RED BELT SENSEI GUIDE

Red Belt Sensei Guide

The Red Belt is different from the previous Unity Belts because it emphasizes creativity and individuality over code replication.

Each game has a Ninja Planning Document that the Ninja must complete and reference throughout their game design process. This document will help them plan out their ideas so they can truly make each of the projects their own.

Throughout the curriculum, the Ninja will encounter Sensei Stops. These are designed to be check-ins between the Ninja and a Code Sensei. Each Sensei Stop will have different requirements. One might ask a Ninja to explain a concept in their own words, while another might require them to write code with little assistance.

At the end of each project, the Ninja is required to complete a Requirements Checklist. This will ensure that the Ninja has a complete and unique game before submitting it in the GDP for grading.

With a focus on creativity, knowledge, and independence, the Red Belt is designed to be the final steppingstone before Black Belt.

Gravity Trails

Section Notes

Project Setup

This section relies heavily on the Purple Belt Scavenger Hunt activity. Many of the instructions tell the Ninja to reference the curriculum from the previous activity to help build the scene. This is intentional to help build the Ninja's abilities and confidence in borrowing existing code and logic from games they previously created.

This section has two Sensei Stops.

A Walk in the Park

Make sure that the Ninja creates an empty object to help them organize their scene. While it is possible use other objects, Quad objects are useful because they can be seen in the scene view but are hidden in the game view.

The Polygon Collider 2D can be tricky to set up because of how Unity handles adding, removing, and editing vertices. Since it is being used to only contain the camera, it does not need to be perfectly aligned with the edges of the background image.

Lighting in a 2D project can be tricky because Unity performs all calculations in 3D space. This means that using a directional light won't work as expected and the skybox environment light should be used. Help the Ninja find good values for the Environment Lighting Intensity Multiplier in the Lighting window.

This section has one Sensei Stop.

What Goes Up, Must Stay Up

Make sure that the Ninja creates an empty object to help them organize their scene. While it is possible use other objects, Quad objects are useful because they can be seen in the

This section has three Sensei Stops.

You Shall Not Pass

This section walks the Ninja through adding enemies to the scene. The Ninja can customize their enemy sprite. The curriculum will work with any enemy as long as there are no existing scripts or components that might conflict with the instructions provided.

This section has two Sensei Stops

Seeing Stars

Encourage the Ninja to use their Ninja Planning Document to help them build their scene and choose their collectable. If the Avatar cannot interact with the collectable objects, make sure that collectable has a 2D Collider and the correct tag.

Have the Ninja explain why they placed their collectables where they did.

This section has one Sensei Stop.

Throwdown

This section focuses on programming the logic of the projectile. Make sure the Ninja carefully programs the Start function in the Projectile script. Because the script is attached to a prefab, Unity will not let you attach other game objects to public variables. The Ninja must search for the Avatar game object and get its Throwable component in the Start function.

This section has two Sensei Stops

Show Me the Shurikens

This section focuses on creating a simple UI and updating it based on what happens in the game. As you check the Ninja's work, stress the importance of clearly communicating information to the player.

This section has one Sensei Stop.

Take it up a Level

This section helps the Ninja create a portal from the first scene to a second scene. This logic will be used later on in Sulky Slimes. Encourage the Ninja to think of other goals for the player besides defeating all the enemies.

Encourage the Ninja to be creative when designing the second level. The curriculum provides several ideas without solutions that are intended to get the Ninja thinking about new ways to apply what they have learned.

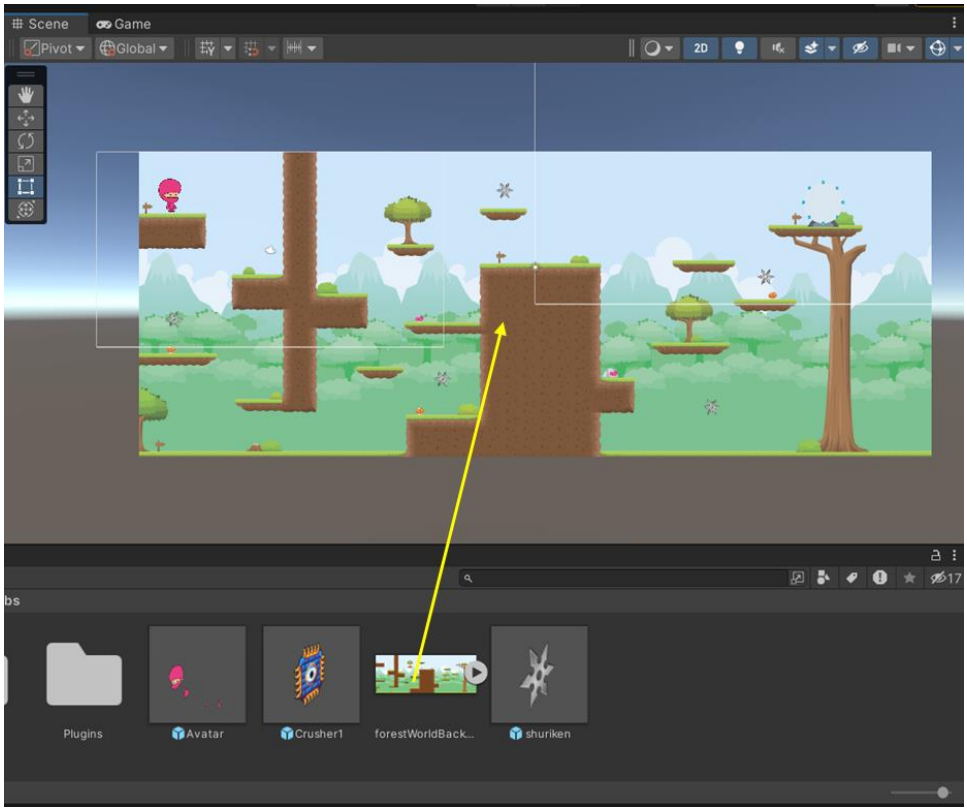
There is one Sensei Stop in this section.

Sensei Stops

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

Ask the Ninja to think back to the Scavenger Hunt activity in Purple Belt. Ask them to explain the steps they took to create an object and add an image to it.

The Ninja should drag the image from the asset folder to the object in the scene. Make sure the Ninja renamed the object to Background.

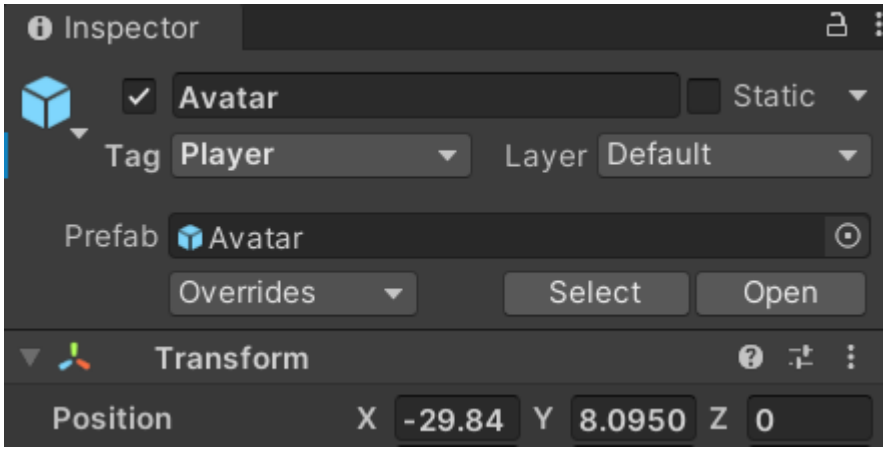


10 *Describe what you must do for the Avatar to appear correctly. Make the changes and show your Sensei that all Avatar parts are showing.*

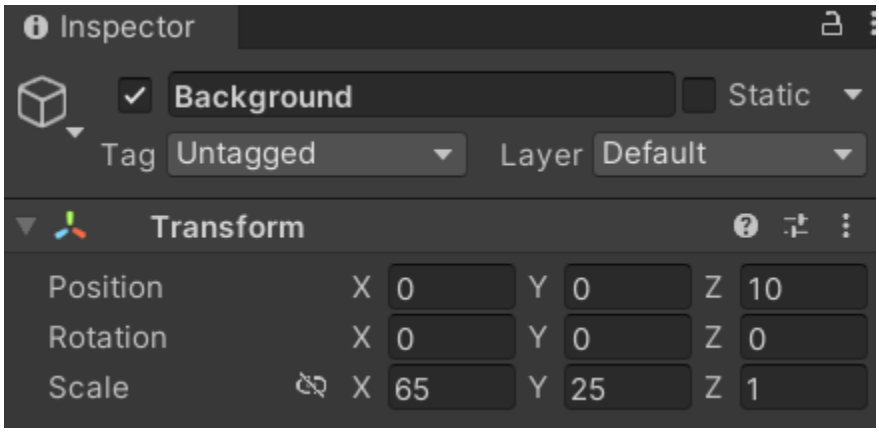
Even though this activity is in 2D, Unity always uses 3D dimensions. If X is horizontal and Y is vertical, have the Ninja explain how the Z coordinate is used in 2D Unity games.

Ask the Ninja to explain the steps they took to fix their Avatar.

One solution is bringing the Avatar object towards the camera by adjusting the transform's Z position to be a larger negative number.



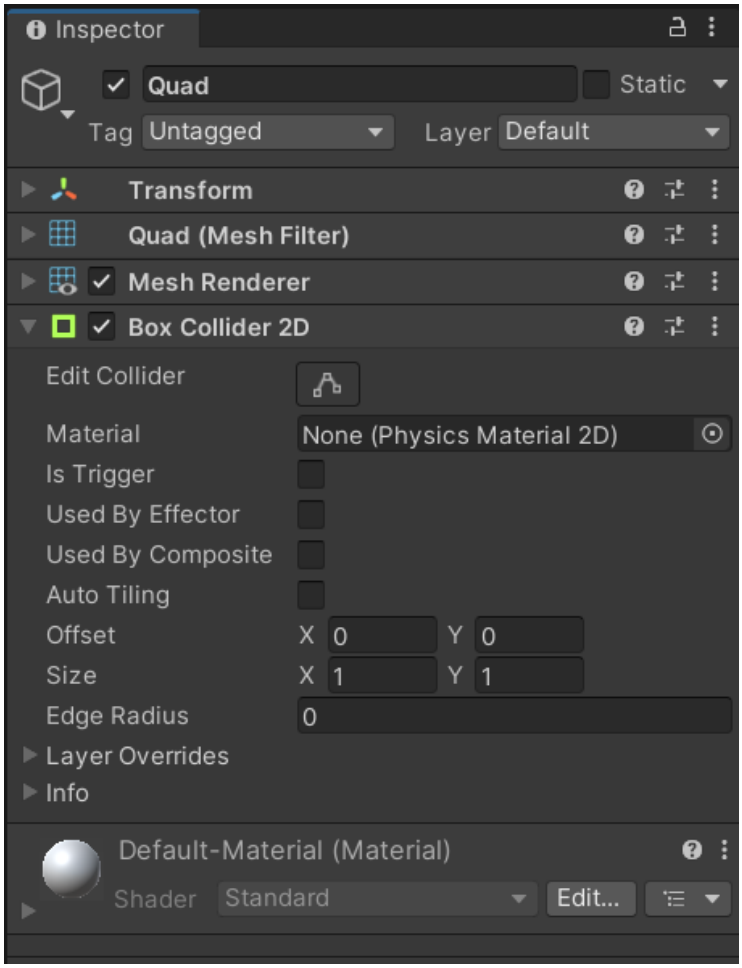
Another solution is to increase the value of the Background object's Z position. This will move it farther away from the camera.



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

Ask the Ninja why they think the Avatar passes through the quad objects. Explain that even though 2D games in Unity use the Z coordinate, Unity uses special 2D components to calculate physics and collisions of 2D objects.

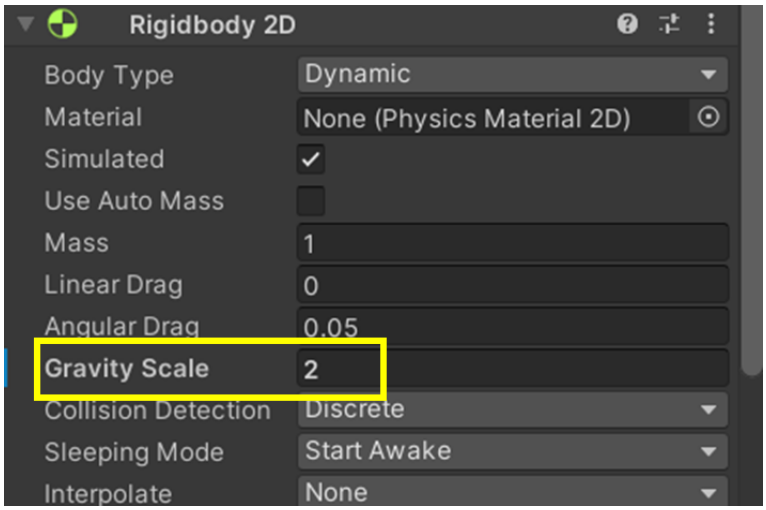
First, the Mesh Collider must be removed since it does not work with 2D objects like the Avatar. Then, a Box Collider 2D must be added to the quad.



