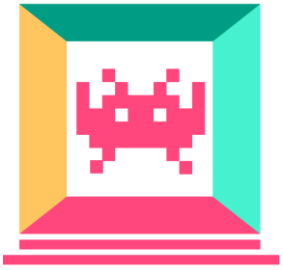


Unity Animation for Beginners

By Tim Bonzon

Unity Animator and Programmer

This book is brought to you by Zenva - Enroll in our **Unity Game Development Mini-Degree** to learn even more about game development with Unity.

Before diving into this eBook, why not check out some resources that will supercharge your coding skills:

This book is brought to you by Zenva - Enroll in our **Unity Game Development Mini-Degree** to learn even more about game development with Unity.

Table of Contents

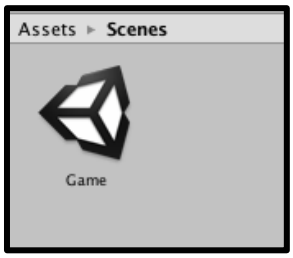
The Comprehensive Introduction to the Unity Animator

Introduction

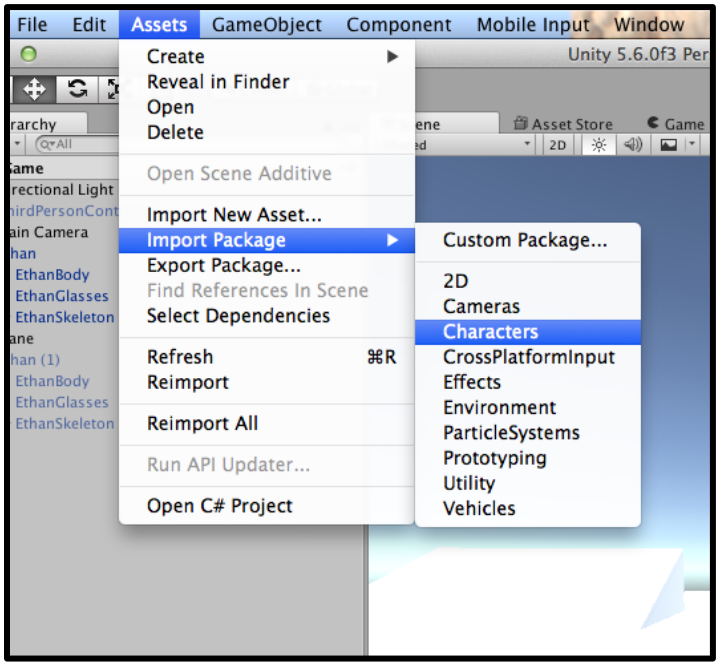
In this lesson we will explore how to use animations in a scene. This tutorial is a good step for those who have read the tutorial on creating animations. Which you can view here. However, no prior knowledge of animation is required. The scripts for this lesson can be downloaded here.

Setting up the project

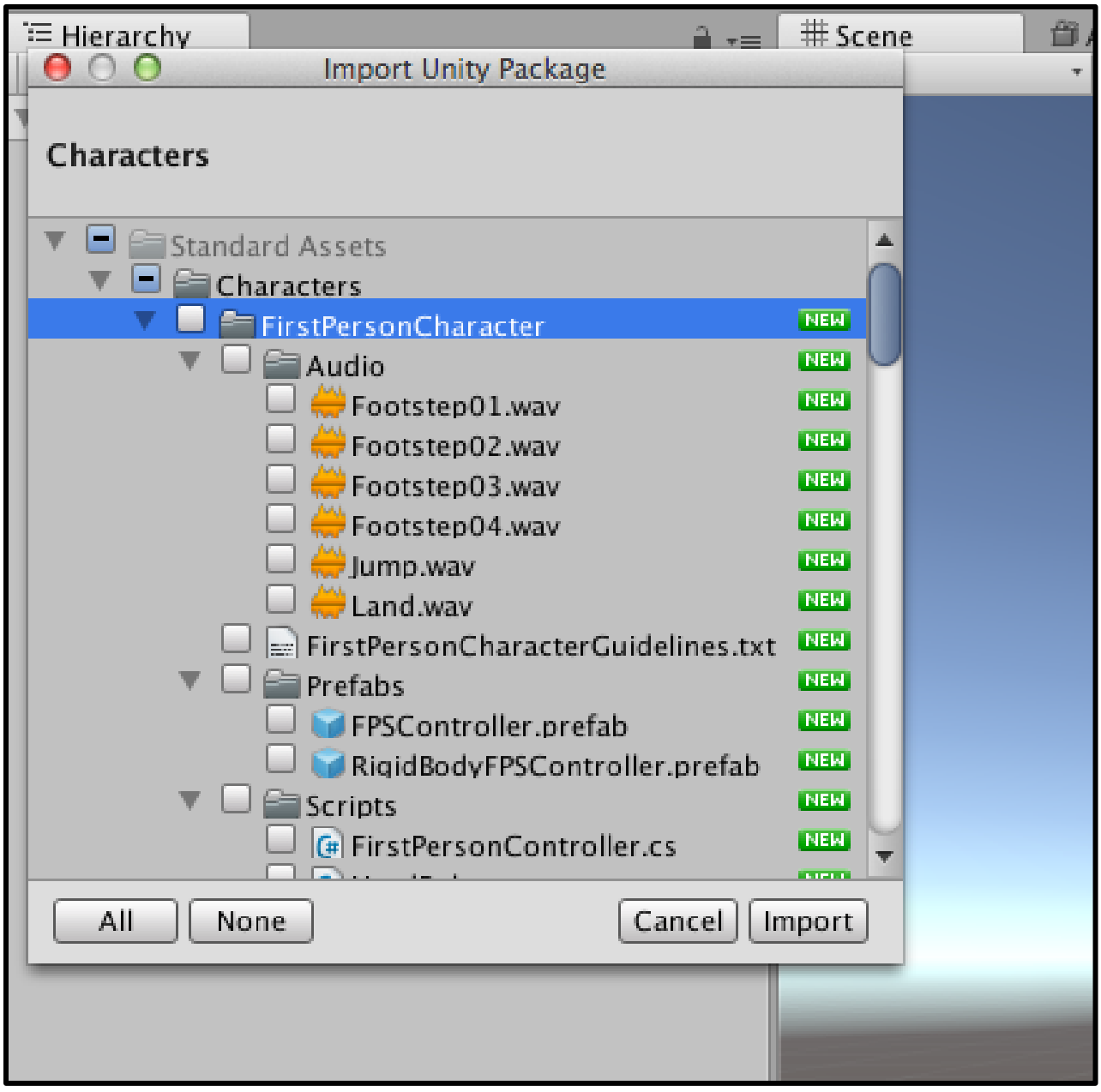
Open up Unity and create a new project. Create a folder called "Scenes" and save the current scene in that folder.



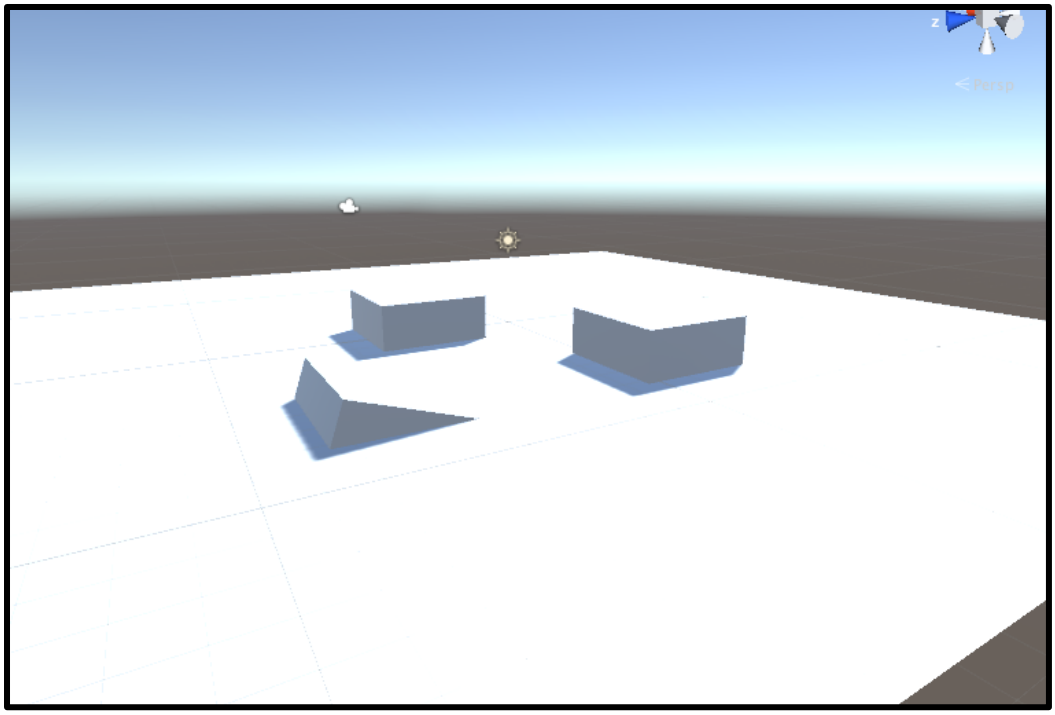
Go to Assets -> Import Package -> and select the "Characters" package.



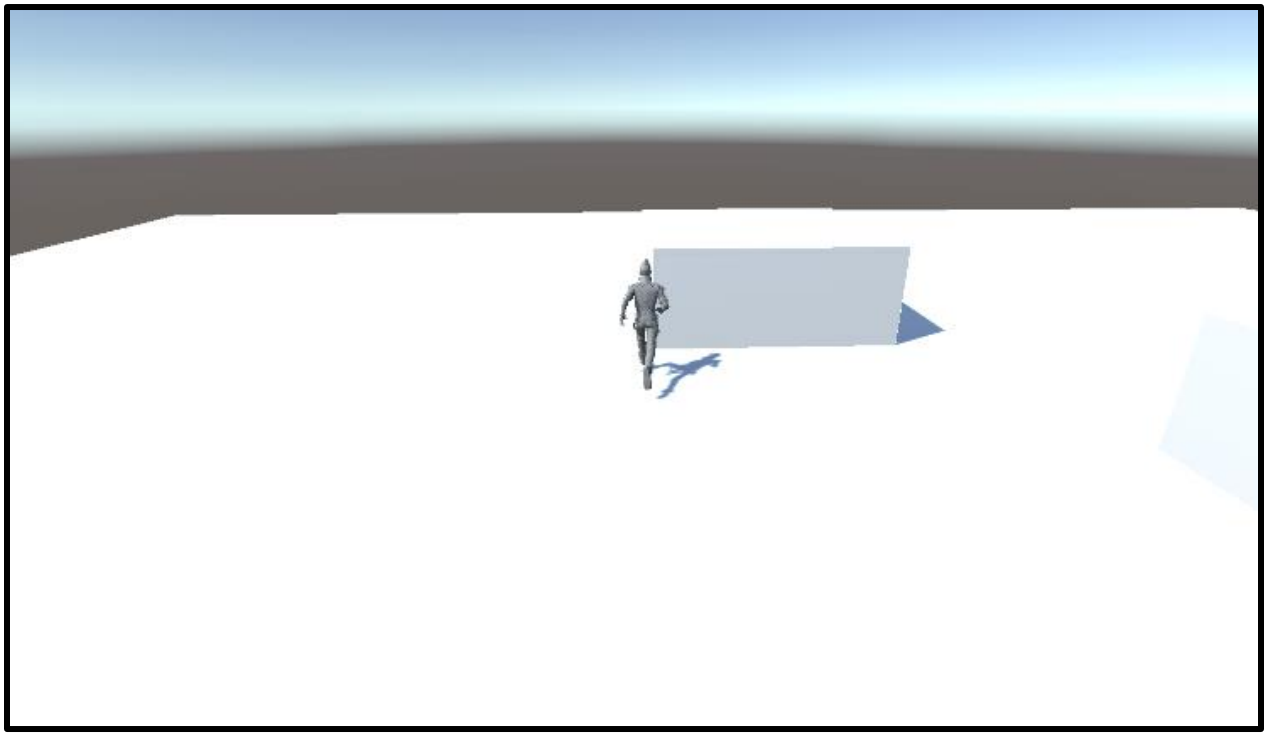
Uncheck the first person character since we won't be working with that.



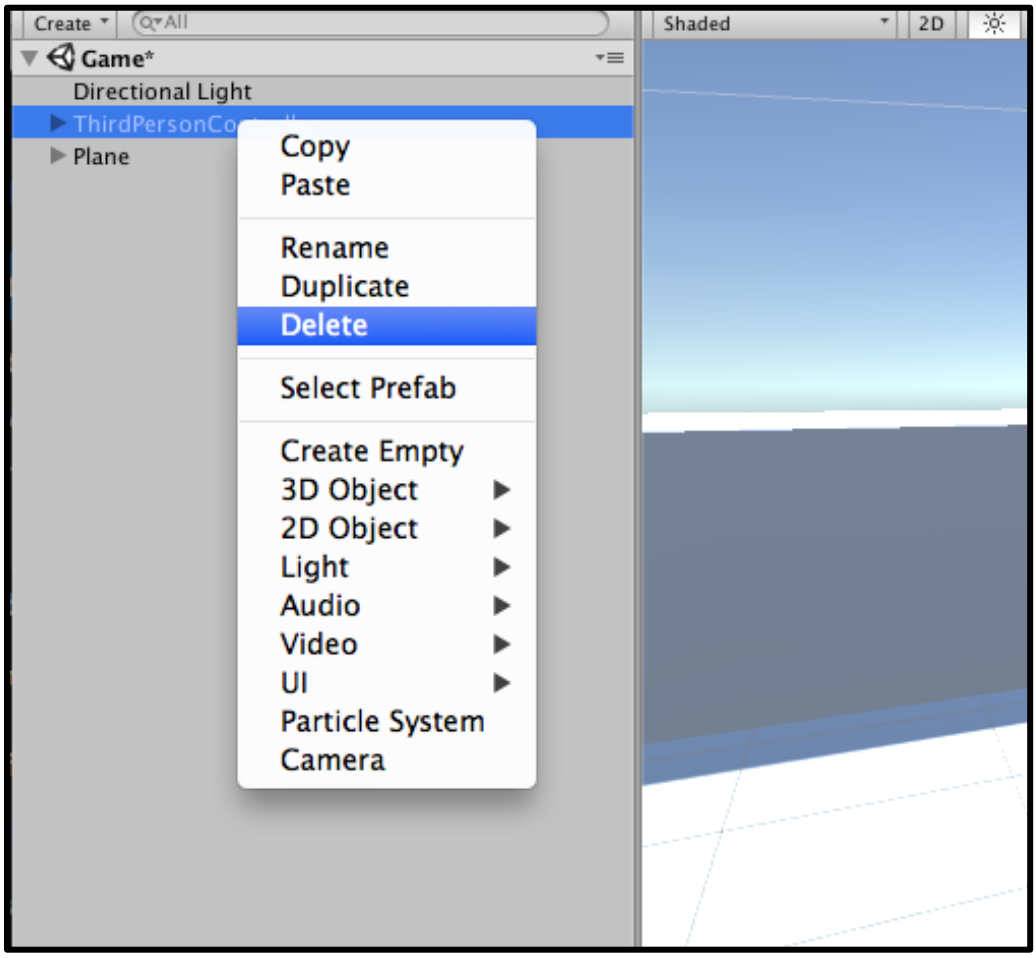
Now make a plane for the character to stand on. Next, make some cubes and ramps for our character to run and walk on.



This is optional but, go ahead and drag in the "ThirdPersonController" into the scene. Play and look how the character moves. We are going to build, from scratch, a character that moves similar to this.

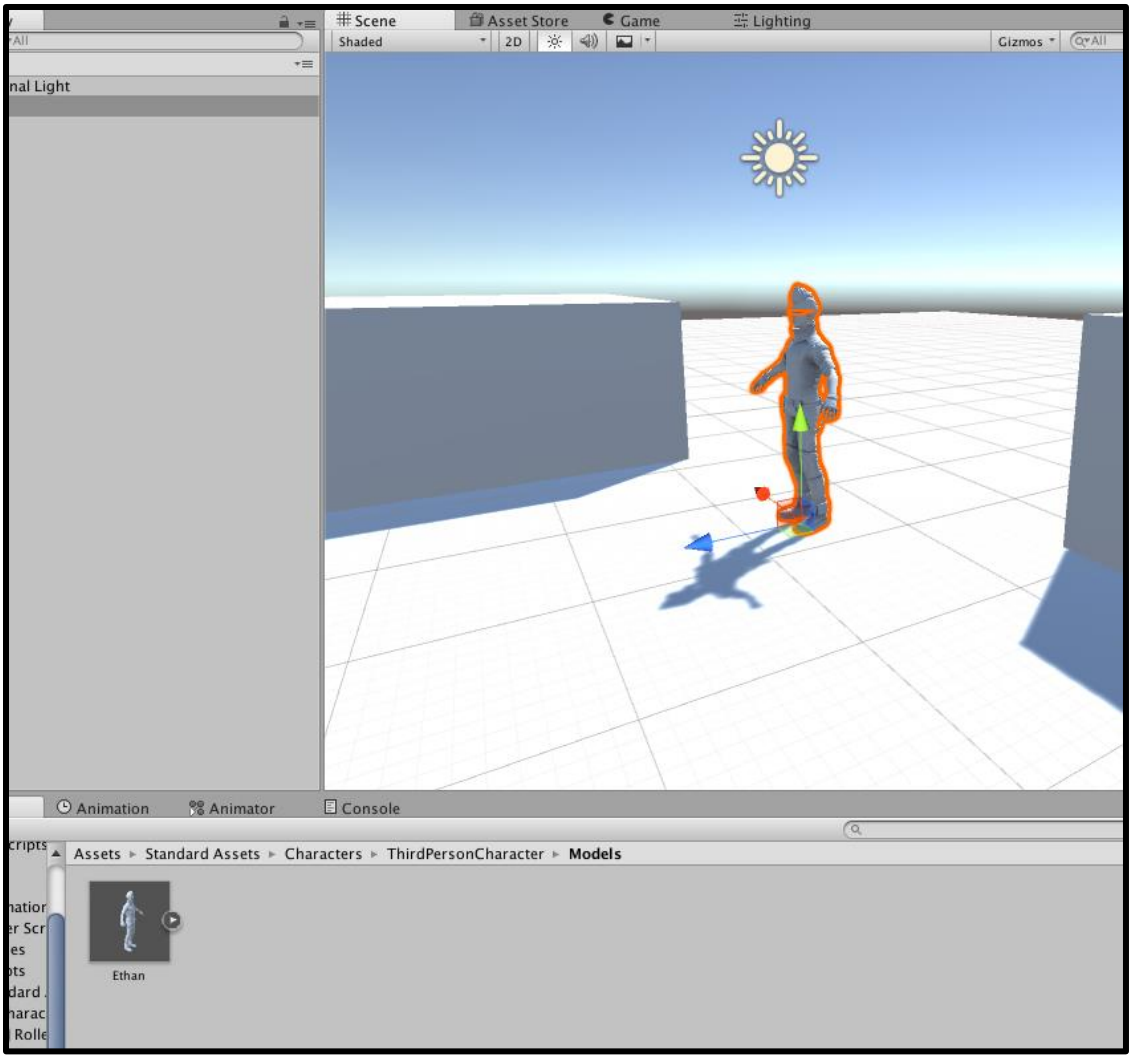


Once you have got a feel for how the character moves, ThirdPersonController is no longer needed and you can delete it.

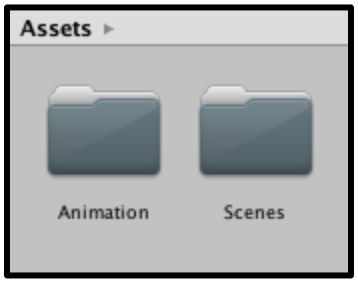


The Animator

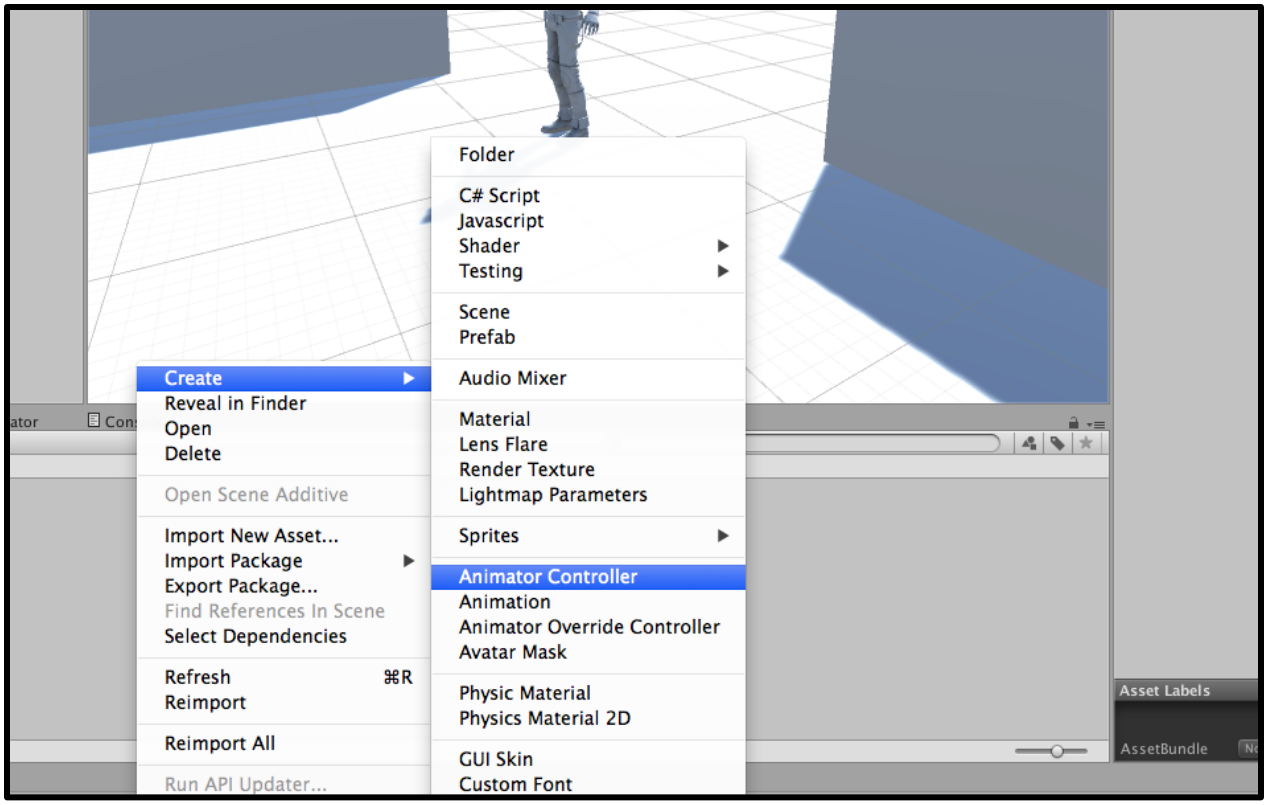
Navigate in your project files to Standard Assets -> Characters -> Models and drag in the "Ethan" model.



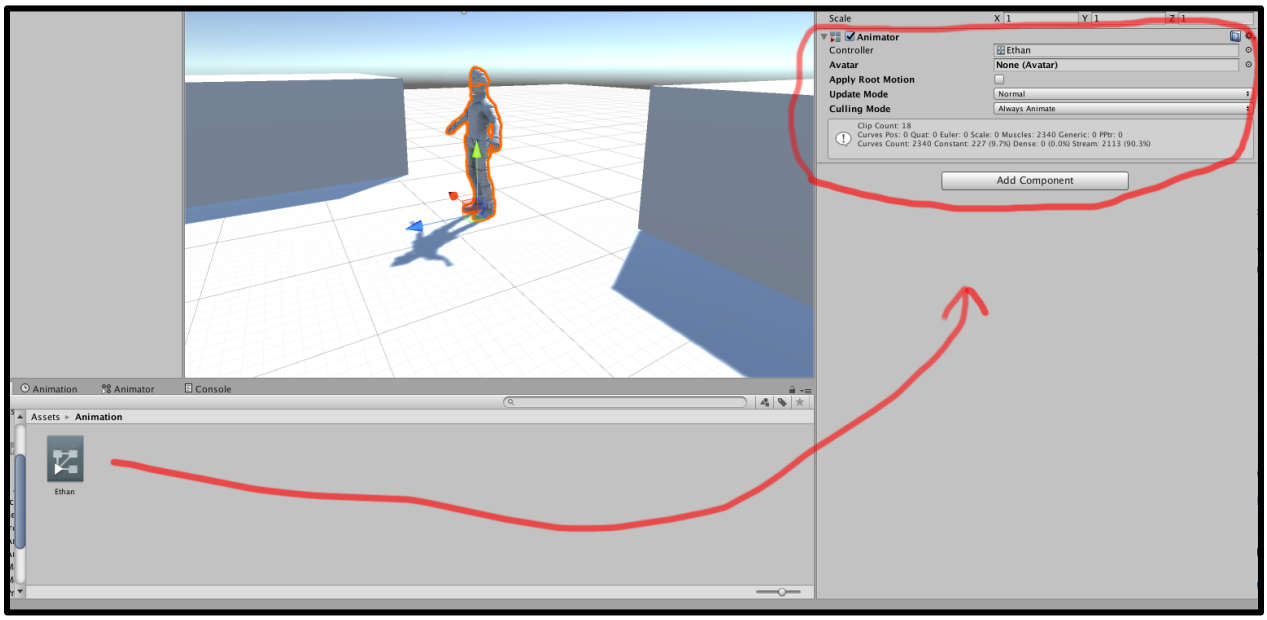
Then create a folder called "Animation".



Right click and go to Create -> "Animator Controller".



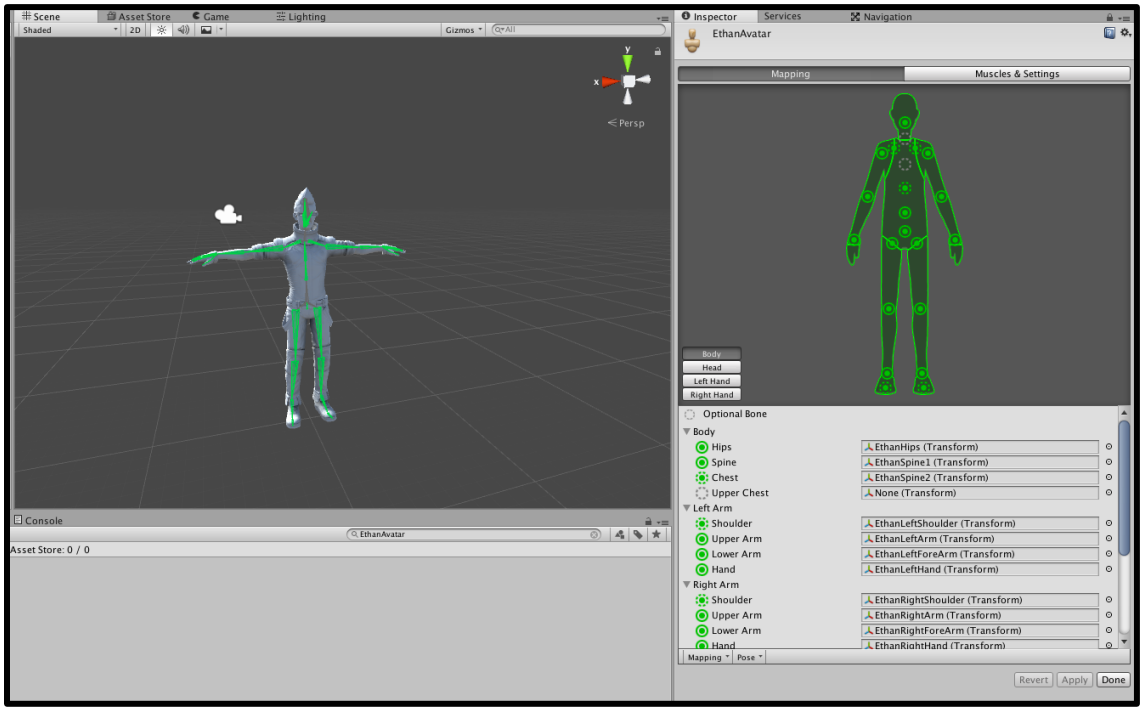
Name it "Ethan". Drag our new Animator Controller onto the Ethan model.



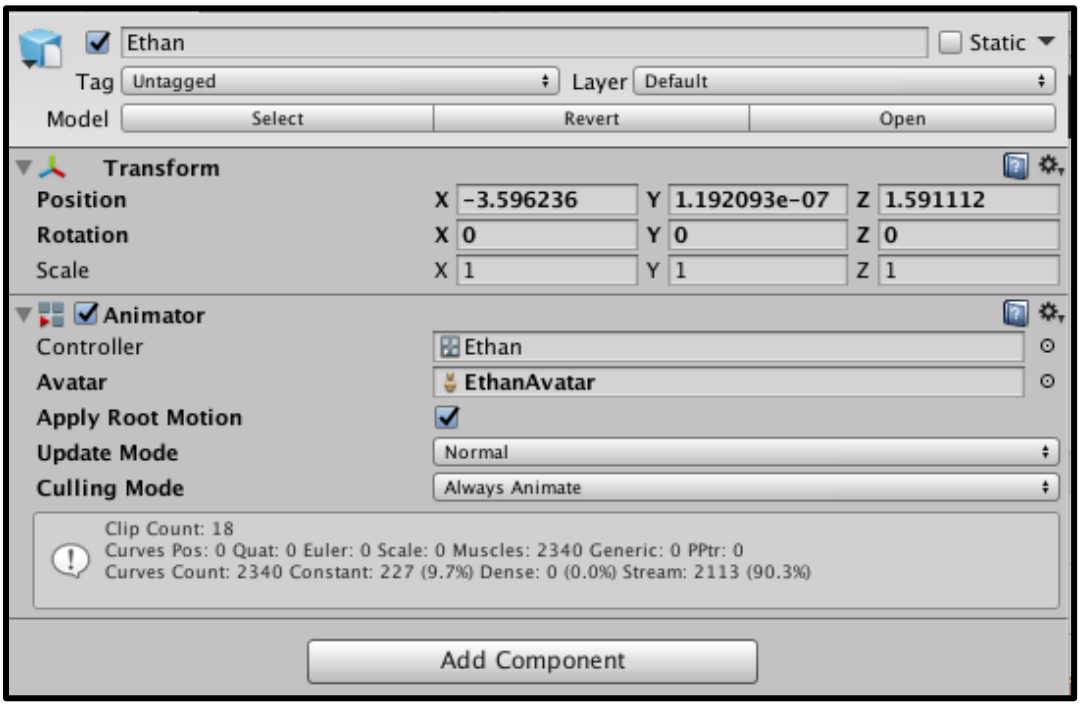
The Animator Controller allows you to assign multiple animations to a model. For humanoid characters, such as ours, it requires something called an "Avatar". Click on the circle next to the Avatar field and find the "EthanAvatar". Lets have a look at the Avatar.

Click "Configure Avatar".

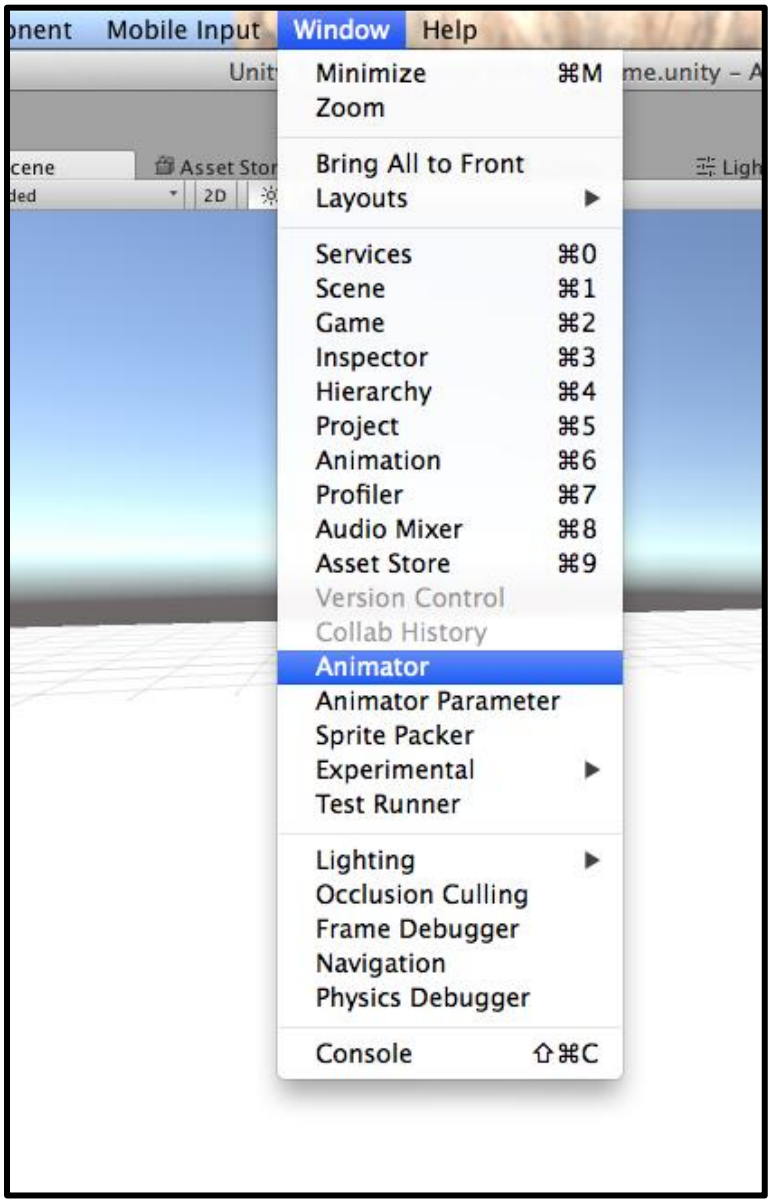
As you can see, the Avatar is just a way for Unity to find and configure the bones in the model.



Press done and go back to the Ethan model. Let's look at some of the settings in the Animator component.



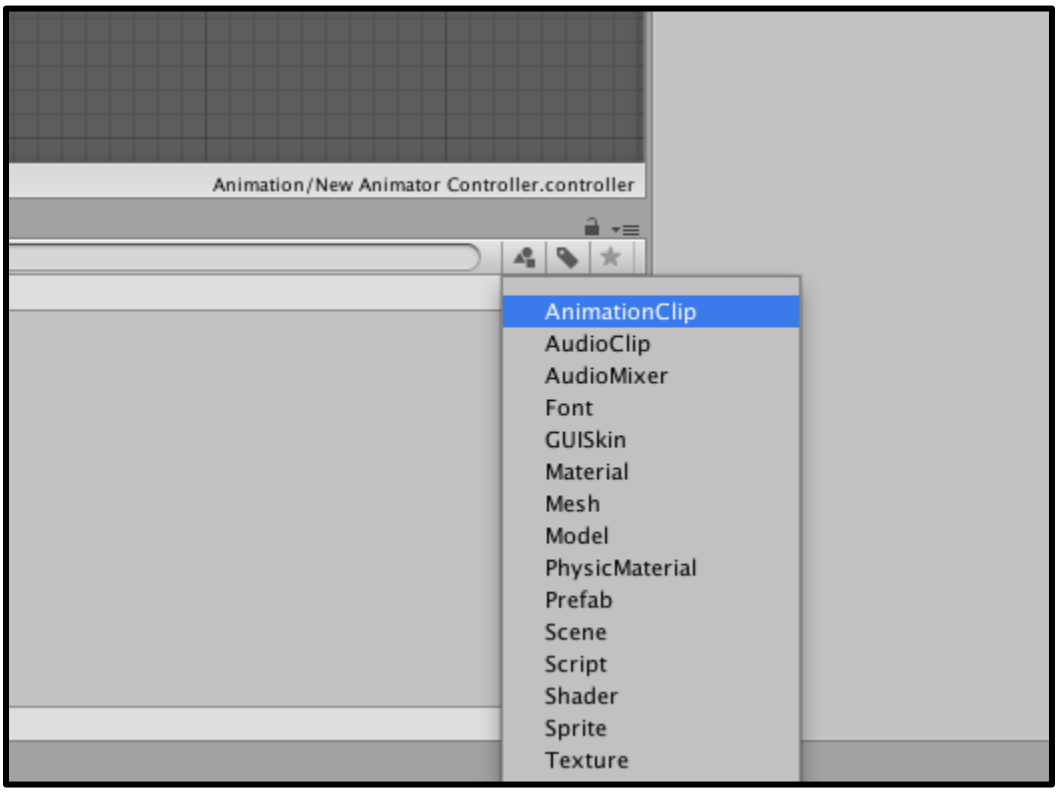
The "Apply Root Motion" boolean determines if Unity will use the movement in the animation or a script. For example, say I have a script that makes my character move forward. I have an animation attached of the character running in place. Since the script is moving the character not the animation then I would want Root Motion set to false. However, as we will see soon, we want to enable Root Motion in this case. With "Update Mode" you can determine how you want your animations to play. Such as with physics, with a fixed speed, or just normal. Culling mode allows you to determine if you want to keep animating even off screen. With the Ethan model selected, navigate to the "Animator" tab.



Here is where we add in the animations that will effect our model.

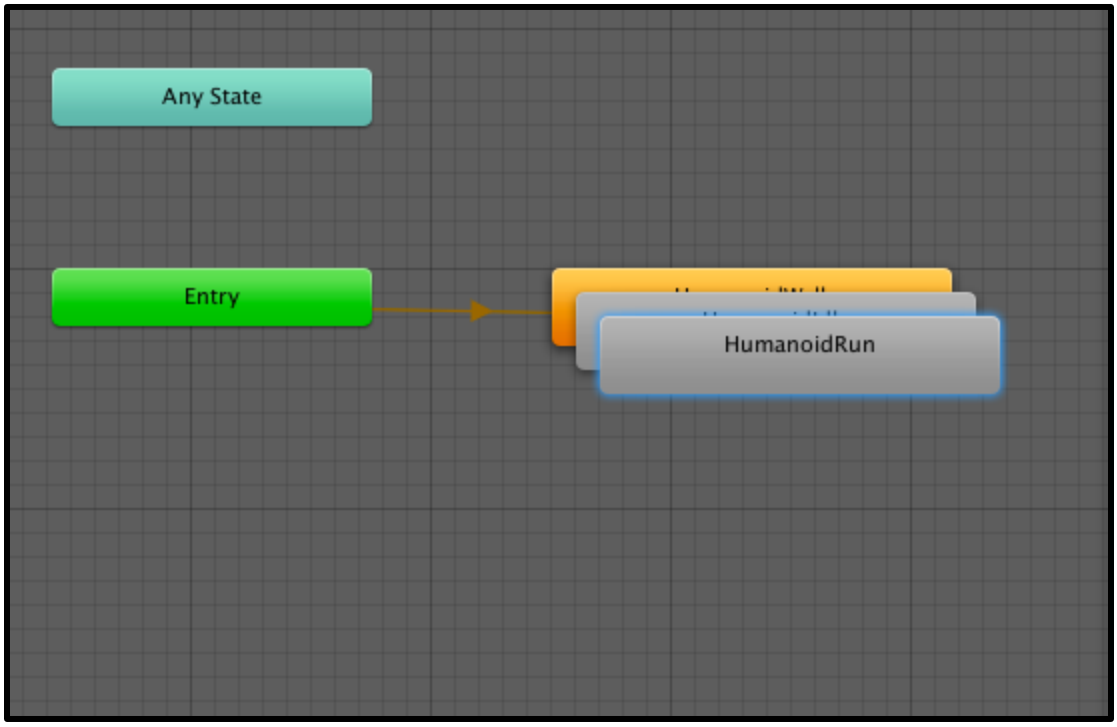
States

In your Project tab set the filter for animations.

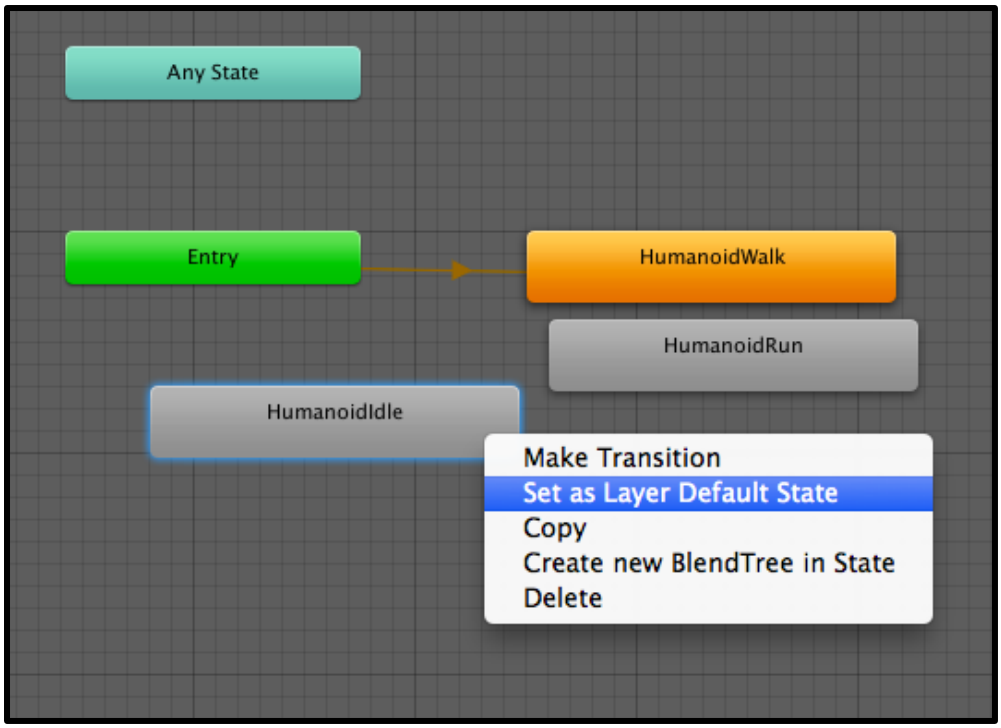


Look for "HumanoidIdle," "HumanoidWalk," and "HumanoidRun."

Drag these three into your Animator tab. Now there are three new boxes in our tab, those are the animations known as State Machines. Also, notice how one of the boxes is orange.

