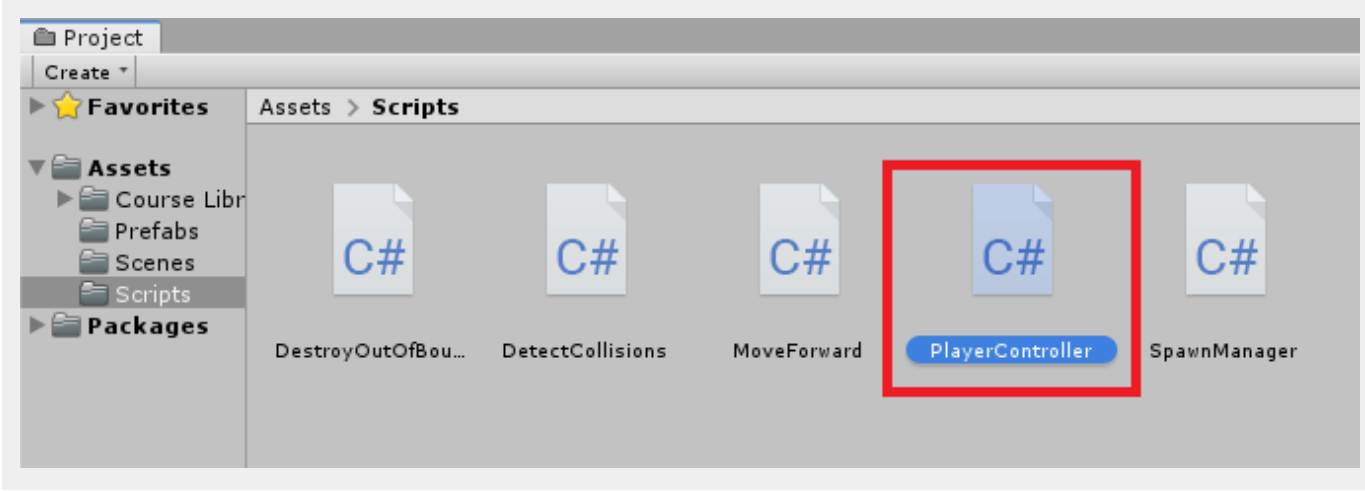
Bonus Features 2

Solution Walkthrough

Easy - Vertical Player Movement

1. In the Project window, navigate to the **Scripts** folder and double-click the **PlayerController** script to open it.



2. By the variable declarations, add the following:

```
public float horizontalInput;
public float speed = 10.0f;
public float xRange = 10.0f;

public GameObject projectilePrefab;

public float zMin;
public float zMax;
public float verticalInput;
```

3. In the Update method, after transform.Translate, we will add the code to move the player forwards

```
 horizontalInput = Input.GetAxis("Horizontal");
transform.Translate(Vector3.right * horizontalInput * Time.deltaTime * speed);

verticalInput = Input.GetAxis("Vertical");
transform.Translate(Vector3.forward * verticalInput * Time.deltaTime * speed);
```

4. Now that the player can move back and forth, we will need to limit the movement so that they don't go out of the screen. We can do that by adding the following before the horizontalInput variables value is set.

```
if(transform.position.z < zMin)
{
    transform.position = new Vector3(transform.position.x, transform.position.y,
zMin);
}

if(transform.position.z > zMax)
{
    transform.position = new Vector3(transform.position.x, transform.position.y,
zMax);
}
```

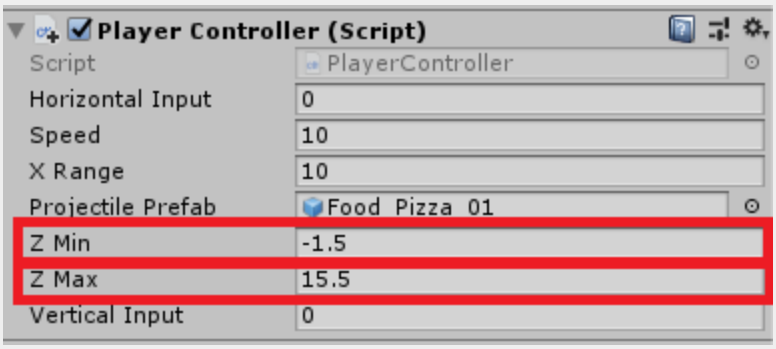
5. The final Update method for the PlayerController script should look like this:

```
void Update()
{
    if (transform.position.x < -xRange)
    {
        transform.position = new Vector3(-xRange, transform.position.y,
transform.position.z);
    }

    if (transform.position.x > xRange)
    {
        transform.position = new Vector3(xRange, transform.position.y,
transform.position.z);
    }

    if(transform.position.z < zMin)
    {
        transform.position = new Vector3(transform.position.x,
transform.position.y, zMin);
    }

    if(transform.position.z > zMax)
    {
        transform.position = new Vector3(transform.position.x,
transform.position.y, zMax);
    }

    horizontalInput = Input.GetAxis("Horizontal");
    transform.Translate(Vector3.right * horizontalInput * Time.deltaTime * speed);
```

```
    verticalInput = Input.GetAxis("Vertical");
    transform.Translate(Vector3.forward * verticalInput * Time.deltaTime * speed);

    if (Input.GetKeyDown(KeyCode.Space))
    {
        Instantiate(projectilePrefab, transform.position,
projectilePrefab.transform.rotation);
    }
}
```

6. Save the script and head back into Unity. In the Hierarchy, select the *Player* GameObject. You will notice that we have some new fields available on the **Player Controller** component.

7. Change the values of the Z Min and Z Max properties to allow for the player to not leave the cameras view. We chose the following values:



8. Save the screen and press play. The player should now be able to move backwards and forwards, while not being able to exit the camera's view.

Medium - Aggressive Animals

1. Navigate to the Scripts folder in the Project view and select the SpawnManager script. We will edit it to allow animals to spawn on the sides. By the variable declarations, add the following new variables:

```
public float sideSpawnMinZ;
public float sideSpawnMaxZ;
public float sideSpawnX;
```

2. After the SpawnRandomAnimal method, create two new methods. We can use the contents of the SpawnRandomAnimal method as a start for our methods. We would need to adjust the spawnPos to better suit which side of the screen the animals will spawn. For the left side, we will need the negative of the x position.

```
void SpawnLeftAnimal()
{
    int animalIndex = Random.Range(0, animalPrefabs.Length);
    Vector3 spawnPos = new Vector3(-sideSpawnX, 0, Random.Range(sideSpawnMinZ,
sideSpawnMaxZ));
    Instantiate(animalPrefabs[animalIndex], spawnPos,
animalPrefabs[animalIndex].transform.rotation);
}
```

For the right side we would need the positive of the x position

```
void SpawnRightAnimal()
{
    int animalIndex = Random.Range(0, animalPrefabs.Length);
    Vector3 spawnPos = new Vector3(sideSpawnX, 0, Random.Range(sideSpawnMinZ,
sideSpawnMaxZ));
    Instantiate(animalPrefabs[animalIndex], spawnPos,
animalPrefabs[animalIndex].transform.rotation);
}
```

3. At the moment, if we were to test this we would have an issue with the rotation of the animals. Thats because we are currently using the rotation that is on the prefab. We can fix this by updating the methods we just created to the following:

```
void SpawnLeftAnimal()
{
    int animalIndex = Random.Range(0, animalPrefabs.Length);
    Vector3 spawnPos = new Vector3(-sideSpawnX, 0, Random.Range(sideSpawnMinZ,
```

```
sideSpawnMaxZ));
    Vector3 rotation = new Vector3(0, 90, 0);
    Instantiate(animalPrefabs[animalIndex], spawnPos, Quaternion.Euler(rotation));
}

void SpawnRightAnimal()
{
    int animalIndex = Random.Range(0, animalPrefabs.Length);
    Vector3 spawnPos = new Vector3(sideSpawnX, 0, Random.Range(sideSpawnMinZ,
sideSpawnMaxZ));
    Vector3 rotation = new Vector3(0, -90, 0);
    Instantiate(animalPrefabs[animalIndex], spawnPos, Quaternion.Euler(rotation));
}
```

Click here to learn about **Quaternion.Euler**.
Save the script and head back to Unity.

4. The next script we will need to adjust is **DestroyOutOfBounds**. Open up the script and add a new variable by the variable declarations.

```
private float topBound = 30;
private float lowerBound = -10;
private float sideBound = 30;
```

5. Next we need to update the Update method to look like this:

```
void Update()
{
    // If an object goes past the players view in the game, remove that object
    if (transform.position.z > topBound)
    {
        Destroy(gameObject);
    } else if (transform.position.z < lowerBound)
    {
        Debug.Log("Game Over!");
        Destroy(gameObject);
    }
    else if(transform.position.x > sideBound)
    {
        Debug.Log("Game Over!");
        Destroy(gameObject);
    }
    else if(transform.position.x < -sideBound)
```

```
    {
        Debug.Log("Game Over!");
        Destroy(gameObject);
    }
}
```

The code we added will make sure the animals are destroyed when they go out of view on the left and right side of the screen. Save the script and head back to Unity.

