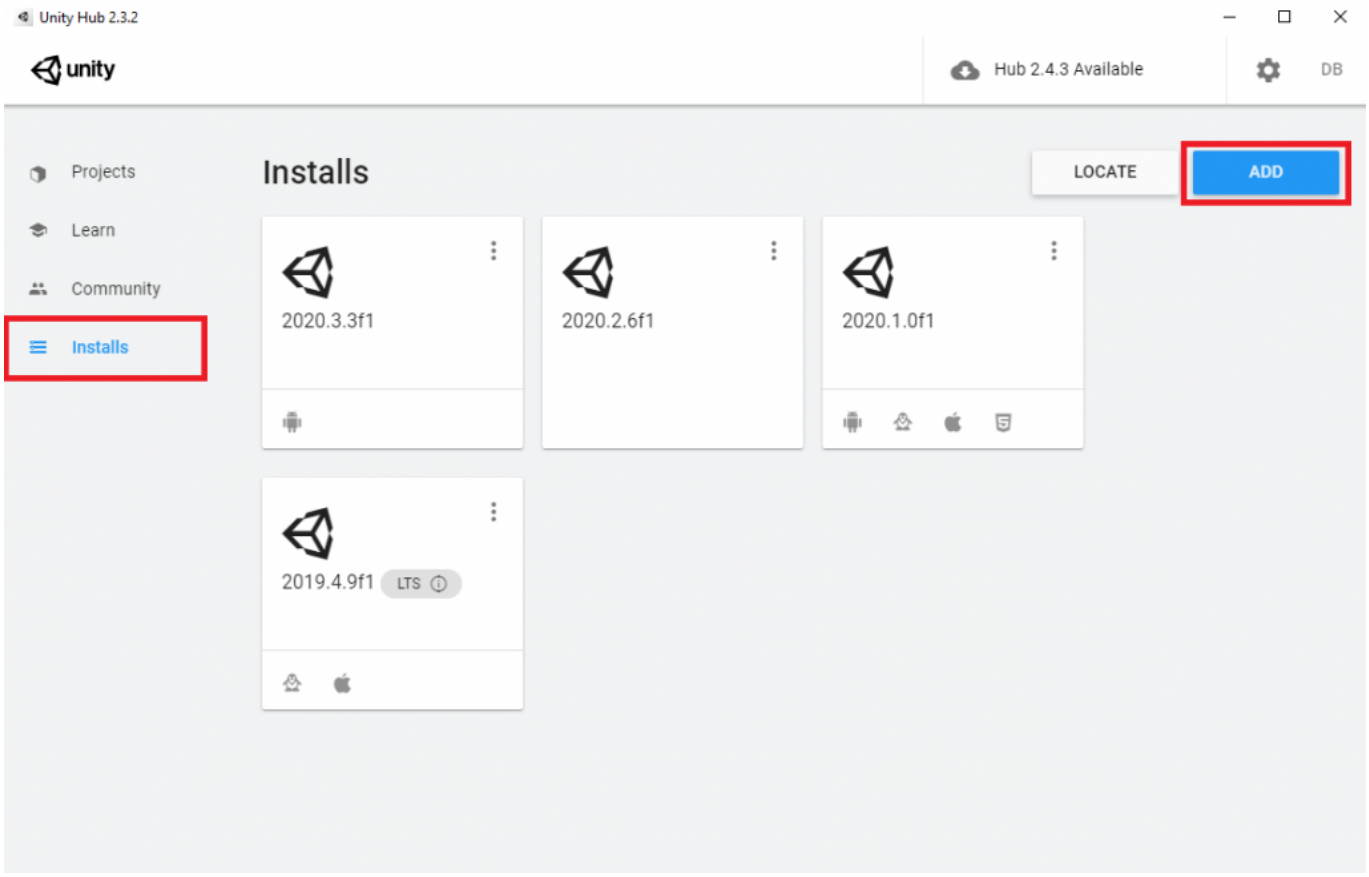
Course Updated to Unity 2020 LTS

For this course, we've updated the project files to Unity version **2020.3 (LTS)**. This is a long term support version, which is recommended by Unity. LTS versions get released each year and are stable foundation for you to build your projects from.

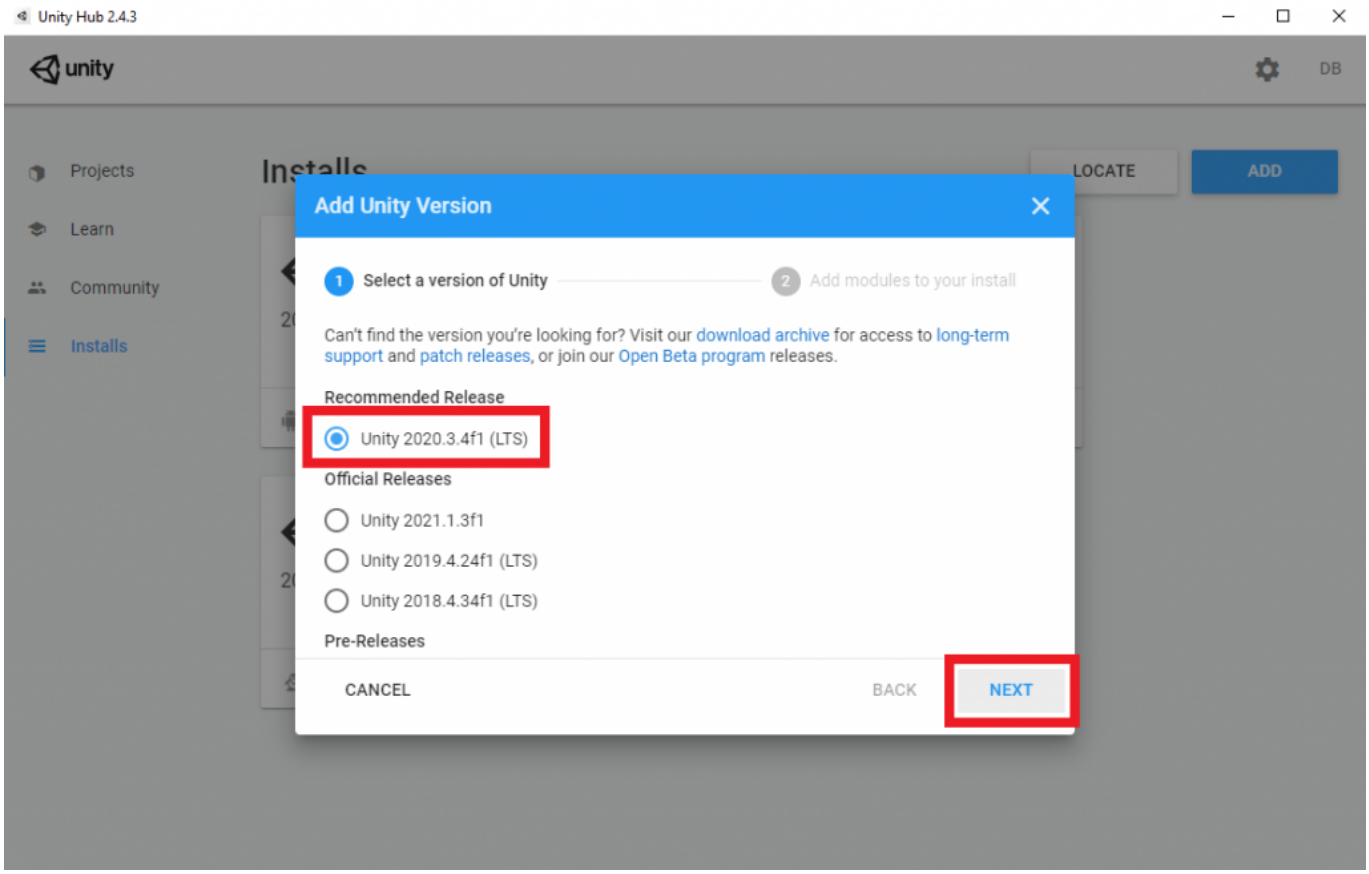
For students, this means consistent project files across all courses – no matter when they were created. No more downloading different Unity versions, or opening old projects filled with errors.

How to Install 2020 LTS

First, open up your Unity Hub and navigate over to the **Installs** screen. Then, click on the blue **Add** button.



This will open up a smaller window with a list of Unity versions. Under the *Recommended Release* header, we want to select **Unity 2020.3 (LTS)**. From here, click **Next** and follow the install process.

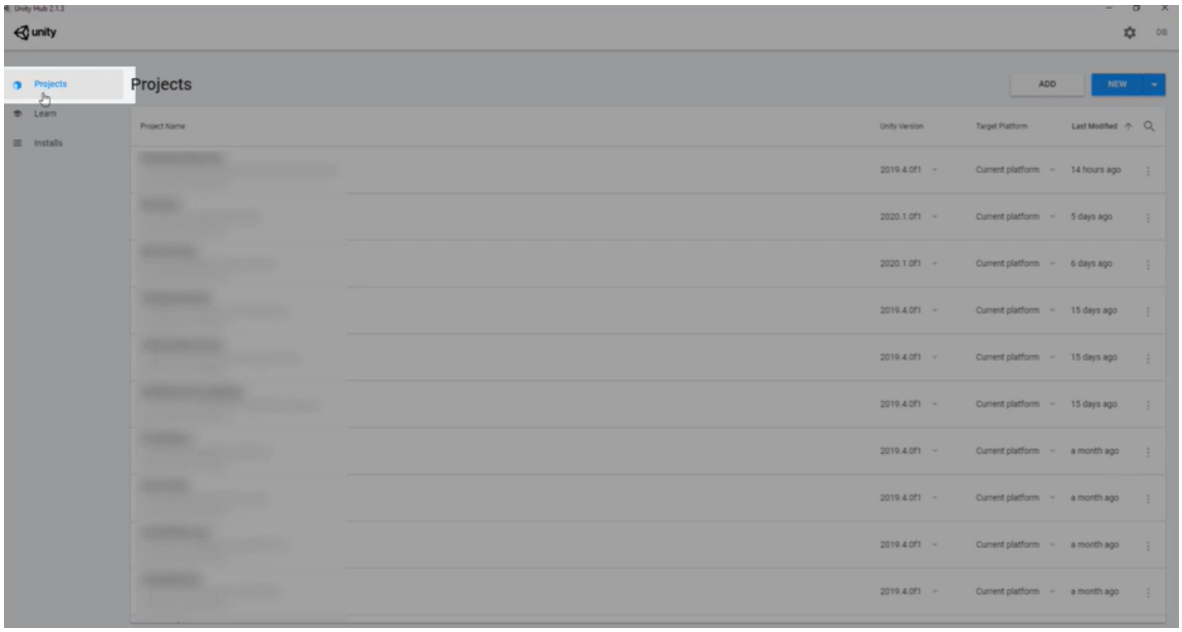


In this lesson, we're going to be setting up the **2D RogueLike Game** in Unity.

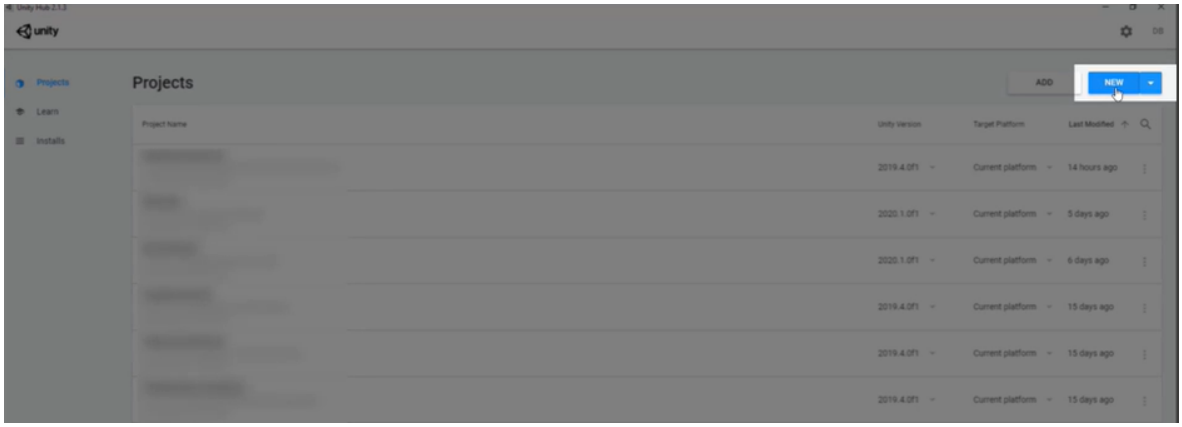
You need to have the latest version of **Unity Hub** installed if you haven't already. Download Unity Hub: https://unity3d.com/get-unity/download

Creating Unity Project

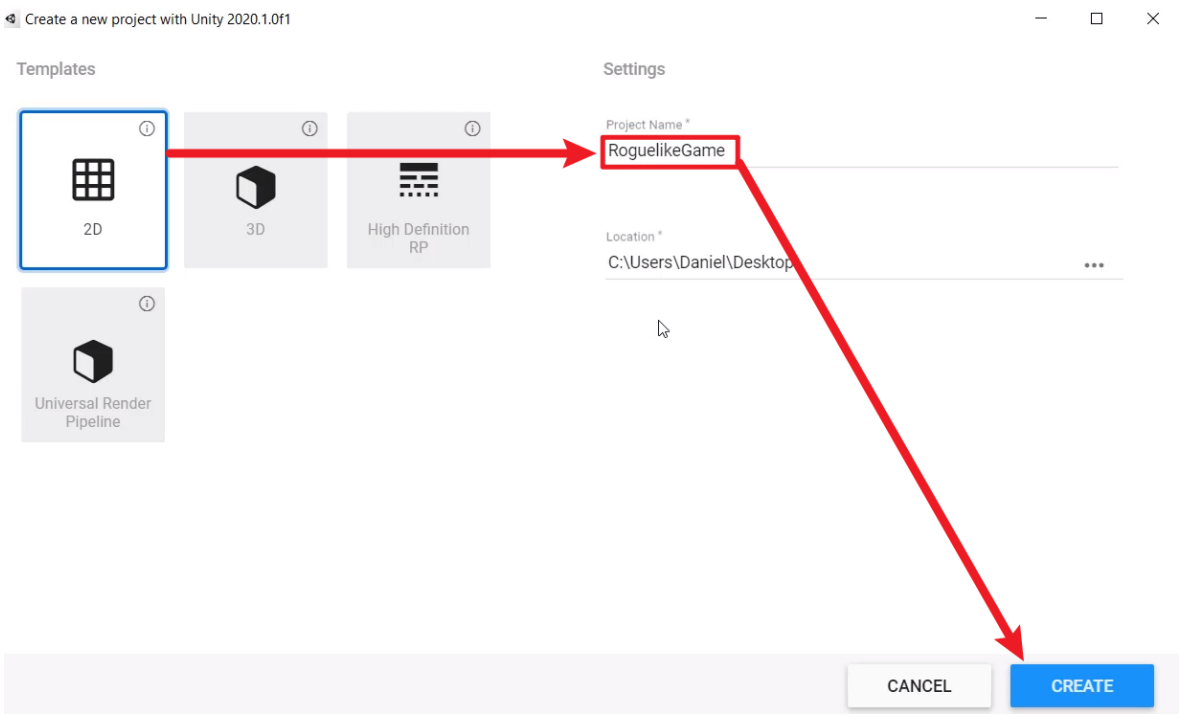
Open up Unity hub, and click on the **Projects** tab:



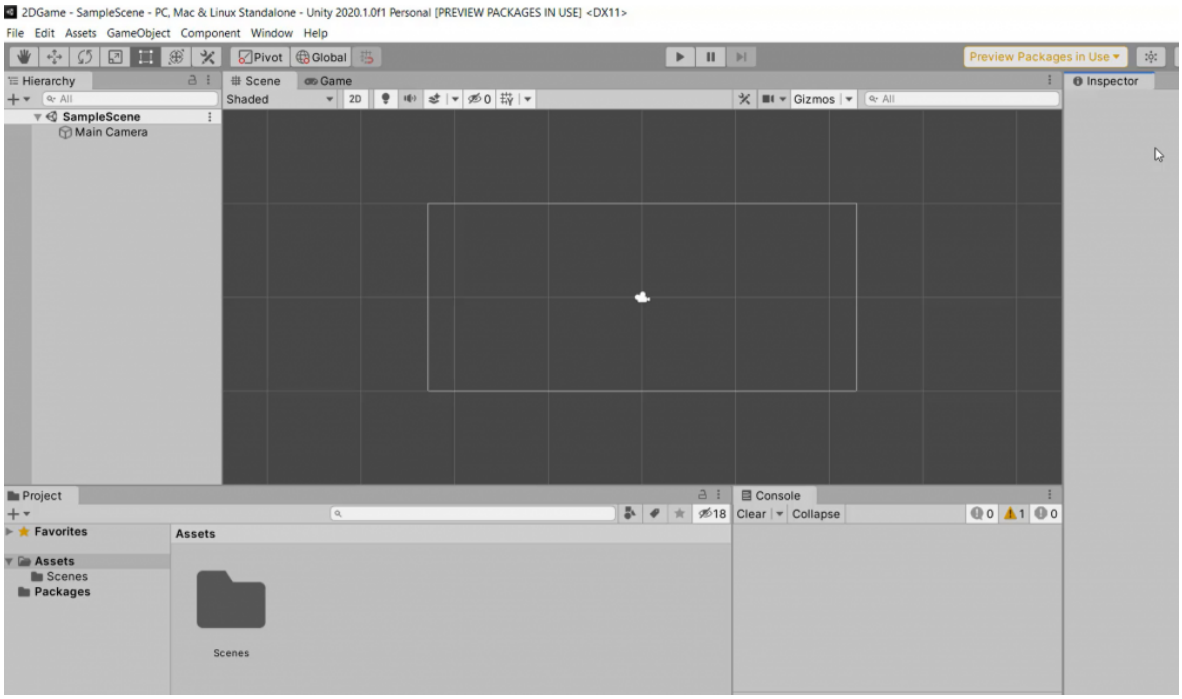
... and click on the **New** button**:**



Select the **2D template**, and set the project's name (e.g. **RoguelikeGame**) and location. Then click on the '**Create**' button.

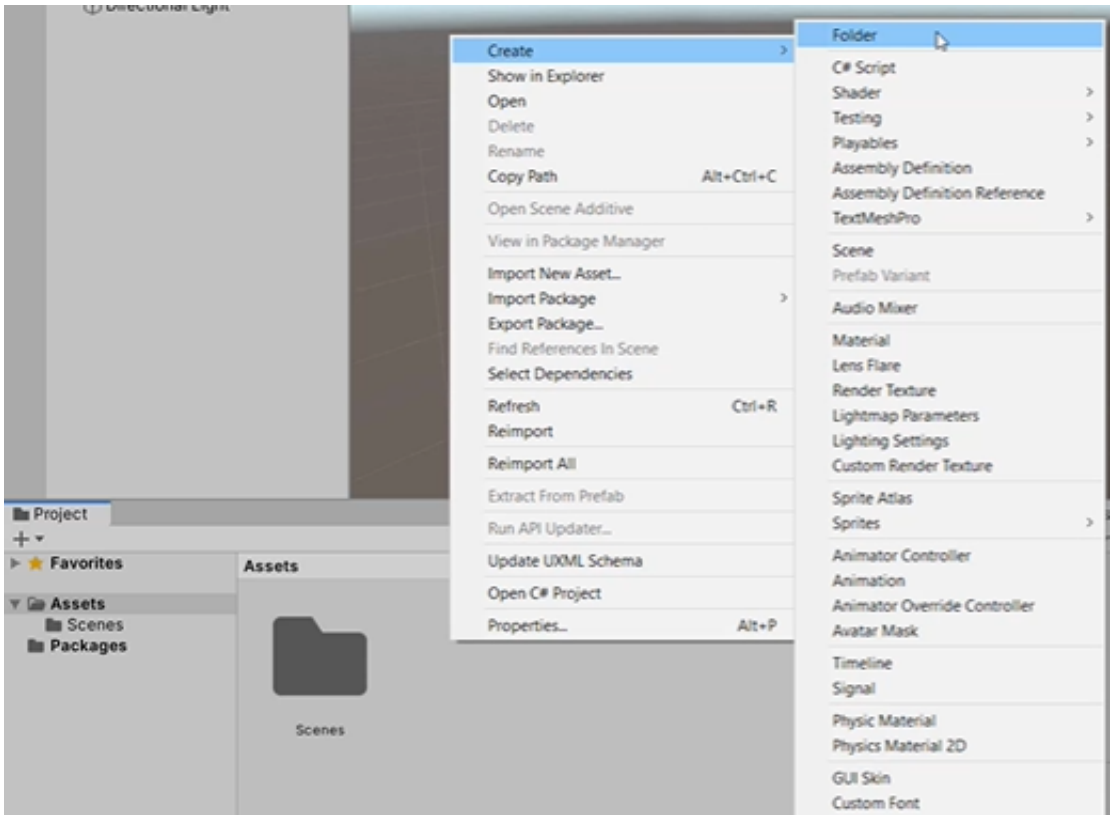


We now have an empty **2D Scene** open inside of our brand new Unity project.

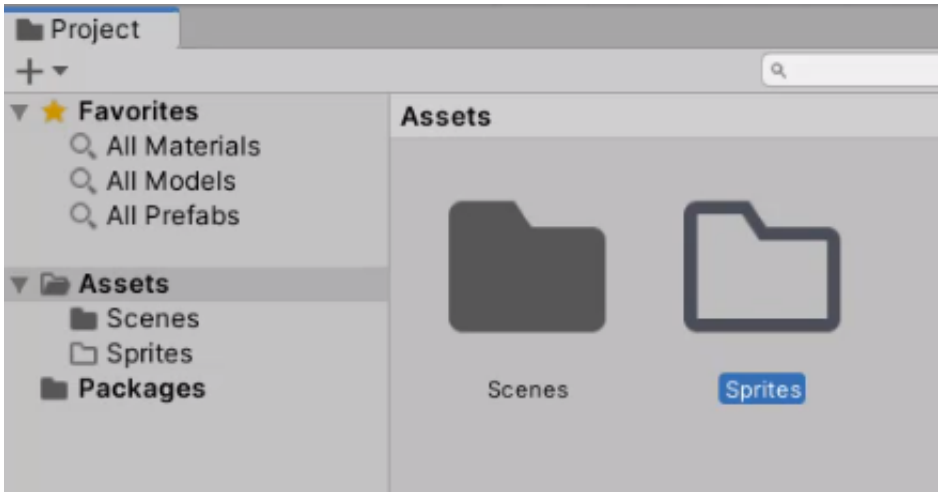


Importing Sprites

First of all, we're going to **create new folders** by **Right-clicking** on the project panel > **Create > Folder**.



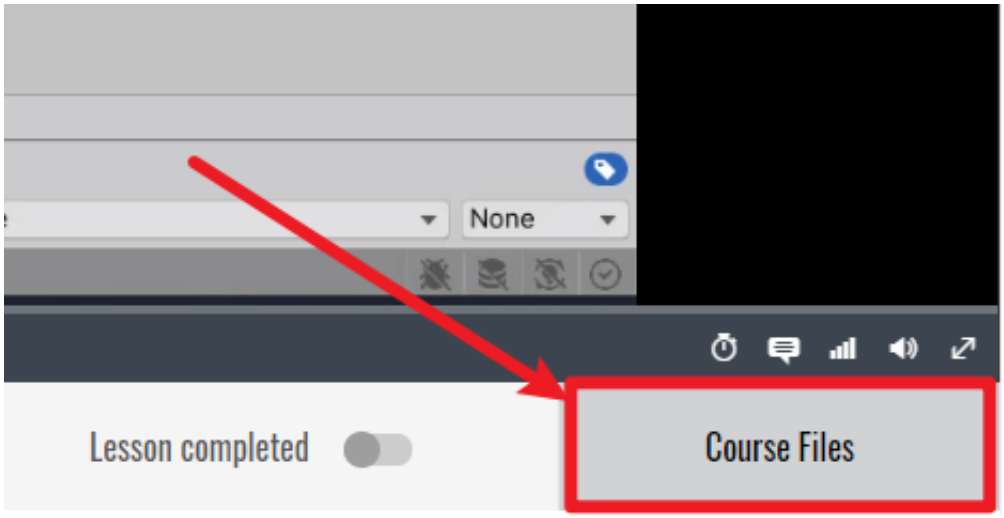
Then we're going to **name** the folder "**Sprites**". This folder is going to store all of our sprites for the game:



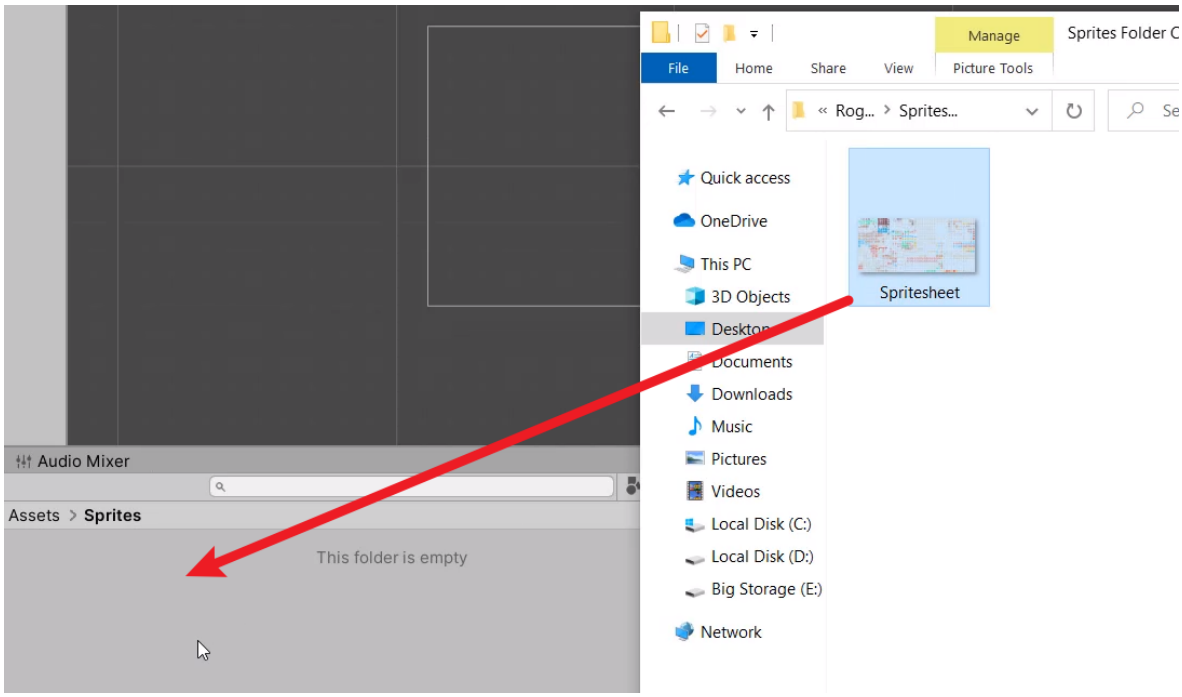
And we're going to create one for **Scripts** and **Prefabs** as well.

Importing Sprites

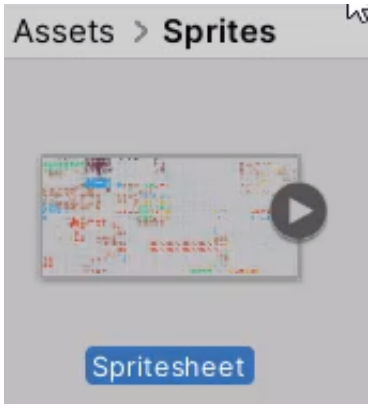
Next, we're going to download the **Assets.zip** file from the **Course Files** tab.



Then we're going to select the file inside the extracted **.zip** file and **drag** that into the **Sprites** folder.



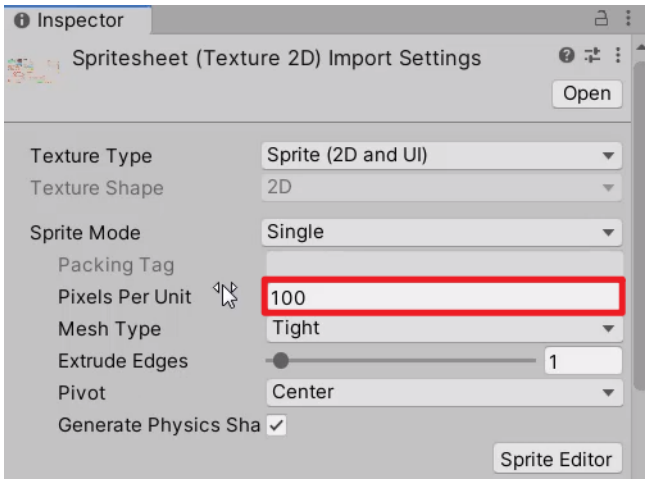
Now we've successfully imported the file into our project.

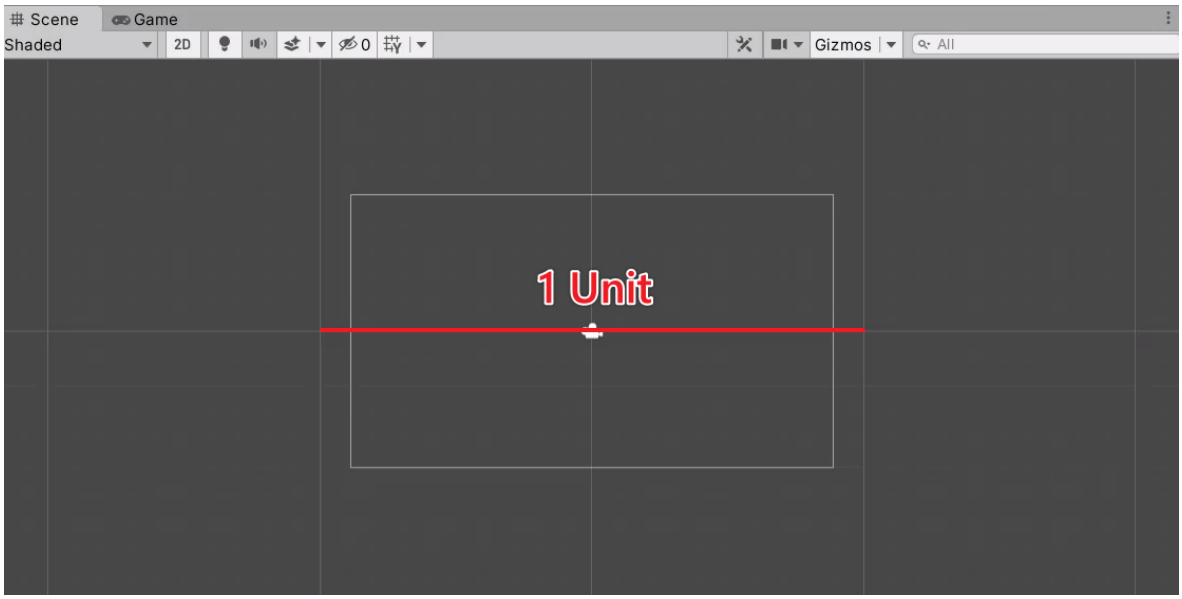


Note that this is called "**Spritesheet**", instead of "Sprite". Unlike a typical sprite that has an individual image for each object, this contains a table of all the different sprites, which can be cut into different sections.

Changing Pixels Per Unit

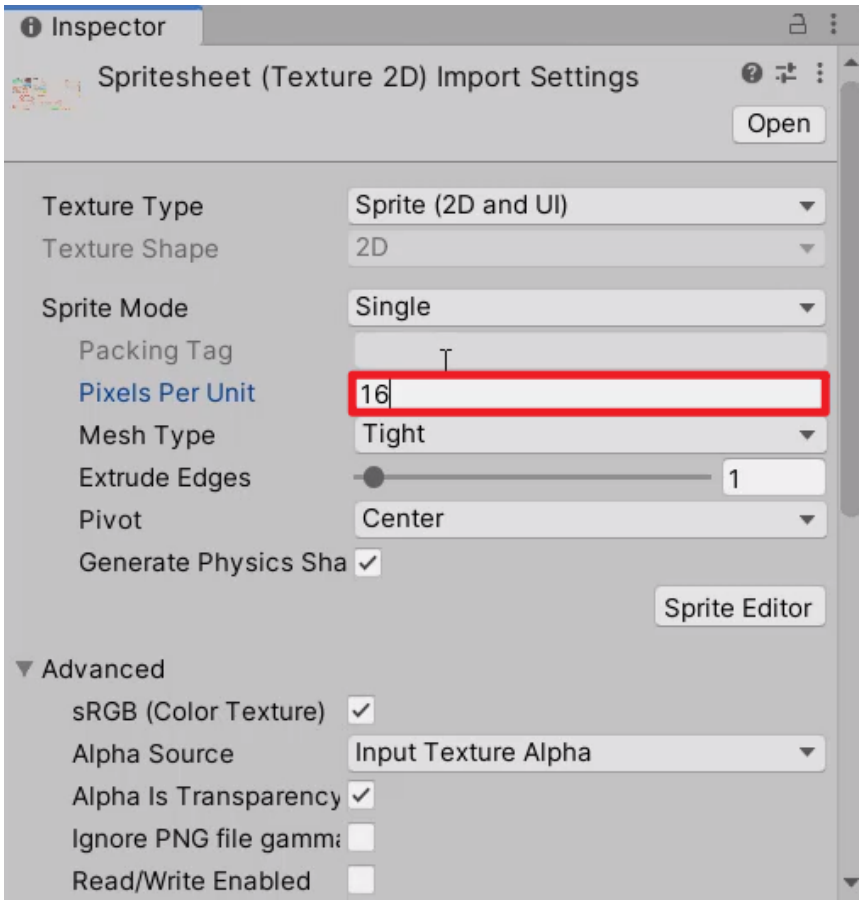
When you click on the sprite, you'll see that the **Pixels Per Unit** is set to **100** by default. This means that every **100** pixels of the sprite will fit in **1** Unity unit.

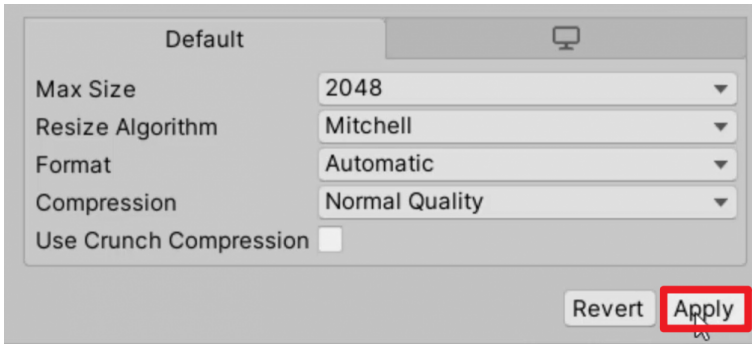




(Each of the grid tiles is half a unit long in Unity.)

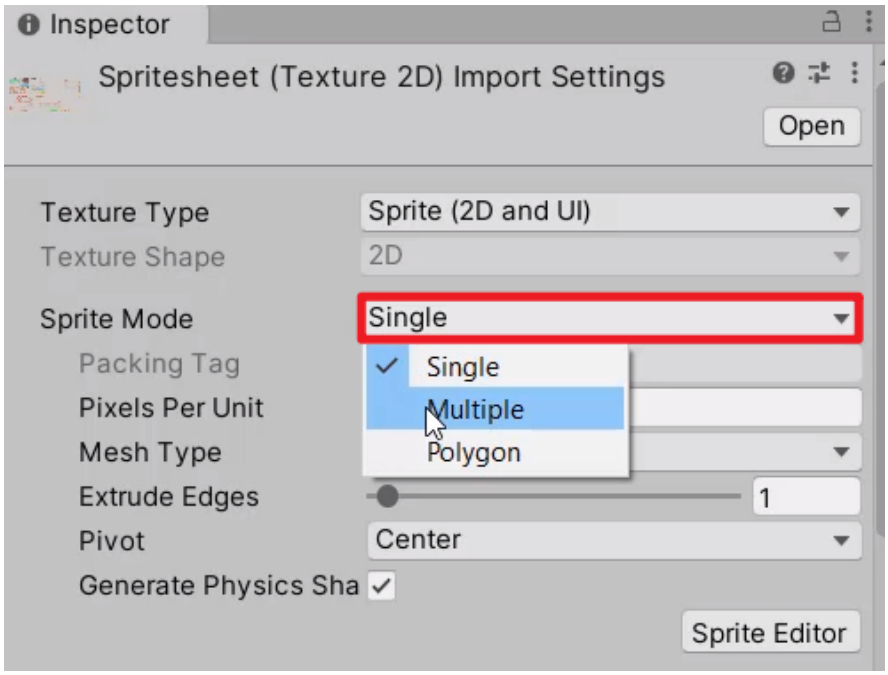
Since each component of the sprite is equal to **16 x 16 pixels**, we're going to change their **Pixels Per Unit** value to be 16, and click on the **Apply** button.



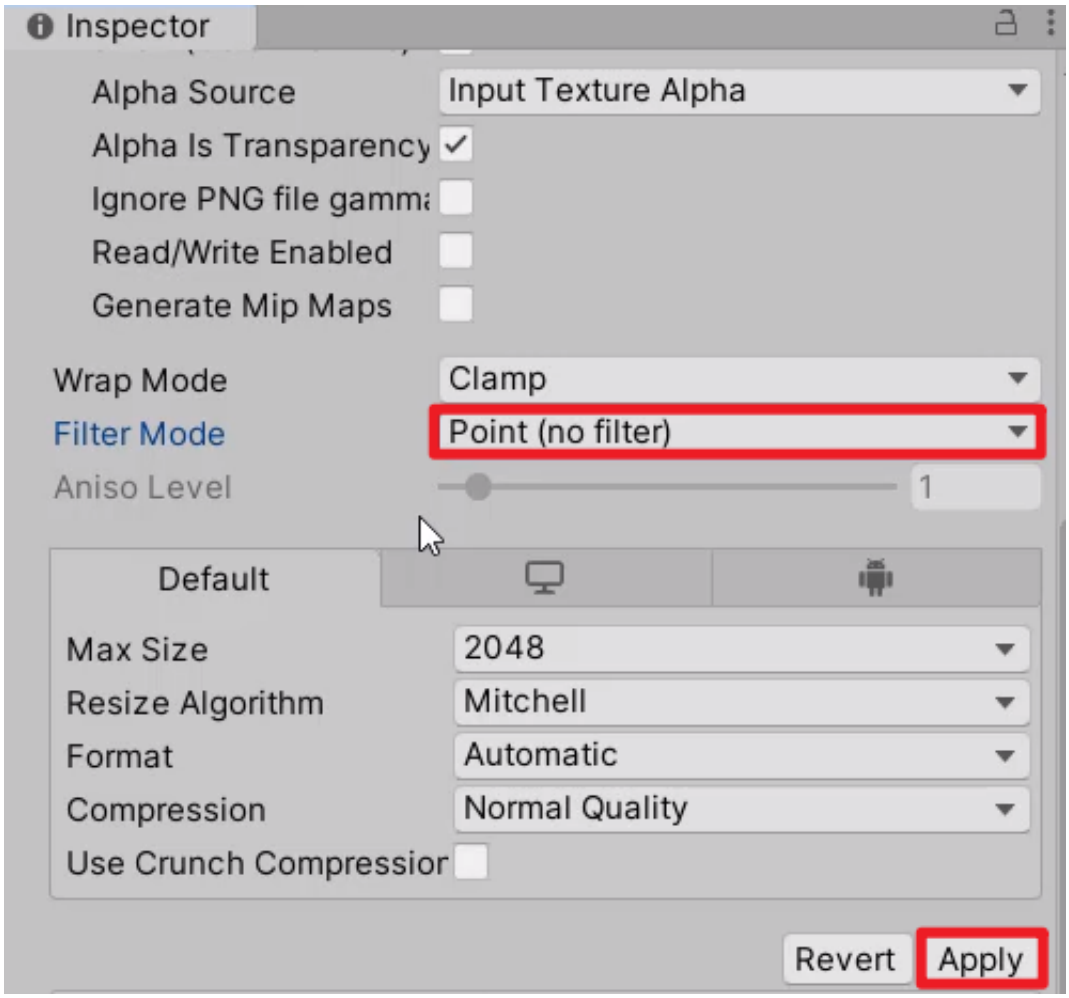


Slicing SpriteSheet

Now what we need to do is changing the **Sprite Mode** from **Single** to **Multiple** so that the sprite can be divided into multiple sprites:

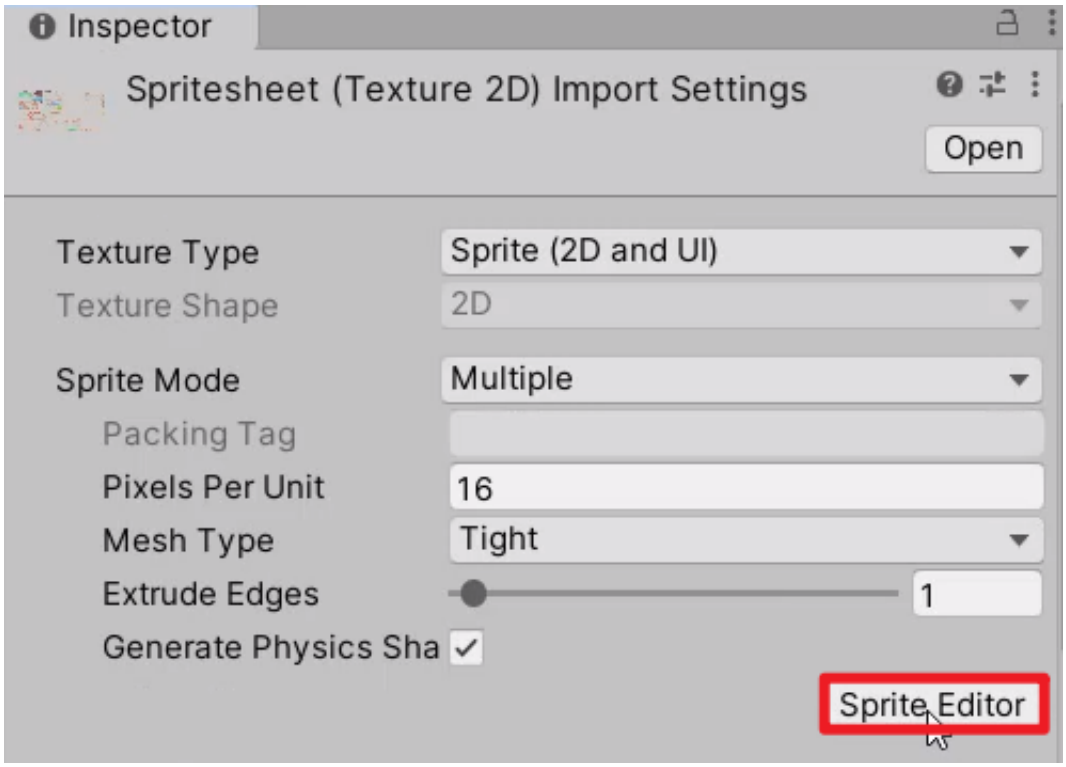


And we'll also change the **Filter Mode** to **Point (no filter)** so that we can see each individual pixel without any filtering or blurring. Normally, if you were to have an image on the screen, you would want this to be **Bilinear** to make it look a bit smoother.

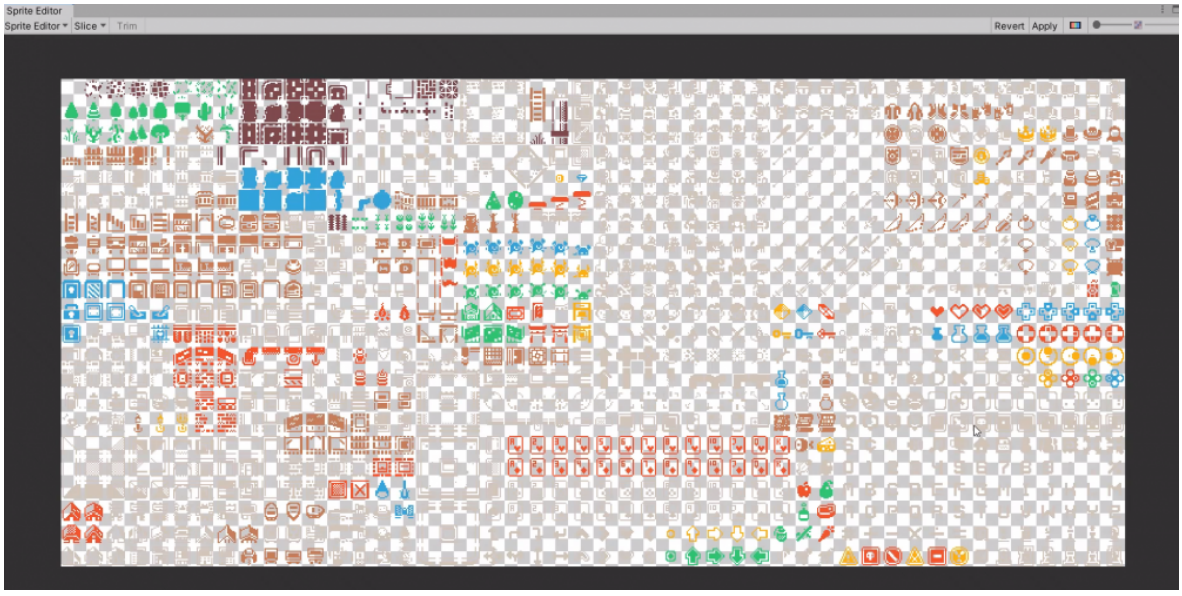


Sprite Editor

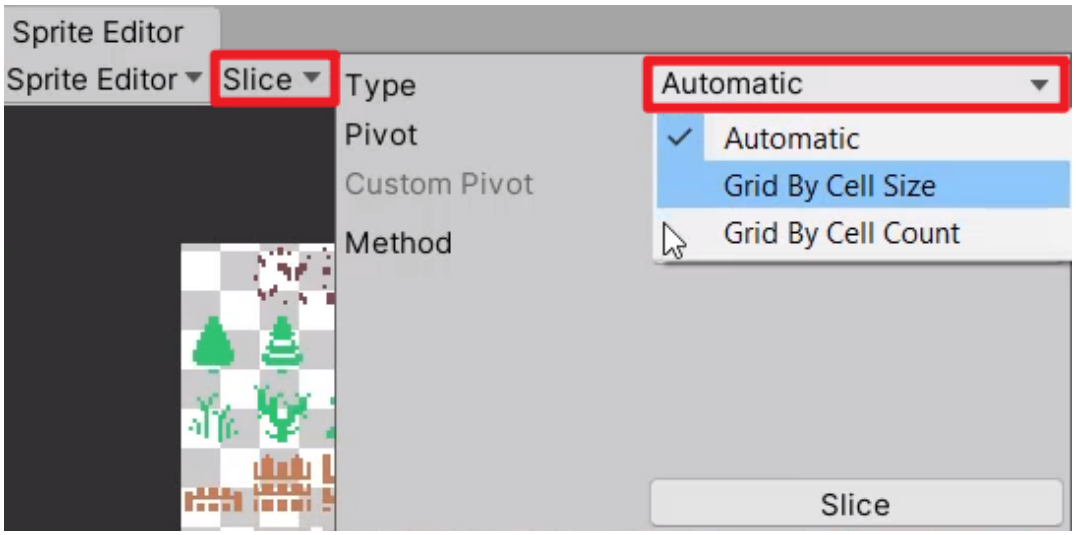
If you scroll up to the top, you can click on the **Sprite Editor** button.



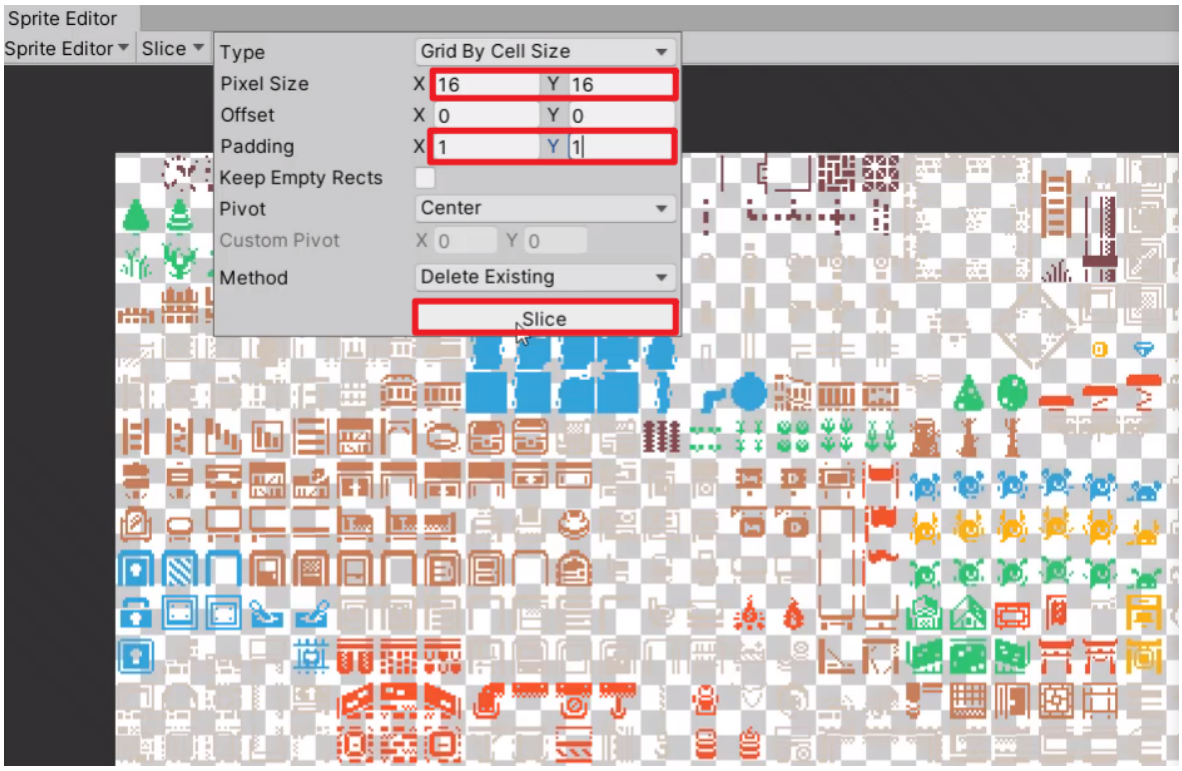
This is going to open up the **Sprite Editor** window. Inside this window, we can define each of the individual Sprite by either manually dragging or using the built-in **Slice** feature.



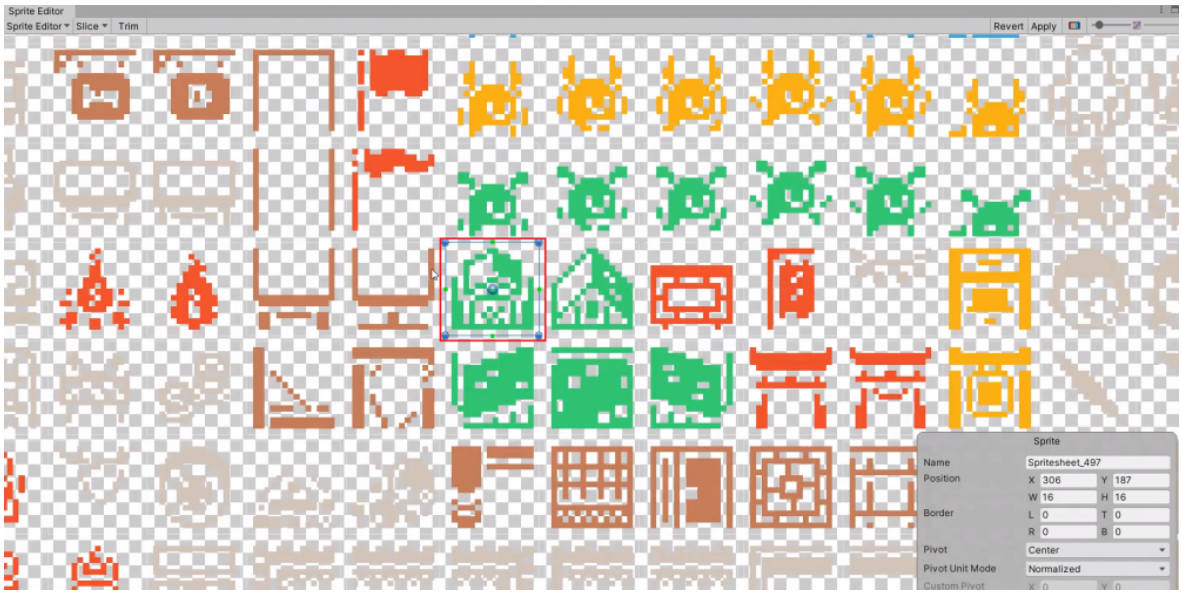
To use that feature, you can click on the **Slice** button at the top left, and choose the **Type** to be **Grid By Cell Size**.

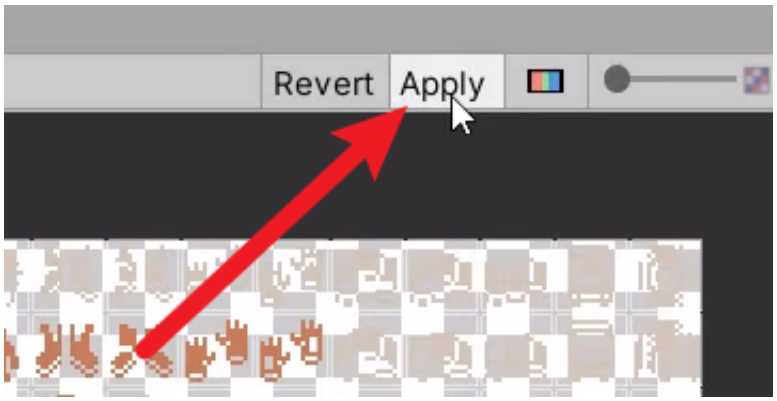


We then need to enter the **size** for each of the Sprites, which is going to be **16 x 16**. Since the sprites are separated by one pixel on each side, we need to enter **1 x 1** on the **padding**, and click **Slice**.

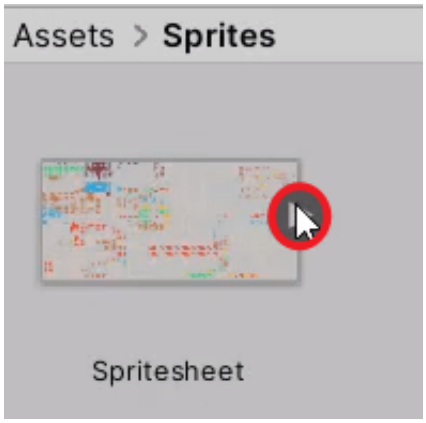


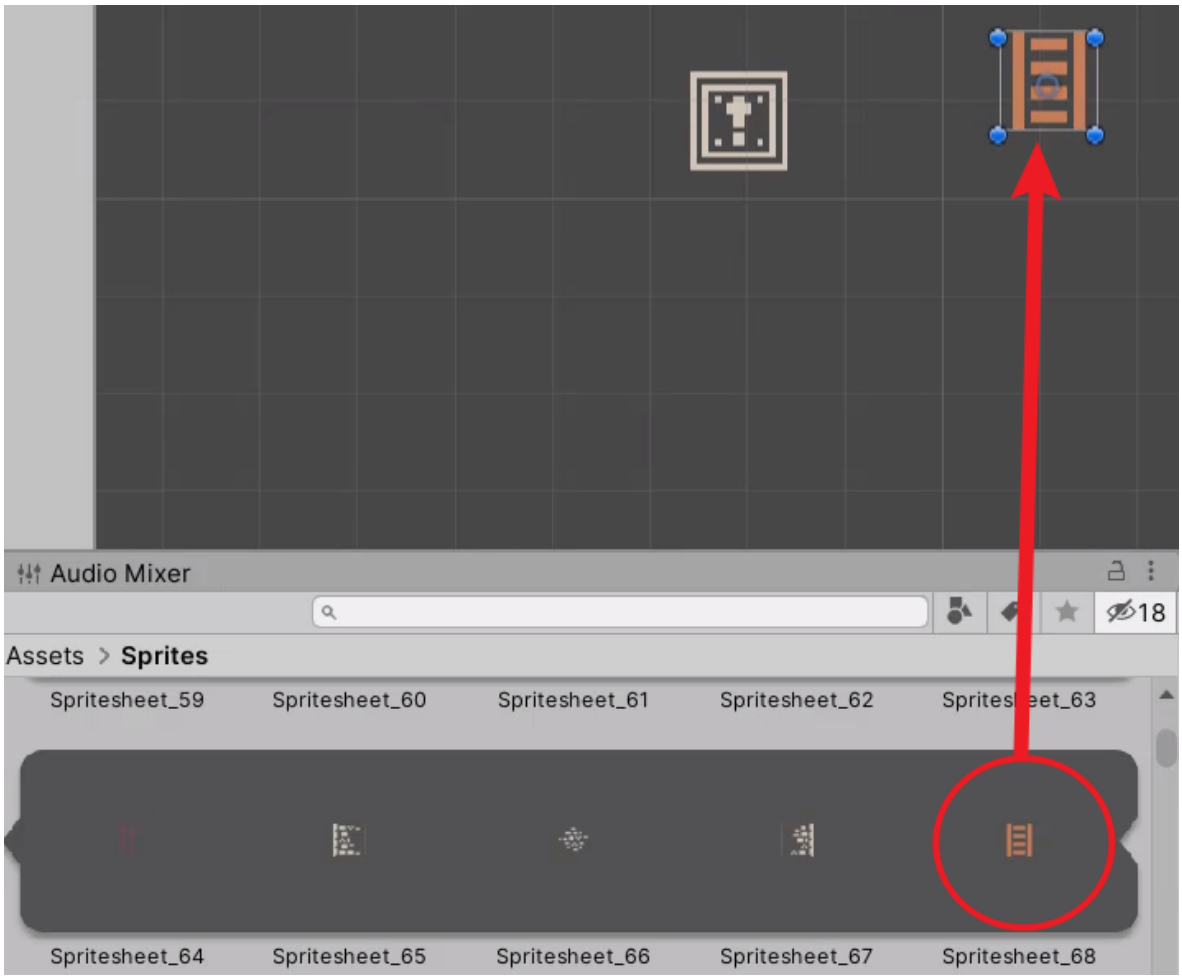
You can now see that the entire sprite sheet is divided up, each separated by **1 x 1** pixel **padding** around them. Make sure to click on the **Apply** button at the top to save the changes.





Once that's done, you can now open up the **Spritesheet** and drag in any sprite that you want into the **Scene** view**.**

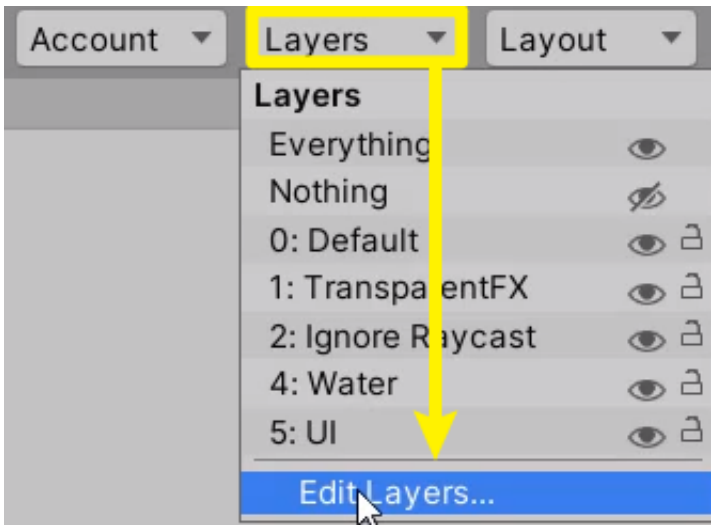




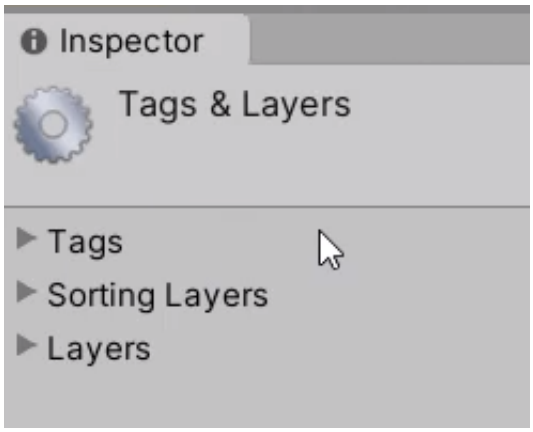
Setting Up Sorting Layers

In Unity, every Sprite has a default **Sorting layer**, and it helps us determine which sprites are going to be rendered on top of others.

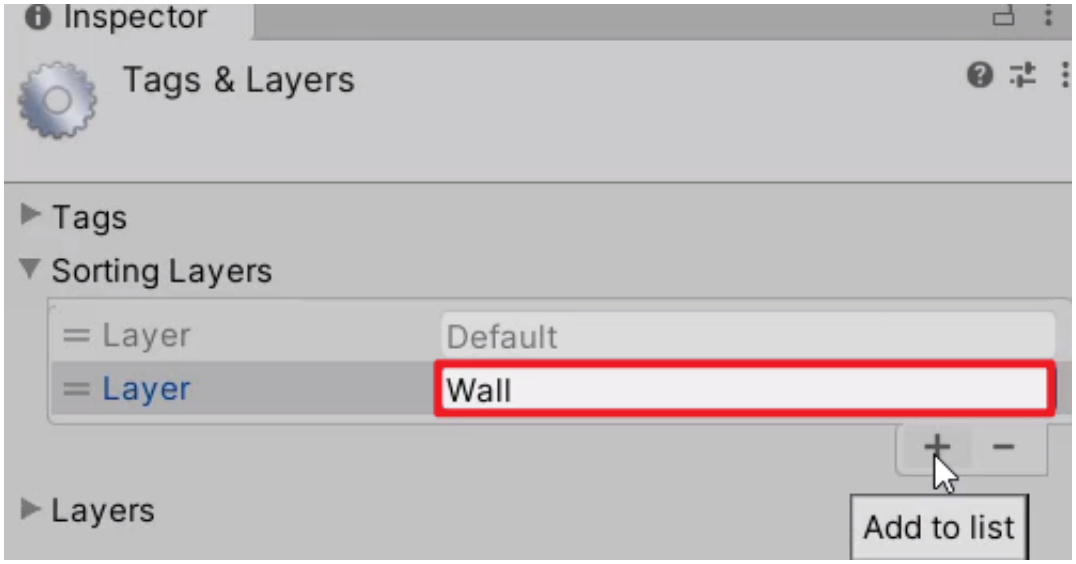
To edit Sorting Layers, select **Layers > Edit Layers** from the top-right corner of the screen.



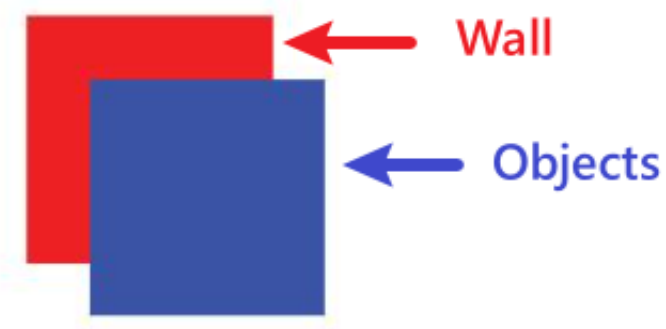
This will open up the **Tags & Layers** window in the Inspector.



