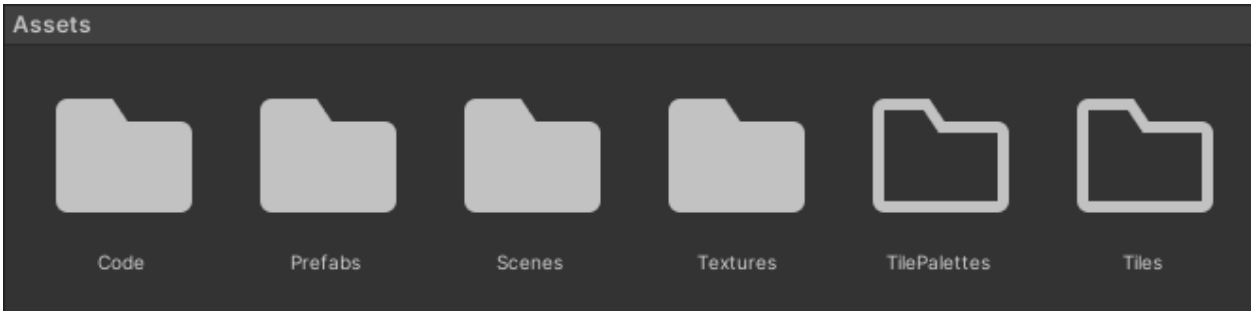
Quest 4 - Steps

1) Pre-organize for TileMap Assets

Creating TileMap graphics in Unity is built on a hierarchy of several asset types:

Grid object can hold any number of TileMap layers.
TileMap object must have a Tilemap Renderer. It might have a Tilemap Collider 2D.
TilePalette is a collection of reusable Tiles that is used to paint a TileMap.
Tiles are reusable blocks of configurable graphics/collision placed throughout the TileMap.
Sprites are the graphics for Tiles (and SpriteRenderers). Also have a collision configuration.

Grids and TileMaps live in the Scene Hierarchy. TilePalettes, Tiles, and Sprites are kept in the Project library.

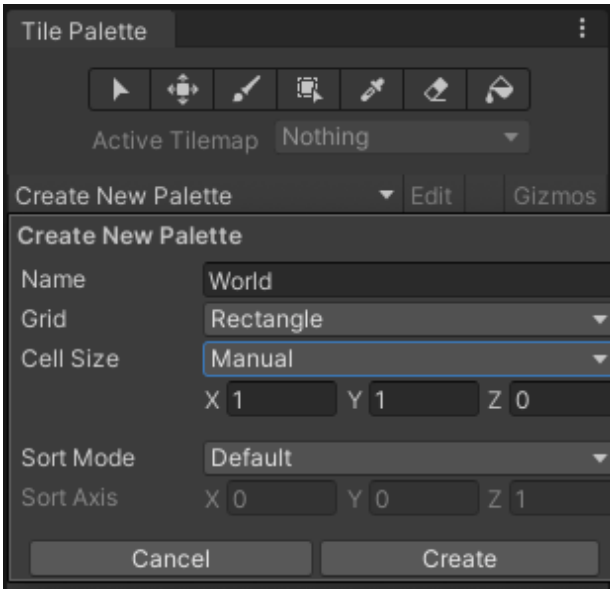


To keep our project organized, we'll prepare a TilePalettes and Tiles folder. We already have a Textures folder for our Sprites.

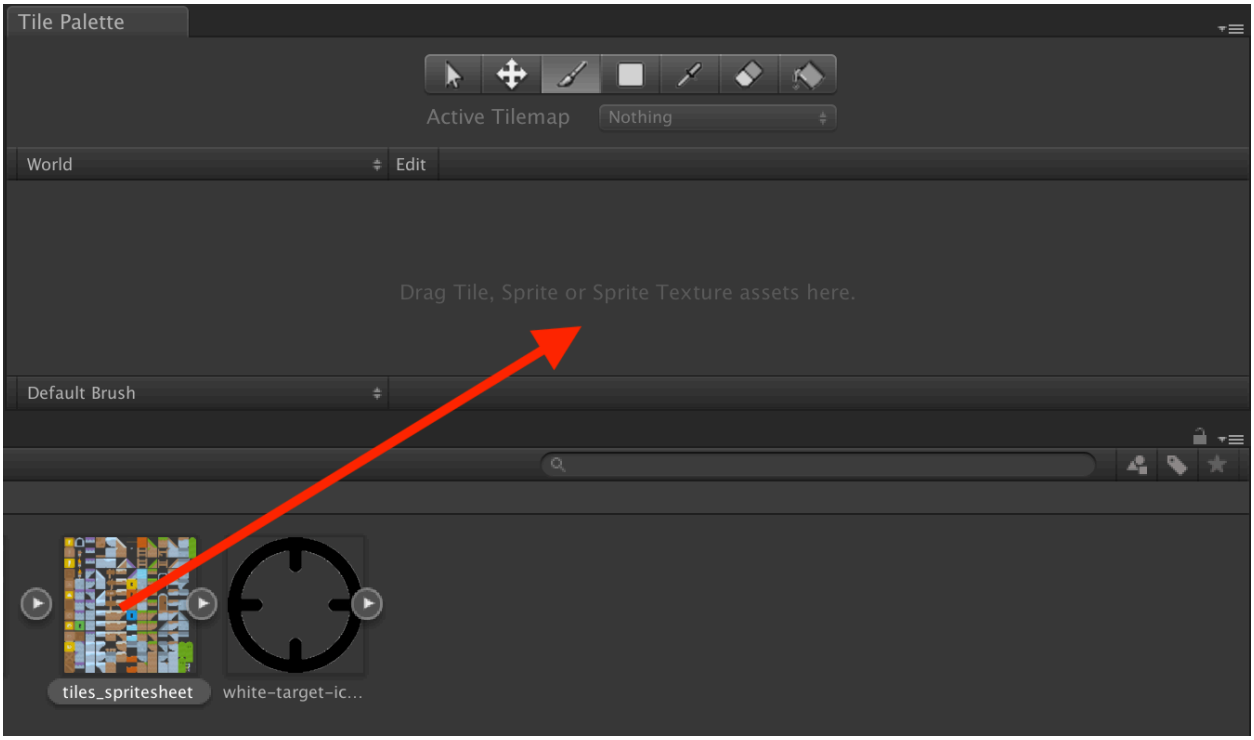
2) Create TilePalette

Before we can paint on a TileMap, we have to create a TilePalette. As a convenience, the Tile Palette tool can create Tiles from Sprites. You might have different Palettes for different collections of graphics. For example, some games might have a Water, Fire, and Castle palette for different game zones.

Open the Tile Palette tool from Window->2D->Tile Palette. Create a new Tile Palette with Manual settings. Save your new Palette in the TilePalette folder.

