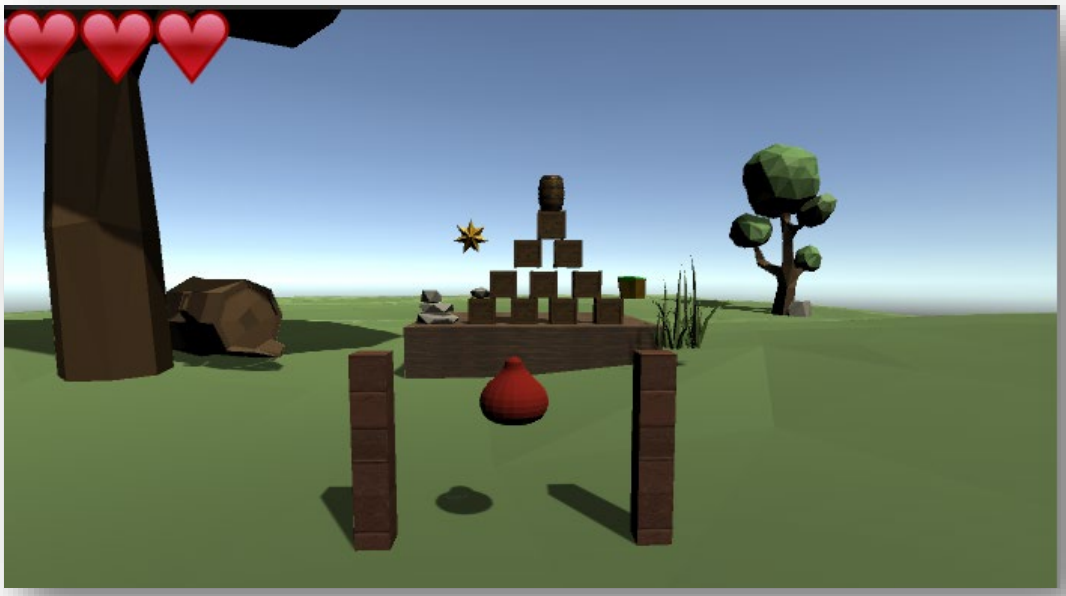
CODE NINJAS®

Platinum Belt Ninja Guide
Activity 03: Sulky Slimes

Sulky Slimes

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

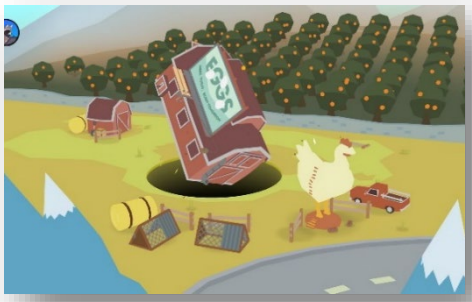


The game we will create together is influenced by the popular Angry Birds game series, but there are many other games and ideas that are built on similar concepts. As you plan and program, think about other similar games and try to come up with a few new ideas on your own.

Plan and Design

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

Kerbal Space Program by Squad
Built in Unity



Donut County by Ben Esposito
Built in Unity

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

Ninja Planning Document

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

Look at the sample projects below for some inspiration!

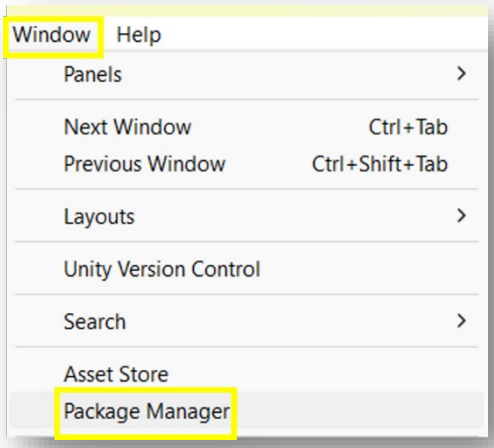
Project Setup

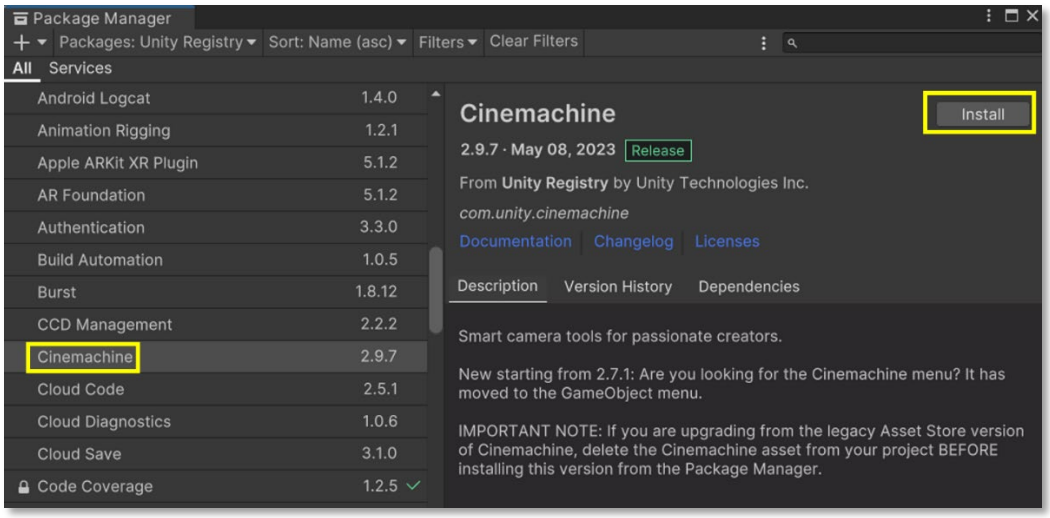
1 Start a new Unity Project and name it *YOUR INITIALS – Sulky Slimes*. Select **3D template**.

2 After it loads, rename the Sample Scene to Sulky Slimes.

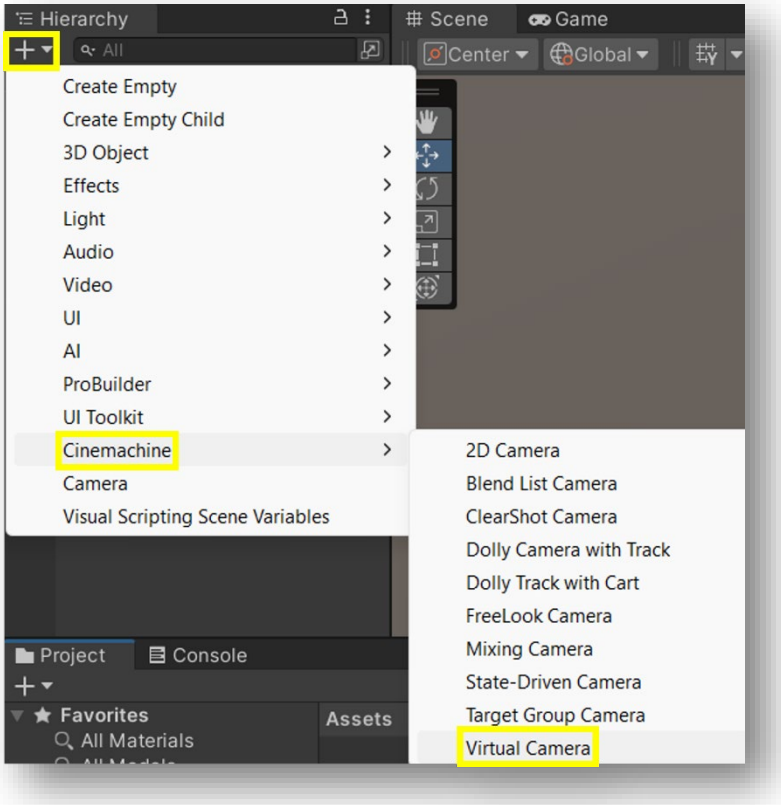
3 Open the **Package Manager** from the **Window Menu**.

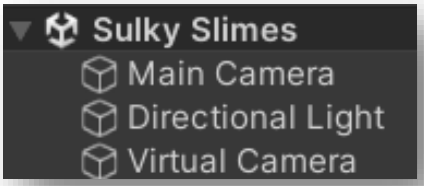
Find **Cinemachine** and click Install in the bottom right of the window.





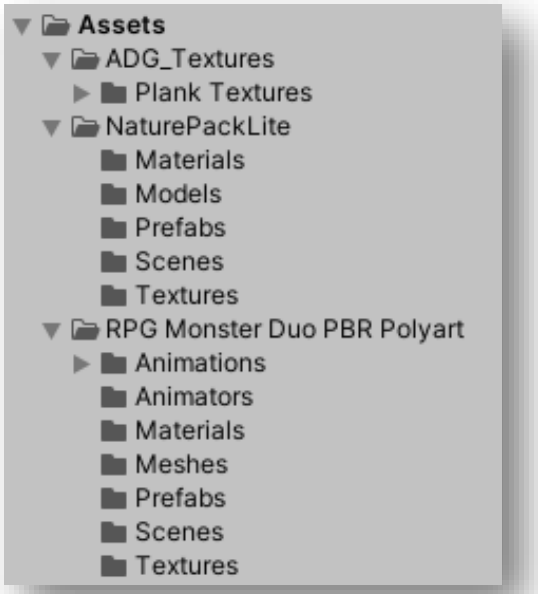
4 You should now see a **Virtual Camera** in your object **Hierarchy**. If not, go to **Cinemachine -> Create Virtual Camera**.





A Whole New World

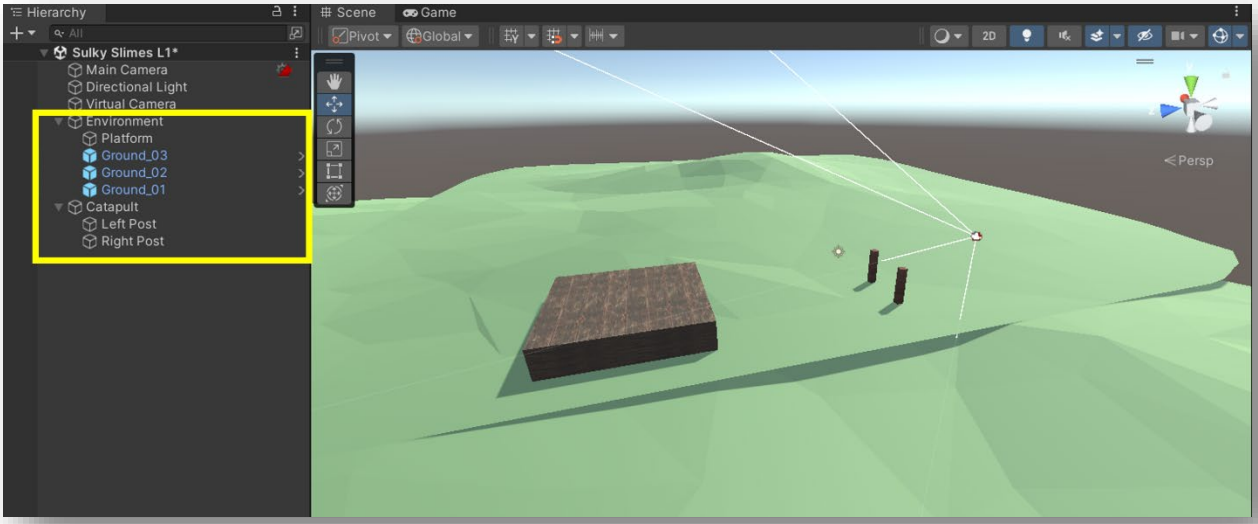
5 Use Code Ninjas assets, assets from the Unity Asset Store, or your own creations to build a **basic ground**, a **platform** for your targets, and **simple catapult** or **slingshot** structure to launch your object. Your platform and object launcher should both be on the ground. Don't worry about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



Ninja Planning Document

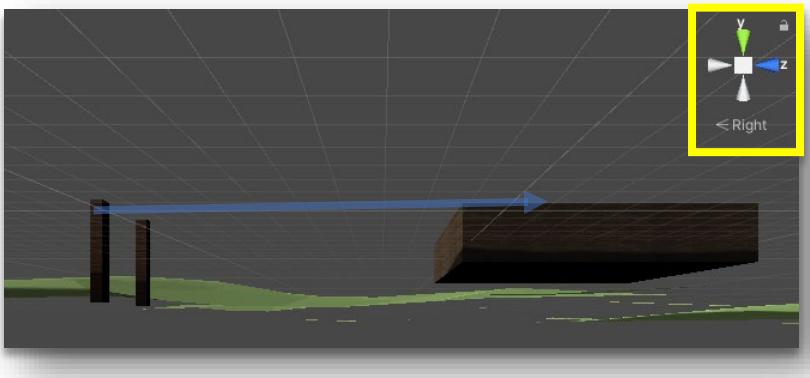
Use your Ninja Planning Document to help you create your vision in Unity!