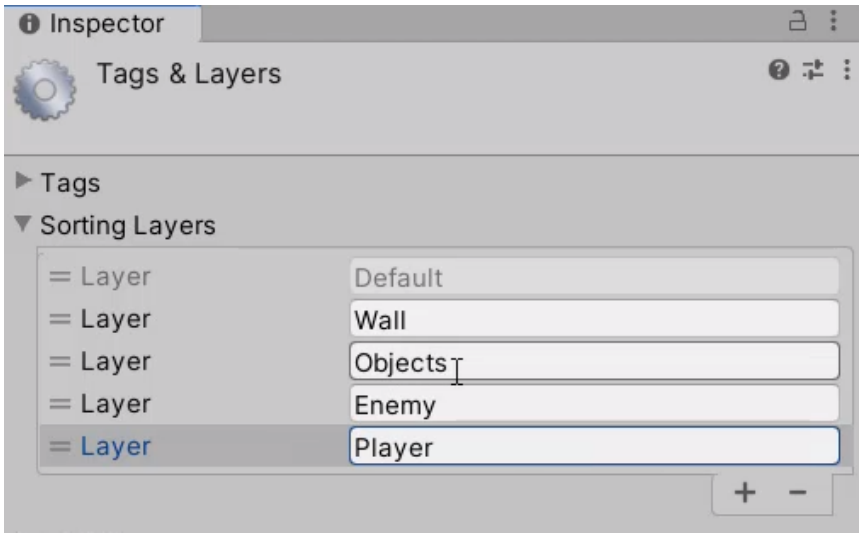
Let's open up the **Sorting Layers** drop-down and add a new layer called "**Wall**". This is what we're going to assign to the **backmost** sprites, so everything from here is going to be on top of the wall.

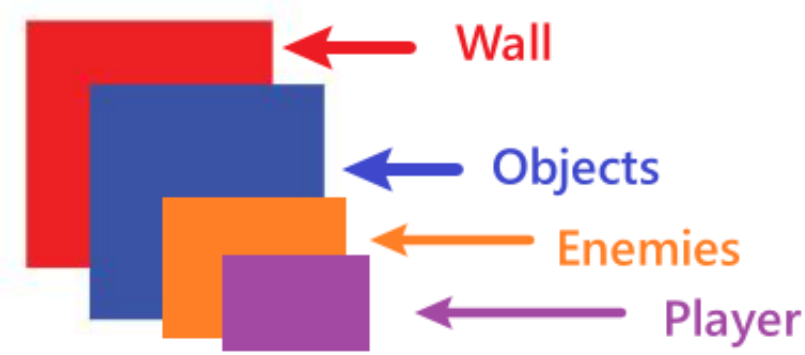


We're going to create another layer called "**Objects**", which will be assigned to the objects that appear on top of the wall such as doors, coins, keys, etc. Since the layer is underneath "Wall", all these objects are going to be rendered **on top** of all the wall sprites.



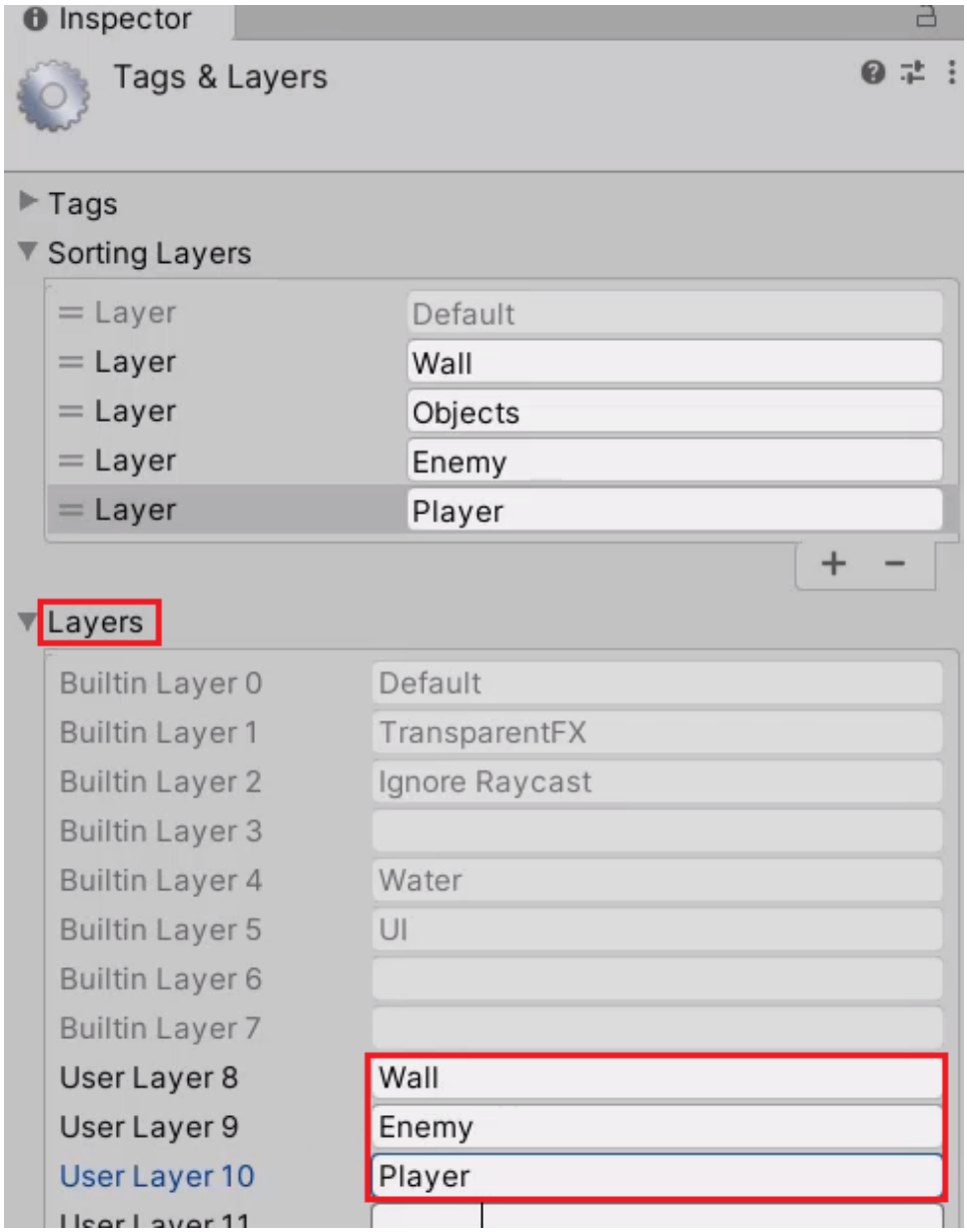
And finally, we're going to make "**Enemies**" and "**Player**" as they are going to appear on top of everything.





Setting Up Layers

Now, aside from **Sorting Layers**, we also need to create some **Layers** for **Walls**, **Enemies,** and **Player**. This is an important step when it comes to detecting **Raycasts** (which we will go over much more later on), but for now, let's just keep in mind that we need layers to refer to these objects inside C# scripts.

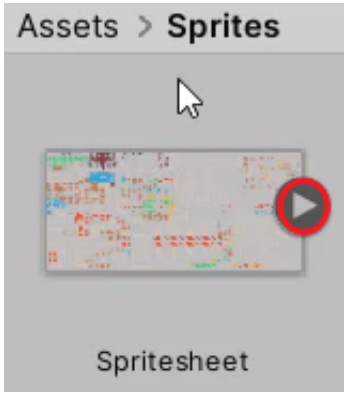


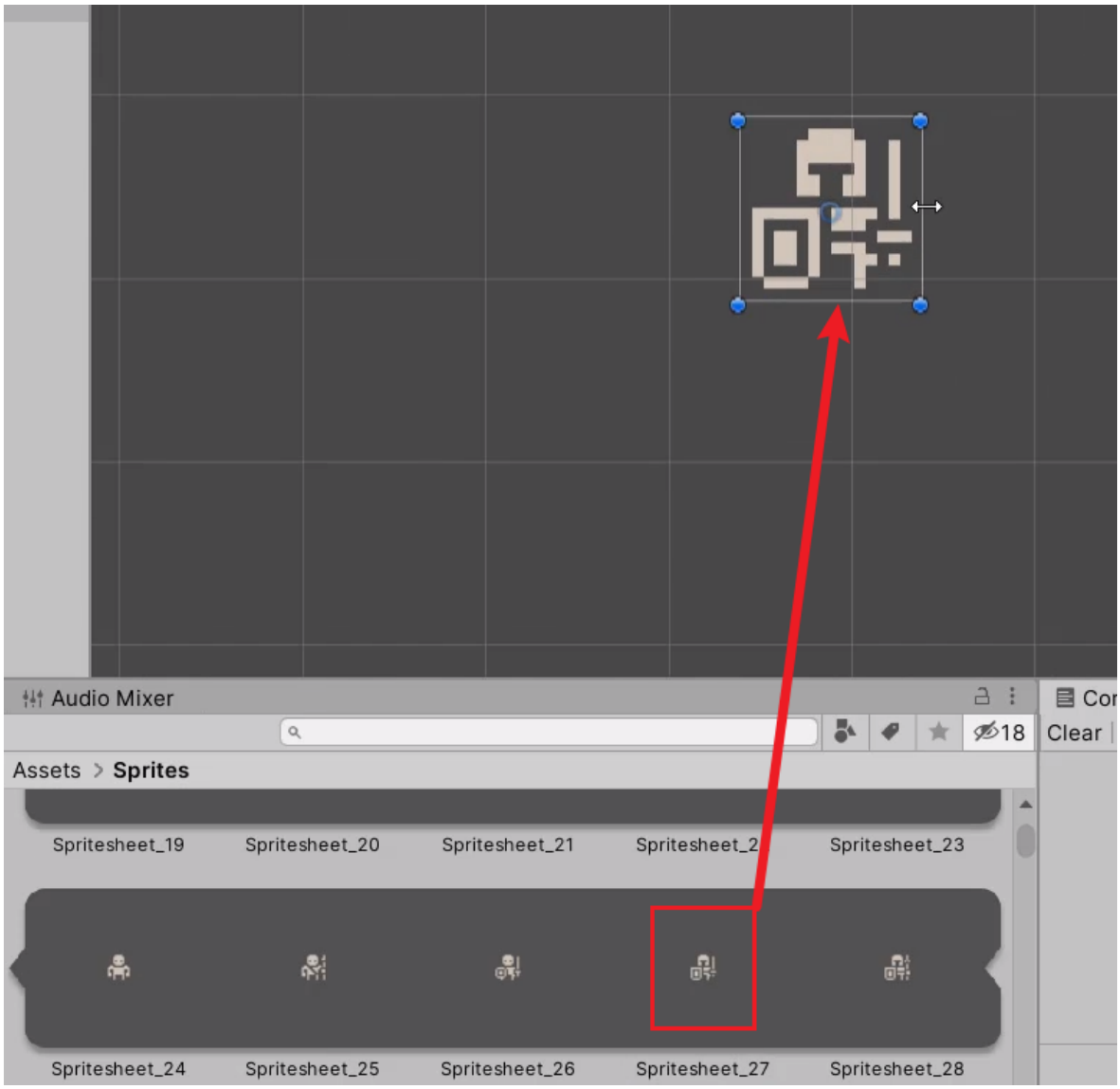
In this lesson, we're going to be creating a Player **GameObject*.**

*(A **GameObject** is the base class for all entities in Unity that you can attach various different components onto.)

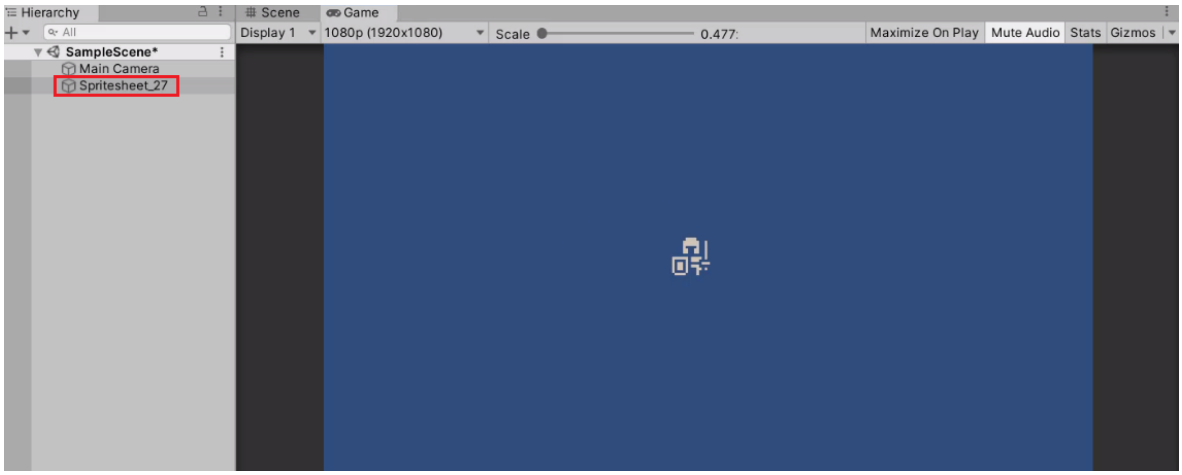
Creating A GameObject

We're going to open up the **Spritesheet** and **drag** our player sprite (**Spritesheet_27**) into the scene.



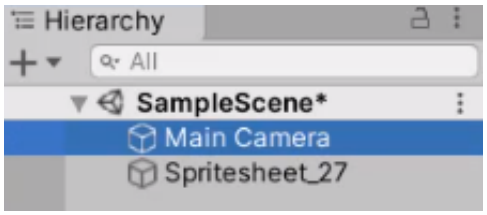


This is going to create a brand new GameObject.

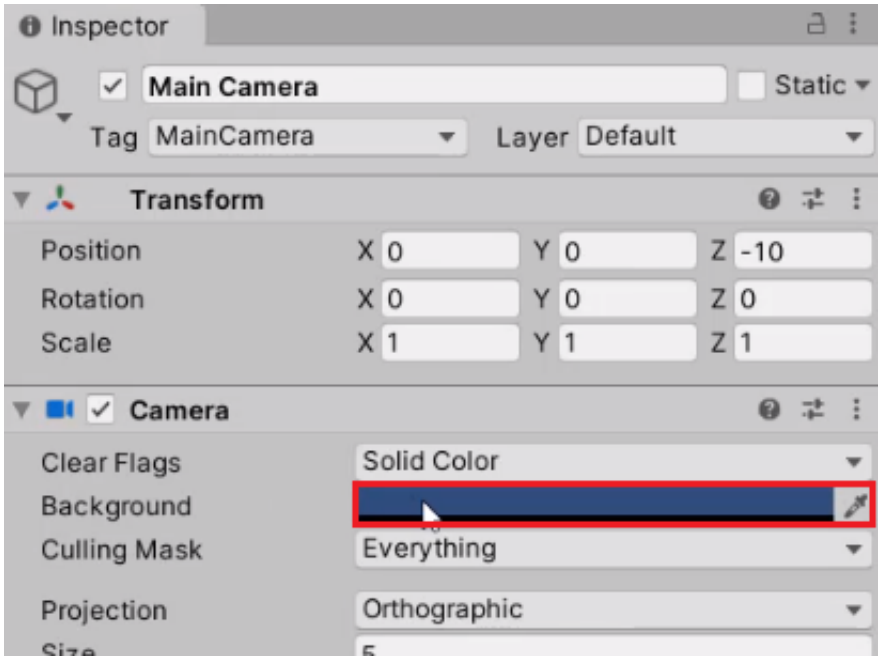


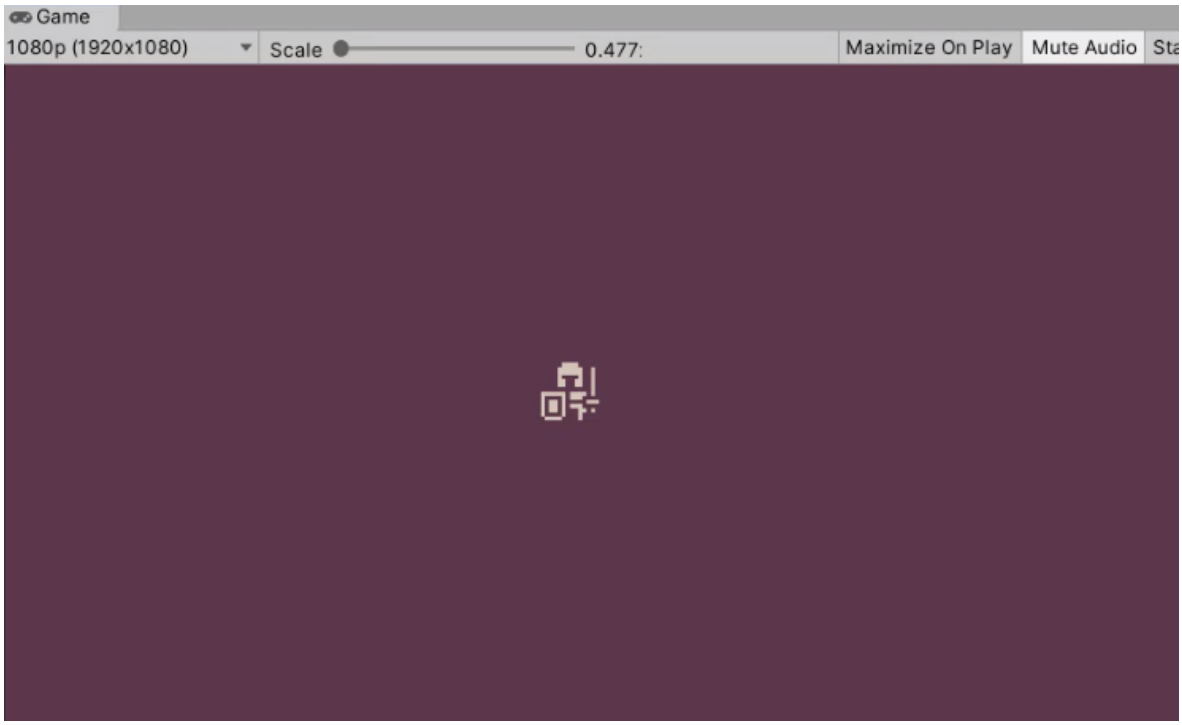
Changing Background Color

To change the background color, we're going to select our **Main Camera** from the Hierarchy:

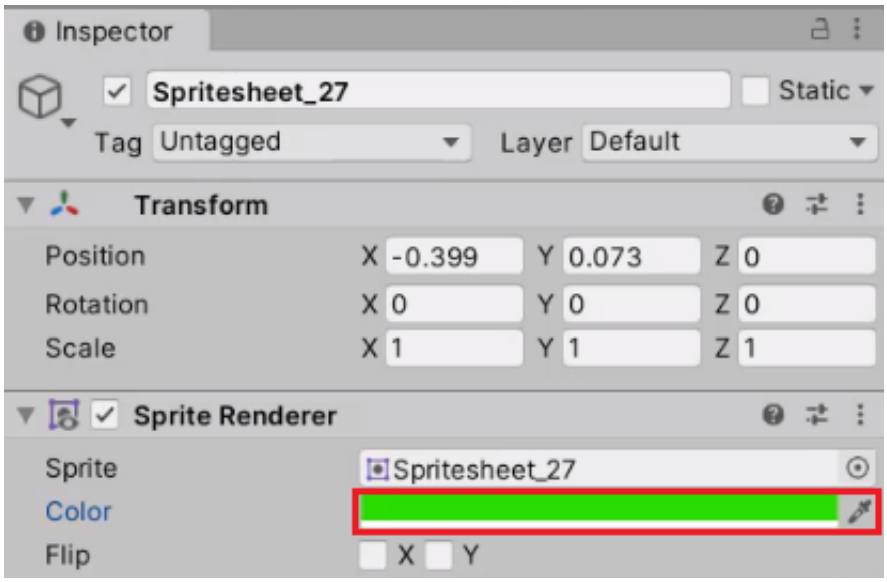


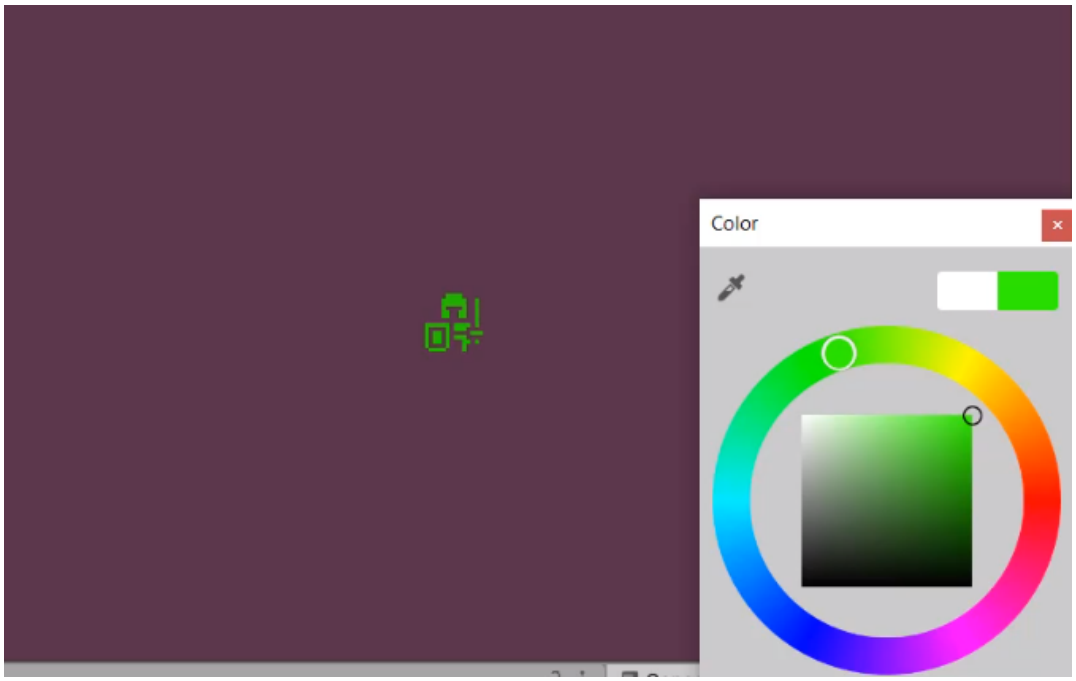
Select the Background and change it to be a dark pink-ish color.





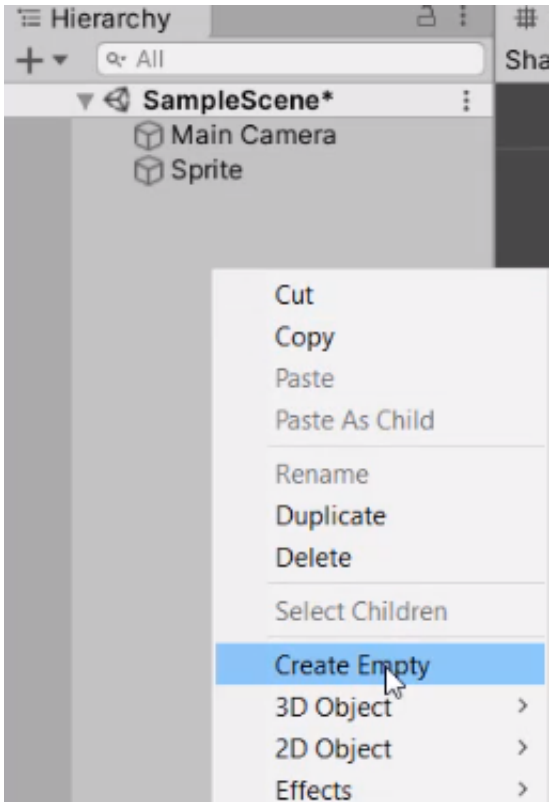
Similarly, we can change the color of our sprite by clicking on the **Color** property on **Sprite Renderer**.



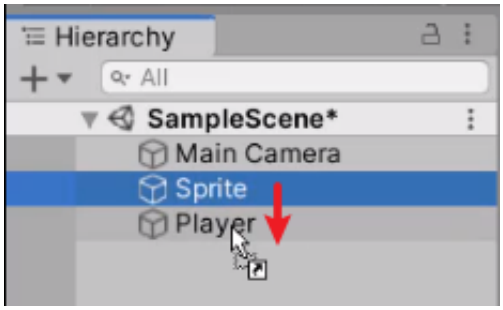


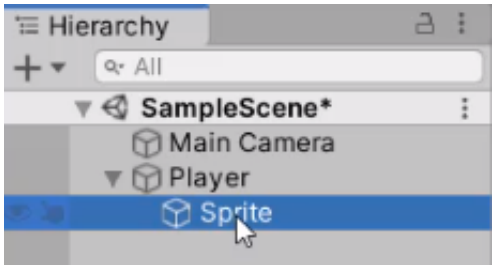
Setting Up Player GameObject

We're going to create a new **empty** GameObject, which is going to have this **Sprite** as a child. **(Right-click > Create Empty)**



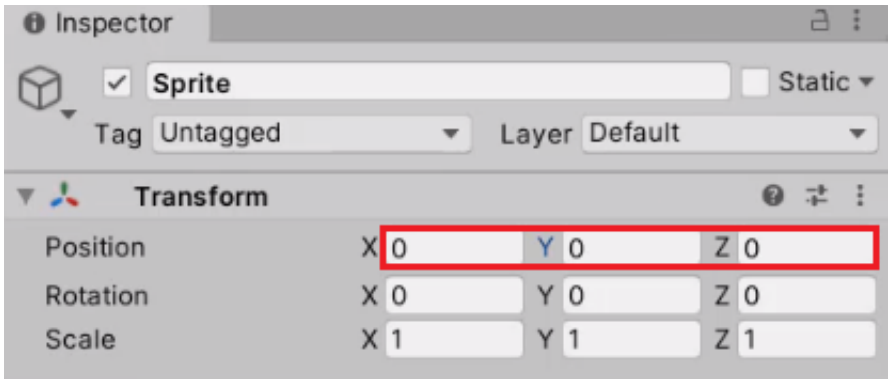
We're going to **rename** this empty object as "Player", and drag our **Sprite** object over it.



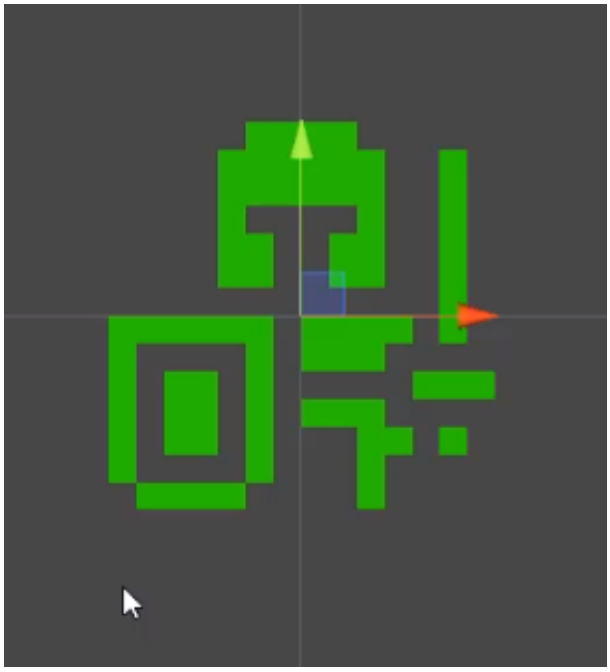


This allows us to finely adjust the child sprite position and pivot without modifying our parent (Player) object.

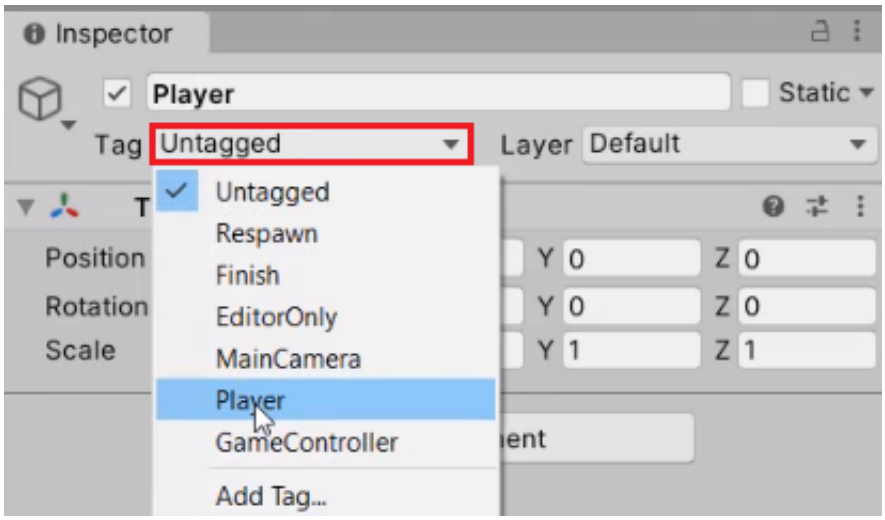
Select the **Sprite**, and reset all the **Position** to be 0, 0, 0.

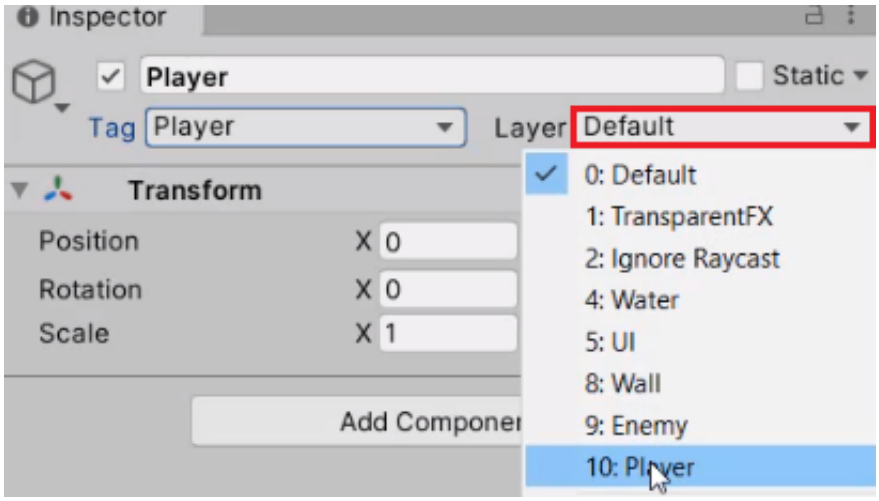


Our player sprite is now at the pivot point (center) of our GameObject– this is going to make things a lot easier, such as implementing ground detection.



Select the **Player** object, and set the **Tag** and the **Layer** as "**Player**".



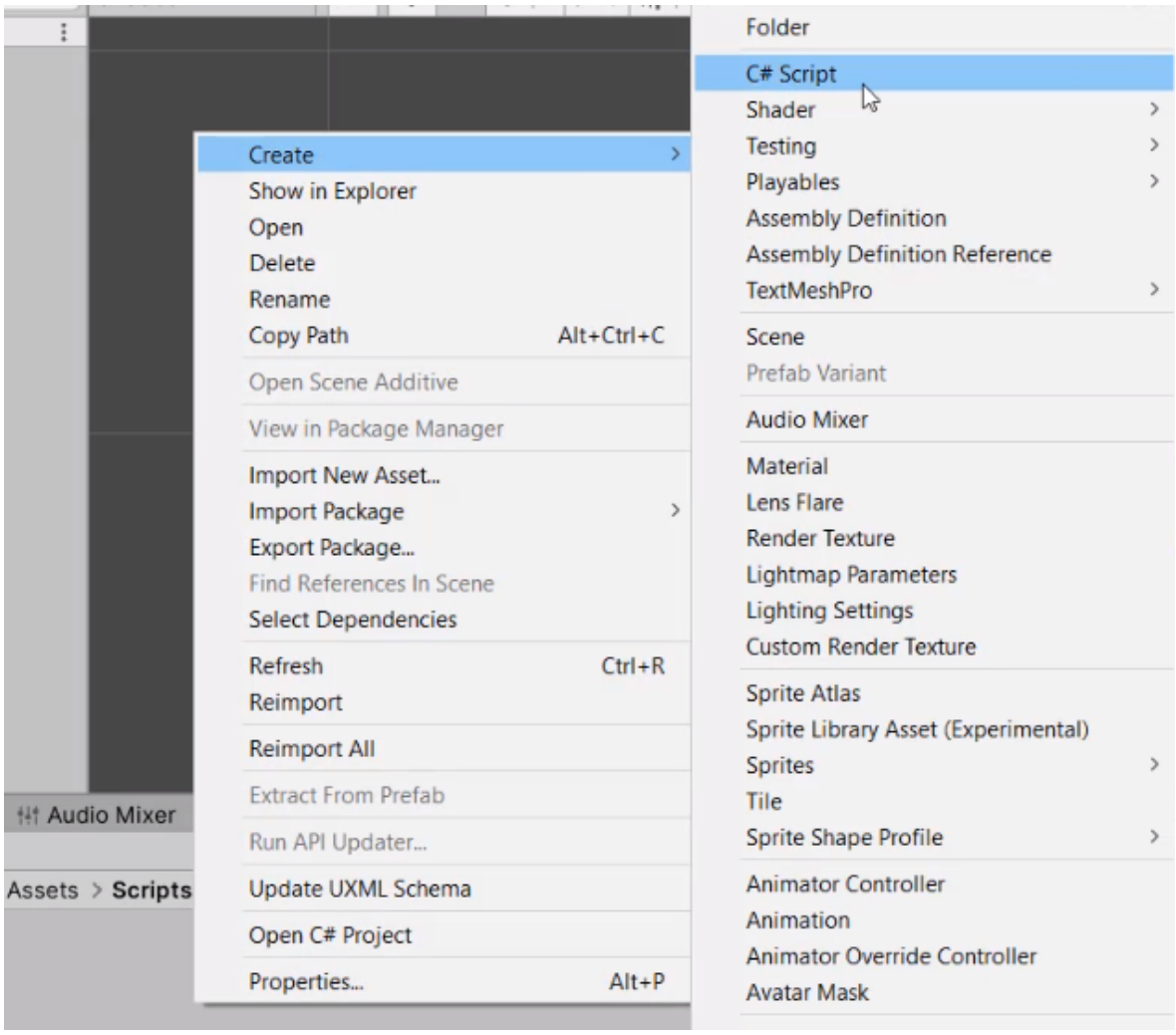


Now we will add the following components to our Player GameObject:

- C# Script
- CapsuleCollider2D
- Rigidbody2D

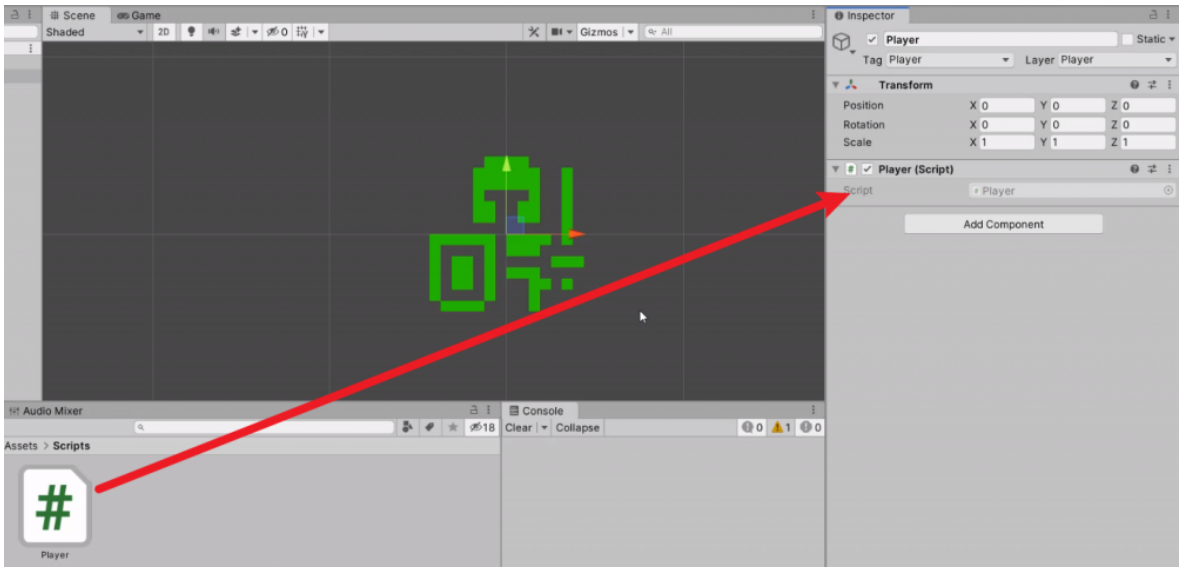
Adding A C# Script

Inside of the **Scripts** folder (**Assets > Scripts**), we're going to create a new **C#** script (**Right-click > Create > C# Script**).



This script will be used to implement all the functionalities related to our player, such as movement, attacking, taking damage, etc.

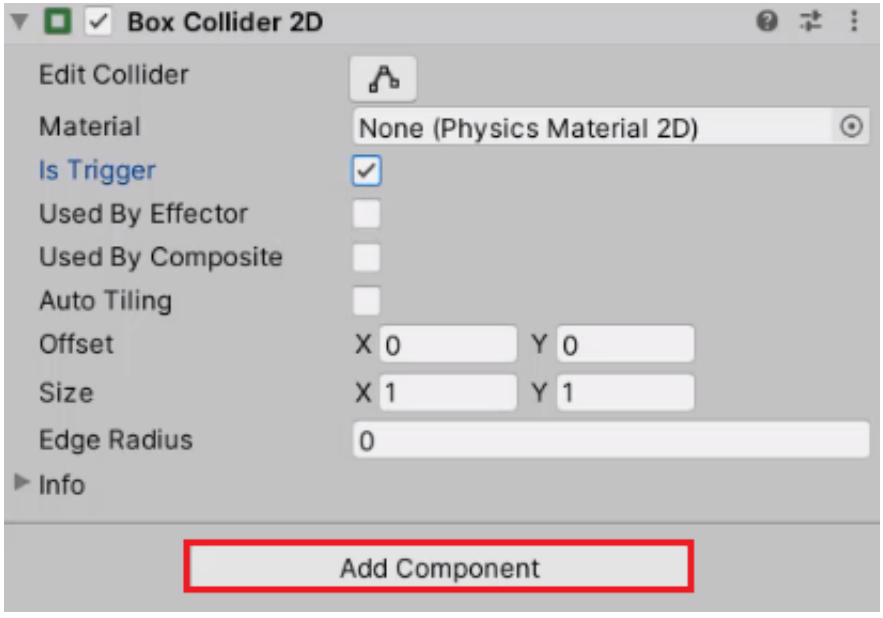
Select the **Player** object, and **drag** the script into the Inspector.



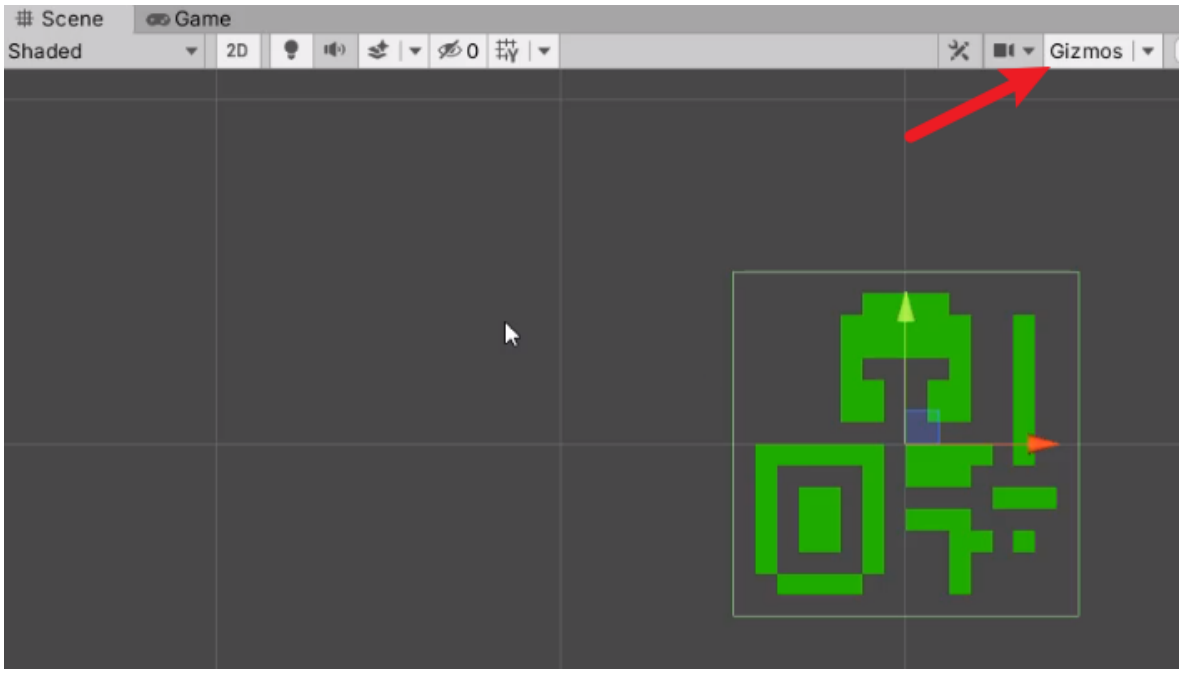
Adding Components (Collider)

Colliders in Unity are components that provide collision detection, which allows our GameObjects to stand on the ground and interact with other objects.

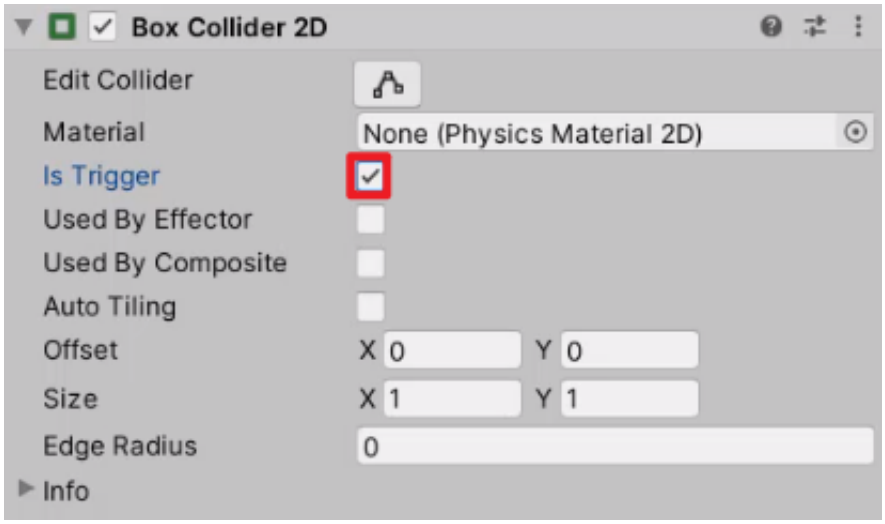
To add a box-shaped collider, go to the Inspector and click on **Add Component** > (search) **Box Collider 2D**.



Make sure to enable the '**Gizmos**' button so the collider (green lines) appears in the Scene view.



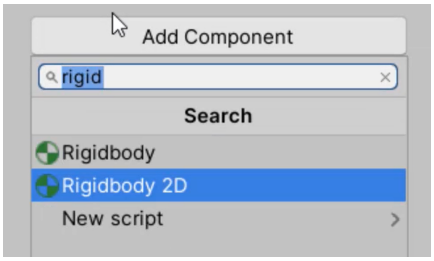
Let's set this to be a **Trigger**, so we can pass through other objects and still be able to detect what object has entered our collider.



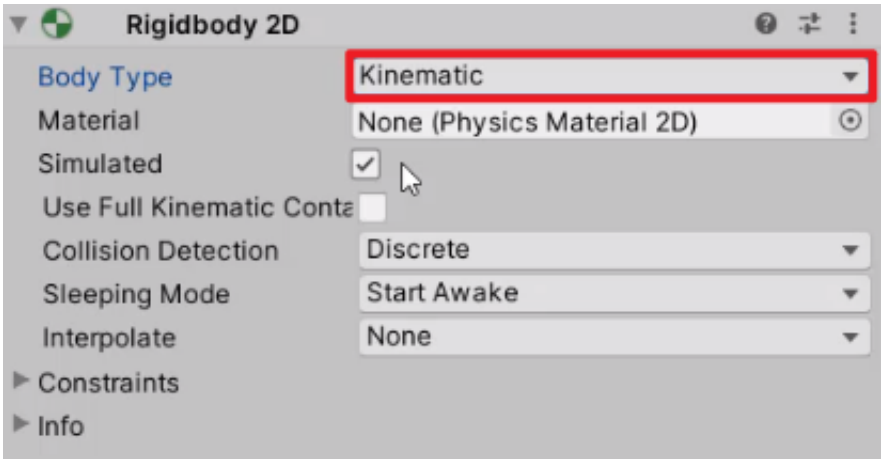
Adding Components (Rigidbody)

If you want the collider to be detected by other colliders, at least one of the colliding objects needs to have a **Rigidbody2D** component.

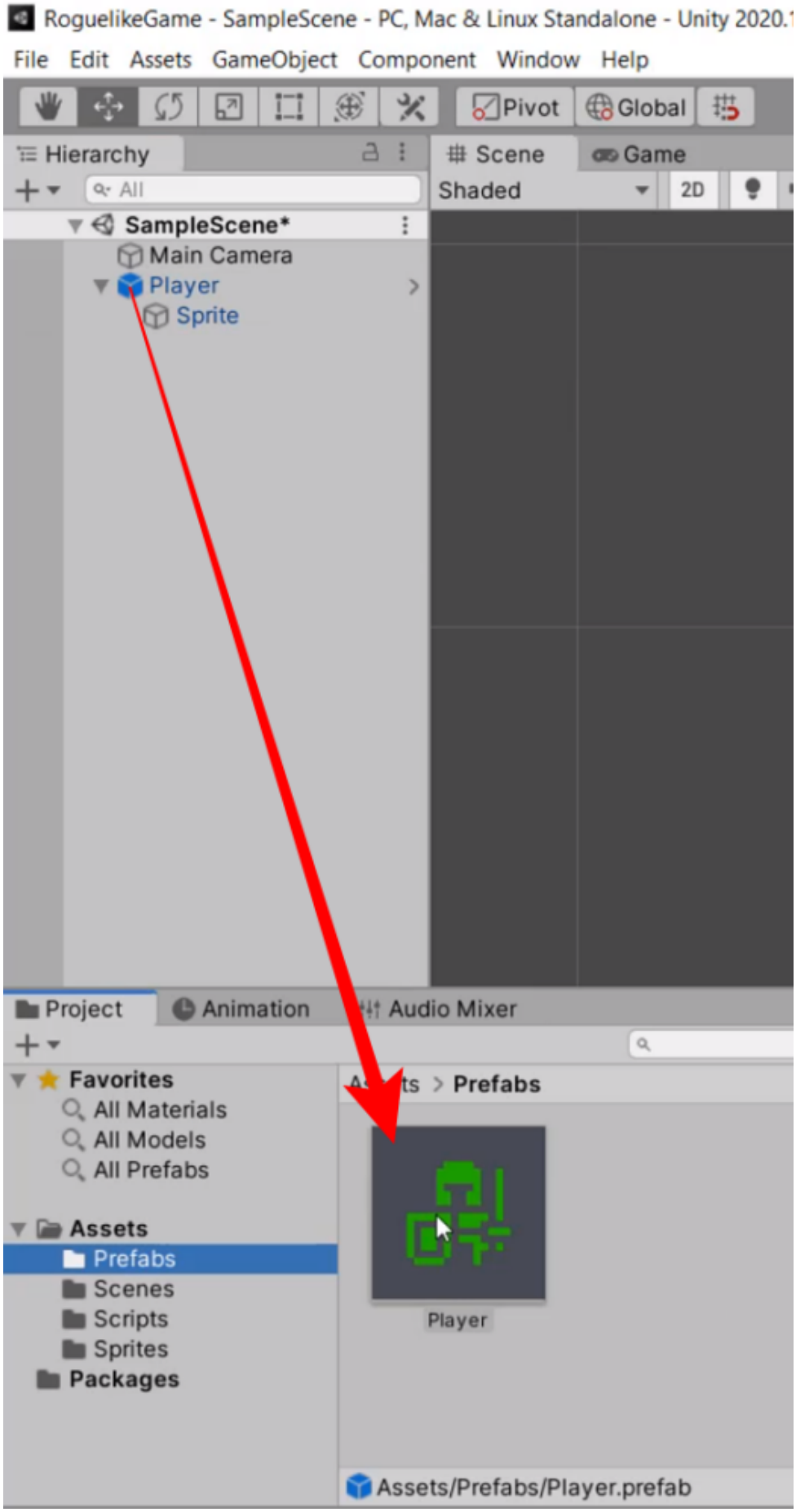
Rigidbody is basically a component that applies physics such as gravity, drag, mass, etc. We're going to click on **Add Component >** (search) **Rigidbody2D**.



By default, the **Body Type** is set to "**Dynamic**". This means our player is going to get affected by gravity and fall through the ground. Let's change it to be "**Kinematic**" so it can only be moved by our C# script.



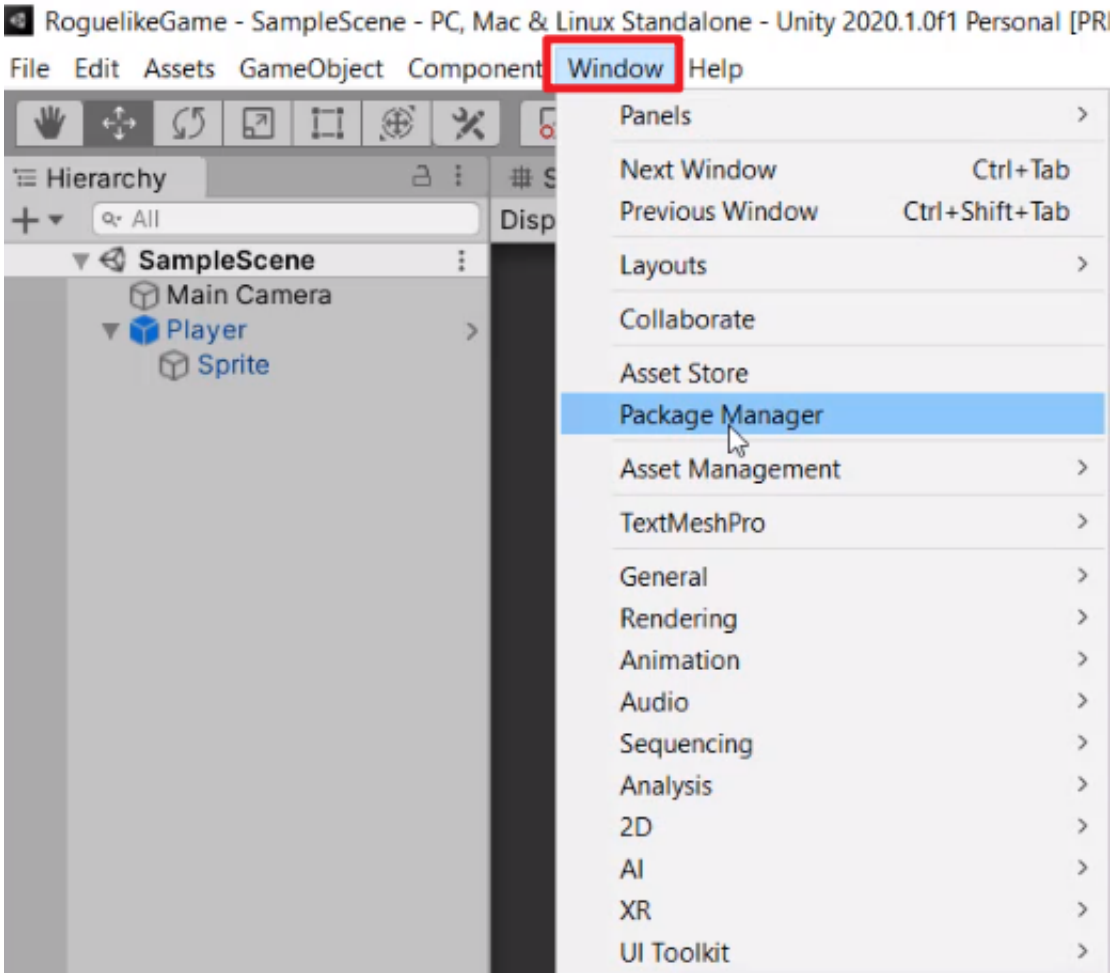
Lastly, create a **Prefab** of the **Player** by dragging it from the scene **Hierarchy** into the **Prefabs folder** in the **Project** window.



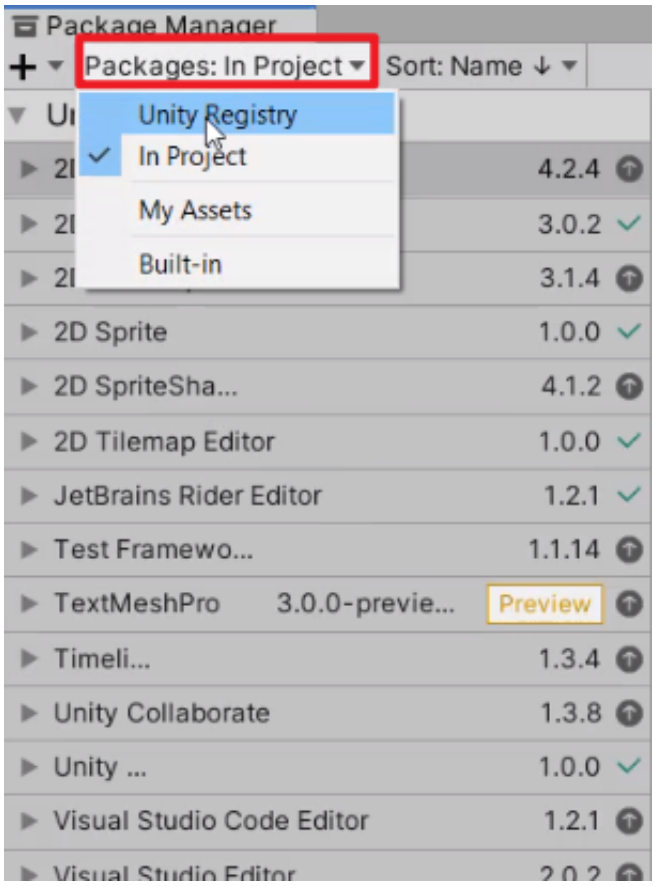
In this lesson, we're going to be setting up our **inputs** using Unity's new **Input System**.

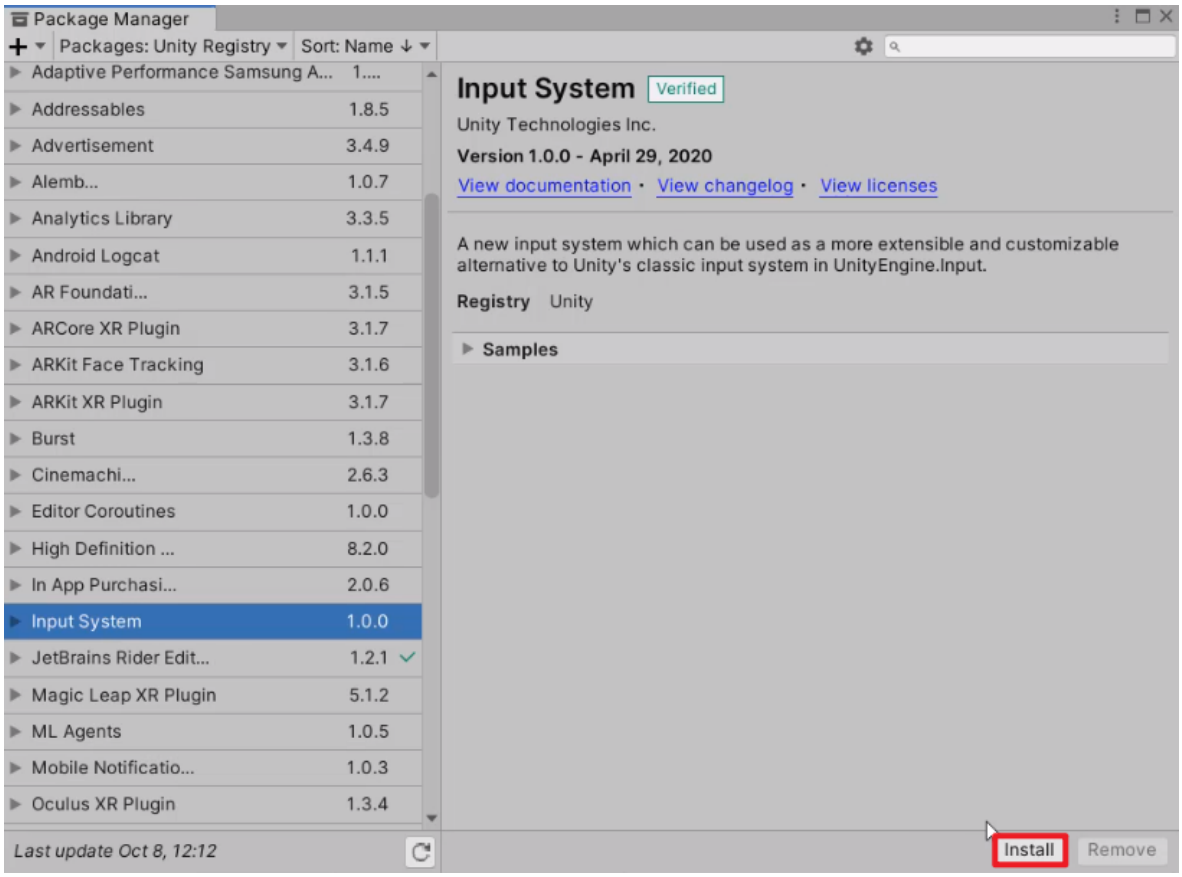
Installing Input System Package

Let's open up the **Package Manager** window (**Window > Package Manager**).



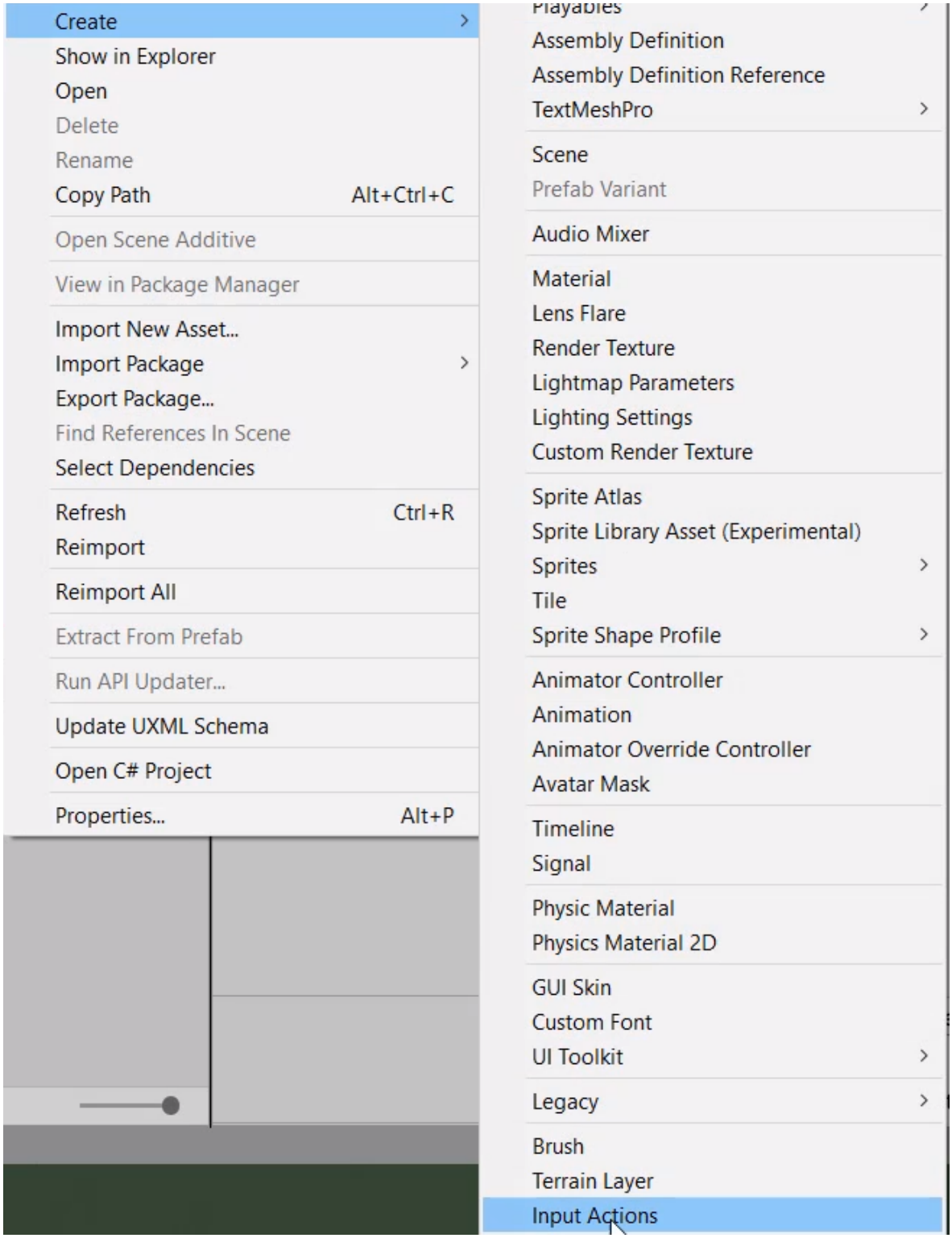
Select **Packages > Unity Registry** to access all the Unity packages, and install **Input System**.





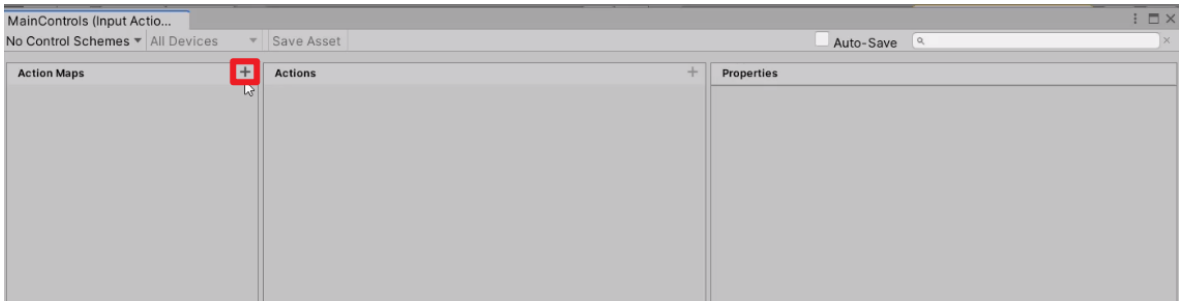
Creating Input Actions

Once that's done, we can **right-click** on the Project window and click on **Create > Input Actions**.

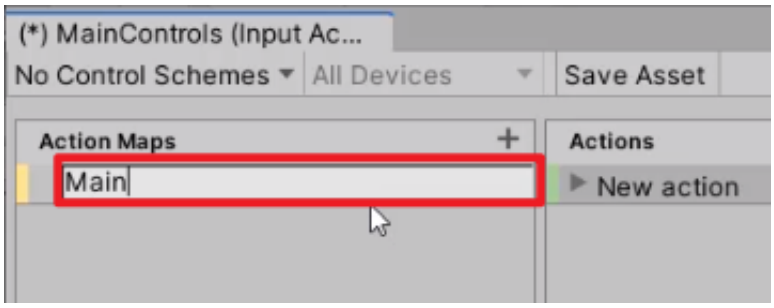


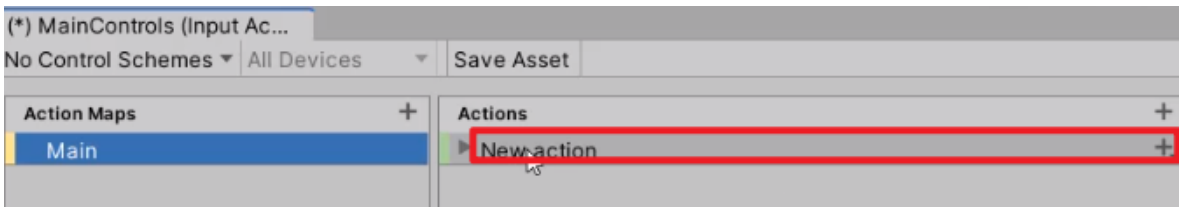
We're going to call this "**MainControls**" and **double-click** to open it up.

This is going to open up the **Input Actions** window. On the leftmost panel, we can create an **Action Map** by clicking on the **+** button. This is basically a category for all your different inputs.