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

Ask the Ninja what components are on the Avatar game object. Have them explain which component controls how the Avatar interacts with the Unity physics engine.

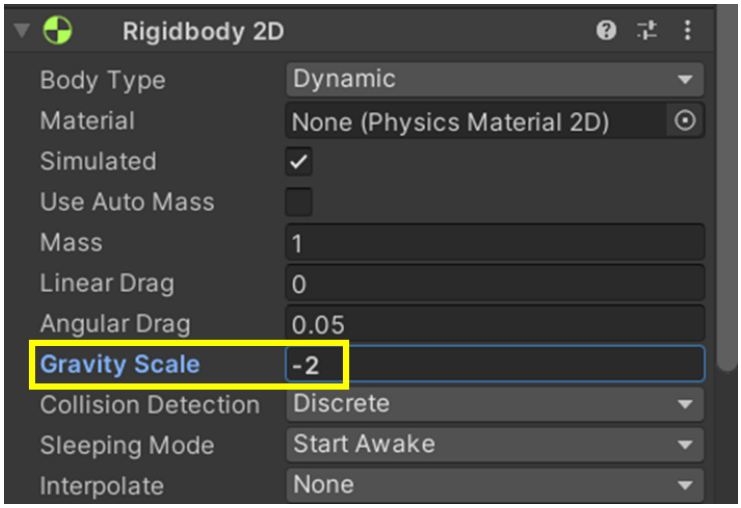
The Ninja should be able to explain that the Rigidbody 2D component's Gravity Scale controls how gravity affects the Avatar.



With your Ninja, go through the following steps to build the connection on what the Gravity Scale component actually does.

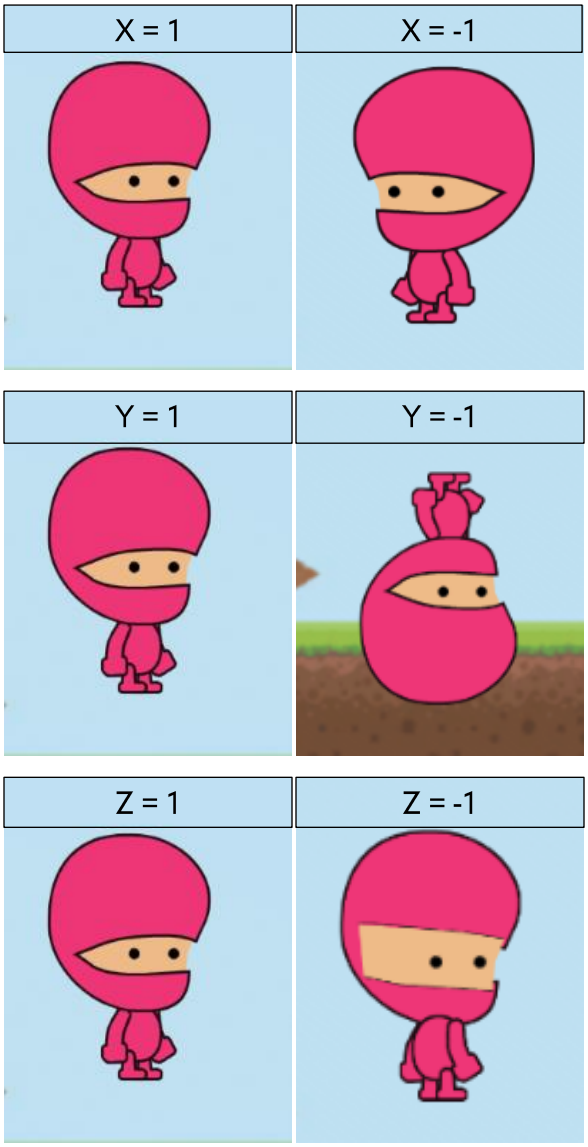
First, playtest the game with a Gravity Scale of 2. The Avatar will fall because gravity is positive.

In the Inspector, change the Gravity Scale to -2. The Avatar will go upwards because gravity is negative.



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

Ask the Ninja to walk you through what happens when each Scale has been changed to -1.



The Ninja correctly state that Scale Y needs to be modified in our code.

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

Ask the Ninja to explain in their own words what each line of code does. Explain the difference between creating a variable using the values of the localScale and actually changing the transform's localScale.

The Ninja needs to add a line of code that sets the transform's localScale property to the newDirection.

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

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

Ensure that the Ninja adds a minimum of 4 enemies to the scene. Encourage the Ninja to place their enemies where they want instead of copying the locations from the provided images.

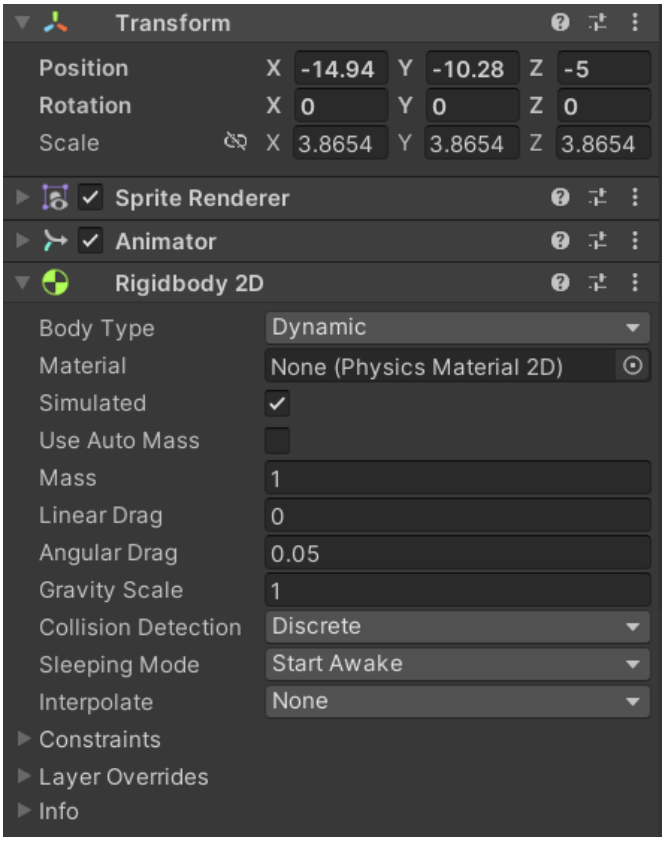
The Ninjas must create a new script for the enemies. In this example we are calling the script EnemyMovement.

The Ninjas should use Robomania steps 33a through 44b to help them write the script. +

The Ninja should create six public variables. Two forces, yForce and xForce, control the jump strength of the enemy. xDirection determines if the enemy moves left or right. The enemyRigidBody will be used to apply forces to the enemy. The maximum and minimum x positions control the area the enemy will move back and forth in.

```
public float yForce;
public float xForce;
public float xDirection;
private Rigidbody2D enemyRigidBody;
public int maximumXPosition;
public int minimumXPosition;
```

The enemies must have a Rigidbody component so we can apply forces in the script.



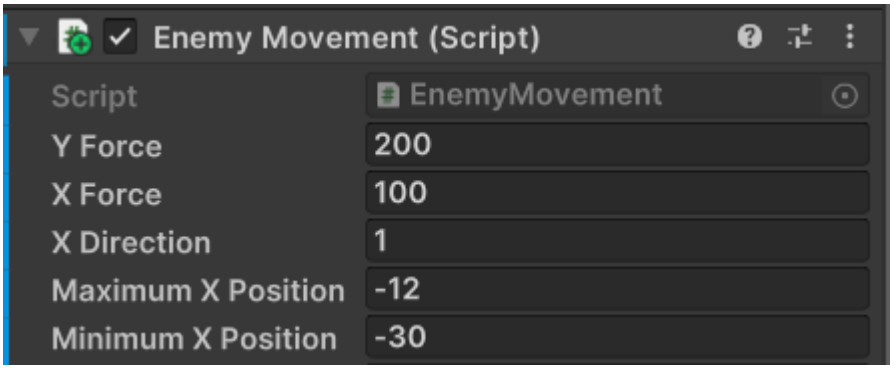
If the Ninja decides to make the enemyRigidbody a private variable, they will have to initialize the variable in a Start or Awake function.

```
void Start()
{
    enemyRigidBody = GetComponent<Rigidbody2D>();
}
void Awake()
{
    enemyRigidBody = GetComponent<Rigidbody2D>();
}
```

The logic to check the boundaries are the same, except for two changes they can make which will help them with the minimum and maximum boundaries.

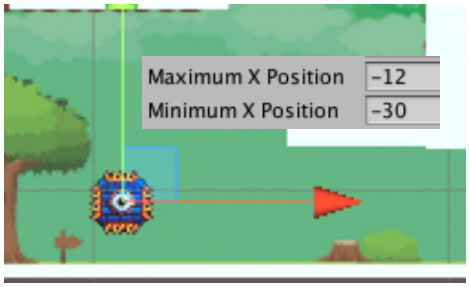
They should use the 2 variables to accurately tell each enemy what their boundaries are.

```
private void FixedUpdate()
{
    if (transform.position.x <= minimumXPosition)
    {
        xDirection = 1;
        enemyRigidBody.AddForce(Vector2.right * xForce);
    }
    if (transform.position.x >= maximumXPosition)
    {
        xDirection = -1;
        enemyRigidBody.AddForce(Vector2.left * xForce);
    }
}
```



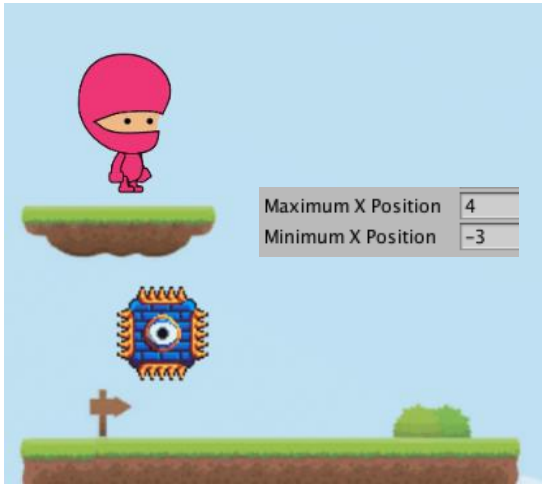
Each enemy now has two new variables that will tell our enemies the left and right bounds of their movement.

The Ninja needs to use numbers that work with the position of each individual enemy.



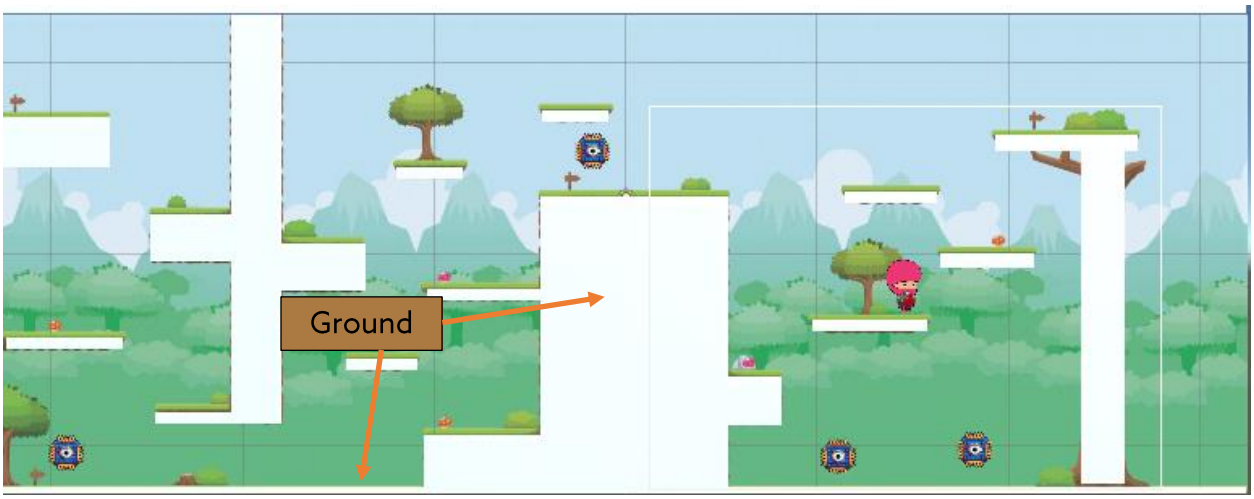
Encourage the Ninja to experiment with numbers to make it seem that the crusher bounces off the platforms and walls.

