

Animator and Blend Trees

Note: This lab will explain how to implement Animations using the Animator and Blend Trees. It will assume that you are comfortable with the creation of Animations, whether modular or frame-by-frame; if you are not, please review the [Animation Lab on the DeCal website](#). This lab was created on Unity 2019.1.2.

Table of Contents

Overview.....	1
Setup.....	2
Load your .anim Files into States	2
Load Your Animation Frames	2
The Animator	3
Transitions	4
Connecting to Scripts	7
Testing	10
Tweaking Animation Speeds	11
Checkoff:.....	11
Blend Trees	12
Checkoff:.....	15

Click the links above to jump to a topic or open the Bookmarks/Navigation tab in your PDF viewer.

Overview

The **Animator** is an element with an associated Component and Window (similar to Inspector, Project, Animation, etc.)

This window is where you connect your code to the visuals by defining the animation transitions.

Blend Trees are used inside of the Animator to organize and *smooth the transition between similar Animations*.

Use cases include transitioning between walk and run animations depending on how far you're pushing your joystick or simply transitioning between up/down/left/right movement animations in a top-down 2D game (think retro Legend of Zelda).

In this lab, we will go through the process of implementing an **Animator component** for a player GameObject using **predefined animations** (either through the provided frames or through your own .anim files). Then we will communicate between the code handling the player movement and the animation transitions to finish implementing a simple 2D platformer player with idle, run and jump animation states.

After that, we will implement a blend tree to transition between eight different animation states for a top-down 2D game.

By the end of this lab, you should be comfortable with creating and customizing Animator mappings as well as recognizing how to simplify mappings using blend trees to support whatever your future games will require.

Setup

Make sure you have the Animator scene open. If it is not, find the Scenes folder in the Project View and double click the Animator scene.

You have two options to set up your Animator: load your animation frames or load your .anim files if you have pre-made animations.

Load Your Animation Frames

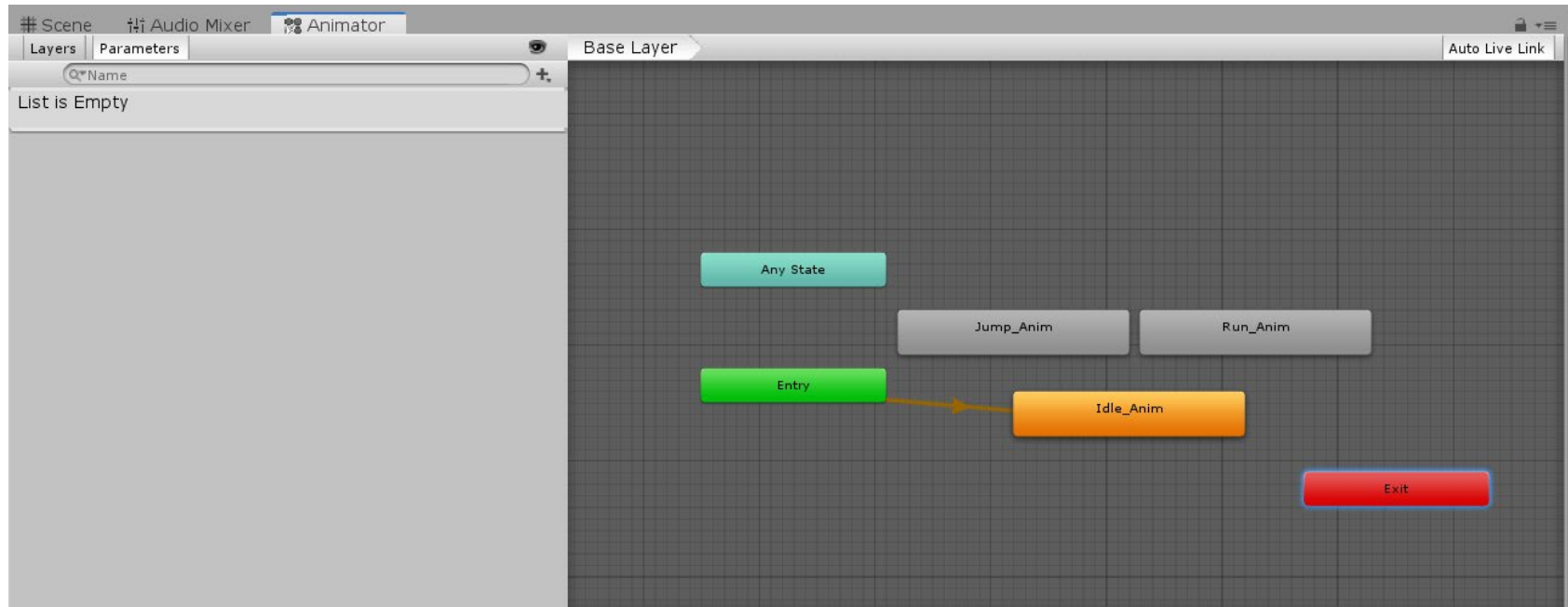
1. For each folder within Animator/Animations/Frames, select all the images, click and drag them over the Player GameObject in the Hierarchy View.
2. This will create animations using each set of images you selected. Name them Idle_Anim, Jump_Anim and Run_Anim respectively and create them inside the Assets/Animator/Animations folder.
3. Be sure to preview the animations to make sure they're playing at the right speed, review the [Animation lab](#) for details on how to use the Animation View.
4. Notice that inside the Animations folder in the Project View, there is a new item called "Player."
5. Select this item to reveal that it is a Component of type Animator.
6. Select the Player GameObject in the Hierarchy, note that this Component has automatically been added as a result of your dragging and dropping.
7. Double-click the Player Animator in the Project View to open the Animator View. Alternatively, you can open the Animator by navigating to Window>Animation>Animator.