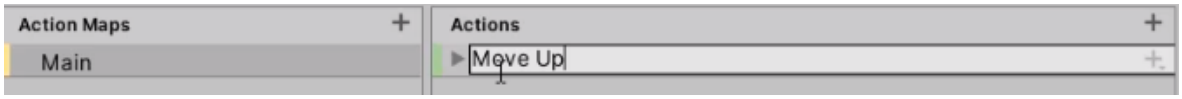


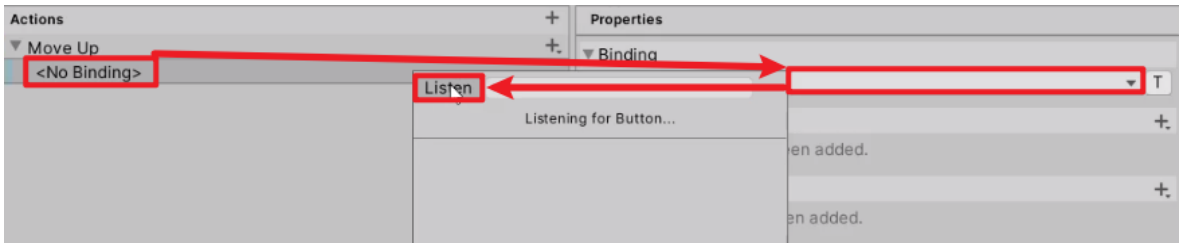
We'll call the first action map "**Main**", and create a new **Action** connected to the "Main" action map. Actions define what we want to do when pressing certain keys.



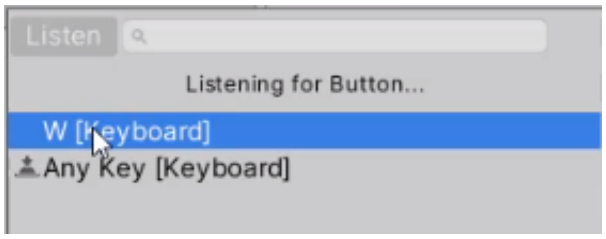


So first, let's create an action called "**Move Up**". Then we can bind a custom key to the action by clicking on **No Binding** > **Path** > **Listen**.



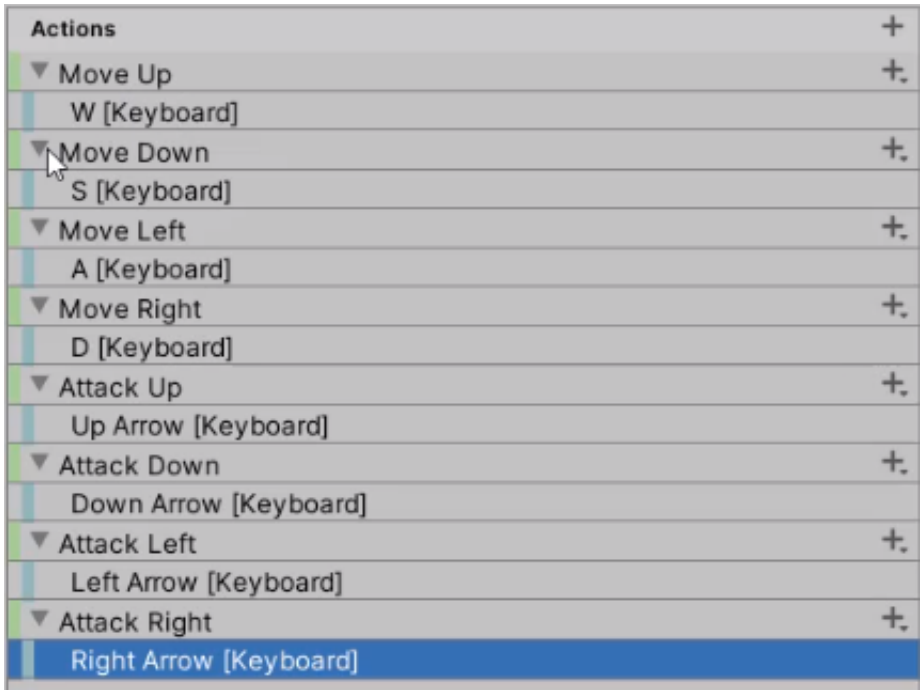


Then we can press the "**w**" key on our keyboard, and select **W [Keyboard]**.

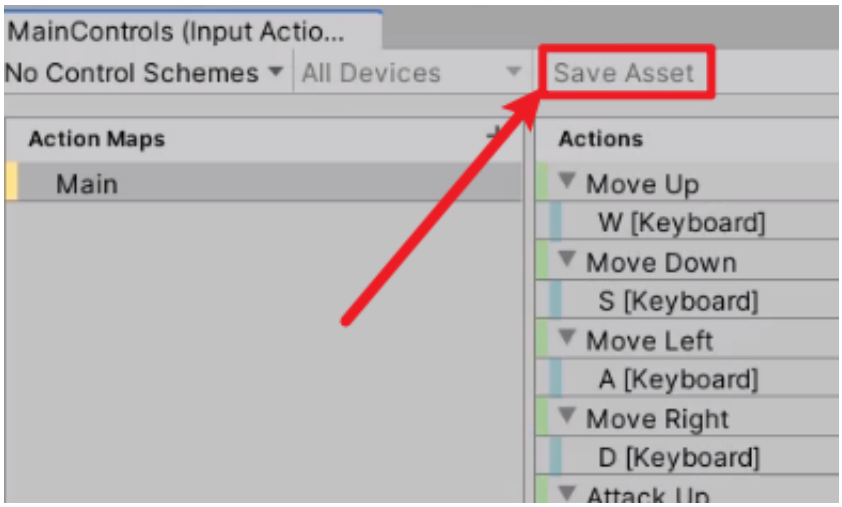


Now you'll see that "Move Up" has a binding of "W":

In our game, we're going to have 8 different buttons: **move up** / **down** / **left** / **right** using the **W** / **A** / **S** / **D** keys, as well as **attack up** / **down** / **left** / **right** using the **arrow** keys.

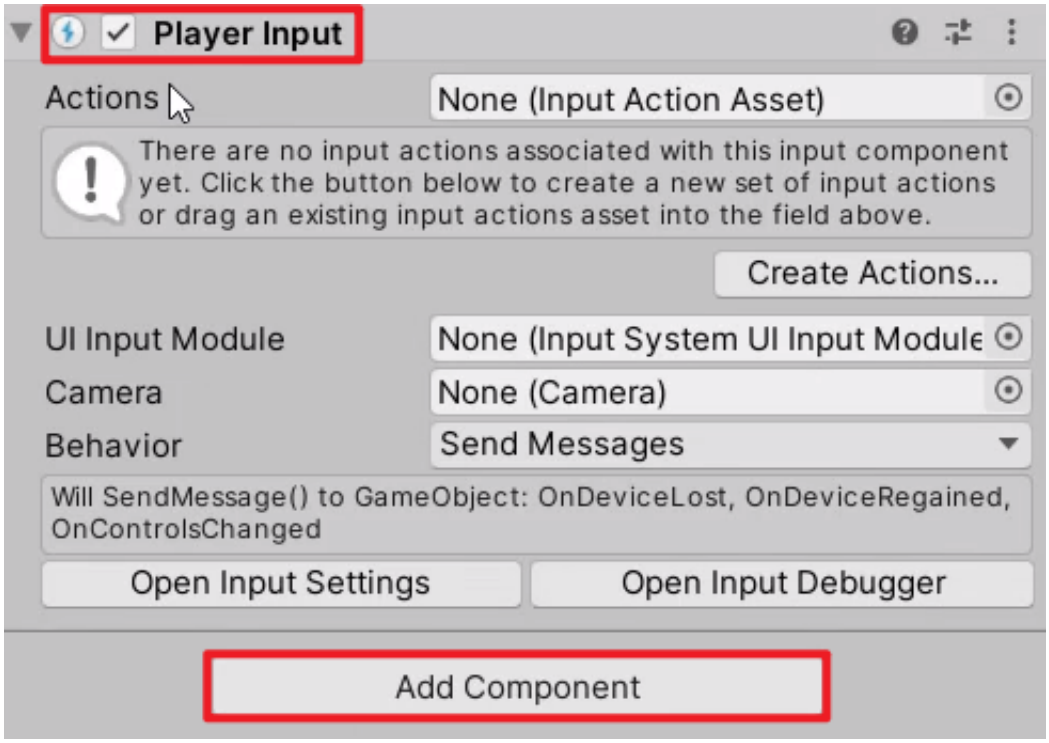


Before you close the window, make sure that the asset is **saved**.

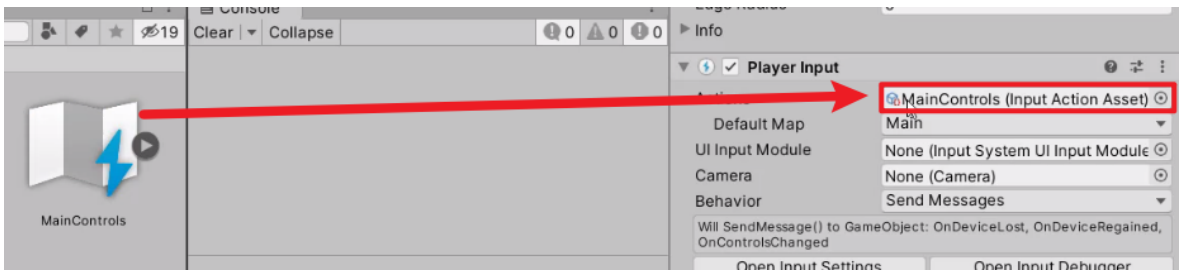


Detecting Inputs

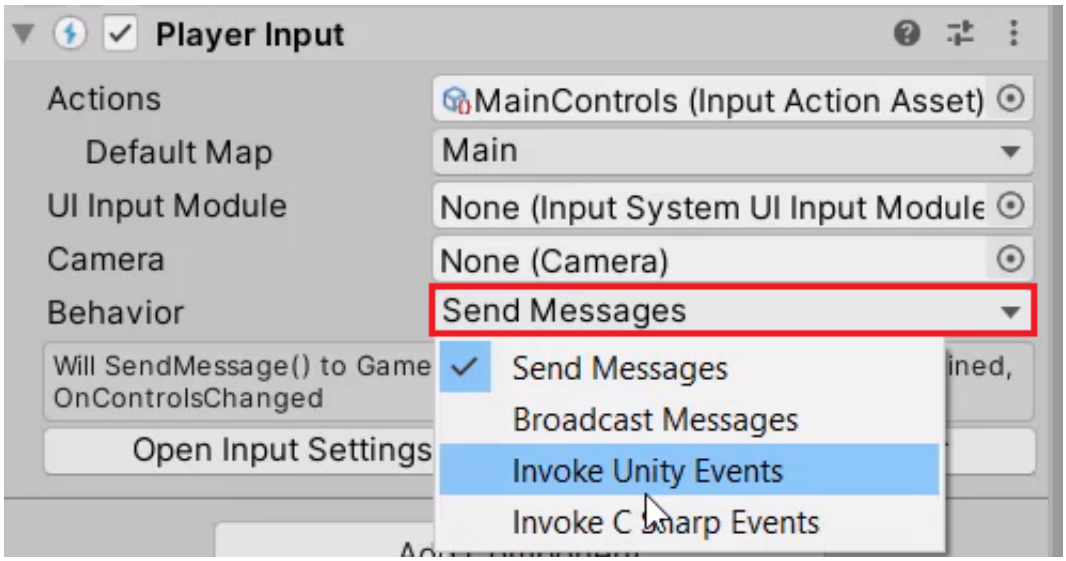
In order to detect and make use of these inputs, we need to select our **Player** object and add a new component called "**Player Input**".



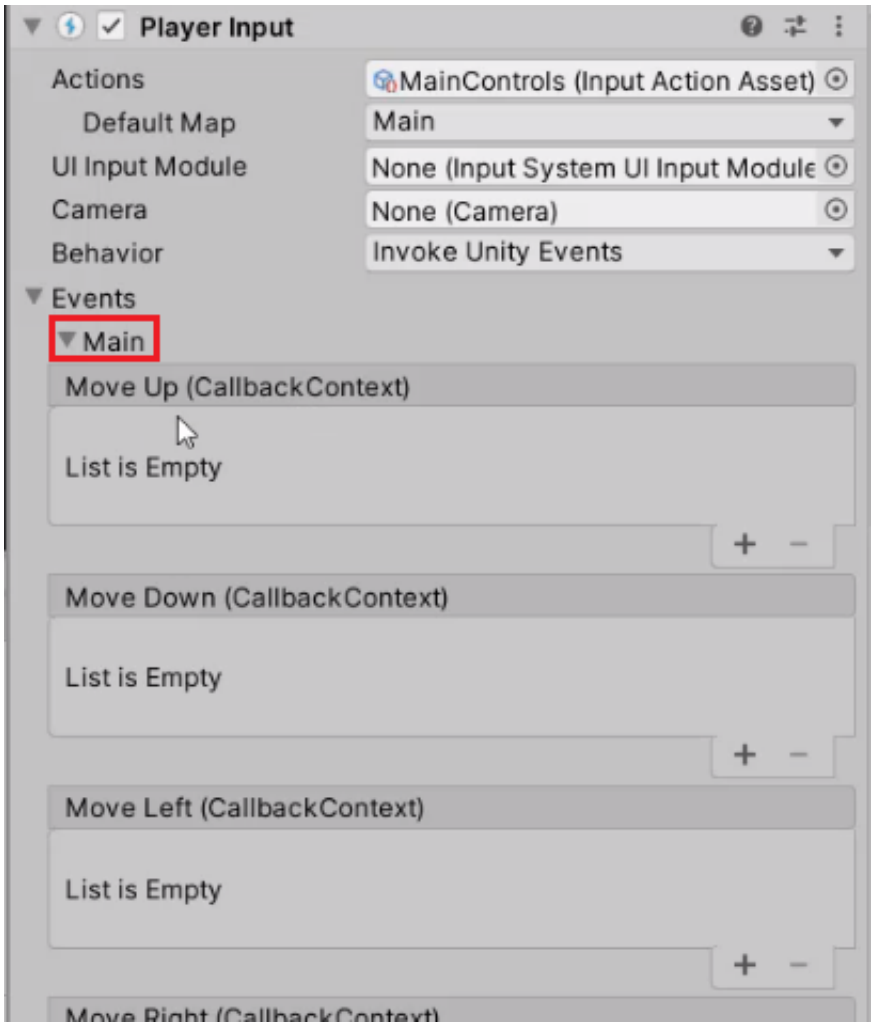
Then, drag the **Input Action** asset into the **Actions** property of Player Input.



The **behavior** property defines how we're going to be detecting when we press our buttons.

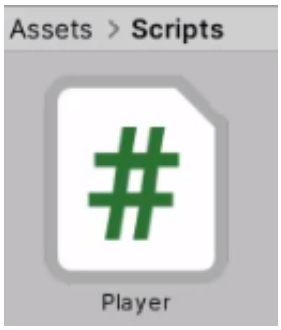


If we change this from '**Send Messages**' to '**Invoke Unity Events**', you'll get access to the '**Main**' Action Map, which contains all the actions that we created earlier. We can now create a function inside of a C# script and link it up here.



Linking Action Through Script

Let's open up **Assets > Scripts**, and double-click on the **Player** script.



First of all, we need to import the library package to get access to the **InputSystem** at the top of the script.

```
using UnityEngine.InputSystem;
```

Then we're going to create a function that will be called whenever we press the '**W**' (**Move up**) key. This function will take in a parameter of type **CallbackContext**, which contains various different information about the input, such as which key was pressed, at which frame the key was pressed, etc.

```
public void MoveUp (InputAction.CallbackContext)
{

}
```

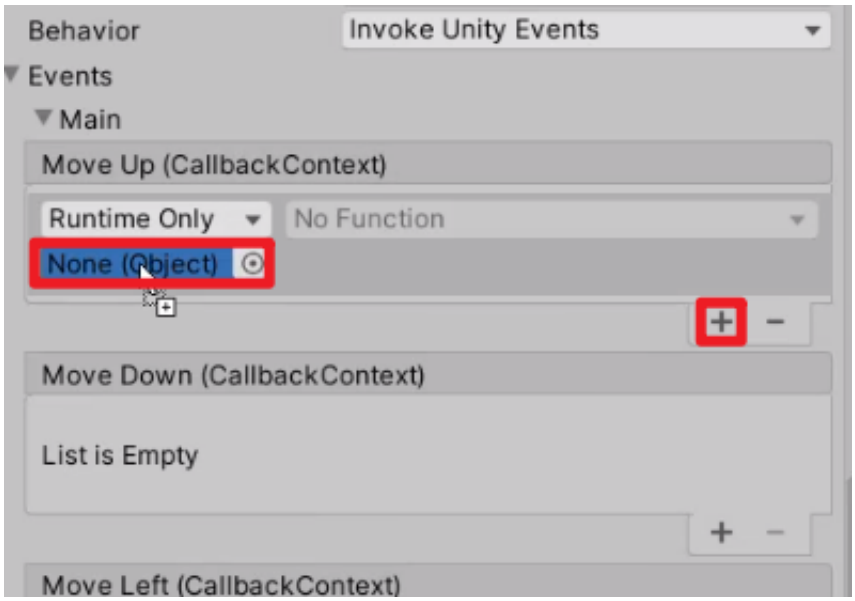
Each action will need its own function to bind with, so let's **duplicate** the function and **rename** it:

```
public void MoveUp (InputAction.CallbackContext)
{

}
public void MoveDown (InputAction.CallbackContext)
{

}
public void MoveLeft (InputAction.CallbackContext)
{

}
public void MoveRight (InputAction.CallbackContext)
{

}
public void AttackUp (InputAction.CallbackContext)
{
```

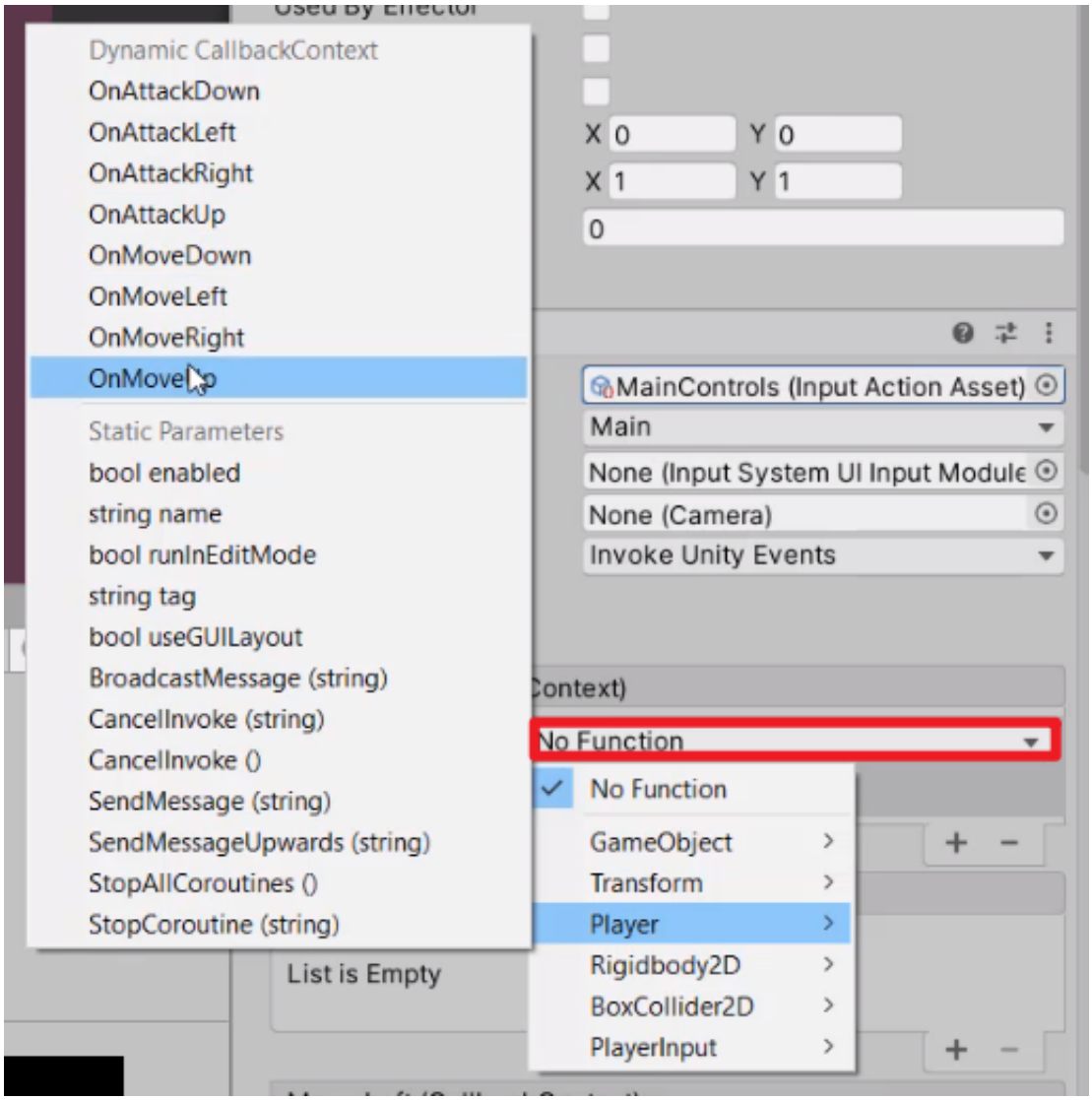


```
}
public void AttackDown (InputAction.CallbackContext)
{

}
public void AttackLeft (InputAction.CallbackContext)
{

}
public void AttackRight (InputAction.CallbackContext)
{

}
```

We can then click on the **Add** (+) button and drag in the **Player** (either the Player **component** above this Player Input component or the Player **object in the scene Hierarchy)** into the **Object** field. This will allow us to select the functions inside the script.

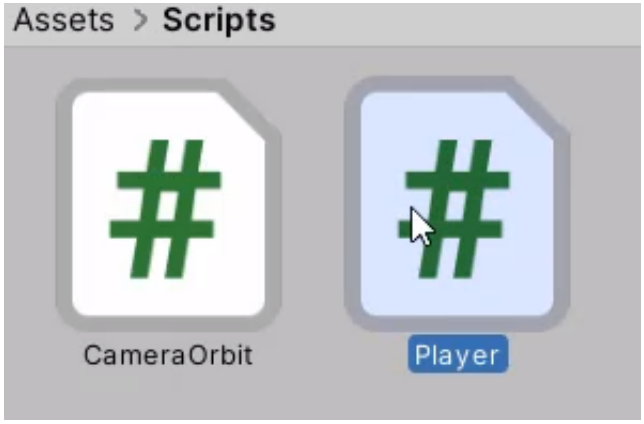


(If they're not appearing here, make sure to check that the functions are set to be 'public', that your script is saved to apply those changes, and that you have NOT dragged in the Player.cs script file directly – you must assign a Player instance from the Scene to be referenced here.)



In this lesson, we're going to begin setting up our player movement.

Let's **double-click** on the "Player" script and open it up in Visual Studio.



Declaring Variables

The first thing we're going to be doing inside of the script is declaring **public variables** to keep track of the player's **current** health points, **max** health points, and the amount of **coins**.

```
public class Player : MonoBehaviour
{
    public int curHp;
    public int maxHp;
    public int coins;
}
```

We then also need to determine if we currently have the **key** to get out of the level.

```
public class Player : MonoBehaviour
{
    public int curHp;
    public int maxHp;
    public int coins;

    public bool hasKey;

}
```

Finally, we want to flash the screen with white or red color whenever we take damage. To do this, we need to get a reference to the **SpriteRenderer** component.

```
public class Player : MonoBehaviour
{
    public int curHp;
    public int maxHp;
    public int coins;
```



```
    public bool hasKey;

    public SpriteRenderer sr;
}
```

Detecting Input

Now we're going to create a function called "**Move**", which will be called when any of our actions are triggered (= when an arrow key or W/A/S/D is pressed down). To prevent our player from moving through walls and objects, we're going to pass the direction of movement as a **Vector2**.

```
public class Player : MonoBehaviour
{
    public int curHp;
    public int maxHp;
    public int coins;

    public bool hasKey;

    public SpriteRenderer sr;

    void Move (Vector2 dir)
    {

    }
}
```

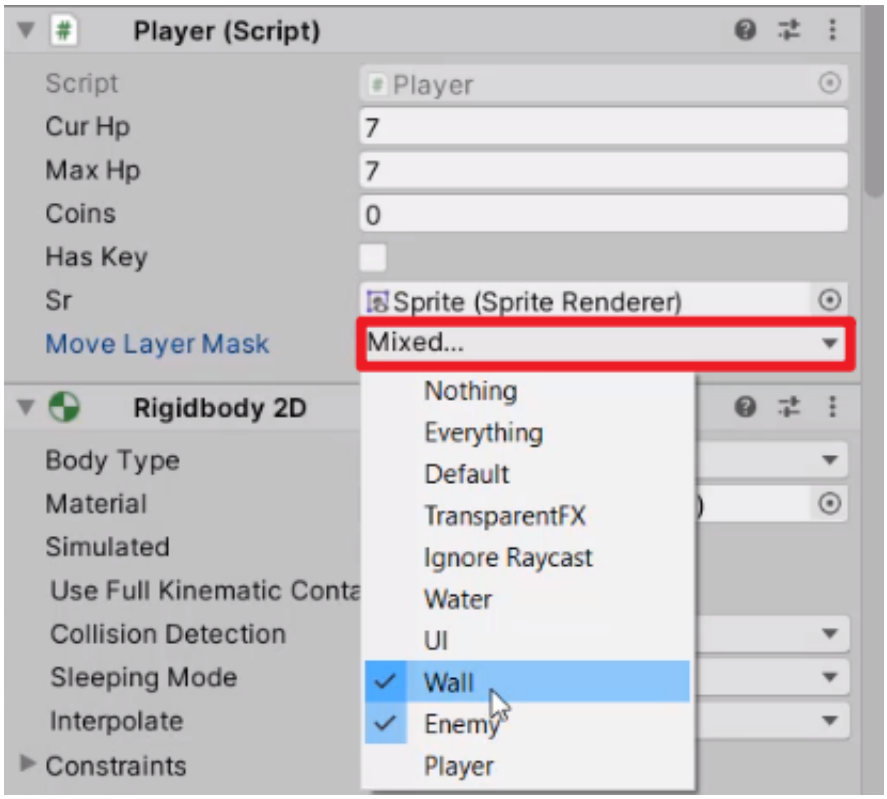
We can then use this **Vector2** "dir" parameter to shoot a ray towards the direction of movement and detect if there is any collider in front of us.

```
public class Player : MonoBehaviour
{
    public int curHp;
    public int maxHp;
    public int coins;

    public bool hasKey;

    public SpriteRenderer sr;

    // layer to avoid (mask)
    public LayerMask moveLayerMask;

    void Move (Vector2 dir)
    {
        // cast a ray from the player's position to the direction of movement, and d
etect colliders that are layered as "moveLayerMask".
        RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f, moveLaye
rMask);
    }
}
```


If there is nothing detected in front of us, the **Raycast.collider** will return **null** and we can move forward. If we collided with a **wall** or an **enemy**, then the ray cast will return something (**!= null**) and so we shouldn't be able to move further.

```
public class Player : MonoBehaviour
{
    public int curHp;
    public int maxHp;
    public int coins;

    public bool hasKey;

    public SpriteRenderer sr;

    // layer to avoid (mask)
    public LayerMask moveLayerMask;

    void Move (Vector2 dir)
    {
         RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f, moveLaye
rMask);

         // if there is no moveLayerMask detected in front of us,
         if(hit.collider == null)
         {
              // move forward
              transform.position += new Vector3(dir.x, dir.y, 0);
         }
    }
}
```

Since **transform.position** can only take in a value of type **Vector3**, we can't simply give it "dir", which is Vector2. Instead, we're creating a new **Vector3** and assigning **0** to the **z-position** when we're moving forward. Now let's go back in the Editor, and assign the **Enemy** and **Wall** layers as "**moveLayerMask**".



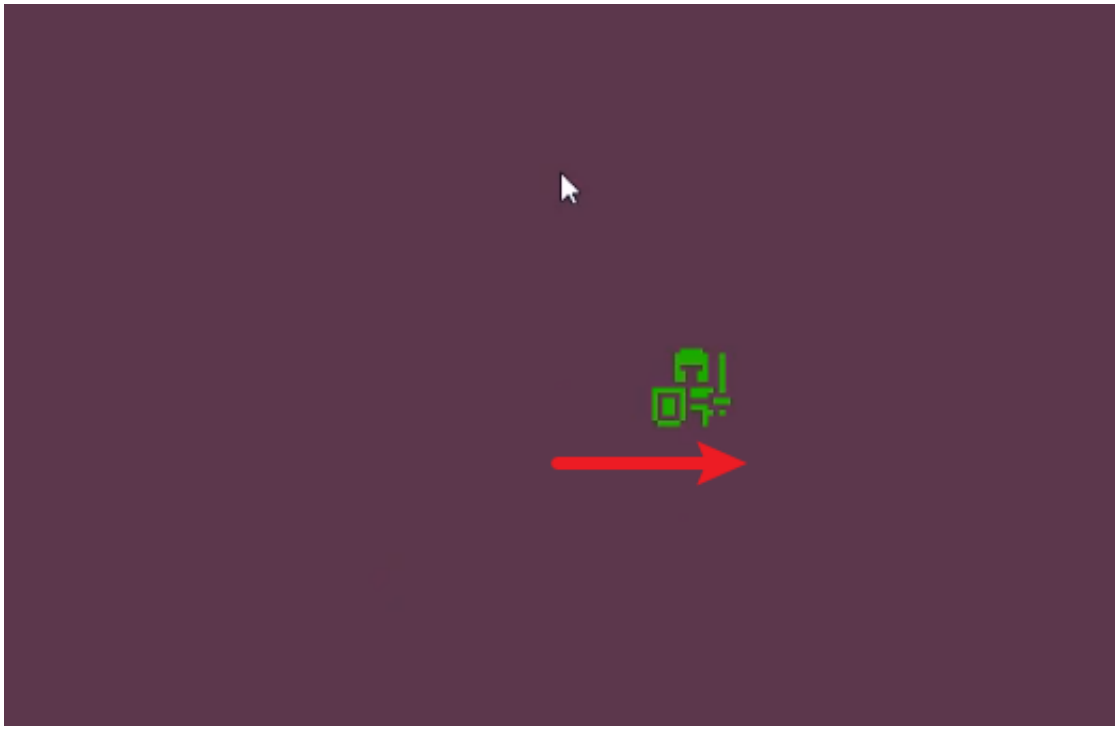
Now we can call this **Move** function whenever our movement keys are pressed.

```
public void MoveUp (InputAction.CallbackContext)
{
     // have we pressed down the corresponding key?
     if(context.phase == InputActionPhase.performed)
         // if so, move towards the given direction.
         Move(Vector2.up);
}
public void MoveDown (InputAction.CallbackContext)
{
      if(context.phase == InputActionPhase.performed)
         Move(Vector2.down);
}
public void MoveLeft (InputAction.CallbackContext)
{
      if(context.phase == InputActionPhase.performed)
         Move(Vector2.left);
}
public void MoveRight (InputAction.CallbackContext)
{
      if(context.phase == InputActionPhase.performed)
         Move(Vector2.right);
}
public void AttackUp (InputAction.CallbackContext)
{

}
public void AttackDown (InputAction.CallbackContext)
{
```



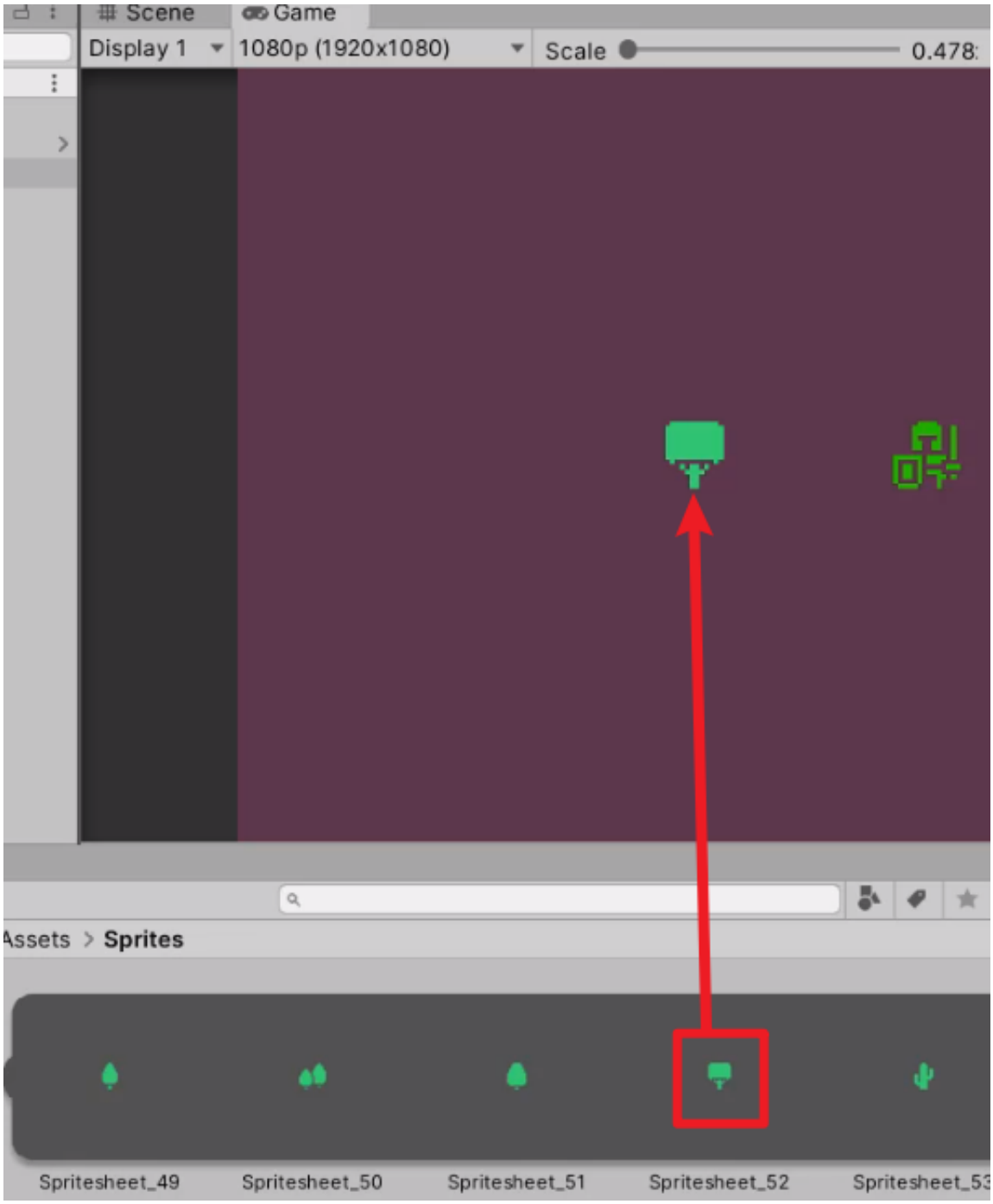
```
}
public void AttackLeft (InputAction.CallbackContext)
{

}
public void AttackRight (InputAction.CallbackContext)
{

}
```

After that, we can hit **Save** (Ctrl+S or ⌘+ S) and move our player around using **W / A / S / D**.

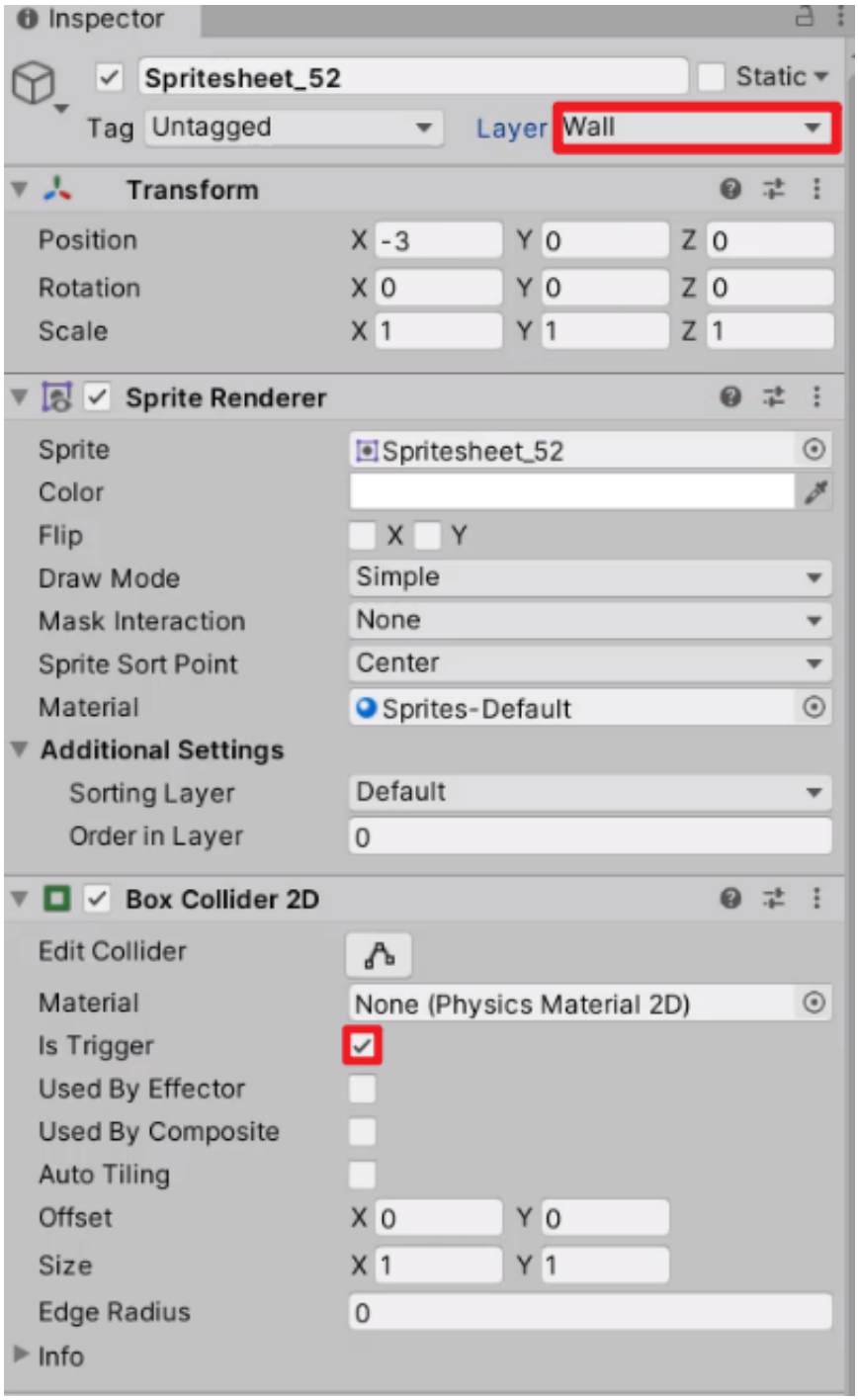


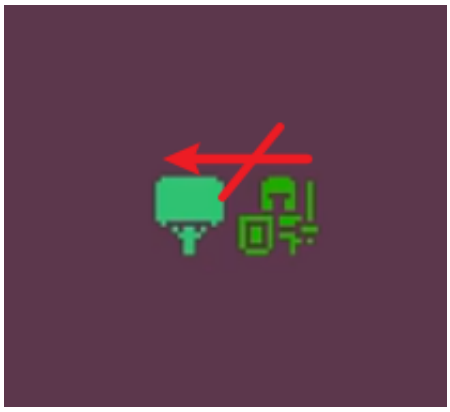
Creating A Wall

To create a wall, we can simply drag in any sprite from **Spritesheet** and assign it to the “Wall” layer.



We also need a **BoxCollider2D** component as a **trigger** in order to get detected by our player's raycast.



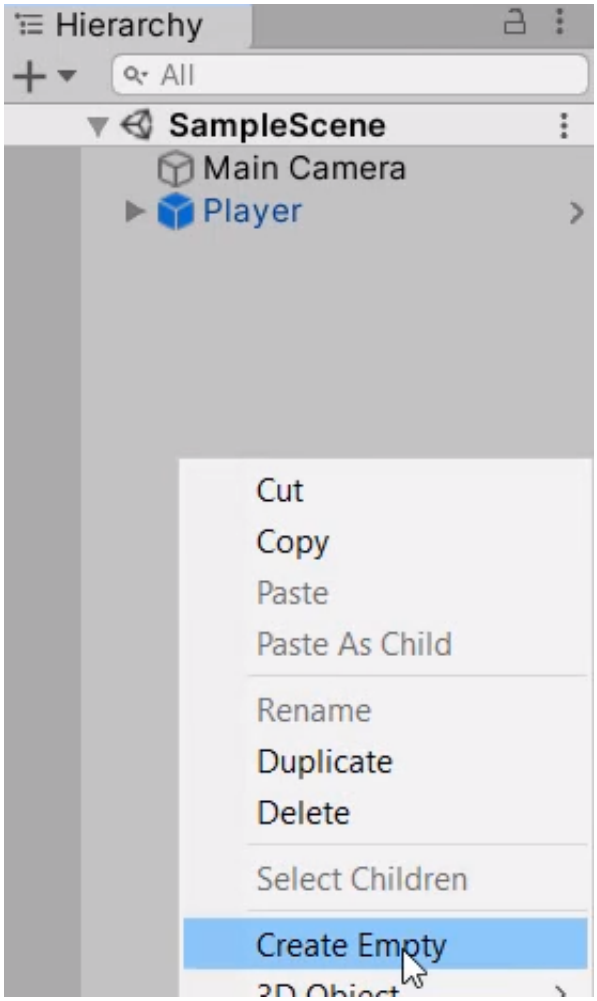


In this lesson, we're going to be creating a **room prefab**.

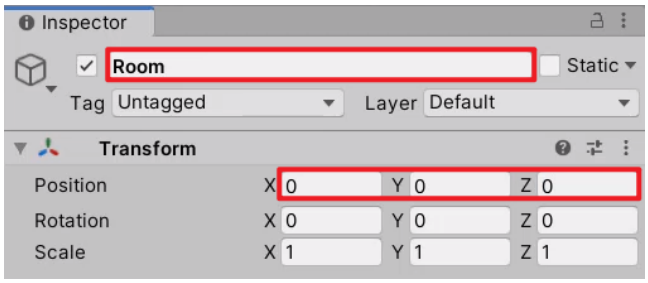
This is going to be a square room, which is going to be instantiated every time we want to place down a room for the player to walk around in. Each level will have around a dozen rooms, and each room will have walls and doors in four directions (East, West, South, North).

Setting Up GameObjects

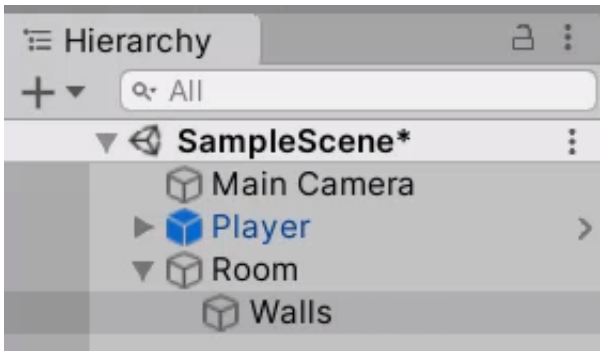
Let's start by creating a brand new object in the Hierarchy (**Right-click > Create Empty**)



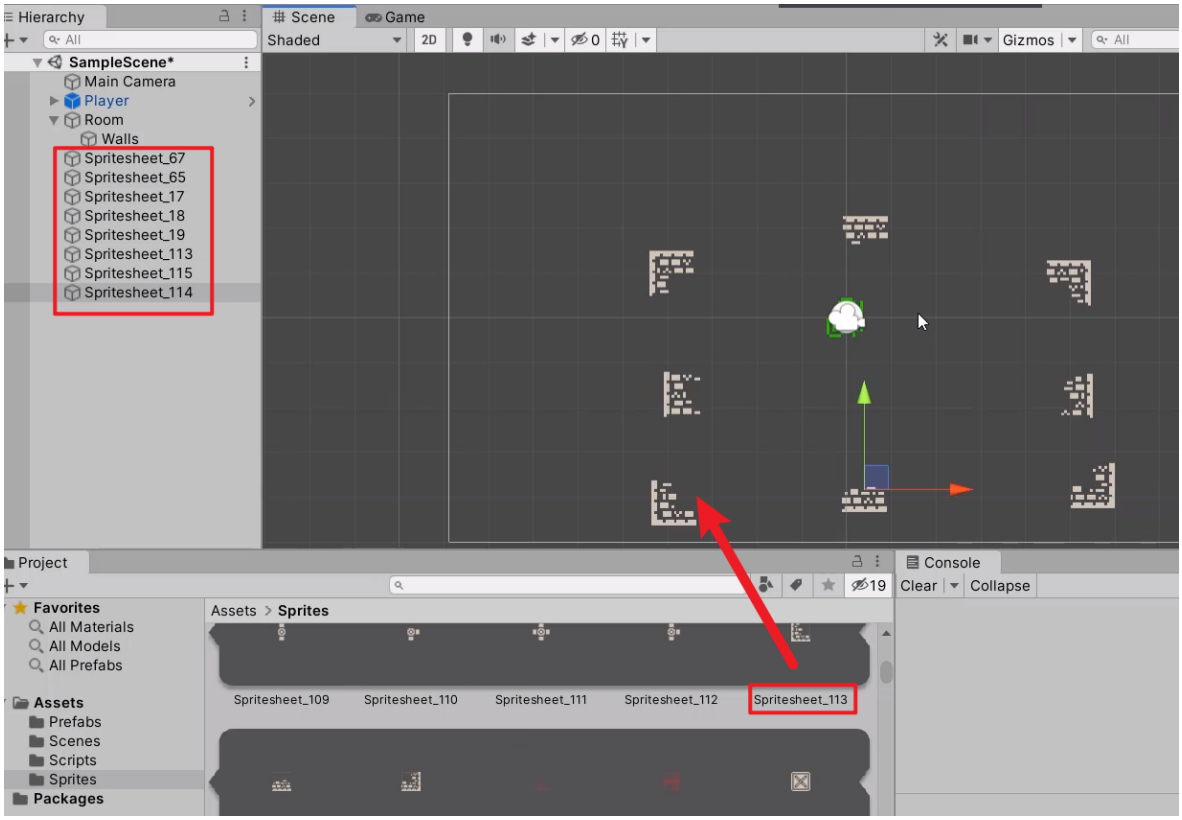
We're going to **rename** this to be "**Room**" and reset the **Position** to **(0, 0, 0)**.

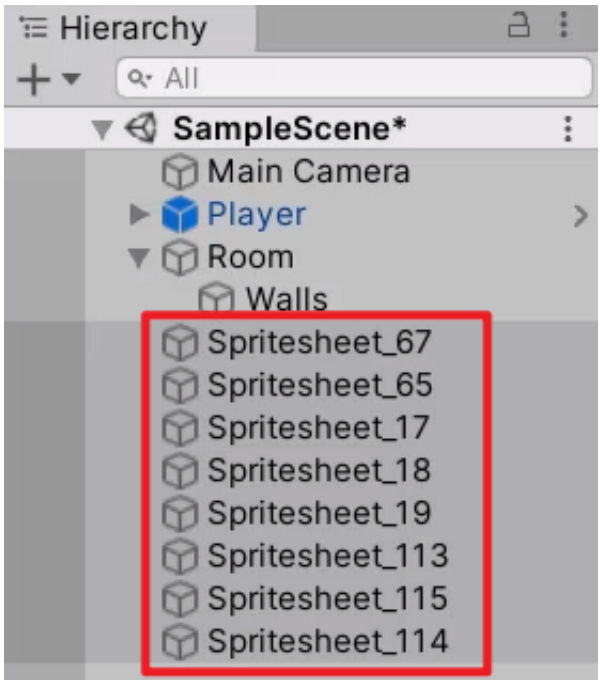


Inside the **Room** object, we're going to create another empty object named "**Walls**".

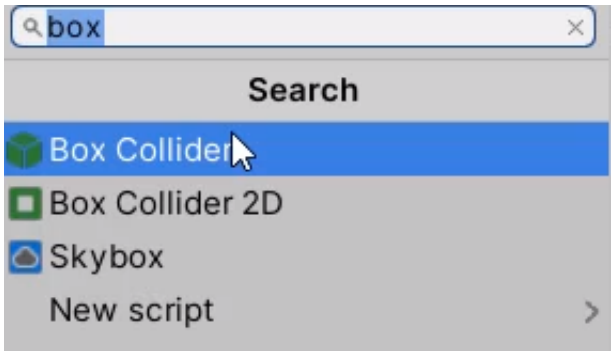


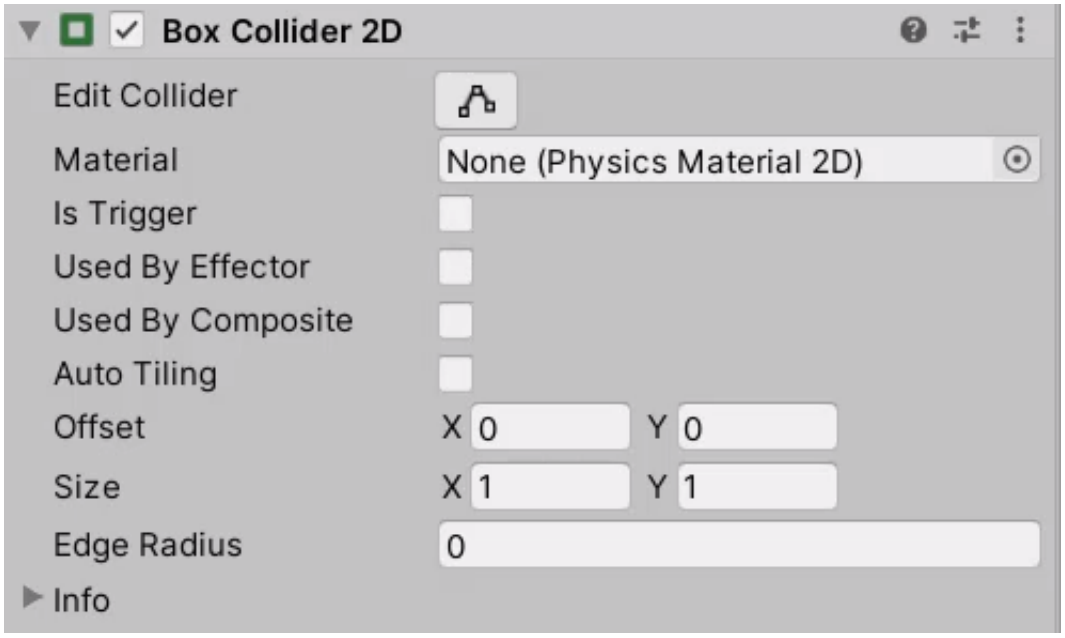
Then we can go to **Assets > Sprites** and look for sprites that we're going to use for our walls.





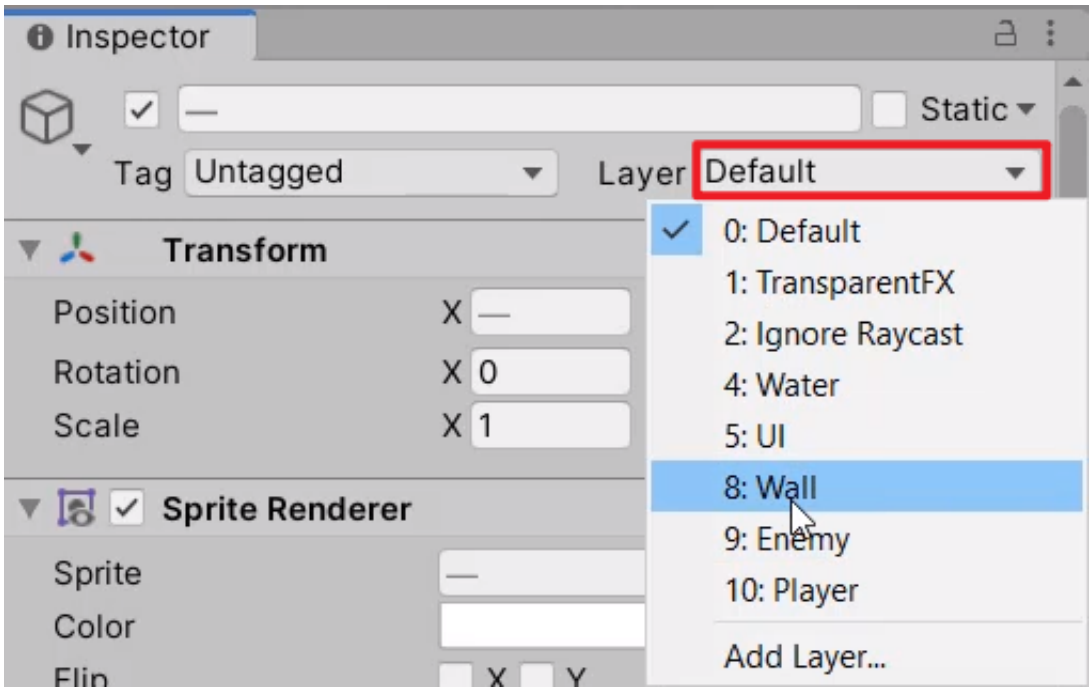
Each of these sprites should have a **Box Collider 2D** in order to interact with the player.





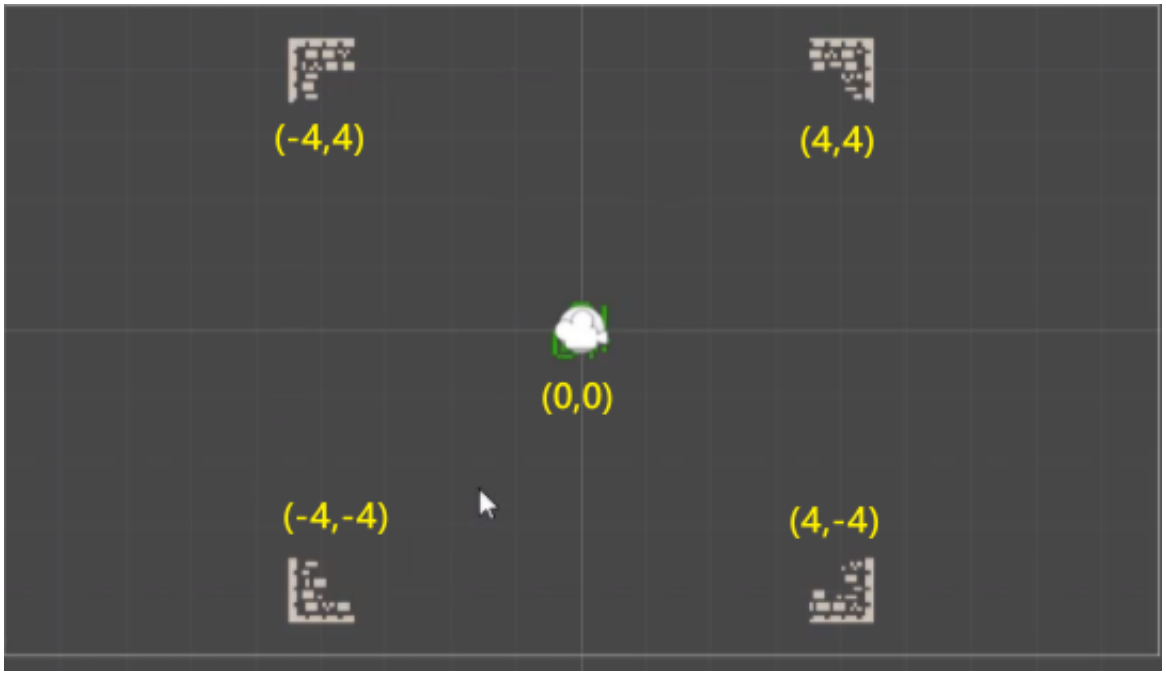
We also need to set the **Layer** to be "**Wall**", as we're going to be using that to detect if there is a

wall in front of us.

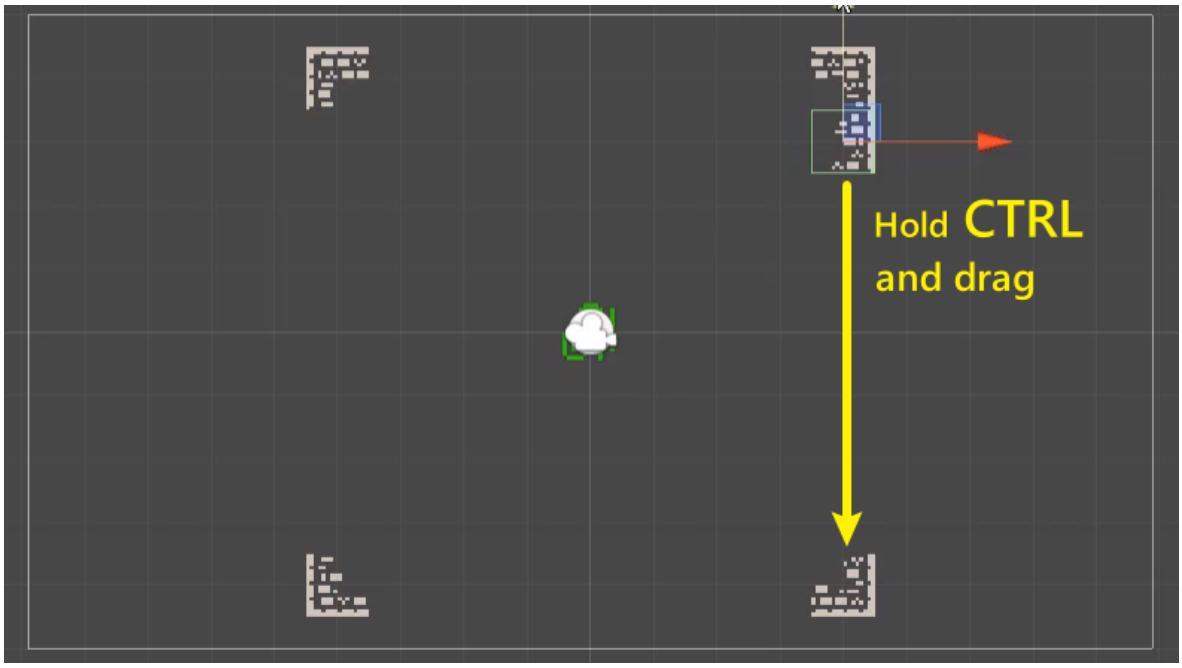


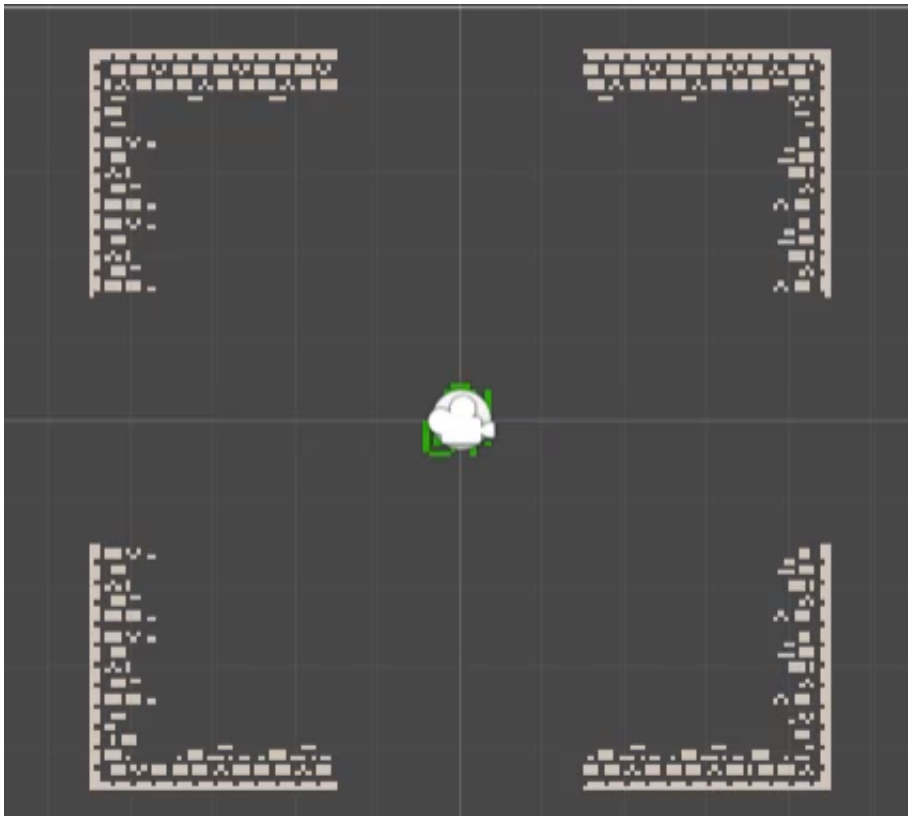
Positioning The Walls

Now what we need to do is positioning the walls around the player to start building up our **room template**. First, we need to select the corner sprites and set their position so that we have a **three-tiles gap** on each side.

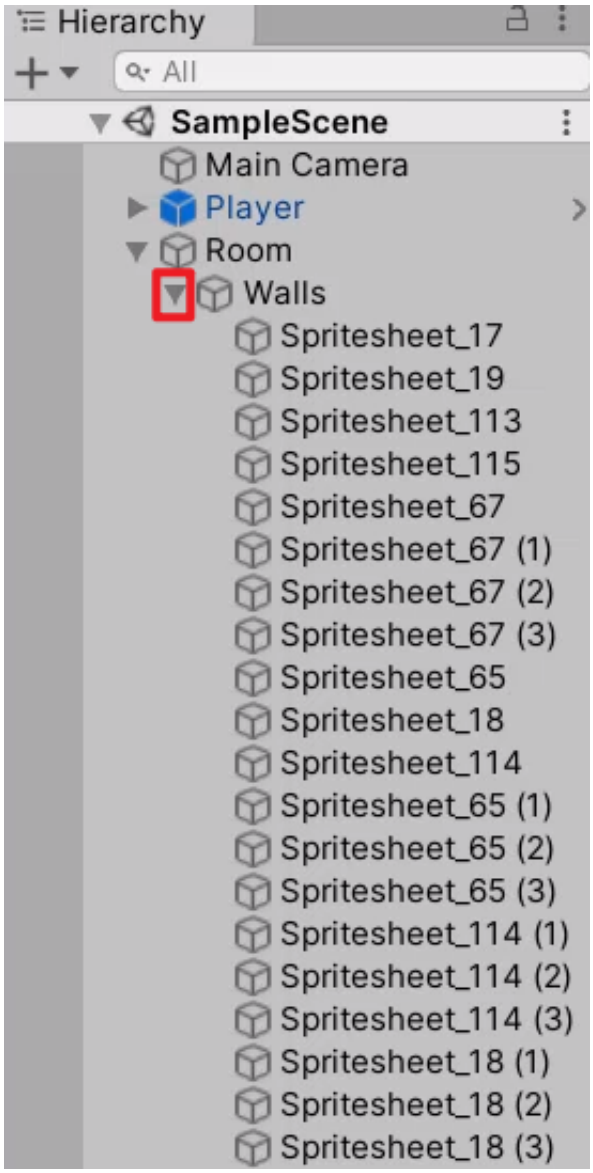


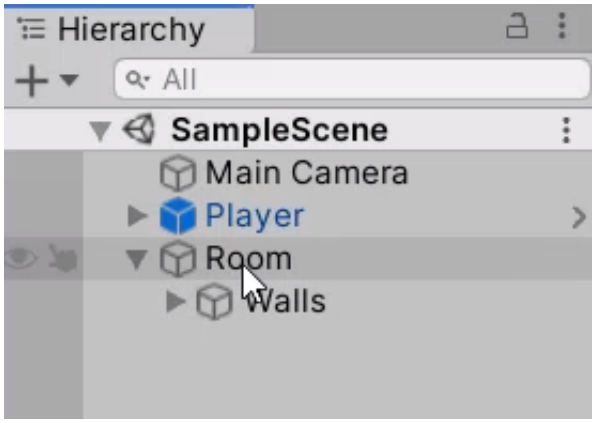
Once the corners are placed down, we can **duplicate** (**Ctrl+D** or **Cmd+D**) the sidewall sprites and fill in the gaps between each corner.