Below are other examples of other minimum/maximum positions.



The crushers will only bounce upwards if they collide with an object below them with a tag. We want to create a new tag and name it **ground**.

Give this tag to the Quad objects that the crushers will collide with.



Just like in Robomania, when the crushers collide with the ground, we want to apply an upward force.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "ground")
    {
        Vector2 jumpForce = new Vector2(xForce * xDirection, yForce);
        enemyRigidBody.AddForce(jumpForce);
    }

}
```

There is a chance that enemies collide with one another. We can add to the **OnCollisionEnter2D** function to solve this potential bug.

A possible solution to resolve this issue is to change the direction of the enemies and apply a new force in the opposite direction.

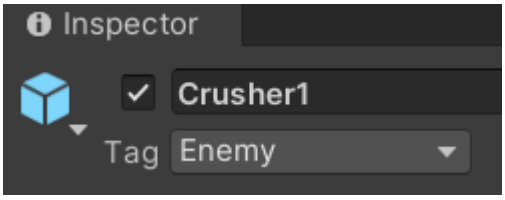
```
if (collision.gameObject.tag == "Enemy")
{
    xDirection *= -1;
    Vector2 jumpForce = new Vector2(xForce * xDirection, yForce * 2);
    enemyRigidBody.AddForce(jumpForce);
}
```

The Ninja should feel free to modify this code to customize the movement of their enemies.

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

Ask the Ninja to explain how they tagged their enemy game objects. Ask if they used any previous games to help them.

Each crusher needs to be given the “Enemy” tag. It is important to note that tags are case sensitive.



Add **Using UnityEngine.SceneManagement** to the top of the script.

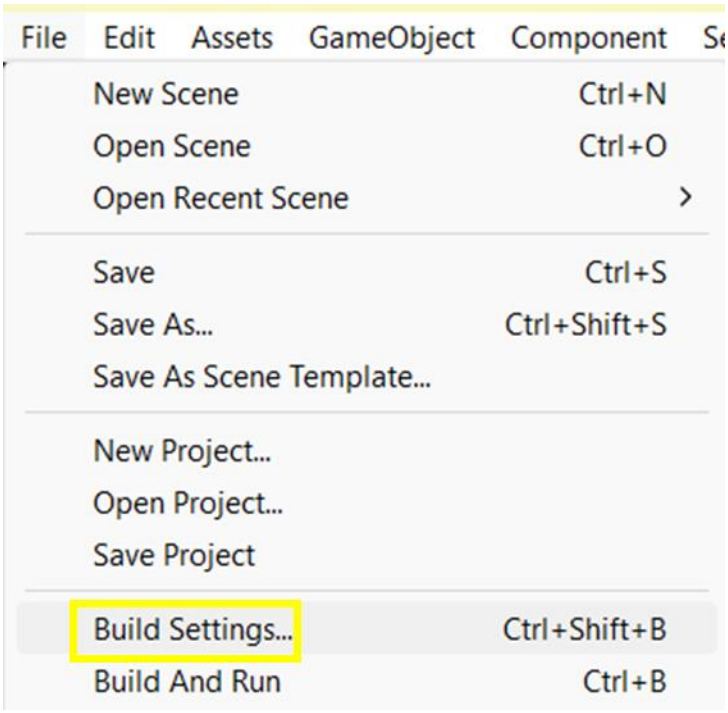
```
using UnityEngine.SceneManagement;
```

Create a **OnCollisionEnter2D** function that checks when the Avatar collides with the crushers. When our Avatar collides with any object with the tag enemy, we want to load scene 0.

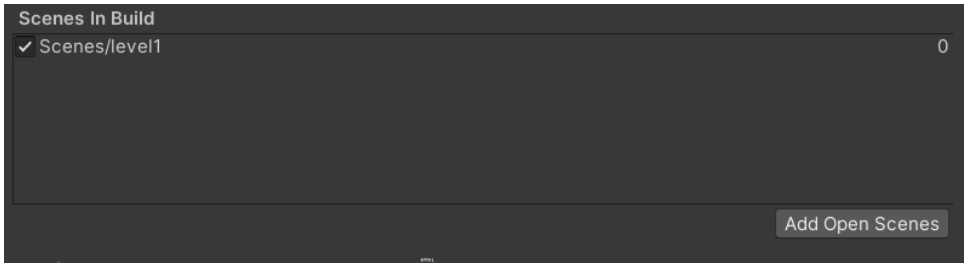
```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.tag == "Enemy")
    {
        SceneManager.LoadScene(0);
    }
}
```

Save your script and return to Unity.

In the tabs above find **File**, click on **Build Settings**.



Click on **Add Open Scenes** in order to add the current scene you are on. It will be assigned an index of 0.



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

The Ninja needs to modify the Throwable script. Help the Ninja create a public int throwableCounter variable and an OnCollisionEnter2D function.

When the Avatar does collide with the object we want to destroy the collision.gameObject and add one to throwableCounter.

```
public int throwableCounter;
```

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Collectible")
    {
        Destroy(collision.gameObject);
        throwableCounter++;
    }
}
```

Modify the if condition to check if we have collected at least one object. Subtract one from throwableCounter inside the conditional statement.

```
void Update()
{
    if (Input.GetButtonDown("Fire1") && throwableCounter > 0)
    {
        throwableCounter--;
        offset = transform.localScale.x * new Vector3(1, 0, 0);
        Vector3 throwablePosition = transform.position + offset;
        Instantiate(offenseObject, throwablePosition, transform.rotation);
    }
}
```

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

Ask the Ninja what special Unity function runs a function after a delay.

The Ninja has to check when a throwable and an enemy collide. They should add their code in the OnCollisionEnter2D function in the EnemyMovement script.

Create and assign the object they are throwing a unique tag. In our example we gave the shuriken a tag of ThrowingObject. When the enemy collides with a ThrowingObject, then the enemy and the throwable need to be destroyed.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Ground")
    {
        Vector2 jumpForce = new Vector2(xForce * xDirection, yForce);
        enemyRigidBody.AddForce(jumpForce);
    }
    if (collision.gameObject.tag == "Enemy")
    {
        xDirection *= -1;
        Vector2 jumpForce = new Vector2(xForce * xDirection, yForce);
        enemyRigidBody.AddForce(jumpForce);
    }

    if(collision.gameObject.tag == "ThrowingObject")
    {
        Destroy(gameObject);
        Destroy(collision.gameObject);
    }
}
```

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

To destroy the throwable when it does not collided with an enemy, they must create a **private void** function that simply destroys the gameObject.

```
private void DestroyOffense()
{
    Destroy(gameObject);
}
```

The Ninja needs to **Invoke** this new function inside the Start() function after a few seconds. The Ninja can change the amount of time by changing the value in the second parameter.

```
void Start()
{
    avatar = GameObject.FindGameObjectWithTag("Player").GetComponent<AnimateMovement>();
    if (avatar.facingLeft)
    {
        xDirection = -8;
    }
    else if (avatar.facingRight)
    {
        xDirection = 8;
    }
    Invoke("DestroyOffense", 1f);
}
```

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

At the very beginning of the game the Ninja should initialize the **offenseText.text** to a string of the number zero.

```
void Start()
{
    offenseText.text = "0";
}
```

This makes sure that the player knows that at the start of the game they have no objects they can throw.

When the Avatar collides with the object, the text must also be updated to match the value of **offenseCount**.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.tag == "Collectible")
    {
        Destroy(collision.gameObject);
        offenseCount++;
        offenseText.text = offenseCount.ToString();
    }
}
```

The final step is to also update the text when the Avatar throws their object in the Update() function.

```
if (Input.GetButtonDown("Fire1") && offenseCount > 0)
{
    offenseCount--;
    offenseText.text = offenseCount.ToString();
```

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

The Ninja must demonstrate that the enemyCount variable decreases by 1 when an enemy is destroyed.

The logic is to defeat enemies is already in the EnemyMovement script. The Ninjas need to create a variable that connects the EnemyMovement script to the Teleport script. We named the variable **counter**.

```
public Teleport counter;
```

Then in the **OnCollisionEnter2D** function, there is logic that destroys the enemy and the object being thrown. Subtract one from the **enemyCount** variable located in the Teleport script.

```
if(collision.gameObject.tag == "ThrowingObject")
{
    Destroy(gameObject);
    Destroy(collision.gameObject);
    counter.enemyCount--;
}
```

Codey Raceway
Section Notes
Project Setup
Have the Ninja start a new Unity project and give it a name so they can identify the project later. Renaming the Sample Scene is important because it prevents imported assets from overwriting their work. Make sure the Ninja imports the correct Codey model.

The Unity Asset Store requires an account to download assets. You can provide a Center account or have Ninjas create their own.

This section has one Sensei Stop.

On Track
Even if the Ninja chooses to find a track from the Unity Asset Store, this section is good practice on how to import and modify pre-made assets.

Navigating a 3D scene can be difficult. Help the Ninjas understand how to use the different modifiers like the right mouse button or the compass in the top right.

This section has one Sensei Stop.

Shopping Spree
This section walks the Ninja through searching for and downloading free track assets from the Unity Asset Store. Each asset might have different object structures, components, and scripts. If a Ninja is having difficulties implementing a pre-made track, encourage them to use the models provided by Code Ninjas to create their own unique track.

This section has no Sensei Stops.

Moving Mountains
There is no wrong way for the Ninja to use the terrain tool. They can make a realistic environment or design something fantastical.

This section has no Sensei Stops.

Exterior Decorating
Encourage the Ninja to be creative when building their scene. Have them reference their Ninja Planning Document throughout to help them implement their vision.

This section has no Sensei Stops.

The End of the Road
The finish line material is included with the Models_CodeyRaceway Unity Package. Ask the Ninja why they placed the

This section has no Sensei Stops.

Eyes on the Road

While they are a good starting point, the Cinemachine values provided in the curriculum might need to be changed based on the scales and positions of the objects in their scene. The camera should be positioned so the player can see Codey and the track.

This section has no Sensei Stops.

Lap it Up

This section walks the Ninja through setting up a finish line and checkpoints. The Ninja can choose any object that they want—what matters is adding a box collider that stretches the width of the track. The curriculum uses four checkpoints, but the Ninja can place as many as they would like because the code is based on finding tagged objects.

The Ninja needs to understand the difference between a trigger collider and a non-trigger collider. This concept has been used before, and it will be reintroduced in Chef Codey.

This section has three Sensei Stops.

It's About Time

If the Ninja needs help setting up their UI, they can reference several games in the Brown Belt to get ideas and assistance. Encourage the Ninja to be creative when they design their text objects.

Make sure Ninjas playtest their tracks to find a good time limit for the player.

This section has four Sensei Stops.

Hazardous Conditions

Encourage the Ninja to use and modify their Ninja Planning Document as they add obstacles and item boxes to their track. This section references several games in the Purple and Brown Belts. Help the Ninja access either their completed games or the curriculum documents.

Most of the coding in this section is done in Sensei Stops. It is very important to check the Ninja's work to make sure they have a working game after each Sensei Stop.

This section has six Sensei Stops.

Feel the Power

Encourage the Ninja to use and modify their Ninja Planning Document as they add obstacles and item boxes to their track. This section references several games in the Purple and Brown Belts.

Most of the coding in this section is done in Sensei Stops. It is very important to check the Ninja's work to make sure they have a working game after each Sensei Stop.

If the Ninja used a track that was not provided by Code Ninjas, they might need help adding a NavMesh and making certain objects walkable.

This section has four Sensei Stops.

Get Off My Lawn!

While this section references Meany Bird, there are several other games that use Scene Management. Help the Ninja determine what game objects should be considered out of bounds for the player.

This section has one Sensei Stop.

Victory Lap

This section is important because the Ninja will be expected to add win and loss screens to the remaining Red Belt games.

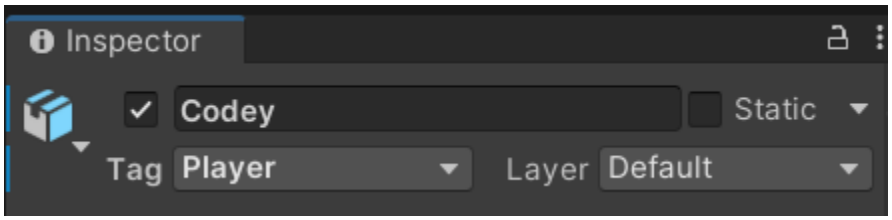
This section has one Sensei Stop.

