CODE NINJAS®

Platinum Belt Ninja Guide
Activity 02: Codey Raceway

Codey Raceway

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

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

Plan and Design

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

Mario Kart Tour by Nintendo

Fall Guys by Mediatonic
Built in Unity

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

Ninja Planning Document

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

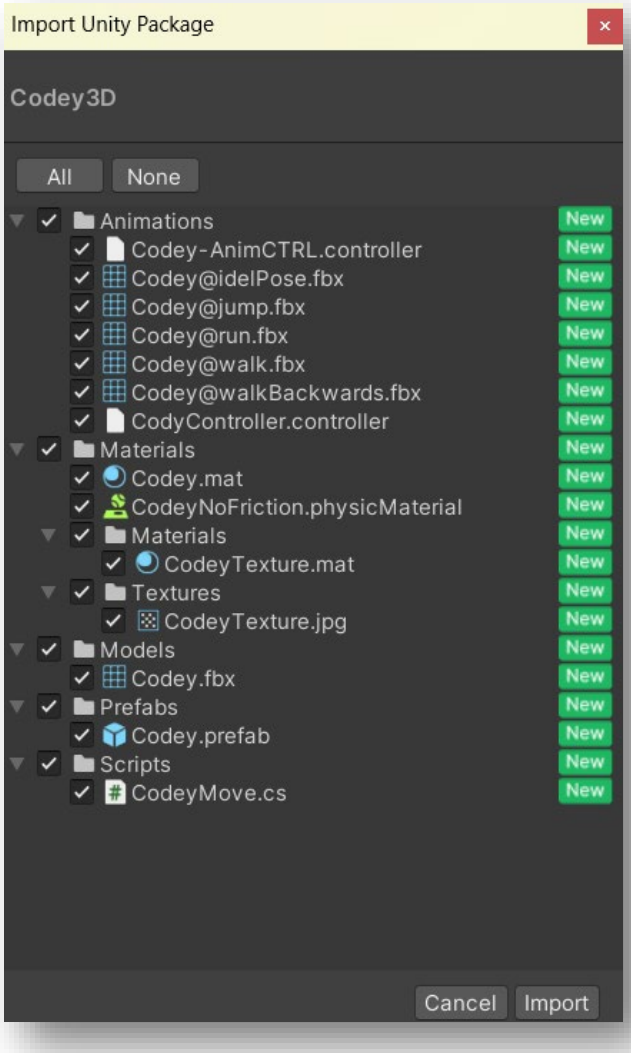
Take a look at the sample projects below for some inspiration!

Project Setup

1 Start a new Unity Project and name it *YOUR INITIALS – Codey Raceway*. Select **3D template**.

2 After it loads, rename the Sample Scene to Codey Raceway.

3 Import **Activity 02 - Codey3D.unitypackage**. This is a Codey **game object** that can run around in 3D space.



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

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

Codey is missing...

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

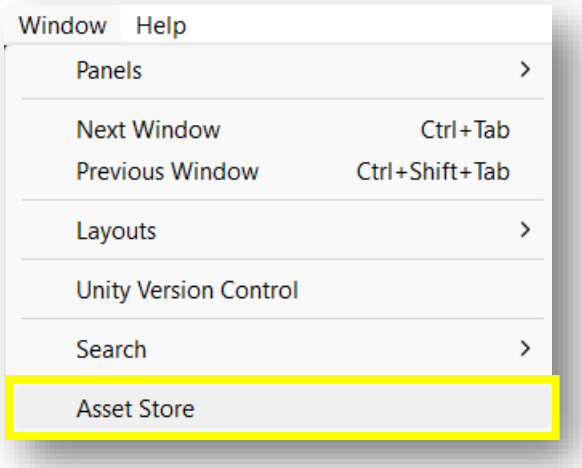
Sensei Stop

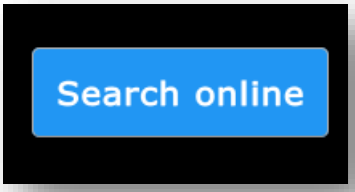
Using the list above, add the required components to the Codey model. If you get stuck, work with your Code Sensei to add the components that you are

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

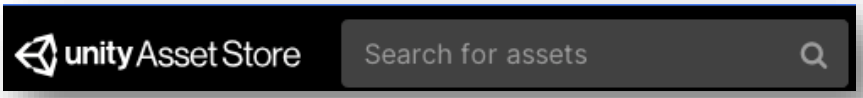
Now, with Unity you have a great way to find assets that are free to use in the **Unity Asset Store**! You can find all kinds of amazing sprites, music, materials, and more to include in your game!

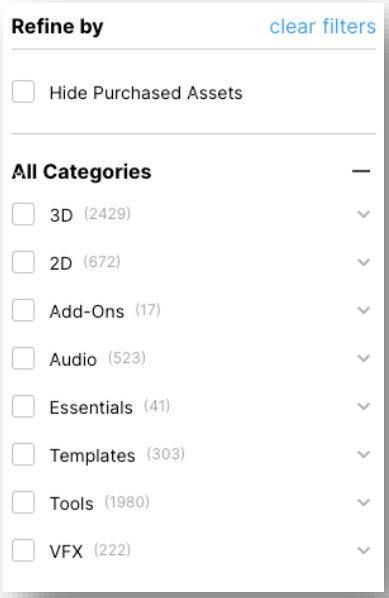
Open the **Unity Asset Store** by clicking on **Window** and then **Asset Store**. This will place an Asset Store tab in your Unity project so you can easily access it at any time. After you go to the asset store Tab, click **Search Online**.





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



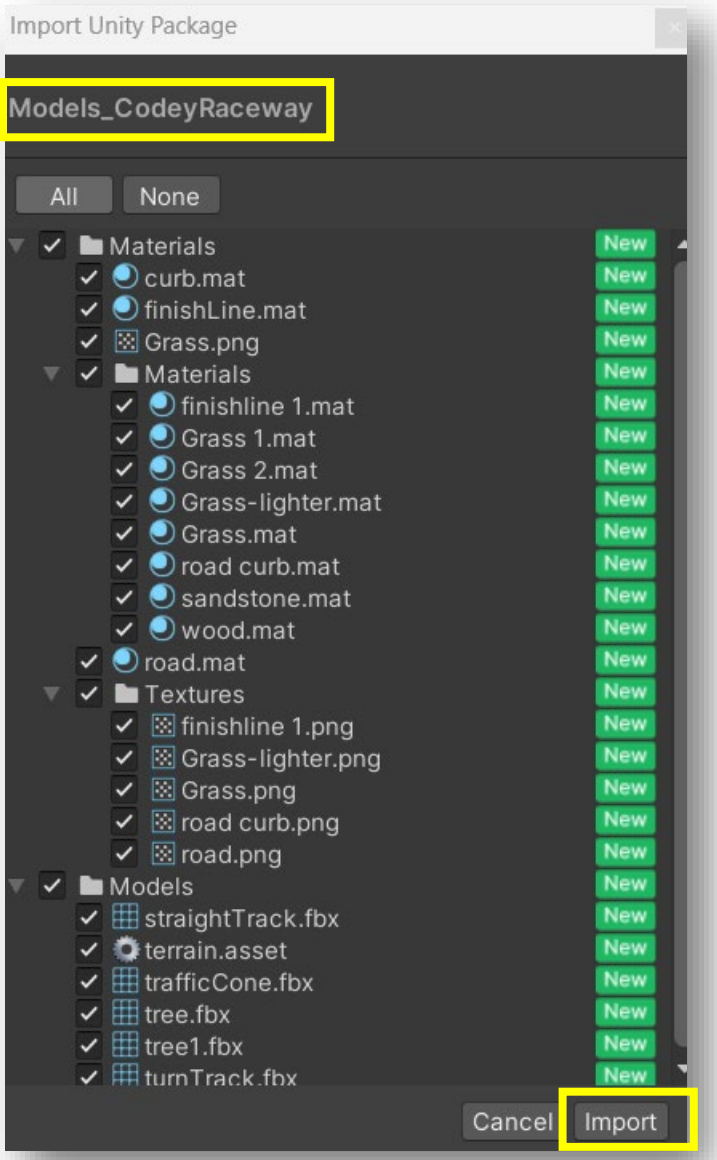


Imported Assets

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

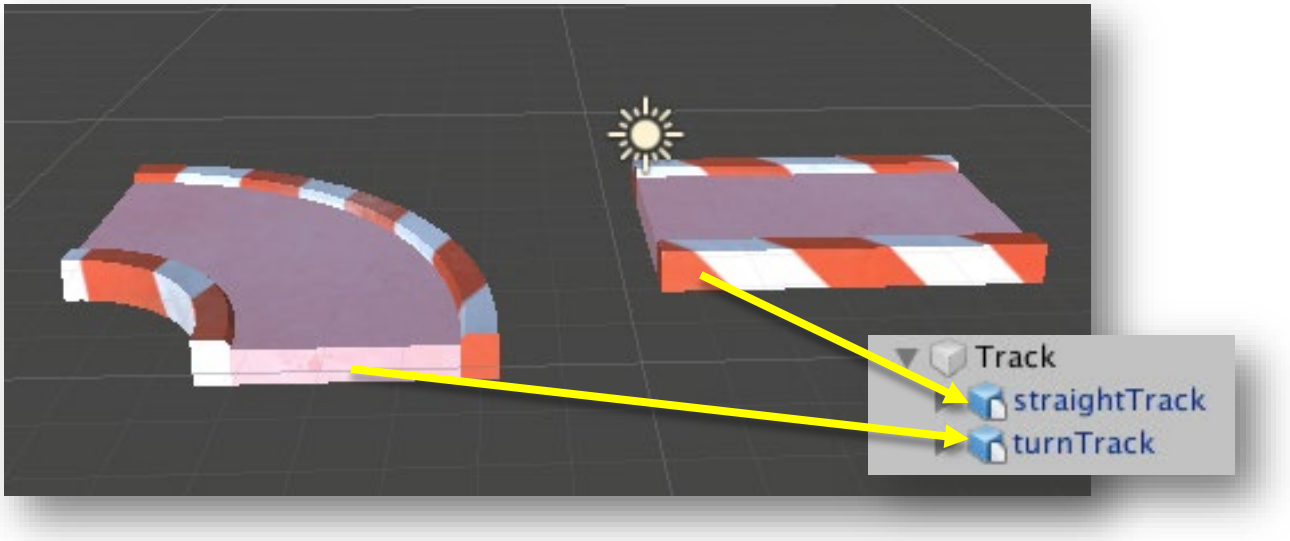
8 For this game, you have two options to customize your world. The next section will show you how to import **assets** from Code Ninjas to build your track. The following section will show you how to use the Unity Asset Store to find **assets** to use in your game.

9 Import the **Activity 02 - Models_CodeyRaceway.unitypackage**.



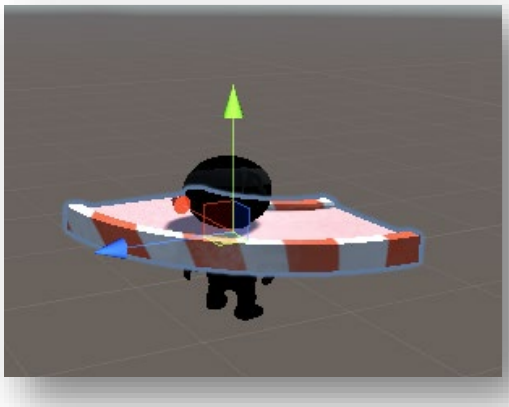
10 Create an empty game object and rename it **Track**. Use this game object to keep your track objects organized.

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



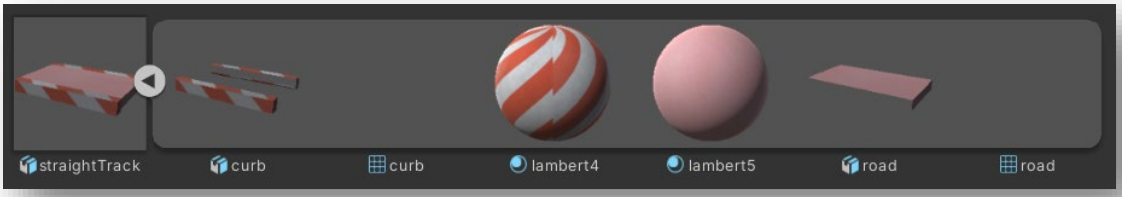
12 Before moving forward, select either the **straightTrack** or the **turnTrack** piece in the **Hierarchy**. Notice how none of the track pieces have a collider. What does this mean? Let's find out!

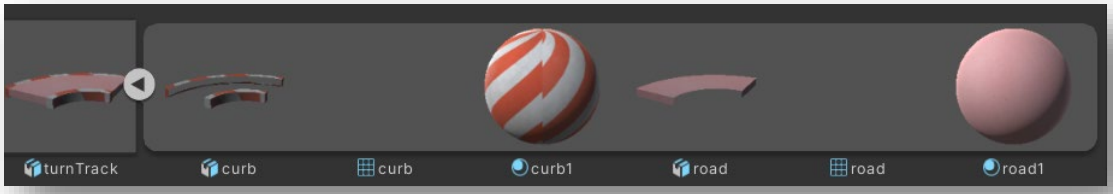
Drag Codey from the **Models** folder on top of one of the track pieces and see what happens.



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

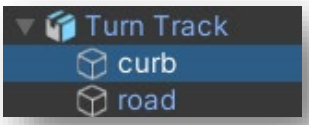
13 In the **Hierarchy** both the turnTrack and the straightTrack have two parts to them. Look in the **Models** folder and expand either the **straightTrack** or the **turnTrack**.

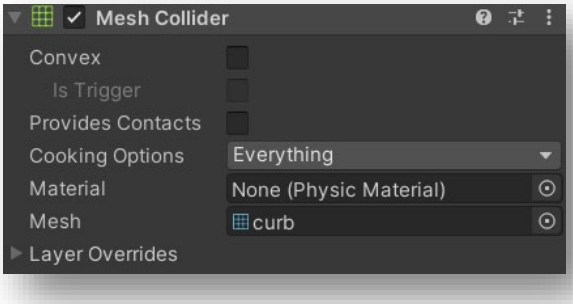




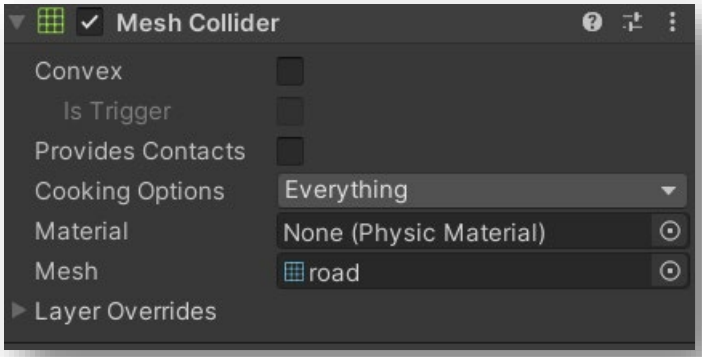
Both tracks have their own smaller parts that make up this one piece. Using the **Mesh Collider,** you will be able to form a collider for the **curb** and **road** parts.

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



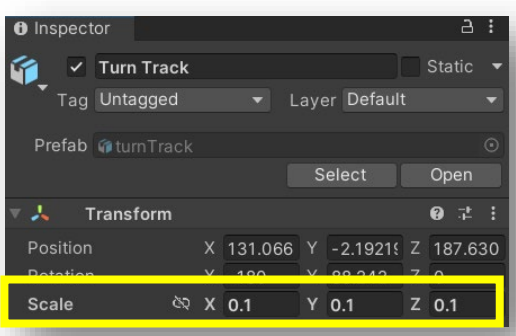


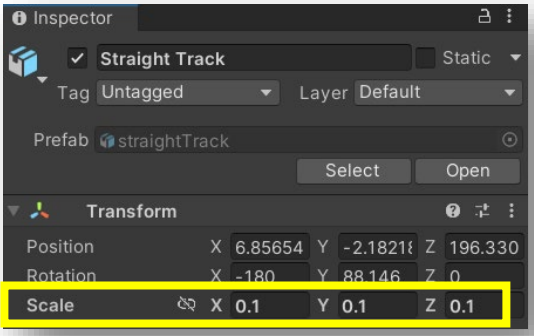
15 Follow the same process and add the **Mesh Collider** to the **road game object**.