Load your .anim Files into States

1. If you have your own .anim files you want to use for this tutorial, add the Animator component to the Player GameObject by selecting "Add Component" and typing in "Animator" and title it "Player." Save it to the Assets/Animator/Animations folder.
2. Double-click the Player Animator in the Project View to open the Animator View. Alternatively, you can open the Animator by navigating to Window>Animation>Animator.
3. Right click the empty space and select "New State."
4. Select this new state so that its properties appear in the Inspector.
5. Drag your .anim file from the Project into the Inspector over the Motion field to load your animation.
6. Rename your new state(s) accordingly.

The Animator

Your Animator should look something like this, with an Entry state, an Any State, three _anim states, and one Exit state:



To navigate this view:

1. LMouse to select/move the various animation states
2. Rmouse to bring up further options
3. Scroll wheel to zoom in and out
4. Scroll Button + drag to pan around

Drag the states around until you're comfortable with how to navigate around the Animator.

Transitions

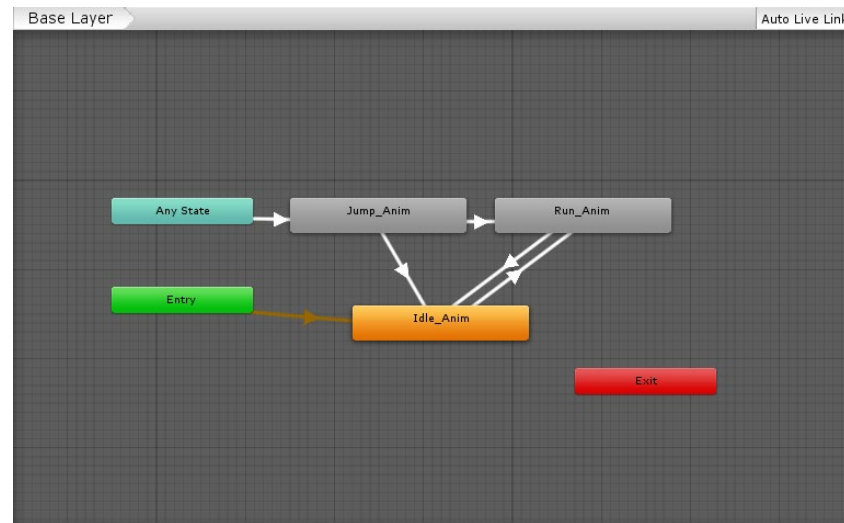
Upon Entry, we will go directly to the Idle animation. Think for a moment about how you would want to be able to transition from one state to another, sketch out or visualize what that would look like in this mapping, then continue reading.

From Idle, we want to be able to transition into Run, and from either Idle or Run we want to be able to transition into a Jump. After we finish Jumping, we want to be able to stand Idle or continue Running, and of course we also want to be able to stop Running and stay Idle.

To create a transition, right-click a state and select “Make Transition” and left-click the destination state.

Try implementing the word vomit above on your own before continuing.