Sensei Stops

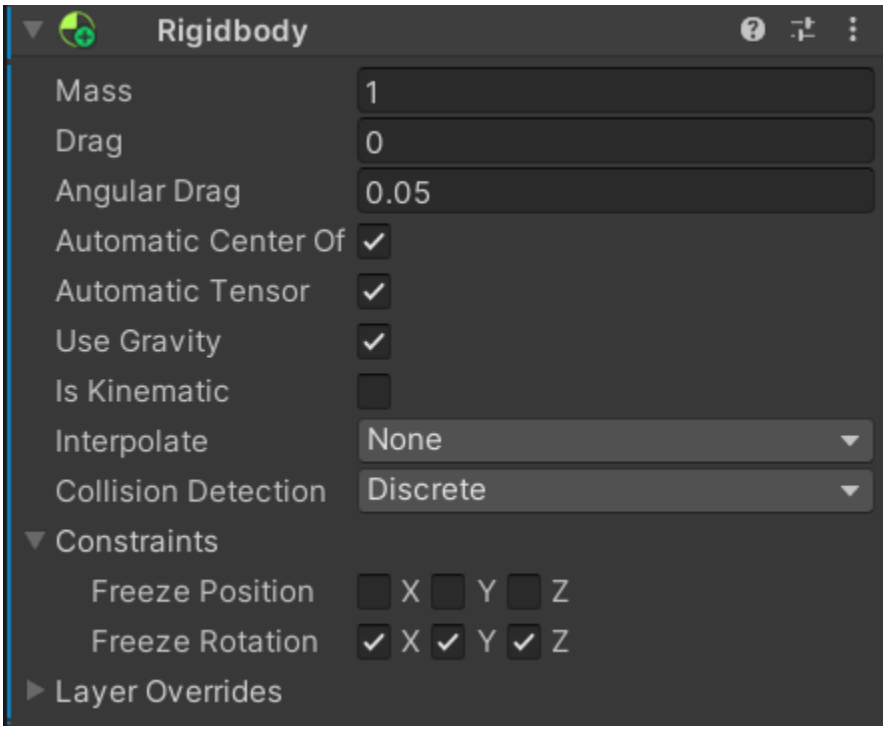
5 *Using the list above, add the required components to the Codey model. If you get stuck, work with your Sensei to add the components that you are missing.*

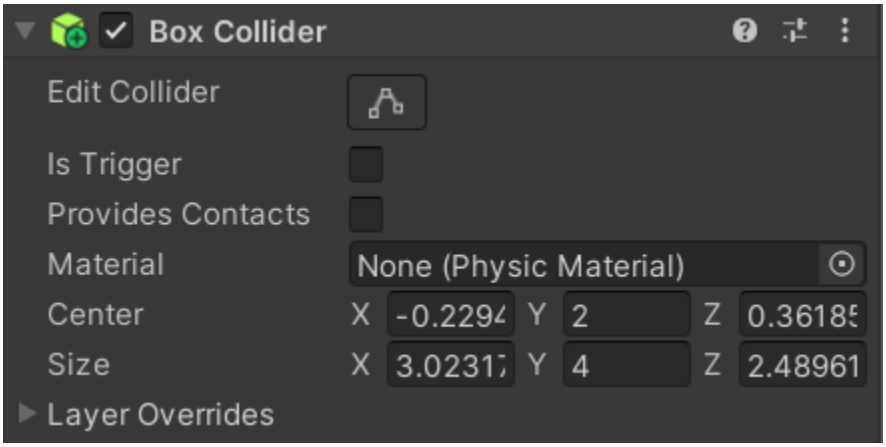
It is important that the Ninja completes this Sensei Stop before moving on. If they do not properly set up their Codey model, the game might not behave as expected.

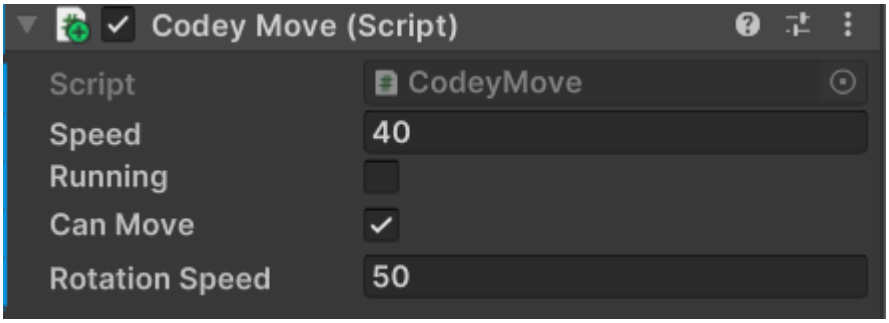
The Codey game object should have the Player tag.



The Codey game object should have Rigidbody component with "Use Gravity" enabled, a Box Collider component that encloses Codey, and the Codey Move script.







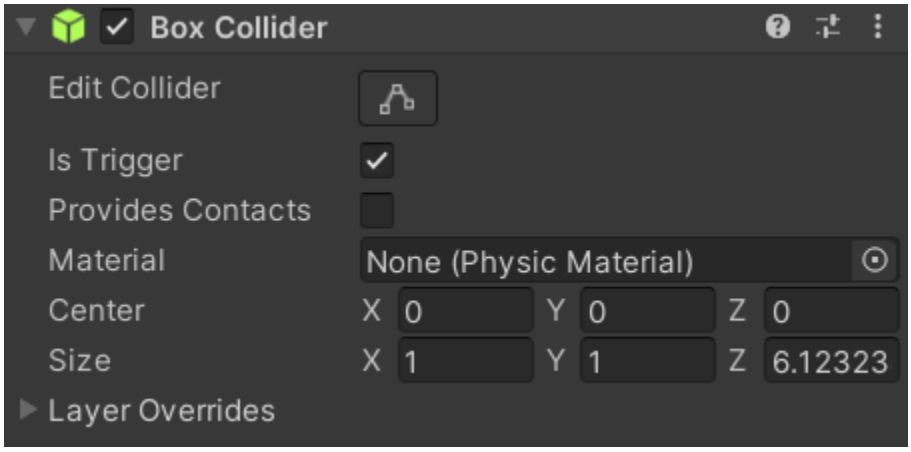
22 *What Unity tools are you using to build your track? Are you changing the rotation, using your mouse, or making the scale negative? Explain what happens to the track depending on the tool you are using.*

Ask the Ninja why they decided to use each tool. Encourage the Ninja to explain how each tool helped them put together the track.

The Move Tool positions the game objects around the scene. The Rotate Tool spins the game objects around an axis. The Scale Tool resizes game objects. The Vertex Tool helps snap together game objects by using corners instead of centers.

59 *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.*

The Ninja must add a box collider to the finishLine game object, enable the Is Trigger property, and adjust the size of the collider.



If the Ninja is struggling with determining what needs to be added, ask them about how Unity knows when two objects are colliding. Have them explain the difference between a regular collider that prevents objects from intersecting and a trigger collider that allows objects to be inside other objects. Ask them why we want our finish line collider to be a trigger. What would happen if it was a solid collider that prevented Codey from crossing?

72 *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.*

Have the Ninja demonstrate what happens in their game when the Is Trigger is disabled and when it is enabled. If the property is disabled, Codey should not be able to "go inside" the checkpoint's collider and the OnTriggerEnter function will not run causing the didCollide variable to never change to true. If the property is enabled, then Codey is able to pass through the checkpoint's collider and the OnTriggerEnter function will run causing the didCollide variable to change to true.

Encourage the Ninja to experiment and discover the differences on their own. Use the following guiding questions to help the Ninja if necessary.

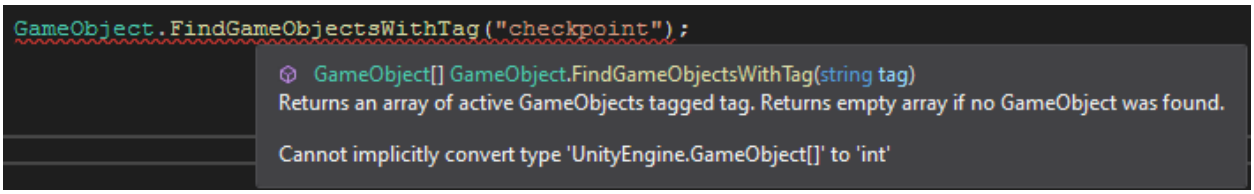
How can we check if two objects have collided?
How can we check if Codey has collided with the checkpoint?
What components are required to detect when the two objects have collided?
What tag is the checkpoint looking for?
What component in the checkpoint do we have to modify to make sure it is a trigger?
Does the size of the colliders matter?

78 *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?*

Since each Ninja was asked to set up their own unique checkpoint system, they might have missed a simple yet essential step for the function to work. First, ask them how many objects they are using as checkpoints. Then, ask them what special property each checkpoint needs to be found by the FindGameObjectsWithTag function. Finally, ask if each of their checkpoints has that special property, the "checkpoint" tag.

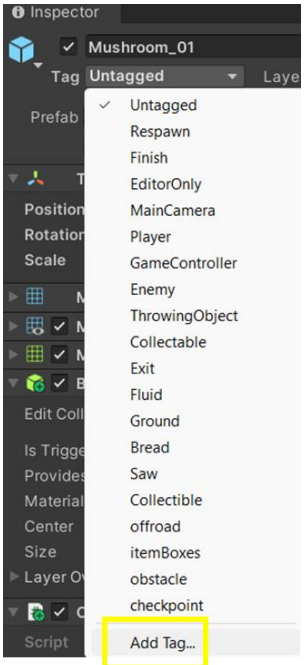
There is an error in the Start function because of a type mismatch. Ask the Ninja what type of object the function returns. Ask what type of object the variable stores.

Show them that they can hover their mouse over errors in Visual Studio to get an explanation.

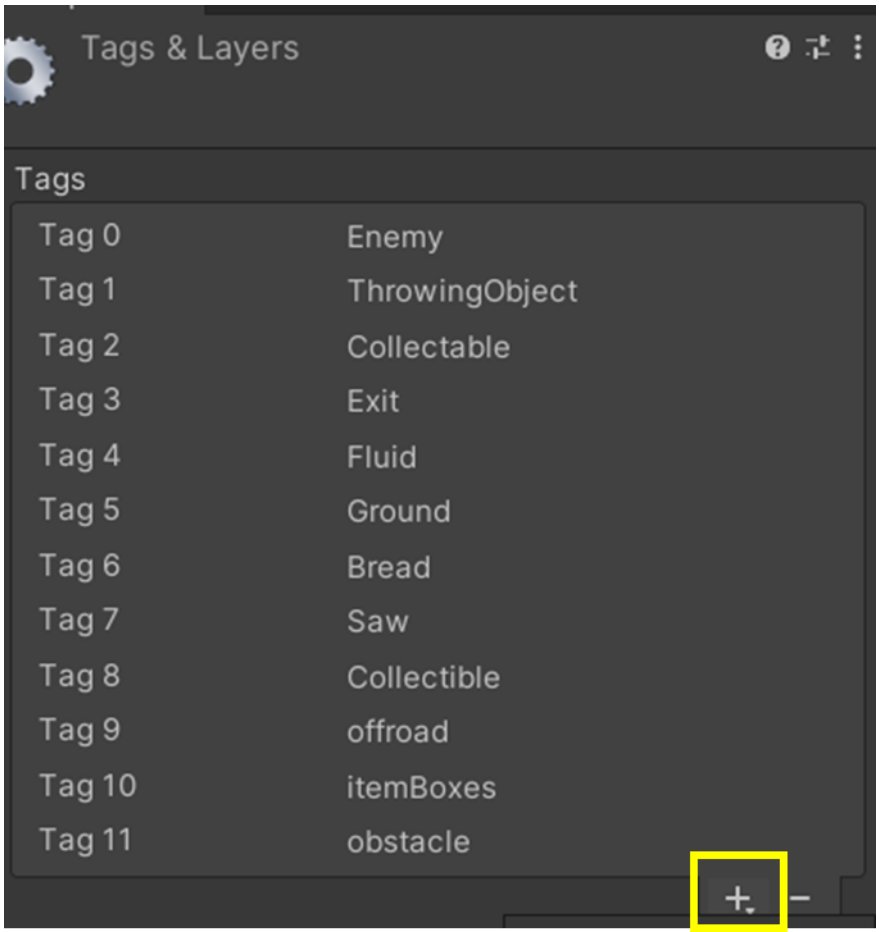


If the Ninja has not set up the custom "checkpoint" tag, then walk them through the following steps.

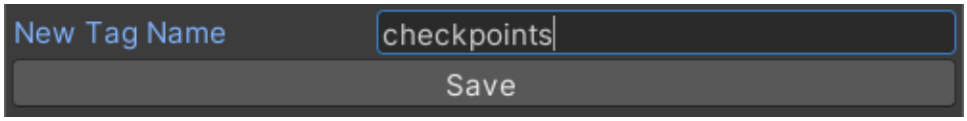
Select one of the child game objects and in the Inspector create a new tag.