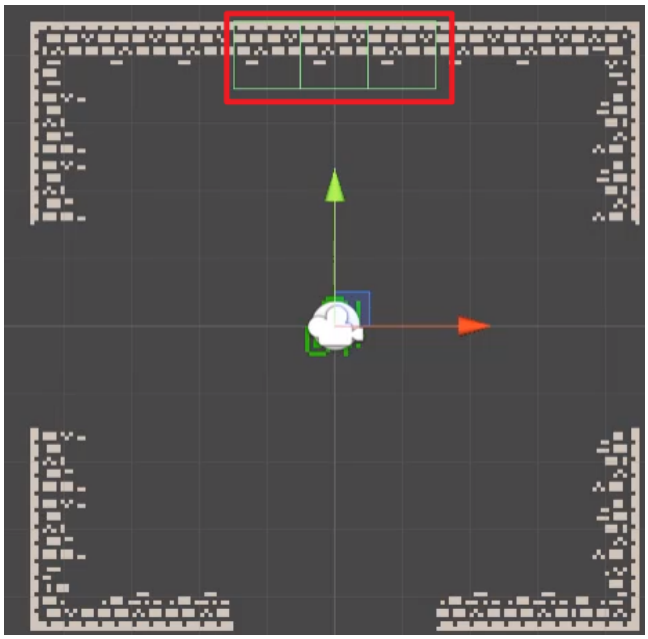
Make sure to set the sprites as **children** to the "**Walls**" GameObject, and close it up in the Hierarchy.

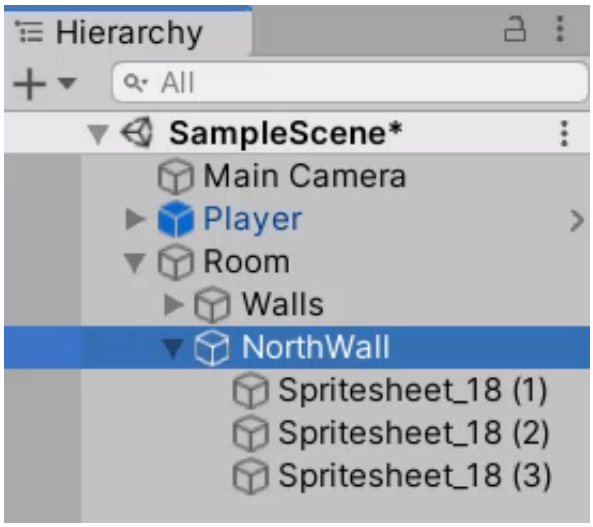




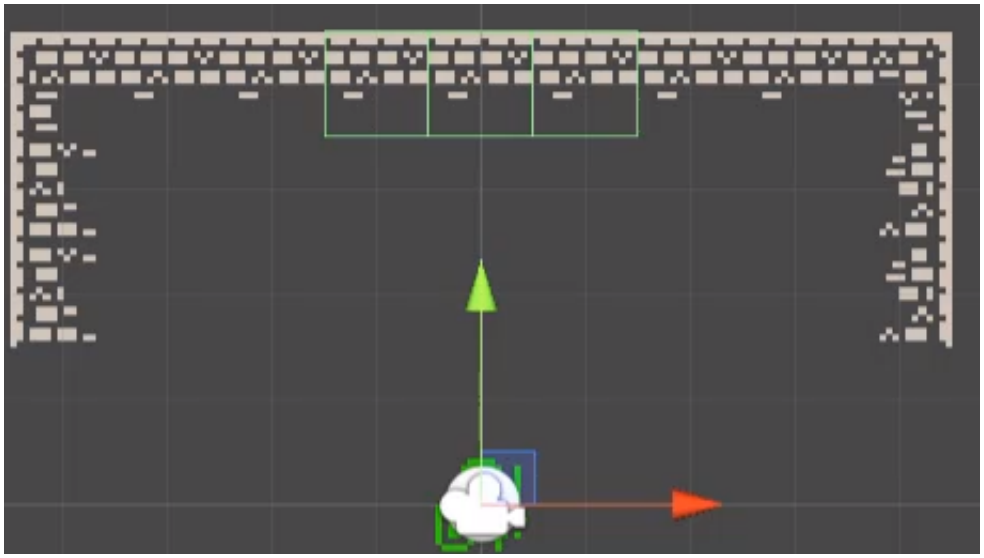
Creating A Wall

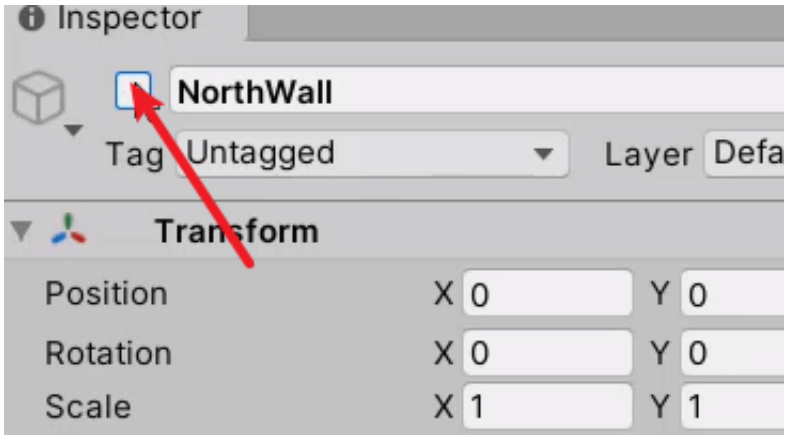
Now we're going to create a new **Empty** GameObject called "**NorthWall**", which will store a wall that is going to cover up the gap on the north side.

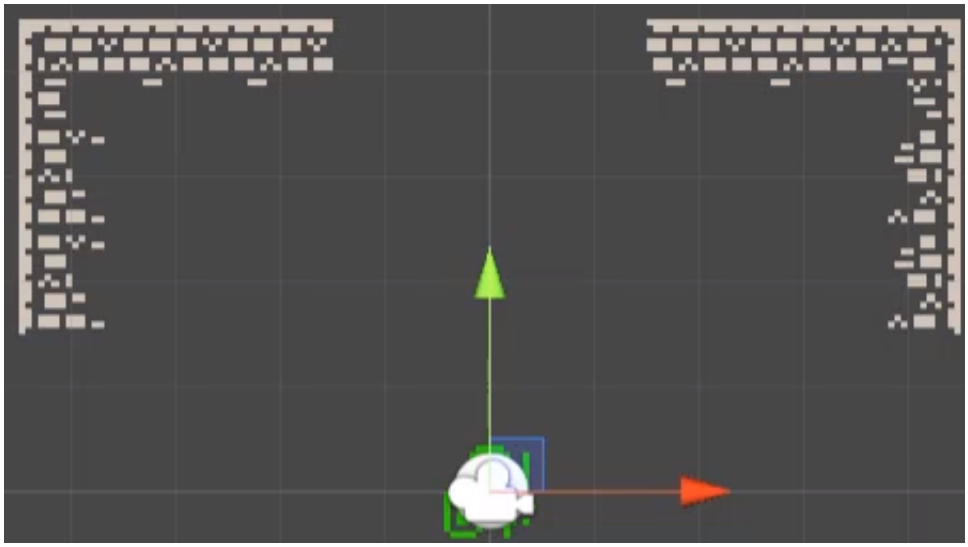




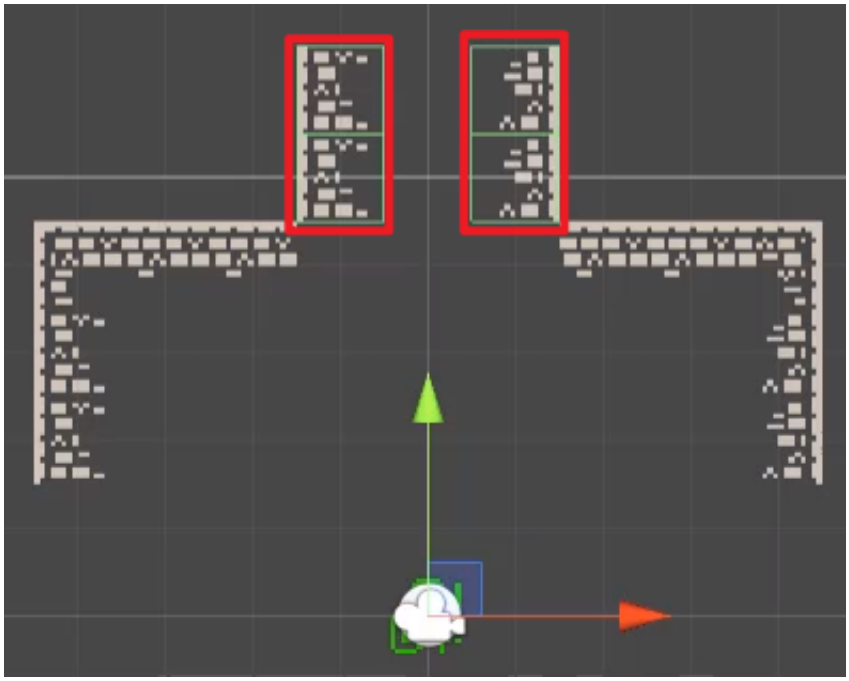
Now we can **disable** the NorthWall object and use it as a switch to toggle on/off the walls.

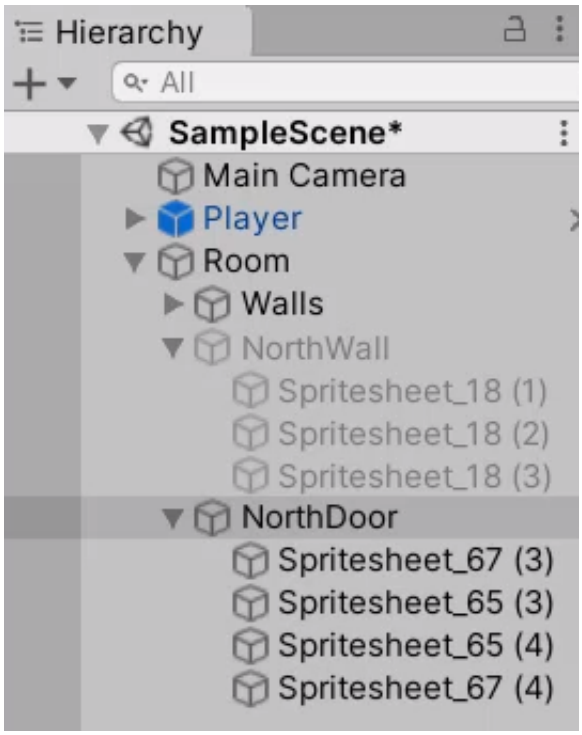




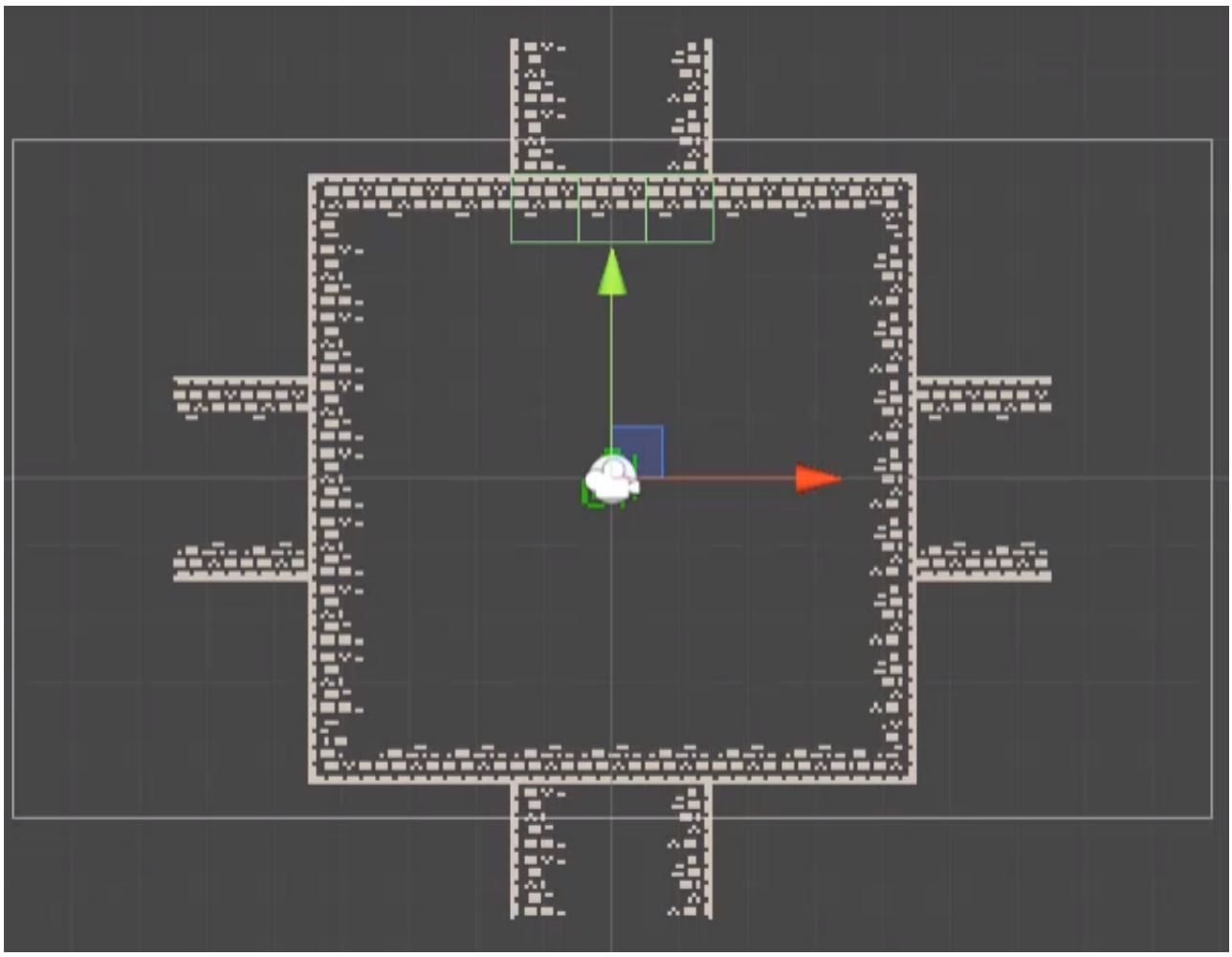


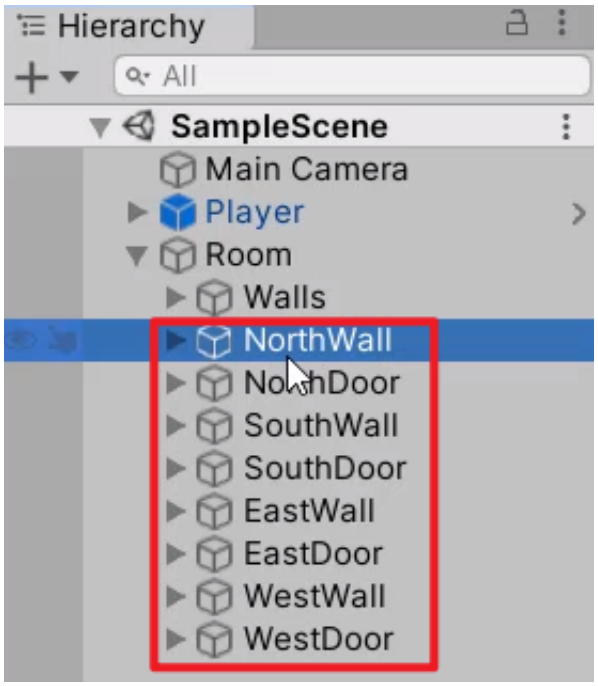
Whenever NorthWall is disabled, we're going to enable another GameObject called "**NorthDoor**", which is going to contain the sprites that lead players into another room.



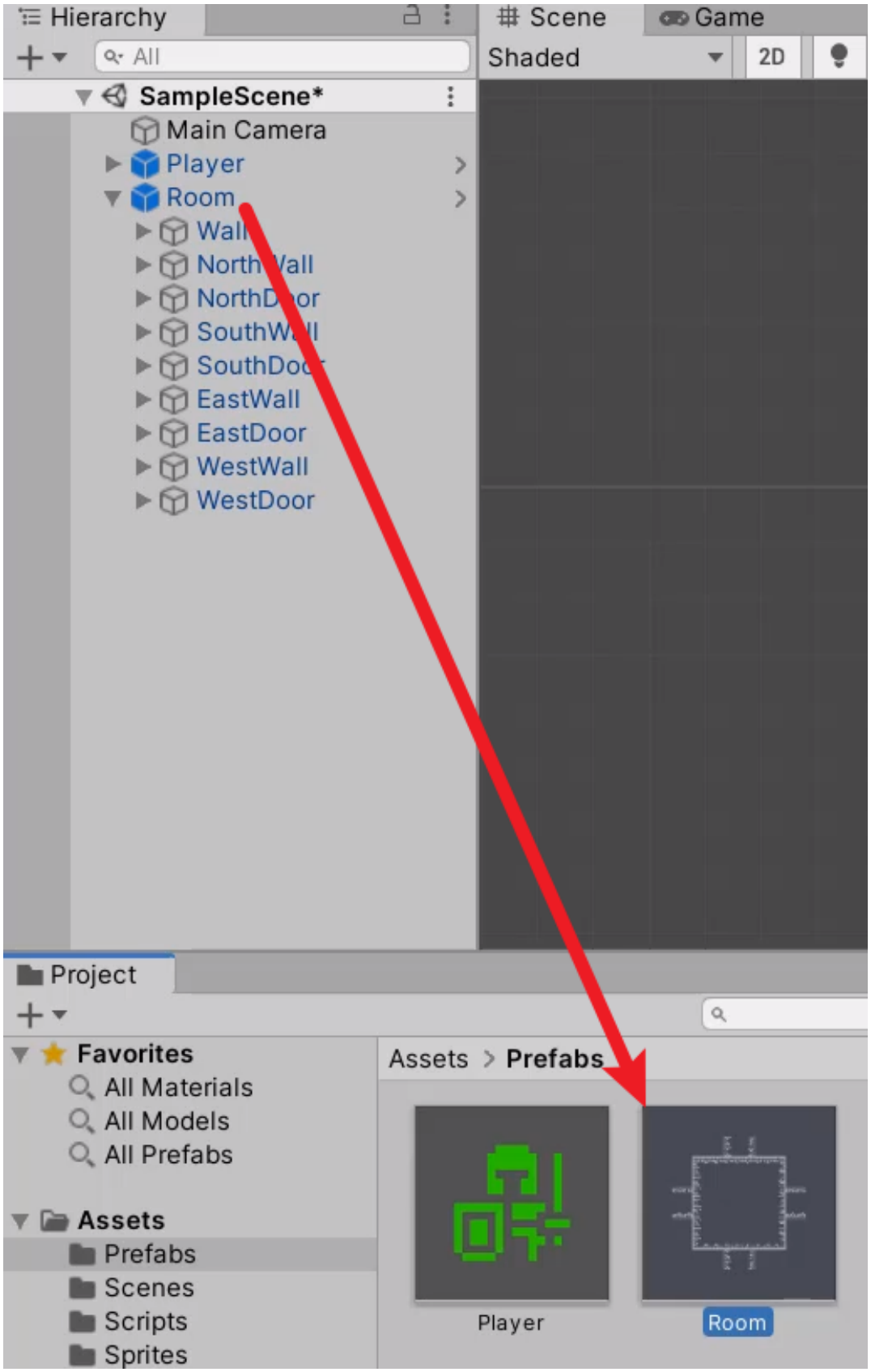


Once you've created a door and a wall for each side, you should have a total of 8 different GameObjects.





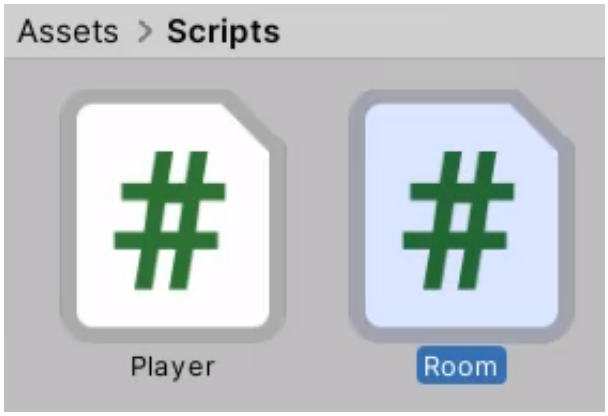
You can then save this as a **Prefab** by simply dragging the root object (“Room”) into **Assets > Prefabs**.



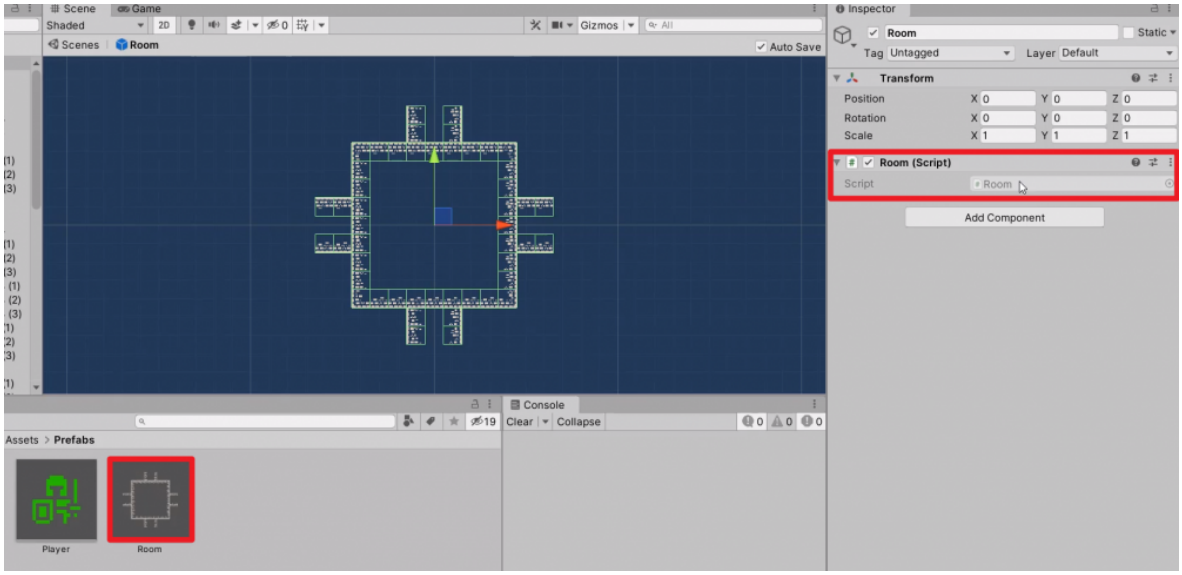
In this lesson, we're going to be setting up our **Room Script**. This is going to be a script that is attached to our room prefab, and it's going to be used by the **Map Generator**.

Setting Up A Room Script

First of all, let's create a new **C# script** called "Room" inside **Assets > Scripts**.



And then **attach** the Room script to the Room **prefab** after opening it up inside of the **Prefab editor**.



Header Attributes

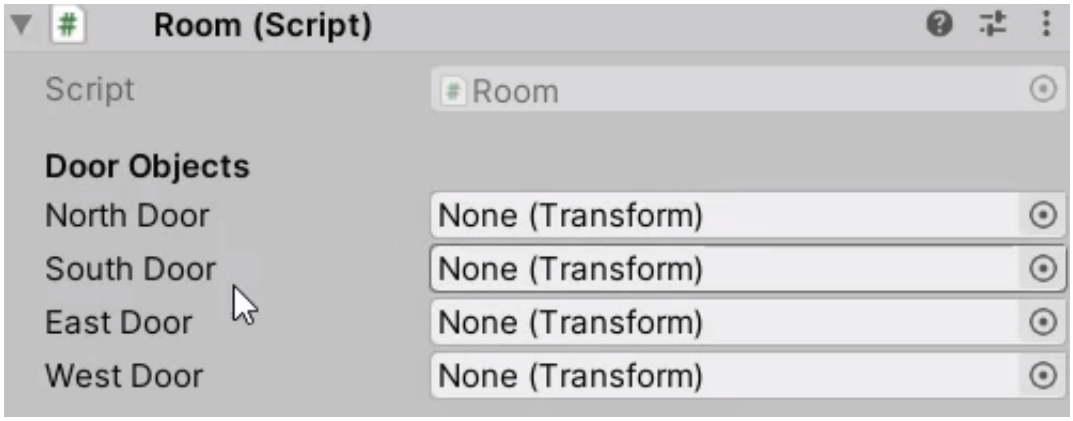
First of all, inside the script, we need to declare variables to get a reference to each object for all four directions. Instead of just having a large list of variables, we're going to make use of **Header Attributes**.

A header attribute in Unity can add a header above variable fields in the Inspector. For example, this example below will display four different Transform fields under the category of "**Door Objects**" in the Inspector.

```
using UnityEngine;

public class Room : MonoBehaviour
```



```
{   //references for the door objects
    [Header("Door Objects")]
    public Transform northDoor;
    public Transform southDoor;
    public Transform eastDoor;
    public Transform westDoor;
}
```



Along with the door objects, we also need to get references for the **walls**.

```
using UnityEngine;

public class Room : MonoBehaviour
{
    [Header("Door Objects")]
    public Transform northDoor;
    public Transform southDoor;
    public Transform eastDoor;
    public Transform westDoor;
    // references for the wall objects
    [Header("Wall Objects")]
    public Transform northWall;
    public Transform southWall;
    public Transform eastWall;
    public Transform westWall;
}
```

Next, we need to keep track of **how many tiles** (**width**, **height**) there are in order to spawn coins, enemies, etc. inside the room.

```
using UnityEngine;

public class Room : MonoBehaviour
{
    [Header("Door Objects")]
    public Transform northDoor;
    public Transform southDoor;
    public Transform eastDoor;
```



```
    public Transform westDoor;

    [Header("Wall Objects")]
    public Transform northWall;
    public Transform southWall;
    public Transform eastWall;
    public Transform westWall;

    // how many tiles are there in the room?
    [Header("Values")]
    public int insideWidth;
    public int insideHeight;
}
```

And of course, in order to spawn some object in the room, we need to know which object to spawn first. Some of these prefabs are going to be created in future lessons.

```
using UnityEngine;

public class Room : MonoBehaviour
{
    [Header("Door Objects")]
    public Transform northDoor;
    public Transform southDoor;
    public Transform eastDoor;
    public Transform westDoor;

    [Header("Wall Objects")]
    public Transform northWall;
    public Transform southWall;
    public Transform eastWall;
    public Transform westWall;

    [Header("Values")]
    public int insideWidth;
    public int insideHeight;

    // objects to instantiate
    [Header("Prefabs")]
    public GameObject enemyPrefab;
    public GameObject coinPrefab;
    public GameObject healthPrefab;
    public GameObject keyPrefab;
    public GameObject exitDoorPrefab;
}
```

Along with this, we need to create a **Vector3** list that is going to contain a number of different positions in order to give each object a unique position to spawn in. This is because we don't want to accidentally spawn something on top of another thing.

```
using UnityEngine;
```



```
public class Room : MonoBehaviour
{
    [Header("Door Objects")]
    public Transform northDoor;
    public Transform southDoor;
    public Transform eastDoor;
    public Transform westDoor;

    [Header("Wall Objects")]
    public Transform northWall;
    public Transform southWall;
    public Transform eastWall;
    public Transform westWall;

    [Header("Values")]
    public int insideWidth;
    public int insideHeight;

    [Header("Prefabs")]
    public GameObject enemyPrefab;
    public GameObject coinPrefab;
    public GameObject healthPrefab;
    public GameObject keyPrefab;
    public GameObject exitDoorPrefab;

    // list of positions to avoid instantiating new objects at
    private List<Vector3> usedPositions = new List<Vector3>();
}
```

Function Templates

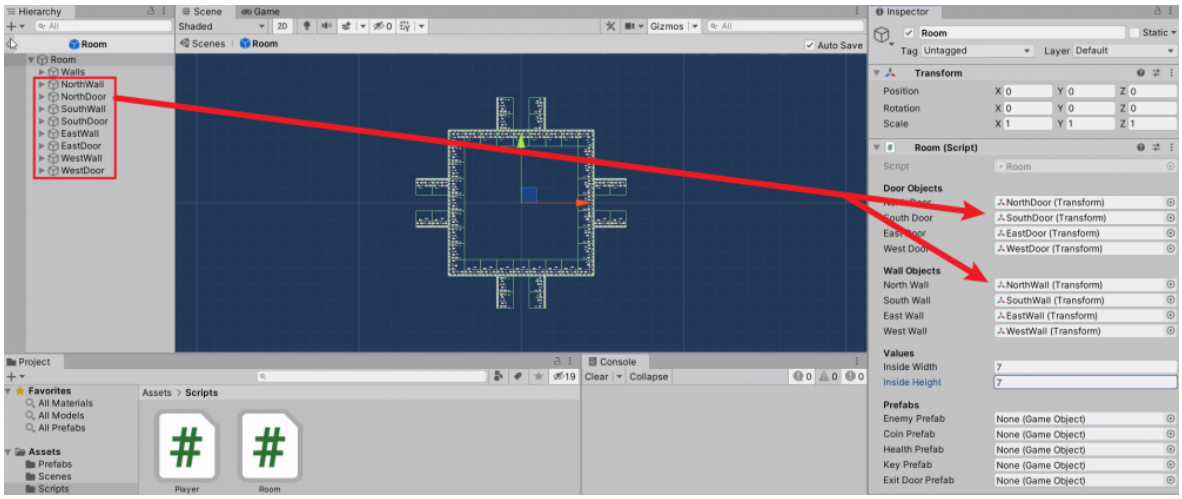
The first function template we're going to create is called "**GenerateInterior**". As the name implies, this function is going to create the coins, enemies, health packs, etc. once the room has been instantiated.

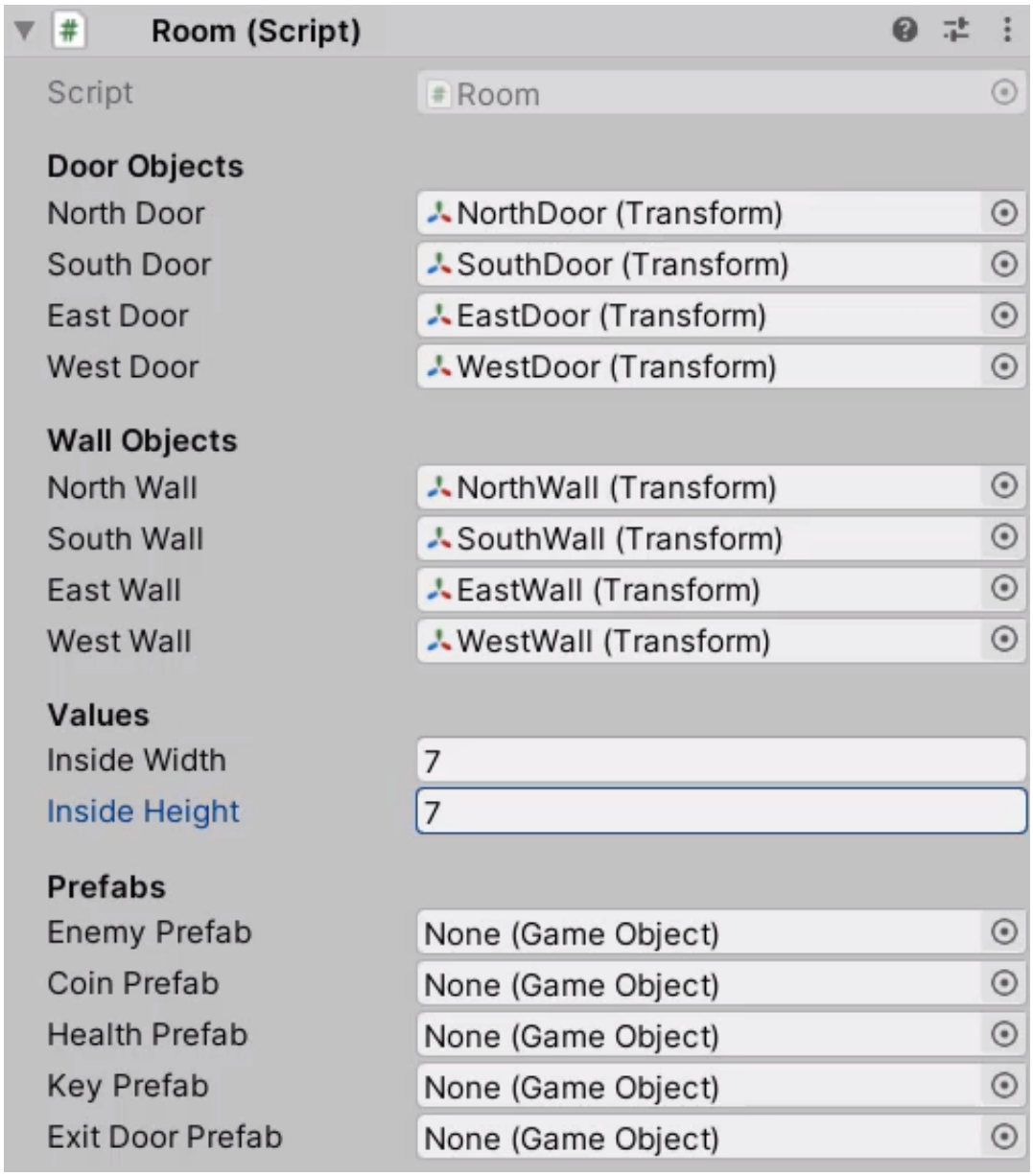
```
public void GenerateInterior()
{
    // create coins, enemies, health packs, etc.
}
```

Then we also need to create another function, which is going to take in three parameters; a **prefab** to spawn, a **minimum** and a **maximum** number of the objects to spawn. For the min/max numbers, the default values are set to be 0.

```
public void SpawnPrefab(GameObject prefab, int min = 0, int max = 0)
{

}
```


Make sure that the script is **saved**, and drag in the appropriate **Transforms** into the empty fields.



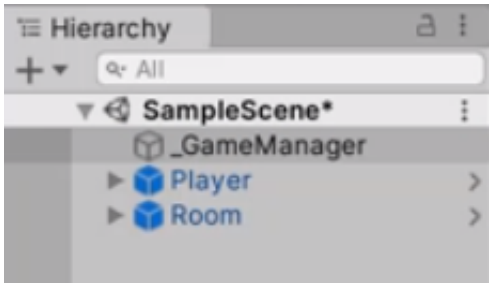


In this lesson, we're going to be creating our **Generation Script**.

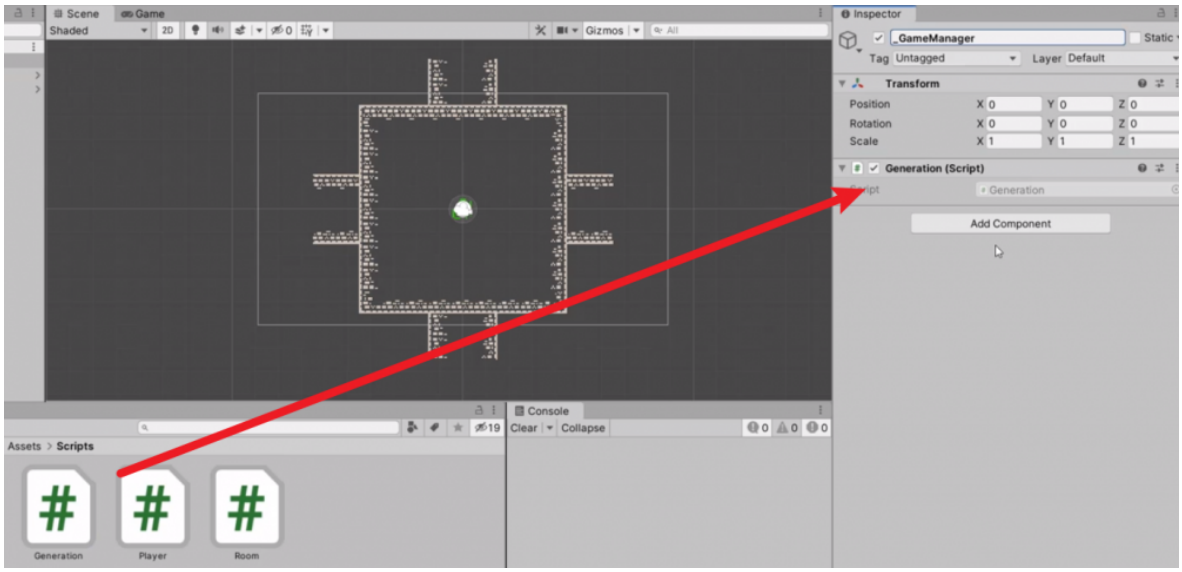
Setting Up GameManager

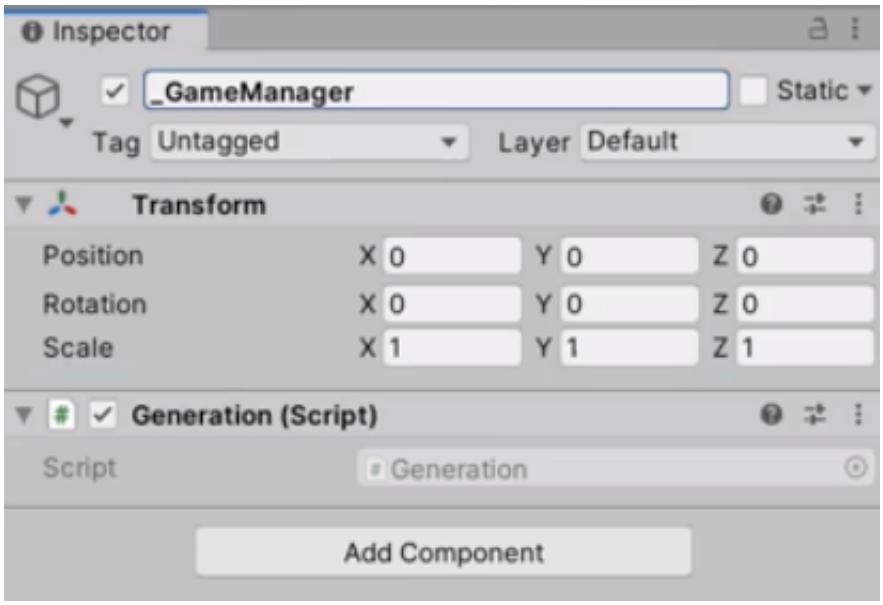
Let's go to **Assets > Scripts**, and create a new C# script called "**Generation**":

This script is going to be in charge of procedurally generating all of our levels. To add this script into the scene, we're going to create an empty object called "**_GameManager**":



... And drag this **Generation** script into the **_GameManager** object to attach it in the Inspector.





Declaring Variables

First of all, we're going to declare variables to map out our level. This includes how large our room is going to be, in terms of **width** and **height**, as well as how many rooms we are going to generate.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    public int mapWidth = 7;
    public int mapHeight = 7;
    public int roomsToGenerate = 12;
}
```

The next variable is going to be a private **integer** to keep track of the current room (**index**) we're in, and a private **boolean** to determine if we have **instantiated** the room prefabs yet.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    public int mapWidth = 7;
    public int mapHeight = 7;
    public int roomsToGenerate = 12;

    private int roomCount;
    private bool roomsInstantiated;
}
```

Along with this, we also need to keep track of where our **first room** is in order to procedurally generate additional rooms. This will be stored in a private **Vector2** variable, and we can use this to branch out left, right, up, or down.


```
using UnityEngine;

public class Generation : MonoBehaviour
{
    public int mapWidth = 7;
    public int mapHeight = 7;
    public int roomsToGenerate = 12;

    private int roomCount;
    private bool roomsInstantiated;

    // store our first room's position for procedural level generation
    private Vector2 firstRoomPos;

}
```

In a spreadsheet, we can identify a cell's position by its row and column. In Unity, we can use a **2D array** to store two indexes to access a specific element (x, y).

Our 2D array is going to store two booleans (**bool[,]**) to check whether this specific room is there or not. This will give us an overview of what these rooms are going to look like together and where they are connected to each other.

As well as this, we also need to keep track of the room prefab that we're going to be instantiating.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    public int mapWidth = 7;
    public int mapHeight = 7;
    public int roomsToGenerate = 12;

    private int roomCount;
    private bool roomsInstantiated;

    private Vector2 firstRoomPos;

    // A 2D boolean array to map out the level
    private bool[,] map;
    // the room prefab to instantiate
    public GameObject roomPrefab;
}
```

Once we instantiate our rooms, we can keep track of them inside a private **List** of class type **Room**. We're using a **List** instead of an **Array** because we'll be adding this sequentially as we play the game.

```
using UnityEngine;

public class Generation : MonoBehaviour
```



```
{
    public int mapWidth = 7;
    public int mapHeight = 7;
    public int roomsToGenerate = 12;

    private int roomCount;
    private bool roomsInstantiated;

    private Vector2 firstRoomPos;

    private bool[,] map;
    public GameObject roomPrefab;

    private List<Room> roomObjects = new List<Room>();
}
```

And finally, we're going to create a **Singleton** for this script here so that we can access it anywhere inside of our project.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    public int mapWidth = 7;
    public int mapHeight = 7;
    public int roomsToGenerate = 12;

    private int roomCount;
    private bool roomsInstantiated;

    private Vector2 firstRoomPos;

    private bool[,] map;
    public GameObject roomPrefab;

    private List<Room> roomObjects = new List<Room>();

    // creating a Singleton
    public static Generation instance;

    void Awake ()
    {
        instance = this;
    }
}
```

Setting Up Function Templates

Now what we're going to do is create some function templates.

The first function we're going to create is called **Generate**. This will be called in the process of

generating our rooms for this specific level.

```
public void Generate ()
{

}
```

And to figure out whether each and every room is here, we're going to create another function called **CheckRoom**. This function will need to take in three integer variables, which are the **x,y positions** of the room, and the number of **remaining rooms** left.

```
void CheckRoom (int x, int y, int remaining)
{

}
```

In addition, we will pass in a **Vector2** for the **general direction** and a **boolean** for whether or not this is the **first room**. This is because we'll be starting with one room in the center, and we're only going to be branching out from there. When we're branching out, we need to give it some general direction so that it doesn't suddenly go back in on itself.

```
void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool firstRoom
 = false)
{

}
```

Once everything has been created, we can then spawn the room prefabs in their correct positions, enabling or disabling the respective doors and walls.

```
void InstantiateRooms ()
{

}
```

Inside each level will be a **key** and a **door** that you have to go through. In order to go through the door, you need to collect the key first. The two will be placed as far away as possible so that the player is encouraged to explore around the map. This will be calculated inside a new function called **CalculateKeyAndExit**.

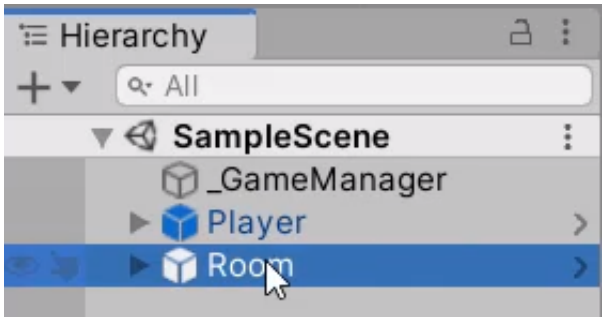
```
void CalculateKeyAndExit ()
{

}
```


Now we've got all of the functions and variables needed to start generating levels.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    public int mapWidth = 7;
    public int mapHeight = 7;
    public int roomsToGenerate = 12;

    private int roomCount;
    private bool roomsInstantiated;

    private Vector2 firstRoomPos;

    private bool[,] map;
    public GameObject roomPrefab;

    private List<Room> roomObjects = new List<Room>();

    public static Generation instance;

    void Awake ()
    {
        instance = this;
    }

    public void Generate ()
    {

    }

    void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool first
Room = false)
    {

    }

    void InstantiateRooms ()
    {

    }

    void CalculateKeyAndExit ()
    {

    }
}
```


In this lesson, we're going to begin the process of generating our rooms.

So to begin, let's delete the **Room** prefab from our scene since we want this to be spawned in with the generation script.



And then we can open up the **Generation** script to start filling in the functions.

Random Number Generator

First of all, we're going to use **Random.InitState** to send over a **Seed** as an integer, and start off the generation process inside of the **Start** function.

In a procedural generation, a **seed** is basically what you give to the game in order for it to generate a specific layout. In our game, we're going to give our players a specific seed so that they can send it to other players and play on the exact same level.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        // random seed assigned to a random number generator
        Random.InitState(74742584);
        Generate();
    }

    // called at the start of the game - begins the generation process
    public void Generate ()
    {

    }
...
}
```

Inside of the **Generate** function, we need to create a **map** variable, which is going to keep track of where each individual room is.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
```



```
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    // called at the start of the game - begins the generation process
    public void Generate ()
    {
        // create a new map of the specified size
        map = new bool[mapWidth,mapHeight];
    }
...
}
```

Return Conditions

Then what we need to do is calling the **CheckRoom** function for the first room, which will be placed in the center of the map. Since our map has a width and height of **7 x 7**, its center is going to be around **(3,3)**. Make sure to mark it as **true** for the **firstRoom** parameter.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    // called at the start of the game - begins the generation process
    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        // check to see if we can place a room in the center of the map. Since this i
s the first room, there will be no branch nor a general direction.
        CheckRoom(3, 3, 0, Vector2.zero, true);
    }

    // checks to see if we can place a room here - continues the branch in the genera
lDirection.
    void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool first
Room = false)
    {

    }
...
}
```


Then we can move on to the **CheckRoom** function, and check if we have generated all of the rooms that we need (**roomsToGenerate**). And if so, we should stop checking the rooms, or in other words, stop the function.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        CheckRoom(3, 3, 0, Vector2.zero, true);
    }

    void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool first
Room = false)
    {
        // if we have generated all of the rooms that we need, stop checking the room
s.
        if(roomCount >= roomsToGenerate)
            return;
    }
...
}
```

Another condition to stop the function is when the room that we're checking is outside the bounds of our map. This can be done by checking that the **x** and **y** parameters are in the range between **0** and the given map **width/height**.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        CheckRoom(3, 3, 0, Vector2.zero, true);
    }
```



```
    void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool first
Room = false)
    {
        // if we have generated all of the rooms that we need, stop checking the room
s.
        if(roomCount >= roomsToGenerate)
            return;

        // if this is outside the bounds of the actual map, stop the function.
        if(x < 0 || x > mapWidth - 1 || y < 0 || y > mapHeight - 1)
            return;
    }
...
}
```

Next, if the **remaining** parameter is less than or equal to **0**, we should stop the function because that simply means there is no more room to check. (Except for when this is the first room)

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        CheckRoom(3, 3, 0, Vector2.zero, true);
    }

    void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool first
Room = false)
    {
        // if we have generated all of the rooms that we need, stop checking the room
s.
        if(roomCount >= roomsToGenerate)
            return;

        // if this is outside the bounds of the actual map, stop the function.
        if(x < 0 || x > mapWidth - 1 || y < 0 || y > mapHeight - 1)
            return;

        // if this is not the first room, and there is no more room to check, stop th
e function.
        if(firstRoom == false && remaining <= 0)
            return;
    }
...
```

```
}
```

And the final return condition here is when there is already an existing room. This ensures that we don't place a room on top of another room.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        CheckRoom(3, 3, 0, Vector2.zero, true);
    }

    void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool first
Room = false)
    {
        // if we have generated all of the rooms that we need, stop checking the room
s.
        if(roomCount >= roomsToGenerate)
            return;

        // if this is outside the bounds of the actual map, stop the function.
        if(x < 0 || x > mapWidth - 1 || y < 0 || y > mapHeight - 1)
            return;

        // if this is not the first room, and there is no more room to check, stop th
e function.
        if(firstRoom == false && remaining <= 0)
            return;

        // if the given map tile is already occupied, stop the function.
        if(map[x, y] == true)
            return;
    }
...
}
```

Now, if this is the **first** room, then we need to specify the **first room position**.

If the function hasn't returned yet, we can add **1** to our **roomCount** and set the map tile to be **true**. (By default, all the map tiles are set to false until we add a room on them.)


```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        CheckRoom(3, 3, 0, Vector2.zero, true);
    }

    void CheckRoom (int x, int y, int remaining, Vector2 generalDirection, bool first
Room = false)
    {
        // if we have generated all of the rooms that we need, stop checking the room
s.
        if(roomCount >= roomsToGenerate)
            return;

        // if this is outside the bounds of the actual map, stop the function.
        if(x < 0 || x > mapWidth - 1 || y < 0 || y > mapHeight - 1)
            return;

        // if this is not the first room, and there is no more room to check, stop th
e function.
        if(firstRoom == false && remaining <= 0)
            return;

        // if the given map tile is already occupied, stop the function.
        if(map[x, y] == true)
            return;

        // if this is the first room, store the room position.
        if(firstRoom == true)
            firstRoomPos = new Vector2(x,y);

        // add one to roomCount and set the map tile to be true.
        roomCount++;
        map[x, y] = true;
    }
...
}
```

In this lesson, we're going to continue on with our **Generation** script.

Now that we have our first room set up in the center of the map, we can finally start branching off from the first room. First of all, we need to check if we're branching **North, South, East**, or **West**.

Random Direction

Let's create a temporary boolean variable called '**north**' here. If this is true, it means our room is branching to the north. Although we randomly decide whether it is true or not, it is more likely going to be **true** if the **generalDirection** is up. Let's see how it works.

First of all, **Random.value** will return a random number between 0 and 1. We're going to see if this value is greater than a certain "value", and feed the boolean result into **'north'**.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
        // 'north' will be true if the random value is greater than ...
        bool north = Random.value > (...);
...
}
```

This "value" that we will be comparing **Random.value** with is going to be either **0.2f** or **0.8f**, depending on the state of **generalDirection**,

If the **generalDirection** is up, then the value we're using is **0.2**. Otherwise, it's going to be **0.8**. That means there's going to be a higher chance that '**north**' is going to be **true**.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
        // 'north' will be true if the random value is greater than 0.2 (if generalDi
rection == up) or greater than 0.8 (when generalDirection != up).
        bool north = Random.value > (generalDirection == Vector2.up ? 0.2f : 0.8f);

    }
...
}
```

Then we can create more booleans for the **south**, **east** and **west**. Note that each **generalDirection** should have a different **Vector2** value (**down**, **left,** and **right**).

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
```



```
        bool north = Random.value > (generalDirection == Vector2.up ? 0.2f : 0.8f);

        bool south = Random.value > (generalDirection == Vector2.down ? 0.2f : 0.8f);

        bool east = Random.value > (generalDirection == Vector2.right ? 0.2f : 0.8f);

        bool west = Random.value > (generalDirection == Vector2.left ? 0.2f : 0.8f);
    }
...
}
```

So we've got all four of these set up now. Basically, **north**, **south**, **east**, and **west** can either be set to true or false based on their **general direction**.

Generating A Room

Now we need to generate a room in that specific direction. Out of **4 possible directions**, the maximum number of rooms we can have in each direction is going to be the amount we can generate (**roomsToGenerate**) divided by 4.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
        bool north = Random.value > (generalDirection == Vector2.up ? 0.2f : 0.8f);

        bool south = Random.value > (generalDirection == Vector2.down ? 0.2f : 0.8f);

        bool east = Random.value > (generalDirection == Vector2.right ? 0.2f : 0.8f);

        bool west = Random.value > (generalDirection == Vector2.left ? 0.2f : 0.8f);

        int maxRemaining = roomsToGenerate / 4;
    }
...
}
```

If this is **north**, we're going to make a room one tile above the current **(y + 1**) by calling the **CheckRoom** function. If this is the **first room**, we're going to start off a branch in all four directions.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
        bool north = Random.value > (generalDirection == Vector2.up ? 0.2f : 0.8f);

        bool south = Random.value > (generalDirection == Vector2.down ? 0.2f : 0.8f);

        bool east = Random.value > (generalDirection == Vector2.right ? 0.2f : 0.8f);
```



```
        bool west = Random.value > (generalDirection == Vector2.left ? 0.2f : 0.8f);

        int maxRemaining = roomsToGenerate / 4;

        // if north is true, make a room one tile above the current.
        if(north || firstRoom)
            CheckRoom(x, y + 1,
...
}
```

For the **remaining** value, we will give it the **maxRemaining** value if this is the first room, otherwise, we'll give it the existing **remaining parameter subtracted by 1**.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
        bool north = Random.value > (generalDirection == Vector2.up ? 0.2f : 0.8f);

        bool south = Random.value > (generalDirection == Vector2.down ? 0.2f : 0.8f);

        bool east = Random.value > (generalDirection == Vector2.right ? 0.2f : 0.8f);

        bool west = Random.value > (generalDirection == Vector2.left ? 0.2f : 0.8f);

        int maxRemaining = roomsToGenerate / 4;

        // ... the branch is going to be a max of four in this specific direction.
        if(north || firstRoom)
            CheckRoom(x, y + 1, firstRoom ? maxRemaining : remaining - 1, );
    }
...
}
```

Again, if this is the first room, then we want to manually override the **generalDirection** value to be **Vector2.up**. Otherwise, we can give the existing gerneralDirection that has been given already.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
        bool north = Random.value > (generalDirection == Vector2.up ? 0.2f : 0.8f);

        bool south = Random.value > (generalDirection == Vector2.down ? 0.2f : 0.8f);

        bool east = Random.value > (generalDirection == Vector2.right ? 0.2f : 0.8f);

        bool west = Random.value > (generalDirection == Vector2.left ? 0.2f : 0.8f);
```



```
        int maxRemaining = roomsToGenerate / 4;

        // ... the generalDirection will be Vector2.up if this is the first room.
        if(north || firstRoom)
            CheckRoom(x, y + 1, firstRoom ? maxRemaining : remaining - 1, firstRoom ?
 Vector2.up : generalDirection);
    }
...
}
```

Now we can fill in the rest of the directions– **south**, **east,** and **west**.

Make sure though that you put in the correct coordinates. For example, if we're moving **east**, we'd have **(x + 1, y)** whereas if we're moving **south**, we'd have **(x, -1)**. Also remember to keep in mind the **Vector2** value (**down**, **right**, **left**) that we're specifying for the **generalDirection**.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
        bool north = Random.value > (generalDirection == Vector2.up ? 0.2f : 0.8f);

        bool south = Random.value > (generalDirection == Vector2.down ? 0.2f : 0.8f);

        bool east = Random.value > (generalDirection == Vector2.right ? 0.2f : 0.8f);

        bool west = Random.value > (generalDirection == Vector2.left ? 0.2f : 0.8f);

        int maxRemaining = roomsToGenerate / 4;

        if(north || firstRoom)
            CheckRoom(x, y + 1, firstRoom ? maxRemaining : remaining - 1, firstRoom ?
 Vector2.up : generalDirection);

        if(south || firstRoom)
            CheckRoom(x, y - 1, firstRoom ? maxRemaining : remaining - 1, firstRoom ?
 Vector2.down : generalDirection);

        if(east || firstRoom)
            CheckRoom(x + 1, y, firstRoom ? maxRemaining : remaining - 1, firstRoom ?
 Vector2.right : generalDirection);

        if(west || firstRoom)
            CheckRoom(x - 1, y, firstRoom ? maxRemaining : remaining - 1, firstRoom ?
 Vector2.left : generalDirection);

    }
...
}
```


And from here, we can instantiate these rooms. So inside the **Generate** function, we're going to call the **InstantiateRooms** function just after we call the **CheckRoom** function.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        CheckRoom(3, 3, 0, Vector2.zero, true);
        InstantiateRooms();
    }
...
}
```

And finally, we can set the **player's** position to be inside the first room.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
...
    void Start()
    {
        Random.InitState(74742584);
        Generate();
    }

    public void Generate ()
    {
        map = new bool[mapWidth,mapHeight];
        CheckRoom(3, 3, 0, Vector2.zero, true);
        InstantiateRooms();
        // Find the player in the scene, and position them inside the first room.
        FindObjectOfType<Player>().transform.position = firstRoomPos * 12;
    }
...
}
```

Note that, by **multiplying 12**, the tile coordinates are converted into global coordinates.

In this lesson, we're going to begin the process of **instantiating** our room prefabs.

Let's open up our **Generation** script and start working on the **InstantiateRooms** function.

Instantiating Rooms

First of all, we need to check if the rooms are already instantiated, and if so, return. Otherwise, we can set **roomsInstantiated** to be true so that the function is called only once.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    // once the rooms have been decided, they will be spawned in and setup
    void InstantiateRooms ()
    {
        // call this function once
        if(roomsInstantiated)
            return;

        roomsInstantiated = true;
    }
...
}
```

Next, we're going to be looping through each and every element inside of our **map 2D array**, and check if it is equal to **false**. If it's false, we don't need to do anything because it doesn't exist, so we'll use '**continue**' to skip over to the next one.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    void InstantiateRooms ()
    {
        if(roomsInstantiated)
            return;

        roomsInstantiated = true;

        // loop through each element inside of our map array
        for(int x = 0; x < mapWidth; ++x)
        {
             for(int y = 0; y < mapHeight; ++y)
             {
                  // if the map tile doesn't exist, skip over to the next one.
                  if(map[x, y] == false)
                       continue;
             }
        }
    }
...
}
```

Otherwise, if it's equal to true, then we want to **instantiate** our new room prefab. The position would be the current **x** and **y multiplied by 12**, since that is the width in terms of Unity unit.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    // once the rooms have been decided, they will be spawned in and setup
    void InstantiateRooms ()
    {
        if(roomsInstantiated)
            return;

        roomsInstantiated = true;

        for(int x = 0; x < mapWidth; ++x)
        {
             for(int y = 0; y < mapHeight; ++y)
             {
                  if(map[x, y] == false)
                       continue;

                  // instantiate a new room prefab
                  GameObject roomObj = Instantiate(roomPrefab, new Vector3(x, y, 0) *
 12, Quaternion.identity);
             }
        }
    }
...
}
```

We then need to get a reference to the **Room** script of the newly created room object. With that reference, we can check if there is a room nearby us and decide which gate to open.

For example, if there is a room to the **left** of us, we can enable the **west** gate. If there is a room **above** us, we can enable the **north** gate.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    // once the rooms have been decided, they will be spawned in and setup
    void InstantiateRooms ()
    {
        if(roomsInstantiated)
            return;

        roomsInstantiated = true;

        for(int x = 0; x < mapWidth; ++x)
        {
```

```
            for(int y = 0; y < mapHeight; ++y)
            {
                if(map[x, y] == false)
                    continue;

                GameObject roomObj = Instantiate(roomPrefab, new Vector3(x, y, 0) *
 12, Quaternion.identity);
                // get a reference to the Room script of the new room object
                Room room = roomObj.GetComponent<Room>();

                // if we're within the boundary of the map, AND if there is room ab
ove us
                if(y < mapHeight - 1 && map[x, y, 1] == true)
                {
                    // enable the north door and disable the north wall.
                    room.northDoor.gameObject.SetActive(true);
                    room.northWall.gameObject.SetActive(false);
                }
            }
        }
    }
...
}
```

We can now enable or disable the corresponding door and wall on all four directions, if there is an adjacent tile/room.

```
using UnityEngine;

public class Generation : MonoBehaviour
{
    // once the rooms have been decided, they will be spawned in and setup
    void InstantiateRooms ()
    {
        if(roomsInstantiated)
            return;

        roomsInstantiated = true;

        for(int x = 0; x < mapWidth; ++x)
        {
            for(int y = 0; y < mapHeight; ++y)
            {
                if(map[x, y] == false)
                    continue;

                GameObject roomObj = Instantiate(roomPrefab, new Vector3(x, y, 0) *
 12, Quaternion.identity);
                // get a reference to the Room script of the new room object
                Room room = roomObj.GetComponent<Room>();

                // if we're within the boundary of the map, AND if there is room ab
ove us
```



```
                if(y < mapHeight - 1 && map[x, y, 1] == true)
                {
                     // enable the north door and disable the north wall.
                     room.northDoor.gameObject.SetActive(true);
                     room.northWall.gameObject.SetActive(false);
                }

                // if we're within the boundary of the map, AND if there is room be
low us
                if(y > 0 && map[x, y - 1] == true)
                {
                     // enable the south door and disable the south wall.
                     room.southDoor.gameObject.SetActive(true);
                     room.southWall.gameObject.SetActive(false);
                }

                // if we're within the boundary of the map, AND if there is room to
 the right of us
                if(x < mapWidth - 1 && map[x + 1, y] == true)
                {
                     // enable the east door and disable the east wall.
                     room.eastDoor.gameObject.SetActive(true);
                     room.eastWall.gameObject.SetActive(false);
                }

                // if we're within the boundary of the map, AND if there is room to
 the left of us
                if(x > 0 && map[x - 1, y] == true)
                {
                     // enable the west door and disable the west wall.
                     room.westDoor.gameObject.SetActive(true);
                     room.westWall.gameObject.SetActive(false);
                }

            }
        }
    }
...
}
```

Generating The Interior

Now that we've created all the rooms, we need to generate the interior. We're going to leave our first room empty by calling the **GenerateInterior** function after checking that the **firstRoomPos** is **not** equal to our current **x** and **y** value.

Once that's done, we can add our room to the **roomObjects** list and then place the key and exit in the level. Make sure to call the **CalculateKeyAndExit** function outside the loop, because we only need to place them once for the entire level.