Create transitions according to the following Animator mapping:



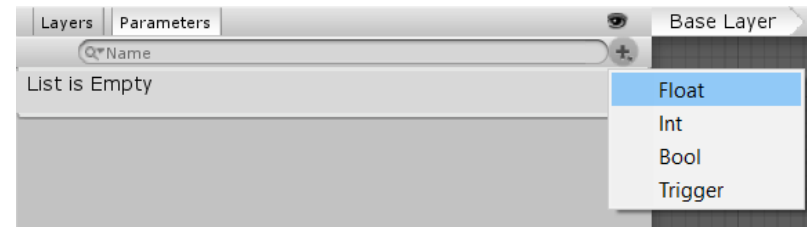
How will we determine when to execute a transition? We need to define parameters that will only activate when our specified conditions are met.

At a high level:

- We will be **idle** if we are not moving
- We will **run** when the magnitude of our velocity (our speed) is anything other than 0
- We know we will **jump** when the jump button is pushed.

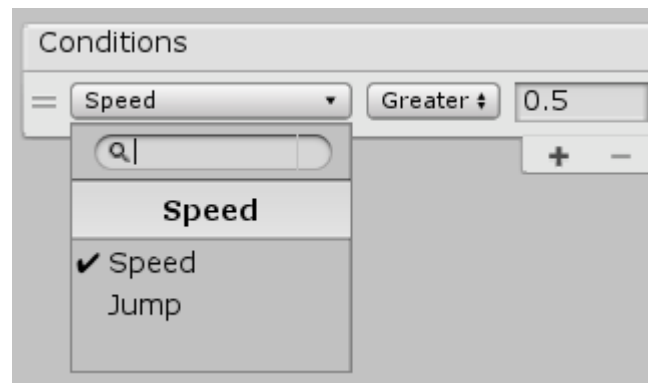
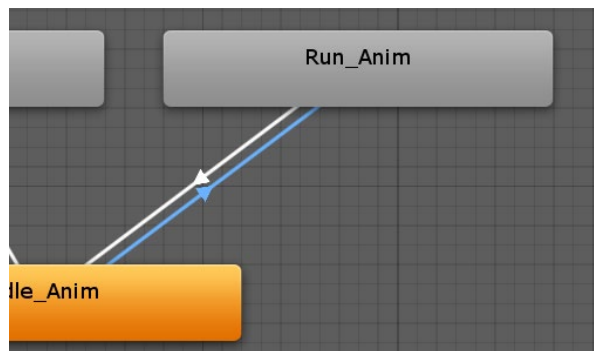
On the left half of the Animator View, observe the Parameters tab. Use the + button to the right of the search field to create a new float parameter called "Speed" and a new Trigger parameter called "Jump." Note that the **capitalization of these variable names do matter**, so try to stay consistent in your naming conventions.

Now we will implement these parameters into the State Transitions we defined above.



Select the transition arrow from Idle to Run.

In the Inspector, click the + button under the Conditions tab and define it to react only when Speed is greater than 0.5. We just need a small number; technically, a number like 0.00001 would suffice, but 0.5 just looks cleaner and functions the same for the purposes of this tutorial so we'll stick with that.



Now make the following changes:

- Select the transition from **Run to Idle** and set it to activate only when **Speed is less than 0.5**
- Select the transition from **Jump to Idle** and set it to activate only when **Speed is less than 0.5**
- Select the transition from **Any State to Jump** and set it to activate only when the **Jump Trigger is activated**.

You may be wondering why we can't use the Any State to transition to Idle and Run the same way we did for Jump. It's not that you can't, there are many ways to implement an Animator, this is just one of them. You could use a Boolean to check isMoving, instead of having a Speed float variable, or have an isJumping Boolean to check for whether a foot collider is touching the ground or not. Be sure to explore, expand or redesign this implementation as part of the extra challenges part of this lab.

Connecting to Scripts

Our Animator logic is all set up, everything makes sense in theory but none of it has truly been implemented yet. To do this, we'll have to communicate our movement to the Animator Component through our movement script.

Create a Movement.cs script and attach it to the Player GameObject.

In order to communicate with the Animator, we'll need to create a reference to it in our script.

You can either (1) create a private reference and use GetComponent to retrieve the Animator or (2) create a public reference and drag/drop the Animator in from the Inspector View.

