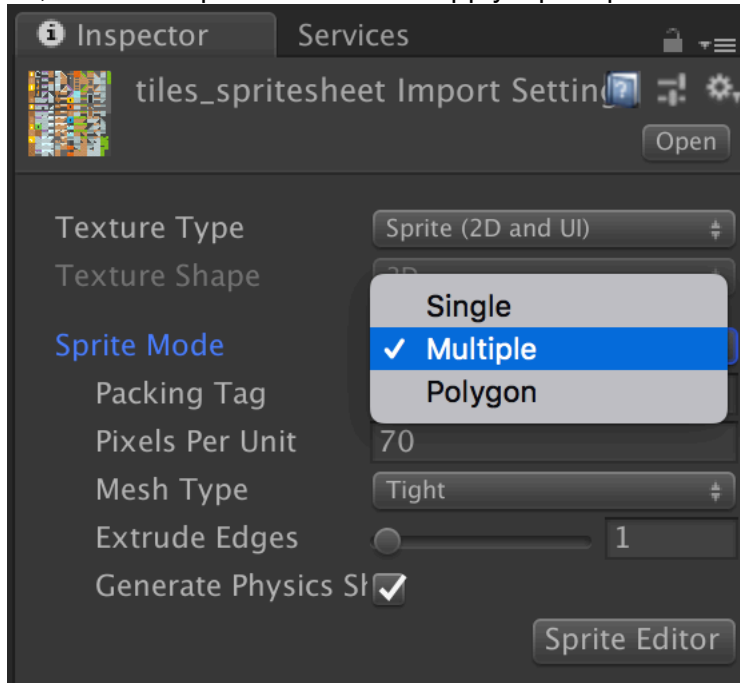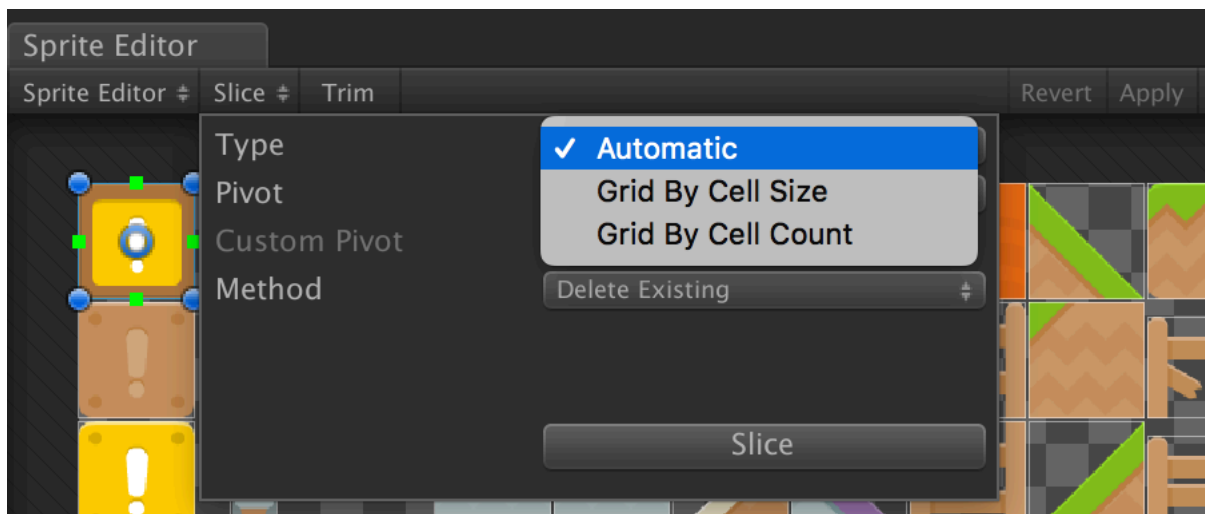# Quest 3 - Platformer - Steps

1) Import and Slice Graphics

Import quest course files. Set the Sprite Sheets' Import Settings to Multiple, Pixels Per Unit to 70, and click Sprite Editor. Click Apply if prompted.

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

Repeat these steps for any other graphics that are Multi-Sprite Sheets instead of single sprite files.
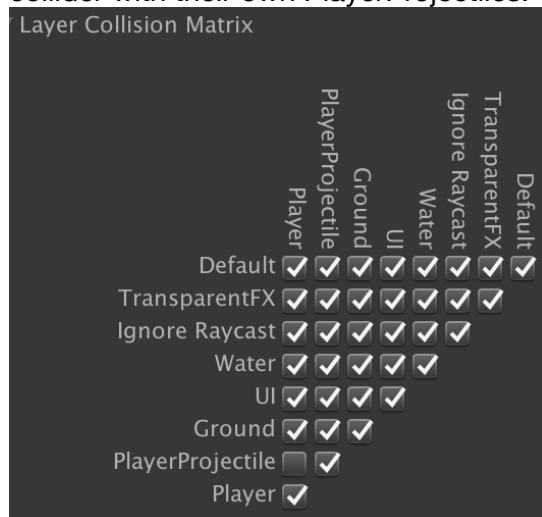
2) Setup Layers and Collisions

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."