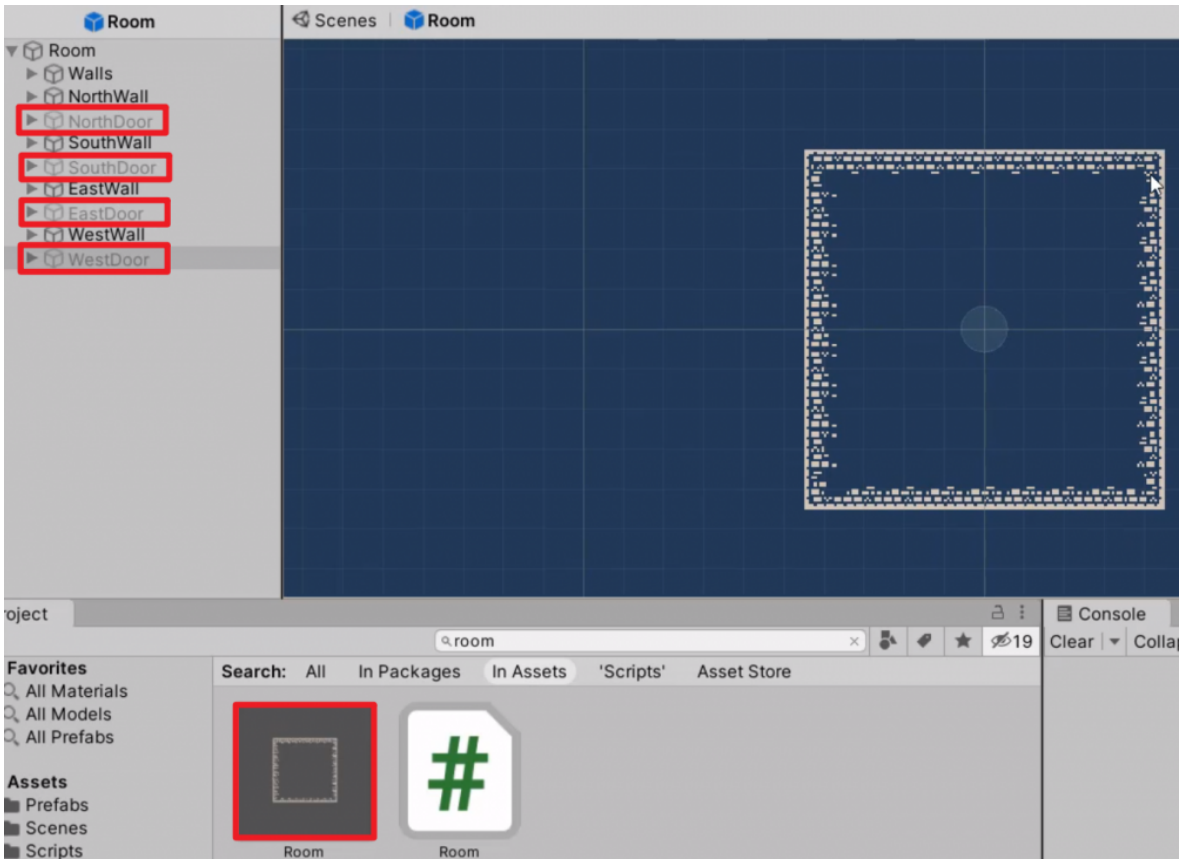
```
using UnityEngine;

public class Generation : MonoBehaviour
```



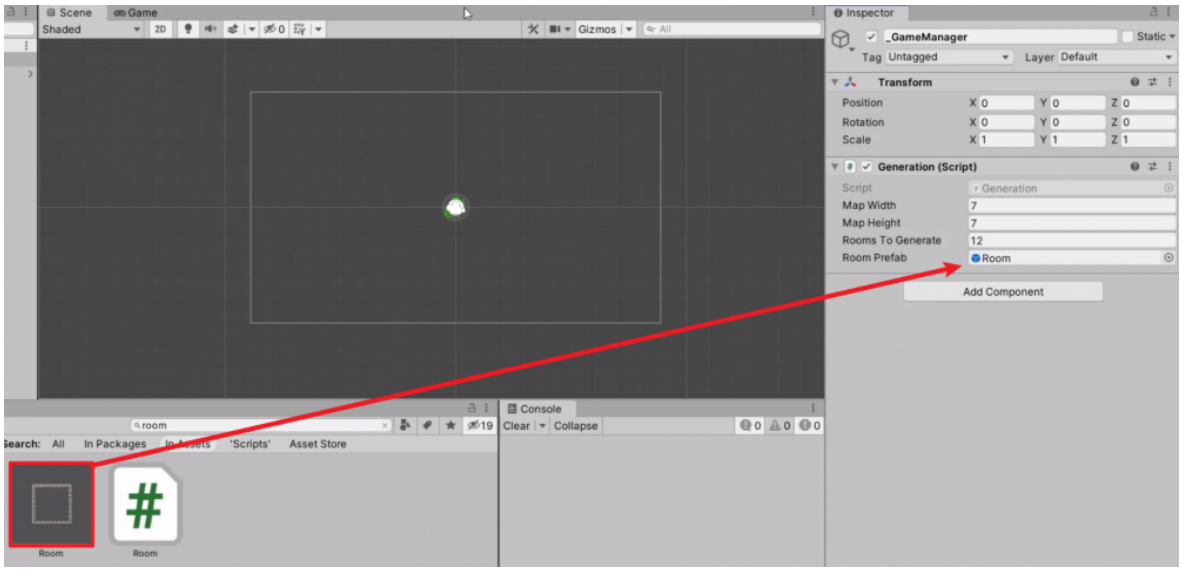
```
{
    // once the rooms have been decided, they will be spawned in and setup
    void InstantiateRooms ()
    {
        if(roomsInstantiated)
            return;

        roomsInstantiated = true;

        for(int x = 0; x < mapWidth; ++x)
        {
             for(int y = 0; y < mapHeight; ++y)
             {
                  if(map[x, y] == false)
                       continue;

                  GameObject roomObj = Instantiate(roomPrefab, new Vector3(x, y, 0) *
 12, Quaternion.identity);
                  Room room = roomObj.GetComponent<Room>();

                  if(y < mapHeight - 1 && map[x, y, 1] == true)
                  {
                       // enable the north door and disable the north wall.
                       room.northDoor.gameObject.SetActive(true);
                       room.northWall.gameObject.SetActive(false);
                  }

                  if(y > 0 && map[x, y - 1] == true)
                  {
                       // enable the south door and disable the south wall.
                       room.southDoor.gameObject.SetActive(true);
                       room.southWall.gameObject.SetActive(false);
                  }

                  if(x < mapWidth - 1 && map[x + 1, y] == true)
                  {
                       // enable the east door and disable the east wall.
                       room.eastDoor.gameObject.SetActive(true);
                       room.eastWall.gameObject.SetActive(false);
                  }

                  if(x > 0 && map[x - 1, y] == true)
                  {
                       // enable the west door and disable the west wall.
                       room.westDoor.gameObject.SetActive(true);
                       room.westWall.gameObject.SetActive(false);
                  }

                  // if this is not the first room, call GenerateInterior().
                  if(firstRoomPos != new Vector2(x, y))
                       room.GenerateInterior();

                  // add the room to the roomObjects list
                  roomObjects.Add(room);
             }
```

```
        }

        // after looping through every element inside the map array, call CalculateKe
yAndExit().
        CalculateKeyAndExit();
    }

    // places the key and exit in the level
    void CalculateKeyAndExit ()
    {
    }

...
}
```

Since our script will automatically enable the doors, we can now disable them. Let's **double-click** on the **Room** prefab to open it up inside the Prefab Editor, and **disable** all the door objects.

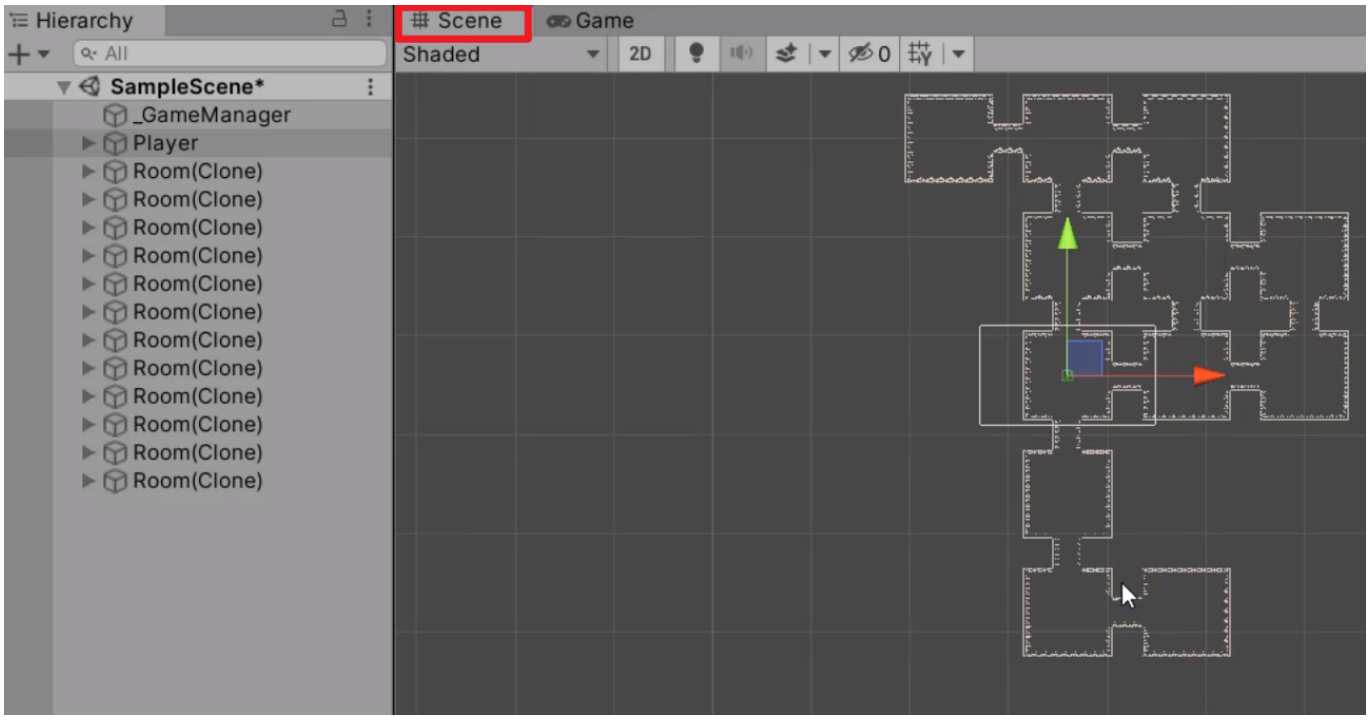


Finally, we can drag it into the '**Room Prefab**' field of the **Generation** script.



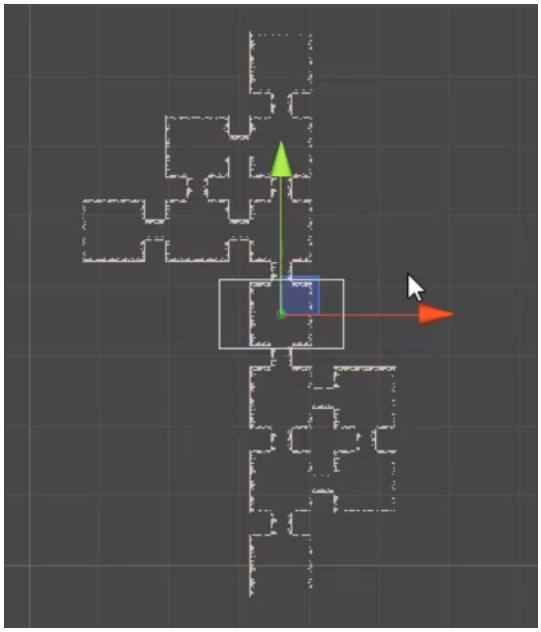
If you press **Play**, you'll see that there are a few room prefabs generated by the script.

You can also see the entire layout of the level by opening up the **Scene** view.



We can generate a different layout by feeding any random number as a **seed** value. This is something that we can add to the menu as an option.

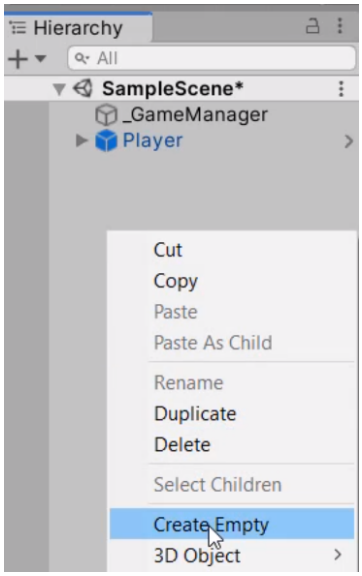
```
void Start ()
{
	// enter any number
	Random.InitState(123123);
	Generate();
}
```



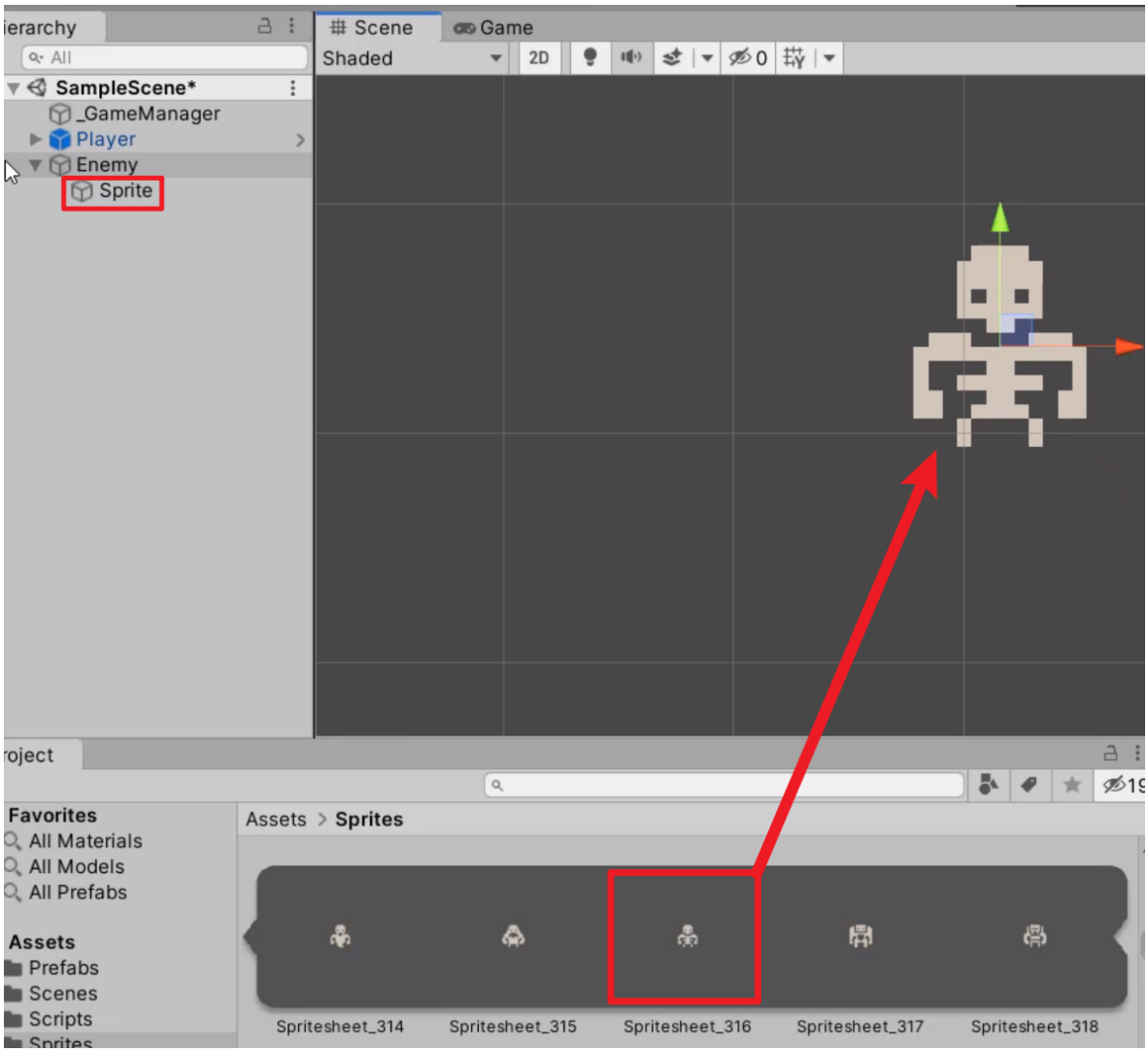
In this lesson, we're going to be working on setting up our enemy object.

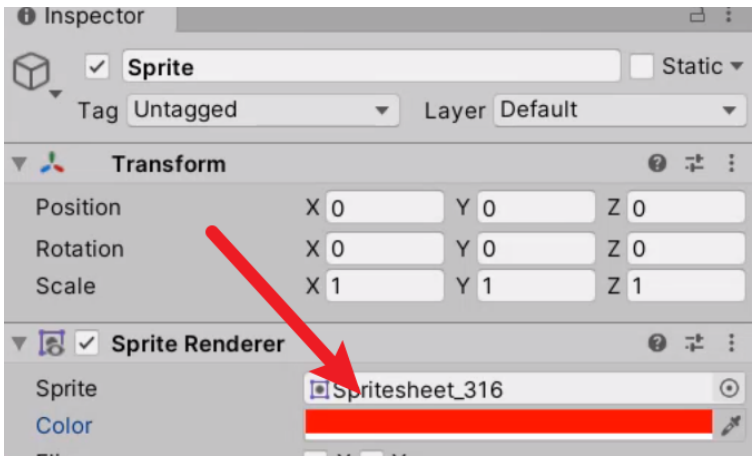
Creating Enemy GameObject

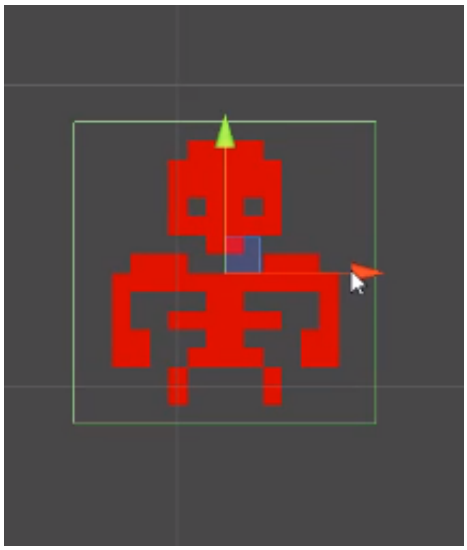
First of all, we're going to right-click on the **Hierarchy > Create Empty** to create an empty object called "**Enemy**".



And then we can add a **Sprite** as a **child** of the enemy object. Feel free to change the color using the **Sprite Renderer** component.

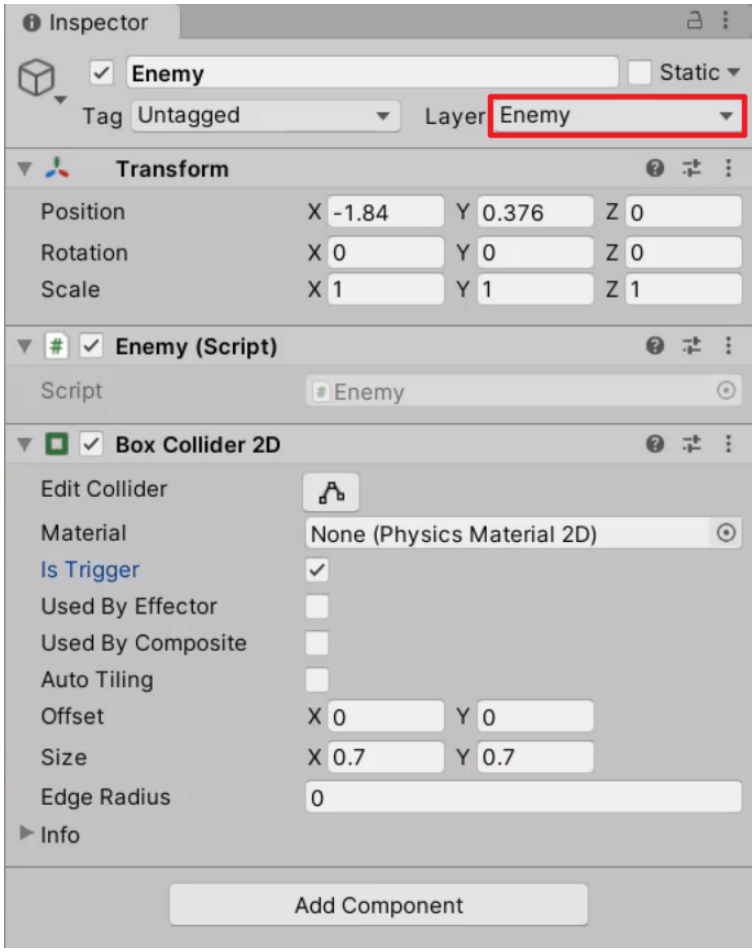




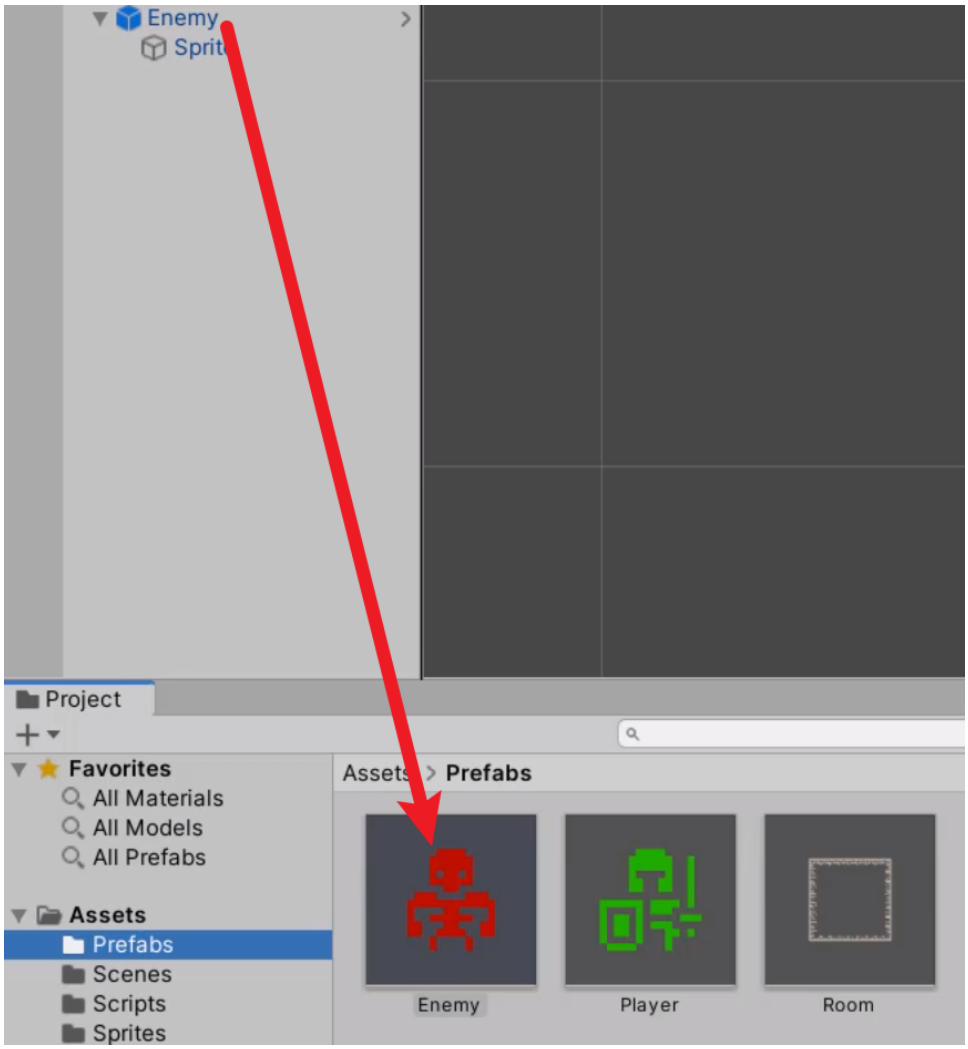


We're then going to add the **Enemy** script and a **Box Collider 2D** component to the **Enemy** object.

Make sure to set the **Layer** to be "**Enemy**" as well.



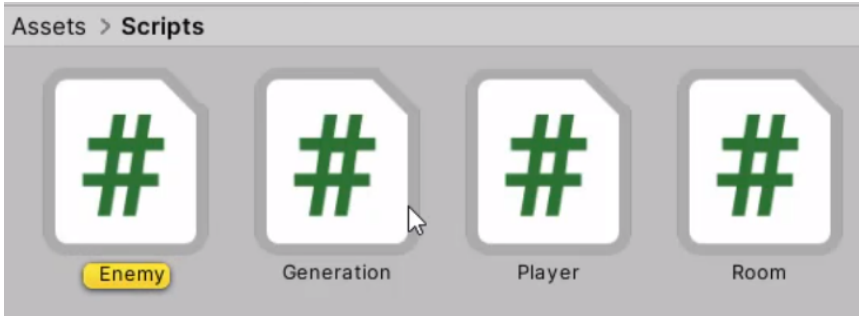
Finally, we can **save** this as a **prefab** by dragging it into **Assets > Prefabs**.



Now let's start actually scripting its functionality.

Scripting Enemy

To begin, **double-click** on the **Enemy** script to open it up inside of Visual Studio.



We're going to start by creating a few variables based on the following questions:

- How much **health** does our enemy have? (**int**)
- How much **damage** will our enemy do? (**int**)
- What object will our enemy drop when we defeat them? (**GameObject**)
- Which colour will their sprite flash when they take damage? (**SpriteRenderer**)

```
// reference to the Player class
public Player player;

public int health;
public int damage;

public GameObject deathDropPrefab;
public SpriteRenderer sr;
```

At the start of the game, we're going to **find** the player and assign that to the **player** variable.

```
void Start()
{
     player = FindObjectOfType<Player>();
}
```

First of all, we're not going to have an Update function, since we want every enemy instance to move based on a script called "**EnemyManager**". So whenever the player moves, the EnemyManager will go through every single enemy and call the **Move** function.

Inside the **Move** function, we're going to make it that there's a **50% chance** the enemy is going to move if the player moves.

```
public void Move ()
{
    if(Random.value < 0.5f)
        return;
}
```

Before moving the enemy, we first need to figure out which direction can we move it. So we're going to make two temporary variables: **Vector3** for direction and **bool** for availability.

While **canMove** is false, we're going to get a random direction by calling a new function called **GetRandomDirection**, which returns a random direction as **Vector3**.

```
public void Move ()
{
    if(Random.value < 0.5f)
        return;

    Vector3 dir = Vector3.zero;
    bool canMove = false;

    // before moving the enemy, get a random direction to move to
    while(canMove == false)
    {
        GetRandomDirection();
    }
```



```
}

Vector3 GetRandomDirection ()
{
    // Get a random number between 0 and 4
    int ran = Random.Range(0, 4);

    if(ran == 0)
       return Vector3.up;
    else if(ran == 1)
       return Vector3.down;
    else if(ran == 2)
       return Vector3.left;
    else if(ran == 3)
       return Vector3.right;

    return Vector3.zero;
}
```

We're going to assign this random direction to "**dir**", and then cast a **ray** into the direction using **Physics2D.Raycast**. This will allow us to determine if we can move to the direction or not.

If the ray hits nothing (**null**), we can set **canMove** to true, and update our **transform.position** outside the while loop.

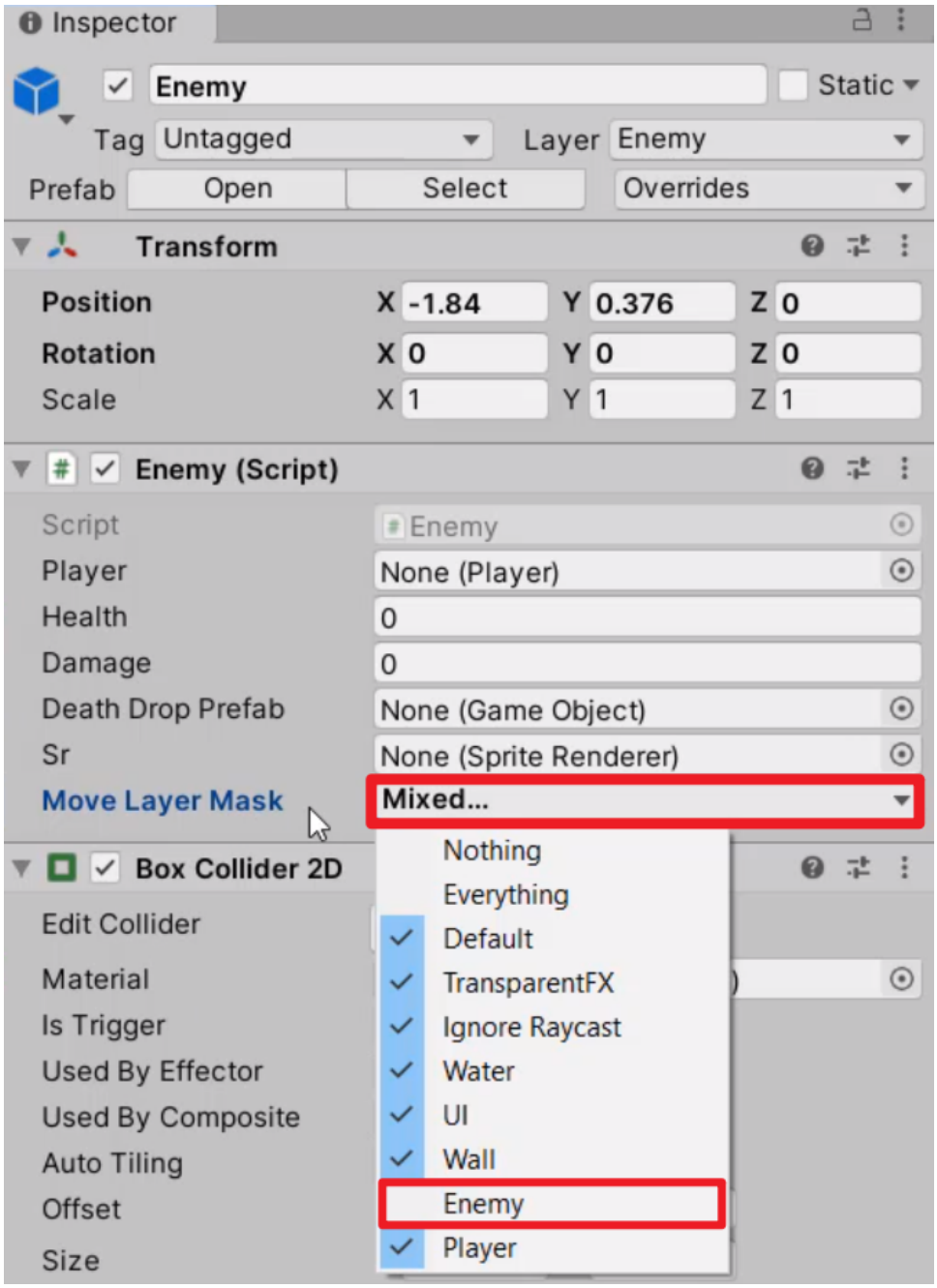
```
public void Move ()
{
    if(Random.value < 0.5f)
        return;

    Vector3 dir = Vector3.zero;
    bool canMove = false;

    while(canMove == false)
    {
        dir = GetRandomDirection();
        // cast a ray into the direction.
        RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f, moveLayer
Mask);

        // if the ray hasn't detected any obstacle,
        if(hit.collider == null)
            // end of while loop
            canMove = true;
    }

    // move towards the direction
    transform.position += dir;

}

Vector3 GetRandomDirection ()
{
```

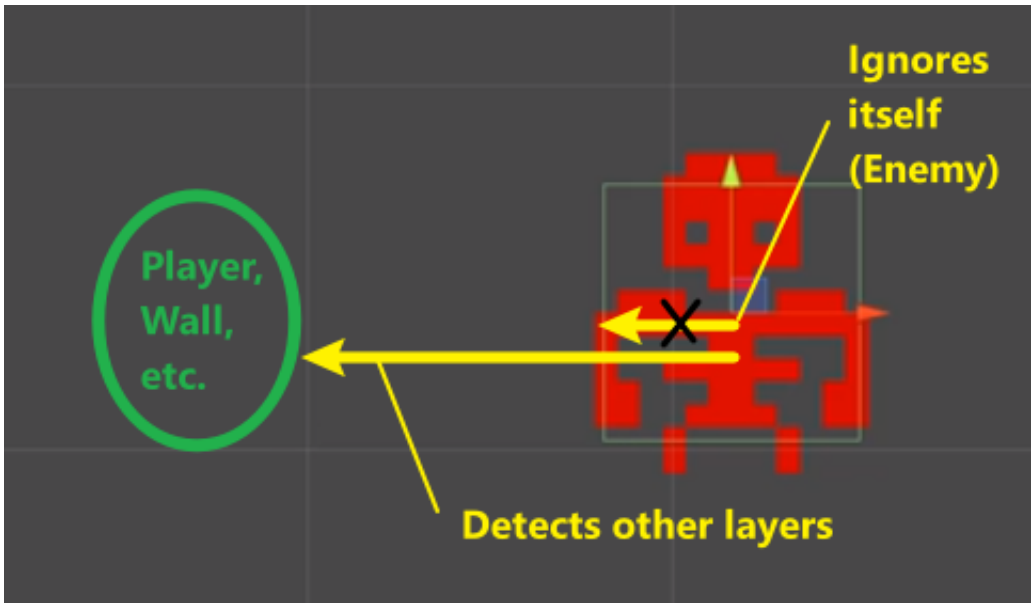


```
    // Get a random number between 0 and 4
    int ran = Random.Range(0, 4);

    if(ran == 0)
       return Vector3.up;
    else if(ran == 1)
       return Vector3.down;
    else if(ran == 2)
       return Vector3.left;
    else if(ran == 3)
       return Vector3.right;

    return Vector3.zero;
}
```

Now we need to make a new **LayerMask** variable for **Physics2D.Raycast**. Because the Raycast is emitting from inside the enemy, this LayerMask should include **everything except** for the "**Enemy**" layer.

```
public LayerMask moveLayerMask;
```





In this lesson, we're going to continue setting up our Enemy.

Taking Damage

First of all, we're going to create a new function called "**TakeDamage**". This is going to be called when the player deals damage to the enemy, taking in an **int** parameter for the amount of damage taken.

```
public void TakeDamage (int damageToTake)
{

}
```

Inside this function, we're just going to **subtract** the damage amount from **health**. If the health is less than or equal to 0, we can drop an item that is marked as **deathDropPrefab** and then destroy the enemy.

```
public void TakeDamage (int damageToTake)
{
    health -= damageToTake;


    if(health <= 0)
    {
        // if there is an item to drop, instantiate it.
        if(deathDropPrefab != null)
            Instantiate(deathDropPrefab, transform.position, Quaternion.identity);

        Destroy(gameObject);
    }
}
```

Now we're going to make our enemy flash its sprite whenever they take damage.

Flashing A Sprite

To do that, we're going to create a new **IEnumerator** called "DamageFlash". An IEnumerator is a **coroutine** that has the ability to pause and wait for a designated time until it resumes. We need this because we're going to change the color of the enemy sprite to white for a short amount of time (0.05 s) and then switch it back to the default color, every time the enemy takes damage.

```
IEnumerator DamageFlash()
{
    // get a reference to the default sprite color (red),
    Color defaultColor = sr.color;
    // set the color to white,
    sr.color = Color.white;

    // wait for 0.05 sec,
    yield return new WaitForSeconds(0.05f);
```



```
    // set the color back to default (red).
    sr.color = defaultColor;
}
```

Since this is a coroutine, we can't just call it like a normal function. Instead, we need to use **StartCoroutine** to execute **DamageFlash()** every time the enemy takes damage.

```
public void TakeDamage (int damageToTake)
{
    health -= damageToTake;


    if(health <= 0)
    {
        // if there is an item to drop, instantiate it.
        if(deathDropPrefab != null)
            Instantiate(deathDropPrefab, transform.position, Quaternion.identity);

        Destroy(gameObject);
    }

    // start the DamageFlash coroutine
    StartCoroutine(DamageFlash());
}
```

Along with this, we also want to have a random chance of dealing damage back to the player. Let's create a float variable called "**attackChance**" and have it equal to 50% by default.

```
public float attackChance = 0.5f;
```

If a **random value** is greater than **attackChance**, the enemy will deal damage back to the player.

```
public void TakeDamage (int damageToTake)
{
    health -= damageToTake;

    if(health <= 0)
    {
        if(deathDropPrefab != null)
            Instantiate(deathDropPrefab, transform.position, Quaternion.identity);

        Destroy(gameObject);
    }

    StartCoroutine(DamageFlash());

    // If a random value is greater than attackChance,
    if(Random.value > attackChance)
```



```
        // deal damage to the player.
        player.TakeDamage(damage);
}
```

Now we're going to go over to the **Player** script and make another similar function called "**TakeDamage**". Note that some variable names may differ from the Enemy script (e.g. **curHp**)

```
// Inside Player.cs:
public void TakeDamage (int damageToTake)
{
    curHp -= damageToTake;

    StartCoroutine(DamageFlash());
}

IEnumerator DamageFlash()
{
    Color defaultColor = sr.color;
    sr.color = Color.white;

    yield return new WaitForSeconds(0.05f);

    sr.color = defaultColor;
}
```

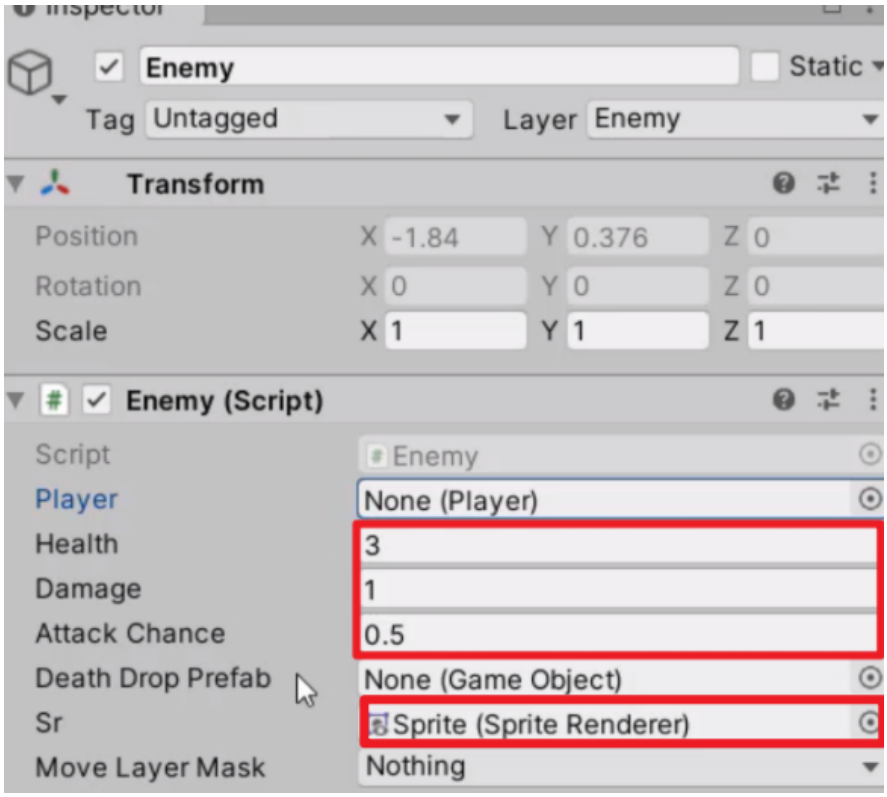
We can then **reset** the scene whenever the player's health falls below 0. To do this, we need to import the **SceneManagement** library first.

```
using UnityEngine.SceneManagement;
```

Then we can use the **SceneManager** to load up the first scene, which we will set as the main menu in future lessons.

```
public void TakeDamage (int damageToTake)
{
    curHp -= damageToTake;

    StartCoroutine(DamageFlash());

    // load the main menu scene
    if(curHp <= 0)
        SceneManager.LoadScene(0);
}
```

Let's save the scripts and open up our Enemy prefab to assign values to the public variables. Make sure to also drag in the **Sprite** component to the '**Sr**' field. (The Player variable is automatically assigned at the start of the game.)



In this lesson, we're going to implement the ability for the player to do damage to the enemy.

Shooting A Raycast

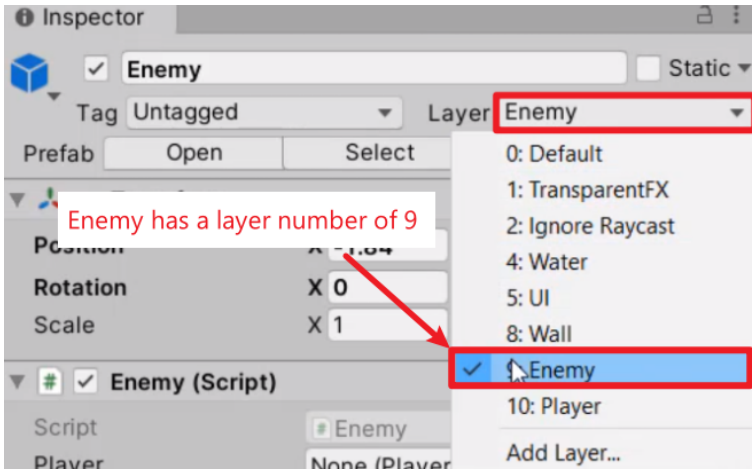
Let's go over to the Player script, and create a new function called "**TryAttack**". This function is going to take in a **Vector2** parameter as the direction of attack.

```
void TryAttack (Vector2 dir)
{

}
```

Inside this function, we're going to be shooting a **ray cast** at the given direction using **Physics2D.Raycast**. At the same time, we can store the information about what the ray hit inside a temporary **RaycastHit2D** variable ('**hit**').

```
void TryAttack (Vector2 dir)
{
    // shoot a raycast from the current position towards 'dir'.
    RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f,
}
```

As the final parameter of **Physics2D.Raycast**, we need to send over the **layer mask**. Since we only want to detect enemies, we're going to enter every layer except the enemy's layer (**1 << 9**) as the layers to ignore.




```
void TryAttack (Vector2 dir)
{
    // ignore the layer 1 to layer 8. (only detect Layer 9: Enemy)
    RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f, 1 << 9);
}
```

If this ray has hit an enemy, then we're going to access the **Enemy** component attached to the object and call its **TakeDamage** function. Make sure to send over the amount of damage as an

integer.

```
void TryAttack (Vector2 dir)
{
    RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f, 1 << 9);

    // if the ray has hit an enemy (collider),
    if(hit.collider != null)
    {
        // the enemy will take damage.
        hit.transform.GetComponent<Enemy>().TakeDamage(1);
    }
}
```

Callback Context

Let's now implement it in each of the attack function calls, which get called when we press down the arrow keys. For example, if the **up arrow** key is pressed down, then we can call the **TryAttack** function inside the **OnAttackUp** function, sending over **Vector2.up** as the direction of attack.

```
public void OnAttackUp (InputAction.CallbackContext context)
{
    // when we press the key down,
    if(context.phase == InputActionPhase.Performed)
        // try attack (up).
        TryAttack(Vector2.up);
}
```

We can repeat this for **OnAttackDown**, **OnAttackLeft**, and **OnAttackRight**. Keep in mind that we need to change the TryAttack parameter to the corresponding direction.

```
public void OnAttackUp (InputAction.CallbackContext context)
{
    // when we press the key down,
    if(context.phase == InputActionPhase.Performed)
        // try attack (up).
        TryAttack(Vector2.up);
}

public void OnAttackDown (InputAction.CallbackContext context)
{
    // when we press the key down,
    if(context.phase == InputActionPhase.Performed)
        // try attack (down).
        TryAttack(Vector2.down);
}

public void OnAttackLeft (InputAction.CallbackContext context)
{
    // when we press the key down,
    if(context.phase == InputActionPhase.Performed)
```

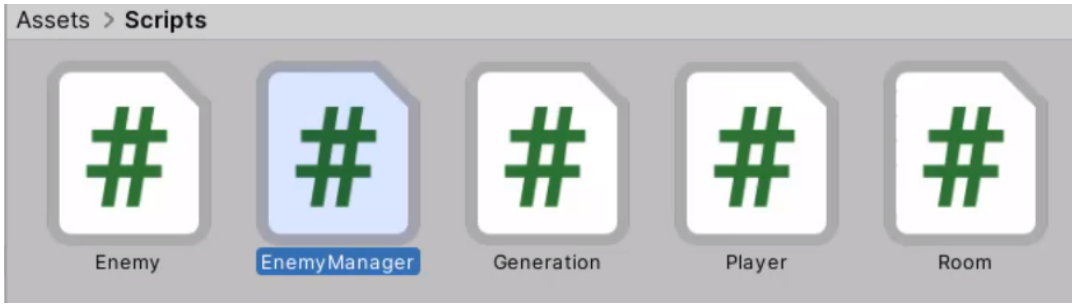


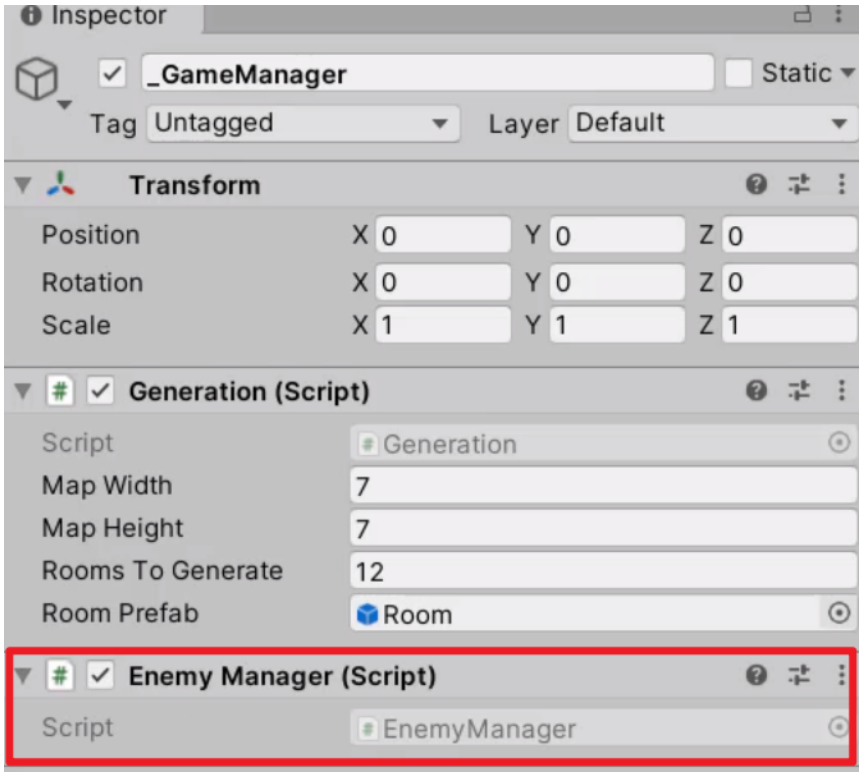
```
        // try attack (left).
        TryAttack(Vector2.left);
}

public void OnAttackRight (InputAction.CallbackContext context)
{
    // when we press the key down,
    if(context.phase == InputActionPhase.Performed)
        // try attack (right).
        TryAttack(Vector2.right);
}
```

Moving Enemy

What we now need to do is make it so that the enemy can actually move when our player moves.

We're going to create a new C# script called "**EnemyManager**" and attach it to the **_GameManager** object.





Inside the script, we're going to create a new **list** of **Enemy** so we can move all the enemies at once.

```
public List<Enemy> enemies = new List<Enemy>();
```

Then we're going to make a **Singleton** instance of this class so we can access it anywhere easily.

```
public List<Enemy> enemies = new List<Enemy>();

public static EnemyManager instance;

void Awake()
{
    instance = this;
}
```

To move our enemies after a certain time delay, we're going to create an **IEnumerator** called "MoveEnemies". Then we can initiate this coroutine on every player move.

```
public List<Enemy> enemies = new List<Enemy>();

public static EnemyManager instance;

void Awake()
{
    instance = this;
}

publci void OnPlayerMove ()
{
     StartCoroutine(MoveEnemies());
}

IEnumerator MoveEnemies ()
{
    yield return new WaitForFixedUpdate();
}
```

And finally, we're going to **move** each enemy inside the '**enemies**' list. Make sure to do a **null check** of the enemy first in case the enemy has been defeated and removed from the scene.

```
public List<Enemy> enemies = new List<Enemy>();

public static EnemyManager instance;

void Awake()
{
    instance = this;
}
```

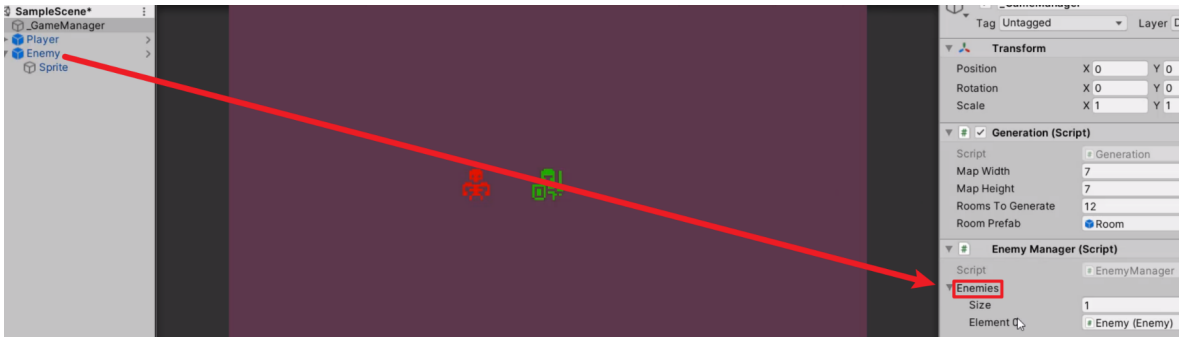


```
publci void OnPlayerMove ()
{
     StartCoroutine(MoveEnemies());
}

IEnumerator MoveEnemies ()
{
    yield return new WaitForFixedUpdate();

    foreach(Enemy enemy in enemies)
    {
        if(enemy != null)
            enemy.Move();
    }
}
```

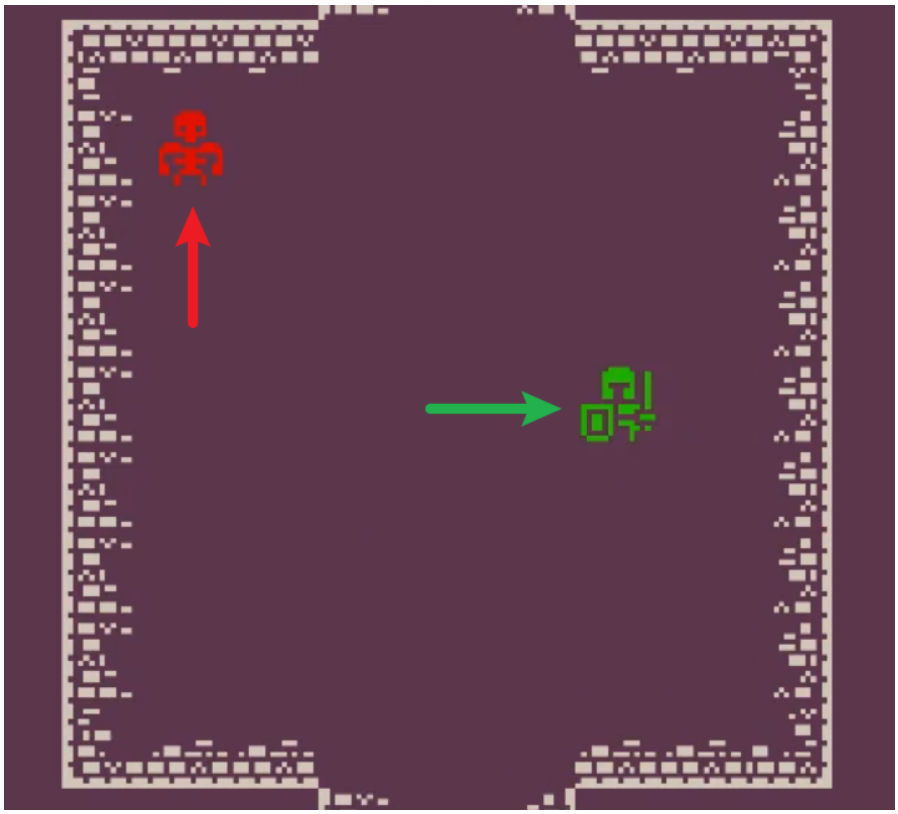
We can now go over to the **Move** function of the **Player** script and access the **EnemyManager** to call the **OnPlayerMove** function.

```
void Move (Vector2 dir)
{
    RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f, moveLayerMask
);

    if(hit.collider == null)
    {
        transform.position += new Vector3(dir.x, dir.y, 0);
        // move enemies
        EnemyManager.instance.OnPlayerMove();
    }
}
```

To test it out, we can drag in any **Enemy** object into the **EnemyManager** script and press **Play**.



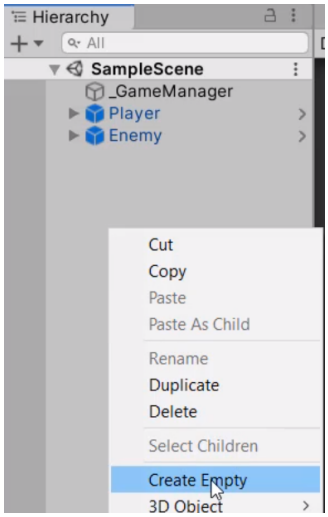
There’s a 50% chance of the enemy to move everytime you press down the arrow keys.



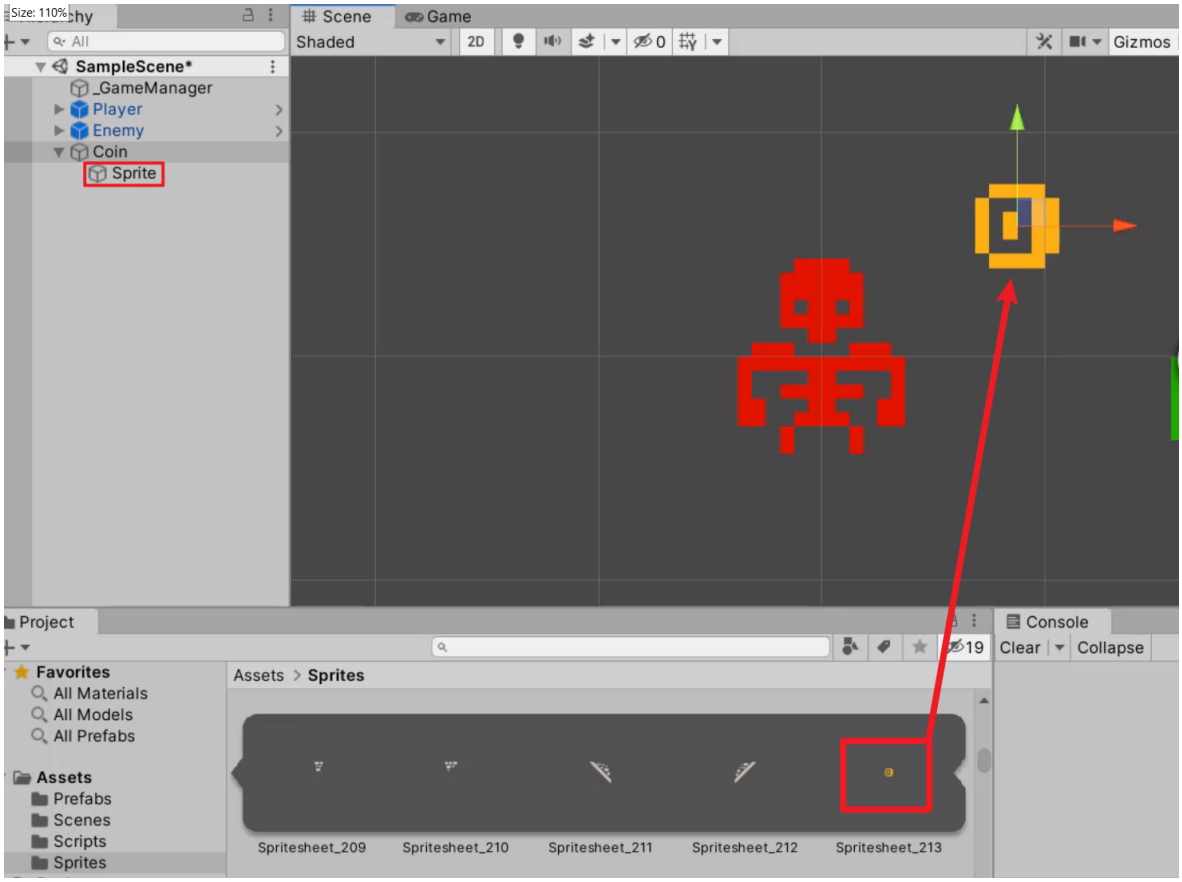
In this lesson, we're going to be working on setting up our **health packs** and **coins**.

Creating A Coin

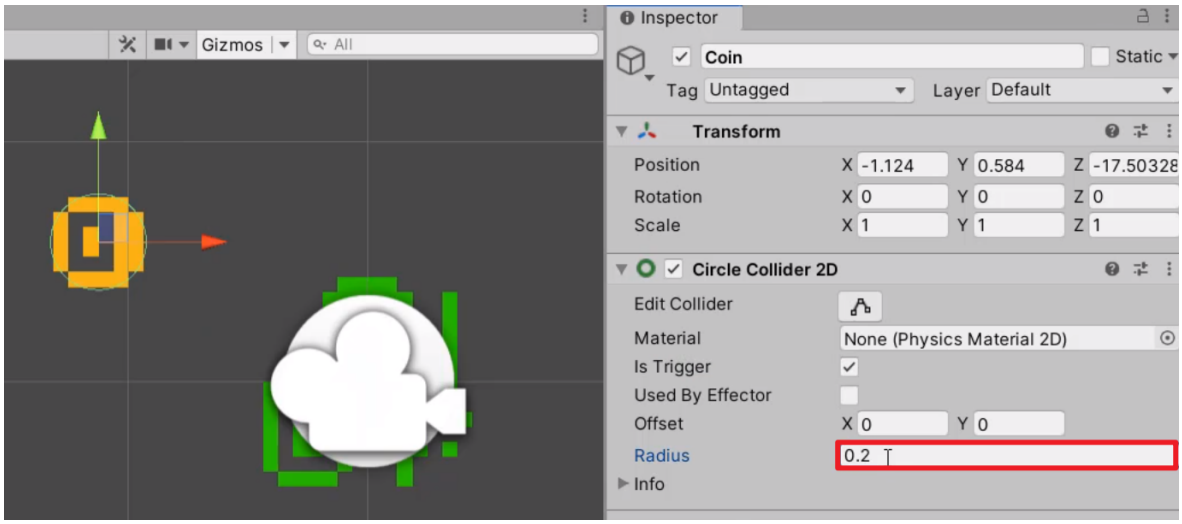
First of all, we're going to create a new **empty** gameObject called "Coin".



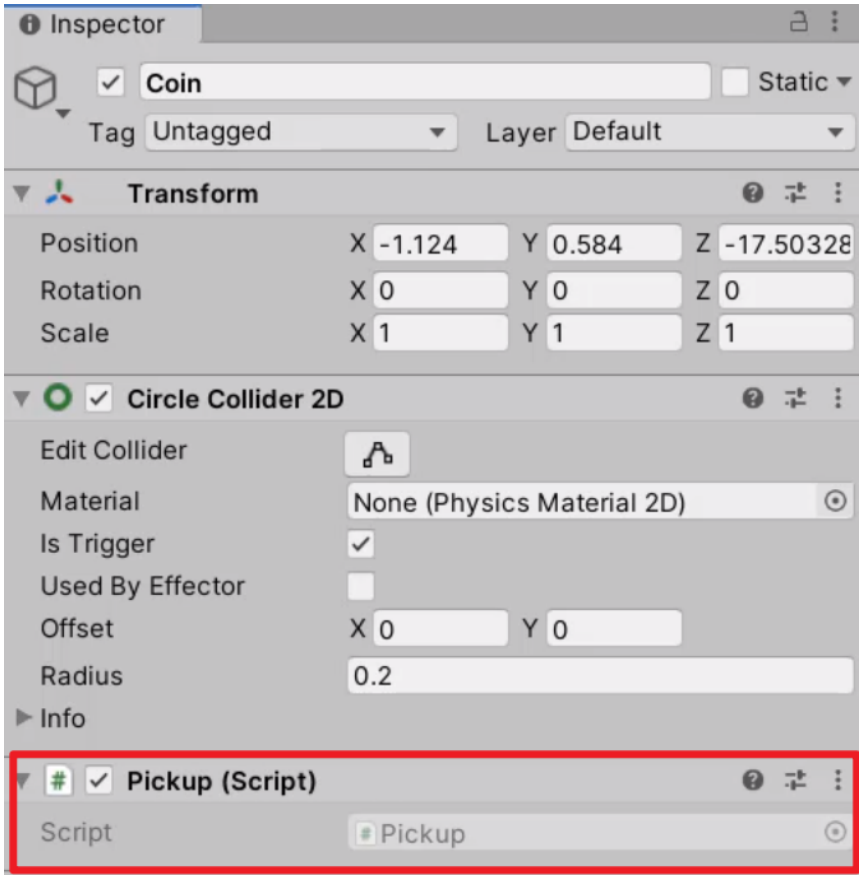
Then we can drag in a coin **Sprite** as a **child** to the "Coin" object.



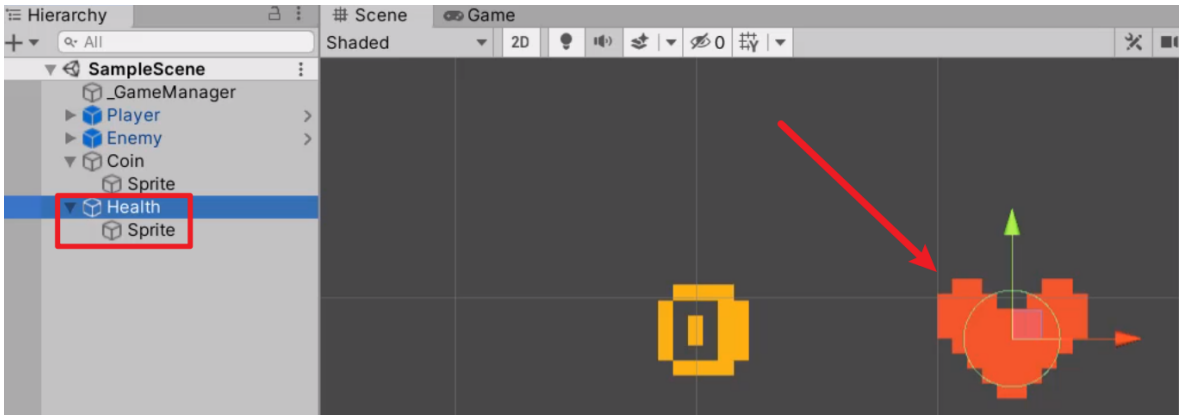
In order for the player object to detect the coin, we need to attach a **Circle Collider 2D** component. Make sure to set it as a **Trigger** and adjust the **Radius** (0.2) so that it fits the size of the sprite.

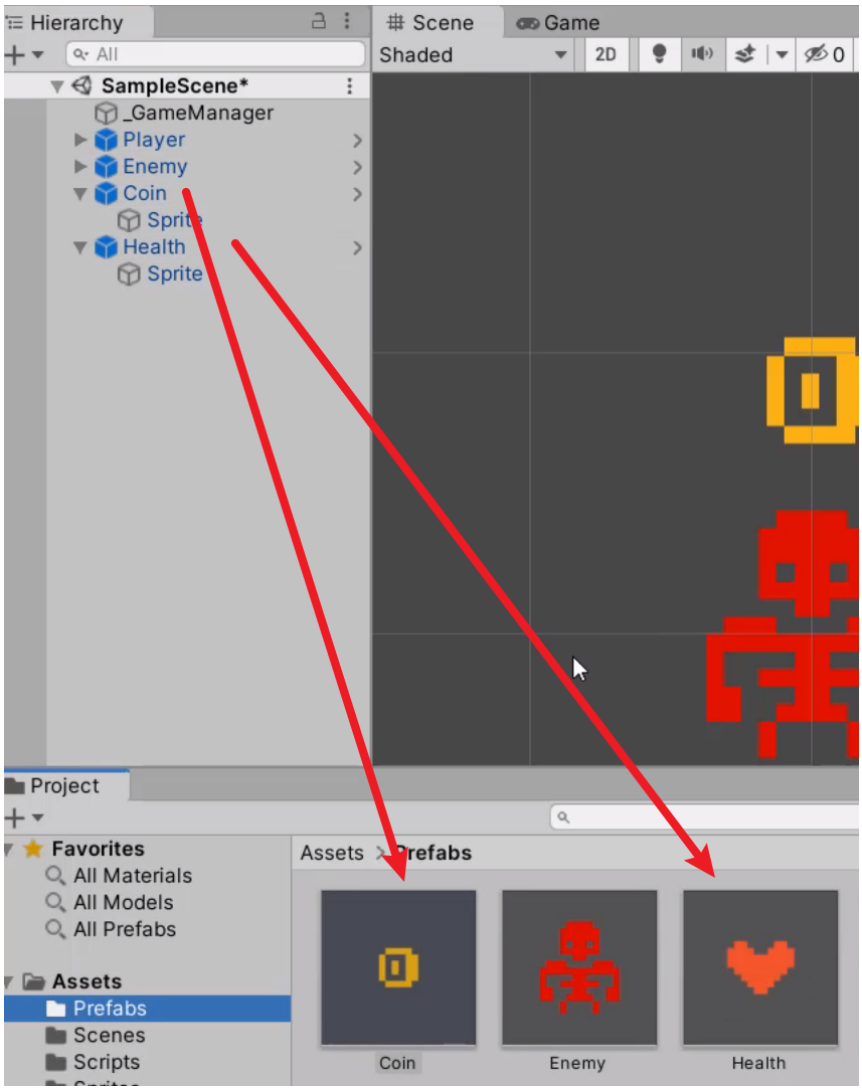


Finally, let's create a new C# script called "**Pickup**" and attach it to the coin object.



