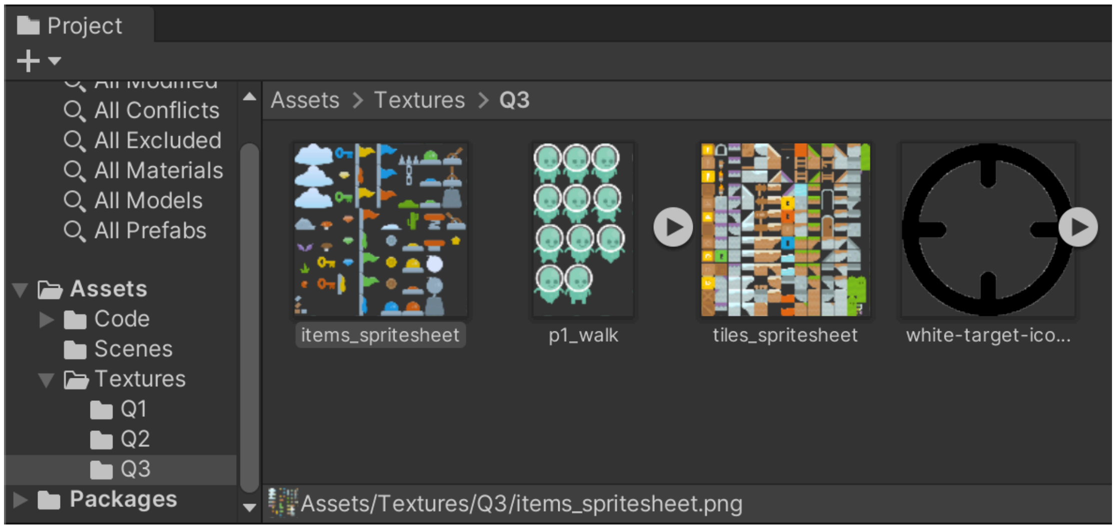
Quest 3 - Platformer - Steps

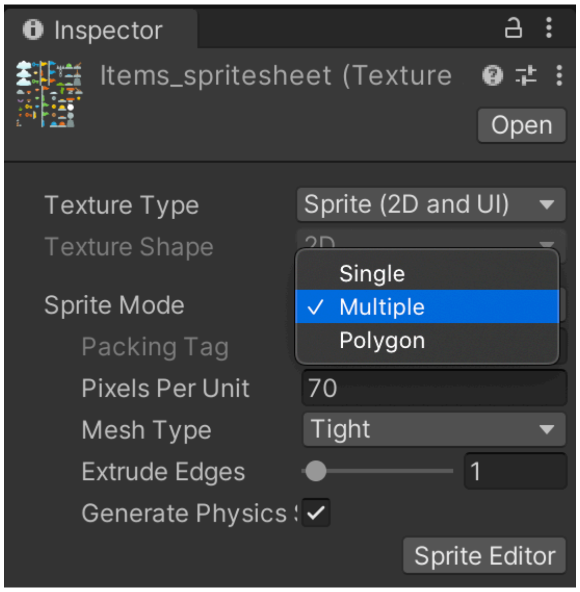
1) Import and Slice Graphics

Import this quest's asset files into /Textures/Q3/



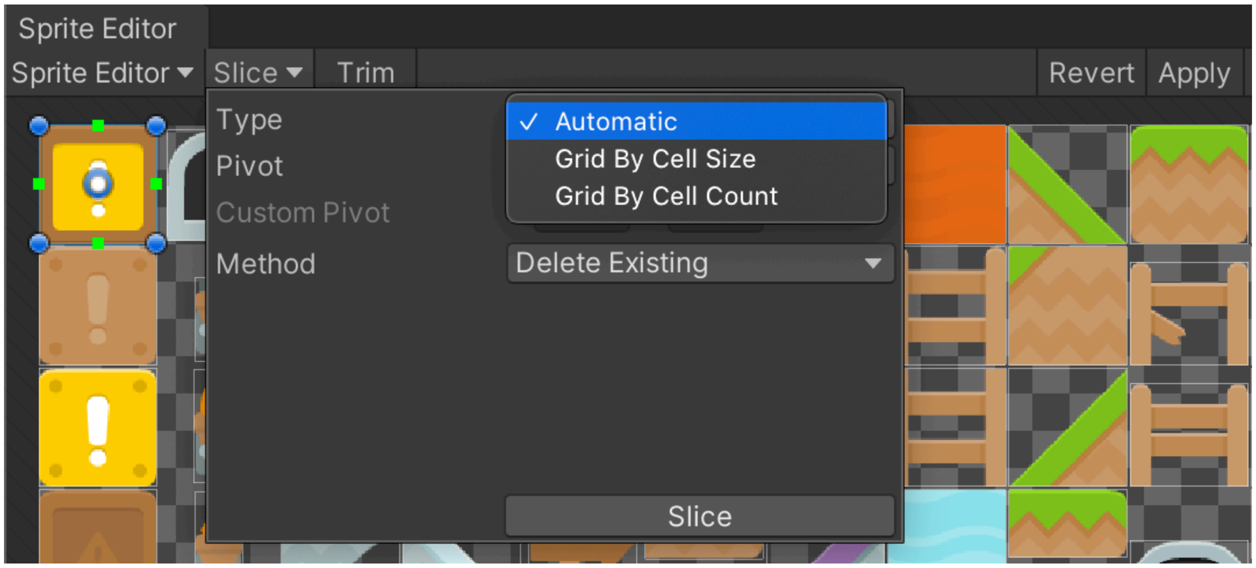
Assets/Textures/Q3/items_spritesheet.png

Set the items_spritesheet, tiles_spritesheet, and p1_walk Sprite Mode to Multiple, Pixels Per Unit to 70, and click Sprite Editor. Click Apply if prompted.



Texture Type: Sprite (2D and UI)
Texture Shape: 2D
Sprite Mode: ✓ Multiple
Pixels Per Unit: 70
Mesh Type: Tight
Extrude Edges: 1
Generate Physics Shape: ✓

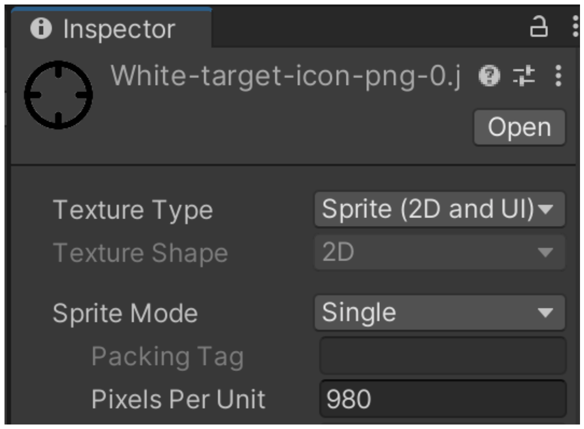
In Sprite Editor, click the Slice dropdown, set Type to Automatic, click the Slice button, then click Apply.



Type: ✓ Automatic / Grid By Cell Size / Grid By Cell Count
Pivot
Custom Pivot
Method: Delete Existing
Slice

Make sure that these steps are duplicated on both items_spritesheet and tiles_spritesheet because they are both multi-sprite sheets instead of single sprite files.

The target-icon will use 980 Pixels Per Unit because it is such a large image.

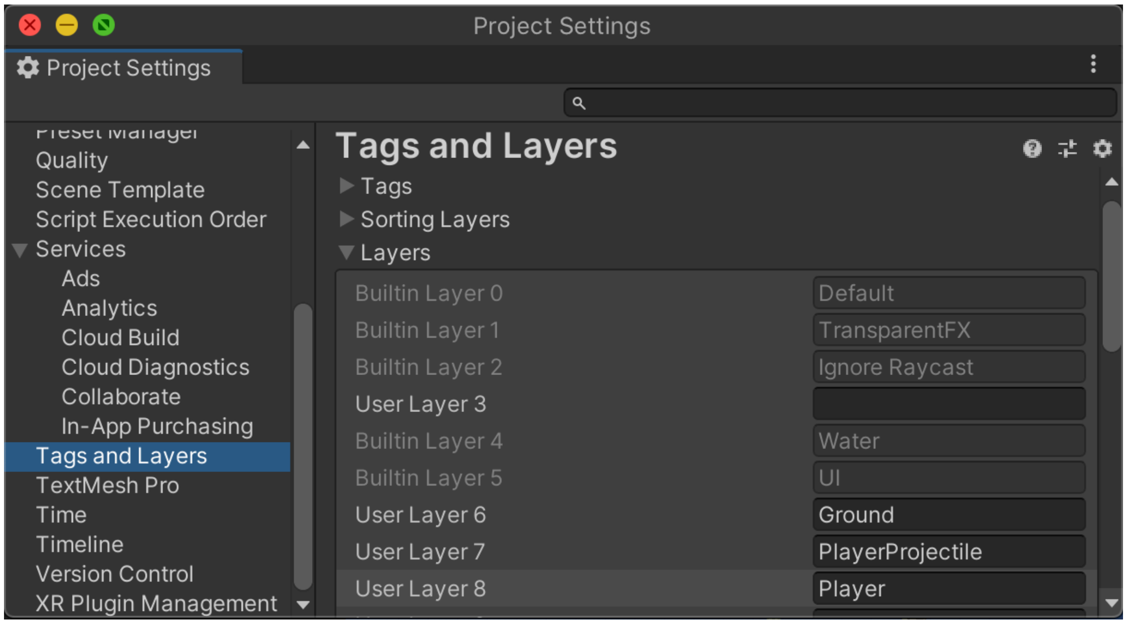


Texture Type: Sprite (2D and UI)
Texture Shape: 2D
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 980

2) Setup Layers and Collisions

We need to prepare Object Layers, so we can control what kinds of objects can collide with each other.