From Edit->Project Settings->Physics2D, alter the collision matrix, so that Players cannot collider with their own PlayerProjectiles.
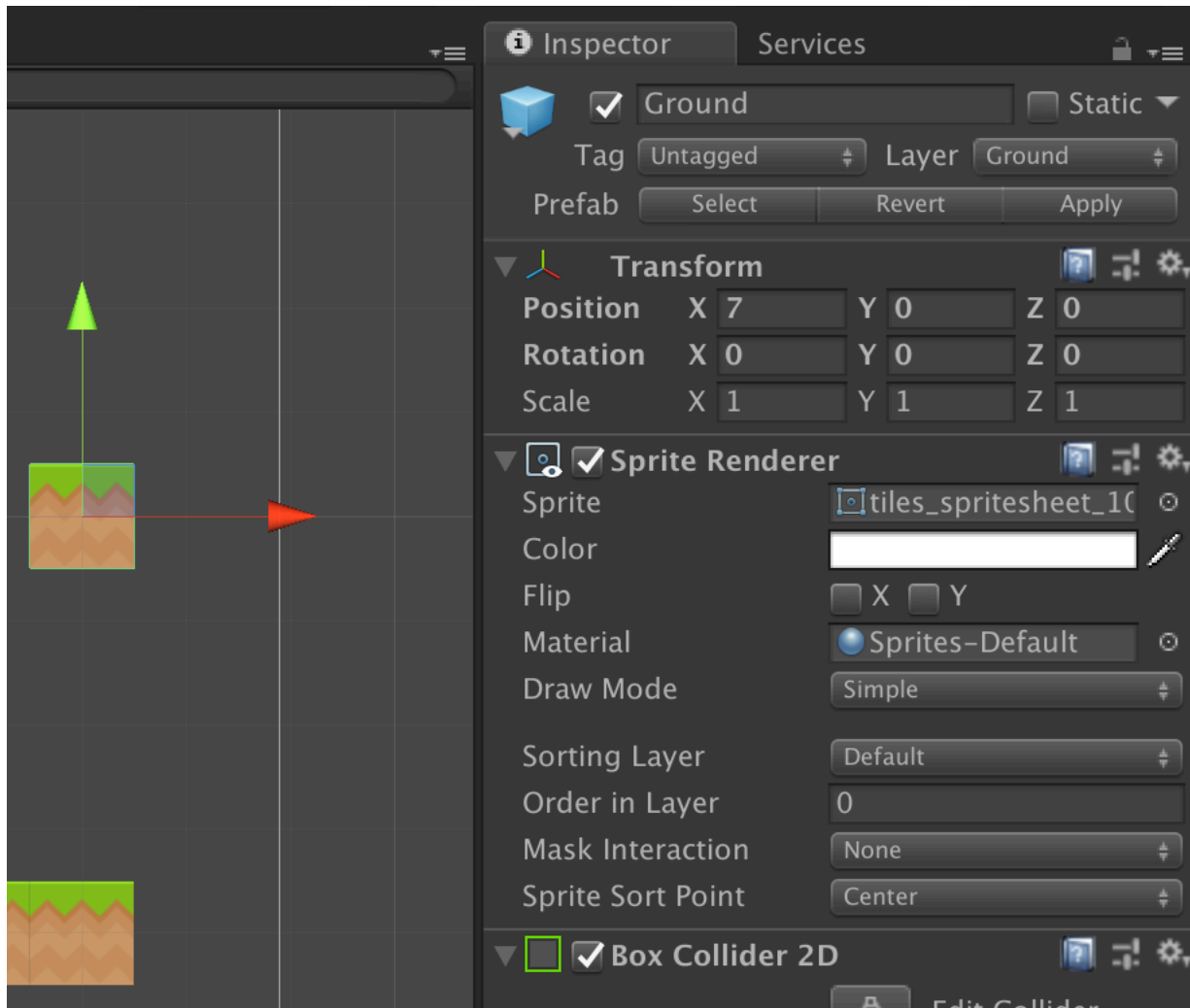
3) Create Ground Prefab

