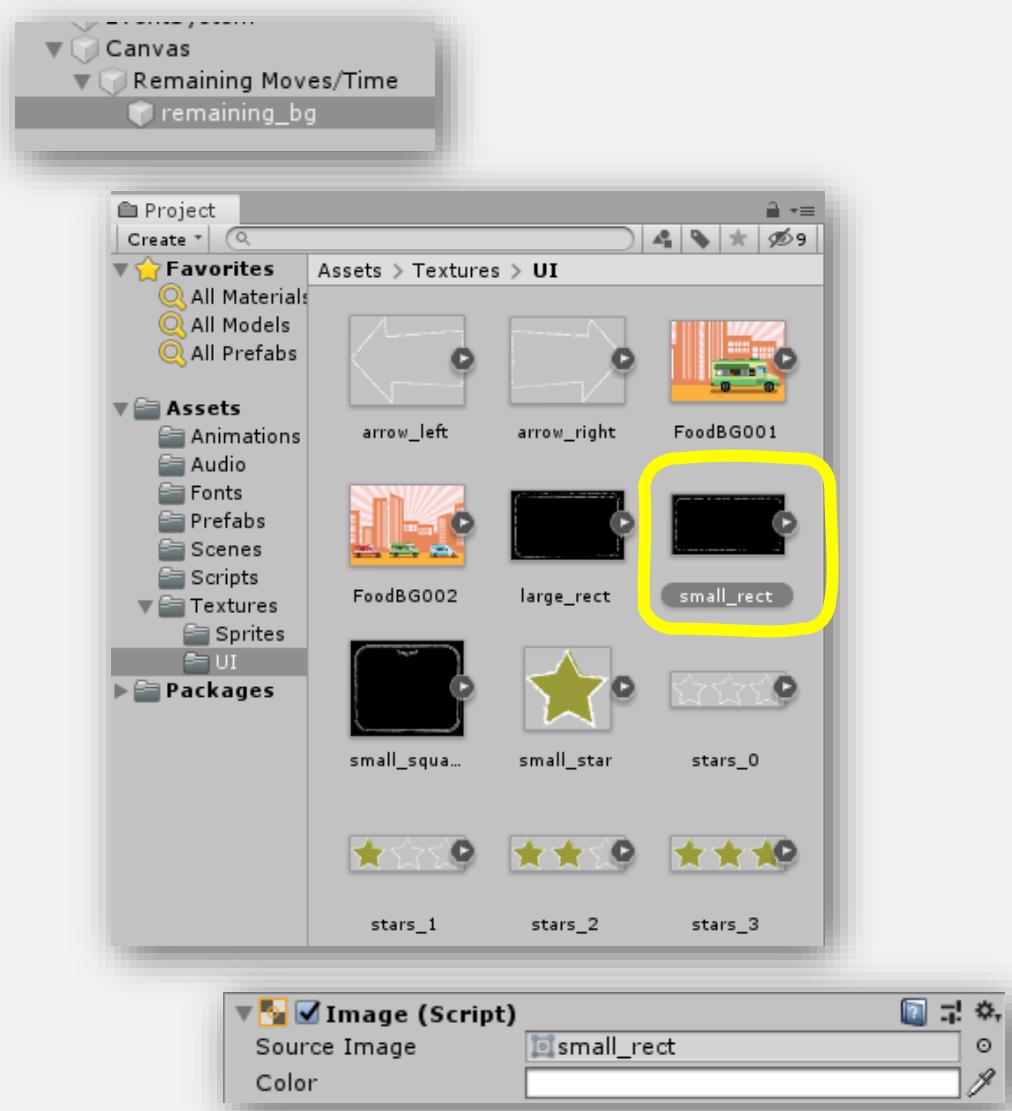
Here you can see the 3D view, notice the small rectangle on the bottom right of the canvas? That's the game itself!



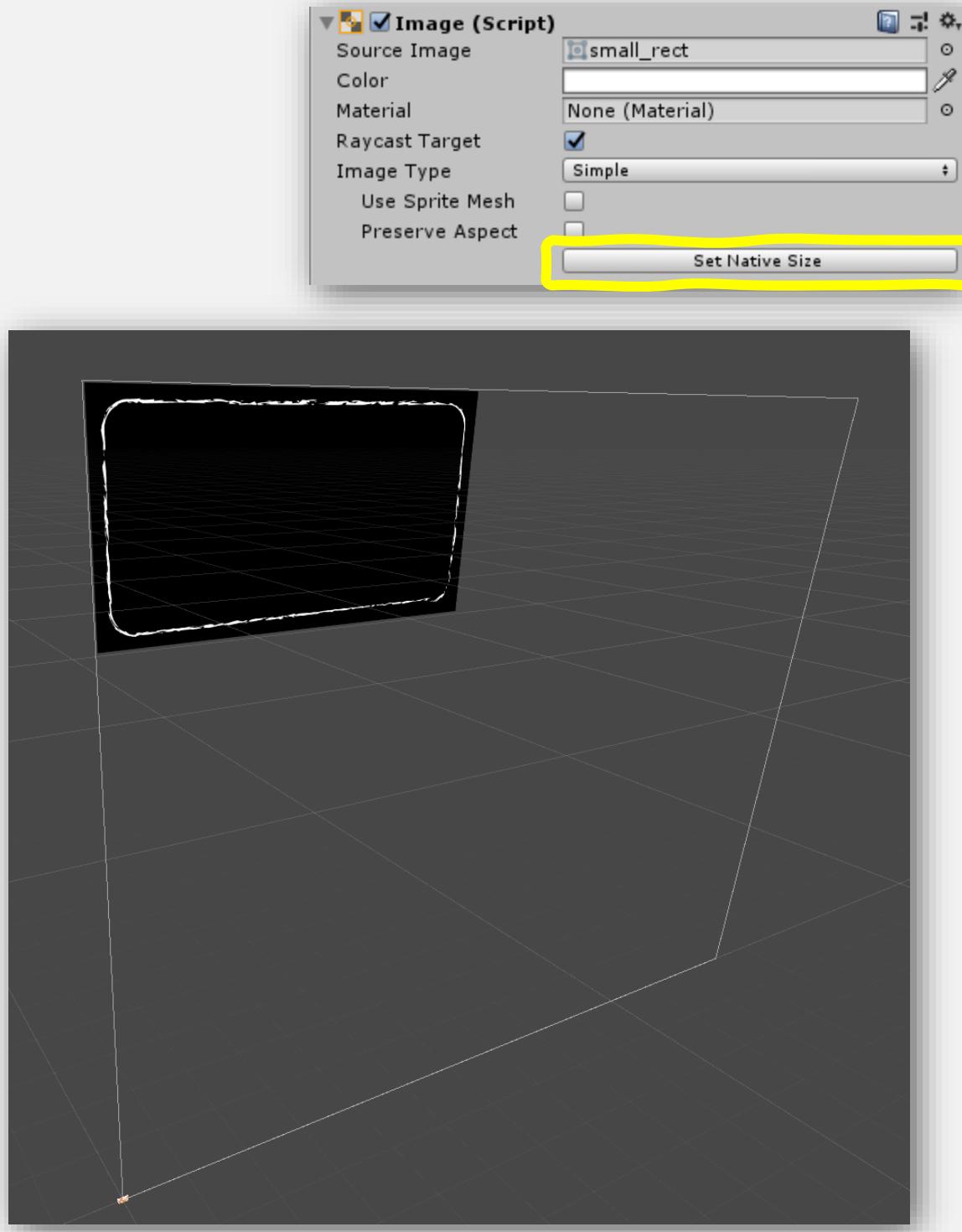
**7** We'll put our interface next to the game. There will be three parts – the **remaining moves**, the **target score** (or goal), and the **current score** along with 0 to 3 stars depending on how well the player has done.

Create an **empty GameObject** in the **Canvas** and give it the name of "**Remaining Moves/Time**".

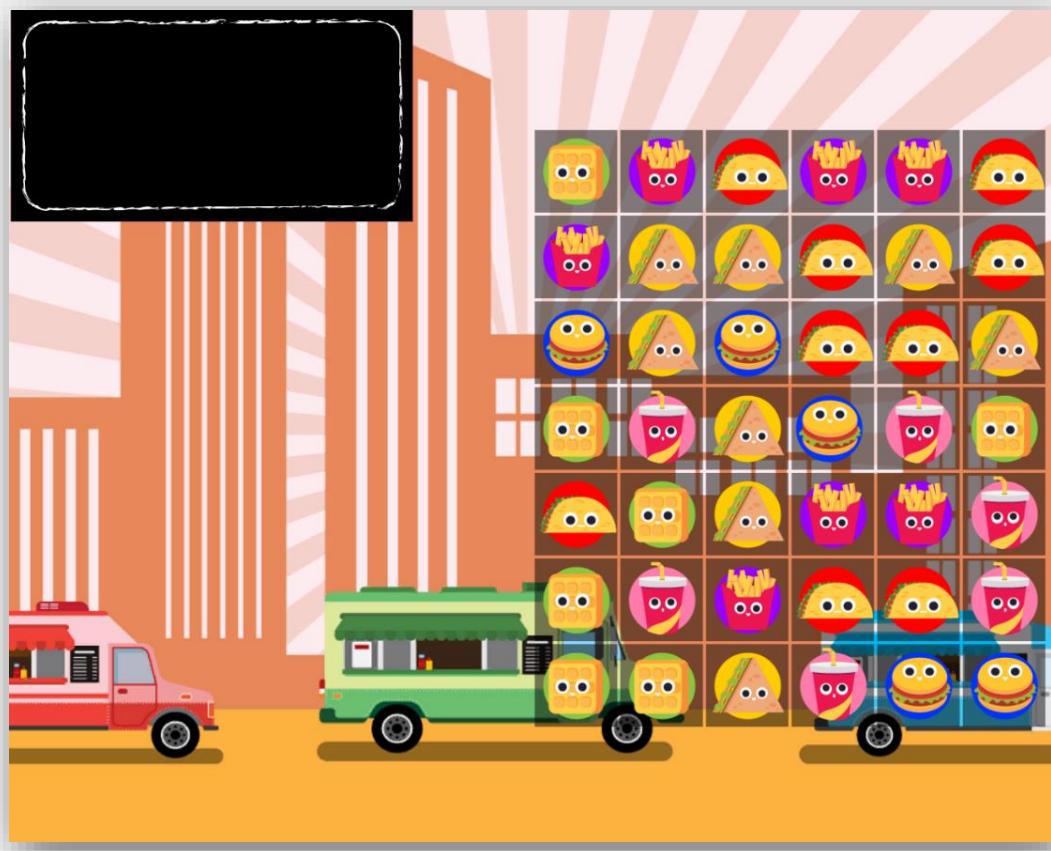
Inside this object, add a **UI > Image** object and call it "**remaining\_bg**". With the **remaining\_bg** object selected in the **Hierarchy**, change the **Source Image** in the **Image** component to the **small\_rect** sprite available in the **Assets > Textures > UI** folder.



- 8** In the **Image** component is the **Set Native Size** button. Go ahead and click the checkbox on to make sure that the image has the proper height to width ratio. Use the editing tools to move the image to the top-left corner of the **Canvas**.

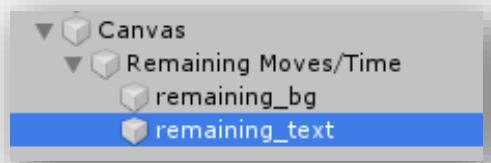


- 9** Play the game to make sure that the UI element you just added looks like it does below.



- 10** The next step is to add text. Add a **UI > Text** object to the **Remaining Moves/Time** object. To do this, right-click on **Remaining Moves/Time**, and navigate to **UI > Text**. Change the name to “**remaining\_text**”.

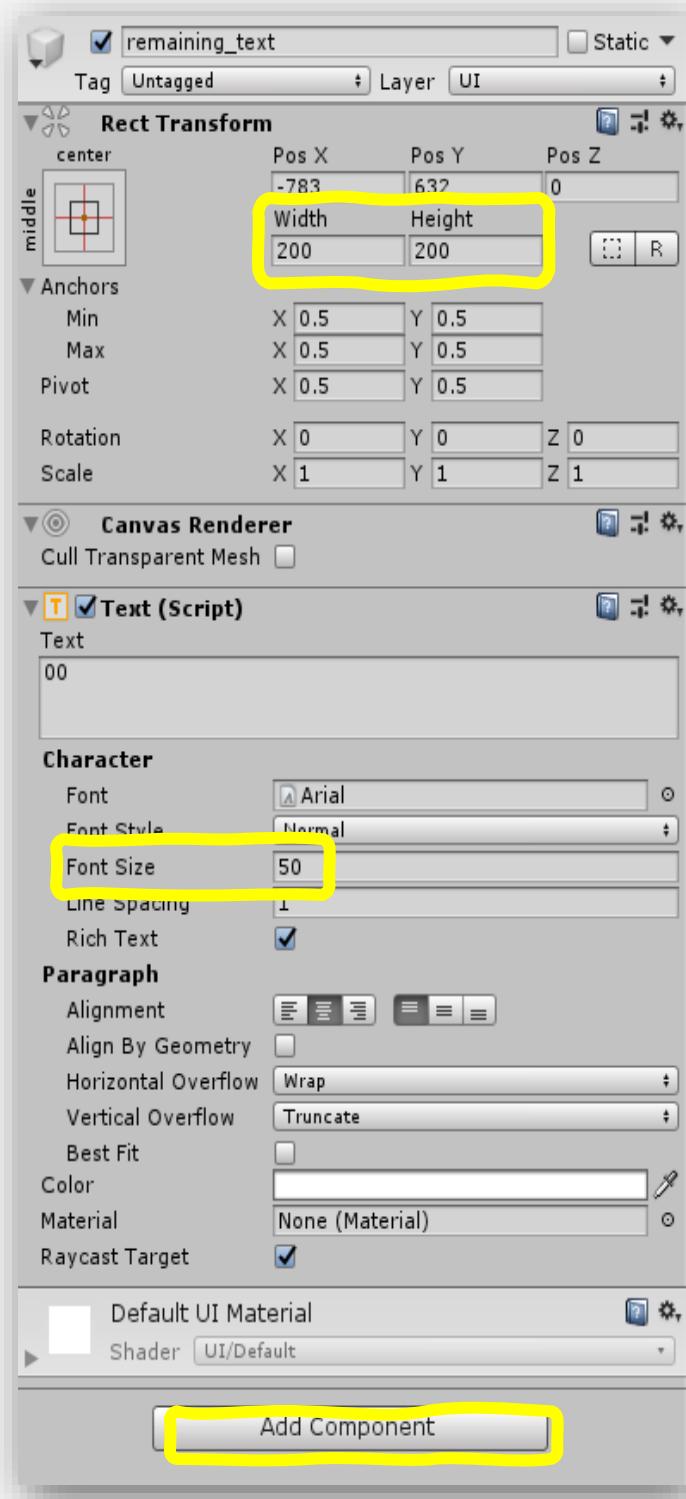
The default **font** of Arial is fine, but if you want to import a different font, you can do so.



Change the **font color** to white and change the **text** to “00” or any other 2-digit number. We’ll be changing the text in code, so we just want to get an idea of how much space we’ll need.

Adjust the **size** as necessary and change the **Paragraph > Alignment** to center.

You can see in our example that we've changed the font to 50 and increased the width and height of the Rect Transform to fit the text. You can customize it however you want by playing around with the text components. The object itself might need to have its size adjusted to account for the size of the font you choose.



**11**

Now make a **duplicate** of this text object and rename it as "**remaining\_subText**".

This will be a label for the number that we'll be putting in this object.

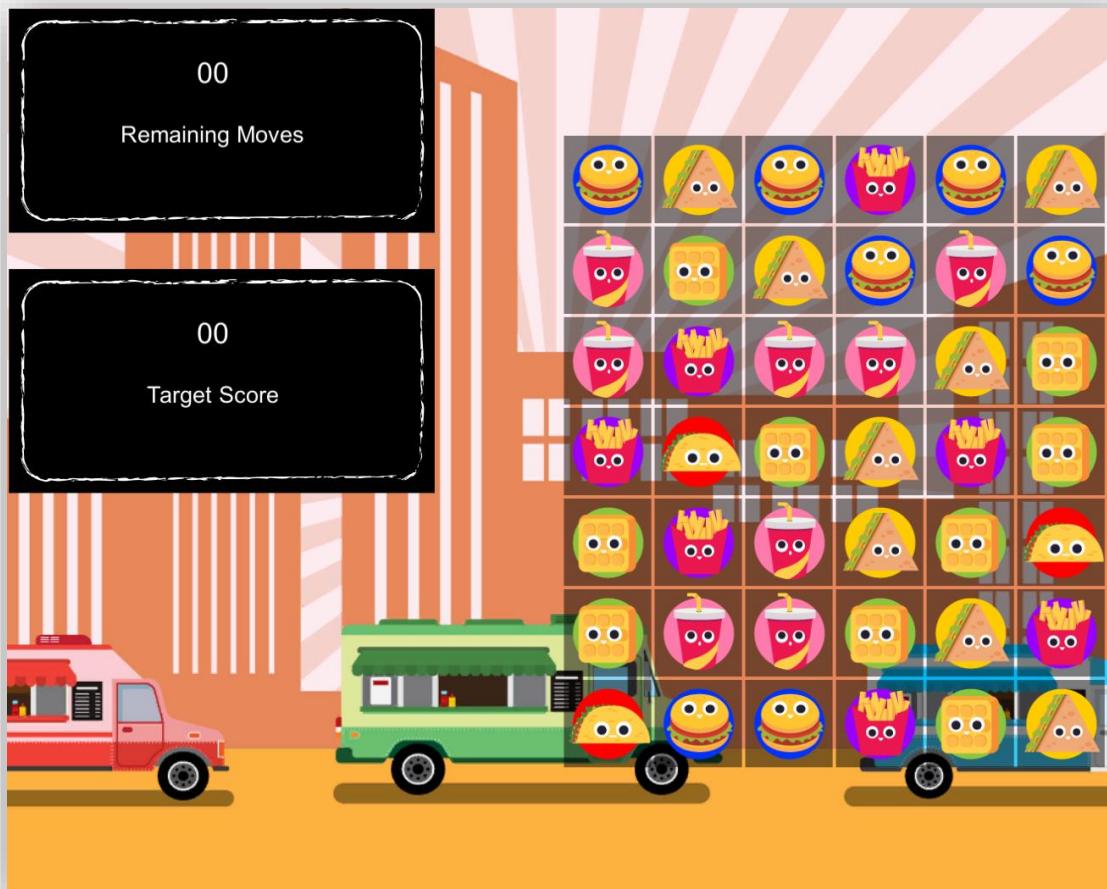
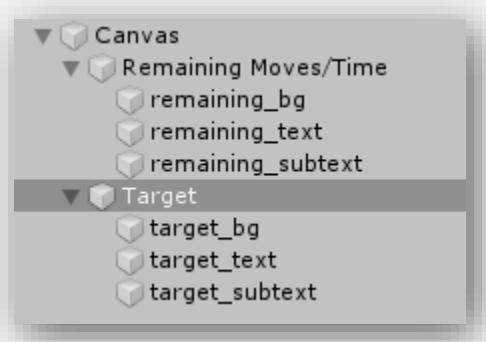
Change the **text** to "**Remaining Moves**" and resize it so that it is smaller and fits beneath the larger number.

As you can see in the image below, we've roughly centered both text objects.



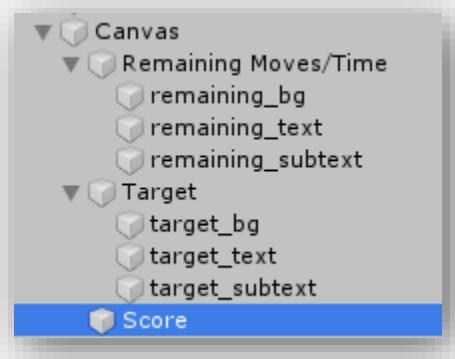
**12**

Make sure that you have the original **Remaining Moves/Time** object selected and make a **duplicate** of it. Move this duplicate so that it is below the Remaining Moves/Time object. Change the name of this new object to "**Target**" and rename the image and text objects inside it to "**target\_bg**", "**target\_text**" and "**target\_subtext**". Select **target\_subtext** and change the **Text** component so that the **text** is now "**Target Score**".



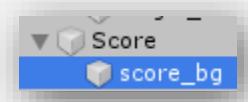
**13**

The next step is to have an object for the score. This will be different from the first two. Create a new **empty GameObject** in the **Canvas** and change the name to "**Score**".

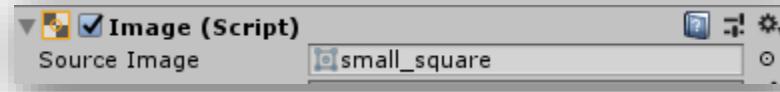


**14**

As you did with the other UI object, create a **UI > Image** inside **Score**. Change the name to "**score\_bg**".



In the **Inspector**, change the **Source Image** to **"small\_square"**.

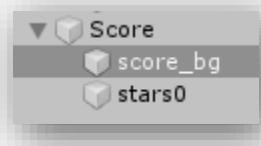


Adjust this so that it fits underneath the first two UI objects.