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



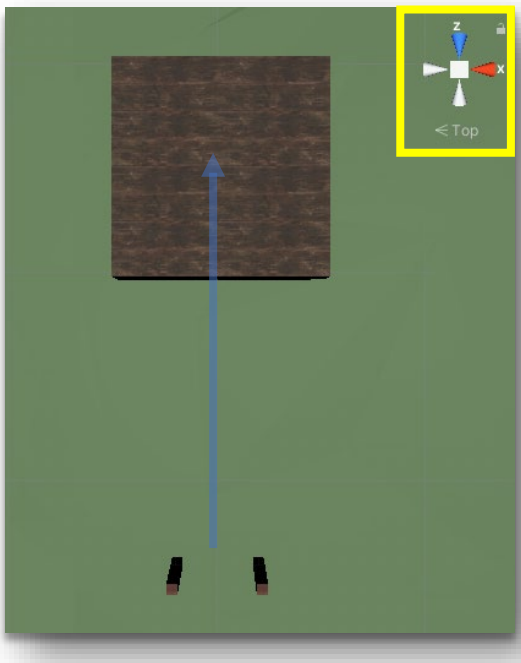
7 To ensure our math works out later in the game, we want to align our catapult with the platform. We want to launch our object in the positive z direction. Click the red, green, and blue arrows on the axis indicator in the top right of the scene view to see your scene from the top and the right. Use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

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



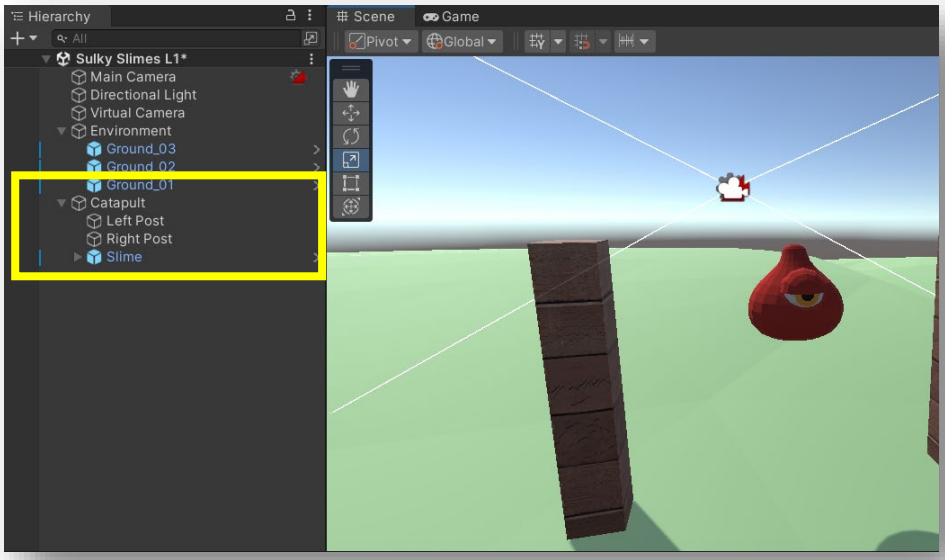
When viewing from the right, the platform should be to the right of the catapult.

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



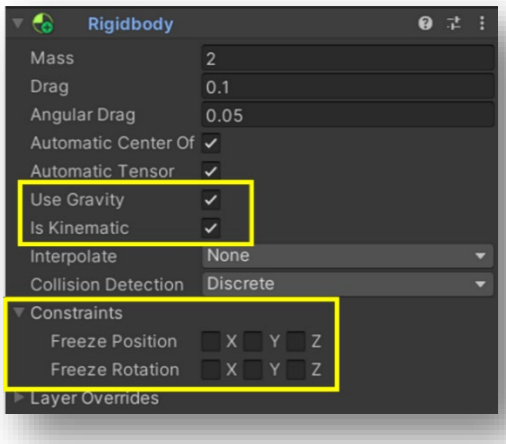
Character Building

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

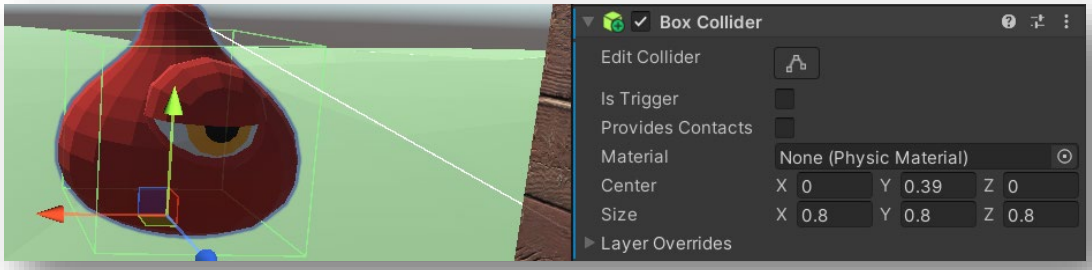


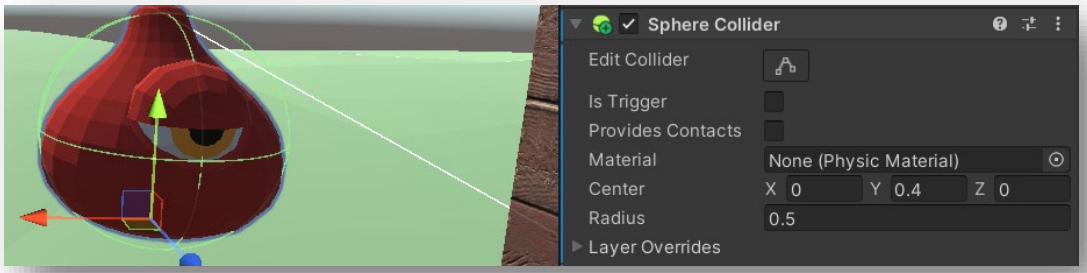
Scale it and have it hover like in the image.

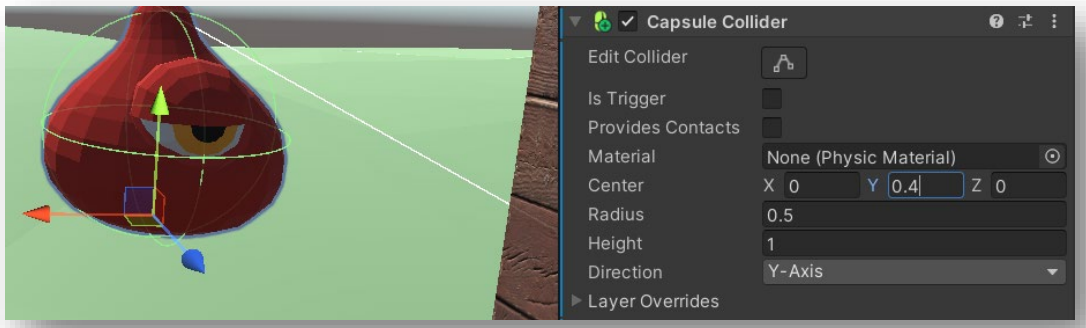
9 Make sure your character has a Rigidbody. We will alter many of these values during our playtests, but make sure **Use Gravity** and Is **Kinematic** are checked and the **Freeze Position** and **Freeze Rotation** boxes are unchecked.

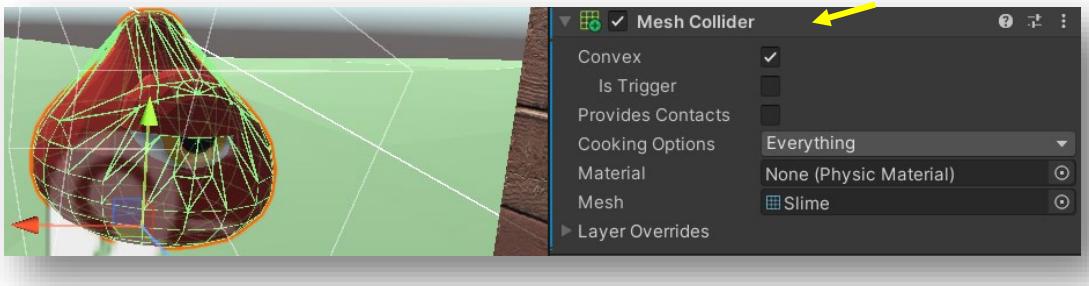


10 Make sure your character has a **Collider** that is positioned and sized to fit your character.



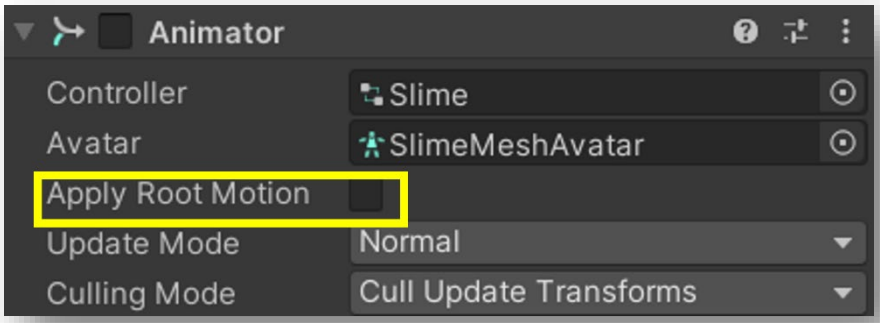






If your character has a **Mesh Collider**, make sure that the **Convex** option is enabled. The **Convex** option tells Unity that you want this character to interact with other types of **colliders**.