6 Open up the **DetectCollision** script and update the **OnTriggerEnter** method to look like this:

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        Debug.Log("Game Over");
        Destroy(gameObject);
    }
    else
    {
        Destroy(gameObject);
        Destroy(other.gameObject);
    }
}
```

This will make sure that when the animal collides with the player, Game Over will be printed to the console.

7. The final bit of scripting we will need to do, is to adjust the **PlayerController** script. The reason behind this is, we need to add a collider to the player to check for a collision between them and an Animal. The issue with adding a collider is that it will make the food prefab collide with the player when they spawn it, resulting in both being destroyed. We can fix this by having a separate spawn location as a transform we can adjust in the editor. Below the variable declarations add the following variable:

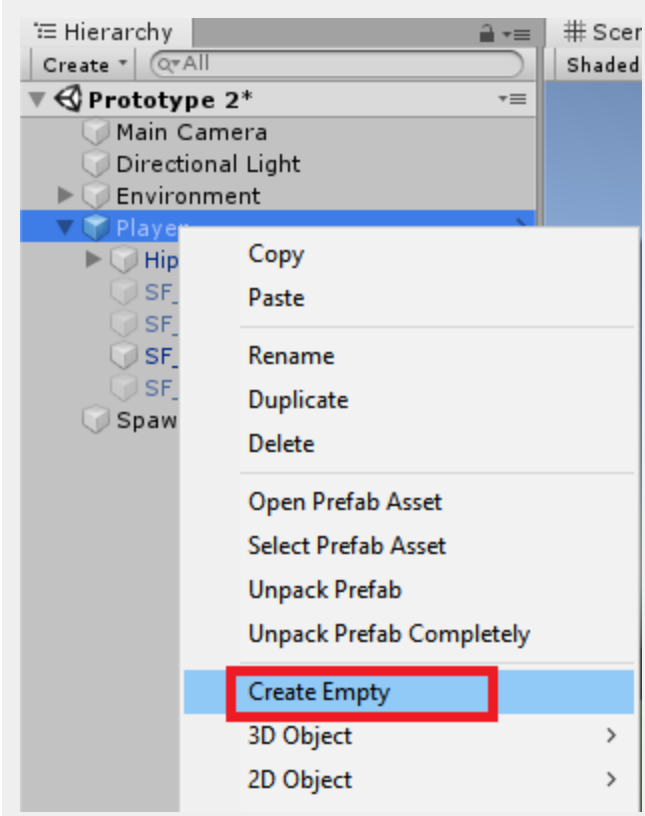
```
public Transform projectileSpawnPoint;
```

8. In the **Update** method, we need to update the way the food is spawned. The updated code looks like this:

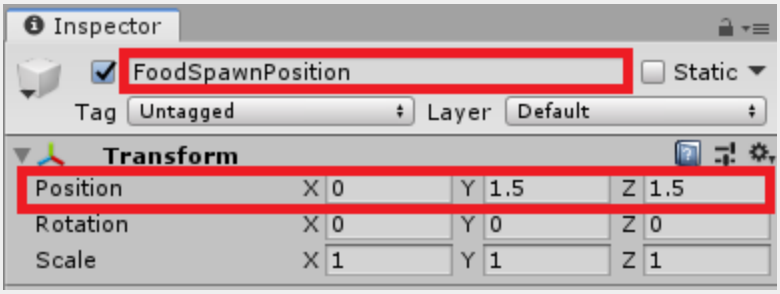
```
Instantiate(projectilePrefab, projectileSpawnPoint.position,
projectilePrefab.transform.rotation);
```

Save the script and return to Unity.

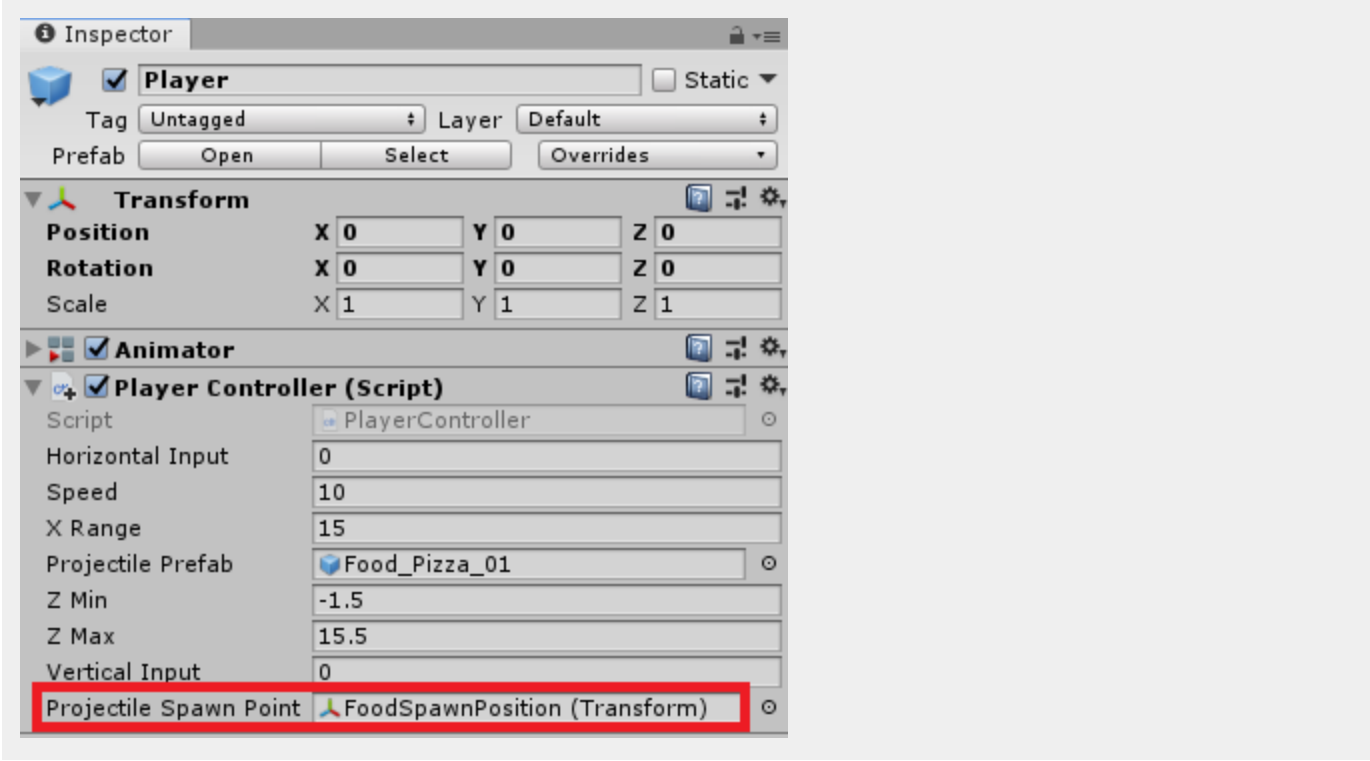
9. Next we need to adjust the Player GameObject. In the Hierarchy, right-click on the Player and select **Create Empty**.



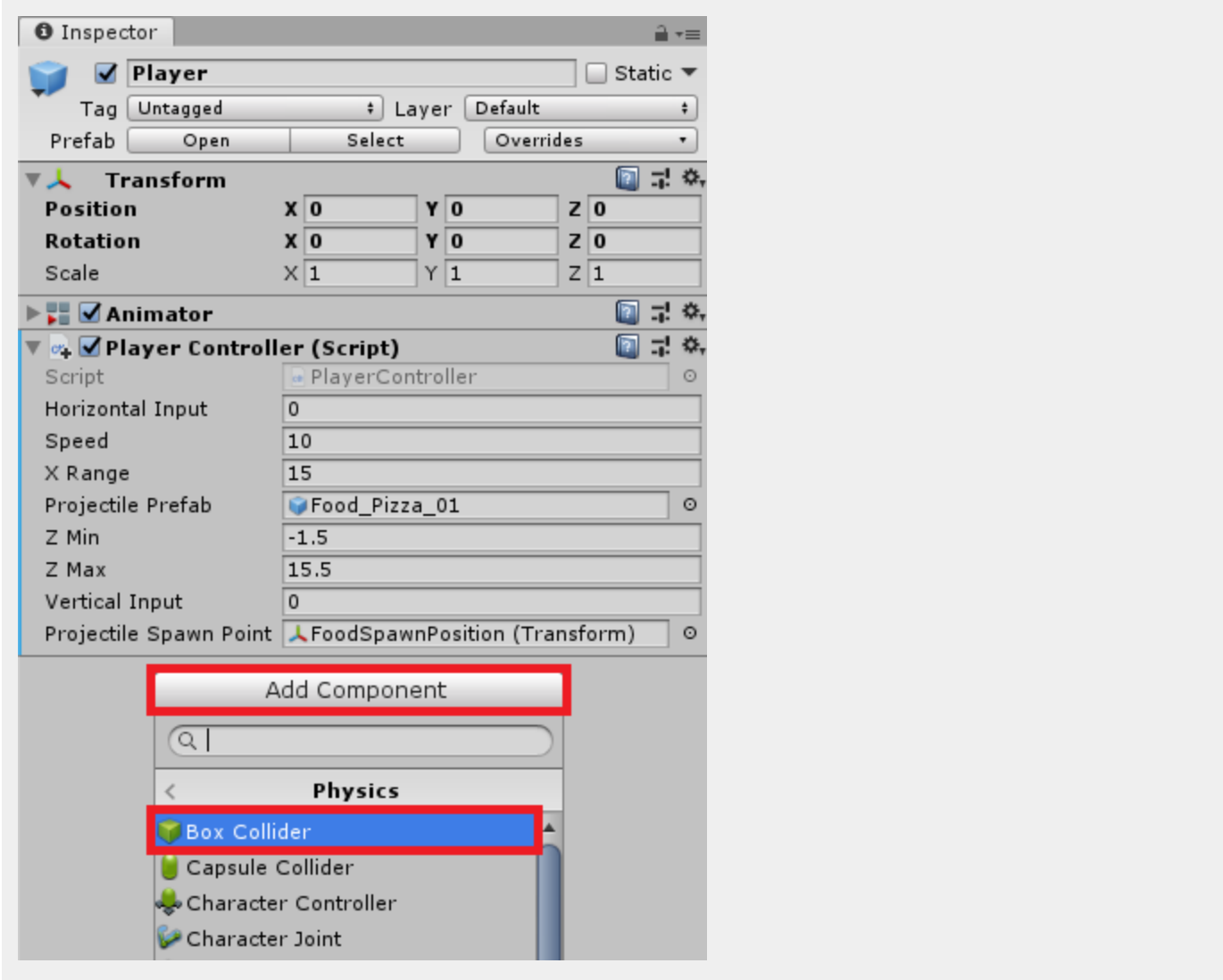
10. Rename the new GameObject to *FoodSpawnPosition* and adjust the position to be slightly in front of the Player.

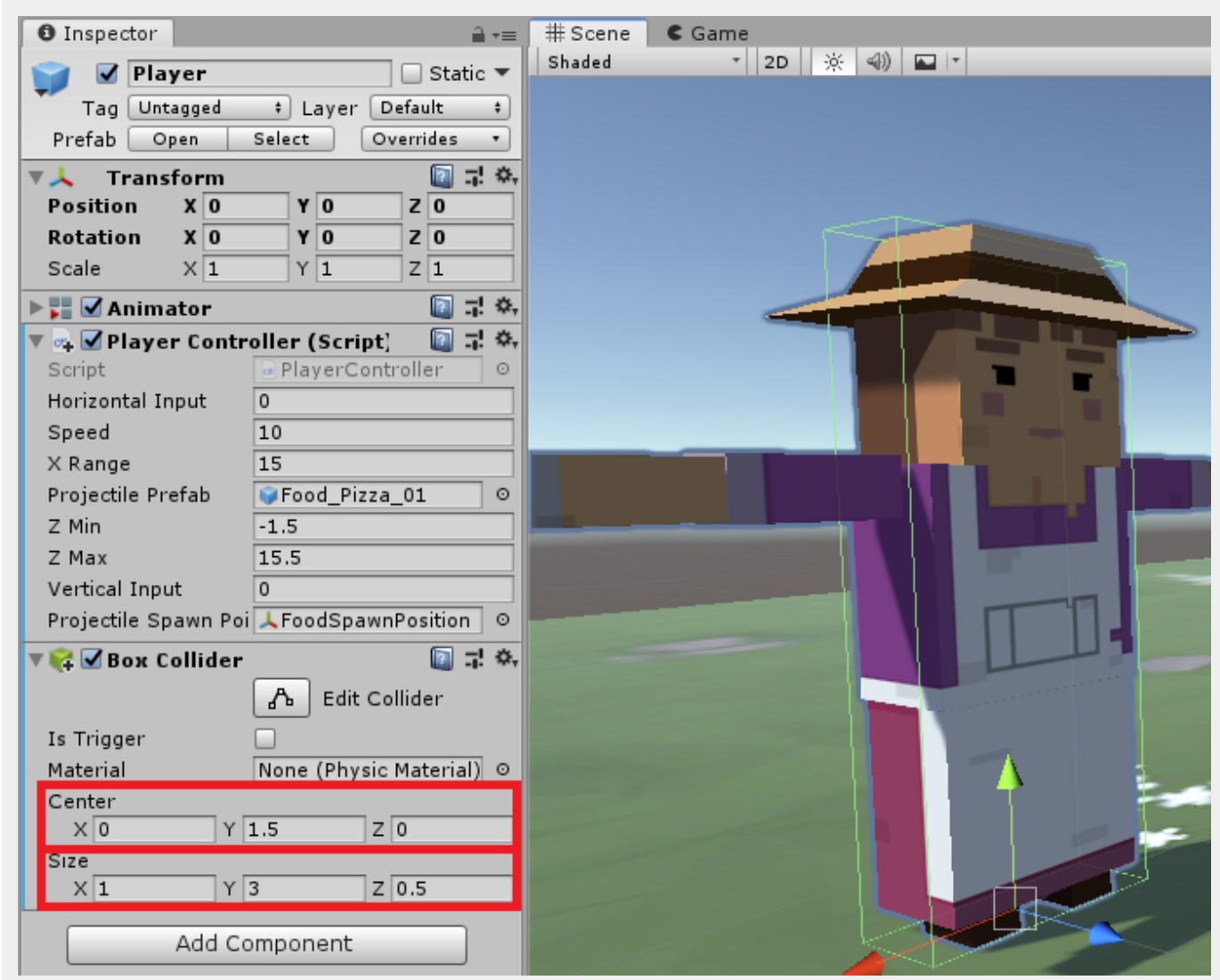


11. Back on the Player GameObject, we now need to assign the *FoodSpawnPosition* to the **Projectile Spawn Point** on the **Player Controller** component.

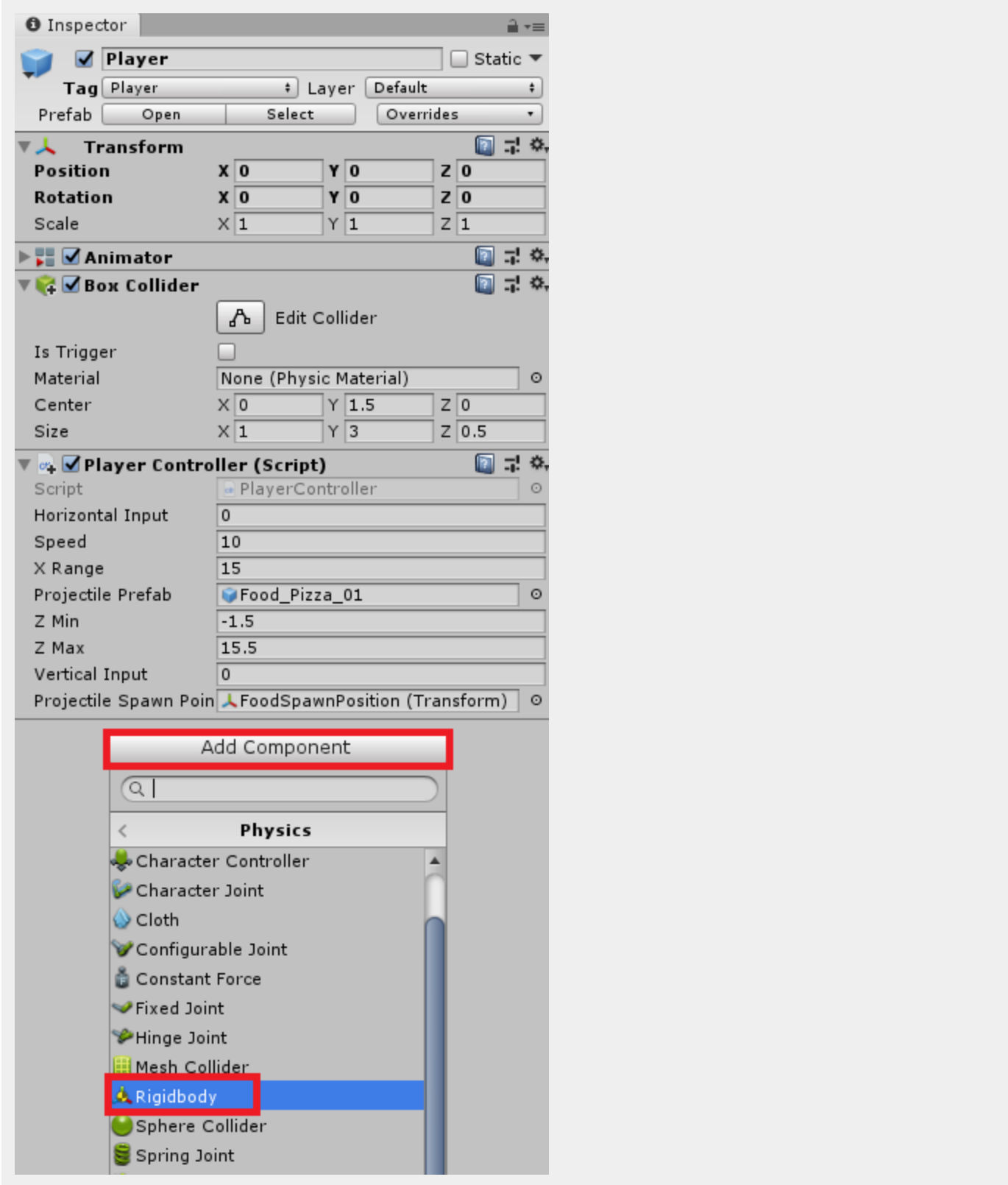


12. We also need to add a Collider to our Player. In the Inspector, select **Add Component > Physics > Box Collider**. Adjust the collider size and center to better suit the player.

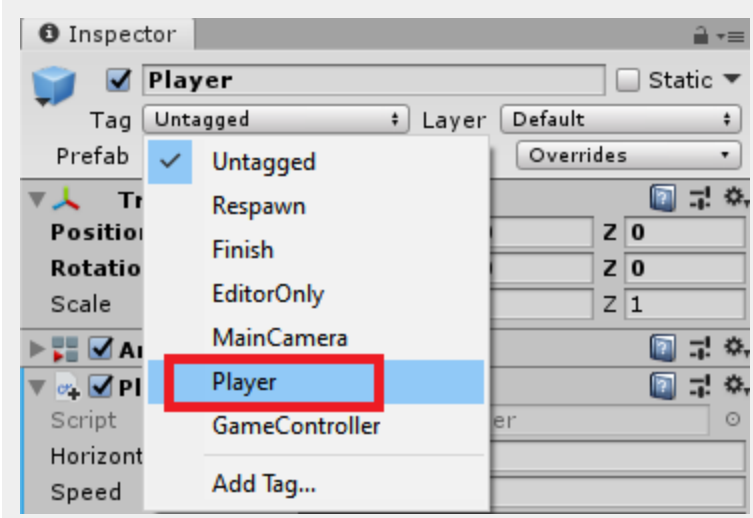

Inspector
Player
Static
Tag
Untagged
Layer
Default
Prefab
Open
Select
Overrides
Transform
Position
X 0
Y 0
Z 0
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Animator
Player Controller (Script)
Script
PlayerController
Horizontal Input
0
Speed
10
X Range
15
Projectile Prefab
Food_Pizza_01
Z Min
-1.5
Z Max
15.5
Vertical Input
0
Projectile Spawn Point
FoodSpawnPosition (Transform)
Add Component
Physics
Box Collider
Capsule Collider
Character Controller
Character Joint



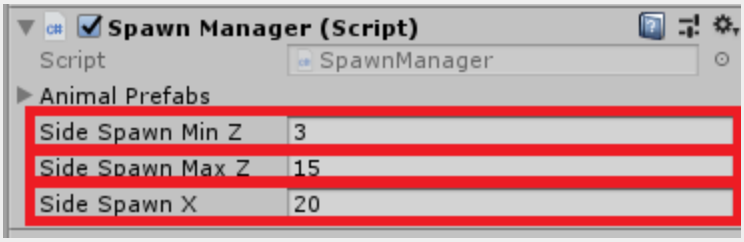
13. Next we need to add a **Rigidbody** to the Player GameObject. This is because if neither the Player nor the Animal has a Rigidbody, the collision events will not be called. To do this, click **Add Component** > **Physics** > **Rigidbody**



13. The last thing we need to do on the Player is to change the Tag to "*Player*"



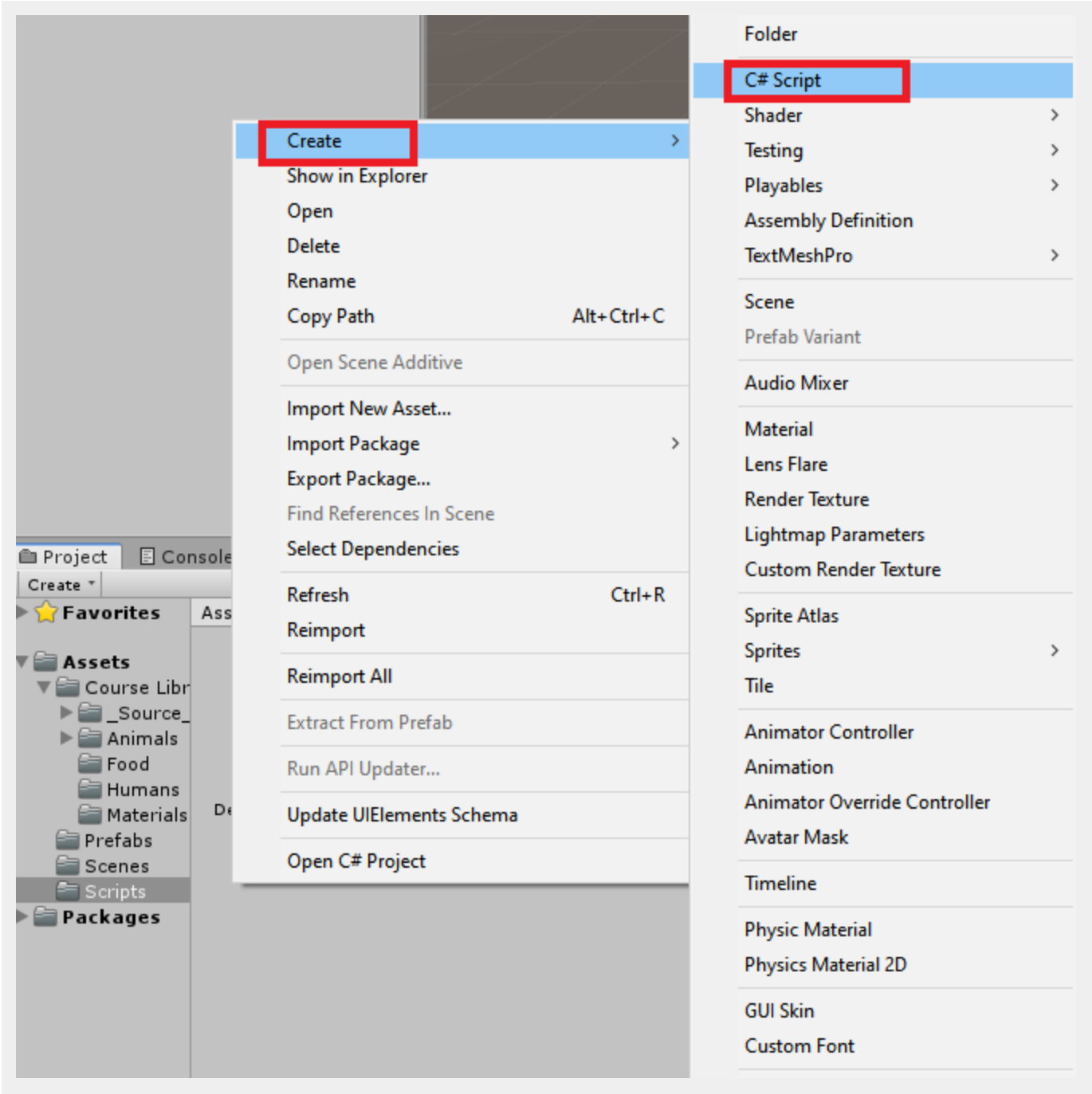
14. Next, select the *SpawnManager* in the Hierarchy. We will need to set up the new variables on the **Spawn Manager** component.



15. Save the scene and hit Play. Animals should now spawn from the edges. When they collide with you, the console outputs “Game Over”.

Hard - Game User Interface

1. The first we will do is create a GameManager script that will handle the lives and score variables. Navigate to the Scripts folder and right-click > **Create** > **C# script**. Name the script *GameManager*.



2. Open up the script. The first thing we will do is create the variables for score and lives. Below the class definition add:

```
private int score = 0;
private int lives = 3;
```

3. After the Update method, we will need to create two methods. The first will be called AddLives, and the second will be AddScore. The methods will take in a value and update the relevant variables, printing out to the console when they are updated.

```
public void AddLives(int value)
{
    lives += value;

    if (lives <= 0)
    {
        Debug.Log("Game Over");
        lives = 0;
    }
    Debug.Log("Lives = " + lives);
}

public void AddScore(int value)
{
    score += value;
    Debug.Log("Score = " + score);
}
```

4. Now we will need to call these methods in other scripts. Open up the **DestroyOutOfBounds** script. We will first need to create a reference to our GameManager script. Add a new variable for the GameManager.

```
private float topBound = 30;
private float lowerBound = -10;
private float sideBound = 30;
private GameManager gameManager;
```

5. In the Start method, we will setup the reference to the new variable.

```
void Start()
{
    gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();
}
```

6. Inside the Update method, we will need to remove all instances of 'Debug.Log("Game Over");'. and instead add in 'gameManager.AddLives(-1);'.

```
void Update()
{
    // If an object goes past the players view in the game, remove that object
    if (transform.position.z > topBound)
    {
        Destroy(gameObject);
    }
    else if (transform.position.z < lowerBound)
    {
        Debug.Log("Game Over");
        gameManager.AddLives(-1);
        Destroy(gameObject);
    }
    else if (transform.position.x > sideBound)
    {
        Debug.Log("Game Over");
        gameManager.AddLives(-1);
        Destroy(gameObject);
    }
    else if(transform.position.x < -sideBound)
    {
        Debug.Log("Game Over");
        gameManager.AddLives(-1);
        Destroy(gameObject);
    }
}
```

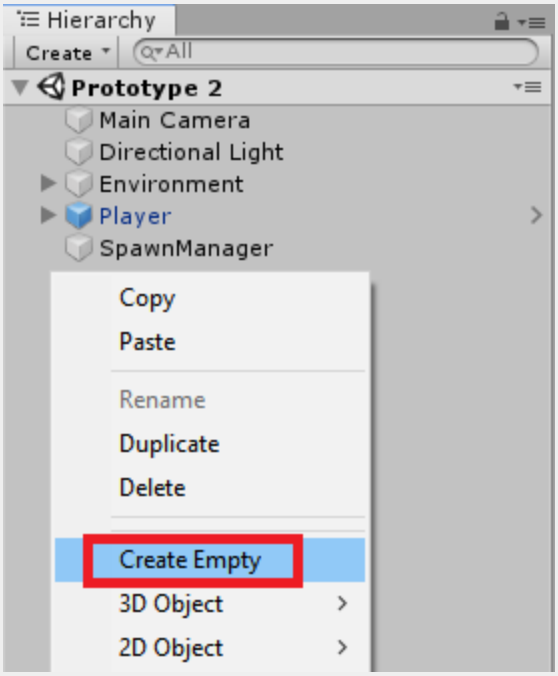
7. The next script we will need to update is **DetectCollisions.cs**. Open up the script, add in a GameManager variable and set it up like we did in **DestroyOutOfBounds.cs.**

```
private GameManager gameManager;

// Start is called before the first frame update
void Start()
{
    gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();
}
```

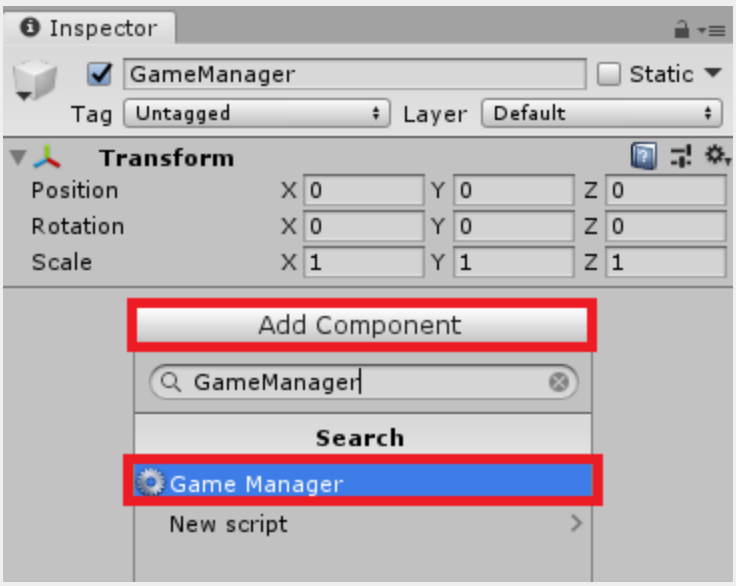
8. Next we will need to update the **OnTriggerEnter** method. The updated OnTriggerEnter method looks like this:

```
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        Debug.Log("Game Over");
        gameManager.AddLives(-1);
        Destroy(gameObject);
    }
    else if (other.CompareTag("Animal"))
    {
        gameManager.AddScore(5);
        Destroy(gameObject);
        Destroy(other.gameObject);
    }
}
```

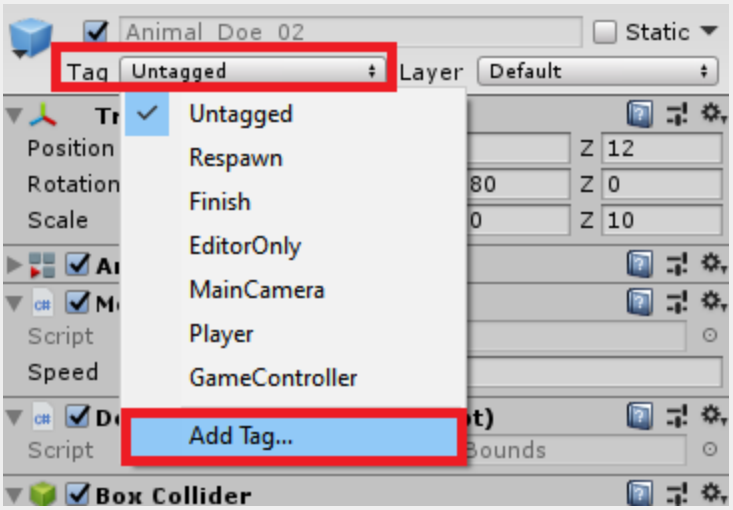
9. Save all the scripts and head back to Unity. In the Hierarchy, right-click and select **Create Empty**. Rename the empty GameObject to **GameManager**.



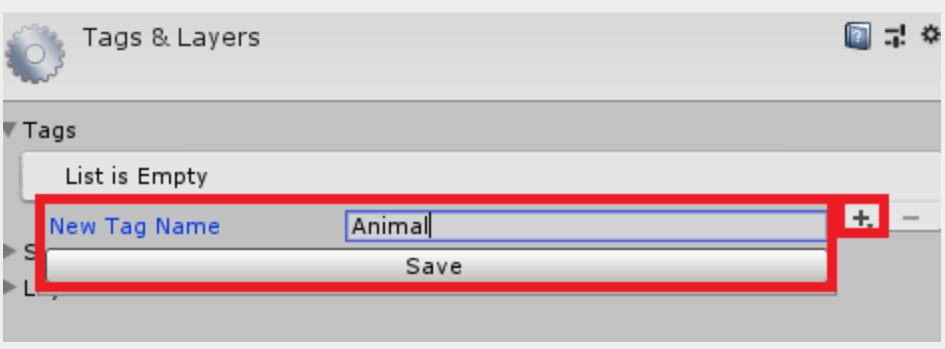
10. In the Inspector for the GameManager GameObject, Click **Add Component** and search for **GameManager**.



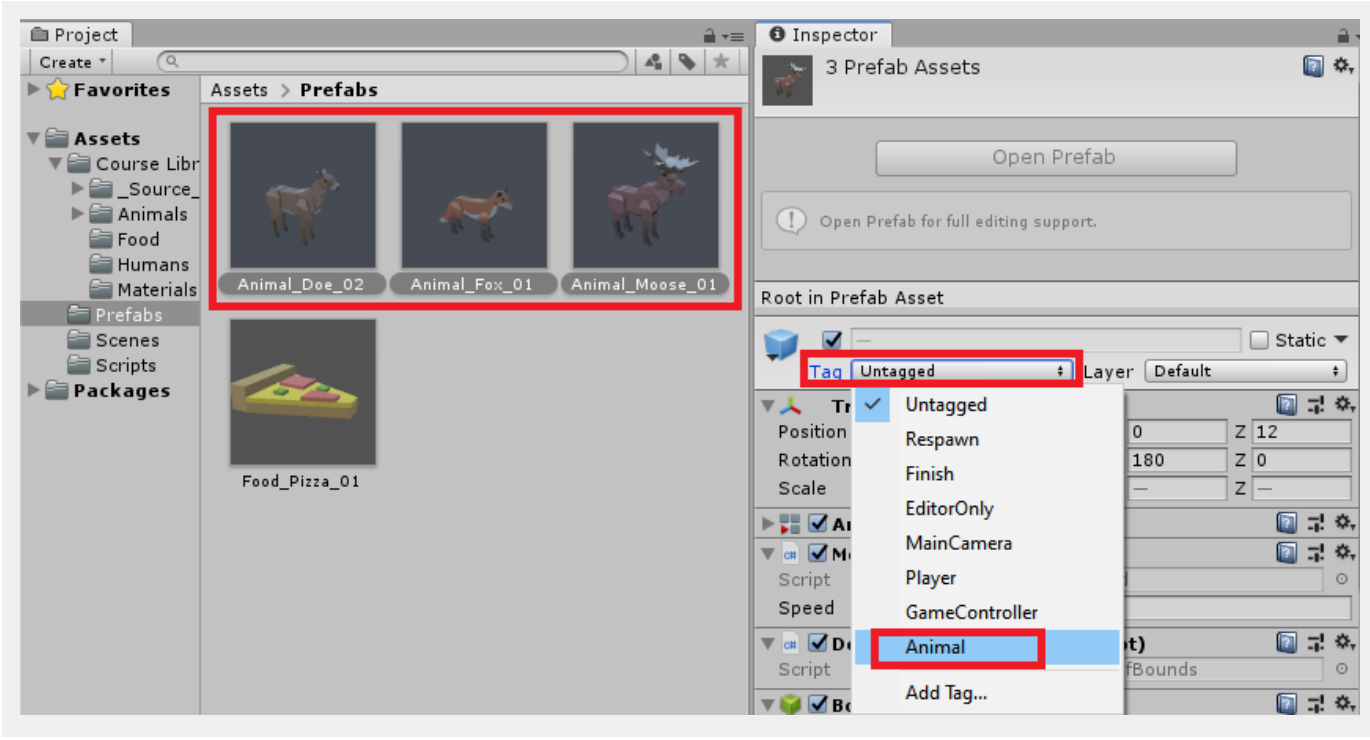
11. Save the scene. In the Project window, navigate to the Prefabs folder. Select one of the animals, then select the Tag property and click **Add Tag…** .



12. The Inspector will now show the Tags & Layers window. Under the tags option, click the **+** and write in **Animal**. Then click **Save**.



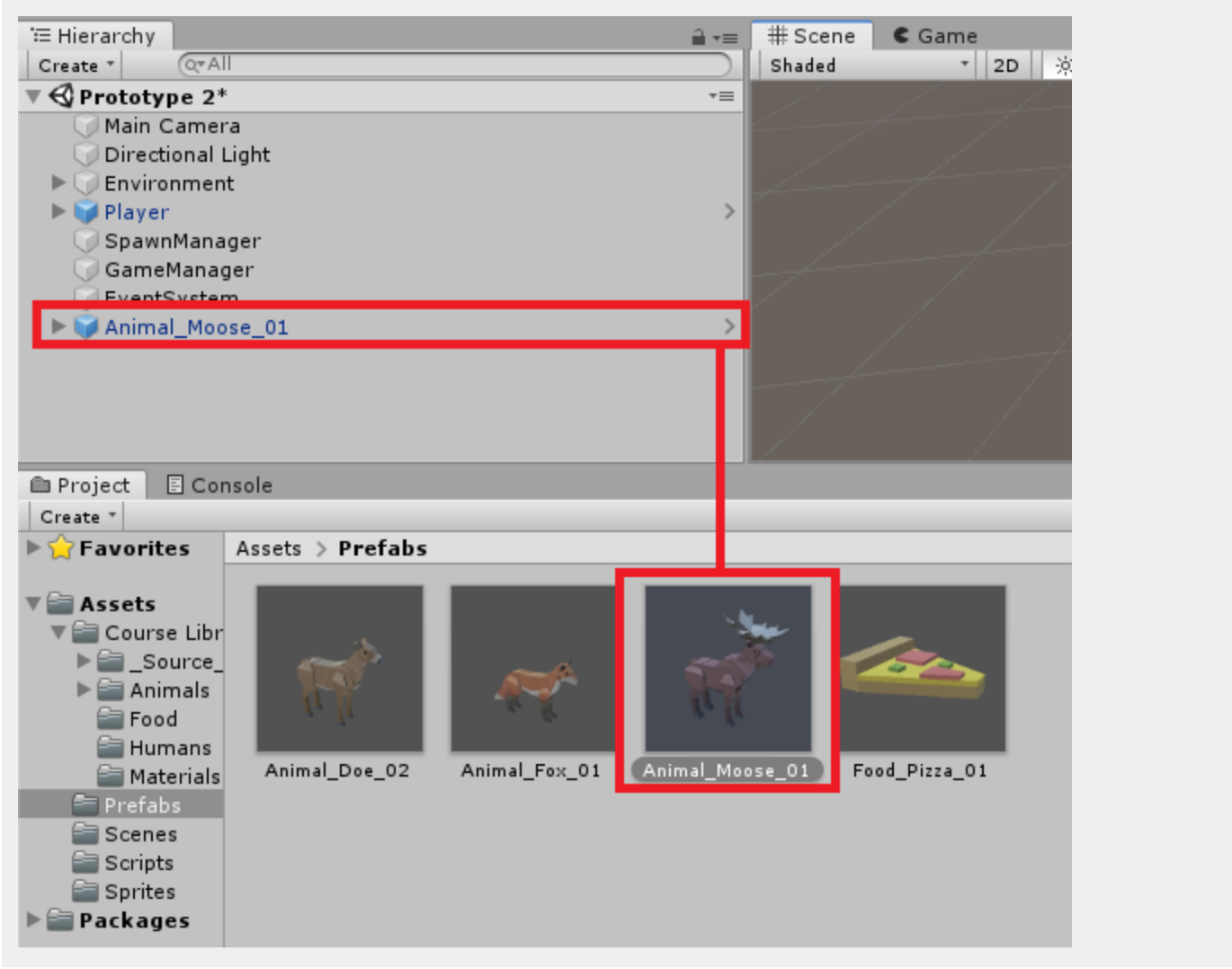
13. Now that we have the tag, assign it to all of your animals. You can do this quick by selecting all the animal prefabs and changing their tag to **Animal**.



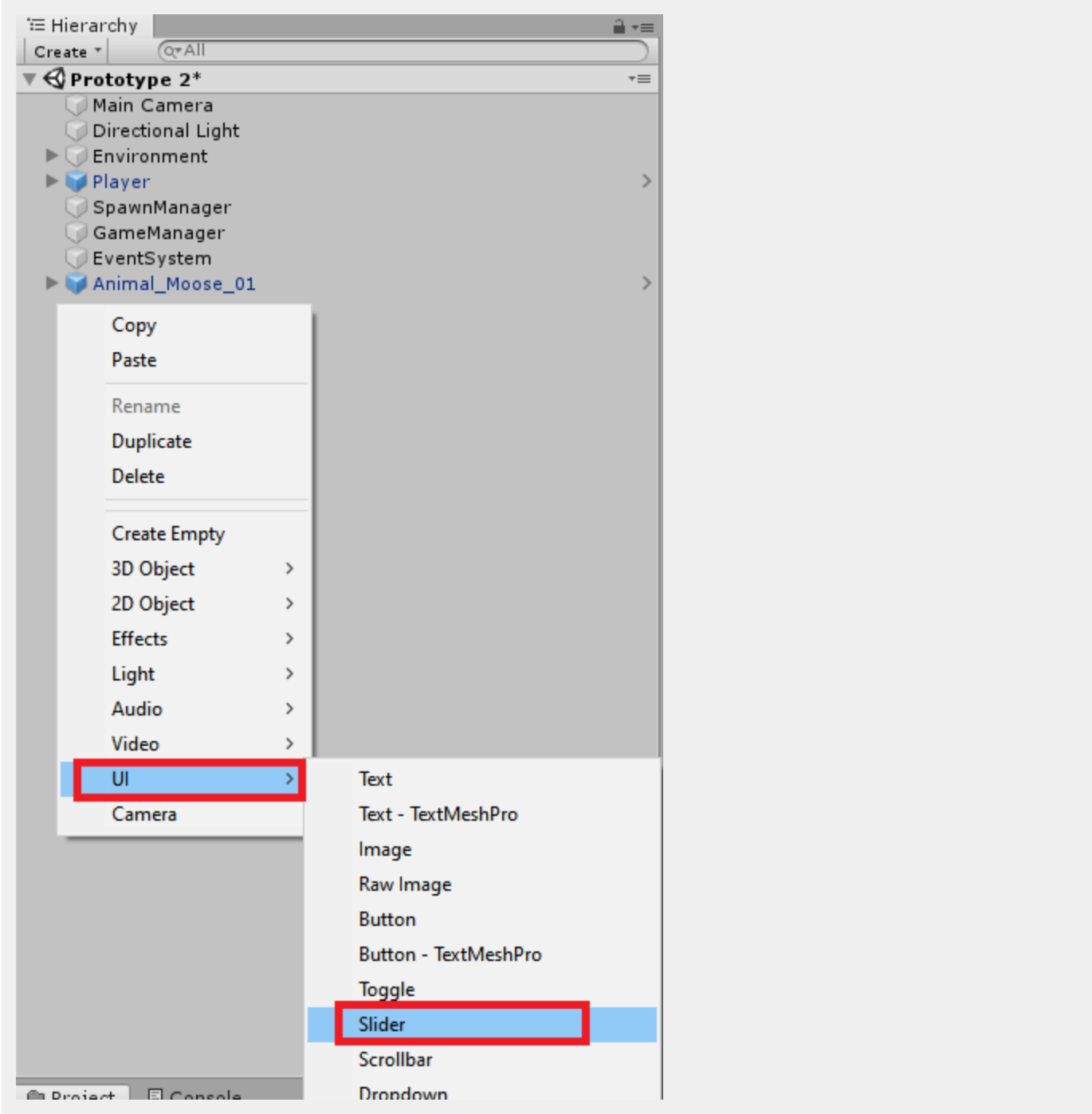