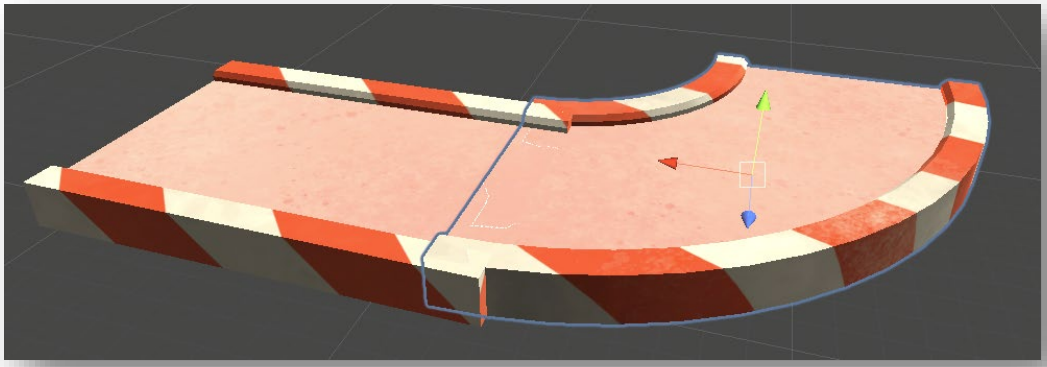
16 Playtest your game. Codey no longer goes through the game objects!

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



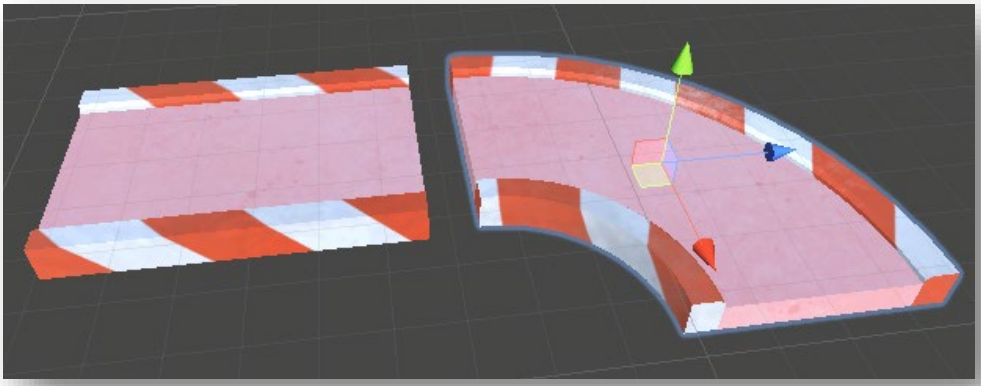


18 The next step is to get the pieces to look like one complete part. If you just use the **Move Tool**, it is hard to tell if pieces are overlapping one another.



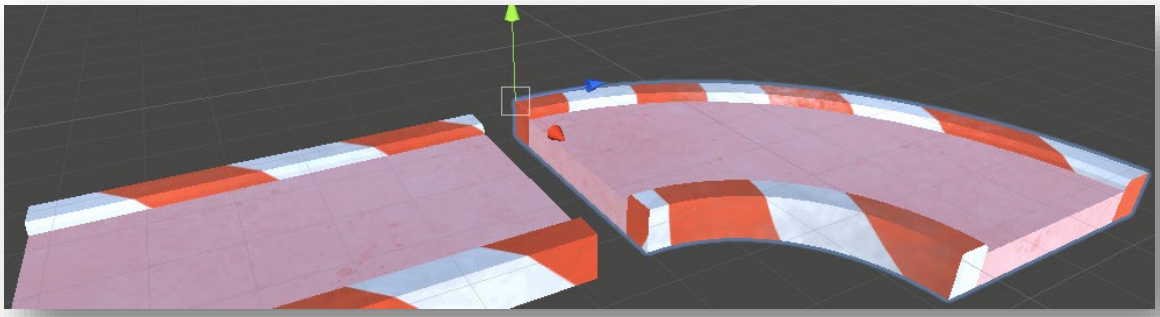
Unity has a great way to help us piece our tracks together – called the **Vertex Tool**. With the Vertex Tool, you can trigger a command in Unity using your keyboard and mouse.

First, select the piece you want to move.



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

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

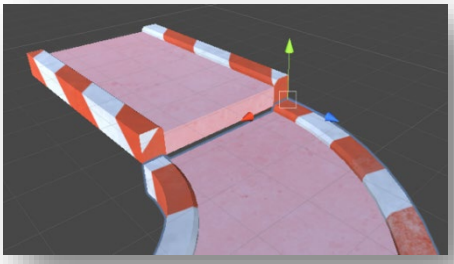


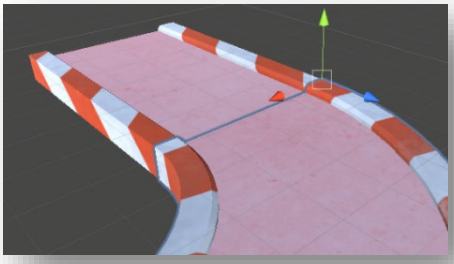
The two track pieces should now fit together without any overlapping parts!

Track Tips

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

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



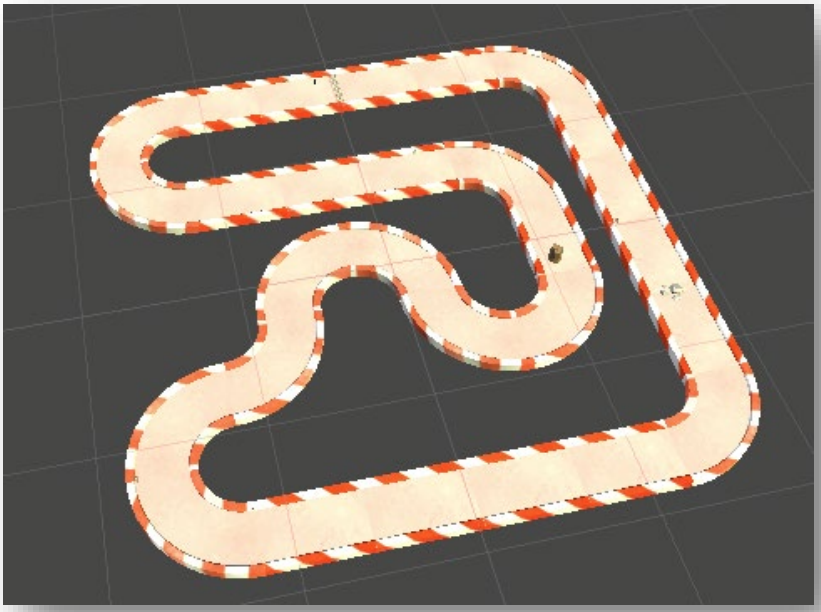


22 Duplicate the **turnTrack** and **straightTrack** in the **Hierarchy** for additional road pieces so you don't have to add the **Mesh Collider** component to track pieces every time!

Sensei Stop

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

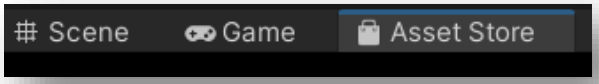
23 Have fun building your track! Take a look below to see the example track what we made.



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

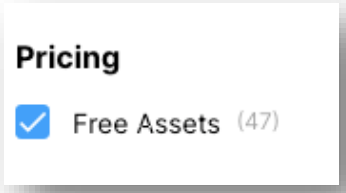
Shopping Spree

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

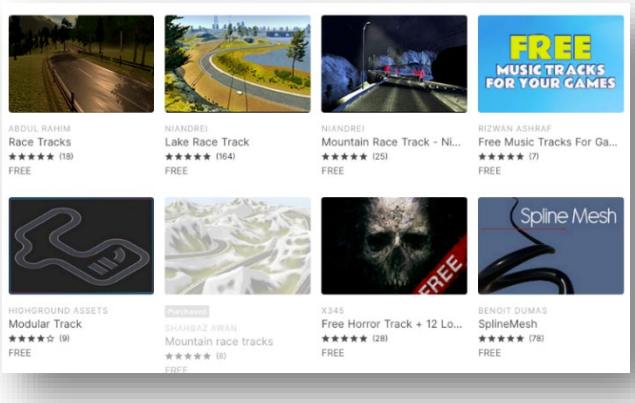


25 Search for “tracks” and press **Enter**.

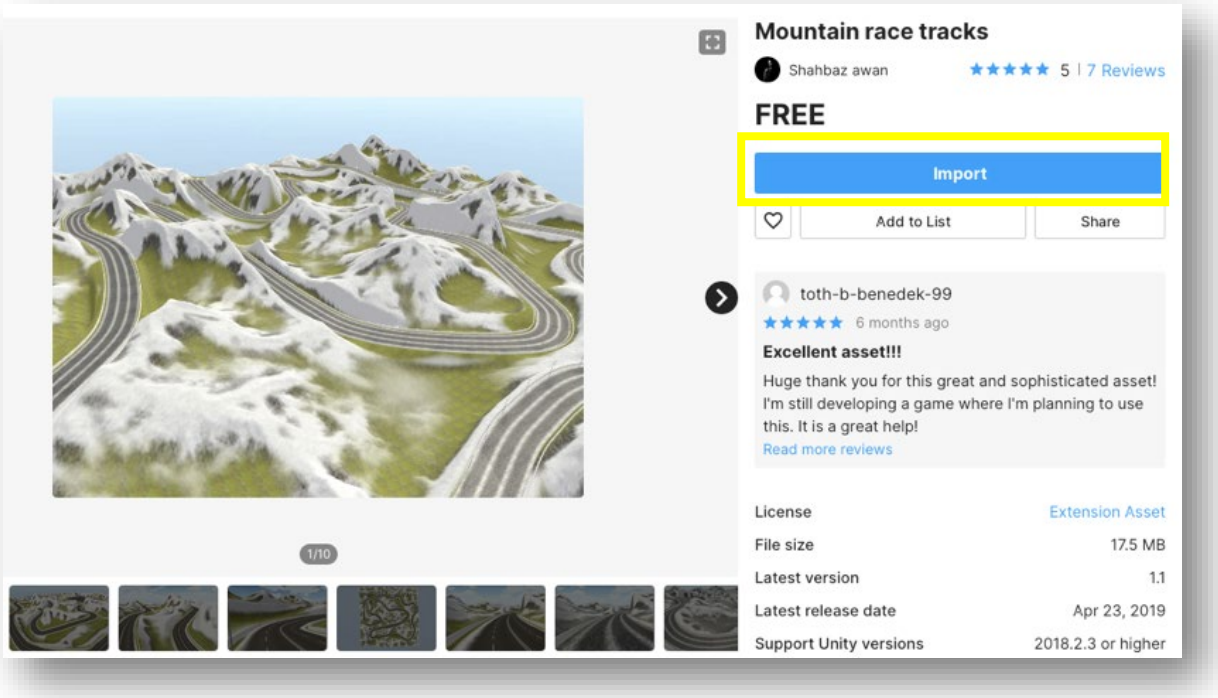
26 On the right side, scroll down to find **Pricing** and check **Free Assets**.



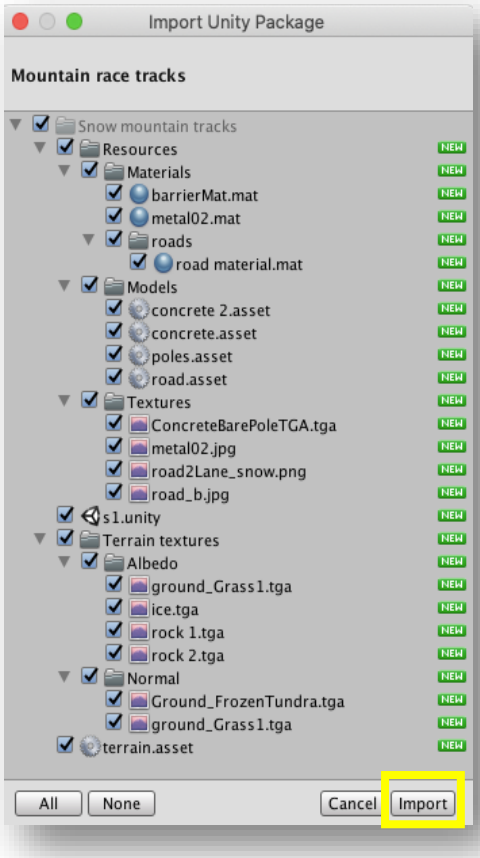
27 You should see a few tracks you can download to your project! Look for one that you like and click on it.



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

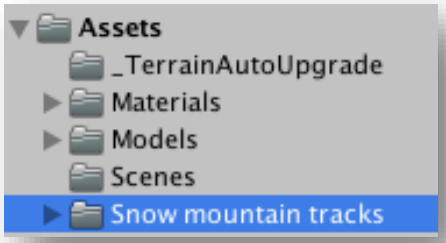


29 An import pop-up will appear. Select **All** then **Import**.



It might take some time to add the entire package.

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

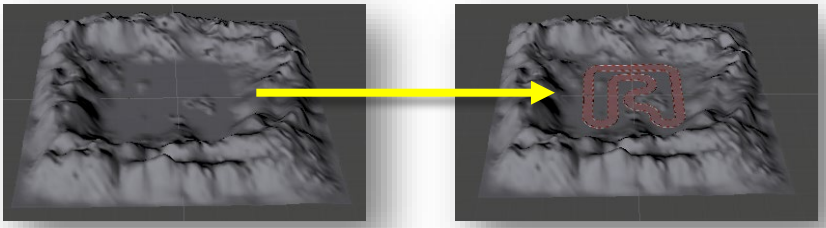


Each package is different. Go through the folder and use the assets you like the most for your track.

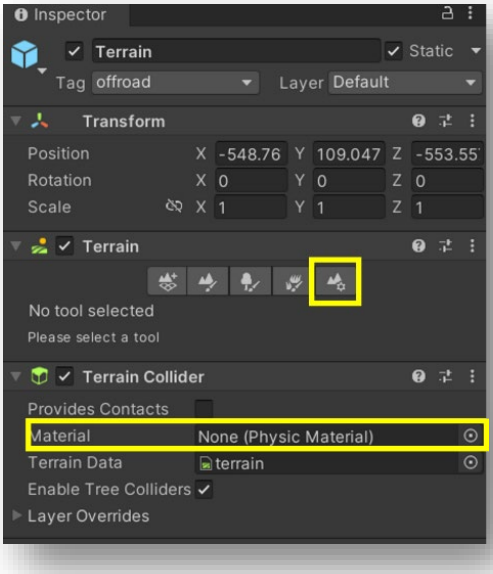
Moving Mountains

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

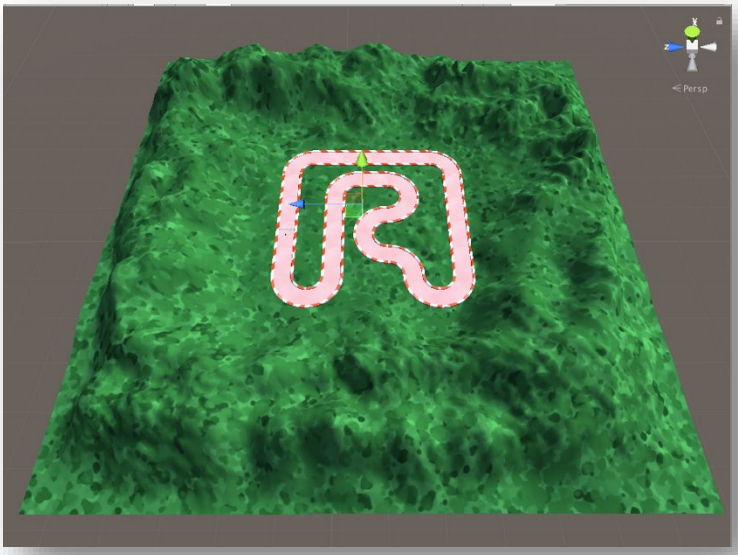
32 The **terrain** can be used to give your track an extra layer of creativity. Adjust the placement of it so the track is above the terrain.