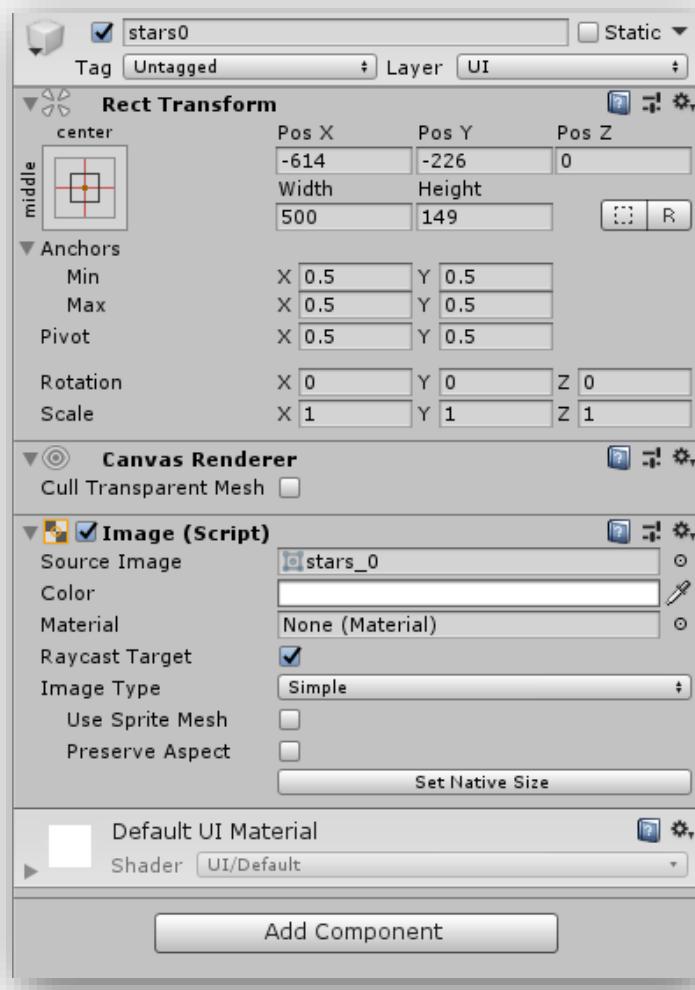


**15**

Add another **UI > Image** to the **Score** object. Name it “**stars0**” and change the **Source Image** to **stars\_0**. Adjust the **stars0** object and the background object until they look how you want them to.



Here's a possible way to set it up:



**16**

Now create three **duplicates** of **stars0**. Change the names of these duplicates to “**stars1**”, “**stars2**” and “**stars3**”. Change the **Source Image** in each of these duplicates to match the name (**stars\_1** for stars1, **stars\_2** for stars2 and so on). We will turn the stars on and off in code.



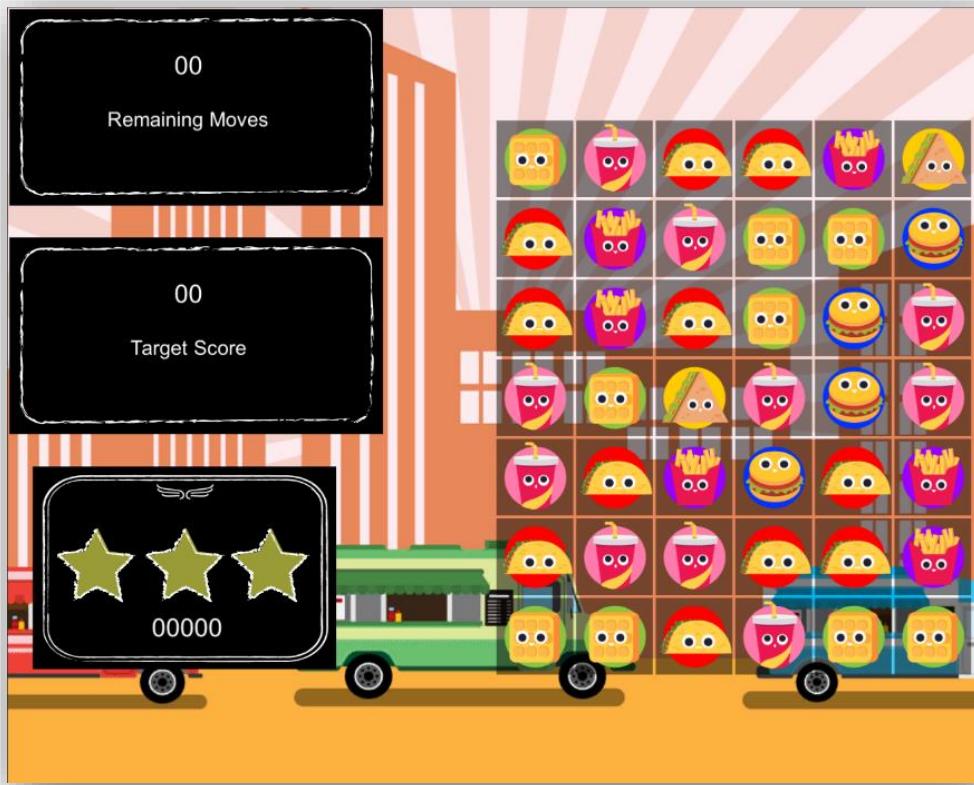
17

Our score will appear beneath the stars. Inside the **Score** object, add a **UI > Text** object. Call it "**score\_text**".

Change the **color** to white and change the **text** to "00000" or a similar 5-digit number. Adjust the **font size and position** of the text object so that it is centered under the stars. Make sure that the edges of the text box fit inside the remaining space in the background. If you do not see your text, it might be because you forgot to adjust the **width** and **height**. Go ahead and change the values until you can see your text.



As you can see, we've adjusted the size of the **score\_bg** object to account for the newly added **score\_text** object.

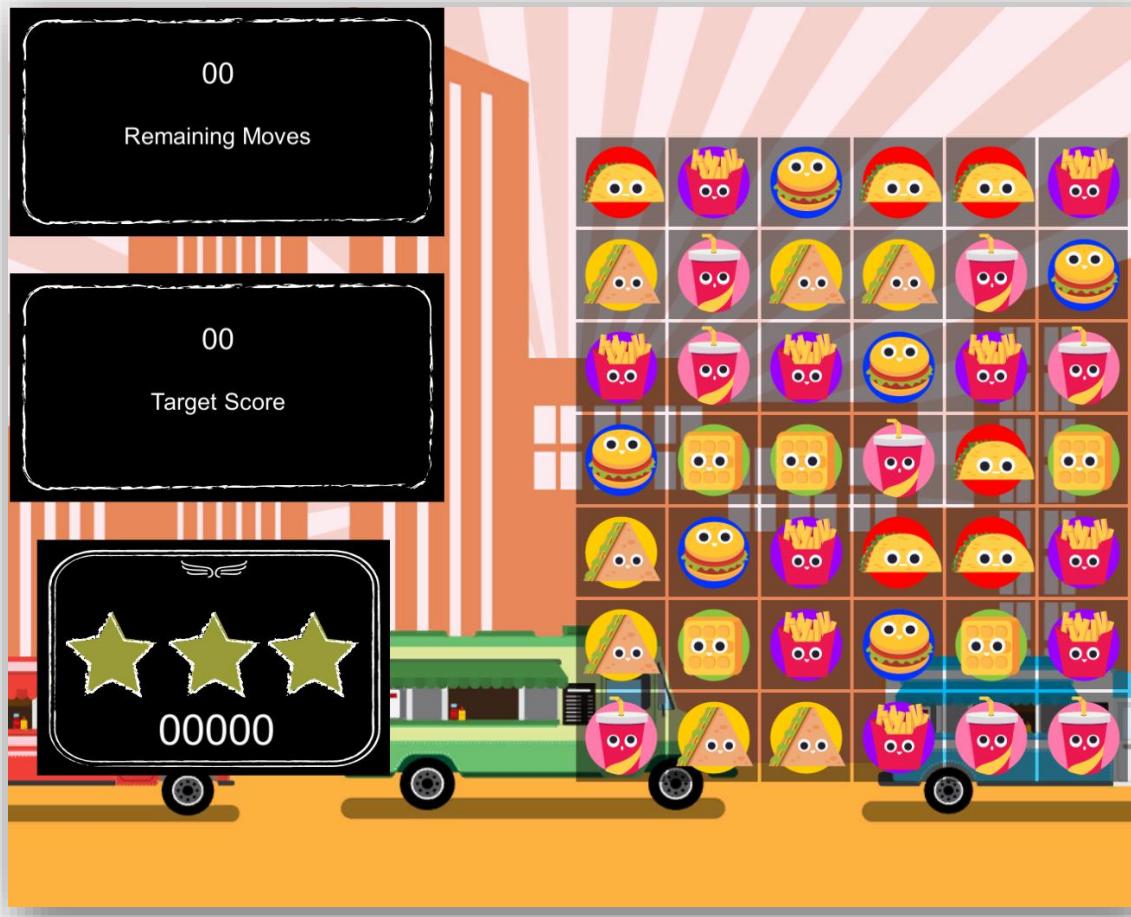


## 18

What happens if the player's Score is a number that's too big to fit in the space?

Unity has a fix for that. Inside the **Text** component in the **Inspector**, there is a check box for **Best Fit**. Make sure it is checked. Set the minimum size to the smallest size you think will fit, and the maximum size you think will fit. Play around with these numbers and the **score\_text** positioning and height/width until you like how it looks. When playing the game, Unity will automatically adjust the font size between the min and the max so that the score always fits the available area.

Make sure that these three elements fit within the boundaries for the Canvas. If necessary, use the shift key to select all three of the **Canvas elements** and adjust them so that they fit within the **canvas**.



**19**

Inside the **Scripts** folder, create a new **C# script** and name it **"HUD"**. "HUD" is short for **Heads Up Display**, meaning the information presented is part of the game window.



**20**

Double-click the **HUD** script to open it. We'll need to add a **directive**, so Unity knows how to handle the UI. At the top of the script, add:

```
using UnityEngine.UI;
```

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5
```

**21**

Inside the **class** we'll set up the **variables** that we need. We will need variables for the **Level** and **object**. There will also be a **GameOver** object, but we'll come back to this script later to add that.

```
public Level level;
```

```
6 public class HUD : MonoBehaviour
7 {
8 public Level level;
9 }
```

**22**

Then we need variables for all the text objects in our Canvas:

```
public Text remainingText;
public Text remainingSubtext;
public Text targetText;
public Text targetSubtext;
public Text scoreText;
```

```
6 public class HUD : MonoBehaviour
7 {
8 public Level level;
9
10 public Text remainingText;
11 public Text remainingSubtext;
12 public Text targetText;
13 public Text targetSubtext;
14 public Text scoreText;
15 }
16
```

**23**

We also need an **array** variable to store the four different star images in the **Score** object:

```
public Image[] stars;
```

```
6 public class HUD : MonoBehaviour
7 {
8 public Level level;
9
10 public Text remainingText;
11 public Text remainingSubtext;
12 public Text targetText;
13 public Text targetSubtext;
14 public Text scoreText;
15
16 public Image[] stars;
17 }
```

Remember that the **brackets** [] let the program know that this is an **array**.

**24**

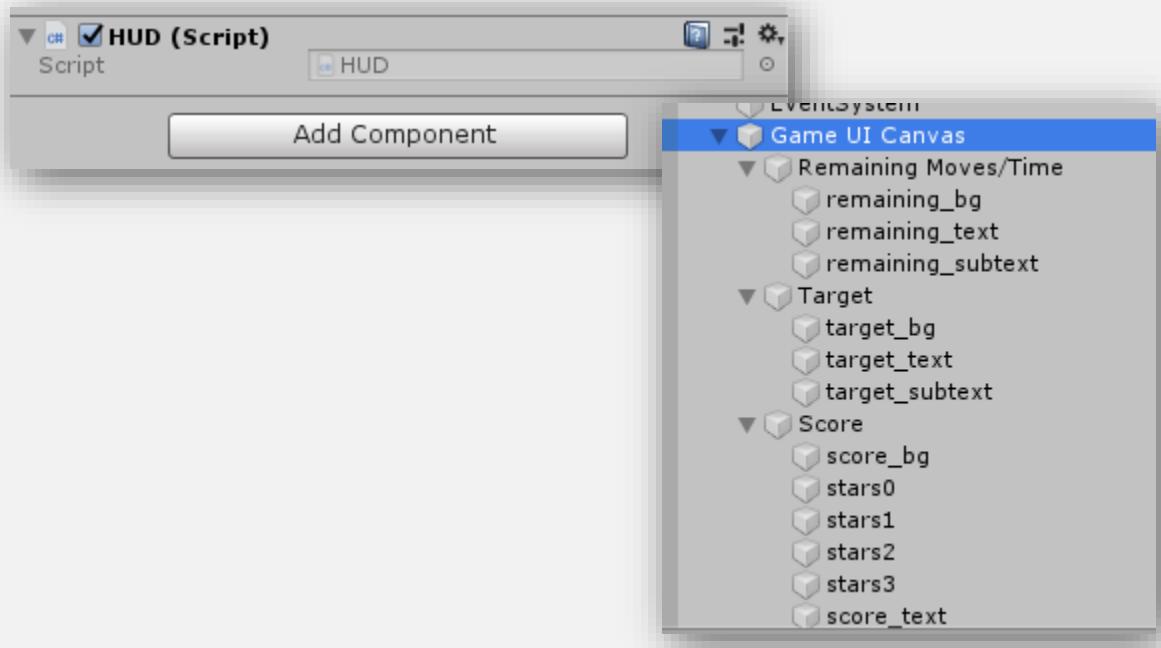
Finally, we'll need a **private variable** to identify which star image we want to show, and then another **private variable** for when the game is over.

```
private int starIndex;
private bool isGameOver;
```

```
6 public class HUD : MonoBehaviour
7 {
8 public Level level;
9
10 public Text remainingText;
11 public Text remainingSubtext;
12 public Text targetText;
13 public Text targetSubtext;
14 public Text scoreText;
15
16 public Image[] stars;
17
18 private int starIndex;
19 private bool isGameOver;
20 }
```

**25**

**Save** the script and go back to Unity. Add the **HUD** script to the **Canvas** object. Rename the **Canvas** object to "**Game UI Canvas**".



**26** In the **Inspector** for the **Game UI Canvas**, link all the variable objects:

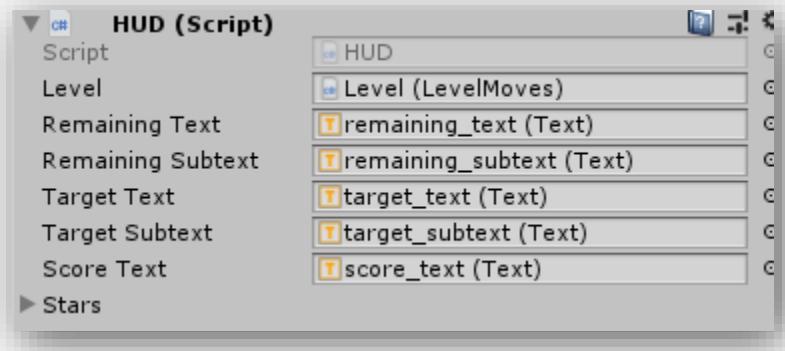
Level: Level gameObject

Remaining Text: remaining\_text

Remaining Subtext: remaining\_subtext

Target Text: target\_text

Target Subtext: target\_subtext

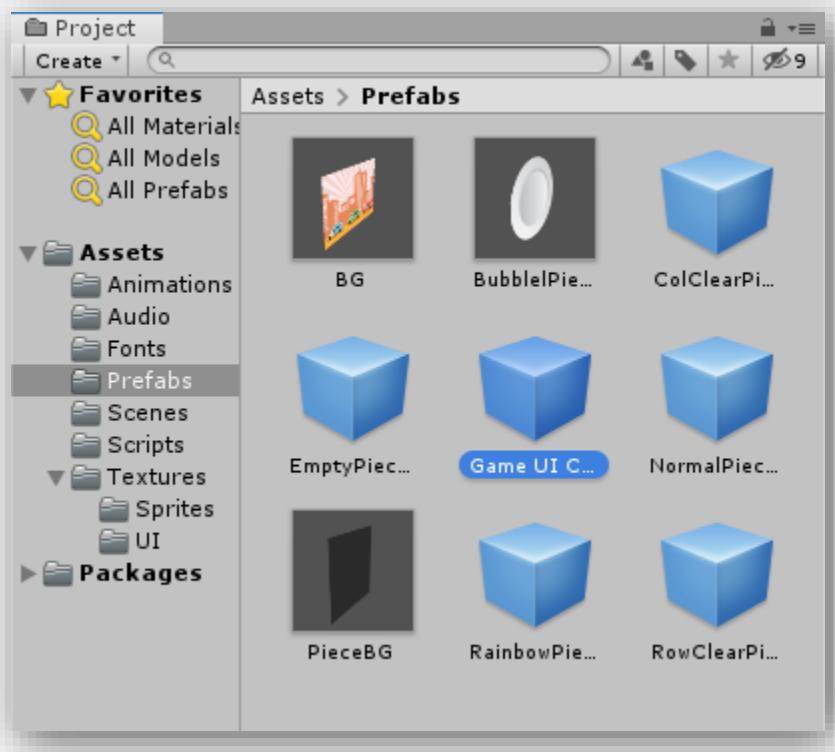


**27** Click the arrow next to **Stars** in the **HUD (Script)** component in the **Inspector**. Where it says **Size**, change the number to 4 to make the **Elements** appear. Link the **stars0** through **stars3** images from the **Score** object in the Hierarchy.



**28**

We'll want to use the UI for all the levels in this game. Drag **Game UI Canvas** into the **Prefabs** asset folder. **Save** this scene.



Open the **LevelTwo** scene and add the **Game UI Canvas** prefab to the **Hierarchy**. Do the same for **LevelThree**.



Make sure that the children of each group are ordered correctly, otherwise you might end up in a situation where the background is covering up the text.

If you're prompted to save while doing this, go ahead and **save**.

**29**

The canvas requires an **Event System**. If you don't see an **EventSystem** in the **Hierarchy** like in the images in the last step, add **UI > EventSystem**. Make sure everything has been **saved**.

**30**

Open the **HUD** script. We'll use the **Start** function to initialize the stars. Since we're updating the stars more than once in our game, we'll use a function for it. Add this to the **Start** function:

`UpdateStars();`

```
20 private void Start()
21 {
22 UpdateStars();
23 }
```

**31**

Next, let's write that function. We'll create a loop to go through the stars array and show the image that matches the current starIndex visible and hide the rest. First, *outside* of both the **Start** and **Update** functions, write a new function: `public void UpdateStars()`

Inside that, create a **for** loop that runs as many times as there are objects in the `stars[]` array.

```
for (int i = 0; i < stars.Length; i++) { }
```

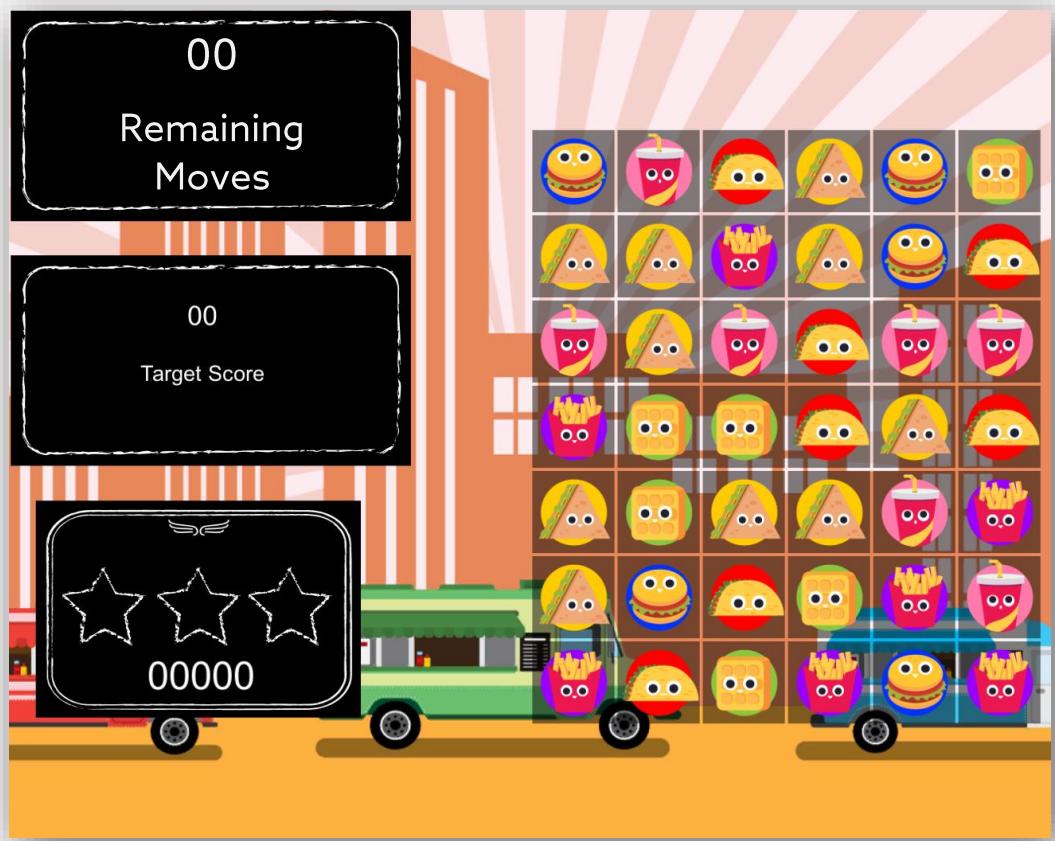
Inside the **for** loop, we check the current member of the `stars[]` array against the **starIndex** variable. **If** they're **equal**, then the object at that address of the `stars[]` array is **enabled**. **Else**, it is disabled by setting **enabled** to **false**.