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

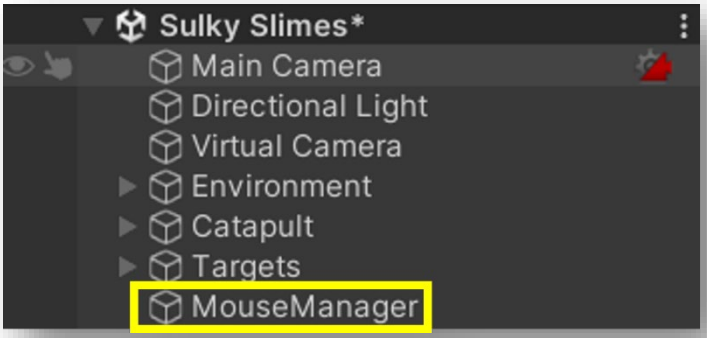


Imported Assets

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

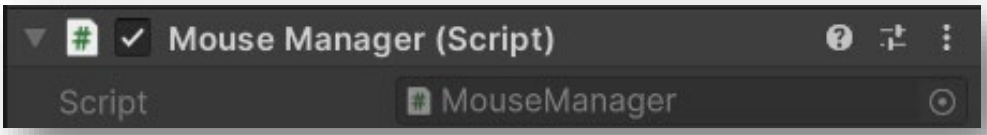
Mighty Mouse

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

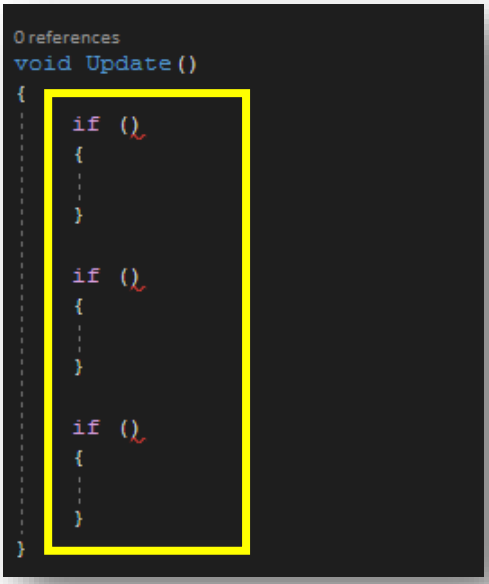


13 Create a new **script** in your Script folder in the **Assets** tab and name it Mouse Manager.

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



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



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

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

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

    }
    if (Input.GetMouseButton(0))
    {

    }
    if (Input.GetMouseButtonUp(0))
    {

    }
}
```

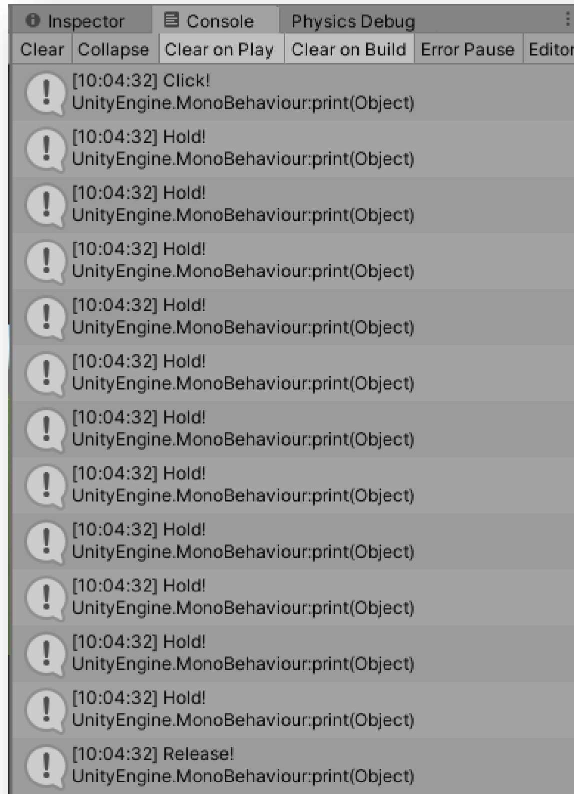
16 Inside each of the **if statements**, put a **print** statement to check to see when each of these will run.

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

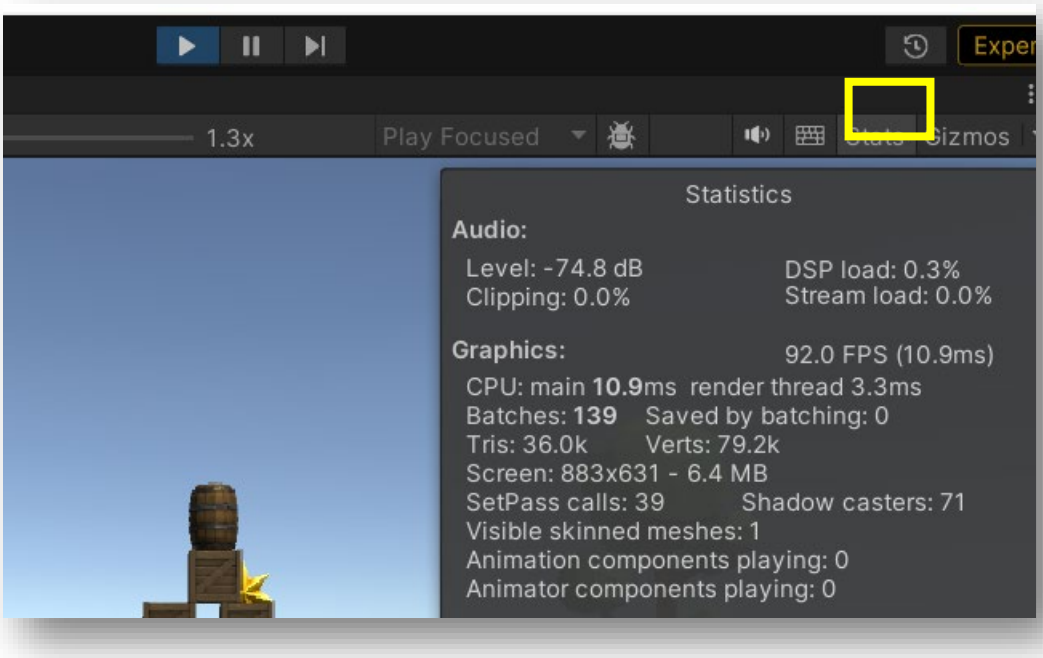
17 **Playtest** your game. Click the left mouse button and look at the **console**.

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

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



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



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

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

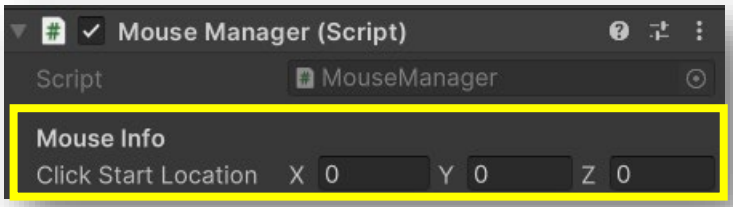
In the **MouseManager script**, create a **public class variable** named **clickStartLocation** to store the mouse position when the left mouse button is first clicked.

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

```
public Vector3 clickStartLocation;

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

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

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

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

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



Sensei Stop

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

21

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

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

```
public Vector3 clickStartLocation;

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

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

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

22

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

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

```
public Vector3 clickStartLocation;

public Vector3 launchVector;
```

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

```
public Vector3 clickStartLocation;
public Vector3 launchVector;

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

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

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

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

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

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

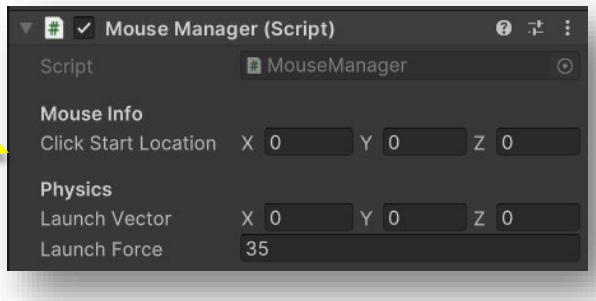
```
public Vector3 clickStartLocation;
public Vector3 launchVector;

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

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

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

25 Add a **public float** named **launchForce** and give it a starting **value** of 10 in the Unity Inspector. We need to **playtest** before we know exactly what a good number will be!

```
public Vector3 clickStartLocation;
public Vector3 launchVector;

public float launchForce;

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

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

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

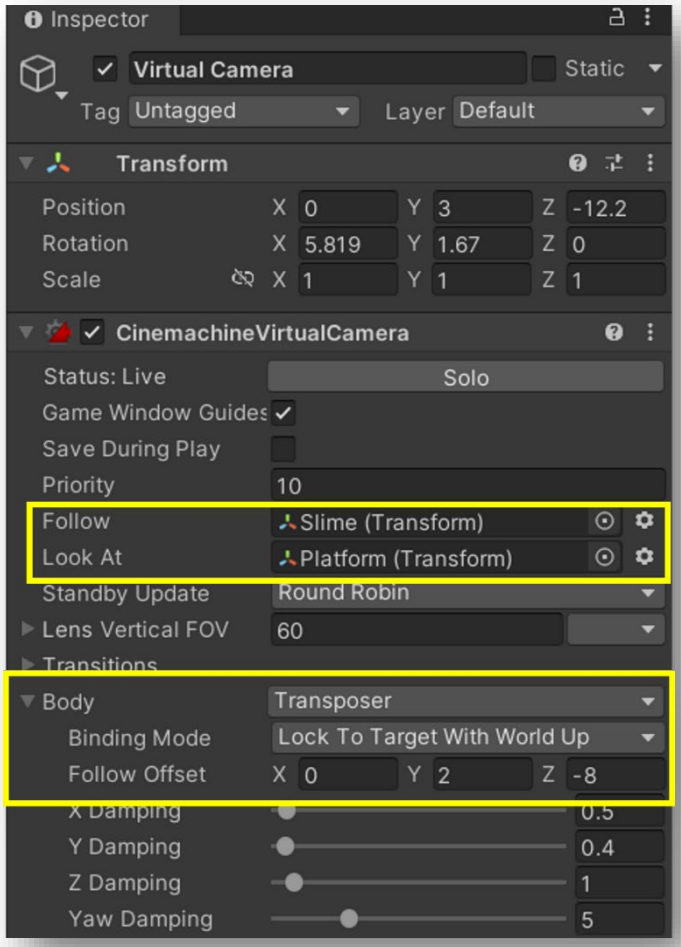
26 Now that we have a few different **public variables**, we can use a special line of code to set up **headers** to help us stay organized. Add **[Header("Mouse Info")]** and **[Header("Physics ")]** above the related **variables**.

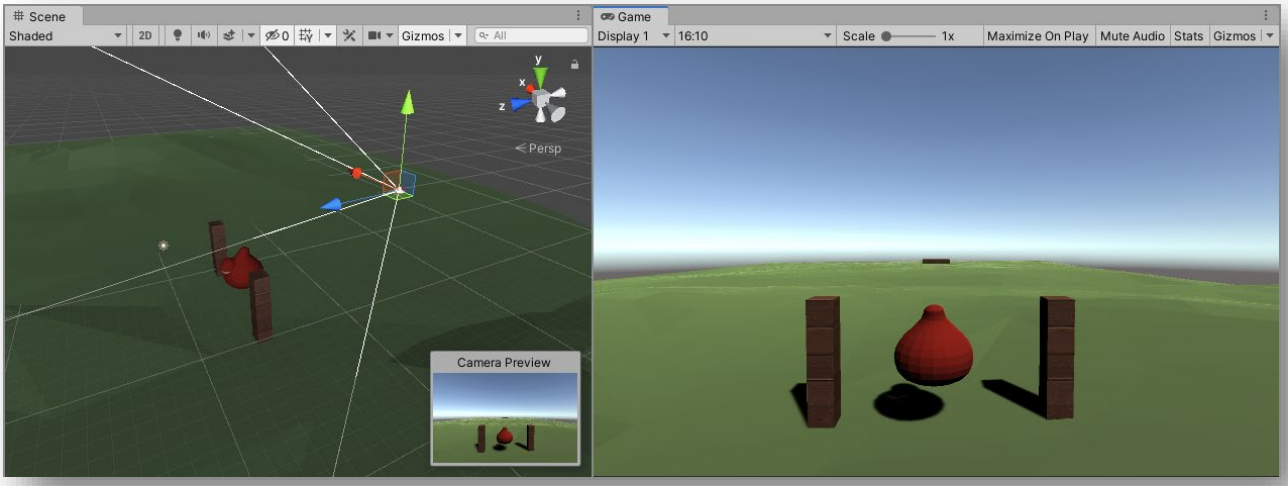
```
[Header("Mouse Info")]
public Vector3 clickStartLocation;

[Header("Physics")]
public Vector3 launchVector;
public float launchForce;
```



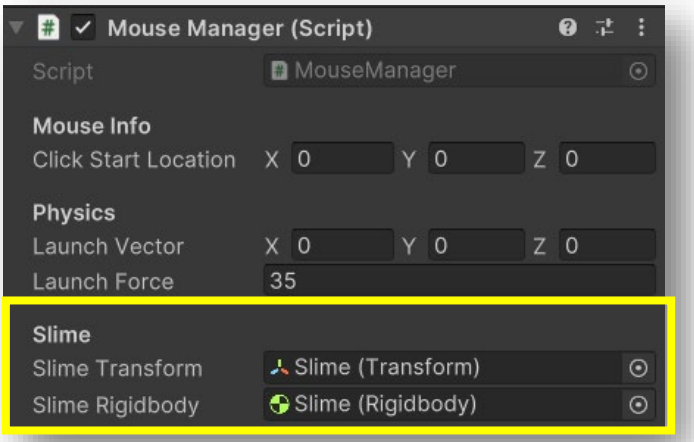
27 Before we write the code that launches the slime, we need to make sure the **camera** is behind the slime.

Open the **Virtual Camera** object and adjust the **Cinemachine Virtual Camera** component. The **camera** should follow the slime and look at the platform. Set the **values** of the **Body's Follow Offset** to be behind and slightly above the slime. Your numbers might be different than the image.





28 Now that we have our **camera** positioned and we are tracking how far the **mouse** has moved, we can use it to launch the slime. Open the **Mouse Manager script** again. We need to add two **variables**, a **public Transform** and **public Rigidbody**.



