CODE NINJAS®

Platinum Belt Ninja Guide
Activity 01: Gravity Trails

Gravity Trails

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

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

Plan and Design

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

Hollow Knight by TeamCherry
Built in Unity

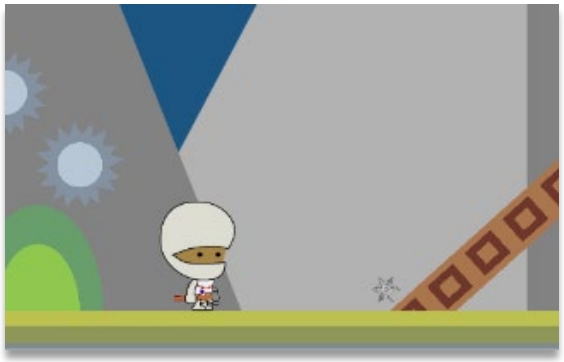
Cuphead by StudioMDHR
Built in Unity

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

Ninja Planning Document

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

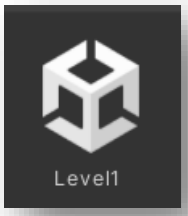
Take a look at the sample projects below for some inspiration!



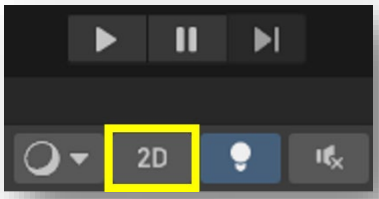
Project Setup

1 Start a new Unity Project and name it *YOUR INITIALS – Gravity Trails*. Select **3D template**. Make sure you are using the **2022.3 LTS** Unity Version!

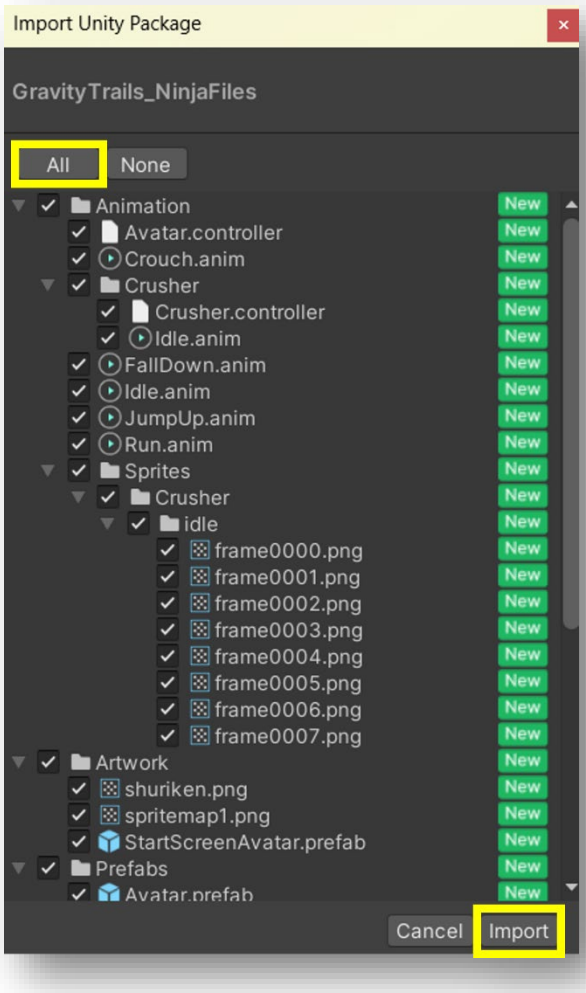
2 After it loads, rename the Sample Scene to Level 1.



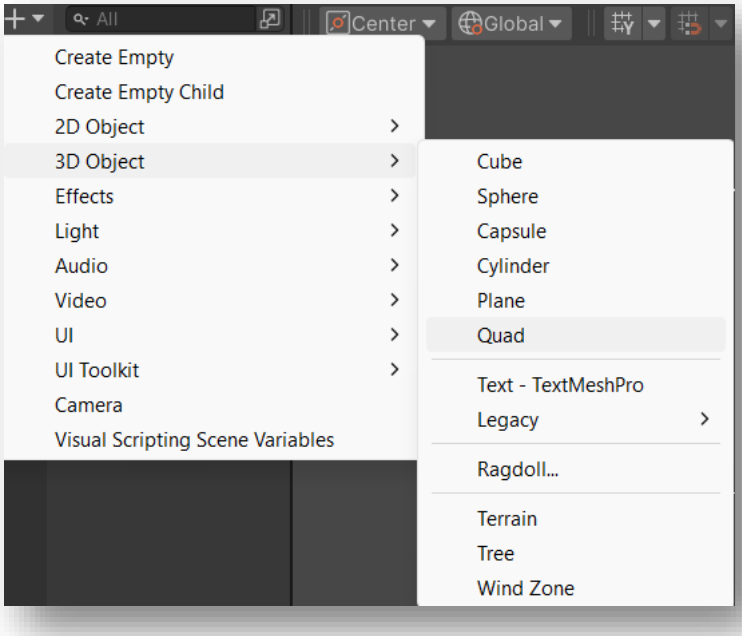
3 When your project first opens, it will have a 3D view. We can change that by clicking on the 2D button near the Play Button.

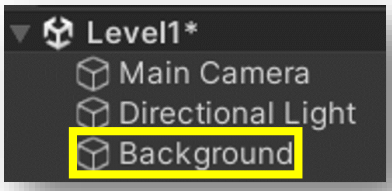


4 Next, import the **Activity 01 - GravityTrails_NinjaFiles.unitypackage** provided. This contains the background, the ninja, the enemies, and the other assets used to build level one. Click **All** to select all folders, then click **Import**.

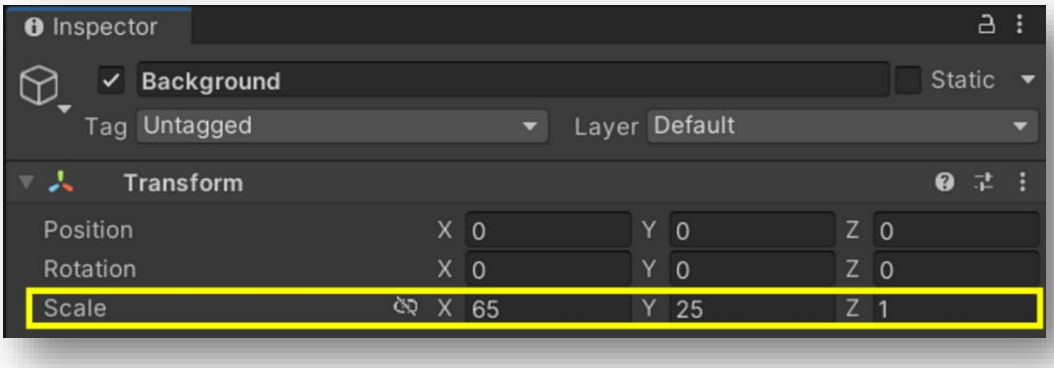


5 Create a **Quad object** in your scene. Rename it to Background.

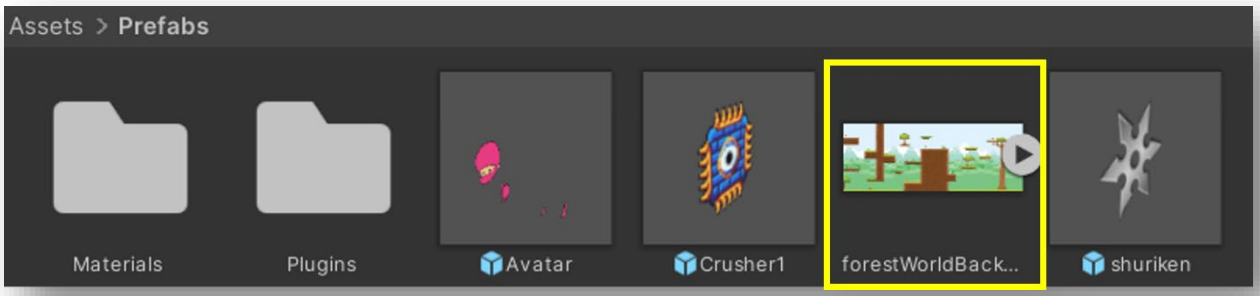




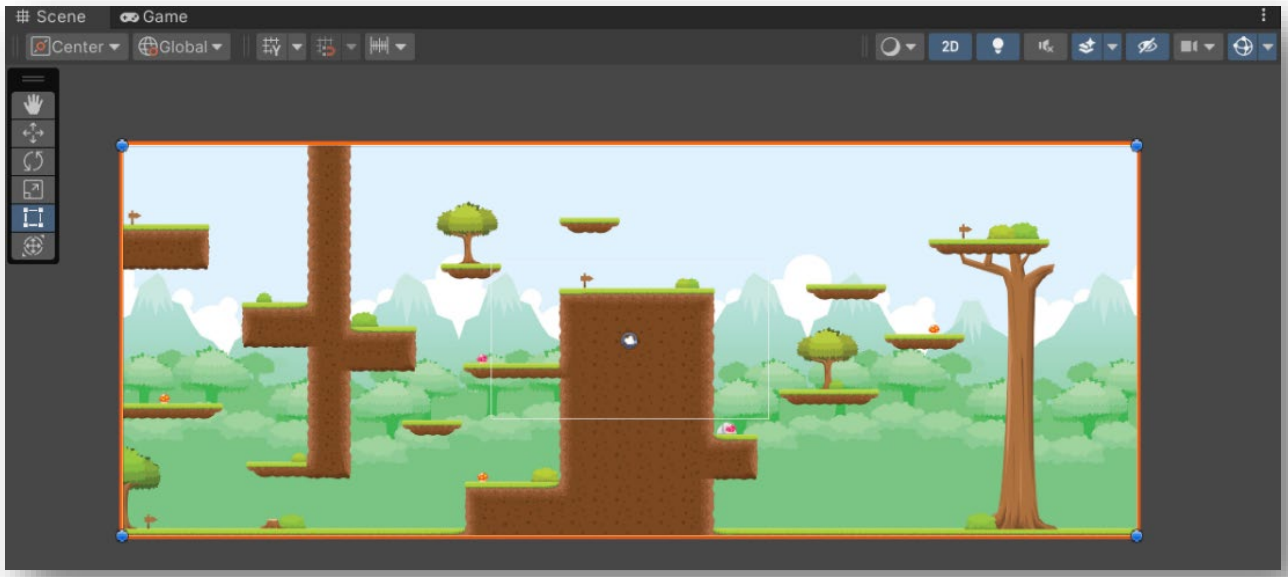
6 Adjust the Background's **Scale X** to **65** and the **Scale Y** to **25**. You may change these values later based on your playtesting.



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



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



Sensei Stop

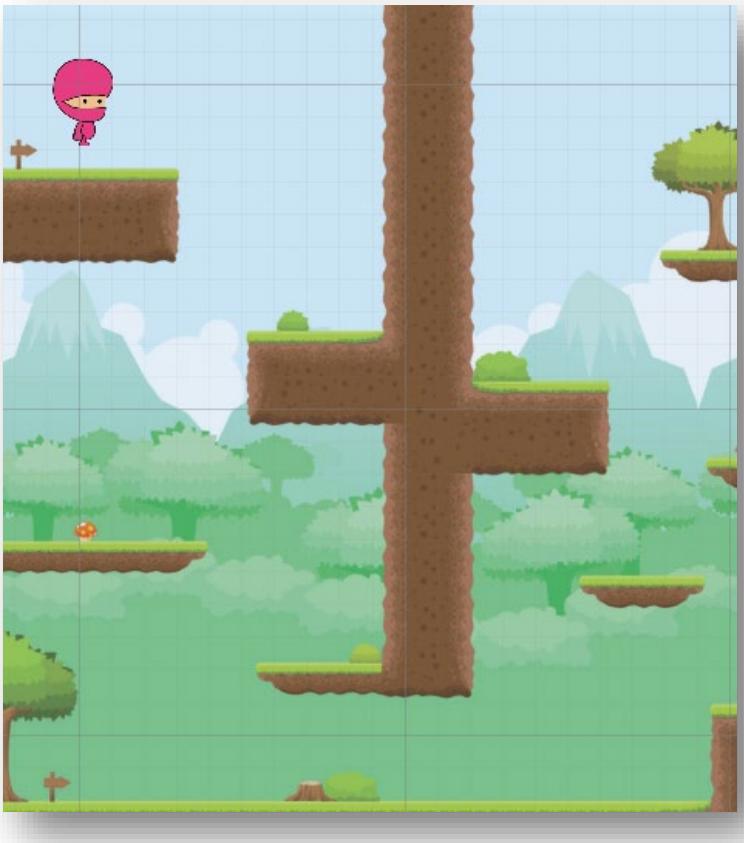
Demonstrate to your Code Sensei that you have created the Quad and attached the forestWorldBackground to it.

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

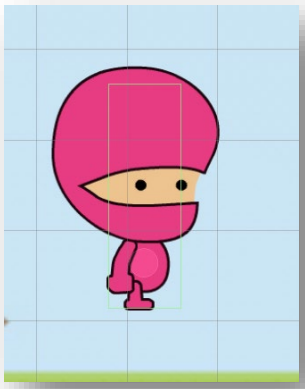
To turn it on, click on the "Auto Generate Lighting" button at the very bottom right of the Unity window.

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

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



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



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

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

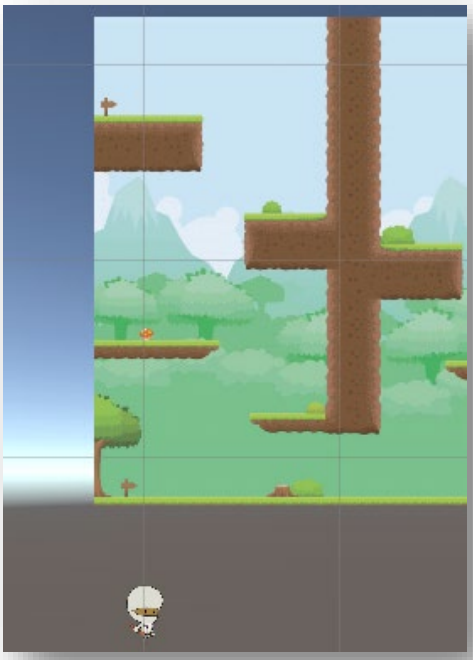
Sensei Stop

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

Your Avatar should like the image below once you make the changes you describe to your Code Sensei.

A Walk in the Park

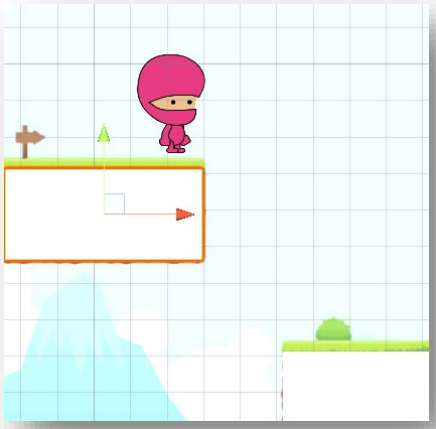
11 Playtest your game. What happens to the Avatar?



The Avatar is falling through the scene!

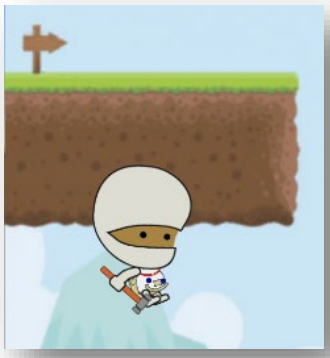
12 Similarly, to the Scavenger Hunt activity, we need to create multiple **Quad** objects to have walking surfaces for our Avatar. To help us to keep things organized, first create an **Empty Game Object,** and rename it **Surfaces**. Then create the Quad objects inside the Surfaces object.

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



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

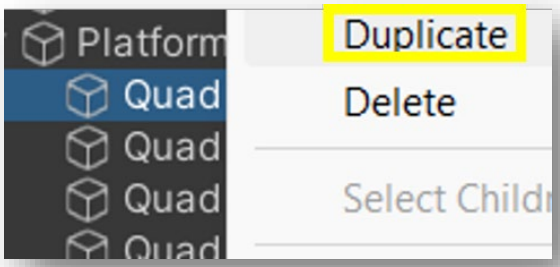
The Avatar is still falling through the Quads. What can we add to stop the Avatar from falling?



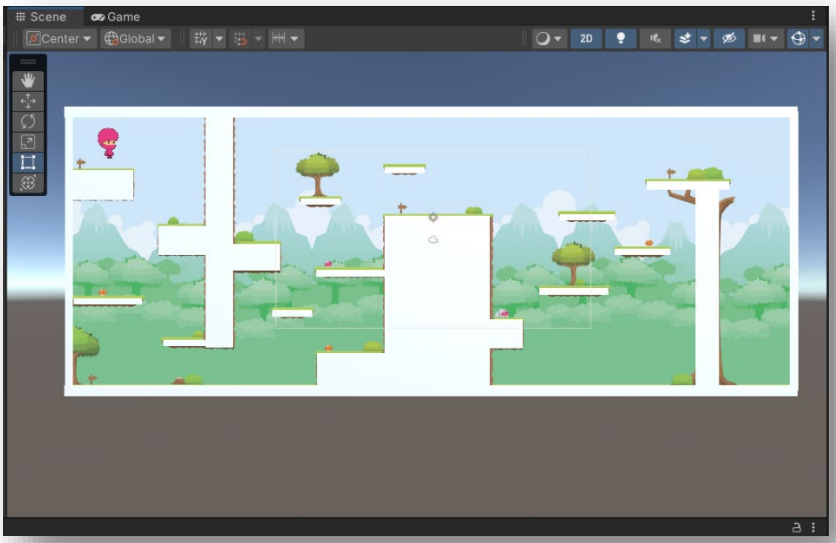
Sensei Stop

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

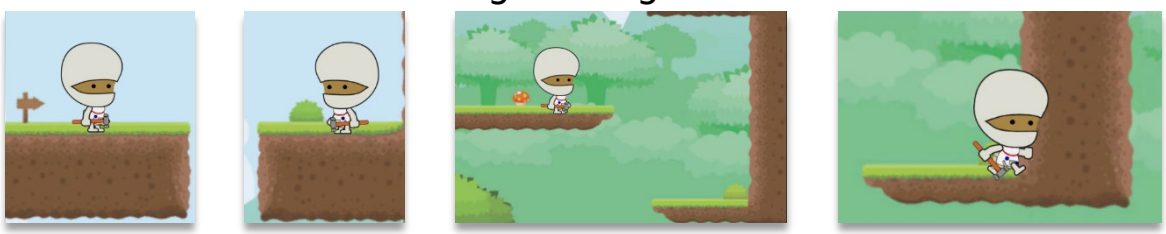
15 After you have modified your Quad objects to work with the Avatar, duplicate the original Quad by right clicking and selecting Duplicate or pressing Control + D.



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

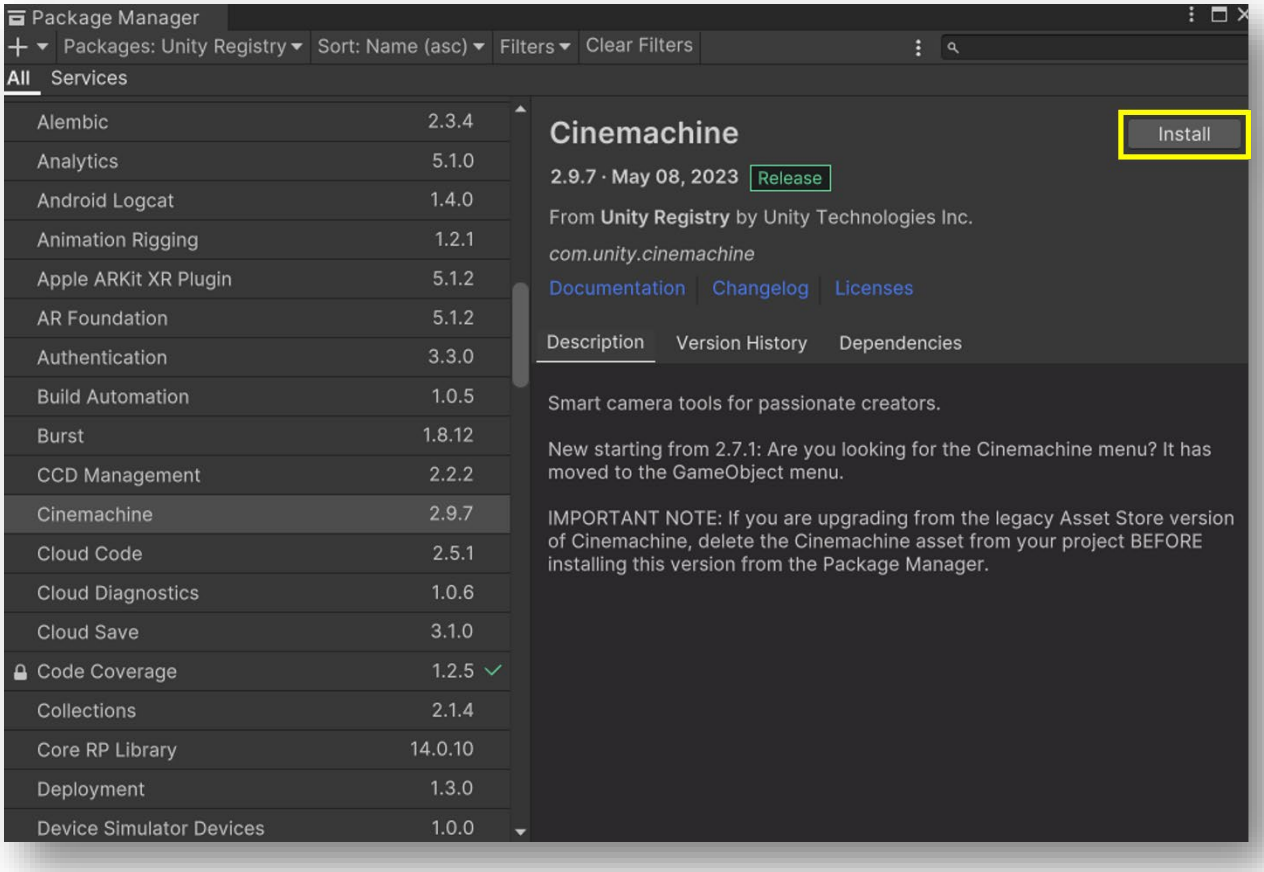


16 Playtest your game. Your Avatar will now be able to land on the Quad pieces and the Avatar will not be able to go through the brown dirt areas.