Add a Sprite Renderer component to an Empty Object. Assign a desired Ground Sprite. Add a BoxCollider2D component.
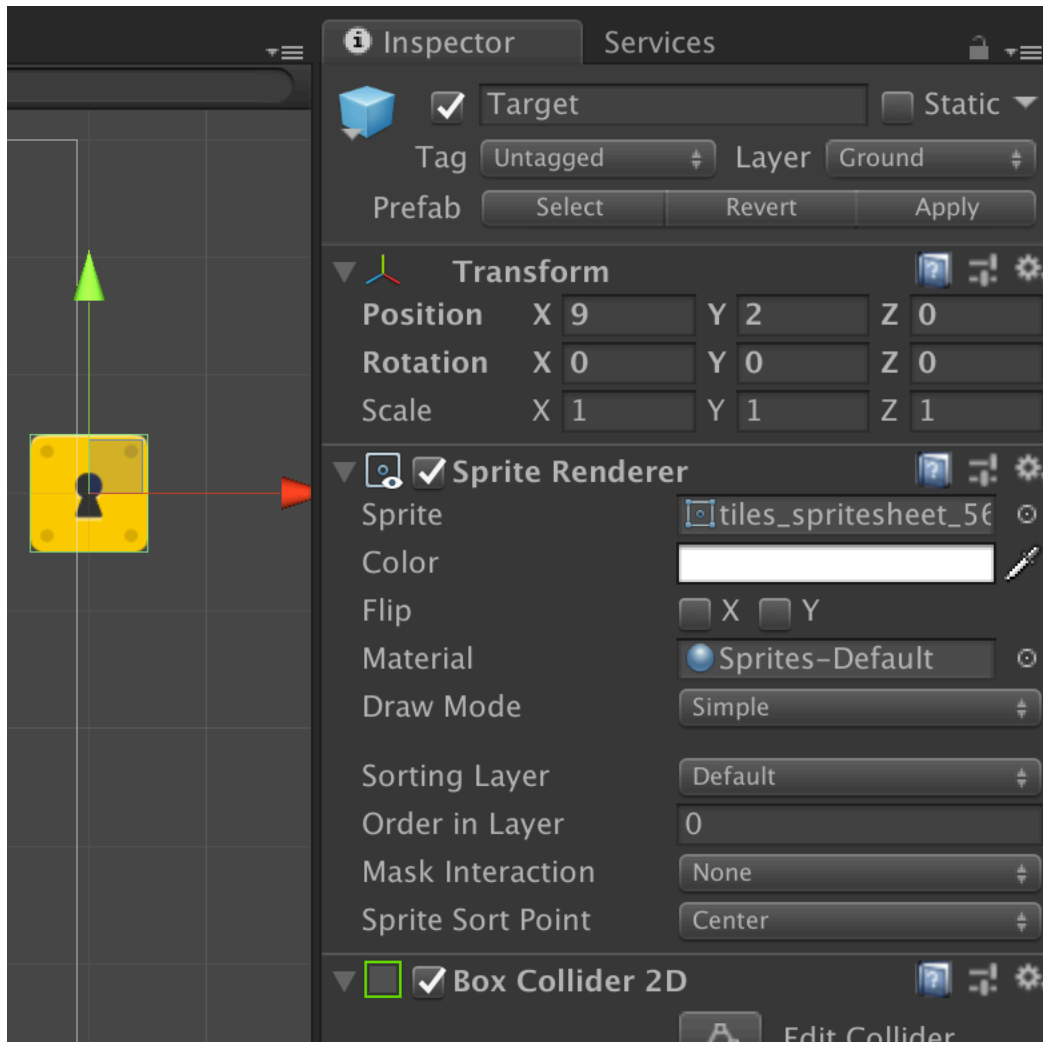
Assign a Layer of "Ground."

Drag the Ground gameObject from the Hierarchy list to the Project library to create a reusable ground Prefab. Notice the Select, Revert, and Apply buttons for updating other copies of this Prefab.

4) Create Target Prefab

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

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

5) Create Character

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

Assign a Layer of "Player."

Add a CircleCollider2D or a CapsuleCollider2D component. Any Collider2D will work as long as it 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 .

6) Create Projectile

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

Assign a Layer of "PlayerProjectile."

Add a CircleCollider2D component or other appropriately-shaped Collider2D component.