From Edit->Project Settings->Tags and Layers, add Layers for “Ground,” “PlayerProjectile,” and “Player.”



Tags and Layers

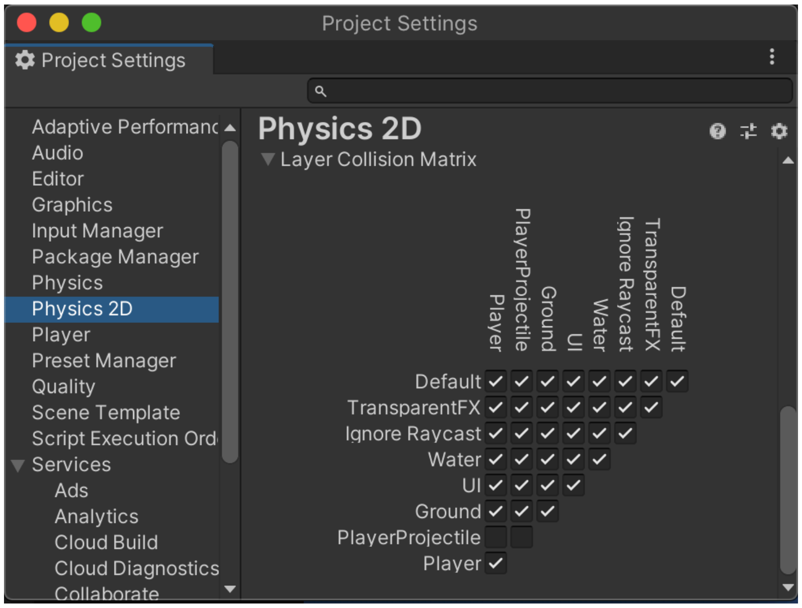
▶ Tags

▶ Sorting Layers

▼ Layers

Builtin Layer 0	Default
Builtin Layer 1	TransparentFX
Builtin Layer 2	Ignore Raycast
User Layer 3	
Builtin Layer 4	Water
Builtin Layer 5	UI
User Layer 6	Ground
User Layer 7	PlayerProjectile
User Layer 8	Player

From Edit->Project Settings->Physics2D, alter the collision matrix, so that Players cannot collider with their own PlayerProjectiles. We also do not want PlayerProjectiles to hit themselves.



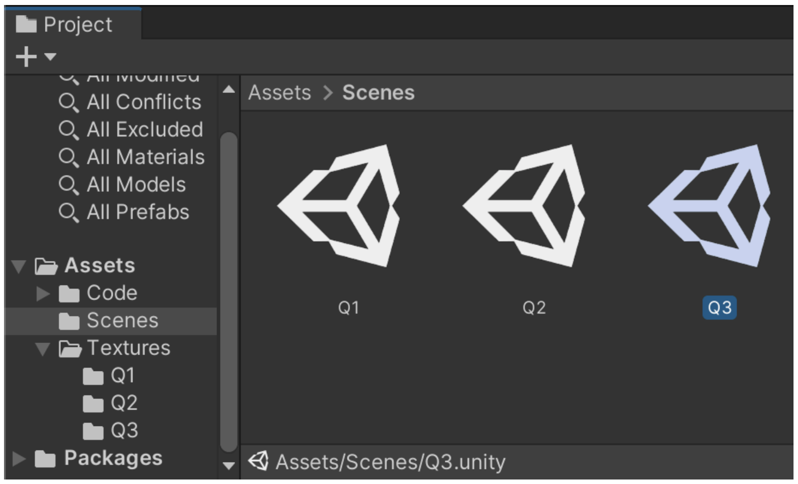
Physics 2D

▼ Layer Collision Matrix

	Player	PlayerProjectile	Ground	UI	Water	Ignore Raycast	TransparentFX	Default
Default	✓	✓	✓	✓	✓	✓	✓	✓
TransparentFX	✓	✓	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓	✓	✓		
Water	✓	✓	✓	✓	✓			
UI	✓	✓	✓	✓				
Ground	✓	✓	✓					
PlayerProjectile	☐	☐						
Player	✓							

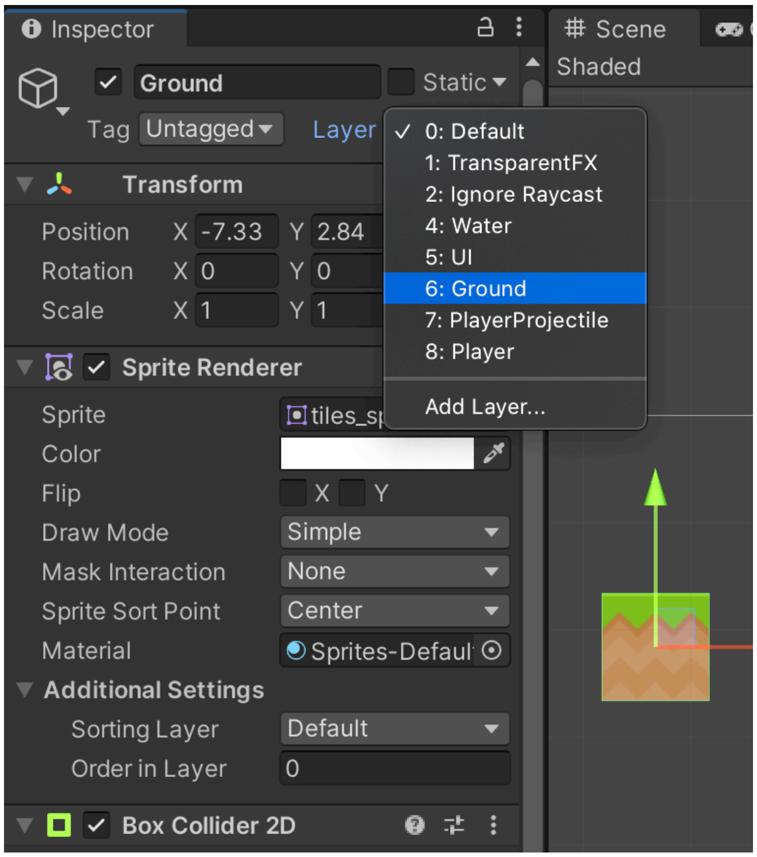
3) Create Ground Prefab

Ensure you are in a new scene for the following steps.



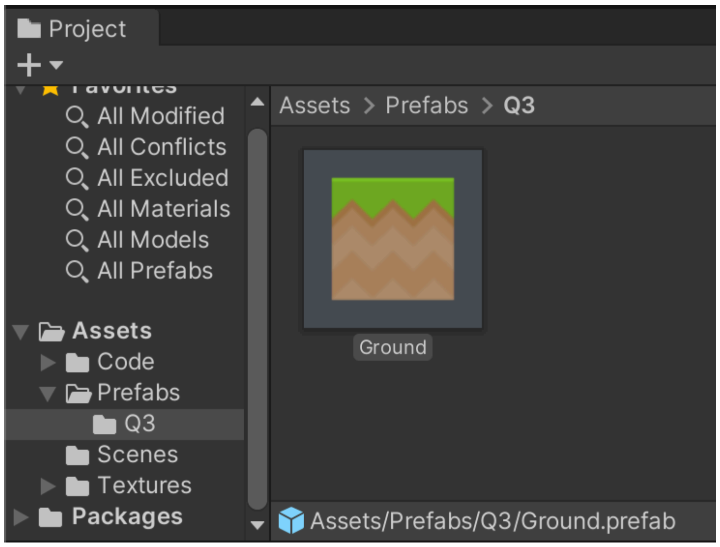
In the Q3 hierarchy, create an Empty Object and add a Sprite Renderer component. Assign a desired Ground Sprite. Add a BoxCollider2D component.

Assign a Layer of “Ground.”

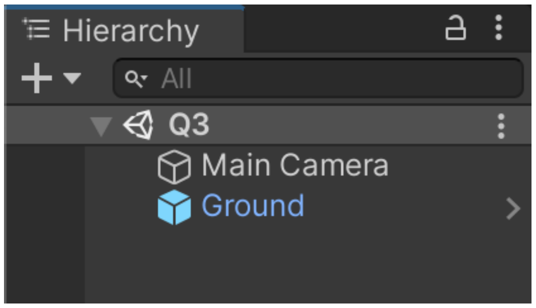


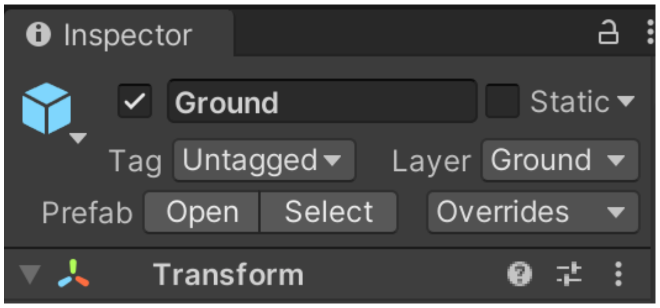
Making Prefabs is a way to prepare a template copy of an object that will be re-used multiple times throughout a project. You only need to change the template and all of the copies will update with those changes. Because we have a lot of ground blocks, it is a good candidate for a prefab.