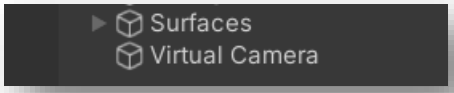


17 We want to use the Cinemachine to have our camera follow the Avatar.

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

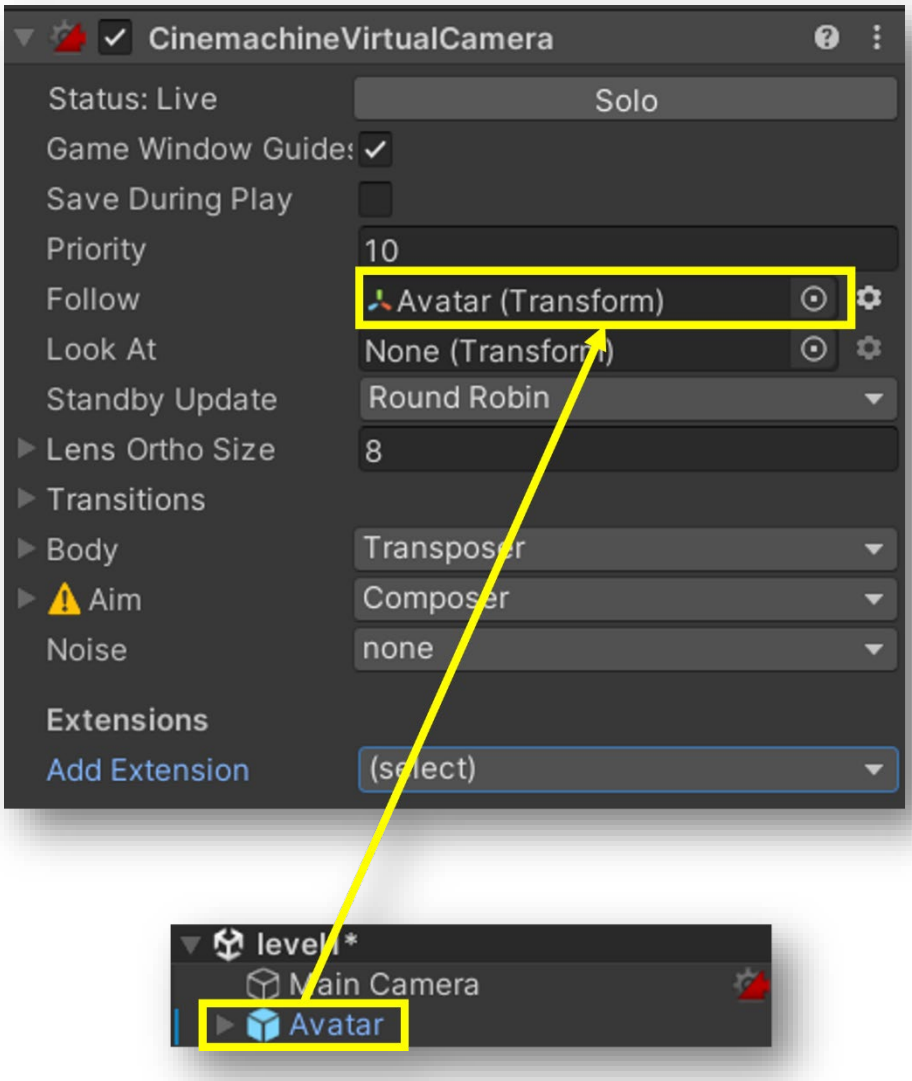


18 In the **Inspector** you should see a new game object Virtual Camera.

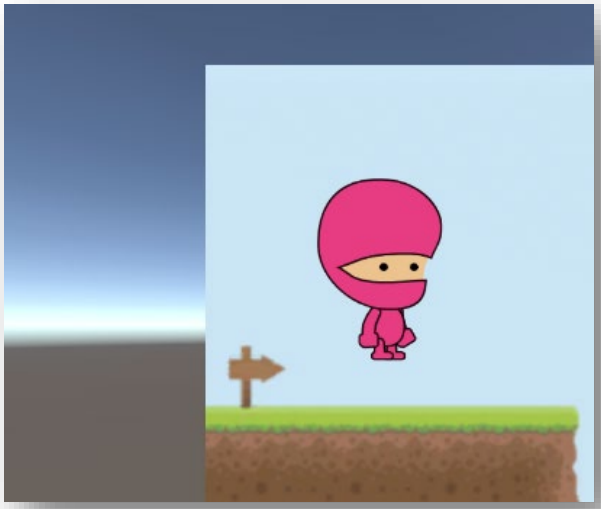


If you do not see this object, then click on **Cinemachine** next to Window at the top and select **Create Virtual Camera**.

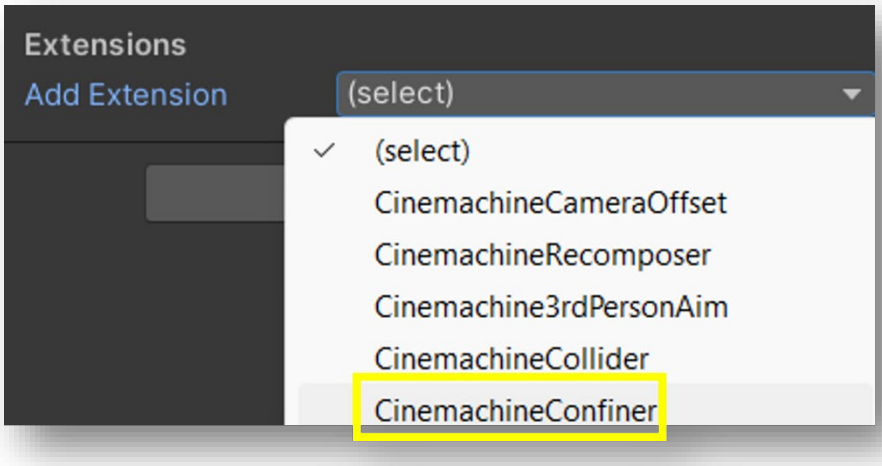
19 Select the **Virtual Camera game object**. In the Inspector, drag the Avatar game object into the **Follow** property.



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

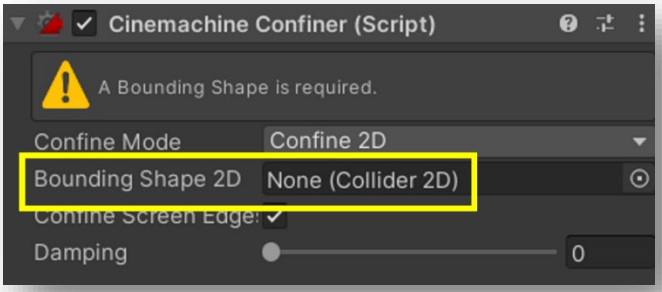


Select the **Virtual Camera game** object. Click **Add Extension** at the bottom of the CinemachineVirtualCamera component. Add the **Cinemachine Confiner** Extension.

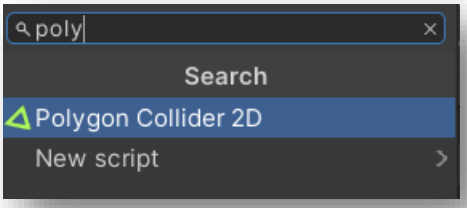


21

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

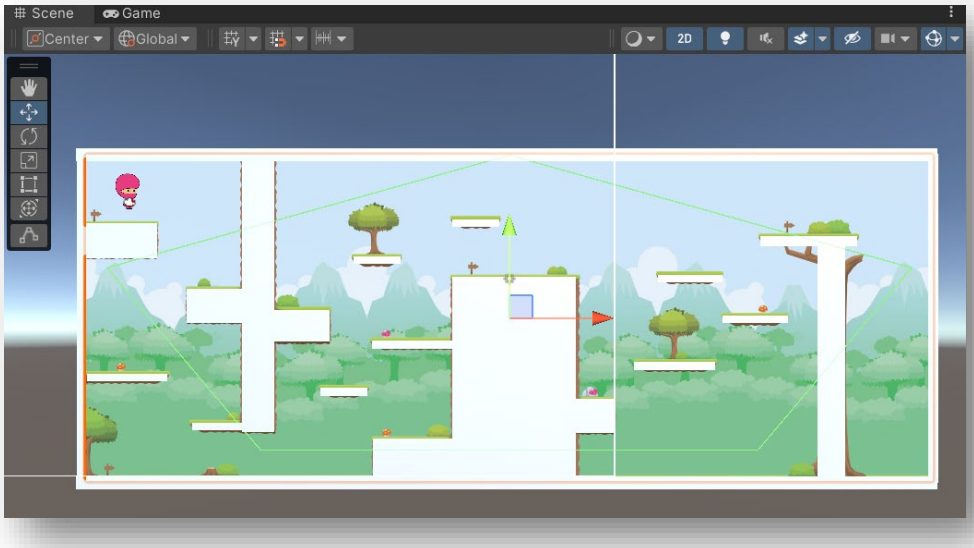


Select your Background game object and add a Polygon Collider 2D component.



22

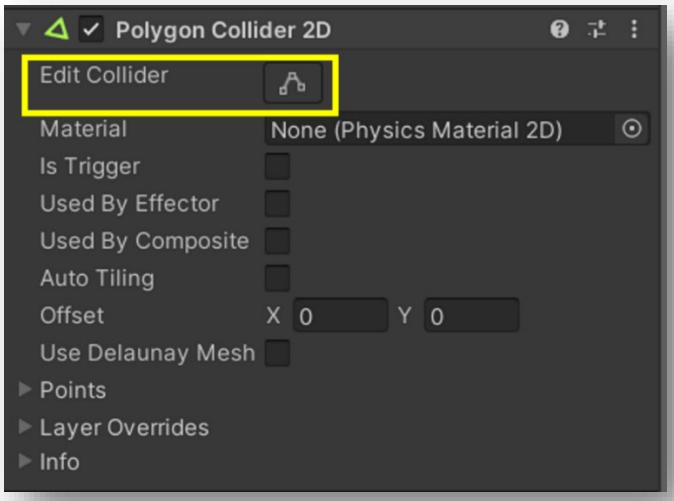
There should now be a green shape in your scene.



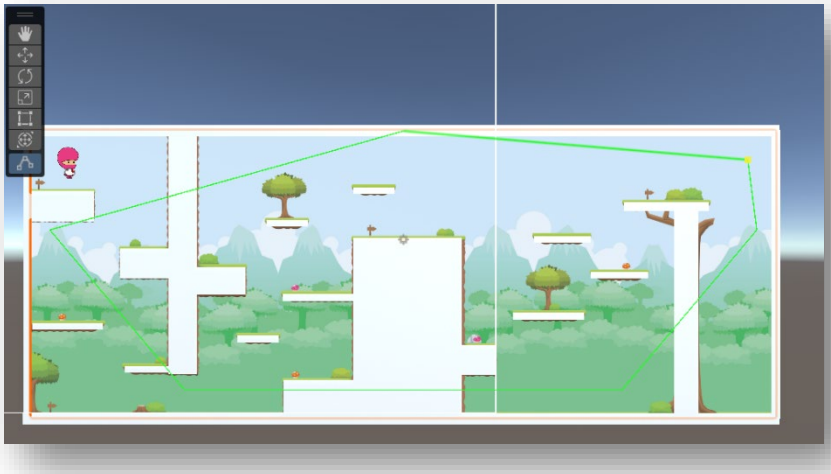
Lighting

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

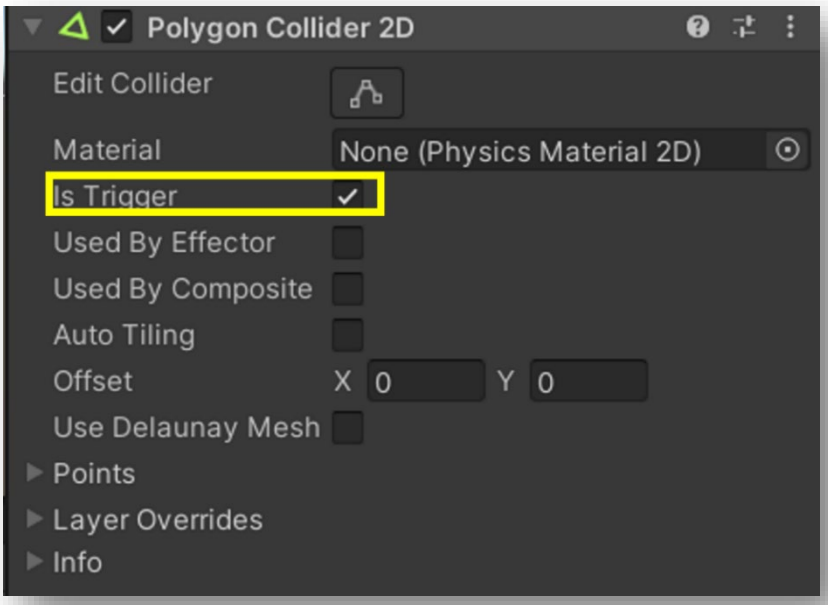
23 Click the Polygon Collider 2D component's Edit Collider button.



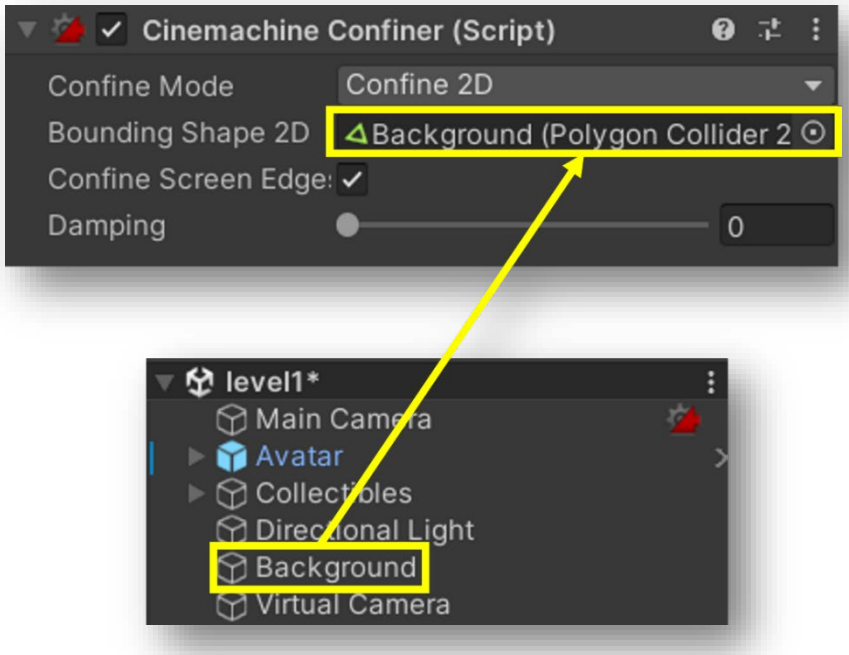
Drag the corners of the green polygon to the corners of the background image. Unity might draw thin green edges between the corners of your collider. When you are done, click the Edit Collider button again.



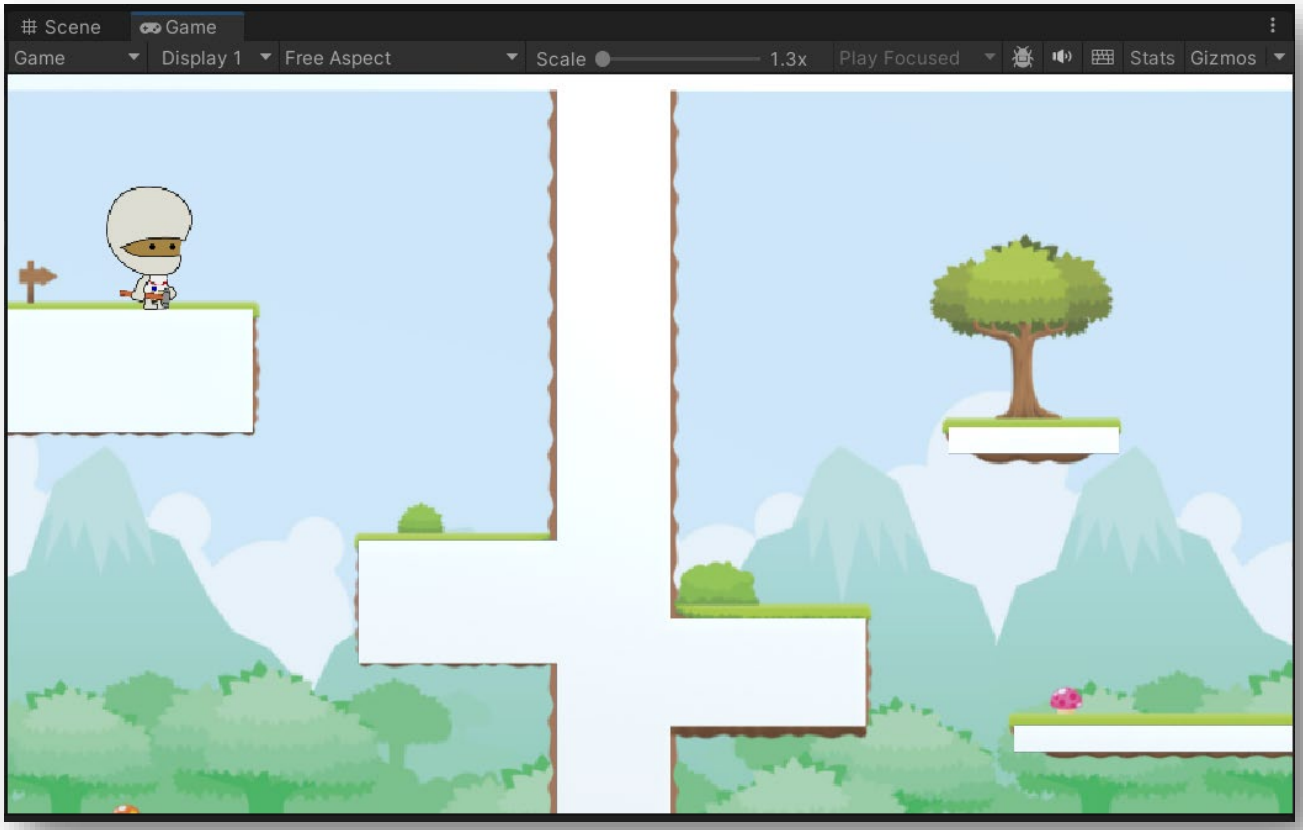
24 Enable the Polygon Collider 2D component's Is Trigger property. If you do not do this, then your Avatar will be forced outside of the background image when you play your game.



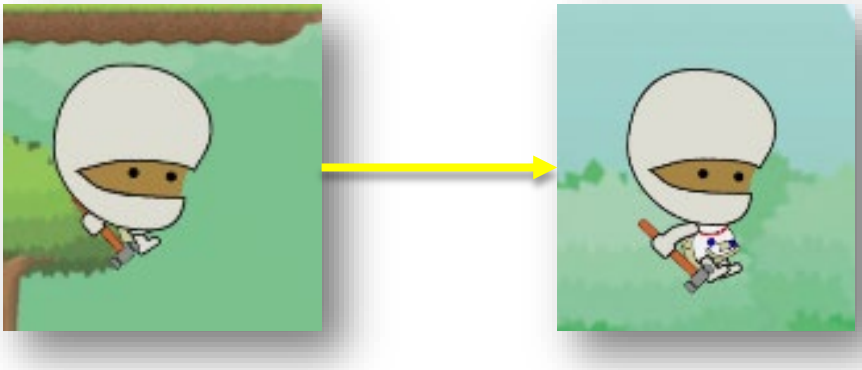
25 Drag in the background game object to the **Polygon Collider 2D** component.



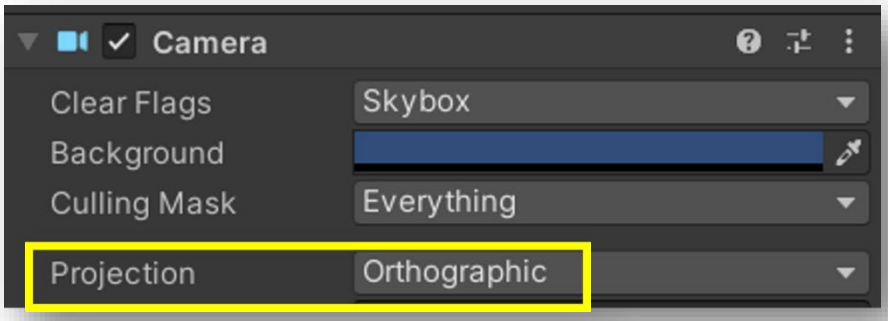
26 Playtest your game. Now the camera will stay inside the collider.



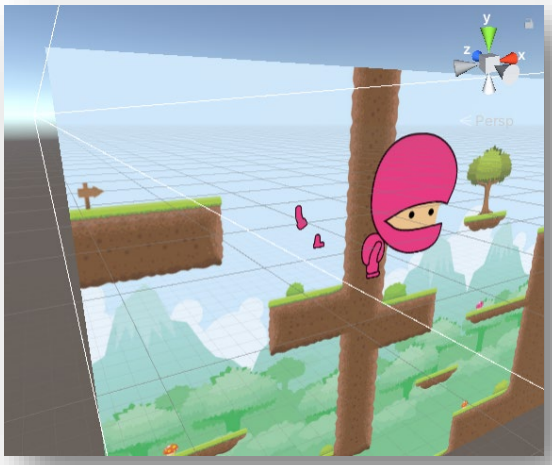
27 Does your Avatar look a little smushed?



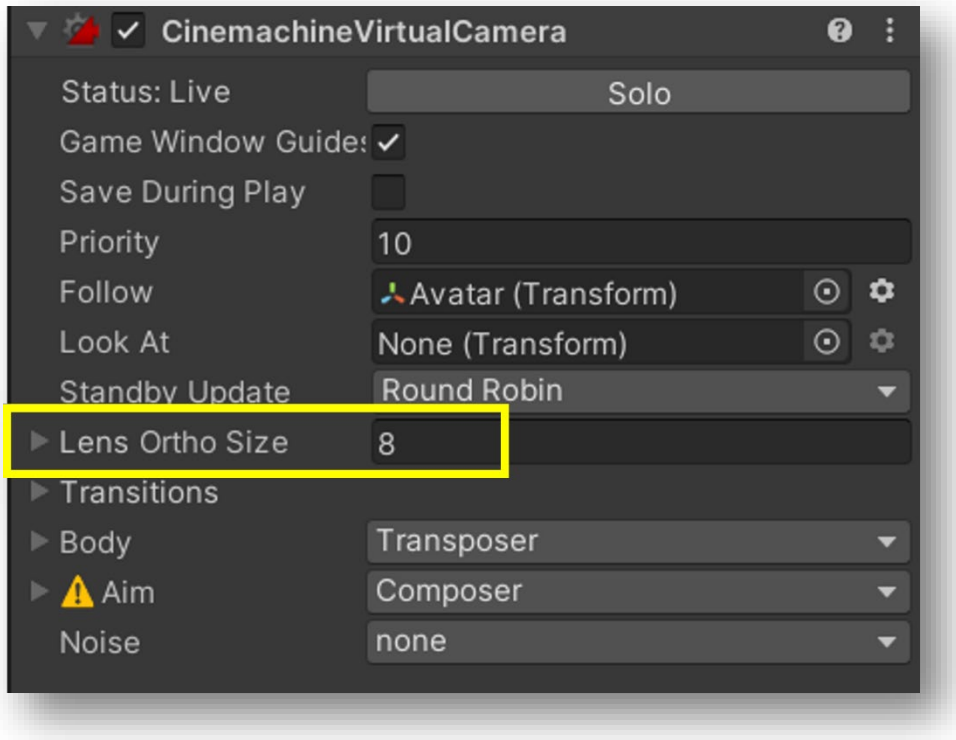
Click on the Main Camera game object and set Projection to Orthographic.



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

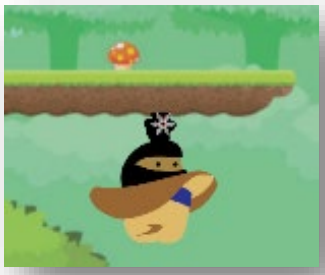


28 Change the zoom of the camera by adjusting the **Virtual Camera** game object's Lens Ortho Size. Try values between 6 and 12 until you find a level of zoom that you like.



What Goes Up, Must Stay Up

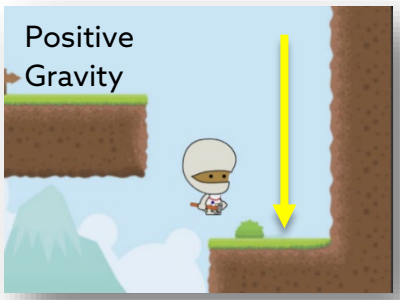
29 Select the Avatar and in the Inspector. We want the player to control the direction of gravity so the Avatar can walk on the ceiling.