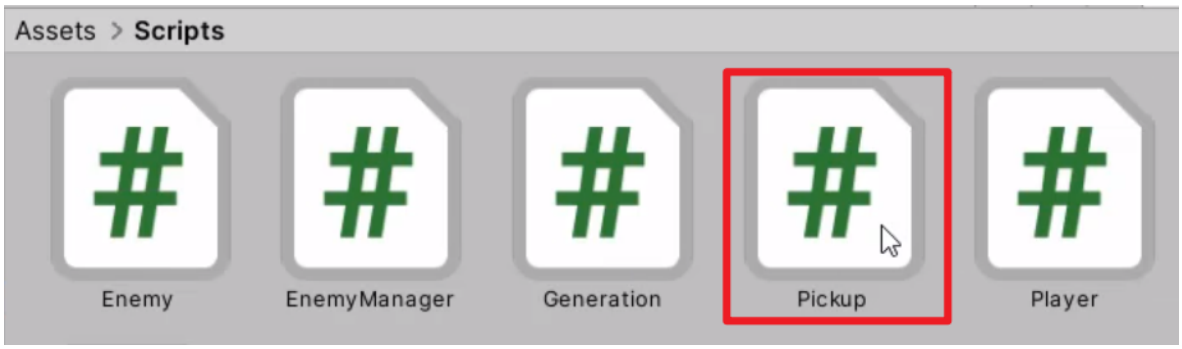
We can take a similar process to create a **Health pack** gameObject, and drag these pickups into the **Prefabs** folder to save as prefabs.





Scripting Pickups

Let's **double-click** on the **Pickup** script to open it up inside of Visual Studio.



Since this script is going to manage all the **pickup types**, we're going to create an **enum**— a custom variable type that you can create in C#.

```
public enum PickupType
{
    Coin,
    Health
}
```

Then we can use this custom variable type inside the **Pickup** class to check which **pickup** type we have collided with.

```
public class Pickup : MonoBehaviour
{
    public PickupType type;
    public int value = 1;

    private void OnTriggerEnter2D(Collider2D collision)
    {
         // if the pickup object has collided with the player,
         if(collision.CompareTag("Player"))
         {
              // if the pickup type is 'Coin'
              if(type == PickupType.Coin)
              {
              }
              // if the pickup type is 'Health'
              else if(type == PickupType.Health)
              {
              }
         }
    }
}
```

What we then want to do is have functions that **add** some amount to the player's **coin**.

```
public void AddCoins (int amount)
{
     coins += amount;
     // To-do: update the UI
```



```
}
```

And likewise, we need a function that adds some amount to the player's **health**. We're also going to make the function return a **boolean** in order to ensure that the health pack is only consumed when **curHp** doesn't exceed **maxHp**.

```
public void AddCoins (int amount)
{
     coins += amount;
     // To-do: update the UI
}

public bool AddHealth (int amount)
{
     if(curHp + amount <= maxHp)
     {
         curHp += amount;
         // update the UI
         return true;
     }

     return false;
}
```

Now it's time to actually call these functions inside **OnTriggerEnter2D**. Once the pickup has been collected, we need to **destroy** it so that it doesn't get collected multiple times.

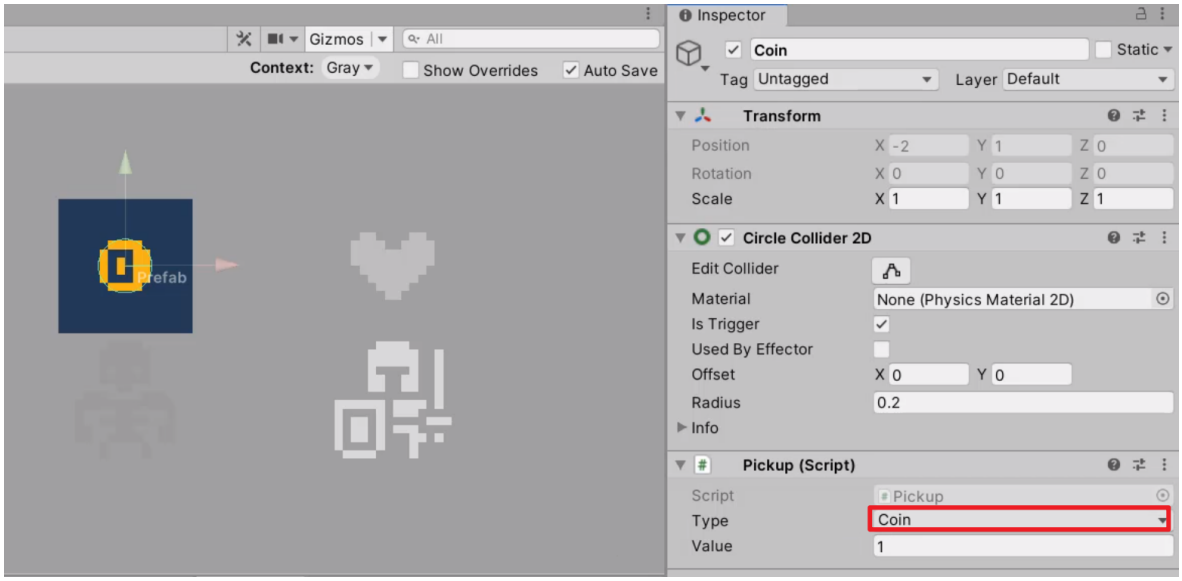
```
public class Pickup : MonoBehaviour
{
    public PickupType type;
    public int value = 1;

    private void OnTriggerEnter2D(Collider2D collision)
    {
         // if the pickup object has collided with the player,
         if(collision.CompareTag("Player"))
         {
              // if the pickup type is 'Coin'
              if(type == PickupType.Coin)
              {
                   collision.GetComponent<Player>().AddCoins(value);
                   Destroy(gameObject);
              }

              // if the pickup type is 'Health'
              if(type == PickupType.Health)
              {
                   collision.GetComponent<Player>().AddHealth(value);
                   Destroy(gameObject);
              }
```

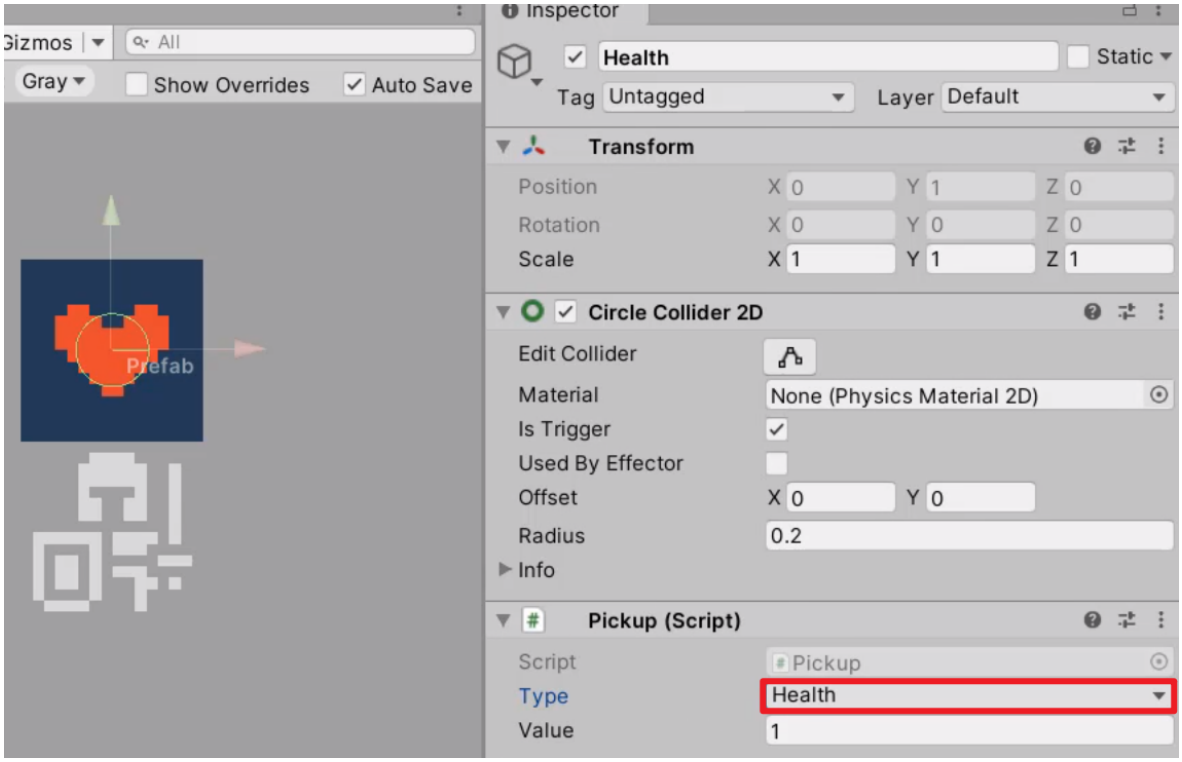


```
            }
        }
}
```

Setting Up Pickup Prefabs

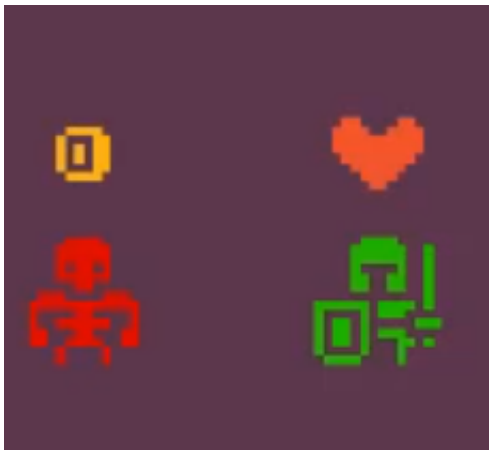
Now let's **save** the script and open up the pickup objects inside the **Prefab** Editor. We can check inside the **Pickup** component so that the corresponding **Type** is set up.

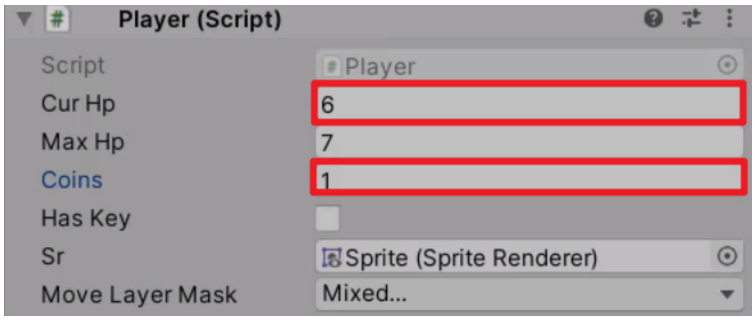




If you press **Play** and collect the pickups, you'll see that the **CurHp** property and the **Coins** property

goes up in the Inspector.





In this lesson, we're going to begin generating our **room interiors**.

Spawning Random Prefabs

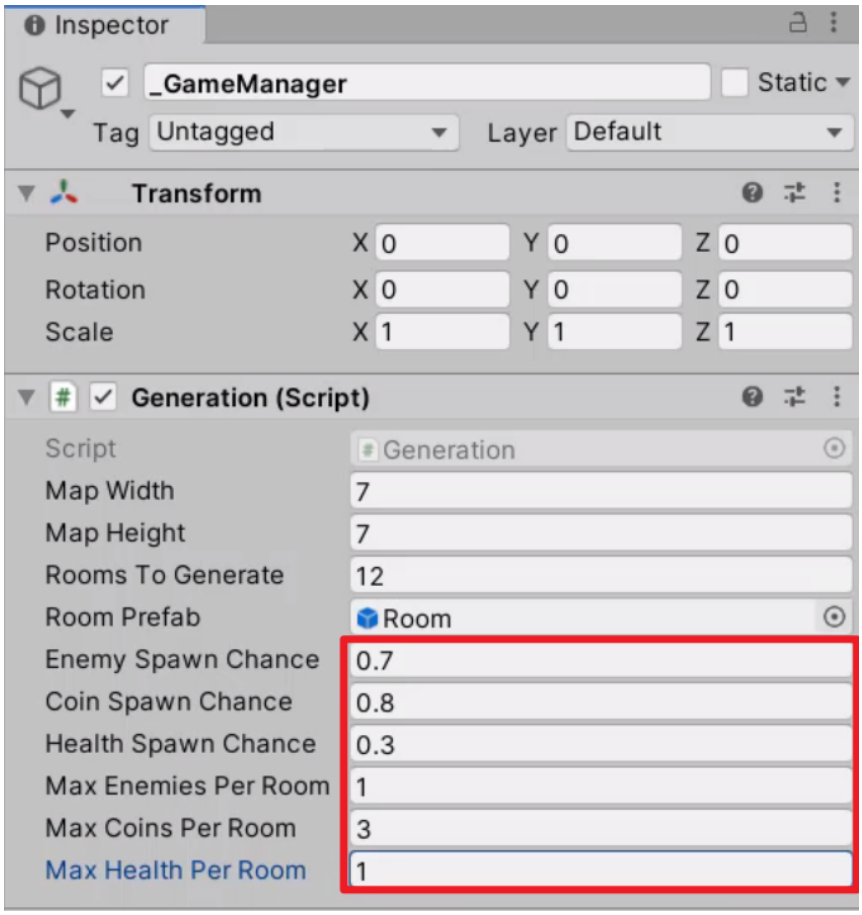
Let's open up the **Room** script and create a new function called "**GenerateInterior**". This function will spawn different objects based on random number generation.

```
public void GenerateInterior ()
{
     //do we spawn enemies?
}
```

Then we'll go over to the **Generation** script and create the variables for different **spawn chances** and the **max number** of entities per room to spawn.

```
public float enemySpawnChance;
public float coinSpawnChance;
public float healthSpawnChance;

public int maxEnemiesPerRoom;
public int maxCoinsPerRoom;
public int maxHealthPerRoom;
```

Make sure to **save** the script, and set the values in the Inspector.



Now we can go back to the **Room** script and spawn different prefabs. We're going to be using **random.value** to do this, which will be a number between 0 and 1. If the value is less than the **enemySpawnChance**, then we can spawn in the enemy prefab.

```
public void GenerateInterior ()
{
    // do we spawn enemies?
    if(Random.value < Generation.instance.enemySpawnChance)
    {
        SpawnPrefab(enemyPrefab, 1, Generation.instance.maxEnemiesPerRoom +1);
    }
}
```

Note that we added 1 to **maxEnemiesPerRoom**, because the maximum is **exclusive** when it comes down to randomly generating a number between a range, meaning that we can actually get the full range of random numbers by adding 1. We can do the exact same for coins and health.

```
public void GenerateInterior ()
{
    // do we spawn enemies?
    if(Random.value < Generation.instance.enemySpawnChance)
    {
        SpawnPrefab(enemyPrefab, 1, Generation.instance.maxEnemiesPerRoom + 1);
    }
```



```
    // do we spawn coins?
    if(Random.value < Generation.instance.coinSpawnChance)
    {
        SpawnPrefab(coinPrefab, 1, Generation.instance.maxCoinPerRoom + 1);
    }

    // do we spawn health?
    if(Random.value < Generation.instance.healthSpawnChance)
    {
        SpawnPrefab(healthPrefab, 1, Generation.instance.maxHealthPerRoom + 1);
    }

}
```

Spawning Prefabs At Random Positions

Now, what we need to do is fill out the **SpawnPrefab** function. We're loop through each of the prefabs we need to spawn using a random number '**num**' between the given '**min**' and '**max**'.

Inside this loop, we're going to **instantiate** the prefab and place it at a **random position**.

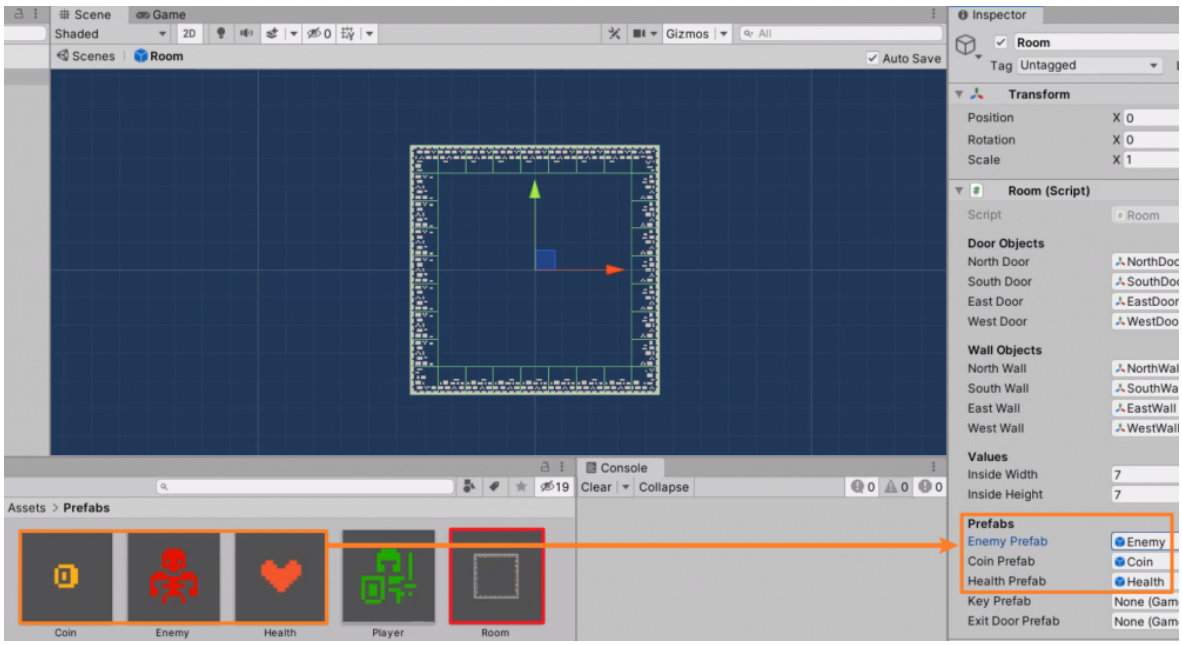
```
public void SpawnPrefab (GameObject prefab, int min = 0, int max = 0)
{
    int num = 1;
    if(min != 0 || max != 0)
        num = Random.Range(min, max);

    // for each of the prefabs,
    for(int x = 0; x < num; ++x)
    {
        // instantiate the prefab
        GameObject obj = Instantiate(prefab);
        // getting the nearest tile to a random position inside the room.
        Vector3 pos = transform.position + new Vector3(Random.Range(-insideHeight / 2
, insideWidth / 2 + 1), Random.Range(-insideHeight / 2, insideHeight / 2 + 1), 0);

        // if the position is already in use, pick another random position.
        while(usedPositions.Contains(pos))
        {
             pos = transform.position + new Vector3(Random.Range(-insideHeight / 2, i
nsideWidth / 2 + 1), Random.Range(-insideHeight / 2, insideHeight / 2 + 1), 0);
        }

        // place the prefab to the random position.
        obj.transform.position = pos;
        // add the current position to the 'usedPositions' list.
        usedPositions.Add(pos);
    }
}
```

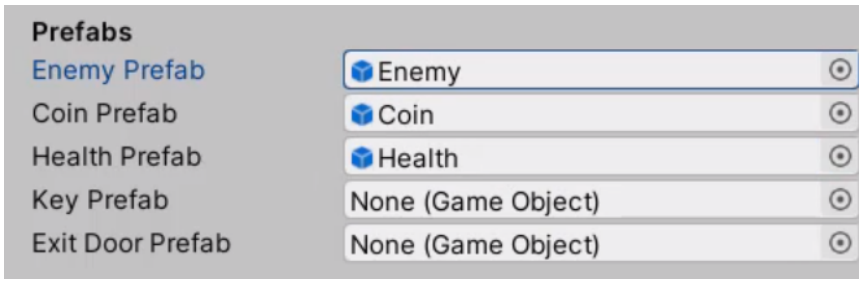
If the spawned prefab is **enemyPrefab**, we need to add it to the '**enemies**' list inside

EnemyManager.

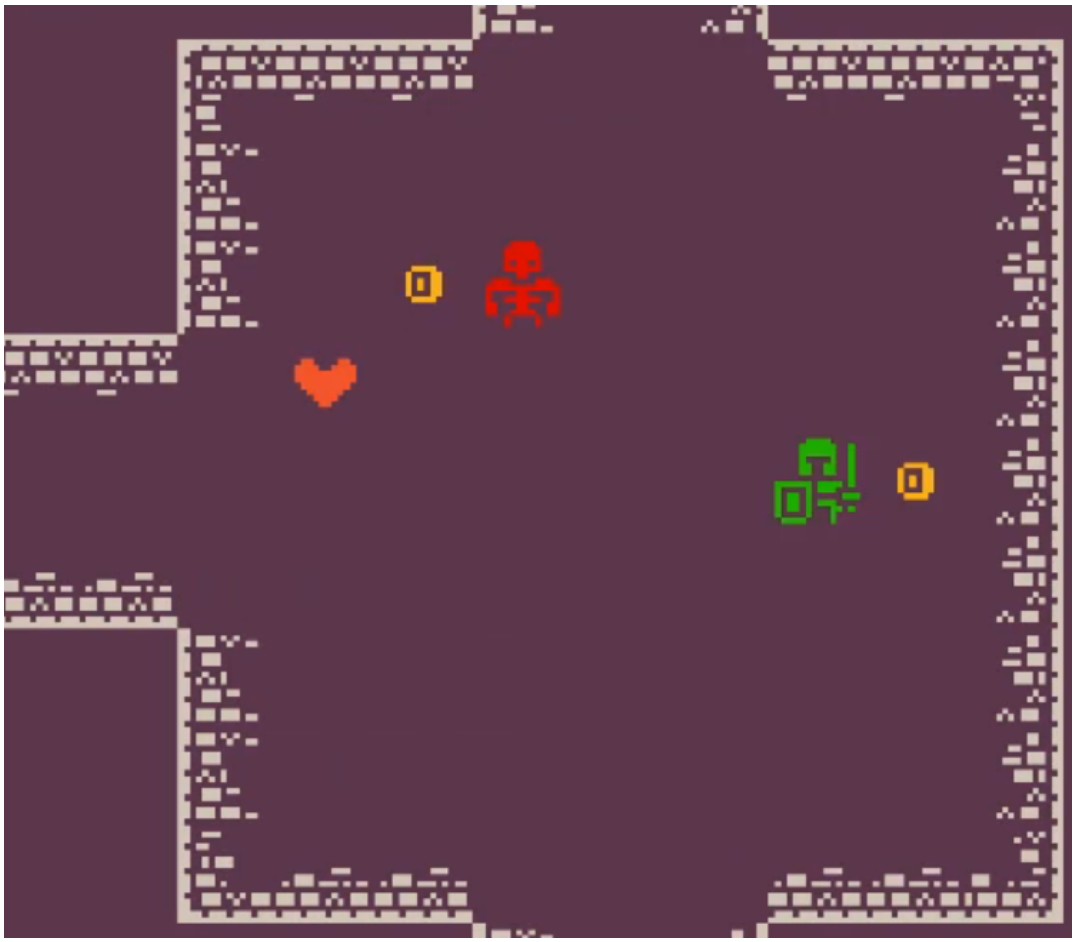
```
public void SpawnPrefab (GameObject prefab, int min = 0, int max = 0)
{
    int num = 1;
    if(min != 0 || max != 0)
        num = Random.Range(min, max);

    for(int x = 0; x < num; ++x)
    {
        GameObject obj = Instantiate(prefab);
        Vector3 pos = transform.position + new Vector3(Random.Range(-insideHeight / 2
, insideWidth / 2 + 1), Random.Range(-insideHeight / 2, insideHeight / 2 + 1), 0);

        while(usedPositions.Contains(pos))
        {
            pos = transform.position + new Vector3(Random.Range(-insideHeight / 2, i
nsideWidth / 2 + 1), Random.Range(-insideHeight / 2, insideHeight / 2 + 1), 0);
        }

        obj.transform.position = pos;
        usedPositions.Add(pos);

        // if the prefab we generated is enemyPrefab,
        if(prefab == enemyPrefab)
            //add it to the EnemyManager's enemies list.
            EnemyManager.instance.enemies.Add(obj.GetComponent<Enemy>());
    }
}
```

Let's **save** that and open up the **Room** prefab. We need to drag in the prefabs into the corresponding fields of the Room script.





Now, if you press **Play**, you'll be able to see that these prefabs get generated at random positions inside the room.

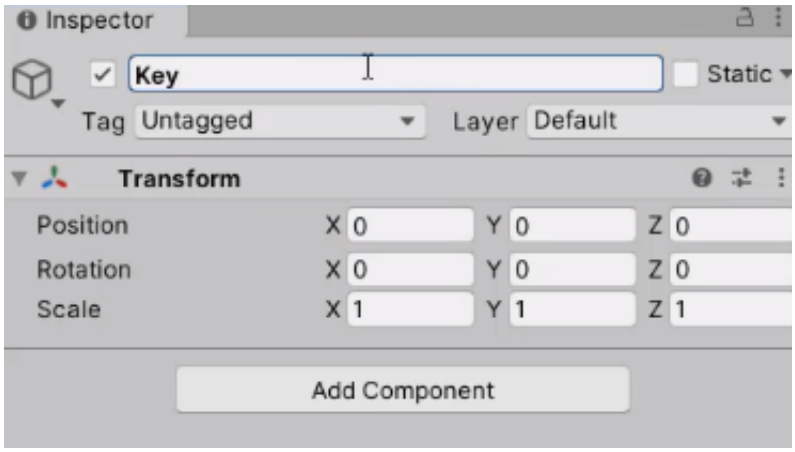


In this lesson, we're going to be setting up our key and exit door object.

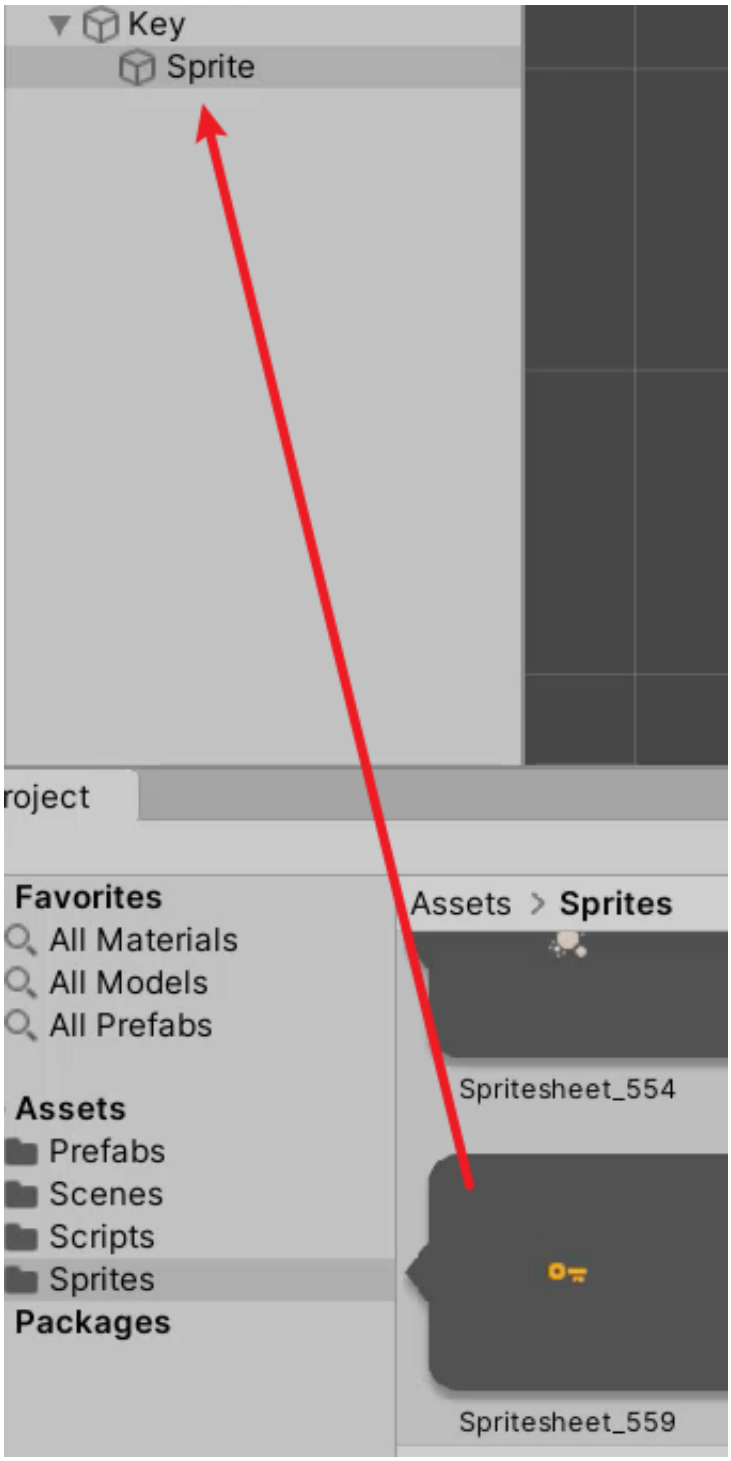
Each level is going to have one key and one exit door. The key and the door will be placed as far away as possible from each other in different rooms, so the player has to collect the key in order to go for the door, which will then take them to the next level.

Creating A Key/ExitDoor GameObject

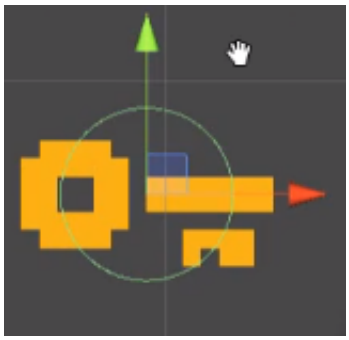
First of all, we're going to create a new **empty** gameObject called "Key":

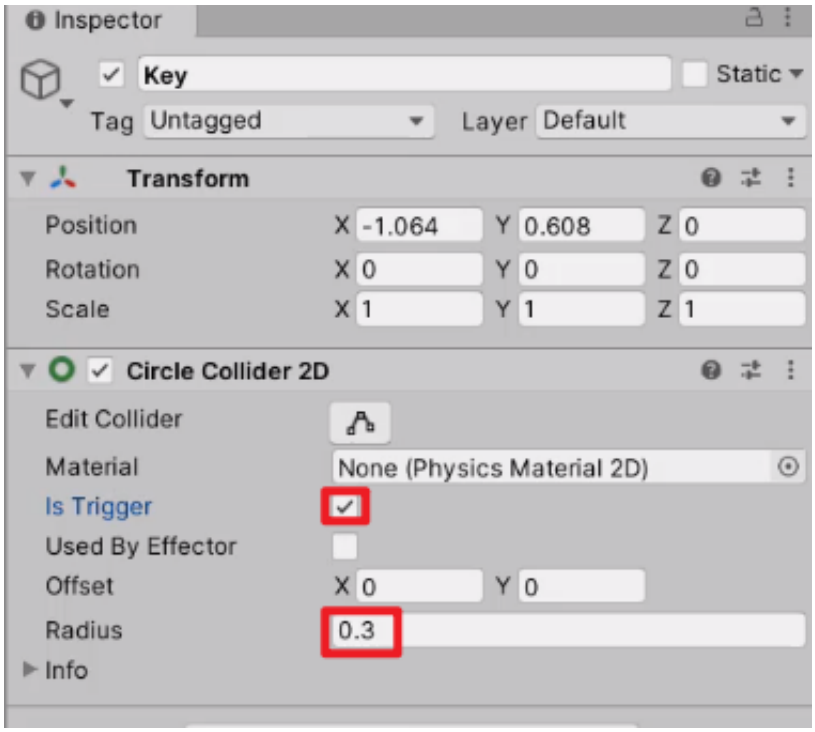


And then drag in the key sprite (**Spritesheet_559**) as a **child** to the "Key" gameObject.

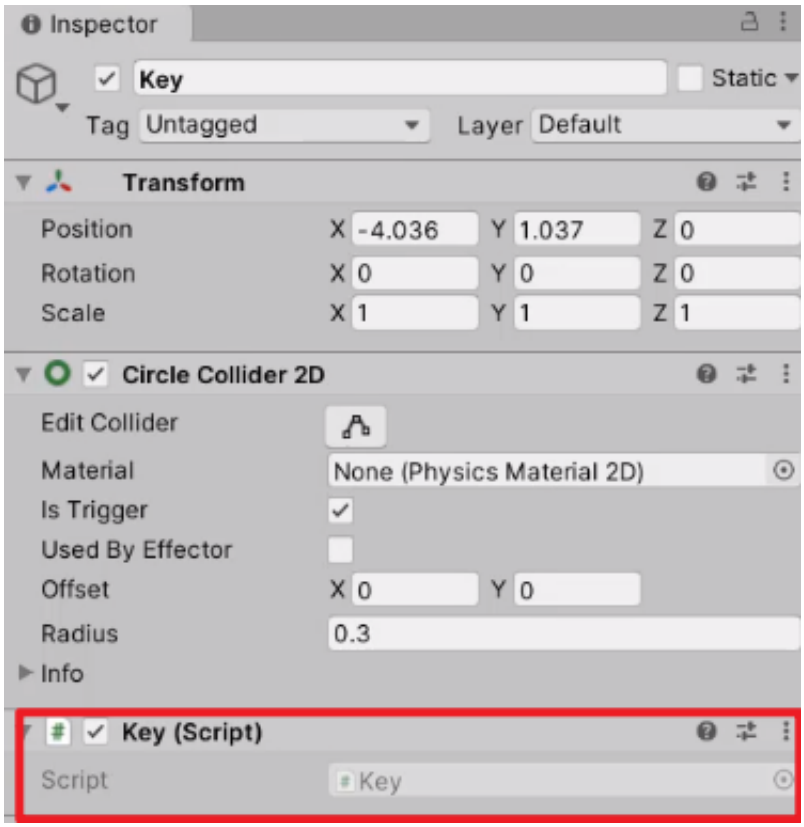


Then we can attach a **Circle Collider 2D** to the key object as a **Trigger**, with a **Radius** of **0.3** to fit the size of the sprite.

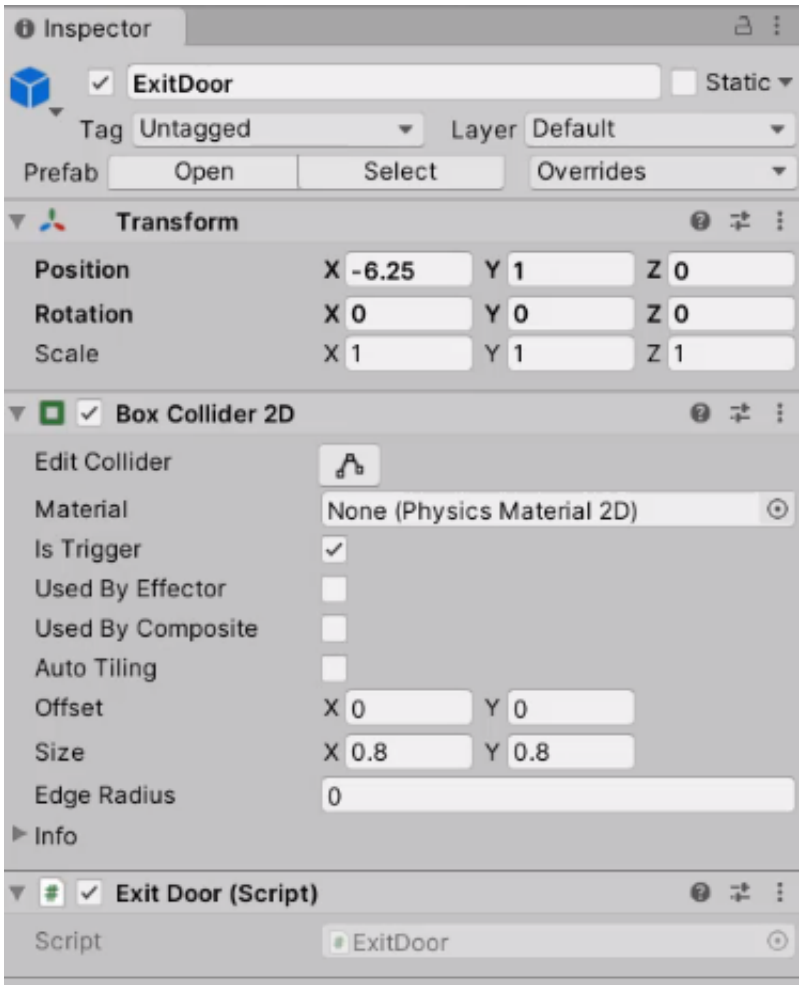


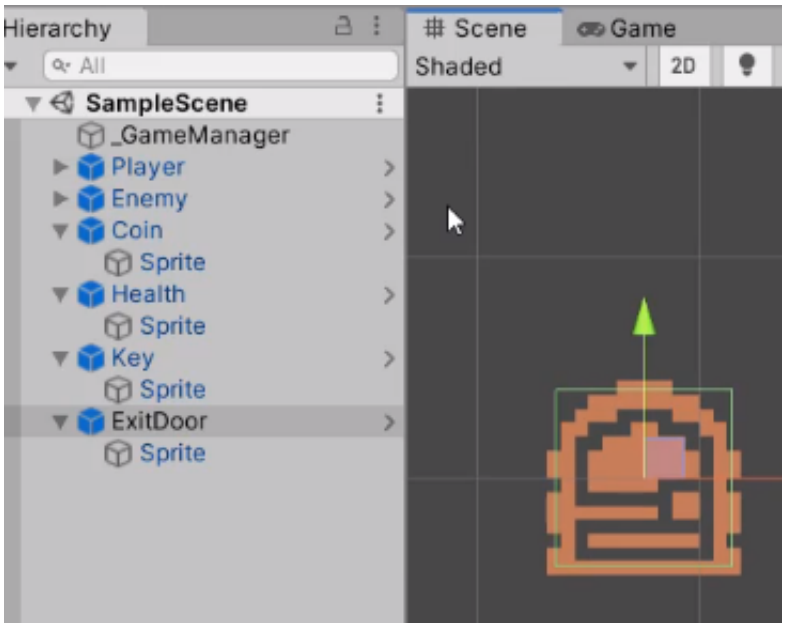


Then what we want to do is go ahead and create a new C# script called “Key”.



We can take the exact same process to create **ExitDoor.**



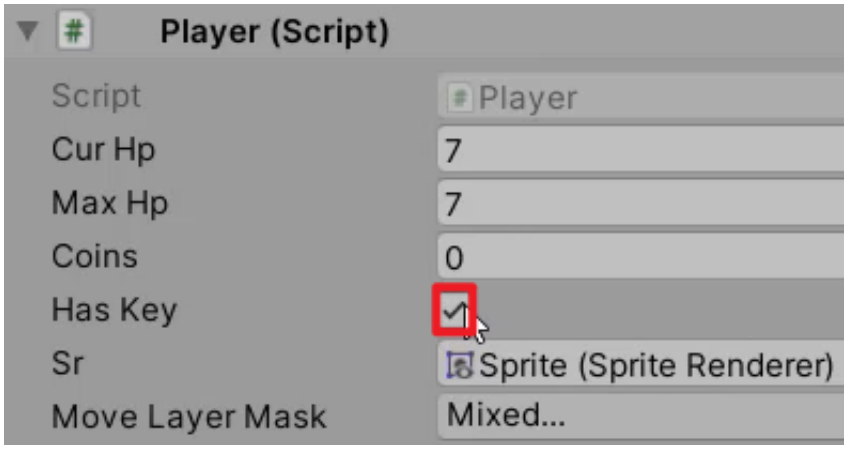


Scripting The Key/ExitDoor

Inside of the **Key** script, we're going to check to see if the player has entered the collider. If so, we can set the **hasKey** property of the Player class to be true, and then **destroy** the key gameObject.


```
private void OnTriggerEnter2D(Collider2D collision)
{
     if(collision.CompareTag("Player"))
     {
          collision.GetComponent<Player>().hasKey = true;
          // update the UI
          Destroy(gameObject);
     }
}
```

Now if you save the script and press **Play**, you'll be able to see that the **hasKey** property gets enabled as soon as you collect the key.



Let's open up the **ExitDoor** script and create a similar function so that the next level loads up if the player has the key equipped.

```
private void OnTriggerEnter2D(Collider2D collision)
{
     if(collision.CompareTag("Player"))
     {
          if(collision.GetComponent<Player>().hasKey)
          Debug.Log("Go to next level");
     }
}
```

Random Placement

Now what we need to do is placing the key and the exit in the level inside the **Generation** script. We're looking for the two rooms which are the furtherest distance apart.

```
void CalculateKeyAndExit()
{
     float maxDist = 0;
     Room a = null;
     Room b = null;
```

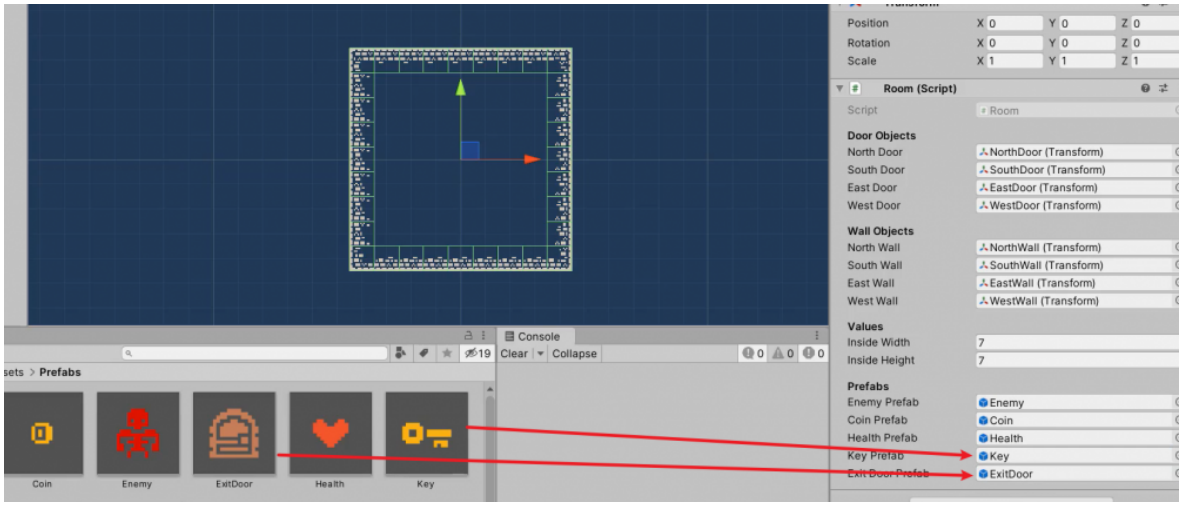


```
}
```

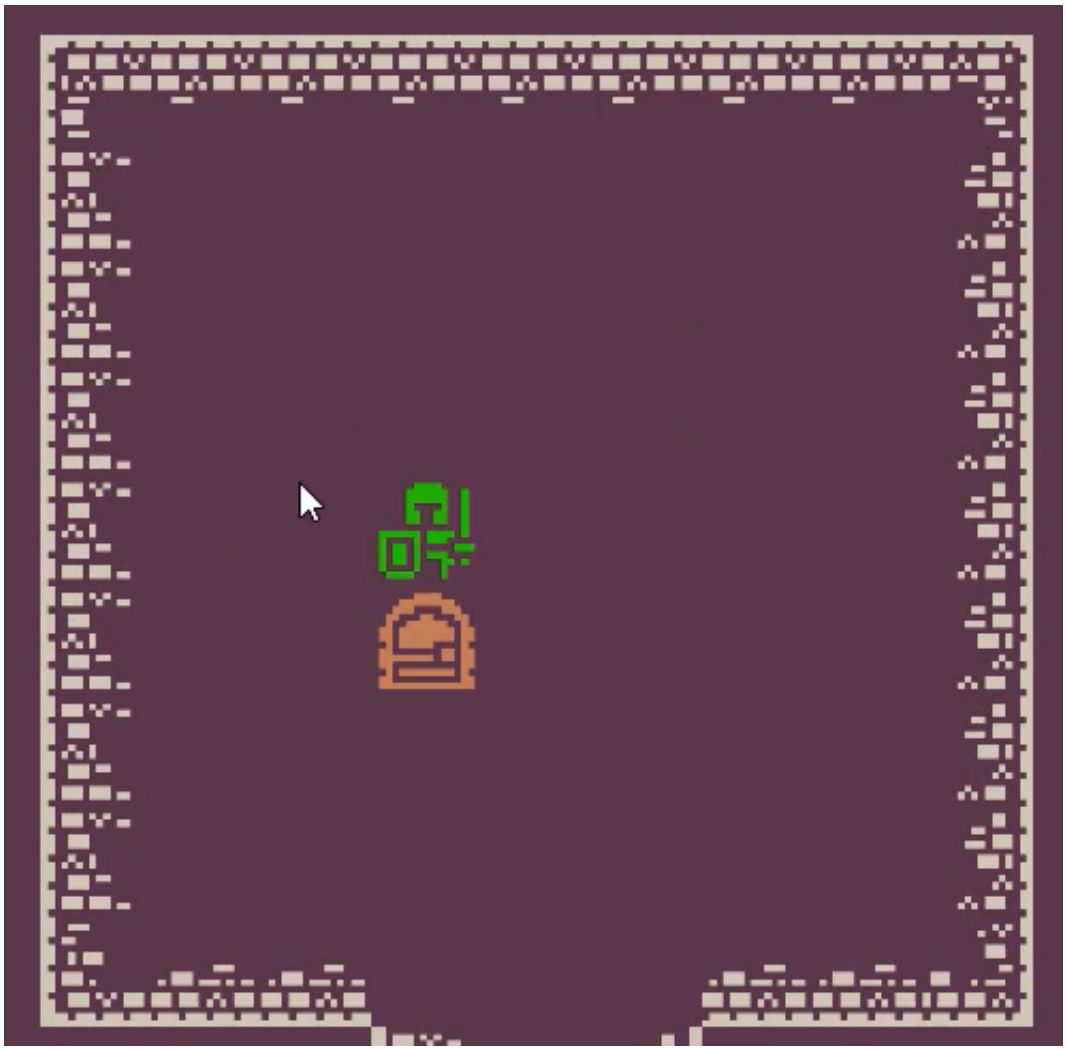
To calculate the maximum distance between Room A and Room B, we're going to use **Vector3.Distance** inside two **foreach** loops. This will allow us to compare each of the rooms and find out which one is the furtherest away.

```
void CalculateKeyAndExit()
{
     float maxDist = 0;
     Room a = null;
     Room b = null;

     foreach(Room aRoom in roomObjects)
     {
          foreach(Room bRoom in roomObjects)
          {
               // compare each of the rooms to find out which pair is the furtherest
away.
               float dist = Vector3.Distance(aRoom.transform.position, bRoom.transfor
m.position);
               if(dist > maxDist)
               {
                    a = aRoom;
                    b = bRoom;
                    maxDist = dist;
               }
          }

          // once room A and room B are found, spawn in the key and the exitdoor.
          a.SpawnPrefab(a.keyPrefab);
          b.SpawnPrefab(b.exitDoorPrefab);
     }
}
```

We then need to open up the **Room** prefab and **drag in** the two prefabs into their correct properties.



If you press Play, you'll see that the exit door and the key are spawned in the level.

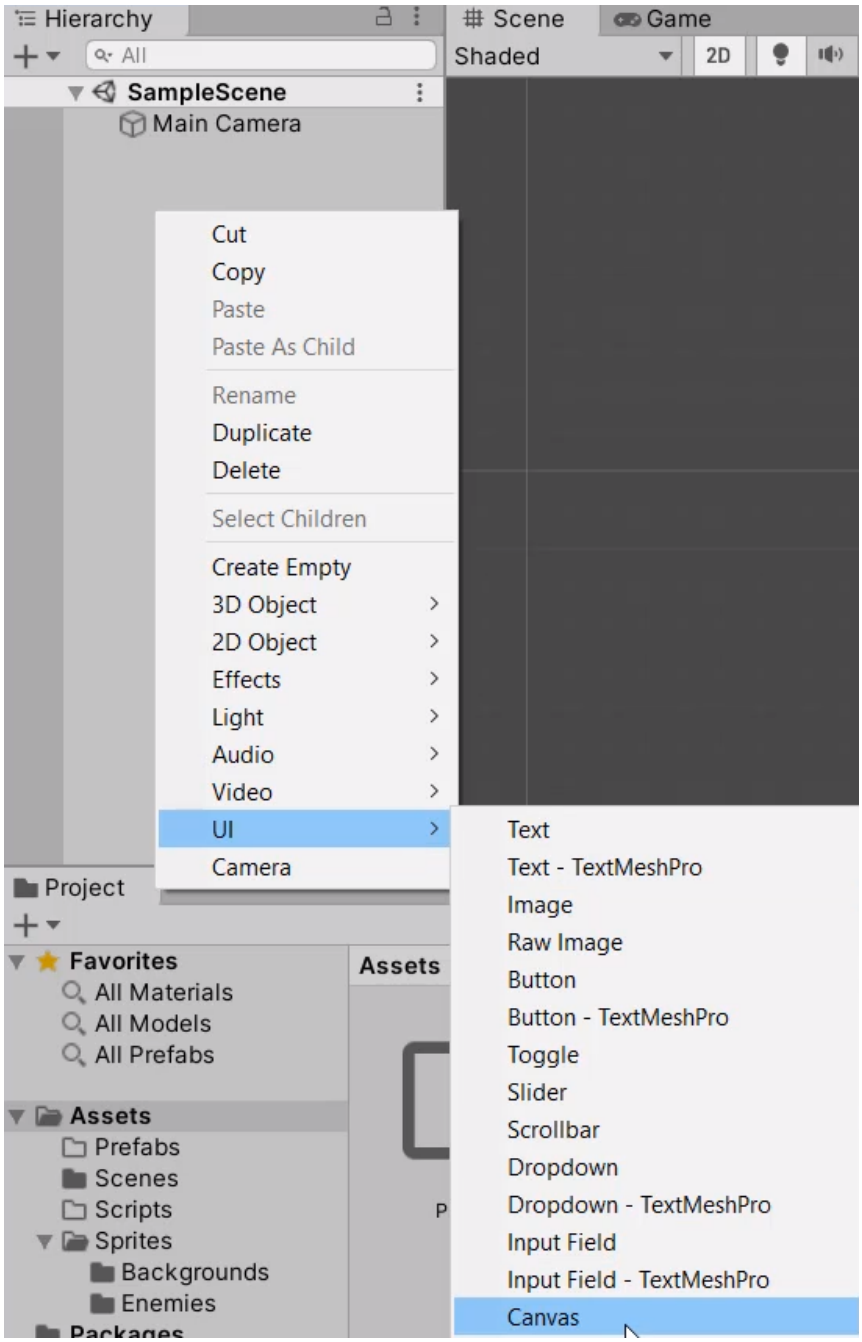


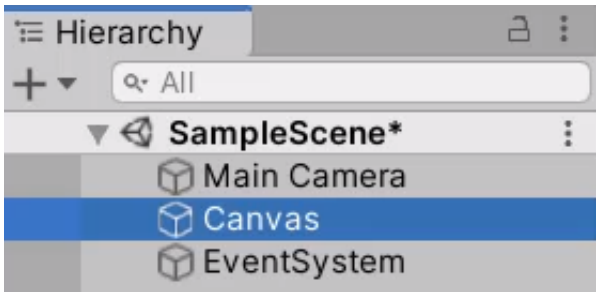
In this lesson, we're going to be setting up the visual **UI** (User Interface) of the game.

Creating UI Canvas

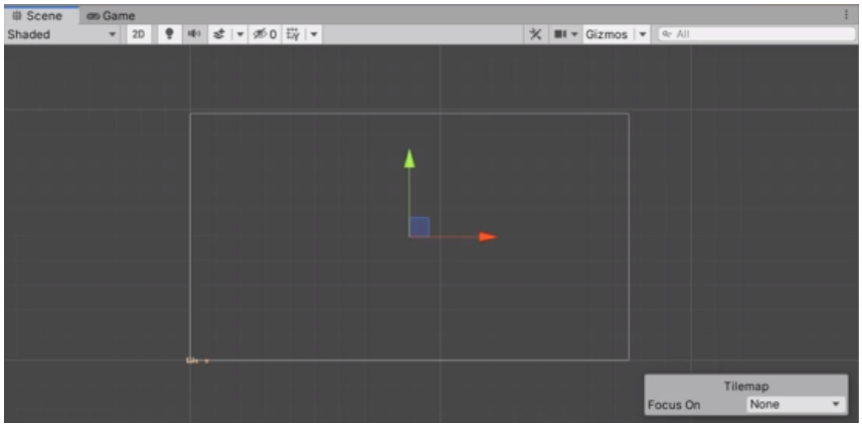
First of all, we're going to create a new gameObject called **Canvas,** which contains all of the UI elements.

To create a canvas, **right-click** on the **Hierarchy > UI > Canvas**.



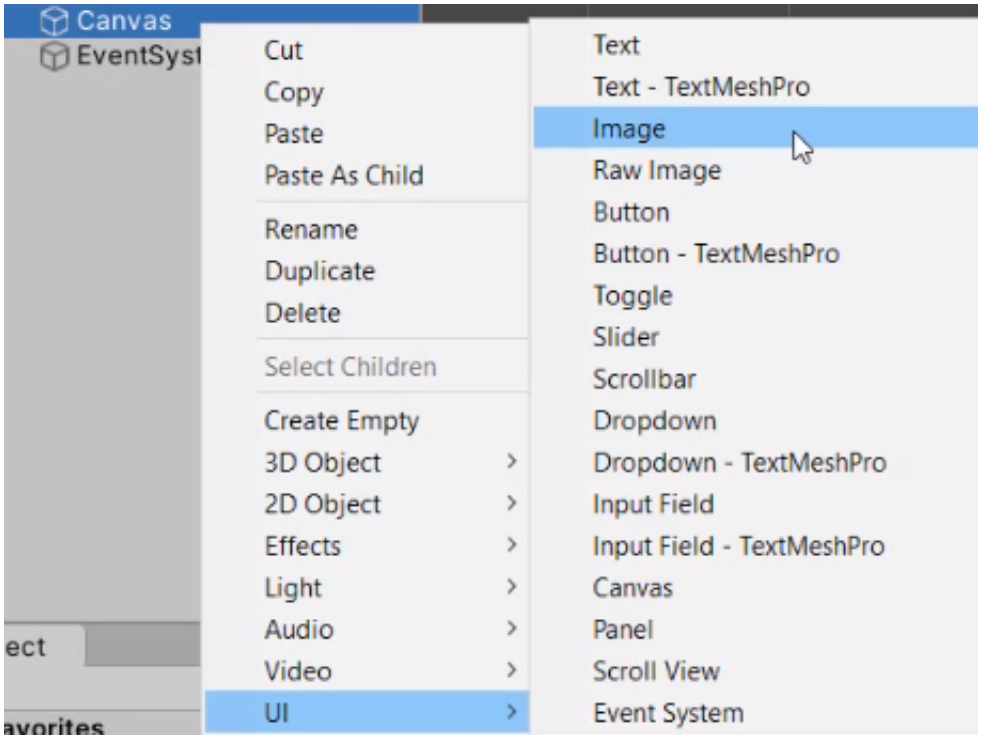


If you select the Canvas and **press F**, the screen will be zoomed out to **focus** on it.

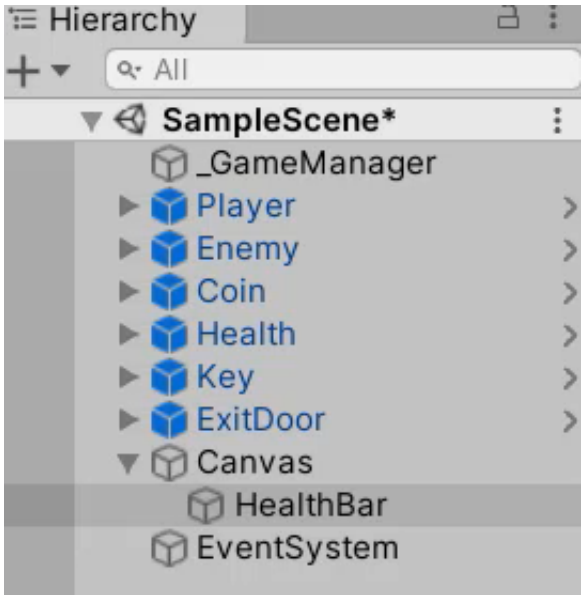


Adding Image To Canvas

To add an **Image** object to the Canvas, we're going to **right-click** on Canvas, and go to **UI > Image**.

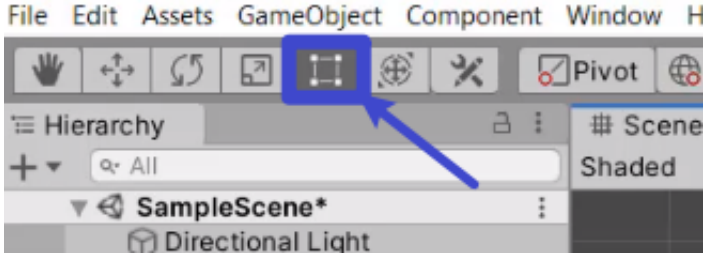


As you can see, it has created a brand new **Image** object, which is a **child** of Canvas. Let's **rename** this Image to '**HealthBar**'.

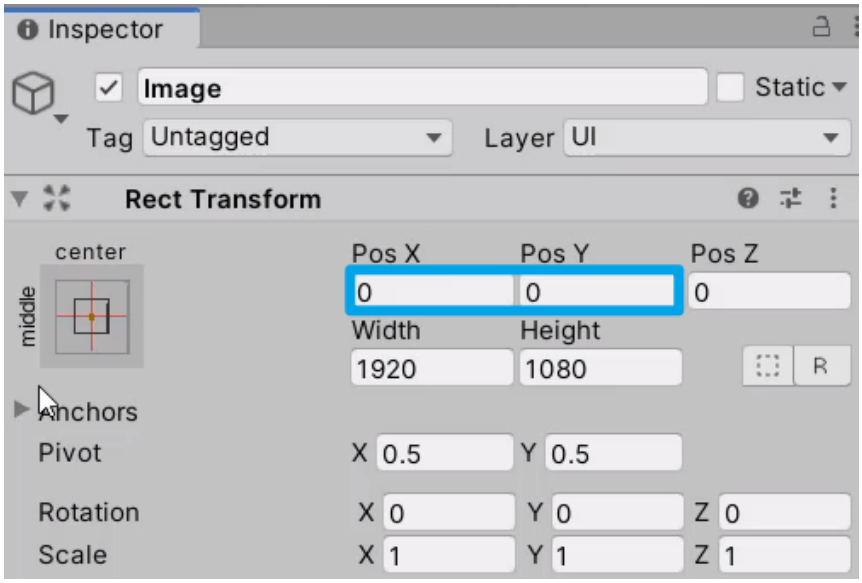


Canvas Components

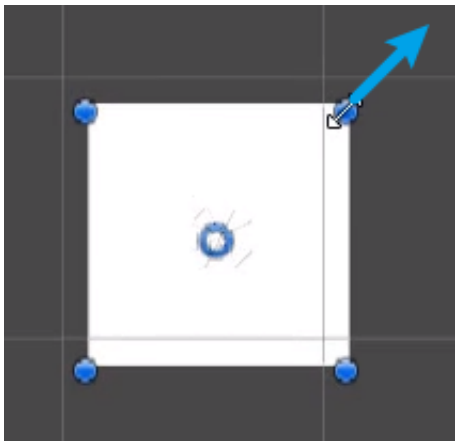
To change the position of the element, you can drag it around with the **Rect tool** selected:



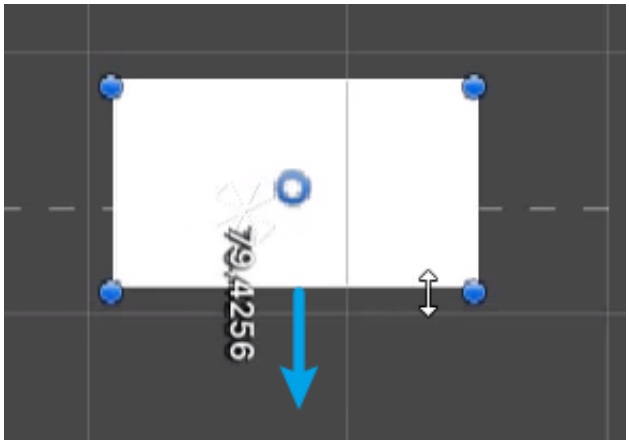
Alternatively, you can set the position using **Rect Transform** in the Inspector.



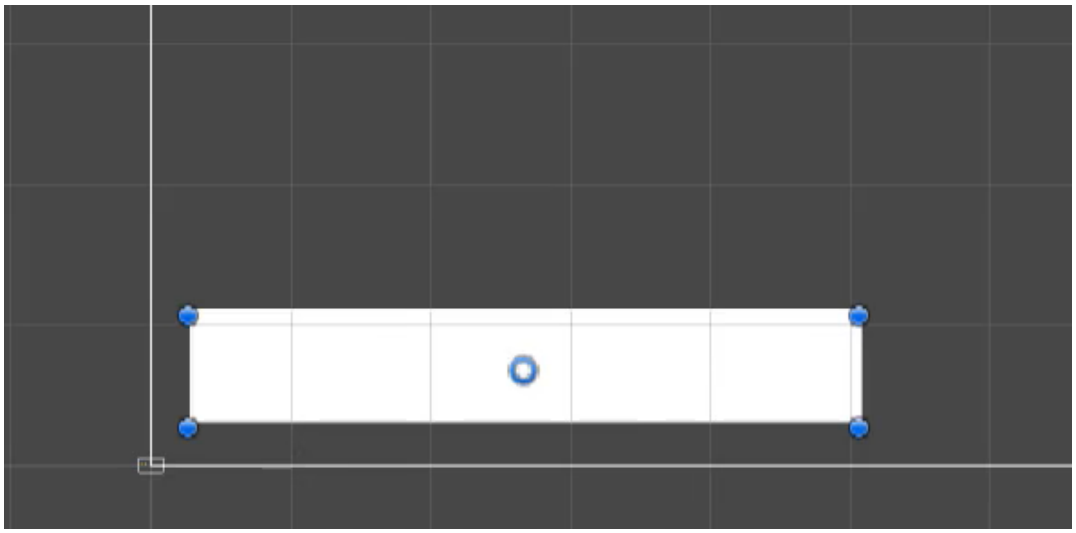
You can also click and drag on the **blue circles** to change the **size** of the image.



Or you can click and drag on the **sides** to resize them along that specific axis.

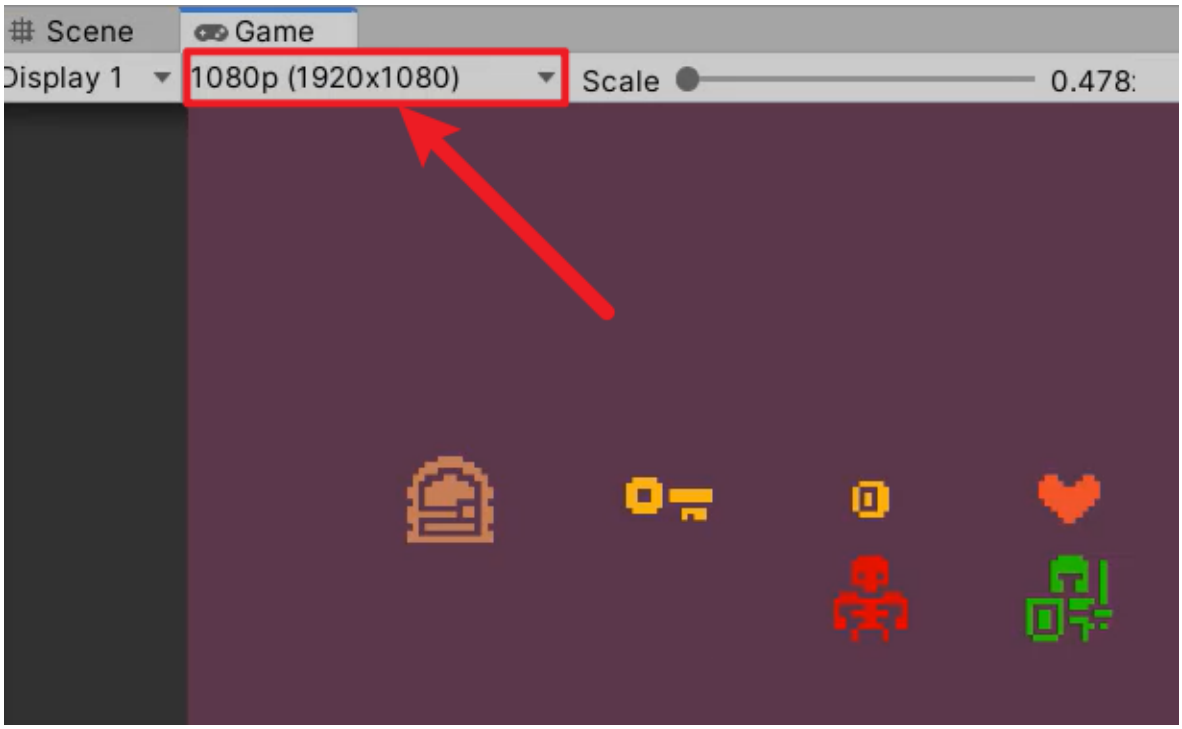


Since this image is going to be our **health bar**, we want to stretch it **horizontally** and place it at the **bottom left** corner of the screen.