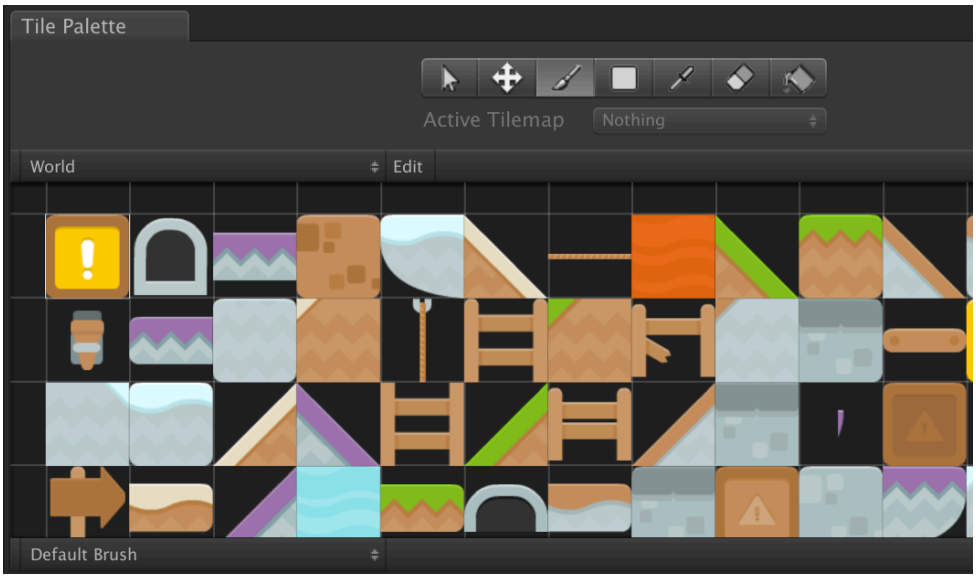


Select the tiles_spritesheet from Textures and drag it into TilePalette drop zone to create Tiles from those sprites. Create an Assets/Tiles/World folder to save these Tiles.

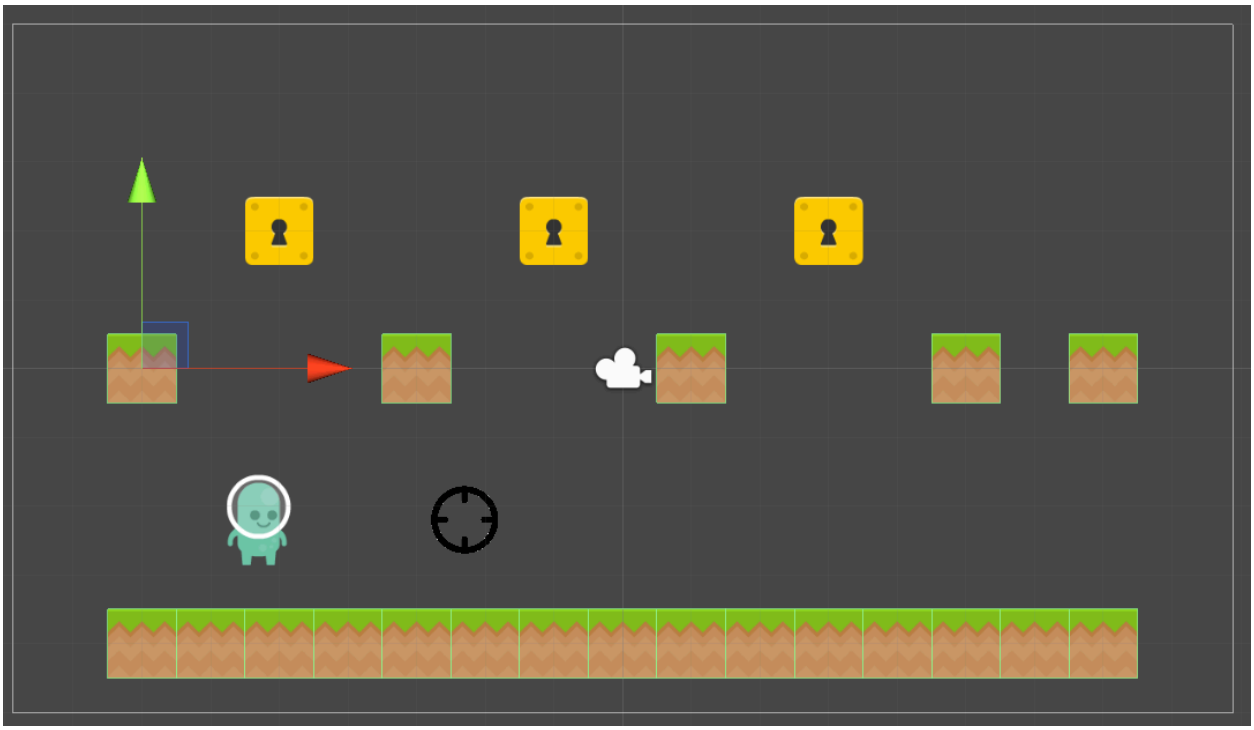


3) Fix TilePalette/Tiles Import Issues (if any)

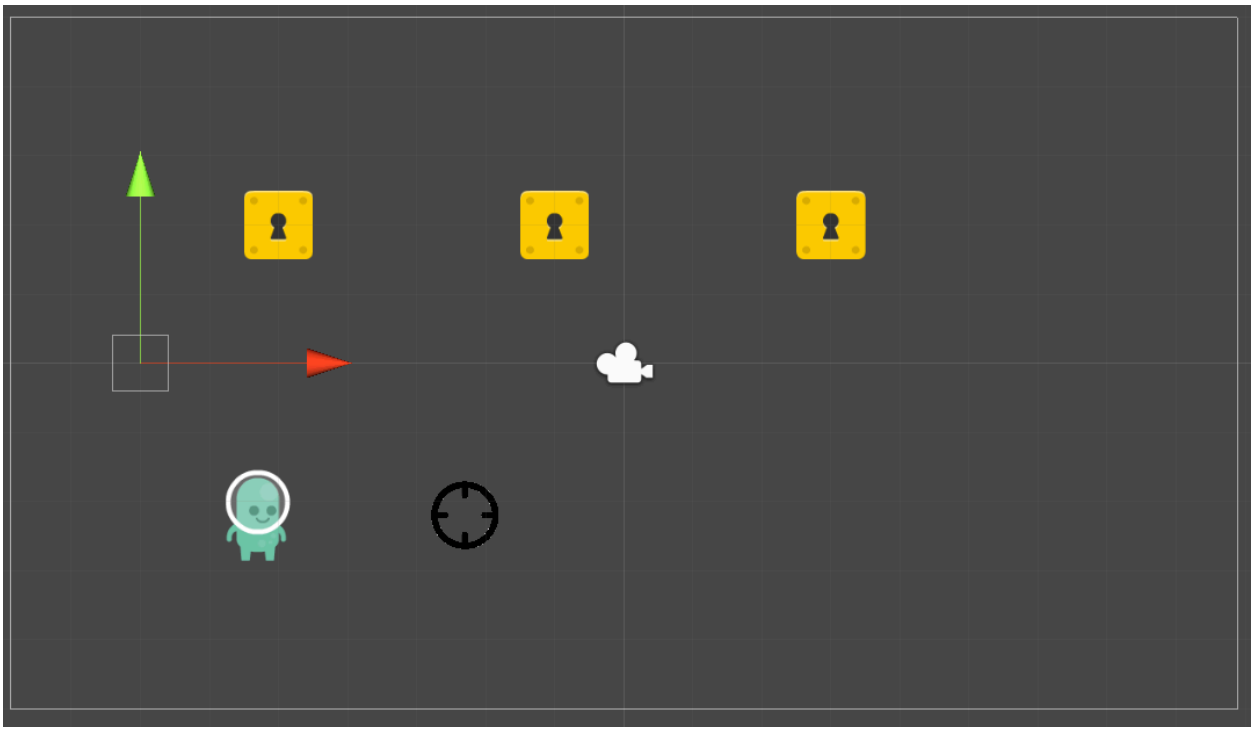
If your TilePalette looks like a mess, you might have forgotten to set your Palette sizing to Manual. You can delete the Palette from the TilePalette folder and the Tiles from the Tiles/World folder to try again.