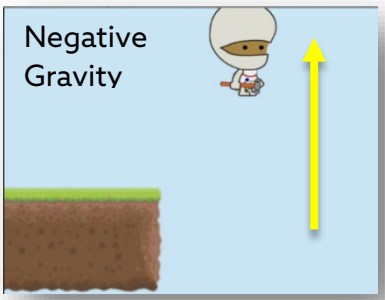


Sensei Stop

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

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





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

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

```
public class GravityFlip : MonoBehaviour
{
    0 references
    void Start()
    {

    }

    0 references
    void Update()
    {

    }
}
```

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

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

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

    0 references
    void Start()
    {

    }

    0 references
    void Update()
    {

    }
}
```

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

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

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

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

    0 references
    void Start()
    {

    }

    0 references
    void Update()
    {
        if (Input.GetButtonDown("Jump"))
        {

        }
    }
}
```

Input Names

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

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

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

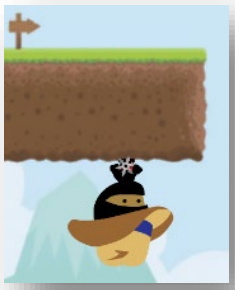
36 Save your **script** and return to Unity. Playtest your game and press the spacebar. Is your Avatar now floating or falling on the ground?

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

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

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

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



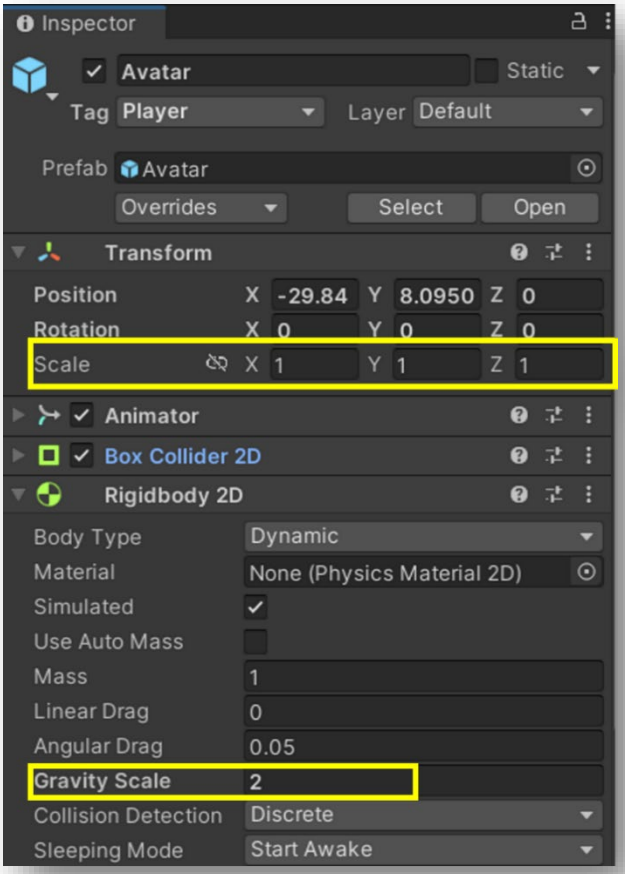
38 Playtest your game and select the Avatar. Begin by making sure your Avatar's **Gravity Scale** is **negative**. Your Avatar should look like it is floating.

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

Sensei Stop

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

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



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

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

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

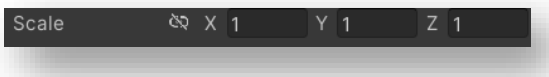
```
void Update()
{
    if (Input.GetButtonDown("Jump"))
    {
        avatarRigidbody.gravityScale *= -1f;
        Vector3 newDirection = transform.localScale;
        newDirection.y *= -1;
    }
}
```

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

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

The Avatar still is not flipping! Why is that?

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

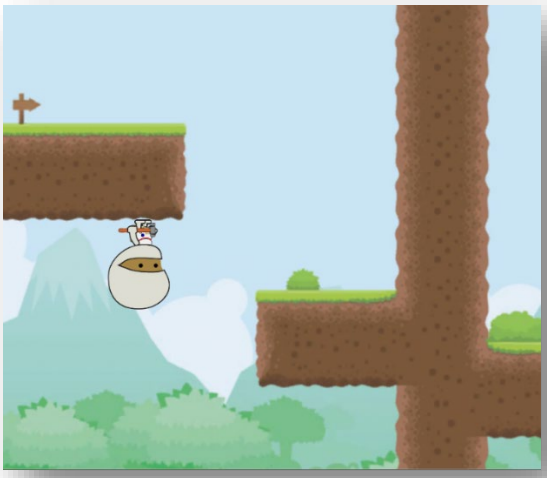


Open the GravityFlip script.

Sensei Stop

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

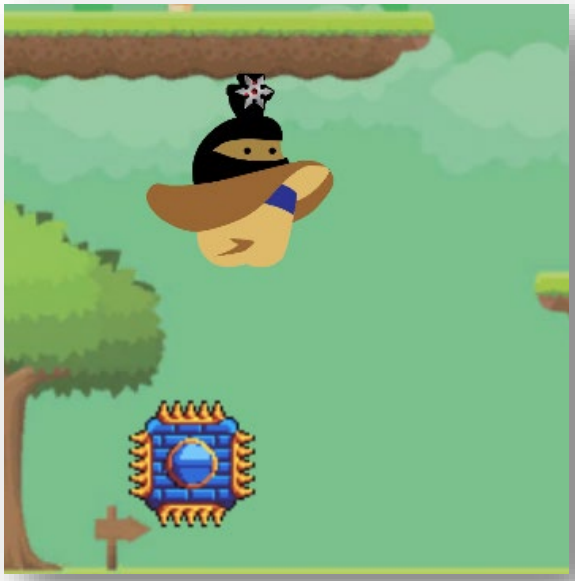
44 After you change your code, save your script and return to Unity. Playtest your game. Your Avatar will now flip based on the direction of gravity.



You Shall Not Pass

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

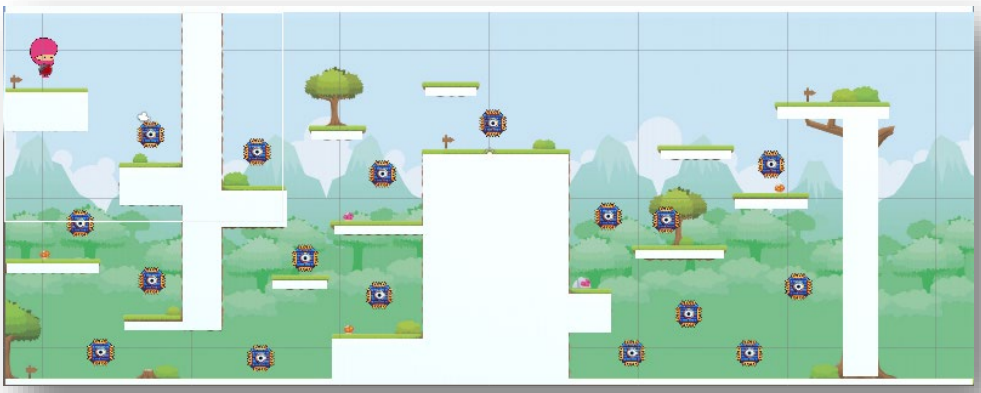
For our enemies, we are going to use the Crushers from Robomania. If you don't want to use the crushers, use the Unity Asset Store to find the enemies you want to use. They can be anything you want, but make sure that the enemies are 2D.



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

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

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



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

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

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

Let's work on our enemy movement first. On your own, program the enemies to move like they do in Robomania. Use steps **14-36** of the Silver Belt Robomania activity.

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

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

Sensei Stop

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

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

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

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

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

```
private void OnCollisionEnter2D(Collision2D collision)
{

}
```

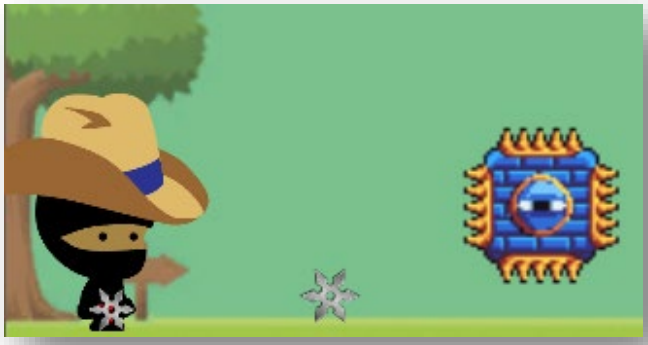
Sensei Stop

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

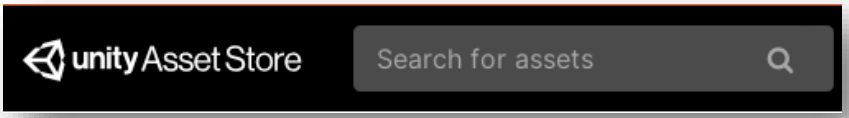
Seeing Stars

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

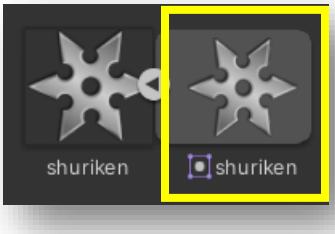
Game design involves making the player feel like they can overcome daunting challenges! Let's place multiple collectables for the Avatar to collect and use to clear enemies!



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



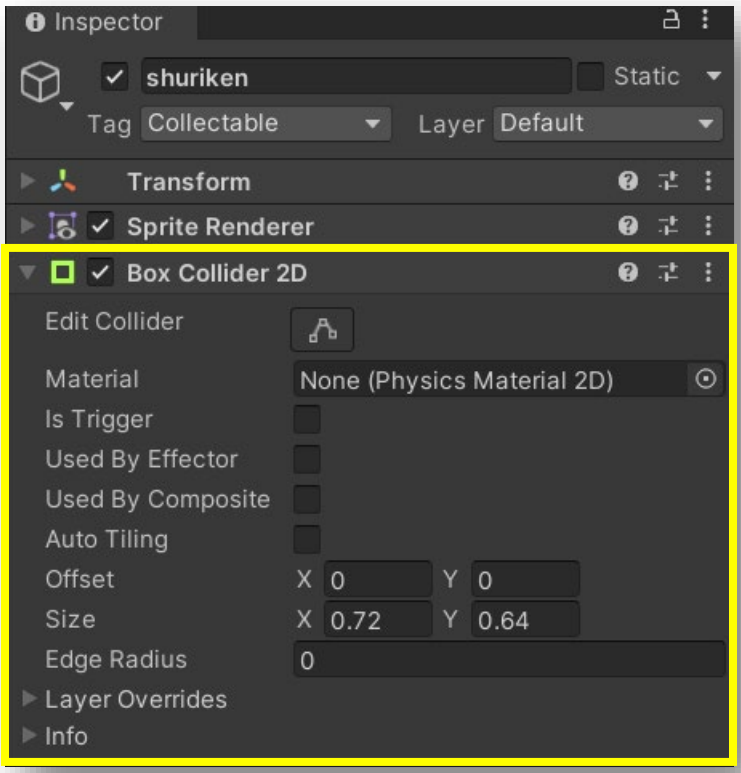
In this example project we are using a shuriken. If you want to use the shuriken, you can find it in the **Artwork** folder.



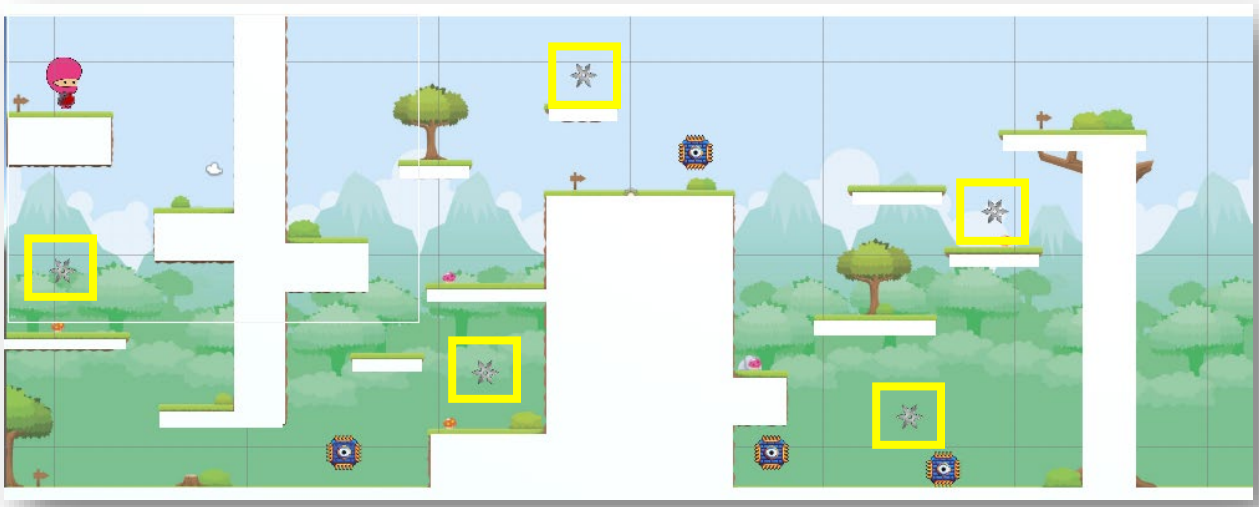
Click to expand the arrow and drag the shuriken on the right onto the scene.

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

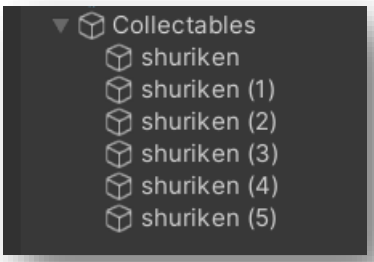
53 Give the object you are using a **Box Collider**.



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



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



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

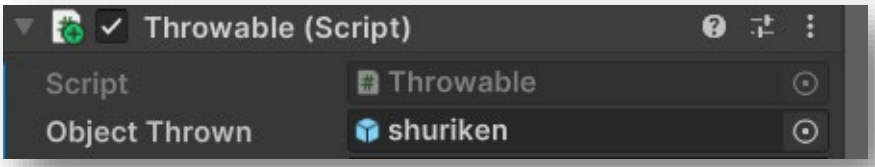
Create a new script in the Scripts folder and name it **Throwable**. Attach the script to the Avatar game object.

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

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

```
public GameObject objectThrown;
```

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

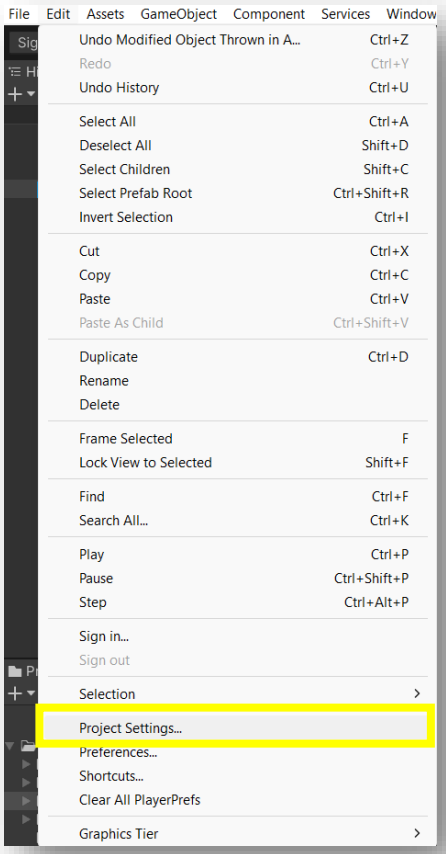


59 Now that the Avatar knows what object it will be using, we need to instantiate it where the Avatar is positioned.

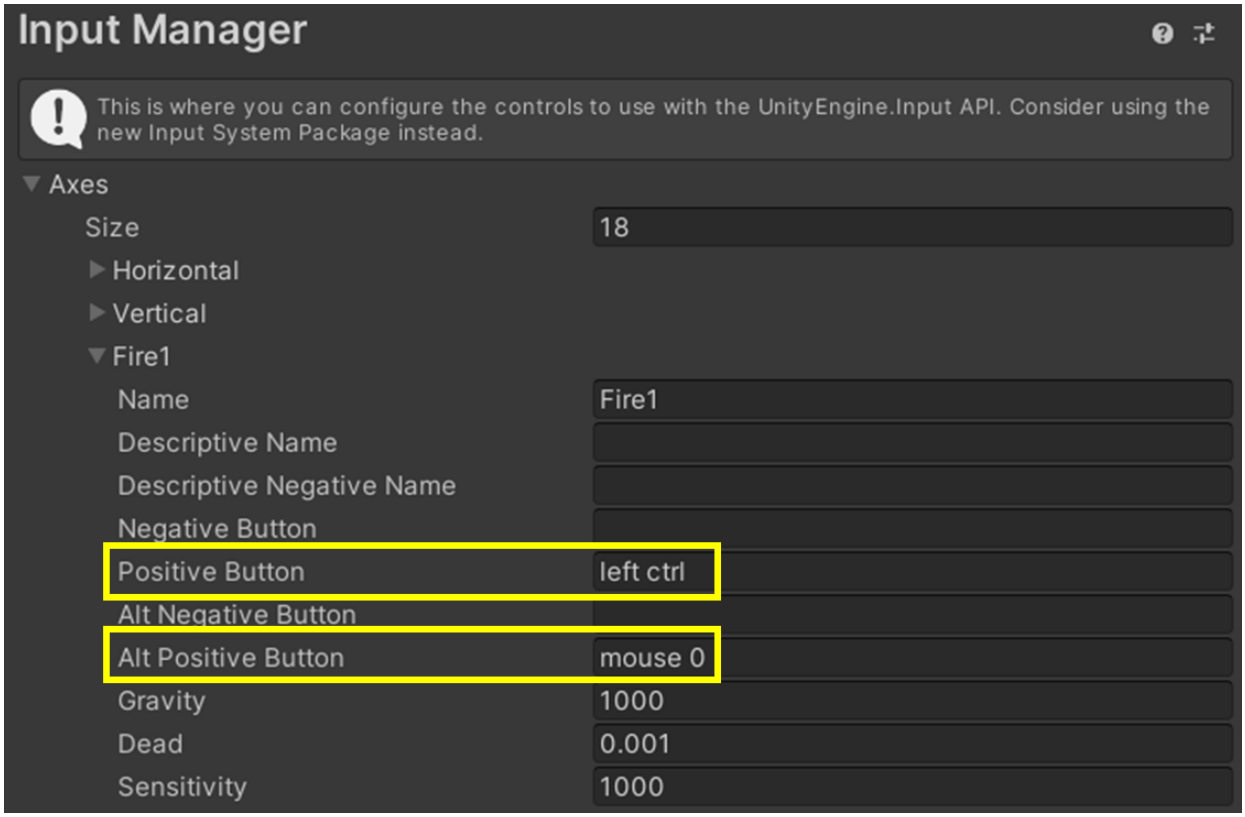
Open the **Throwable** script. In the **Update** function we want to place the object in front of our Avatar.

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

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



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



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

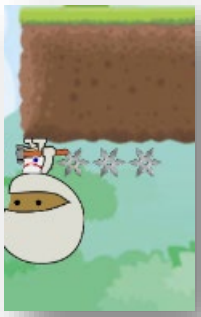
62 If you don't want to use the left control key as a way to throw your object, you can delete "left ctrl" and replace it with the key you want. To see the exact key value you need to type, click on the question mark button at the top right of the Project Settings.

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

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

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

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



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

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

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

```
public Vector3 offset;
```

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

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

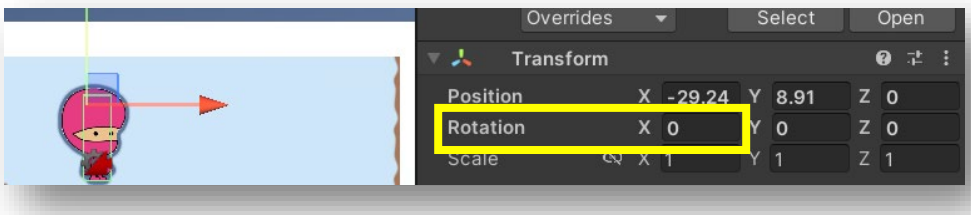
The **X value** moves the object from left or right.
The **Y value** moves the object up or down.
The **Z value** moves the object forward or back a layer.

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

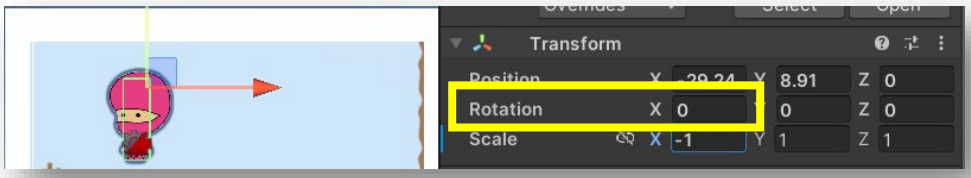
```
offset = transform.localScale.x * new Vector3(1, 0, 0);
```

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

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



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



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

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

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

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

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

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

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

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

```
public int throwableCounter;
```

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

Create an **OnCollisionEnter2D** function that:

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

Modify the "Fire1" conditional statement so it:

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

Sensei Stop

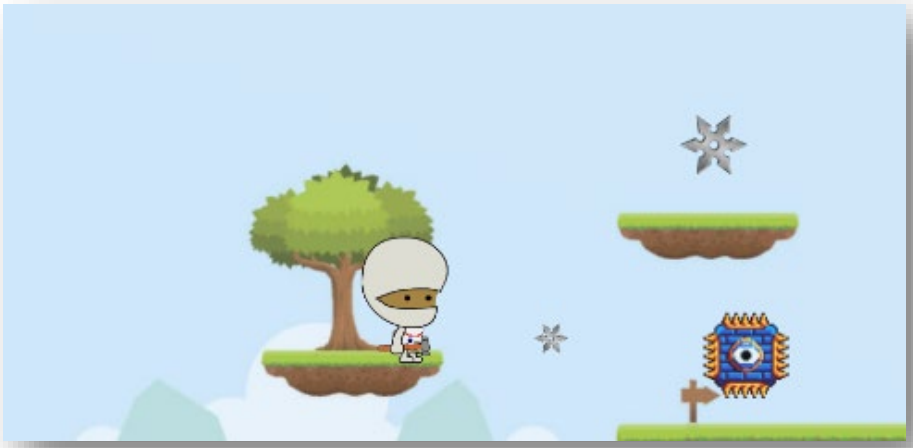
Work with your Code Sensei to answer the following questions based on the above pseudocode. How does the Avatar pick up an item? How many items can the Avatar throw? If the Avatar has not collected an item, can it throw an object?

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

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

Throwdown

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



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

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

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

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

```
public Throwable direction;
```

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

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

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

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

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

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

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

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

79 Save your script and return to Unity. Playtest your game again. The throwable is really slow now! How can we adjust the speed?

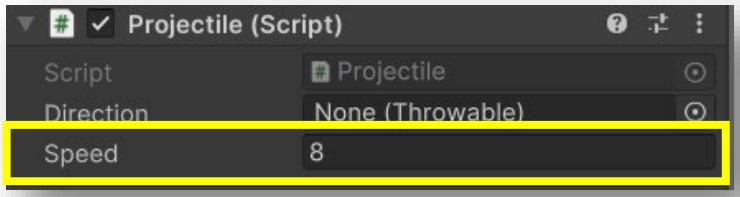
Stop the game and open the **Projectile** script. One way we can speed up the throwable is by creating a **public float** variable named **speed**.

```
public float speed;
```

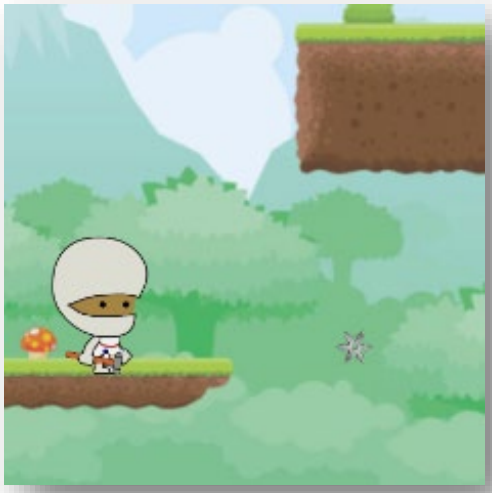
80 Multiply **Time.deltaTime** by **speed**.