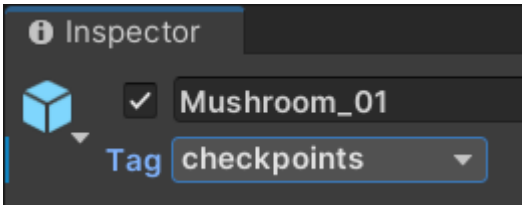
Click on the **+.**



Name the new tag **checkpoint** and click Save.



In the Hierarchy select the game object, in our example it is the Mushroom_01 and give it the checkpoint tag you created.



Change the tag for all the game objects that are in the Checkpoints game object.

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

The Ninja should be able to accurately show you that the numberOfCheckpoints and triggeredCheckpoints variables equal when Codey goes through all the checkpoints. Ask the Ninja to explain why triggeredCheckpoints is 0 but numberOfCheckpoints is 4 at the start of the game.

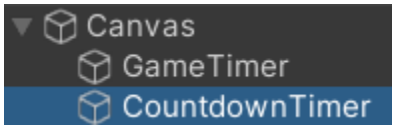
Have the Ninja playtest a scenario where Codey does not go through one of the checkpoints. Have them explain what happens if Codey goes through the checkpoints out of the intended order.

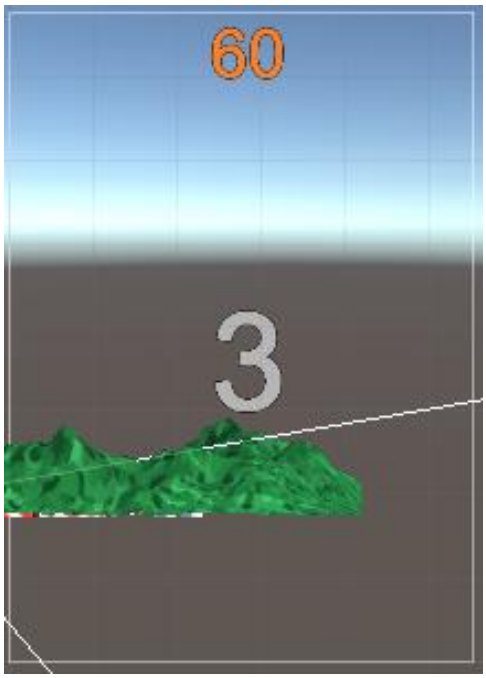
88 *Describe to your Code Sensei how your if condition has changed. Then demonstrate what happens when you cross the finish line without going through all the checkpoints versus when you have gone through all checkpoints.*

The Ninja should be able to explain that the TriggerFinishLine script accesses the variables on the CheckpointCounter script through the checkpointTracker variable. When the Codey object crosses the finish line, the values of triggeredCheckpoints and numberOfCheckpoints are compared. If they are equal, then print a console message saying the player wins. If the two values are not equal, that means the player has not crossed all checkpoints and a cheater message is printed to the console.

101 *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.*

The Ninja will either go through the entire process again in the previous steps or they can duplicate the **GameTimer** object and rename it to **CountdownTimer**. The text objects must look different from one another.





105 *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?*

If the Ninja is struggling with understanding the difference between a float and an int, provide examples for each type. Ask them what is different between the numbers on a stopwatch and the number of songs in a playlist.

The Ninja should be able to explain that an integer is like a whole number and a float can contain decimals. Ask them to provide specific examples.

106 *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?*

Ask the Ninja which text object they attached to the Lap Time and the Start Countdown variables. Ask them how much time Codey needs to complete their track and how they came to that value. Ask them how many seconds they want to give the player to get ready before the race.

The **Lap Time** component should have the **GameTimer** text object.

The **Start Countdown** component should contain the **CountdownTimer** text object.

The **Total Lap Time** component will match the amount of time the **GameTimer** text. In our example the **GameTimer** has the text set to "**60"** so our **Total Lap Time** component will be **60**.

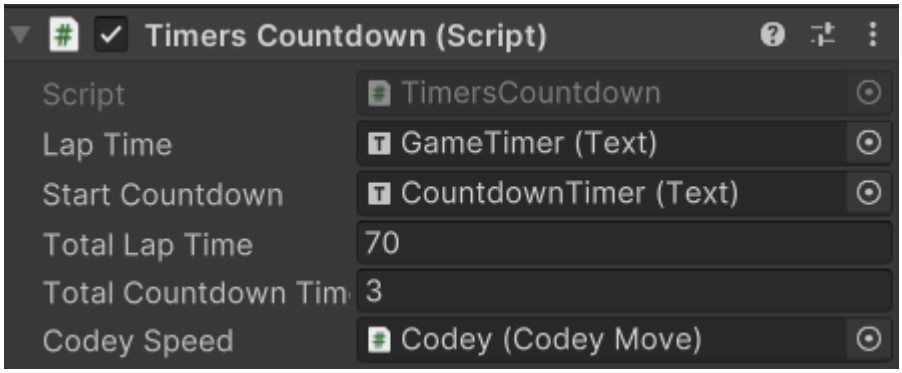
The **Total Countdown Time** component will match the **CountdownTimer** text. In our example the **CountdownTimer** has the text set to "**3"** so our **Total Countdown Time** component will be **3**.

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

The Ninja must first create a variable that allows us to grab the speed variable from the **CodeyMove** script.

```
public CodeyMove codey;
```

Save the script and return to Unity. Select the GameManager and in the script component make sure you add CodeyMove in the component.



Then they will need to check when if the totalCountdownTime is greater than 0. Only then can they countdown and they must set Codey's speed to 0.

```
if (totalCountdownTime > 0)
{
    totalCountdownTime -= Time.deltaTime;
    startCountdown.text = totalCountdownTime.ToString();
    codey.Speed = 0;
}
```

Once the totalCountdownTime is less than or equal to zero, the totalLapTime counter will begin counting down and Codey's speed can be set to any number that they want.

```
if (totalCountdownTime <= 0)
{
    startCountdown.text = "";
    totalLapTime -= Time.deltaTime;
    lapTime.text = totalLapTime.ToString();
    codey.Speed = 40;
}
```

The last conditional checks when the totalLapTime is less than 0. When this is true, the "Time is up" message will show in the console.

```
if (totalLapTime < 0)
{
    print("Time is up!");
}
```

Final example script.

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

    public float totalLapTime;
    public float totalCountdownTime;

    public CodeyMove codey;

    0 references
    void Update()
    {
        if (totalCountdownTime > 0)
        {
            totalCountdownTime -= Time.deltaTime;
            startCountdown.text = Mathf.Round(totalCountdownTime).ToString();
            codey.Speed = 0;
        }

        if (totalCountdownTime <= 0)
        {
            startCountdown.text = "";
            totalLapTime -= Time.deltaTime;
            lapTime.text = Mathf.Round(totalLapTime).ToString();
            codey.Speed = 40;
        }

        if (totalLapTime < 0)
        {
            print("Time is up!");
        }
    }
}
```

Make sure the Ninja changed the code and didn't just add additional lines based on the pseudocode. For example, if they kept the original lines that decrease the timers by Time.deltaTime and then added the lines a second time in the new if statements, then the timers will count down twice as fast.

131 *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!*

This is a good opportunity to check the Ninja's understanding of a for loop. Ask them how many parameters are needed in a for loop in C# and in JavaScript. To ensure they have this script's loop set up properly, have them point out and explain their starting value and the ending value and how that relates to the maxItemBoxes variable.

The Ninja does not need to use i as the index variable, but it is conventional. Ensure that the Ninja starts the loop at index 0. The run condition must use the less than comparison operator to ensure the correct number of boxes is spawned. The Ninja can increment the index variable by using ++ or += 1.

```
void Start()
{
    for (int i = 0; i < numberOfBoxes; i++)
    {

    }
}
```

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

Ask the Ninja if they looked back at Shape Jam to review the three parameters that the Instantiate function needs. Have them explain where they want the objects to spawn.

The first parameter of the Instantiate function should be the itemBox game object.

The second parameter should be the spawnLocation's transform's position.

The third parameter is the spawnLocation's transform's rotation.

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

In the next few steps, the Ninja will update this code so the boxes are spread out through the track.

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

This Sensei Stop is an opportunity to have the Ninjas demonstrate their understanding of how the index variable of a for loop works. Ask them to demonstrate and explain what happens when you have a value greater than zero for none, one, and both modifyXPosition and modifyZPosition. Ask them to demonstrate what happens when a large number like 100 is used.

Since all scenes will be different, encourage the Ninja to playtest their game to determine the best values for their track.

147 *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.*

This is a good opportunity to have the Ninja explain the difference between 2D and 3D.

Ask the Ninja what parts of the Super Shapes activity can be used to help us rotate the item boxes. Ask them to point out how they modified the code to account for the Z axis.

```
0 references
void Update()
{
    transform.Rotate(new Vector3(100, 100, 100), Time.deltaTime * 100);
}
```

We want the itemBox to rotate constantly so we place it in the Update function.

For the most part the code is the same, except for the new Vector3 change. By using new Vector3 the ninja can tell the item boxes how strong the rotation is on each axis. The Ninja can change the values of 100 to anything they want.

The second argument rotates the itemBox on each frame using **Time.deltaTime**. If the Ninja wants to speed up or slow down the rotation, then they can change the multiplier.

149 *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?*

This is a good opportunity to check the Ninja's understanding of Booleans. Have them explain how the parameter of SetActive changes the state of the game object. Ask the Ninja to explain how they need to check to see if Codey collided with the item box. Ask them what happens if an object other than Codey collides with a box.

The Ninja should use the OnCollisionEnter function to detect when an object collides with the item box. If the other object that collides with the box has a "Player" tag, then we know it is Codey. Since this script is attached to the itemBox game object, we can use gameObject.SetActive(false) to disable it in the scene.

```
0 references
public class itemBoxFeatures : MonoBehaviour
{

    0 references
    void Update()
    {
        transform.Rotate(new Vector3(100, 100, 100), Time.deltaTime * 100);
    }
    0 references
    private void OnCollisionEnter(Collision other) {
        if (other.gameObject.tag == "Player") {
            gameObject.SetActive(false);
        }
    }
}
```

143 *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.*

This is a good opportunity to check the Ninja's understanding of functions. Have them explain how to run the code in the body of functions. Ask if they understand the difference between calling a function and invoking a function.

The Invoke function's first parameter is a string of the function name – capitalization matters. The second parameter is the time to wait as a float. Make sure the Ninja places an f after the number so Unity interprets it as a float and not another number type.

```
private void OnCollisionEnter(Collision other)
{
    if (other.gameObject.tag == "Player")
    {
        gameObject.SetActive(false);
        Invoke("itemBoxRespawn", 5f);
    }
}
0 references
private void itemBoxRespawn()
{
    gameObject.SetActive(true);
}
```

168 *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.*

This is a good opportunity to check the Ninja's understanding of player input and conditional statements. Ask the Ninja to think back to previous games to help them remember how to check for user input. Help them use the Project Settings to find and change all the Input keywords.

Ask the Ninja about the importance of Instantiate's three parameters.

While there may be slight variations, the Ninja's code must check for user input and a chosen powerup before instantiating the chosen powerup at Codey's position.

```
void Update()
{
    if (Input.GetKeyDown("space") && chosenPowerup)
    {
        Instantiate(chosenPowerup, transform.position, transform.rotation);
    }
}
```

```
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (chosenPowerup != null)
        {
            Vector3 powerupLocation = new Vector3(transform.position.x, transform.position.y, transform.position.z);
            Instantiate(chosenPowerup, powerupLocation, transform.rotation);
        }
    }
}
```

This code will get updated in the next Sensei Stop.

170 *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.*

Ask the Ninja to explain if the X, Y, or Z values of the spawn position need to be modified. Have them explain why setting chosenPowerup to null after a powerup is spawned stops Codey from instantiating more than one at a time.

The solution to the positioning problem can vary based on how the Ninja completed the previous Sensei Stop. Since each Ninja's scene is different, they need to experiment with any numbers they use to find good values for their game.