4) Delete old terrain

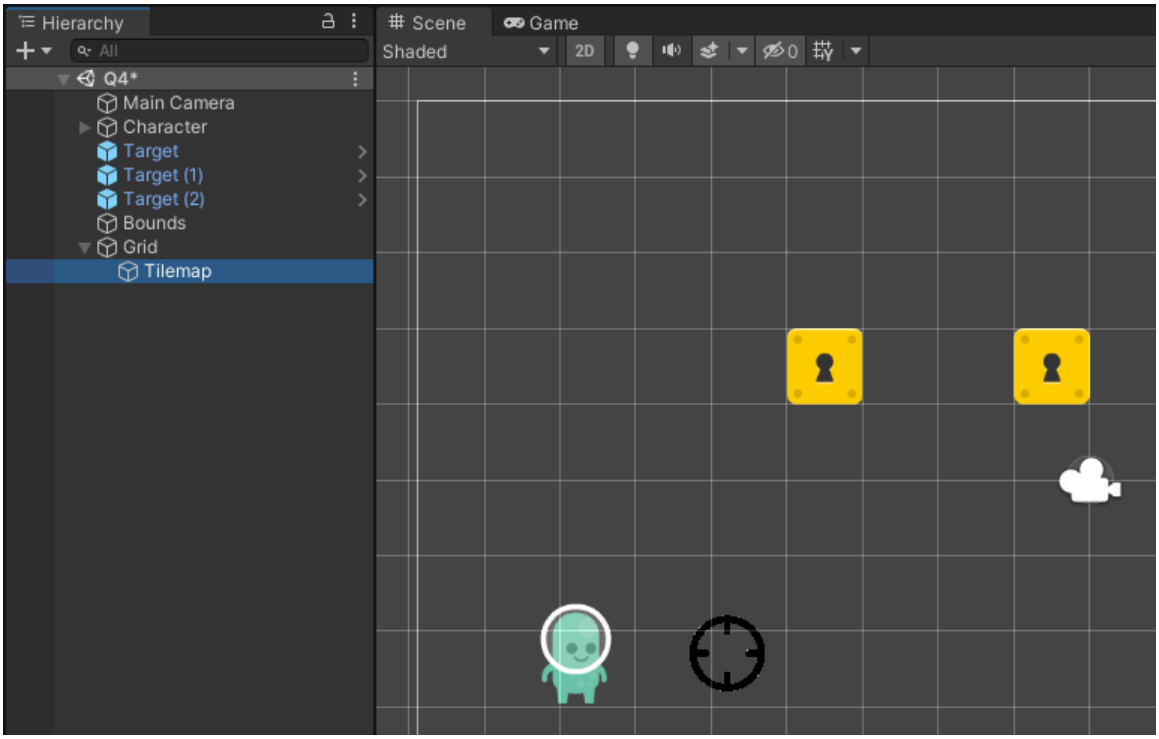


We are going to replace our old Q2 terrain with a TileMap. You should still leave the Target blocks as normal objects. Delete your old Terrain objects.



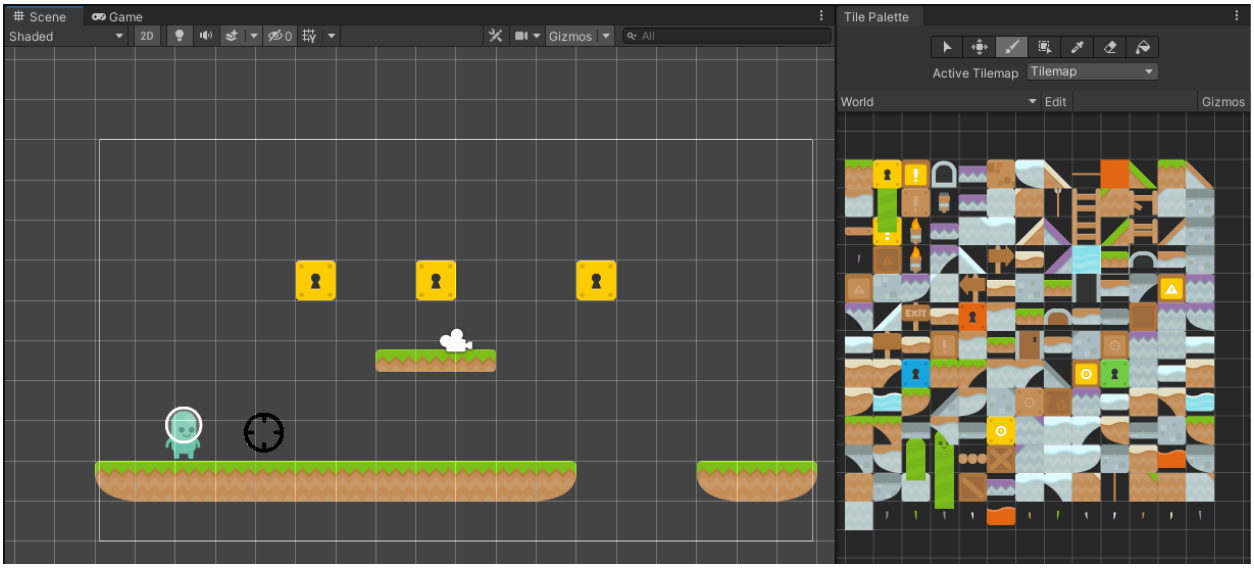
5) Create new TileMap

Create a TileMap from the Hierarchy by clicking Create->2D Object->Tilemap->Rectangular. Notice how it creates two nested objects. A parent Grid and a Child Tilemap. You can create additional Tilemaps inside the Grid for additional graphical layers.