Create a /Prefabs/Q3/ folder to hold prefabs for this exercise. Drag the Ground game object from the Hierarchy list to the Project library to create a reusable ground Prefab.



Notice how prefabs have a different icon in the Hierarchy and a new row of Prefab options in the Inspector.



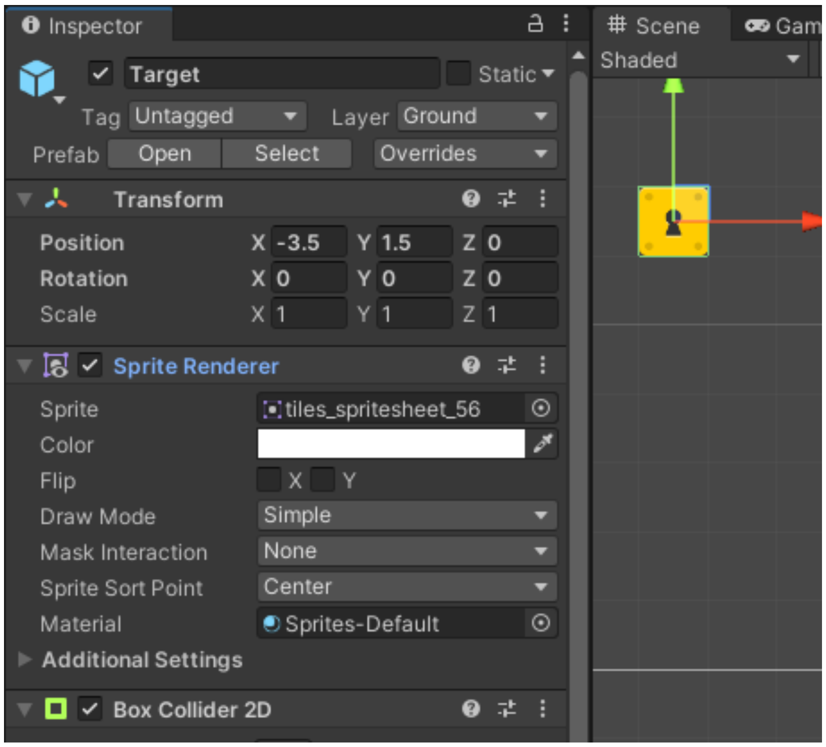


4) Create Target Prefab

Add a Sprite Renderer component to an Empty Object. Assign a desired Target Block Sprite. Add a BoxCollider2D component.

Assign a Layer of “Ground.”

Drag the Target game object from the Hierarchy list to the Project library to create a reusable target Prefab.



Target
Static
Tag Untagged
Layer Ground
Prefab Open Select Overrides

Transform

Position	X -3.5	Y 1.5	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Sprite Renderer

Sprite	tiles_spritesheet_56
Color	
Flip	X Y
Draw Mode	Simple
Mask Interaction	None
Sprite Sort Point	Center
Material	Sprites-Default

Additional Settings

Box Collider 2D

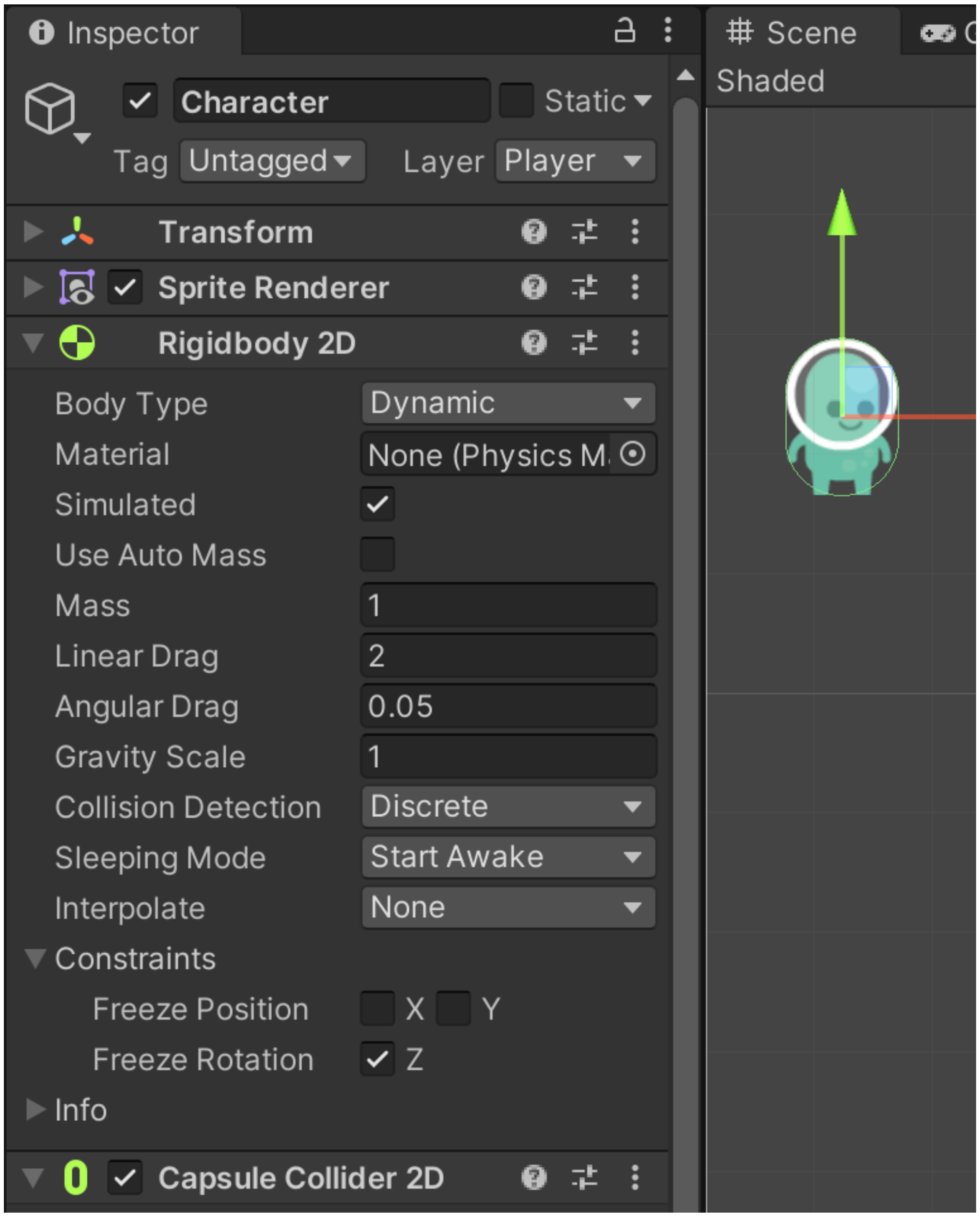
5) Create Character

Add a Sprite Renderer component to an Empty Object. Assign a desired Character Sprite.

Assign a Layer of “Player.”

Add a CapsuleCollider2D component. We use a Collider2D that has a rounded bottom to prevent the character from getting snagged on terrain.

Add a Rigidbody2D component. To prevent the character from rolling, check the Constraints->Freeze Rotation Z checkbox .


Inspector

Character — Static

Tag: Untagged — Layer: Player

Transform

Sprite Renderer

Rigidbody 2D

Body Type	Dynamic
Material	None (Physics M
Simulated	✓
Use Auto Mass	
Mass	1
Linear Drag	2
Angular Drag	0.05
Gravity Scale	1
Collision Detection	Discrete
Sleeping Mode	Start Awake
Interpolate	None
Constraints	
Freeze Position	X Y
Freeze Rotation	✓ Z
Info	