14. Save the project and press Play. You should now see a score in the console every time you feed an Animal. The current amount of lives will also be displayed when an animal gets passed, or collides with, the player.

Expert - Animal Hunger Bar

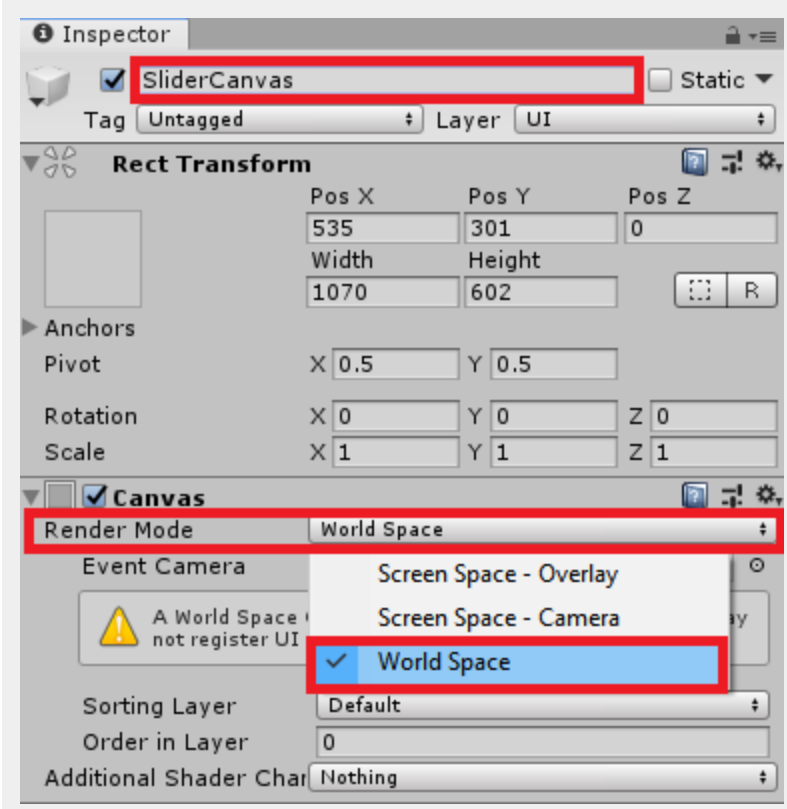
1. We will start out by creating the UI for the hunger bar. Navigate to the Prefabs folder and drag one of the animals into the Hierarchy



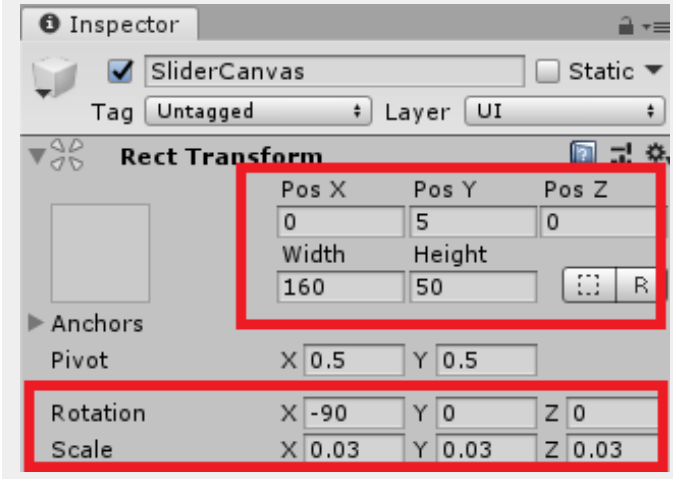
2. Right-click in the Hierarchy and select **UI** > **Slider**.



3. On the Canvas that was just created, rename the GameObject to *SliderCanvas*. Find the **Canvas** component and change the **Render Mode** to **World Space**.

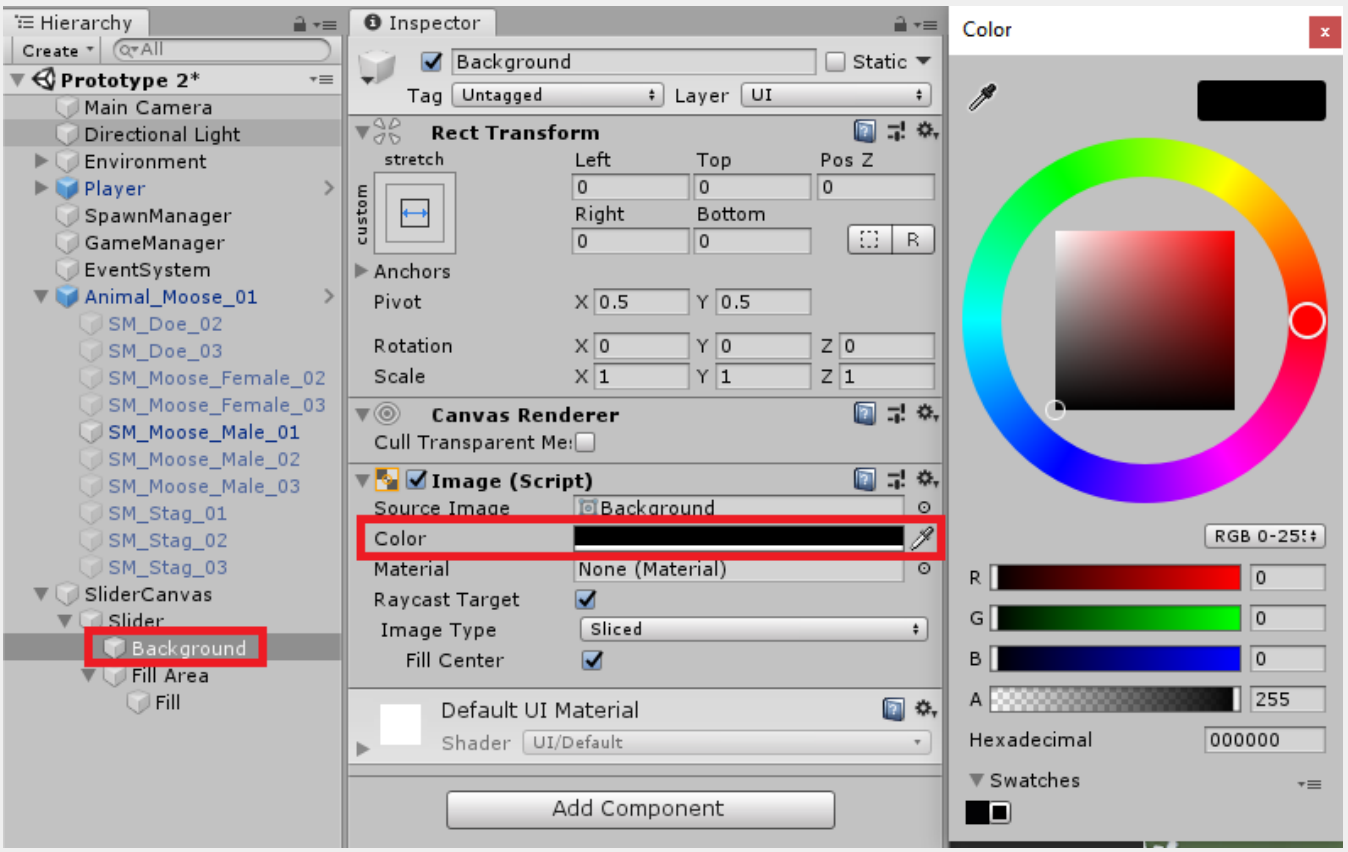


4. If you zoom out of the scene view, you will see that the slider is currently massive. We need to adjust the size of the canvas so that it will be viewable. The settings we used for the **Rect Transform** of the *SliderCanvas* can be seen below:

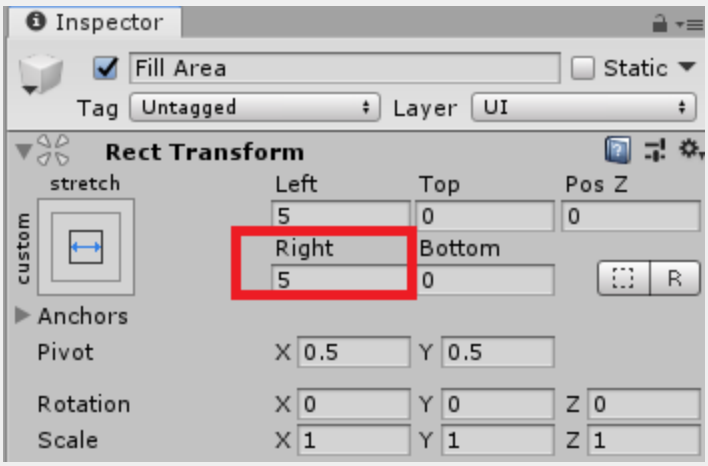


5. Next we will adjust the Slider. In the Hierarchy, click the arrow to show the child GameObjects of the Slider.

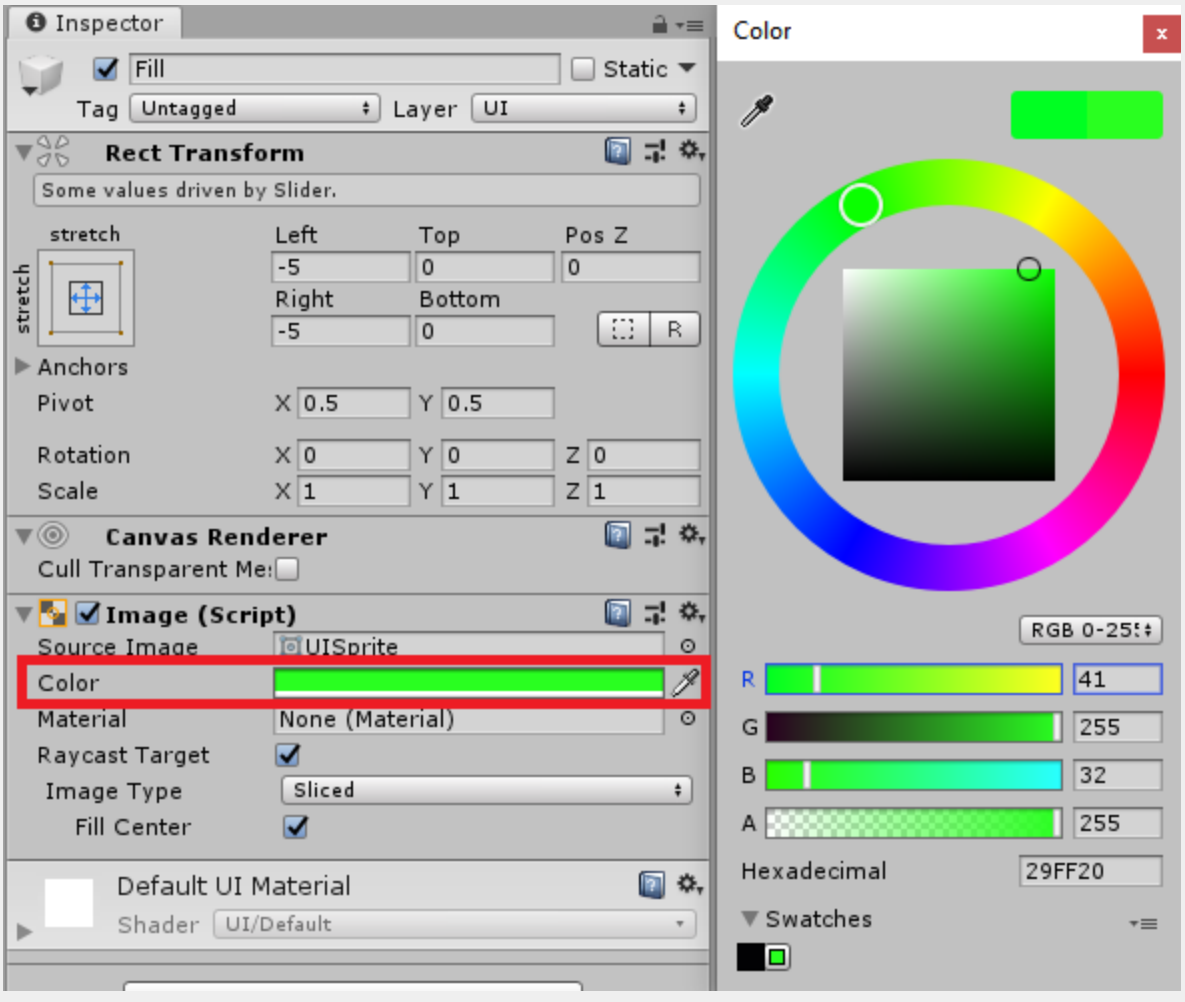