(1) :

```
private Animator animator;

...

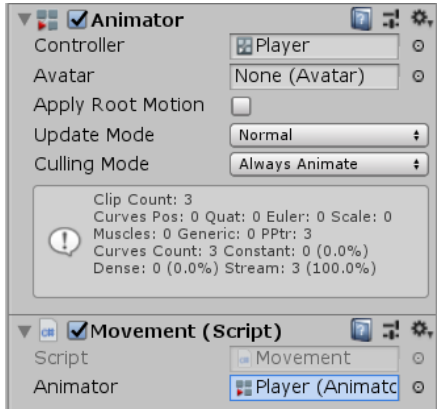
void Start() {
    animator = GetComponent<Animator>();
}

...
```

(2) :

```
public Animator animator;

...
```



Inside the script, create the Update() function. Inside Update(), check for inputs corresponding to whichever keys you've assigned to be left, right and jump using either Input.GetKey() or Input.GetButtonDown. For jump, use GetKeyDown or GetButtonDown, as this will only trigger once when you push it, and will not continue to fire if you hold it.

Using our reference to the animator, you can use the various .SetFloat/.SetBoolean/.SetTrigger methods to change the parameters inside the Animator. In our case, you'll make the following calls at least one time:

```
animator.SetFloat("Speed", 1);  
animator.SetFloat("Speed", 0);  
animator.SetTrigger("Jump");
```

Remember that because of the way we implemented our Speed check, you'll have to manually reset the speed to 0 whenever you do not detect a left/right input. **Try implementing the script yourself before moving on to the staff solution.**

Movement.cs

Since this isn't really a scripting lab, the code for this script is below. Feel free to ctrl+c/ctrl+v the whole thing, but you will be required to understand how to modify the Animator parameters via code in order to check off:

```
using UnityEngine;

public class Movement : MonoBehaviour
{
    public Animator animator;
    private Rigidbody2D rb;
    private SpriteRenderer sr;

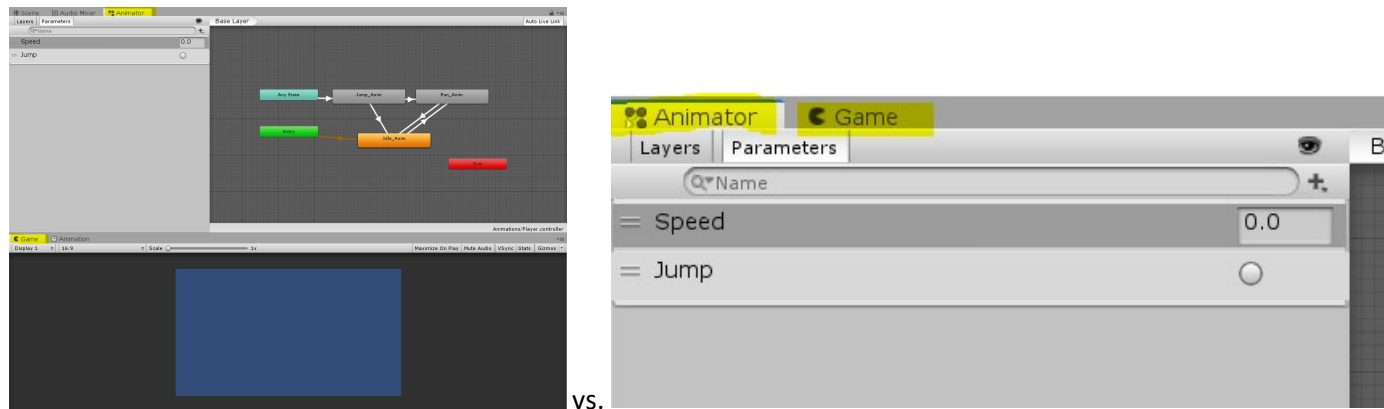
    private void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        sr = GetComponent<SpriteRenderer>();
        animator = GetComponent<Animator>();
    }

    private void Update()
    {
        if (Input.GetKey(KeyCode.A))
        {
            animator.SetFloat("Speed", 1);
            transform.position = (Vector2)transform.position + new Vector2(-5, 0) * Time.deltaTime;
            sr.flipX = true;
        }
        else if (Input.GetKey(KeyCode.D))
        {
            animator.SetFloat("Speed", 1);
            transform.position = (Vector2)transform.position + new Vector2(5, 0) * Time.deltaTime;
            sr.flipX = false;
        }
        else
        {
            animator.SetFloat("Speed", 0);
        }

        if (Input.GetKeyDown(KeyCode.Space))
        {
            animator.SetTrigger("Jump");
            rb.AddForce(new Vector2(0, 5), ForceMode2D.Impulse);
        }
    }
}
```

Testing

Ensure that the Game View and Animator View are open side-by-side and not in the same Window. This will let you view the animator at work during live play and modify the parameters directly.