Capsule Collider 2D

6) Create Projectile

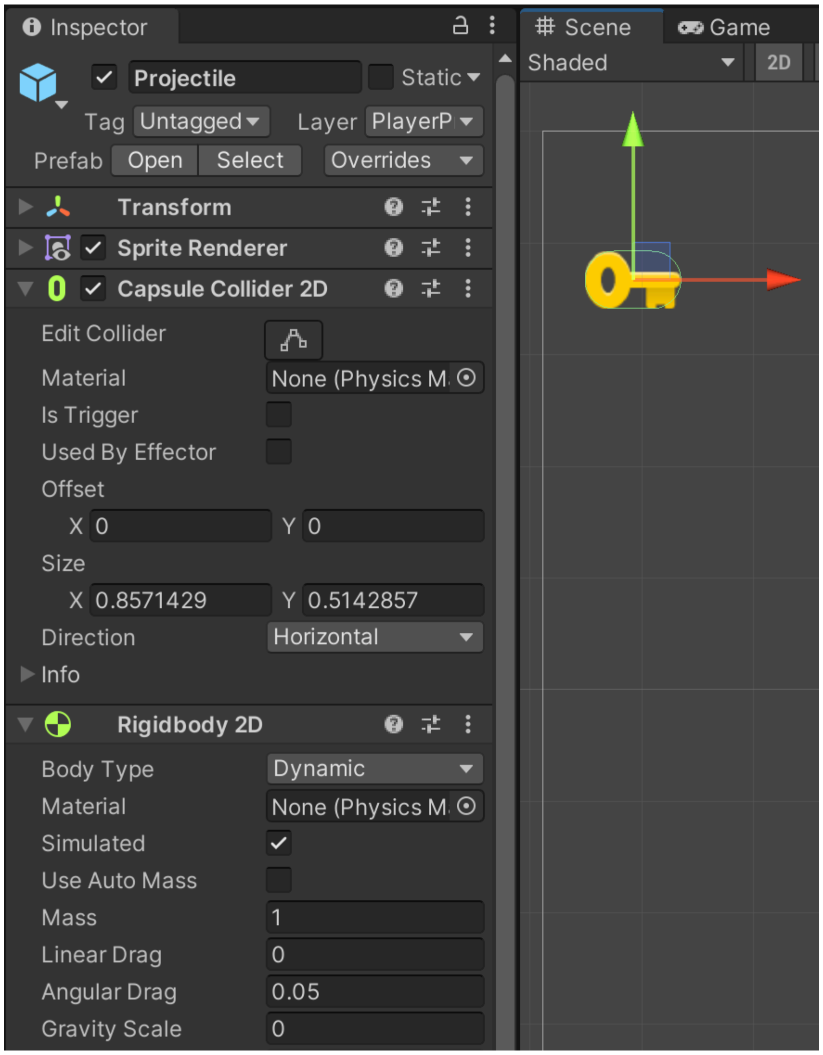
Add a Sprite Renderer component to an Empty Object. Assign a desired Projectile Sprite.

Assign a Layer of “PlayerProjectile.”

Add a CapsuleCollider2D component with a Direction setting of Horizontal.

Add a Rigidbody2D component and set Gravity to 0.

Drag the Projectile gameObject from the Hierarchy list to the Project library to create a reusable projectile Prefab.


Inspector

Projectile — Static

Tag: Untagged — Layer: PlayerP

Prefab: Open | Select | Overrides

Transform

Sprite Renderer

Capsule Collider 2D

- Edit Collider
- Material: None (Physics M
- Is Trigger: ☐
- Used By Effector: ☐
- Offset: X 0, Y 0
- Size: X 0.8571429, Y 0.5142857
- Direction: Horizontal
- Info

Rigidbody 2D

- Body Type: Dynamic
- Material: None (Physics M
- Simulated: ✓
- Use Auto Mass: ☐
- Mass: 1
- Linear Drag: 0
- Angular Drag: 0.05
- Gravity Scale: 0

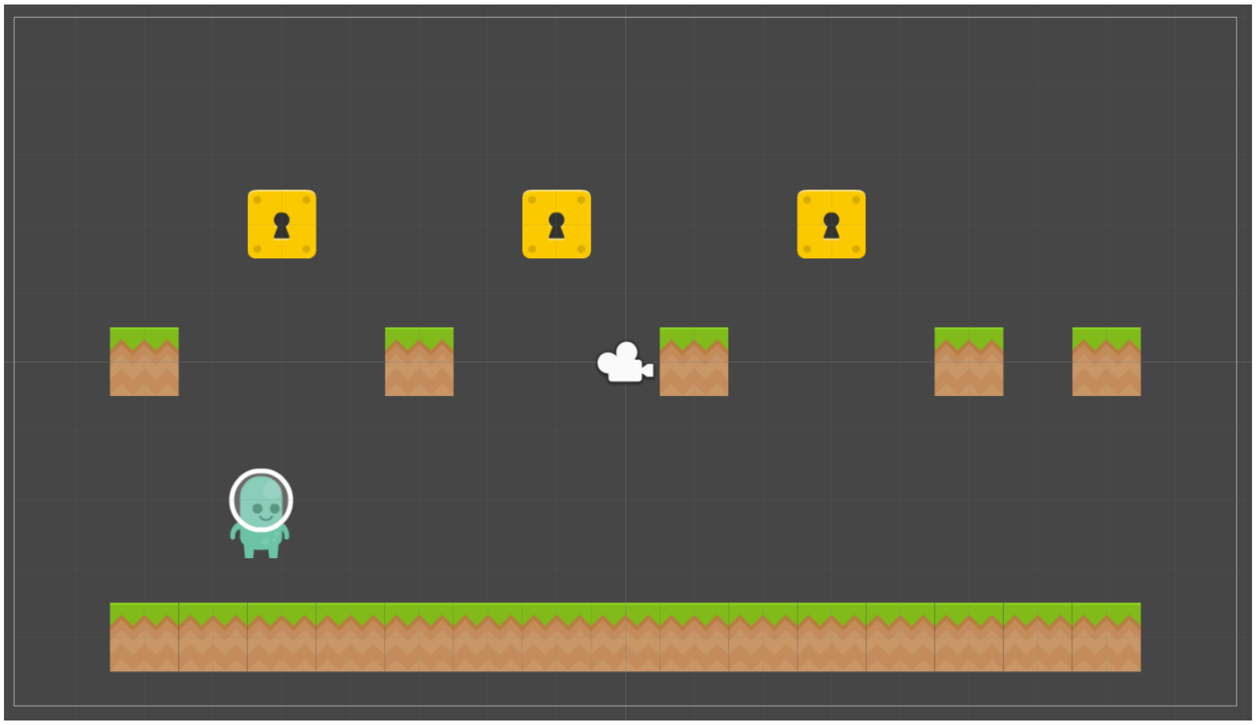
With the Projectile prefab in the Library, DELETE the Projectile from the Scene Hierarchy. Do NOT leave an unfired Projectile in the scene.

7) Create a Level Layout

Create a Level Layout using your Character, Ground prefabs, and Target prefabs. You need ground for the character to stand on, a hole in the ground the player might fall through, and targets that pose some kind of challenge to hit.

To use a prefab, you just drag it from the Project library to the Scene. If you edit one Prefab, you can update all of the other Copies of that prefab to match by clicking the Apply button in its inspector.

Similarly, updating the Prefab in the Project library will update all copies of it in the Scene.



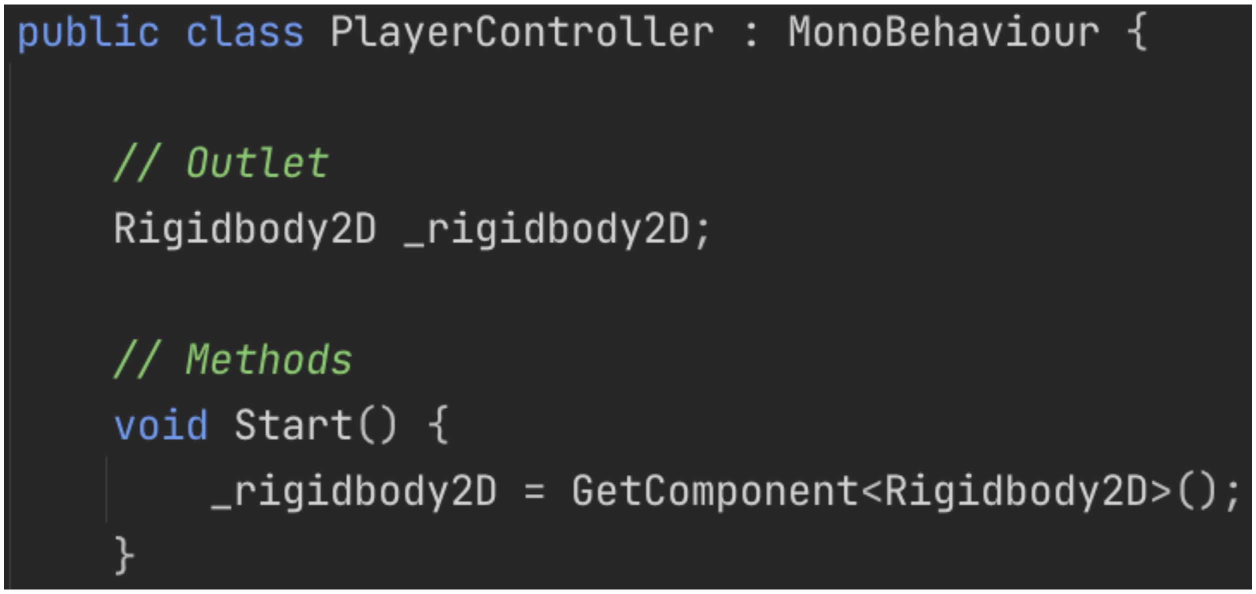
8) Program Character Movement

Create a new C# Script called “PlayerController” in /Code/Q3/ and attach it to the Character game object.

We will be using namespaces to help avoid contradicting code from different exercises. Enclose ALL classes created as part of this exercise in Q3.

```
namespace Q3 {
    public class PlayerController : MonoBehaviour{...}
}
```

Create a Reference Outlet to the Character gameObject’s Rigidbody2D component, so that we can reference it in code.