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

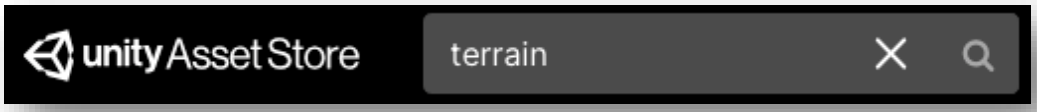


34 You can create a new **material**, use an existing **material**, or find a **material** in the **Unity Asset Store**. In the example below, we used the grass **material**!



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

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

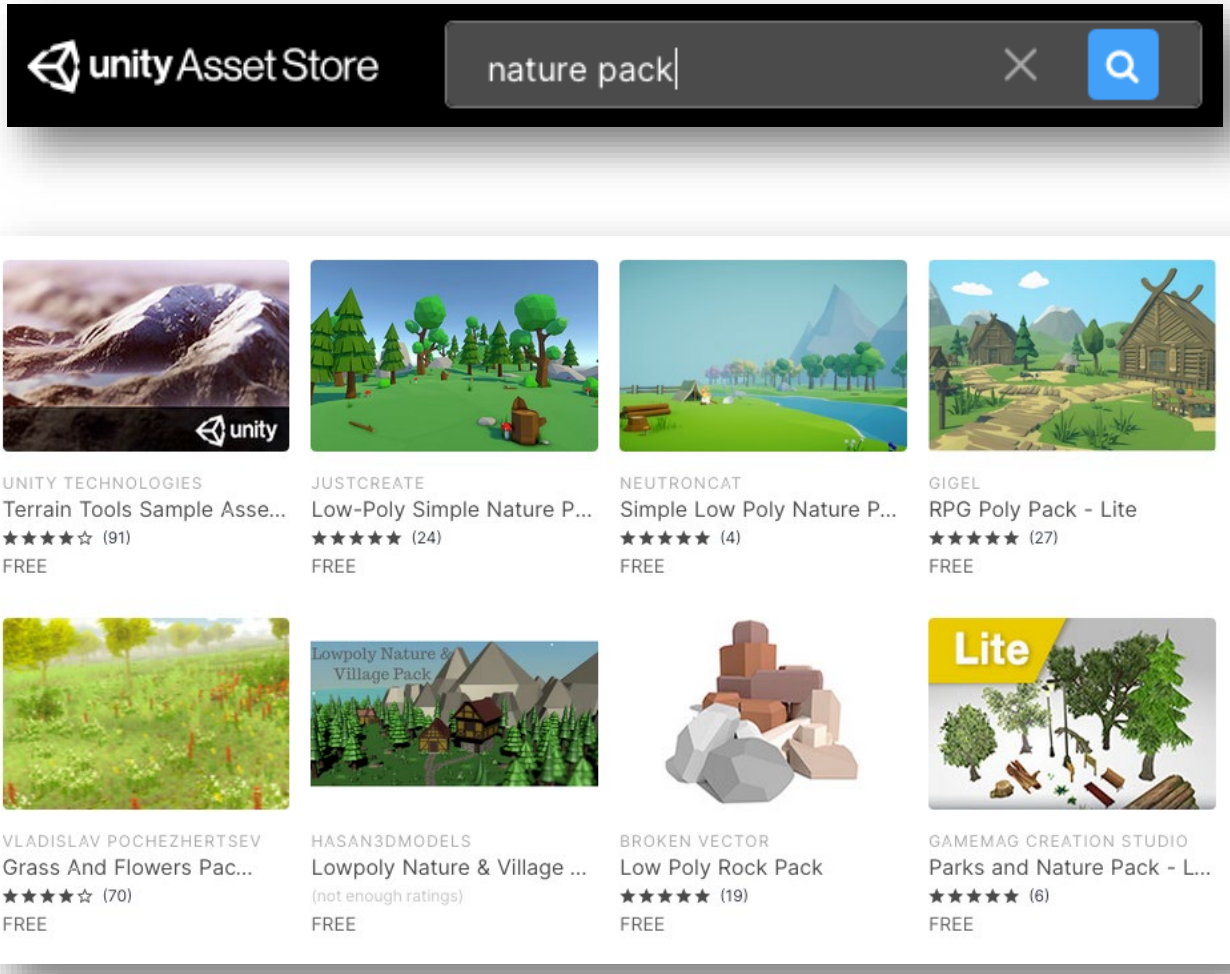


Assets

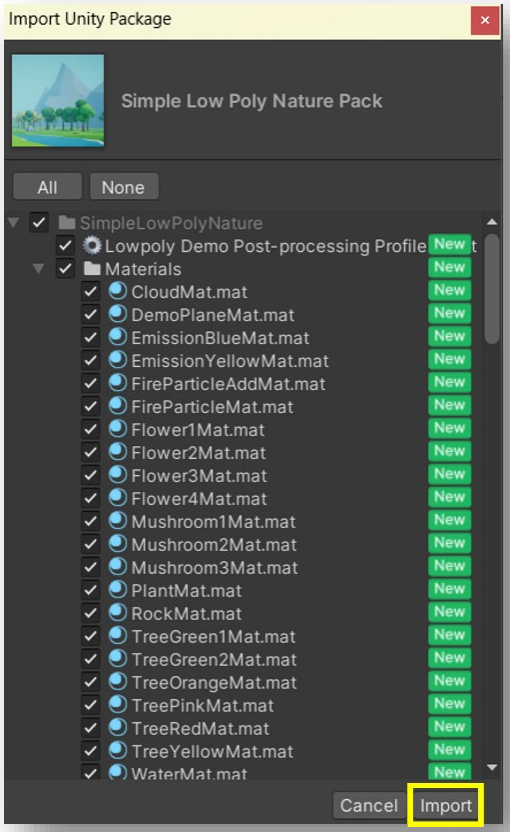
Some of the packages may have more than what you are looking for. For example, the track we imported already came with a terrain.

Exterior Decorating

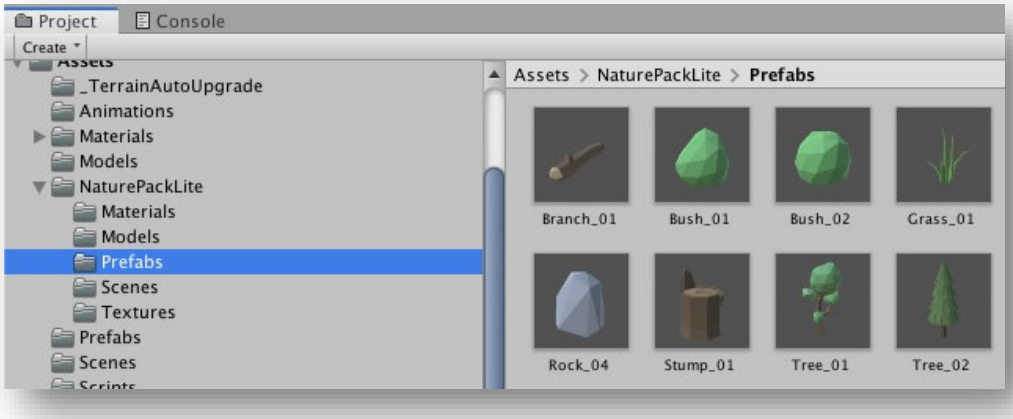
36 Let's make your scene even more unique! Use the **Unity Asset Store** to add more to your scene. Start by searching for **nature pack** and selecting **Free Assets** in the **Pricing** category.



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

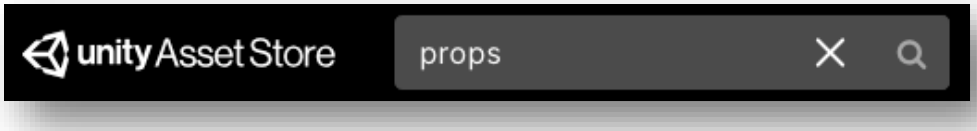


38 You will find a new folder in the **Project** tab. In this folder there will be **prefabs** that you can add into your **scene**.



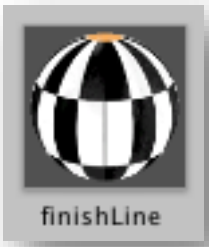
39 To spark your creativity, take a look at what we added to make the world more unique!

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

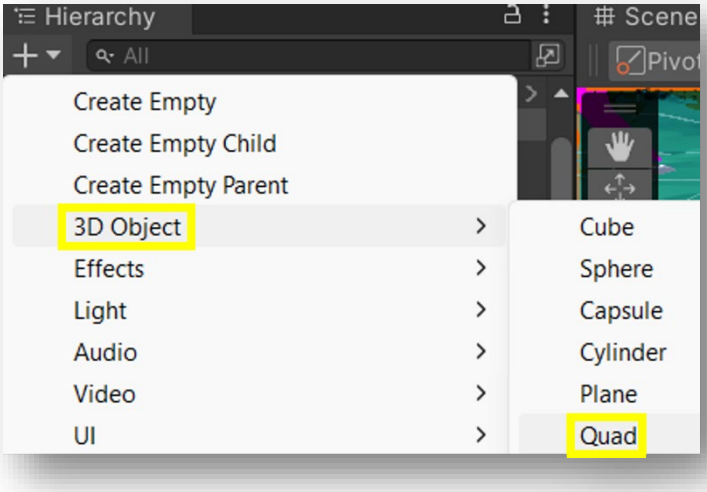


The End of the Road

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

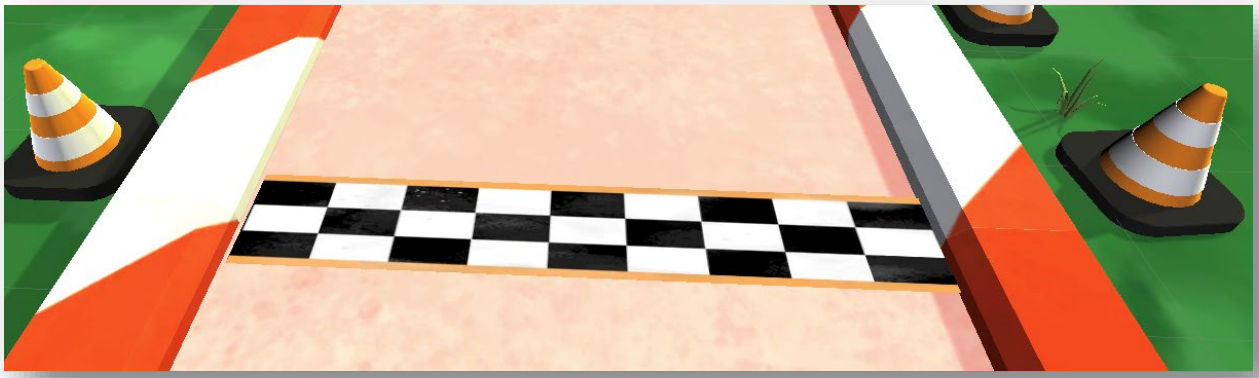


42 This material looks the best on a **Quad game object**. Add a new **Quad game object** to your **scene**.



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

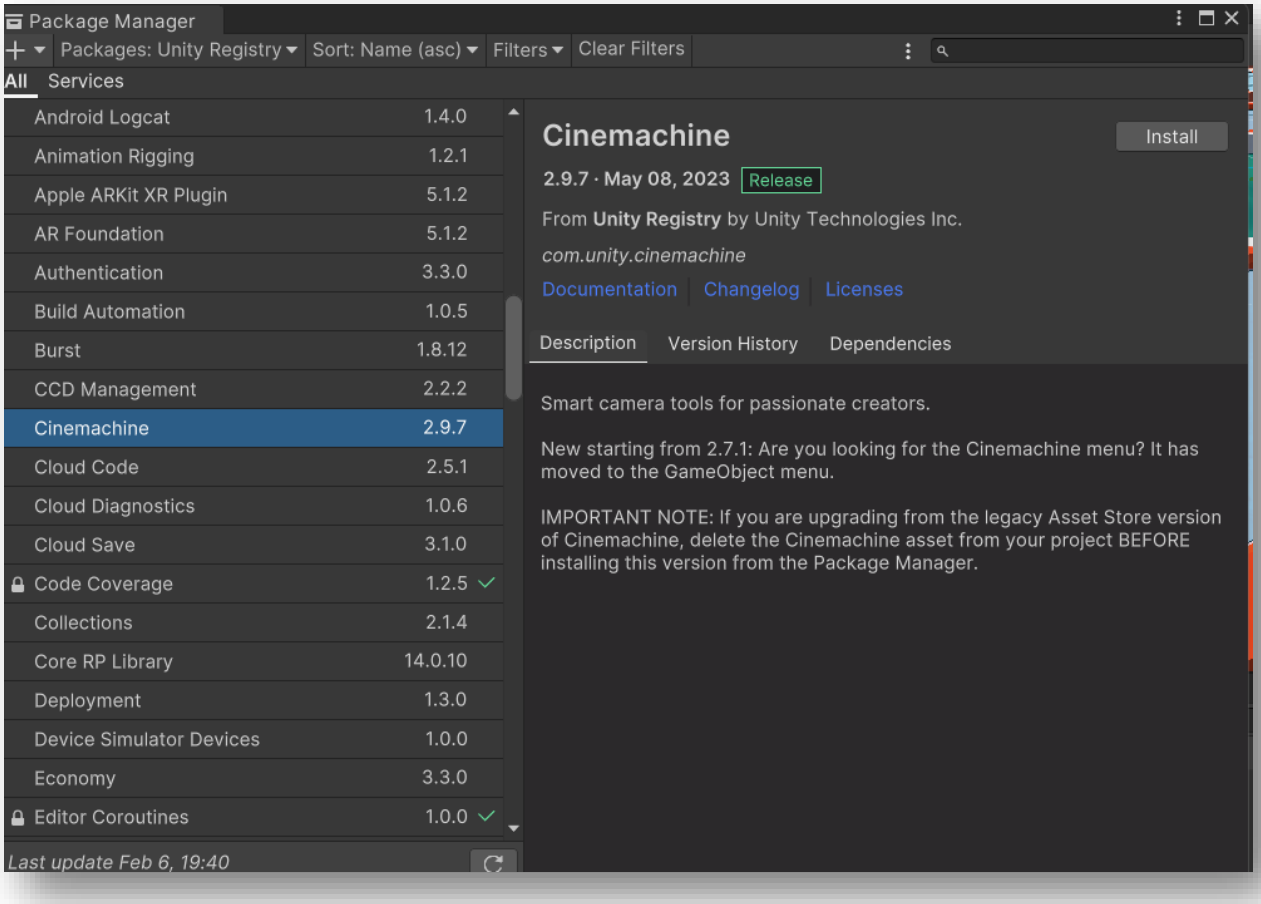
Change the **position**, **size**, and **rotation** of the Quad so it is somewhere on your track. Drag the finishLine **material** onto the Quad. Rename the **game object** to Finish Line.