6) Brush Tiles

Using the Tile Palette window in conjunction with the Scene window, you can use the Brush tool to paint different Tiles onto your scene.

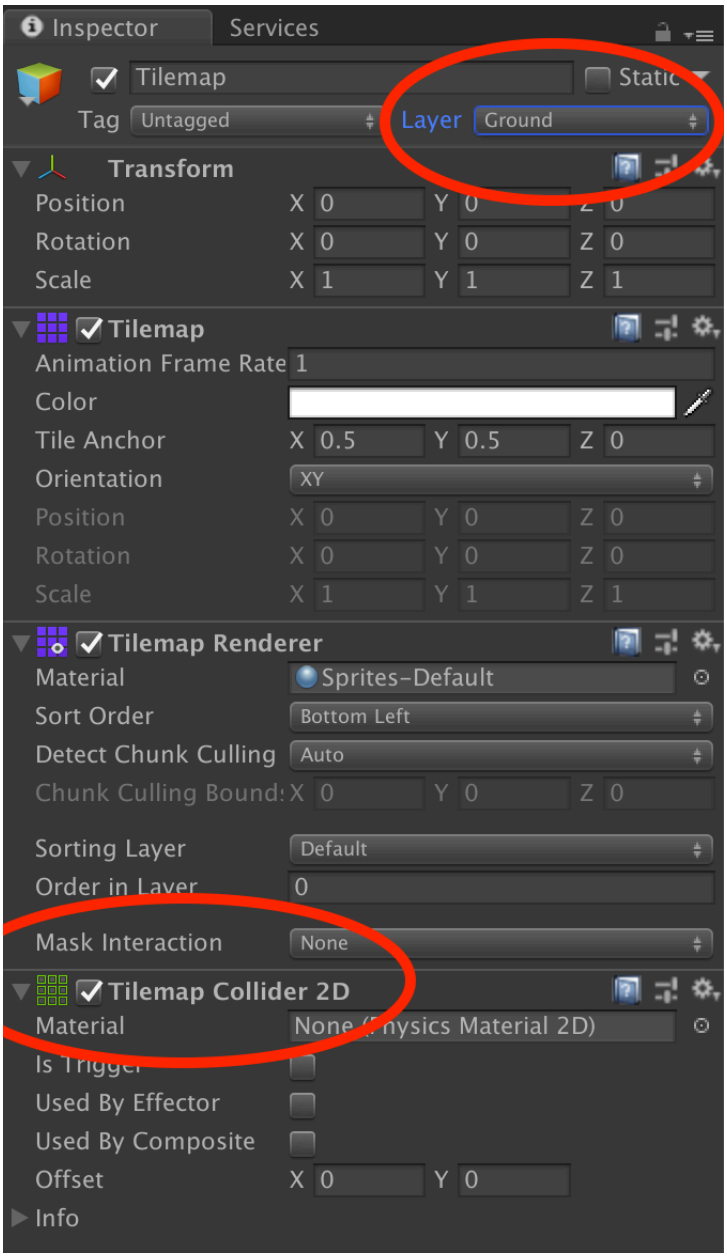


7) Setup Collisions

If you play your game right now, you'll notice that your player just falls through the landscape. Just like a SpriteRenderer object, you also need to add a Collider to make it solid.

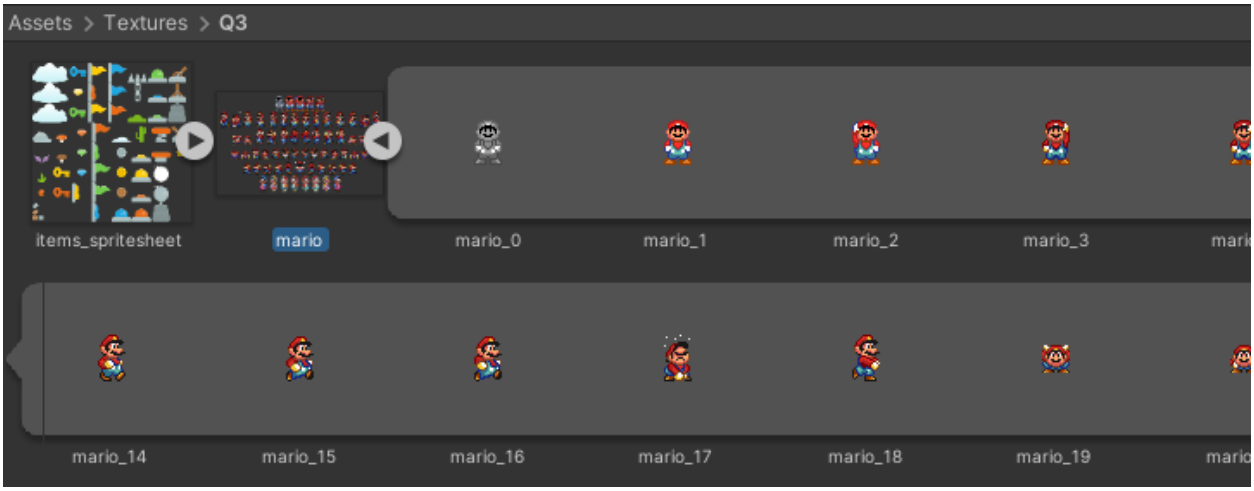
The correct collider for a TileMap is a Tilemap Collider 2D.

Also notice we change the Layer to Ground because our Jump mechanics check for Ground collisions.



8) Import New Character Graphics

Download mario.gif which is the Mario character sprite sheet from the Canvas Files. Set your Import Settings to Sprite Mode = Multiple, Pixels Per Unit = 16, Filter Mode = Point, Compression = None. Use the Sprite Editor to do an Automatic Slice.



9) Switch to New Character Graphics

As the default graphic, change your Character's Sprite to #13. Reset your Capsule Collider to match the new graphics.