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.
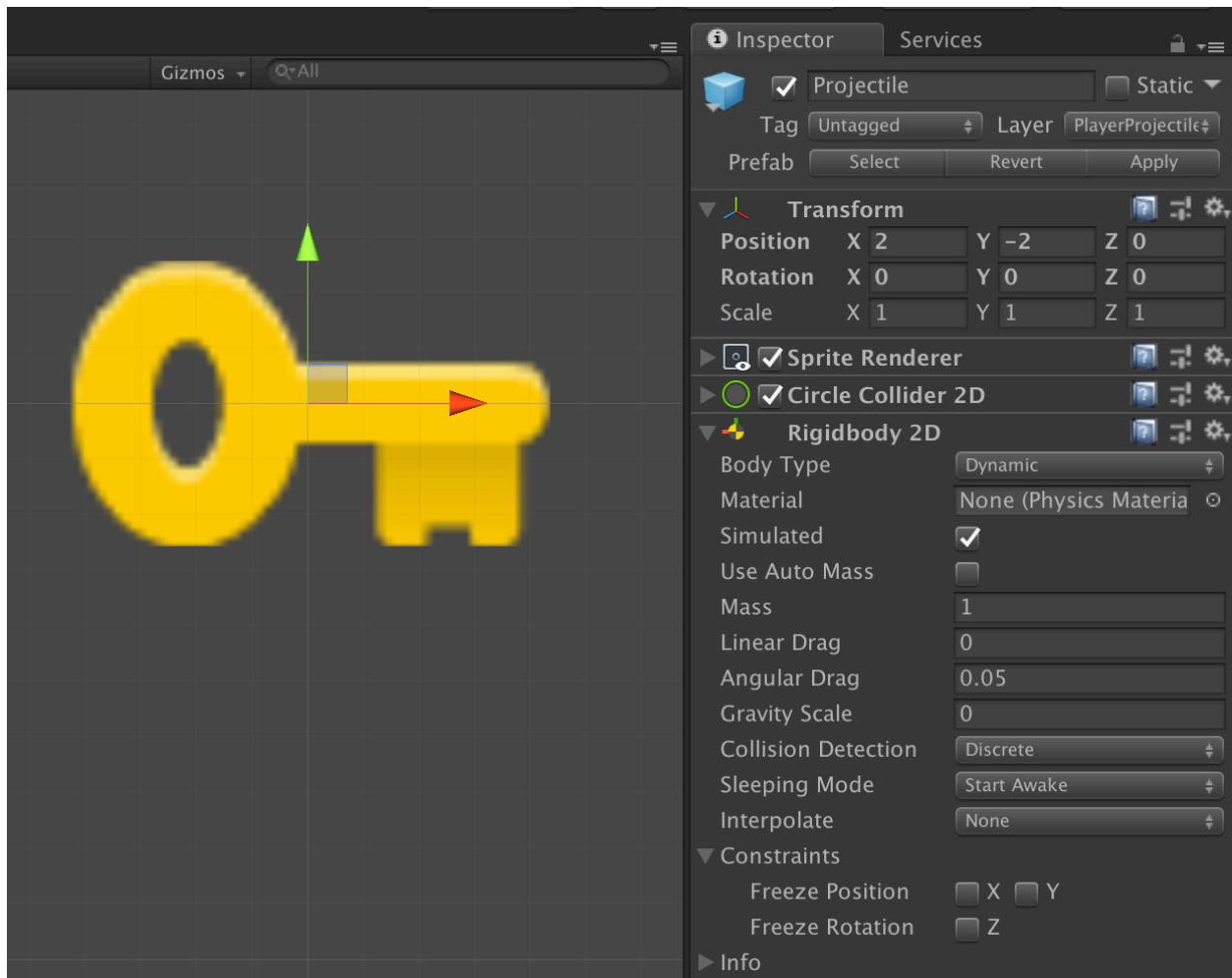
7) Create a Level Layout