Hit the play button and notice that even though we didn't assign a Sprite to the Player, the animation provides it all the same.

Also note that you can view the current animation playing inside of the Animator denoted by the little progress bar in the active state looping.

Test our Jump logic by clicking the little bubble next to Jump in the Parameters list. If all goes as planned, you should play through most of your Jump animation. The trigger acts as a one-time button rather than a Boolean switch; it flips on for a single frame and then automatically switches off.

Change your speed variable to be greater than 0.5, then change it back to 0 to view the transitions.

This is useful for testing your logic before connecting it all to your scripts.

One issue here is that the Jump animation looks like it's been cut short. This is due to the transition time from one animation to another.

Exit playmode and select the transition leading into Jump.

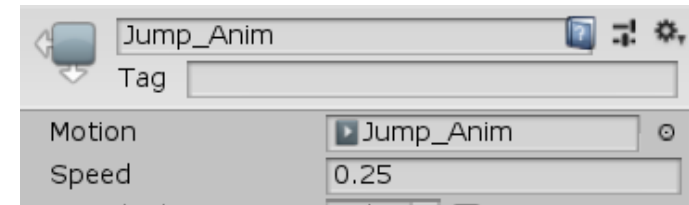
Disable "Has Exit Time" and set Transition Duration to 0. This will make the transition from Idle/Run to Jump instantaneous with no smoothing.

For player actions that require quick visual feedback like jumping, attacking or dashing, you should consider reducing or removing exit and transition time. For things like idle into running, you may want to leave the transition to make it feel smoother. This is all up to your preference.

Tweaking Animation Speeds

Press the Play button and try jumping. Notice how the jump animation is too quick; we can fix this by selecting the jump animation state and slowing the speed down. Tweak it around to see what feels right for you and feel free to change the jump strength in the script. I found a speed of 0.25 to work well with this implementation.

Don't be afraid of going into the Animation View and modifying the length of the animation itself instead of just changing the overall playback speed!



Checkoff:

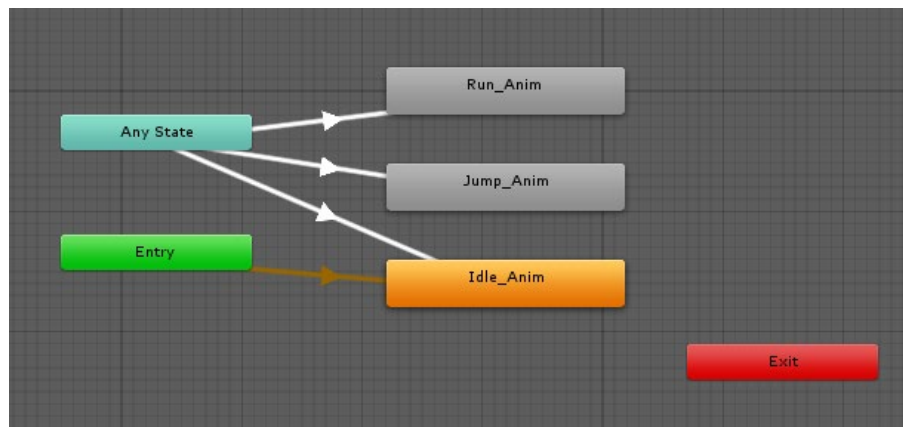
Change the Speed parameter from a float to a Boolean and make the appropriate changes in the script to ensure the same functionality.

Explain the process of implementing transitions in the Animator and setting parameters in code.