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

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

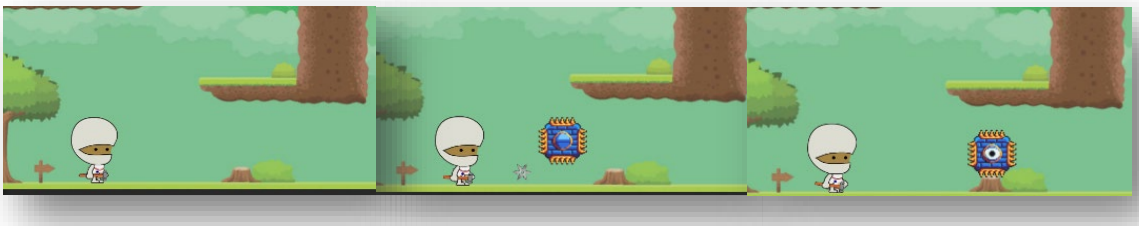


82 Playtest your game. The throwable should now be moving at a good speed for the player – not too fast or slow! It will also move in the direction the Avatar is facing!

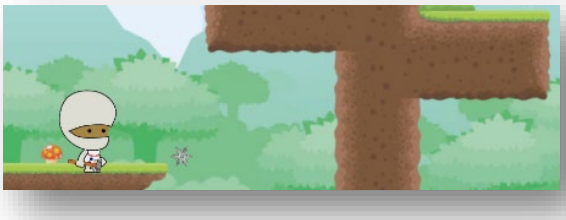


83 Notice that the object you instantiate never gets removed from the scene.

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



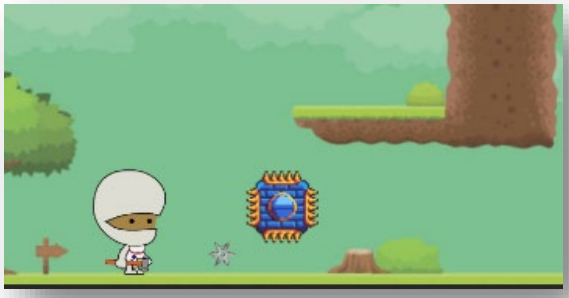
We also need to destroy the object the Avatar throws even if it does not collide with an enemy.



In order to destroy the enemies, you must:

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

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



Sensei Stop

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

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

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

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

```
private void DestroyThrowable()
{

}
```

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

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

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

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

Sensei Stop

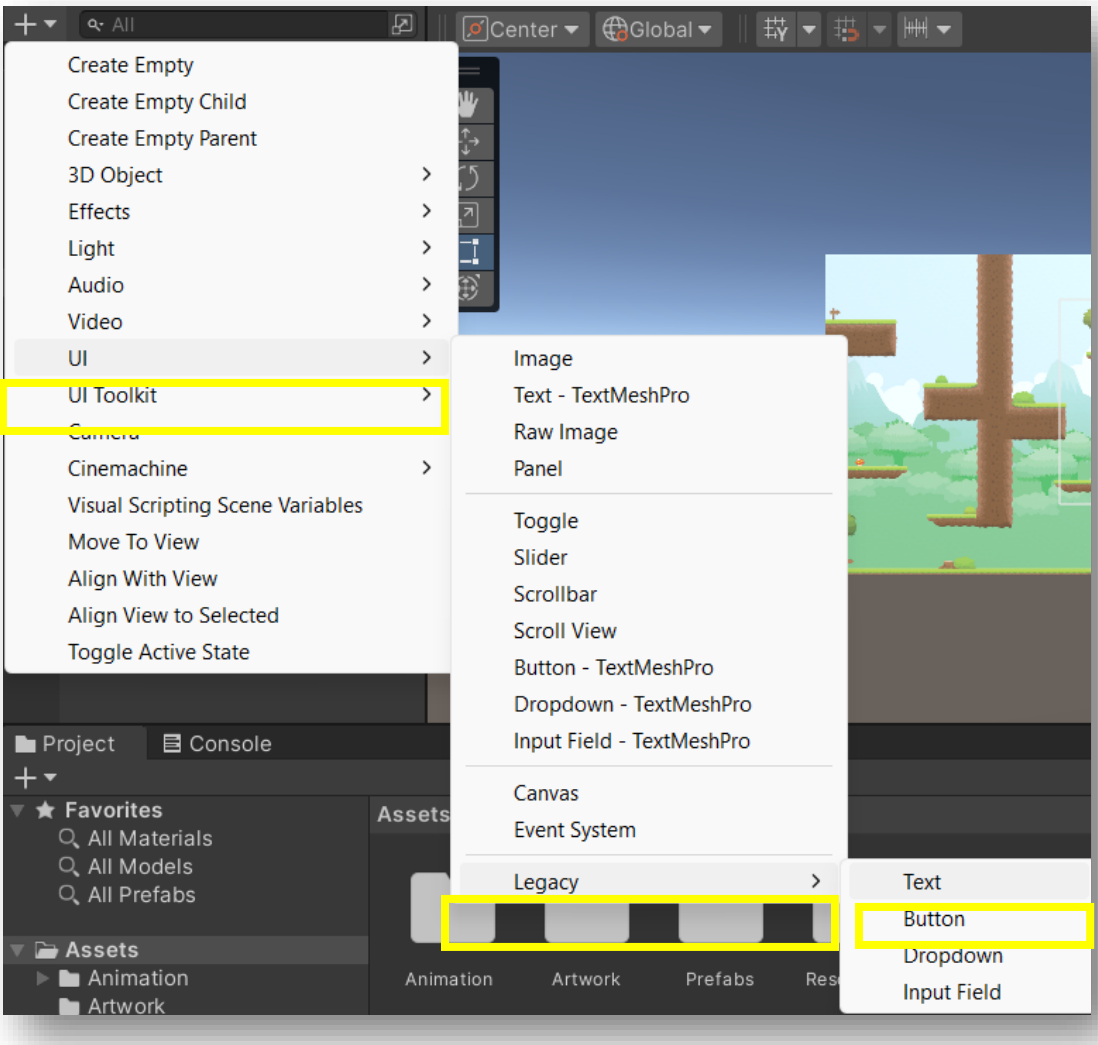
Demonstrate to your Code Sensei that your throwable is destroyed after a few seconds. If you are not able to see the throwable vanishing, use the scene view and zoom out so you can see it disappear.

Show Me the Shurikens

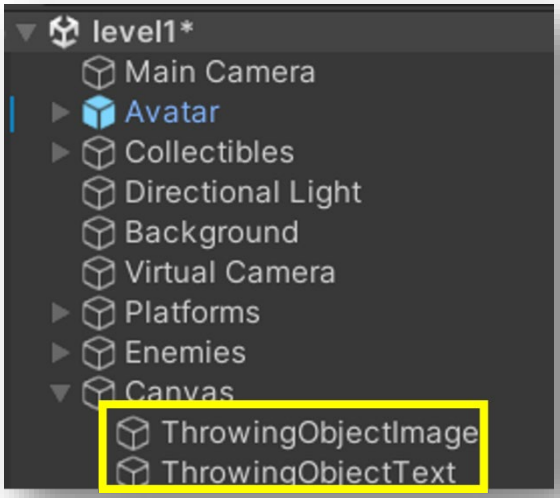
88 Our Avatar is now able to collect items and throw them at enemies to get past them! However, the person playing our game won't be able to click the Inspector to see how many throwable items they have collected.

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

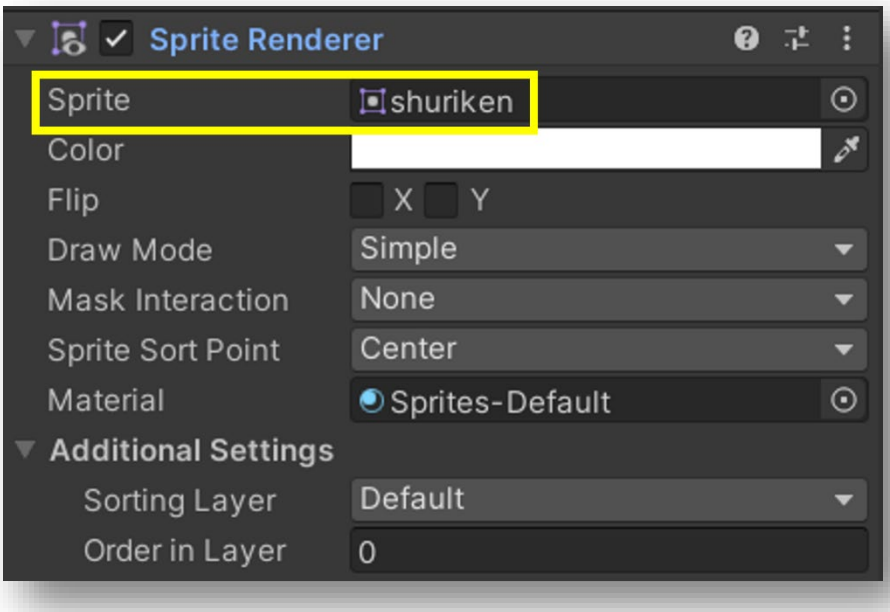
89 Create a **UI Legacy Text** and **UI Image** in the Hierarchy.



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

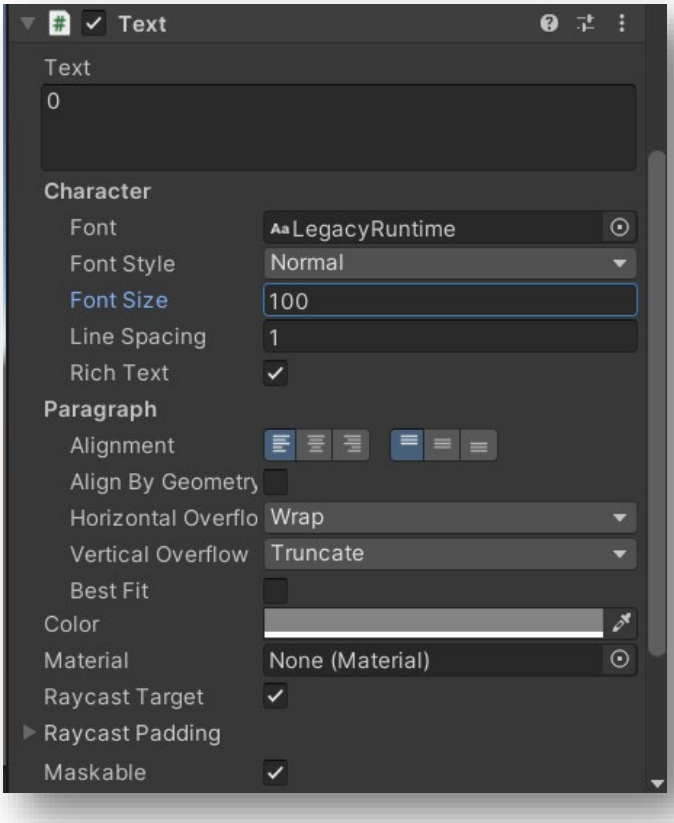


91 Select the **ThrowingObjectImage**. In the Inspector find the **Image** component, we are going to change the **Source Image** to the throwable you are using. Click on the circle and choose your throwable.



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

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



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

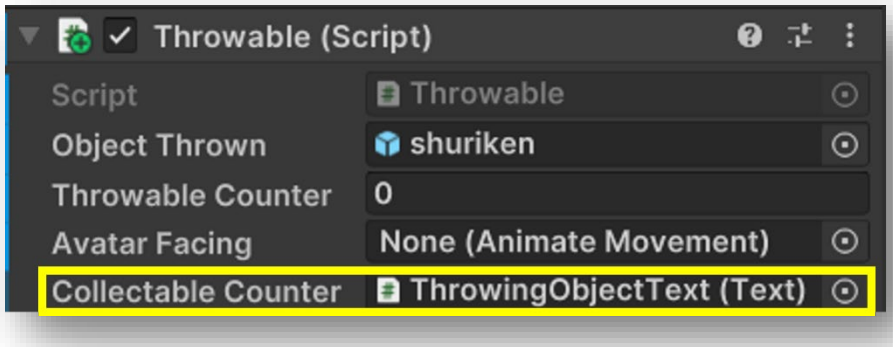
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
```

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

```
public Text collectableCounter;
```

Save your script and return to Unity.

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



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

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

If our Avatar collides with a collectable:

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

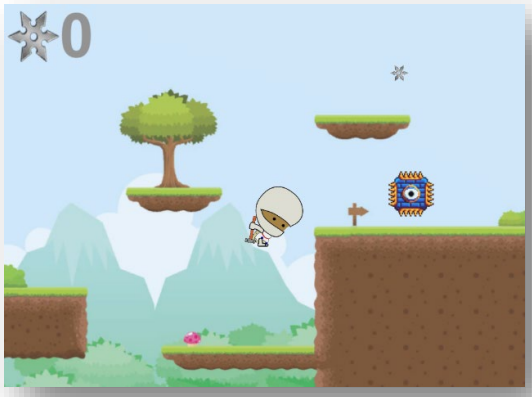
When the player throws a collectable:

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

Sensei Stop

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

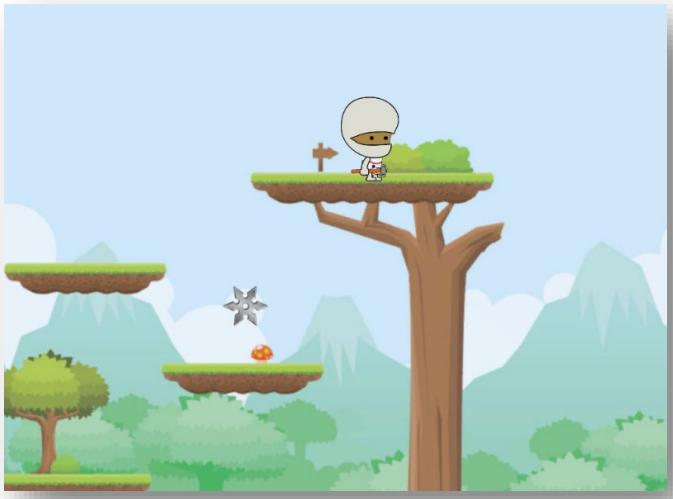
Your game should update every time you collect or throw an object.



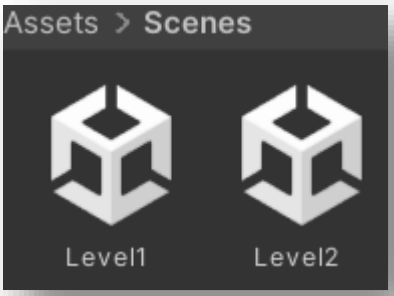
Take it up a Level

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

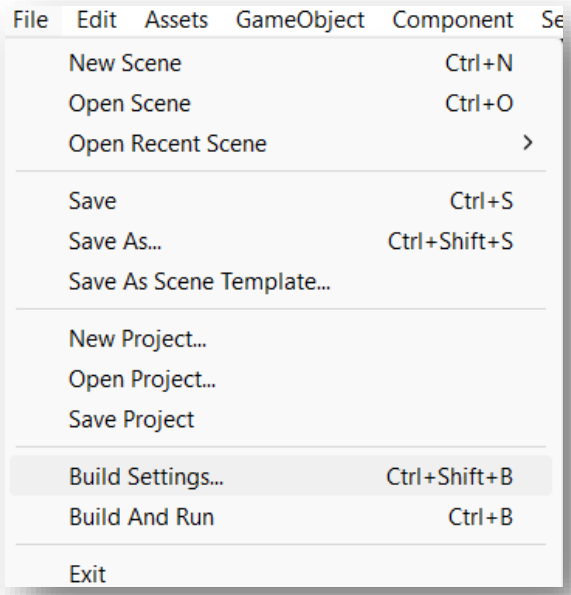
For the forest background, the player needs to reach the top right of the scene. If you used a different background image, your goal area may be different.

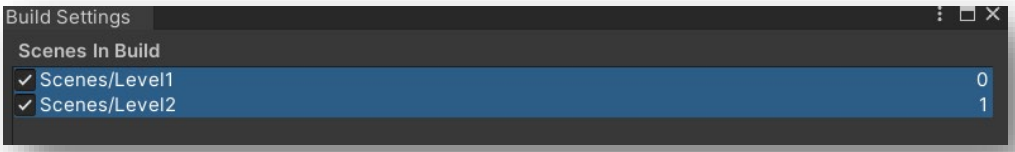


99 In the Scenes folder, duplicate your **Level1** scene and rename it **Level2**.



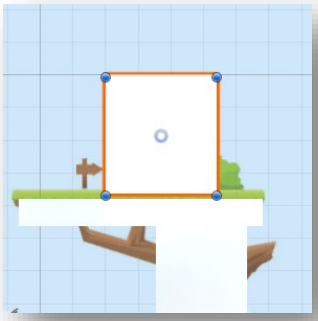
100 Add this new scene to the **Scenes in Build** list located in **Build Settings**.



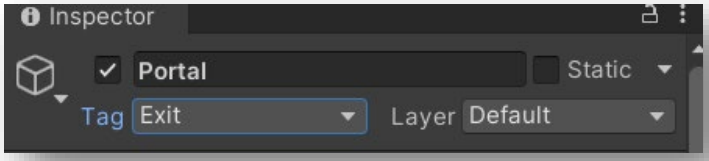


101 Open the Level 1 Scene. Create a Quad object and rename it portal. This object will be used to teleport our Avatar to level 2 only if all the enemies have been destroyed. You can replace the Quad object with any other game object you like. In our example project we found a portal and are going to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

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

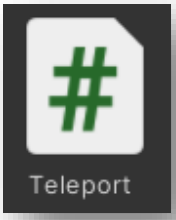


102 Create a new tag named **Exit** and assign it to the Portal game object.



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

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



105 Open the Teleport script in Visual Studio.

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

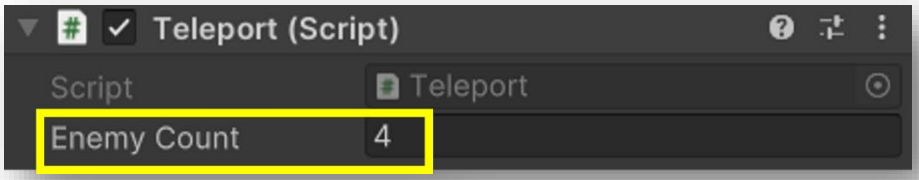
```
public int enemyCount;
```

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

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

107 Save your script and return to Unity.

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



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

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

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

Sensei Stop

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

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

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

```
private void OnCollisionEnter2D(Collision2D collision)
{
    if(collision.gameObject.tag == "Player" && enemyCount == 0)
    {

    }
}
```

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

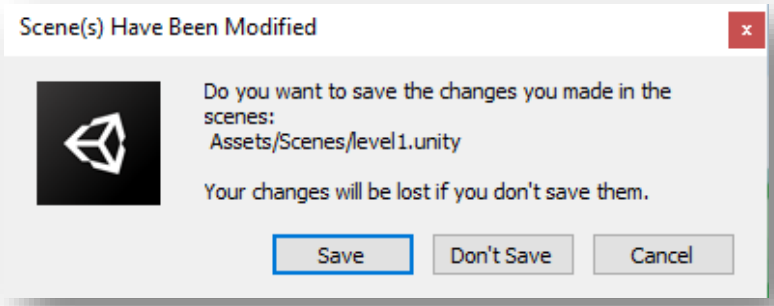
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
```

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


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

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

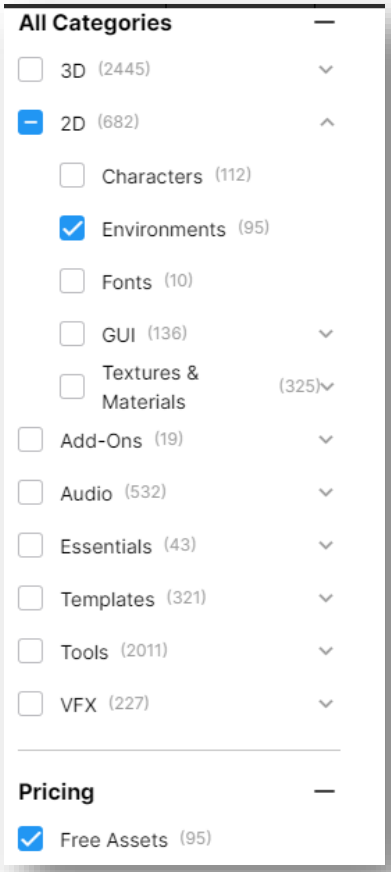
Stop your game. In the Scenes folder, open up level2. If Unity asks you to save your changes, select **Save**. If you don't, your work will be deleted.



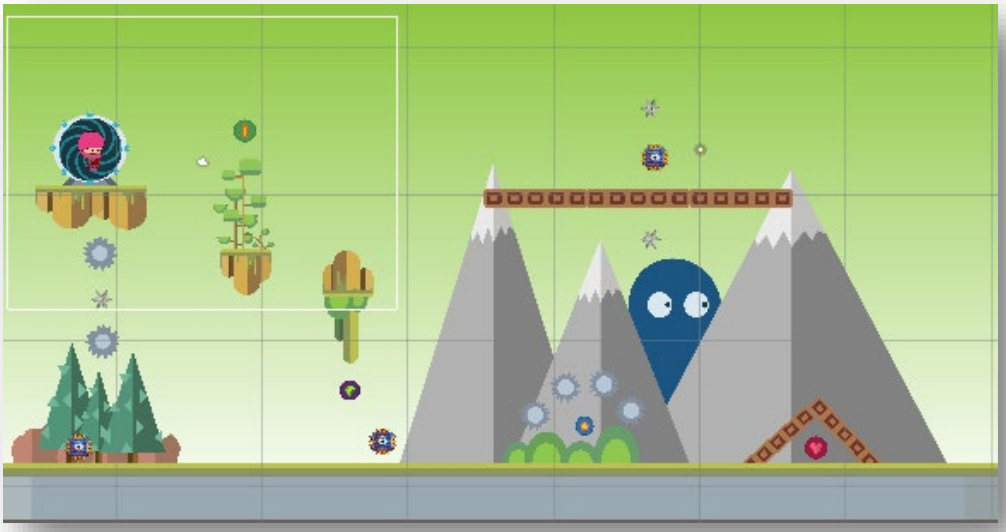
113 You might be wondering what we are going to do in level 2. It is now your turn to create your very own 2D scene using the Unity Asset store!

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

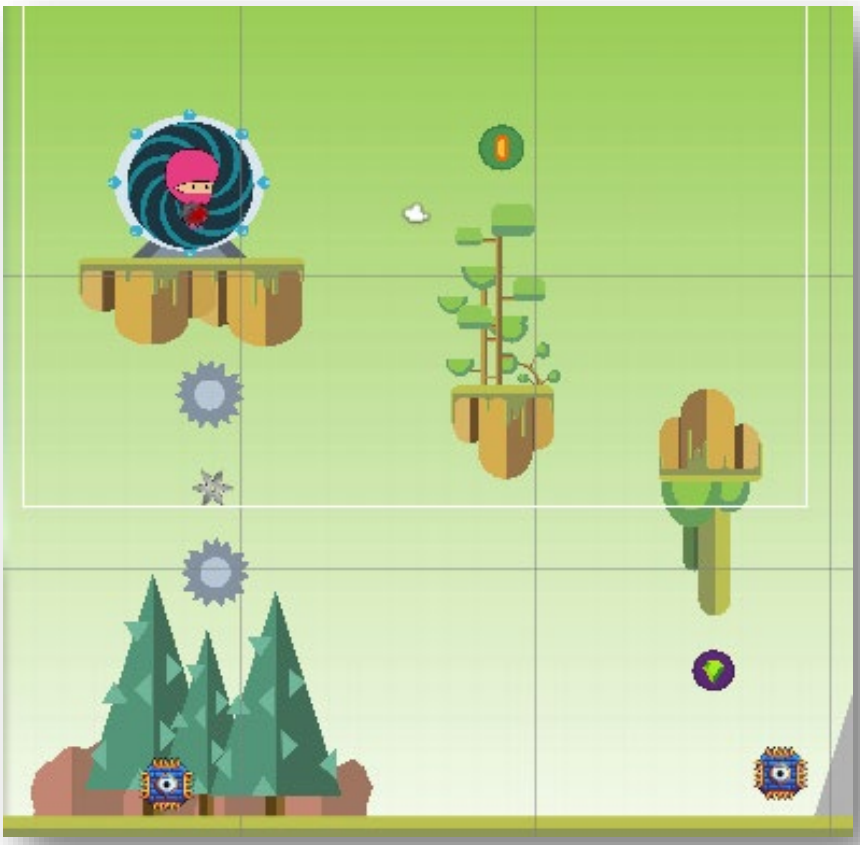
114 Go into the Unity Asset Store tab. In the Categories, apply the filters we have below so you can find Artwork you like.