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.
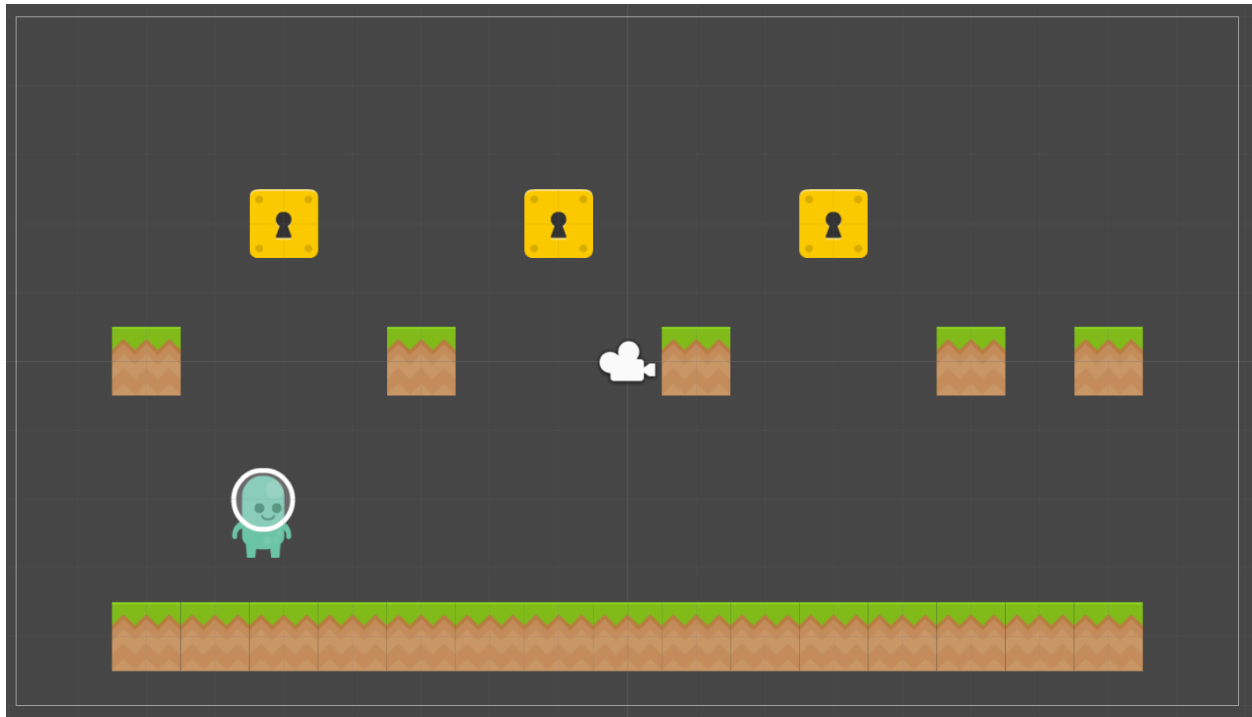
8) Program Character Movement

Create a new C# Script called "PlayerController" and attach it to the Character gameObject.

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

```csharp
public class PlayerController : MonoBehaviour {

    // Outlet
    Rigidbody2D rigidbody;

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

    void Update() {
```

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 than 12.

```
void Update() {
    // Move Player Left
    if(Input.GetKey(KeyCode.A)) {
        rigidbody.AddForce(Vector2.left * 12f);
    }

    // Move Player Right
    if(Input.GetKey(KeyCode.D)) {
        rigidbody.AddForce(Vector2.right * 12f);
    }
```

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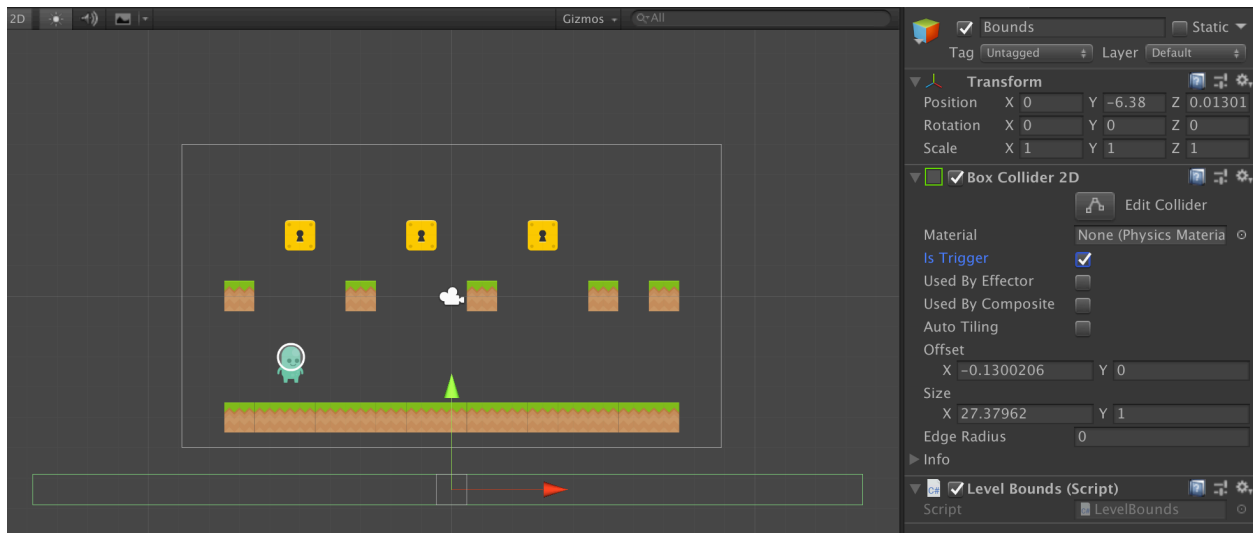
In your script, add the SceneManagement namespace to give us easier access to the SceneManager functions.