```
if (i == starIndex)
{
 stars[i].enabled = true;
}
else
{
 stars[i].enabled = false;
}
```

```
public void UpdateStars()
{
 for (int i = 0; i < stars.Length; i++)
 {
 if (i == starIndex)
 {
 stars[i].enabled = true;
 } else
 {
 stars[i].enabled = false;
 }
 }
}
```

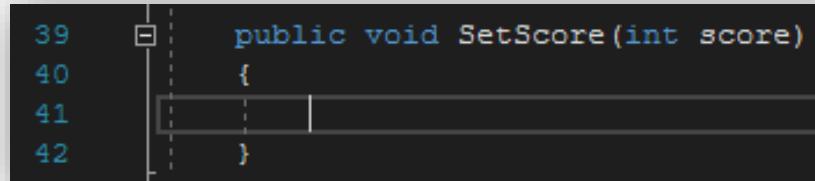
**32**

Save and do a playtest. When the game begins, this loop will make only the **star0** image visible.



- 
- 33** The next step is to create a new function for SetScore:

```
public void SetScore(int score) { }
```



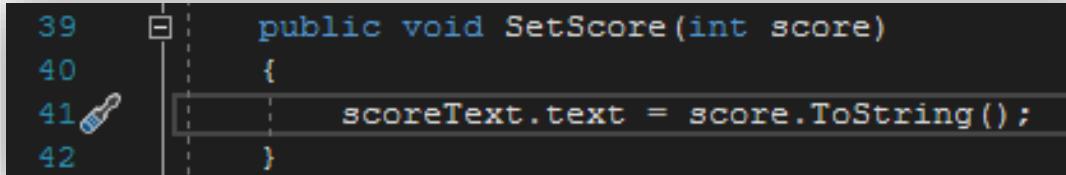
A screenshot of a code editor showing the declaration of a new function. The code is as follows:

```
39 public void SetScore(int score)
40 {
41 }
42 }
```

The cursor is positioned at the end of the opening brace on line 41.

- 
- 34** This function takes the parameter sent to it, **score**, and uses that integer in the rest of the function. First, we'll apply the score to the **scoreText** object:

```
scoreText.text = score.ToString();
```



A screenshot of a code editor showing the addition of a line of code to set the scoreText. The code is as follows:

```
39 public void SetScore(int score)
40 {
41 scoreText.text = score.ToString();
42 }
```

A small edit icon is visible next to the line of code on line 41.

**ToString** takes the **integer** and turns it into **text** for the **Text** object.

---

**35**

Then we need to compare the score to the minimum number defined in the Level object and display the right star:

```
int visibleStar = 0;
if(score >= level.score1Star && score < level.score2Star)
{
 visibleStar = 1;
} else if (score >= level.score2Star && score <
level.score3Star)
{
 visibleStar = 2;
}
else if (score >= level.score3Star)
{
 visibleStar = 3;
}
starIndex = visibleStar;
UpdateStars();
```

```
39 □ public void SetScore(int score)
40 {
41 scoreText.text = score.ToString();
42
43 int visibleStar = 0;
44
45 □ if (score >= level.score1Star && score < level.score2Star)
46 {
47 visibleStar = 1;
48 }
49 □ else if (score >= level.score2Star && score < level.score3Star)
50 {
51 visibleStar = 2;
52 }
53 □ else if (score >= level.score3Star)
54 {
55 visibleStar = 3;
56 }
57
58 starIndex = visibleStar;
59
60 UpdateStars();
61 }
```

The [UpdateStars\(\)](#) function is called after it has been decided which **star** image is **visible**, and that value is set to the **starIndex** variable.

**36**

Then we need **functions** to set the information for the remaining **moves**, **time**, and **target**. We'll need three functions:

```
public void SetTarget(int target)
{
 targetText.text = target.ToString();
}

public void SetRemaining(int remaining)
{
 remainingText.text = remaining.ToString();
}

public void SetRemaining(string remaining)
{
 remainingText.text = remaining;
}
```

```
63 □ public void SetTarget(int target)
64
65 {
66 targetText.text = target.ToString();
67 }
68 □ public void SetRemaining(int remaining)
69
70 {
71 remainingText.text = remaining.ToString();
72 }
73 □ public void SetRemaining(string remaining)
74
75 {
76 remainingText.text = remaining;
77 }
```

We might be sending an **integer** or a **string** to **SetRemaining**. By setting up the **parameters** for either possibility, we are ready for either situation. Remember, the **integer** needs to be converted to a **string** while the **text** object can display a **string** without having to convert it.

## 37

What's left to do is modifying the **subtext** objects depending on which of the three levels is being played. Inside a `SetLevelType` function, we'll be taking the `Level.LevelType type` parameter, and applying a **switch/case** statement to adjust **text** objects accordingly.

```
public void SetLevelType(Level.LevelType type)
{
 switch (type)
 {
 case Level.LevelType.MOVES:
 remainingSubtext.text = "moves remaining";
 targetSubtext.text = "target score";
 break;
 case Level.LevelType.OBSTACLE:
 remainingSubtext.text = "moves remaining";
 targetSubtext.text = "dishes remaining";
 break;
 case Level.LevelType.TIMER:
 remainingSubtext.text = "time remaining";
 targetSubtext.text = "target score";
 break;
 }
}
```

```
79 public void SetLevelType(Level.LevelType type)
80 {
81 switch (type)
82 {
83 case Level.LevelType.MOVES:
84 remainingSubtext.text = "Moves Remaining";
85 targetSubtext.text = "Target Score";
86 break;
87 case Level.LevelType.OBSTACLE:
88 remainingSubtext.text = "Moves Remaining";
89 targetSubtext.text = "Dishes Remaining";
90 break;
91 case Level.LevelType.TIMER:
92 remainingSubtext.text = "Time Remaining";
93 targetSubtext.text = "Target Score";
94 break;
95 }
96 }
```

We're using a **switch** statement to take the parameter of the **level type** and setting the appropriate **subtext** for the **target** and **remaining** objects.

Each level can only be one of these types. The way the game is set up is that Level One contains the amount of moves and the target score. Level Two contains dishes as obstacles, so it shows moves remaining and dishes remaining. Level Three has a timer, hence it shows time remaining and target score.

## 38

Finally, we need to add functions for `OnGameWin` and `OnGameLose`. In both of them, we will simply change the `isGameOver` variable to true:

```
public void OnGameWin(int score)
{
 isGameOver = true;
}
public void OnGameLose()
{
 isGameOver = true;
}
```

```
98 □ public void OnGameWin(int score)
99
100 | {
101 | | isGameOver = true;
102 | }
103 □ public void OnGameLose()
104
105 | {
106 | | isGameOver = false;
107 | }
```

Currently, these functions don't do much since there's nothing calling them.

**39**

The **Level** object takes care of the game functions such as **score** and so on. For the **Level** class to communicate with the **HUD**, we need to link to it. Open the **Level** script and add this variable:

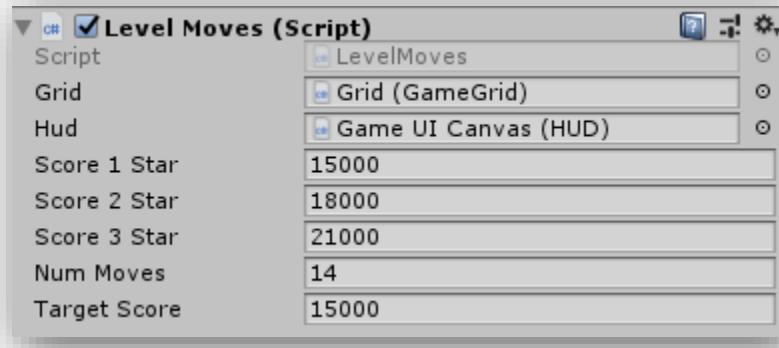
```
public HUD hud;
```

```
5 public class Level : MonoBehaviour
6 {
7 public enum LevelType
8 {
9 TIMER,
10 OBSTACLE,
11 MOVES,
12 };
13
14 public GameGrid grid;
15 public HUD hud;
16
17 public int score1Star;
18 public int score2Star;
19 public int score3Star;
20 }
```

The **Level** script will now be able to access any public variables or functions the **HUD** script contains.

**40**

**Save** the script and then go to Unity. Select the **Level** object and in the **Level** script component, link the **HUD** variable to the **Game UI Canvas** object. **Save** the scene.



**41**

Repeat this step for the **LevelTwo** and **LevelThree** scenes.

**42**

Back in the **Level** script, we can now pass the **score** to the **HUD** object. In the [Start\(\)](#) function, add this:

```
hud.SetScore(currentScore);
```

```
32 □ private void Start()
33 {
34 □ hud.SetScore(currentScore);
35 }
```

All this does is send the **currentScore** as a parameter to the **SetScore** function in the **HUD**. Then the **SetScore** function can update the score.

**43**

For the **GameWin** and **GameLose** results, we'll take the value of **didWin** and check to see if the grid has finished filling. If so, then we'll call **OnGameWin** and **OnGameLose** in the **HUD** script. Add this to [WaitForGridFill\(\)](#):

```
if (didWin && !grid.IsFilling)
{
 hud.OnGameWin(currentScore);
}
else
{
 hud.OnGameLose();
}
```

```
68 □ protected virtual IEnumerator WaitForGridFill()
69 {
70 □ while (grid.IsFilling)
71 {
72 □ yield return 0;
73 }
74
75 □ if (didWin && !grid.IsFilling)
76 {
77 □ hud.OnGameWin(currentScore);
78 }
79 □ else
80 {
81 □ hud.OnGameLose();
82 }
83 }
84 }
```

The first `if` statement checks if the **game has been won** and if the game is **done filling up the grid**. When **both** of these conditions are **true**, we call the `OnGameWin()` function. If those conditions are **not** met, we call the `OnGameLose()` function.

## 44

We'll also need to update the score when a piece is cleared. Find `OnPieceCleared` in the **Level** script and add this:

```
hud.SetScore(currentScore);
```

```
62 public virtual void OnPieceCleared(GamePiece piece)
63 {
64 //Update Score
65 currentScore += piece.score;
66
67 hud.SetScore(currentScore);
68 }
```

## 45

We also need to update the **HUD** based on which level is being played. Let's start with **LevelMoves**. Open the **LevelMoves** script (In the **Assets > Scripts** folder) and add these lines to the `Start()` function:

```
hud.SetLevelType(type);
hud.SetScore(currentScore);
hud.SetTarget(targetScore);
hud.SetRemaining(numMoves);
```

```
13 void Start()
14 {
15 type = LevelType.MOVES;
16
17 hud.SetLevelType(type);
18 hud.SetScore(currentScore);
19 hud.SetTarget(targetScore);
20 hud.SetRemaining(numMoves);
21 }
```

**46**

Each time the player makes a move, we need to update the remaining moves on the **HUD**. Find the `OnMove()` function and add this beneath `movesUsed++`:

```
hud.SetRemaining(numMoves - movesUsed);
```

```
23 public override void OnMove()
24 {
25 base.OnMove();
26
27 movesUsed++;
28
29 hud.SetRemaining (numMoves - movesUsed);
30 }
```

We need to update the number of moves remaining. So, every time we make a move, subtract the number of `movesUsed` from `numMoves`.

**47**

We will do the same for **LevelObstacles** and **LevelTimer**. Open the **LevelObstacles** script and add this to the `Start()` function:

```
hud.SetLevelType(type);
hud.SetScore(currentScore);
hud.SetTarget(numObstaclesLeft);
hud.SetRemaining(numMoves);
```

```
14 void Start()
15 {
16 type = LevelType.OBSTACLE;
17
18 for(int i=0; i<obstacleTypes.Length; i++)
19 {
20 numObstaclesLeft += grid.GetPiecesOfType(obstacleTypes[i]).Count;
21 }
22
23 hud.SetLevelType(type);
24 hud.SetScore(currentScore);
25 hud.SetTarget(numObstaclesLeft);
26 hud.SetRemaining(numMoves);
27 }
```

**48**

Once again, each time the player makes a move, we need to update the remaining moves on the **HUD**. Find the **OnMove** function and add this beneath `movesUsed++`:

```
hud.SetRemaining(numMoves - movesUsed);
```

```
35 public override void OnMove()
36 {
37 base.OnMove();
38
39 movesUsed++;
40
41 hud.SetRemaining(numMoves - movesUsed);
42 }
```

This code is exactly the same as the code added to **LevelMoves**, don't get them confused with each other!

**49**

Once we make a move and there is a match, we also need to update the hud score. In the **OnPieceCleared** function add the following:

```
public override void OnPieceCleared(GamePiece piece)
{
 base.OnPieceCleared(piece);

 for (int i = 0; i < obstacleTypes.Length; i++)
 {
 if(obstacleTypes[i] == piece.Type)
 {
 numObstaclesLeft--;

 hud.SetTarget(numObstaclesLeft);

 if (numObstaclesLeft == 0)
 {
 currentScore += 1000 * (numMoves - movesUsed);
 hud.SetScore(currentScore);
 GameWin();
 }
 }
 }
}
```

This way our final score will match in the HUD we created.

**50**

Then open the **LevelTimer** script. Since we're using time, things will be slightly different:

```
hud.SetLevelType(type);
hud.SetScore(currentScore);
hud.SetTarget(targetScore);
hud.SetRemaining(string.Format("{0}:{1:00}",timeInSeconds/60, timeInSeconds%60));
```

```
14 void Start()
15 {
16 type = LevelType.TIMER;
17
18 hud.SetLevelType(type);
19 hud.SetScore(currentScore);
20 hud.SetTarget(targetScore);
21 hud.SetRemaining(string.Format("{0}:{1:00}", timeInSeconds / 60, timeInSeconds % 60));
22 }
```

The `string.Format` function will take the format given in the first parameter and apply the second and third parameters to that format. So, that means that the `{0}` is replaced by the value given as `timeInSeconds/60`, and `{1:00}` is replaced by `timeInSeconds%60`.

**51**

In the `Update()` function, add this after `timer += Time.deltaTime;`:

```
hud.SetRemaining(string.Format("{0}:{1:00}",
(int)Mathf.Max((timeInSeconds-timer)/60,0),
(int)Mathf.Max((timeInSeconds-timer)% 60,0)));
```

```
void Update()
{
 if (!timeOut)
 {
 timer += Time.deltaTime;