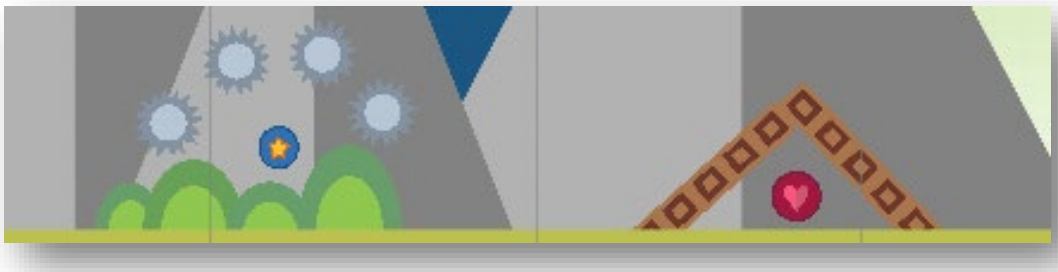


115 Go through the different artwork and import it into your project. Look at what we have made below using different downloaded art!

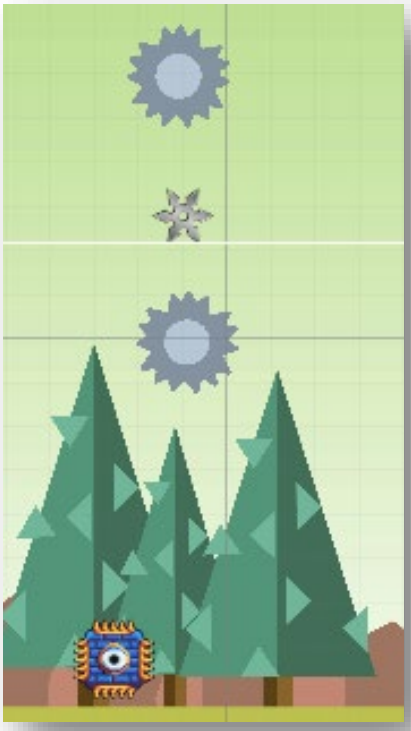


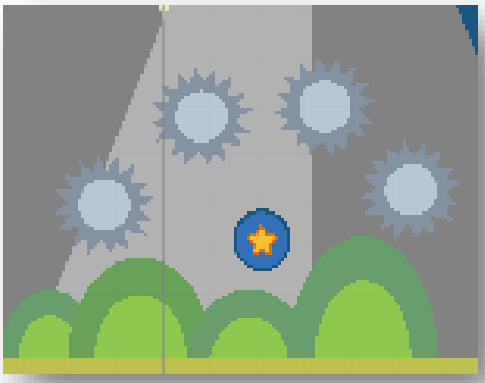
In this level, you have the option to add more enemies or more collectables to make your level more challenging and fun. We have added additional collectables all around the scene that the Avatar must collect to pass level2.





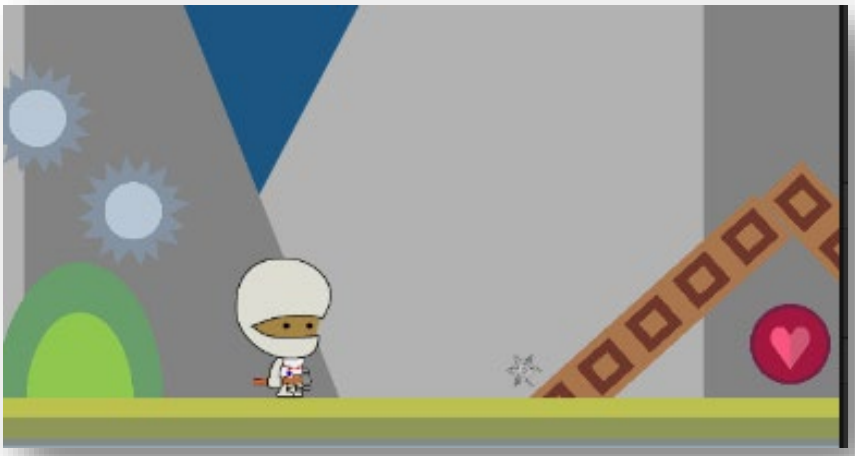
117 The crushers are still our enemies, but now there are additional traps! There are rotating spikes that resets Avatar if it gets too close to them.



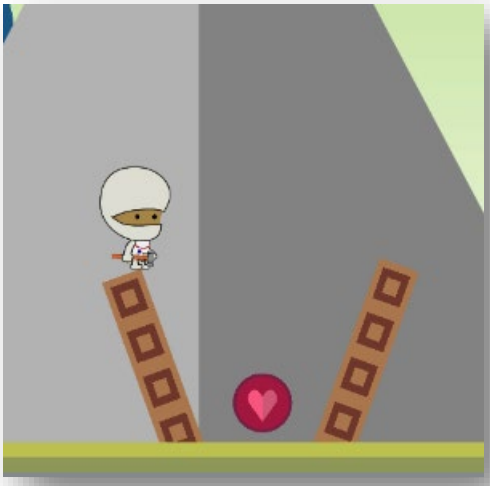


You can even block off a collectable and program a way for the player to unlock it!

Can the Avatar throw an object and break the barriers?



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



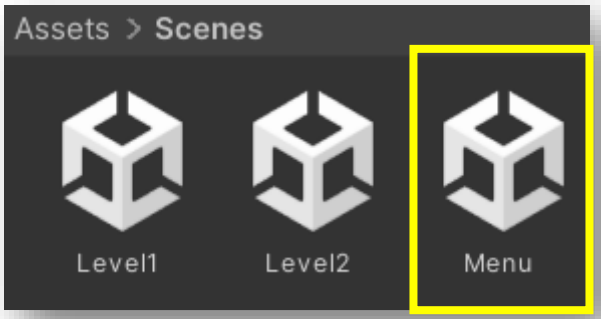
119 We have also modified level1 in order to align better with level2. In game design, it is always ok to go back and make changes to your game so things can match up together.

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

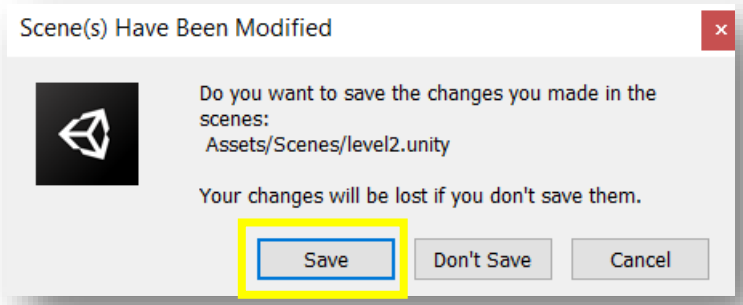
120 Refer to your Gravity Trails Planning Document to build out the scene you planned!

Creating a Start Screen

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



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

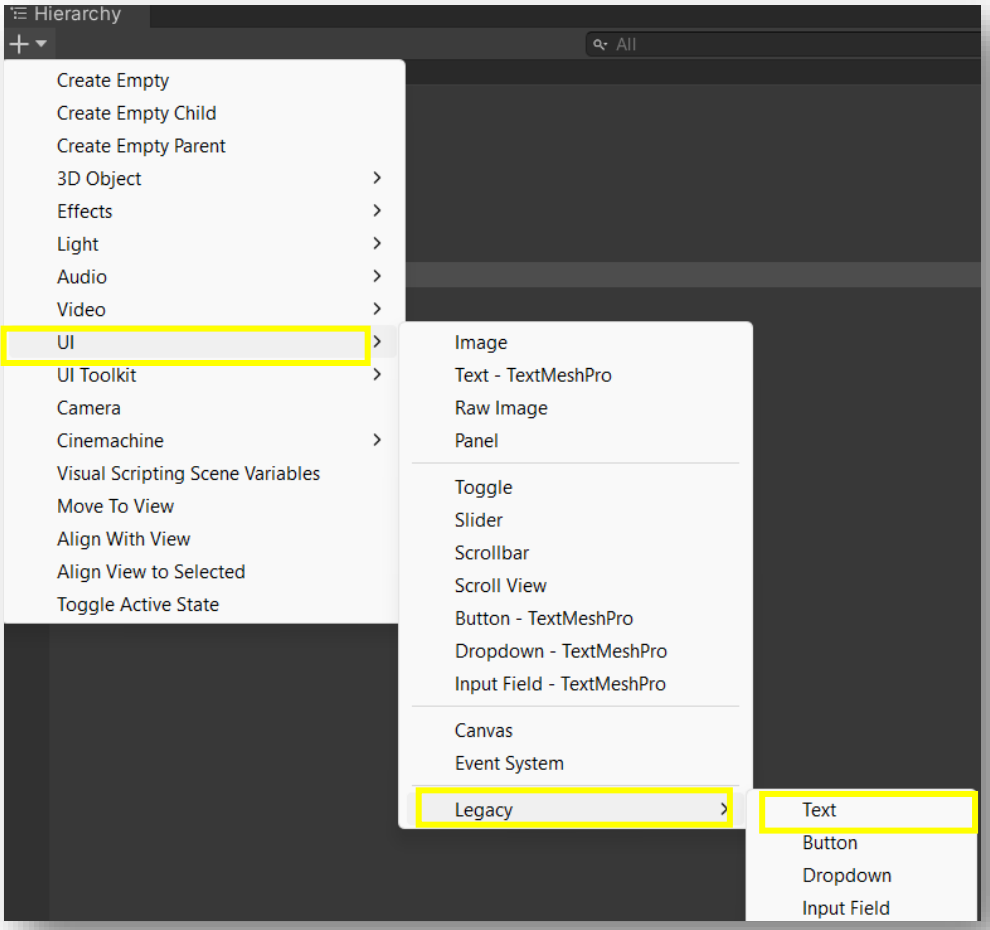


123 Our game menu will display four things:

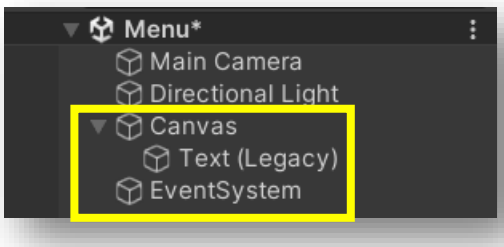
- the title of the game,
- a background,
- a start game button, and
- the controls the player needs to know to play your game.

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

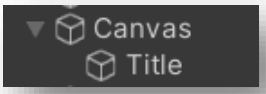
124 In the Hierarchy, create a new **UI Legacy Text** object. This object will display the title of our game.



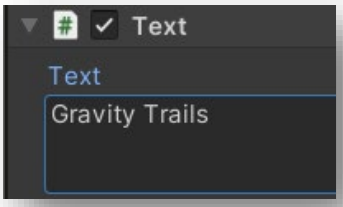
This will create a **Canvas** with your **Text** inside of it and an **EventSystem** object.



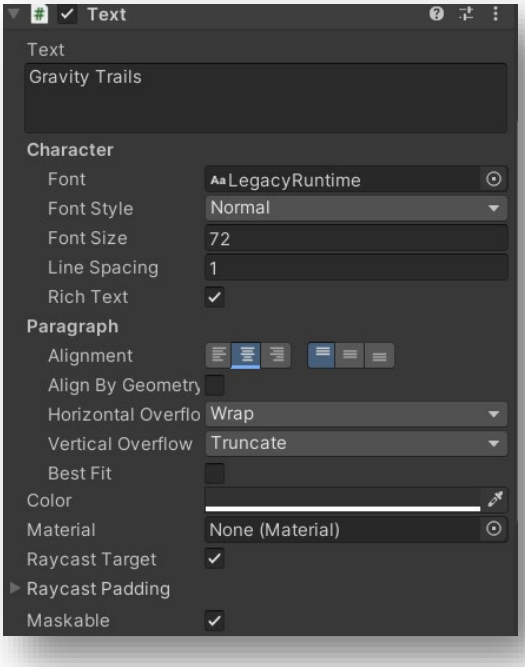
125 Rename the Text object to **Title**.



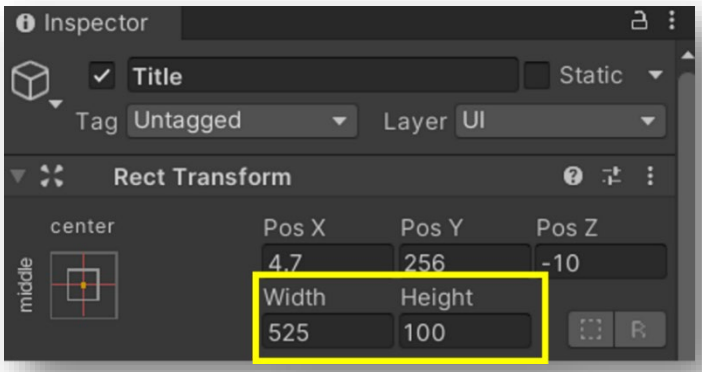
126 Click on the Tile object and in the **Inspector**. Change the value of the **Text** property to the title of the game.



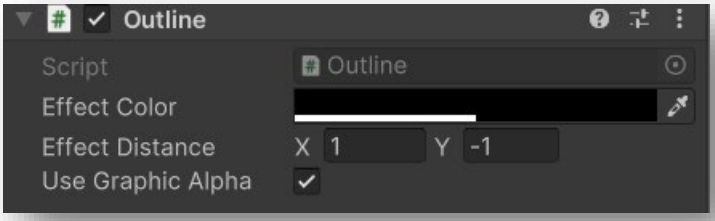
127 Change the other properties of the Text component to customize your title.



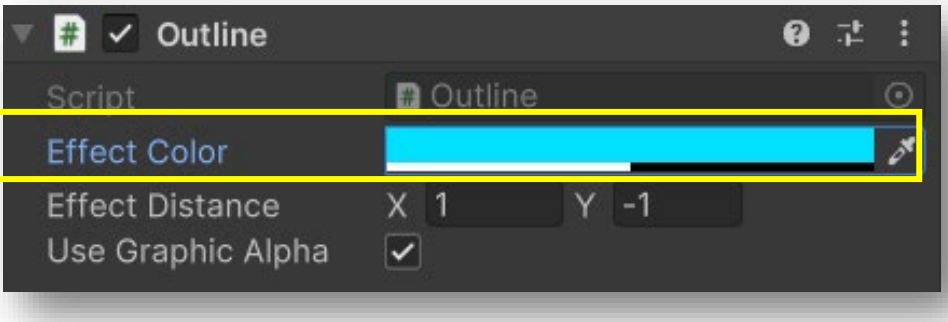
128 If you are unable to see the changes you are making, you might have to adjust the **Width** and **Height of** your text box.



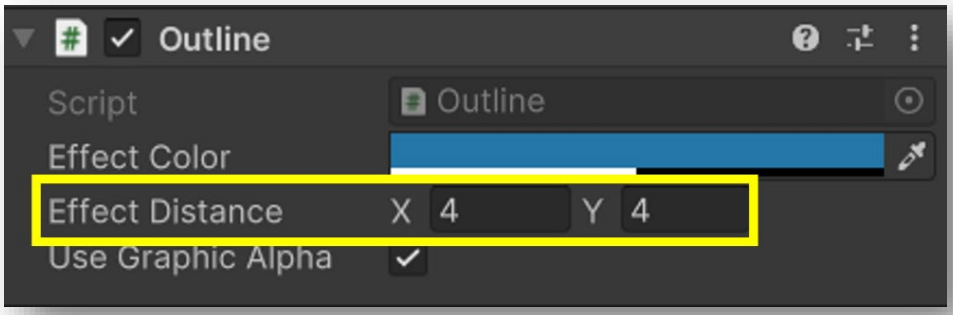
129 Another way we can customize the Title is by adding the **Outline** component. Click on **Add Component** in the Inspector, search for **Outline**, and click on it.



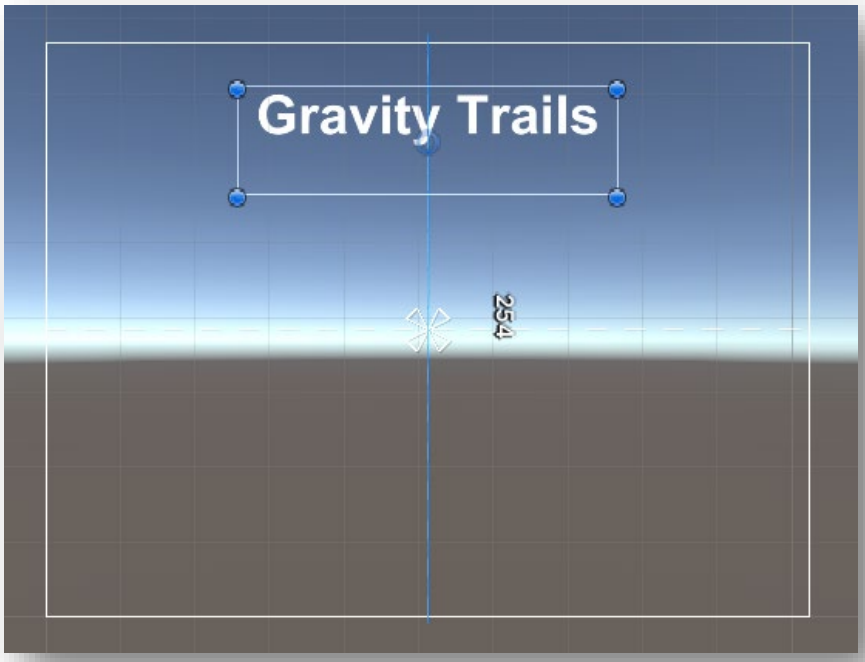
130 Change the **Effect Color** to add a border to the text.



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

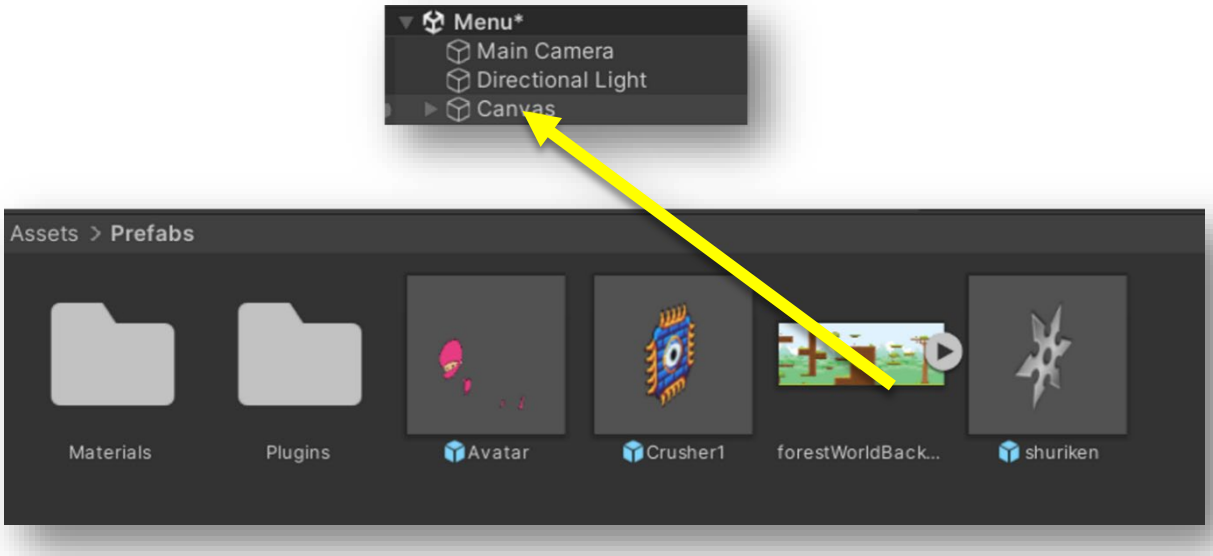


132 Use the **Rect Tool** to center the Title on the Canvas. You will know it is centered when a Vertical Blue line appears in the screen. You may need to resize the textbox to a larger size to see the blue line.

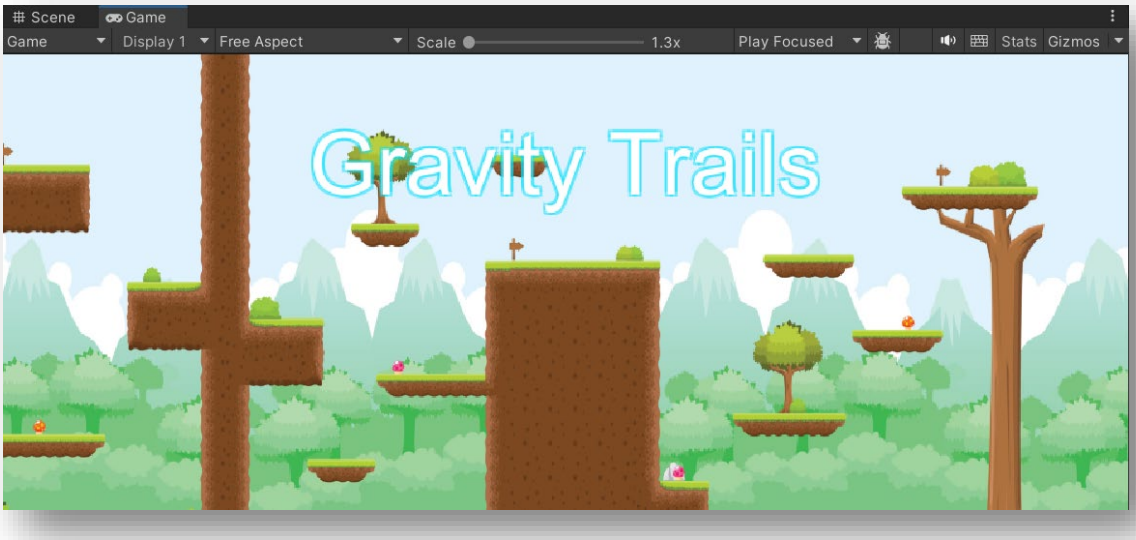


133 The Start Screen is not quite complete! Let's work on the background. You can use the **forestWorldBackground** that we used previously or find a new background in the Unity Asset store.