Because our Character is now taller, we have to increase how far below the character we Raycast looking for Ground to reset our Double Jump mechanic.

```
// Check what is directly below our character's feet
RaycastHit2D[] hits = Physics2D.RaycastAll(transform.position, -transform.up, 0.85f);
//Debug.DrawRay(transform.position, -transform.up * 0.85f); // Visualize Raycast
```

10) Left vs. Right Graphics

Having a player face left or right can be handled a few different ways. For simplicity, we are going to use the Flip X setting to mirror our Character Sprite when the player pushes A or D.

```
public class PlayerController : MonoBehaviour {

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

    // State Tracking
    public int jumpsLeft;

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

    void Update() {
        // Move Player Left
        if(Input.GetKey(KeyCode.A)) {
            _rigidbody2D.AddForce(Vector2.left * 12f * Time
            sprite.flipX = true;
        }

        // Move Player Right
        if(Input.GetKey(KeyCode.D)) {
            _rigidbody2D.AddForce(Vector2.right * 12f * Tim
            sprite.flipX = false;
        }
```

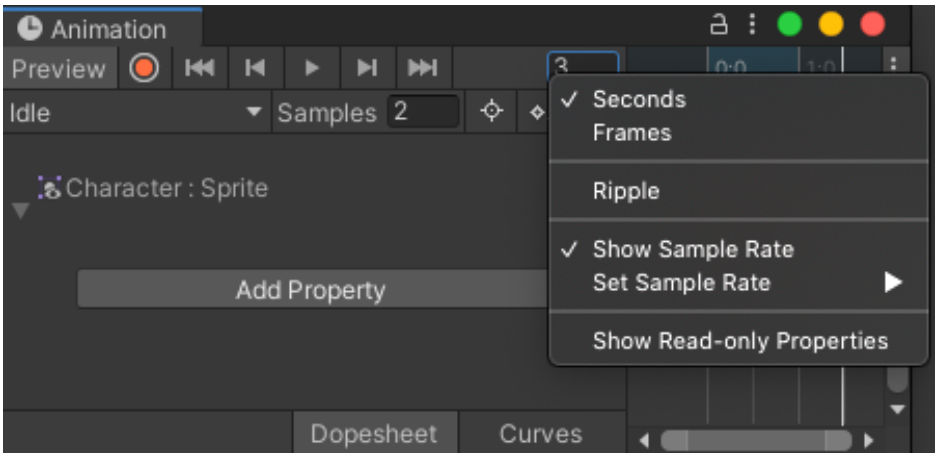
11) Animator Component and Idle Animation

Add an Animator Component to your character and open the Animation tool from Window->Animation->Animation. The Animation tool is contextual, so make sure you have your Character selected.

From the Animation tool, click “Create New Clip…” to make a new Animation called Idle and save it in the /Assets/Animations/Mario/ folder.

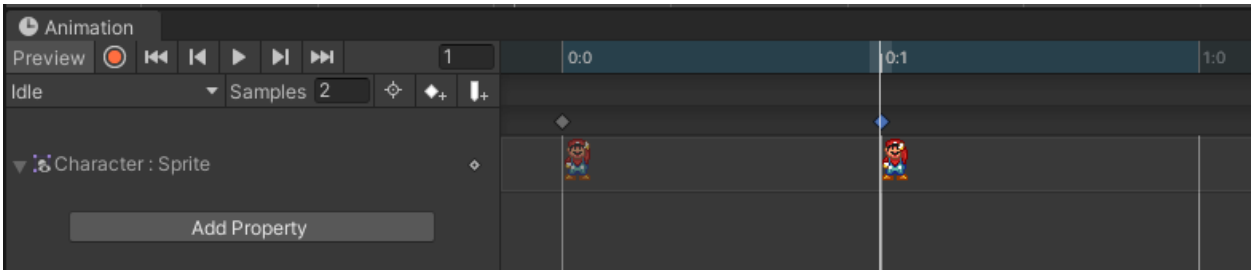
Some versions of Unity hide the “Samples” setting which allows you to adjust the frame rate of the animation. You can reveal this option by clicking the Settings dots in the upper-right of the Animation tool and choosing “Show Sample Rate.”

Be sure to set the proper Samples setting FIRST before placing your animation frames.



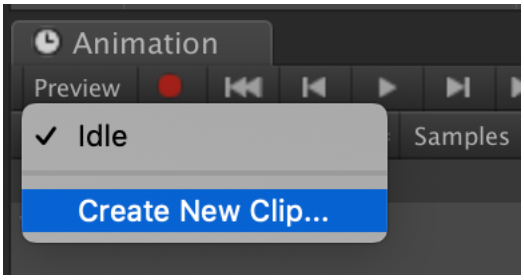
Our character’s Idle animation will consists of frames #3 and #4, which will show Mario waving at you.

Always DELETE any excess frames that may have been added automatically if they aren’t shown in these screenshots.

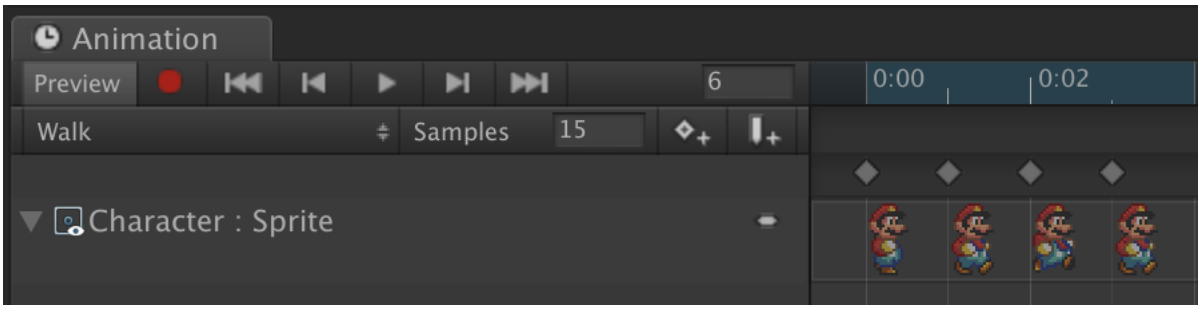


12) Walk Animation

Create a New Animation called Walk and save it in the Resources/Animations folder.



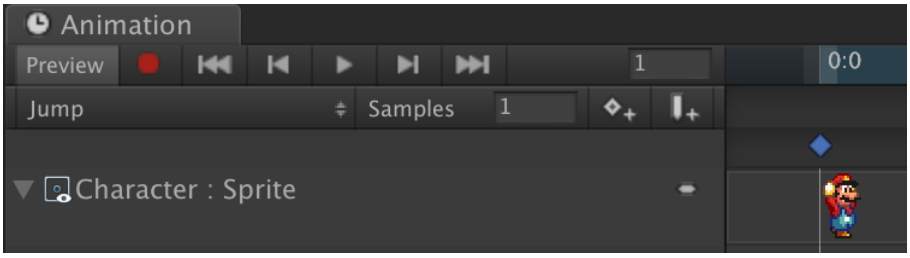
Our Walk animation will run at 15 frames per second and uses frames #13, #14, #12, and #14.



13) Jump Animation

Create a New Animation called Jump and save it in the Resources/Animations folder.