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.

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

10) Create Aiming Reticule

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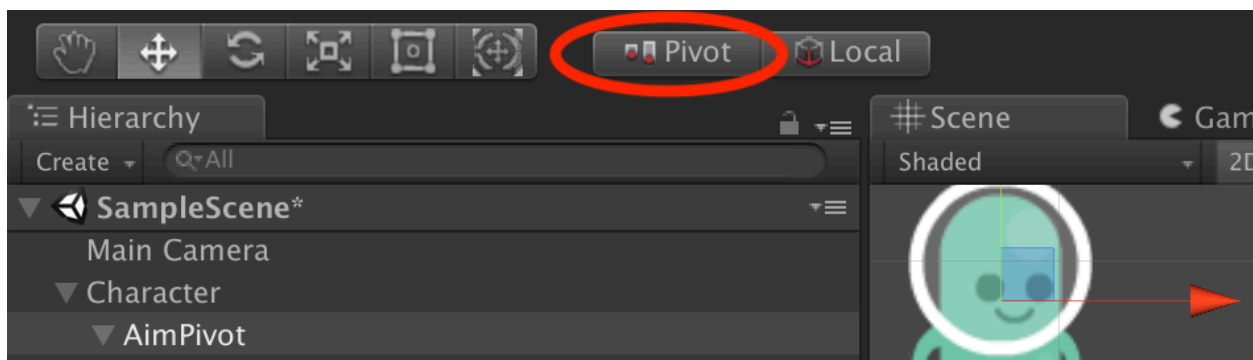
Create an Empty Child Game Object within AimPivot and name it Reticule. Add a SpriteRenderer component and assign an appropriate graphic. You may need to adjust the Pixels Per Unit to get the graphic the correct size.

Notice how the Pivot Point of Reticule is positioned outside of the character.
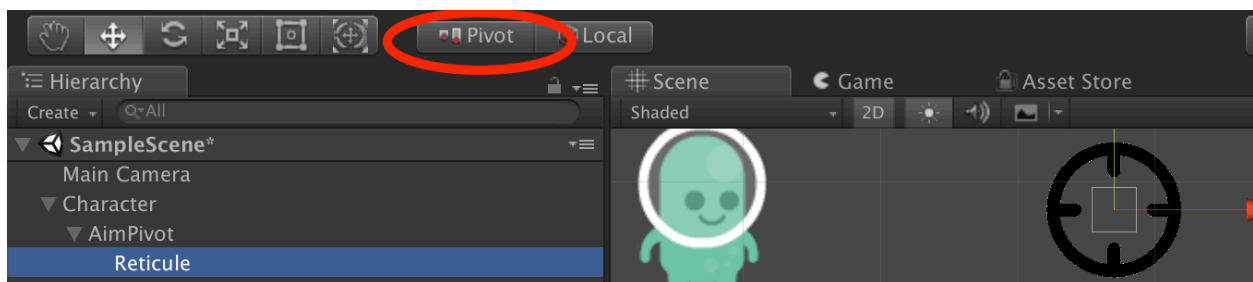
11) Program Aiming Reticule

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 :

    // Outlet
    Rigidbody2D rigidbody;
    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.
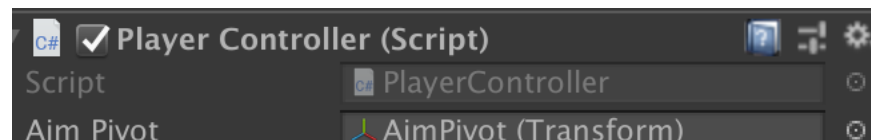

We will add Mouse Aiming code to the Update event of our PlayerController script on our Character gameObject.