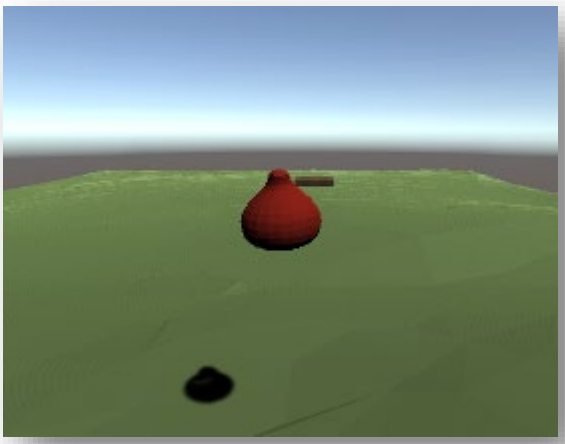

```
[Header("Slime")]
public Transform slimeTransform;
public Rigidbody slimeRigidbody;
```

29 Now that we access to the slime's transform and rigid body, we can program the logic to launch it! When the mouse button is released, set the slime's rigid body is kinematic to false to enable forces. Then, add a force.

Sensei Stop

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

30 **Playtest** your game. Adjust the **launch force** based on your playtests. Did your slime go straight up in the air? Try changing the **z coordinate** of the **launch vector**. Did your slime go in a random direction? Make sure that your launcher and platform are aligned properly. Did it flop to the ground or fly too far? Adjust your **launchForce**.



31 We need to give the player a way to reset the slime. We need to create **variables** that store the slime's original **position** and **rotation** when the game starts. When the user presses a key or right clicks the mouse, we need to reset the slime's **isKinematic property**, **position** (**Vector3**), and **rotation** (**Quaternion**).

Sensei Stop

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

You can use **Input.GetMouseButtonDown(1)** to check to see if the user right clicked or **Input.GetKeyDown("space")** to see if the user pressed the space bar.

32 One problem with our game is that we aren't giving **feedback** to the player before they launch the slime. We can write code that makes the slime follow the mouse movements as the player is dragging the mouse to launch.

Before we **normalize** the launch **vector**, we can use it to move the slime. Because the mouse **coordinates** are based on the size of the computer screen and not the game scene, we need to **divide** our launch **vector** by a large number to make sure the slime doesn't fly out of the scene when the user drags the mouse.

```
if (Input.GetMouseButton(0))
{
    Vector3 mouseDifference = clickStartLocation - Input.mousePosition;
    launchVector = new Vector3(
        mouseDifference.x * 1f,
        mouseDifference.y * 1.2f,
        mouseDifference.y * 1.5f
    );
    slimeTransform.position = originalSlimePosition - launchVector / 400;
    launchVector.Normalize();
}
```

That's all the programming we need to create a physics-based game. We respond to user input, calculate and apply forces, and reset the game objects.

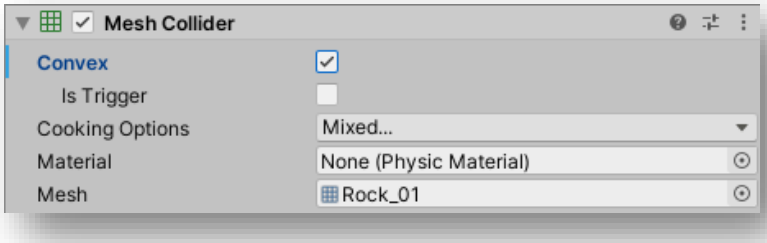
Reach for the Star

33 We now need to give the player a **goal** to complete. We need to add **objects** for the player to launch into and stars that the player needs to collect, but you should design your scene based on your planning document.

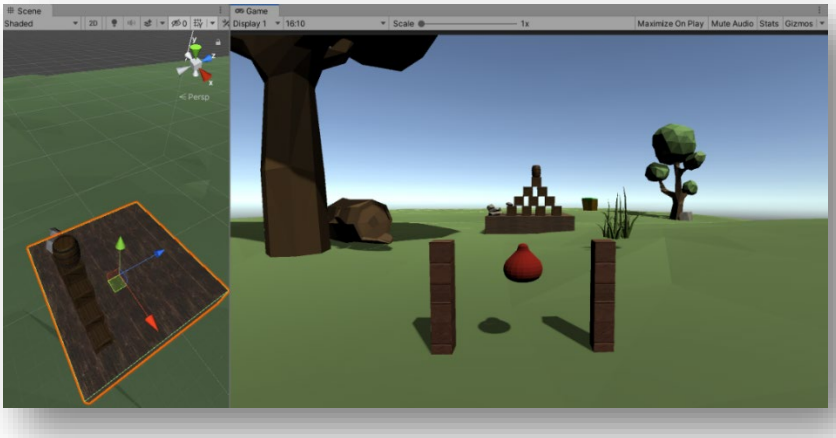
Search for **assets** or make your own and place them on your platform **object**. Create **empty objects** named "targets" and "collectables" to help you organize your scene.

Make sure your models have **colliders** and **rigidbodies**. Make sure that the **colliders** match the shape and the size of the **object**. Experiment by giving different objects different **masses**.

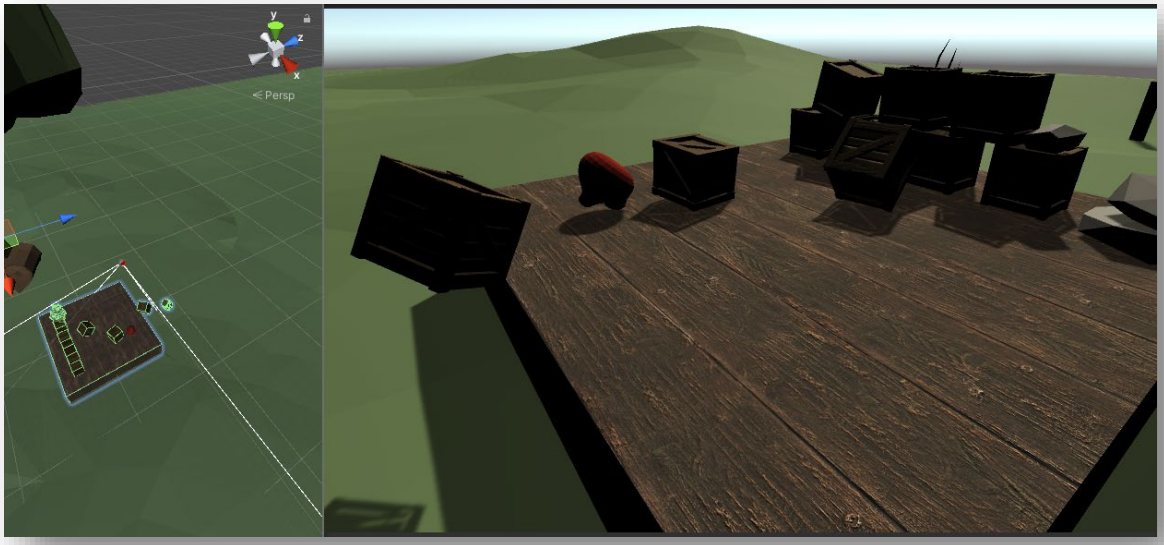
If any of your **objects** already have a **Mesh Collider**, make sure that **Convex** is enabled.



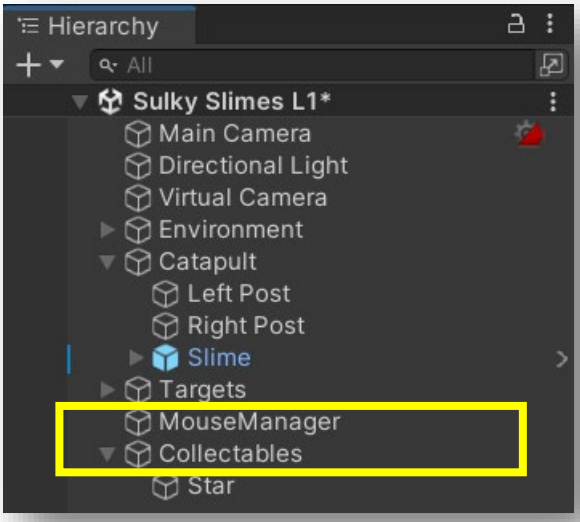
34 Place your objects inside the **Targets** game object.

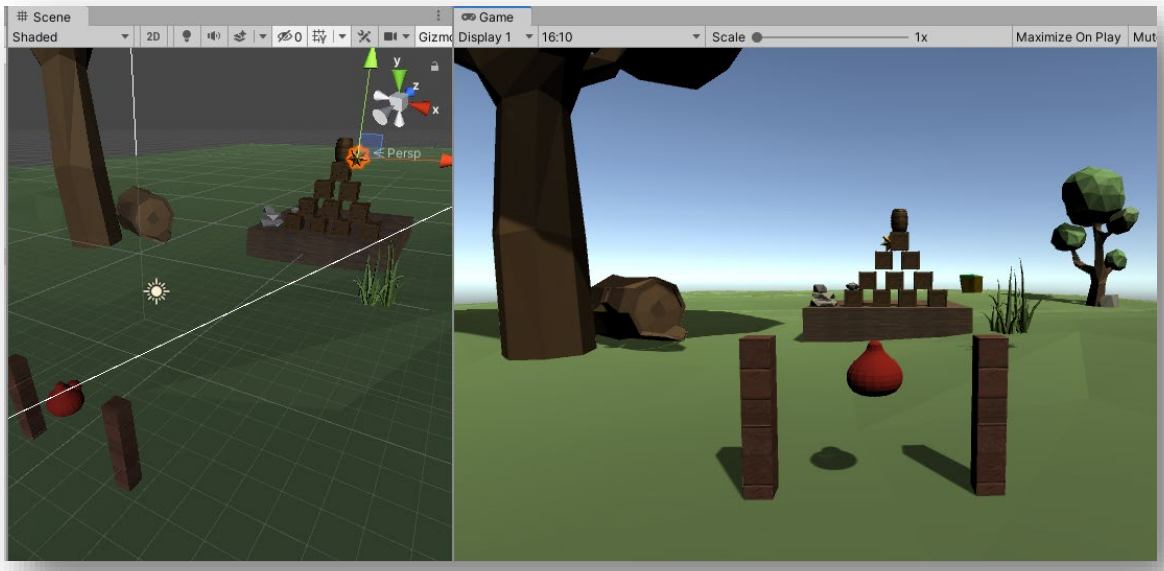


35 Playtest your game and make sure that the slime can collide and interact with all the objects that you placed.



36 Find an asset for your player to collect and place it behind the obstacles in the scene. Make it a child of the **Collectables** object.

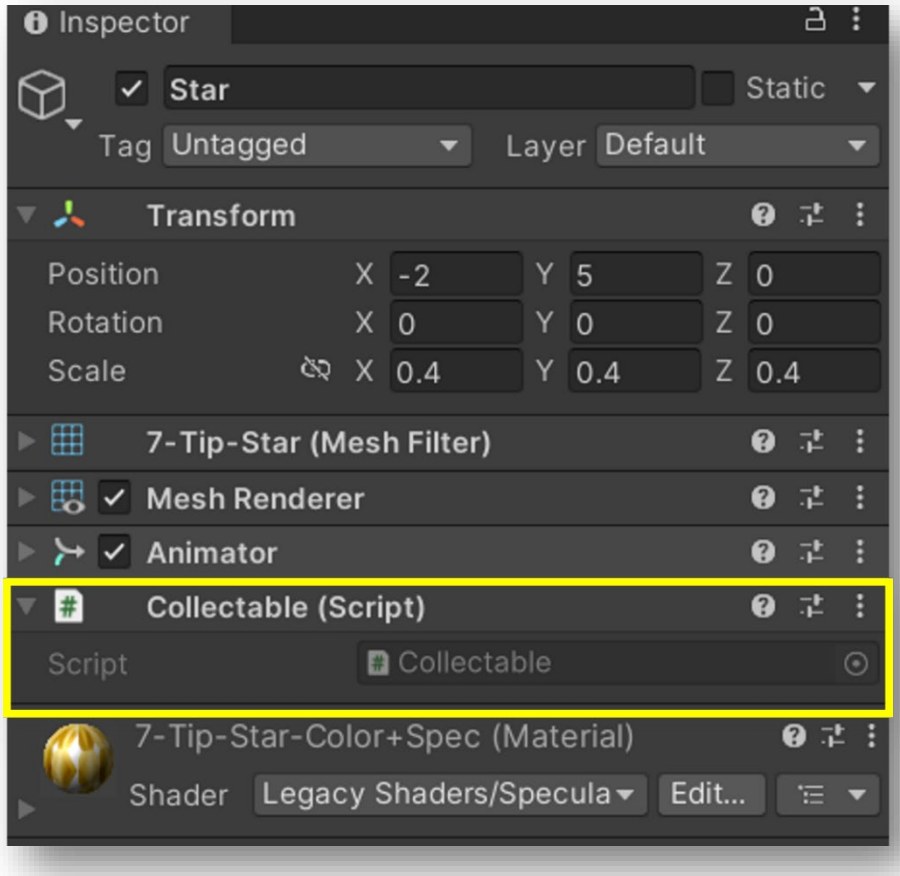




37 Based on your **Ninja Planning Document**, decide what challenge you want to present to the player. Are the **collectables** moving or changing size? Is there a time or attempt limit?