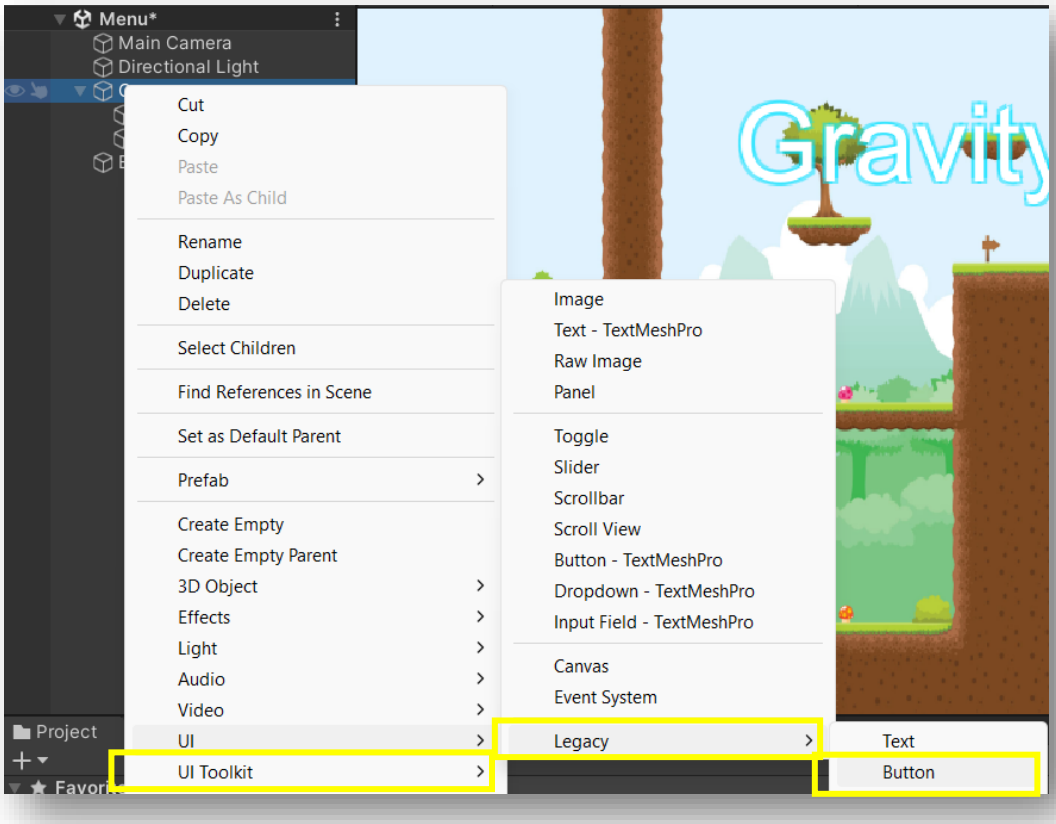
Once you have the background you want, we are going to add it to the scene. Drag the background you chose into the Canvas in the hierarchy. If you don't see the background in the game view, make sure that the background image is positioned where your camera is.



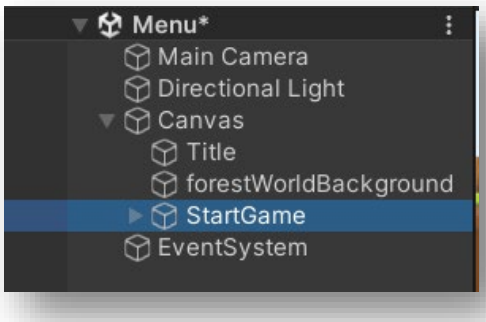
134 Your background image should now be visible on the game screen. Use the **Move Tool** to adjust the position of your image.



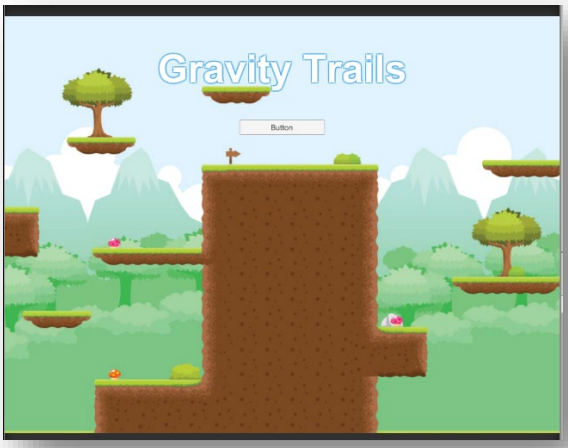
135 Now that we have our background setup, let's move on to creating our **Start Game** button. In the Canvas object, right click and go to **UI** then **Legacy** then **Button**.



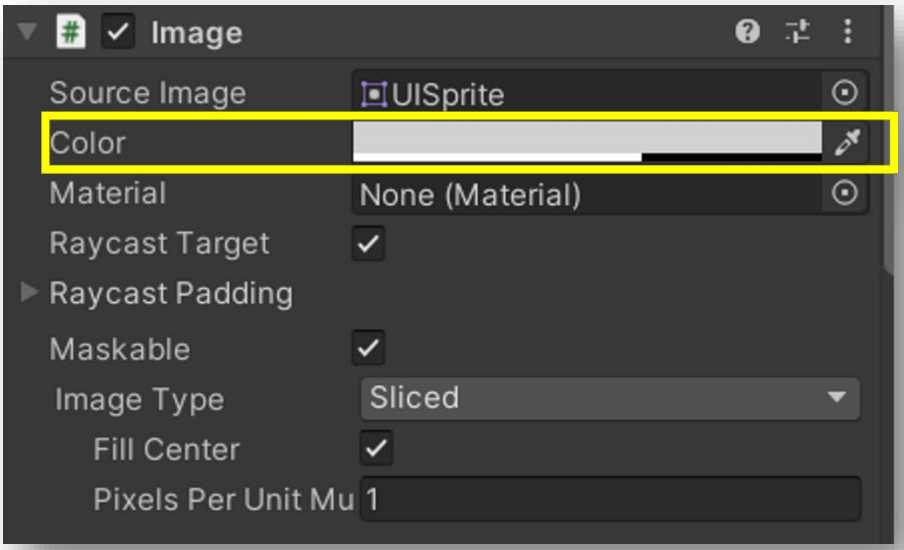
136 Rename the button to **StartGame**.



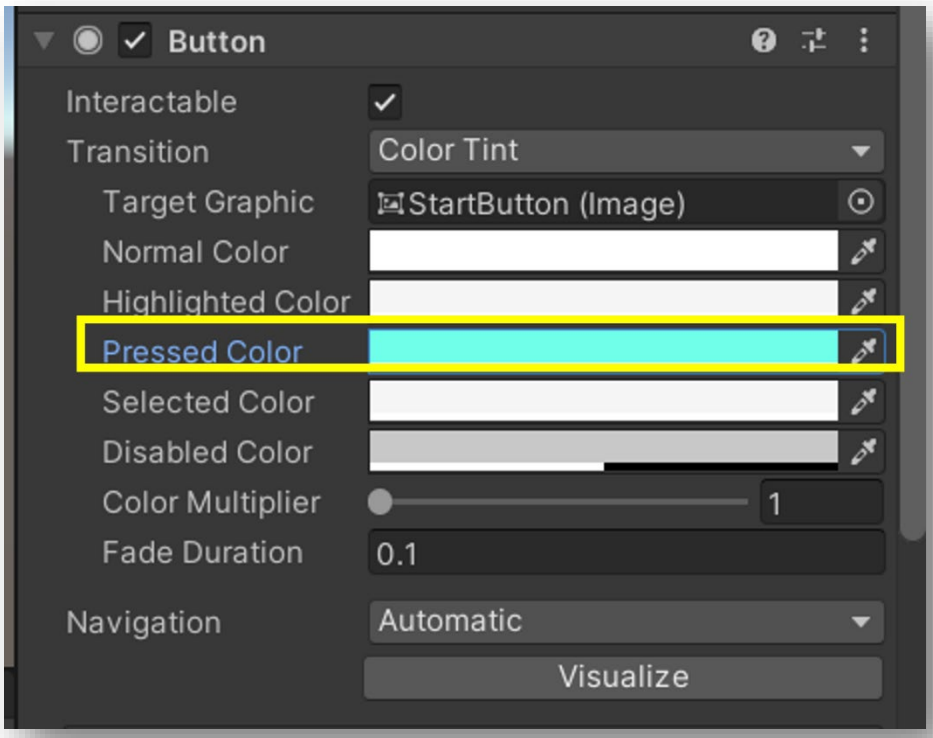
137 You should see a button created in the middle of the scene. Feel free to position it anywhere that is easy for the player to see it.



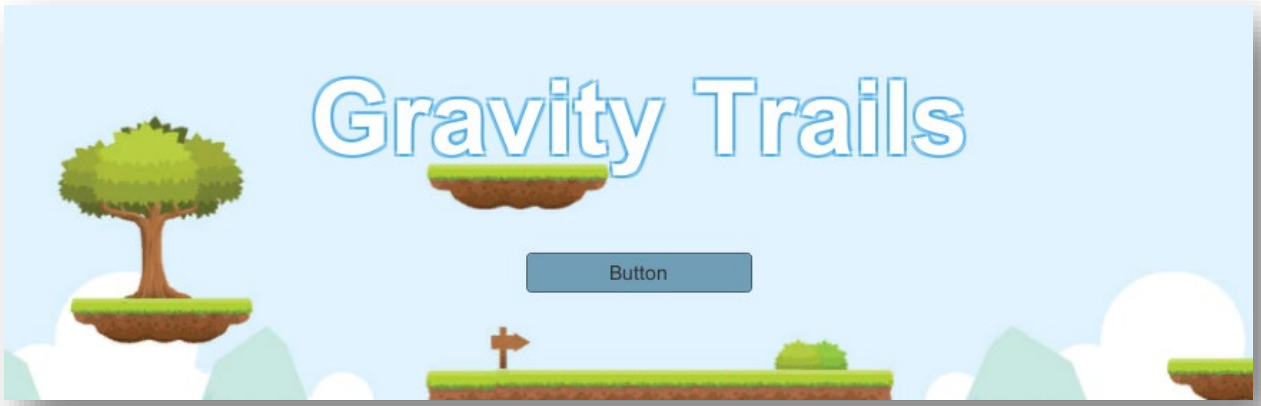
138 We can customize the buttons appearance by making changes to the **Image** and **Button** component in the Inspector. Try changing the **Color** of your **button!**



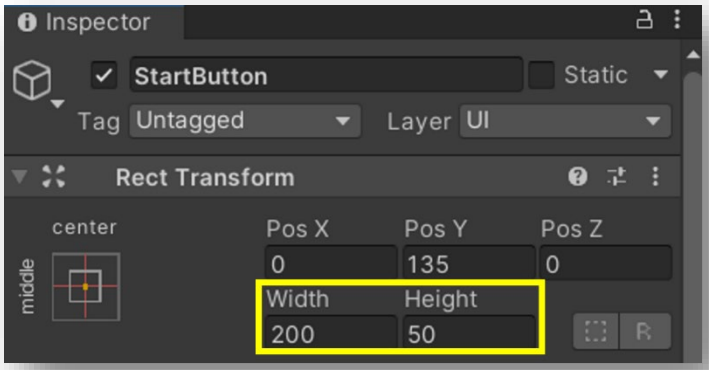
139 In the **Button** component, you can change the **Pressed Color** so when a player wants to start the game, they are given feedback that the button has been pressed.



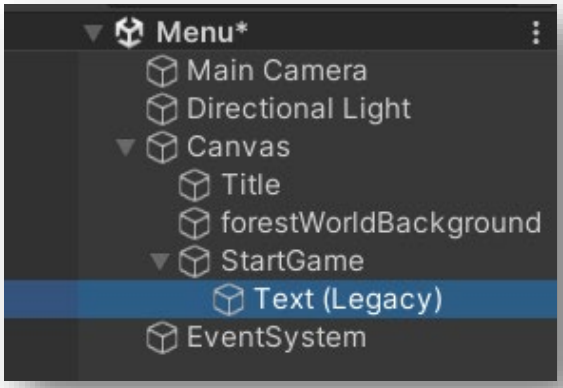
140 Playtest the scene and click on the Button. Notice how it changes colors when clicked.



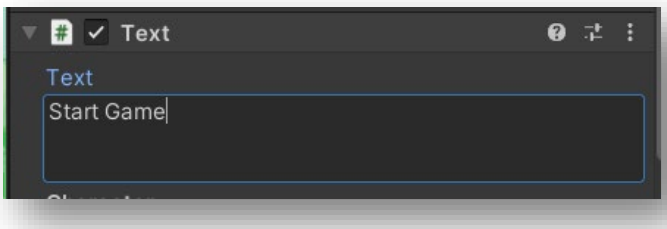
141 Stop your game. If you feel your button is to small adjust the **Width** and **Height** to make it larger.



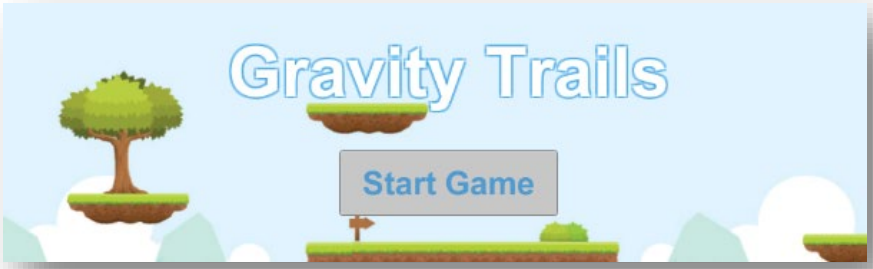
142 The last part of our button that we want to customize is the **Text**. Expand the StartGame object and select the **Text**.

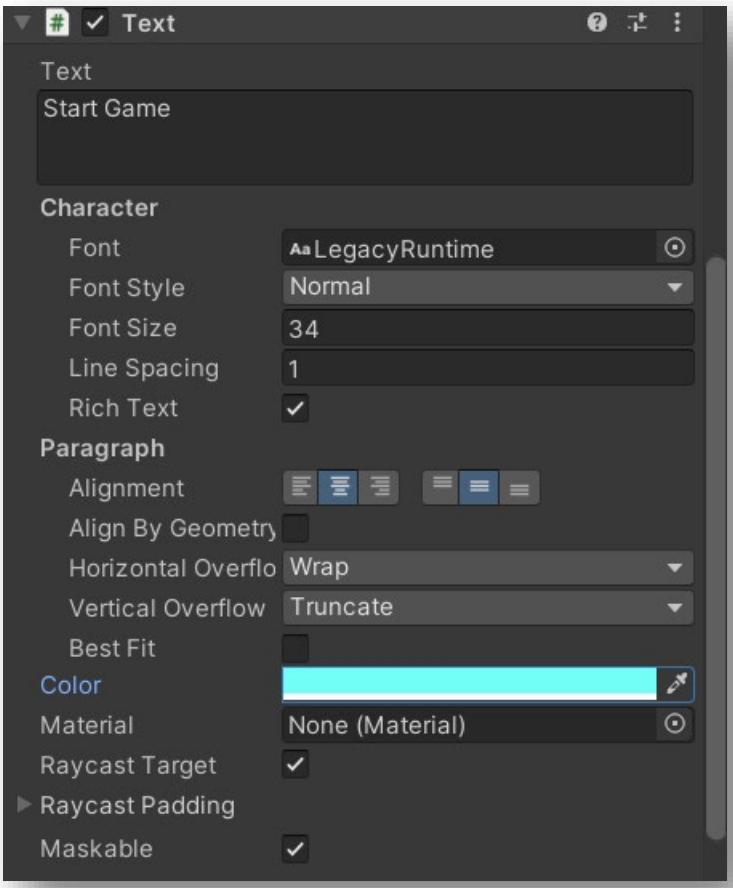


143 In the Inspector change the **Text** value to **Start Game**.



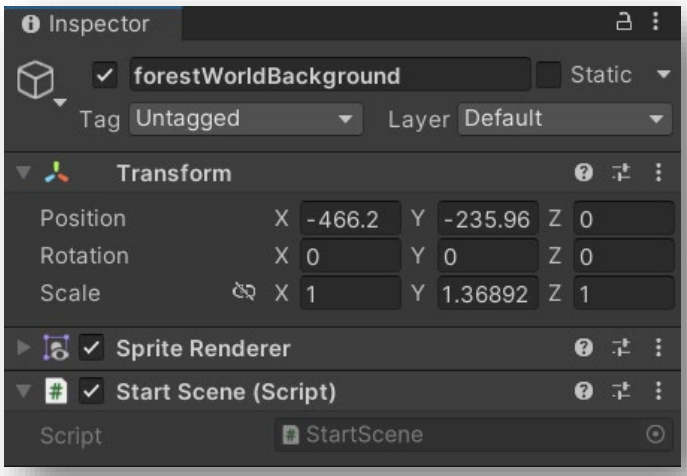
144 Customize the text to your liking by using the other options in the **Text components**.





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

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



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

```
public void OnMouseDown()
{

}
```

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

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

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

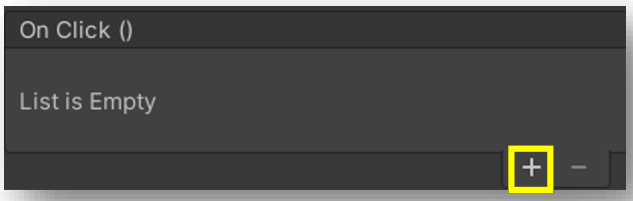
```
public void OnMouseDown()
{
    SceneManager.LoadScene(1);
}
```

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

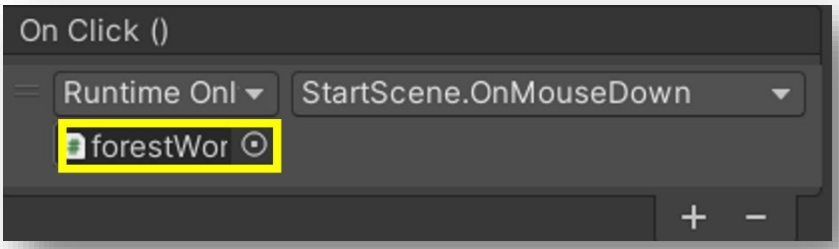
150 Save your script and return to Unity. Playtest your game and try clicking on the Start game. The button still doesn't change the scene!

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

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

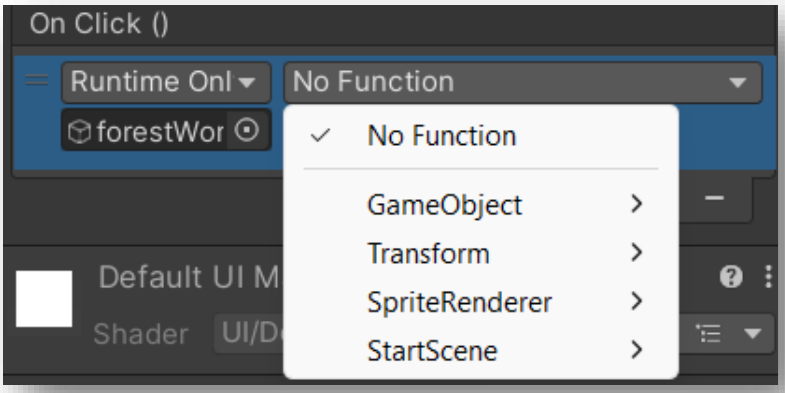


151 You should now see three slots, we first need to fill the one that says “None” with the object that has the StartScene script. Drag in the background you are using into that component.

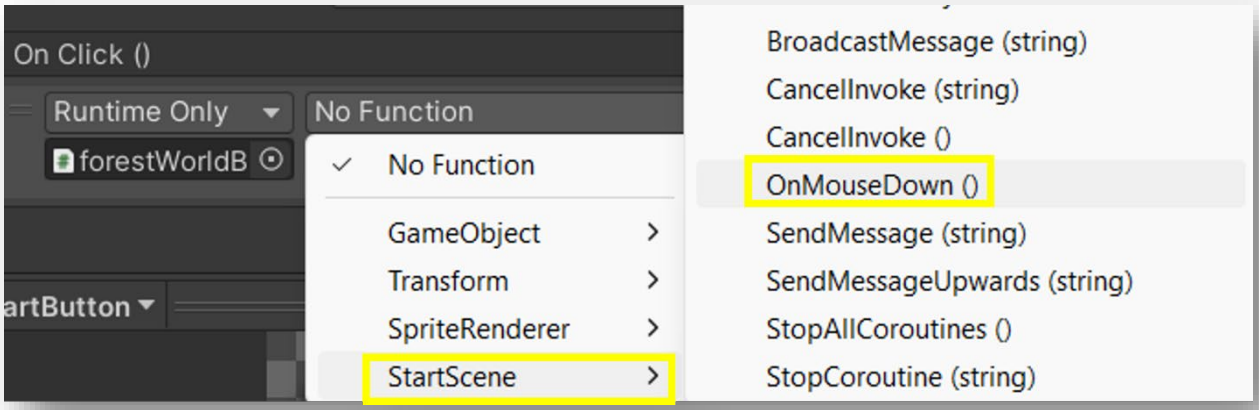


Notice how we can now change the property on the right that says **No Function**.

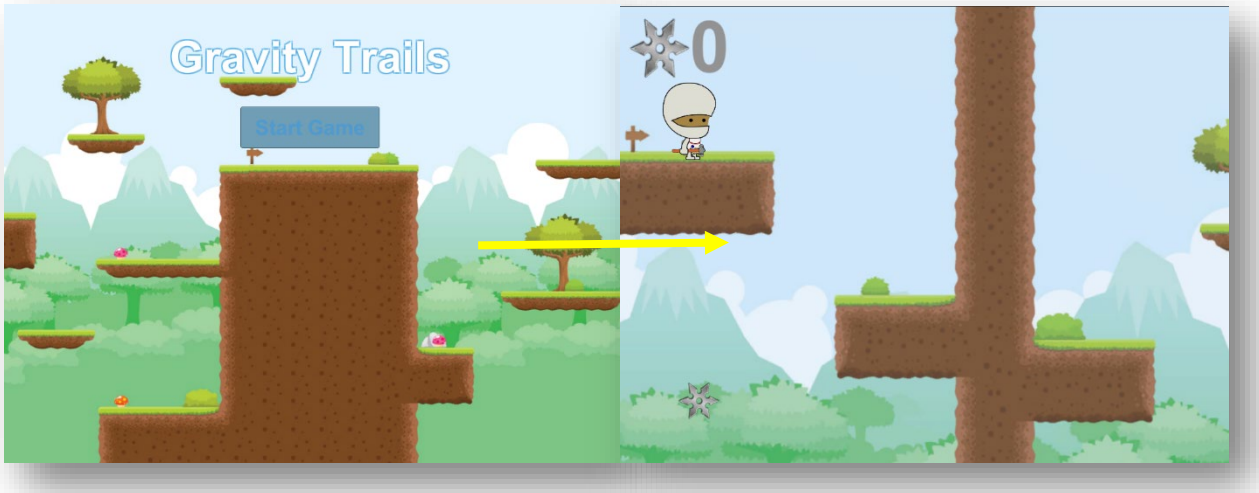
152 The component on the right is how we tell the button what script to look at and the function to run. Click on the arrow next to **No Function** to see the options we have.



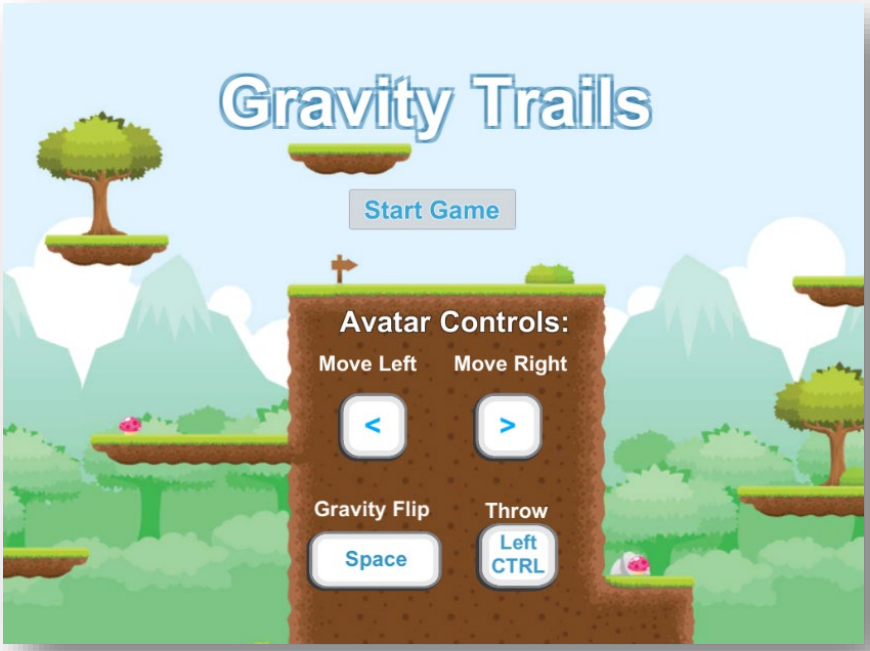
153 We want to access the function found in the **StartScene** script. Click on **StartScene** then **OnMouseDown()**.