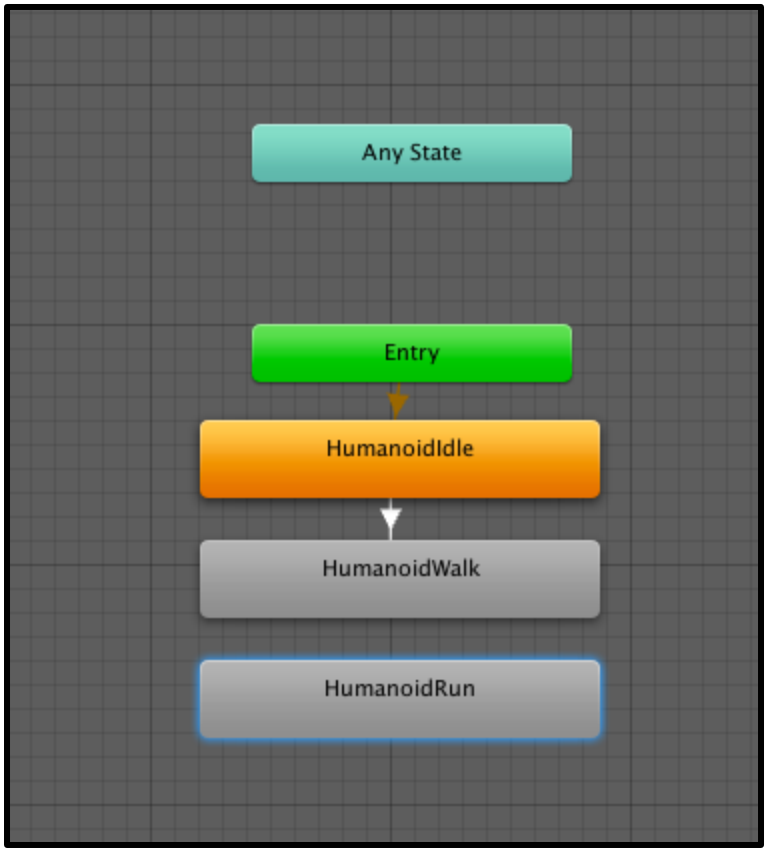


The orange one is the first one you selected in your Project tab. That is known a "Default State". We want the "HumanoidIdle" to be the Default State. To do this right click on "HumanoidIdle" and select "Set as Layer Default State".

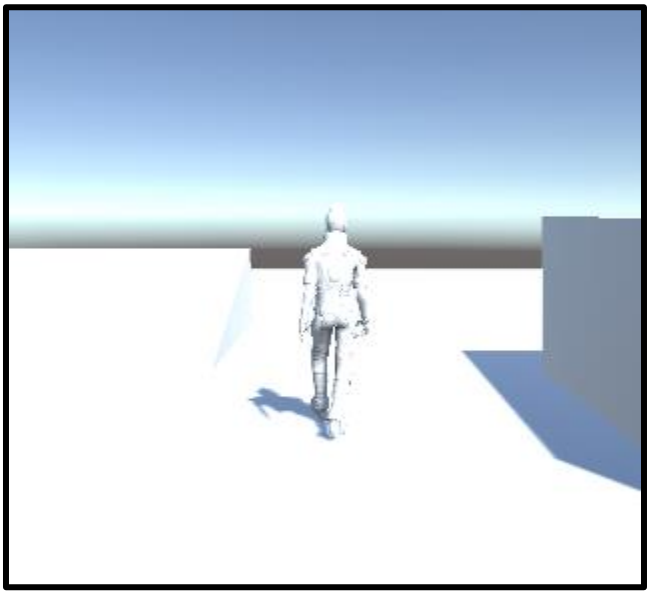


See how the Default State has an orange arrow coming from the box that says Entry. This means that as soon as the game starts it will go to the Default State. Press play and see what happens. Our model is idling!

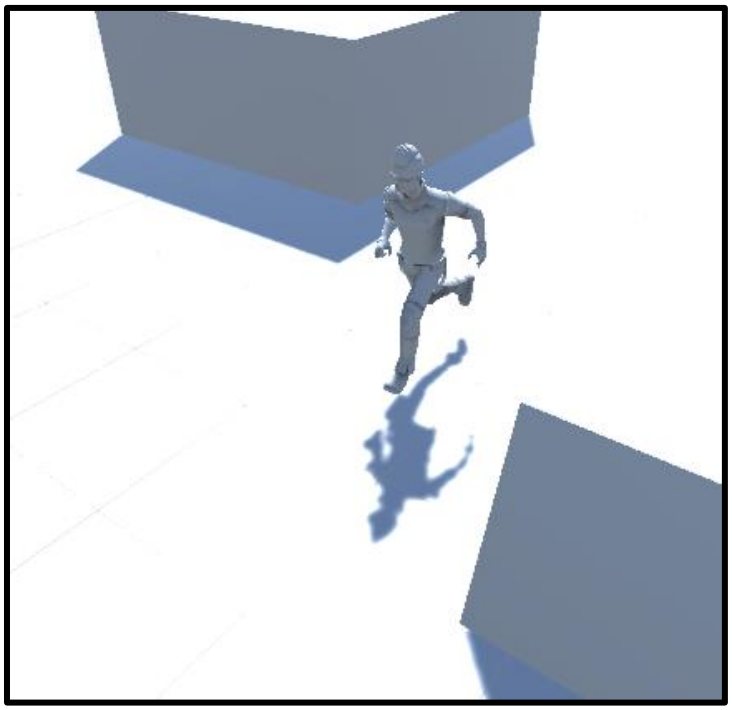
That is pretty neat, but it makes for a really boring game. We want our character to be able to run and walk around the scene. To do that, right click on the Default State and select "Make transition". Then click on the "HumanoidWalk" state.



Now press play and look what happens. Ethan idles for a bit then starts walking.

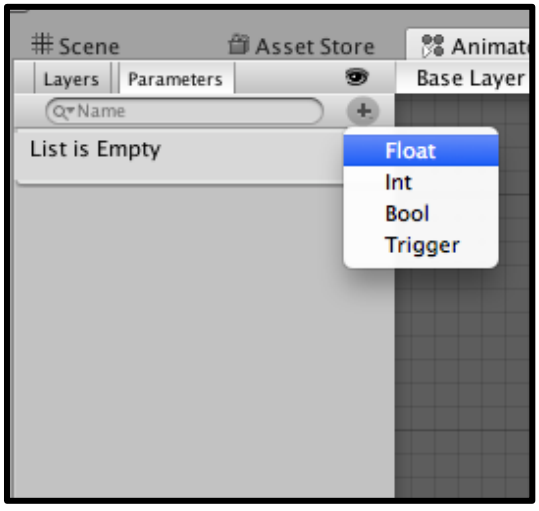


Now do the same thing for the run state. Then press play and see what happens. Our player idles, walks, then runs.



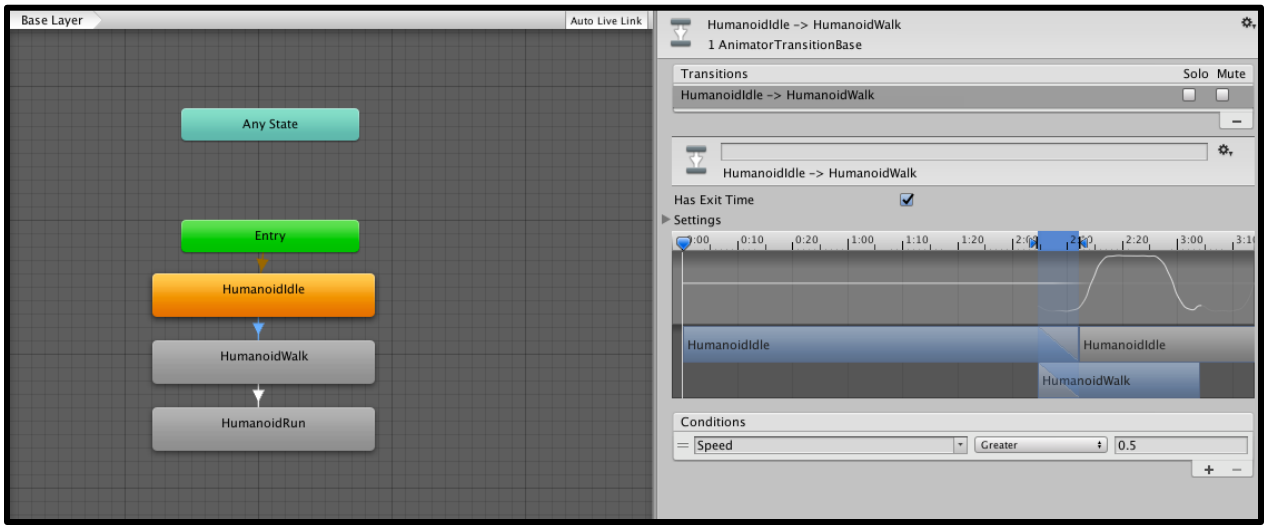
Parameters and Transitions

We have our model animating but it lacks control. It idles and runs without us doing anything. The way we add control is through Parameters. Parameters are variables that are defined within the animator that scripts can access and assign values to, this is how a script can interact with the Animator. There are three types: Int (or Integer), Float, Boolean, and Trigger.



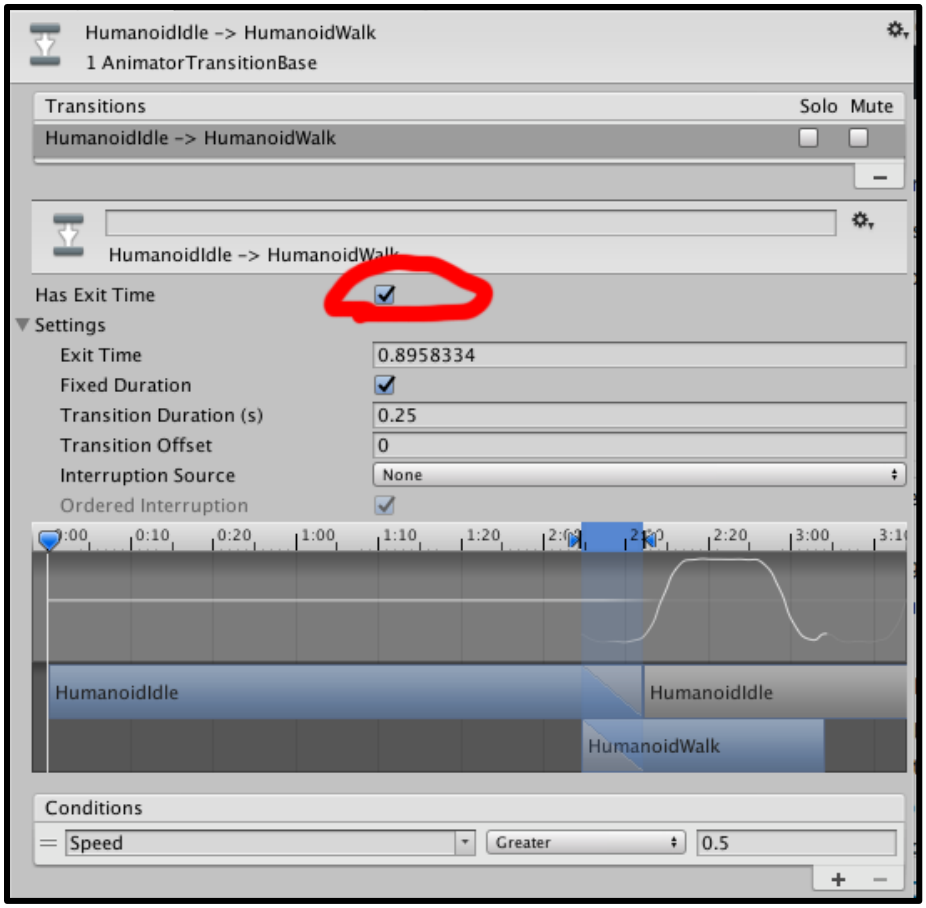
All of them are pretty self explanatory except for Trigger. A trigger is like a boolean but as soon as it is set to true it immediately goes back to false. We will realize it's usefulness soon. For now make a float parameter and call it Speed.

Then click on the idle to walk transition and set the condition to be if Speed is greater than 0.5. This means that it will only transition if Speed is greater than 0.5.

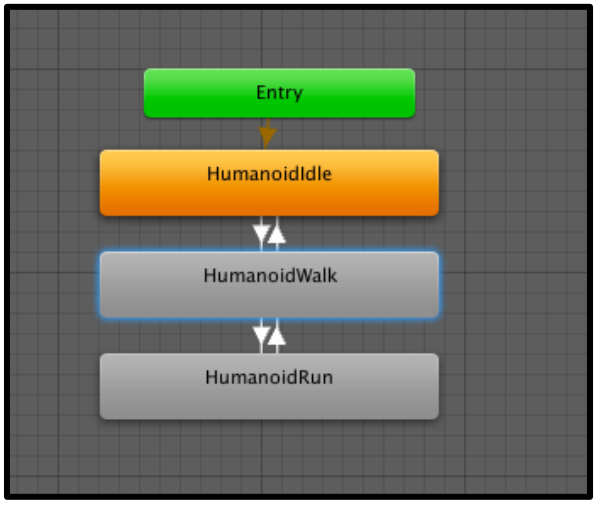


Then click on the walk to run transition and set the condition to if Speed is greater than 0.9. Now play, increase or decrease the Speed parameter to see what happens.

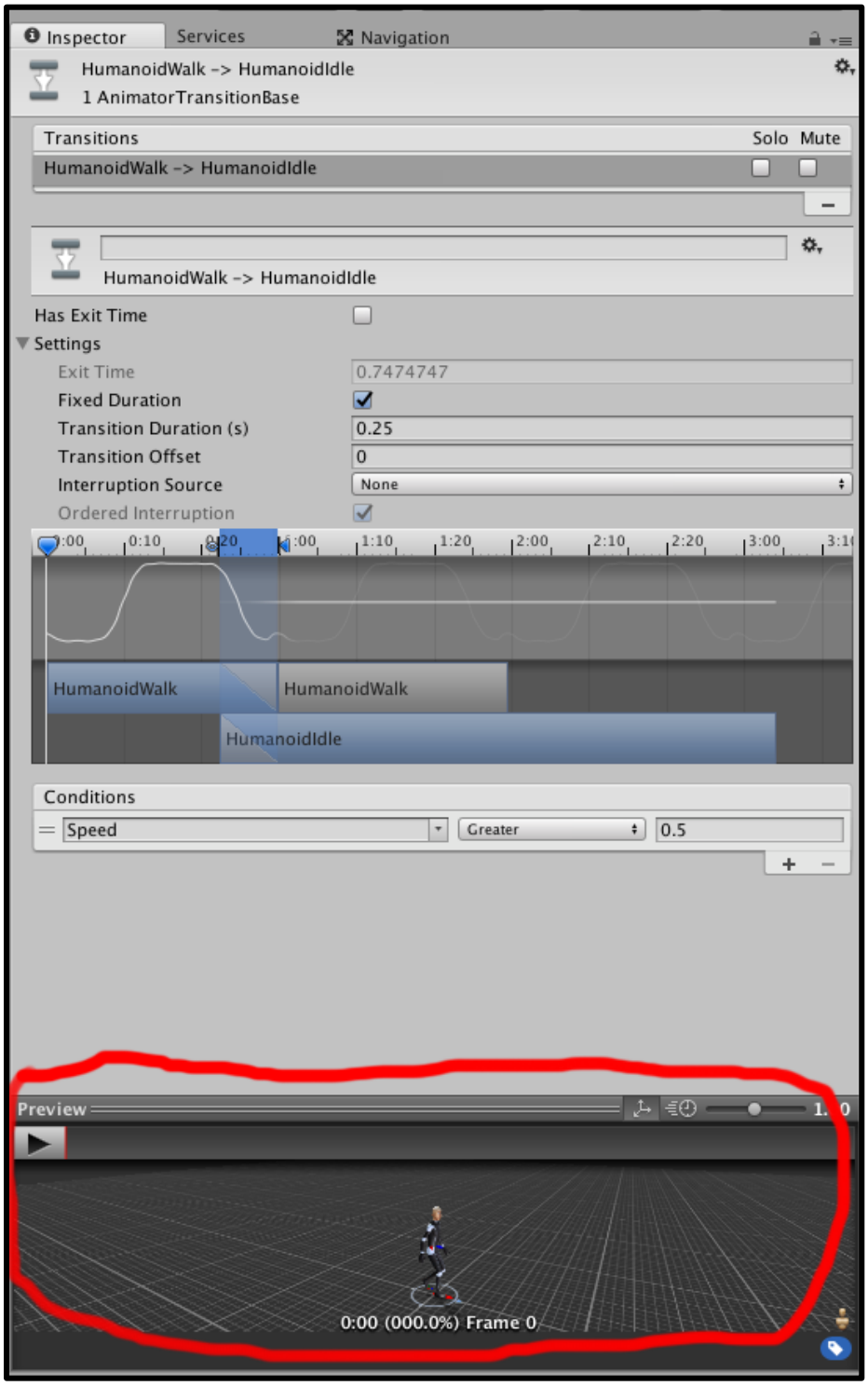
Okay not too bad but could use some tweaking. First off I am going to go to each transition and uncheck "Has Exit Time."



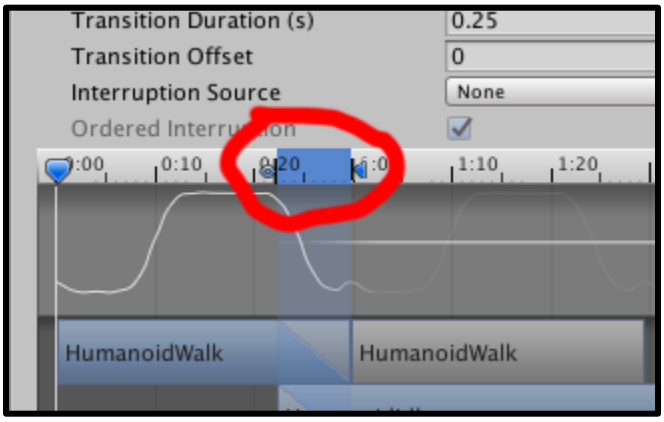
With this enabled the transition would go through the whole animation before transitioning. Which means there will be a delay if the player presses the run key. Second, make two other transitions (disabling Has Exit Time of course), one going from run to walk, and then walk to idle.



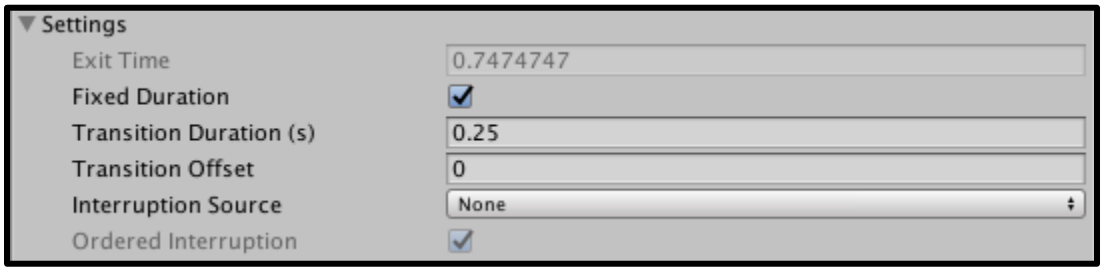
Set the Run to Walk condition to Speed is less than 0.9 and the condition of Walk to Idle to Speed is less than 0.5. Okay now that we have that done let's look at the transition settings. Notice how you can preview the transition?



The blue section is where and how long the transition will be. You can change at what point a certain clip will transition.



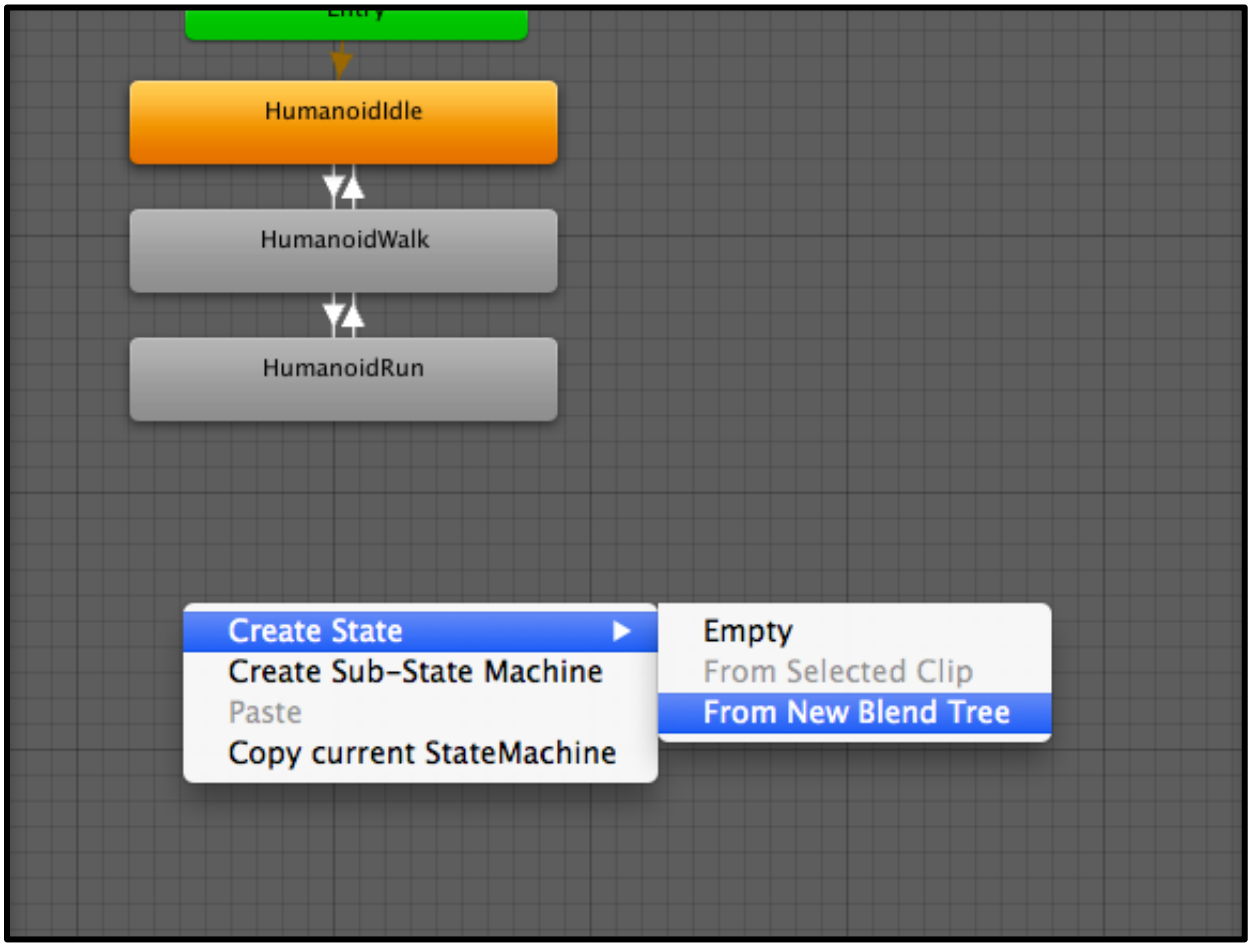
This can be used to make your transitions look more realistic. Open up Settings on a transition.



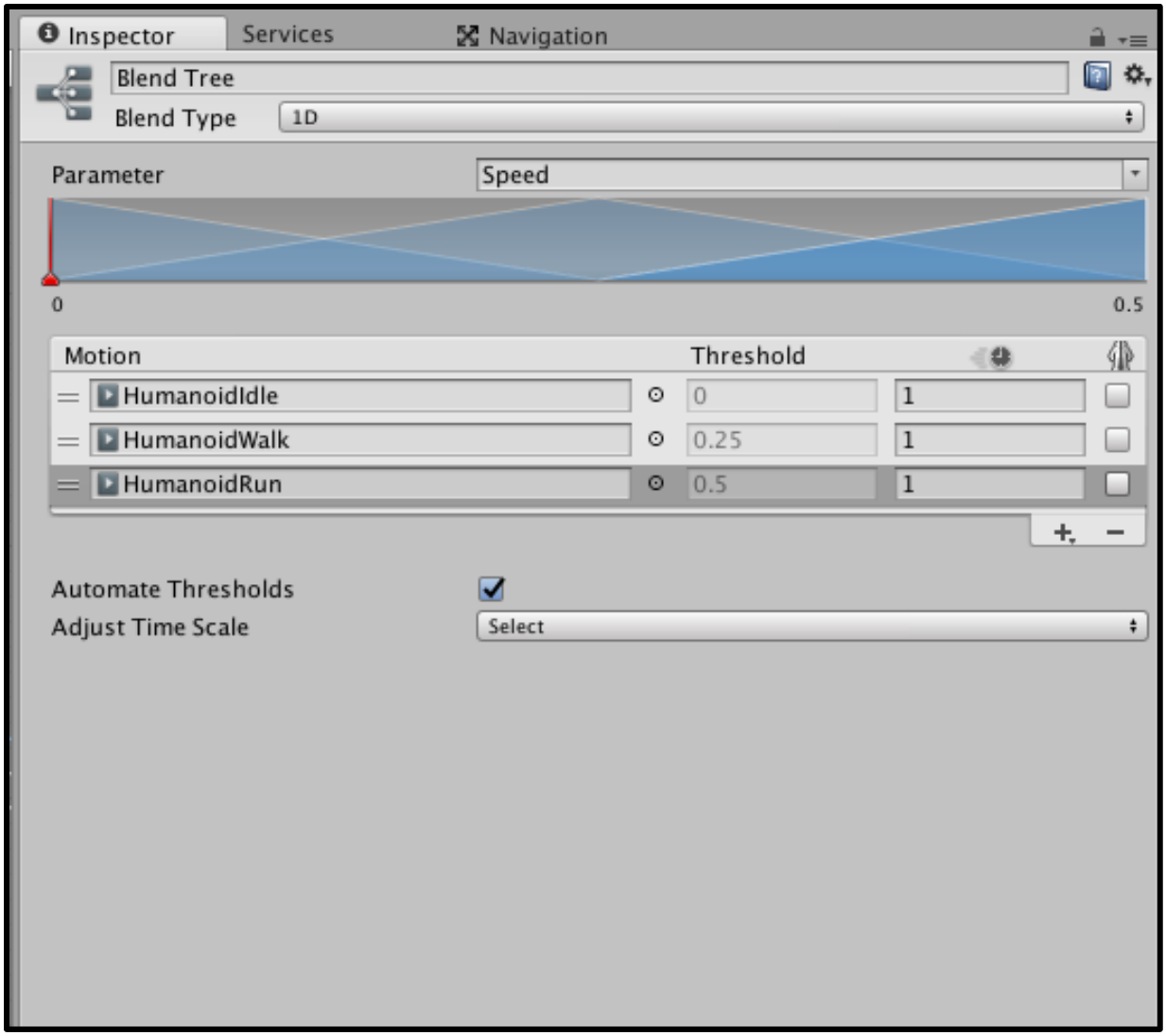
Exit time is how long before the transition starts to exit. With Fixed Duration enabled we can use the blue marker for transitioning. Transition Duration is pretty self explanatory, the length of the transition! Interruption Source is pretty interesting. If a transition's condition becomes false as it is transitioning, it has been interrupted. When Interruption Source is set to None it can be interrupted be the Any State (which we will look at later). When set to Current State it will go through the transition then be interrupted. Next State runs the next transition then can be interrupted. Current State Then Next State runs the current transition then the next transition before being interrupted. Next State Then Current State is basically the reverse, if an interruption occurs it runs the next transition then the current transition. It doesn't matter what you set these settings to because we will actually not be using it!

Blend Trees 1D

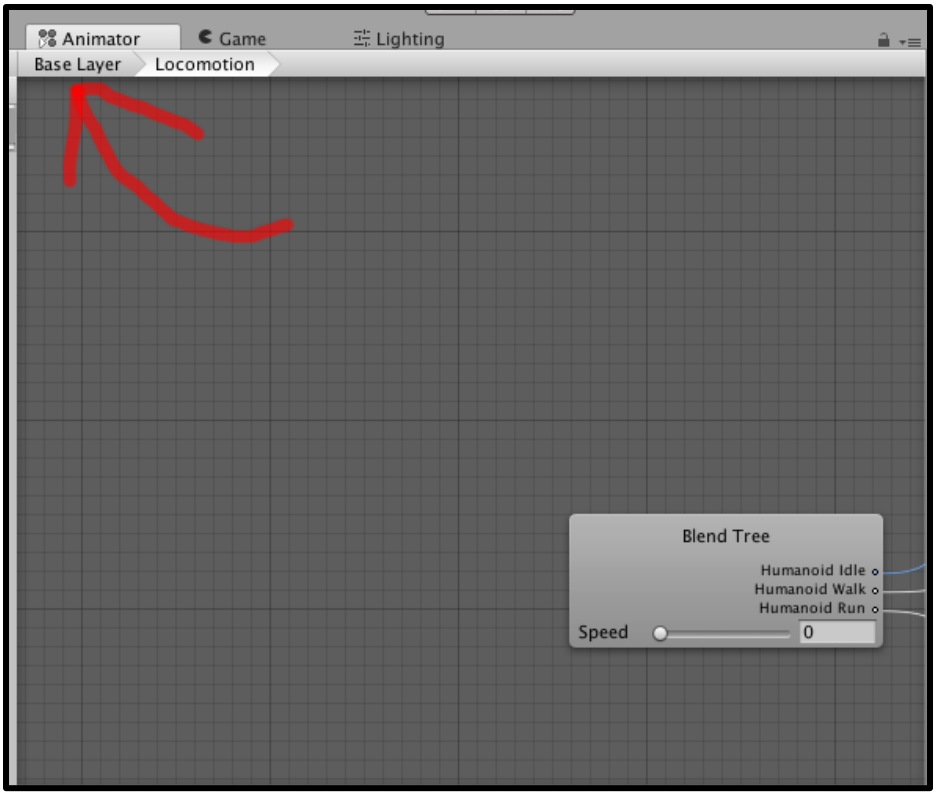
This locomotion system works but it isn't very modular. We can't add turning left or right very easily. A better and cleaner way to do this would be through Blend Trees. Blend Trees are a way to blend multiple animations together smoothly in varying degrees. To make a Blend Tree right click in the Animator and select Create State -> From New Blend Tree. Name it "Locomotion" or "OnGround".



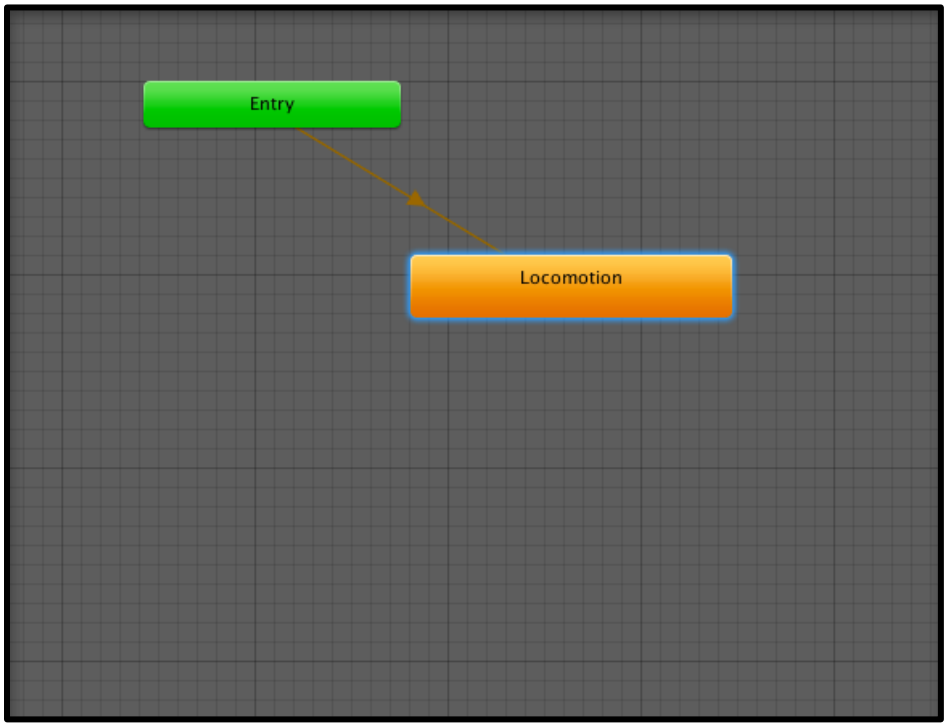
Double click on it, add three motion fields and put the three animations in it idle, walk, run.



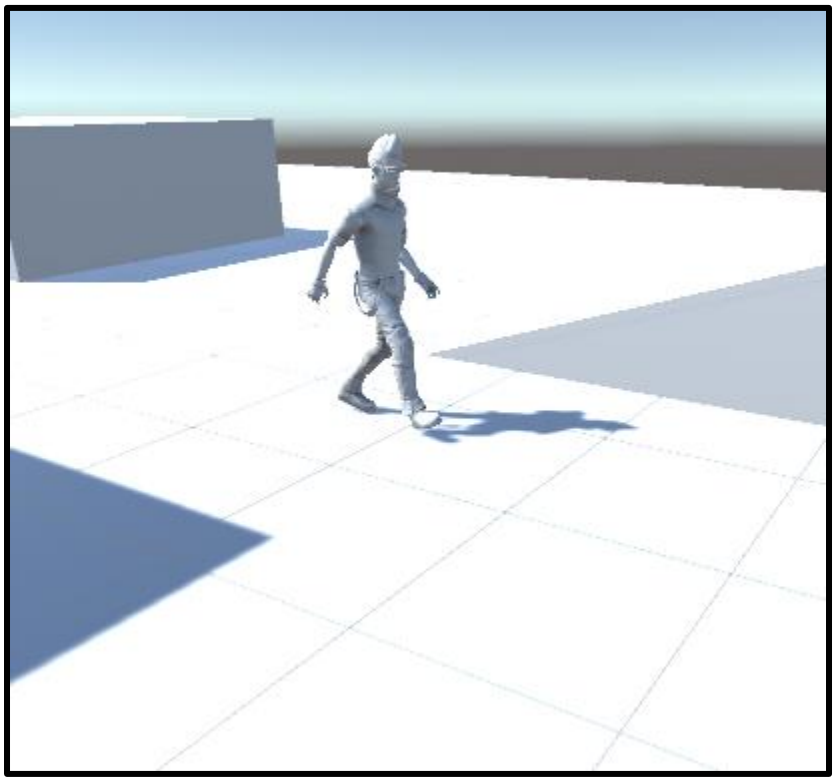
The thresholds are already setup for us but if you want to customize them uncheck "Automate fields". Basically, it is everything we have already done but in a nicer format. When the Speed parameter gets over 0.5 the character walks, when Speed equals 1 the character runs. Go back to the Base Layer and delete the three other states.



Then make a transition (unless it does it for you) from entry to our Locomotion Blend Tree.

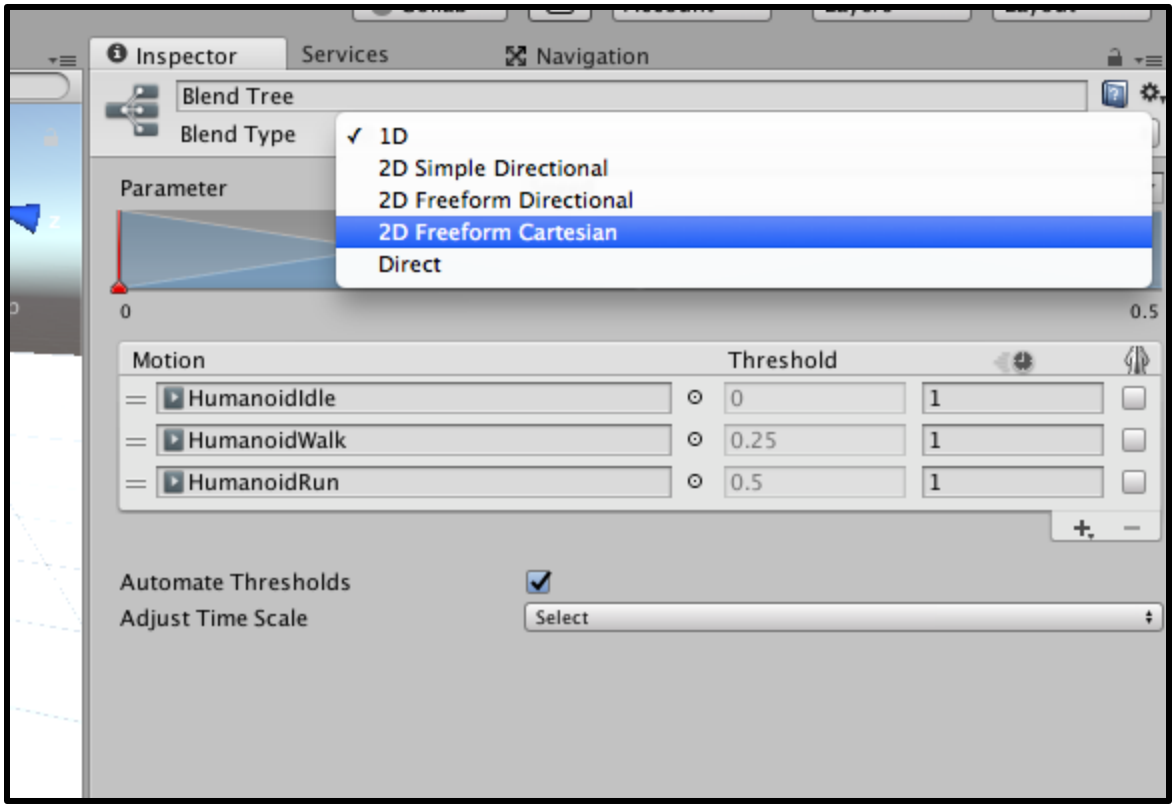


Now play and see what happens. Looks good, if you notice it blends the walk and run animation together if Speed is between 0.5 and 1.

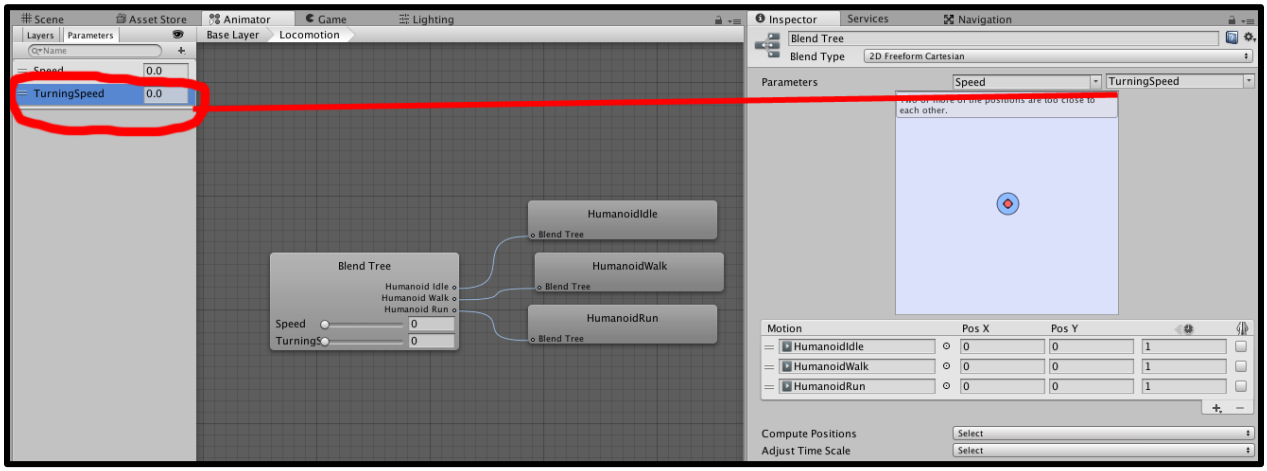


Making Our Character Turn

So our character can run, walk, and idle but it isn't very exciting. Unless you have a game where the character only runs forward (which, who does?) it isn't even remotely useful. We need our character to be able to turn left and right (obviously!). In order to do this go back to our Locomotion Blend Tree and set the type to "2D Freeform Cartesian." A Blend Tree has five different types of blending modes: 1D, 2D Simple Directional, 2D Freeform Directional, 2D Freeform Cartesian, and Directional.

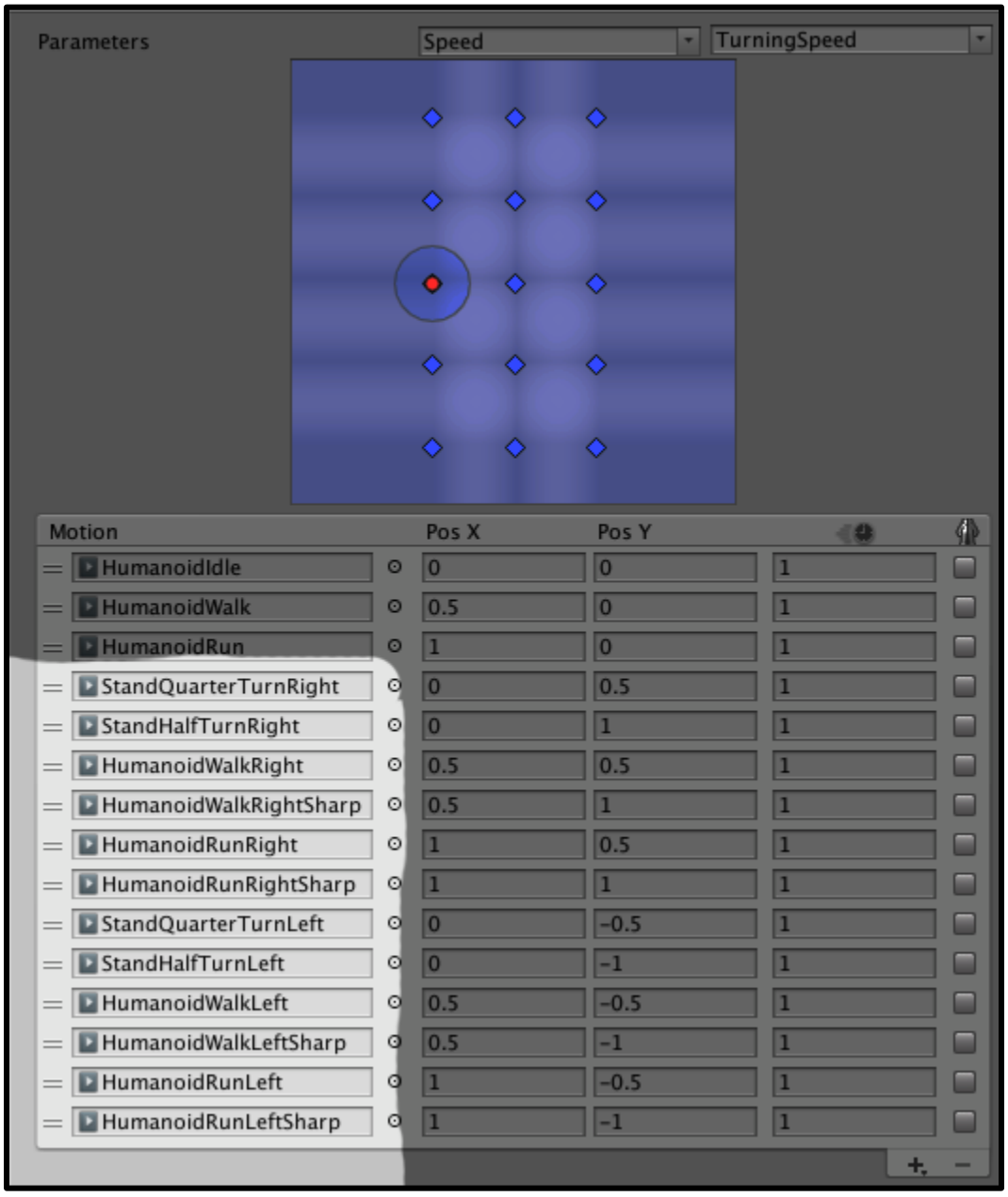


We have already looked at 1D. 1D uses just one parameter to check with, while any of the other types require two. 2D Simple Directional is if you need something slightly more complicated than 1D. An example would be if you just need Walk Forward, Walk Right, or Walk Left motions. 2D Freeform Directional is slightly more complicated. It would be able to handle, Walk Forward, Run Forward, Walk Right, Run Right etc. 2D Freeform Cartesian adds another level of complexity to the mix. It allows Run, Walk, and Turning. Before we do anything though we need to create a new float parameter called "TurningSpeed" and put it in the other parameter slot.