6. We won't need the Handle, so delete that GameObject. Select the *Background* GameObject, and adjust the **Color** of the **Image** component.



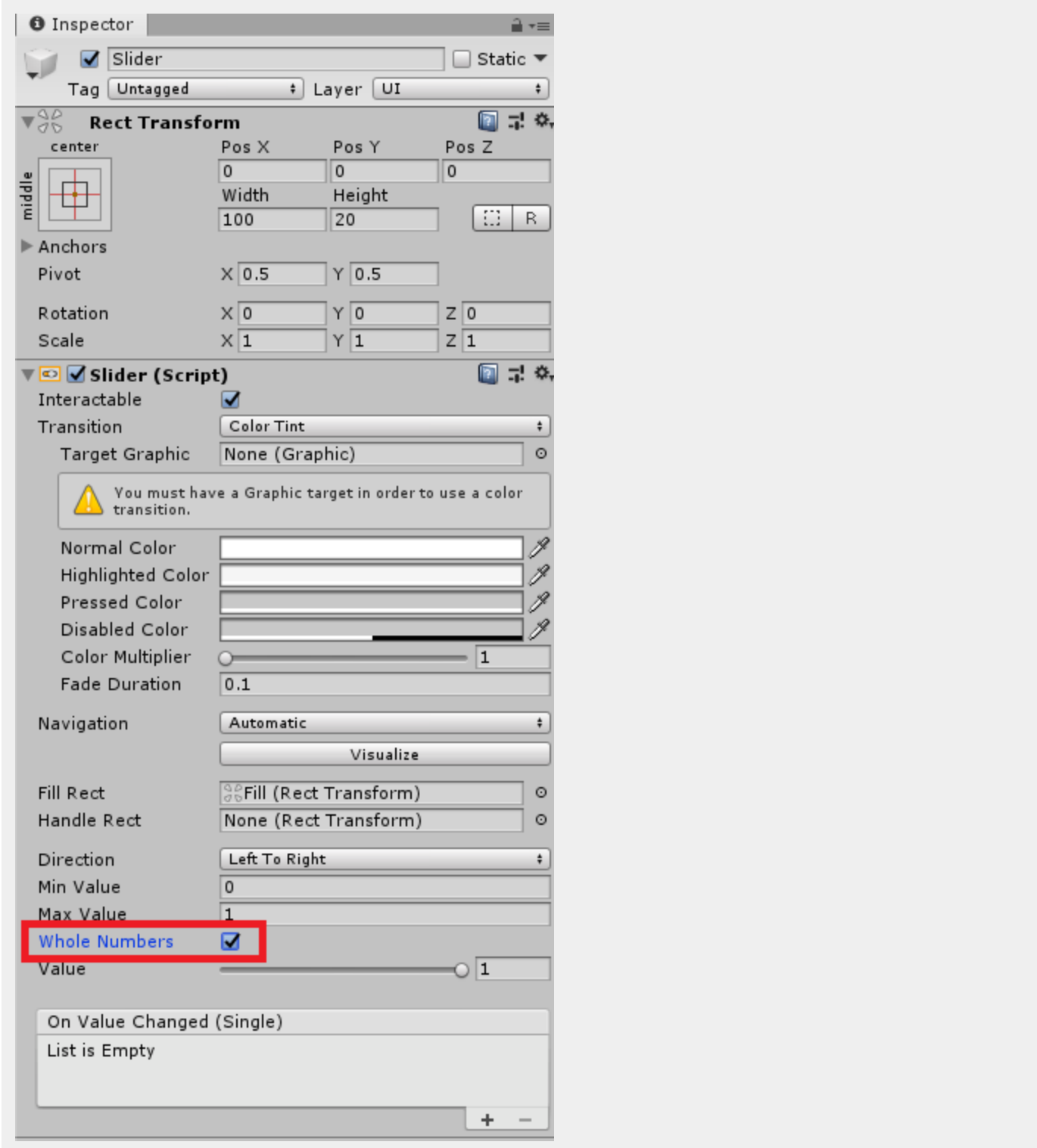
7. On the *Fill Area* GameObject, we will need to adjust the **Rect Transform** component. Change the **Right** value to **5**.



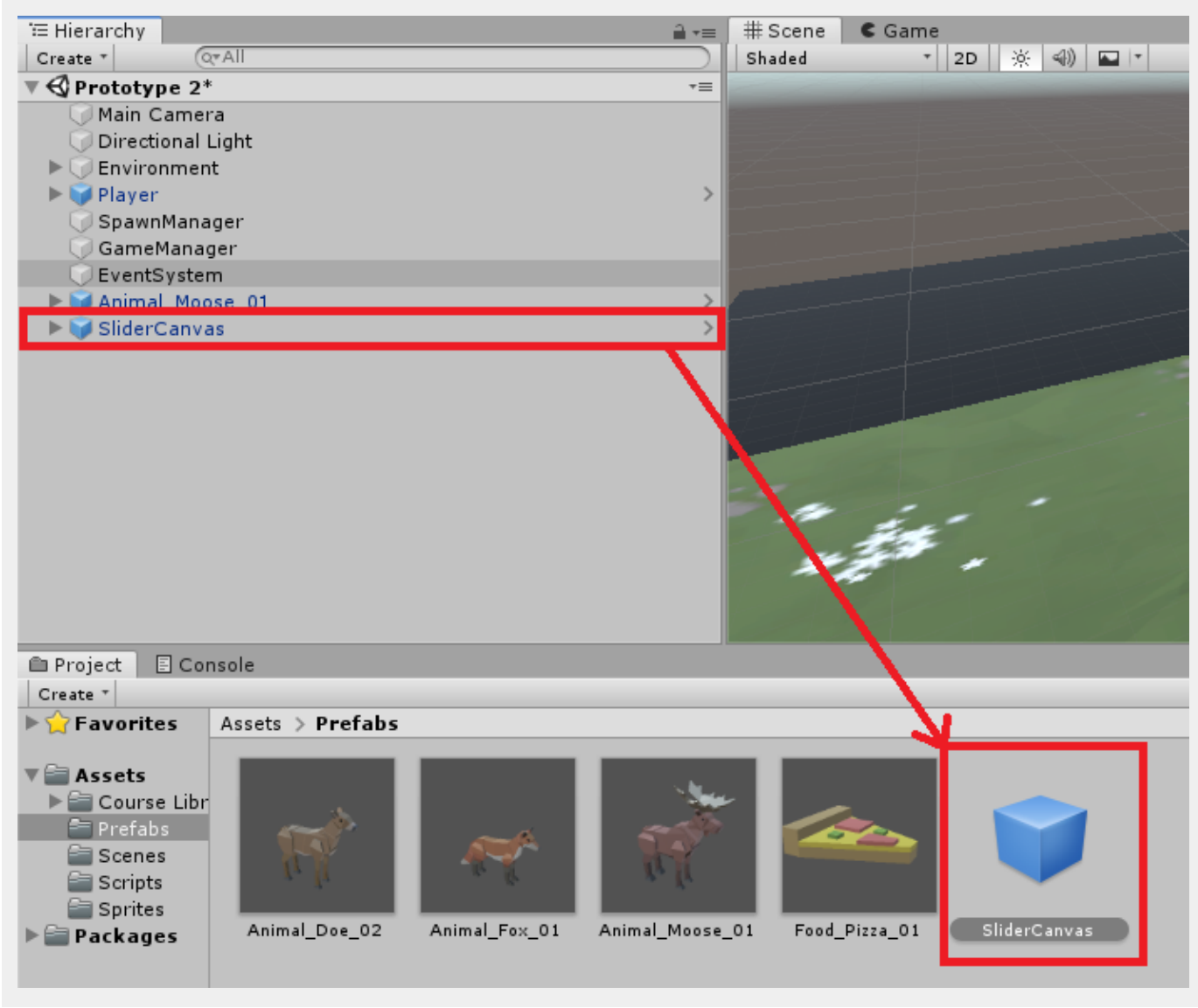
8. On the *Fill* GameObject, we will adjust the **Color** of the **Image** component. We set ours to green.



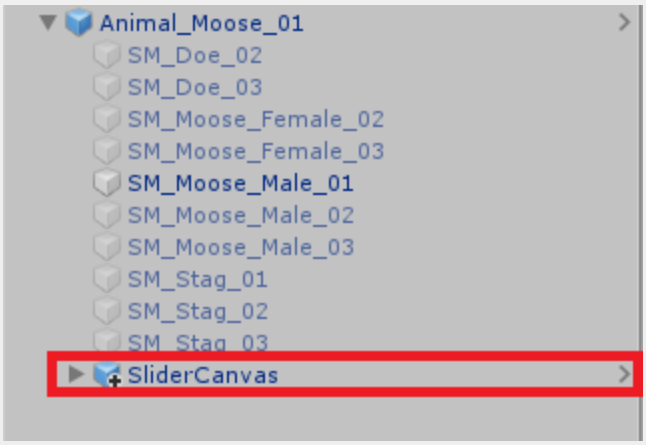
9. The next thing we need to do for the *Slider*, is adjust the **Slider** component. On the **Slider** component, check the **Whole Numbers** property.



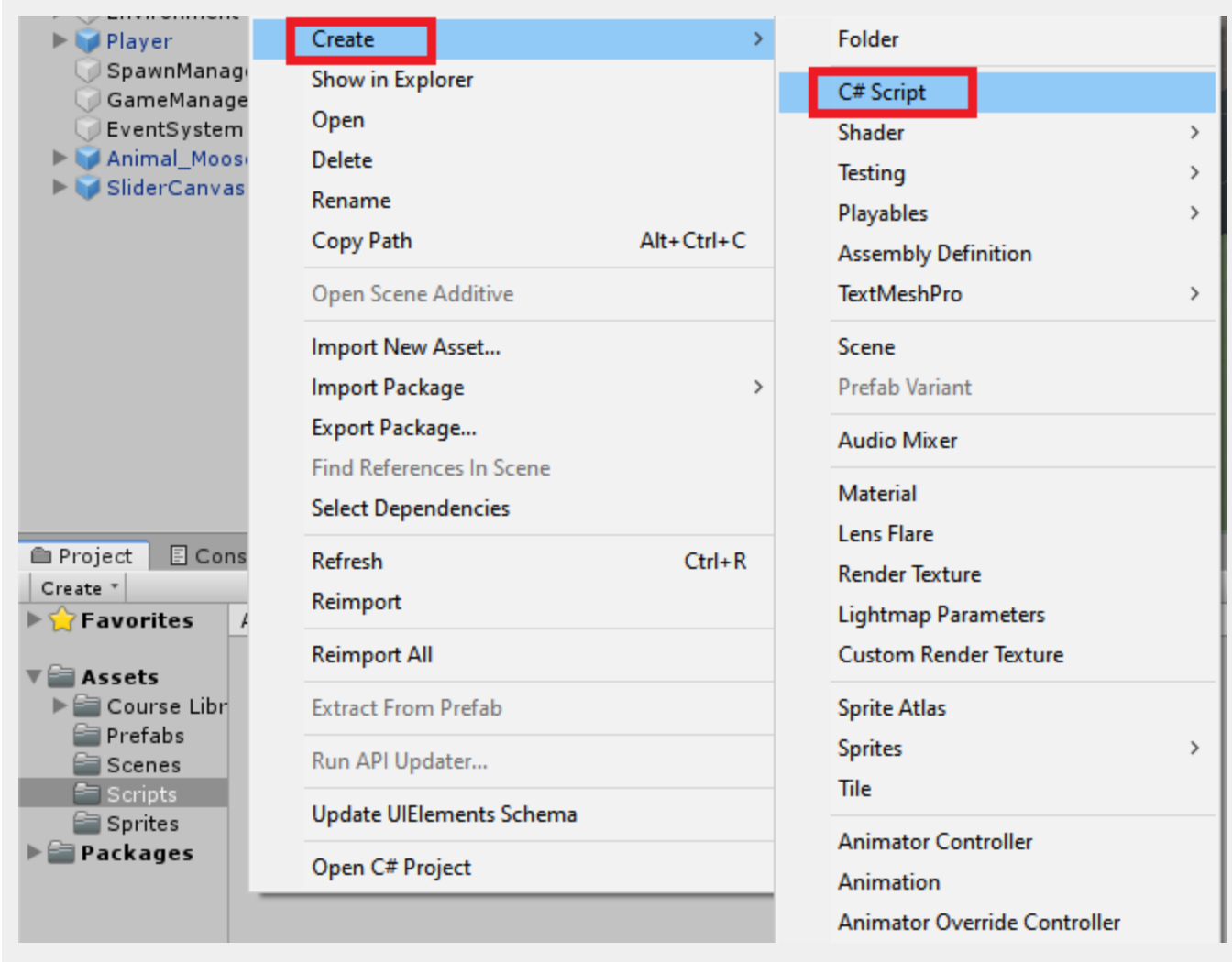
10. Drag the *SliderCanvas* from the Hierarchy into the Prefabs folder in the Project View



11. Add the *SliderCanvas* prefab to the animal you have within the scene.



12. Navigate to the Scripts folder, right-click and select **Create** > **C# Script**. Name the new script *AnimalHunger*.



13. Open up the new script. The first thing we need to do is include another namespace. At the top of the script, update the 'using' section to look like this:

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

14. Next, let's set up the variables. Before the **Start** method, add the following:

```
public Slider hungerSlider;
public int amountToBeFed;

private int currentFedAmount = 0;
```

```
private GameManager gameManager;
```

15. In the **Start** method, we will set up the maxValue of the slider, as well as the GameManager variable. The updated **Start** method should look like this:

```
void Start()
{
    hungerSlider.maxValue = amountToBeFed;
    hungerSlider.value = 0;
    hungerSlider.fillRect.gameObject.SetActive(false);

    gameManager = GameObject.Find("GameManager").GetComponent<GameManager>();
}
```

16. Below the **Update** method, we will create a new method that will be called to update the current fed amount of the animal.

```
public void FeedAnimal(int amount)
{
    currentFedAmount += amount;
    hungerSlider.fillRect.gameObject.SetActive(true);
    hungerSlider.value = currentFedAmount;

    if(currentFedAmount >= amountToBeFed)
    {
        gameManager.AddScore(amountToBeFed);
        Destroy(gameObject, 0.1f);
    }
}
```

Save the script.

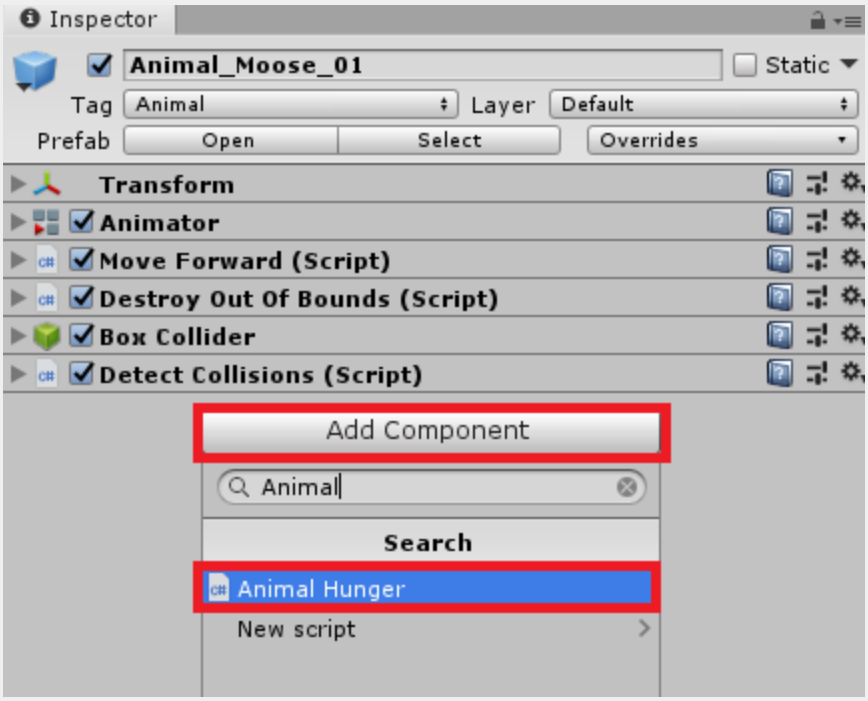
17. Open up the **DetectCollisions** script. We now need to adjust the collision when the food hits the animal. Update the OnTriggerEnter method to look like this:

```
private void OnTriggerEnter(Collider other)
{
    //Check if the other tag was the Player, if it was remove a life
    if (other.CompareTag("Player"))
    {
        gameManager.AddLives(-1);
        Destroy(gameObject);
    }
    //Check if the other tag was an Animal, if so add points to the score
    else if (other.CompareTag("Animal"))
```

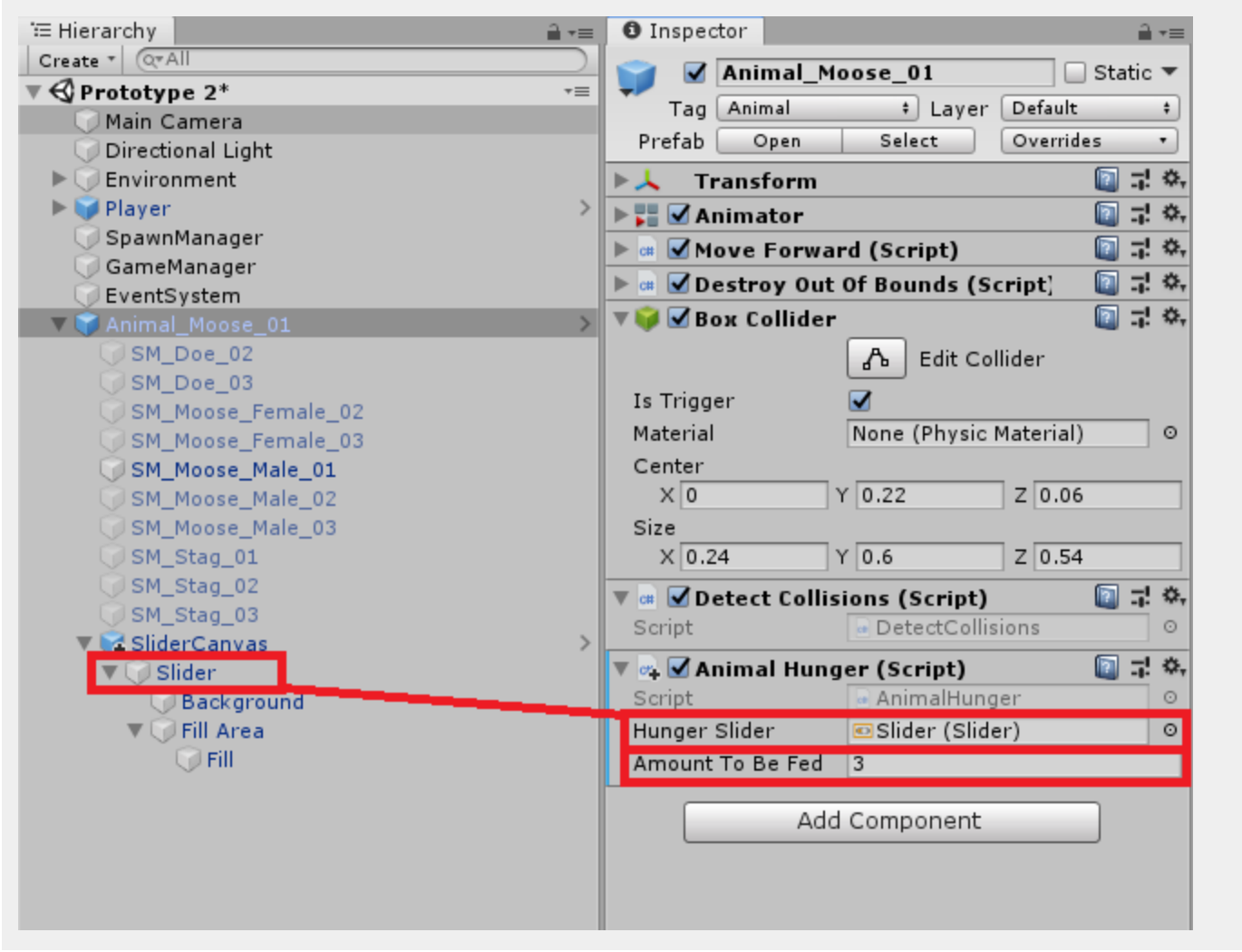
```
    {
        gameManager.AddScore(5);
        other.GetComponent<AnimalHunger>().FeedAnimal(1);
        Destroy(other.gameObject);
        Destroy(gameObject);
    }
}
```

Save the script and head back to Unity.

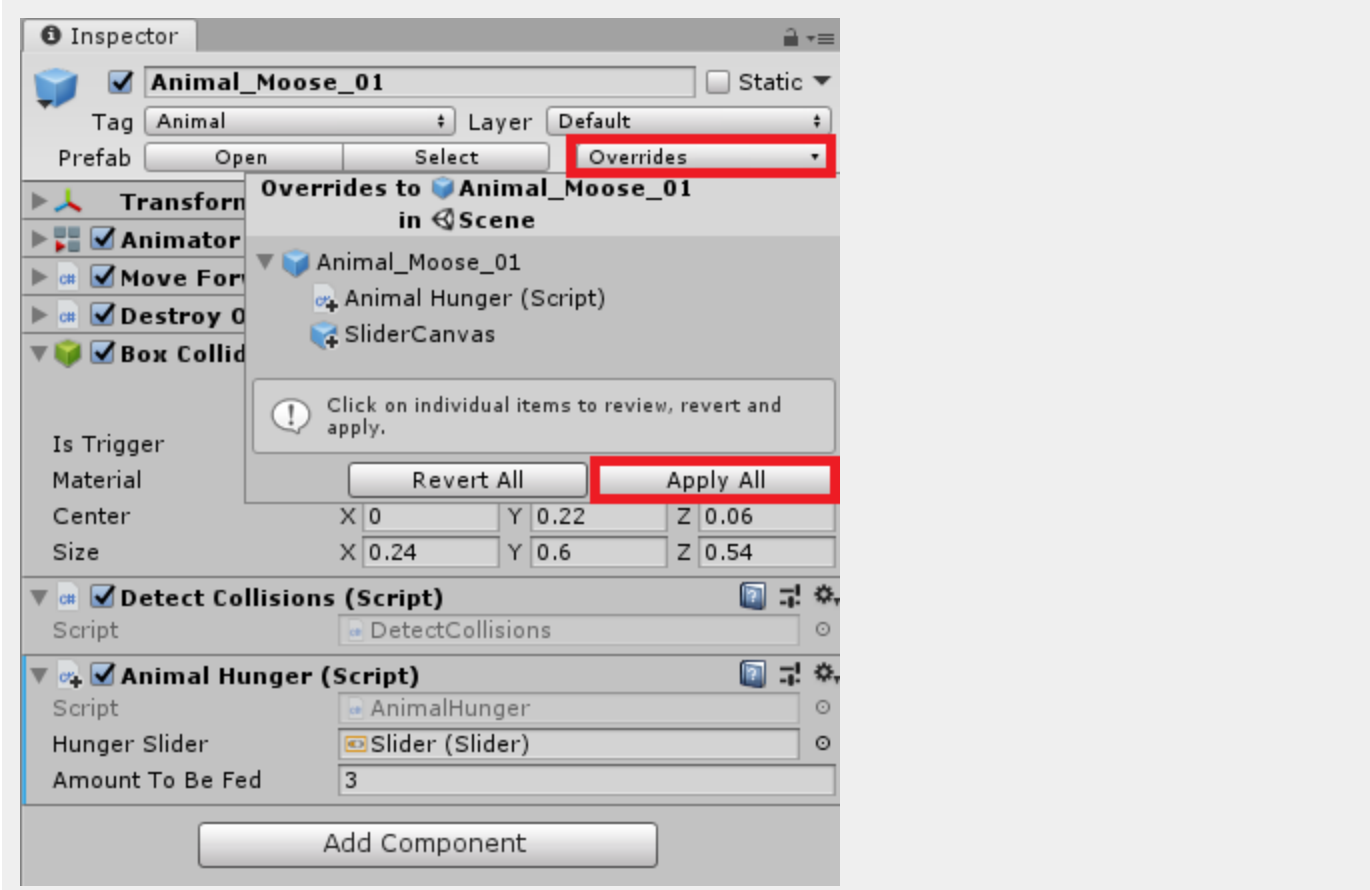
18. In the Hierarchy, select the Animal prefab that you placed earlier in the scene. In the Inspector, click **Add Component** and search for the *AnimalHunger* script.



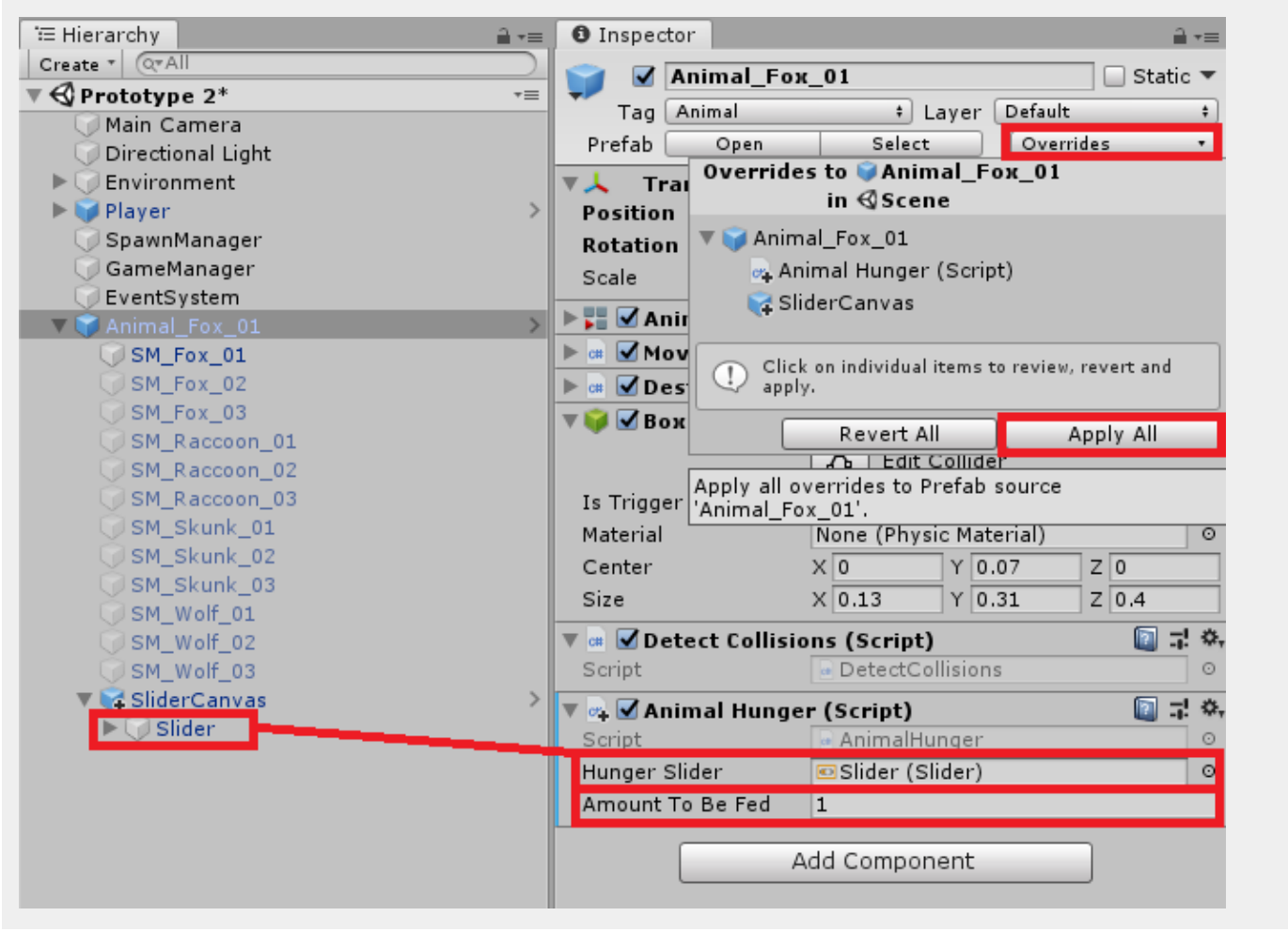
19. Drag the Slider that is a child of the animal, into the **Hunger Slider** parameter. Adjust the amount to be fed based on the animal being selected. For the Moose we used the value of **3**.

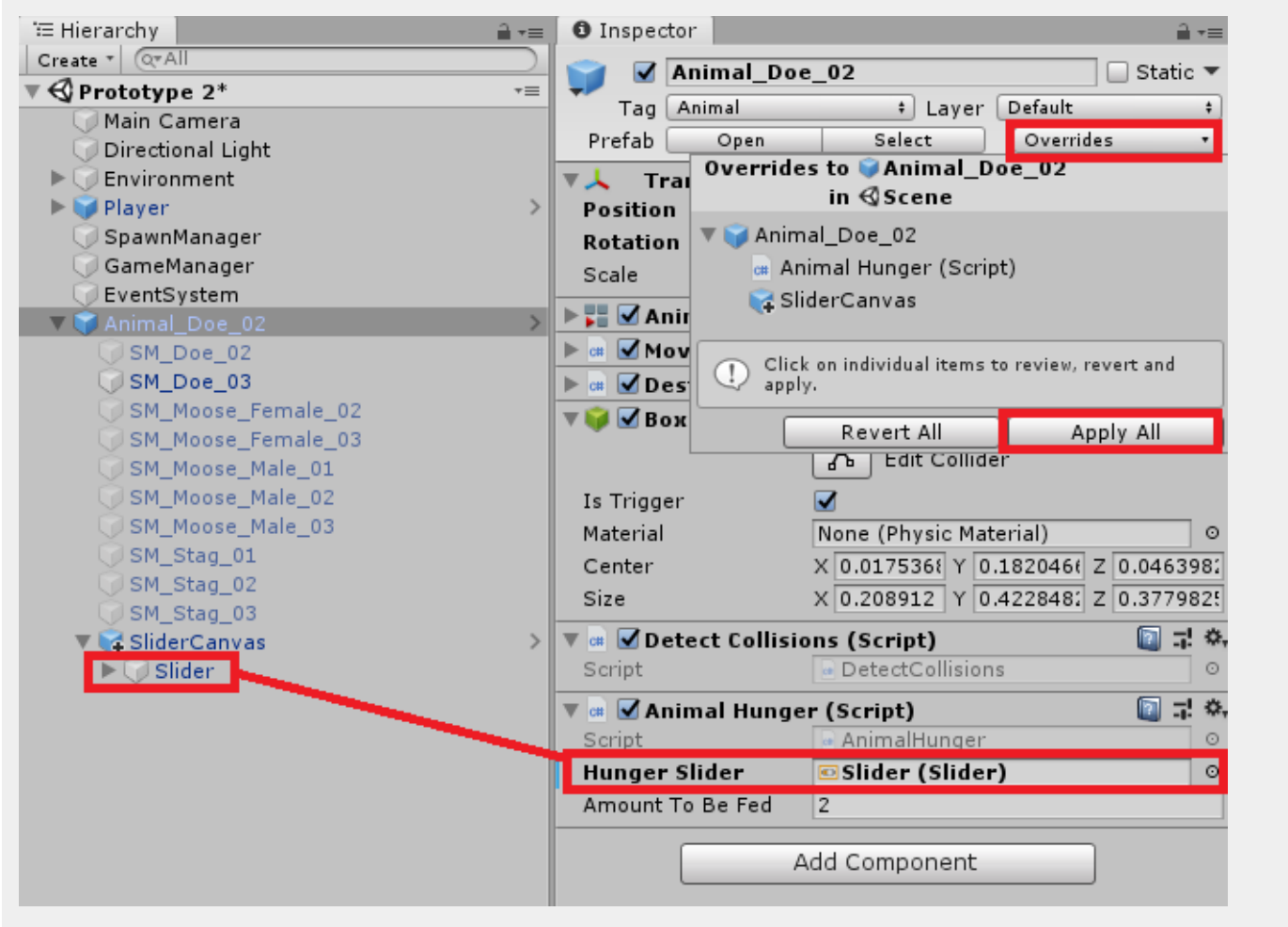


20. Let's apply the changes to the prefab. In the Inspector, click **Overrides** and then **Apply All**.



21. Now repeat the process for each of the other animal prefabs. Ensure they have the *SliderCanvas* within their hierarchy and the **Animal Hunger** script applied to the root. The amount to be fed value we used for the other animals are:
    - Fox = 1
    - Doe = 2


Hierarchy
Create
All
Prototype 2*
Main Camera
Directional Light
Environment
Player
SpawnManager
GameManager
EventSystem
Animal_Fox_01
SM_Fox_01
SM_Fox_02
SM_Fox_03
SM_Raccoon_01
SM_Raccoon_02
SM_Raccoon_03
SM_Skunk_01
SM_Skunk_02
SM_Skunk_03
SM_Wolf_01
SM_Wolf_02
SM_Wolf_03
SliderCanvas
Slider
Inspector
Animal_Fox_01
Static
Tag
Animal
Layer
Default
Prefab
Open
Select
Overrides
Overrides to Animal_Fox_01 in Scene
Animal_Fox_01
Animal Hunger (Script)
SliderCanvas
Position
Rotation
Scale
Click on individual items to review, revert and apply.
Revert All
Apply All
Edit Collider
Is Trigger
Apply all overrides to Prefab source 'Animal_Fox_01'.
Material
None (Physic Material)
Center
X 0
Y 0.07
Z 0
Size
X 0.13
Y 0.31
Z 0.4
Detect Collisions (Script)
Script
DetectCollisions
Animal Hunger (Script)
Script
AnimalHunger
Hunger Slider
Slider (Slider)
Amount To Be Fed
1
Add Component



After you have set up the prefabs, remember to apply the changes and delete them from the scene.

22. Save the scene and press play. The animals should spawn with a black bar, the bar will go green the more you fire food at them.