 hud.SetRemaining(string.Format("{0}:{1:00}", (int)Mathf.Max((timeInSeconds - timer) / 60, 0), (int)Mathf.Max((timeInSeconds - timer) % 60, 0)));
 }
}
```

**52**

Now **load** the **LevelOne** scene and **play** the game. Is everything showing up in the **HUD**? If not, you might need to adjust the remaining and target text boxes. Be sure to hit **apply** to update the prefab. **Test** the levels and adjust the **Game UI Canvas** elements however you want.

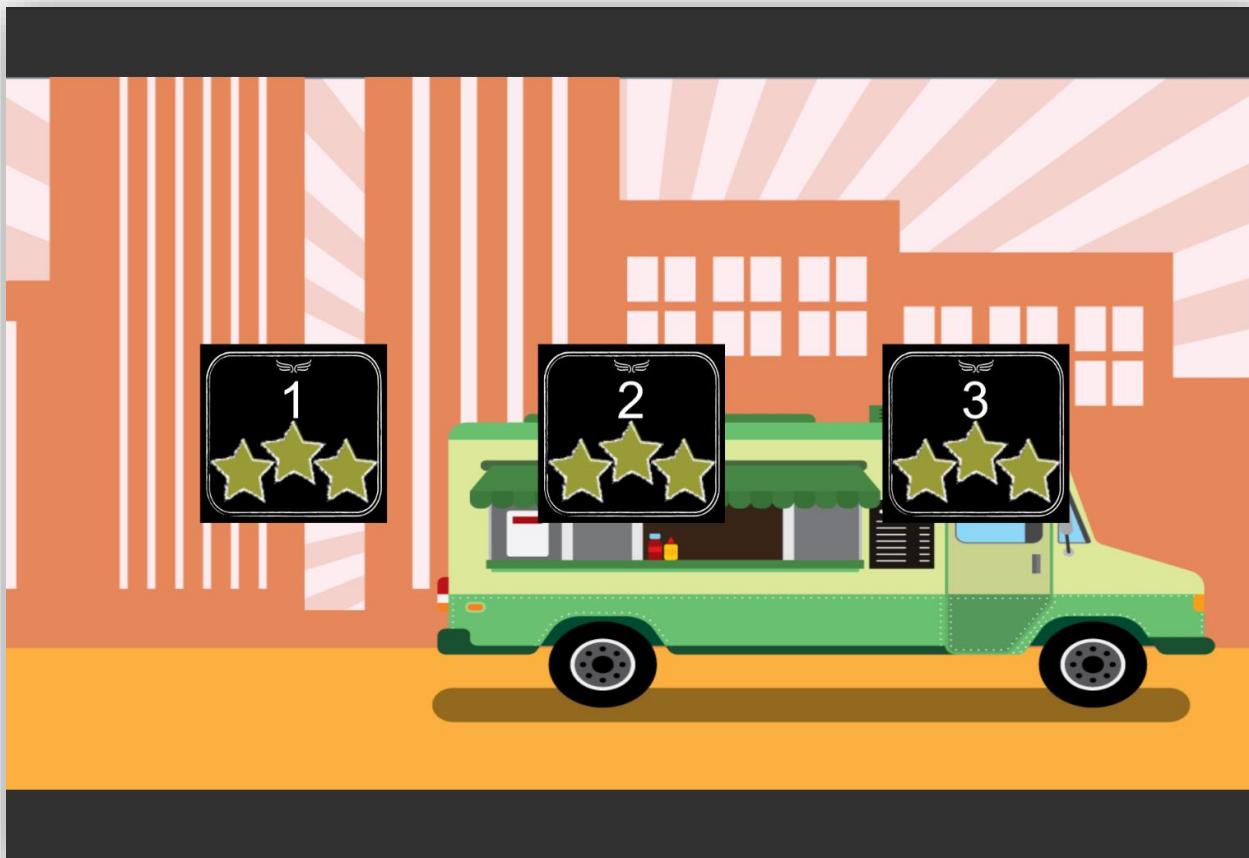




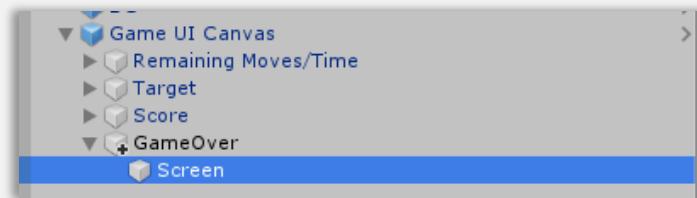
## Activity 17

# Food Frenzy Part 2

Well done! Now it's time for the finishing touches on our final Brown Belt game! The interface isn't quite done yet. There needs to be something after the game has finished to either let the player **play again or continue**, in other words, the **Game Over** screen. These next steps create the **Game Over** object to do that. We'll also be creating a **Level Select** to let players choose what type of level they want to play.



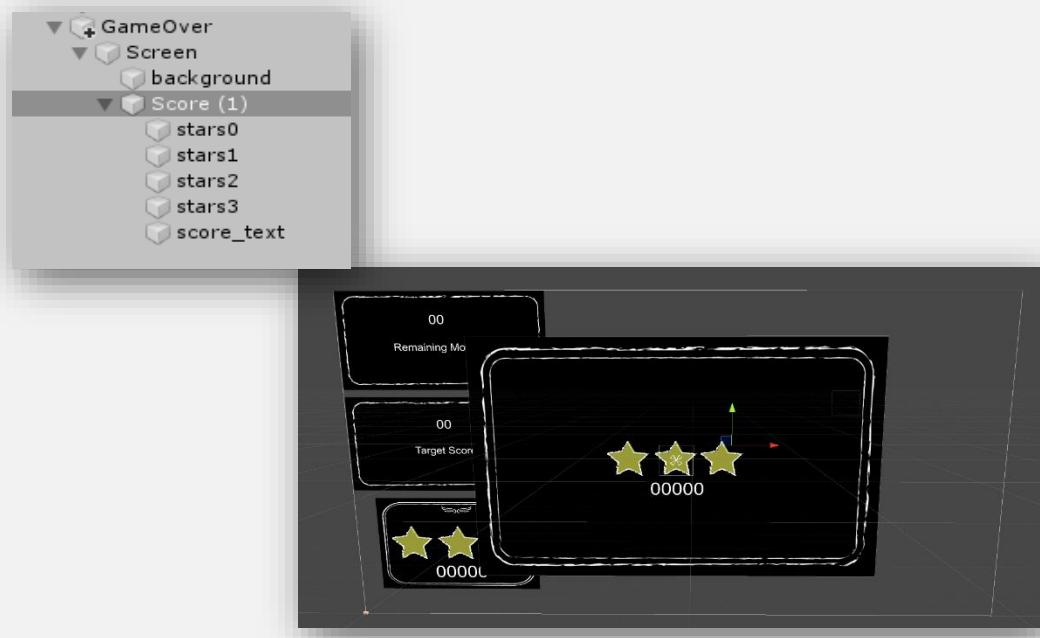
- 
- 1** We'll be continuing on the same project from Part 1. Re-open this project if you have closed it.
  - 2** In the **Hierarchy**, select the **Game UI Canvas** object.  
Add an **Empty GameObject** and give it the name of "**GameOver**".  
Inside the **GameOver** object, create another **empty GameObject** and name it "**Screen**".  
The reason we are putting an **empty GameObject** inside another **empty GameObject** is so we can disable or enable the UI elements of the **Screen** object without interfering with the script in the **GameOver** object.



- 
- 3** Inside the **Screen** object, add a **UI > Image** object.  
Rename this object as "**background**".  
In the **Inspector**, find the **Image** component and set the **Source Image** to **large\_rect**.  
Keep the image centered and adjust the image so that it fills most of the screen. (We used a width of 1197 units and a height of 800 units).



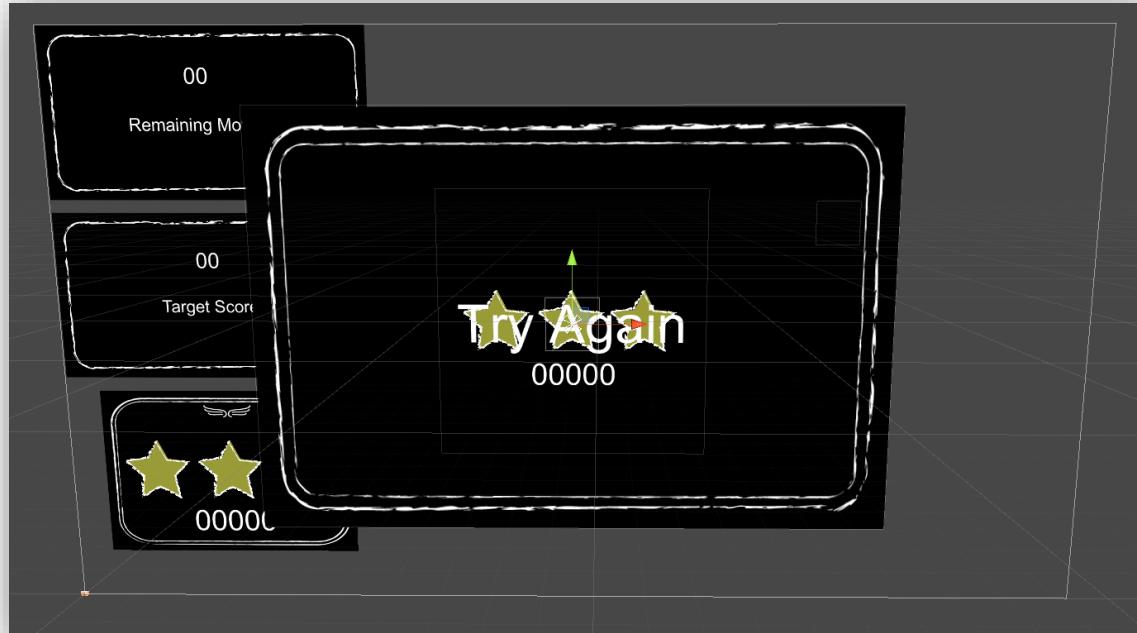
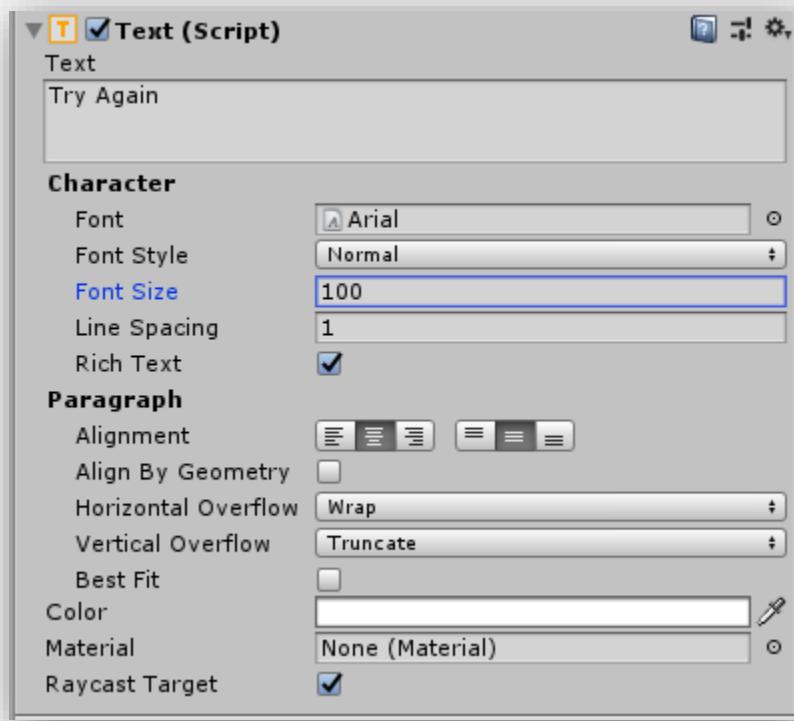
- 4** Make a **duplicate** of the **Score** object and place the duplicate inside **Screen**. Remove the **score\_bg** image from the **duplicate** and adjust the new **Score** elements so that it is in the center of the **Screen** object.



- 5** If the player loses, we won't show the stars or score, just a **message**. Inside the **Screen** object, create a **UI > Text** object and give it the name of "**lose\_text**". Change the **color** of the text to white and **resize** it so that it fits in the center of the **Screen** background.

Do not worry if it is hard to see in front of the stars, we won't be showing the stars if the player loses. Change the **text** to "**Try Again**".

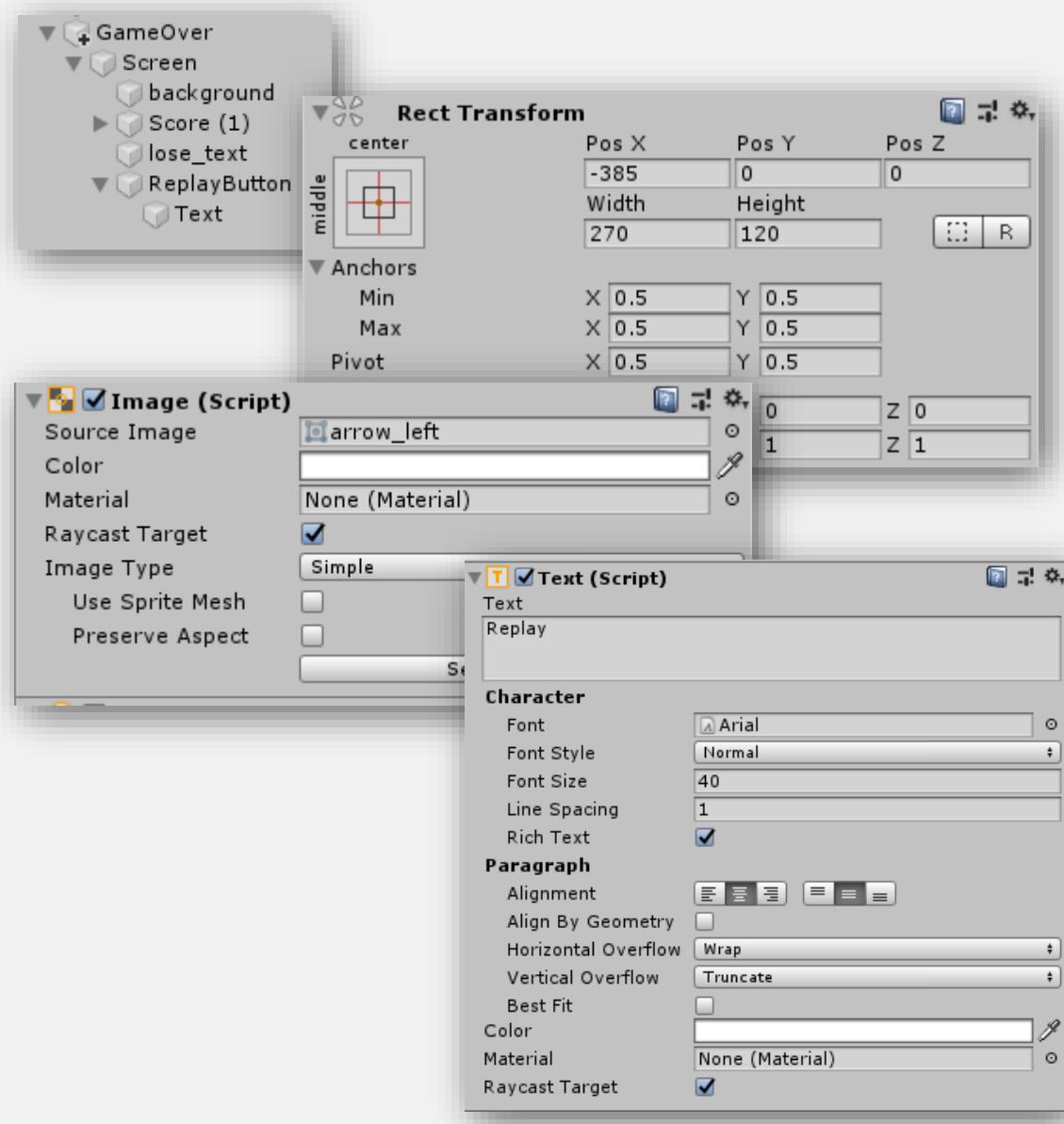


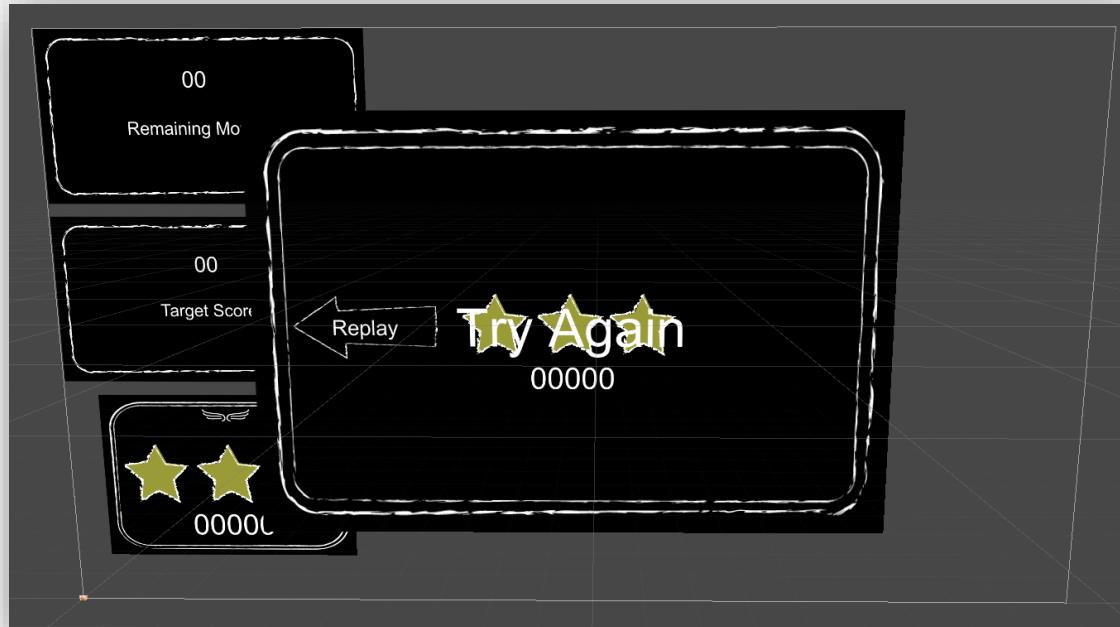


## 6 Our screen needs buttons to give the user choices on what to do next.

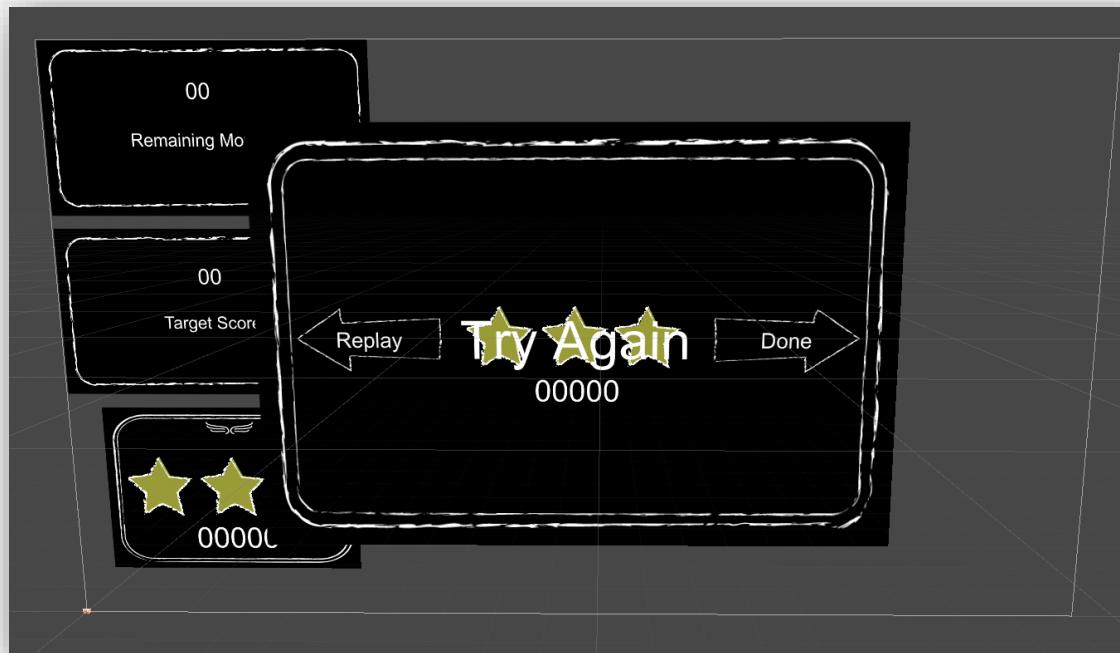
Add a **UI > Button** to the **Screen** object and give it the name of "**ReplayButton**". The **button** object comes with its own **sprite** and **text**, but we can change that. Find the **Image** component in the **Inspector** and replace the **Source Image** with **arrow\_left** from the assets.

Select the **text** object that is inside the **ReplayButton** object and make the **text color** white and change the **text** to "**Replay**". Resize the arrow so that the text can be seen and position it on the left side of the screen.





- 7 Duplicate the **ReplayButton** object and change the name to "**DoneButton**". Replace the **Source Image** with **arrow\_right** and change the **text** to "**Done**". Move this button to the right side of the **Screen** object as shown below.