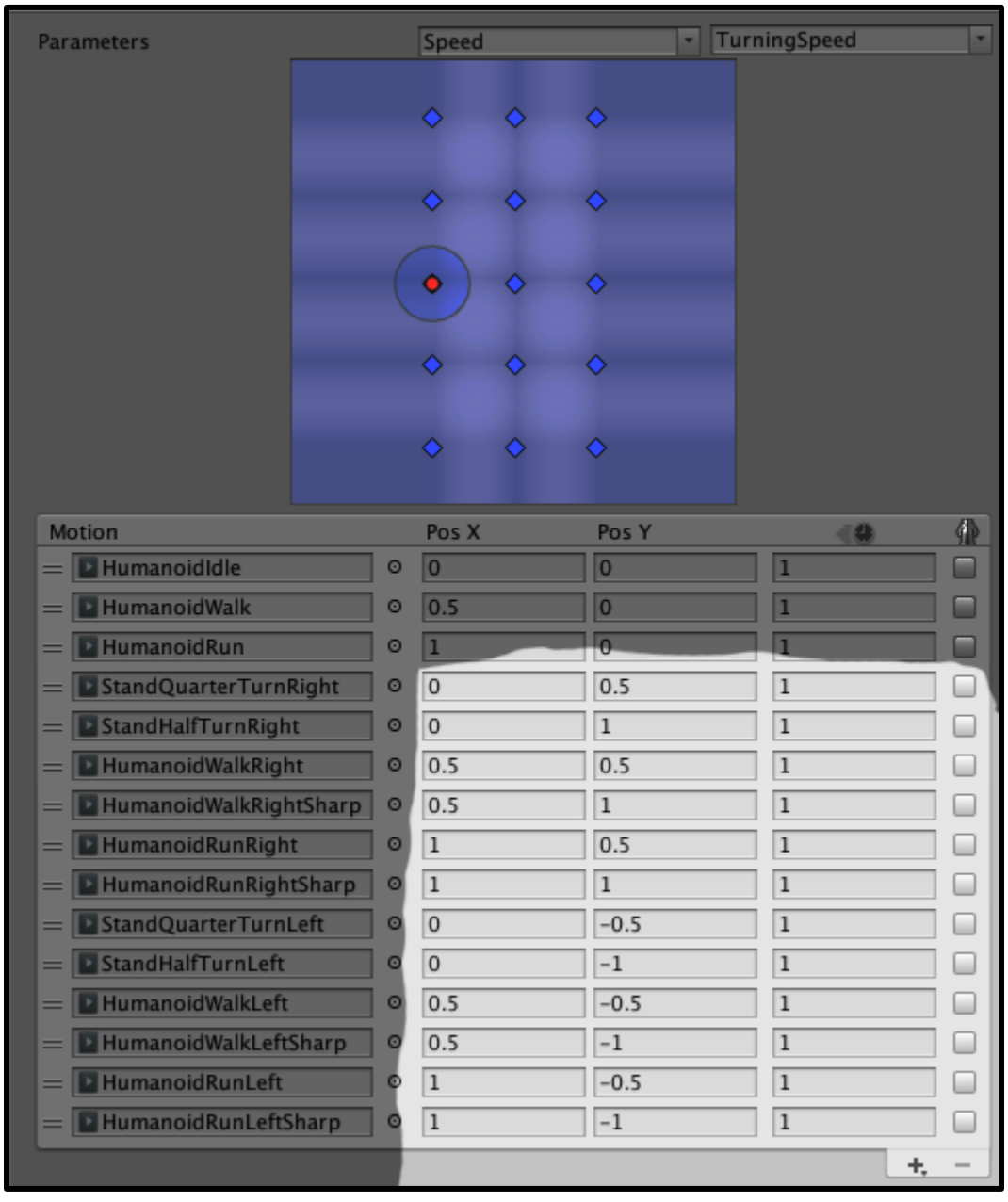


Now these next steps may not make much sense but stick with me and hopefully I can explain it well enough. Add twelve more motion slots. Look for and assign these animations to it in this order:

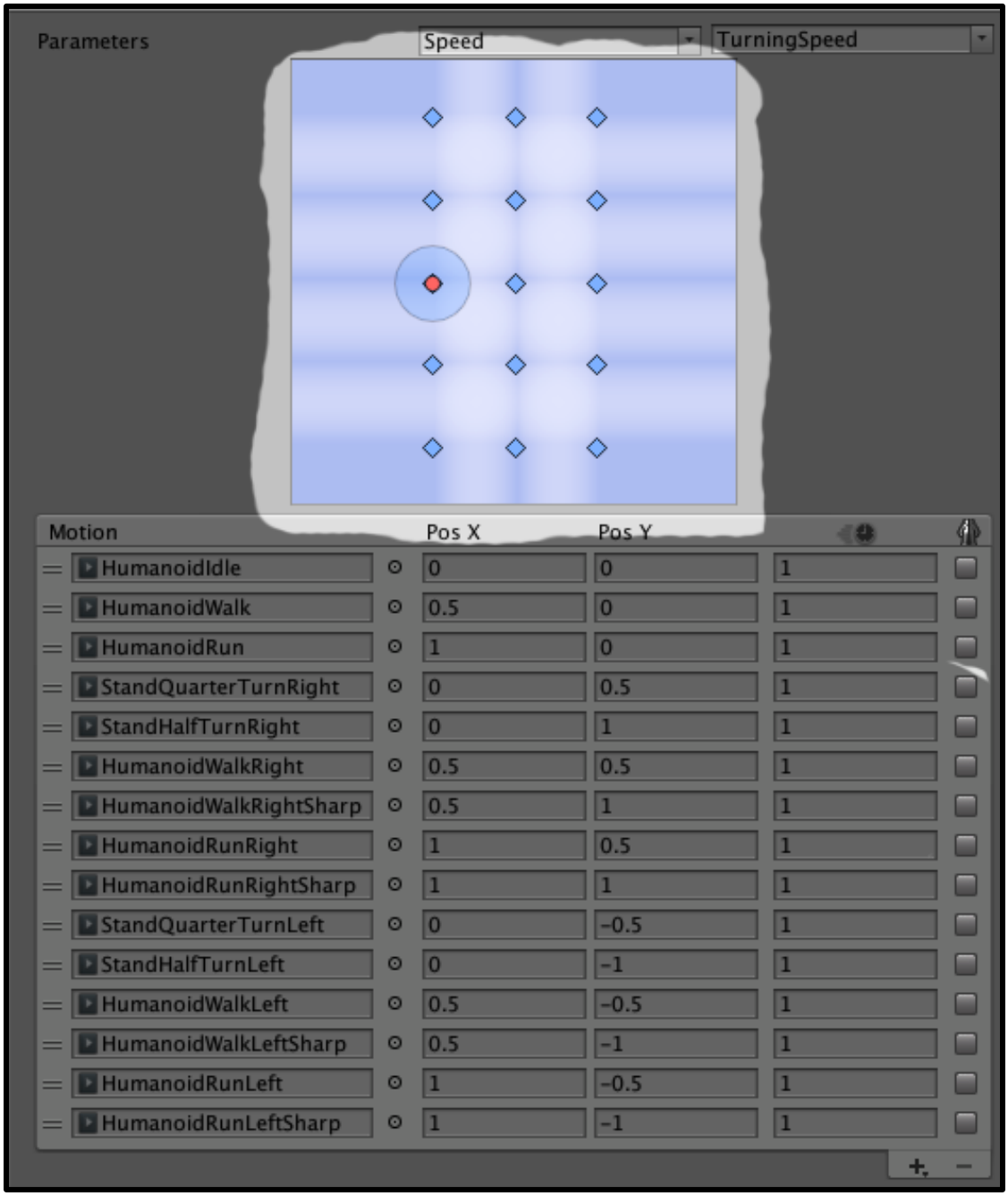
"StandQuarterTurnRight", "StandHalfTurnRight", "HumanoidWalkRight", "HumanoidWalkRightSharp", "HumanoidRunRight", "HumanoidRunRightSharp", "StandQuarterTurnLeft", "StandHalfTurnLeft", "HumanoidWalkLeft", "HumanoidWalkLeftSharp", "HumanoidLeftRight", and "HumanoidRunLeftSharp."



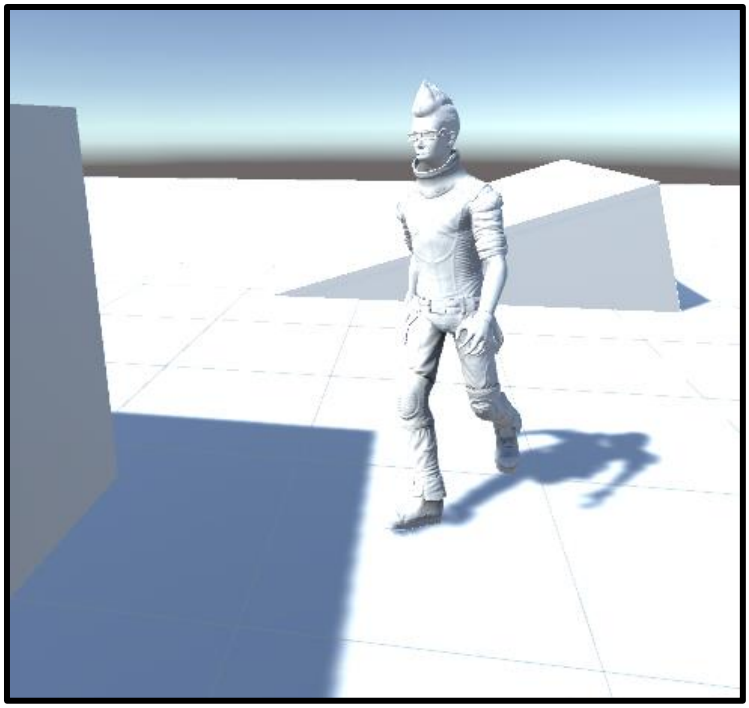
Then assign the values to look like the following:



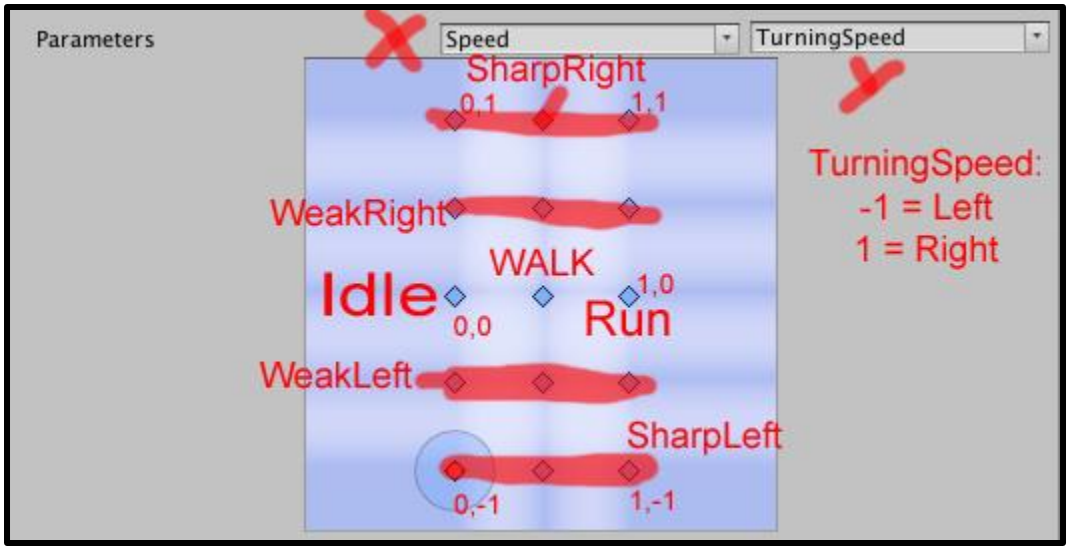
Your graph should look like this:



Okay, hopefully you are still with us after that huge list of unexplained tasks. If you did everything you can now play and change the values for Speed and TurnSpeed to see what happens.



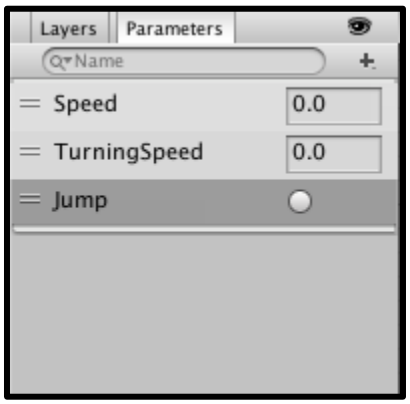
Pretty neat! But why does it do that? To answer this you need to understand how 2D Freeform Cartesian works. Think of it like a graph. The Speed parameter is the x value and TurnSpeed is the y value. If the both values equal 0 then the player is standing still. If TurnSpeed is -1 the character turns left. If it is 1 the player turns right, all while blending between "StandQuarterTurnRight" or "StandHalfTurnRight." If Speed equals 1 the player runs, 0.5 and the character walks. If TurnSpeed is 1 and Speed is 1 then the player runs right. If both equal 0.5 the player walks right. If you understand that then you should understand why we set the values to be what they are. Look at this graph for extra visualization:



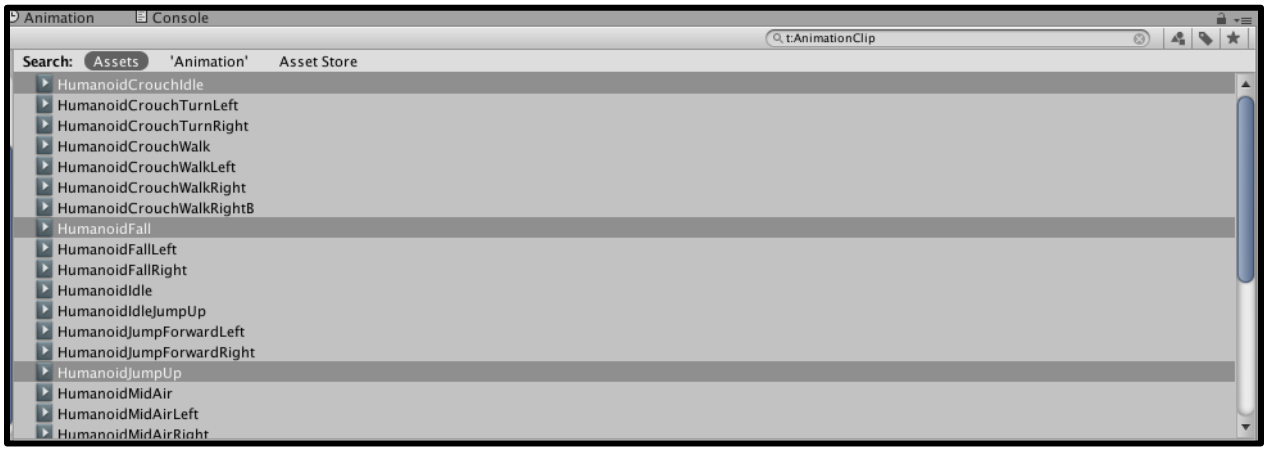
Hopefully, this has cleared up most questions about 2D Freeform Cartesian. If not don't worry! Just think about it, it'll come to you.

The "Jumping" sub-state machine

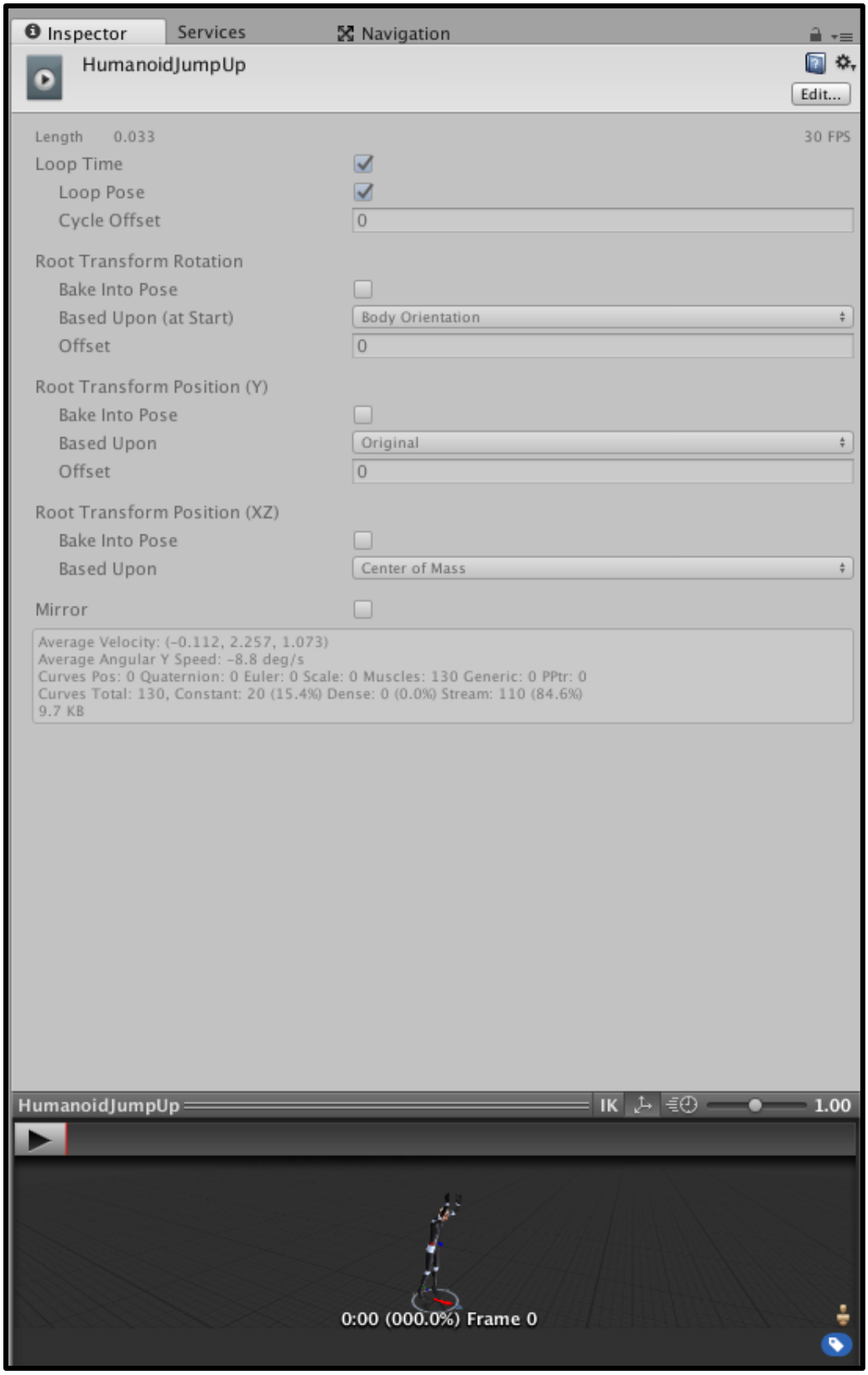
In order to make our character jump we need a few things. A jump up pose, a fall pose, a landed pose, a way to check if the player presses the jump key, and a way to check if we hit the ground. I am going to do the last two first. Create a new trigger called "Jump".

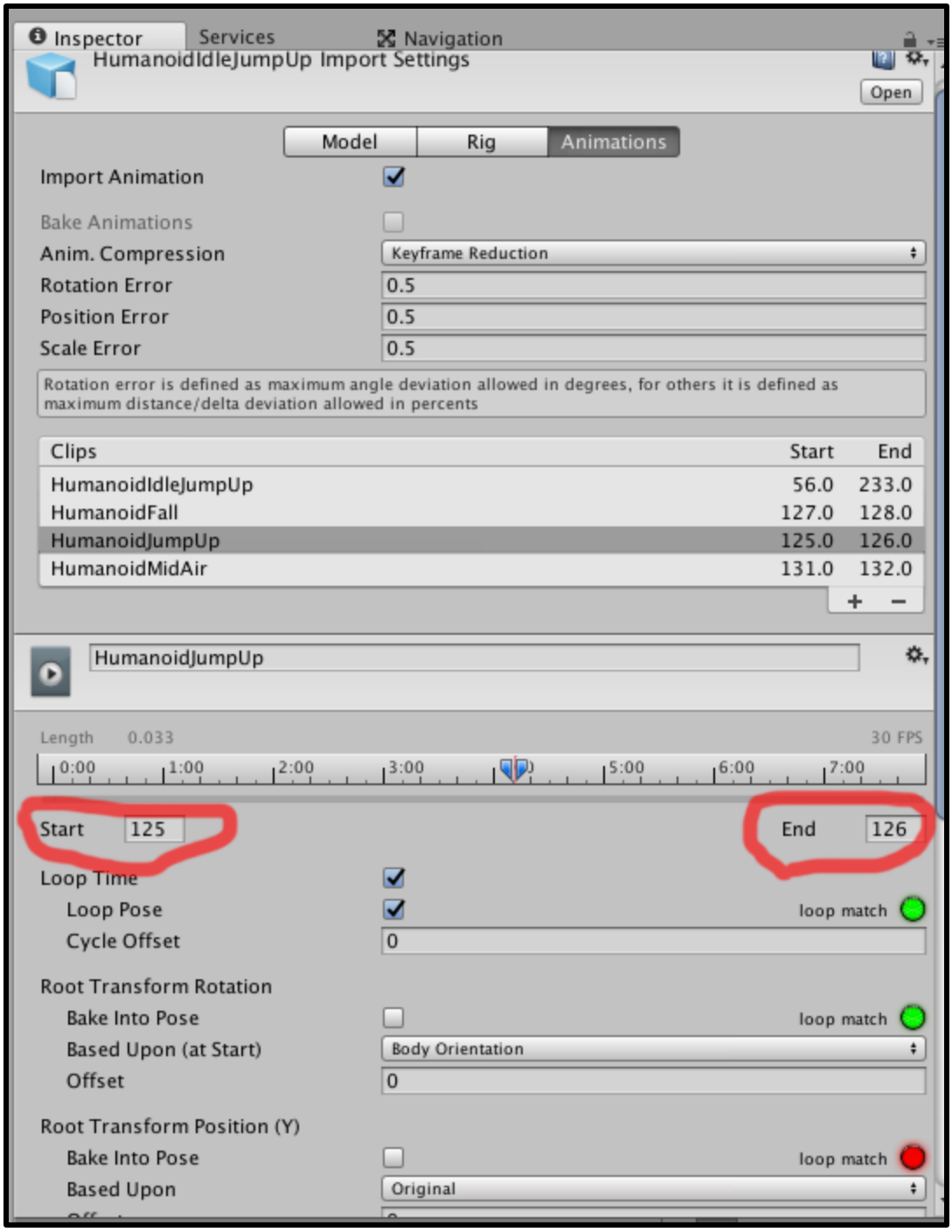


And then create a boolean called "grounded". Okay that is done now we can move on. Filter search for the following animations "HumanoidCrouchIdle" (this is our landed pose), "HumanoidFall", and "HumanoidJumpUp".

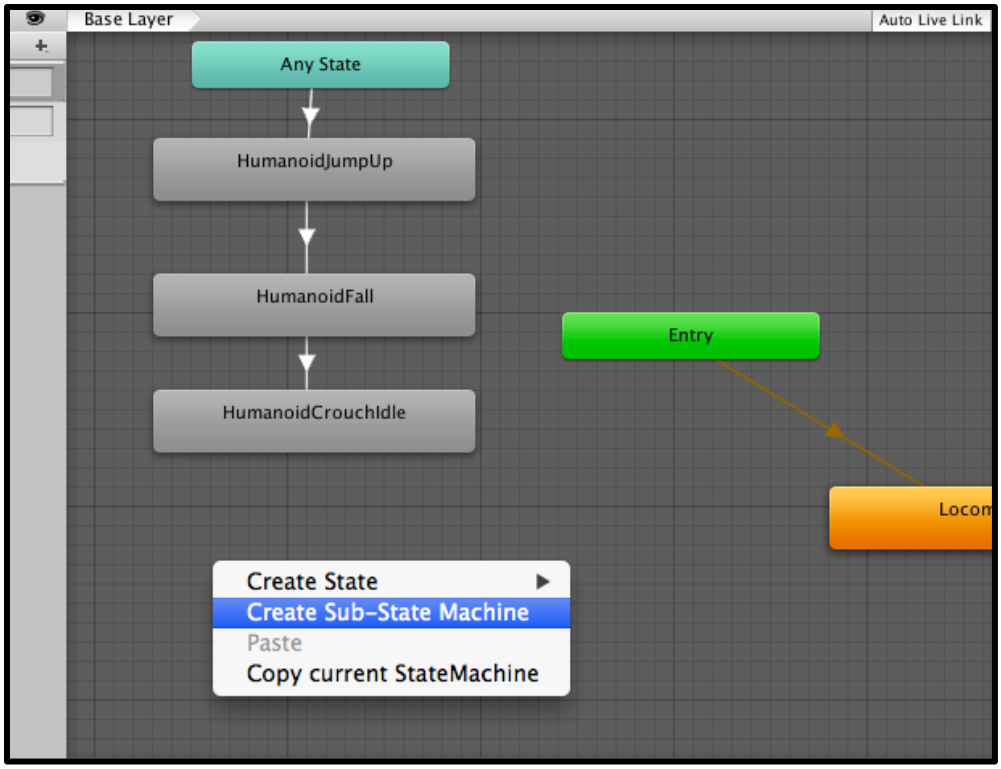


Before we drag those in we need to change something. If you preview one of the animations you will notice an unwanted twitch. To fix this click edit, then set the motion to be only one frame long.

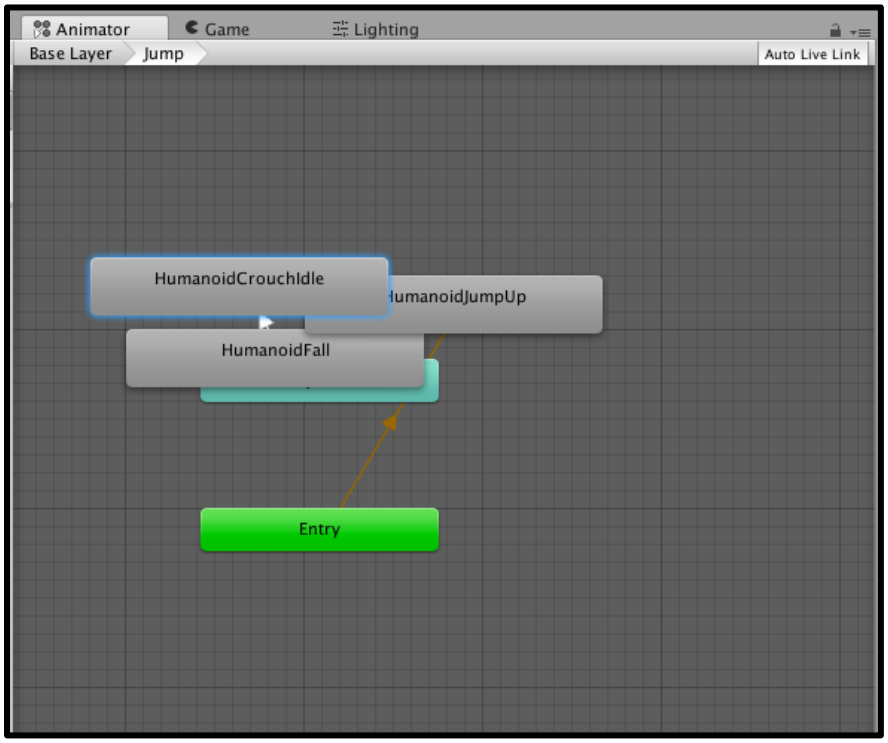




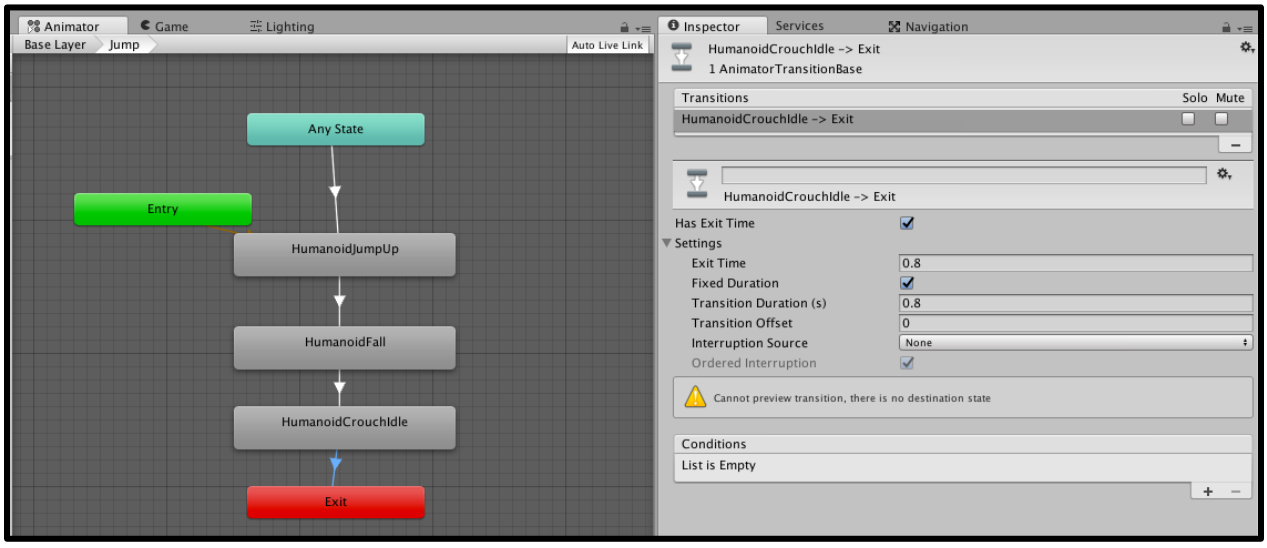
But we still can't drag them in yet! We first need to create something called a Sub-State Machine. A Sub-State Machine is a bunch of motions put together that act as one. This is a good thing to use for jumping actions since a jump action isn't really just one action but a bunch of them put together. Right click in the animator and press "Create Sub-State Machine".



Name it jump. Now drag all our Jump poses onto the new Sub-State Machine.



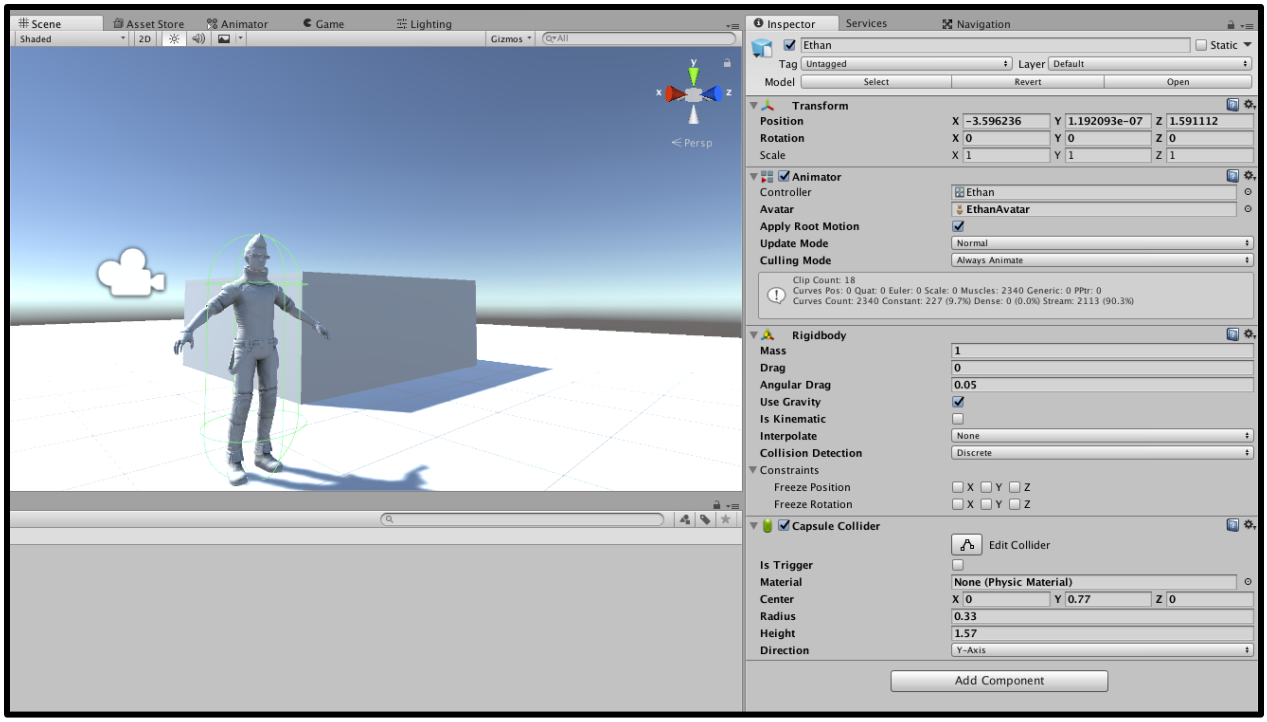
Now we can start constructing our Jump action. Make a transition from Any State to the JumpUp pose. The Any State is a very useful tool. It will transition into a motion, if the condition is true, regardless of where that motion is. Set the condition to be our Jump trigger. Then make a transition from JumpUp to Fall. Make a Fall to Crouch transition but remember we only want it to transition if it hits the ground. Set the condition to be if grounded equals true and uncheck Has Exit Time. Where do we transition to after this? From crouch into the Exit state. When something transitions into the Exit state on a Sub-State Machine it will go back to the Entry state in the Base Layer.



Okay, we have our Jump action setup! Although we still need to add physics, play and tweak the settings to get the look you're after.

Scripting Our Animator

Now we come to the last and arguably most important part of this lesson, scripting. We have already seen how pretty much everything in the Animator is controlled by parameters. So our script should control those parameters. Before we start we need a capsule collider and a Rigidbody on our character. Set the capsule collider to roughly surround our character.



Then create a new script called "PlayerController". The way we control parameters through script is first getting access to the Animator:

```
1  private Animator thisAnim;
2
3  Start (){
4      thisAnim = GetComponent<Animator>();
5  }
```

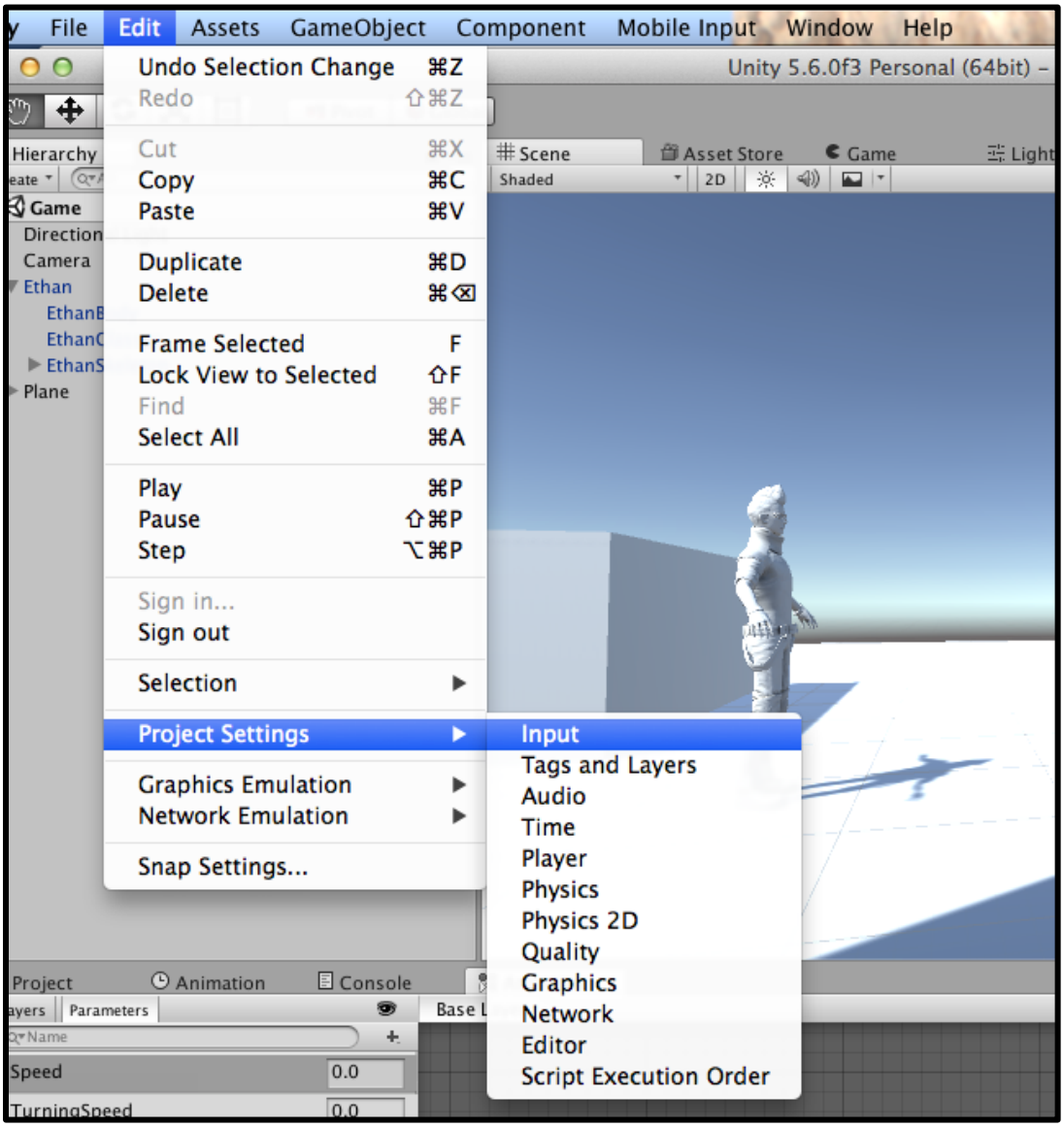
Then write something like the following command:

```
1  thisAnim.SetTrigger ("Jump");
```

Let's think about what we need to do. First, we need a forward and backward input:

```
1 void Update (){
2 var v = Input.GetAxis ("Vertical");
3 thisAnim.SetFloat ("Speed", v);
4 }
```

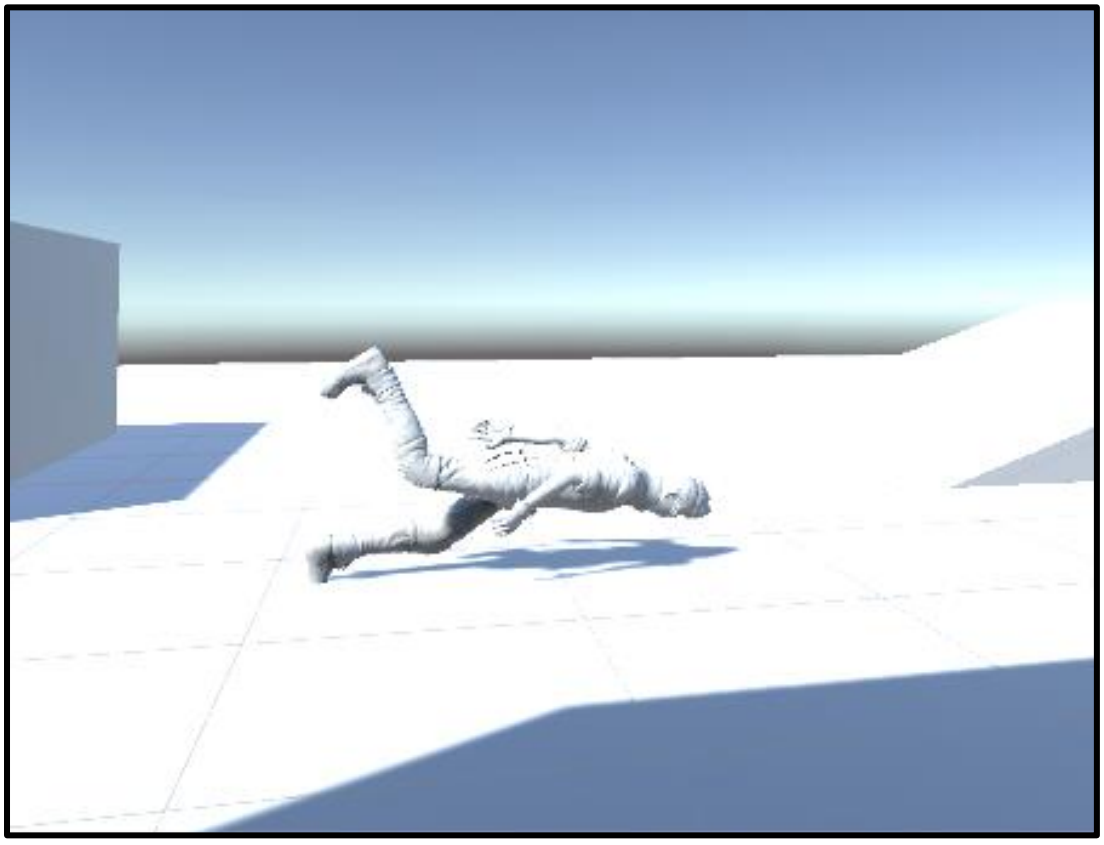
Use the Unity Input axis Vertical, which by default is the up and down arrows, then put it in a variable and assign that variable to the parameter "Speed". The Input settings can be configured here:



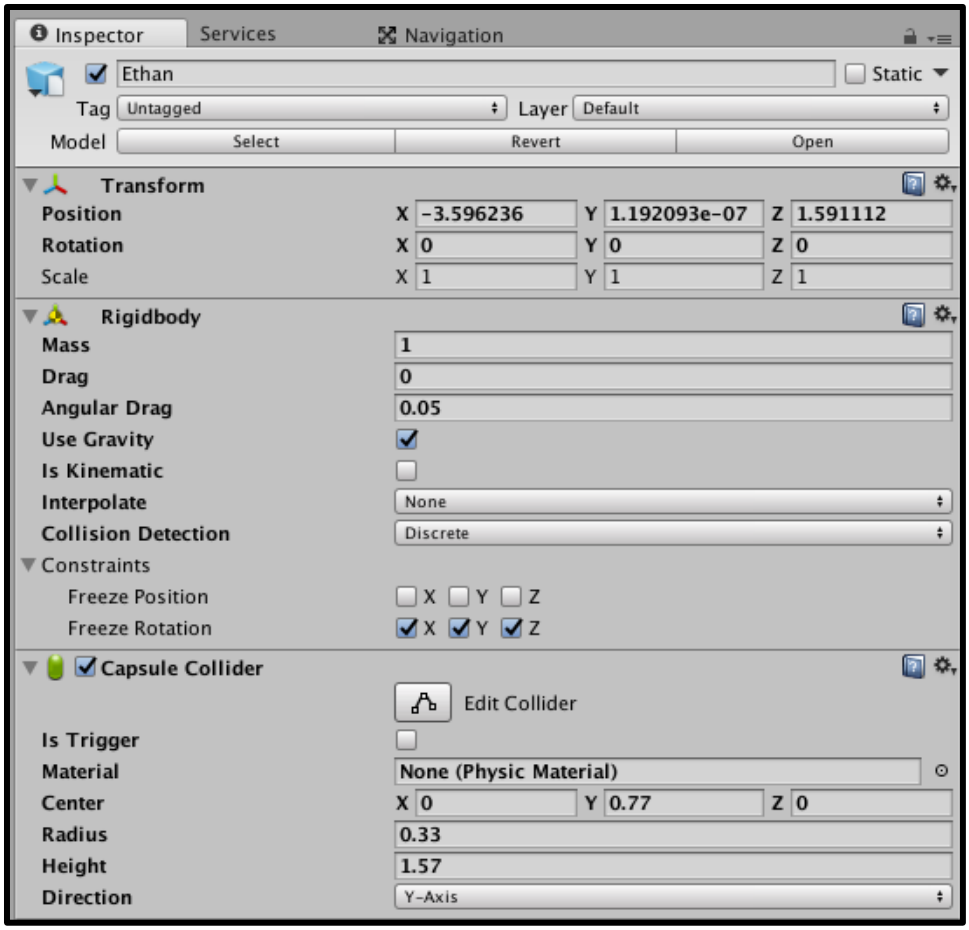
Next we need a turning input:

```
1 void Update () {
2     var h = Input.GetAxis ("Horizontal");
3     var v = Input.GetAxis ("Vertical");
4     thisAnim.SetFloat ("Speed", v);
5     thisAnim.SetFloat ("TurningSpeed", h);
6 }
```

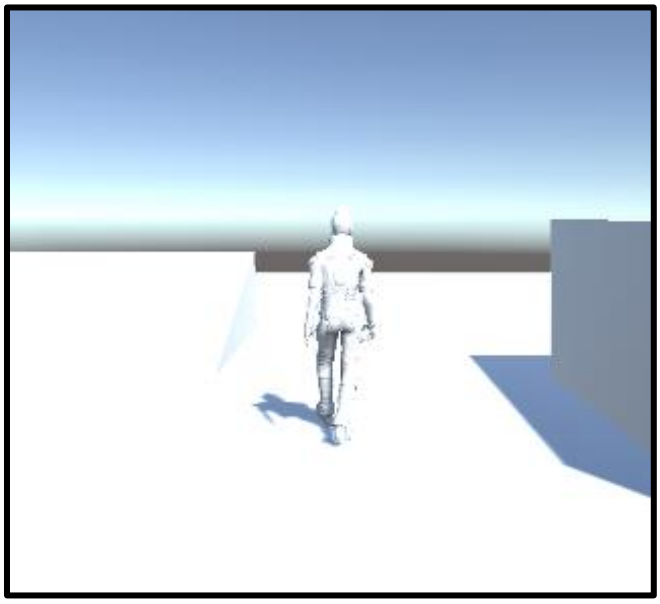
We use the axis Horizontal, which by default is the left and right arrows, just like we used the Vertical axis only this time assigning it to TurningSpeed. Play and see what happens.