```
void Update()
{
    if (Input.GetKeyDown("space") && chosenPowerup)
    {
        Instantiate(chosenPowerup, transform.position + new Vector3(5, 2, 0), transform.rotation);
        chosenPowerup = null;
    }
}
```

```
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (chosenPowerup != null)
        {
            Vector3 powerupLocation = new Vector3(
                transform.position.x + 5,
                transform.position.y + 2,
                transform.position.z);
            Instantiate(chosenPowerup, powerupLocation, transform.rotation);
            chosenPowerup = null;
        }
    }
}
```

174 *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?*

Ask the Ninja what function they used to detect collision. Ask them how they can use tags to make sure the shell only interacts with obstacles.

The Destroy function should be used to completely remove the objects from the game. The obstacle can be referenced using other.gameObject and the shell can be referenced using gameObject.

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

180 *Following steps 2 through 8 of Labrinth, make your road walkable and program the logic in the NavMeshMovement script. Add the collision code from the Shell object so the NavMeshShell can also destroy obstacles.*

Help the Ninja find the relevant sections of Brown Belt's Labyrinth activity. Have the Ninja explain what a NavMesh does. Have them explain what pieces of the track should and should not be walkable. Have them explain what steps can be replicated to help program the NavMeshShell object.

The Ninja is instructed to follow steps 2 through 8 in the Labyrinth activity. If the Ninja used the track pieces provided by Code Ninjas, they can make the **road** object is walkable but not the **curb** object. You may select multiple objects at once by holding down the control key.

Only the road should turn blue when you activate the NavMesh.

Ask the Ninja what must be add to the beginning of the script so the NavMesh can be used. Have the Ninja define the NavMeshShell's destination. Ask them what code they copied from the Shell object.

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


0 references
public class NavMeshMovement : MonoBehaviour
{
    private NavMeshAgent agent;
    private GameObject obstacle;
    0 references
    void Start()
    {
        obstacle = GameObject.FindGameObjectWithTag("obstacle");
        agent = GetComponent<NavMeshAgent>();
        agent.destination = obstacle.transform.position;
    }

    0 references
    private void OnCollisionEnter(Collision other) {
        if(other.gameObject.tag == "obstacle"){
            Destroy(other.gameObject);
            Destroy(gameObject);
        }
    }
}
```

You must include **using UnityEngine.AI** in order to use the NavMesh.

When the **NavMeshShell** is Instantiated, we need to find the nearest obstacle by using the FindGameObjectWithTag function. Setting the NavMeshAgent's destination to the position of the found obstacle will cause the shell to navigate to it based on the "walkable" track.

The Shell's collision code can be used.

The only big difference between this game's implementation and Labyrinth's is the use of FindGameObjectWithTag instead of using a public variable. It is important to note that Unity will not always fine the obstacle that is closest to Codey!

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

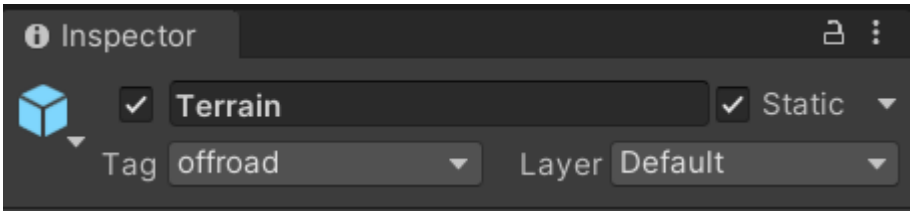
Ask the Ninja to explain what logic was coded in Meany bird and how it can be applied to this game. Ask the Ninja how they are checking to see if Codey is colliding with terrain.

You must include **using UnityEngine.SceneManagement** in order to use the SceneManager.

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

0 references
public class Respawn : MonoBehaviour
{
    0 references
    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.tag == "offroad")
        {
            SceneManager.LoadScene(0);
        }
    }
}
```

The Ninja needs to use a unique tag for the terrain. In this example solution, we are using the “offroad” tag.

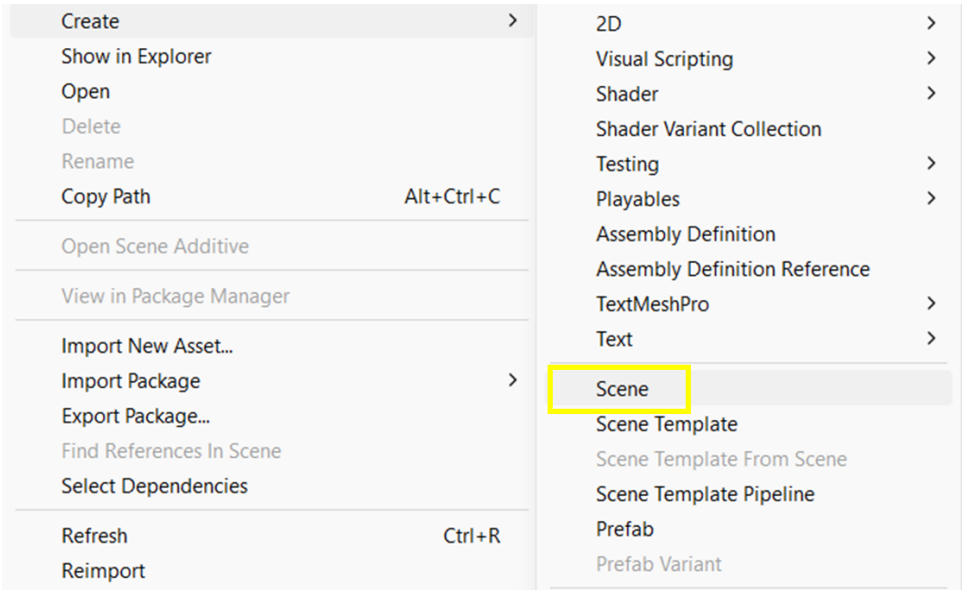


The steps to add the scene to the Build Settings of the Project are in the curriculum.

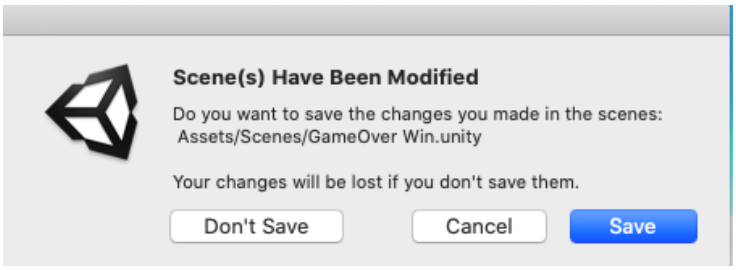
190 *You can create a new canvas with a simple win or lose message! You have all creative freedom in this activity! If you need help, discuss your message idea with your Code Sensei.*

Ask the Ninja to explain how to create new scenes. Ask them what special game object is required for user interfaces. Ask the Ninja where this new scene management code should go.

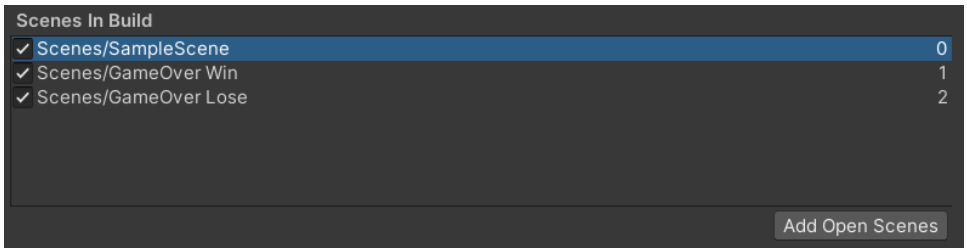
Ninjas need to create two new Scenes. While the names don't matter for the logic, they should still be given appropriate names.



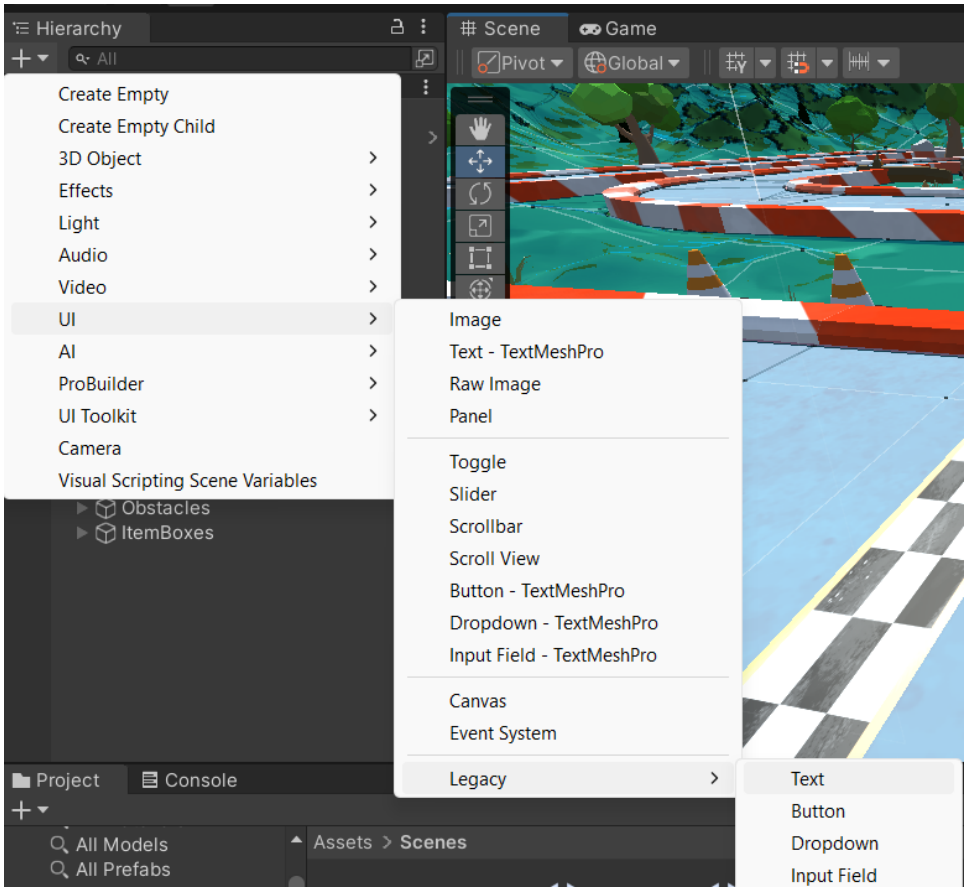
Make sure the Ninja clicks **Save** when changing scene so none of their work is lost.



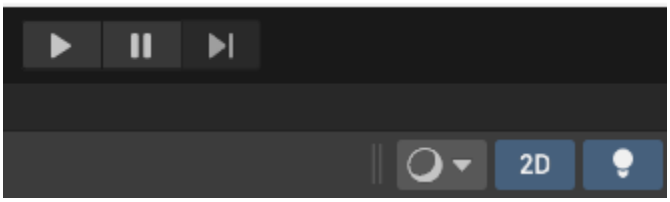
The Ninjas must add their scenes to the **Build Settings** before they can be loaded by the Scene Manager.



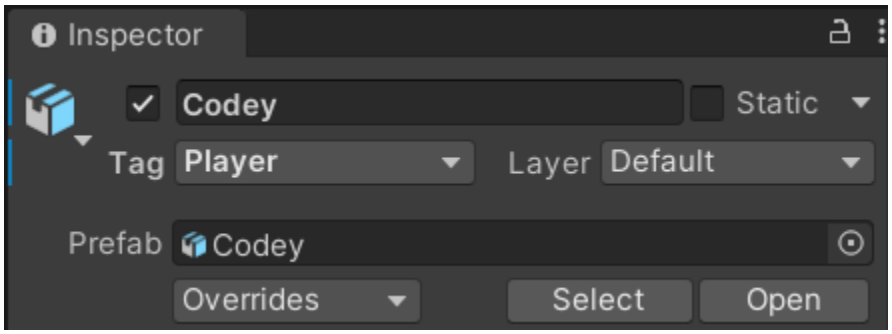
The scenes should at least have one Legacy UI Text object.

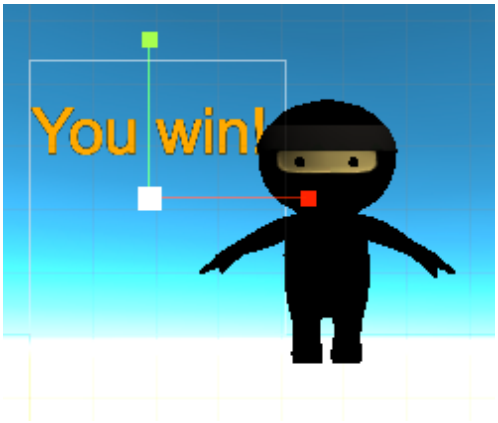


Remind the Ninja that using the 2D scene view makes it is easier to work with user interfaces.

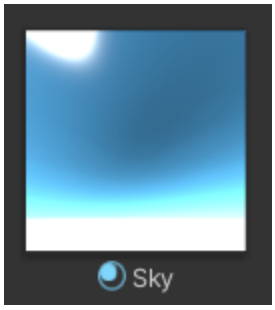


If they want to include Codey in the scene, they must position the camera or scale Codey up.

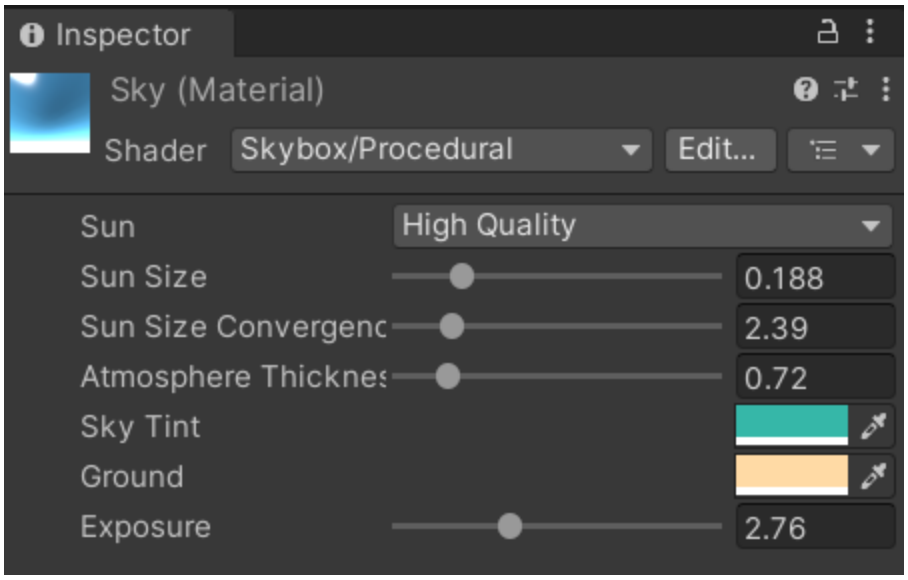




In the **Materials** folder, you can drag the **Sky** object onto the scene.



Select the **Sky** object and in the Inspector modify the **Sky Tint** and **Ground** colors to your liking.



Repeat these steps for the **GameOver Lost** scene.

In the TimersCountdown script, add using UnityEngine.SceneManagement to the top and load the game over scene if the total lap time is less than zero. The parameter of LoadScene should be based on the game over scene's index in the Build Settings menu.