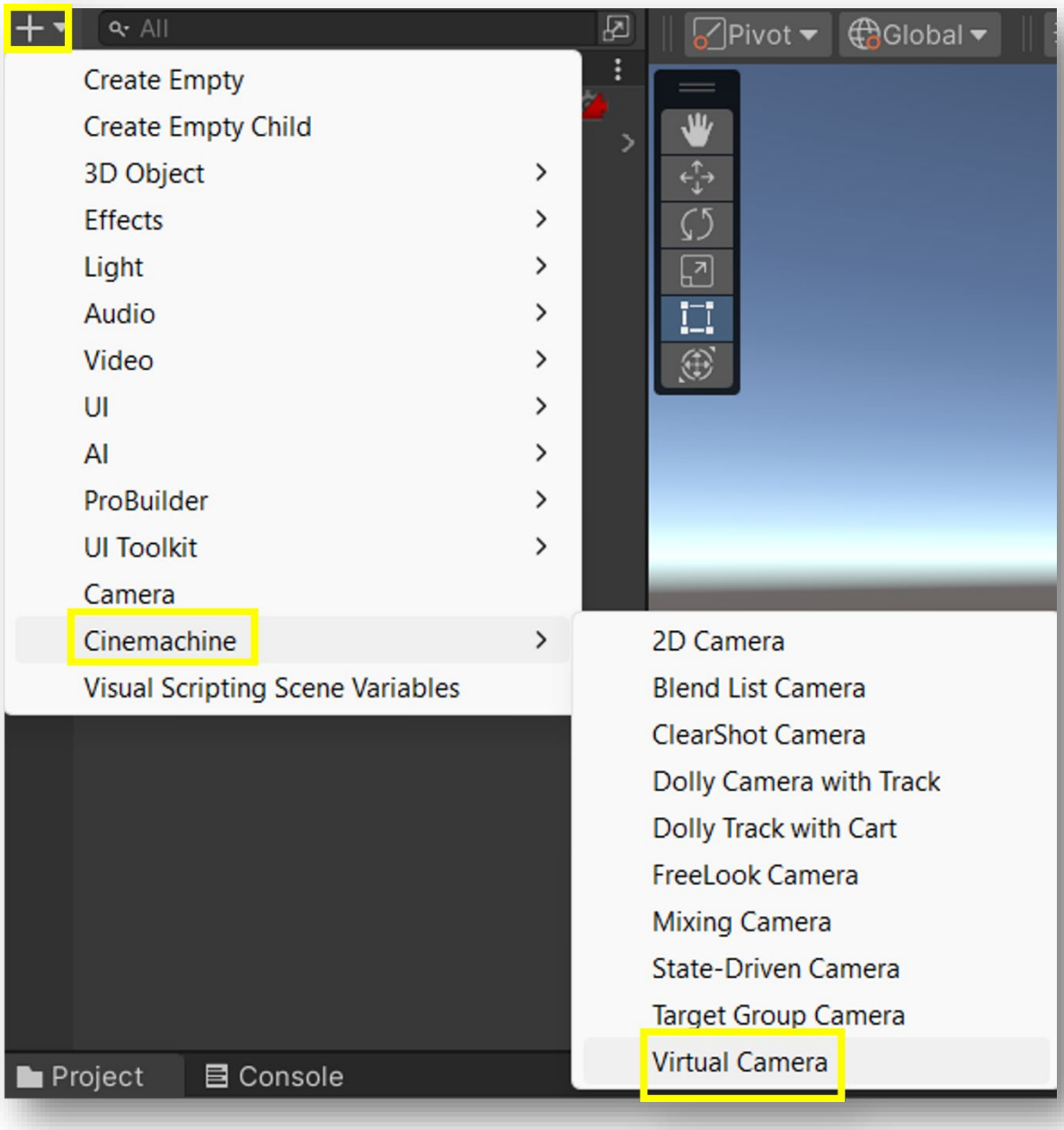
Notice how the finish line looks like part of the road. That's how you'll want yours to look when you're done.

Eyes on the Road

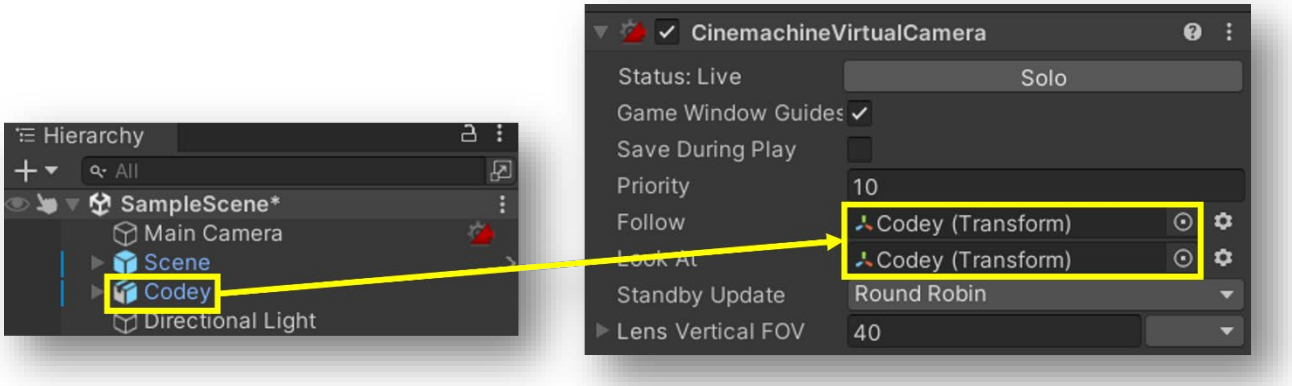
44 Generally, in a game we want the camera to follow the main character. This section will teach you how to make the camera follow behind Codey. To do this, we will use a **Cinemachine**. In the tabs, click on **Window** followed by **Package Manager**. Make sure to click on **Packages** under the **Package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



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



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

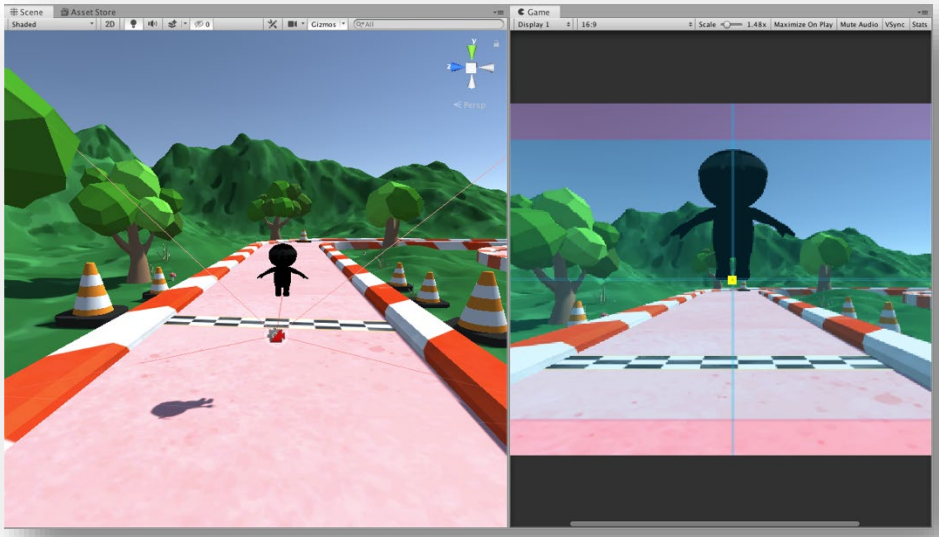


Make sure you have the **CM vcam1 game object** selected. Notice that the camera is always behind the player?

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

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

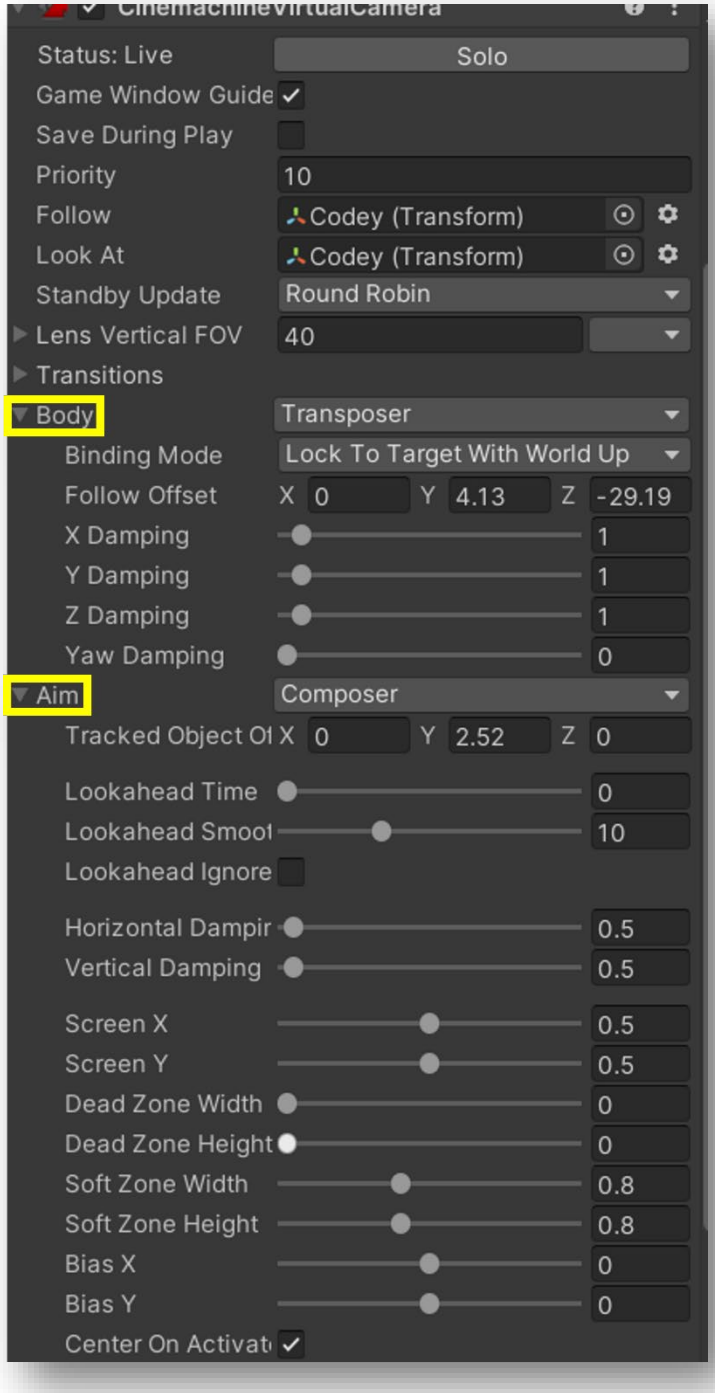
48 Move your **Game Tab** to the right of your **Scene Tab** so you can see both at the same time.



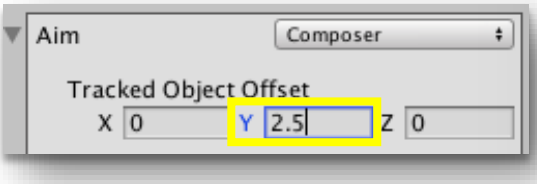
49 While the **CM vcam1** object is still selected in the **Hierarchy**, you should notice a small yellow square in the **Game Tab**. This square tells our camera where to look.

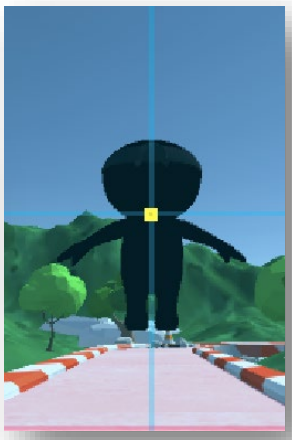
See how the yellow square is at Codey's feet? This is what makes the camera's view so low. Since we don't want the camera to always look at Codey's feet, we will adjust it to look between Codey's body and head.

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

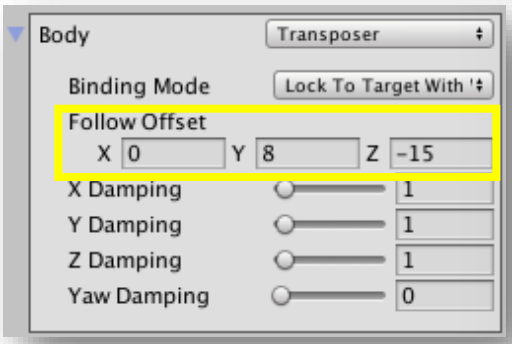


51 Adjust the **Aim** first. This allows you to move the yellow square between Codey's body and head. Since the square is already in the center, we do not want to move it in the **X** or **Z** directions.



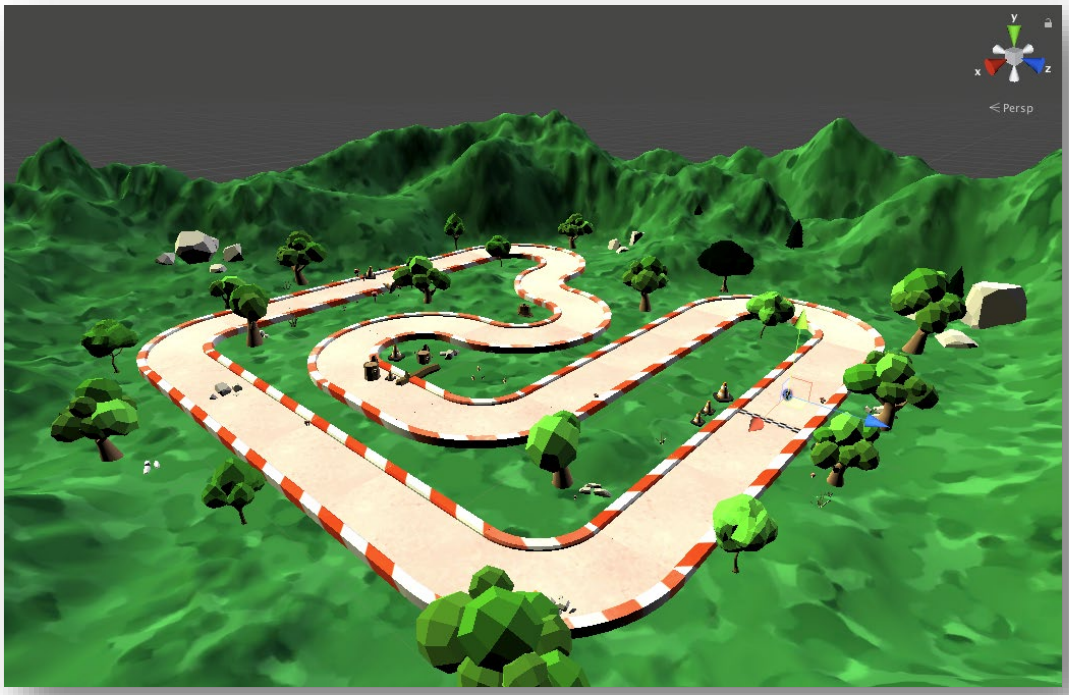


52 The **Body** component allows you to zoom in or zoom out of the view. Adjust the three Follow Offset values.



Lap it Up

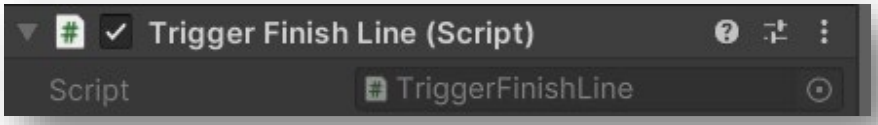
53 At this point, you should have Codey and your **scene** set up! Take some time to play your game, making sure all the game objects in the scene are placed where you want them. Feel free to go back into your scene and make any changes.