Okay our character does a face plant. To fix this, constrain our rotation on the X, Y, and Z axis.



Now play and see what happens.



Wow, just wow. It looks really good! Now we just need to add the jumping input. In order to jump we need a Rigidbody to act force upon, trigger the Jump parameter, check to see when our character lands, and set "grounded" to be true if the player does land. First, we get the Rigidbody and trigger jump with the input button "Jump":

```
private Animator thisAnim;
private Rigidbody rigid;
public float JumpForce = 500;

void Start () {
    thisAnim = GetComponent<Animator> ();
    rigid = GetComponent<Rigidbody> ();
    }
 void Update () {
        var h = Input.GetAxis ("Horizontal");
        var v = Input.GetAxis ("Vertical");

        thisAnim.SetFloat ("Speed", v);
        thisAnim.SetFloat ("TurningSpeed", h);
        if (Input.GetButtonDown ("Jump")) {
            rigid.AddForce (Vector3.up * JumpForce);
            thisAnim.SetTrigger ("Jump");
        }
}
```

"Jump" by default is set to the space bar in the Input Manager. Then we raycast downwards to see if we have hit the ground while changing the value of "grounded":

```
if (Physics.Raycast (transform.position + (Vector3.up * 0.1f),
    Vector3.down, groundDistance, whatIsGround)) {
    thisAnim.SetBool ("grounded", true);
    thisAnim.applyRootMotion = true;
} else {
    thisAnim.SetBool ("grounded", false);
}
```

This part needs a float, for how far down check for ground, and a layer mask, to define what ground is. A Layer Mask is a way to group objects, kind of like tags. Declare those here:

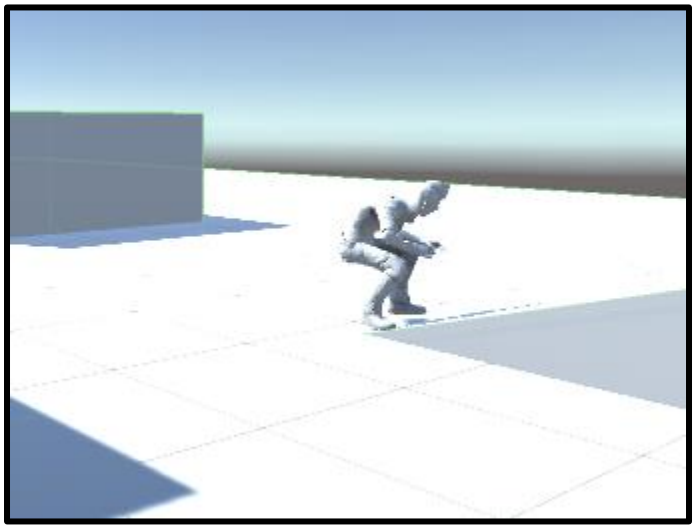
```
1  private Animator thisAnim;
2  private Rigidbody rigid;
3  public float groundDistance = 0.3f;
4  public float JumpForce = 500;
5  public LayerMask whatIsGround;
```

This is the full script:

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PlayerController : MonoBehaviour {
6      private Animator thisAnim;
7      private Rigidbody rigid;
8      public float groundDistance = 0.3f;
9      public float JumpForce = 500;
10     public LayerMask whatIsGround;
11
12     // Use this for initialization
13     void Start () {
14         thisAnim = GetComponent<Animator> ();
15         rigid = GetComponent<Rigidbody> ();
16     }
17
```