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.
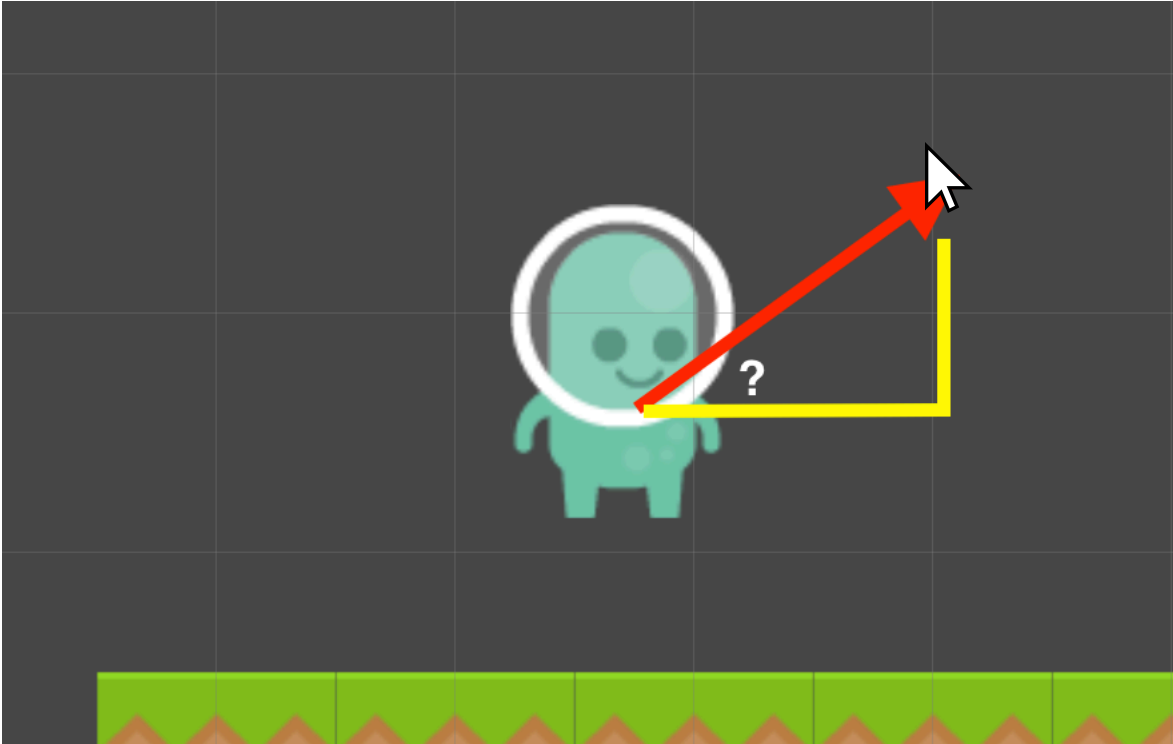
4) We can use Arc Tangent to derive the angle from the hypotenuse, but this is in radians.
5) We convert radians to degrees to get our final angle value and assign it as the rotation for the pivot.

```
void Update() {
    // Move Player Left
    if(Input.GetKey(KeyCode.A)) {
        rigidbody.AddForce(Vector2.left * 12f);
    }

    // Move Player Right
    if(Input.GetKey(KeyCode.D)) {
        rigidbody.AddForce(Vector2.right * 12f);
    }

    // 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 * 180f / Mathf.PI;

    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 : MonoBehavi

    // Outlet
    Rigidbody2D rigidbody;
    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.
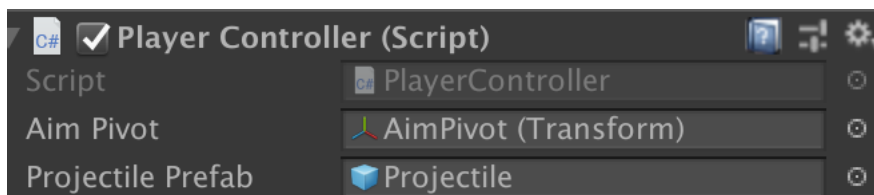
Beneath the code for our Mouse Aiming within the Update Event of our PlayerController, we will add code for shooting projectiles.

```
// Aim Toward Mouse
Vector3 mousePosition = Input.mousePosition;
Vector3 mousePositionInWorld = Camera.main.ScreenToWorldPoint(
Vector3 directionFromPlayerToMouse = mousePositionInWorld - tr

float radiansToMouse = Mathf.Atan2(directionFromPlayerToMouse.
float angleToMouse = radiansToMouse * 180f / Mathf.PI;

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

// Shoot
if(Input.GetMouseButtonDown(0)) {
    GameObject newProjectile = Instantiate(projectilePrefab);
    newProjectile.transform.position = transform.position;
    newProjectile.transform.rotation = aimPivot.rotation;
}
```
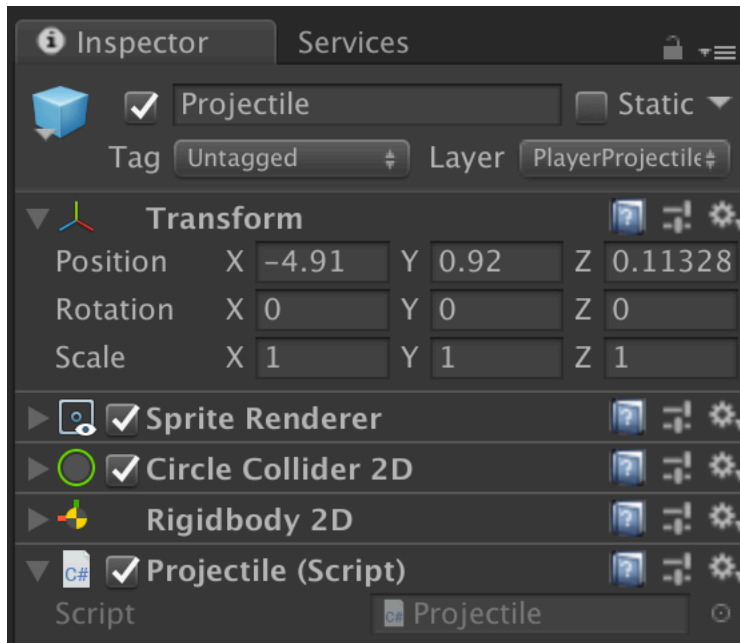
When the MouseButtonDown (not MouseButton "held") event of the left-click occurs, we create a new instance of our projectile prefab, position it where our character is, and rotate it to match our aimPivot. Right now the projectile just floats in mid-air.

13) Program Projectile

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

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.