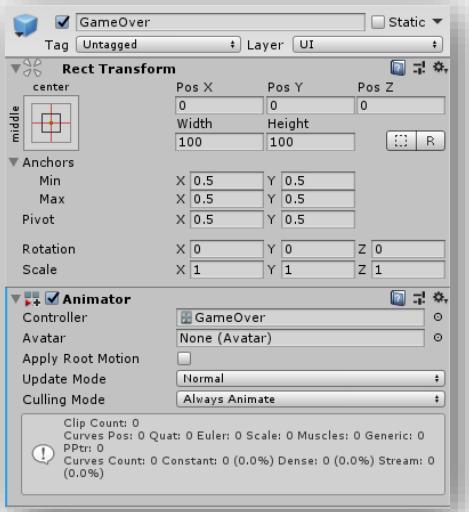
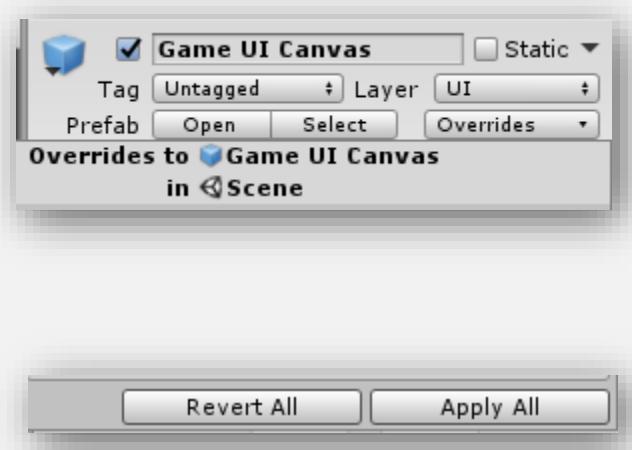
- 8** Update the **Game UI Canvas** prefab so that the changes are applied to all the scenes. You can do this by pressing the “**Override**” button at the top-right with the **Game UI Canvas** selected.

You will see a list of changes that have been made to the prefab. Make sure they’re correct and click **Apply All**.

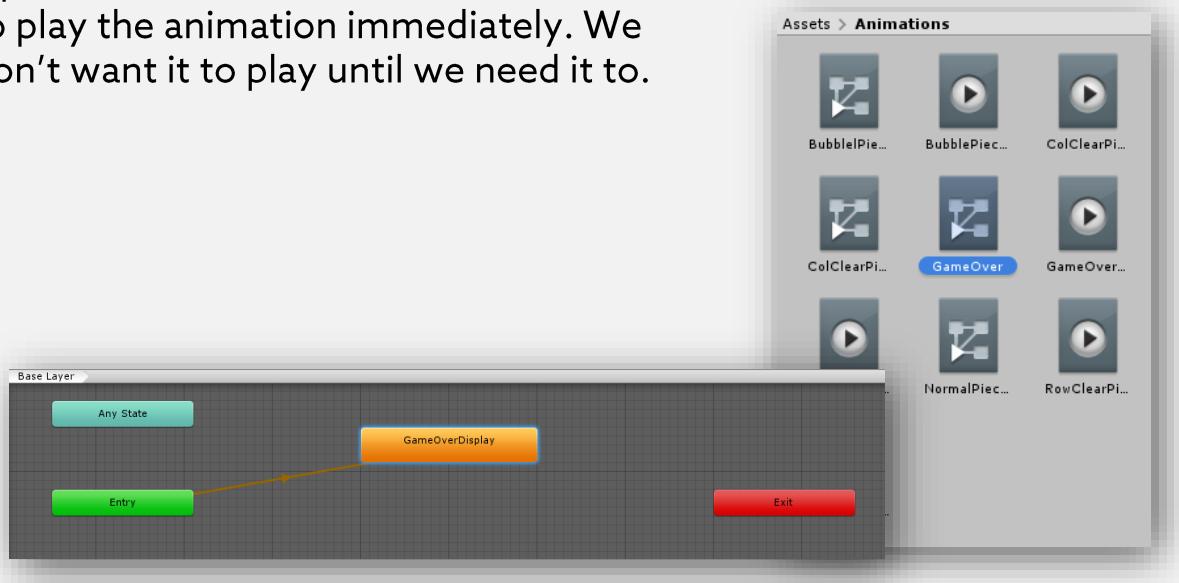
Next, we will have a little fun with how this object appears.

- 9** We could just have the game over screen show up, but where’s the fun in that?

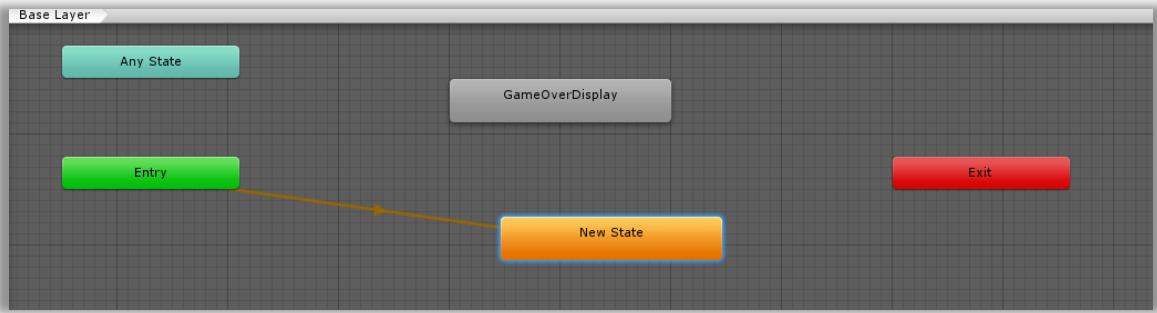
In the **Animations** folder, create a new animation by right-clicking on an empty space in the folder, and going to **Create > Animation**. Call it “**GameOverDisplay**”. Add this component to the **GameOver** object in the Hierarchy by dragging it into the **Inspector**.



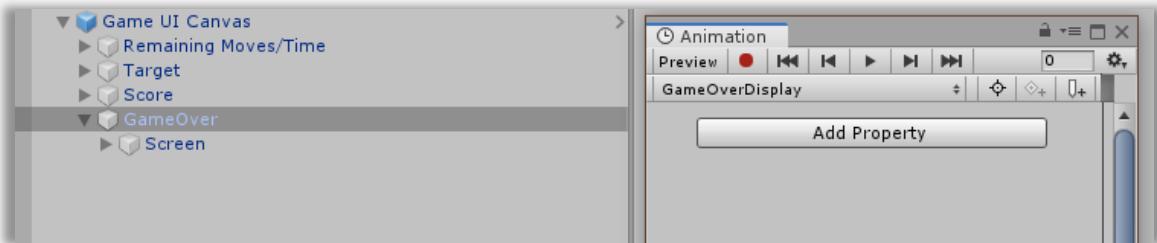
- 10** Open the **GameOver** controller. This is set to play the animation immediately. We don't want it to play until we need it to.



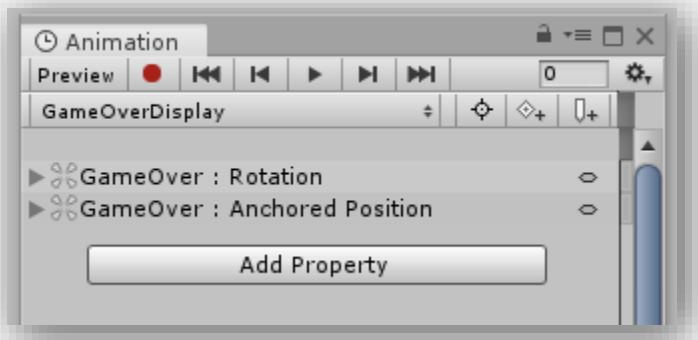
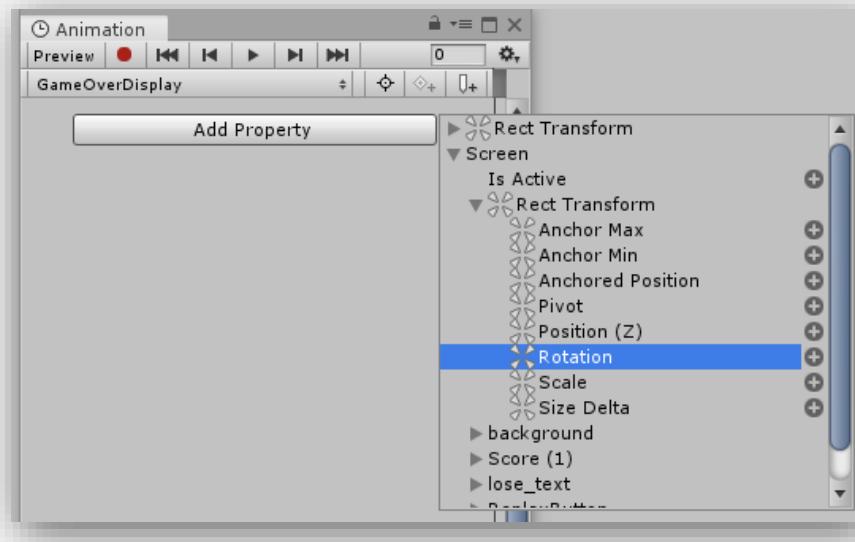
- 11** To fix this, we will create an empty animation state in the animator (right-click anywhere that does not have a rectangle, select **Create State > Empty**) and make this new state the default by right-clicking on the new state and selecting **Set as Layer Default State**.



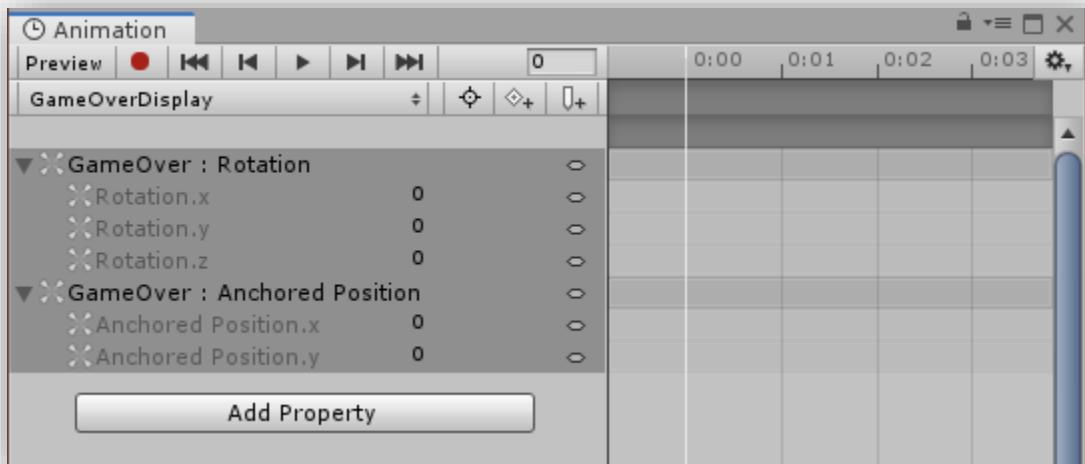
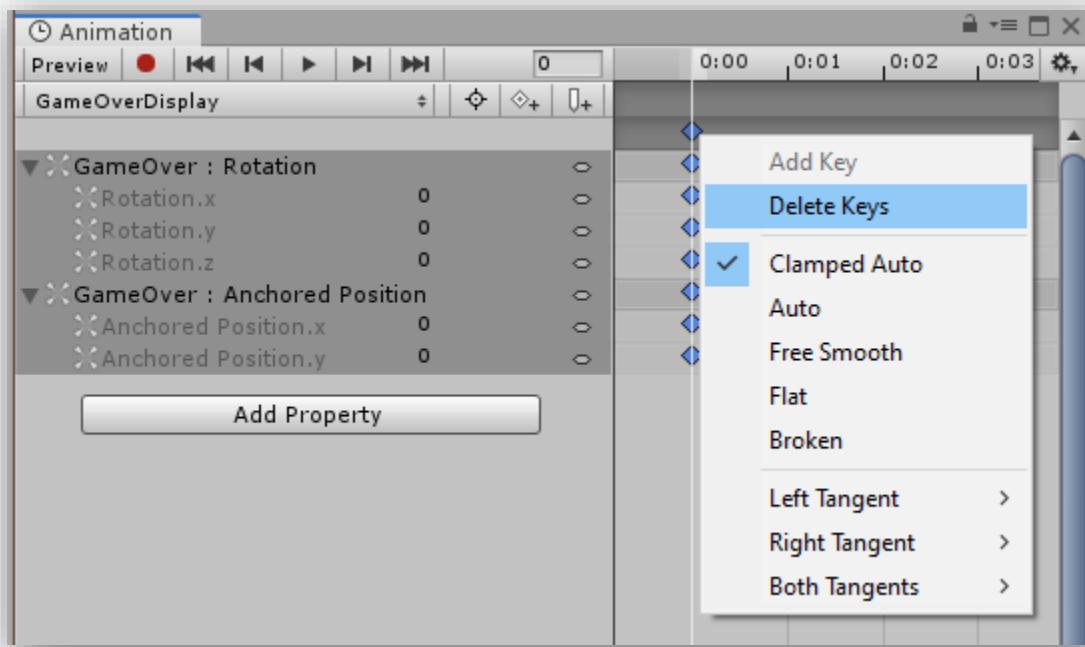
- 12** Now that the animator is set up, let's make our animation. In the **Animations** folder, double-click on **GameOverDisplay** to open the animation window. With the window open, confirm that you're editing the **GameOverDisplay** animation by selecting the **GameOverObject** in the **Hierarchy**.



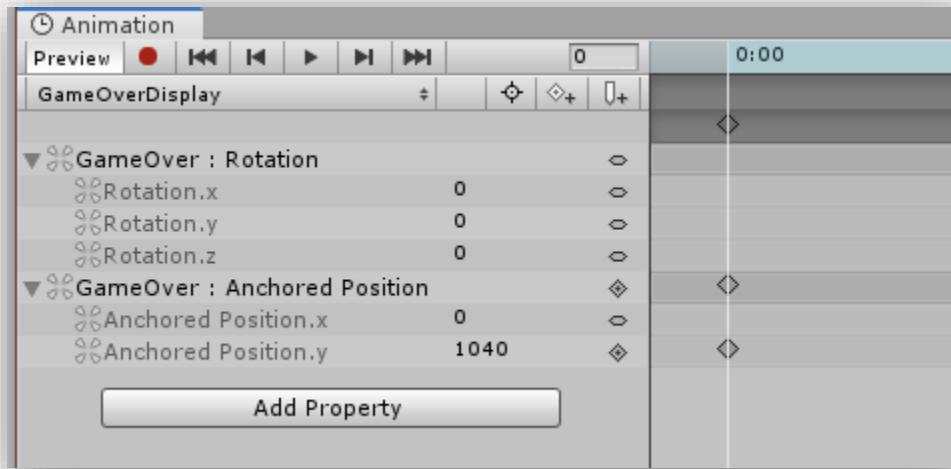
**13** Click on **Add Property** and select **Screen > Rect Transforms > Rotation**. Repeat these steps to add **Screen > Rect Transforms > Anchored Position**.



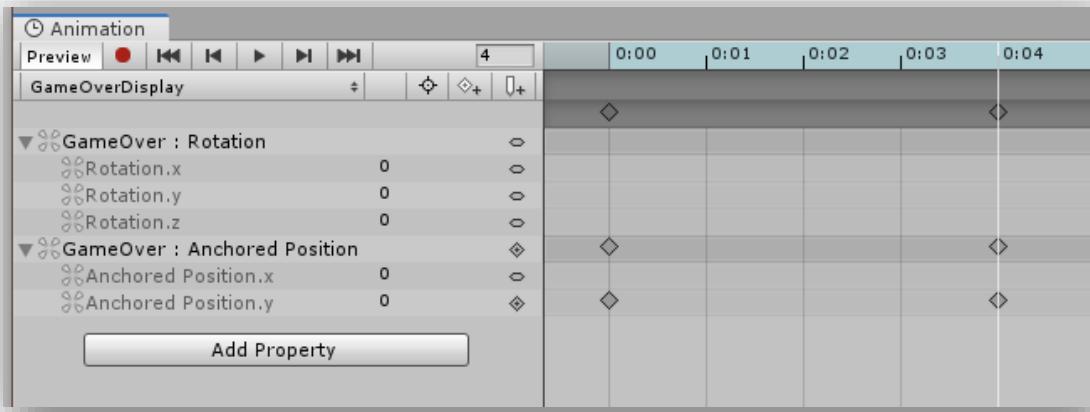
**14** Delete the last key frame by right-clicking on it. Expand the properties by clicking on the triangle next to each of the properties.



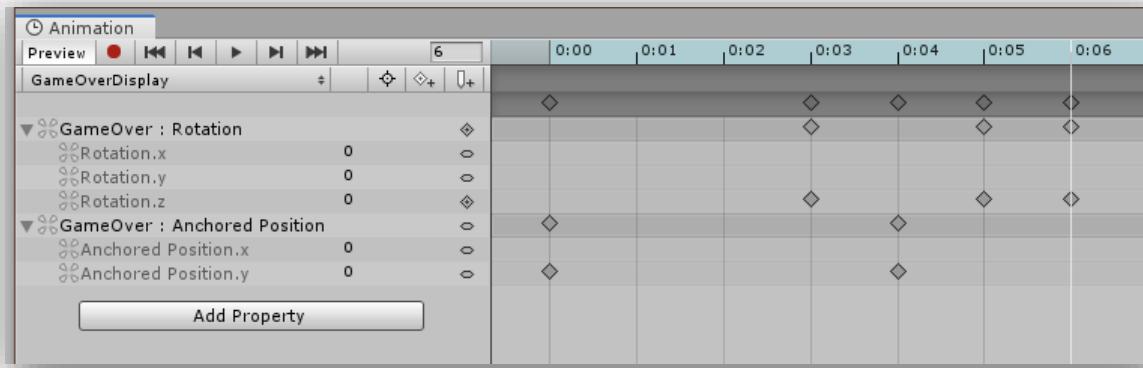
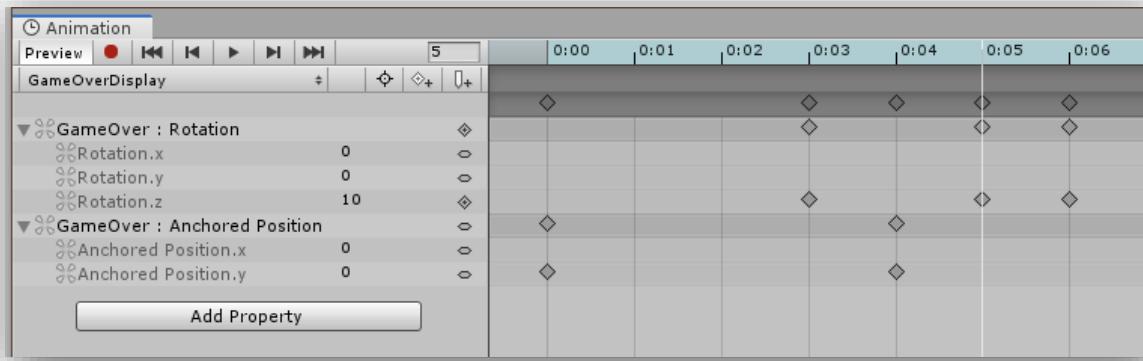
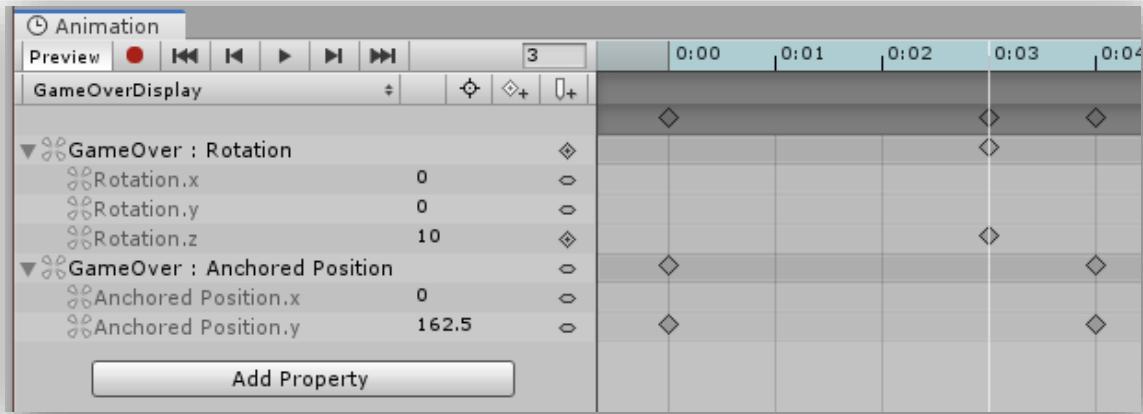
- 15** We want to start with the **Screen** object beyond the top of the canvas. Click on the **Anchored Position.y** property and give it a positive number like 1040.



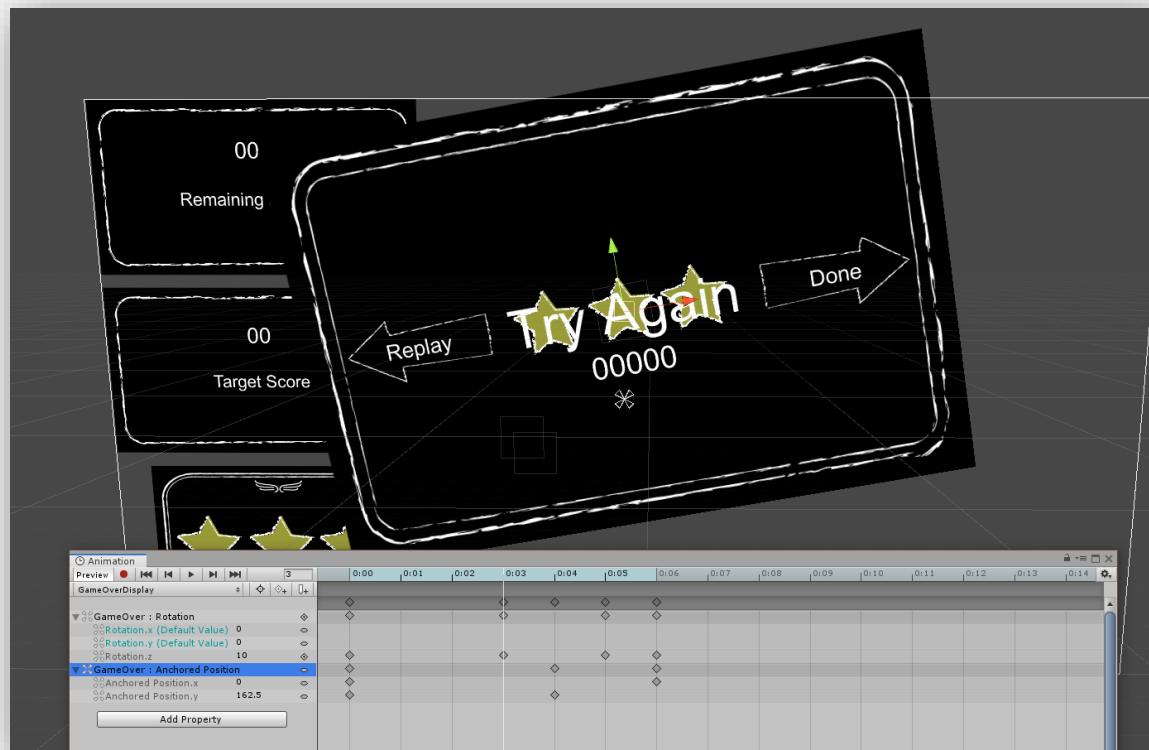
- 16** Move the playback line to the fourth second and set the **Anchored Position.y** back to 0, so that the screen is back in the center of the canvas.



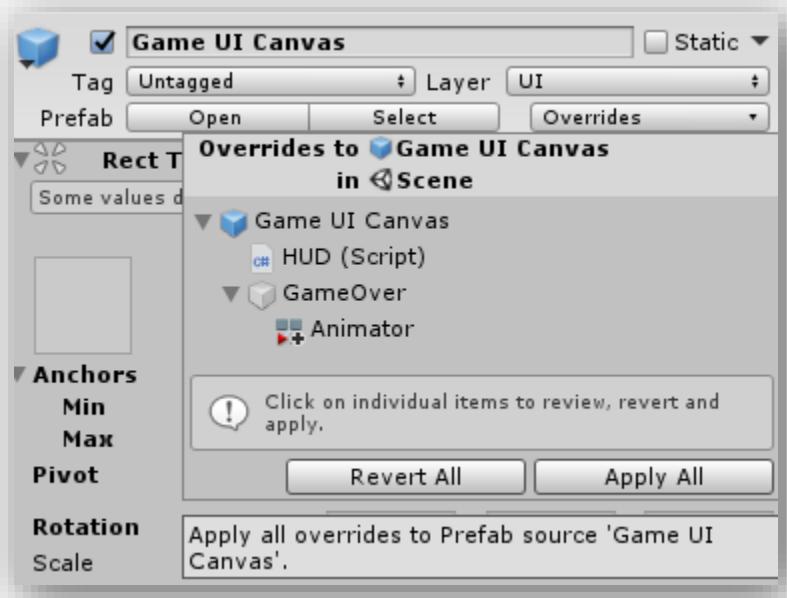
**17** Let's give the screen a little shake when it plays. Move the playback line to the **third** frame and set **Rotation.z** to 10. Then at the **fifth** frame, set the **Rotation.z** to 10. And finally, at the **sixth** frame, set the **Rotation.z** back to 0.



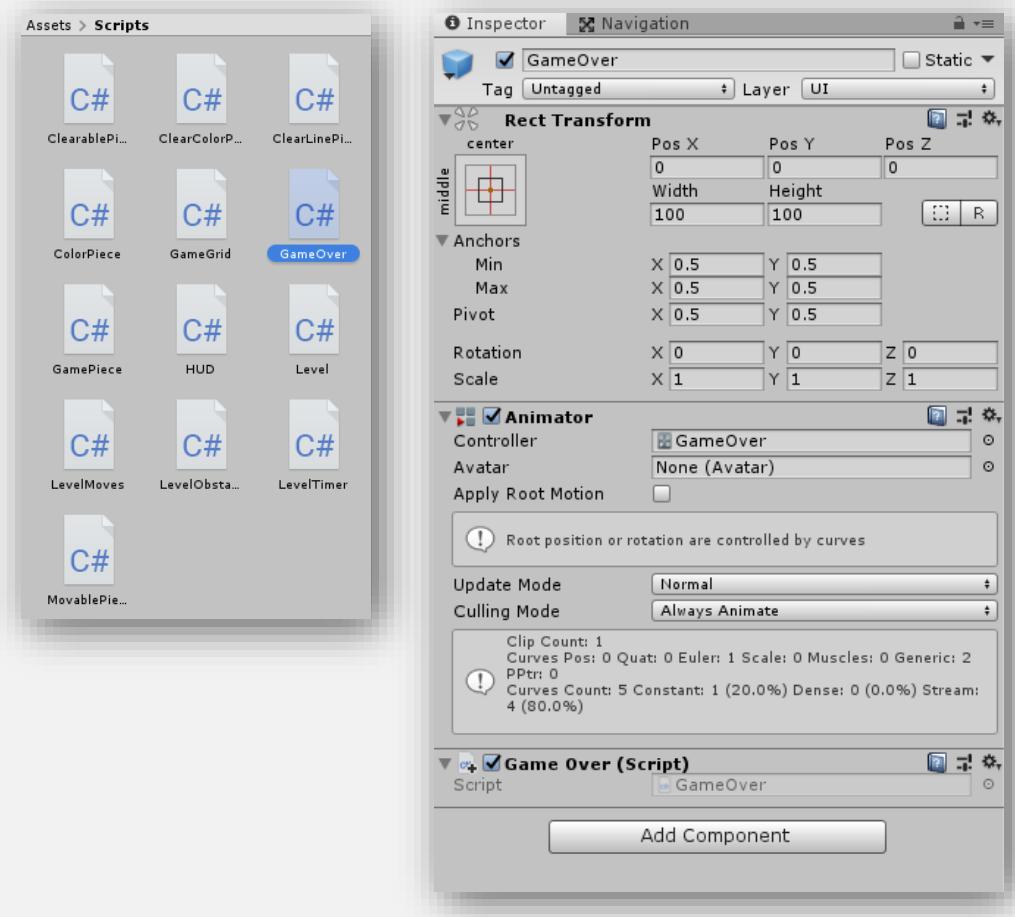
**18** Move the playback line back and forth (this is known as “scrubbing”) to see the animation.



**19** Select the **Game UI Canvas** in the **Hierarchy** and apply the **Overrides** to the prefab so that the animation is the same for all scenes. The next step is writing the script that controls this animation.



- 20** In the **Scripts** folder, create a **new C# script** and give it the name "**GameOver**". Add this script to the **GameOver** object in the **Hierarchy**. Open the script.



- 21** To start with, this Script needs some additional **directives** to handle everything. At the top of the script, add:

```
using UnityEngine.UI;
using UnityEngine.SceneManagement;
```

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5 using UnityEngine.SceneManagement;
```

- 22** For the script to work with the objects, we'll need variables for the **screen**, **score**, **text** and **image** objects:

```
public GameObject screenParent;
public GameObject scoreParent;
public Text loseText;
public Text scoreText;
public Image[] stars;
private Animator animator;
```

```
7 public class GameOver : MonoBehaviour
8 {
9 public GameObject screenParent;
10 public GameObject scoreParent;
11 public Text loseText;
12 public Text scoreText;
13 public Image[] stars;
14 private Animator animator;
15 }
```

- 23** When the game starts, we want the **screen** object and the **star** images to be hidden. We also need to define the animator. In the **Start()** function, add this:

```
screenParent.SetActive(false);
for (int i = 0; i < stars.Length; i++)
{
 stars[i].enabled = false;
}
animator = GetComponent<Animator>();
```

```
16 private void Start()
17 {
18 screenParent.SetActive(false);
19
20 for (int i = 0; i < stars.Length; i++)
21 {
22 stars[i].enabled = false;
23 }
24
25 animator = GetComponent<Animator>();
26 }
```

**24** Next, we need a **function** for when the player doesn't get enough points to earn a star. In this function we'll show the screen and the lose text, then hide the score object before playing the animation:

```
public void ShowLose()
{
 screenParent.SetActive(true);
 scoreParent.SetActive(false);
 loseText.enabled = true;

 if (animator)
 {
 animator.Play("GameOverDisplay");
 }
}
```

```
28 public void ShowLose()
29 {
30 screenParent.SetActive(true);
31 scoreParent.SetActive(false);
32 loseText.enabled = true;
33
34 if (animator)
35 {
36 animator.Play("GameOverDisplay");
37 }
38 }
```

**25** We will need to do the same for the when the player gets one or more stars. In this case, we'll hide the lose text and show the score object.

When this function is triggered, we'll need parameters for the score and the number of stars. However, we won't show the score immediately (we'll explain why in a moment).

Our `ShowWin` function will look like this:

```
public void ShowWin(int score, int starCount)
{
 screenParent.SetActive(true);
 scoreParent.SetActive(true);
 loseText.enabled = false;
 scoreText.text = score.ToString();
 scoreText.enabled = false;
 if (animator)
 {
 animator.Play("GameOverShow");
 }
}
```

```
40 public void ShowWin(int score, int starCount)
41 {
42 screenParent.SetActive(true);
43 scoreParent.SetActive(true);
44 loseText.enabled = false;
45 scoreText.text = score.ToString();
46 scoreText.enabled = false;
47
48 if (animator)
49 {
50 animator.Play("GameOverDisplay");
51 }
52 }
```

**26** At the moment, the object isn't showing the score or the stars. We want them to appear one at a time with a pause. Normally, C# does everything at once. If we want a pause, we will need a **coroutine**. In the `ShowWin()` function, add this at the end:

```
StartCoroutine(ShowWinCoroutine(starCount));
```

```
40 public void ShowWin(int score, int starCount)
41 {
42 screenParent.SetActive(true);
43 scoreParent.SetActive(true);
44 loseText.enabled = false;
45 scoreText.text = score.ToString();
46 scoreText.enabled = true;
47
48 if (animator)
49 {
50 animator.Play("GameOverDisplay");
51 }
52
53 StartCoroutine(ShowWinCoroutine(starCount));
54 }
```

`ShowWinCoroutine()` doesn't exist yet, so let's make it.

**27** A **coroutine** acts a lot like a **function**, but it needs a **special syntax** to work. Any **coroutine** needs to be called with **IEnumerator**:

```
private IEnumerator ShowWinCoroutine(int starCount){}
```

```
56 private IEnumerator ShowWinCoroutine(int starCount)
57 {
58
59 }
```

**28** In this case, we want to wait half a second to make things more dramatic. Add this to the `ShowWinCoroutine()`:

```
yield return new WaitForSeconds(0.5f);
```

```
56 private IEnumerator ShowWinCoroutine(int starCount)
57 {
58 yield return new WaitForSeconds(0.5f);
59 }
```

## 29 Then we'll set up a loop to show the earned stars one by one:

```
if(starCount < stars.Length)
{
 for(int i = 0; i <= starCount; i++)
 {
 stars[i].enabled = true;
 if (i > 0) {
 stars[i-1].enabled = false;
 }
 yield return new WaitForSeconds(0.5f);
 }
}
```

```
56 private IEnumerator ShowWinCoroutine(int starCount)
57 {
58 yield return new WaitForSeconds(0.5f);
59
60 if (starCount < stars.Length)
61 {
62 for (int i = 0; i <= starCount; i++)
63 {
64 stars[i].enabled = true;
65
66 if (i > 0)
67 {
68 stars[i - 1].enabled = false;
69 }
70
71 yield return new WaitForSeconds(0.5f);
72 }
73 }
74 }
```

- 30** After each star is shown, we'll hide the previous star and wait half a second. This means that there will be a half second pause before we show the score with the code below:

```
scoreText.enabled = true;
```

```
56 private IEnumerator ShowWinCoroutine(int starCount)
57 {
58 yield return new WaitForSeconds(0.5f);
59
60 if (starCount < stars.Length)
61 {
62 for (int i = 0; i <= starCount; i++)
63 {
64 stars[i].enabled = true;
65
66 if (i > 0)
67 {
68 stars[i - 1].enabled = false;
69 }
70
71 yield return new WaitForSeconds(0.5f);
72 }
73 }
74
75 scoreText.enabled = true;
76 }
```

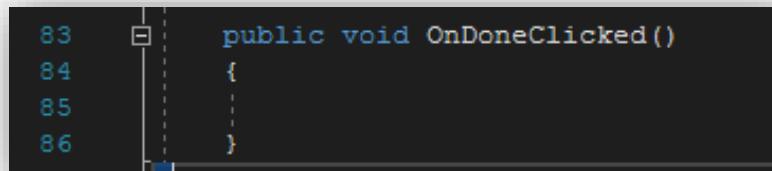
- 31** Finally, we need functions for when the **Replay** and **Done** buttons are clicked. If the player clicks **Replay**, we'll reload the current scene by using **SceneManager**:

```
public void OnReplayClicked()
{
 SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}
```

```
78 public void OnReplayClicked()
79 {
80 SceneManager.LoadScene(SceneManager.GetActiveScene().name);
81 }
```

- 32** We want to do the same for the **Done** button, but that scene hasn't been created yet. We'll leave that blank for now:

```
public void OnDoneClicked() { }
```



- 33** Save this script and go back to Unity. Select the **GameOver** object to link the objects and images to the variables:

Screen Parent: Screen

Score Parent: Score

Lose Text: lose\_text

Score Text: score\_text

Expand the **Stars** section in the **Inspector** by clicking the arrow.

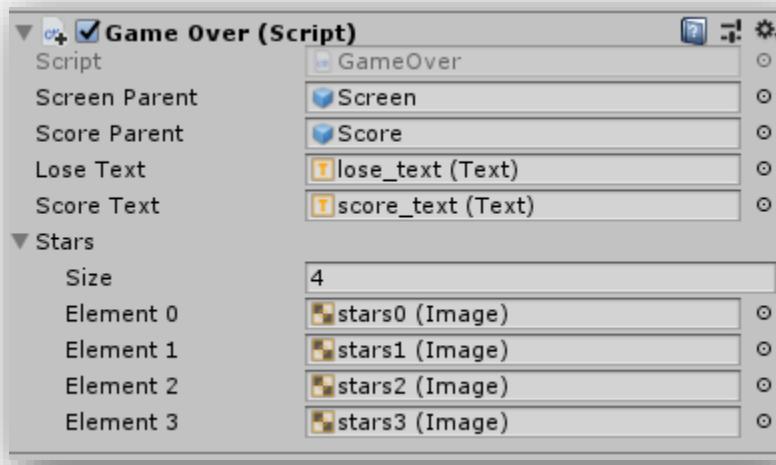
Size: 4

Element 0: stars0

Element 1: stars1

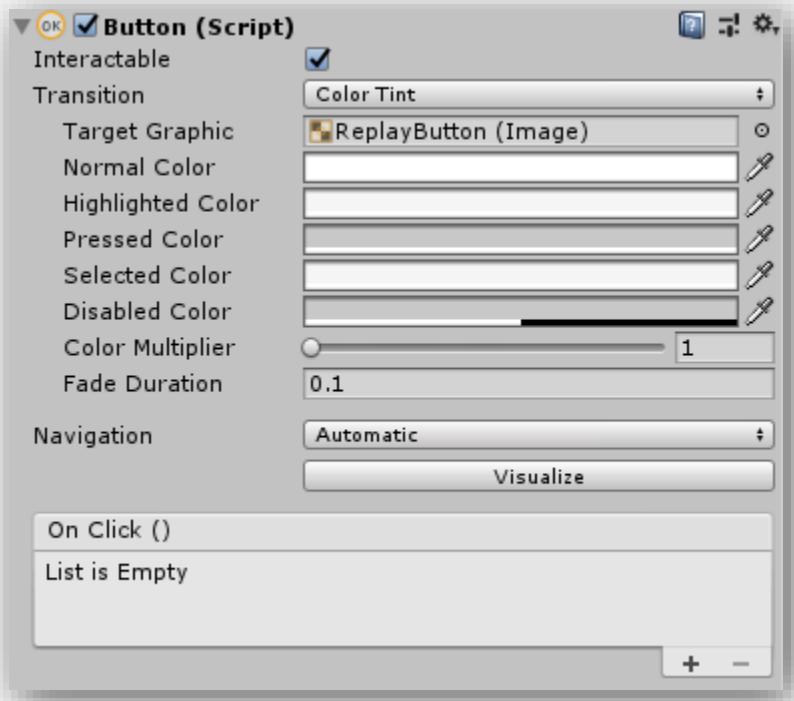
Element 2: stars2

Element 3: stars3

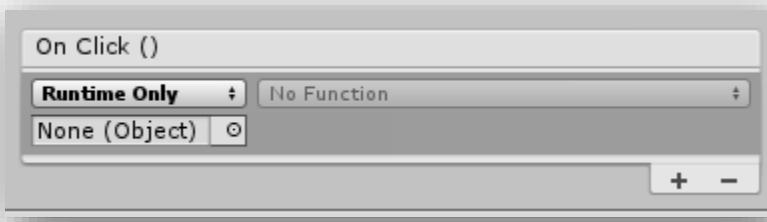


**34** Find each of the **button** objects in the **Screen** object. Each button has its own **script** by default. We just need to tell each button what to do when it is clicked.

Starting with the **Replay** button, find the **Button** component in the **Inspector**. At the bottom is an empty list called **On Click ()**. We will use this to link to the **OnReplayClicked()** function in the **GameOver** script.

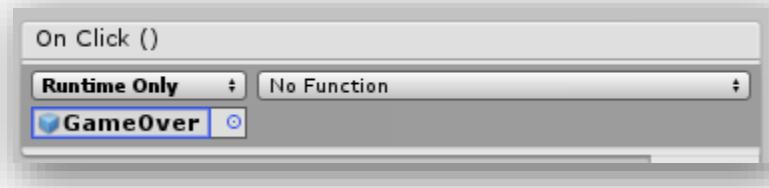


To add to the **On Click ()** list, click on the **+** symbol at the bottom of the list.



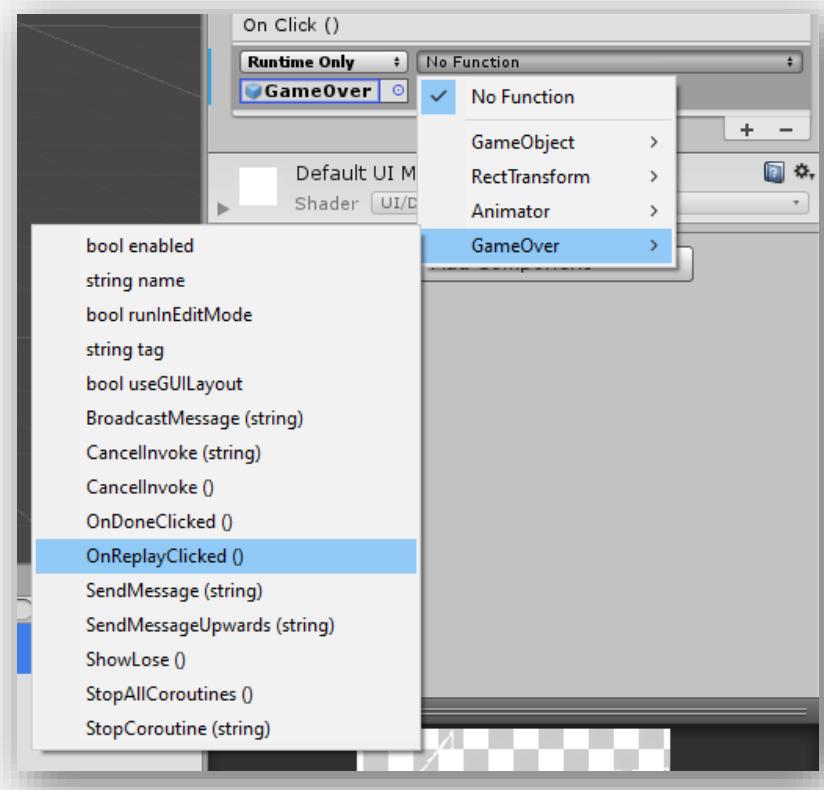
The first box says Runtime Only – we'll leave it as is. There are very few situations where you would need something other than that, so it's a good default.

Beneath that, is a slot for our script to be attached to. Click on the circle and link the **GameOver** object (or drag it over from the **Hierarchy**).

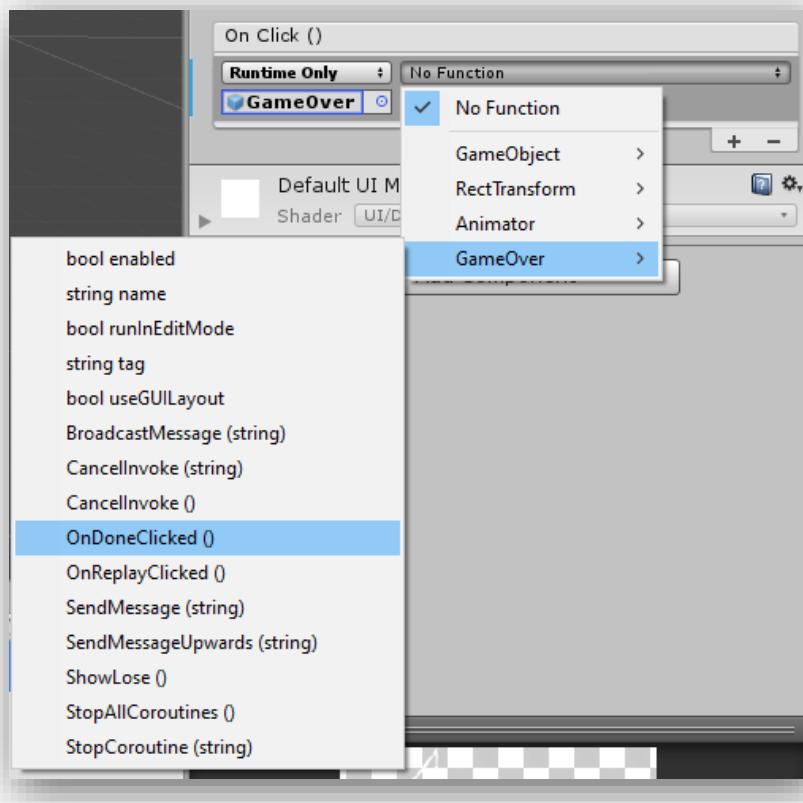


Make sure that you've attached the **GameOver object** and not the **C# script** named the same thing.

Click on the box that says **No Function** to link the function that we want. There will be a drop-down menu. Find **GameOver** and then find the **OnReplayClicked** function. Now the **Replay** button will call that function when it is clicked.



**35** Repeat the above steps in the **Done** button to link the **OnDoneClicked** function to the **Done** button.



**36** Once again, select the **Game UI Canvas** in the **Hierarchy** and apply the override to the prefab.



- 37** The **Replay** and **Done** buttons may be connected, but the **Screen** object itself is not. Open the **HUD** script.



- 38** We already have the functions for `OnGameWin()` and `OnGameLose()` in the script. Now we need to use them to call the correct function in the **GameOver** script. In the variable declarations, add the following line of code.

```
public GameOver gameOver;

6 public class HUD : MonoBehaviour
7 {
8 public Level level;
9 public GameOver gameOver;
10 }
```

- 39** In the `OnGameWin()` function, replace the `isGameOver` code with this code:

```
gameOver.ShowWin(score, starIndex);
```

```
99 public void OnGameWin(int score)
100 {
101 gameOver.ShowWin(score, starIndex);
102 }
```

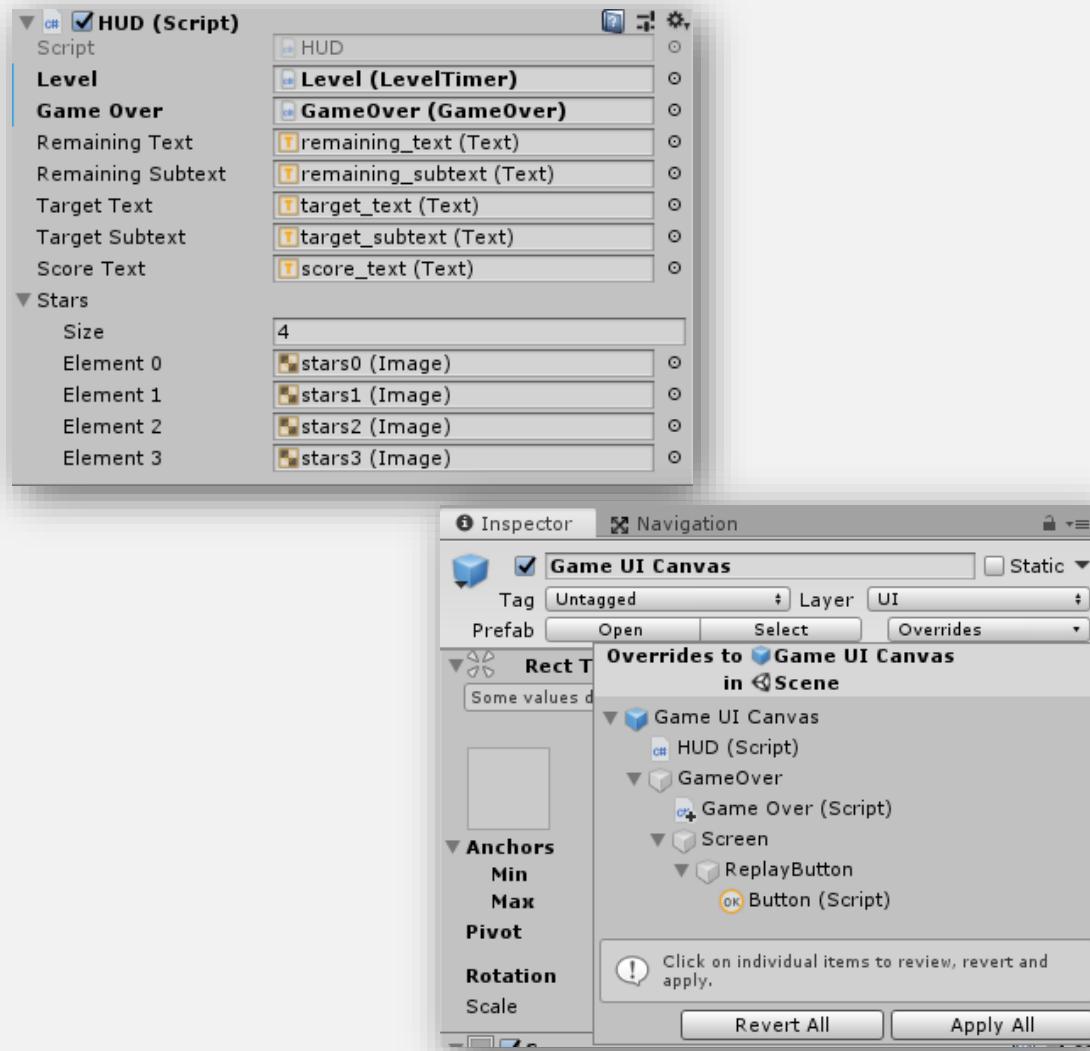
## 40 We'll make a similar change in the `OnGameLose()` function:

```
gameOver.ShowLose();
```

```
104 public void OnGameLose()
105 {
106 gameOver.ShowLose();
107 }
```

Unlike `ShowWin()`, which takes parameters, `ShowLose()` doesn't need any.