```
18      // Update is called once per frame
19      void Update () {
20          var h = Input.GetAxis ("Horizontal");
21          var v = Input.GetAxis ("Vertical");
22
23          thisAnim.SetFloat ("Speed", v);
24          thisAnim.SetFloat ("TurningSpeed", h);
25          if (Input.GetButtonDown ("Jump")) {
26              rigid.AddForce (Vector3.up * JumpForce);
27              thisAnim.SetTrigger ("Jump");
28          }
29          if (Physics.Raycast (transform.position + (Vector3.up * 0.1f),
30                      Vector3.down, groundDistance, whatIsGround)) {
31              thisAnim.SetBool ("grounded", true);
32              thisAnim.applyRootMotion = true;
33          } else {
34              thisAnim.SetBool ("grounded", false);
35          }
36
37      }
38
39  }
```

Make sure WhatIsGround is set to Default. Play and see what happens.



Yes! It looks amazing!

Conclusion

Wow! We have covered a lot! I hope this tutorial was helpful and that you get an understanding of how complicated the Unity Animator is.

Keep making great games!

Rigging a 2D Character in Unity

Introduction

2D animation has been around for over 100 years. Since its inception, this entertainment type has gone through quite a few different evolutions, each time becoming more and more complex. Now, in the latter part of the 20th-century to the present time, 2D animation is a huge part of video games - with thousands of games using characters and environments solely constrained to two dimensions. I personally like the look of a 2D game more than its 3D counterpart.

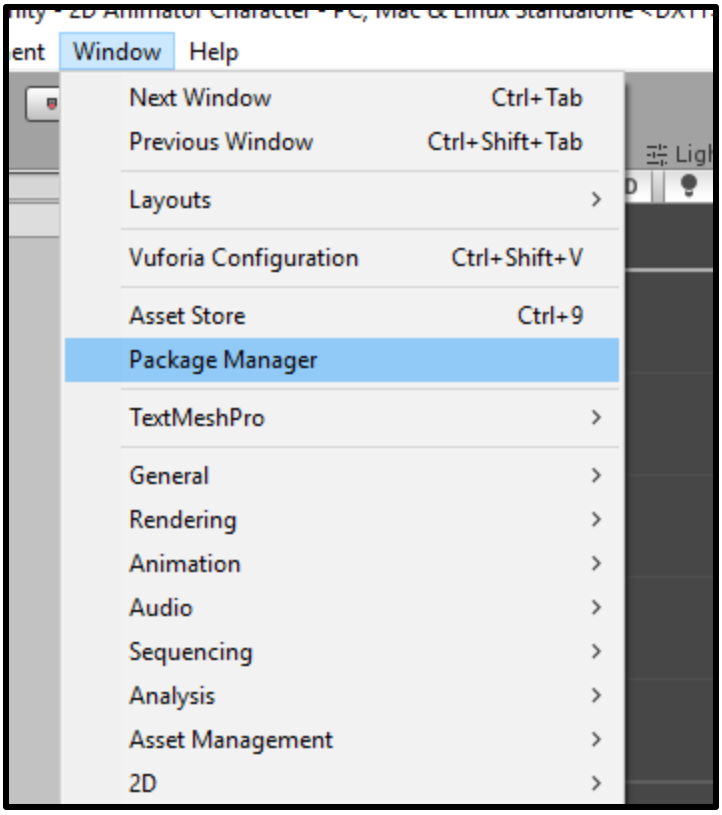
In this tutorial, you will learn how to take a 2D character, rig it, and use it in your own 2D game. You will learn Unity's sprite rigging system, along with how to use the Unity Animator for 2D characters. No previous knowledge of the Unity Animator will be necessary, however, I suggest you check out The Comprehensive Introduction to the Unity Animator tutorial, which goes more in-depth with the Unity Animator. Some experience of Unity itself (how to navigate, how to toggle the 2D and 3D view, how to import assets, etc.) is going to be necessary, however.

Assets

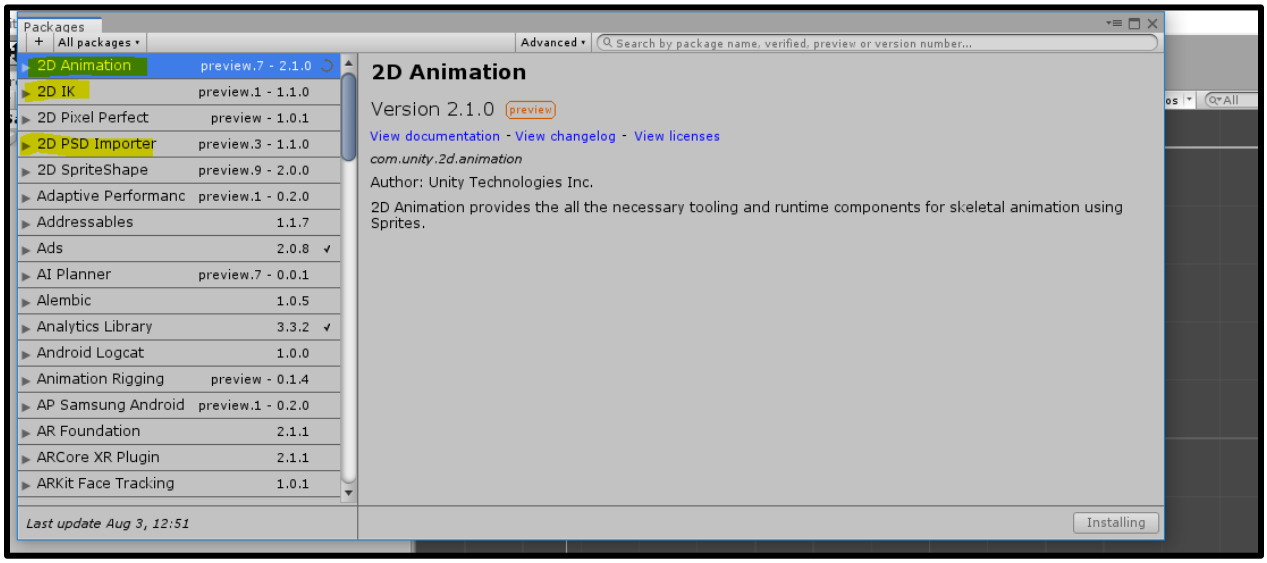
You can get the assets for this project here: 2D Adventurer. This is a character from kenney.nl. I separated the limbs and head in photoshop as separate layers, then I exported it as a .PSB file. You can do the same thing to characters you want to use. For the full source code files, you can download them here.

Importing the packages

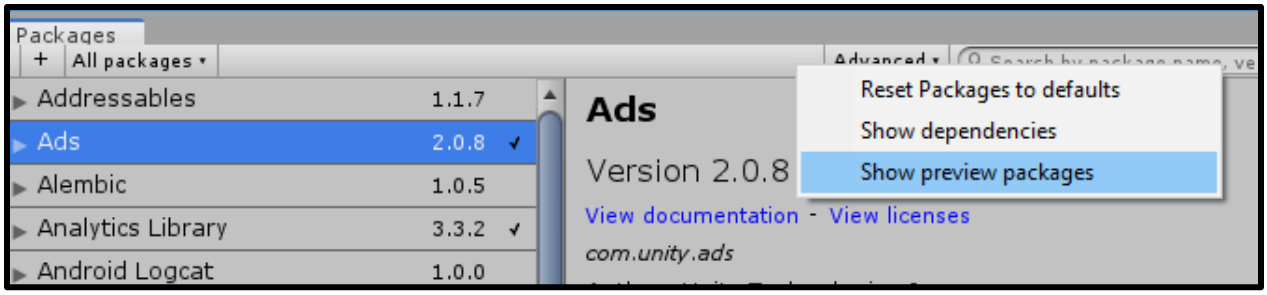
Create a new Unity 2D project and then go to Window -> Package Manager.



Here we need to import the "2D Animation," "2D IK," and "2D PSD Importer" packages.



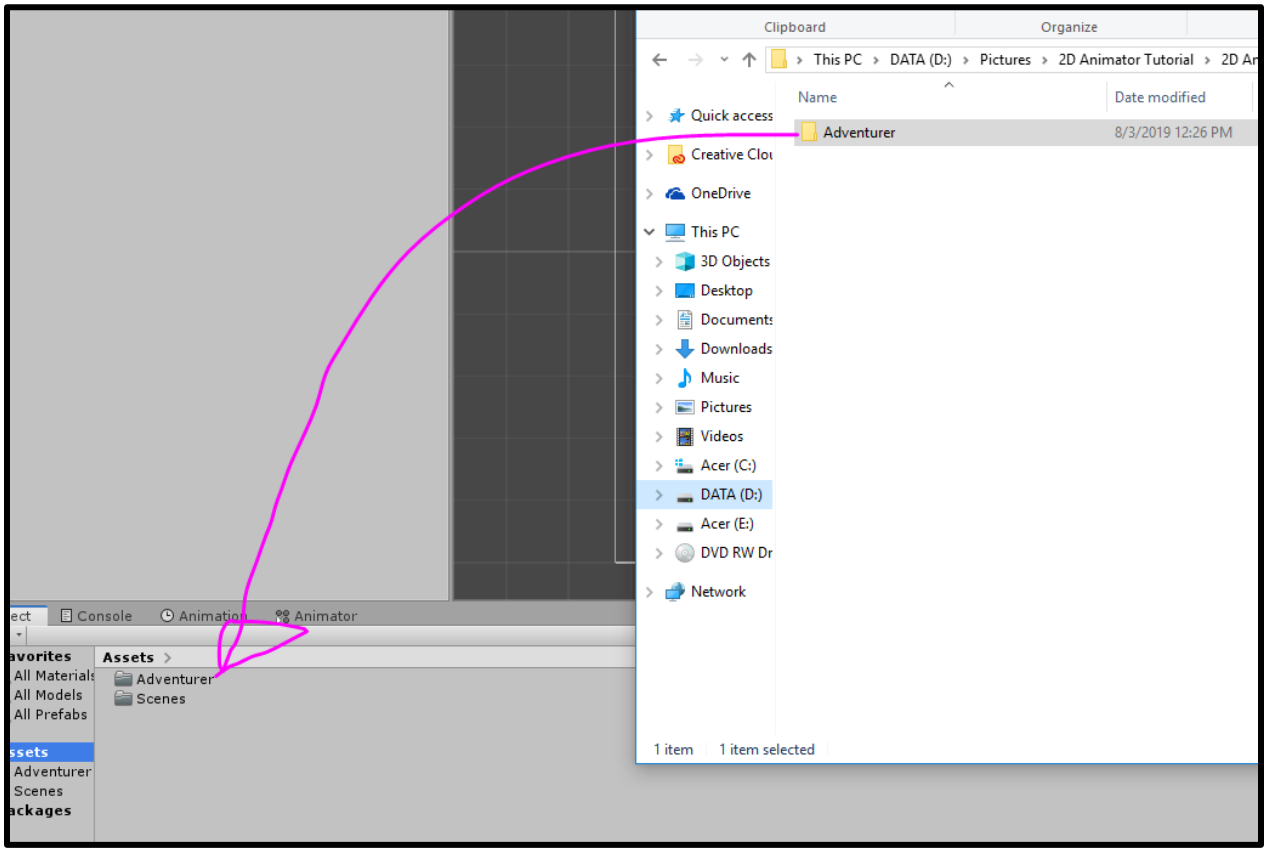
If you cannot see these packages, then you need to click "Advanced" and then "Show Preview Packages."



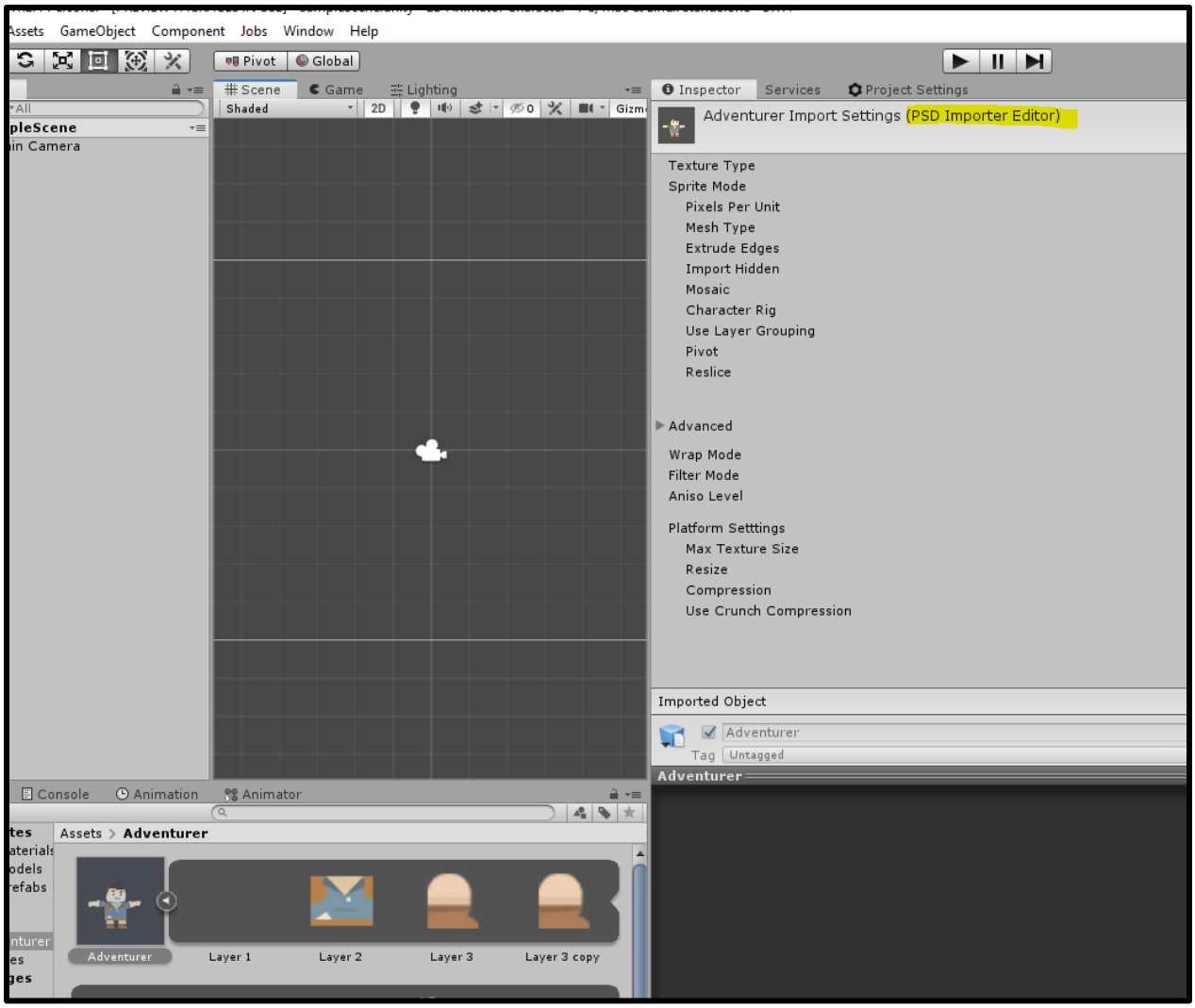
The 2D Animation package is where we will get the sprite rigging tools, 2D IK will help us in animation, and 2D PSD Importer will recognize our .PSB file and each individual character layer. Once these are done downloading, we are now ready to import our character!

Importing the Character

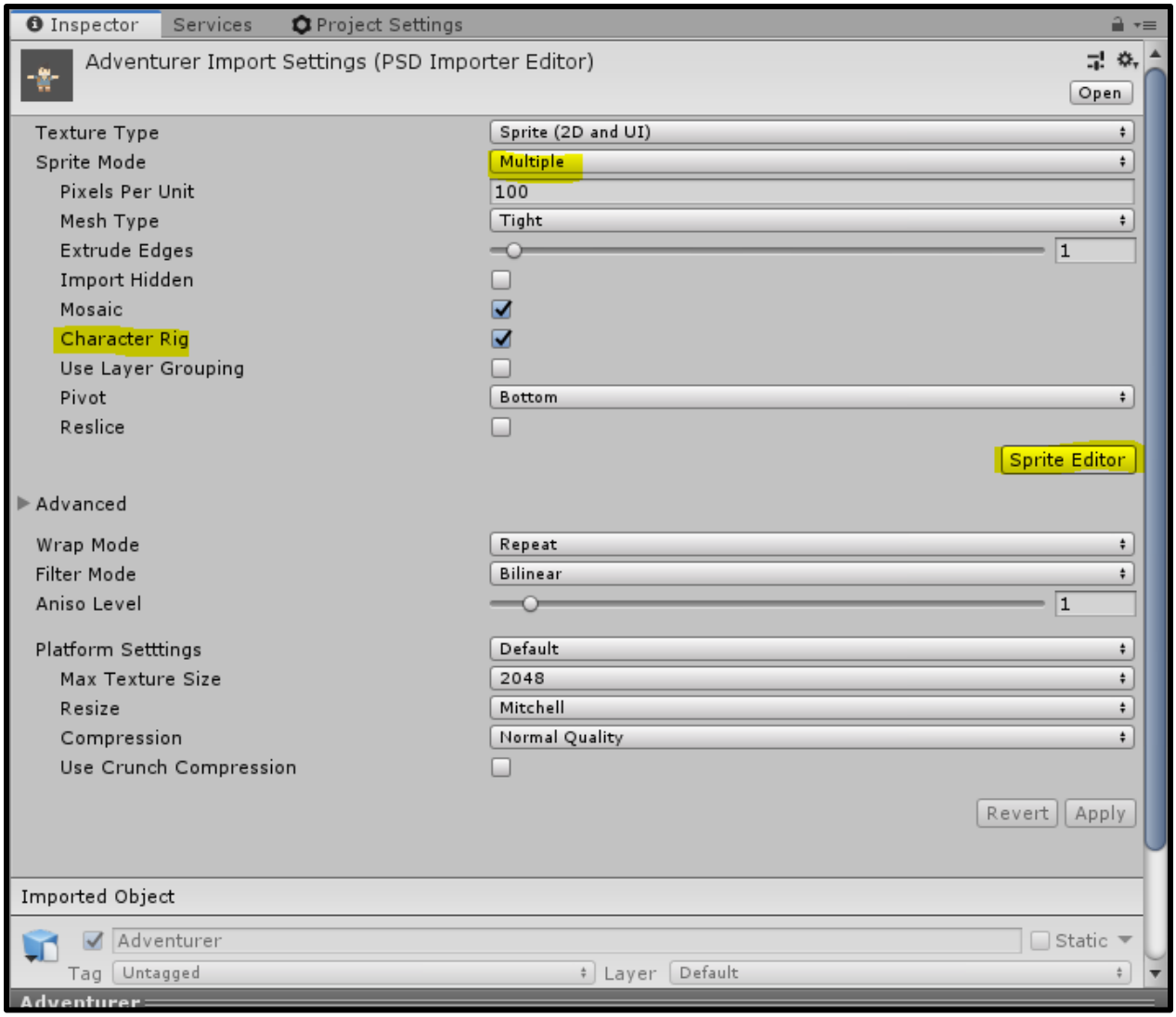
Open up the assets folder and drag the "Adventurer" folder into the project tab.



Once it is done importing, you will notice the .PSB file has been imported with the PSD Importer Package.



Now, we need to configure a few things in the import settings.

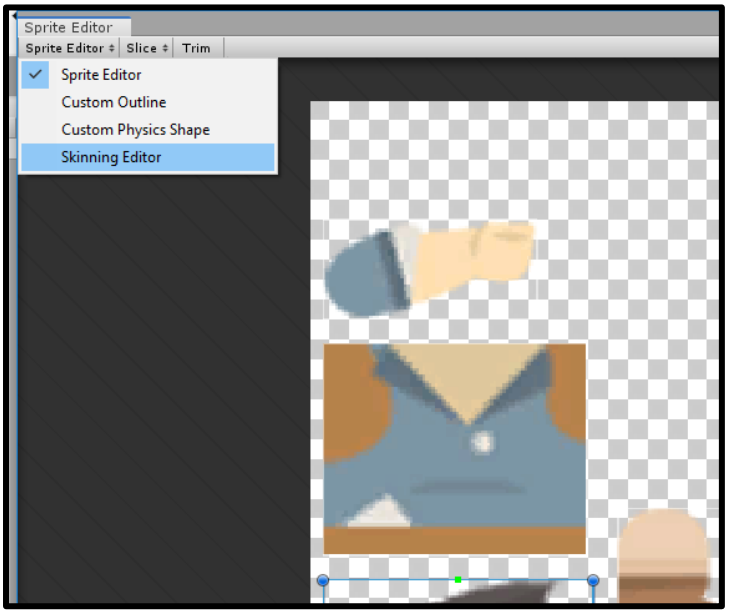


Change the Sprite Mode to "Multiple"; this will tell the Unity Editor that there is more than one sprite in this file and that we would like to use them. It will also recognize each limb layer in the .PSB file. The final thing we need to do here is to check "Character Rig." This simply tells the Unity Editor that we are going to be rigging this sprite. Now, let's hit apply and click "Sprite Editor." This will open up the Sprite Editor and, as you can see, it has taken each layer in the .PSB file and sliced it into its own box.

Here we can tweak the sliced sprites if we see anything that isn't quite right. It all looks good, however, so we can move onto rigging our sprite!

Rigging the Character

Go from the Sprite Editor to the Skinning Editor.



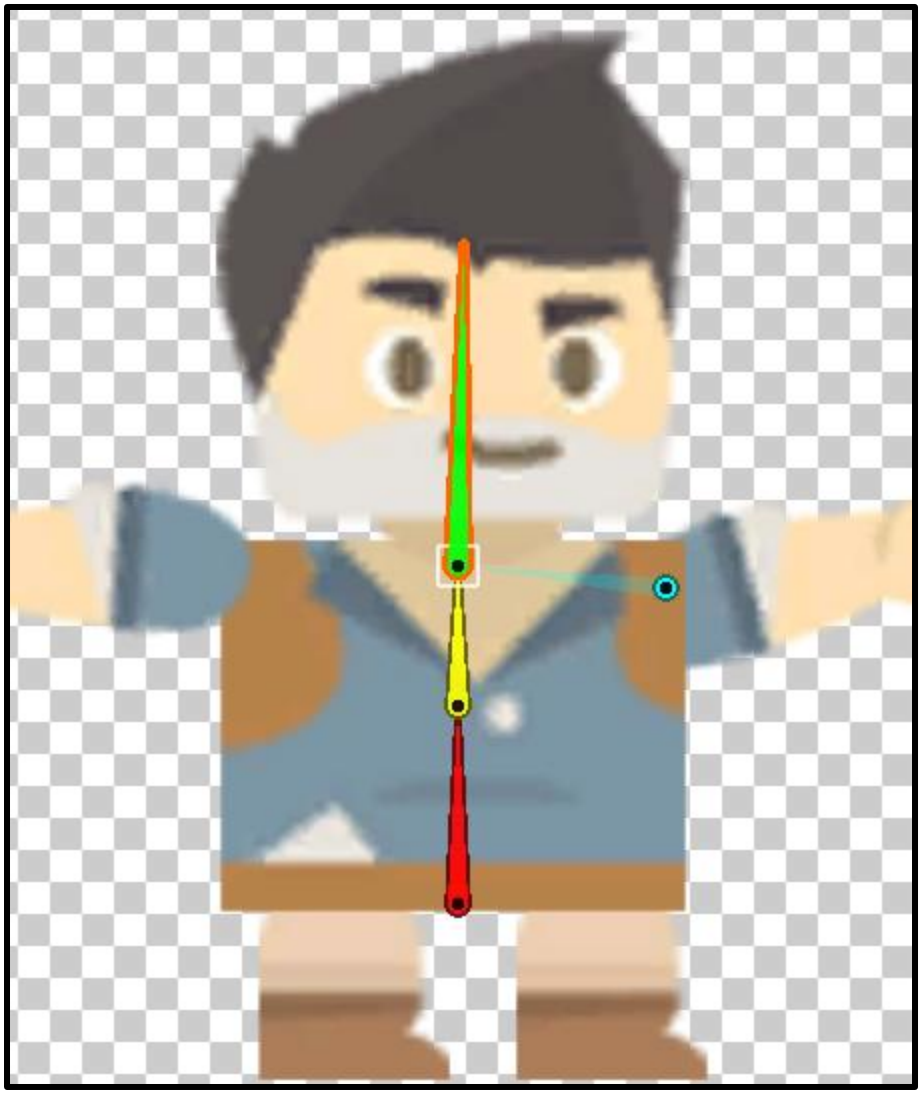
Here, we will create the armature that we will use to pose our character. On the left side of the editor, you will see a list of buttons. "Preview Pose" simply is for when we want to test our rig by posing the character. "Edit Joints" allows us to change the position of the bones without affecting the mesh. This is useful if we find that, for instance, an arm bone isn't quite in line with the arm sprite. "Create Bone" is self-explanatory, let's go ahead and use this right now.

You will see that you can click once to place a bone point, and then again to place the final bone point. Let's go ahead and add one bone near the lower portion of the torso…

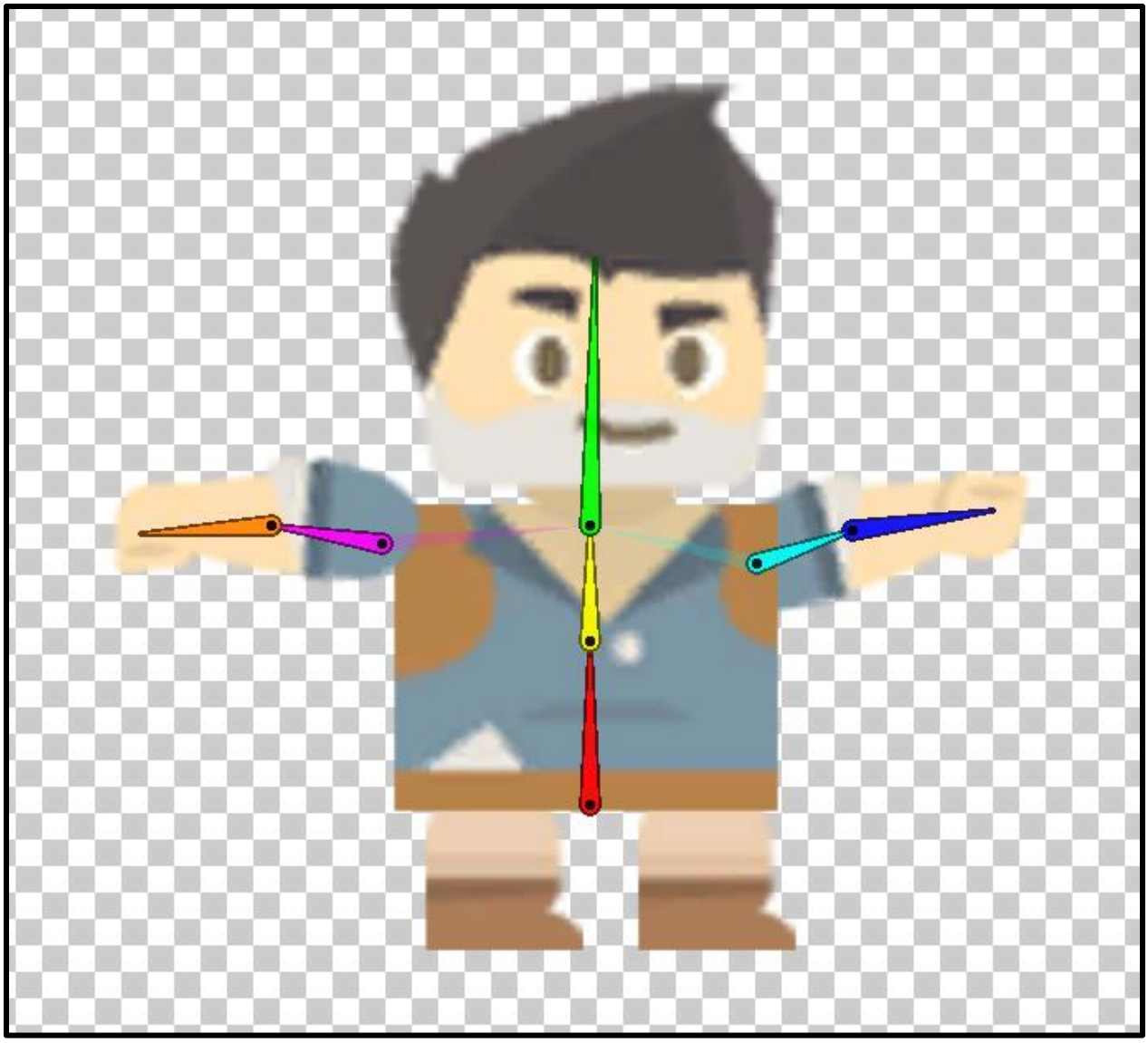
… then another halfway up right before the head…

… and finally, one that will serve as the head bone.

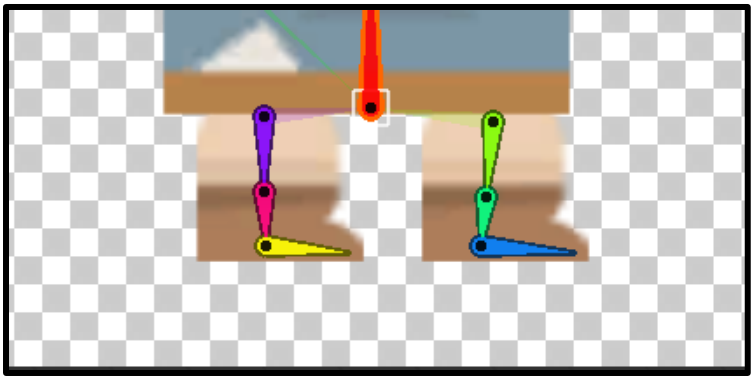
Now, we need to create the arm bones. We could have made all the bones connected, however, because of the way the layers are separated, it's best to give each layer it's own independent armature. Right-click, and you will see that we are free from the created bone chain. Before we create the bone armature, however, we need to parent our new bones to the torso bones. Hover over the origin of the second torso bone, and left-click.