We will program a moving object and a lives system. Let's first work on making our star move so the player needs to aim their slime carefully.

Create a new **script** in the **Asset** panel, name it **Collectable**, and add it to the collectable **object**.



Making It Your Own

In our example game, we want the player to collect the object, but your game's goal might be different! Remember to use your Ninja Planning Document to help you make your game unique!

38 Open the Collectable **script**. We will program **movement** code like **Robomania** and **Gravity Trails**.

In Robomania, we knew where the edges of the screen were, and we changed the enemy's direction based on that information. Since we have an open 3D space in Sulky Slimes, we can calculate how far to move the collectable in its **Start function**.

In the Collectable **script**, add a **public float variable** named **distanceToMove**. We also need two variables to store the starting and ending positions. Since we are calculating the **values** inside the **script**, they can be **private**.

```
0 references
public class Collectable : MonoBehaviour
{
    public float distanceToMove;

    private Vector3 startingPosition;
    private Vector3 endingPosition;

    0 references
    void Start()
    {

    }



    0 references
    void Update()
    {

    }

}
```

39 In the **Start function** we need to store the collectable's original **position** and set its ending **position** based on the **value** of **distanceToMove**.

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

Hint: Use the **value** of **transform.position** when you set the **values** of **startingPosition** and **endingPosition**.

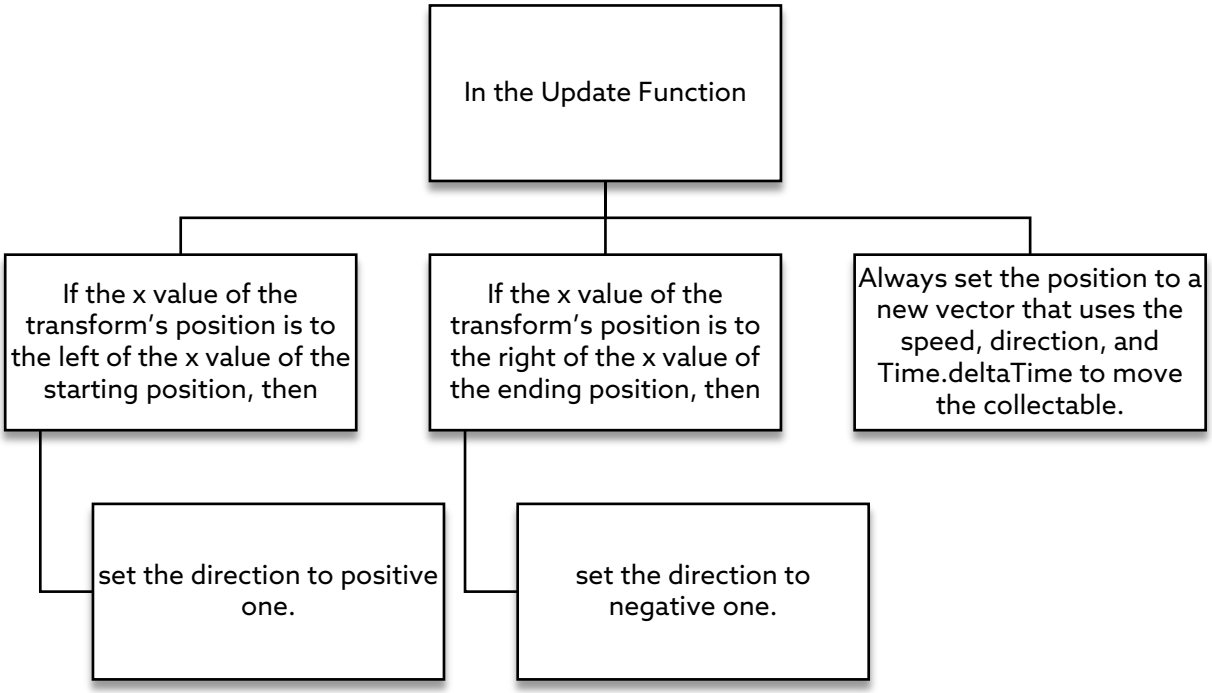
40 If we **playtest** the game now, nothing will happen because we haven't written our **update function** yet.

We need to check the current **x position** of our collectable and compare it to our starting and ending **x positions**. We then need to change its direction based on if it has gone to the right of the ending position or to the left of the starting position. Then we need to move the collectable using a speed, a direction, and **Time.deltaTime**.

Create two **public variables** that store the direction and speed. You can change the speed **value** based on your **playtesting**. Your direction **variable** should be equal to either **1f** or **-1f** to move the collectable left or right, forward or backwards, or up or down.

```
public float distanceToMove;

private Vector3 startingPosition;
private Vector3 endingPosition;

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

41 Use the following pseudocode to help you write your own Update function. Make sure you create variables that store the direction and speed.

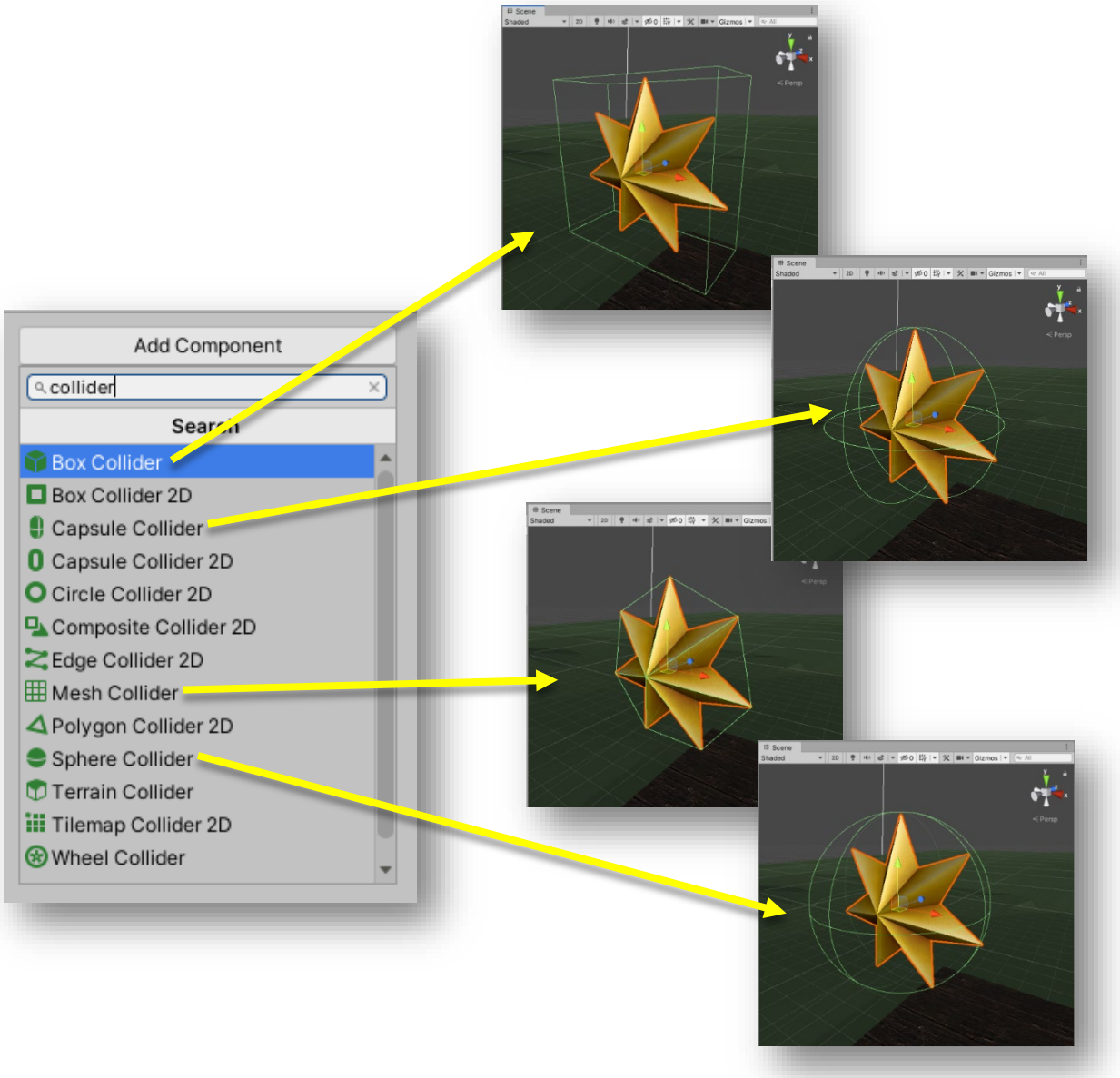


Sensei Stop

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

42 **Playtest** your game. Adjust the **speed**, **distance**, and **direction** to fit your game design.

43 We can now program the logic that lets our player collect the collectable object.

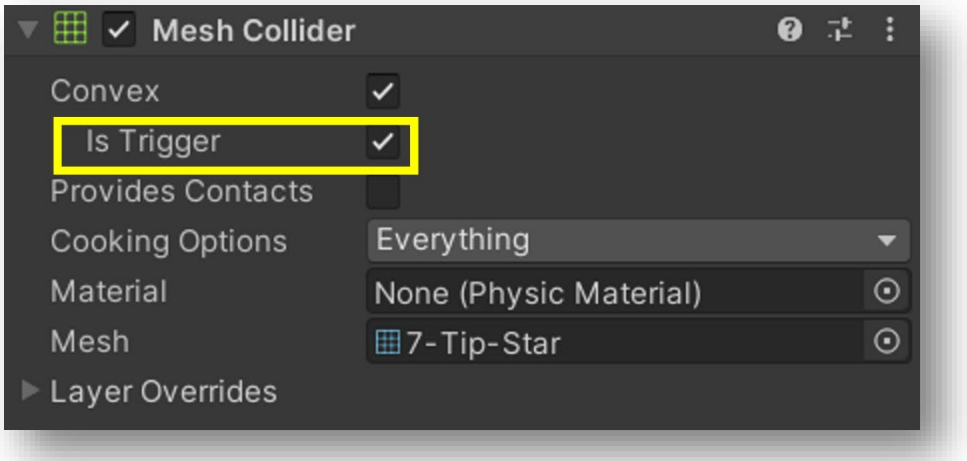


If your collectable does not already have a **collider component**, add one that makes sense for the shape of your object. Make sure you do not pick a **2D collider**!

44 Unity does its best to set the size and position of **colliders** automatically. A **convex mesh collider** will most closely mold to the object's shape.

All the **colliders** use the same **functions** to detect when **collision** with another object starts (**OnCollisionEnter**), continues (**OnCollisionStay**), and stops (**OnCollisionExit**).

If we want to check **collision** without using **physics**, if the collider is a trigger then we can use **OnTriggerEnter**, **OnTriggerStay**, and **OnTriggerExit**.



Sensei Stop

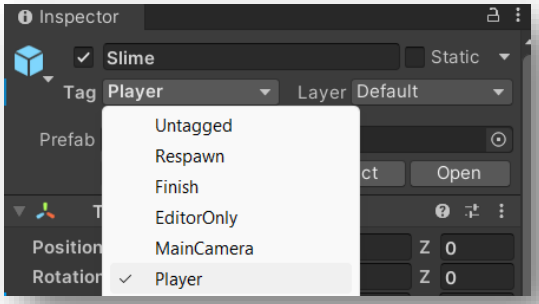
Use the OnTriggerEnter function to disable the collectable from the scene when the slime collides with it.

Hint: Use the **gameObject**'s **SetActive** function.

45 **Playtest** your game. Can any **object** destroy the collectable? Right now, we are checking to see if any other **object collided** with our collectable. We need to make sure that we destroy it only if the slime touches it and not any of the obstacles.

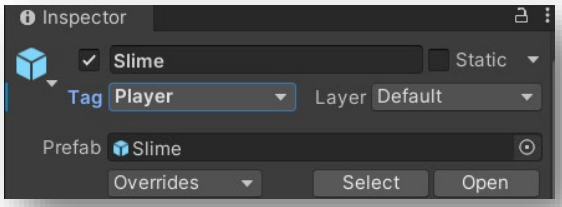
Look back at previous games and decide how you want to check to see if the slime is **colliding** with the collectable. You can set up **tags** or use the **other object**'s name.

Tags




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

Name

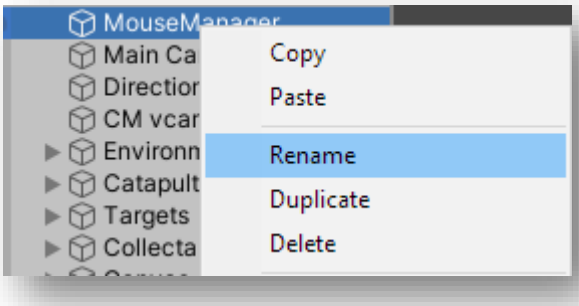


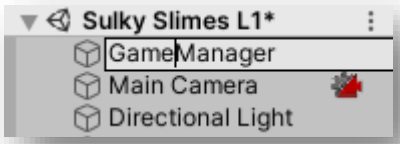