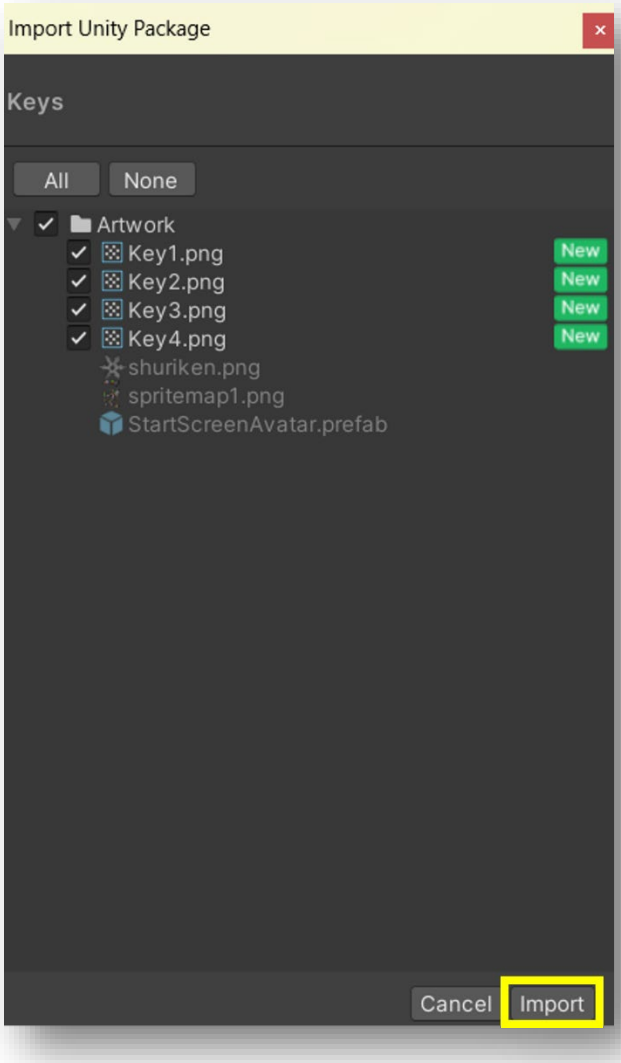
154 We programmed our **OnMouseDown()** function so that the game takes the player to level1. We then connected the logic in our script to our button. Playtest your scene, when you click the StartGame button does it load level1?



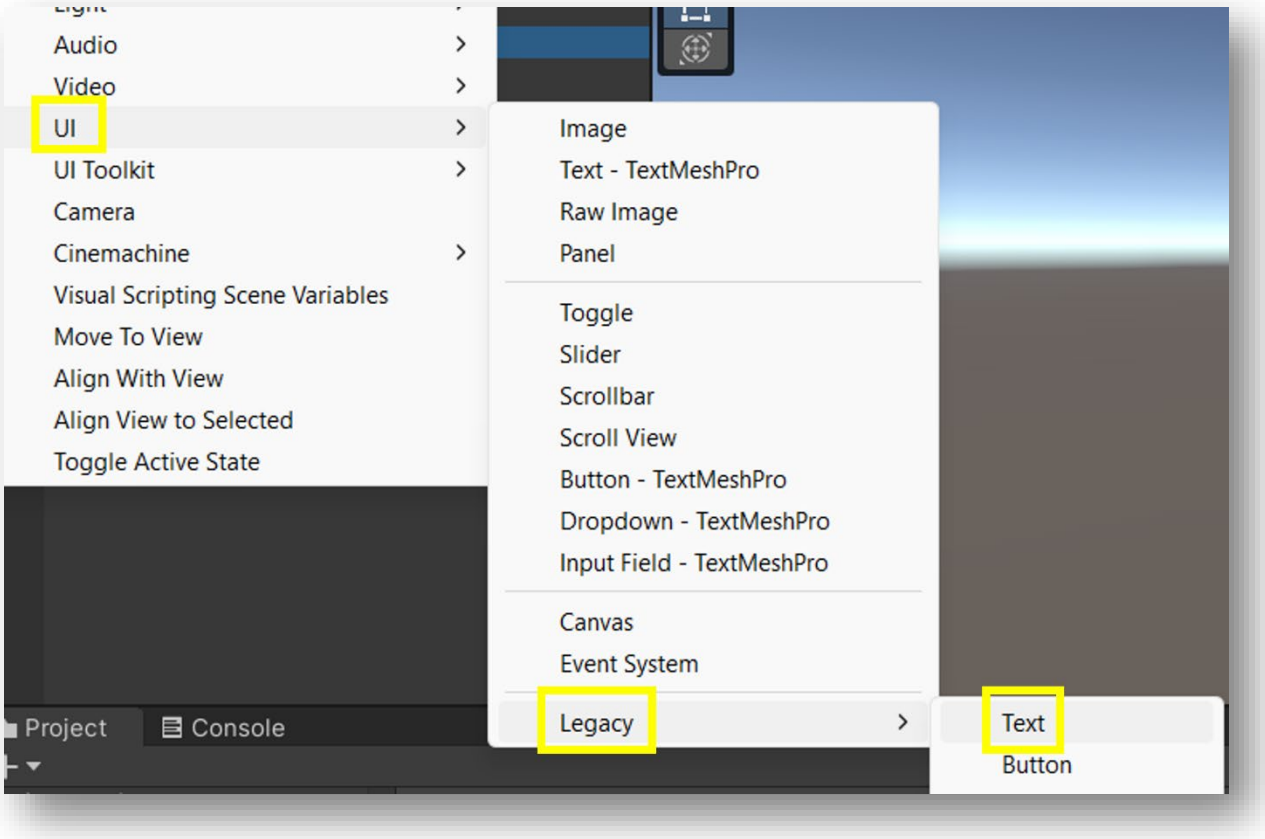
155 Finally, we will add the controls the player needs to know to play the game! It will look something like the image below.

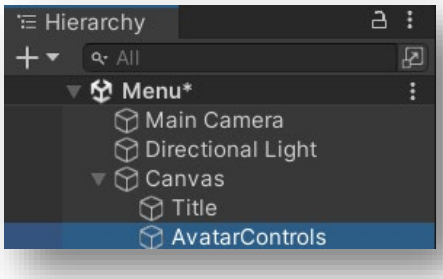


156 Import the **Activity 01 - Keys.unitypackage.** This package contains the keyboard images we will use to make the keys.

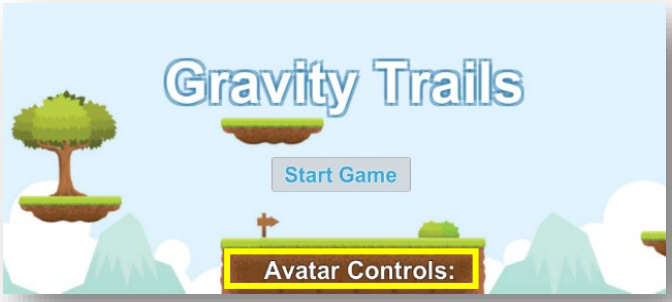


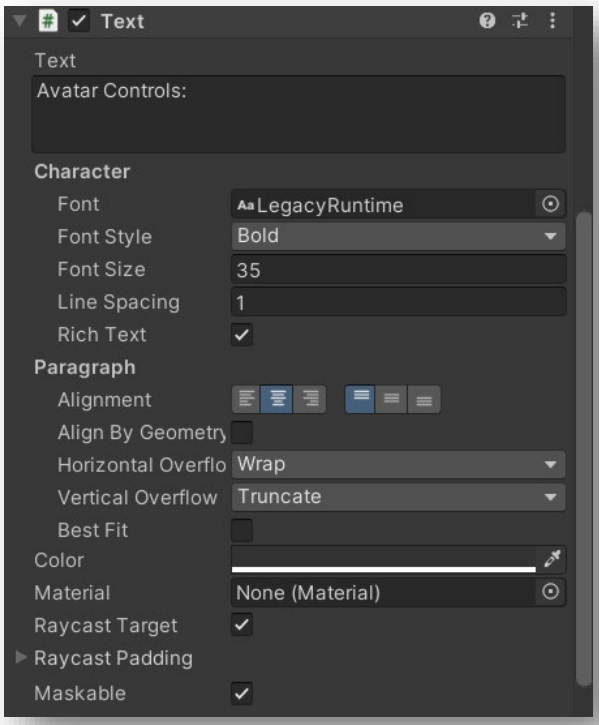
157 Create a **UI Legacy Text** object and rename it **AvatarControls**.





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

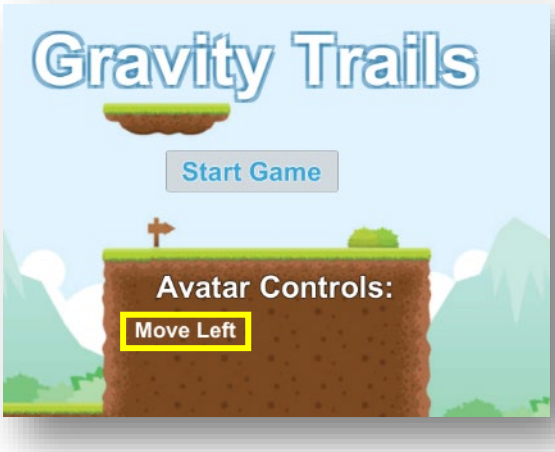




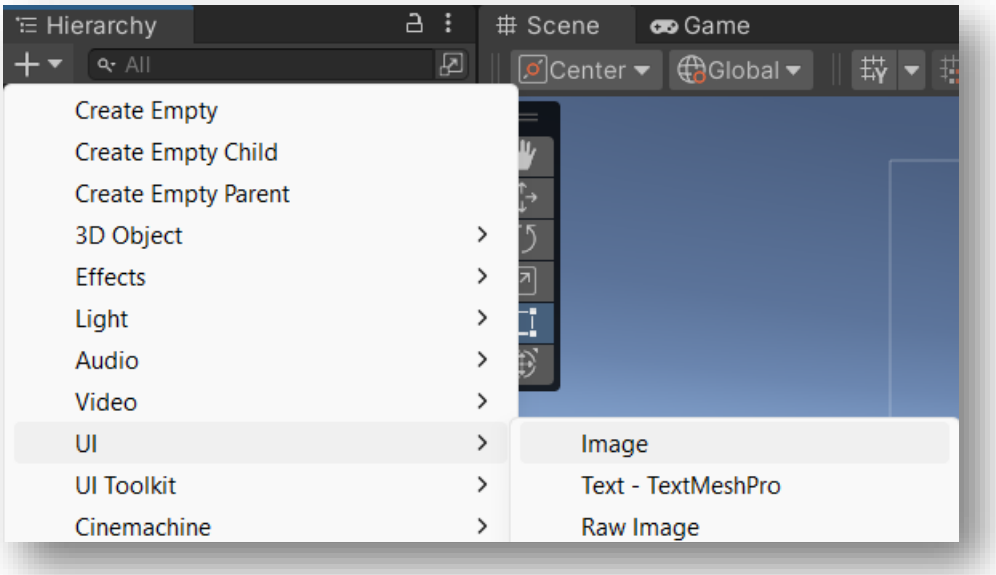
In this activity our main character is the Avatar, but in other games the main player might have a different name. You should change the name before the word "Controls" for other game menus that you create.

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

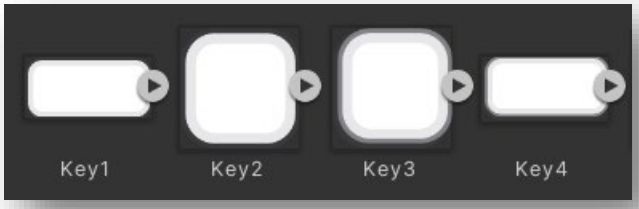
Create another text object and rename it **MoveLeft**. In the Inspector change the **Text** to **"Move Left"**. Customize the text and position it to your liking! Look at our example below:



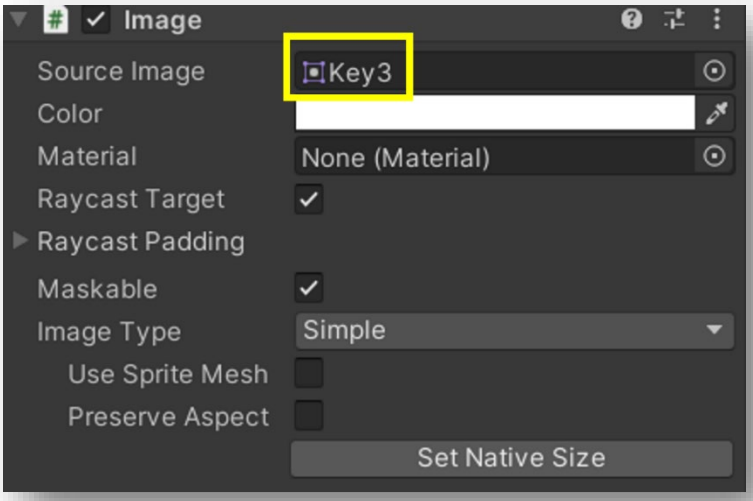
160 Here we tell the player that they can move left, but using what keyboard key? Below this let's add a key image! Create **UI Image** object and rename it **key**. Position it below the text.



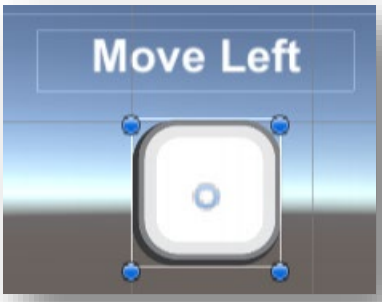
161 We have made a few key images that you can use. In the **Artwork** folder you can find 4 different keys.

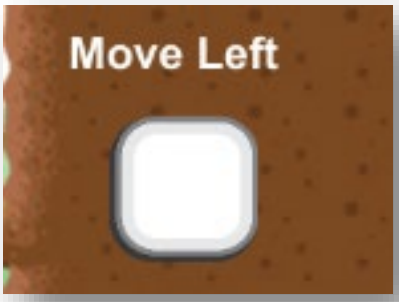


162 Make sure you have the **key** object selected, and in the Inspector drag the key sprite that you like into the **Source Image** component.

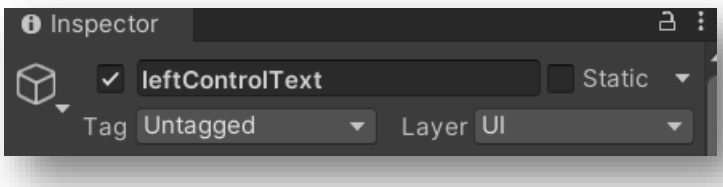


We used the **Key3** sprite. You should see the change in both the **Game** and **Scene** view.

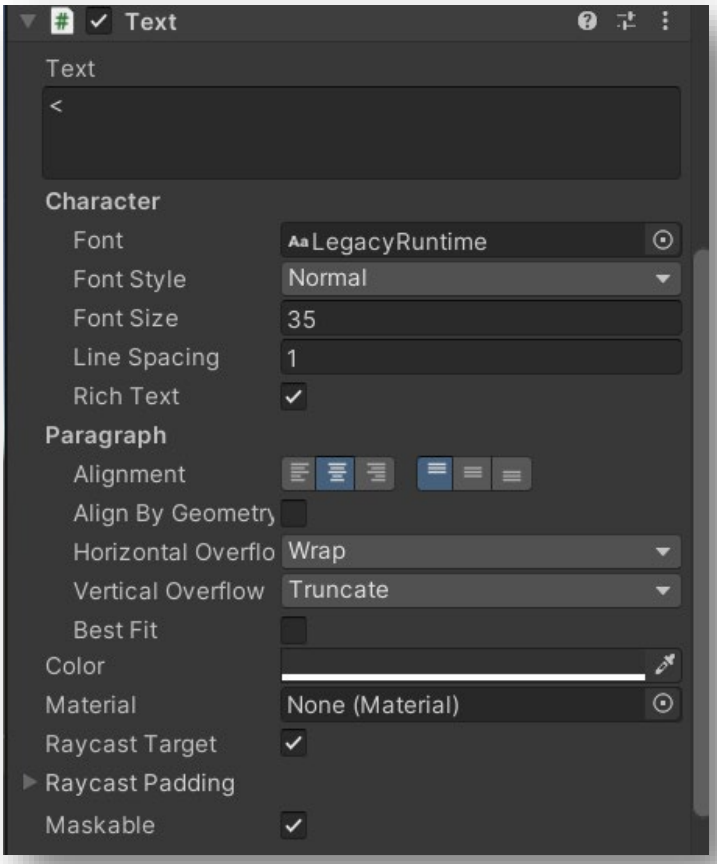




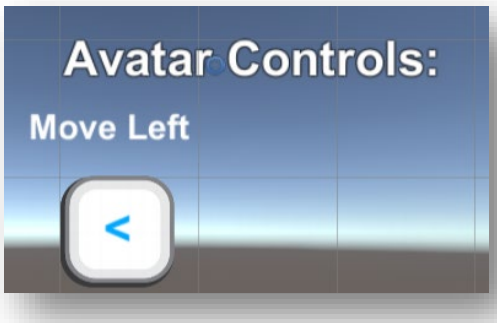
163 To complete the left control image, we want to place a text object inside of our image. Create another **UI Legacy Text** object. Rename it **leftControlText**.

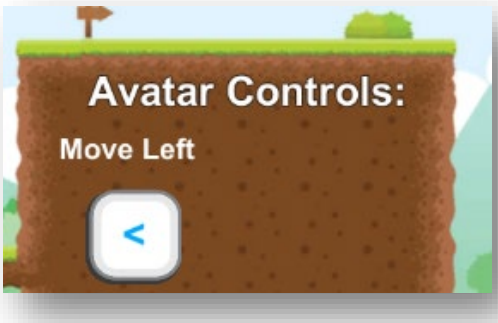


164 In the Inspector change the text to a left facing arrow **<** and customize it to your liking!



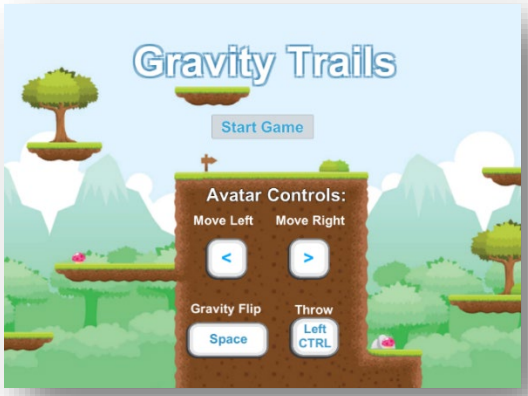
165 Your key that tells the player how to move the Avatar to the left is complete.





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

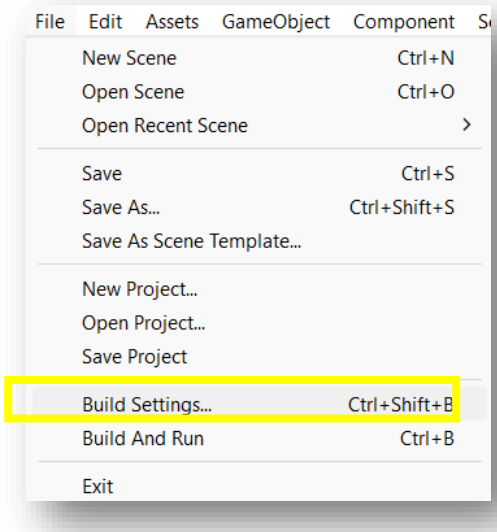
At the end you should have four total controls like the example below!



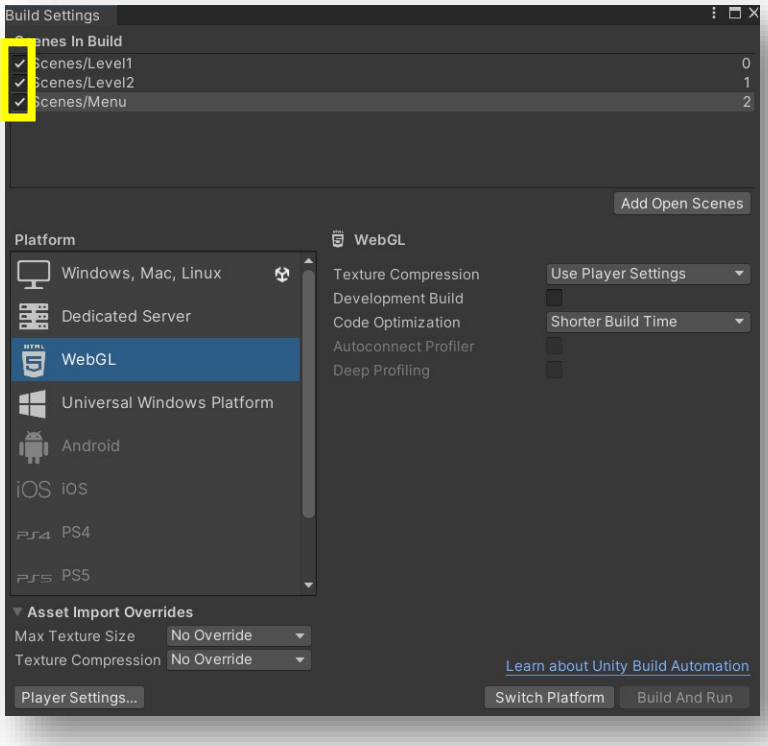
167 Playtest your Menu and have fun on level 1!

Sharing Your Unity Project

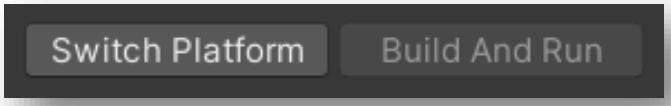
168 In Unity, click File, then Build Settings.



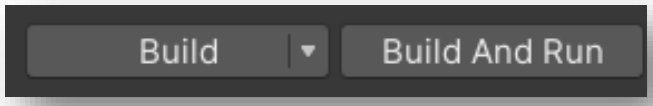
169 Make sure all scenes are checked. Under Platform, click WebGL.



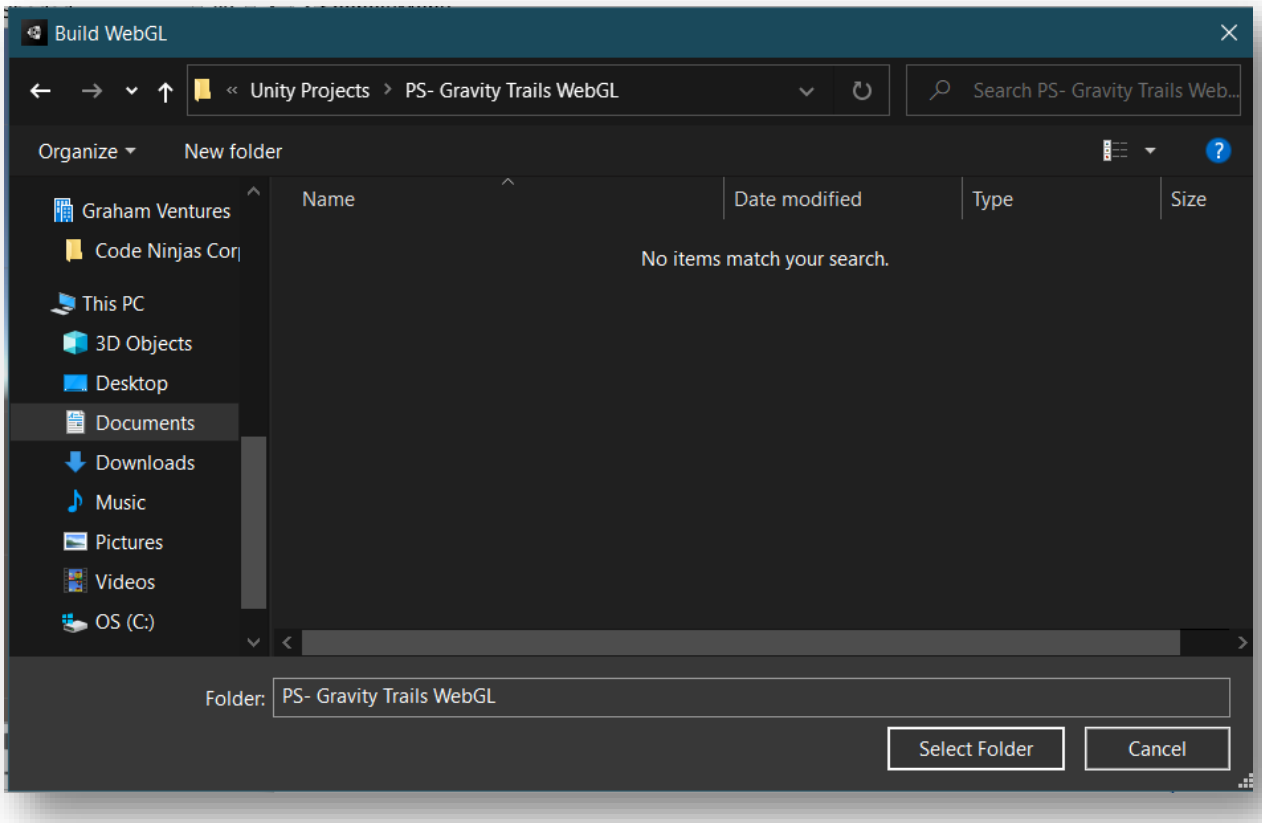
170 Click “Switch Platform.” This may take a few minutes.