```
if (totalLapTime < 0)
{
    print("Time is up!");
    SceneManager.LoadScene(2);
}
```

In the **TriggerFinishLine** script, add using UnityEngine.SceneManagement to the top and load the you win scene if they player crossed the finish line. The parameter of LoadScene should be based on the win scene's index in the Build Settings menu.

```
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Player")
    {
        if (checkpointTracker.triggeredCheckpoints == checkpointTracker.numberOfCheckpoints)
        {
            print("You Win!");
            SceneManager.LoadScene(1);
        }
        else
        {
            print("Cheater!");
        }
    }
}
```

Sulky Slimes

Section Notes

Project Setup

Have the Ninja start a new Unity project and give it a name so they can identify the project later. Renaming the Sample Scene is important because it prevents imported assets from overwriting their work. Instructions to setup Cinemachine will be provided in a later step.

A Whole New World

Have the Ninja use the Unity Asset Store. The free assets used in the manual are from Nature Pack Light and ADG Textures. The Ninja should have two empty game objects to hold the static elements in the scene. If the Ninja does not have the Environment and the Catapult objects, explain the importance of keeping an organized Hierarchy. The Platform object can be a simple cube or an asset from the store, but it must be flat so the objects can sit properly.

The catapult the platform must be oriented so a line can be drawn from the catapult to the platform in the positive Z direction. The code to launch the object will throw the it in the positive Z direction. If the Ninja's scene does not follow this orientation, then the object will launch unpredictably.

Character Building

The free asset used in the manual is called RPG Monster Duo PBR Polyart. The Ninja can use any asset as their object if it is a reasonable size and has the correct components. A character with a basic shape will work best – a complex object like a ragdoll might have unpredictable behaviors when interacting with the Unity Physics Engine.

The character's Rigidbody component should look just like the image in step 9. The mass, drag, and angular drag do not matter at this point in the curriculum, but they can be altered based on playtesting.

The character can have any type of Collider. If the Ninja can't decide, explain that a Box Collider will behave like rolling dice and the others will be more like rolling a marble. It is up to them how they want their character to behave. The Collider type can be changed at any point.

A Mesh collider requires a Mesh object to work properly. If the Ninja happens to choose an asset that does not have its own Mesh object, use one of the other Collider types.

The character must have exactly one Rigidbody and exactly one Collider. It is a good idea to click through each game object attached to the character to ensure that there isn't an extra Rigidbody or Collider buried in a child game object.

Some Assets will have an Animator component. The Ninja can choose to enable or disable the component, but Apply Root Motion must be unchecked. When enabled, this option tells Unity to use the object's animations in its physics calculations. For example, with

Apply Root Motion enabled, character that lunges forward would dynamically move the character forward in the scene. Disabling Apply Root Motion would let the animation play without altering the true location of the object in the scene.

Mighty Mouse

The Ninja must add a script named MouseManager to the MouseManager game object. Later in the curriculum, the game object will be renamed. This will help teach that the code and the scripts matter more than the name of any of the objects.

The Unity Input object handles listening for user input. The object provides different functions for different input types. There are three methods for the three states of a mouse button. GetMouseButtonDown will return true on the frame that the user clicks the mouse button. GetMouseButton will return true on every frame that the user is pressing the mouse button. GetMouseButtonUp will return true on the frame that the user releases the mouse button. Each function takes one integer as an argument. The left mouse button is 0, the right mouse button is 1, and the middle mouse button is 2.

The multiplier values in the components of the launchVector will need to be changed based on the Ninja's scene. Each of the three values need to be fine-tuned by the Ninja based on distance between the catapult and the platform and the mass of the object

There is one Sensei Stop in this section.

To Infinity...

Help the Ninja see the connection between the user's input and the game world. If Ninjas struggle with understanding normalizing vectors, review steps 7 and 8 of Shape Jam.

The Cinemachine component can be tricky to get right. If the Ninja changes too many properties, the CM vcam1 object can be removed from the scene and a new one can be added via the Cinemachine menu.

Help the Ninja adjust the values of all the numbers until their object launches how they want.

Dragging the slime with the mouse can be tricky because the slime will not interact with other objects in the scene. Help the Ninja find a good value that will clearly show the trajectory of the slime while also not letting it sink below the ground.

There are two Sensei Stops in this section.

Reach for the Star

Help the Ninja find objects in the asset store. Make sure that all their objects have colliders that are not triggers. Make sure that all objects have rigidbodies. These objects should be affected by gravity. If they do not fall to the ground, then check the properties of the components.

Help the Ninja find a collectable object in the asset store. Make sure the object has a collider that is set up as a trigger.

If the Ninja struggles with any of the Sensei Stops, the solution code provided should be general enough to work for any game as long as the curriculum was followed.

There are three Sensei Stops in this section.

A Slime's Life

If the Ninja does not make their health image a Sprite in step 52, then it will not behave properly. The Ninja can use any number of lives and the logic should still work properly.

Make sure the Ninja keeps the order of the images consistent across the UI and the array.

Though anchoring UI elements can be tricky, it is necessary because the UI will move and scale based on the screen size.

There are two Sensei Stops in this section.

A Change of Scenery

The Ninjas should not start their new scene from scratch. Duplicating the original scene keeps all of the objects, scripts, and relationships. If they followed the curriculum, they should be able to replace objects in the Environment scene without impacting any game logic or scripts.

If they choose to replace any objects that have scripts attached, they will have to reattach the scripts and reassign any public variables.

There are no Sensei Stops in this section.

Ninja Touches

Encourage the Ninja to extend and add to their game. Ask them what previous games they enjoyed and if they can borrow any concepts or code to add to their Sulky Slimes game.

Sensei Stops

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

The Unity Mouse Coordinate System's origin is the bottom left of the scene. Since the position is calculated in 2D, the z coordinate will always be zero.

29 *Attempt to code these two lines of code. When the user releases the mouse button, set the slime's rigidbody's isKinematic property to false. Then, add an impulse force to rigidbody in the direction of our launch vector with the strength of our launch force. Reference Robomaina from Brown belt if you need help adding the force.*

The Ninja should add two lines of code in the MouseManager's Update function inside of the third if statement.

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

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

    if (Input.GetMouseButtonUp(0))
    {
        slimeRigidbody.isKinematic = false;
        slimeRigidbody.AddForce(launchVector * launchForce, ForceMode.Impulse);
    }
}
```

When isKinematic is true, Unity expects the coder to program all the physics interactions of the object. If isKinematic is false, then the Unity Physics engine can act on the object. Since we want Unity to handle all of the physics, we must first ensure that the Rigidbody's isKinematic is set to false.

We can then add a force using the Rigidbody's AddForce method. The first argument is a Vector3 that determines the direction of the force. We are using the normalized launchVector Vector3 multiplied by the launchForce float. Normalizing the vector and multiplying it by a constant lets us finetune the individual components of the launchVector while being able to simply adjust the overall strength of the force. The second argument uses the ForceMode enum to instruct Unity to apply this force as an Impulse that acts instantly and only once.

31 *Write the code described to reset the slime. What types of variables do you need? Playtest your game and make sure your game works after multiple resets.*

There should be one Vector3 class variable and one Quaternion class variable. They can be public or private. Inside the Start function, these variables need to be set to the transform's position and rotation properties respectively.

```
private Vector3 originalSlimePosition;
private Quaternion originalSlimeRotation;

0 references
void Start()
{
    originalSlimePosition = slimeTransform.position;
    originalSlimeRotation = slimeTransform.rotation;
}
```

Inside of the Update function, there should be a new if statement that checks to see if a button or key has been pressed. Posible options are Input.GetMouseButtonDown(1) to check to see if the user right clicked or Input.GetKeyDown("space") to see if the user pressed the space bar.

Inside this if statement, the rigidbody's isKinematic property must be set to true so it ignores the Unity physics engine. The transform's position and rotation properties must be reassigned to the original values stored in the new variables.

```
if (Input.GetMouseButtonDown(1))
{
    slimeRigidbody.isKinematic = true;
    slimeTransform.position = originalSlimePosition;
    slimeTransform.rotation = originalSlimeRotation;
}
```

39 *Program the two lines of code described above. What direction do you want your collectable to move in?*

Inside of the Collectable script's Start function, set the startingPosition variable equal to the transform's position. Set the endingPosition variable equal to a new vector that uses the value of the original position and the distanceToMove variable. Though the provided solution below moves the collectable in only the x direction, the Ninja should choose the direction that the collectable moves. Once the remaining logic is complete, encourage the Ninja to playtest different scenarios.

```
0 references
void Start()
{
    startingPosition = transform.position;
    endingPosition = new Vector3(
        transform.position.x + distanceToMove,
        transform.position.y,
        transform.position.z
        );
}
```

41 *Use the pseudocode to help you write the collectable's Update function.*

The Ninjas are provided with this pseudocode outline:

In the Update Function...

- If the x value of the transform's position is to the left of the x value of the starting position, then
    - set the direction to positive one.
- If the x value of the transform's position is to the right of the x value of the ending position, then
    - set the direction to negative one.
- Always set the position to a new vector that uses the speed, direction, and Time.deltaTime to move the collectable.

Both of the following possible solutions have the same two if statements. The difference is how the object's position is changed.

This solution sets the transform's position to a new vector that is composed of the individual coordinates of the position. The x component is being incremented by the speed, direction, and time elapsed since the last Update call.

```
public float speed = 0.1f;
public float direction = -1f;

0 references
void Update()
{
    if (transform.position.x < startingPosition.x)
    {
        speed = -1f;
    }
    if (transform.position.x > endingPosition.x)
    {
        speed = 1f;
    }
    transform.position = new Vector3(
        transform.position.x + speed * direction * Time.deltaTime,
        transform.position.y,
        transform.position.z);
}
```

This solution takes the transform's original position and increments it in the x direction by an amount based on the speed, direction, and time elapsed since the last Update call. Note that y and z components are 0.

```
public float speed = 0.1f;
public float direction = -1f;

0 references
void Update()
{
    if (transform.position.x < startingPosition.x)
    {
        speed = -1f;
    }
    if (transform.position.x > endingPosition.x)
    {
        speed = 1f;
    }
    transform.position += new Vector3(speed * direction * Time.deltaTime, 0, 0);
}
```

The Ninja might have slightly different code if they decide to move their collectable in a different direction.

44 *Use the OnTriggerEnter function to disable the collectable from the scene when the slime collides with it.*

```
0 references
private void OnTriggerEnter(Collider other)
{
    gameObject.SetActive(false);
}
```

The Ninja might use the Destroy function. From the player's perspective, Destroy does the same thing as SetActive – the object disappears. However, Destroy will remove the object from the scene and will stop all active scripts. Later on in the curriculum, we will need to Invoke a function a few seconds after the collectible is disabled. Destroy will prevent the Invoke from activating the function.

48: Plan out all the jobs that the LivesManager has. It needs to know how many lives the player has. What else? You can draw a flowchart or diagram to help you organize your thoughts.

The LivesManager script needs to know the total number of lives, how to remove a life, and if the game is over. The curriculum does not cover adding lives, but encourage the Ninja to program that logic on their own.