```
0 references
private void OnTriggerEnter(Collider other)
{
    if (other.name == "Slime")
    {
        gameObject.SetActive(false);
    }
}
```

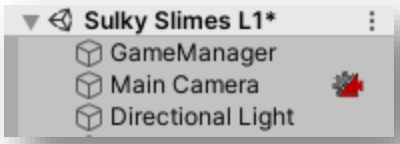
A Slime's Life

46 Now that the player has a goal, we need to program our lives system so they don't have infinite launches! As a game designer, you need to decide how you want to challenge your player. Providing a player with a clear goal with restrictions will give them a sense of accomplishment when they win!

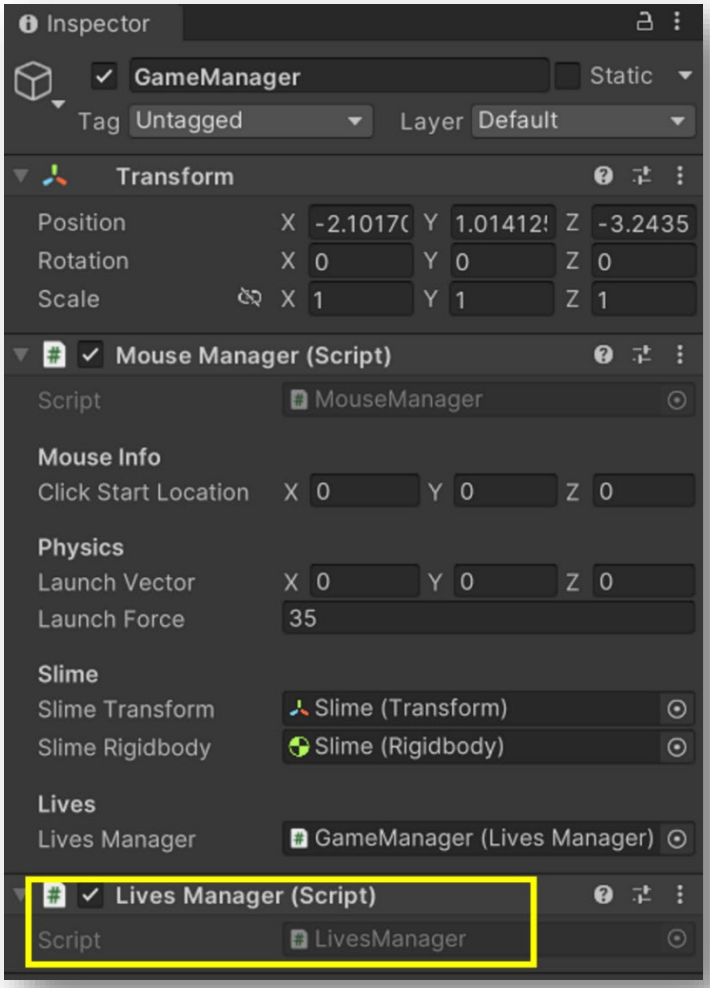
Let's first program the **logic** and then work on the **UI**. We have a game object named **MouseManager** that right now has only one job – managing the player's interaction with the slime. We can create another new **game object** or repurpose this one to be a **GameManager**! Remember that we used a **GameManager** in Codey Raceway.







47 Create a new **script** in the **Assets** tab, name it **LivesManager**, and attach it to the **GameManager object**.



48 Before we program the script, we need to think about what it needs to do. Make a flowchart or diagram to help you organize your ideas!

Sensei Stop

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

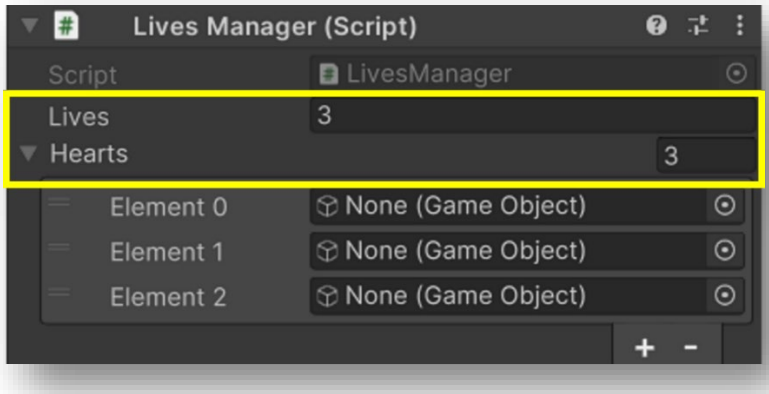
49 Open the LivesManager script and create two variables.

Create a **public int** variable named **lives**. This number will represent how many lives the player has.

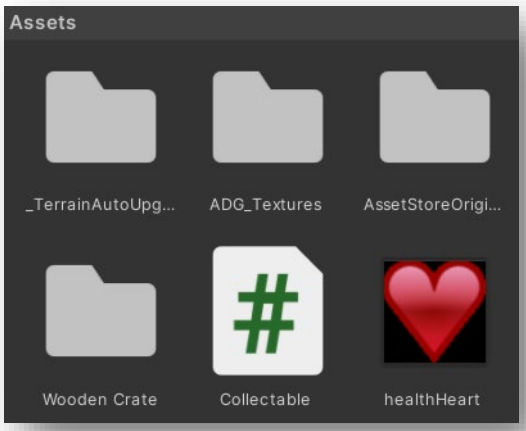
Create a **public GameObject[]** variable named hearts. This is an **array** that will contain **images** that will represent our lives.

```
1 reference
public class LivesManager : MonoBehaviour
{
    public int lives;
    public GameObject[] hearts;

    0 references
    void Start()
    {

    }

    0 references
    void Update()
    {

    }
}
```

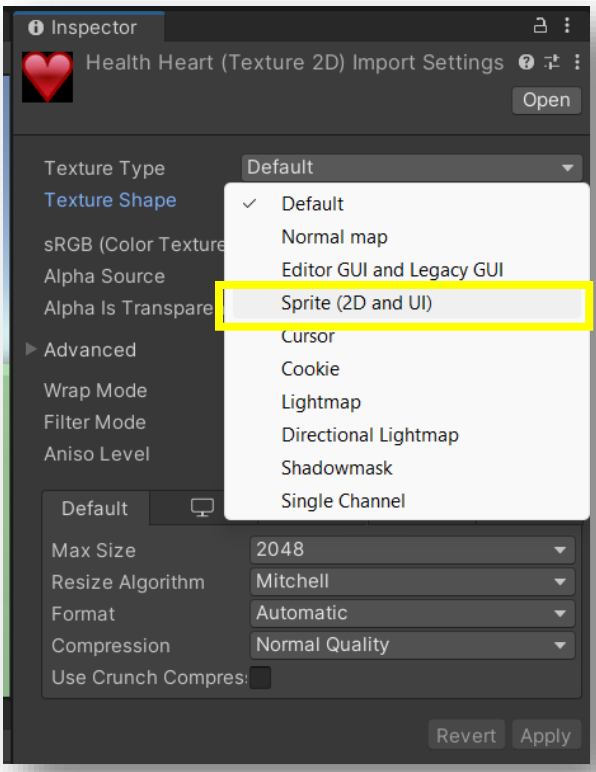
50 Save the **script** and look at the **inspector**. Set the **value** of **Lives** to **3** and the **size** of **Hearts** to **3**.



51 We now need three hearts to fill our three empty **elements**. Find your own image or use the **Activity 03 - healthHeart.png** file provided. Place the image in your Assets folder.



52 Open the **image asset** in the **Inspector**. Change its **Texture Type** to **Sprite (2D and UI)**.



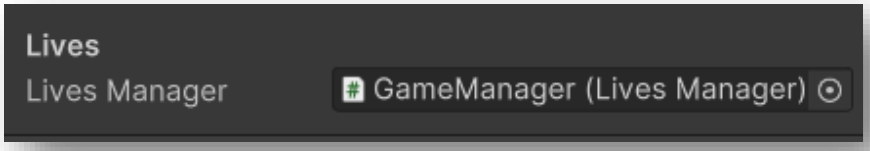
53 Create a **public void function** named **RemoveLife**. Inside this function, subtract one from the **lives variable** and **print** a message to the **console**

```
public class LivesManager : MonoBehaviour
{
    public int lives;
    public GameObject[] hearts;

    0 references
    public void RemoveLife()
    {
        lives -= 1;
        print("You lost a life! Lives: " + lives);
    }
}
```

54 We need to connect the **LivesManager** with the **MouseManager** so we can call the **RemoveLife function** when the player resets the slime.

Open the **MouseManager script** and add a **public LivesManager variable** named **livesManager**. Since we declared our **variable** to be the **LivesManager** type, the **Inspector** will only give us one option. **Unity** knows to only show us the types of **objects**, **scripts**, and **variables** that we want!

```
[Header("Lives")]
public LivesManager livesManager;
```



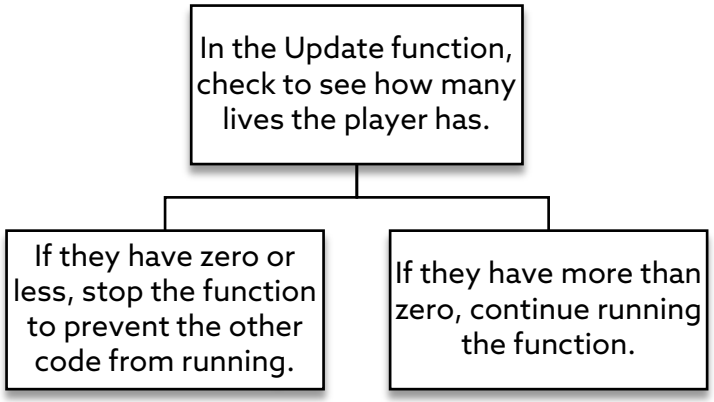
55 Now that we have a reference to our **LivesManager** inside our **MouseManager**, we need to call our **RemoveLife** function.

Sensei Stop

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

Hint: Look out for **console** messages to see exactly when your game runs the **RemoveLife function**.

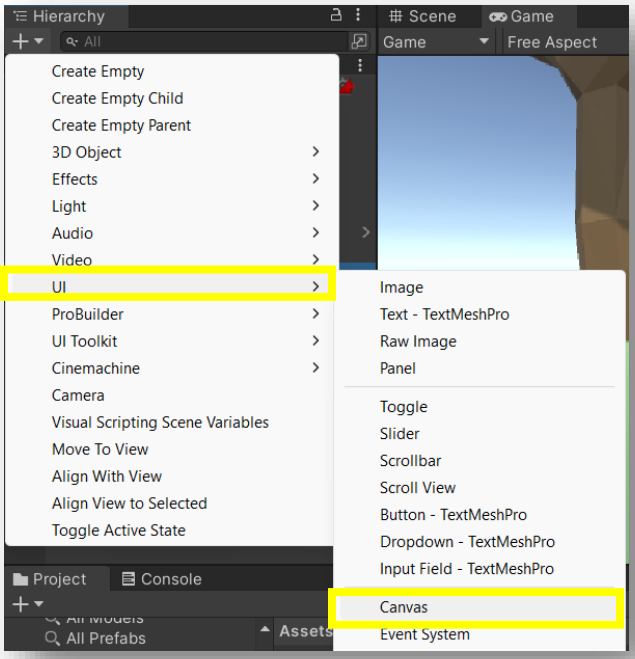
56 We now need to make sure the game ends if the player runs out of lives. We can use **pseudocode** to help us add to our **MouseManager**'s **Update function**.

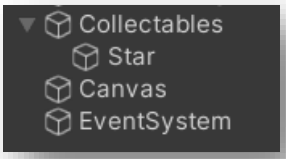