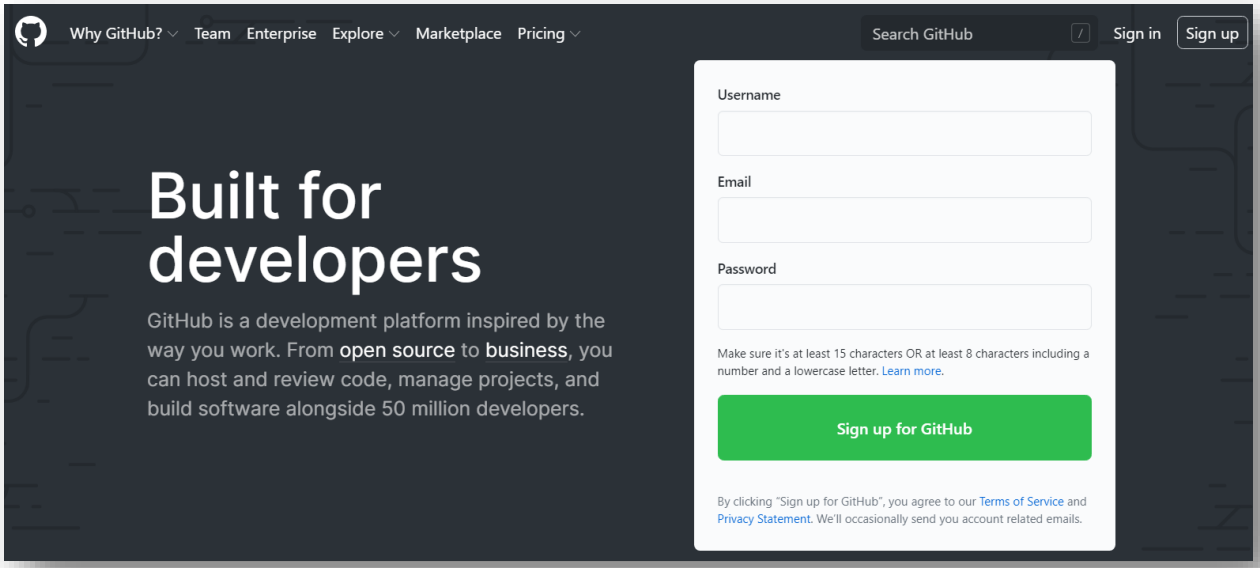
171 Click Build And Run.



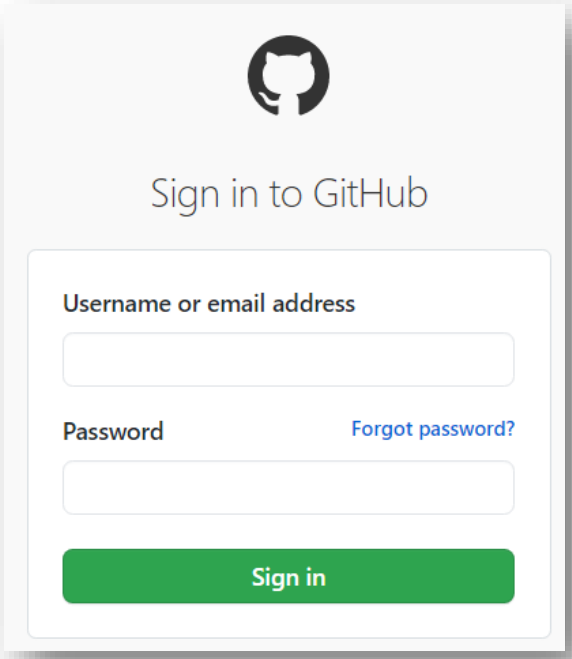
172 Create a folder and save the WebGL files in a location you will remember.



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

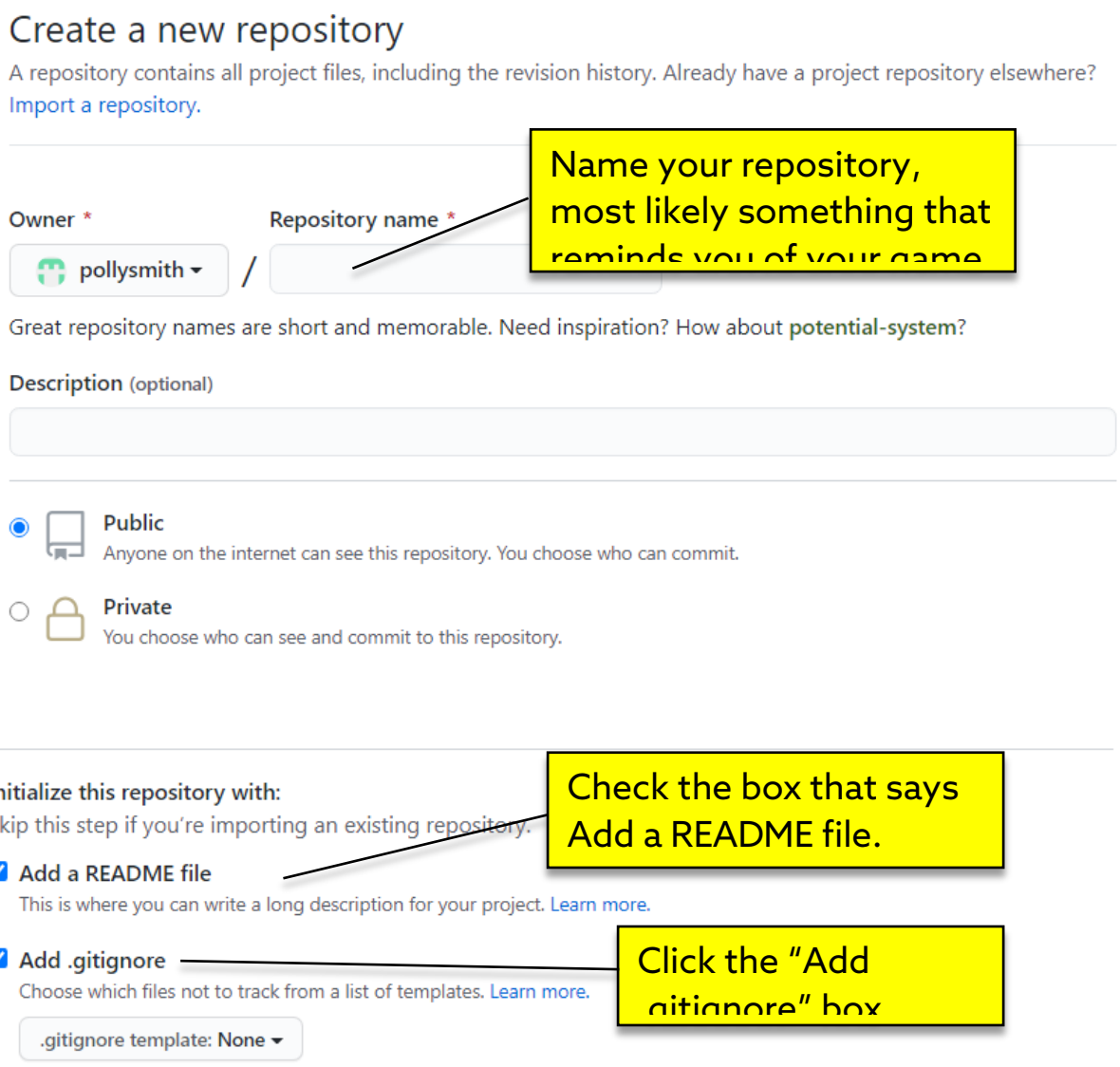


174 Sign into GitHub.

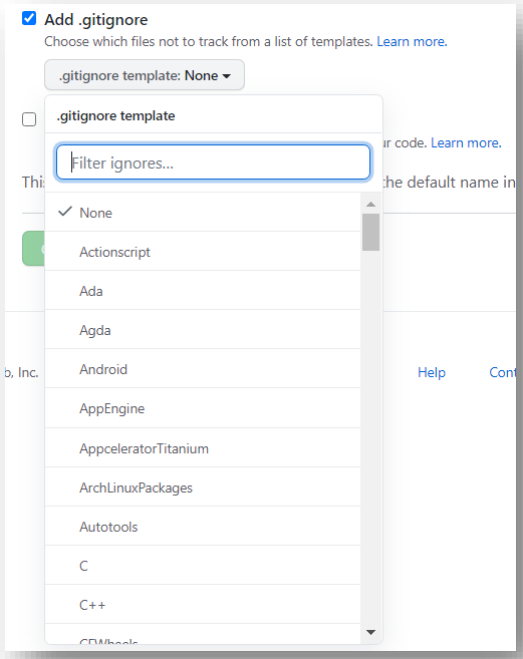


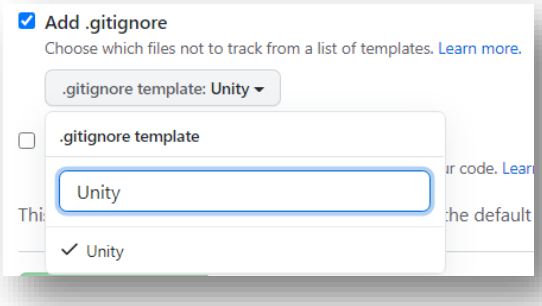
175 In your dashboard, click the "+ New Repository" button.

176 Create a new repository.

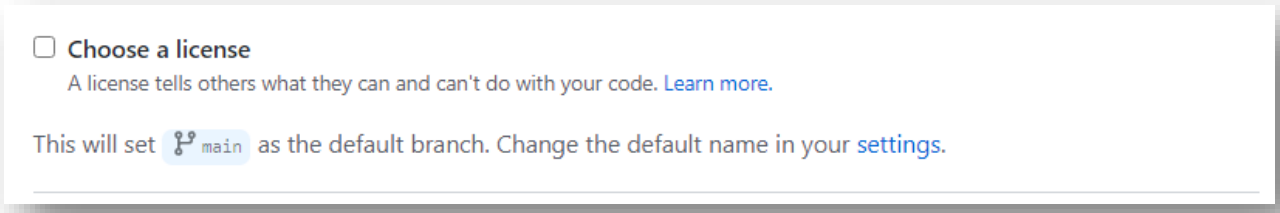


177 Click the .gitignore template: None dropdown menu, and type “Unity”.

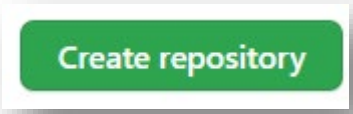


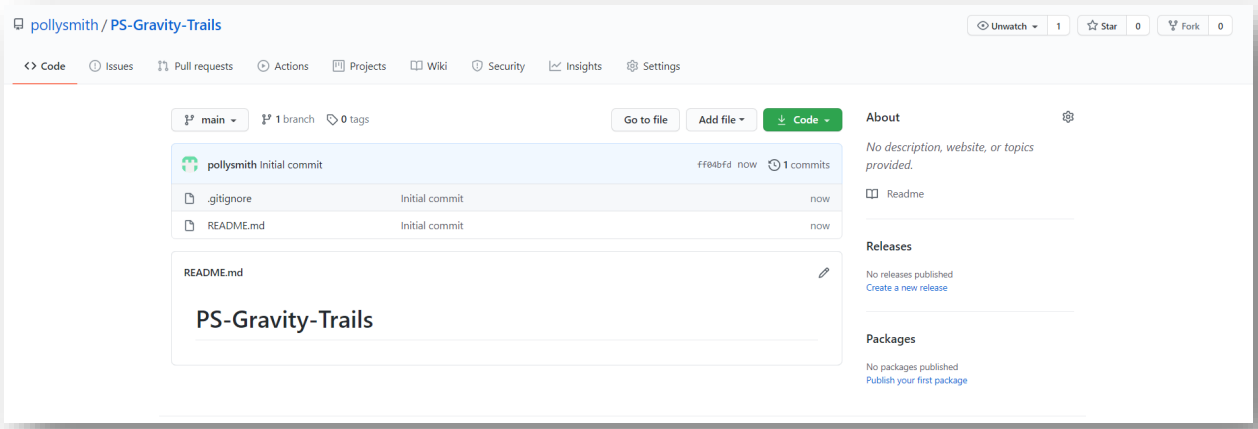


178 Leave the “Add a license” box alone unless you know what kind of license you want to use.

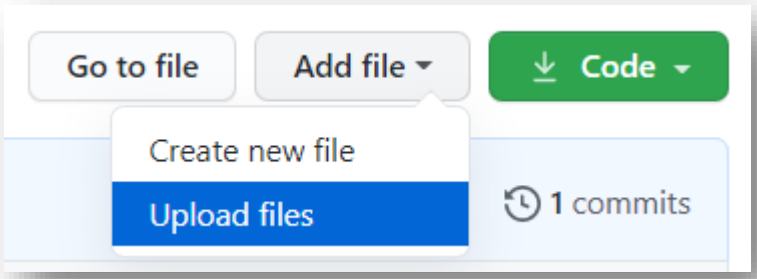


179 Click the Create repository button.

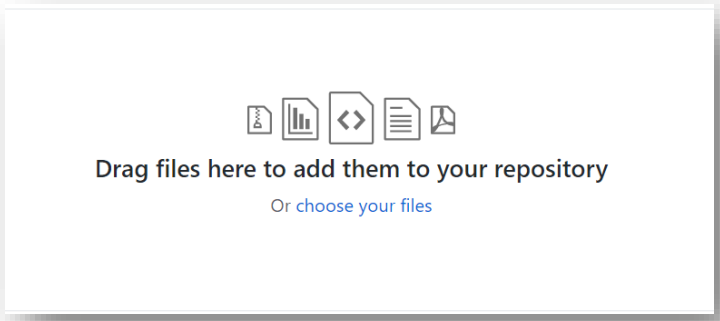




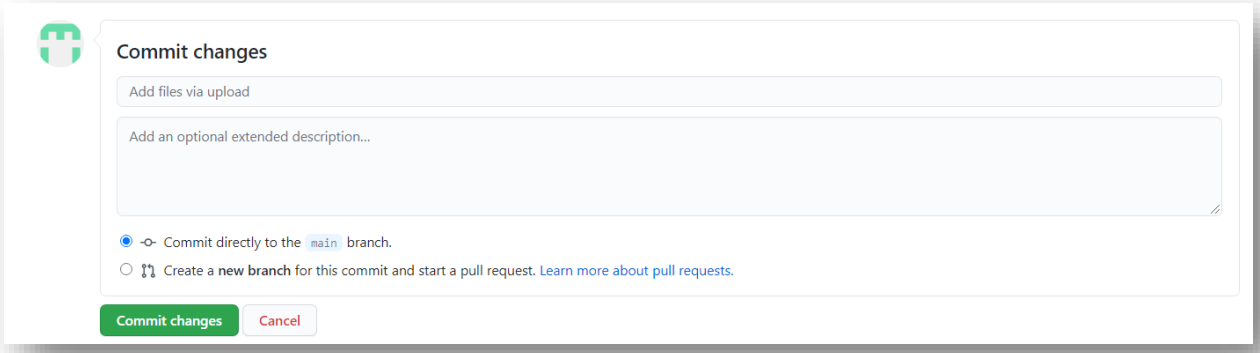
180 Click Add file, then Upload files.



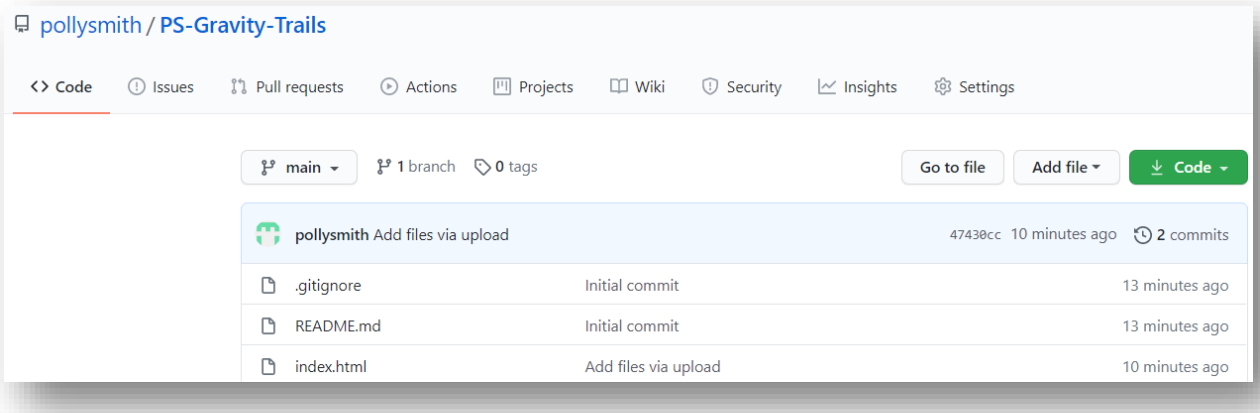
181 Click “choose your files,” then locate the folder where you saved your game’s WebGL files. Upload *index.html*, *Build/*, and *TemplateData/*



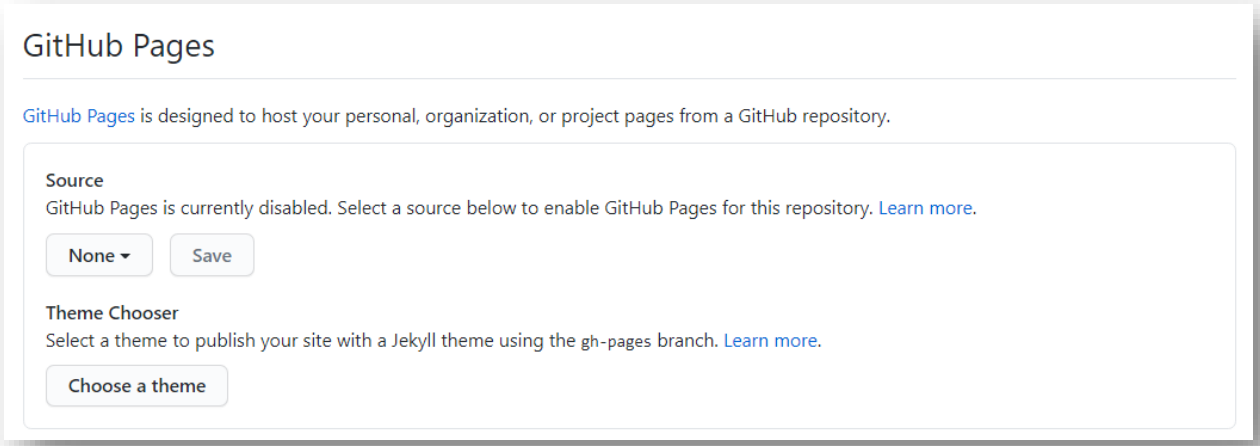
182 Click Commit changes.



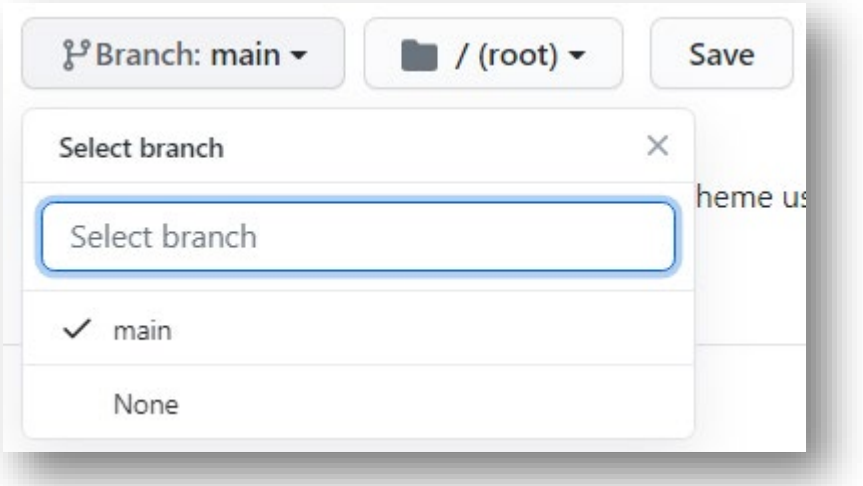
183 Once your files are uploaded, click on “Settings”.



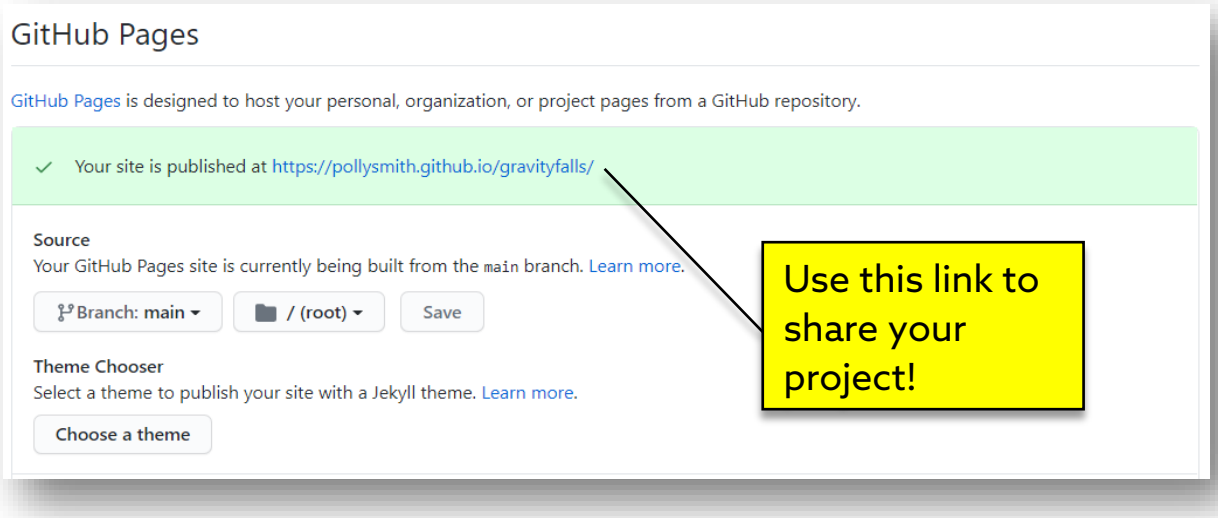
184 Scroll down to “GitHub Pages”.



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



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



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

188 Your game is now published! You can share the GitHub URL to have friends and family play your game!

Project Submission and Reflection

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

Once you feel like you have a good product that represents your vision of the game, have a 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 Gravity Trails Project Requirements Checklist to make sure your game has all the required features.

Ninja Planning Document

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