This will create a semi-transparent line from the torso bone to our bone point. Now, create two bones for one arm then repeat the process for the other.



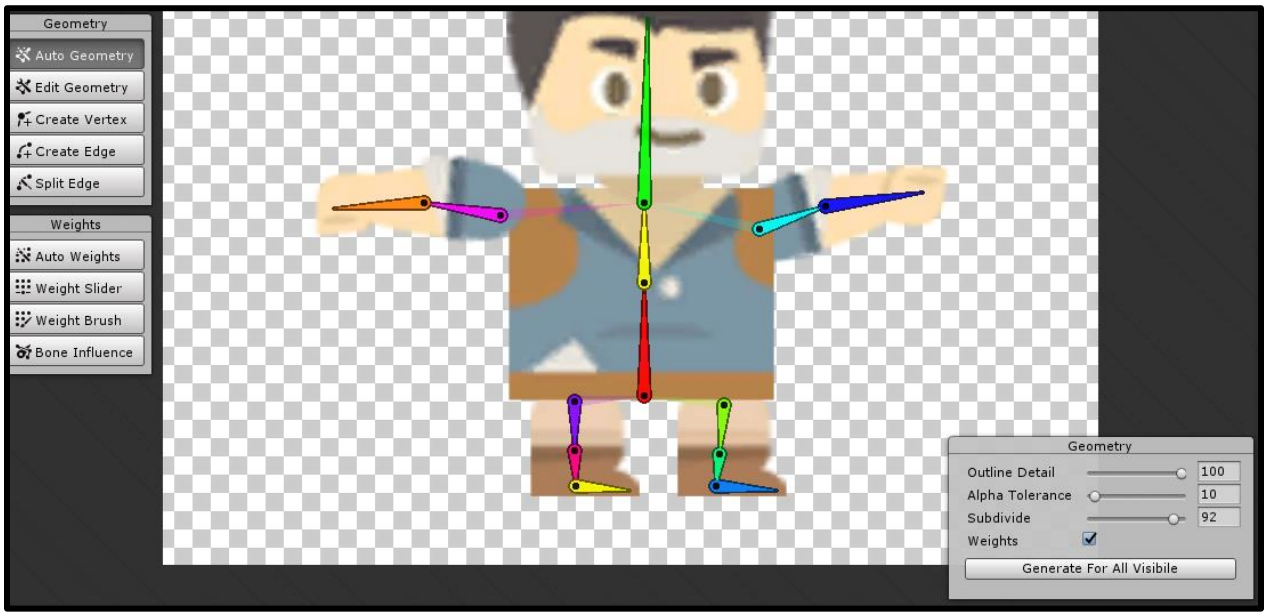
Next, we need to create the leg bones. These need to be parented to the origin of the lower torso bone and this will have a bone for the toe.



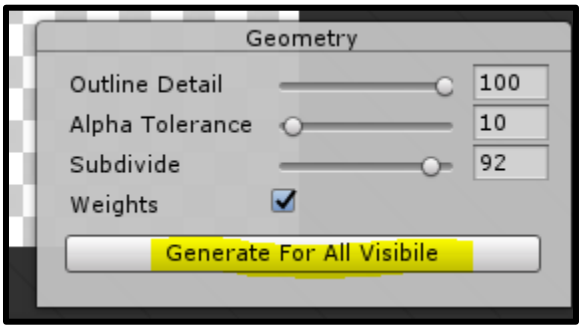
Repeat the process done on the arm bones.

Now that the armature is complete, let's have a further look at the tools in the toolbar.

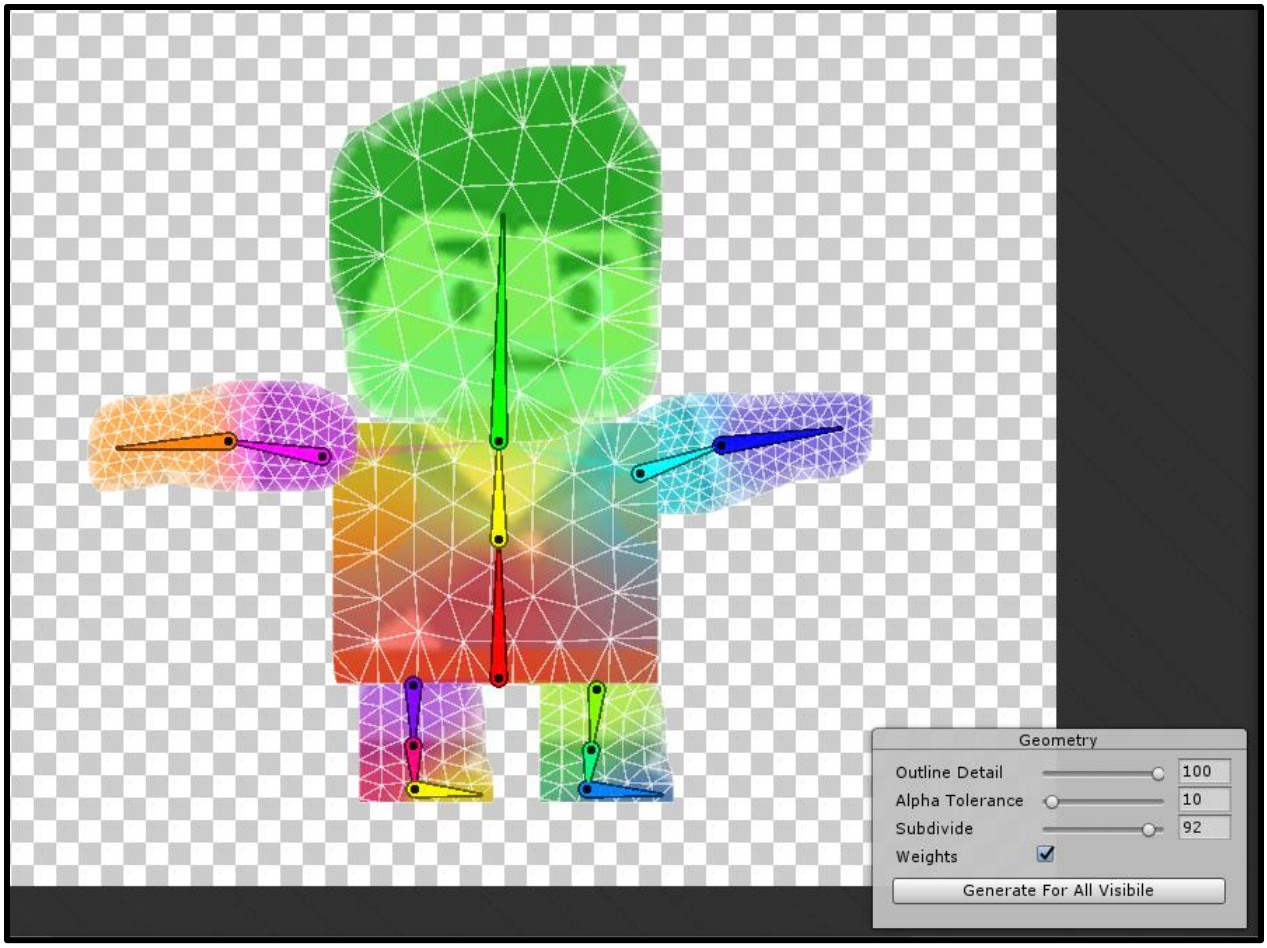
"Split Bone" will split (or subdivide) a single bone into two bones. "Reparent Bone" allows you to see exactly how bones are parented to each other reparent if necessary. "Auto Geometry" will automatically create the geometry that will be used by the armature. If you select this, you will see that it brings up a window in the lower right corner. "Outline Detail" determines how accurately the mesh will stick to the edge of the sprite. I found that a value of about 100 works best for this project. "Alpha Tolerance" means that a pixel will be counted transparent if it's alpha value is less than this slider. We don't have any alpha on our character so leave this set to default. "Subdivide" is how many vertices will be generated within the mesh. I found that we need a fairly high value of 92 in order to get the best results. And finally, "Weights" will automatically assign parts of the mesh to a bone if we have created one already.



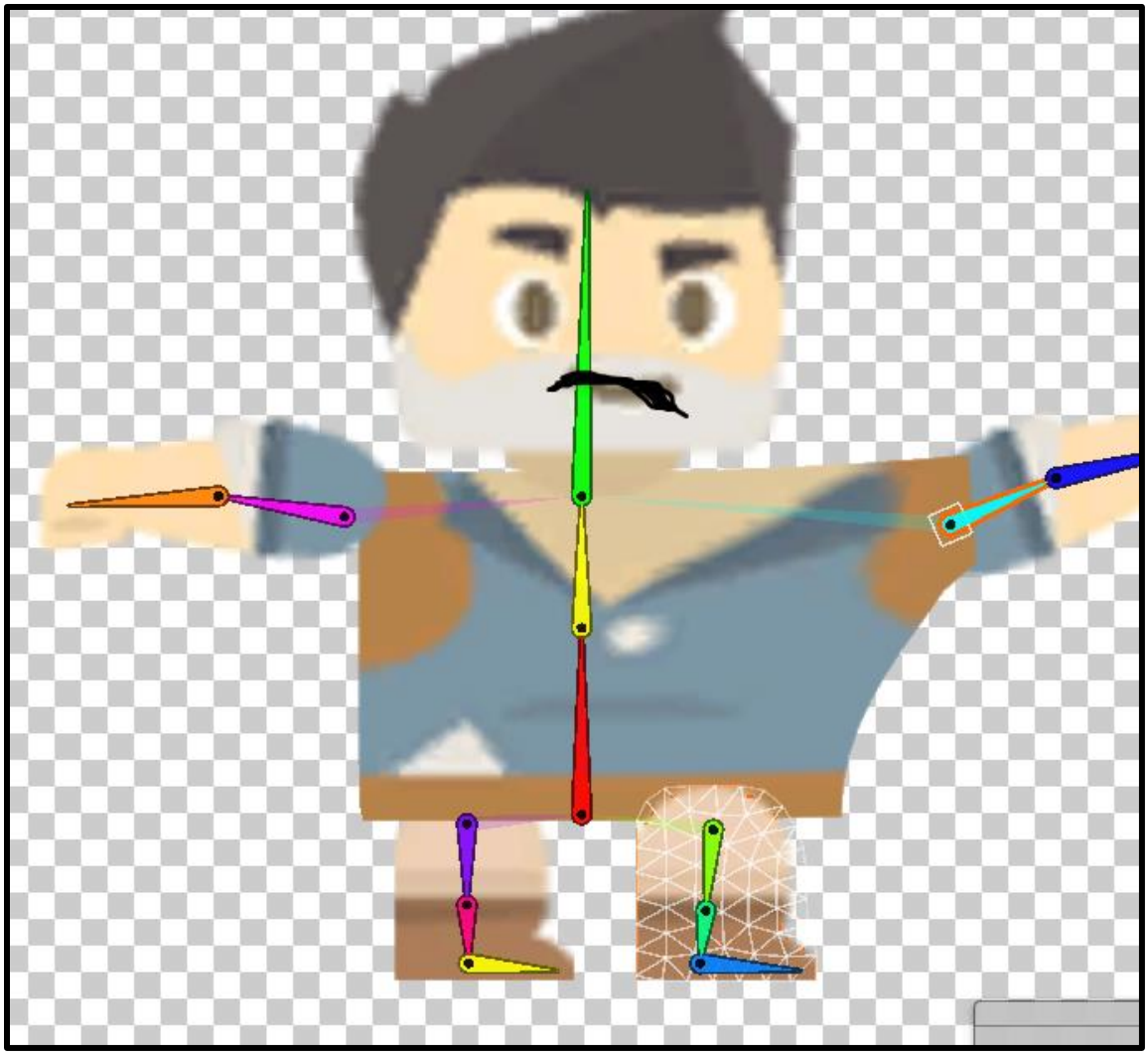
Go ahead and leave this checked and hit "Generate For All Visible."



You will see that Unity generates a mesh with lots of colors.

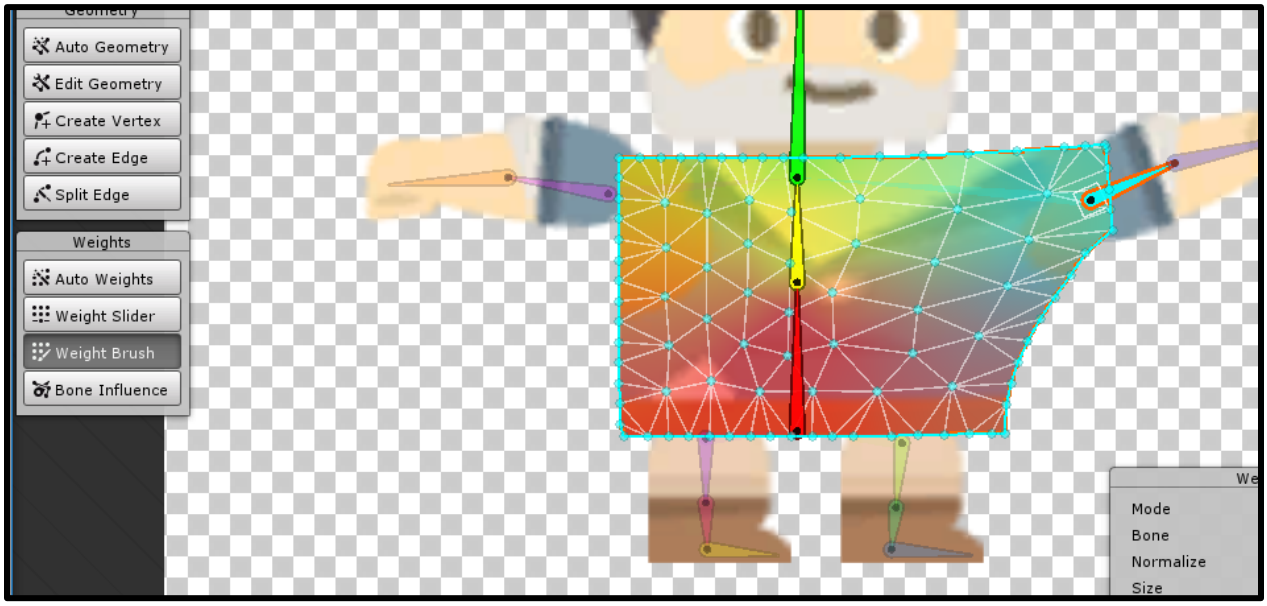


The colors on the mesh signify which bones are going to affect that portion of the mesh. If you select a bone and try moving it, you'll see a couple of things that should be fixed. An obvious one is in the arm and leg bones. If you test the arm and leg bones, the left arm (relative to the character) is moving a portion of the torso.

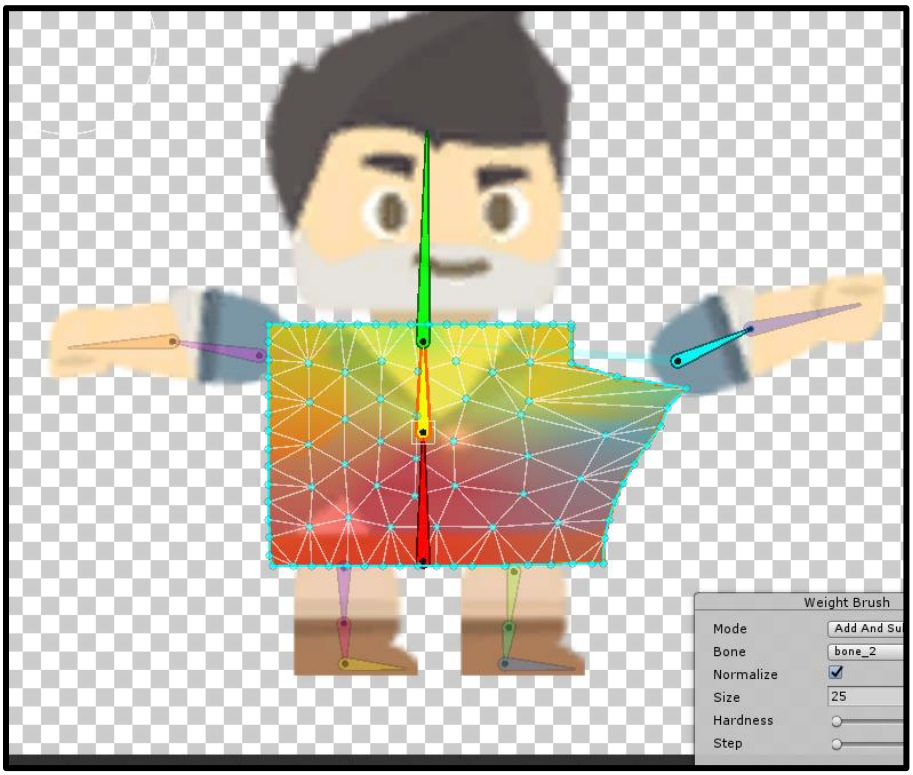


This doesn't look right so let's fix this. Under the "Weights" toolset, the "Auto Weights" does essentially what "Auto Geometry" did for us, "Weight Slider" allows you to use a slider to define which part of mesh a bone will influence, and "Weight Brush" uses a brush tool to edit how the bone affects the mesh. We are going to use Weight Brush to correct the arm and leg problem.

Select this tool and double click on the torso.

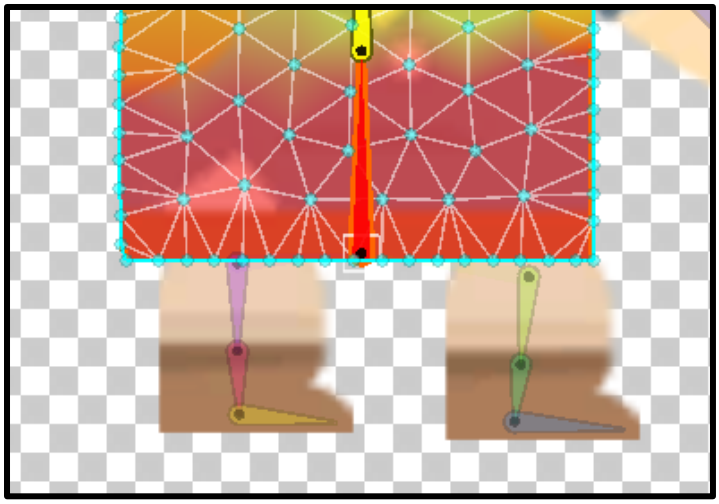


As you can see, the color of the left corner of the torso is the same color as the shoulder bone on the left arm. This means that the arm bone is influencing part of the mesh. To get rid of this, click on the top torso bone and paint out the arm portion on the torso.

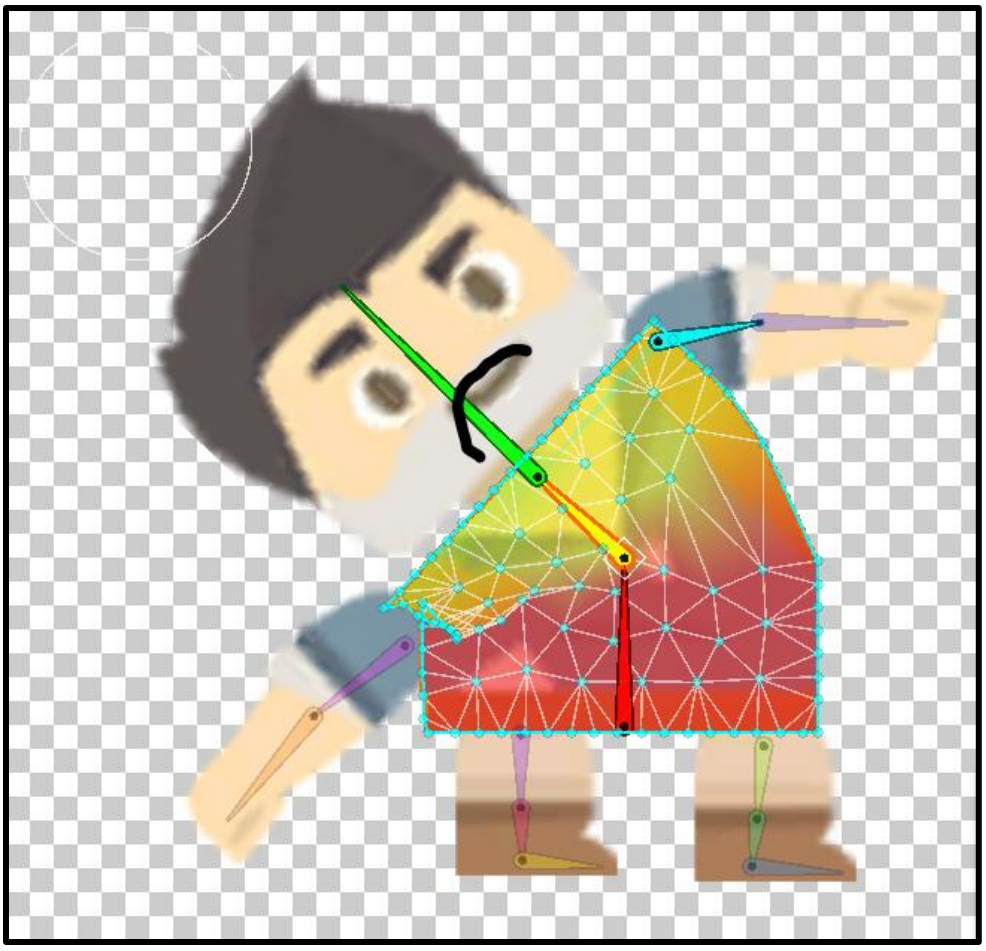


If you increase the hardness, the brush will paint one color faster and stronger. You can select the arm bone and rotate it to see if we are close to removing its influence.

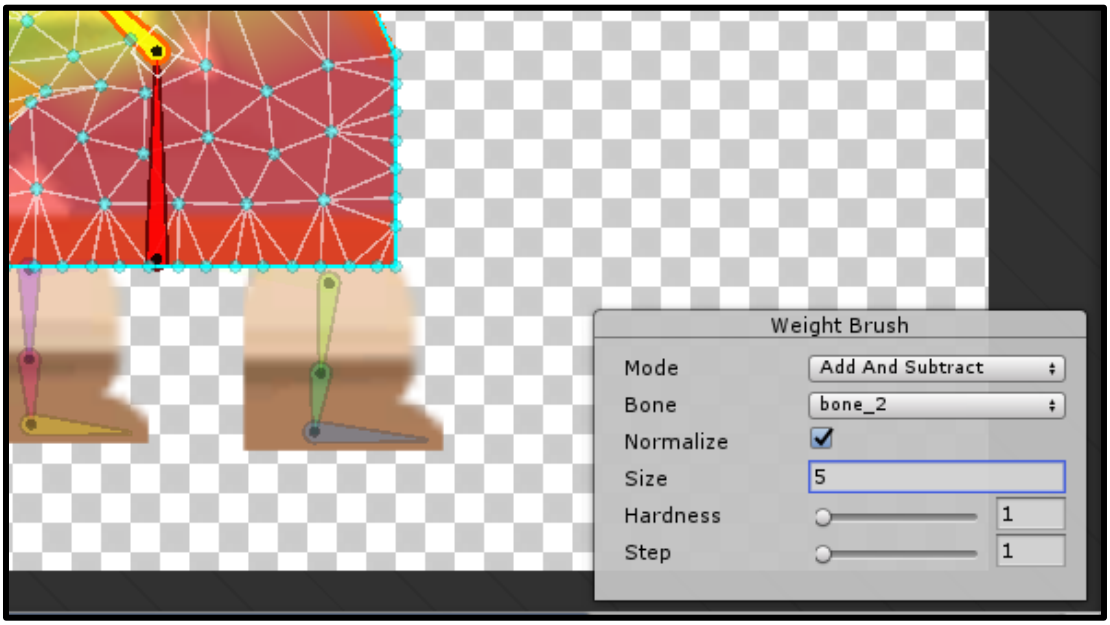
Select the lower torso bone and paint out the leg bone's influence on the torso.

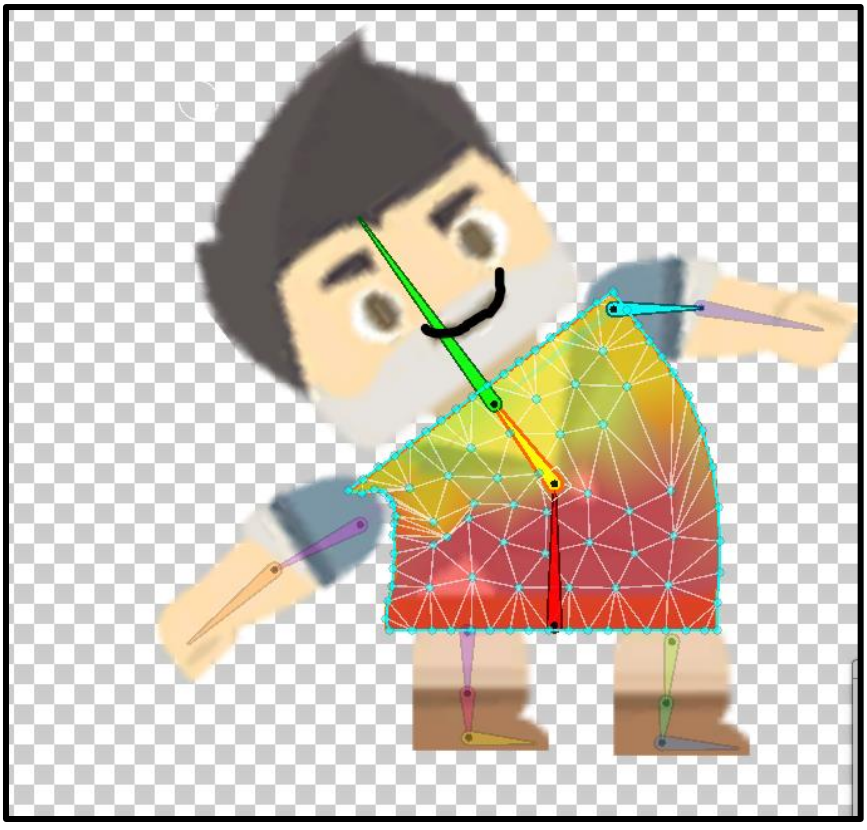


Now that we have corrected our bone influence problem, there is one thing we need to do to the torso in order for this to look right. You'll notice that portions of the torso bend properly while others look too rigid and strange.

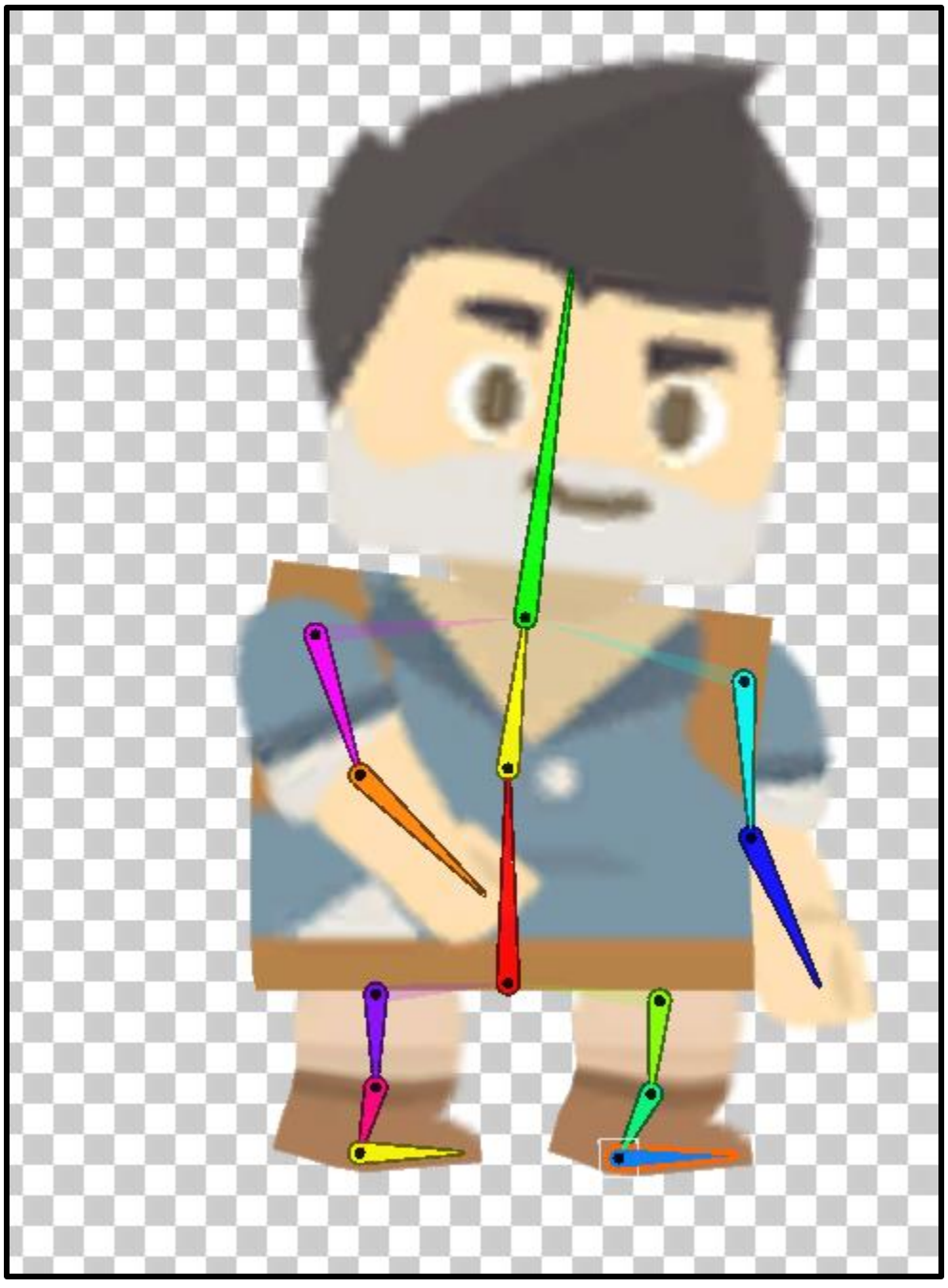


A really easy way to fix this is to rotate the top bone, bring the hardness down to 1, set the size to 5, and then paint on the problematic side. It may take a couple of tweaks to get the right influence painted, but this is how we correct any strange looking armature problems.

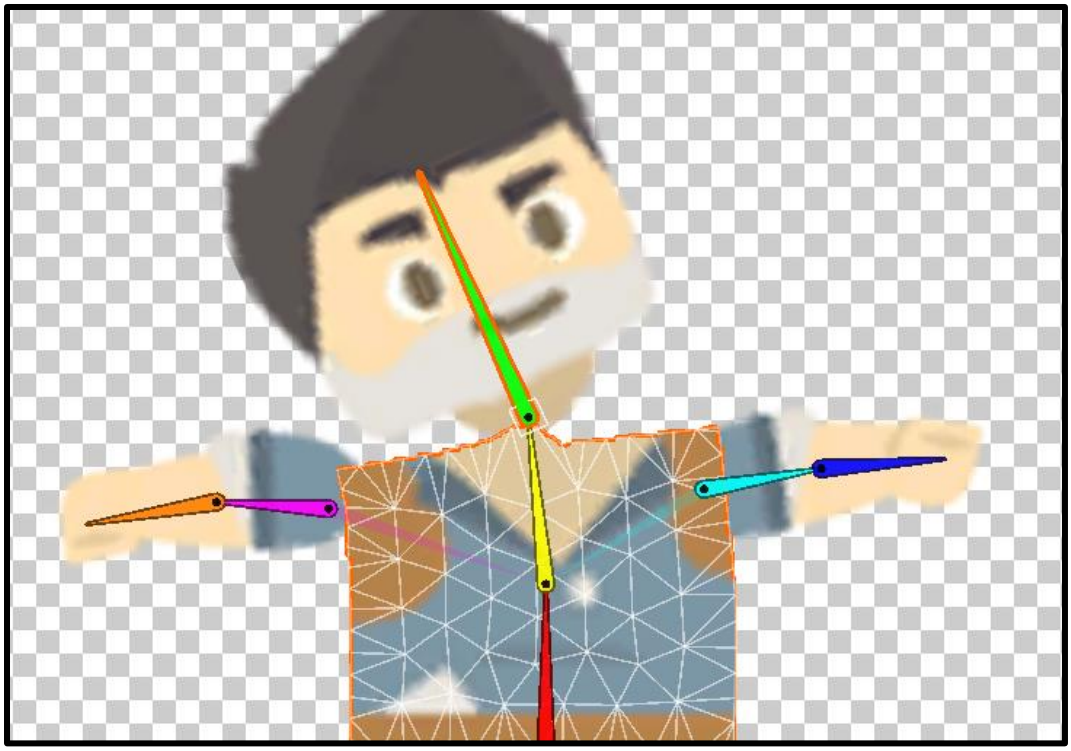




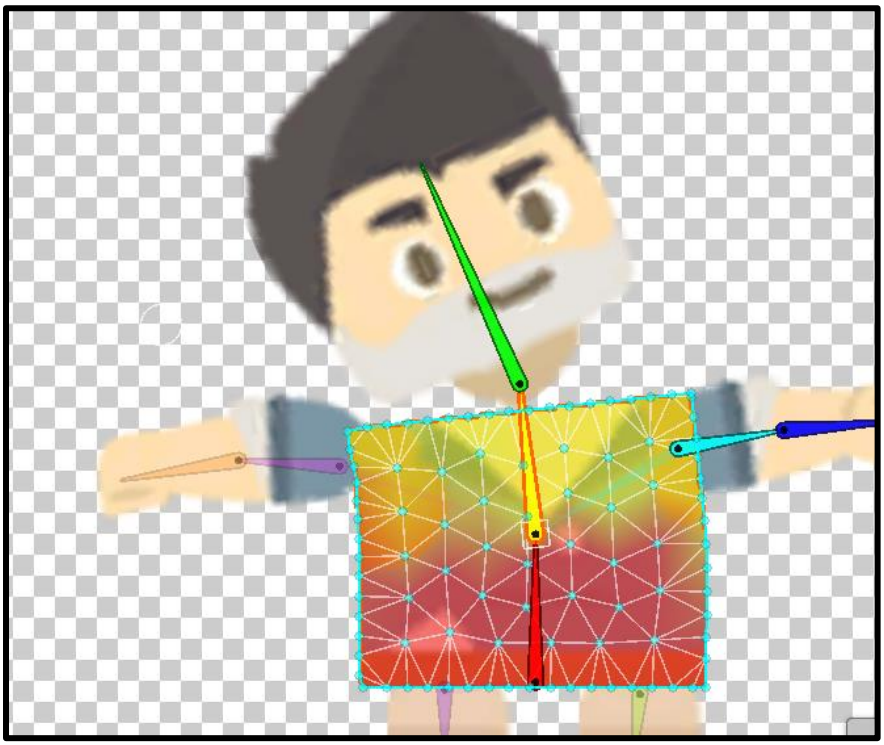
Alright! Now, let's go into Preview pose and give our character a final look over.



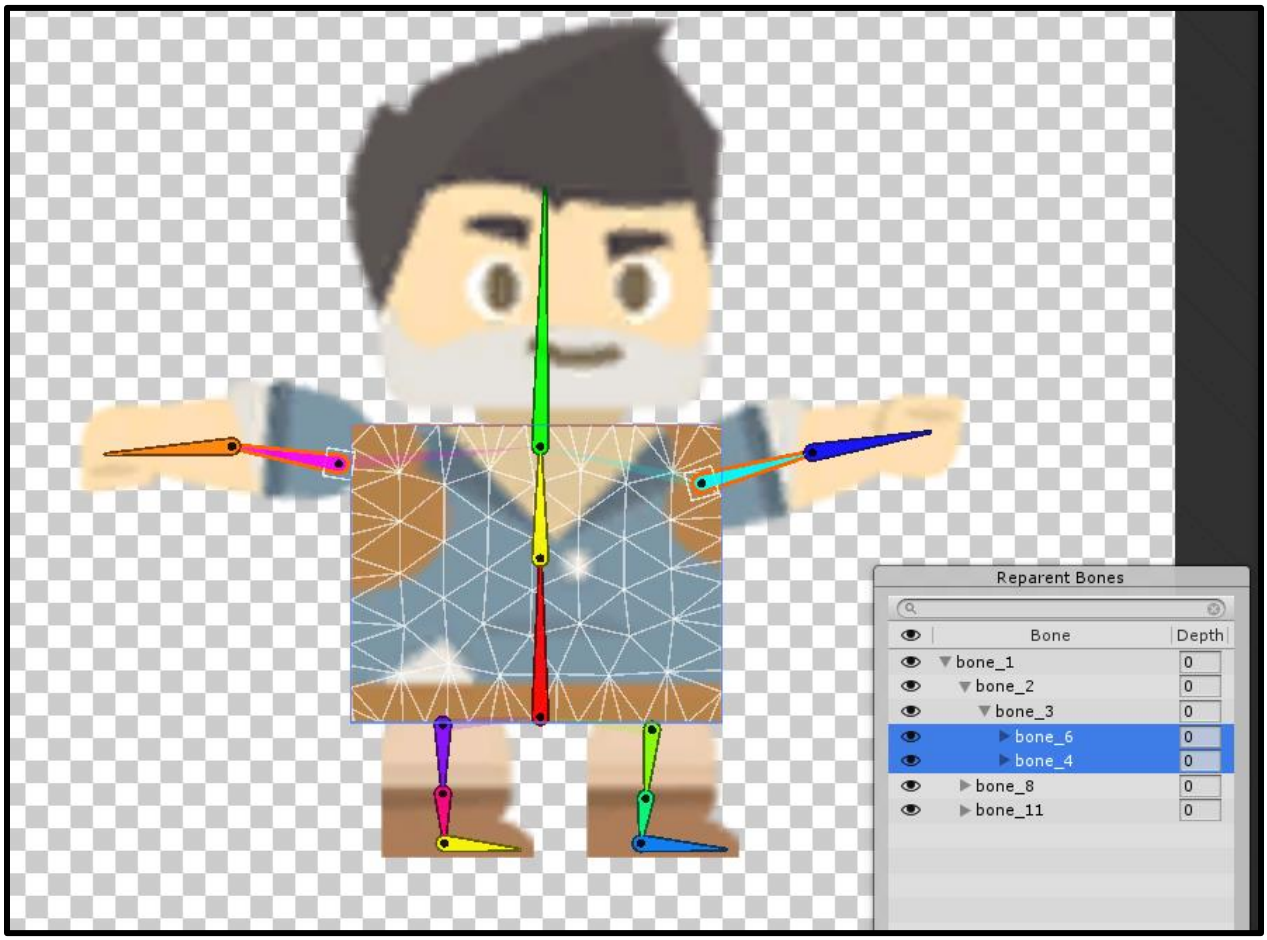
Move every single bone to see if there is anything that looks strange.



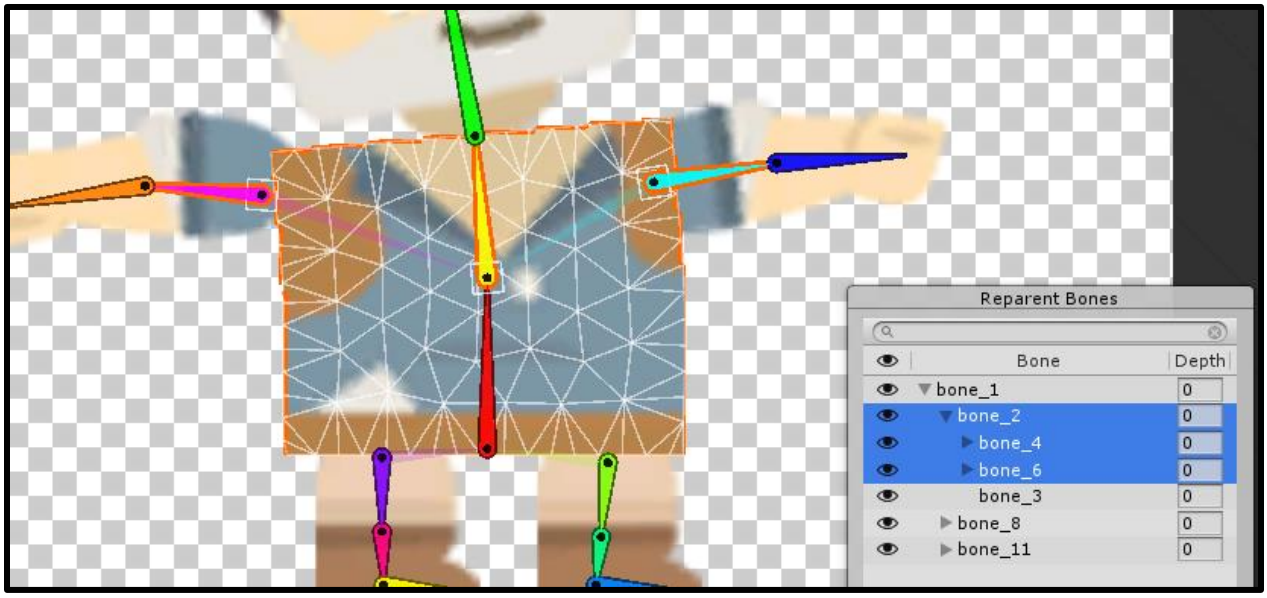
I noticed that the head was influencing portions of the torso, this looked wrong so I corrected it by using the Weight Brush to assign that portion to the top torso bone.



I also had to use the Reparent Bone tool to fix my arms that were parented to the head bone.

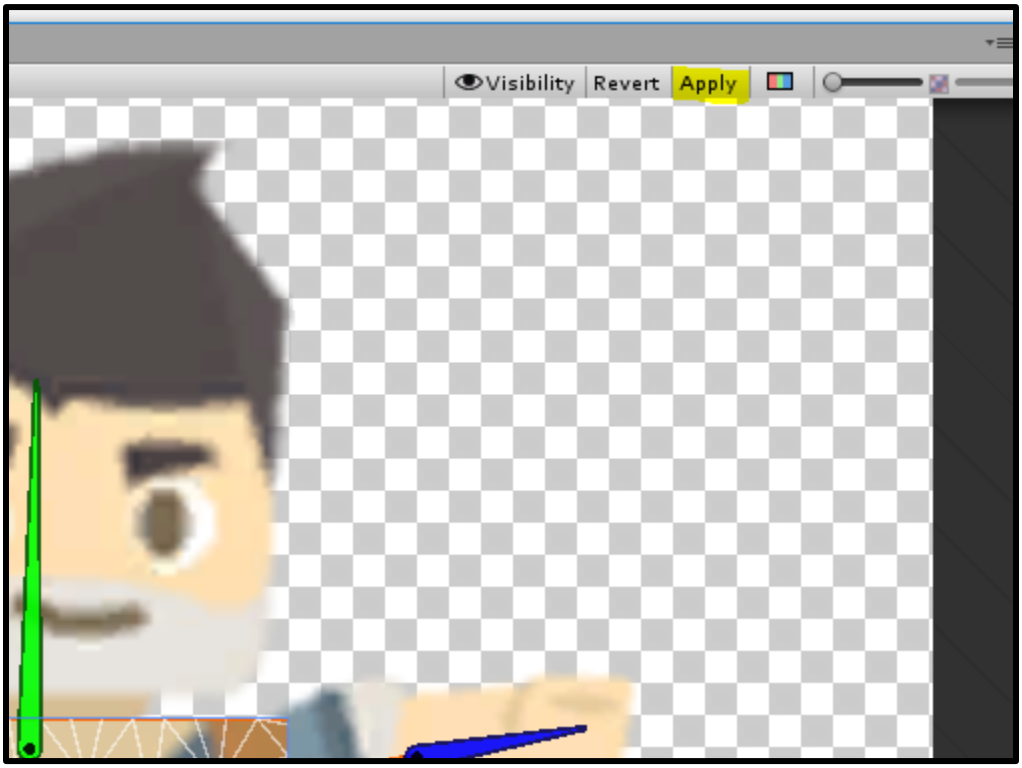


Before



After

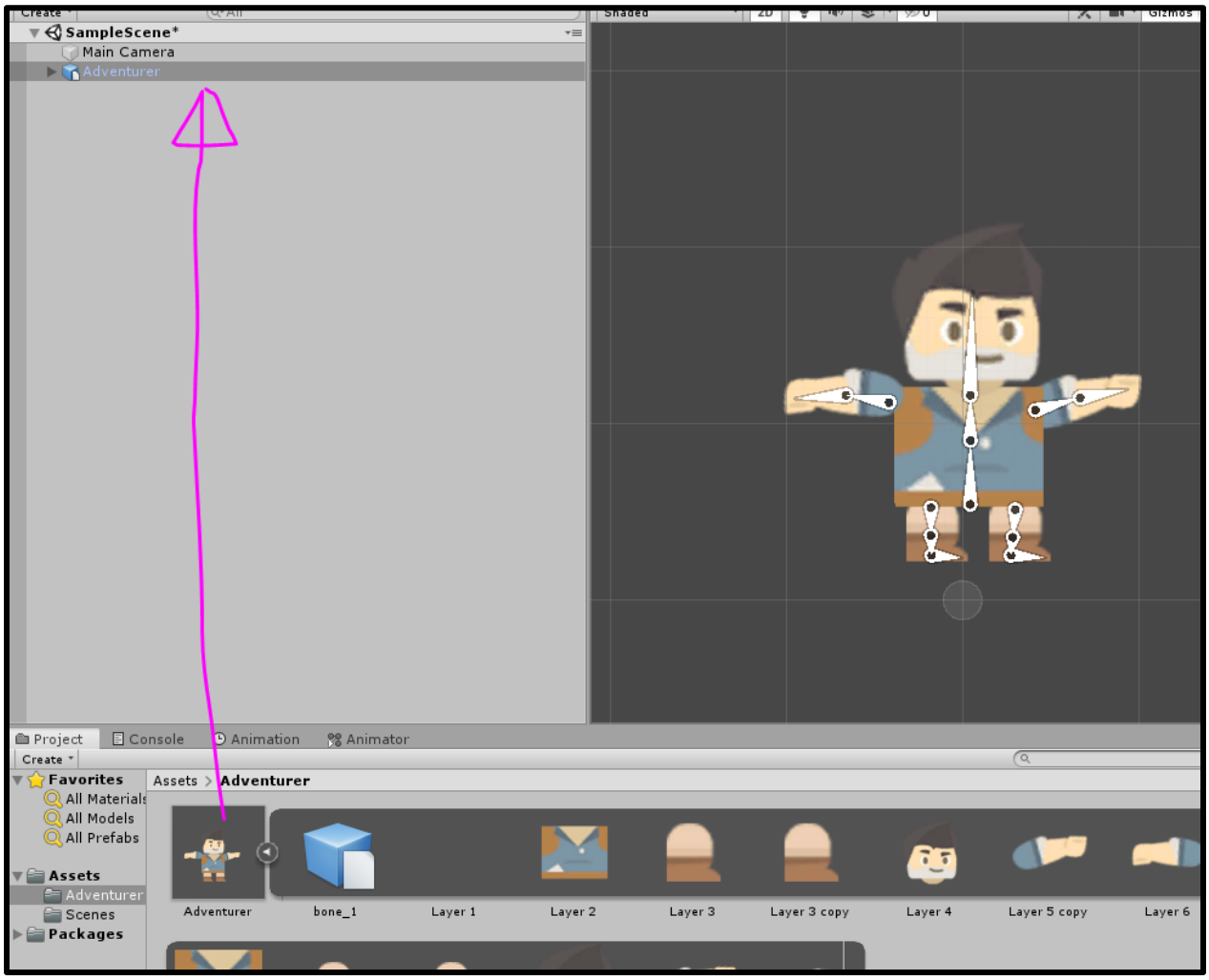
Okay, everything looks good so hit apply.



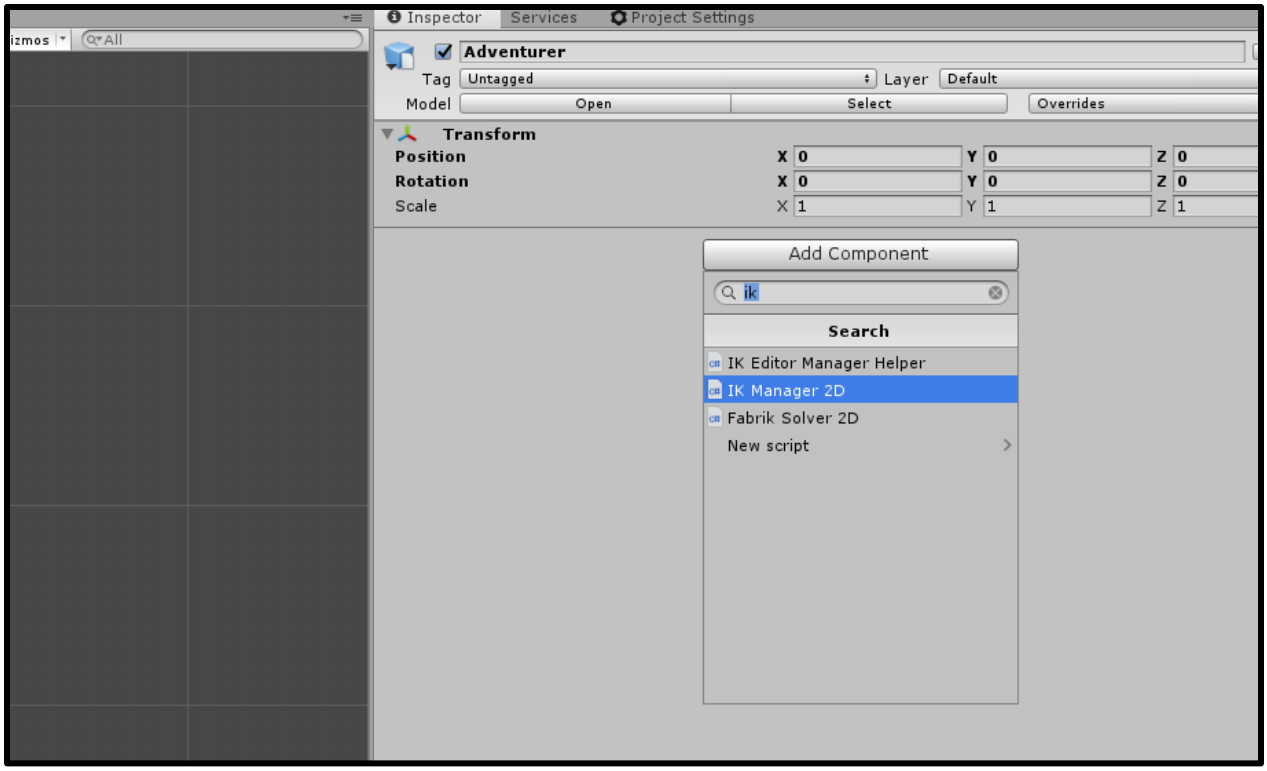
Setting up IKs

Arm IKs

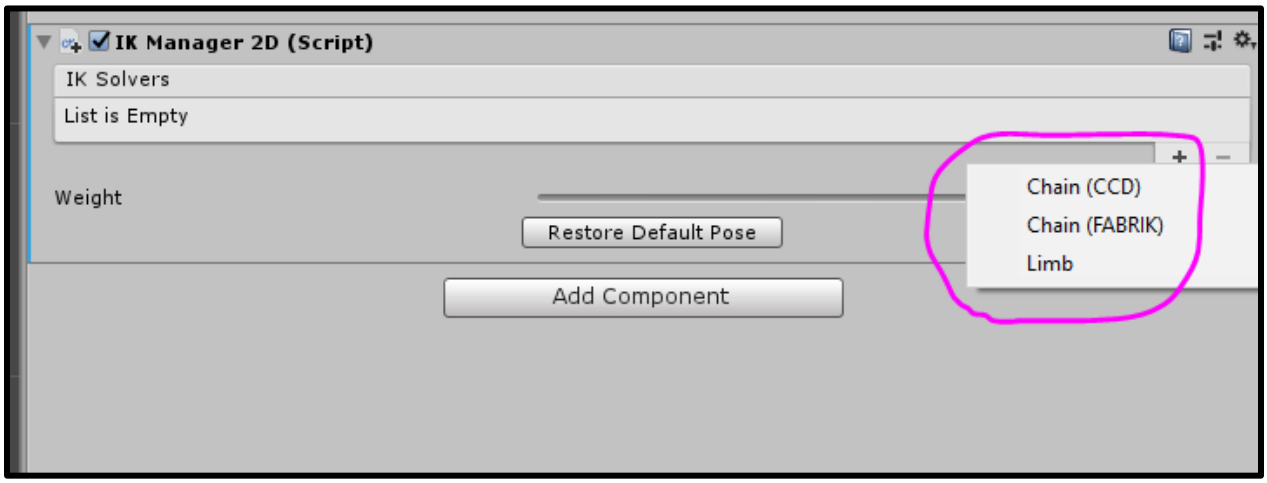
Drag the Adventurer character into the scene.



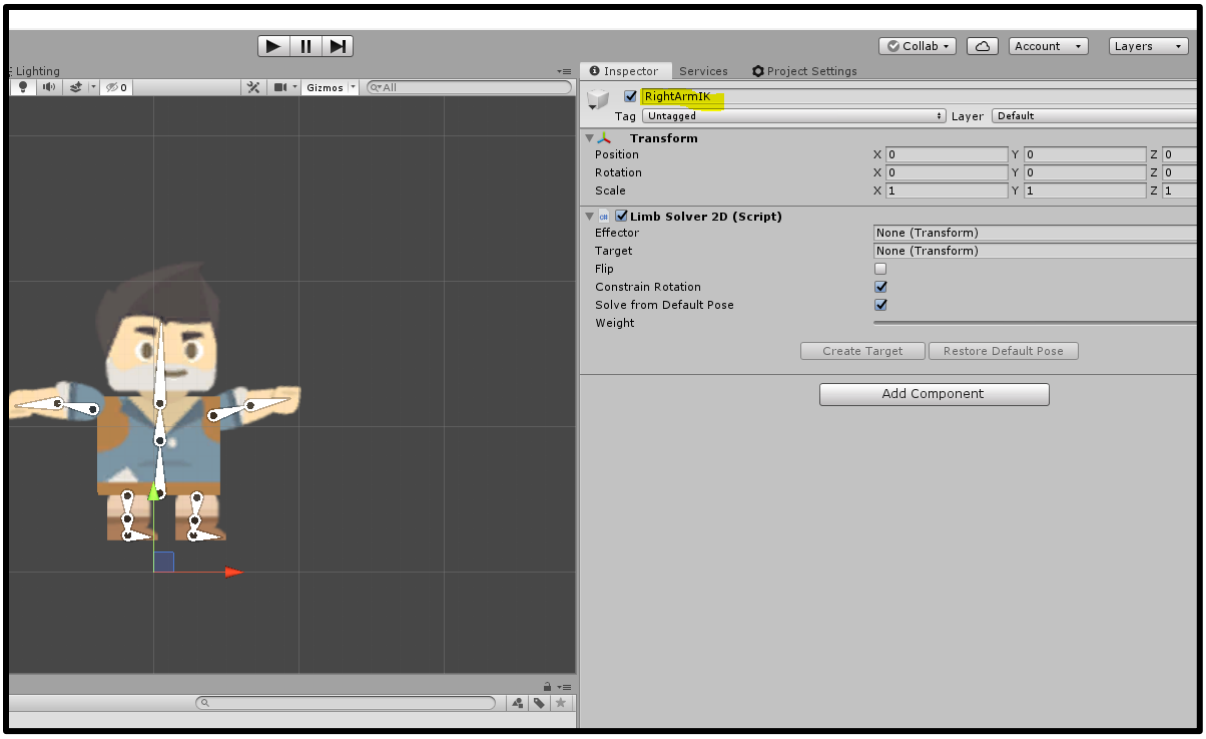
You'll see that the bones show up and we can rotate and manipulate them. We are now animation ready but we are not animation optimized. Our rig needs "Inverse Kinematics." Inverse Kinematics is when the computer predicts the position and rotation of a series of bones based on the position and rotation of a single "target" bone. This is useful for creating a realistic arm and leg rig. Click on the Adventurer character and add an "IK Manager 2D" component.



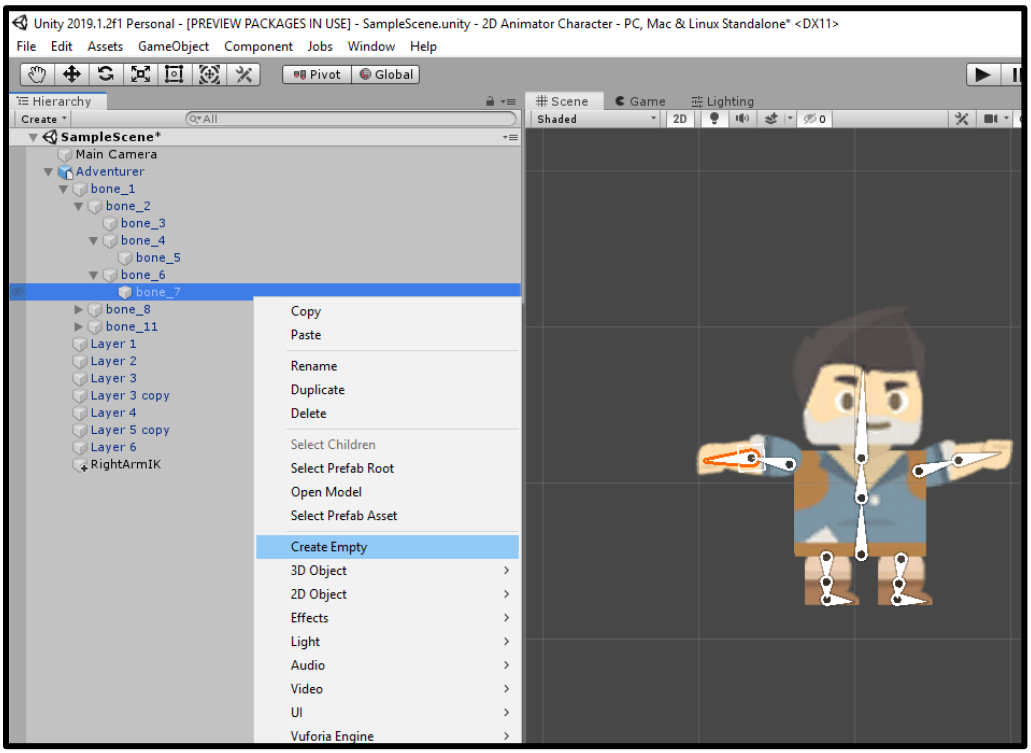
Hit plus icon and you'll see that it gives us three options.

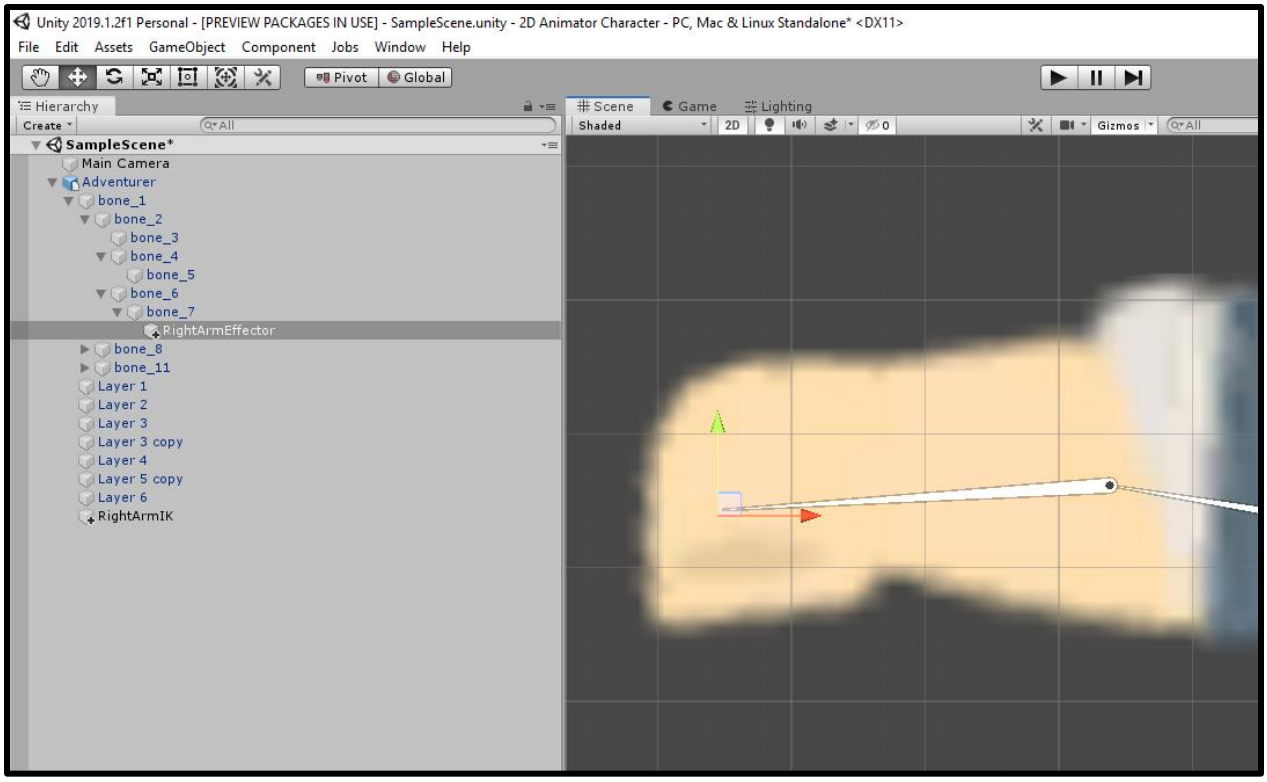


"Limb" is for any bone that has two bones in its chain. The other two, "Chain (CCD)" and "Chain (Fabrik)," are for any bone chain with more than just two. For the arm, we will be using simply "Limb" so go ahead and create it. When you do that, you'll see that it created a new object within the Adventurer game object. This will be our right arm IK so rename it "RightArmIK."