## 41 Save the script and go back to Unity. Select **Game UI Canvas** in the Hierarchy and link the **Game Over** variable to the **GameOver** object and apply the override to the prefab.



**42** Now **test** the game to make sure that the **GameOver** screen shows up at the end.

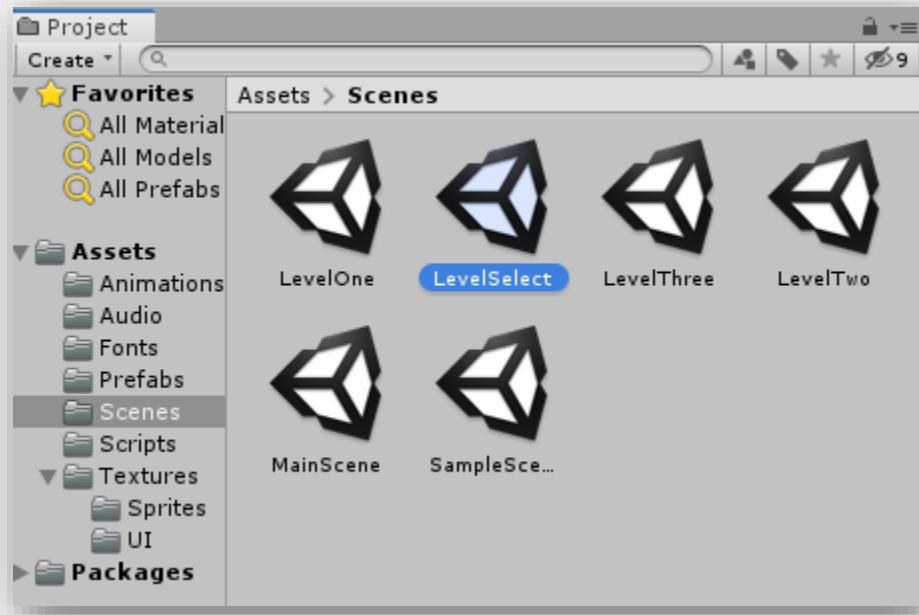
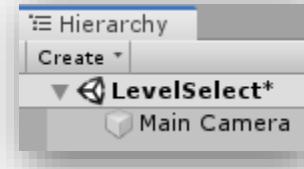
The settings for the star scores and other parameters for each level have already been placed in the script attached to the **Level** object in each scene. If you think the game is too easy or too hard, you can modify these numbers.

You might have noticed that the **Done** button doesn't do anything.

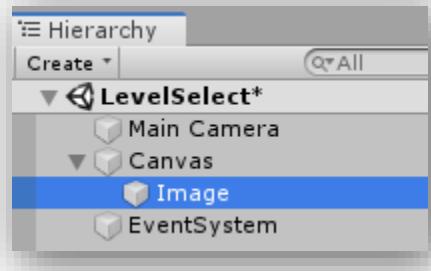
Let's fix that!

**43** To complete the game, we need a menu to start from and return to after each game is finished. This menu will let the player choose any level they want to play.

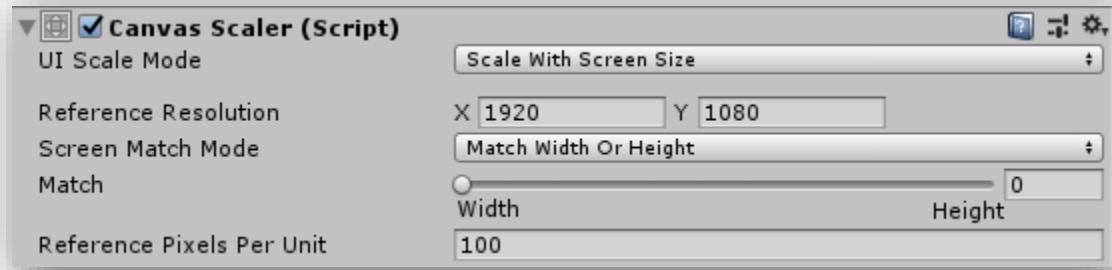
Go to the **Scenes** folder, and right-click to create a scene. Give it the name of "**LevelSelect**". Delete the light, you won't need it.



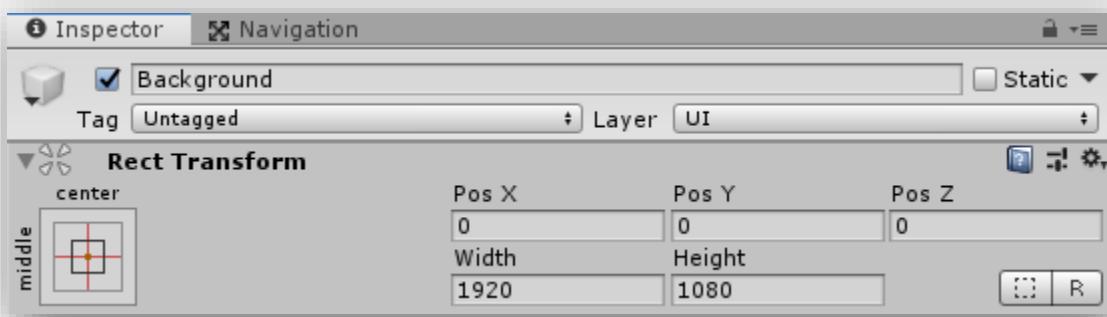
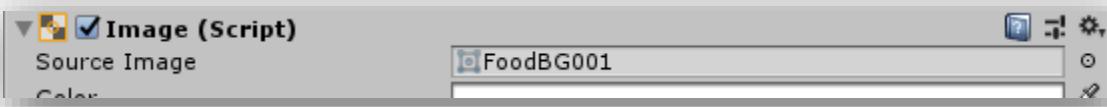
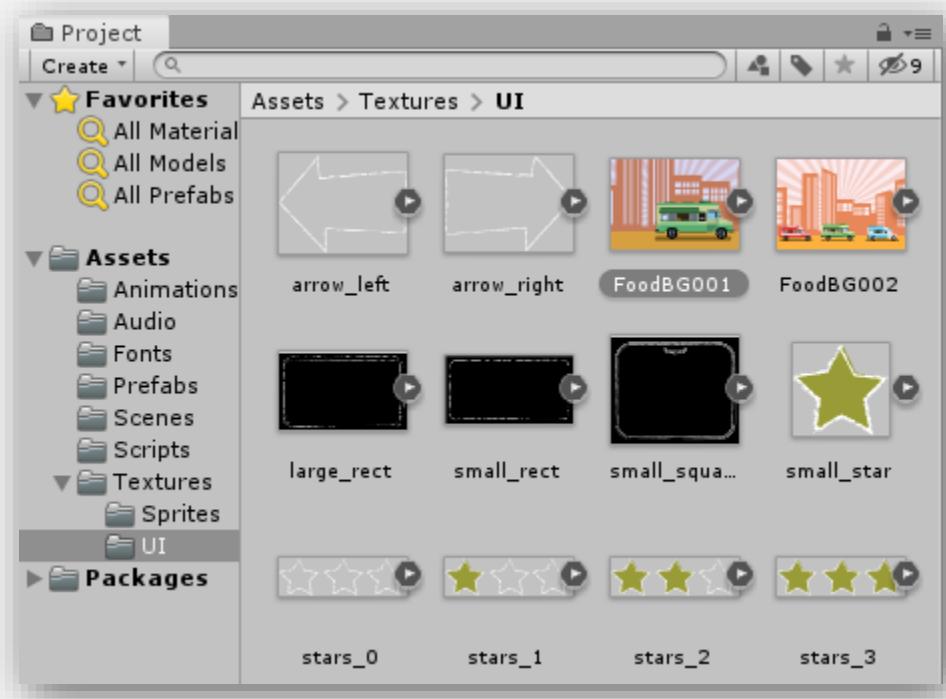
**44** Use **UI > Image** to add an image to the scene. When you do that, Unity automatically adds a canvas and an event system.



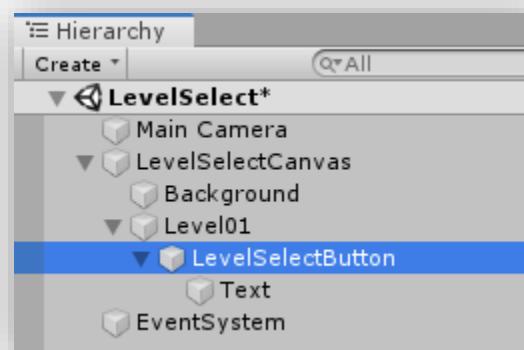
**45** Rename the **Canvas** to **LevelSelectCanvas**. Find the **Canvas Scaler** component in the **Inspector** and set it to **Scale With Screen Size**. Change the **reference resolution** to x: 1920 and y: 1080 to match the other scenes.



**46** Change the name of the **Image** to **"Background"** and change the **Source Image** to **FoodBG001**. Adjust the image so that it is the same width and height of the canvas (When adjusting the image size, Unity will "snap" the corners to the corners of the Canvas, making your job a lot easier).

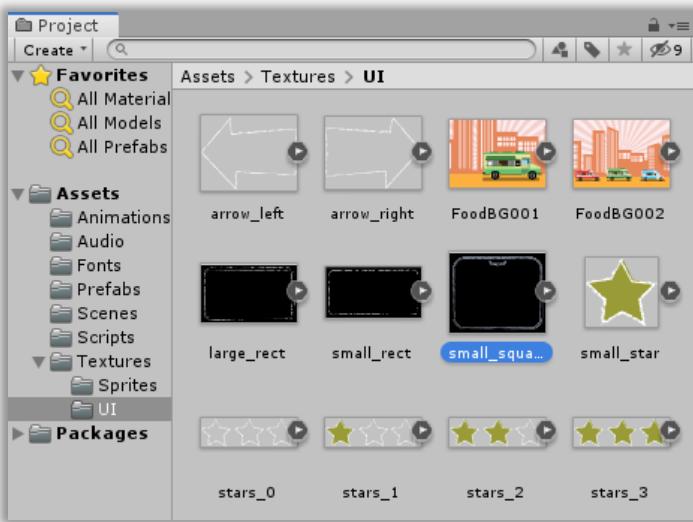


**47** A menu needs buttons. In the **Canvas**, add an **empty object** and give it the name of **Level01**. Inside this object, use **UI > Button** to add a button and give it the name of "**LevelSelectButton**".

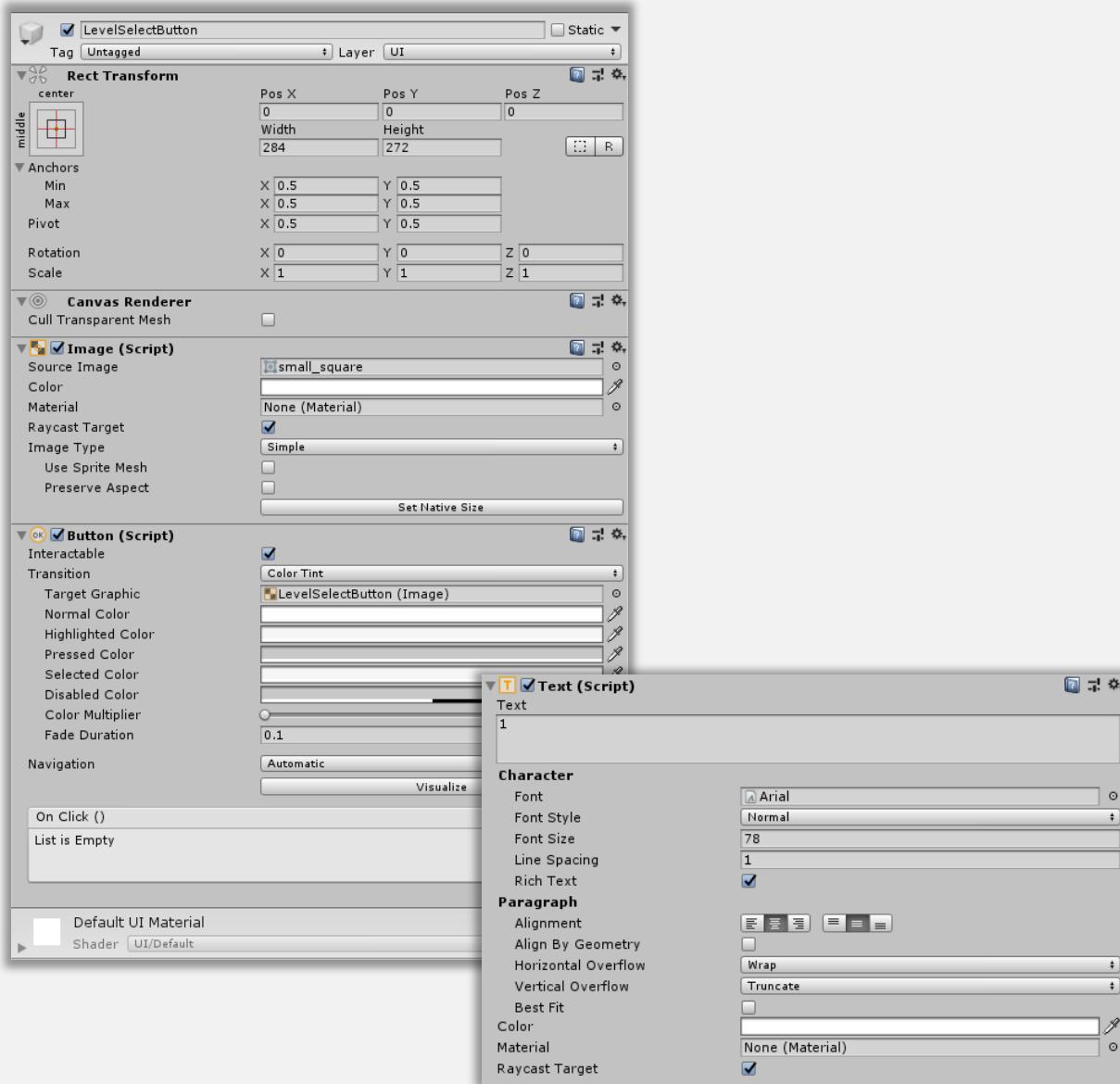


## 48 Change the Source Image of the button to **small\_square** and set the width to 284 and the height to 272.

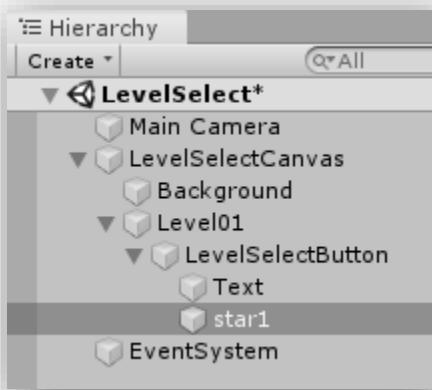
Change the button **Text** to "1", change the **color** to white, and make it large enough to fill the upper half of the button. A font size of 78 should work.



If the number disappears, adjust the size of the object rectangle until the number is visible.



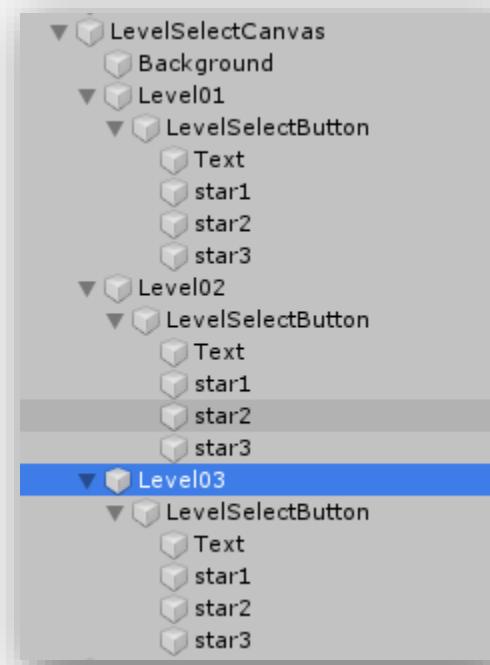
**49** In the button object, add **UI > Image**. Change the **Source Image** to **small\_star** and give it the name of “**star1**”. This will be the first of three stars. Adjust the image so that it fits under the number text and leaves room for two more stars.



- 50** Duplicate **star1** two times. Move each new star to the right or left of the first one so that all three can be seen. Rename the duplicate stars as “**star2**” and “**star3**”. If necessary, select all three stars and adjust their size and position.

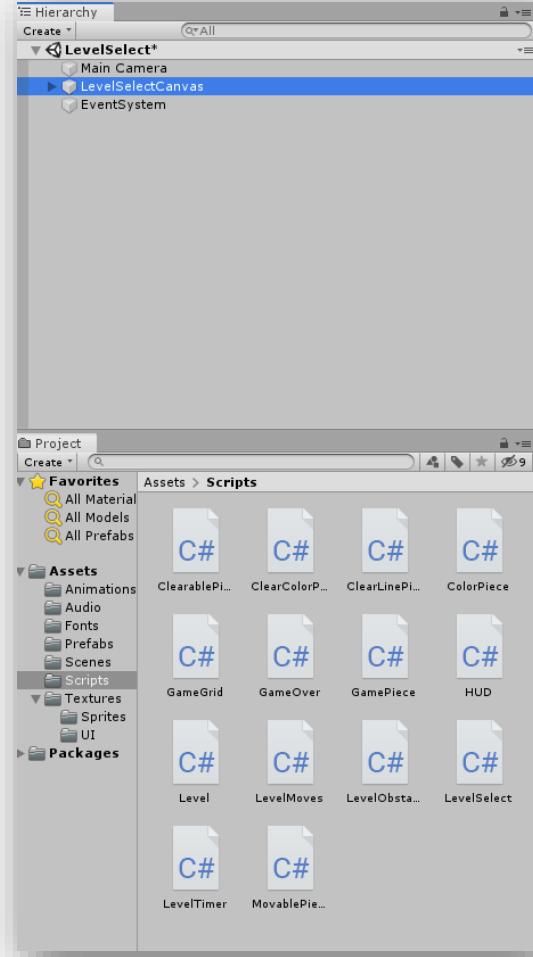


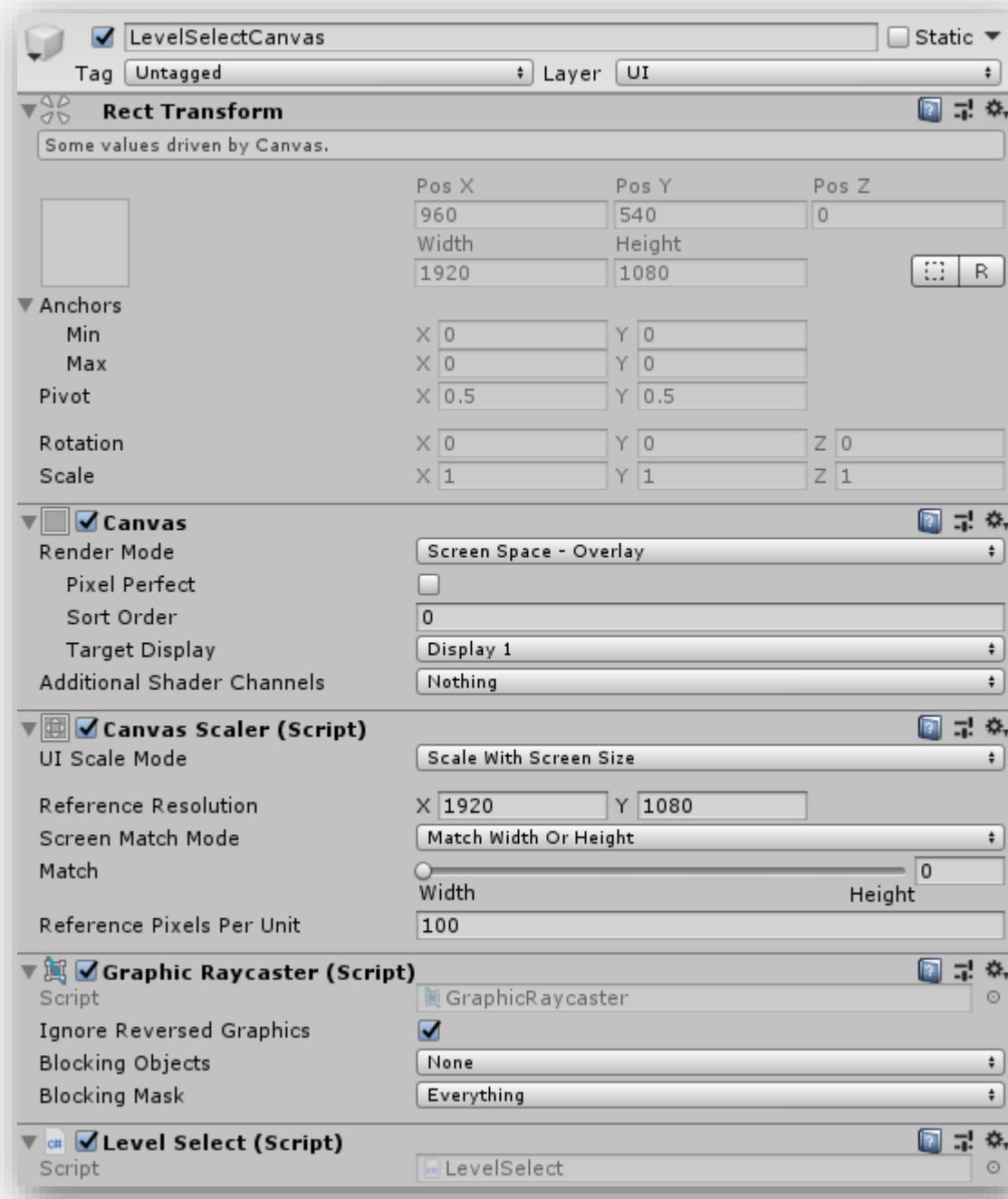
- 51** Currently this game has three levels so we will need three buttons. Select **Level01** and **duplicate** it twice. Move **Level01** to the left and the new buttons to the right of **Level01** and rename them as “**Level02**” and “**Level03**”. Change the **button text** for each of the buttons to the appropriate number, adjusting the size of the **Text** rectangle as needed.