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

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

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



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

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

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

    }
}
```

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

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

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

        }
    }
}
```

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

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

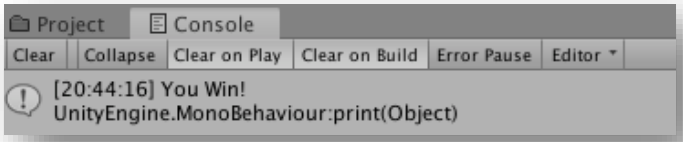
59 Save your **script** and return to Unity. Playtest your game and try to trigger the finish line. Check to see if the "You Win!" message shows up in your console.

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

Sensei Stop

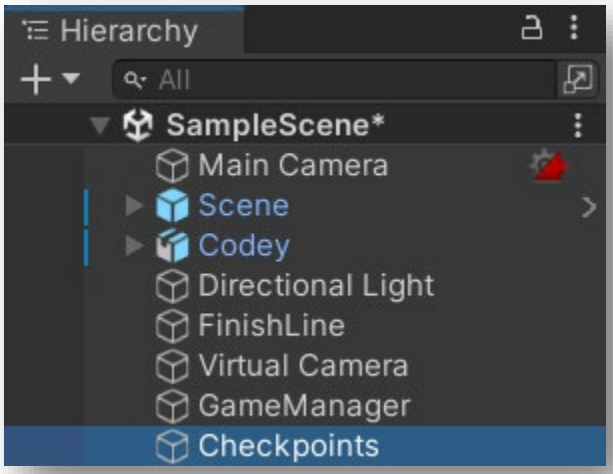
Discuss with your Code Sensei and share why you think the message inside the OnTriggerEnter function did not appear. Change the finish line game object to make the message appear.

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



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

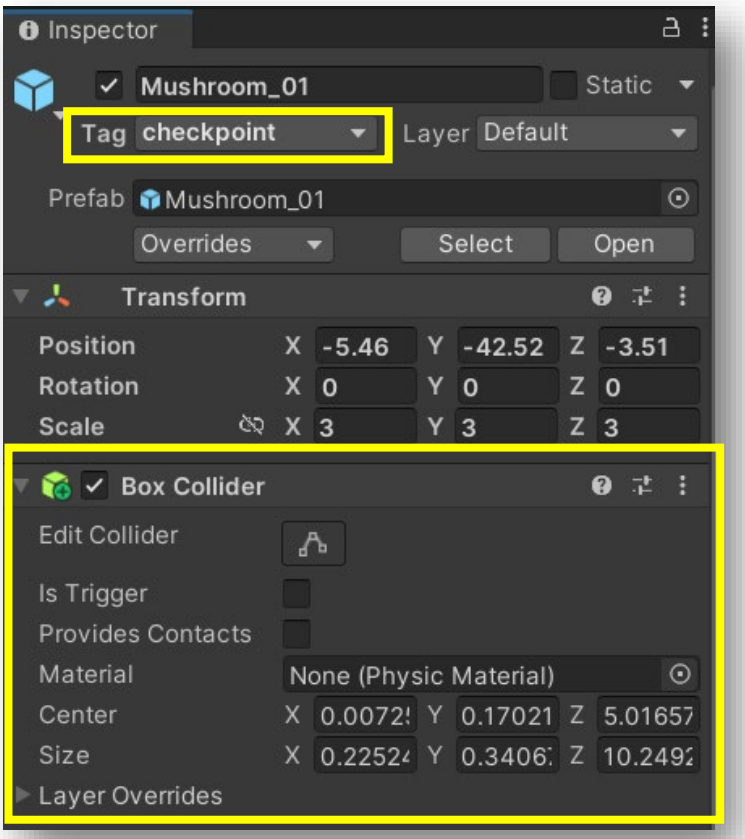
Create an empty game object and rename it **Checkpoints**. We will use this object to organize the checkpoints. You can use checkpoints in any game to make sure your player goes through a specific path while playing!



62 Our example will use the **Mushroom_01** prefab. You can find this object in the **NaturePackLite's Prefabs** folder.

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

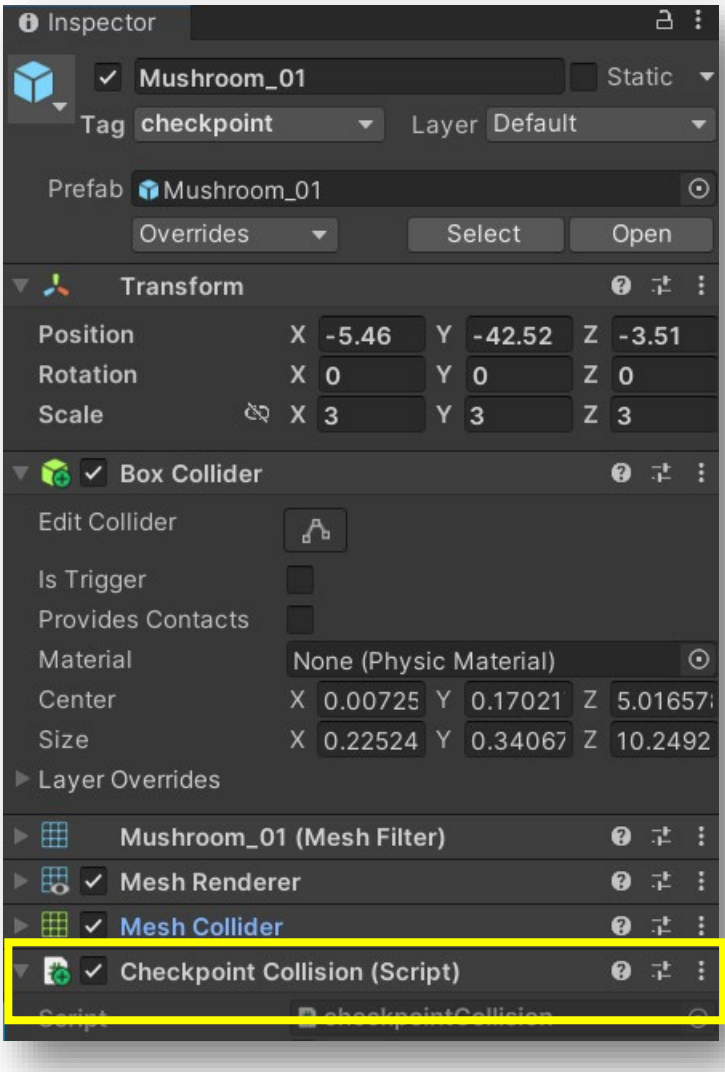
63 Give your game object the **checkpoint** tag and attach a **Box Collider**.



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

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

66 Attach this script to the object you are using as the first checkpoint. Remember, we used the **Mushroom_01** object in our example, but you could pick any object you want.



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

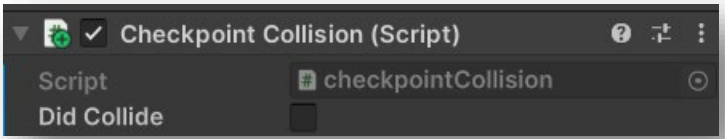
```
public class checkpointCollision : MonoBehaviour
{
    public bool didCollide;

    0 references
    void Start()
    {

    }
}
```

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

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

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

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

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



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

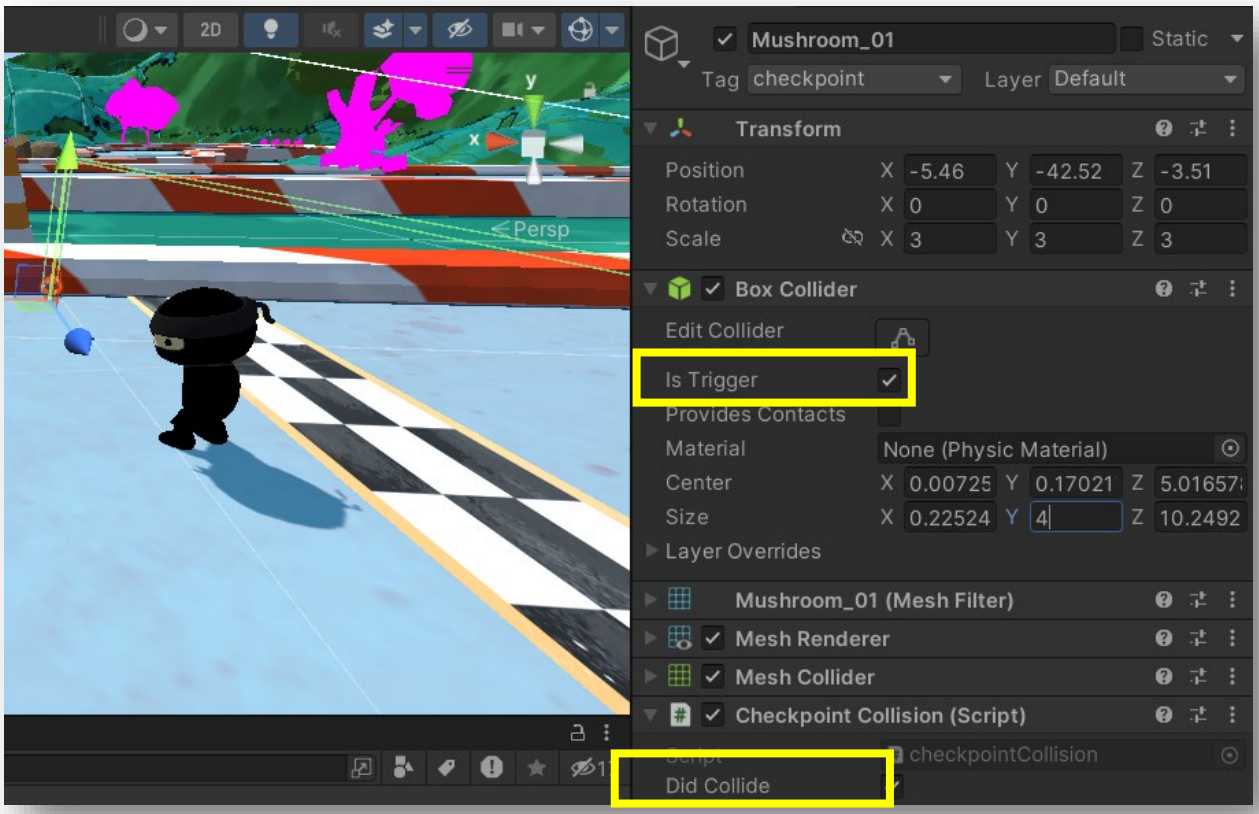
```
public class checkpointCollision : MonoBehaviour
{
    public bool didCollide;

    0 references
    void Start()
    {
        didCollide = false;
    }

    private void OnTriggerEnter(Collider other)
    {

    }
}
```

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

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

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

    0 references
    void Start()
    {
        didCollide = false;
    }

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

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

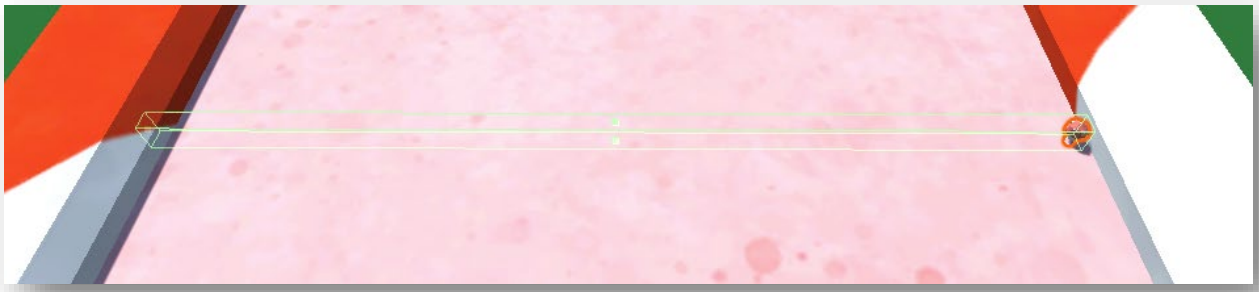


Sensei Stop

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

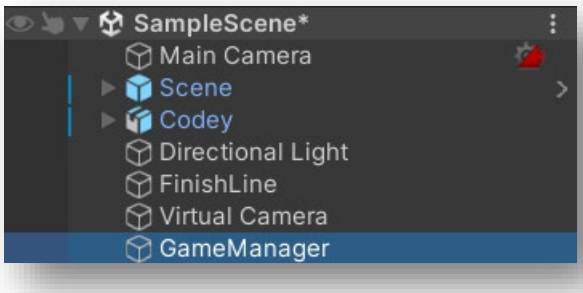
73 If we only have one checkpoint, players can still cheat. Add at least **3 more checkpoint triggers** throughout your track. To make this easier, duplicate the game object you are using as the checkpoint in the **Hierarchy**.

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



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

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



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

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

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

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

    }
}
```

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

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

    }
}
```

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

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

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

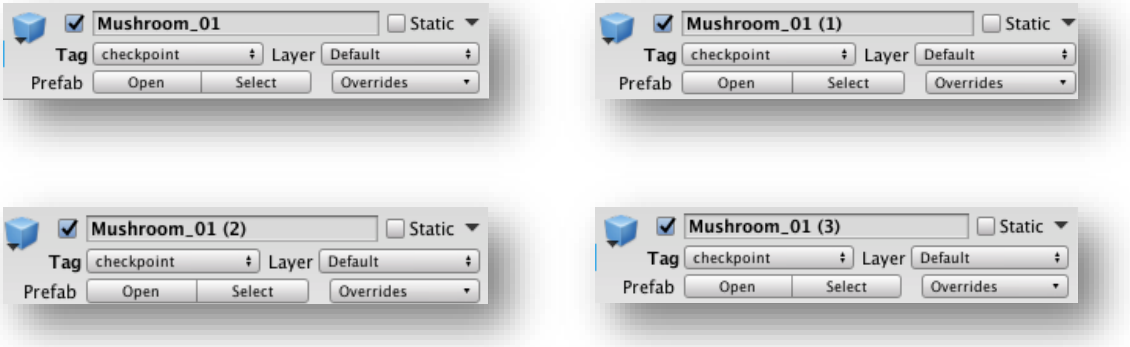
Sensei Stop

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

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

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

[Mushroom_01, Mushroom_01 (1), Mushroom_01 (2), Mushroom_01 (3)]



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

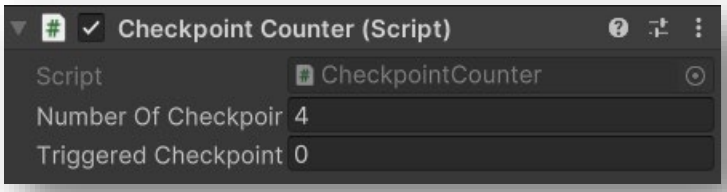
```
public class CheckpointCounter : MonoBehaviour
{
    public int numberOfCheckpoints;
    public int triggeredCheckpoints;
    0 references
    void Start()
    {
        numberOfCheckpoints = GameObject.FindGameObjectsWithTag("checkpoint").Length;
    }
}
```

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

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

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

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



81 Stop your game.

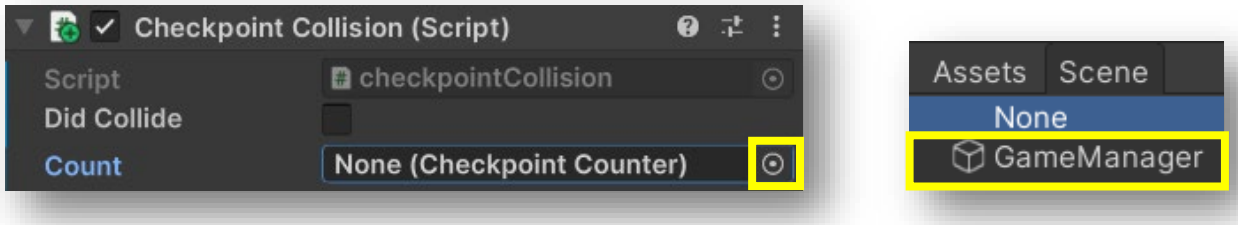
Remember the **Triggered Checkpoints** variable we also created? That variable needs to be updated every time we cross a checkpoint. Since the **Triggered Checkpoints** variable is public, other scripts can use it too. Luckily, we already created that condition in our **checkpointCollision** script.