```csharp
lic class Projectile : MonoBehaviour {

    // Outlets
    Rigidbody2D rigidbody;

    // Methods
    void Start() {
        rigidbody = GetComponent<Rigidbody2D>();
        rigidbody.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.
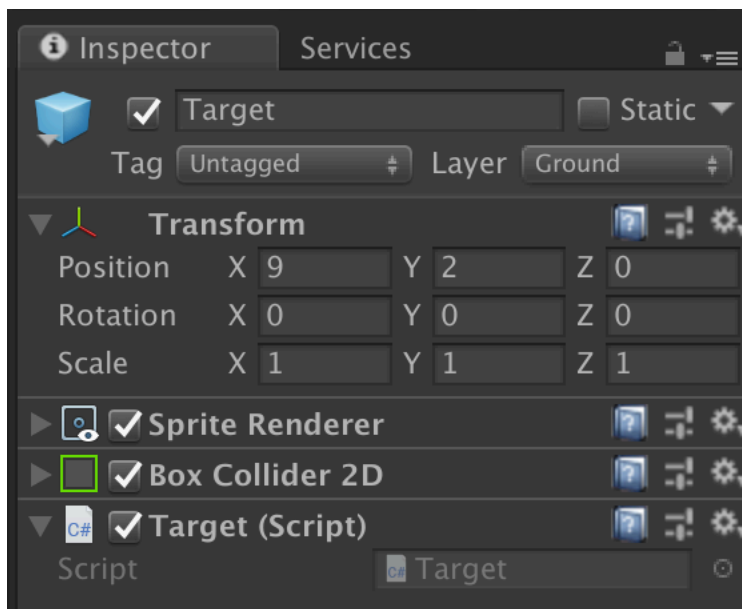
We will use another OnCollisionEnter2D event, but we will specifically check if the colliding object has a Projectile component before destroying our Target block. We do this to make sure touching the Player does not mis-trigger this game mechanic.

```
public class Target : MonoBehaviour {

    void OnCollisionEnter2D(Collision2D collision) {
        if(collision.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.

```
lic class PlayerController : MonoBehaviour {

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


    // State Tracking
    public int jumpsLeft;


    // Methods
```

This Public Property creates a Fill-in-the-Blank in the Character's Inspector. For Double Jump, set the Jumps Left to 2.
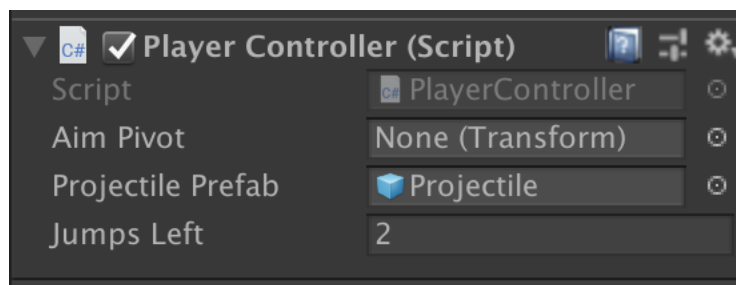
Below our shooting code in PlayerController, we are going to add code for Jumping.

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.

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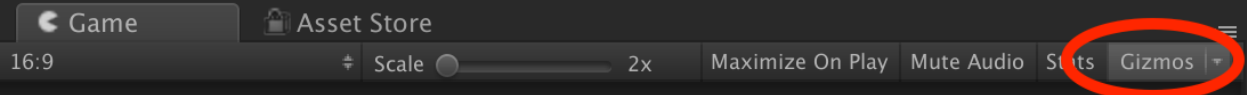
16) Program Double Jump

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.


```
// Collision Event
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;
            }
        }
    }
}
```