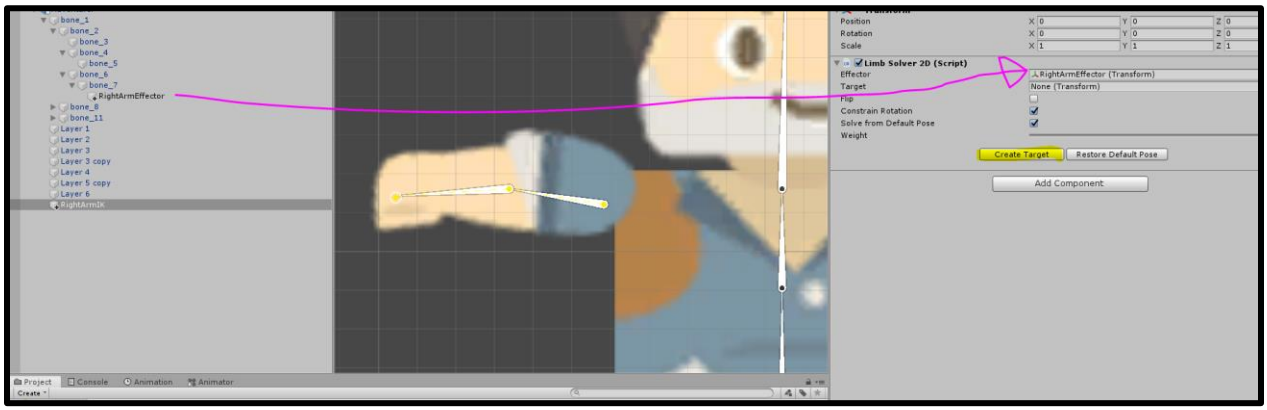


Now, as you can see, we need an effector and a target in the Limb Solver 2D. Click on the right arm forearm bone and create a new empty game object called "RightArmEffector."

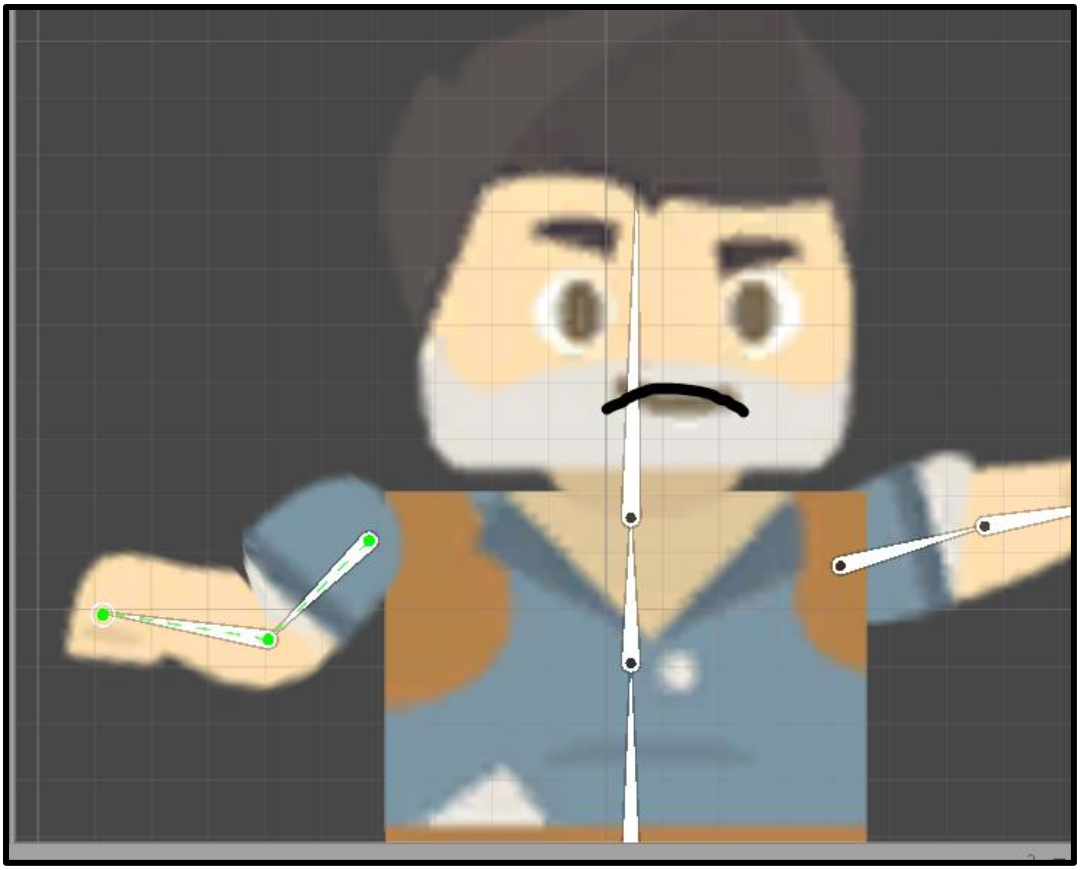




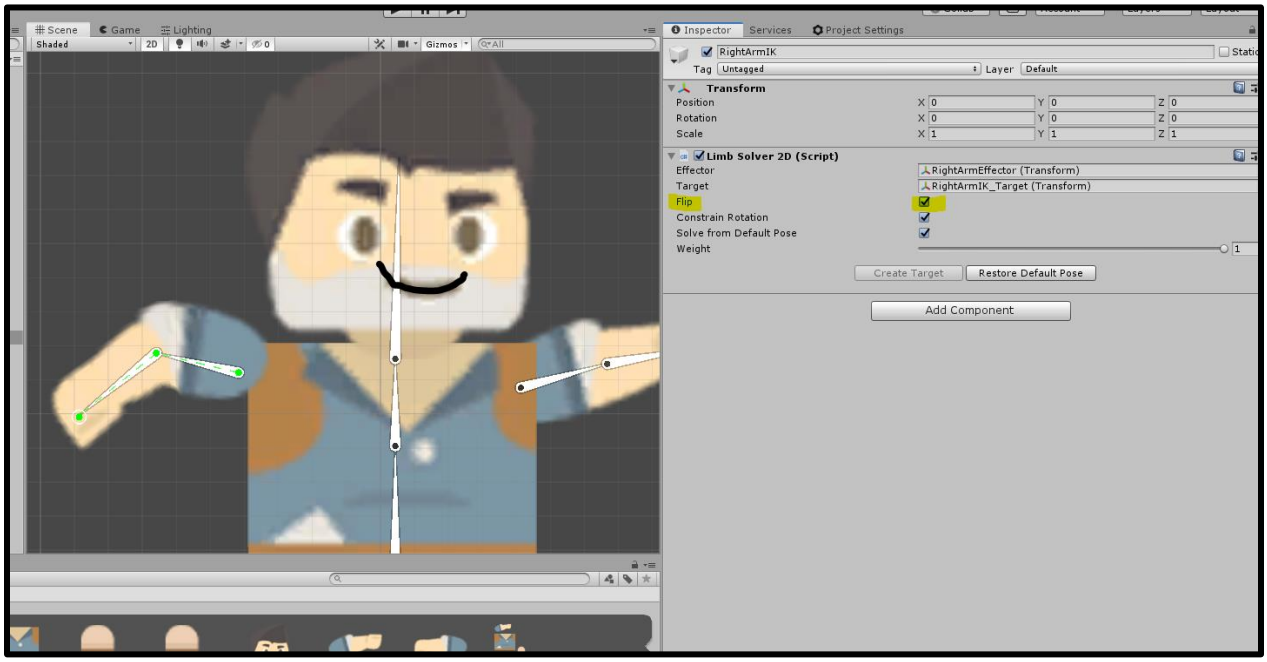
Drag this to about the position of the hand and then drag this into the "Effector" in the Limb Solver 2D on the RightArmIK.



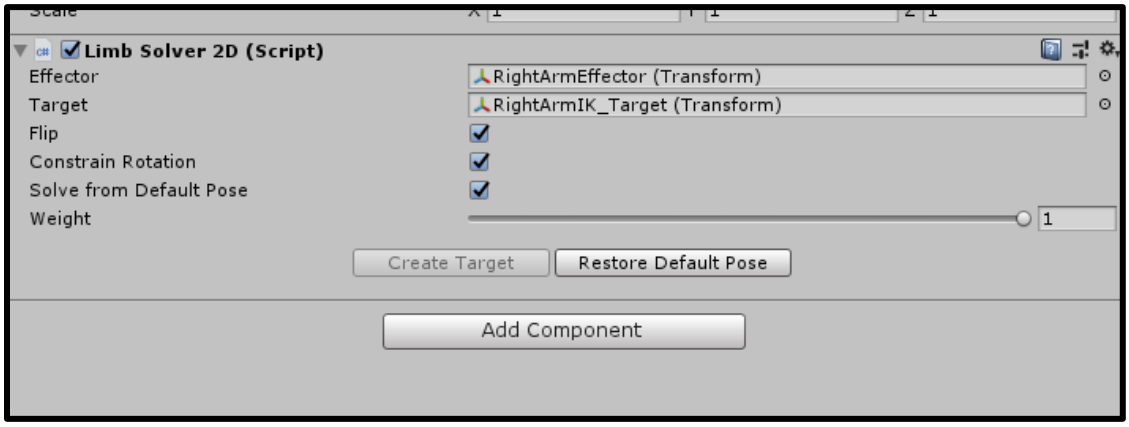
Now, hit "Create Target." You'll notice that when you drag the RightArmIK around (that's the little circle at the end of the arm bones), the elbow is in the wrong position.



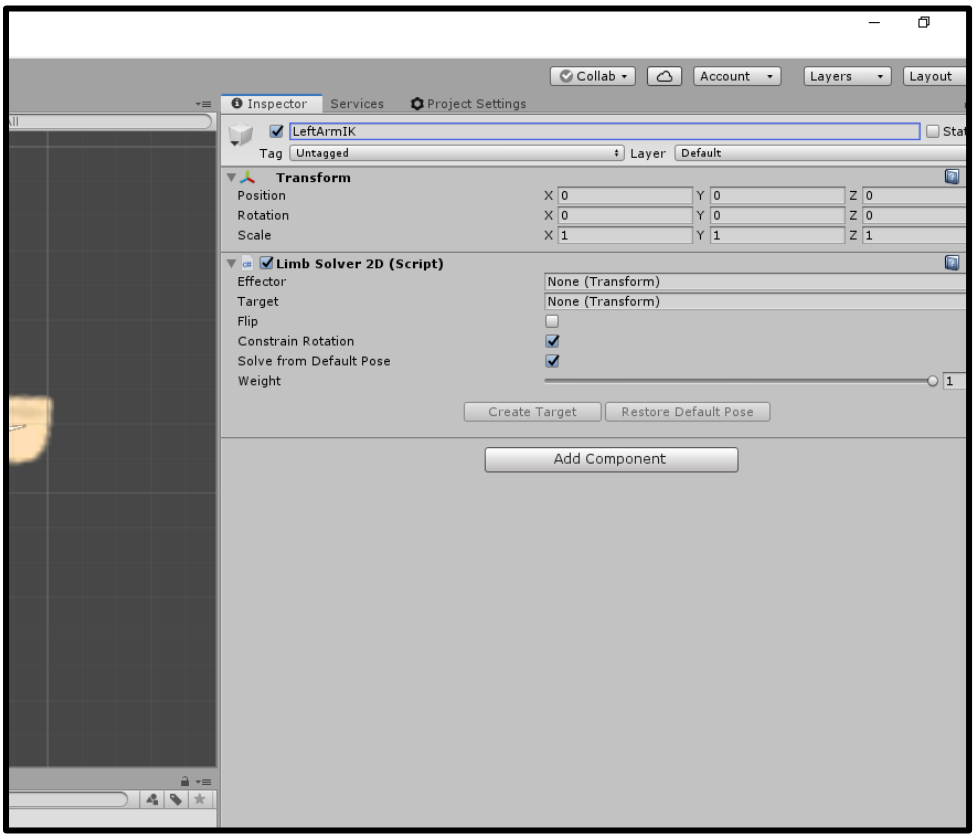
We can fix this by simply hitting "flip" in the Limb Solver 2D.



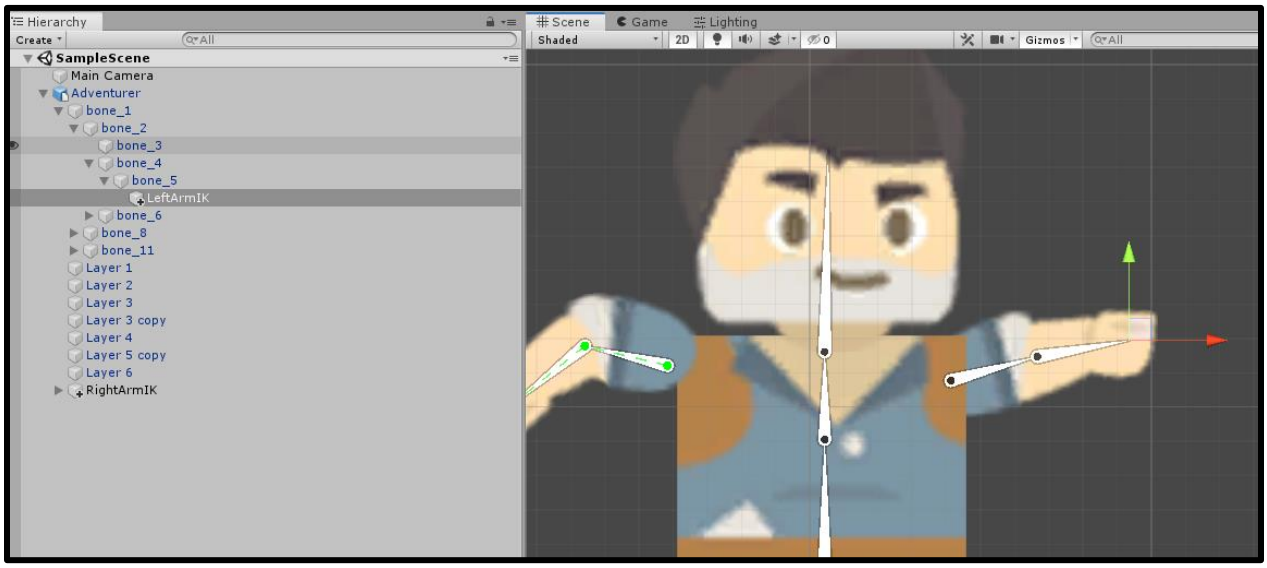
As you can see, this fixed our problem! The other settings here do not really affect our project.



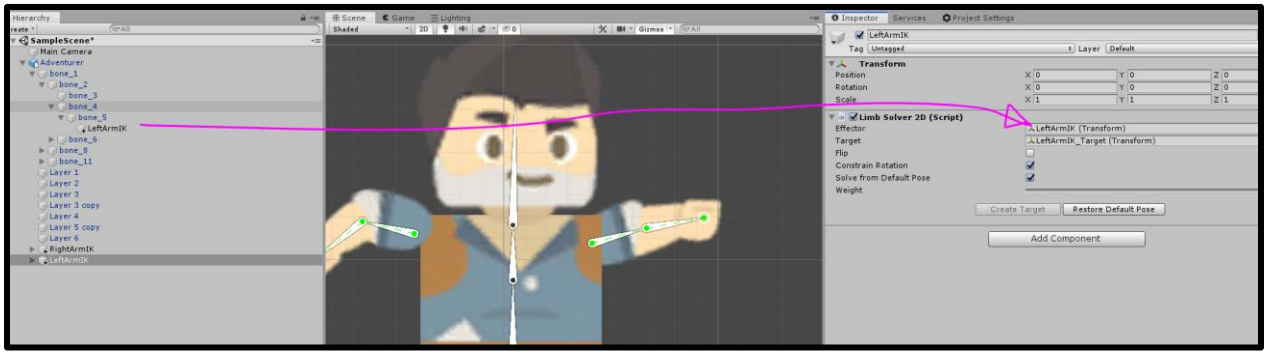
You can tweak the "Weight" slider which will change how much the IK will influence the other bones. "Solve from Default Pose" will calculate IKs from the default pose of the character and not what the current pose is. Leaving this checked or uncheck doesn't seem to affect anything in our project. "Constrain Rotation" is speaking of the rotation on the IK. Leave this checked even though we aren't using the rotation of the IK for anything. Now, we can do the exact same process for the Left Arm. Create a Limb IK, name it LeftArmIK.



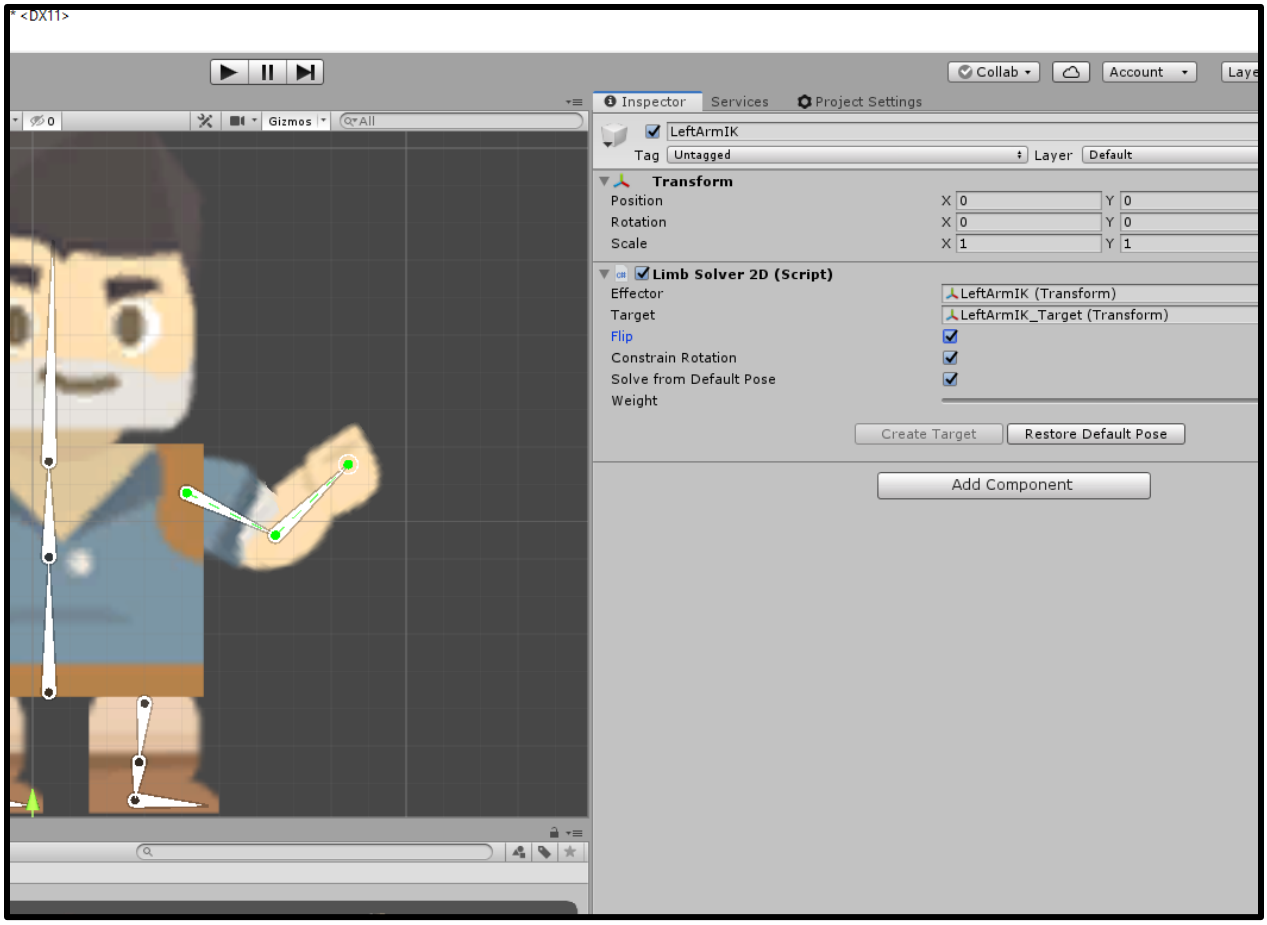
Create an effector at the left-hand position.



Drag it in on the LeftArmIK and hit "Create Target."

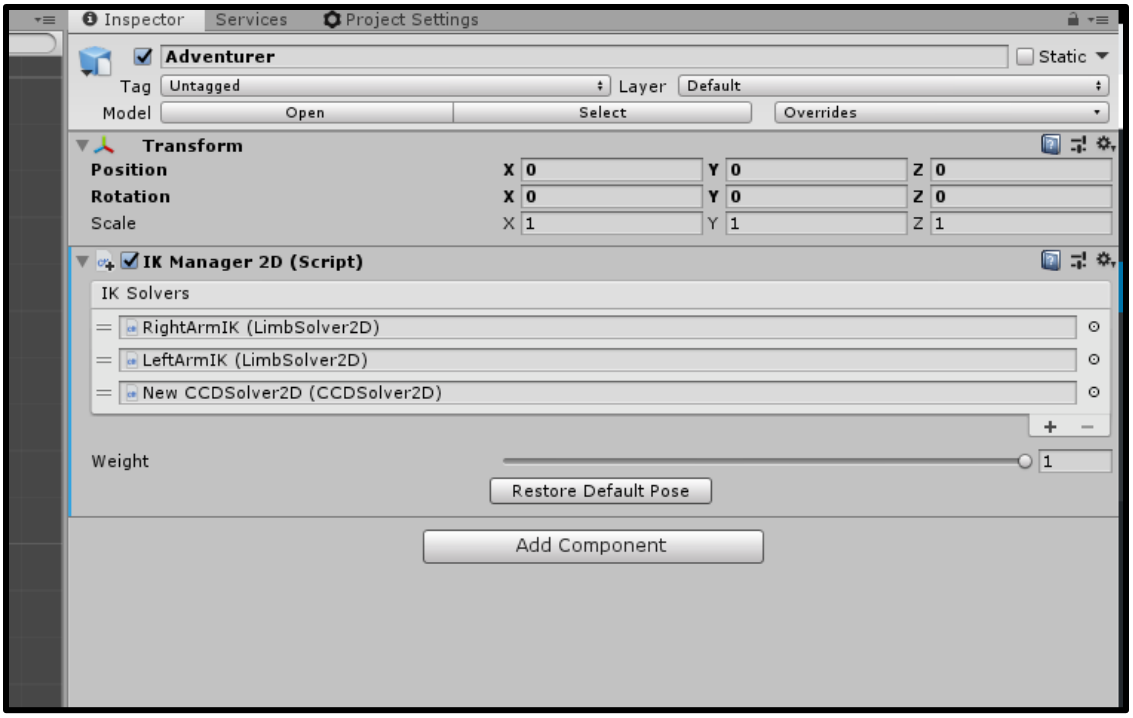


And enable "flip" if it is necessary.

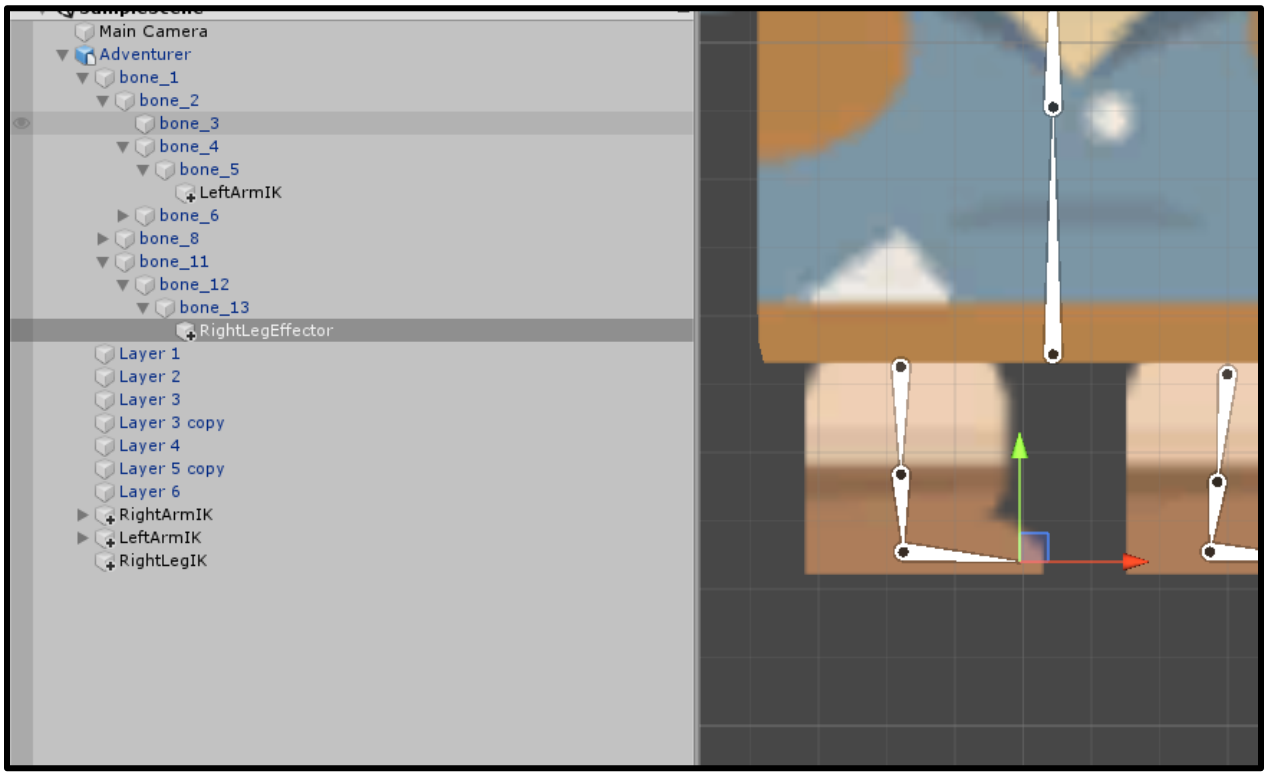


Leg IKs

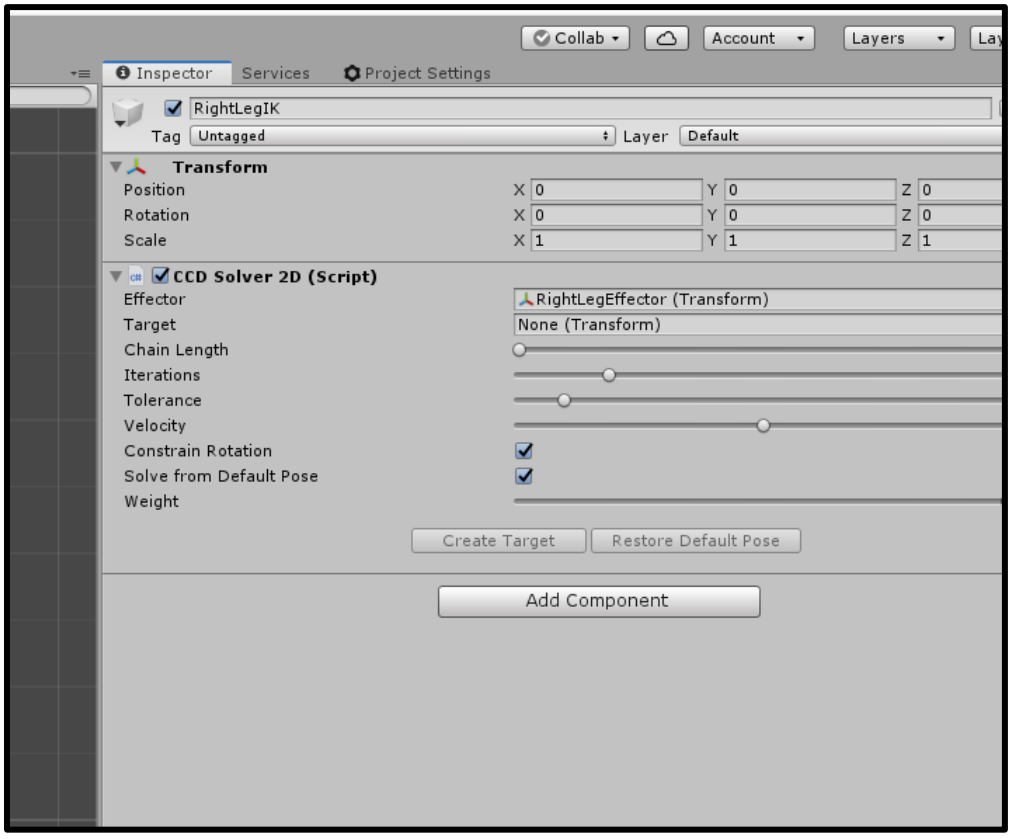
The final thing we need to do for our character rig is to create the Leg IKs. In this case, we are going to have one IK control the entire leg rig. Because this is more than two bones, we cannot us the Limb solver method. Instead, go to your IK Manager 2D script and create a Chain (CCD) solver.



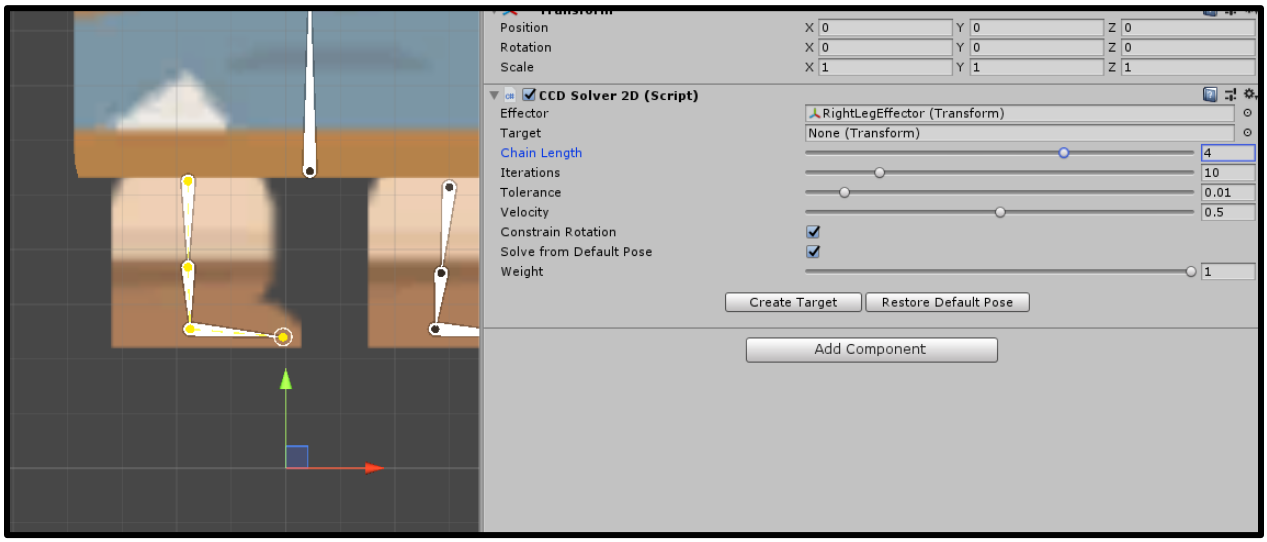
Rename this to "RightLegIK" and then select the toe bone on the right leg. Create an empty game object called "RightLegEffector" and place it at the end of the toe bone.



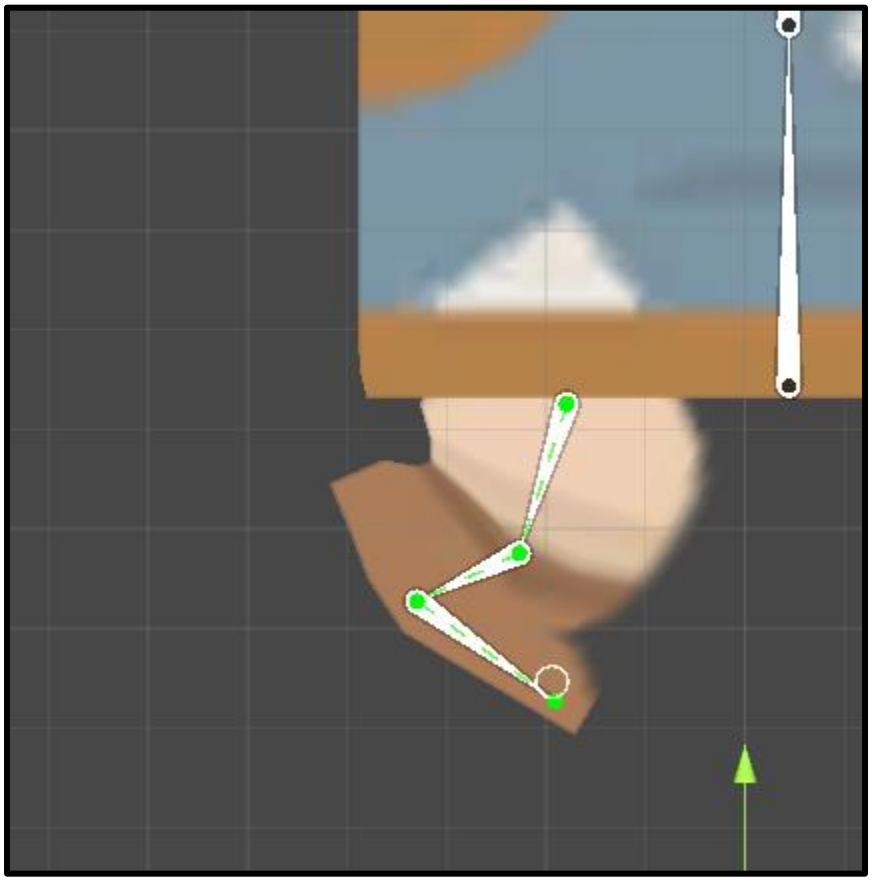
Drag this into the Effector field in CCD Solver 2D on the RightLegIK.



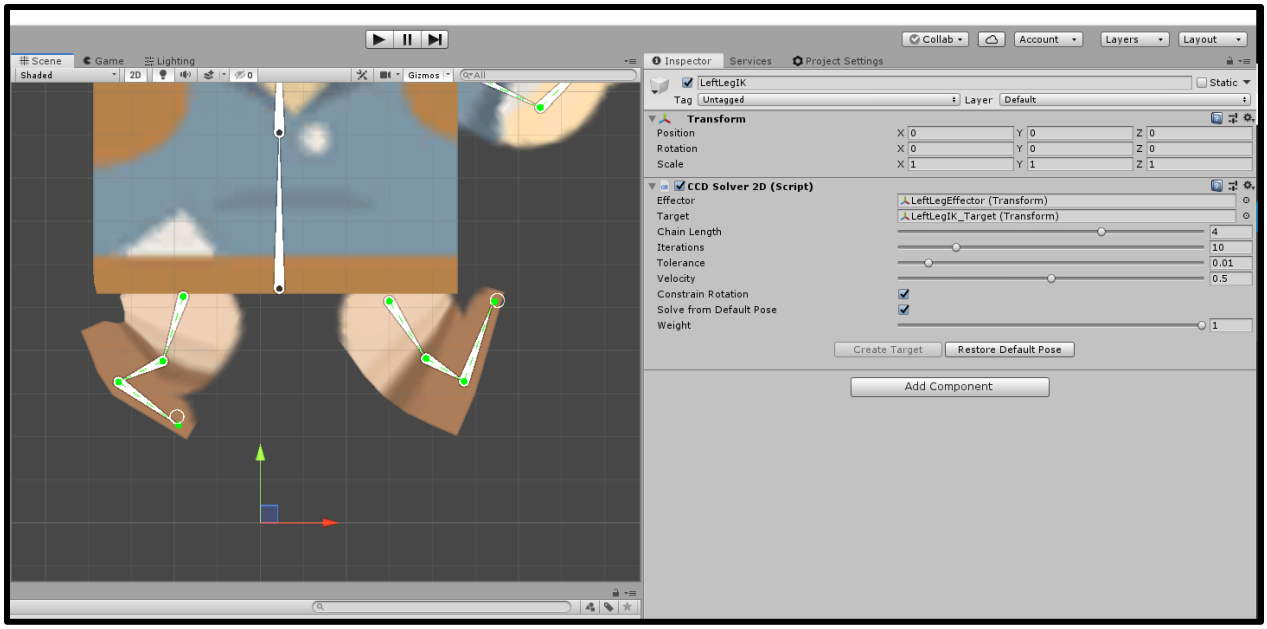
Now, we need to specify a chain length, basically, how many bones are going to be involved in the IK calculation. As you can see, if you drag the "Chain Length" slider up, the active bones turn yellow.



Therefore, a chain length of 4 is what we need. Let's have a look at the other settings before we press "Create Target." "Iterations" is the number of times per frame the IK algorithm will be run. Because our rig isn't that complicated, a value of 10 is fine. "Tolerance" is basically how precise the IK calculation will be. A lower value will be more precise. And finally, "Velocity" is basically how fast the bones will move to the effector. We don't need to change this value much because it is just a four bone chain. Now that we've had a look at the settings, hit "Create Target" and we'll see what we've got.



As you can see, we are now able to control the entire leg from this one IK. This is important, as you will see later when it comes to animating a run or walk cycle. We now have toe and knee control all in one effector! Now, just duplicate this process for the left leg.



Conclusion

Congratulations! You now know how to rig a 2D sprite, plus how to optimize that rig for animating by setting up IKs. This is a very important thing to know when it comes to 2D games. As I mentioned before, if you would like to do this to your own character, simply separate the limbs in Photoshop as different layers, and then export it as a PSB file. And now that we have a rigged character, we can animate it and script its controllers. All of this we will cover in a separate tutorial.

Keep making great games!

Animating a 2D Character in Unity

Introduction

At the very core of animation is an illusion. The "Illusion of Life" is what it is called. We think an animation looks the best when it produces a greater illusion of a living character. This makes sense, does it not? When we see an animation of a living character, it triggers a memory or recognition of a live being we have seen in real life. Therefore, the task of the animator is to closely follow the rules of movement that we see in the real world because that will greatly increase the illusion that what you are seeing is a real being, moving and functioning. And this is the goal of this tutorial. We are going to start creating this "Illusion of Life" for a 2D character.

This tutorial uses a character that was rigged in this tutorial: Rigging a 2D Character in Unity. Please check this out if you would like to learn more about the sprite rigging process in Unity and if you want to be able to follow along in this tutorial. We will be animating a character to run, walk, idle, and jump. The end result is a sprite ready to be scripted into a 2D character for your game.

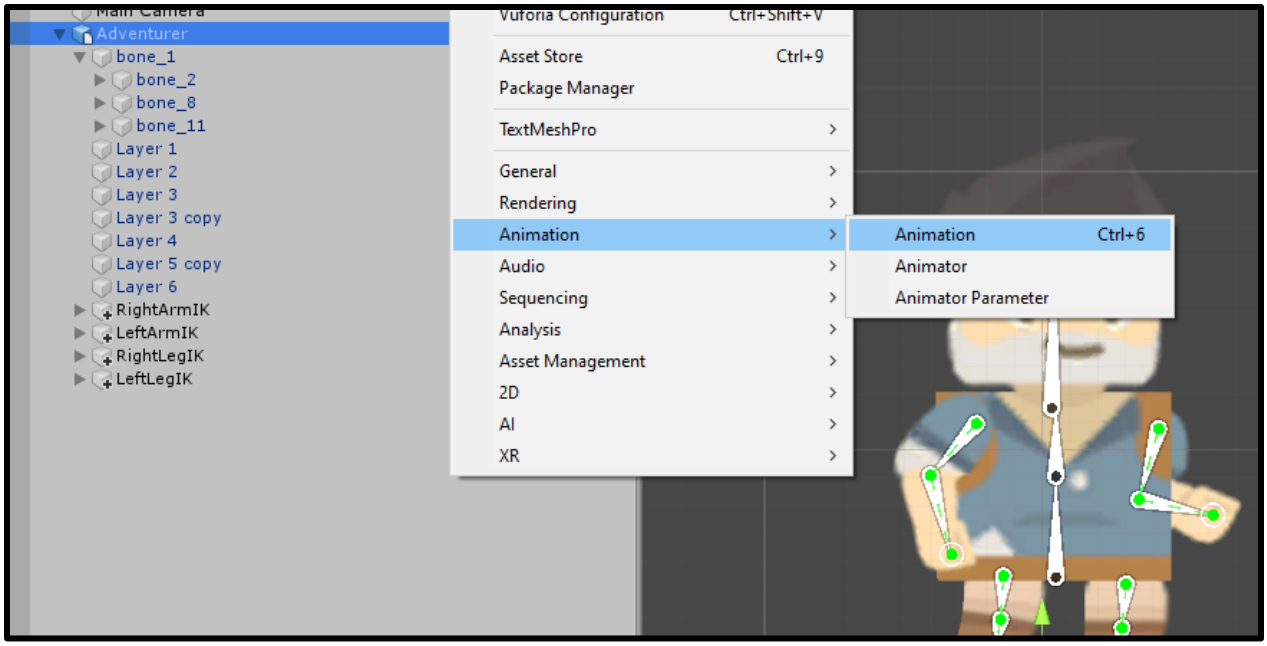
Requirements and Assets