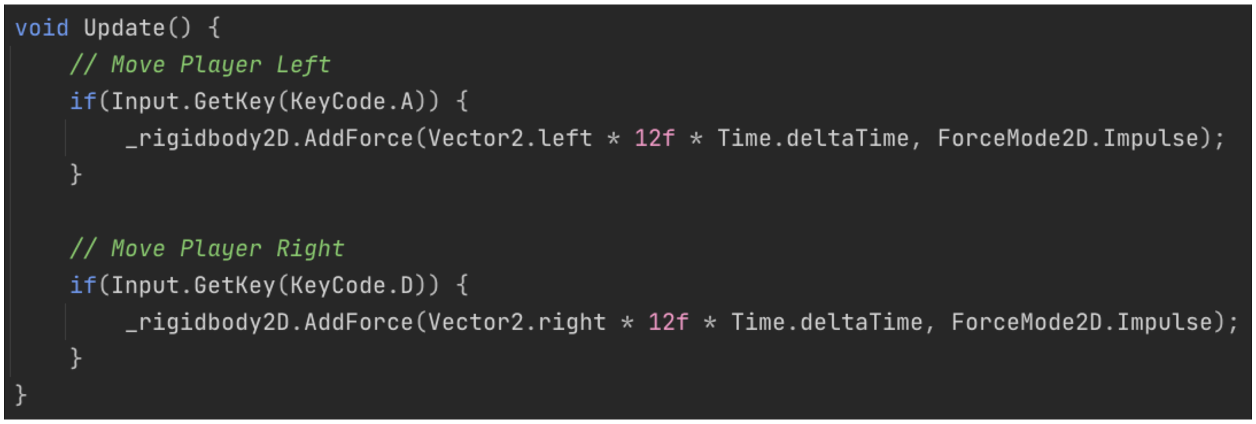


```
public class PlayerController : MonoBehaviour {

    // Outlet
    Rigidbody2D _rigidbody2D;

    // Methods
    void Start() {
        _rigidbody2D = GetComponent<Rigidbody2D>();
    }
```

Every Frame Update, we will check if the A or D keys are held down to move the character Left or Right using physics forces. The appropriate amount of force depends on Game Feel and might be a different number. Using a Force Mode of Impulse means we want the force to be applied instantly for that frame rather than computed as force per second.



```
void Update() {
    // Move Player Left
    if(Input.GetKey(KeyCode.A)) {
        _rigidbody2D.AddForce(Vector2.left * 12f * Time.deltaTime, ForceMode2D.Impulse);
    }

    // Move Player Right
    if(Input.GetKey(KeyCode.D)) {
        _rigidbody2D.AddForce(Vector2.right * 12f * Time.deltaTime, ForceMode2D.Impulse);
    }
}
```

On the Character gameObject, adjust the Linear Drag of the Rigidbody2D component for a proper Game Feel. This controls how the character slows down while moving.

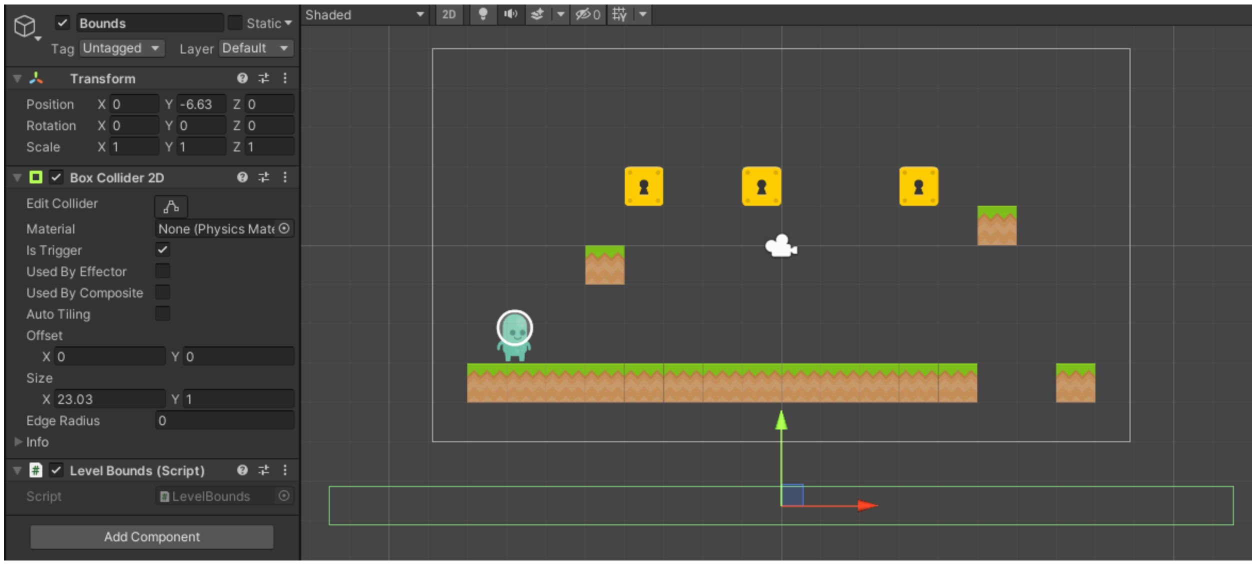
9) Create and Program Level Boundary

Add a BoxCollider2D to an empty object. Position the object below the Level Layout and use Edit Collider to ensure the green collider zone encompasses any area where the player might fall.

Do NOT use Scale to size the collider.

Check the Is Trigger option. This means this collision zone is not solid.

Create a new C# Script called “LevelBounds” (no spaces) and attach it to the game object.

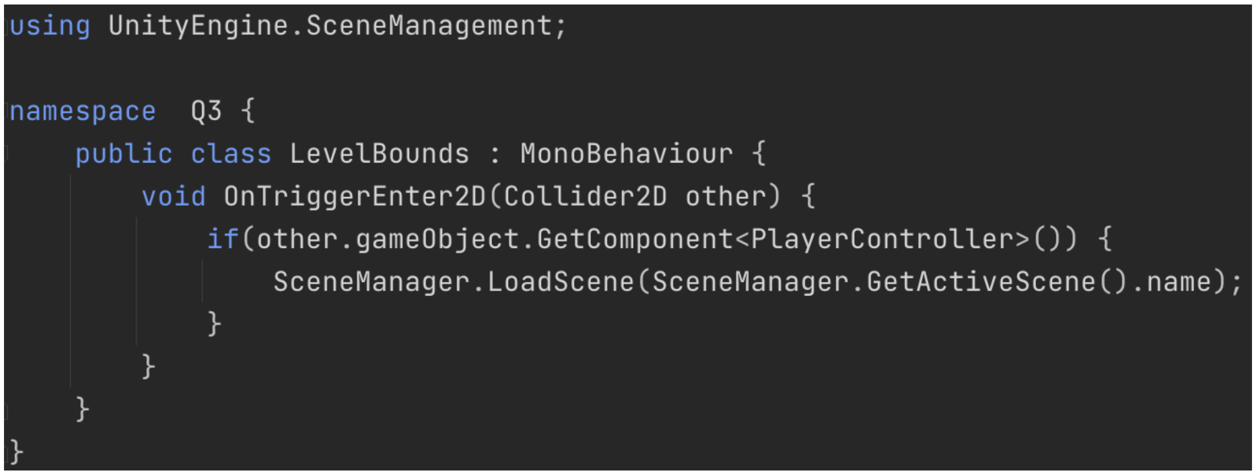


```
Bounds                         Static
Tag Untagged        Layer Default
Transform
Position   X 0     Y -6.63   Z 0
Rotation   X 0     Y 0       Z 0
Scale      X 1     Y 1       Z 1
Box Collider 2D
Edit Collider
Material            None (Physics Mate
Is Trigger          ✓
Used By Effector
Used By Composite
Auto Tiling
Offset
  X 0        Y 0
Size
  X 23.03    Y 1
Edge Radius         0
Info
Level Bounds (Script)
Script              LevelBounds
Add Component
```

In your script, add the SceneManagement namespace to give us easier access to the SceneManager functions.

We will use a Trigger2D event instead of a Collision2D because our green zone is a non-solid Trigger.

You may have a different scene name than “SampleScene.” You must also make sure your Scene is added to Unity’s File->Build Settings.