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

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

    0 references
    void Start()
```

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

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

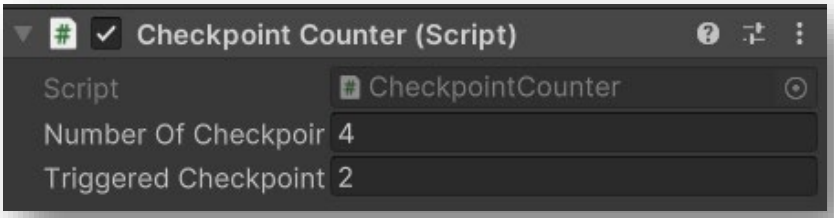


Repeat this for all of your checkpoint objects.

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

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

    0 references
    void Start()
    {
        didCollide = false;
    }

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

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



Sensei Stop

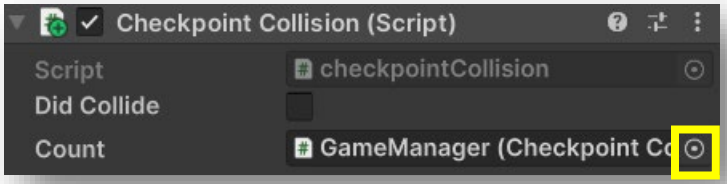
Go through your entire track with your Code Sensei to confirm that both the numberOfCheckpoints and triggeredCheckpoints values match at the end.

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

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

```
public class TriggerFinishLine : MonoBehaviour
{
    public CheckpointCounter checkpointTracker;

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.tag == "Player")
        {
            print("You Win!");
        }
    }
}
```

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



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

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

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

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

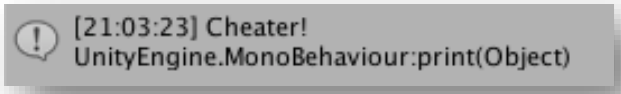
88 We also want to add an else condition that prints "Cheater!" in the console if a player has not gone through all the checkpoints.

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

Sensei Stop

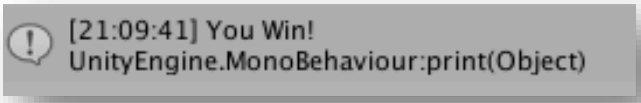
Describe to your Code Sensei how your if condition has changed. Then demonstrate what happens when you cross the finish line without going through all the checkpoints versus when you have gone through all checkpoints.

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



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

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

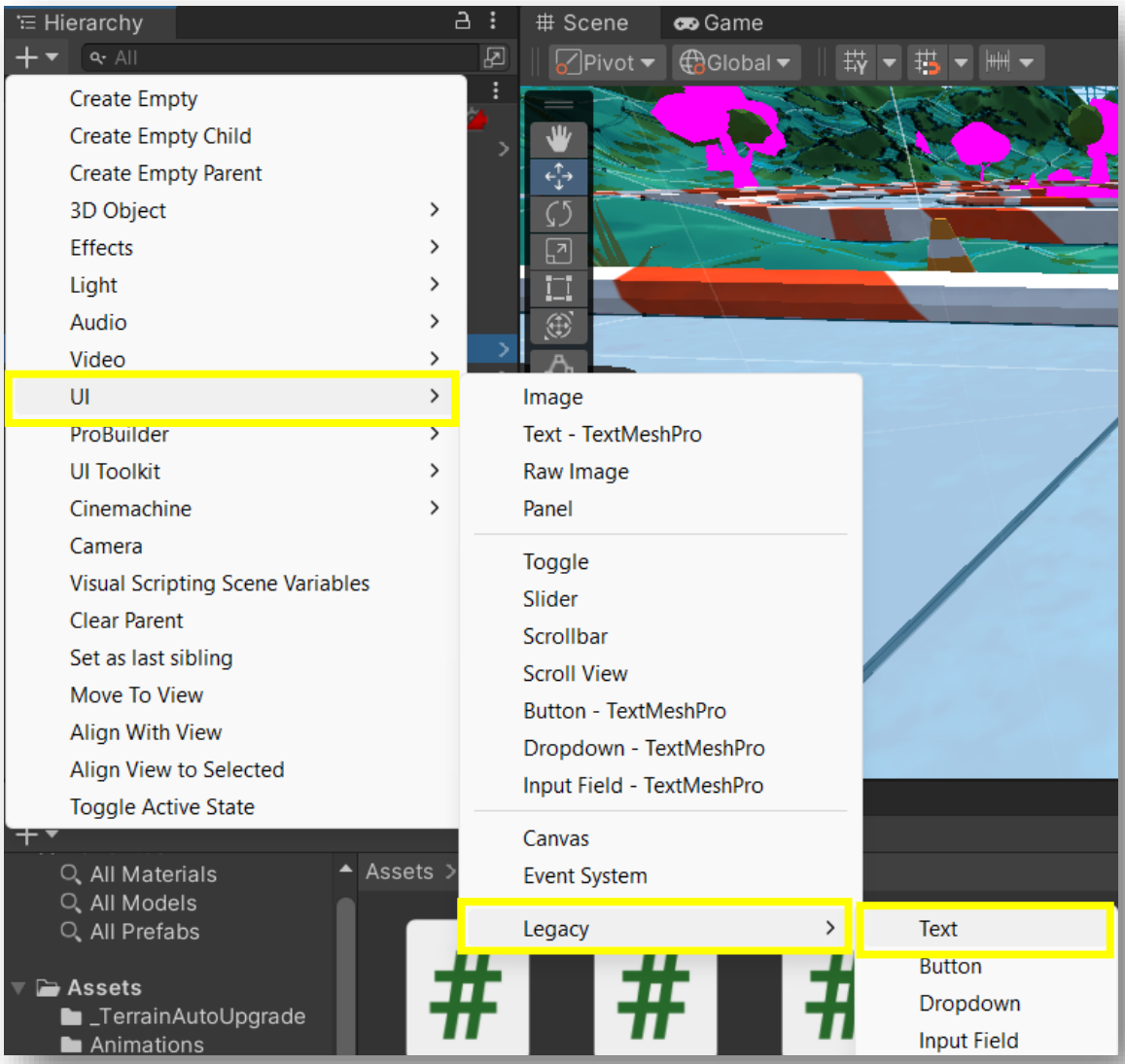


You win! Awesome, we have the lap system working! We'll continue building on this in the next section.

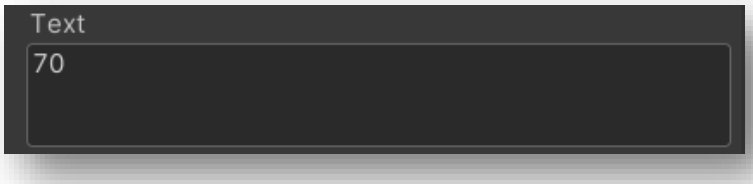
It's About Time

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

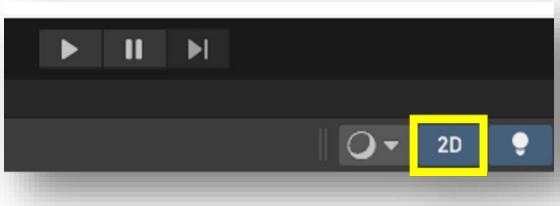
92 In the **Hierarchy**, create a **UI Legacy Text** object.



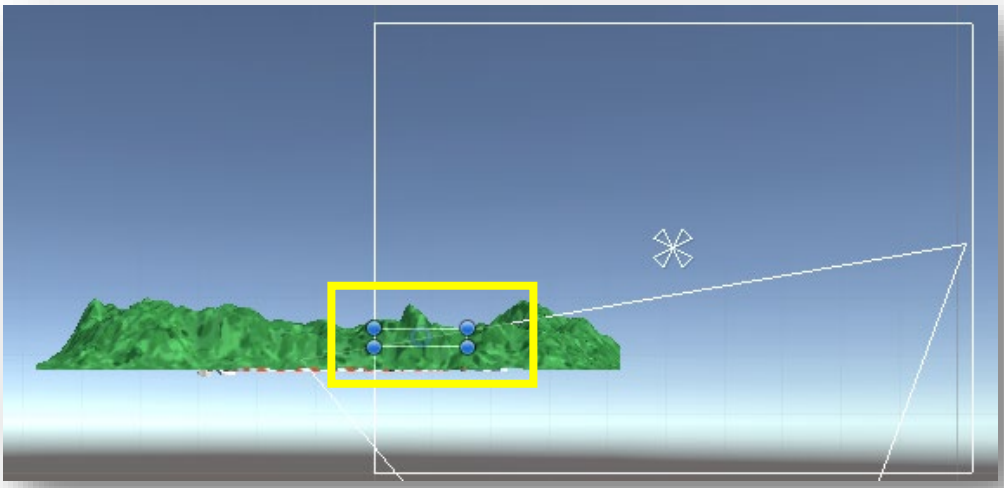
93 Rename this object **GameTimer.**in the **Inspector** change the **Text** to the amount of time you want the player to have to finish the track. In the example image below, we are giving Codey 70 seconds.



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

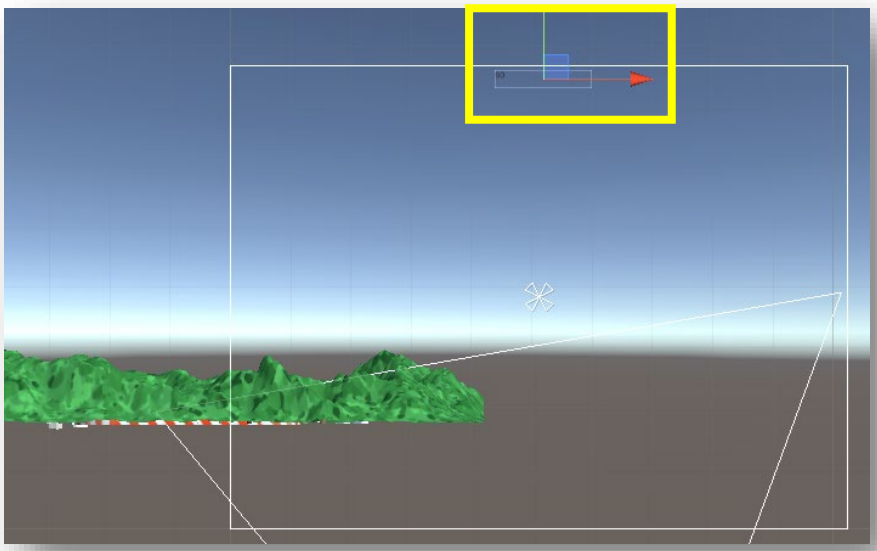


95 Select the **GameTimer** in the Hierarchy. Using your mouse, zoom out until you can see the **Text Box**.

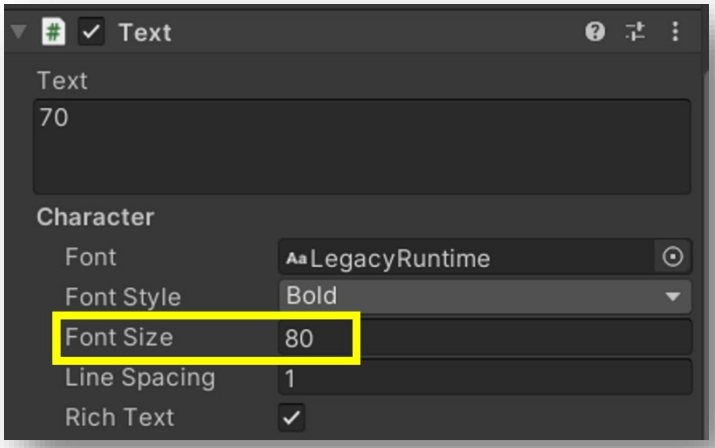


Do you see the other much bigger white box? That is the **Canvas** object you see in the Hierarchy. As long as you position the **GameTimer** object inside this box, you will see it in your Game screen.

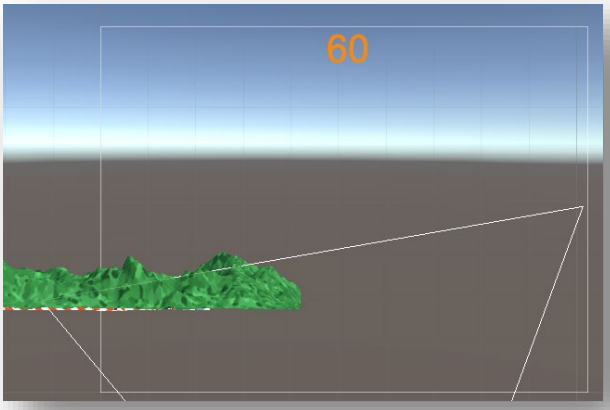
96 Position your timer somewhere near the top of the Canvas.

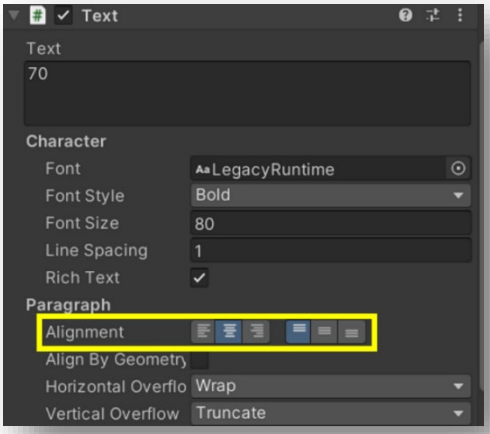


97 The text is a bit small. Adjust it by changing the **Font Size** in the Inspector.

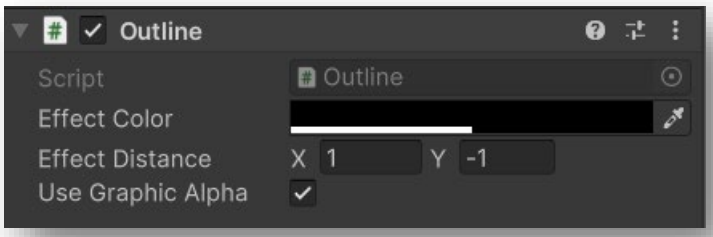


98 In the **Inspector,** change the **Alignment** component to the center button. This will center the text. Use the **Character** and **Paragraph** components to adjust the text how you like!





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



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

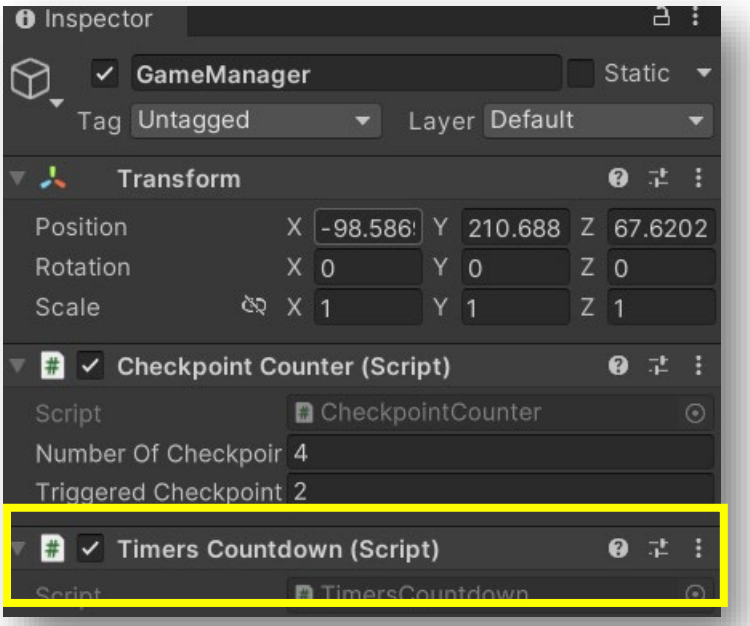
101 We also want to create a **CountdownTimer** that tells our player when the race will begin. This timer will count down from 3.

Sensei Stop

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

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

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



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

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

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

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

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

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

    public float totalLapTime;
    public float totalCountdownTime;

    0 references
    void Update()
    {

    }
}
```

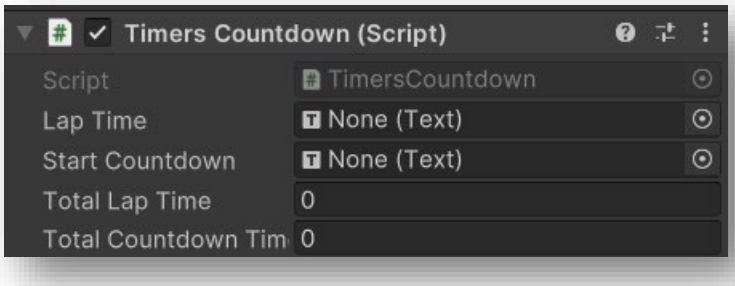
We made all four variables public so we can update them from anywhere in Unity, not just from inside of this script.