55 *Where in the Update function should we call the RemoveLife function? Call the livesManager's RemoveLife function and playtest your game. Talk with your Sensei about where you placed it in your code.*

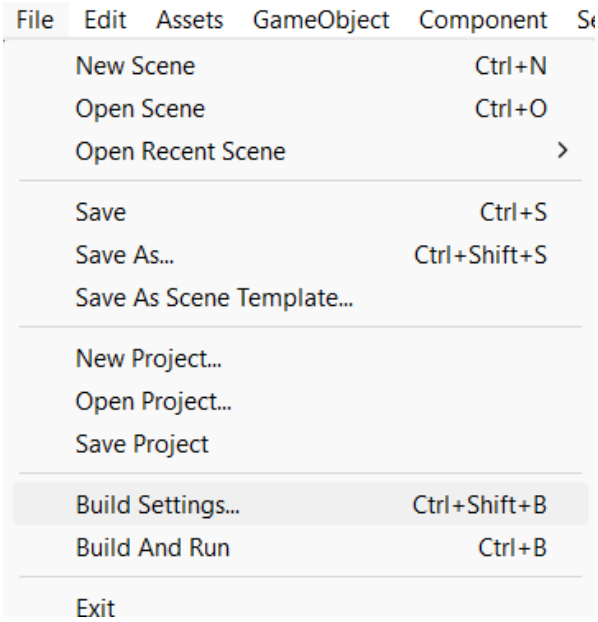
The Ninja should add livesManager.RemoveLife(); inside of the if statement that resets the slime.

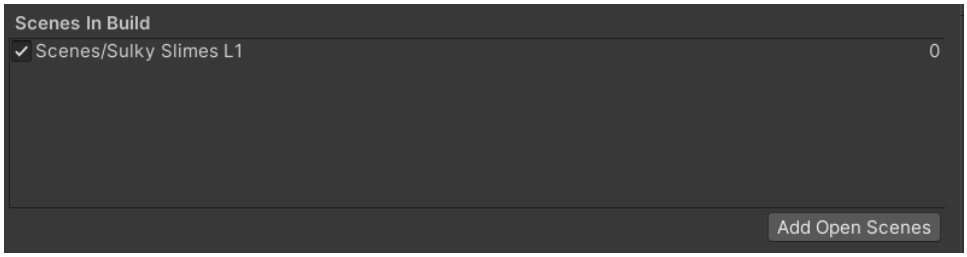
```
if (Input.GetMouseButtonDown(1))
{
    slimeRigidbody.isKinematic = true;
    slimeTransform.position = orignalSlimePosition;
    slimeTransform.rotation = originalSlimeRotation;
    livesManager.RemoveLife();
}
```

63 *Use Brown Belt's World of Color to use Unity's Scene Manger to reload the game when the user runs out of lines. Start with Step 5.*

The Ninja should be able to follow the steps from World of Color.

First, the scene must be added in the build settings menu.





Inside the LivesManager script, they must add using UnityEngine.SceneManagement; to the top of the script.

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

In the RemoveLife function, check to see if lives is less than 0 and then use the SceneManager's LoadScene function to load scene 0.

```
public void RemoveLife()
{
    lives -= 1;
    hearts[lives].SetActive(false);
    if (lives == 0)
    {
        SceneManager.LoadScene(0);
    }
}
```

Chef Codey

Section Notes

Project Setup

Have the Ninja start a new Unity project and give it a name so they can identify the project later. Renaming the Sample Scene is important because it prevents imported assets from overwriting their work. Instructions to setup Cinemachine will be provided in a later step.

A Room with a... Ninja

The Ninja should build their area based on their Ninja Planning Document. Help the Ninja find texture assets that are reasonably sized. It is important to note that 4k and other "mega" texture packs can be hundreds of megabytes.

This section has one Sensei Stop.

Extreme Café Makeover

The Ninja should add assets from the Unity Asset Store to decorate their scene. Help the Ninja find and import assets that support their vision. Encourage the Ninja to edit and update their Ninja Planning Document based on their game scene.

Help the Ninja add colliders to the objects in the scene.

Cinemachine can be tricky to get right since there are so many properties. Make sure the Follow and Look At properties are set to Codey's transform. Since Cinemachine will dynamically change the position and rotation of the camera based on Codey's position, use the Body section's Follow Offset to control the position of the camera.

This section has no Sensei Stops.

The Toast with the Most

The food and some other items in the scene are from the CoffeeShop Starter Pack by Puzzled Wizard.

A prefab is just a game object or group of game objects that are stored outside of the game scene as a template. The advantage of prefabs is being able to edit the template object and have it change all linked objects in the scene. Chef Codey does not require using prefabs, so the Ninja should use the "Unpack Prefab Completely" option to break the game object from the template. This will make sure that any changes the Ninja makes are not permanent – if they mess up they can re-add the original asset.

Encourage the Ninja to create empty game objects to help organize the objects in their scene.

If the Ninja is having trouble getting Codey to interact with the source object, make sure the correct collider is set to be a trigger. If it is not a trigger, then Codey will essentially be

interacting with an invisible wall. Make sure the Ninja is using OnTriggerEnter and OnTriggerExit and not OnCollisionEnter and OnCollisionExit.

When comparing strings, capitalization and extra spaces matter. For example, “bread”, “Bread”, and “bread “ are not equivalent strings. The Ninja must be careful when naming their game objects.

The programming instructions have several iterations. Ensure that the Ninja is replacing old code as the game becomes more complex.

There are several different versions of the Instantiate function. Make sure the Ninja’s Visual Studio did not autocomplete the wrong one.

This section has one Sensei Stop.

Now We’re Cooking

This section is very heavy on programming, but the curriculum takes the Ninja through each step. All of the code in this section is done in the Interact and Stove scripts. The Ninja will frequently be switching back and forth between the two scripts. This is to help enforce the idea that different scripts and objects have different jobs to perform. Make sure the Ninjas understand that how scripts and objects can interact with other scripts and objects.

This section has three Sensei Stops.

Order Up

Make sure the Ninja understands that a function is just a shortcut to run lines of code. As the script continues to grow, help the Ninja keep their code neat.

This section has one Sensei Stop.

How to Program an Egg

Ninjas can use the steps to set up the bread source object to help them set up their second source. The first step of this section can also be used to help the Ninja add more interactable objects to their scene.

This section has three Sensei Stops.

Time to Sizzle

Unity’s particle system can seem very complicated, but this curriculum only uses the Play and Stop functions. Encourage the Ninja to change all the settings inside the component.

Invoke has been used in previous games. Encourage the Ninja to experiment with the values of the second parameter to change how long the items need to cook. It is important for the first argument to be a string of the exact function name – case matters.

This section has two Sensei Stops.

Ninja Touches

Encourage the Ninja to extend and add to their game. Ask them what previous games they enjoyed and if they can borrow any concepts or code to add to their Sulky Slimes game.

Sensei Stops

21 *Discuss your room design with your sensei. Does this align with your planning document? Play test to make sure that Codey cannot run through walls. What type of textures did you use? How does this fit your theme?*

This is an opportunity for a review of the Ninja's game scene before any models are placed or coding is started. If the Ninja built a simple room like in the curriculum, but their planning document has a more complex area, then encourage them to build a scene that represents their vision.

Verify that Codey cannot run through any of the walls. Ensure that all barriers have colliders.

50 *While the slice of bread might look good at (0, 2, 2), it might not be quite right for other objects. Change the values of your Vector3 in the code to position your object. Tell your Sensei how you tested your different values.*

The Ninja was given the option to use their own theme. This is a good chance to check in with the Ninja to see if they are following along exactly with the curriculum or if they are using it as a guide to help them program the logic.

Ask the Ninja how the decided on their three coordinates. Did they randomly decide? Did they try big and small numbers? Did they change one coordinate at a time?

53 *Using the same conditional that we used for the bread trigger, can you code a message to appear if the space bar is pressed when Codey is inside the stove's box collider trigger?*

The Ninja should add an additional if statement under the check to see if the trigger name is bread. Help them understand the pattern of checking to see if Codey is inside a trigger. This logic will be applied throughout the game. This is very similar to step 42.

```
0 references
void Update()
{
    if (Input.GetKeyDown("space"))
    {
        if (triggerName == "Bread")
        {
            heldItem = Instantiate(breadPrefab, transform, false);
            heldItem.transform.localPosition = new Vector3(0, 2, 2);
        }

        if (triggerName == "Stove")
        {
            print("I'm at the stove!");
        }
    }
}
```

63 *Write a few sentences or design a flowchart to describe the ToastBread function's job. What's the first thing that it needs to do when Codey interacts with the stove? What does it need to do before Codey can interact with the stove again?*

The most important part of this Sensei Stop is to hear the Ninja's thought process on how Codey, the bread, and the stove are going to interact. Encourage the Ninja to write pseudocode or draw a diagram to help them.

The stove needs to check to see if Codey has a valid item (bread). It must then take that item from Codey and process it into something else (toast). Codey can then pick up the new item.

69 *Find the correct if statement and call the ToastBread function, destroy the object that Codey is holding, and reset the name of the object Codey is holding.*

In the Interact script's Update function, the Ninja needs to replace the "I'm at the stove" print statement and add three lines of code. The first line should use the local stove variable to call the ToastBread function. The next line should use Unity's Destroy function to destroy the heldItem game object. The last line should set the value of heldItemName to an empty string.

```
if (triggerName == "Stove")
{
    print("I'm at the stove!");
    if (heldItemName == "breadSlice")
    {
        stove.ToastBread();
        Destroy(heldItem);
        heldItemName = "";
    }
    else
    {
        print("Codey is empty handed!");
    }
}
```

87 *There's one other place that we have repeated code – we will create a function to help us soon! Talk with your Sensei about what code can be placed into a function.*

There are no wrong answers to this Sensei Stop. This is an opportunity to talk with the Ninja about the code they have written. Ask them if they can explain what the word refactor means.

93 *Discuss with your Sensei what two elements of the code determine the held item's model and name. What special feature can we add to a function that lets us change its inputs?*

The two arguments or parameters that need to be created are for the GameObject itemPrefab and the string itemName.

102 *Look back at how we first disabled the toast and then wrote a public function for Codey to use. Can you do the same thing for the fried egg? First, disable the fried egg object when the script starts. Create a public void function to fry the egg that sets the object to active and updates the value of cookedFood to "friedEgg".*

The new FryEgg function should mirror the ToastBread function.

```
0 references
void Start()
{
    toast.SetActive(false);
    friedEgg.SetActive(false);
}

0 references
public void FryEgg()
{
    friedEgg.SetActive(true);
    cookedFood = "friedEgg";
}
1 reference
public void ToastBread()
{
    toast.SetActive(true);
    cookedFood = "toast";
}
```

107 *Oh no! There's still egg in the stove. Something must be wrong with our CleanStove function. Investigate and tell your Sensei what one line of code needs to be added to fix this bug.*

The Ninja must add friedEgg.SetActive(false) somewhere in the CleanStove function.

```
2 references
public void CleanStove()
{
    toast.SetActive(false);
    friedEgg.SetActive(false);
    cookedFood = "";
}
```

126 *What happens when you try to pick up food before it's complete? Does anything stop Codey from picking up food immediately? Discuss with your Sensei how you could fix this bug.*

The purpose of this Sensei Stop is to get the Ninja thinking about how to analyze problems as they come up in game design and programming. The curriculum will lead them through a fix, but now is a good time to check the Ninja's understanding.

129 *Talk with your Sensei about how you can accomplish this. Before you add any code to your scripts, write a few sentences of pseudocode that describe when Codey can and cannot interact with the stove. Implement your code and playtest your game.*

The Ninja needs to add additional conditional checks to see if the stove is not cooking. Two possible solutions are below. Remind the ninja that && means "and" and ! means "not".

```
if (triggerName == "Stove" && !stove.isCooking)
{
    if (heldItemName == "breadSlice")
    {
        stove.ToastBread();
        PlaceHeldItem();
    }
    else if (heldItemName == "egg")
    {
        stove.FryEgg();
        PlaceHeldItem();
    }
    else
    {
        if (stove.cookedFood == "toast")
        {
            PickUpItem(breadPrefab, "toastSlice");
            stove.CleanStove();
        }
        if (stove.cookedFood == "friedEgg")
        {
            PickUpItem(friedEggPrefab, "friedEgg");
            stove.CleanStove();
        }
    }
}
```

```
if (triggerName == "Stove")
{
    if (stove.isCooking) return;
    if (heldItemName == "breadSlice")
    {
        stove.ToastBread();
        PlaceHeldItem();
    }
    else if (heldItemName == "egg")
    {
        stove.FryEgg();
        PlaceHeldItem();
    }
    else
    {
        if (stove.cookedFood == "toast")
        {
            PickUpItem(breadPrefab, "toastSlice");
            stove.CleanStove();
        }
        if (stove.cookedFood == "friedEgg")
        {
            PickUpItem(friedEggPrefab, "friedEgg");
            stove.CleanStove();
        }
    }
}
```