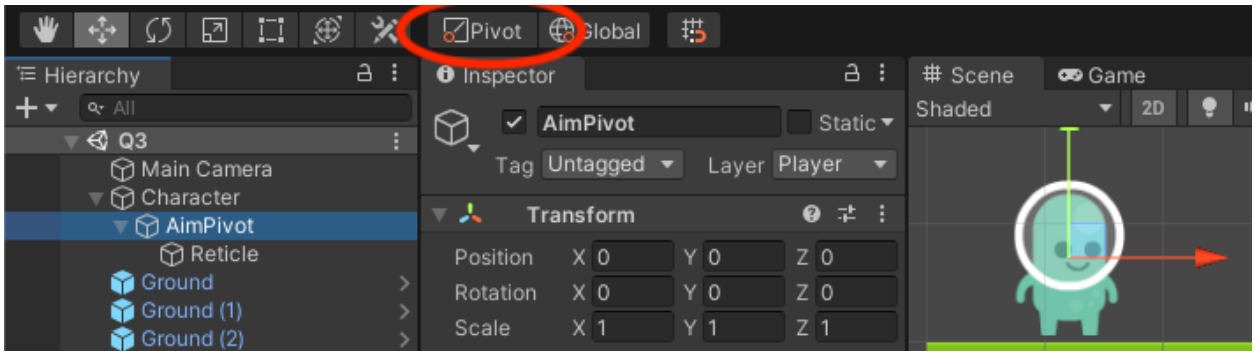
```
using UnityEngine.SceneManagement;

namespace  Q3 {
    public class LevelBounds : MonoBehaviour {
        void OnTriggerEnter2D(Collider2D other) {
            if(other.gameObject.GetComponent<PlayerController>()) {
                SceneManager.LoadScene(SceneManager.GetActiveScene().name);
            }
        }
    }
}
```

10) Create Aiming Reticle

Create an Empty Child Game Object within the Character and name it AimPivot. Not additional Components are necessary. Ensure that your Move Tool is set to Pivot mode. Notice how the Pivot Point of AimPivot is centered within Character.

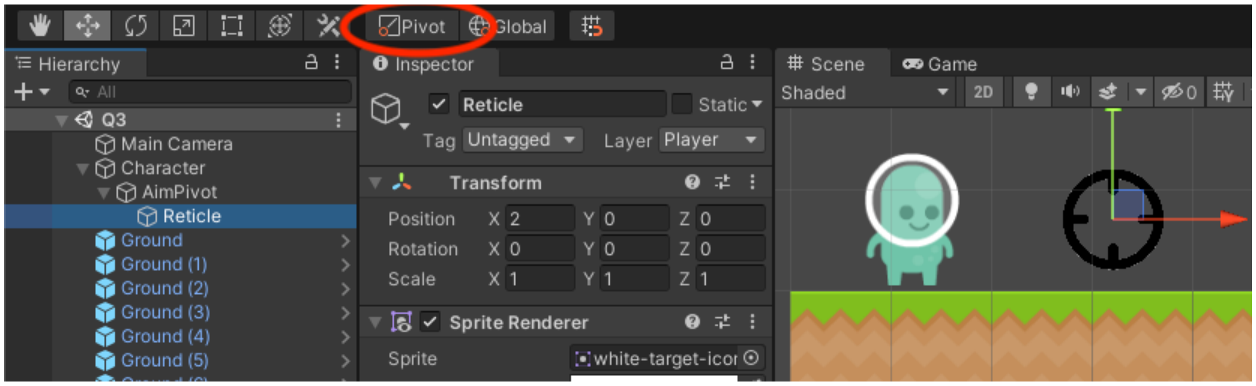
AimPivot is the gameObject we will rotate to aim with the mouse.



AimPivot Static
Tag Untagged Layer Player
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Create an Empty Child Game Object within AimPivot and name it Reticle. Add a SpriteRenderer component and assign an appropriate graphic. You may need to adjust the Pixels Per Unit to get the graphic the correct size.

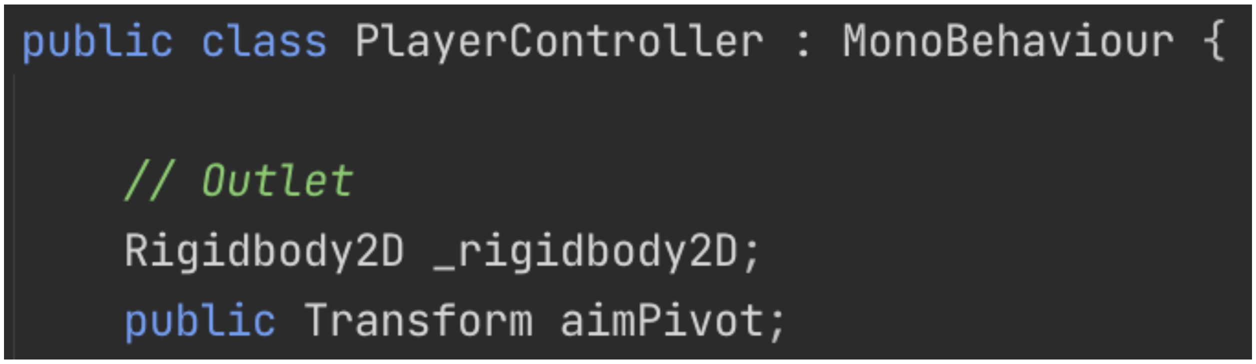
Notice how the Pivot Point of Reticle is positioned outside of the character.



Reticle Static
Tag Untagged Layer Player
Transform
Position X 2 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Sprite Renderer
Sprite white-target-icor

11) Program Aiming Reticle

In your PlayerController script, add a new Public Outlet for aimPivot, so that we can reference in code and rotate it towards the mouse.

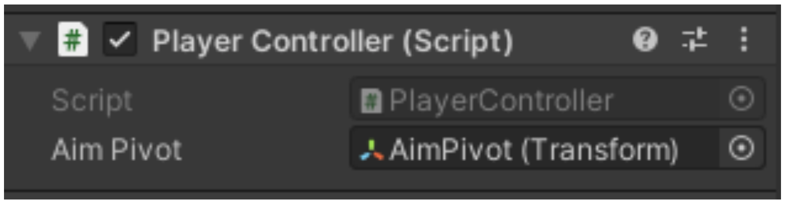


```
public class PlayerController : MonoBehaviour {

    // Outlet
    Rigidbody2D _rigidbody2D;
    public Transform aimPivot;
```

Because this is a Public Outlet, we will get a Fill-in-the-Blank field in the Inspector for our Character gameObject where we can tell Unity what Transform component we want to use as aimPivot.

Assign AimPivot to this blank by dragging or through the Selection Window circle.



Player Controller (Script)

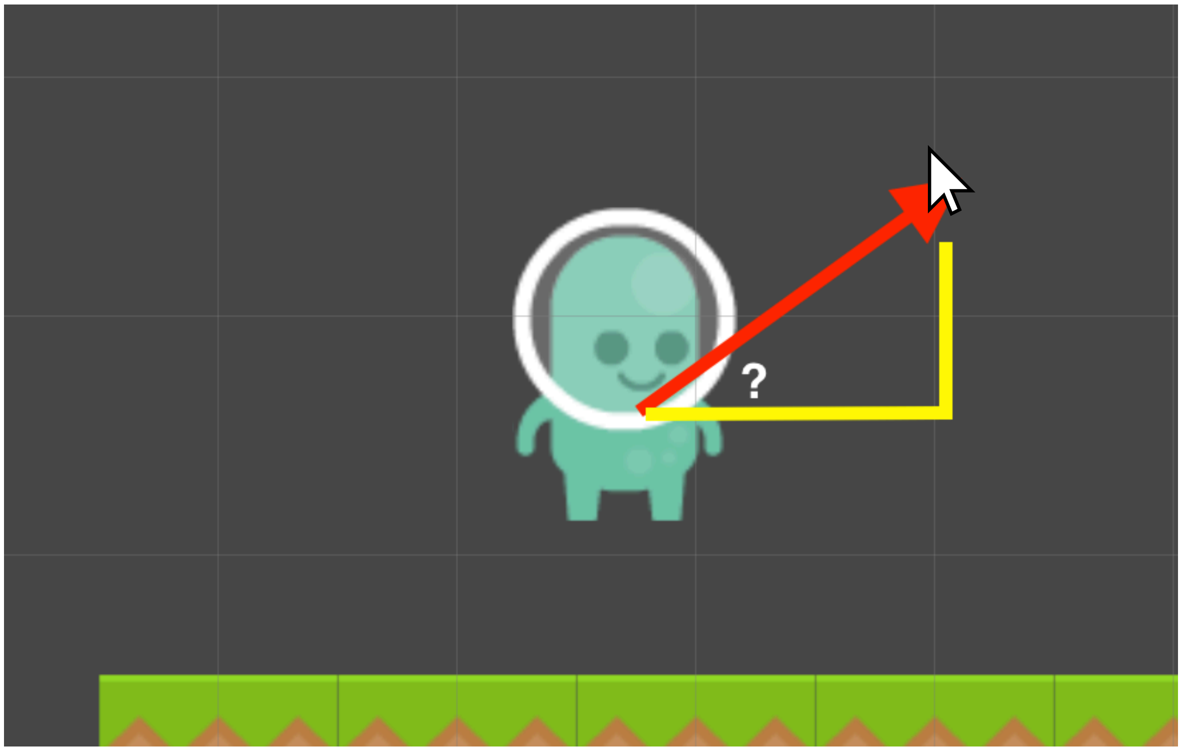
Script	PlayerController
Aim Pivot	AimPivot (Transform)

We will add Mouse Aiming code to the Update event of our PlayerController script on our Character gameObject.

1) First we get the Mouse Position. This is in Screen Space which is the mouse's position on the computer's flat screen.
2) We convert this Screen Space coordinate to World Space which is a position inside the 3D game world. This is a Camera Function since we are interpreting a flat screen position from the 3D perspective of a given camera.
3) Our goal is to get the Angle between the Character and the Mouse, so our technique will be to establish a triangle and use Trigonometry to solve for the angle.