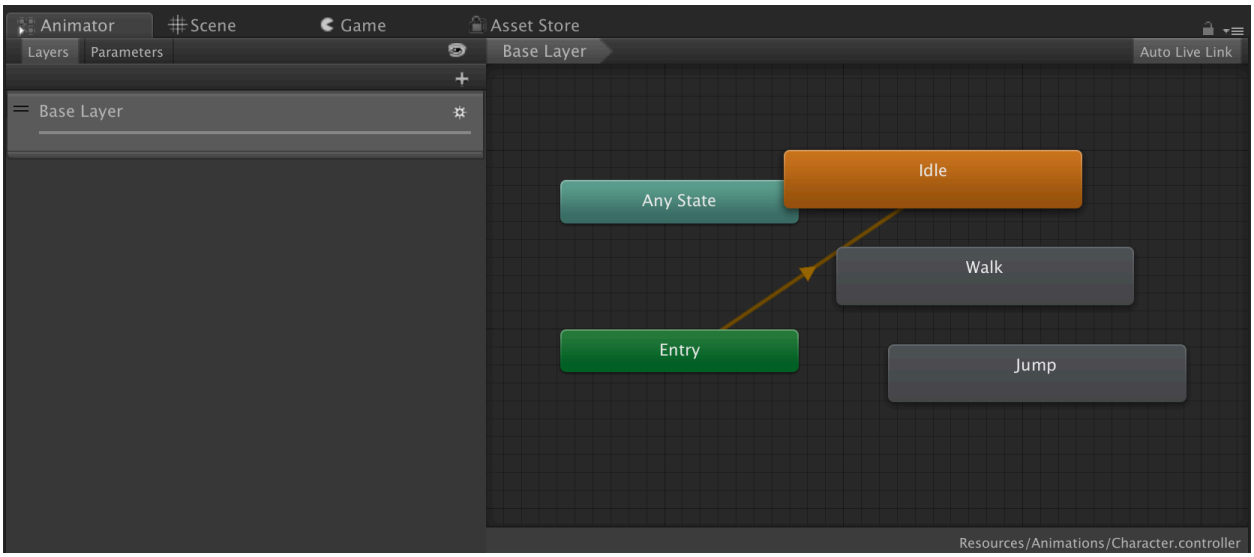
Our Jump animation will run at 1 frame per second and uses frame #26.



14) Animator Transitions

Advanced Animations require two windows. You are already familiar with the **Animation** window. You will use the **Animator** window to control transitions between animations.

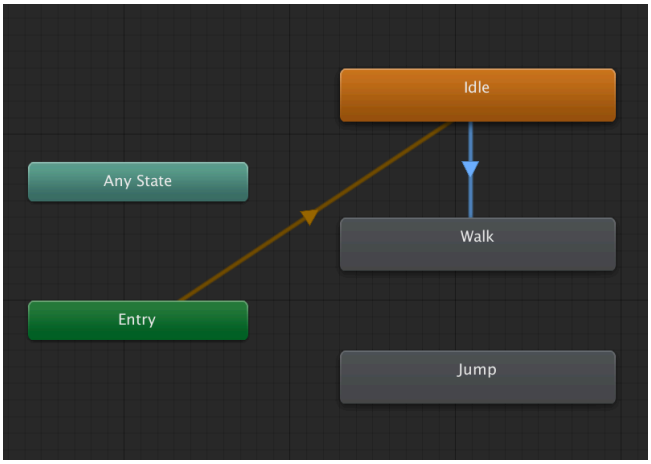
Open the Animator screen from Window->Animation->Animator. Like the Inspector, the Animator window is contextual. Select your Character to see its Animator details.



Our Idle animation is highlighted in orange to represent that it is our default animation. The Animator window shows all Animations created for the selected object and allows us to create contextual transitions between our animations.

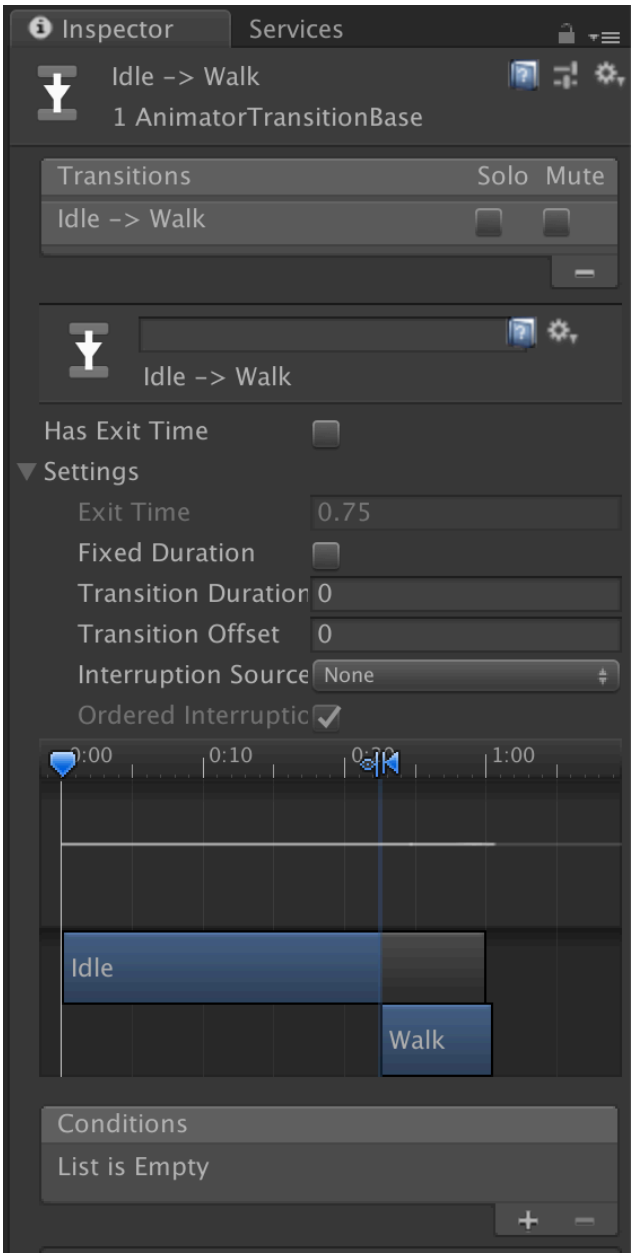
Animations can transition between each other. “Any State” is a shortcut for saying that all animations can be a valid source for a transition.

To create a transition, right-click the Source animation, click “Make Transition,” then click the Destination animation. Create a Transition from Idle to Walk.



Click the Transition Arrow so that Inspector can show configuration options. These same animation tools are used for 3D animations, so there are a lot of extra options for blending that we don’t need for 2D graphics. We will uncheck “Has Exit Time,” expand Settings, uncheck “Fixed Duration,” and set Transition Duration to 0.