Challenges:

Remove the ability to infinite jump while in the air (i.e. you can only jump if you're on the ground)

Make whatever changes necessary to make this Animator map function the same way as our current implementation:



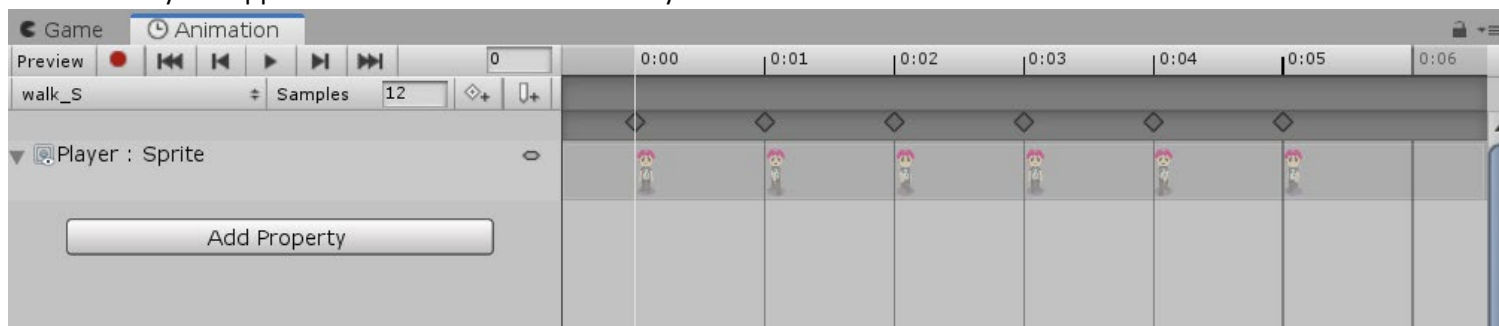
Note how in our first implementation, we effectively defined this behavior already; you can get from any one state to any other state. Explore to see if this mapping is just as viable, easier or more difficult to implement. Consider what parameters you'll need to add/remove/modify, our initial design/script may not be optimal!

Blend Trees

In the Scenes folder in the Project View, double click the Blend Tree scene.

In this lab, you'll create smooth transitions between 2D animations using Blend Trees. Blend trees are a must for organizing complicated animation transitions in Animator. They're used to quickly handle transitions between animations when the player provides new input. In 3D games, they actually interpolate between the two animations, creating a smooth transition, but the same concept of Blend Trees still apply to both 2D and 3D games.

1. Inside the **Blend Trees/Animations folder**, there are 2 folders each containing 8 different animations, one for each cardinal direction of movement.
2. We want to set it up so that a different animation will be played when the player changes direction, so that the player faces the direction it's moving in.
3. Press play and move the character around by right clicking on the screen. Notice that it follows your mouse but is always facing forward. Let's fix this with some animations and a blend tree.
4. All the sprites have been imported, but most of the walking animations we need don't exist yet. We will create them ourselves. There are two ways you can do this: use the same method in the previous part and drag every set of frames associated with one animation onto the Player GameObject and save it, or manually create a new Animation using the Animation View to immediately change things like framerate, frame spacing, etc. For this part of the lab we will go through the second process.
5. Click on the Player GameObject in the project hierarchy and look at the Animation View. If your Animation View isn't already open, you can get it from **Window > Animation** and move it to where you'd like. To see the sprites better, click the gray arrow next to the Sprite label and they will appear on the timeline. This is what you should see.



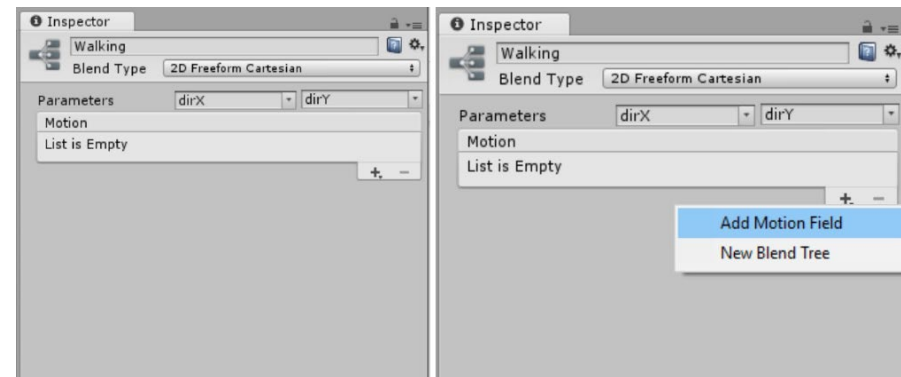
6. Each of those gray diamonds represents a keyframe in the animation and is its own unique sprite. Now, let's create the missing seven animations, one for each cardinal direction: west, north, east, and the four intermediate directions, northeast, northwest, southeast, southwest.