   Trigonometry treats the positive X-axis as 0 degrees and degrees rotate in the counter-clockwise direction.

   Subtracting our character's position from the mouse's world position gives us the hypotenuse of a right triangle.



4) We can use Arc Tangent to derive the angle from the hypotenuse, but this is in radians.

5) We convert radians to degrees to get our final angle value and assign it as the rotation for the pivot.

```
void Update() {
    // Move Player Left
    if(Input.GetKey(KeyCode.A)) {
        _rigidbody2D.AddForce(Vector2.left * 12f * Time.deltaTime, ForceMode2D.Impulse);
    }

    // Move Player Left
    if(Input.GetKey(KeyCode.D)) {
        _rigidbody2D.AddForce(Vector2.right * 12f * Time.deltaTime, ForceMode2D.Impulse);
    }

    // Aim Toward Mouse
    Vector3 mousePosition = Input.mousePosition;
    Vector3 mousePositionInWorld = Camera.main.ScreenToWorldPoint(mousePosition);
    Vector3 directionFromPlayerToMouse = mousePositionInWorld - transform.position;

    float radiansToMouse = Mathf.Atan2(directionFromPlayerToMouse.y, directionFromPlayerToMouse.x);
    float angleToMouse = radiansToMouse * Mathf.Rad2Deg;

    aimPivot.rotation = Quaternion.Euler(0, 0, angleToMouse);
}
```

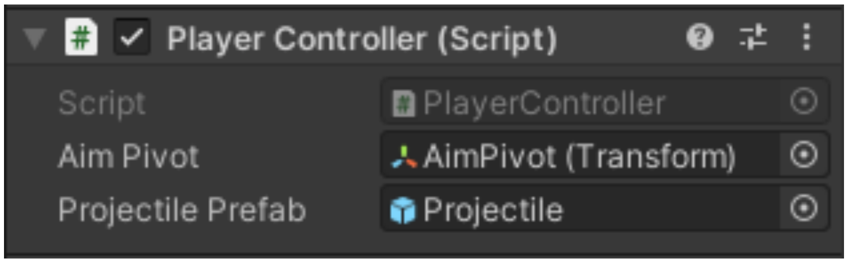
12) Program Shooting

We need to add a Public Outlet for the Prefab gameObject our character will shoot as a Projectile.

```
public class PlayerController : MonoBehaviour {

    // Outlet
    Rigidbody2D _rigidbody2D;
    public Transform aimPivot;
    public GameObject projectilePrefab;
```

This creates a Fill-in-the-Blank in the character gameObject’s Inspector. Assign our Projectile Prefab from the Project library to this blank.


Player Controller (Script)
Script PlayerController
Aim Pivot AimPivot (Transform)
Projectile Prefab Projectile

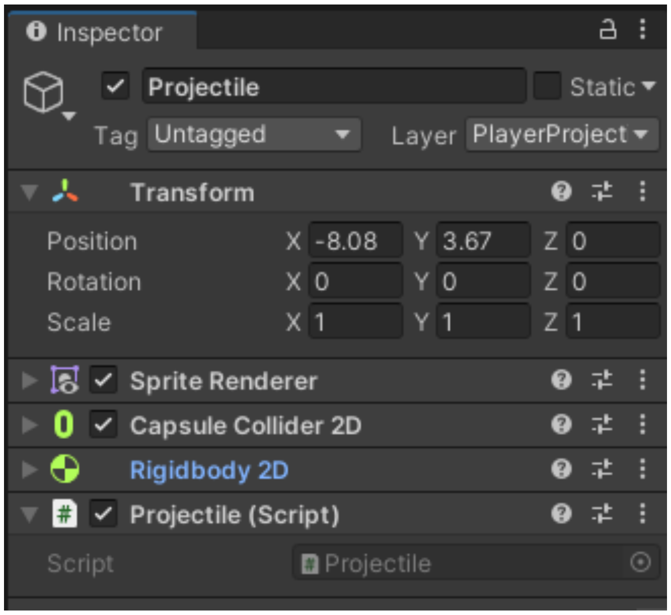
Beneath the code for our Mouse Aiming within the Update Event of our PlayerController, we will add code for shooting projectiles.

```
// Shoot
if(Input.GetMouseButtonDown(0)) {
    GameObject newProjectile = Instantiate(projectilePrefab);
    newProjectile.transform.position = transform.position;
    newProjectile.transform.rotation = aimPivot.rotation;
}
```

When the MouseButtonDown (not MouseButton "held") event of the left-click occurs, we create a new instance of our projectile prefab, position it where our character is, and rotate it to match our aimPivot. Right now the projectile just floats in mid-air.

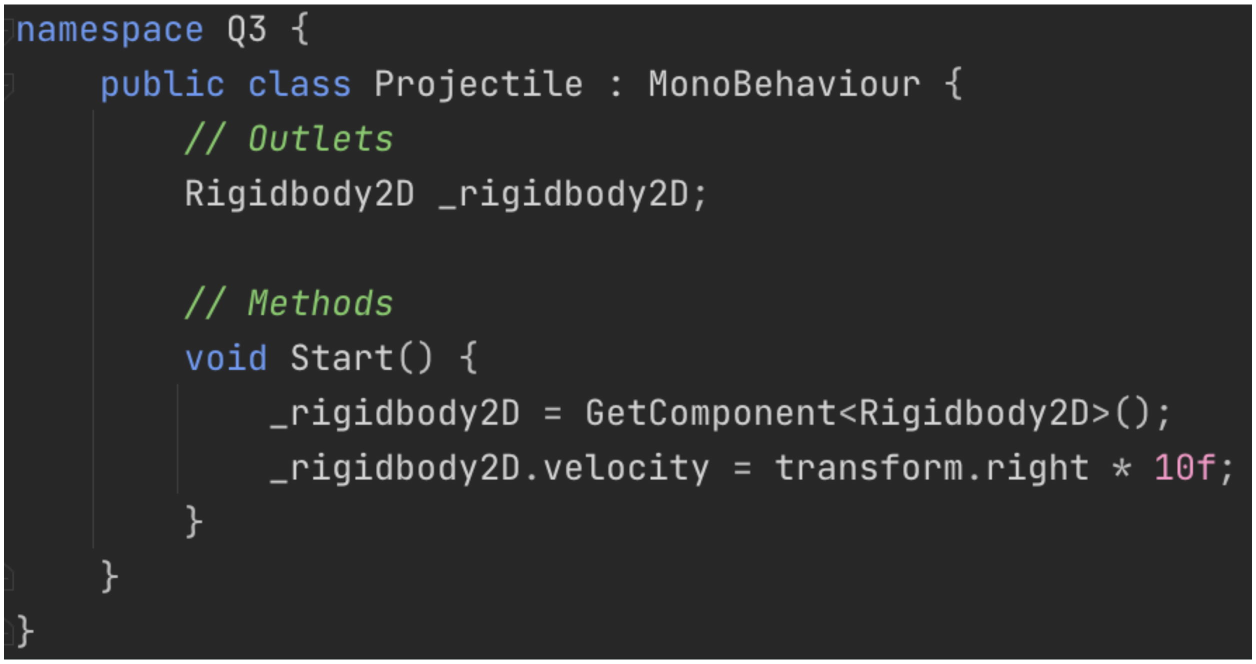
13) Program Projectile

Create a new C# script called Projectile and add it to your Projectile Prefab.


Inspector
Projectile Static
Tag Untagged Layer PlayerProject
Transform
Position X -8.08 Y 3.67 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Sprite Renderer
Capsule Collider 2D
Rigidbody 2D
Projectile (Script)
Script Projectile

We will control our Projectile movement with physics, so we must create an Outlet for Rigidbody2D to reference it in code.

At the Start event of our component, we will set the relative velocity of our projectile toward the Right. Because we are rotating the projectile through angles relative to the positive x-axis, our projectile will appear to travel in whatever direction we aim. You may want a different speed than 10.

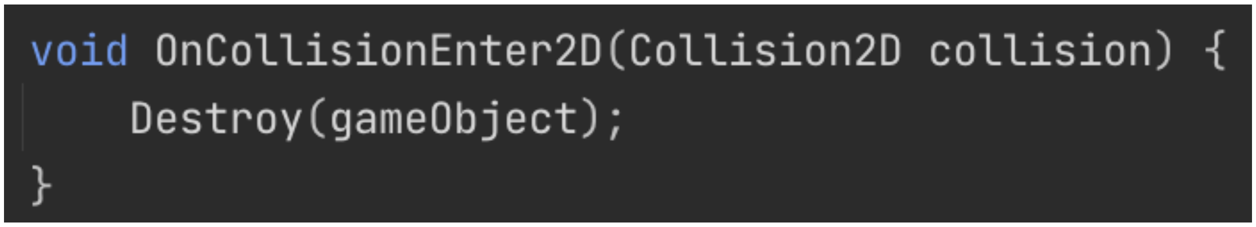


```
namespace Q3 {
    public class Projectile : MonoBehaviour {
        // Outlets
        Rigidbody2D _rigidbody2D;

        // Methods
        void Start() {
            _rigidbody2D = GetComponent<Rigidbody2D>();
            _rigidbody2D.velocity = transform.right * 10f;
        }
    }
}
```

We also want our projectile to disappear any time it his another object. We do not have to worry about the projectile hitting the player when it’s fired because we setup a collision matrix in the Physics2D settings.