**52** To make the **LevelSelect** scene work, we're going to need a script. In the **Scripts** folder, create a **new C# script** and give it the name of "**LevelSelect**". Attach this script to our **canvas** object by dragging it onto the canvas in the **Hierarchy**.





**53** This script is going to be used to load scenes, so we need to add [using UnityEngine.SceneManagement;](#) to the directives at the top:

```

1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.SceneManagement;
5

```

**54** Just like the buttons in the Game UI, we just need a function to call when `On Button Click ()` is activated. Add a function called `OnButtonPress()` and give it a **string** parameter “`levelName`”:

```
public void OnButtonPress(string levelName) { }
```

```
8 public void OnButtonPress(string levelName)
9 {
10 }
11 }
```

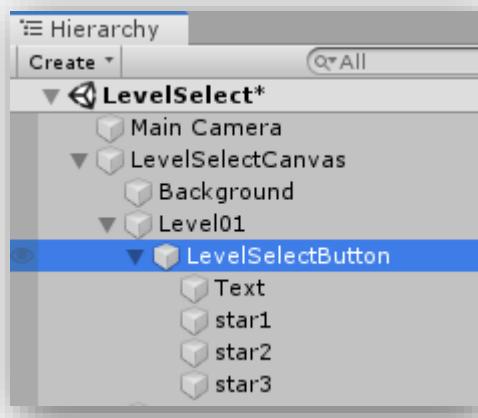
**55** Inside this function is where we will use the **SceneManager** to load a scene:

```
SceneManager.LoadScene(levelName);
```

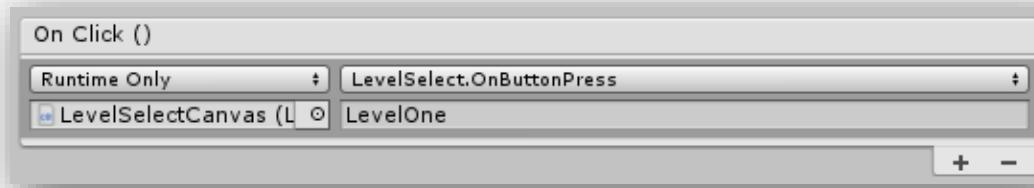
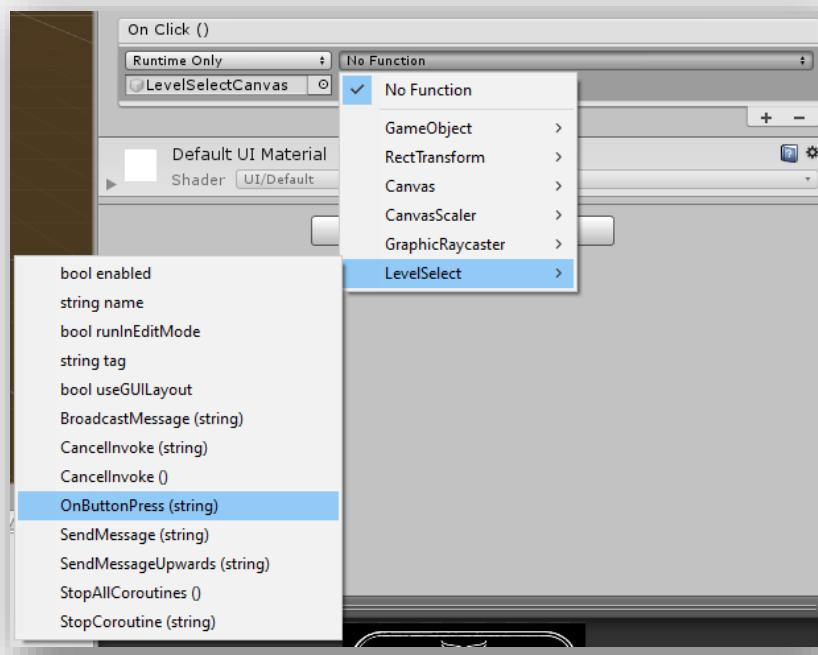
```
8 public void OnButtonPress(string levelName)
9 {
10 SceneManager.LoadScene(levelName);
11 }
12 }
```

**56** Save the script and go back to Unity.

**57** Select the first **LevelSelectButton** in the **Hierarchy**.

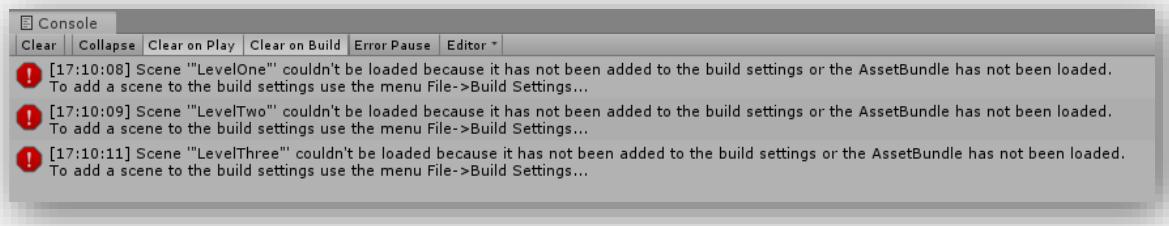


**58** Use the same steps that you used for the **Replay** and **Done** buttons to add the **LevelSelectCanvas** Object and have it call the **OnButtonPress()** function. Since this function requires a string parameter, we will need to specify the name of the first scene exactly as it appears in the Scenes folder: **LevelOne**.



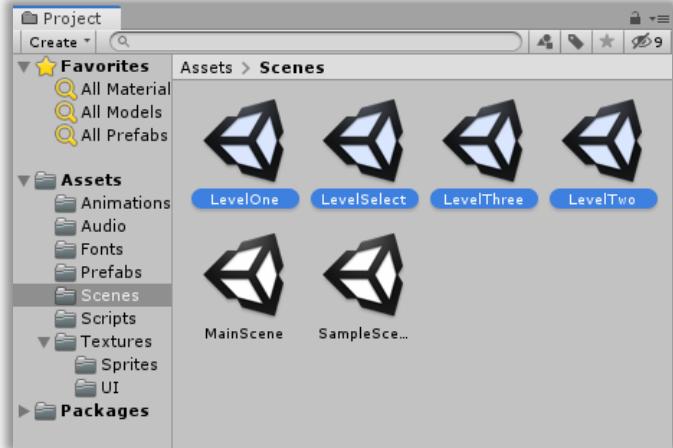
**59** Repeat these steps for the second and third buttons.

# 60 Try playing the game. If you click on a **button**, it will give you an **error**.

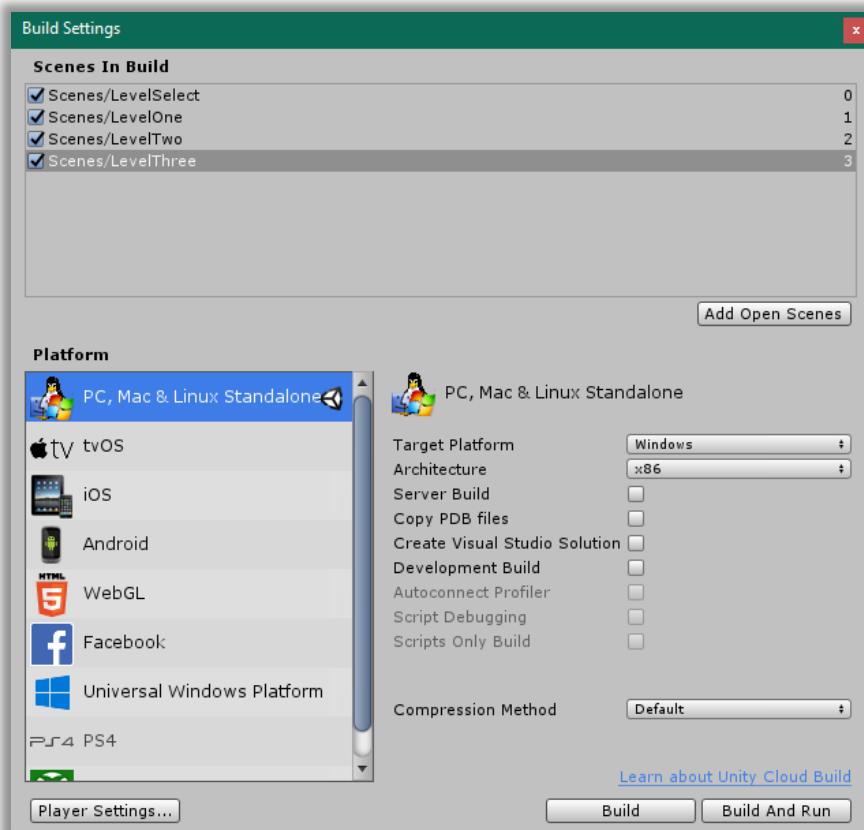


What's up with that?

Unity needs to have the **levels defined in the Build Settings**. Click on **File and Build Settings**. Drag the **three level scenes** and the **LevelSelect** scene from the **Scenes** folder into the **"Scenes In Build"** window.



**Rearrange** the scenes so that **LevelSelect** is first and then you can close the window.



---

## 61 Now we can get to the levels. How do we get back?

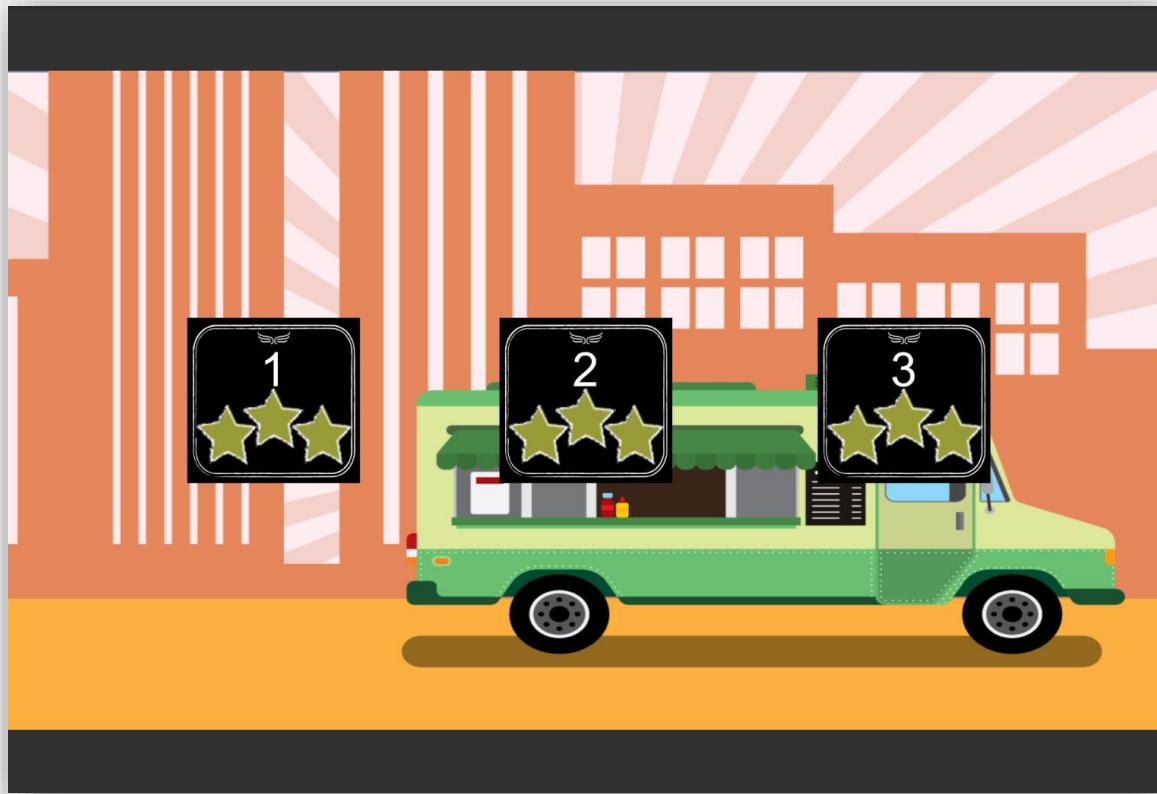
Open the **GameOver** script. Inside the `OnDoneClicked()` function, add:

```
SceneManager.LoadScene("LevelSelect");
```

```
83 public void OnDoneClicked()
84 {
85 SceneManager.LoadScene("LevelSelect");
86 }
```

---

## 62 Save the script. Now, when you play the **LevelSelect** scene, you can load any level and get back to the **LevelSelect** scene when you're finished.



## 63

We can save the player's progress and show it to the player when they play again.

To start, open the **HUD** script. We will need to use the **UnityEngine.SceneManagement** directive to keep track of the player's progress from each scene. Add this to the top of the script:

```
using UnityEngine.SceneManagement;
```

```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4 using UnityEngine.UI;
5 using UnityEngine.SceneManagement;
6
```

## 64

There is no need to check the saved progress if the player loses. Only if they win. Add this **if** statement to the **OnGameWin()** function:

```
if (starIndex >
PlayerPrefs.GetInt(SceneManager.GetActiveScene().name, 0))
{
 PlayerPrefs.SetInt(SceneManager.GetActiveScene().name,
starIndex);
}
```

```
100 public void OnGameWin(int score)
101 {
102 gameOver.ShowWin(score, starIndex);
103
104 if (starIndex > PlayerPrefs.GetInt(SceneManager.GetActiveScene().name, 0))
105 {
106 PlayerPrefs.SetInt(SceneManager.GetActiveScene().name, starIndex);
107 }
108 }
```

If the **starIndex** value is greater than what was previously saved for the **current scene**, we will save the current value of **starIndex** to the **PlayerPrefs** associated with the **name of the current scene**.

It helps to think of **PlayerPrefs** as just another variable, but one that gets remembered between scenes. The name of the current scene becomes the **key**, or variable name, for this.

---

## 65 Now to use this information in our **LevelSelect** scene. Open the **LevelSelect** script.

Our script needs to know the **PlayerPrefs** associated with each scene. To do this, we'll use a **struct**. In the **class**, set up the **variables** like this:

```
[System.Serializable]
public struct ButtonPlayerPrefs
{
 public GameObject gameObject;
 public string playerPrefKey;
}
public ButtonPlayerPrefs[] buttons;
```

```
13
14 [System.Serializable]
15 public struct ButtonPlayerPrefs
16 {
17 public GameObject gameObject;
18 public string playerPrefKey;
19 }
20 public ButtonPlayerPrefs[] buttons;
```

## 66 Save the script and in Unity, select the LevelSelectCanvas.

Find the **LevelSelect** script component in the **Inspector** and add each of the buttons and the names of the associated scenes to the array:

Buttons

Size: 3

Element 0

GameObject: LevelSelectButton

Player Pref Key: LevelOne

Element 1

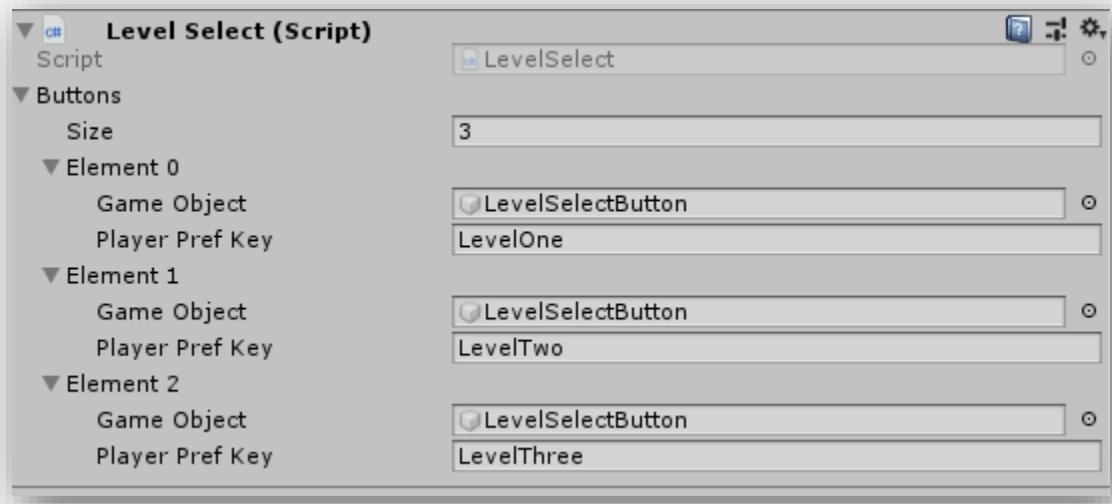
GameObject: LevelSelectButton (1)

Player Pref Key: LevelTwo

Element 2

GameObject: LevelSelectButton (2)

Player Pref Key: LevelThree



**67** Save this and go back to the **LevelSelect** script. In the **Start()** function, we will add two loops, one for all the buttons and one for each of the stars for each of the buttons. These are nested loops. The outside loop doesn't advance until the inside loop is finished.

The first loop checks against the number of objects in the **buttons[]** array.

```
for (int i = 0; i < buttons.Length; i++) { }
```

Inside that loop, set a new variable **score** to the value found in the **playerPrefKey** at that address.

```
int score = PlayerPrefs.GetInt(buttons[i].playerPrefKey,
0);
```

Below that is the second loop, which goes through the 3 possible **starIndex** values.

```
for (int starIndex = 1; starIndex <= 3; starIndex++) { }
```

At each of those values, 1 through 3, we will use **transform.Find** to identify the star by its number.

```
Transform star =
buttons[i].gameObject.transform.Find("star" + starIndex);
```

If the star number is less than or equal to **score**, it's visible. Otherwise, it's not.

```
if (starIndex <= score)
{
 star.gameObject.SetActive(true);
}
else
{
 star.gameObject.SetActive(false);
}
```

```
22 private void Start()
23 {
24 for (int i = 0; i < buttons.Length; i++)
25 {
26 int score = PlayerPrefs.GetInt(buttons[i].playerPrefKey, 0);
27
28 for (int starIndex = 1; starIndex <= 3; starIndex++)
29 {
30 Transform star = buttons[i].gameObject.transform.Find("star" + starIndex);
31 if (starIndex <= score)
32 {
33 star.gameObject.SetActive(true);
34 }
35 else
36 {
37 star.gameObject.SetActive(false);
38 }
39 }
40 }
41 }
```

**68** Save your work and play the game. Try to get at least one star in one of the levels and click on Done. You will see that the level button now shows that number of stars that you earned. Feel free to create additional levels with more challenging stars to earn!