```
0 references
void Update()
{
    if (livesManager.lives < 0)
    {
        return;
    }

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

    if (Input.GetMouseButton(0))
    {
```

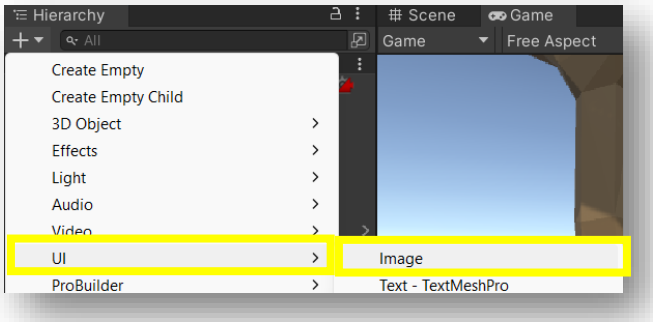
57 **Playtest** the game to see what happens after you run out of lives. The player is not able to interact with the game!

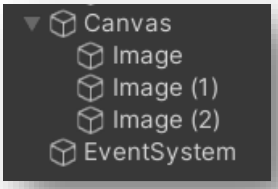
We now will let the player know how many lives they have left with a **UI**. First, add a **Canvas** object to your **scene**. This will also automatically add an **EventSystem** to the **hierarchy**.



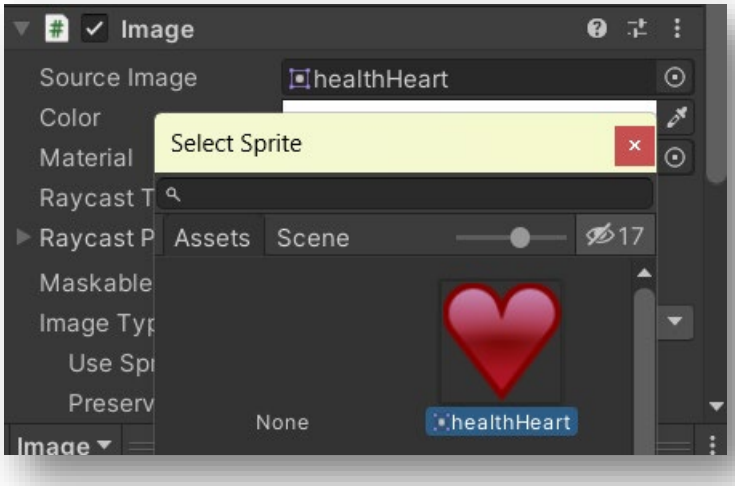


58 Add three **UI Images** objects to the **Canvas object**.



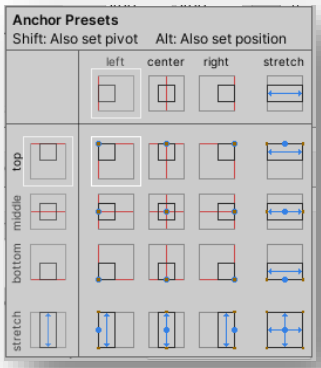


59 Change the three **Image object**'s **Source Image property** to your chosen **image**.

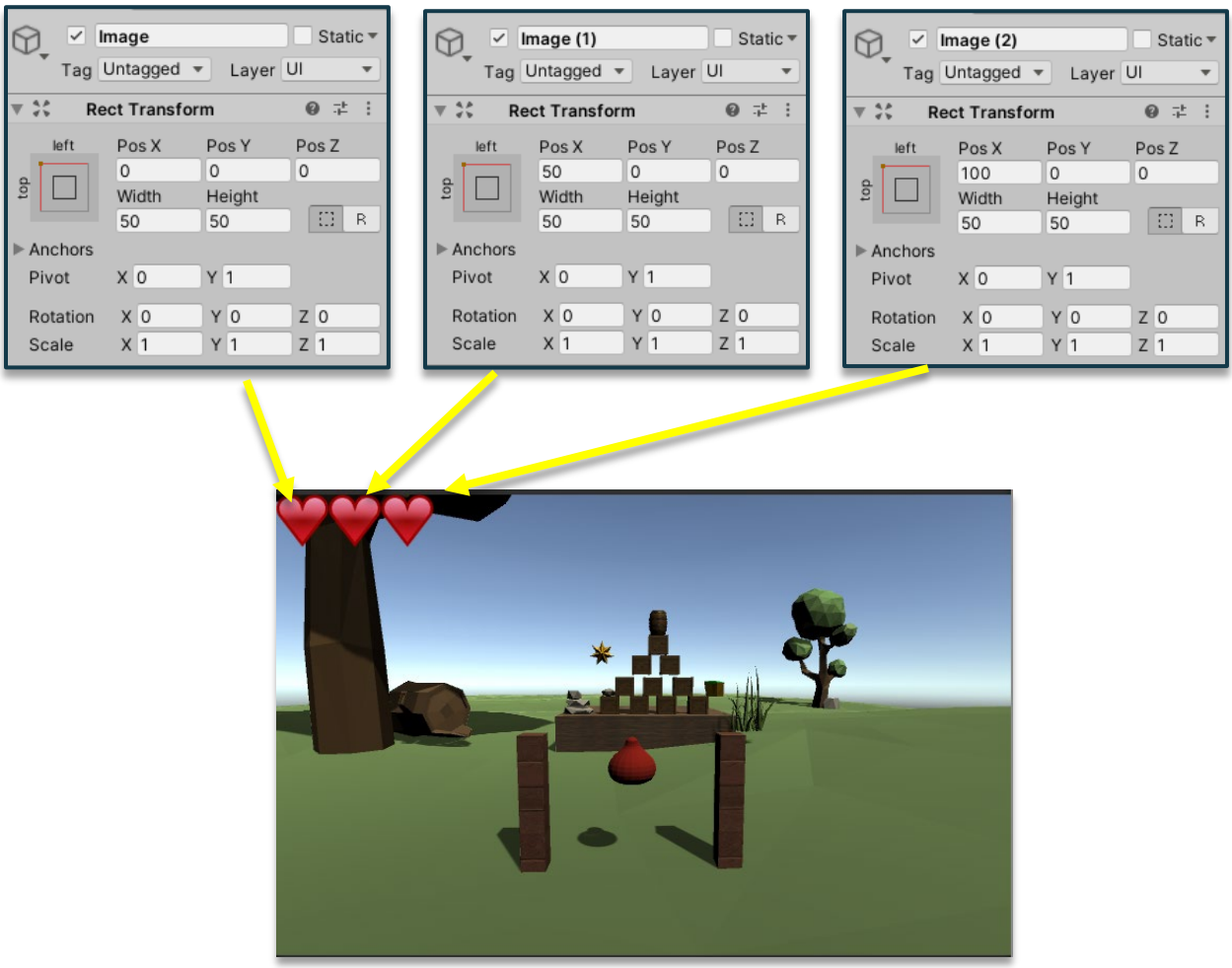


You should be able to see your **UI** in your **game** window.

60 We now need to reposition our three images. Look at its Rect Transform in the Inspector. Use the Anchor settings to place it somewhere on your canvas. Hold shift and alt when you click the location to pin it in place.

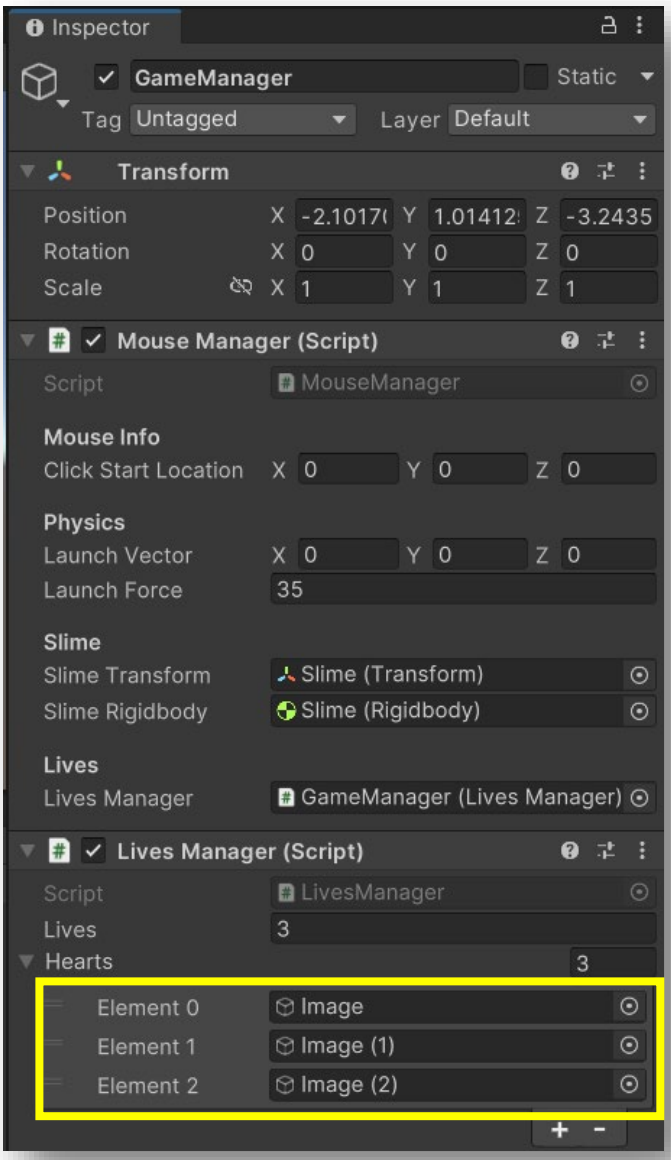


Adjust the width, height, and positions to align your three images. Make sure you keep them in the correct order!



61 With our UI set up, we can write the code to update it with how many lives the player has left.

First, let's connect our three images to the LivesManager script. Place the images in the correct Element of the Hearts property.



62 Since our three heart images are active when the game starts, we just need to disable a heart each time a player loses a life.

When the player loses a life, disable the heart object in the hearts array based on the current value of lives.

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

63 When the player runs out of lives, we want to reload the scene to let them try again.

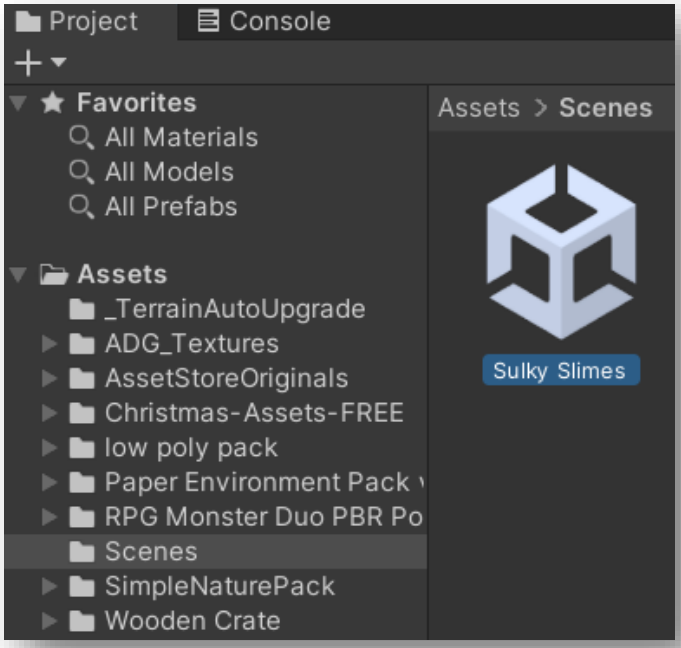
Sensei Stop

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

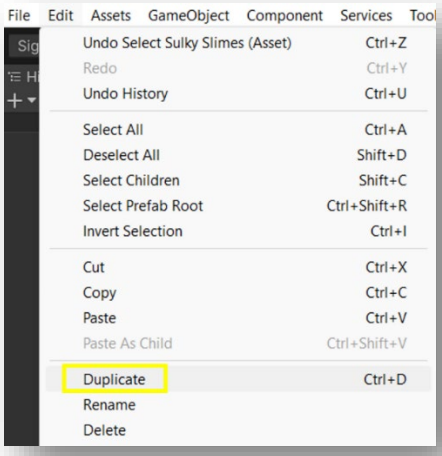
Playtest your game to make sure your game reloads when the lives run out! To speed up the playtest, you can right click however many times you have lives.

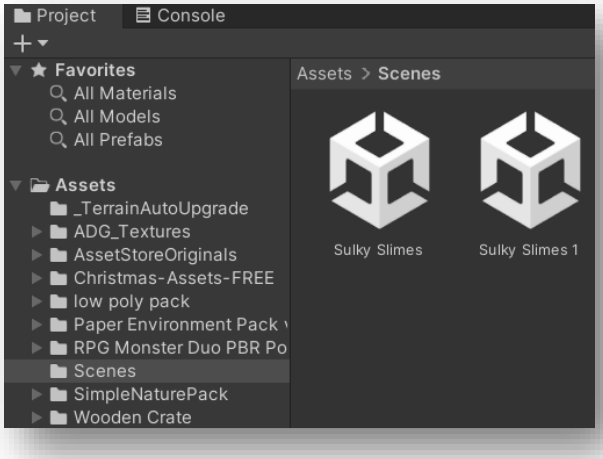
A Change of Scenery

64 As a reward for getting the collectable, we can load a second level for the player. Save your current **scene**. Instead of starting from scratch, we can use our first scene as a starting point. Select the scene in the **Project** window.

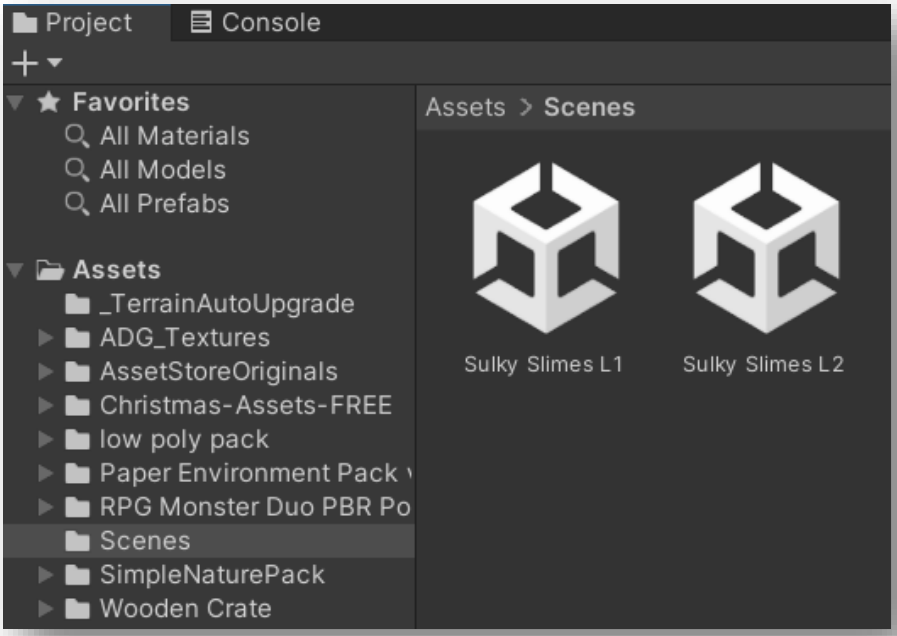


Press **Control + D** or go to **Edit** -> **Duplicate**

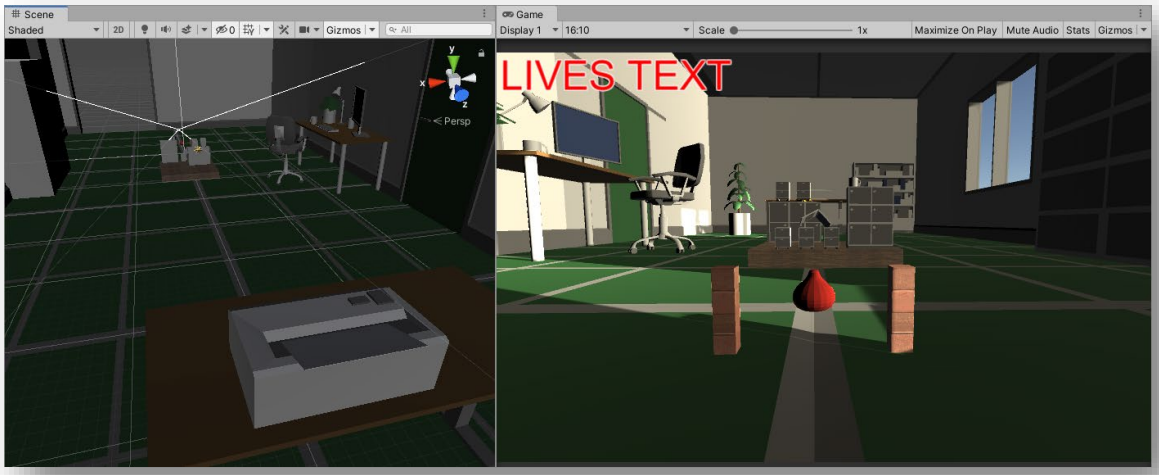




65 Rename your **scenes** so it is easier to tell them apart.

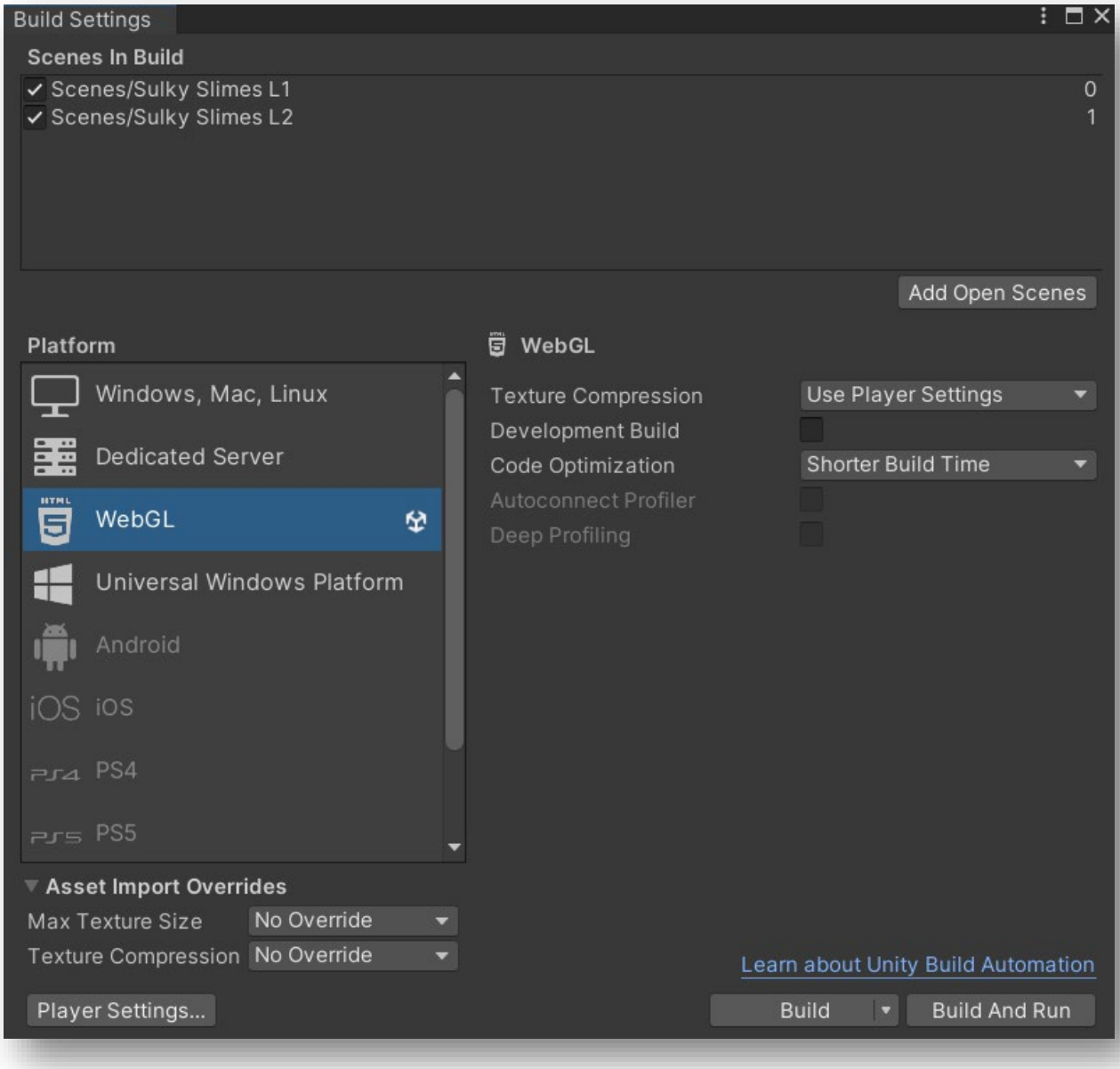


66 Open the level 2 **scene** and make it unique. Change the **environment**, the **collectables**, or the **targets**. Be creative, like recreating a giant version of your Code Ninjas Dojo!



67 **Playtest** your game and make sure everything is working properly. Since we changed only the **models**, all our **logic** still works. We just need to connect our **scenes** so we can dynamically load the next level!

First, go to **File** -> **Build Settings** and add your new **scene** to the "Scenes in Build" menu.



68 When the player collects our collectable, we want to load the next scene. Open the **Collectable script** and add using **UnityEngine.SceneManagement** to the top.

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

69 We also need to know which **scene** to load based on the number assigned in the "Scenes In Build" menu.

In the Collectable **script**, create a **public int variable** named **sceneNumber**. Now that we have a few different **public variables**, add **Headers** to help you stay organized.