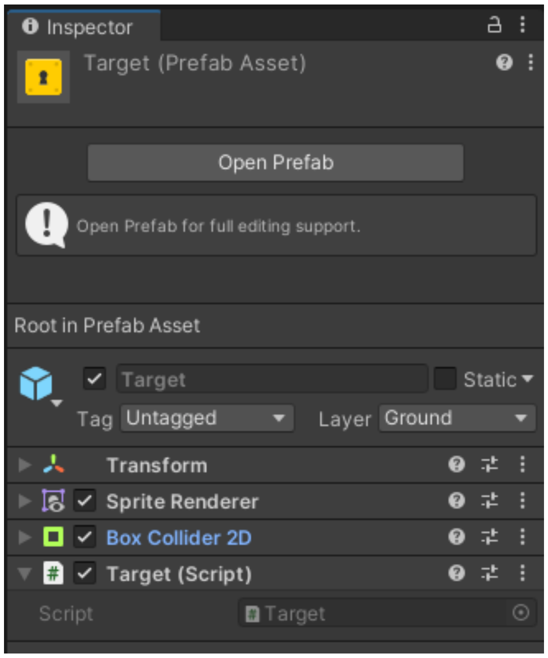
We setup an OnCollisionEnter2D event where we tell our projectile to remove itself from the game.



```
void OnCollisionEnter2D(Collision2D collision) {
    Destroy(gameObject);
}
```

14) Program Target Prefab

Create a new C# script named Target and assign it to the Target Prefab in your Project library to ensure that all Target instances obtain the new script.


Inspector

Target (Prefab Asset)

Open Prefab

Open Prefab for full editing support.

Root in Prefab Asset

Target | Static

Tag: Untagged | Layer: Ground

Transform

Sprite Renderer

Box Collider 2D

Target (Script)

Script: Target

We will use another OnCollisionEnter2D event, but we will specifically check if the colliding object has a Projectile component before destroying our Target block. We do this to make sure touching the Player does not mis-trigger this game mechanic.

```
namespace Q3 {
    public class Target : MonoBehaviour {
        void OnCollisionEnter2D(Collision2D other) {
            if(other.gameObject.GetComponent<Projectile>()) {
                Destroy(gameObject);
            }
        }
    }
}
```

15) Program Jump

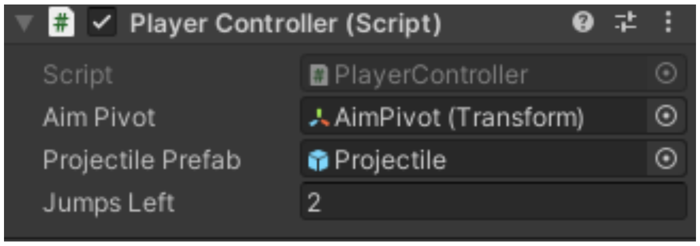
In our PlayerController, we can make jump count configurable by creating a Public Property to keep track of how many jumps we have left.

```
public class PlayerController : MonoBehaviour {

    // Outlet
    Rigidbody2D _rigidbody2D;
    public Transform aimPivot;
    public GameObject projectilePrefab;

    // State Tracking
    public int jumpsLeft;
```

This Public Property creates a Fill-in-the-Blank in the Character's Inspector. For Double Jump, set the Jumps Left to 2.



Player Controller (Script)

Script	PlayerController
Aim Pivot	AimPivot (Transform)
Projectile Prefab	Projectile
Jumps Left	2

Below our shooting code in PlayerController, we are going to add code for Jumping.

We check to see if SpaceBar has just been pressed (not held) and if we have any jumps left in the property we were using as a counter.

When our character jumps, we reduce the number of jumps left and add an upward physics force. You may want a force strength different than 10 depending on your Game Feel.

We use a ForceMode2D of Impulse so that the full jump force is applied instantaneously.

Right now, our player will jump twice and then become stuck to the ground.

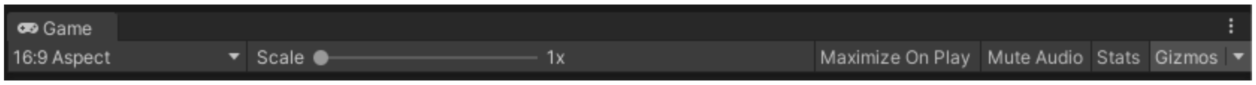
```
// Jump
if(Input.GetKeyDown(KeyCode.Space)) {
    if(jumpsLeft > 0) {
        jumpsLeft--;
        _rigidbody2D.AddForce(Vector2.up * 10f, ForceMode2D.Impulse);
    }
}
```

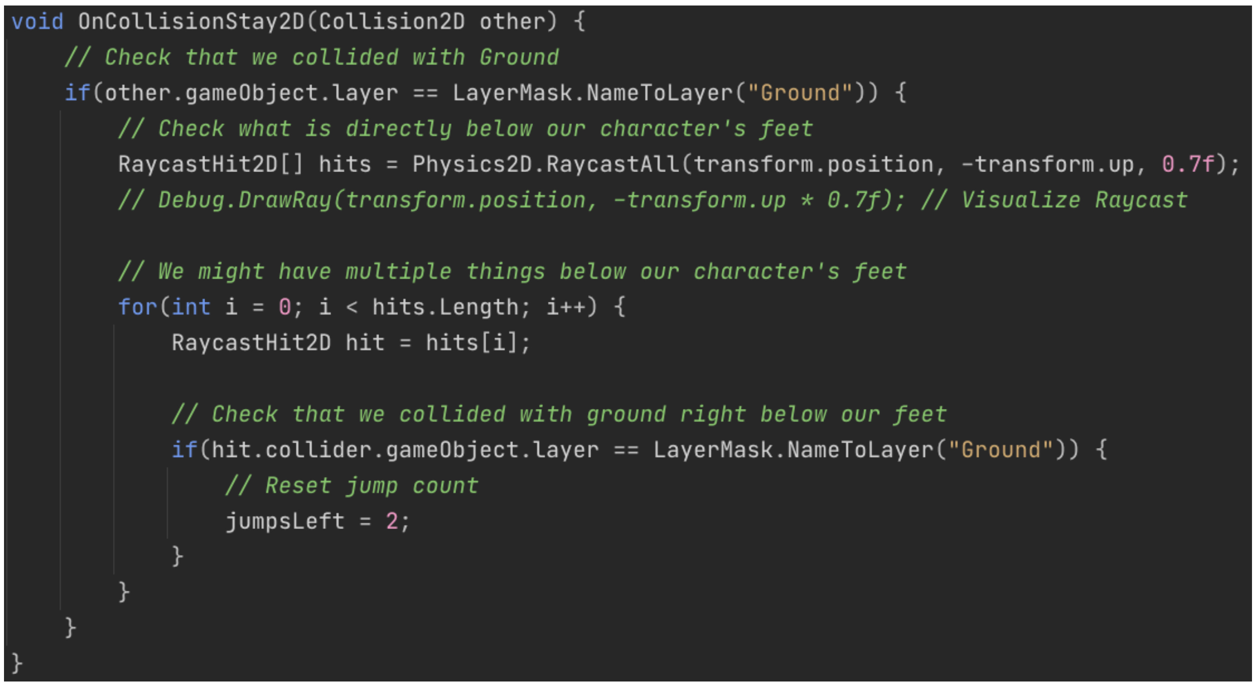
16) Program Double Jump

For a proper Double Jump, we must reset the jump counter when the player solidly lands on ground.

INCORRECT implementations involve resetting the jump counter if the player touches the side of the ground (Wall Jumping) or if the player touches their head to the ground (Ceiling Climbing).

1) We set up an OnCollisionEnter2D event since the only time we reset our Jumps Left is when we collide with something.
2) Specifically we only want to reset Jumps Left if we collide with the ground, so we check that the colliding object is part of the “Ground” layer next.
3) Last, we must verify that it is our feet touching the ground, and not the side or top of our character. We use a technique called Raycasting to see what objects are directly below our character.
4) If one of those objects beneath our feet is also part of the “Ground” layer, then we can be certain it is a valid time to reset Jumps Left to 2.
5) This example has a line of Debug code you can uncomment to visualize the Raycast. Be sure to turn on “Gizmos” during gameplay.





```
void OnCollisionStay2D(Collision2D other) {
    // Check that we collided with Ground
    if(other.gameObject.layer == LayerMask.NameToLayer("Ground")) {
        // Check what is directly below our character's feet
        RaycastHit2D[] hits = Physics2D.RaycastAll(transform.position, -transform.up, 0.7f);
        // Debug.DrawRay(transform.position, -transform.up * 0.7f); // Visualize Raycast

        // We might have multiple things below our character's feet
        for(int i = 0; i < hits.Length; i++) {
            RaycastHit2D hit = hits[i];

            // Check that we collided with ground right below our feet
            if(hit.collider.gameObject.layer == LayerMask.NameToLayer("Ground")) {
                // Reset jump count
                jumpsLeft = 2;
            }
        }
    }
}
```