Changing Screen Resolution

If we change our **aspect ratio** to something else– e.g. **4:3**, you'll see that the image is zoomed in a lot.



This is because the image still **maintains** the exact pixel dimensions that we have given it, although the screen has changed its size. Let's set the screen size to be **1920 x 1080**.

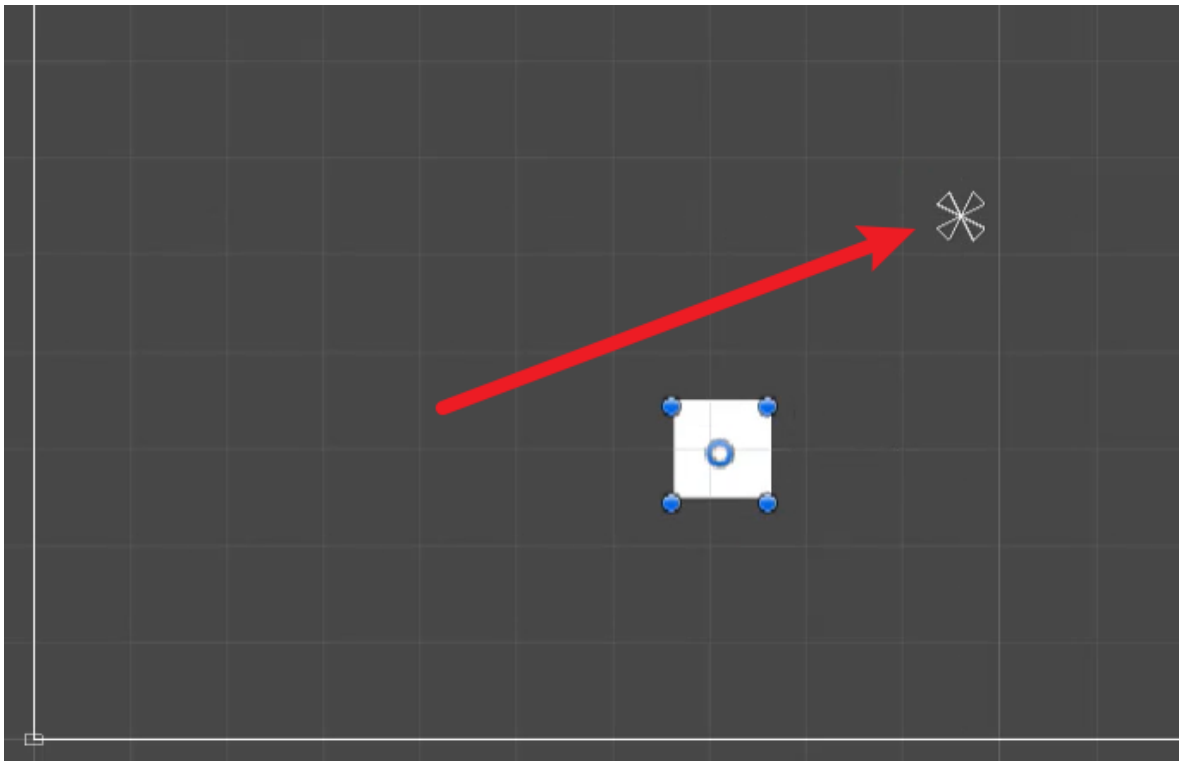
To make the image **scale** with the screen size, we need to set the image to be **anchored** to a certain corner of the screen.

Modifying Anchor Point

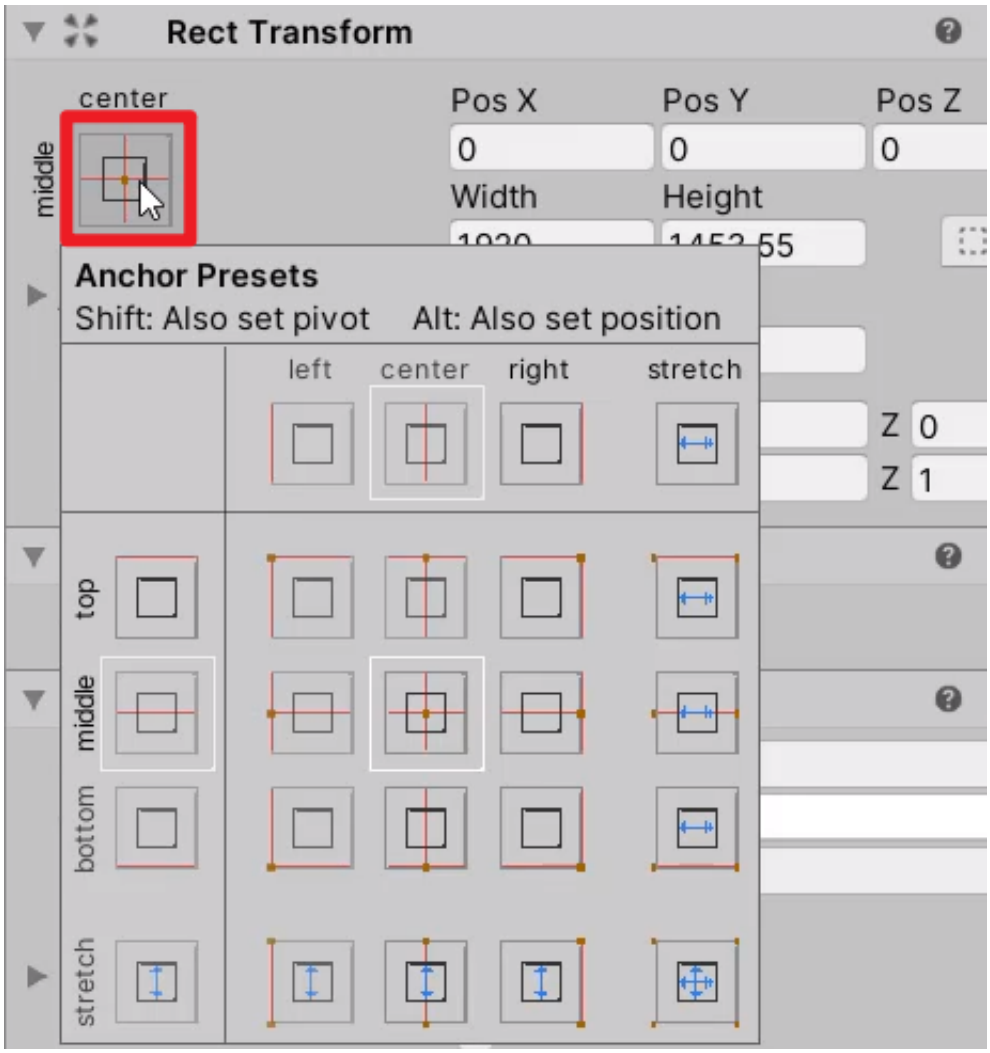
The **Anchor Handles** are represented by four triangles as shown below:

These handles ensure that our UI element is **positioned** and **scaled** appropriately to fit varying resolutions (aspect ratios).

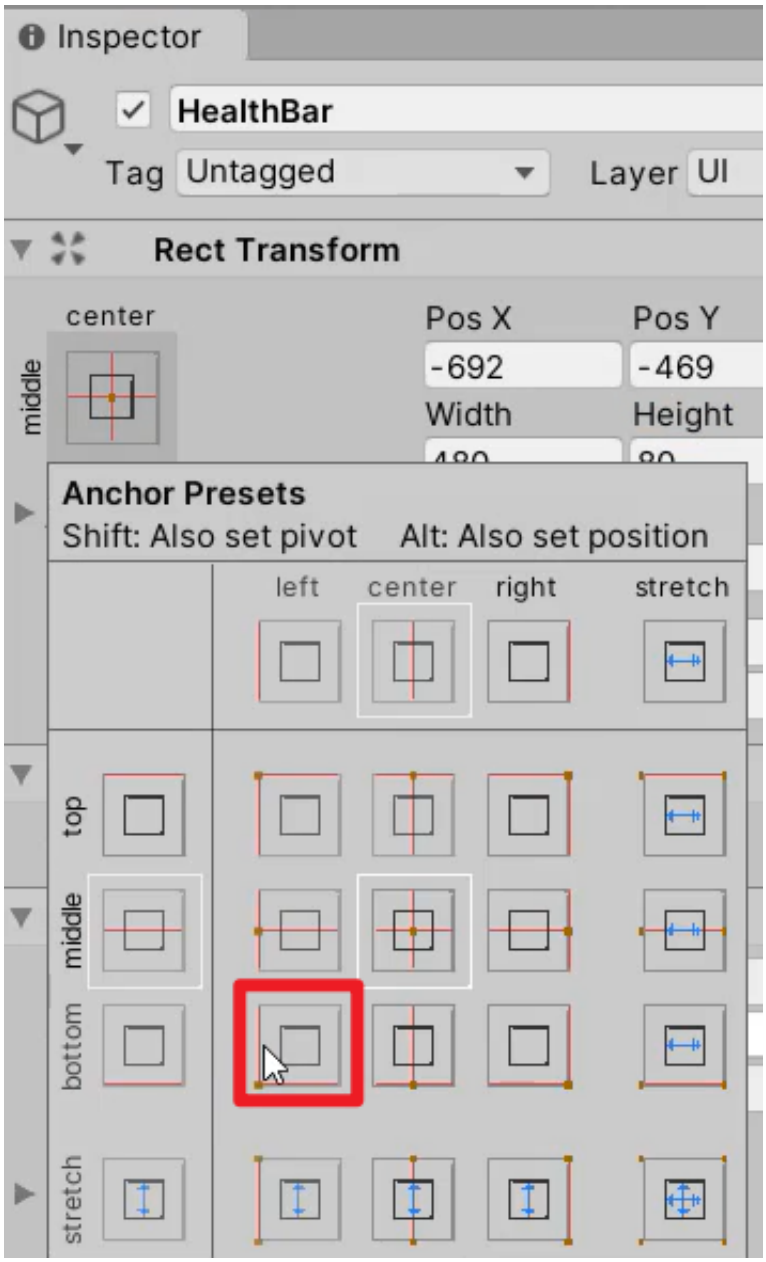
The anchor handles of our image are currently positioned at the **center** of the screen.

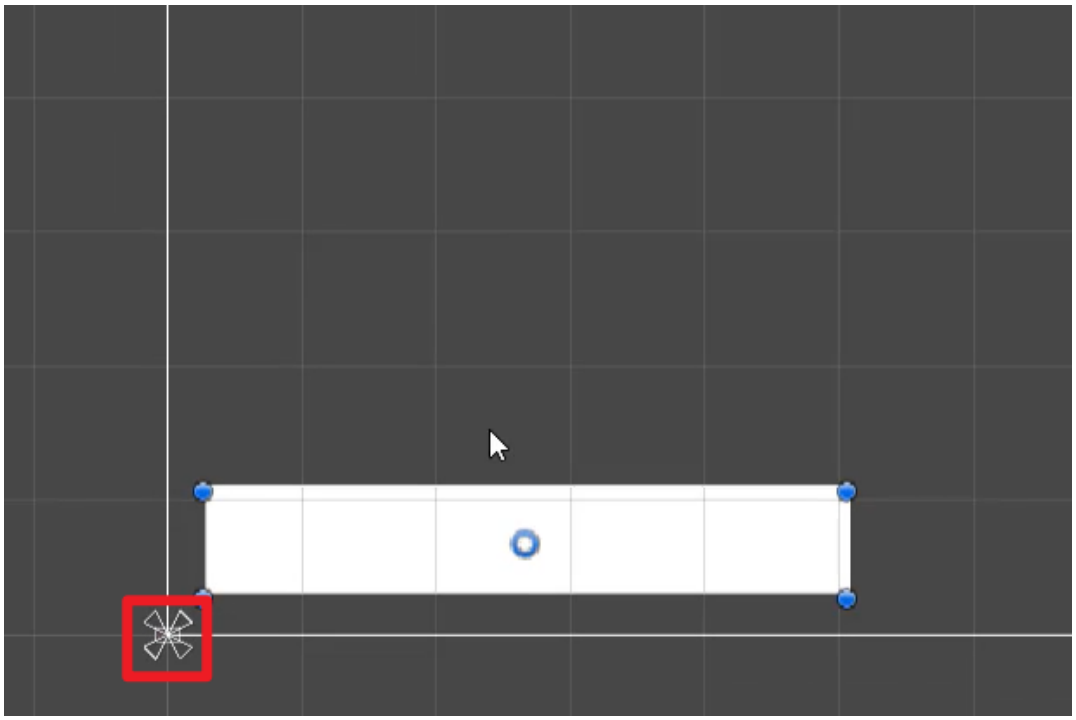


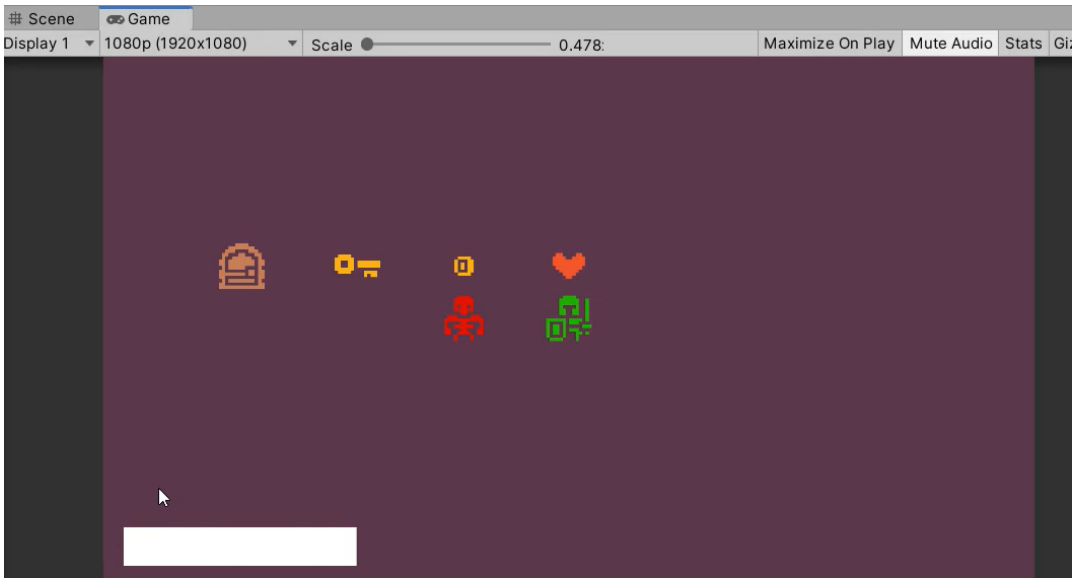
To change that, we can click on the **Anchor Presets** icon in the **Rect Transform**:



And click on the **pivot** icon on the bottom left corner, which snaps our anchors to that specific corner of the canvas.

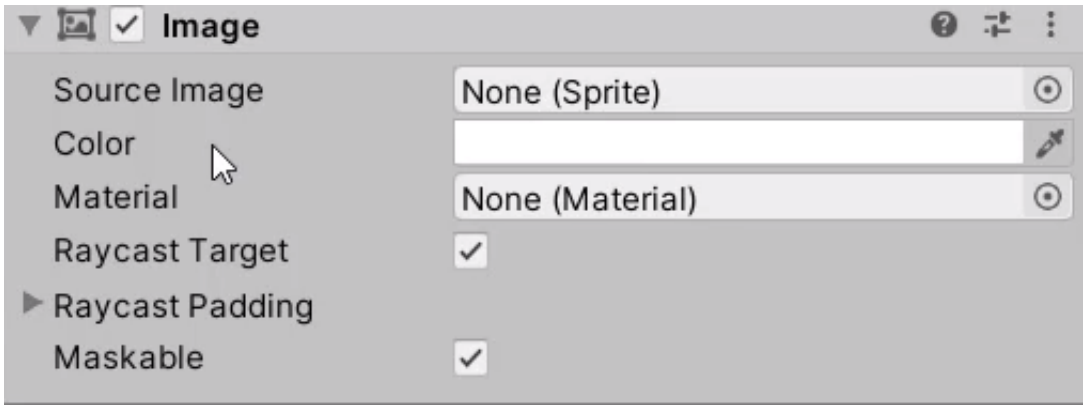




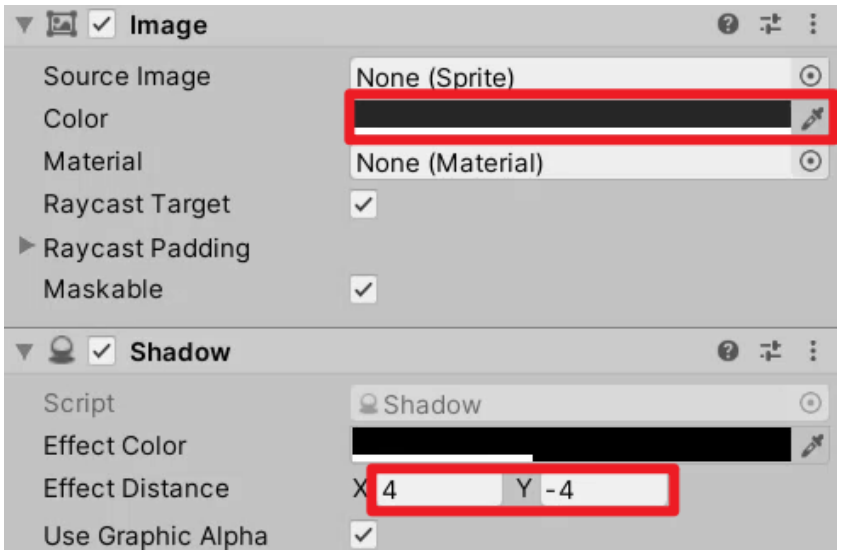


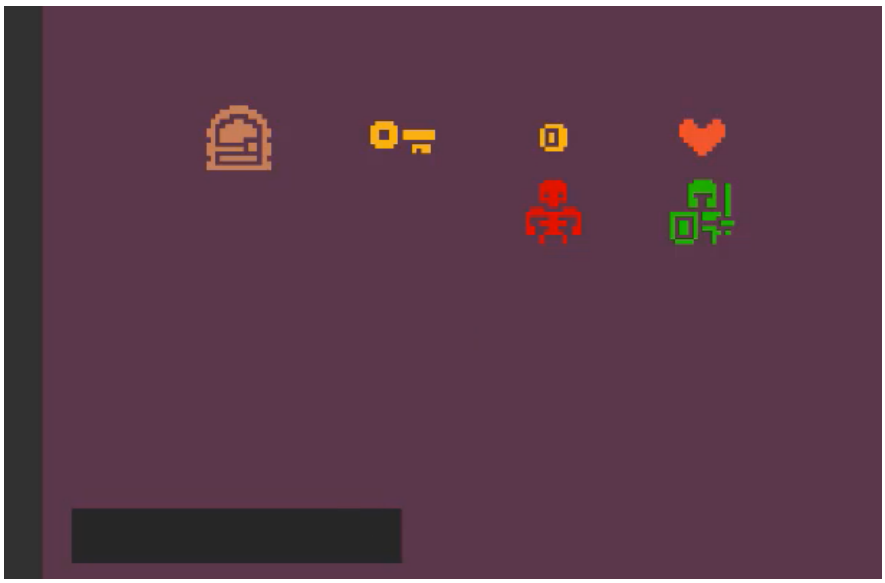
Creating A Health Bar

And inside of the Inspector, you'll see that we have an **Image** component. This component will give us the visual for this UI element.

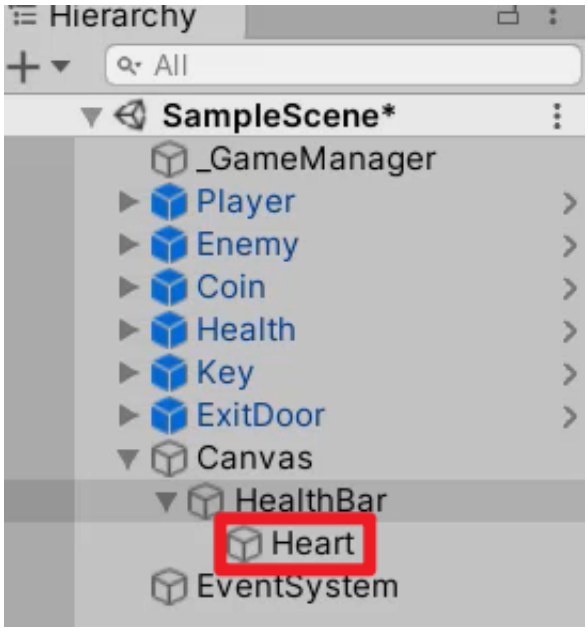


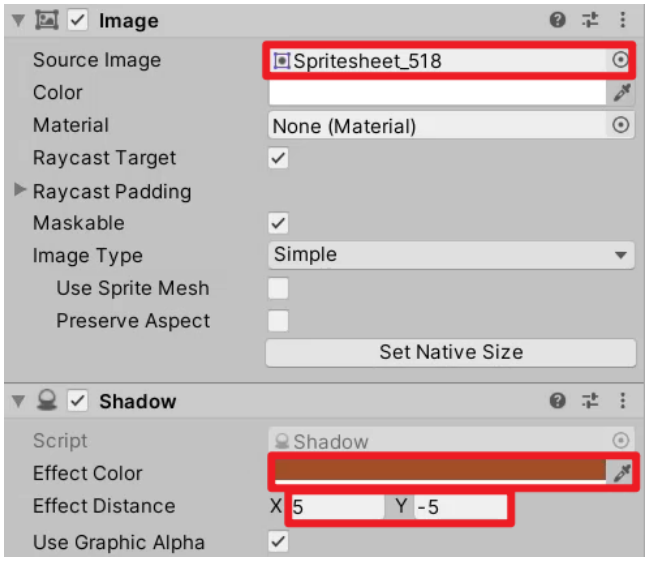
We're going to change the **Color** to be **black**, and add a **Shadow** component.

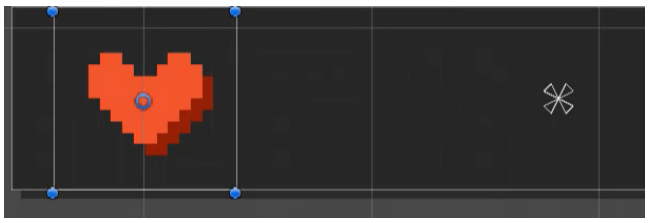




We will then add another **Image** of the same height to the **HealthBar** object, and assign a heart sprite (**Spritesheet_518**) and a **brown shadow** component.

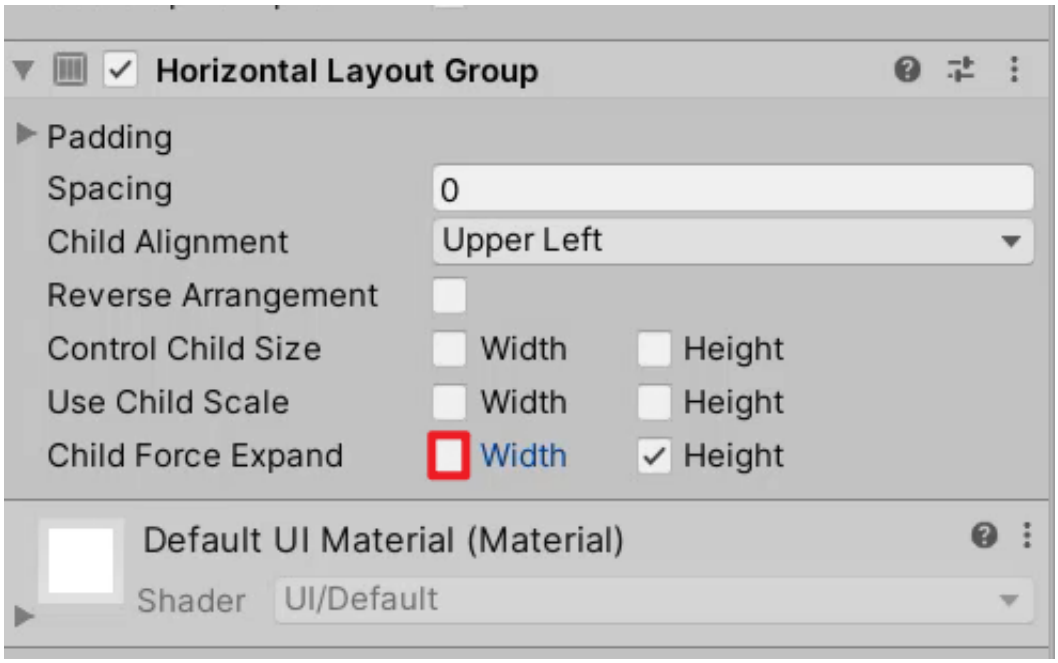


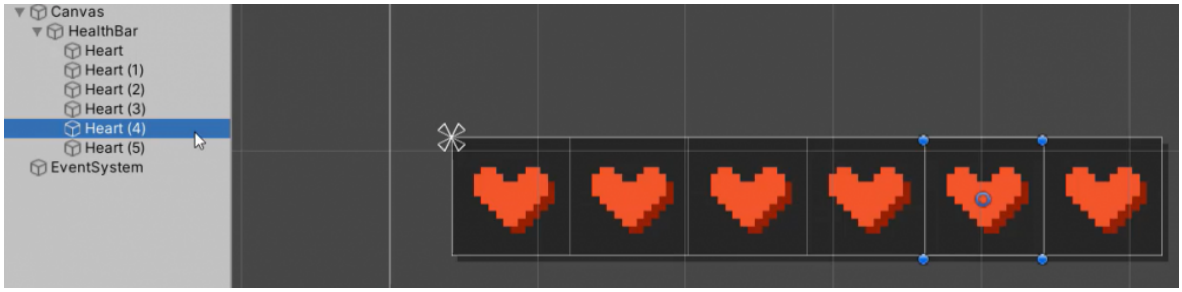




We'll then add a **Horizontal Layout Group** component to the heart, so we can **duplicate** (Ctrl+D or Cmd+D) it and the copies will get placed right next to another.

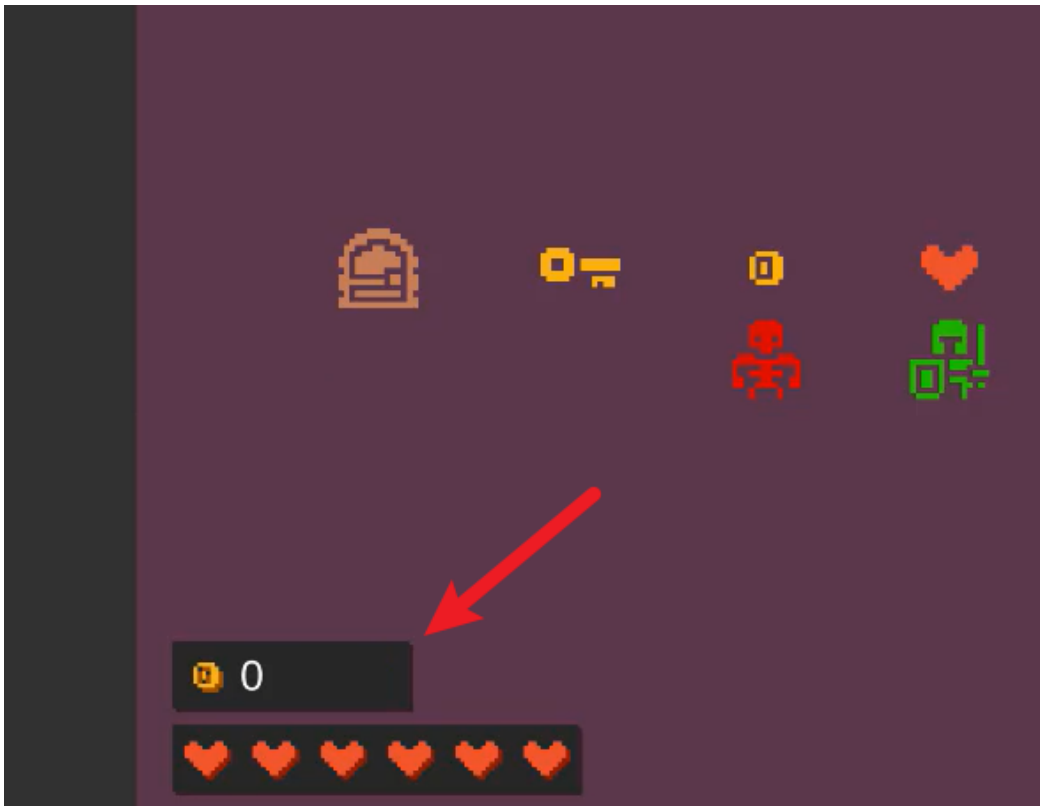
Make sure to **disable** the **Child Force Expand (Width)**, otherwise the copies will stretch across the screen.

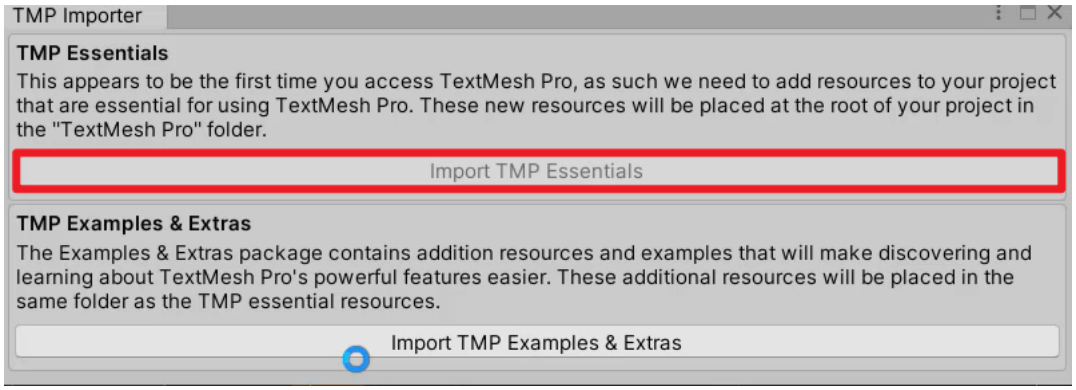




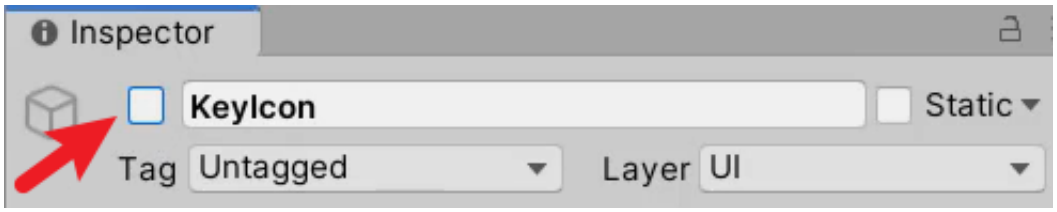
Creating A Coin Bar

Let's also create another bar UI for **Coins** using the exact same method. For the text component, we recommend you using **TMP (TextMesh Pro)** instead of Unity's default Text system because it provides more flexibility of font settings.





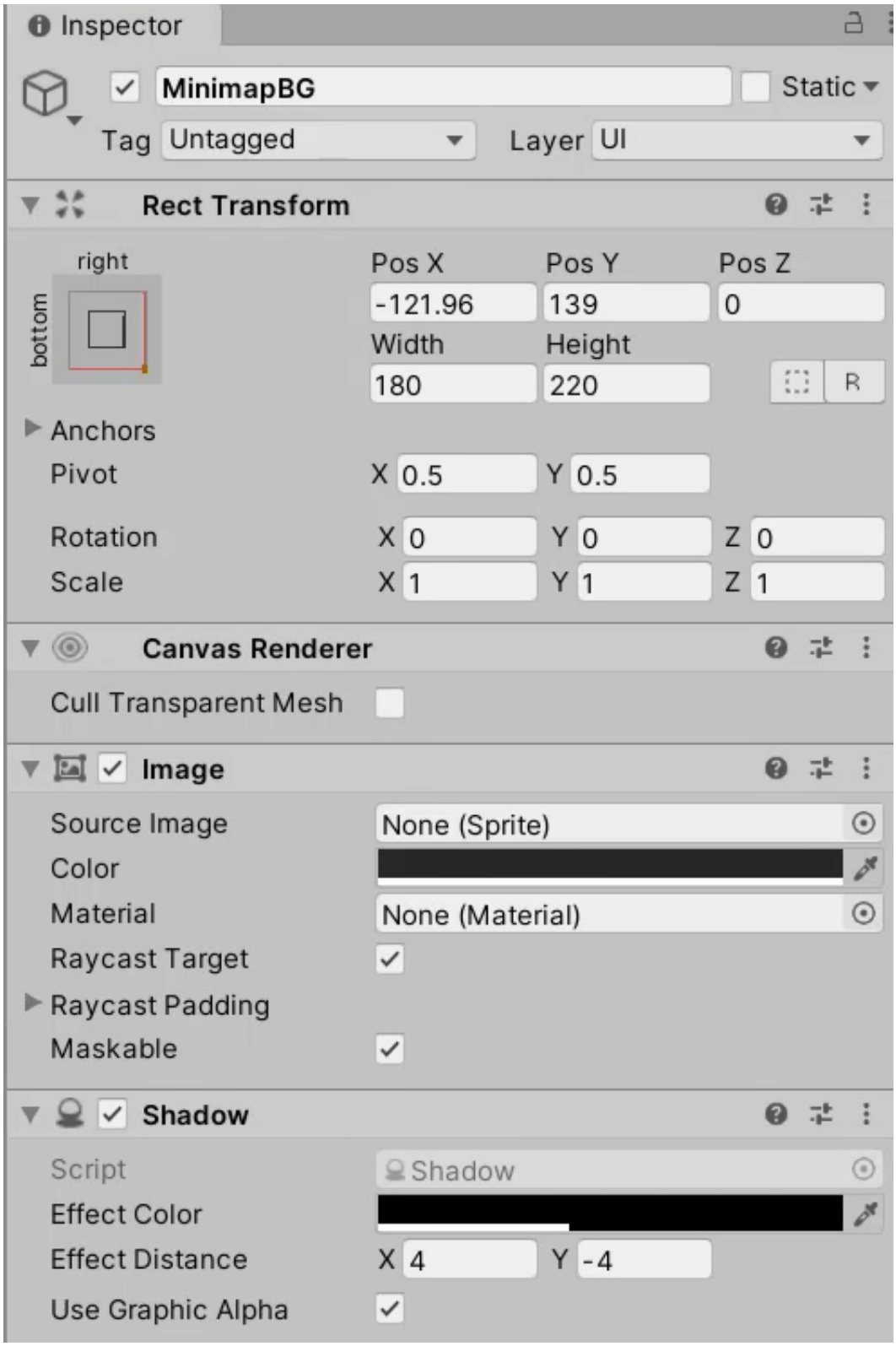
In addition, we're going to place a **key** icon inside the coin bar and **disable** it, so that it only appears once the player has collected the key.

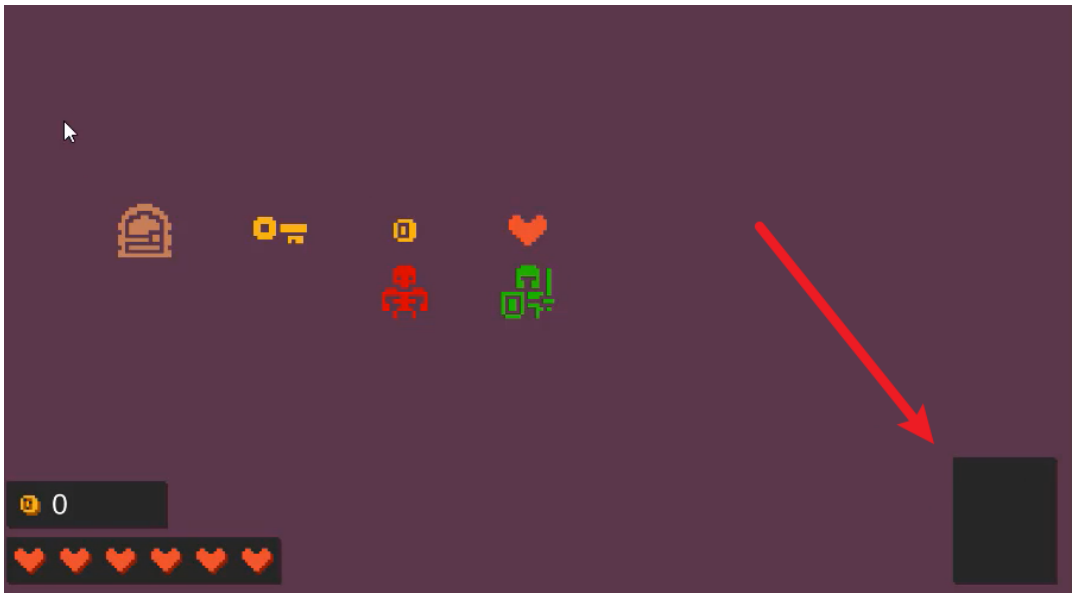


Creating A Minimap

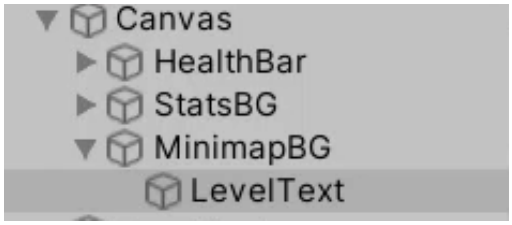
Now, as the last UI component, we're going to create a minimap.

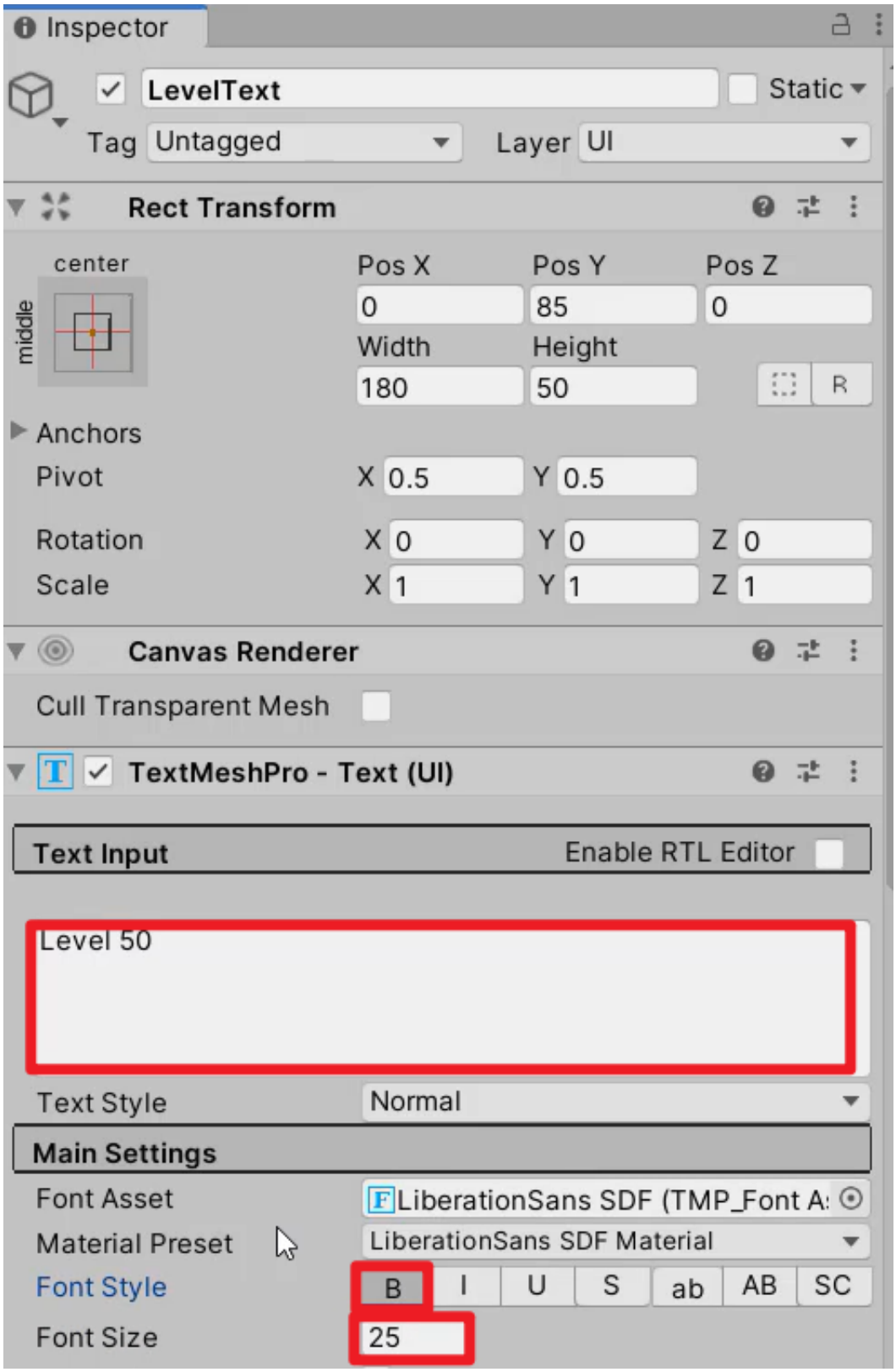
Let's go ahead and add another **Image** component called "**MinimapBG**". This image is going to be anchored at the **bottom right** corner of the screen.

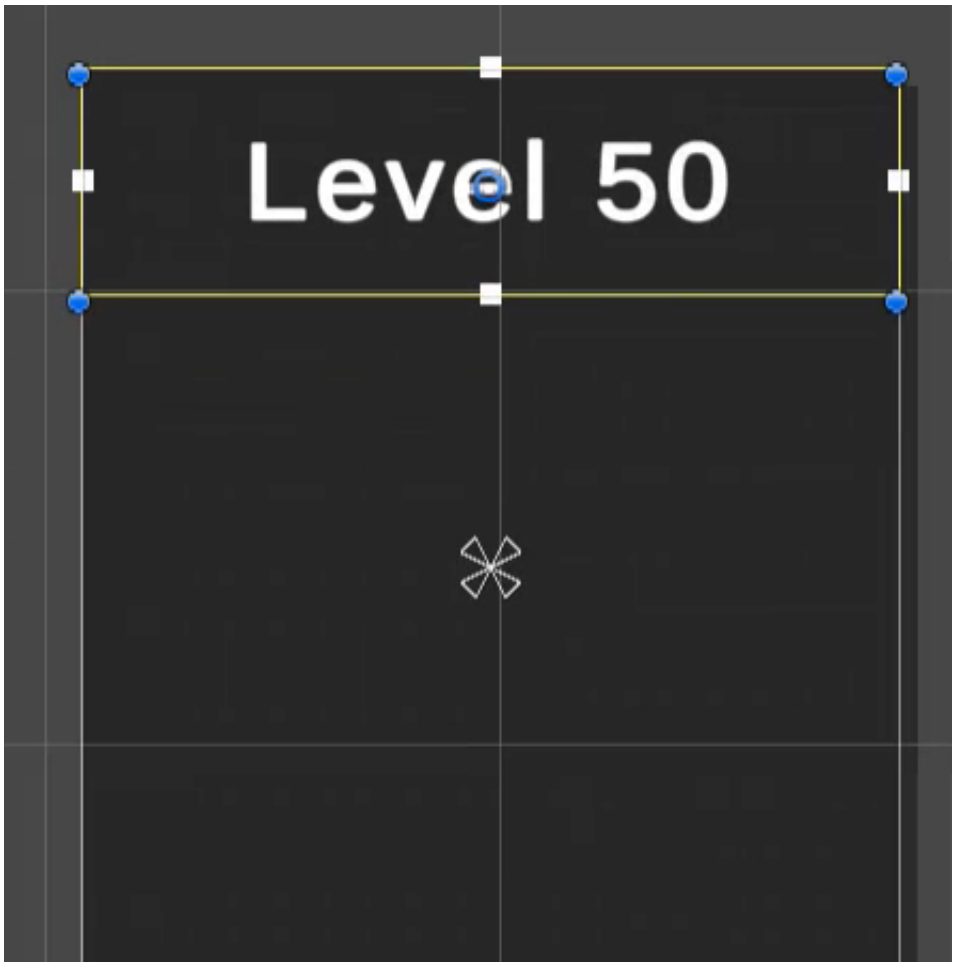




Then we'll add a **Text** component on top of the map to display which **level** we're currently on.

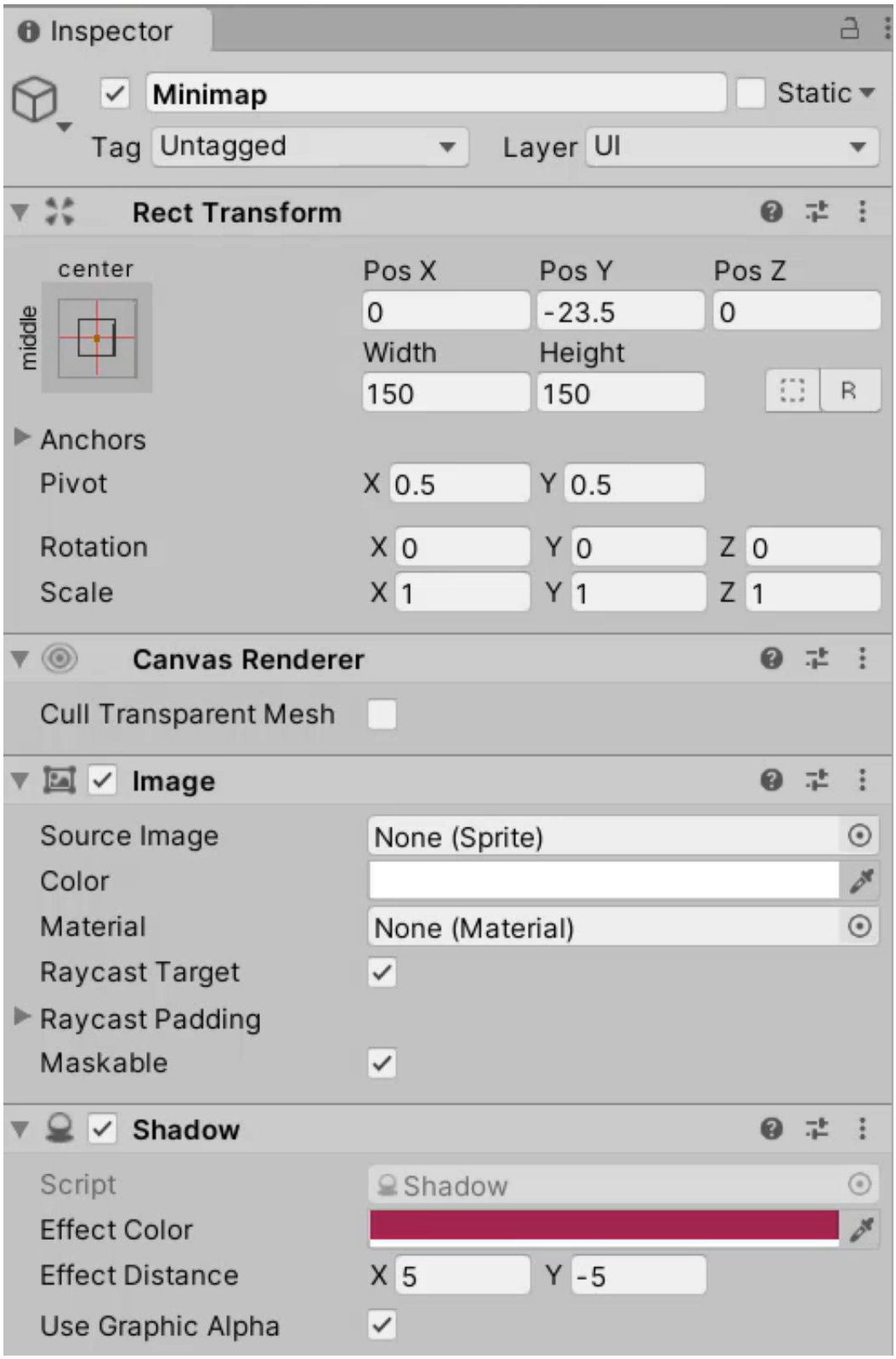


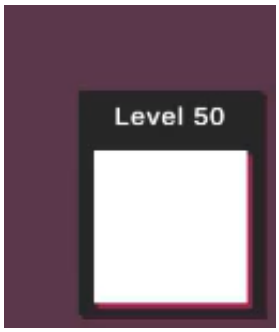




Finally, we're going to add another **Image** component on top of the background as the actual minimap.

We'll set the **size** to be **150 x 150**, and the **Shadow** color to be **dark pink**.

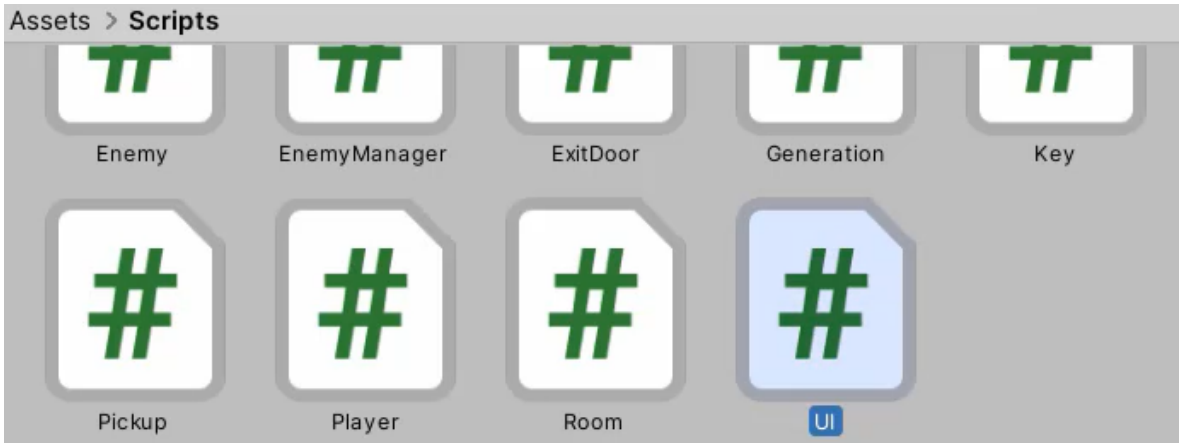




In the next lesson, we’re going to link this image to the script and have it display the current position.

In this lesson, we're going to be working on setting up a **UI** script. This script is going to update a **coinText**, a **healthBar** and a **key icon** when we collect it.

Let's go ahead and create a brand new C# script called "**UI**".



Let's attach it to the **Canvas** object, and open it up inside of Visual Studio.

Creating Variables

First, we need to import the **TMPro** and the **UI** library at the top of our script.

```
using TMPro;
using UnityEngine.UI;
```

We now need to declare **public** variables to reference our **hearts**, **keyIcon**, **levelText, coinText**, and the **map**.

```
public GameObject[] hearts;
public TextMeshProUGUI coinText;
public GameObject keyIcon;
public TextMeshProUGUI levelText;
public RawImage map;
```

Lastly, we're going to make a **static** UI instance for the **Singleton** pattern. This ensures that there is only one instance of UI canvas that is globally accessible and available at all times.


```
public static UI instance;
```

Then inside of the **Awake** function, we can assign this class to the one and only instance.

```
private void Awake()
{
    instance = this;
}
```

Updating Health UI

Now that our variables are all set up, let's create a new function called "**UpdateHealth**". As the name implies, this function will update our **healthBar** when we take damage or collect a health pickup.

```
// called when we take damage or collect a health pickup
public void UpdateHealth (int health)
{
    // for each heart inside the hearts list
    for(int x = 0; x < hearts.Length; ++x)
    {
        // activate to match the current HP
        hearts[x].SetActive(x < health);
    }
}
```

Then we can go over to the Player script and call this function whenever we add or subtract health.

```
// adds health to the player
// returns true or false for if the health can be added
public bool AddHealth (int amount)
{
    if(curHp + amount <= maxHp)
    {
        curHp += amount;
        // update UI
        UI.instance.UpdateHealth(curHp);
        return true;
    }

    return false;
}
```

```
// called when the enemy deals damage to the player
public void TakeDamage (int damageToTake)
{
    curHp -= damageToTake;
```



```
    // update UI
    UI.instance.UpdateHealth(curHp);

    StartCoroutine(DamageFlash());

    if(curHp <= 0)
        SceneManager.LoadScene(0);
}
```

Updating Coin UI

And similarly, we're going to create a function that gets called when we pickup a coin. Note that we need to convert the amount of coins (**int**) into a string (**ToString**) when we're updating the coinText.

```
// called when we pickup a coin
public void UpdateCoinText (int coins)
{
    coinText.text = coins.ToString();
}
```

Again, we'll call this function inside the **Player** script where we add coins to the player:

```
// adds coins to the player
public void AddCoins (int amount)
{
    coins += amount;
    UI.instance.UpdateCoinText(coins);
}
```

Toggling Key Icon

We also need to have a function that gets called when we collect the **key**. The parameter we send over in this case would be a **boolean,** which will intially be false and be **SetActive** when we collect the key.

```
// called when we collect the key
public void ToggleKeyIcon (bool toggle)
{
    keyIcon.SetActive(toggle);
}
```

This should be called inside the **Key** script, where it detects the collision with the player:

```
public class Key : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
```

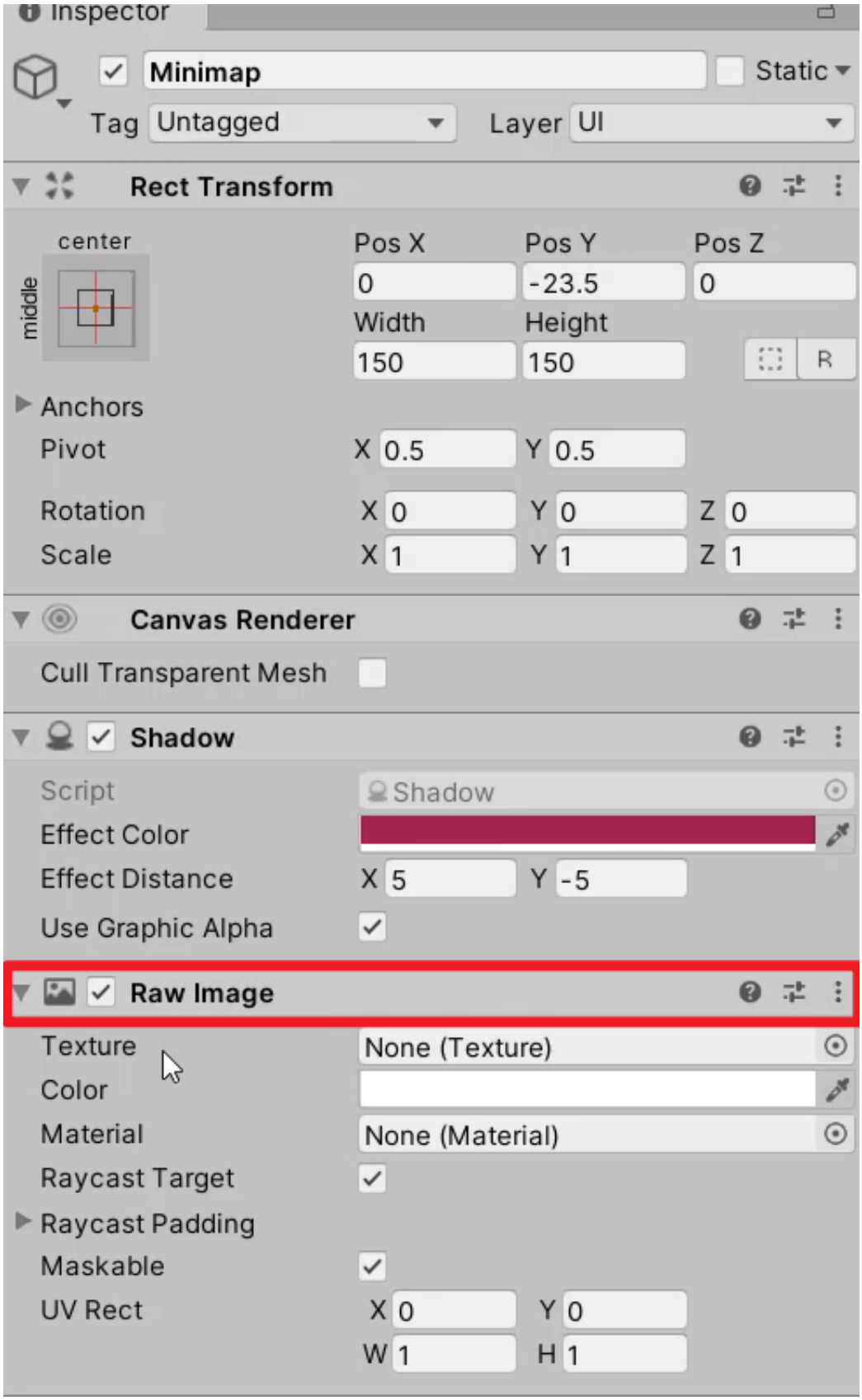


```
        if(collision.CompareTag("Player"))
        {
            collision.GetComponent<Player>().hasKey = true;
            UI.instance.ToggleKeyIcon(true);
            Destroy(gameObject);
        }
    }
}
```

Updating Minimap

Finally, we're going to update our minimap so that it displays the current position of the player.

Since we're going to be passing the minimap data as a **Texture**, we need to select the minimap object and replace the **Image** component with the **RawImage** component that can receive a Texture data.



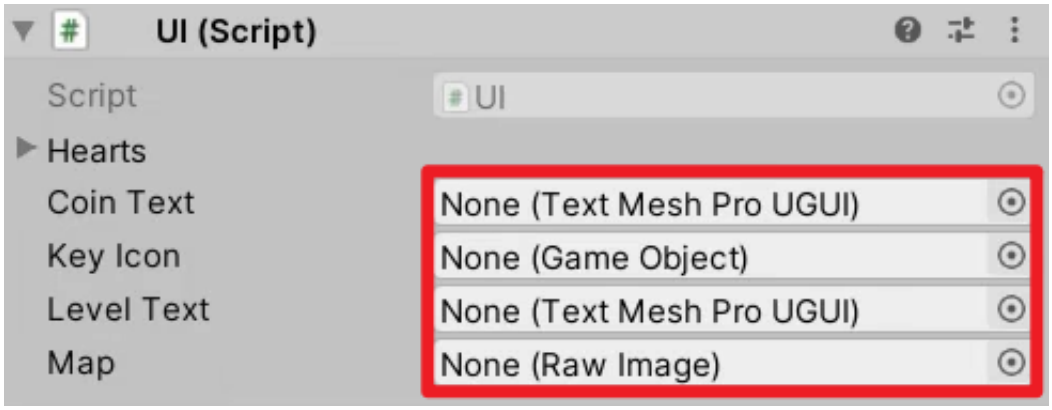
Then we can go back to the UI script and update the **levelText** to display the current level we're on.

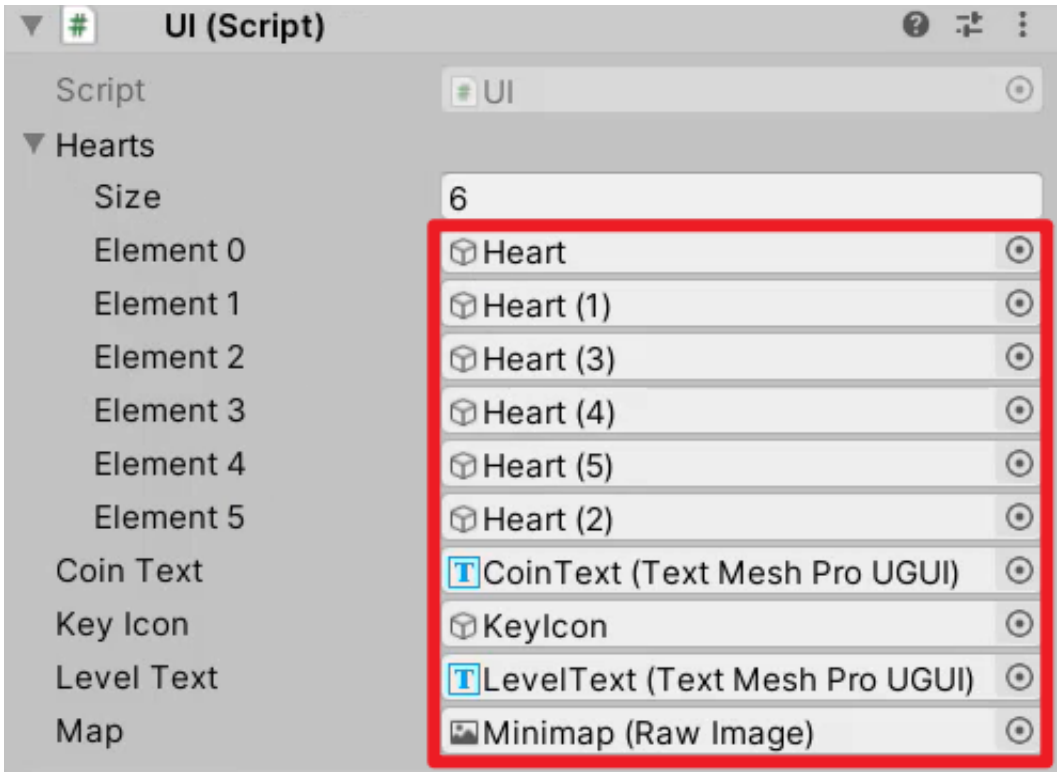
```
// called at the start of the level
```



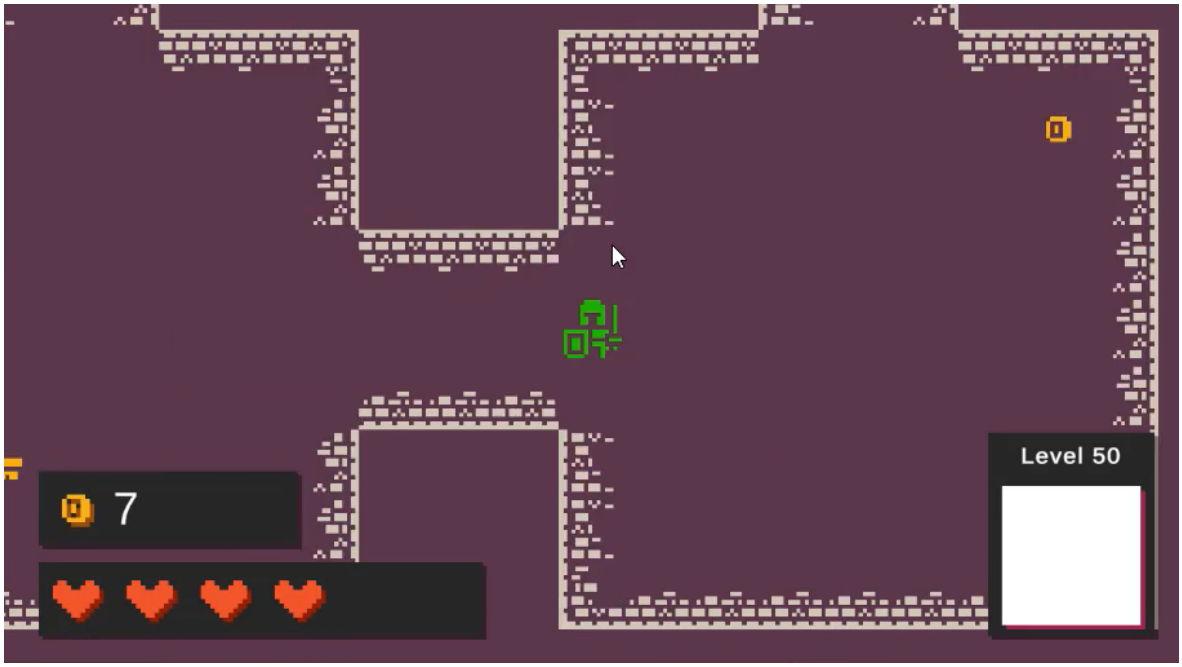
```
public void UpdateLevelText (int level)
{
    levelText.text = "Level " + level;
}
```

Make sure to **save** the script and **drag in** all the UI objects into the corresponding fields.