```
[Header("Movement Values")]
public float distanceToMove;

private Vector3 startingPosition;
private Vector3 endingPosition;

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

[Header("Scene to Load")]
public int sceneNumber;
```

70 In the Collectable **script**, create a new **private void function** named **LoadNextScene** that takes no **parameters**.

```
0 references
private void LoadNextScene()
{

}
```

71 Use the **SceneManager**'s **LoadScene function** to load the **scene** associated with the **sceneNumber variable**.

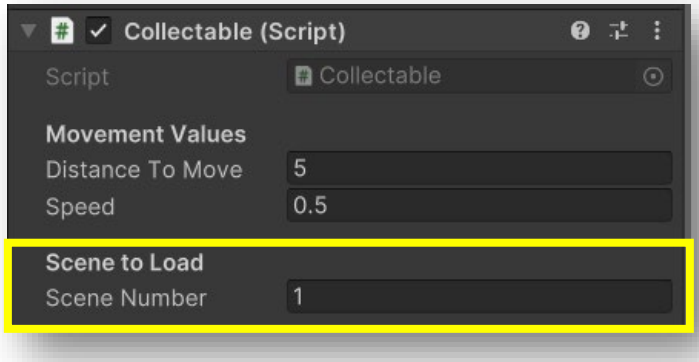
```
0 references
private void LoadNextScene()
{
    SceneManager.LoadScene(sceneNumber);
}
```

72 We want to give the player some time to enjoy their victory.

In the Collectable **script**'s **OnTriggerEnter function**, **Invoke** the **LoadNextScene function** after 2 seconds.

```
0 references
private void OnTriggerEnter(Collider other)
{
    if (other.name == "Slime")
    {
        gameObject.SetActive(false);
        Invoke("LoadNextScene", 2f);
    }
}

0 references
private void LoadNextScene()
{
    SceneManager.LoadScene(sceneNumber);
}
```

73 In the **Inspector**, give **Scene Number** a **value** of 1 to load the corresponding **scene**.



74 **Playtest** your game and see what happens after you get the collectable. The player is transported to the second **scene**!

Project Submission and Reflection

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

Once you feel like you have a good product that represents your vision of the game, have a 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 Code Sensei and fill out the reflection section of your Ninja Planning Document.

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

Ninja Planning Document

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