To start, **click on the drop-down menu labeled “walk_S”** and select **Create New Clip**. If you do not have the drop-down menu, be sure to have the Player GameObject selected in the Hierarchy. Let’s start with north, so name your new clip **walk_N**. In the Sprites folder, find the sprites that make up the north walking animation (there should be 6). Select these and drag them into the Animator View.

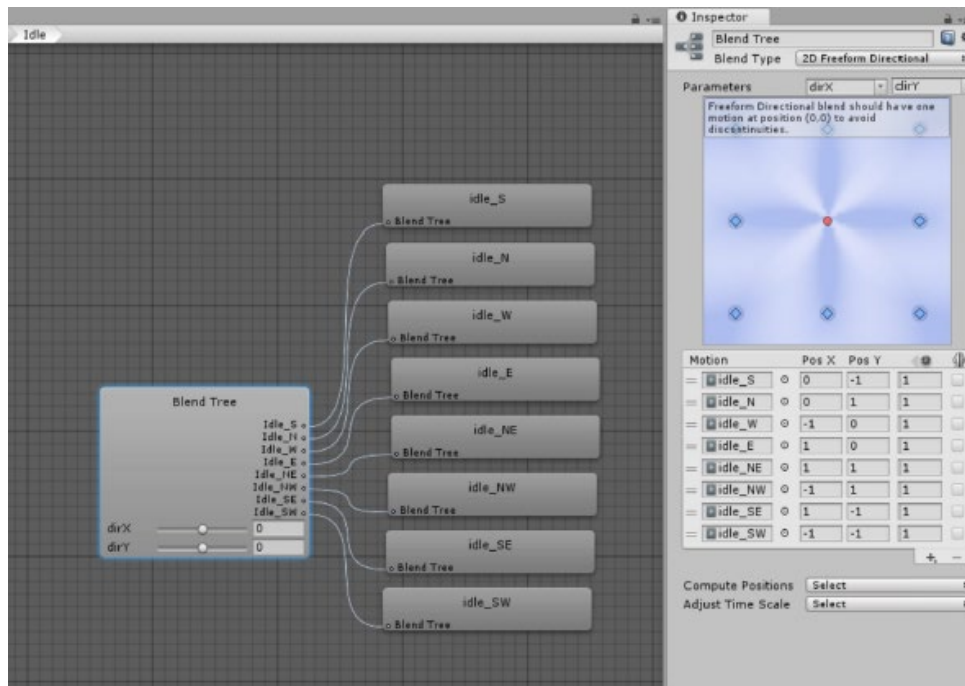
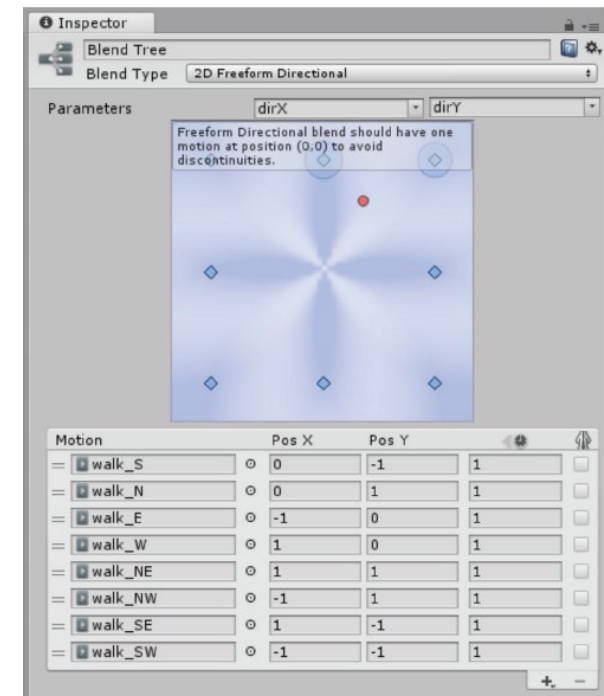
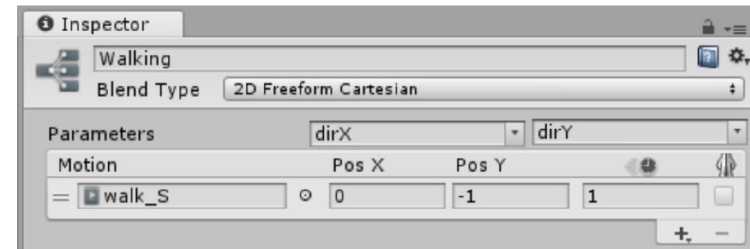
7. That’s it! Repeat this process for the other directions, using the correct sprites and naming conventions for the animations (the naming will make your life easier later).