If you now press **Play**, you'll be able to see that the coins and the health bar gets updated correctly.



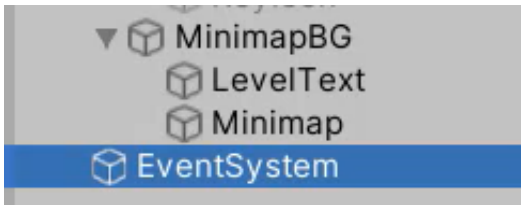
In the next lesson, we will continue working on the minimap texture.

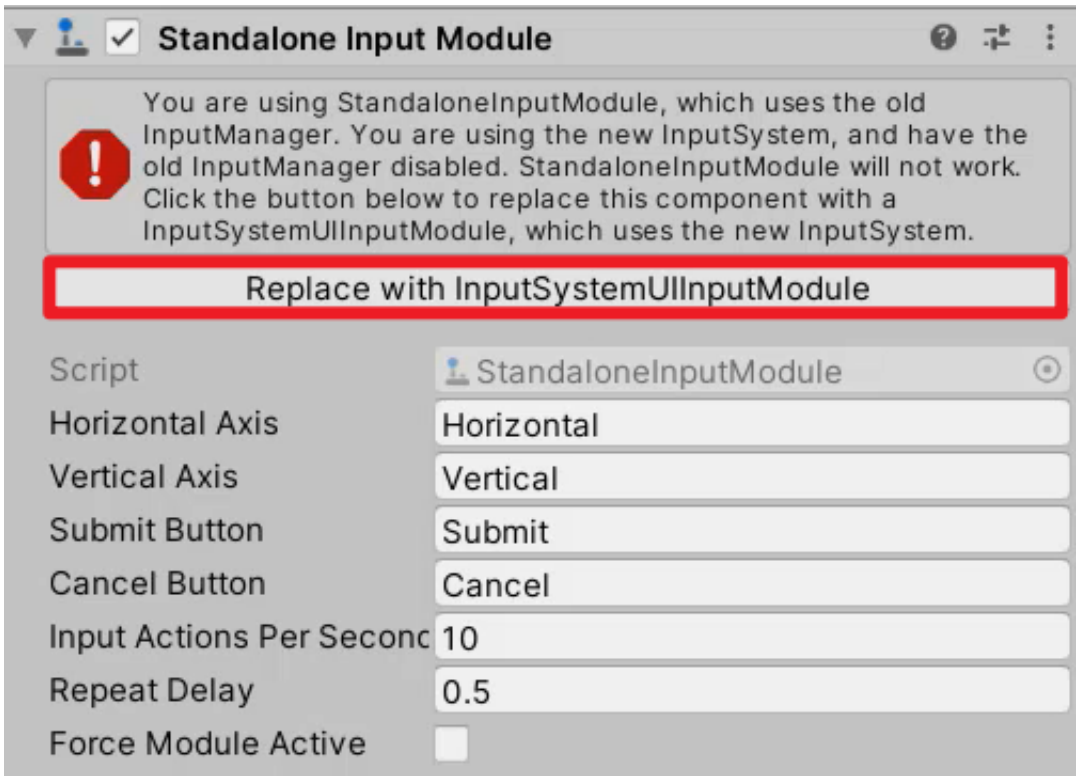
Updating Event System

If you've encountered the error "**InvalidOperationException: You are using StandaloneInputModule**", this is because we're using the new Input system.

To fix this, we can simply select the **EventSystem** object and click **'Replace With**

InputSystemUIInputModule'.

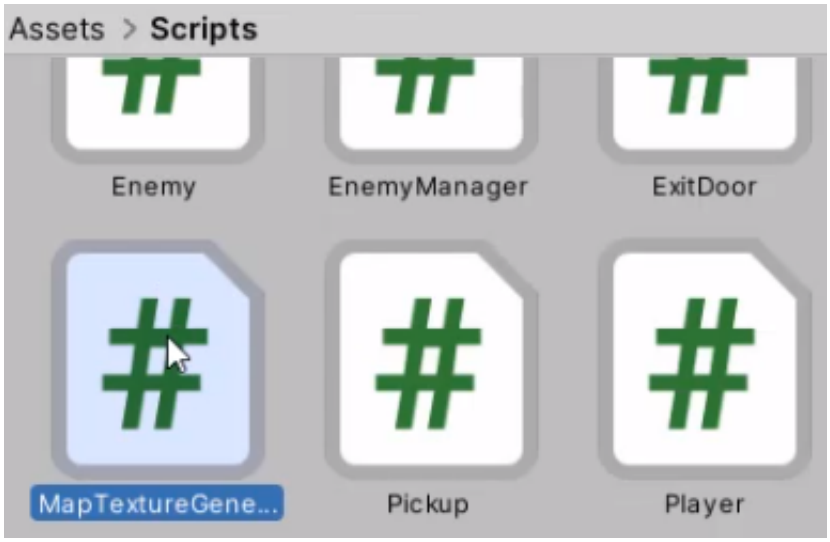




In this lesson, we're going to be setting up our minimap. This is going to display a layout of all the rooms, as well as showing us where we are in the level.

Map Texture Generator

To begin, we're going to create a brand new **script** called "**MapTextureGenerator**".



Inside the script, we're going to create a **static** function called "**Generate**", which returns a **Texture2D** of the same **width** and **height** as our map.

Since this image is very small (around 7 x 7 pixels), we want to remove the pixel filtering by setting the **FilterMode** to **Point.** This will make the image look pixellated, while the default filter mode will blur the pixel details to make it seem a higher resolution.

```
public static Texture2D Generate (bool[,] map, Vector2 playerRoom)
{
    // create a new Texture2D of the map's size
    Texture2D tex = new Texture2D(map.GetLength(0), map.GetLength(1));
    // pixellate the image
    tex.filterMode = FilterMode.Point;

    return tex;
}
```

Now we need to basically fill in this texture using a **Color array** (**pixels**).

For each pixel inside the array, we can divide its index (**i**) by the width of the map (**map.GetLength(0)**), and the remainder will give us the current **column** we're on. Similarly, the remainder after dividing the index by the height of the map will give us the current **row**.

This can be simply calculated using the **modulo %** operator, which returns the remainder after dividing its left-hand side by its right-hand side.

```
public static Texture2D Generate (bool[,] map, Vector2 playerRoom)
{
    Texture2D tex = new Texture2D(map.GetLength(0), map.GetLength(1));
```



```
    tex.filterMode = FilterMode.Point;

    // make a color array of the map's size
    Color[] pixels = new Color[map.GetLength(0) * map.GetLength(1)];

    // for each pixel inside the array
    for(int i = 0; i < pixels.Length; ++i)
    {
        // get the current column
        int x = i % map.GetLength(0);
        // get the current row, rounding down to the nearest int
        int y = Mathf.FloorToInt(i / map.GetLength(1));
   }

    return tex;
}
```

If this is the room that the player is **currently in**, then we can set this pixel color to be **green**. Otherwise, if it's just the normal room, we can set it to be **white**. And if the room doesn't exist at all, we can set it to be **clear** so that it's going to be **invisible**.

```
public static Texture2D Generate (bool[,] map, Vector2 playerRoom)
{
    Texture2D tex = new Texture2D(map.GetLength(0), map.GetLength(1));
    tex.filterMode = FilterMode.Point;

    Color[] pixels = new Color[map.GetLength(0) * map.GetLength(1)];

    for(int i = 0; i < pixels.Length; ++i)
    {
        int x = i % map.GetLength(0);
        int y = Mathf.FloorToInt(i / map.GetLength(1));

        // is this the room that the player is in?
        if(playerRoom == new Vector2(x, y))
            // if so, set this pixel color to be green.
            pixels[i] = Color.green;
        else
            // otherwise, set it to be white if the room exists, or set it to be invi
sible if it doesn't exist.
            pixels[i] = map[x, y] == true ? Color.white : Color.clear;
    }

    return tex;
}
```

Finally, we can apply the pixels to the actual texture using the built-in **SetPixels** function and the **Apply** function of the Texture2D variable.

```
public static Texture2D Generate (bool[,] map, Vector2 playerRoom)
{
```



```
    Texture2D tex = new Texture2D(map.GetLength(0), map.GetLength(1));
    tex.filterMode = FilterMode.Point;

    Color[] pixels = new Color[map.GetLength(0) * map.GetLength(1)];

    for(int i = 0; i < pixels.Length; ++i)
    {
        int x = i % map.GetLength(0);
        int y = Mathf.FloorToInt(i / map.GetLength(1));

        if(playerRoom == new Vector2(x, y))
            pixels[i] = Color.green;
        else
            pixels[i] = map[x, y] == true ? Color.white : Color.clear;
    }

    // apply the pixels to the actual texture.
    tex.SetPixels(pixels);
    tex.Apply();

    return tex;
}
```

Updating The Raw Image Minimap

Now we can go over to the **Generation** script and set the raw image minimap to show this texture, sending over our map data.

```
// called at the start of the game - begins the generation process
public void Generate ()
{
    map = new bool[mapWidth, mapHeight];
    CheckRoom(3, 3, 0, Vector2.zero, true);
    InstantiateRooms();
    FindObjectOfType<Player>().transform.position = firstRoomPos * 12;

    // apply the texture to the raw image minimap.
    UI.instance.map.texture = MapTextureGenerator.Generate(map, firstRoomPos);
}
```

Then we're going to create a new function called "**OnPlayerMove**" to update the minimap whenever the player moves. First, we need to get the player's position, and then figure out which room coordinate they're currently in. Then we can call the **Generate** function of the **MapTextureGenerator** to apply the update, sending over the updated map data.

```
public void OnPlayerMove ()
{
    // get the position of the player
    Vector2 playerPos = FindObjectOfType<Player>().transform.position;
    // get the position of the room that the player is in, in terms of map scale.
    Vector2 roomPos = new Vector2(((int)playerPos.x + 6) / 12, ((int)playerPos.y + 6)
```

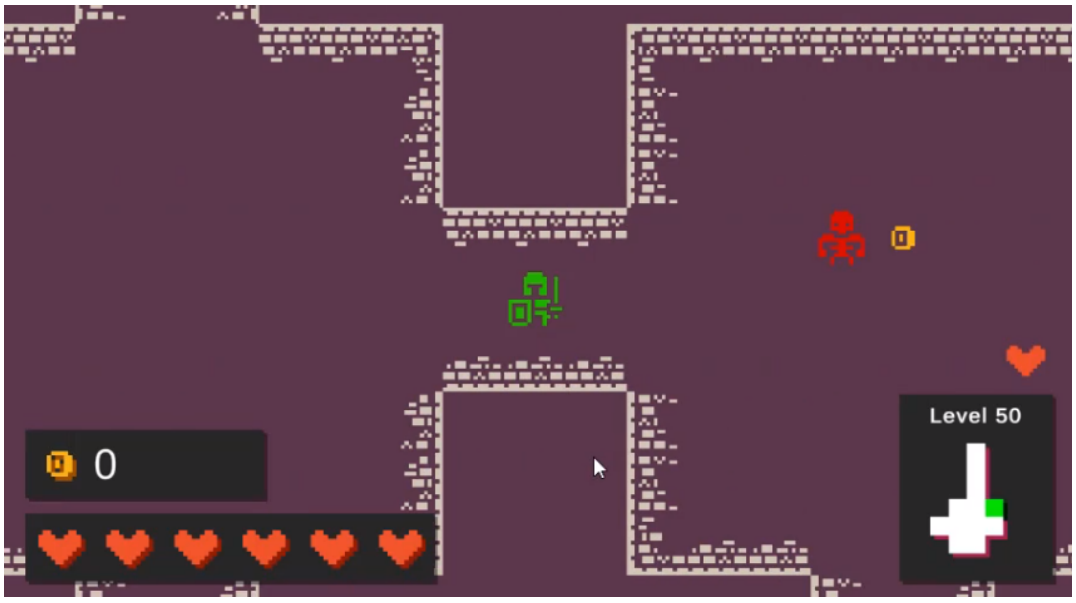


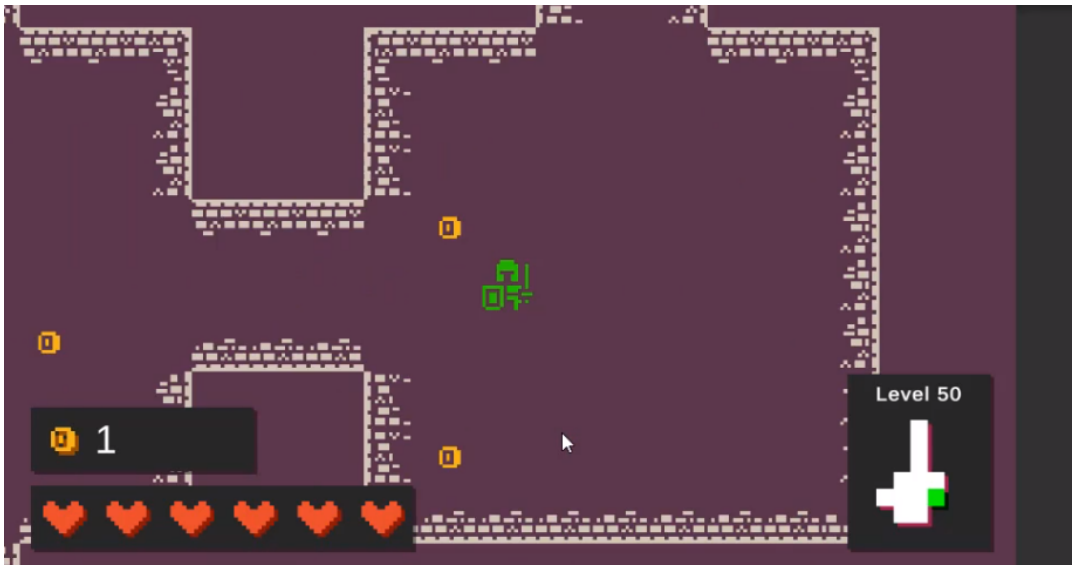
```
 / 12);

    // generate a newer version of the map
    UI.instance.map.texture = MapTextureGenerator.Generate(map, roomPos);
}
```

And just as with the enemy attack, this **OnPlayerMove** function should be called inside the **Move** function of the **Player** script.

```
// called when we want to move in a certain direction
void Move (Vector2 dir)
{
    // make sure we're not hitting a wall or enemy
    RaycastHit2D hit = Physics2D.Raycast(transform.position, dir, 1.0f, moveLayerMask
);

    if(hit.collider == null)
    {
        transform.position += new Vector3(dir.x, dir.y, 0);
        EnemyManager.instance.OnPlayerMove();
        // update minimap
        Generation.instance.OnPlayerMove();
    }
}
```

Now if you **Save** the script and press **Play**, you'll be able to see that the map updates whenever you move into a new room.





The instructions in the video for this particular lesson have been updated, please see the lesson notes below for the corrected instruction.

In this lesson, we're going to be setting up a menu scene. Make sure to have the **TextMeshPro** package up-to-date inside **Package Manager**.

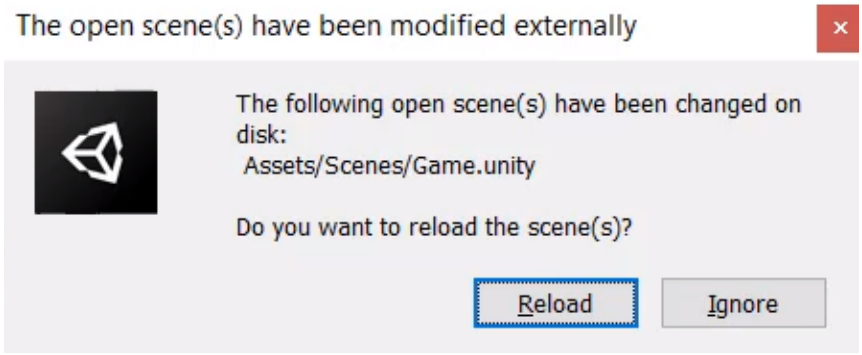
Creating A New Scene

The following instructions have been updated, and differs from the video:

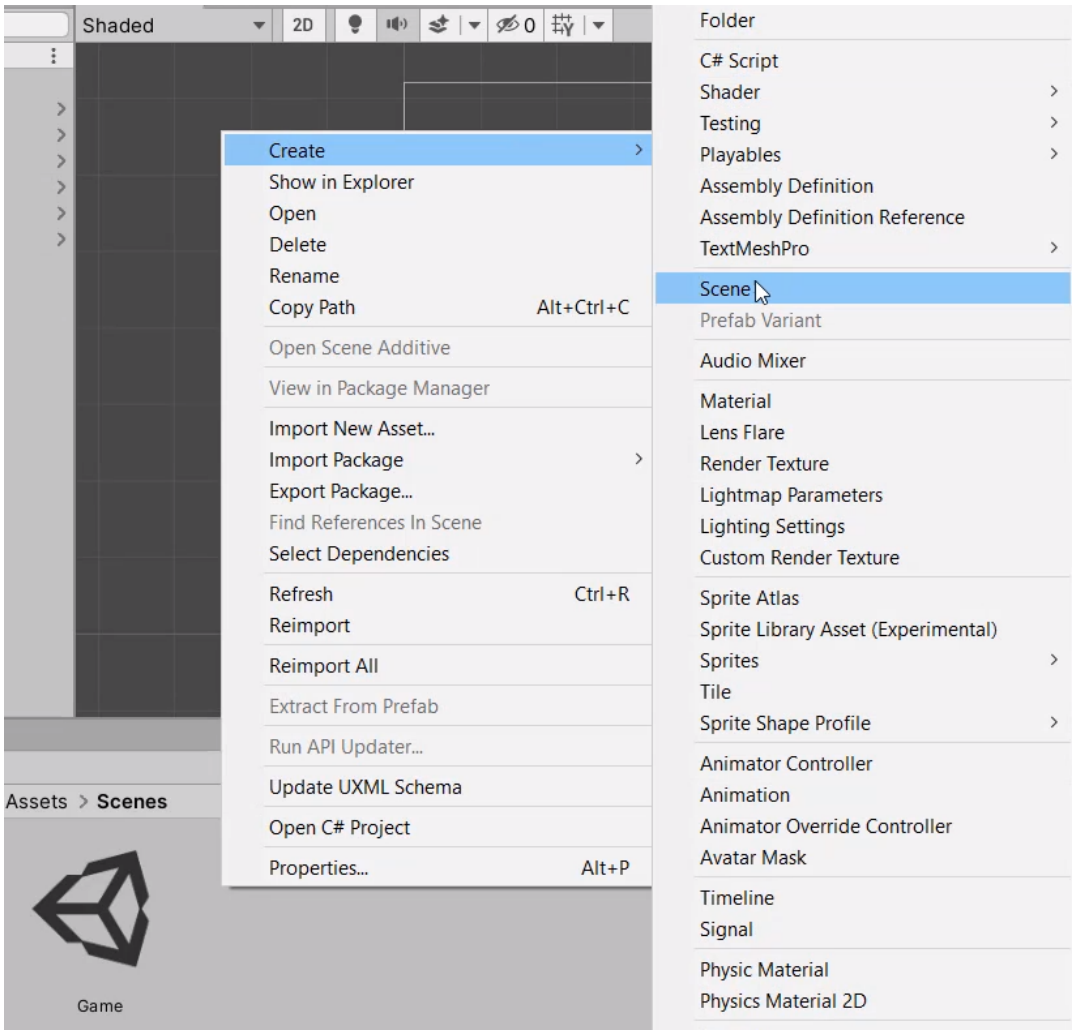
Save the scene at this time (Ctrl + S (Windows) or Cmd + S (macOS)) because we will be renaming it next. *Renaming a scene without saving or closing it first will result in changes being lost.*

Let's go over to the Scenes folder (**Assets > Scenes**), select the current scene (**SampleScene**), and **rename** it to '**Game**'.

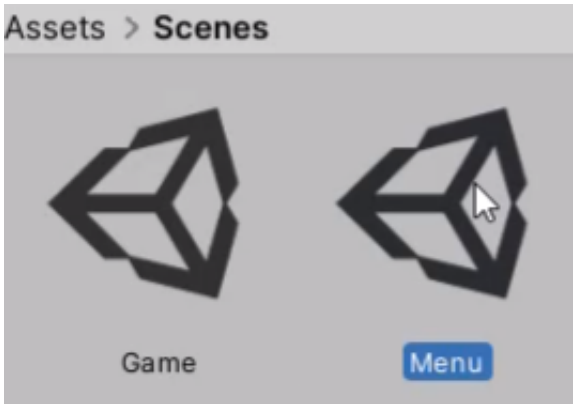
Click on '**Reload**' if the following prompt pops up.



Now we're going to create a new scene by right-clicking on **Project > Create > Scene**.

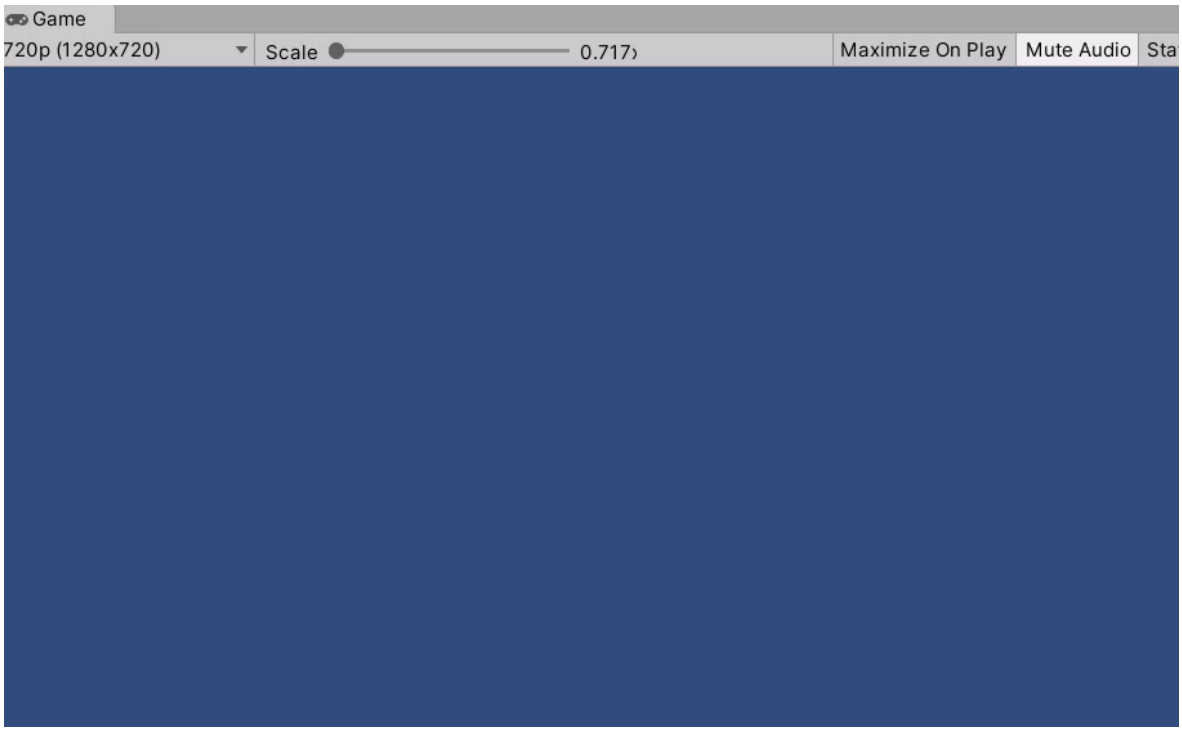


We're going to **rename** this scene to be "Menu", and **double-click** to open it up.

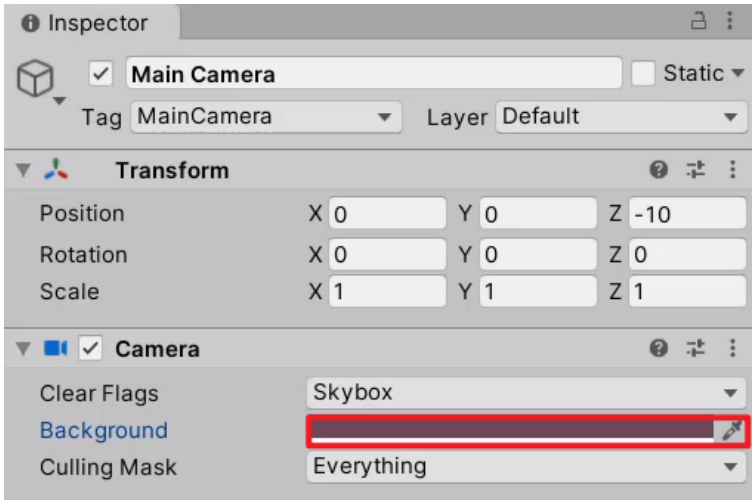


Setting Up Menu UI

Right now our scene's background is set to the default color.

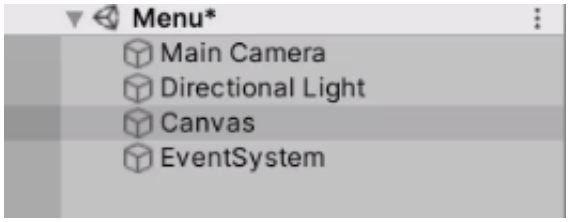


To change the background color, we can select the **Main Camera** and change the **Background** property.

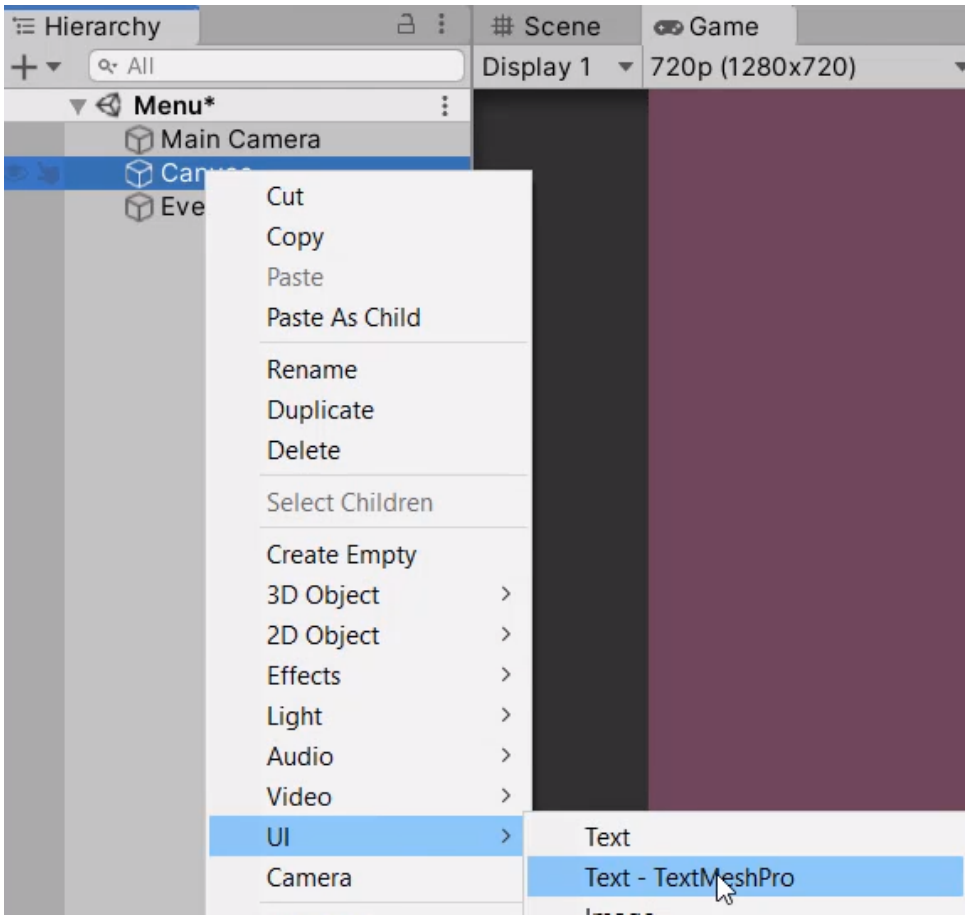


Creating A Title

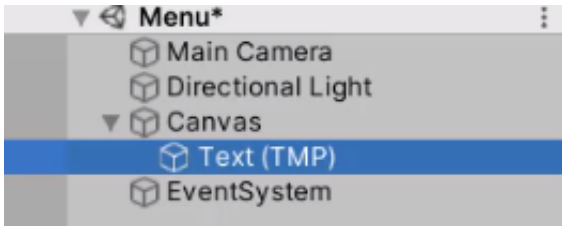
Let's create a new **Canvas** game object by going to **Create > Canvas**.

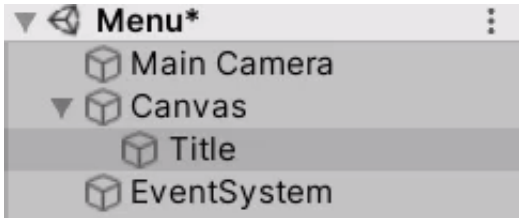


Then we can create a new **Text (TMP)** object as a **child** to our Canvas (**Right-click** > **UI > Text – TextMeshPro**).

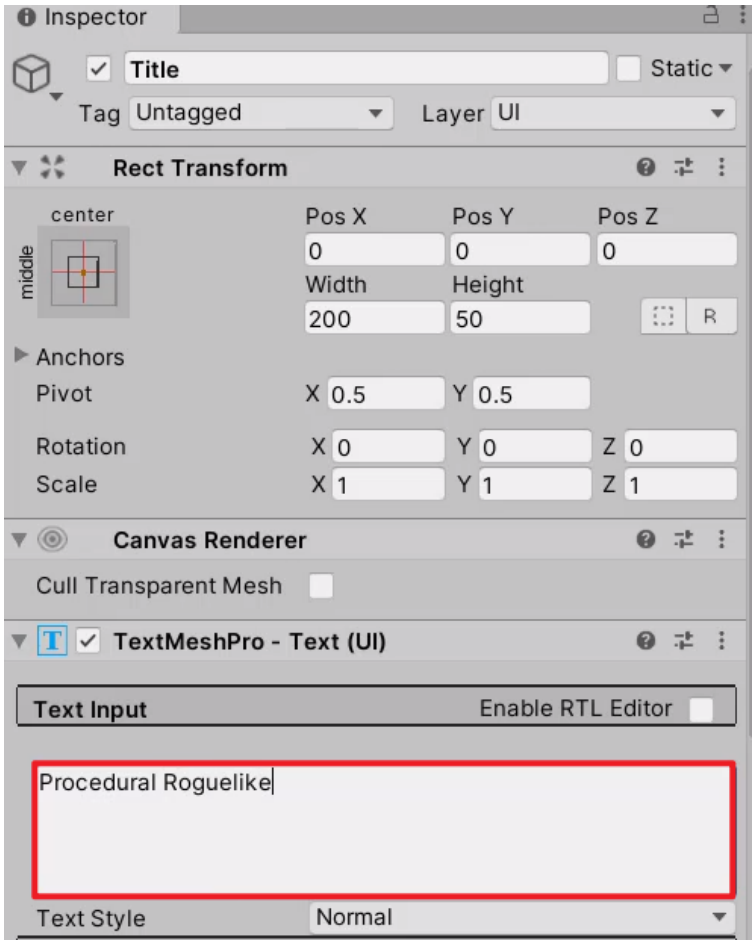


We'll **rename** this to "**Title**",

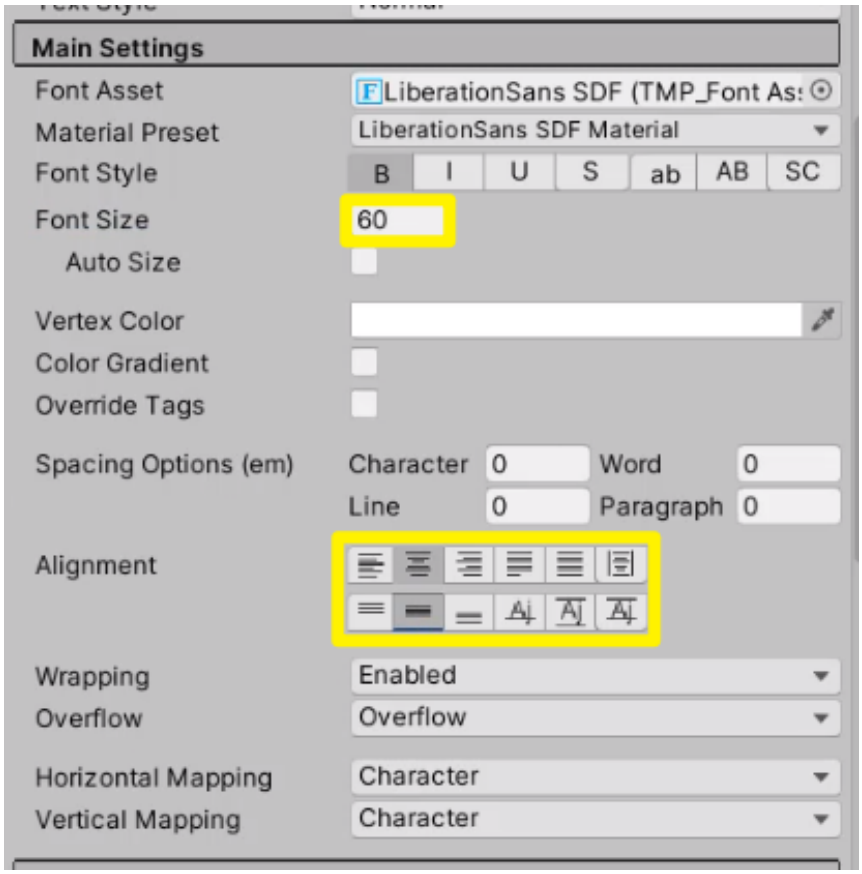




...and change the **text input** to "Procedural Roguelike".



Make sure to **align** it to the center, and increase the **font size**.

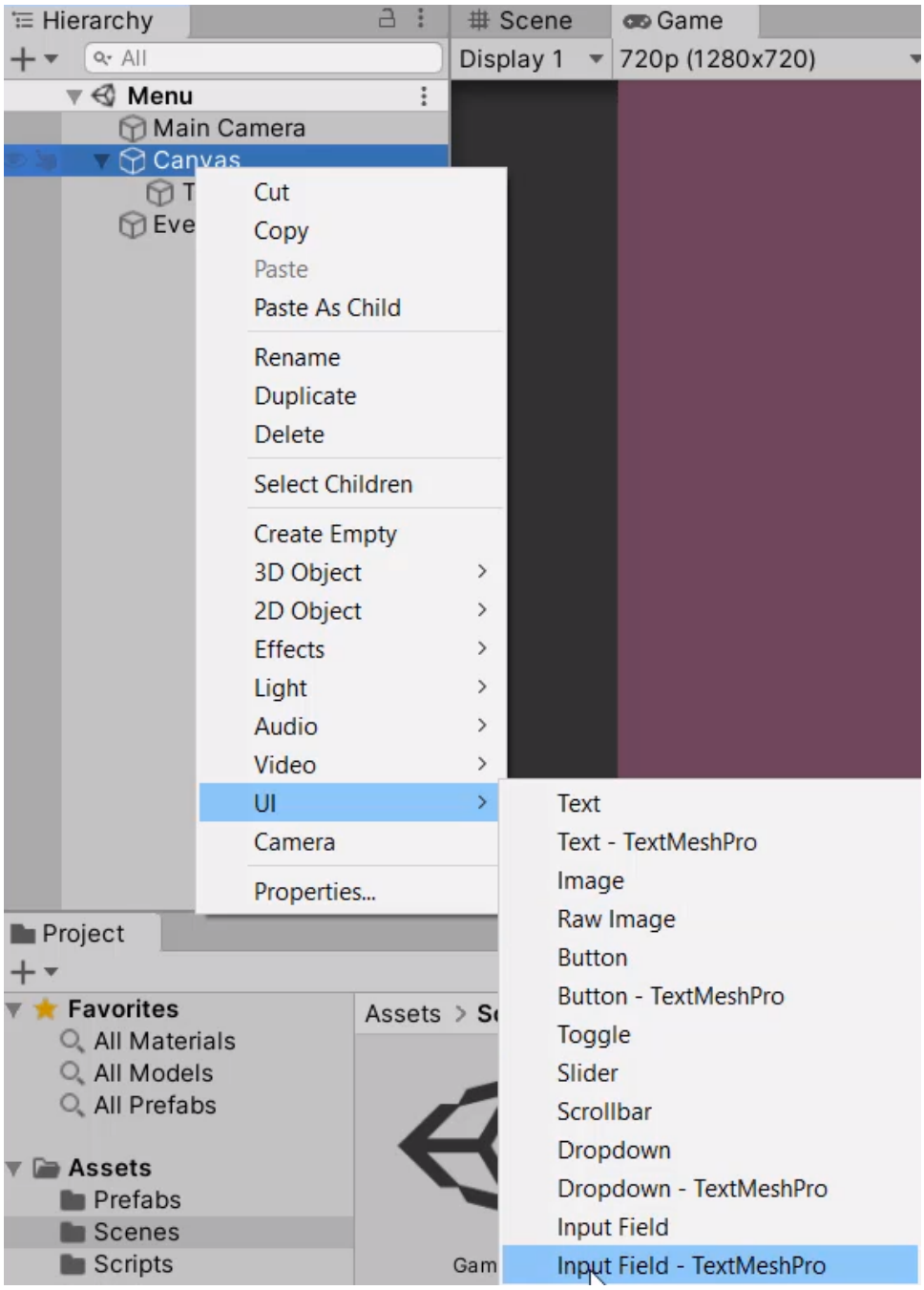


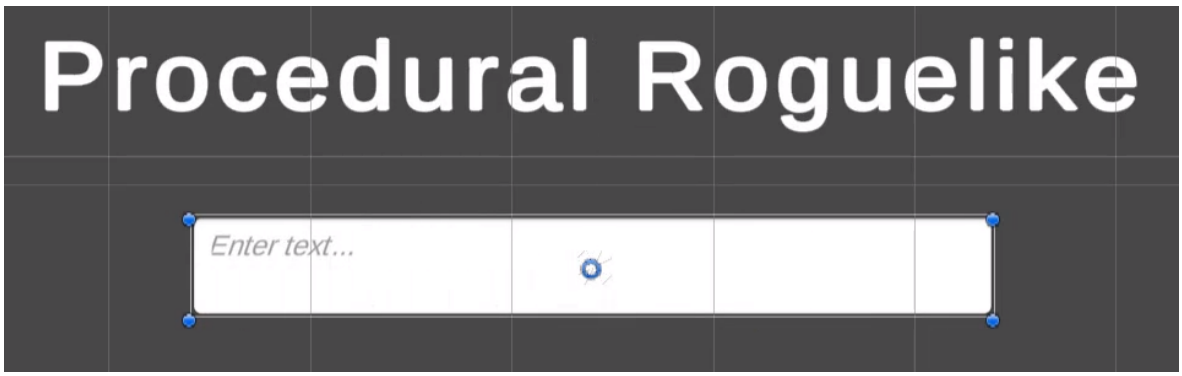
Now we have a simple **Title** for the main menu.

Creating An Input Field

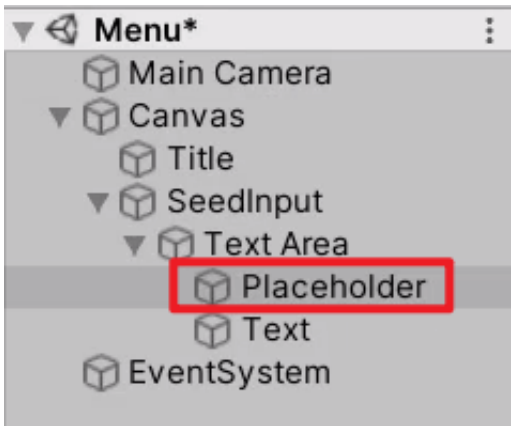
Now we need to create an **Input Field** called "**SeedInput**", which is a text field to enter our **game seed**.

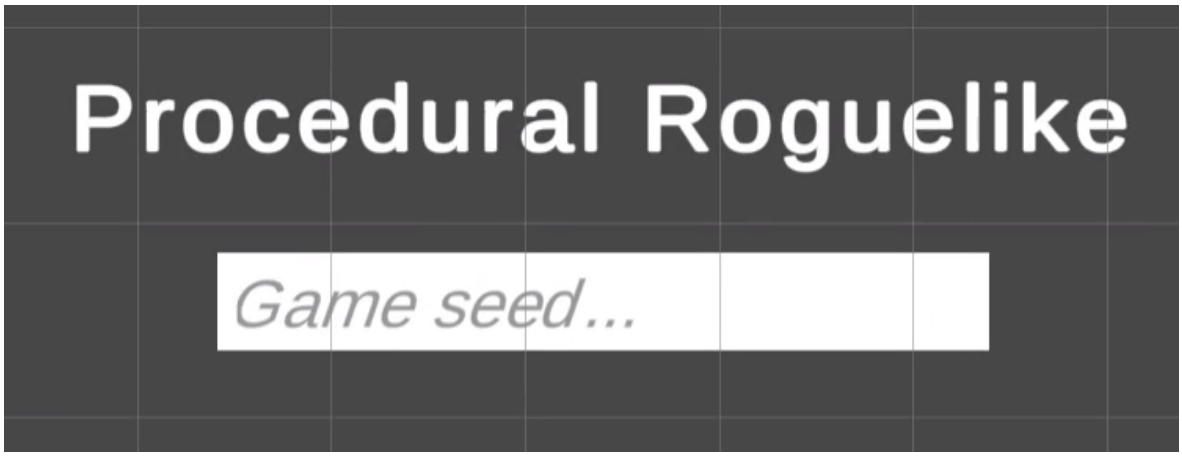
(**Right-click > UI > Input Field – TextMeshPro**)





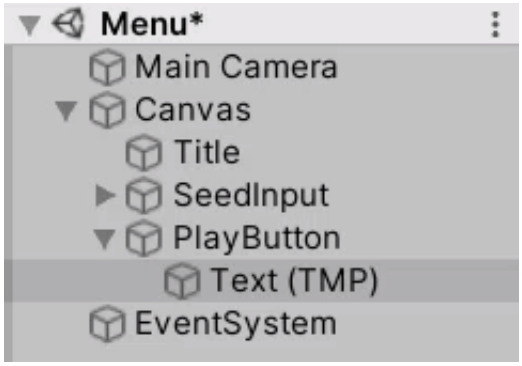
Then we can select the **Placeholder** object to change the default text from '**Enter text...**' to '**Game seed...**'





Creating A Play Button

Finally, we're going to create a new **button** object to our Canvas, and call this "**PlayButton**".

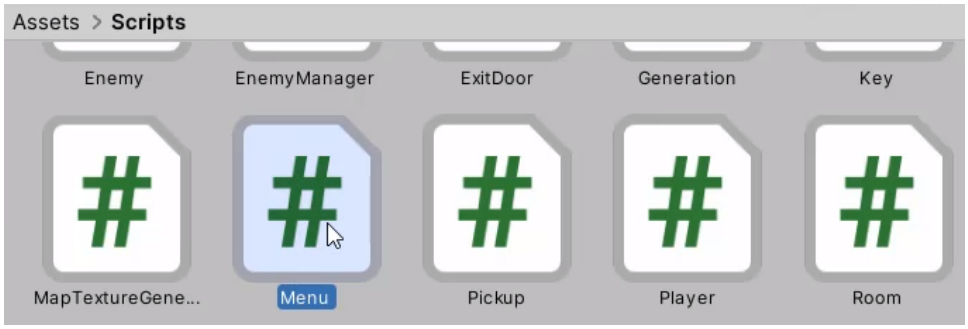


Let's place the button under the input field, and edit the text to '**Play**':

Setting up Menu Script

You will notice that the button responds to clicks, but nothing happens. In order to make it load up a different scene, we need to link the button up with a script.

So let's create a new C# script called "**Menu**" inside **Assets > Scripts**.



At the top of the script, we need to import the libraries we're going to use, i.e. **TMPro** and **SceneManagement**.

```
using TMPro;
using UnityEngine.SceneManagement;
```

Then we need to create a new function called "**OnUpdateSeed**" to save the **Seed** input (string) into **PlayerPrefs** as an integer.

PlayerPrefs is an easy way to store and access important data between game sessions.

For more information, refer to the documentation:
https://docs.unity3d.com/2020.1/Documentation/ScriptReference/PlayerPrefs.html

```
public TMP_InputField seedInput;
```



```
public void OnUpdateSeed ()
{
    // save the seed value into PlayerPrefs
    PlayerPrefs.SetInt("Seed", int.Parse(seedInput.text));
}
```

Now even if you close and reopen the game, you can still load up the same level using the "**Seed**" value saved inside PlayerPrefs.

Upon opening up the main menu scene, we can load this value and update the input field's text so that the player knows which level they're currently in.

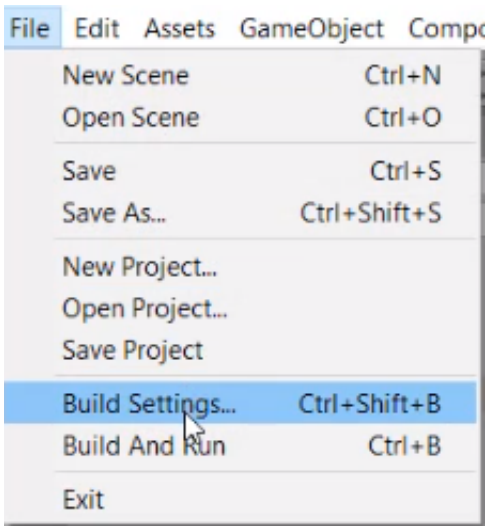
```
public TMP_InputField seedInput;

public void OnUpdateSeed ()
{
    PlayerPrefs.SetInt("Seed", int.Parse(seedInput.text));
}

void Start ()
{
    // load the seed value and display it in the main menu
    seedInput.text = PlayerPrefs.GetInt("Seed").ToString();
}
```

Finally, when we click on the **Play** button, we want to load up the game scene.

```
public TMP_InputField seedInput;

void Start ()
{
    seedInput.text = PlayerPrefs.GetInt("Seed").ToString();
}

public void OnUpdateSeed ()
{
    PlayerPrefs.SetInt("Seed", int.Parse(seedInput.text));
}

public void OnPlayButton ()
{
    SceneManager.LoadScene("Game");
}
```

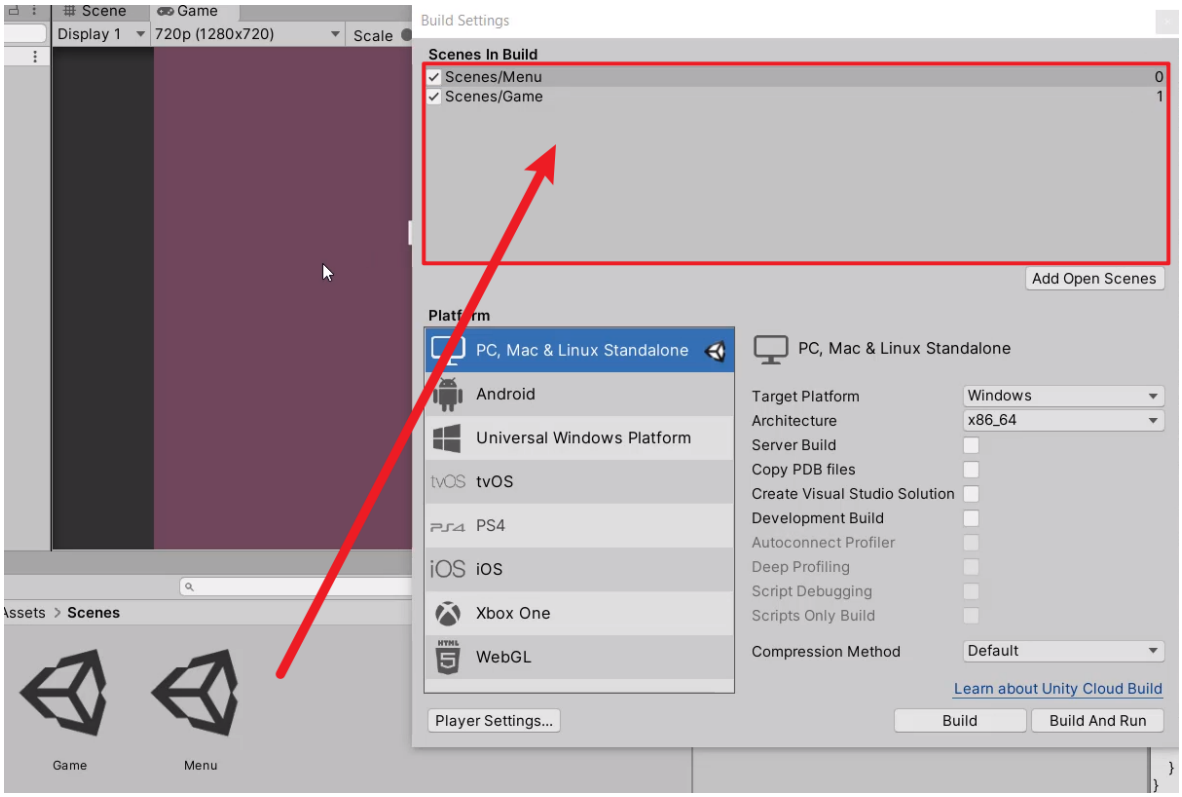
Build Settings

Now, in order for the menu script to load our scenes, we need to first add the scenes to the **Build Settings (File > Build Settings...)**.



We need to include all the scenes that we want our game to have within '**Scenes In Build**':

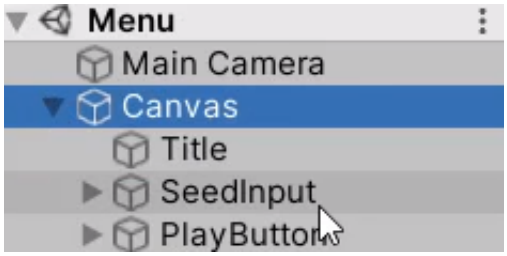
To add scenes to the build list, simply **drag and drop** the scene files into the highlighted box.

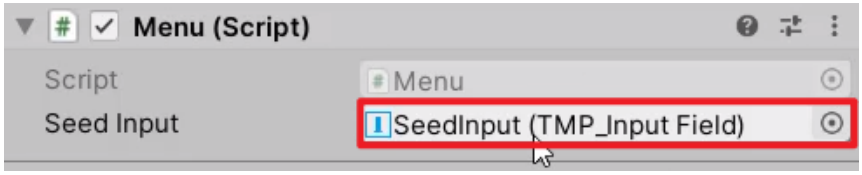


Note that the **build index** (e.g. Menu: 0) is displayed on the right side. We can now switch between these scenes by referring to their index number. By default, the game will launch into the first scene that is marked as index 0.

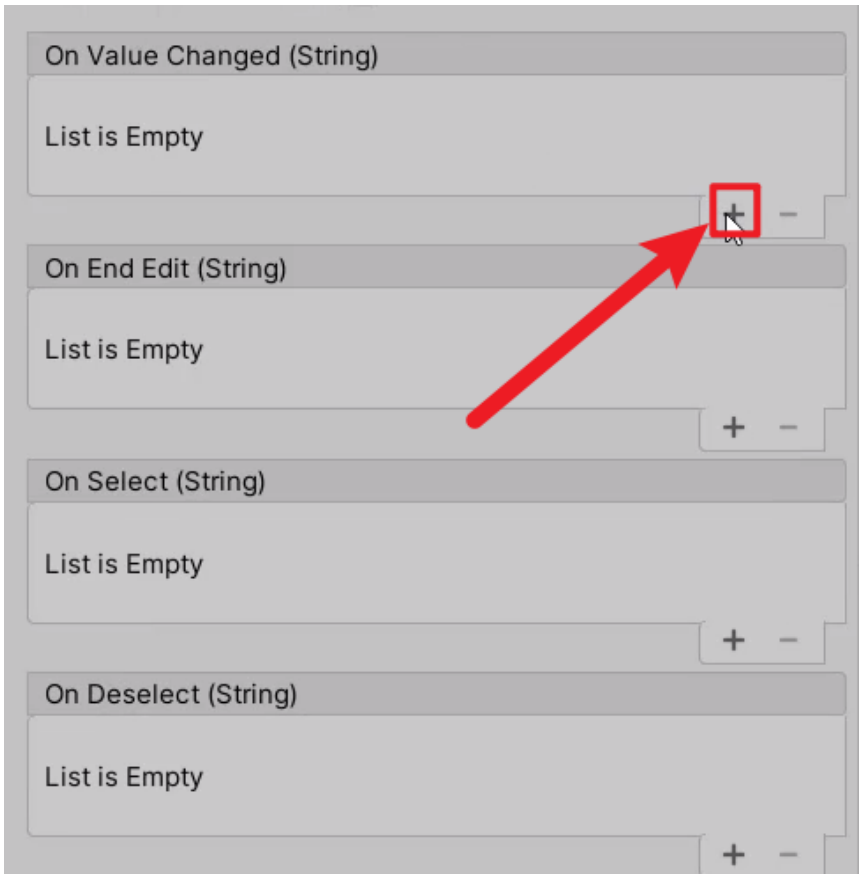
Adding Script Events To Buttons

Make sure to **attach** this script to our **Canvas** and drag in the **TMP InputField** object into the **Seed Input** field.

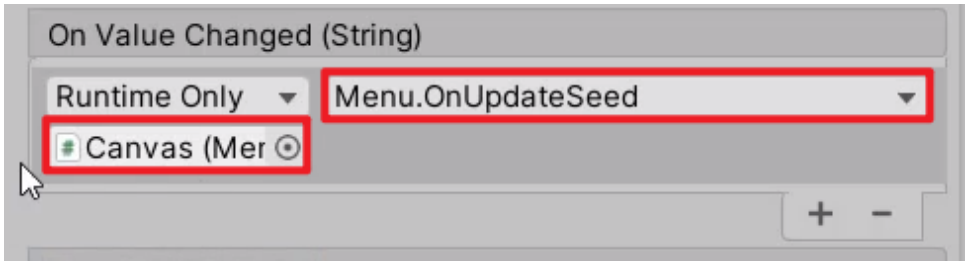




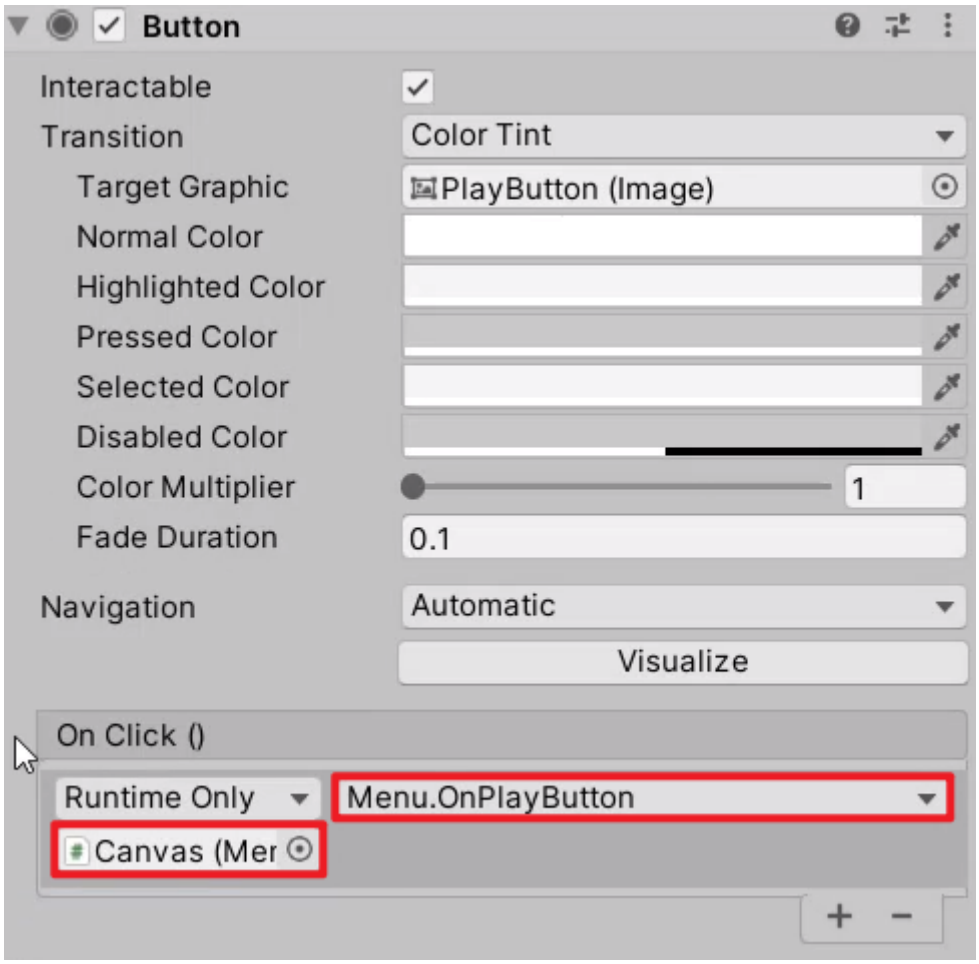
We can then select the **SeedInput** and add an **On Value Changed** event by clicking on the **+** icon.



Inside here, we're going to drag in the **Canvas** object and select the **OnUpdateSeed** function as the function to be called whenever the seed value changes.



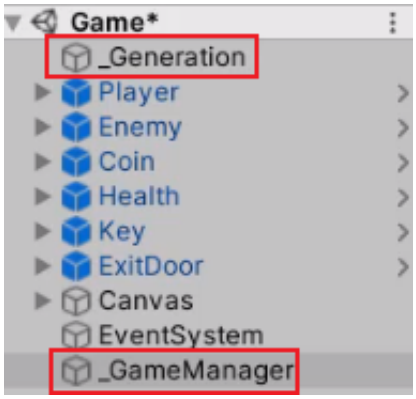
Similarly, we need to link the **OnPlayButton** function with the **Play** button.



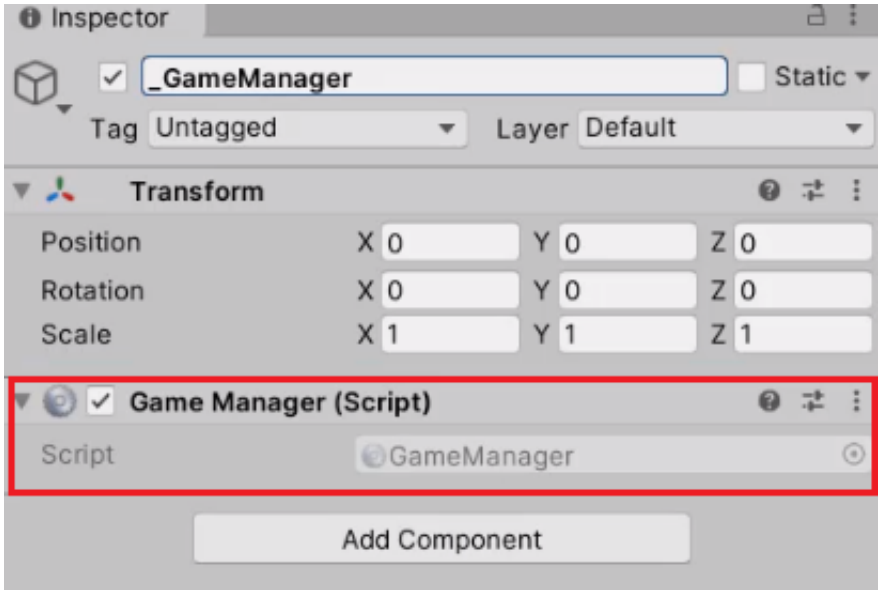
In this lesson, we're going to be setting up a game loop.

Creating GameManager

First of all, let's create an **empty** GameObject called "**_GameManager**" (Rename the existing "_GameManager" to be "**_Generation**" instead):



Then let's add a new **C# script** called "**GameManager**" to the "_GameManager" object.



Scripting GameManager

Inside the script, we're going to import the **SceneManagement** library first.

```
using UnityEngine.SceneManagement;
```

Then we can create a few variables based on the following questions:

- What **level** are we currently in? (**int**)
- What **seed** value does the level have? (**int**)
- How much **health** were we on, in the previous room? (**int**)
- How many **coins** did we have, in the previous room? (**int**)

```
public int level;
public int baseSeed;

private int prevRoomPlayerHealth;
private int prevRoomPlayerCoins;

// reference to the player object
private Player player;
```

DontDestroyOnLoad Singleton

We're also going to set up a **Singleton** instance for this script, and use **DontDestroyOnLoad** to keep track of all the game information between game sessions.

That means when we load up the game scene again, the **GameManager** is going to be carried over with us. But the problem there is, if we were to reload the game scene and bring our game manager over with us, we would have a duplicate of the same game manager.

So we need to check if there is already an existing instance of this script first, and then **destroy** it if this is a **clone**.

```
public static GameManager instance;

void Awake ()
{
    // if there is already an (original) instance of this script existing, which is n
ot 'this' instance
    if(instance != null && instance != this)
    {
        // destroy this gameObject(clone).
        Destroy(gameObject);
        return;
    }

    // if this is not a clone, don't destroy on load.
    instance = this;
    DontDestroyOnLoad(gameObject);
}
```

Generating The Level

Inside the **Start** function, we're going to begin **generating** the level based on the **baseSeed** stored in **PlayerPrefs**. The level should have an updated **level text**, as well as the **player** object.

```
// this can replace the Start function of Generation.cs
void Start ()
{
    level = 1;
    baseSeed = PlayerPrefs.GetInt("Seed");
```



```
    Random.InitState(baseSeed);
    Generation.instance.Generate();
    UI.instance.UpdateLevelText(level);

    player = FindObjectOfType<Player>();
}
```

Then we can make a new function called **OnSceneLoaded**, and subscribe that to the **sceneLoaded** event, which will get called whenever a scene has been loaded. In other words, we're going to call the **OnSceneLoaded** function whenever we change scenes.

```
void Start ()
{
    level = 1;
    baseSeed = PlayerPrefs.GetInt("Seed");
    Random.InitState(baseSeed);
    Generation.instance.Generate();
    UI.instance.UpdateLevelText(level);

    player = FindObjectOfType<Player>();

    // subscribe to the sceneLoaded event which gets called whenever a scene has been
 loaded
    SceneManager.sceneLoaded += OnSceneLoaded;
}

void OnSceneLoaded (Scene scene, LoadSceneMode mode)
{

}
```

If we're back at the menu, we need to **destroy** this gameObject so that there is only one instance of this script running. If this is the **Game** scene, then we can increment the **level** and the **base seed** and generate a new level. Once that's done, we can transfer the existing game data from the previous level to the new level.

```
void Start ()
{
    level = 1;
    baseSeed = PlayerPrefs.GetInt("Seed");
    Random.InitState(baseSeed);
    Generation.instance.Generate();
    UI.instance.UpdateLevelText(level);

    player = FindObjectOfType<Player>();

    SceneManager.sceneLoaded += OnSceneLoaded;
}

void OnSceneLoaded (Scene scene, LoadSceneMode mode)
```



```
{
    // if this is the main menu scene, destroy this gameObject to prevent a duplicate
    if(scene.name != "Game")
    {
        Destroy(gameObject);
        return;
    }

    player = FindObjectOfType<Player>();
    level++;
    baseSeed++;

    // generate a new level with the updated baseSeed
    Generation.instance.Generate();

    // transfer the previous game data to the new level
    player.curHp = prevRoomPlayerHealth;
    player.coins = prevRoomPlayerCoins;

    UI.instance.UpdateHealth(prevRoomPlayerHealth);
    UI.instance.UpdateCoinText(prevRoomPlayerCoins);
    UI.instance.UpdateLevelText(level);
}
```

Finally, we need a function to save the **current** game data to the private variables (i.e. **prevRoomPlayerHealth**, **prevRoomPlayerCoins**). This function will reload the current scene once we hit the exit door with the key.

```
public void GoToNextLevel ()
{
    prevRoomPlayerHealth = player.curHp;
    prevRoomPlayerCoins = player.coins;

    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
}
```

Testing Out

Now, we can enter some random seed in the main menu and press **Play**.

When we go up to the exit door with the key, we should see the player data (coins, health) being maintained across the level.