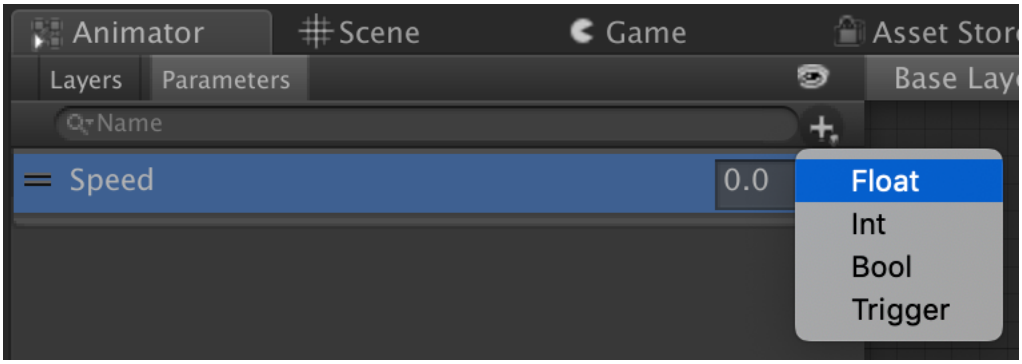
You’ll see the transition graphic update to portray a sudden transition rather than a blended transition that wouldn’t make sense for 2D sprite art.



Notice the empty Conditions box at the bottom. Conditions define the scenarios when a Transition will occur. Two requirements must overlap: 1) Our current animation must be the source animation at the base of the transition arrow, and 2) All Conditions must be met.

15) Animator Parameters

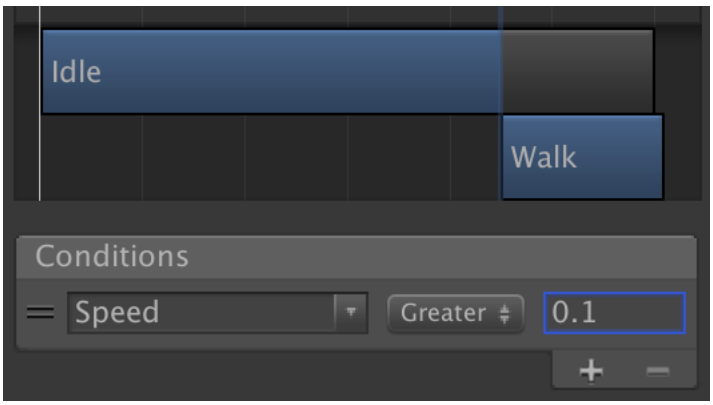
Conditions are configured using Parameters. Parameters can be variety of variable types. For example, we will use a Float parameter to keep track of our characters Speed.



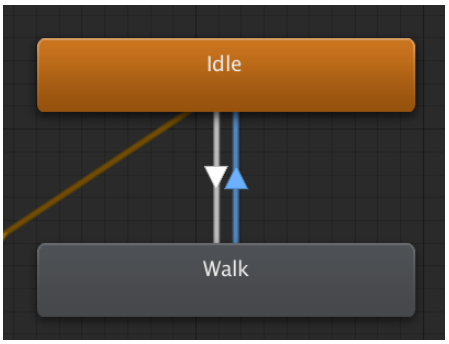
An Animator is sometimes called an Animation Controller because it is like a control center for managing animations. The logic flow works like this:

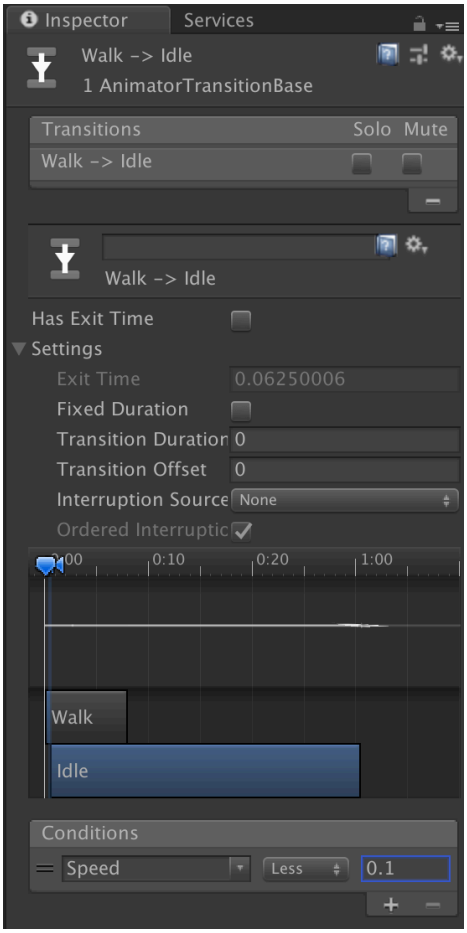
Animations let us show a sequence of graphics.
Transitions let us switch between Animations based on Parameter conditions.
Parameters are variables whose values can be changed during gameplay.
Animators manage the interplay of Parameters, Transitions, and Animations.
Scripts tell the Animator to set Parameters to different values.

Make sure your Transition Arrow is selected and click the + button in the Conditions box. We will Transition from Idle to Walk if our Speed parameter is Greater Than 0.1.



Create a Transition going the other direction with the same settings. We will Transition from Walk to Idle if our Speed parameter is Less Than 0.1.



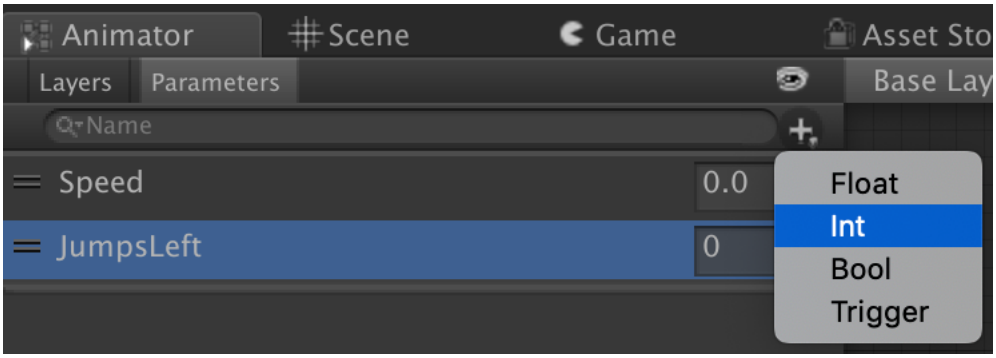


Right now, our Speed parameter never changes. We must modify our PlayerController script to update our Animator parameters as appropriate.

```
public class PlayerController : MonoBehaviour {

    // Outlet
    Rigidbody2D _rigidbody2D;
    public Transform aimPivot;
    public GameObject projectilePrefab;
    SpriteRenderer sprite;
    Animator animator;

    // State Tracking
    public int jumpsLeft;

    // Methods
    void Start() {
        _rigidbody2D = GetComponent<Rigidbody2D>();
        sprite = GetComponent<SpriteRenderer>();
        animator = GetComponent<Animator>();
    }

    void FixedUpdate() {
        // This Update Event is sync'd with the Physics Engine
        animator.SetFloat("Speed", _rigidbody2D.velocity.magnitude);
    }
```