Note: These animations will appear as nodes in your Animator View. These aren’t particularly important, so you may right click on them and delete them if you want, although this isn’t necessary.

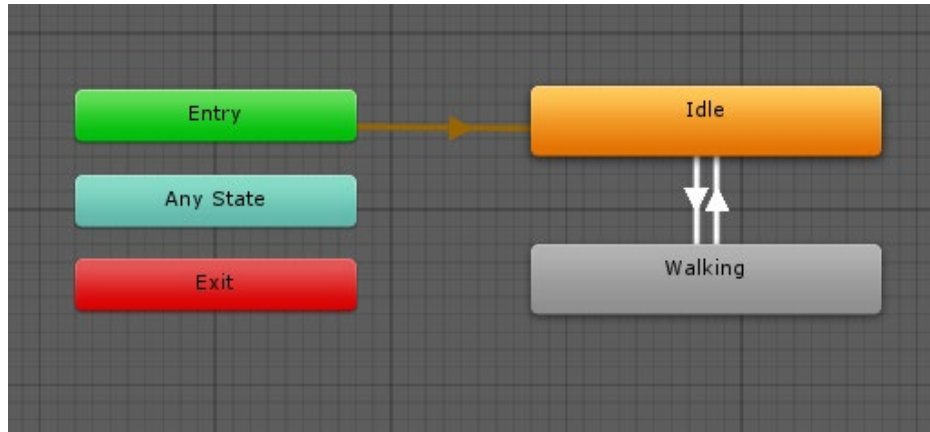
8. With the animations done, **double click on the “Player” animation controller** in the Animations folder.
9. In the Animator View, right click the background, then click **Create State > From New Blend Tree** to create a new blend tree node. Name this blend tree **“Walking”**.
10. **Double click** the blend tree you just created to open it and **click on the node that appears**. In the Inspector, select **“Freeform Directional”** from the drop-down menu and make sure the parameters that appear are **“dirX”** and **“dirY”**. These have been configured in the player movement script to correspond to the relative direction of the player’s mouse from the character’s location. The blend tree will check these values to decide which walking animation to play. Now click the plus button and select **Add Motion Field**.



11. Click the circle next to **None (Motion)** and select “**walk_S**” from the list that appears. Update the “Pos X” field with 0, and the “Pos Y” field with -1. This is because the coordinates correspond to (0, -1). The third field is the animation speed; leave it as it is.
12. Add seven more fields for each of the rest of the walking animations. **walk_E** corresponds to (1, 0), **walk_NW** corresponds to (-1, 1), and so on. In the end, your blend tree should look something like this →→→→→→→→
13. Press play and move the character around by right clicking on the screen. Notice how now, a different animation is played when the player moves in a different direction!
14. The character currently doesn’t know how to stand still when it’s not moving. We can fix this with another blend tree. In the Animator View, click **Base Layer** to return to the lowest mapping level. Repeat the previous steps, but with the Idle animations instead of the walking animations.



15. Finally, all that's left to do is to create transitions between the idle and walking animations. Return to the **Base Layer** and **right click on the Walking blend tree node**, then select **Make Transition**. A white arrow will appear; **click on the Idle blend tree node** to anchor it.
16. Click on the white arrow that appears between Walking and Idle. **Press the plus button under "Conditions"** and select **"walking"** from the drop-down menu, with a value of **false**. **Uncheck the "Has Exit Time" box as well as the "Fixed Duration" box**.
17. Create a second transition, this time from Idle to Walking, and **set "walking" to true**.
18. Right click the Idle blend tree to make it the new Layer Default State. When you're done, it should look like this:



19. That's it! Press play to see the animations and transitions in action.

Checkoff:

Move your character around the screen in windowed play mode to show the blend tree changing.

Challenge:

Go back to the Animator Scene and apply what you now know about Blend Trees in a 1D blend tree to transition between its Idle and Walking animations with the parameter based on our Speed float. Create a "Slow Walk" animation by slowing down the playback of the Walking animation and set it to only play if the speed is in between the values of Walk and Idle. This way, if you're using a joystick with variable input you can walk slowly and play a slower walking animation with it. This idea also works incredibly well with the top-down type of game in the Blend Tree scene if you have controller input and not mouse movement.