Sensei Stop

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

106 Save your script and return to Unity.

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



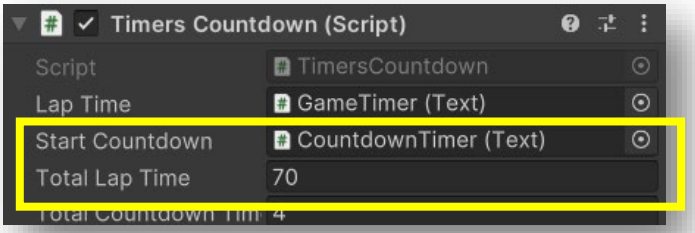
Sensei Stop

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

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

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

    public float totalLapTime;
    public float totalCountdownTime;

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

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



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

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

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

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

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

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

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

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

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

Mathf

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

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

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

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

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

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

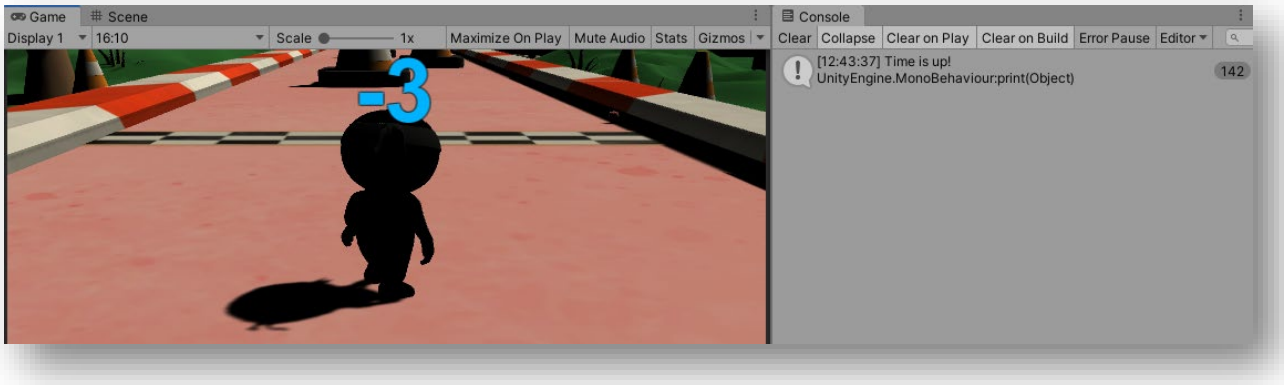
- **If** the **totalCountdownTime** is greater than 0:
    - subtract **Time.deltaTime** from **totalCountdownTime**,
    - update the **startCountdown** text to the new **value**, and

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

Sensei Stop

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

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



Hazardous Conditions

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

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

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

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

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

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

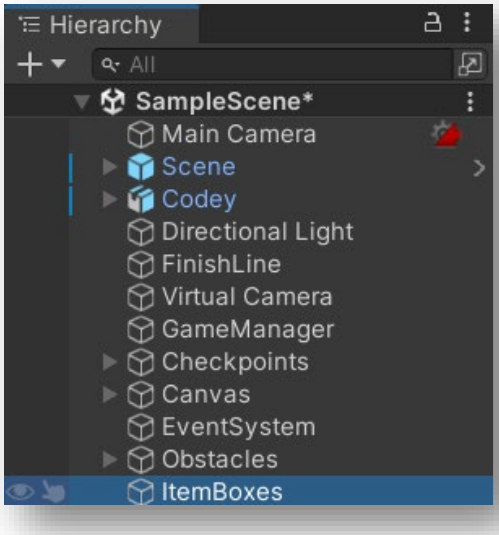
118 Use your Ninja Planning Document and the screenshots below as inspiration for your track.

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

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

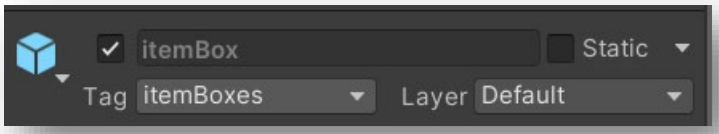
In this section we will create multiple item boxes that will allow Codey to access, then use a power-up. We will use the **Instantiate** function that we learned from Shape Jam!

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



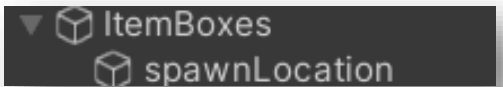
121 In the **Hierarchy,** create a **Cube** and rename it **itemBox**. Drag this into the **Prefabs** folder.

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



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

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



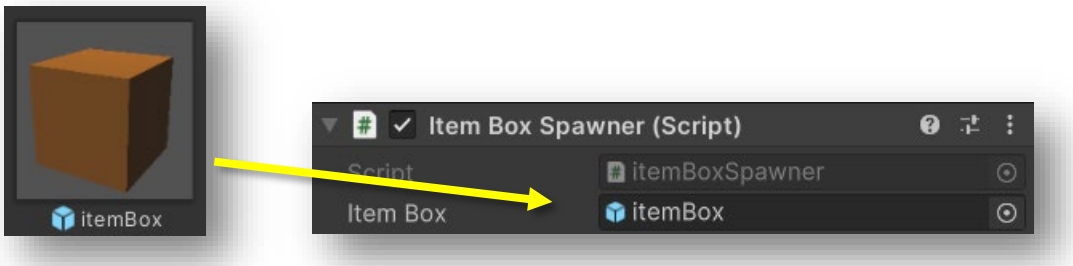
125 This **spawnLocation** object will help us place our item boxes when we clone them in their place. Move the **spawnLocation** object somewhere on your track. It will be hard to see the position since there is no actual object there but in the next few steps, you will be able to see where your boxes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

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

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

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

    0 references
    void Start()
    {

    }
}
```

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



129 Open the **ItemBoxSpawner** script.

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

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

    public int numberOfBoxes;

    0 references
    void Start()
    {

    }
}
```

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

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

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

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

Sensei Stop

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

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

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

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

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

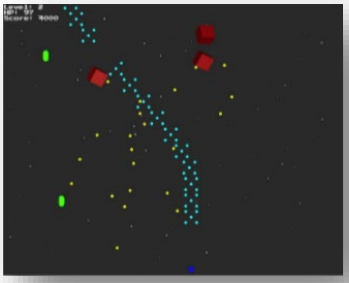
- the object we want to clone,
- the position where we want to place the object, and
- the rotation of the object.

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

Sensei Stop

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

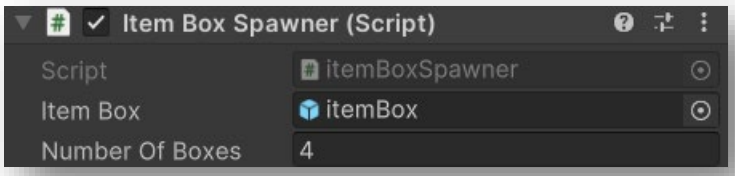
If you get stuck, look back at the Shape Jam activity.



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

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

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



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

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

Why can't we see all of them?

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

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

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

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

137 Open the **itemBoxSpawner** script.

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

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

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

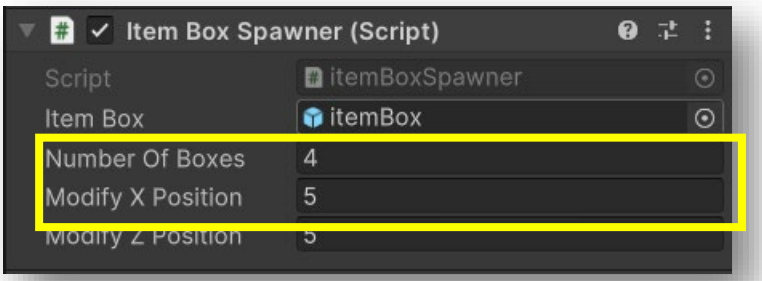
Stop your game. Create two **public int** variables that we will use to modify the position when we instantiate the boxes.

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

    public int numberOfBoxes;

    public int modifyXPosition;
    public int modifyZPosition;

    0 references
    void Start()
    {
```

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

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

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



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

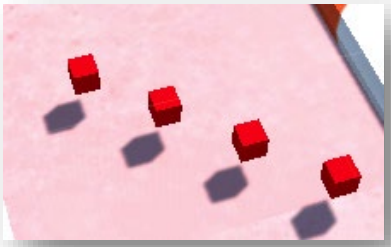
```
GameObject itemBoxClone = Instantiate(
    itemBox,
    new Vector3(
        transform.position.x + modifyXPosition,
        transform.position.y,
        transform.position.z + modifyZPosition
        ),
    transform.rotation);
```

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

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

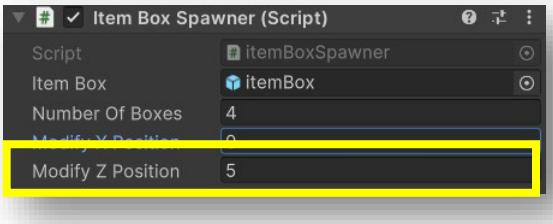
```
void Start()
{
    for (int i = 0; i < numberOfBoxes; i++)
    {
        GameObject itemBoxClone = Instantiate(
            itemBox,
            new Vector3(
                transform.position.x + modifyXPosition * i,
                transform.position.y,
                transform.position.z + modifyZPosition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

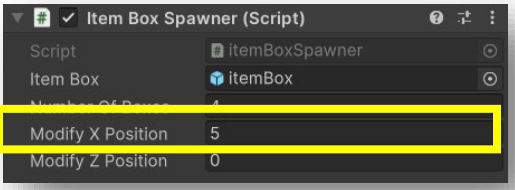
143 Save your script and return to Unity. Playtest your game!



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

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



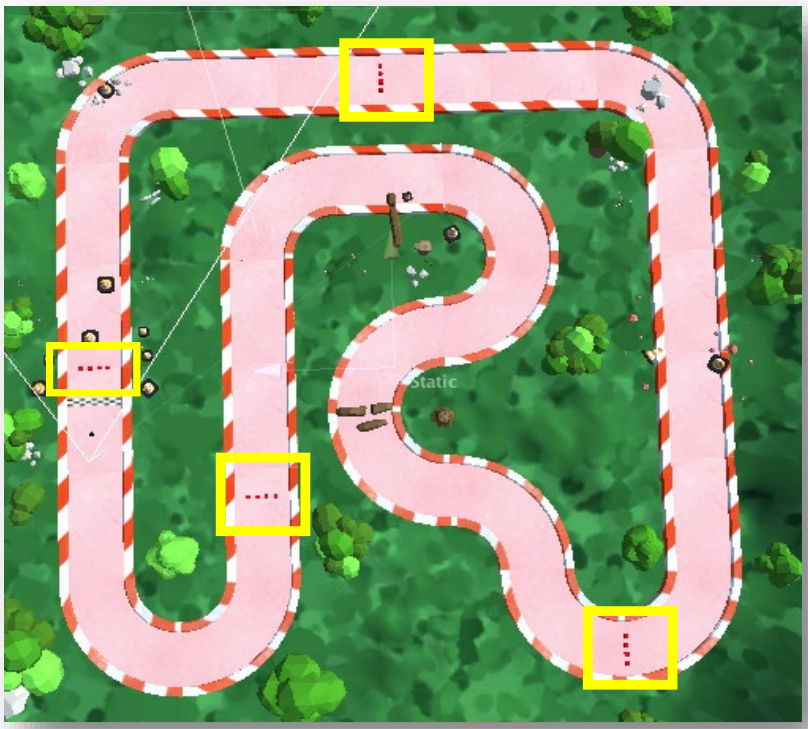


Sensei Stop

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

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

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

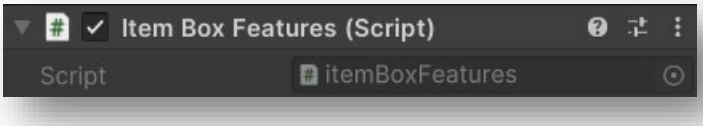


If you collide with these new item boxes they currently don't do anything. In the next steps we will focus on adding additional features to them.

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

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

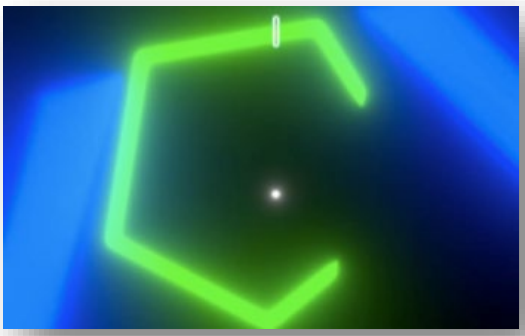
146 In the **prefabs** folder, select the **itemBox** we created. Click **Add Component** in the **Inspector** and add the **itemBoxFeatures** script.



147 Open the script in Visual Studio.

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

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



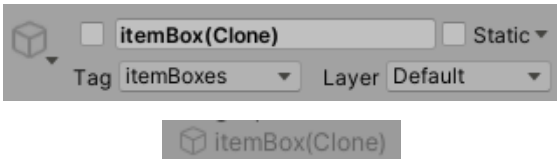
Sensei Stop

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

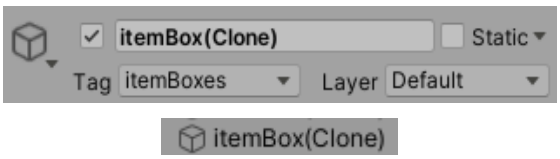
148 We want to make the itemBox disappear when Codey collides with it. Then it should reappear after a few seconds. We can do this using the **Invoke** function!

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

```
gameObject.SetActive(false);
```




```
gameObject.SetActive(true);
```



Sensei Stop

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

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

151 We need to create a function that will make our item box reappear after Codey collects it.
In the **itemBoxFeatures** script create a new **private void function** named **itemBoxRespawn**.

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

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

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

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

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

Sensei Stop

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

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

Feel the Power

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

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

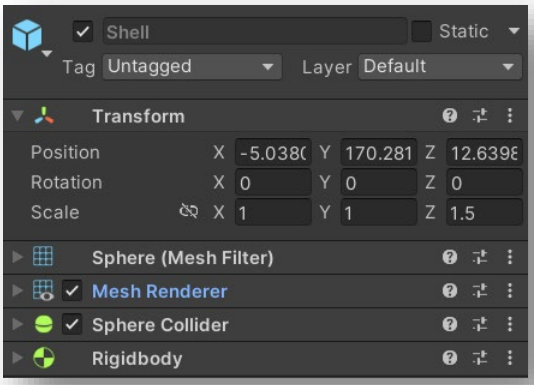
Once you are done shaping the Shell, drag it into the **Prefabs** folder and delete it from the **Hierarchy**.