16) Finalize Transitions and Parameters

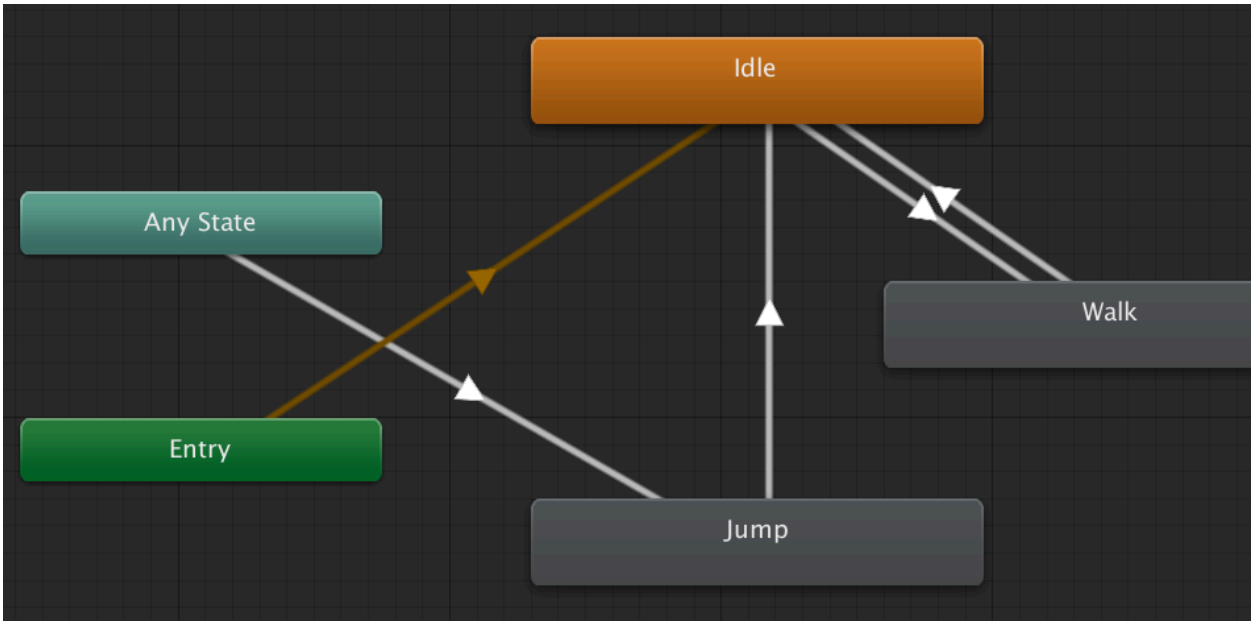
To configure the conditions for Jump animation transitions, we need a new Integer parameter called JumpsLeft.



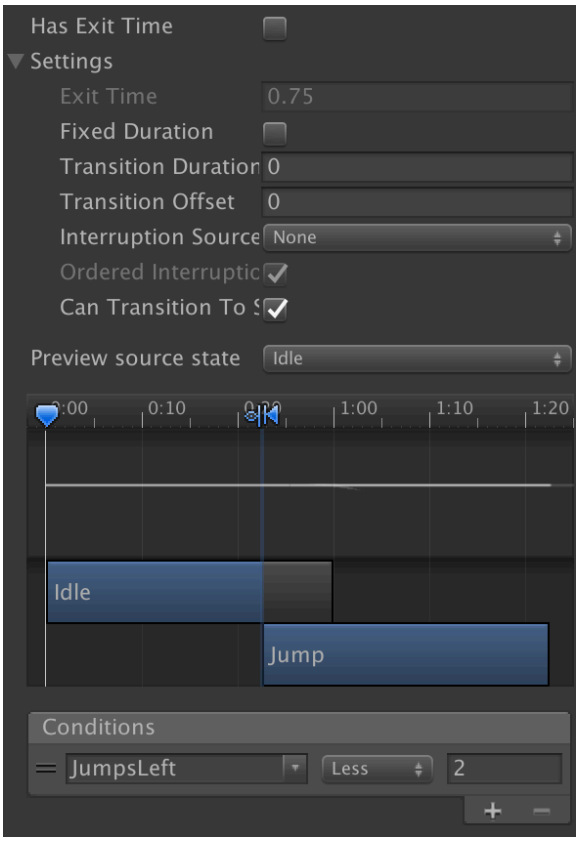
We update the number of JumpsLeft every Update event. Do NOT put this statement inside the if blocks, otherwise it won't update often enough.

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

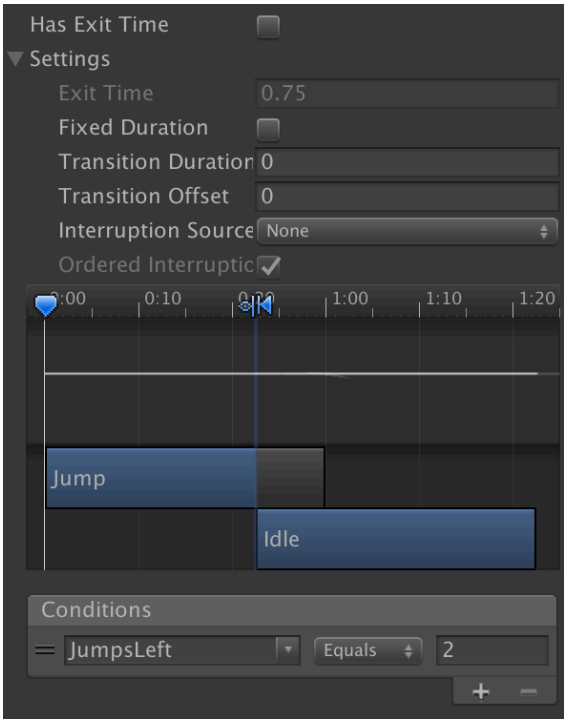
Because we can enter a Jump state from either Idle or Walk, we will Transition from “Any State” to Jump. For simplicity, our Jump will always return to Idle on landing. Create these two Transitions as described.



For the “Any State” to Jump transition:



For the Jump to Idle transition:



17) Dynamic Animation Speed

One last detail is the speed of our Walk animation. If you play the game now, you'll notice the Walk animation plays at the same speed no matter how fast the character is actually going. This means that at some movement speeds, the motion of the feet does not match the movement of the character.

We can solve this by dynamically adjusting the speed of the Walk animation based on the actual movement speed of the character.

```
void FixedUpdate() {
    // This Update Event is sync'd with the Physics Engine
    animator.SetFloat("Speed", _rigidbody2D.velocity.magnitude);
    if(_rigidbody2D.velocity.magnitude > 0) {
        animator.speed = _rigidbody2D.velocity.magnitude / 3f;
    } else {
        animator.speed = 1f;
    }
}
```