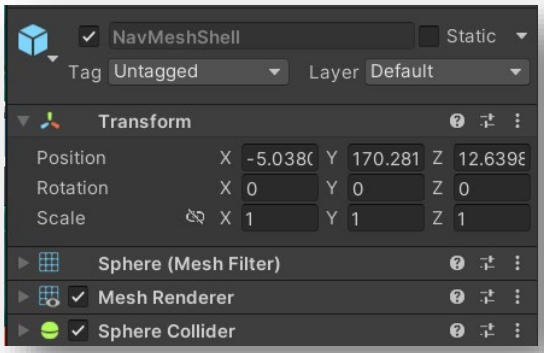
We will be programming this **Shell** object to **Instantiate** from Codey's position and go straight from where Codey is standing. The Shell will then destroy any obstacle in its way.

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

Make this new shell visually different from the first!

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





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

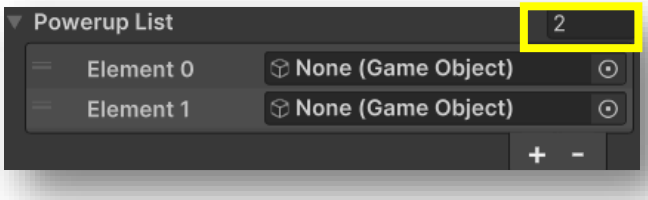
Create a new script in the **Scripts** folder named **SelectRandomPowerup** and attach it to Codey.

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

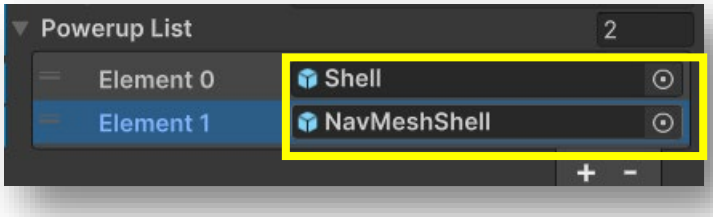
```
public class SelectRandomPowerup : MonoBehaviour
{
    public List<GameObject> powerupList;

    0 references
    void Update()
    {

    }
}
```

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

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



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



162 Open the **SelectRandomPowerup** script.

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

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

    0 references
    void Update()
    {

    }
}
```

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

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

    0 references
    void Update()
    {

    }

    0 references
    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.tag == "itemBoxes")
        {

        }
    }
}
```

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

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

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

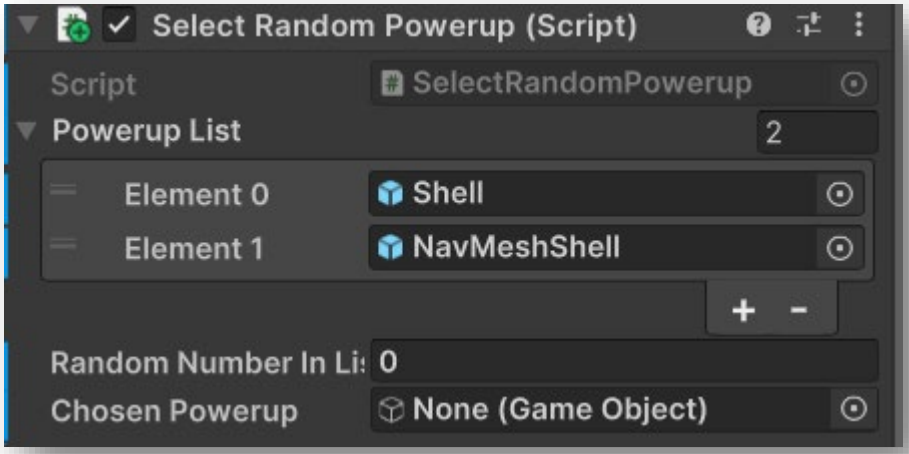
Create a new **public Game Object** named **chosenPowerup**.

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

    0 references
    void Update()
    {
```

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

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

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



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

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

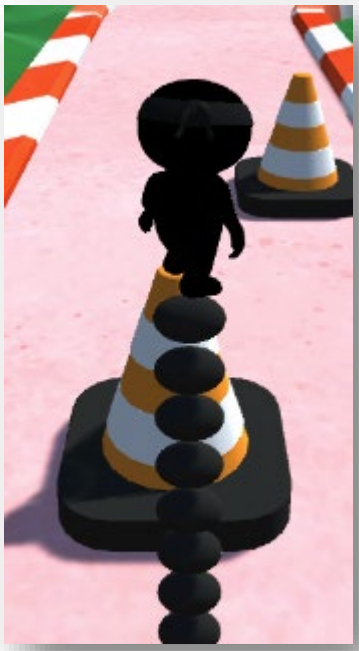
Collect more than one box to make sure both your powerups are being randomly selected!

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

Sensei Stop

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

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



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

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

Sensei Stop

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

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

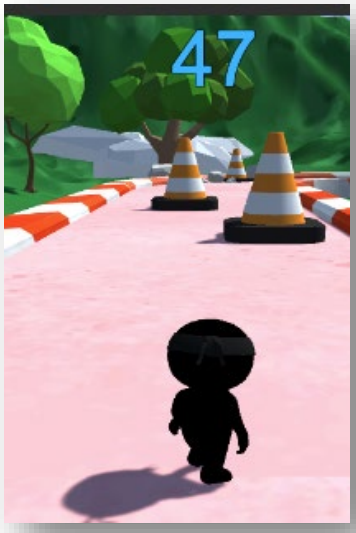
172 The last thing to do is make our Game Objects destroy the obstacles in our track! In the **Scripts** folder, create a new script and name is **ShellMovement**. We will use this script to create the basic movement for the **Shell**. Attach this script to the Shell game object in the prefabs folder.

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

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

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

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



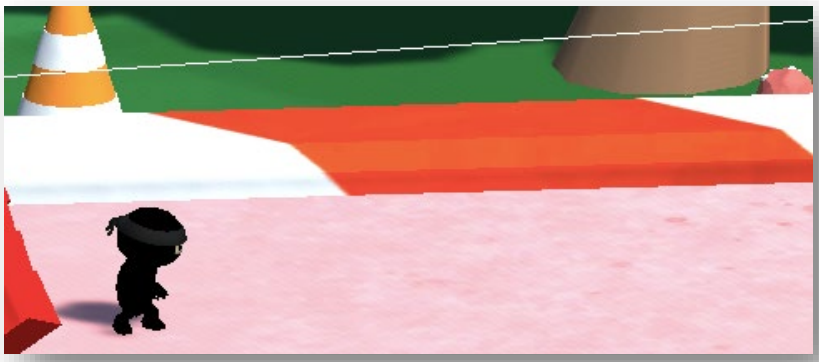
Sensei Stop

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

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

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

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



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

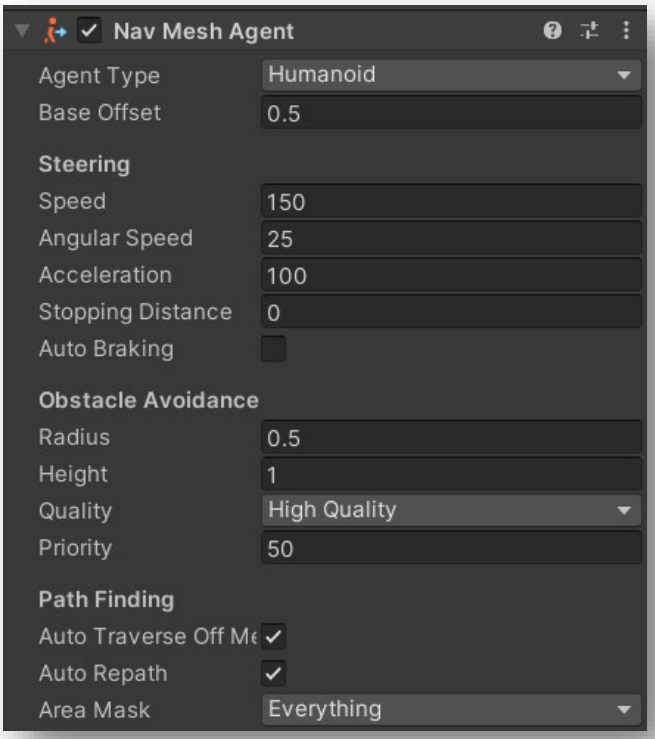
To make your game more unique, you can replace the Shell game object with any other game object you want!

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

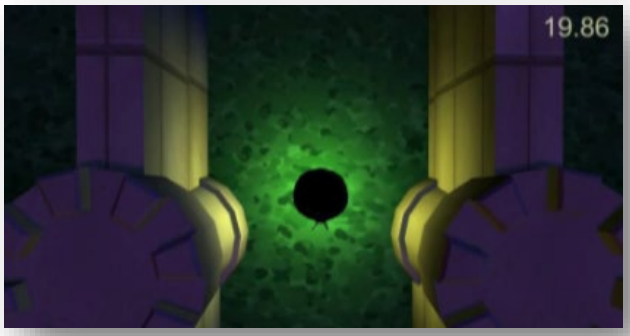
Create a new script in the **Scripts** folder and name it **NavMeshMovement**.

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

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



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



Sensei Stop

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

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

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

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

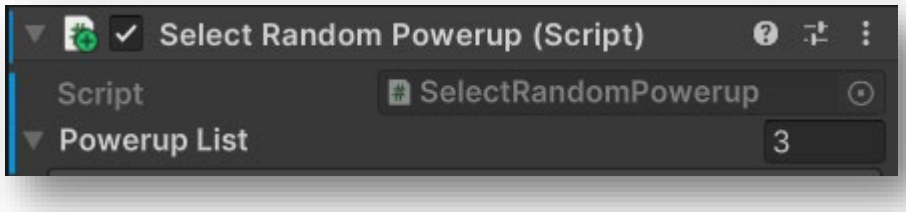
You can add other power-ups that give Codey an advantage in your game, or you can program something that makes it harder for Codey to get to the finish line.

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

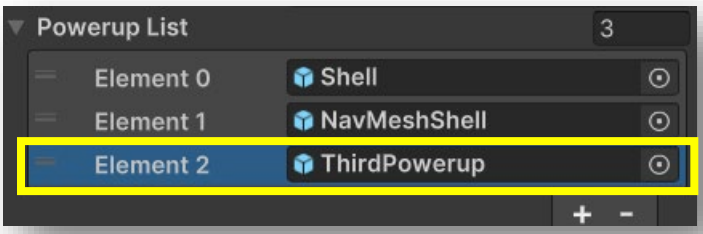
If you want to make the game harder, you could slow Codey down or make Codey spin randomly.

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

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



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



Have fun and get creative!

Get Off My Lawn!

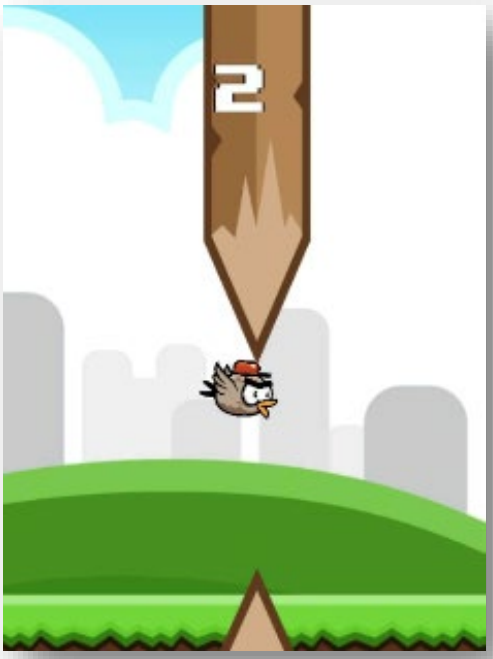
184 There is always the possibility that Codey falls off the track and falls onto the Terrain.

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

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

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

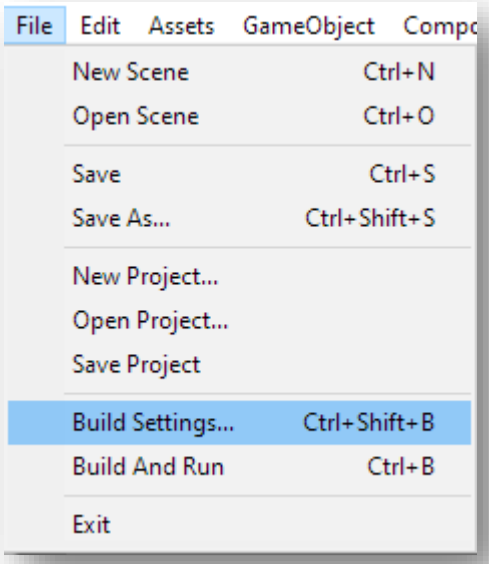
Create the logic to restart the level when Codey touches the Terrain. Use what you have learned so far in Codey Raceway to distinguish the Terrain from all the other game objects in your collision code.



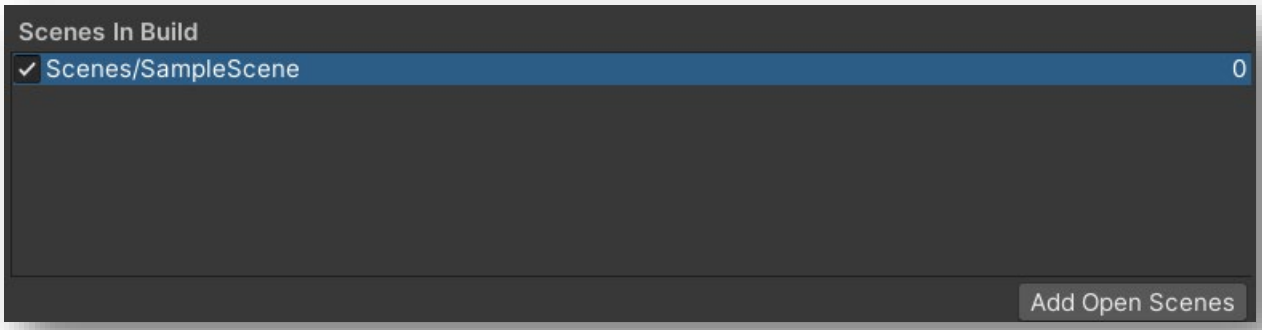
Sensei Stop

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

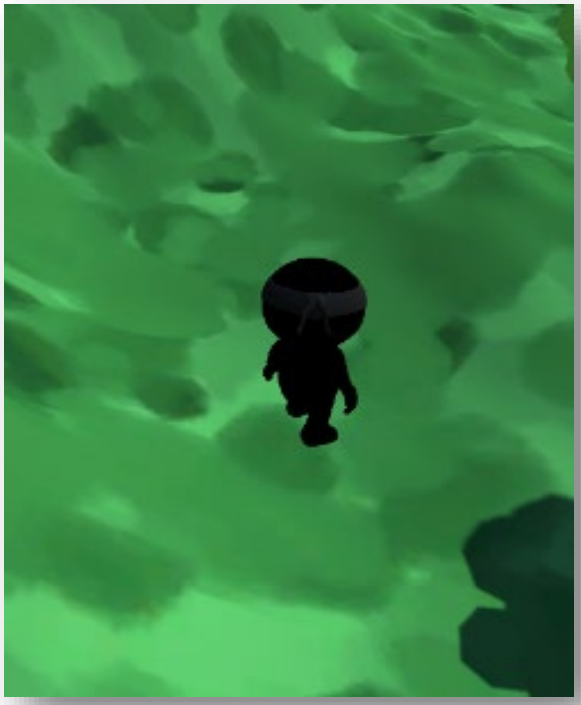
187 Save your script and return to Unity. In the tabs at the very top of the screen, click on **File** then **Build Settings**.



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



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



Victory Lap

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

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

Sensei Stop

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

Project Submission and Reflection

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

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

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

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

Ninja Planning Document

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