Basic knowledge of the Unity Editor (how to navigate and open tabs) is required along with how to work with the Unity Animation window (here is a tutorial that will tell you all you need to know about it: How to create animations from models and sprites). You will also need the character that was rigged in the previous section. You will not be able to follow along without that character. This was rigged with the Sprite Skinning features in Unity and this contains the "2D Animation," "2D IK," and "2D PSD Importer" packages that are necessary to run this project. Also, this tutorial will focus on animating *not* the Unity Animator. If you are looking for information on the Unity Animator, check out our comprehensive guide in the first section.

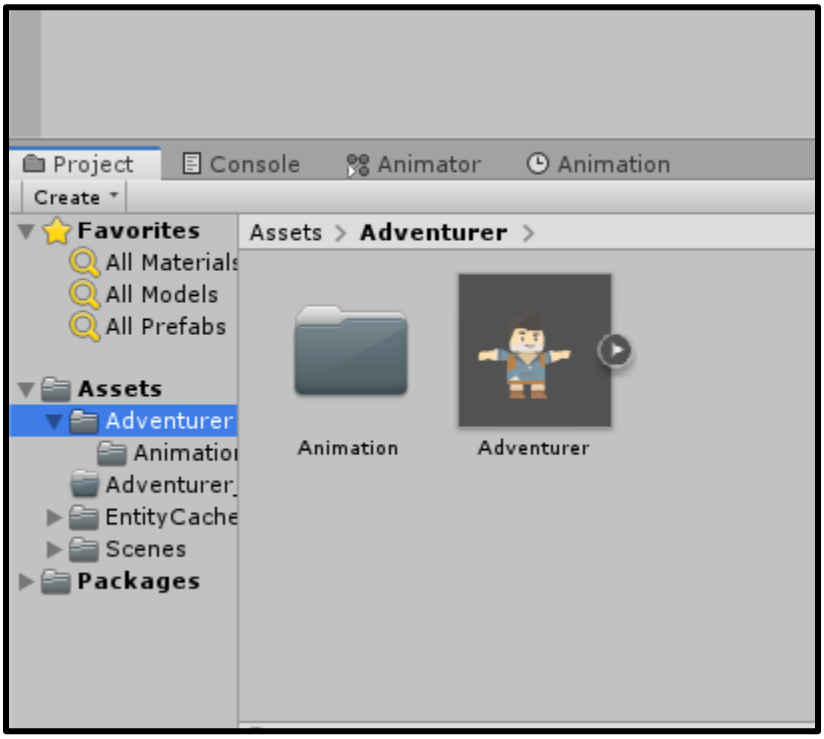
The source code for this project can otherwise be found here.

Animating!

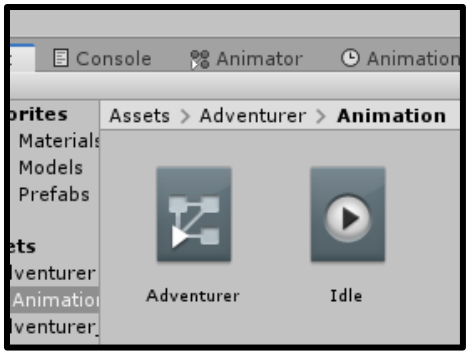
Select the Adventurer character and open up the Animation tab.



Dock this next to your project and console tab. Now, go to your project tab and open up the Adventurer folder. Create a new folder called "Animation."



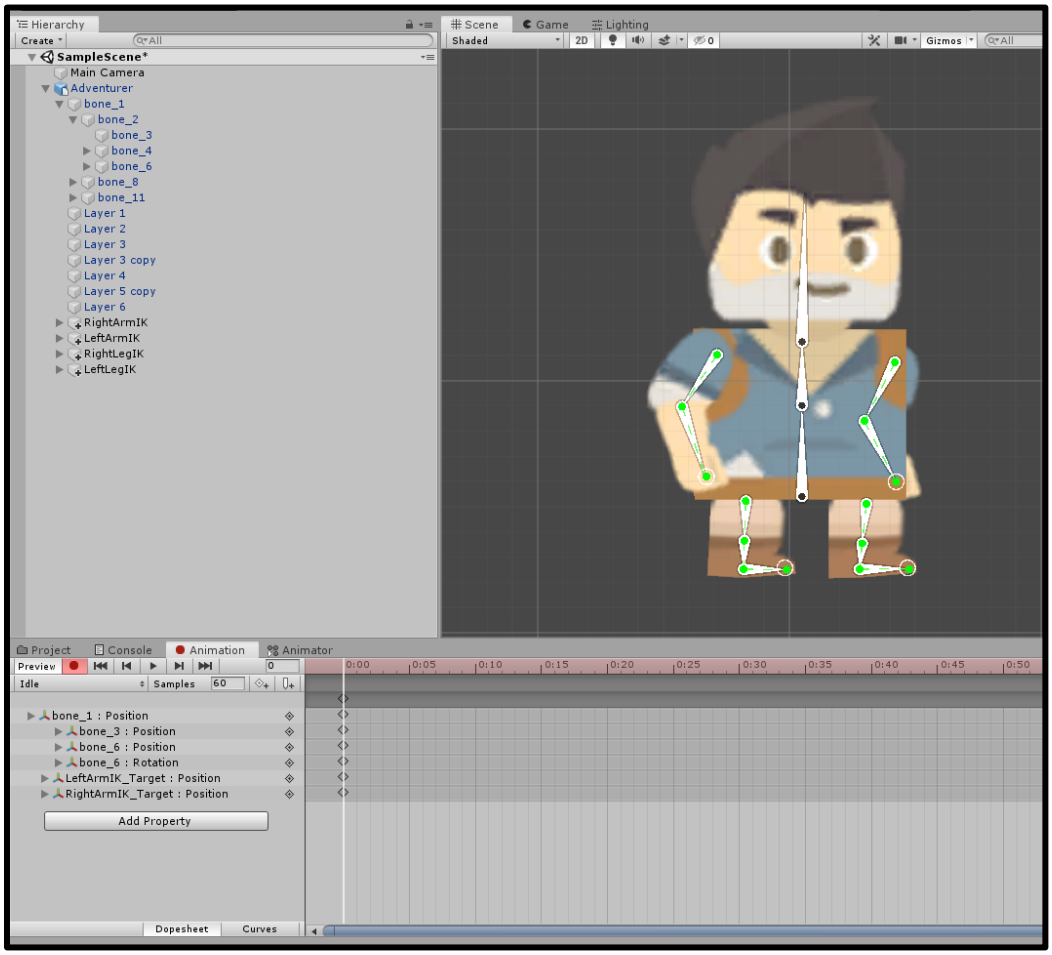
Now we go to the Animation tab and create a new animation called "Idle" that will be housed in our Animation folder. This will also create an Animator Controller in that same folder.



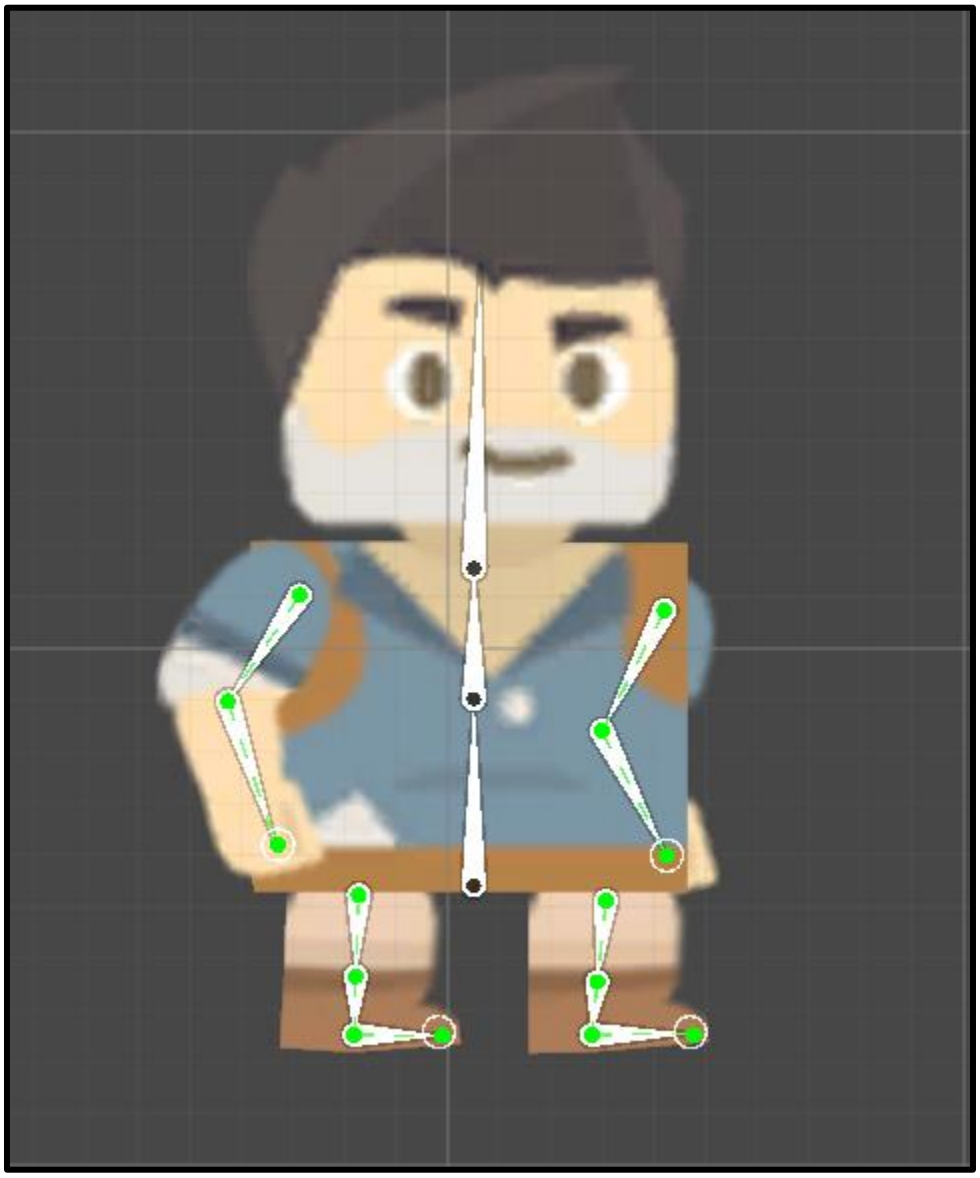
Now, let's start animating this character!

The Idle Animation

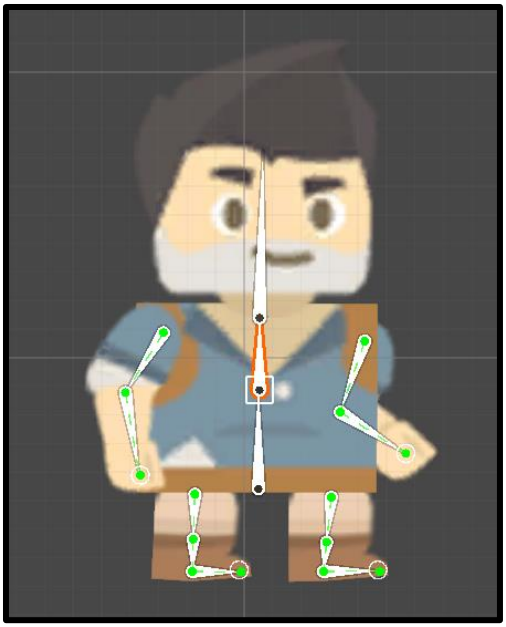
When it comes to the idle animation, the IKs really contribute a lot to the efficiency and speed of animating. We just need to animate a subtle movement, almost like just a breathing motion. Hit the red record button and move the character to this position:



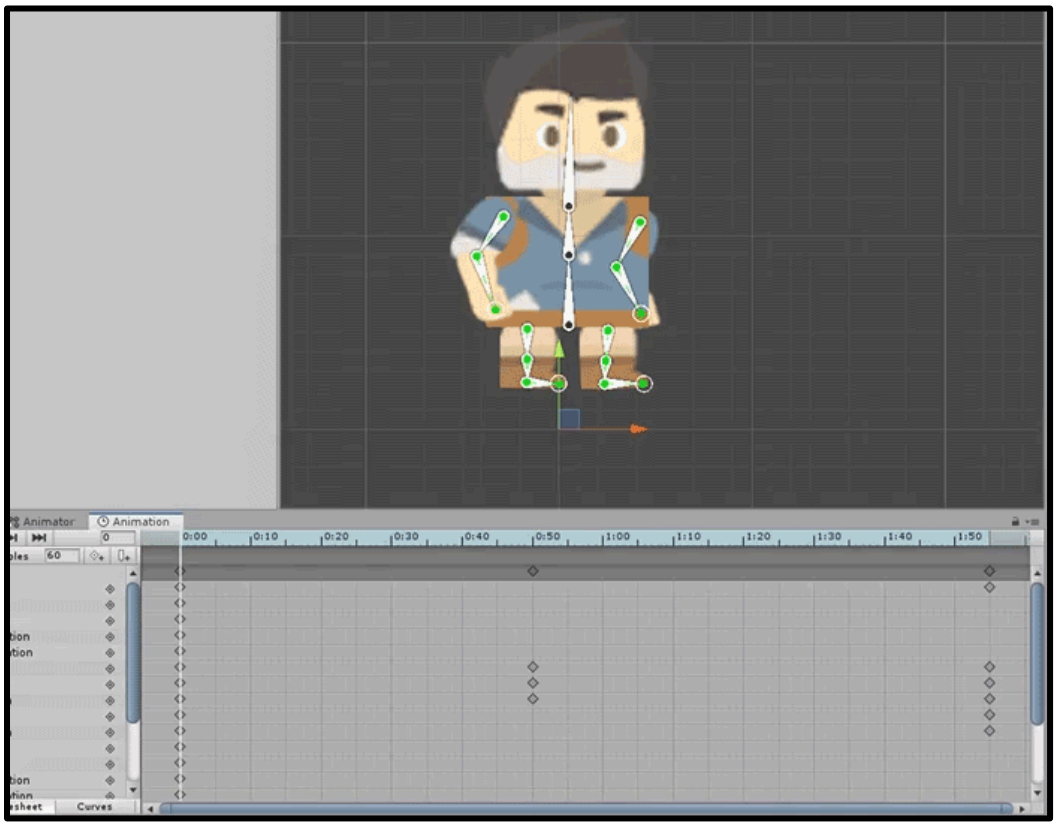
Now go about fifty frames ahead and just barely move the top torso bone straight down. Move the arm IKs to the side so that there is a bit less bend in them and so that there is a sort of "swinging" motion. Here is the pose that I chose:



This is just a subtle tweak. If we make large movements, it makes the idle animation look strange. Next, select the first group of keyframes and hit Ctrl-C (Command-C on a Mac). Then paste these keyframes on frame number 1:50.

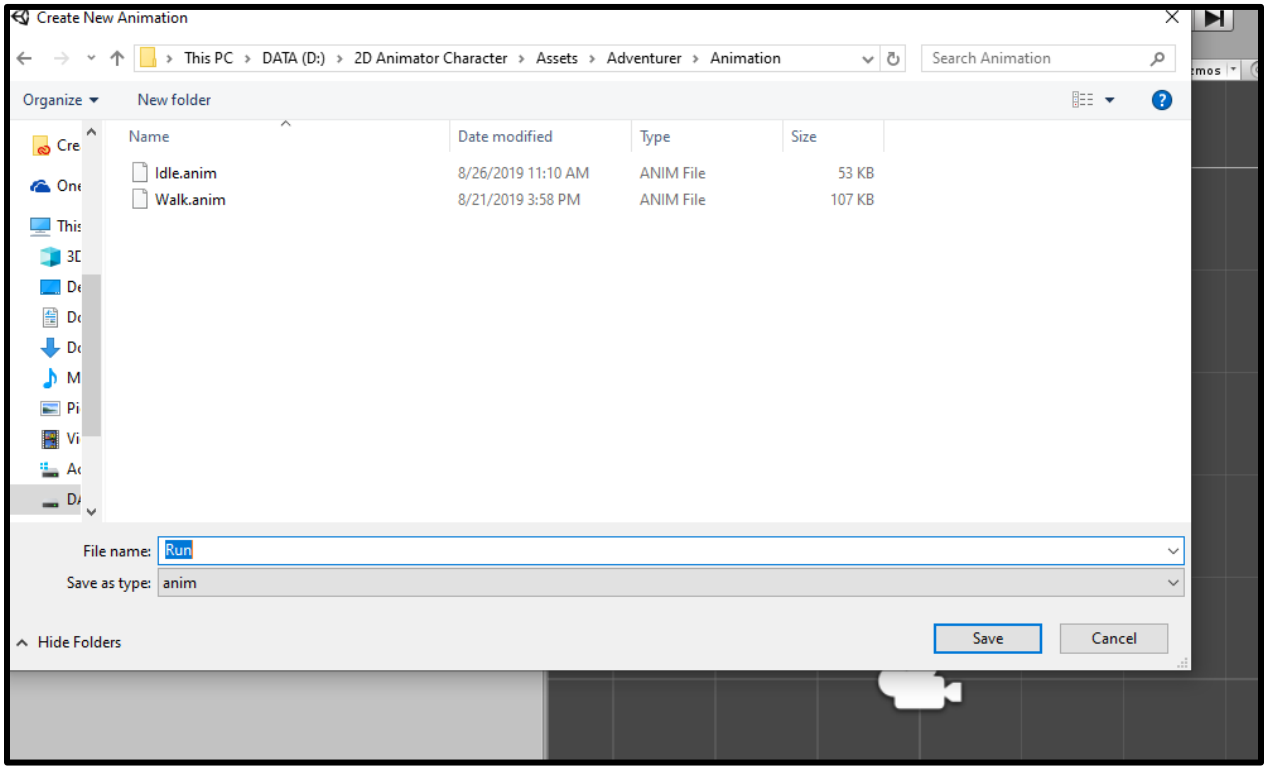


Now, we have a nice looping animation! It looks decent but I think we need to add some head motion to enhance the illusion. Just have the head rotate backward ever so slightly. As you can see, it makes a huge difference. Alright! Congratulations! We now have an idle animation!

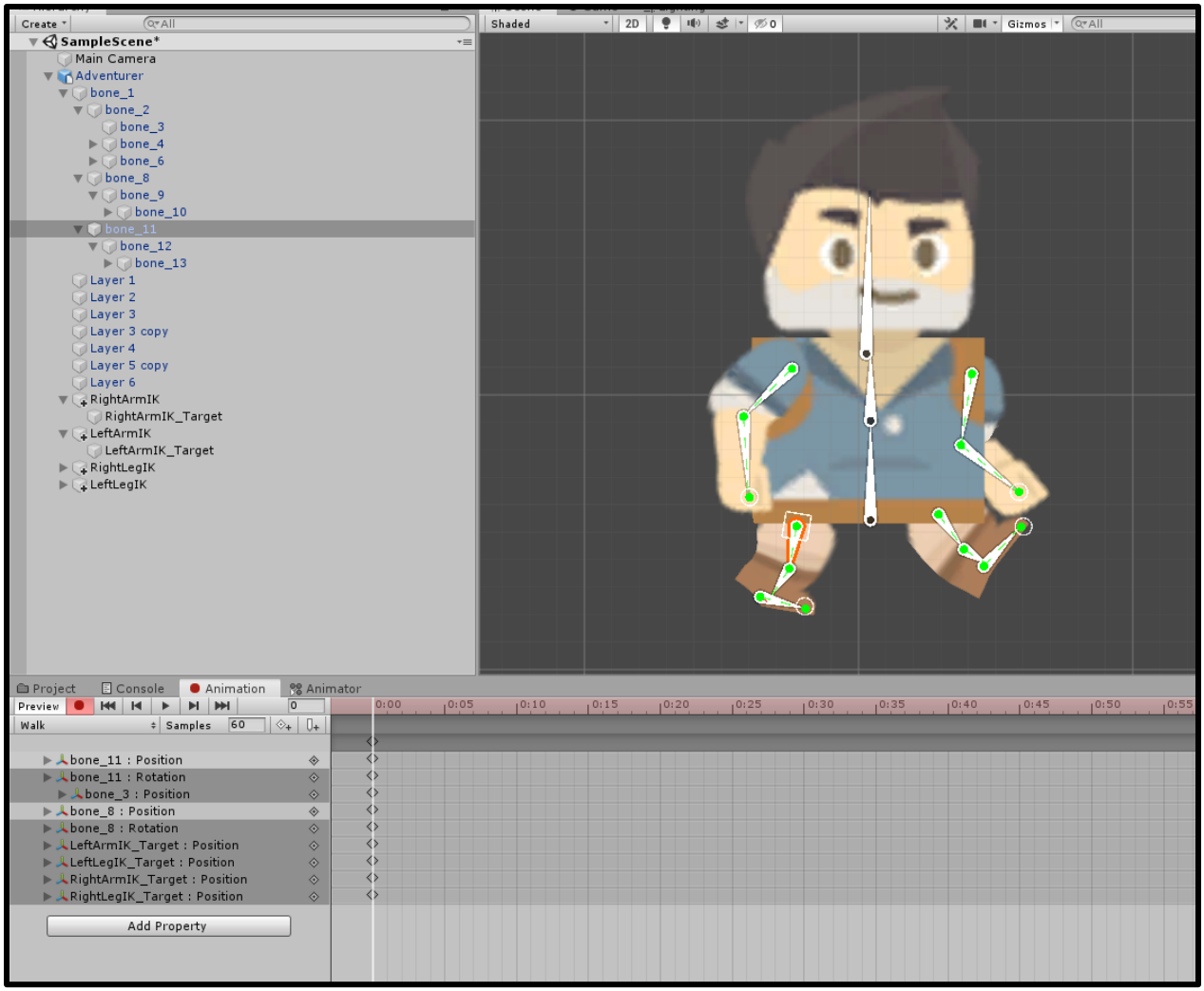


Walk Animation

Create a new animation clip called "Walk."

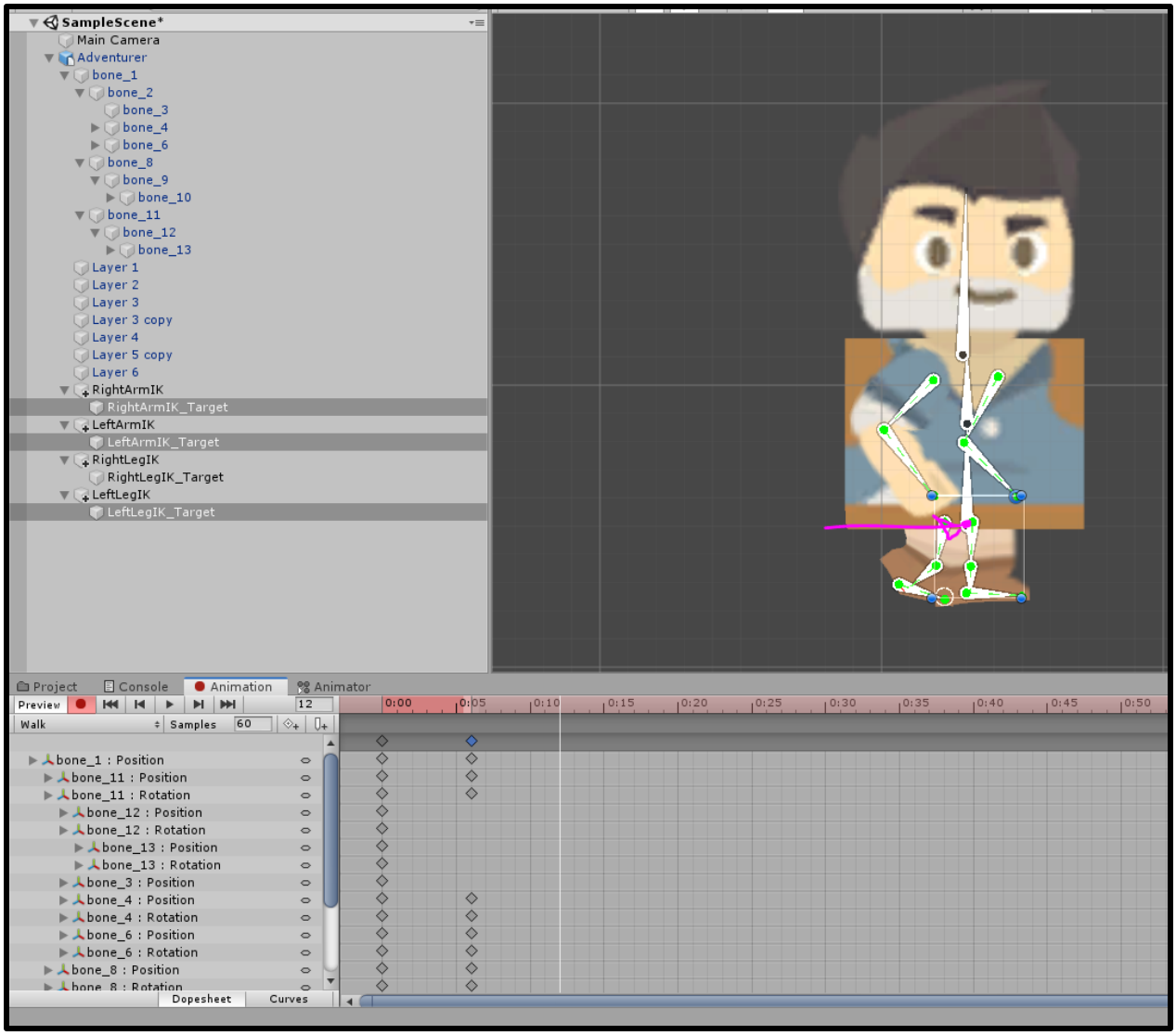


This animation is a bit more technical. Here, we need to know a few important things about the walk locomotion. The main feature of a walk animation is that the character is always touching the ground. We should never have a pose where our character is airborne. Also, I use the word "pose" intentionally. When we animate in 2D, an action is made up of a few key poses. In the case of the walk animation, we have one pose like this:



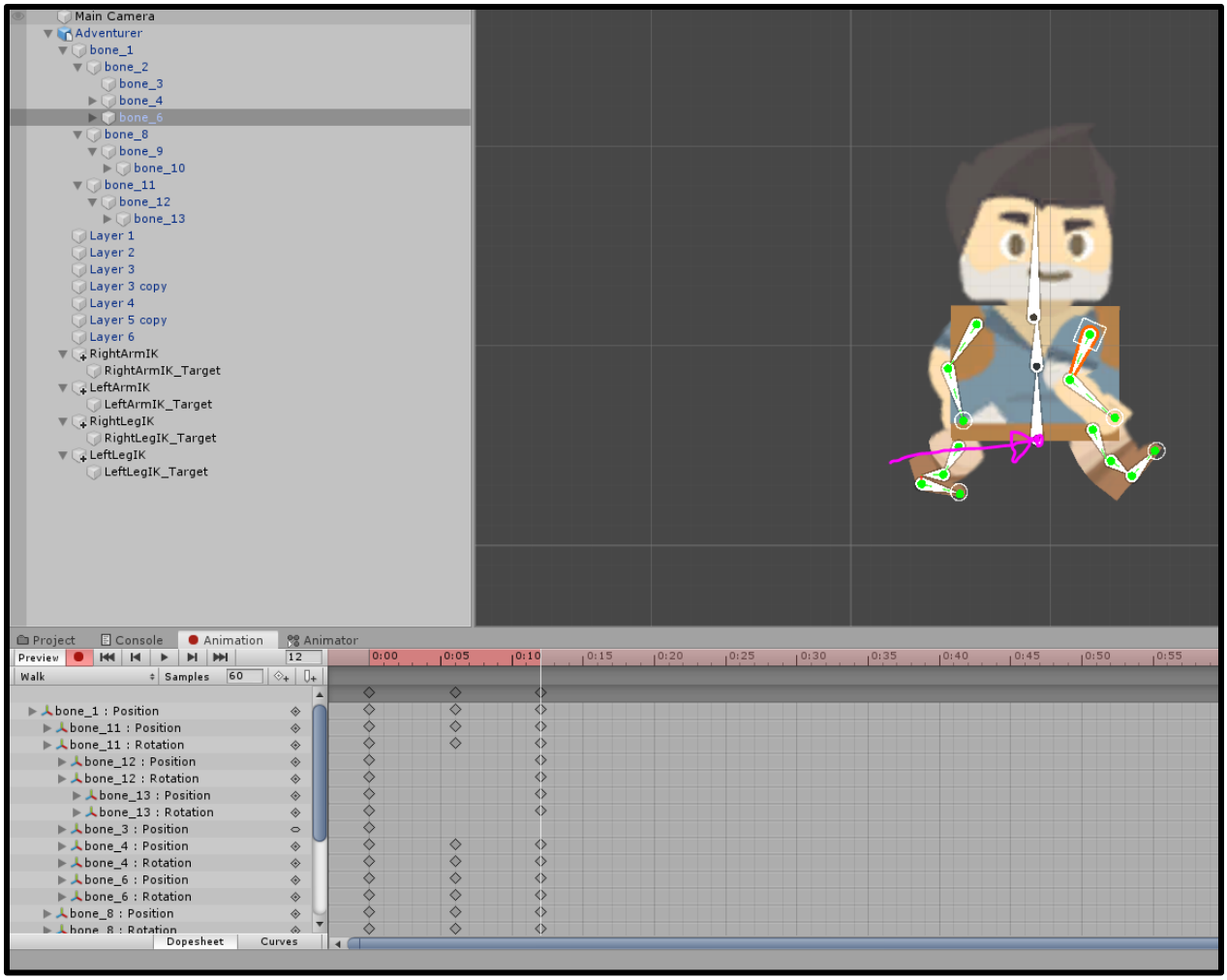
As you can see, here we have the right foot (relative to the character) touching the ground.

Then, about six frames ahead, we have what's called a "passing pose."



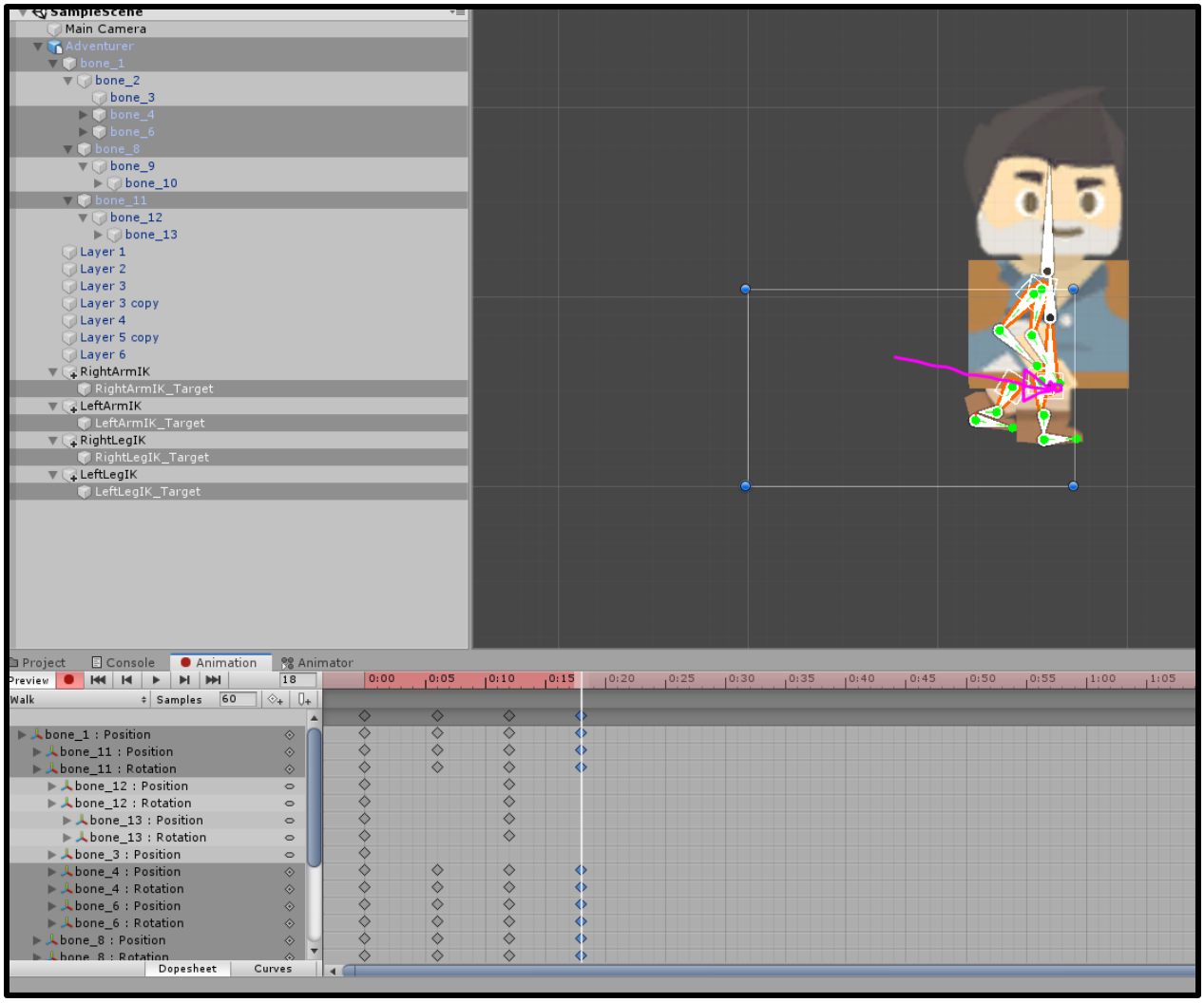
Here, there is a slight movement downwards. This adds a realistic bobbing motion. We also animate the character moving forward by moving the entire character game object. Don't move the character by any other means except by dragging the entire character object.

And then another "dynamic pose" with the left foot touching the ground this time.

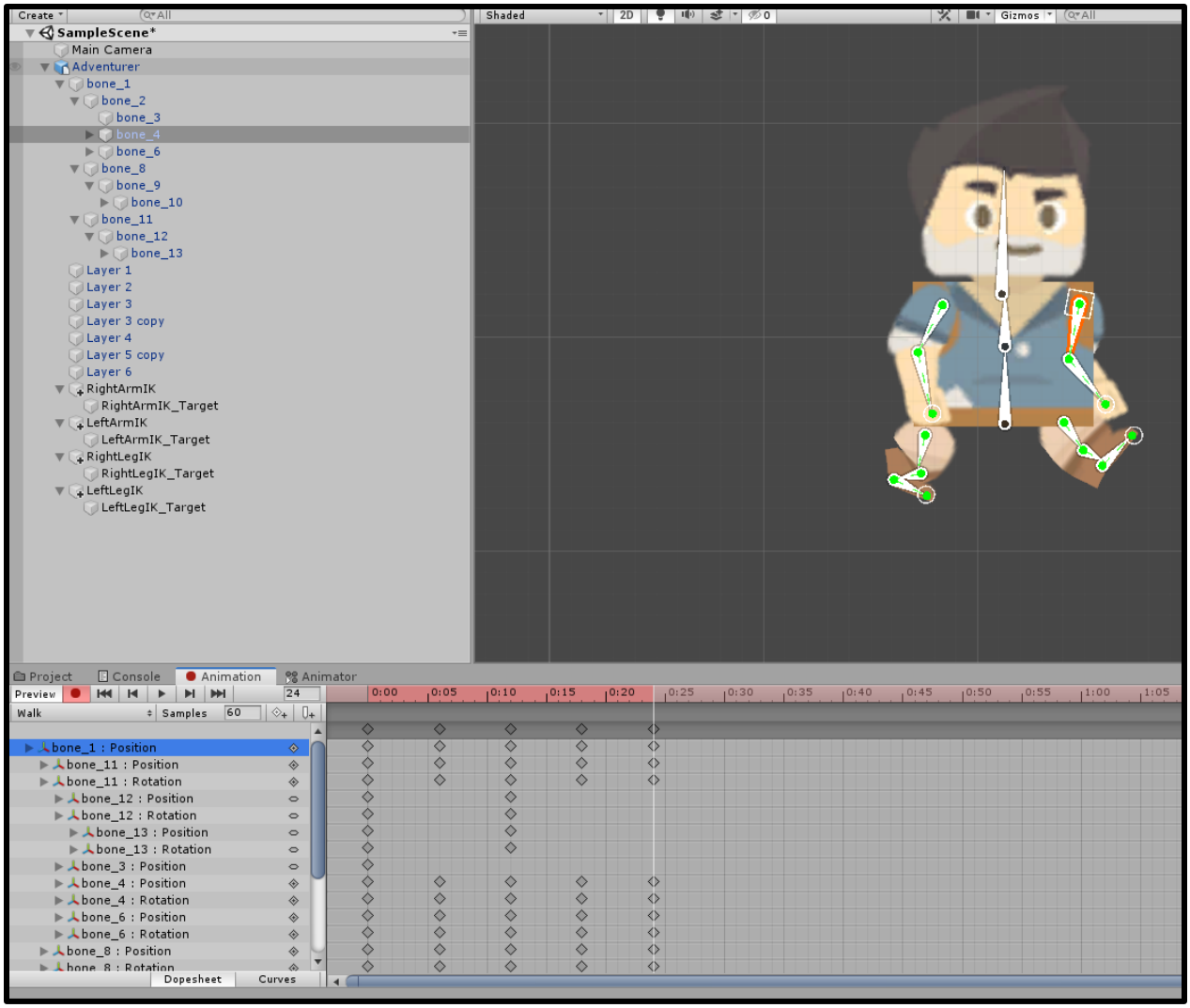


This occurs at frame twelve and we are still moving the character forward.

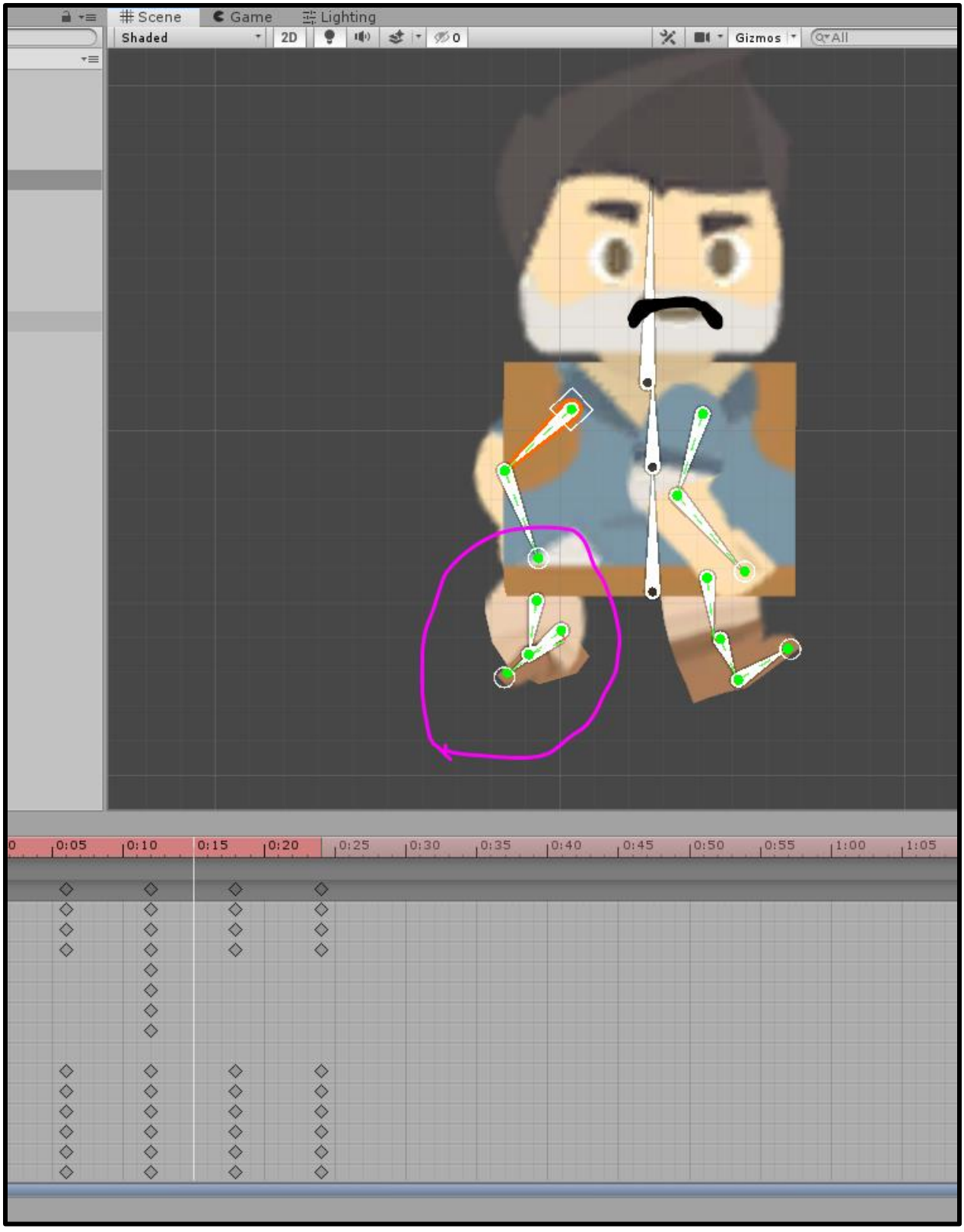
Now, we just need a final passing pose.

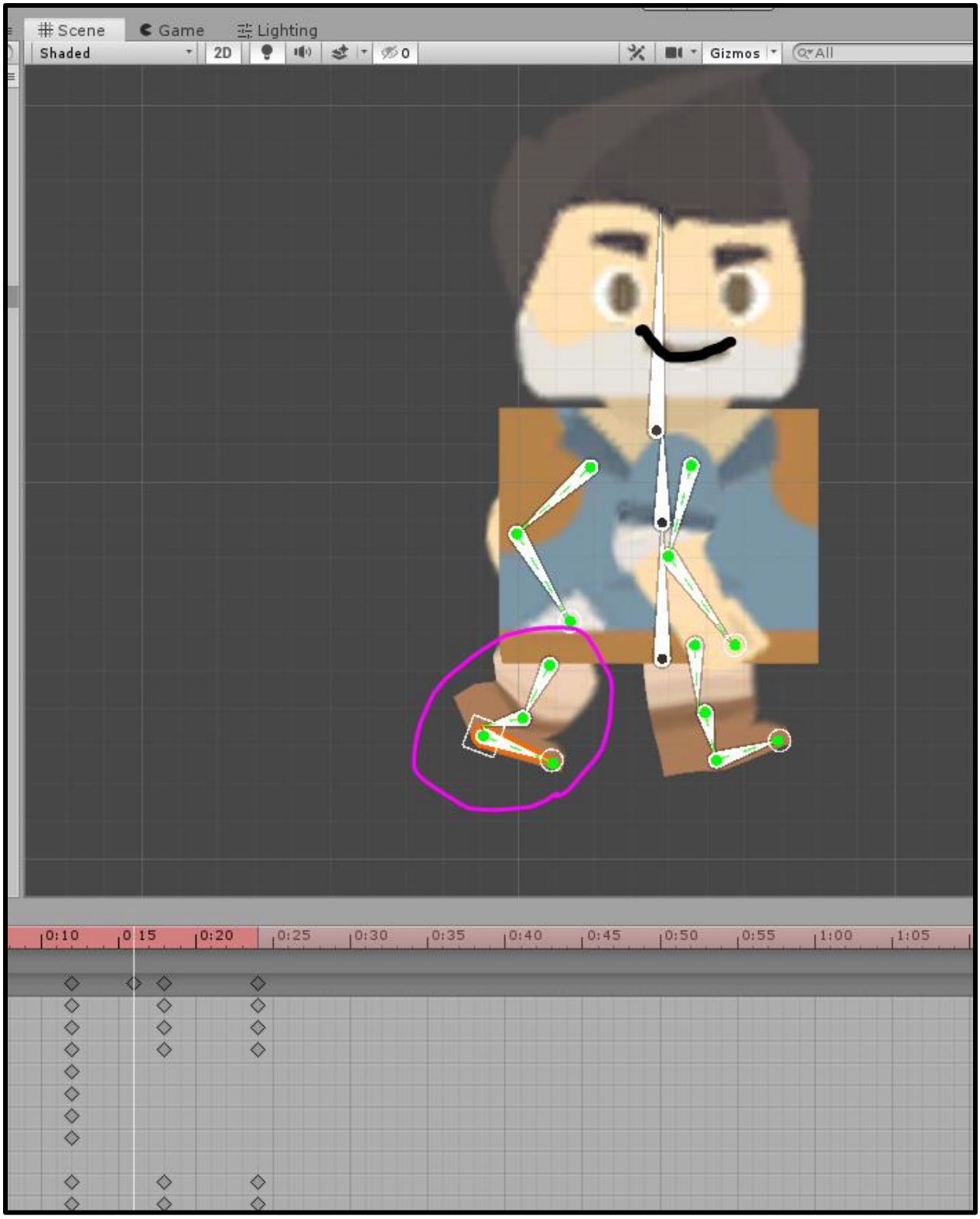


And then we recreate the same pose as the very first frame!

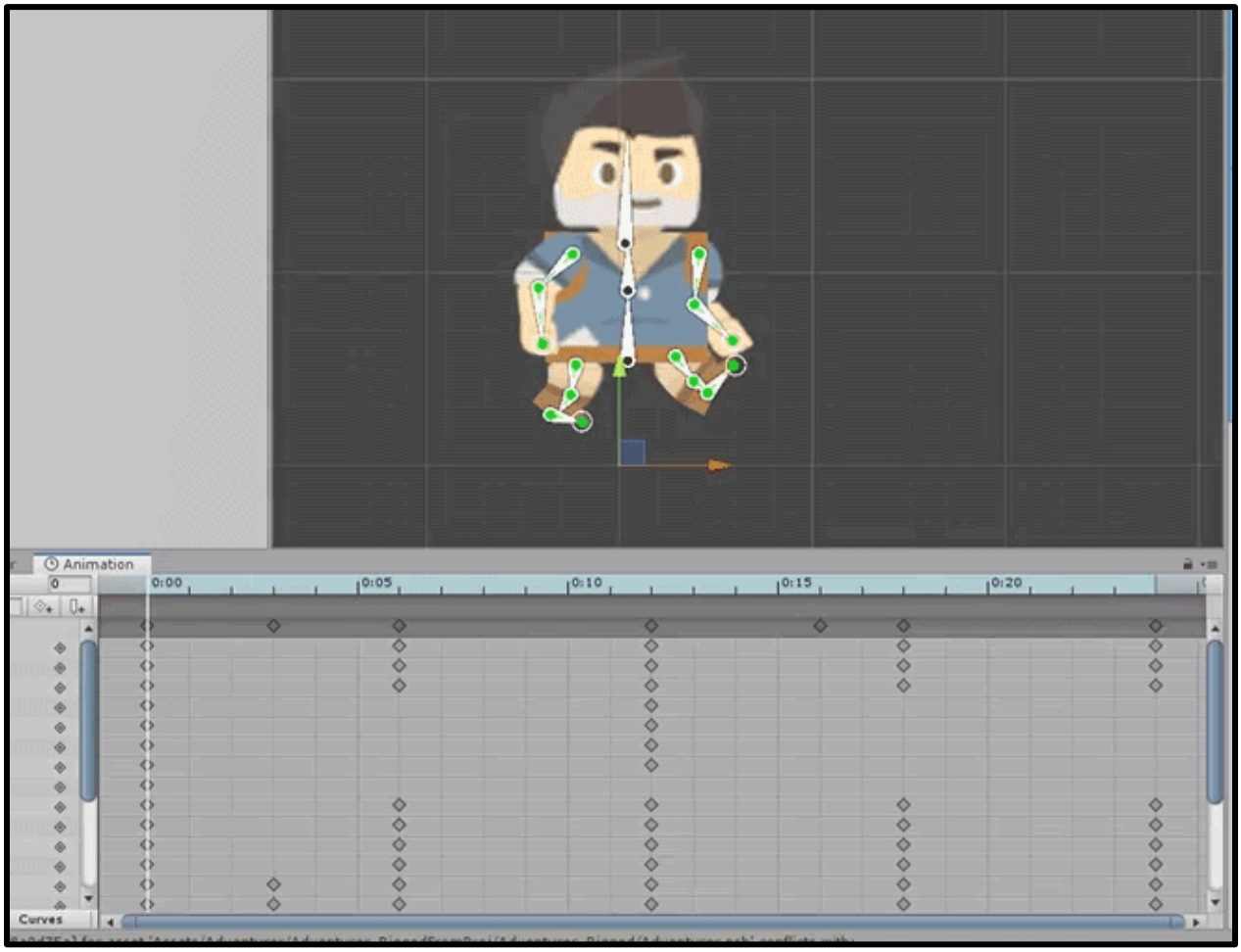


Simply copy the first keyframe and paste it at this frame. Reposition the character on the X-axis so that our forward motion matches. Now, just go through your animation and tweak any limbs that are bending the wrong direction.





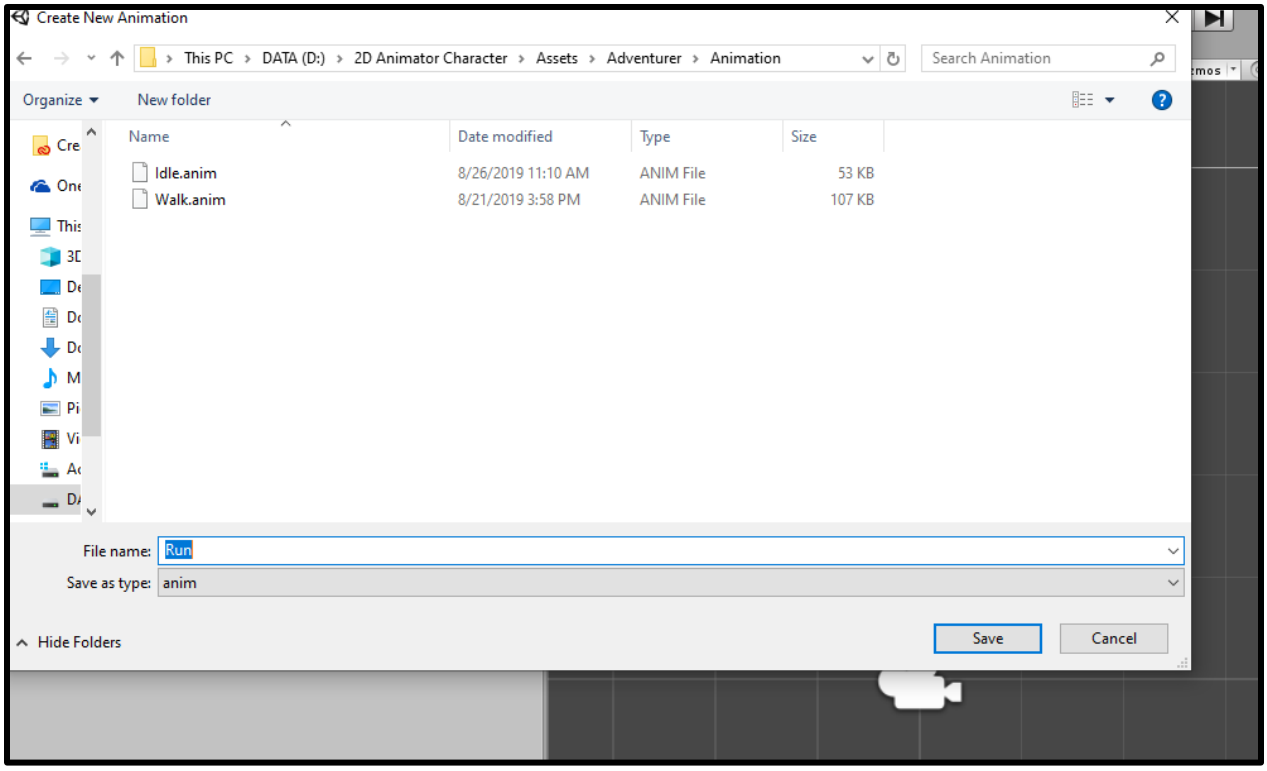
We are done!



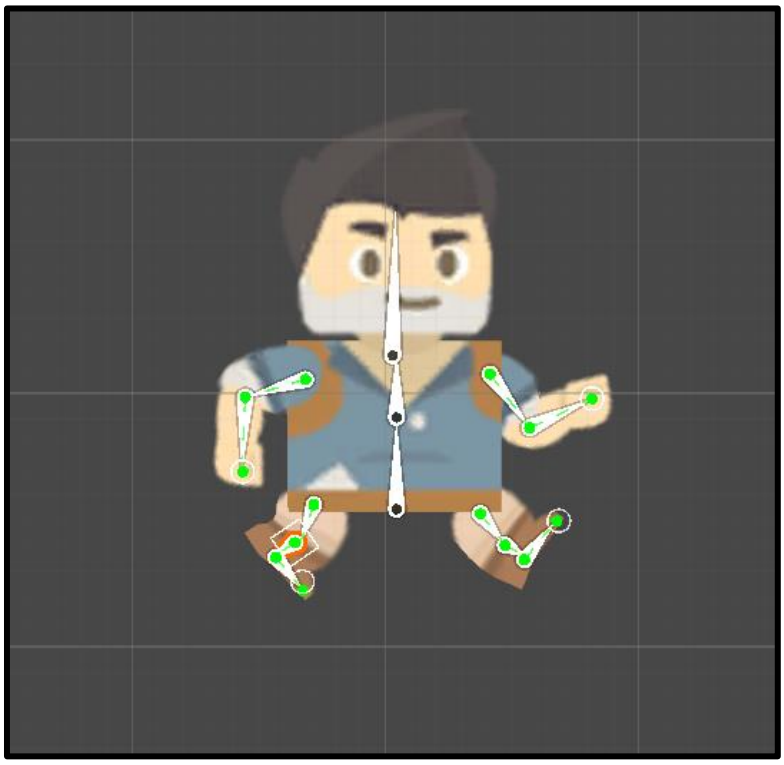
Run Animation

The Run animation is similar to the Walk animation except, this time, we need to have one pose where the character is airborne. The distance covered by one step is also greater than the walk animation. Other than that, the run animation has the same general structure (*dynamic pose to passing pose to dynamic pose to passing pose etc)* but we add one extra pose in order to enhance the illusion.

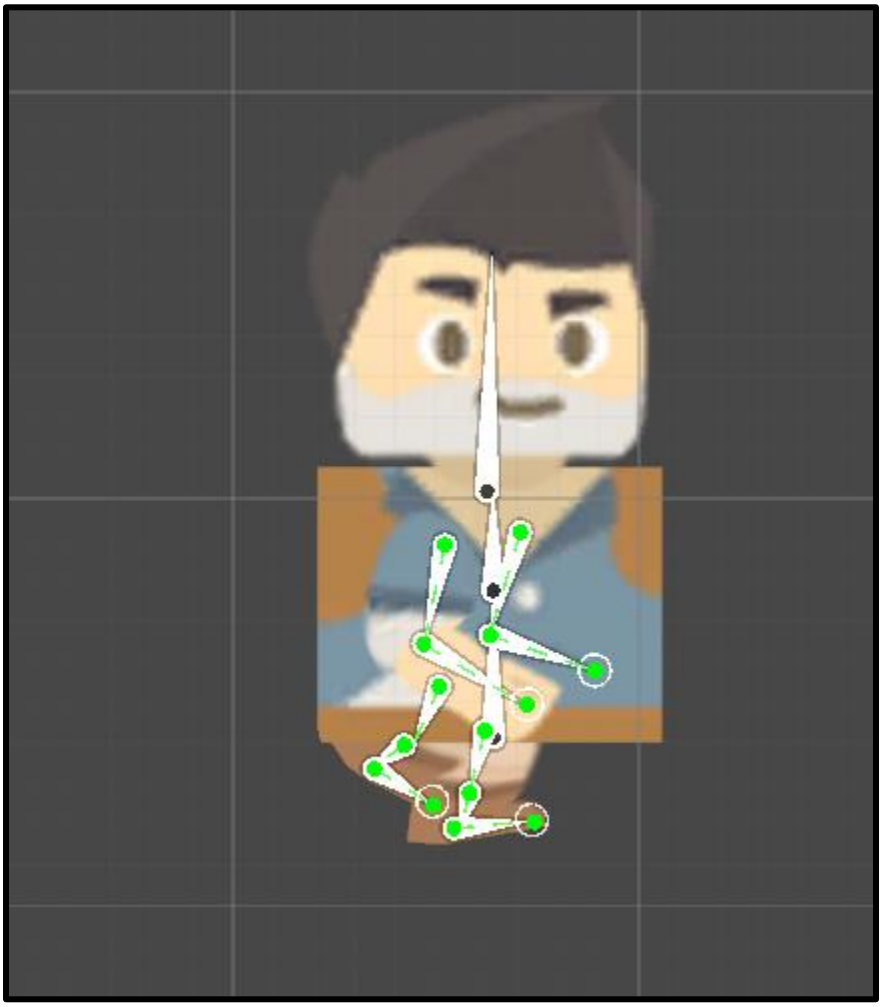
Create a new animation clip called "Run."



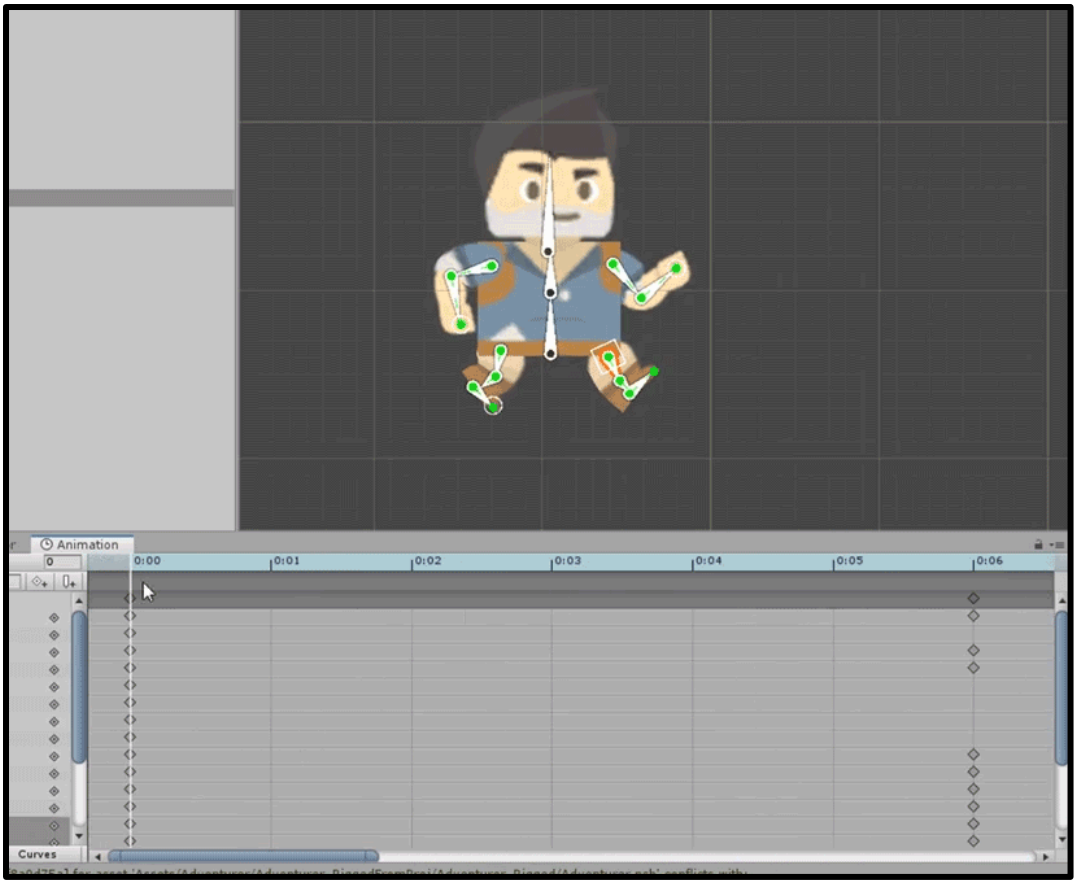
Hit the record button and move the limbs to this pose:



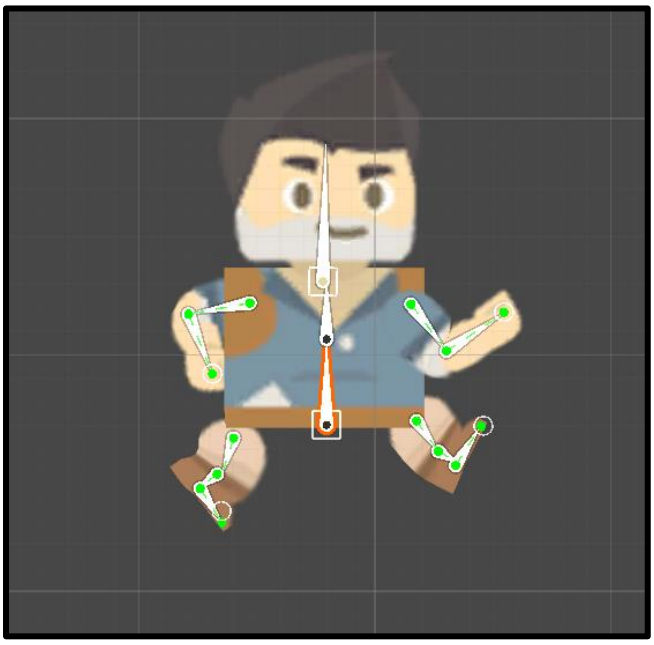
The armature, I found, was very fiddly so be patient while trying to get it to this pose. This is, of course, our first dynamic pose which means that we need to have the entire character move slightly up on the y-axis. Then, after moving six frames forward, we drag the character forward and down while repositioning the limbs to create our first passing pose.



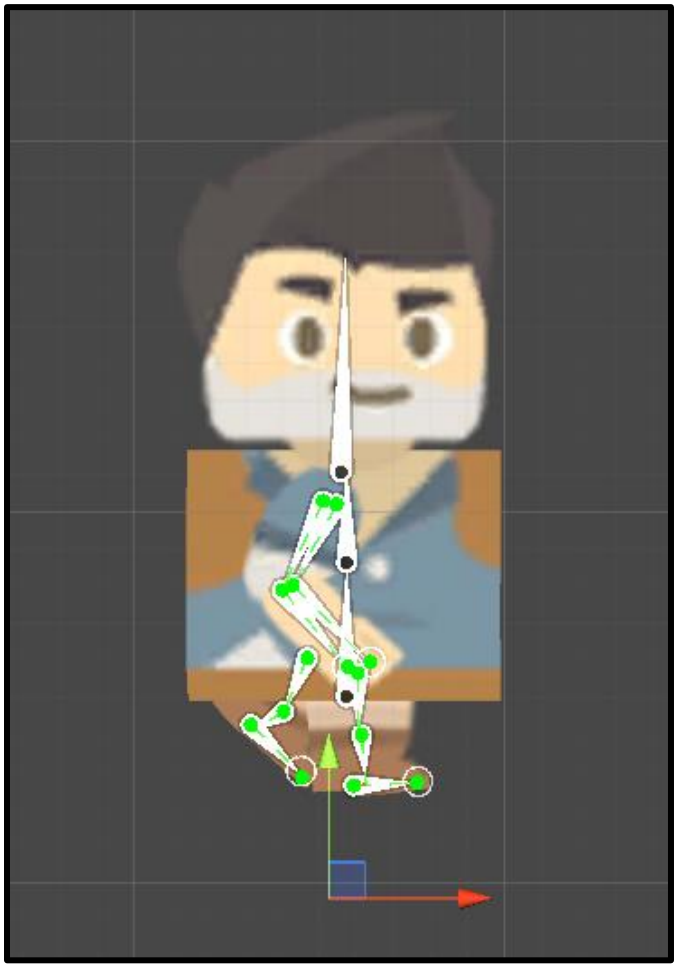
Scrub through these two keyframes to see if you have an animation that works. Reposition the feet so that there is no foot sliding.



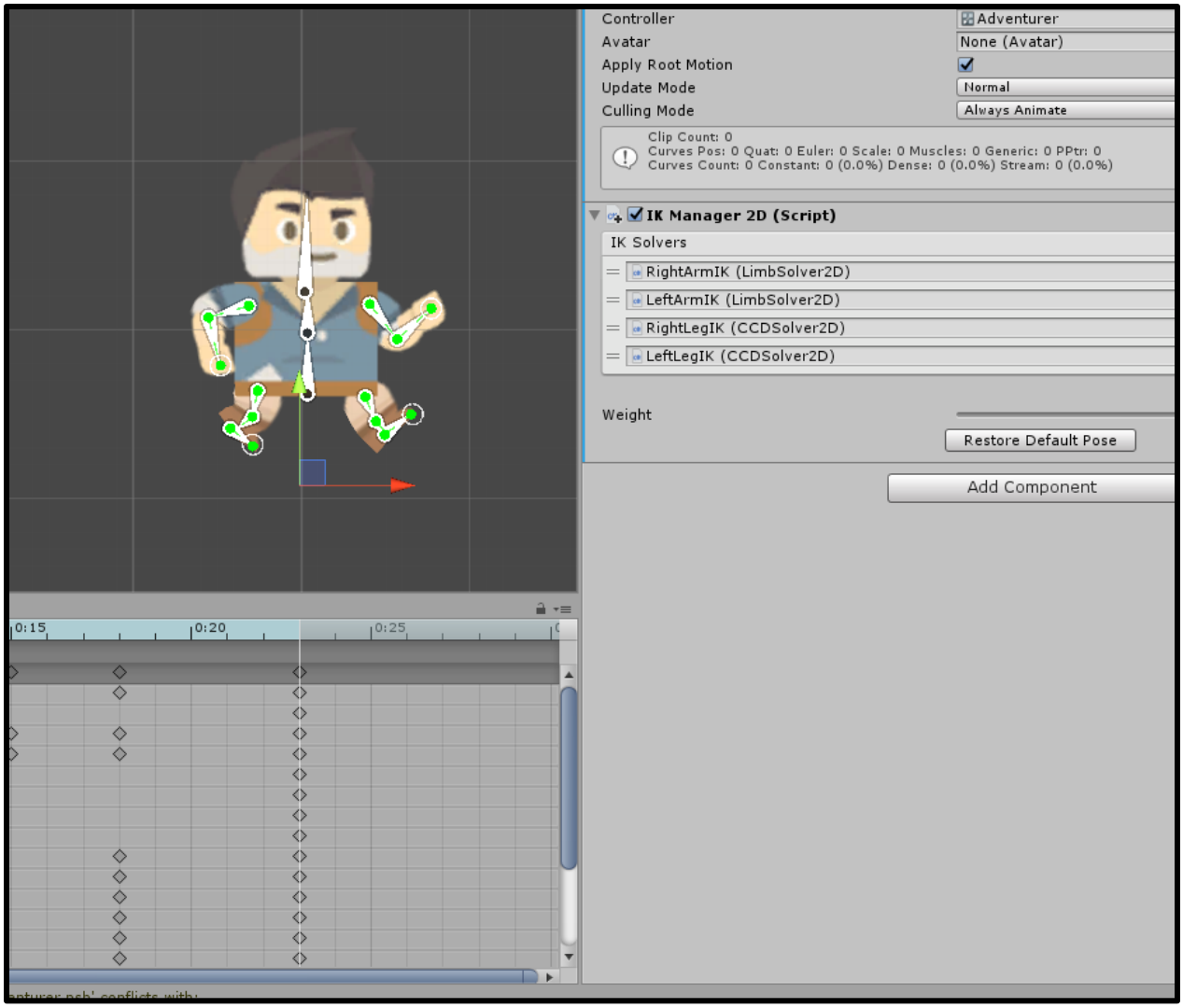
Now, our second dynamic pose occurs six frames forward:



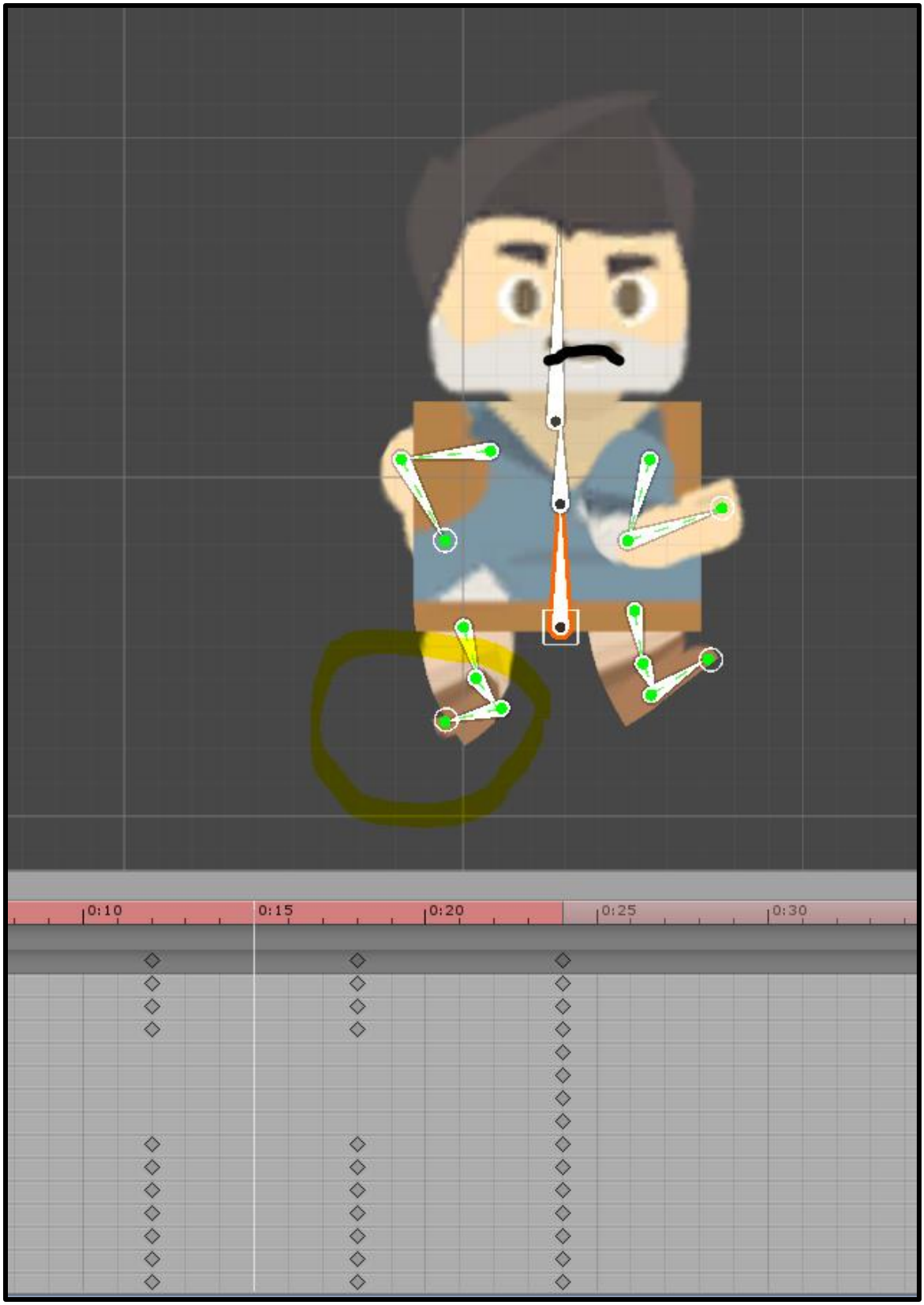
And then we have another passing pose at frame number 18:

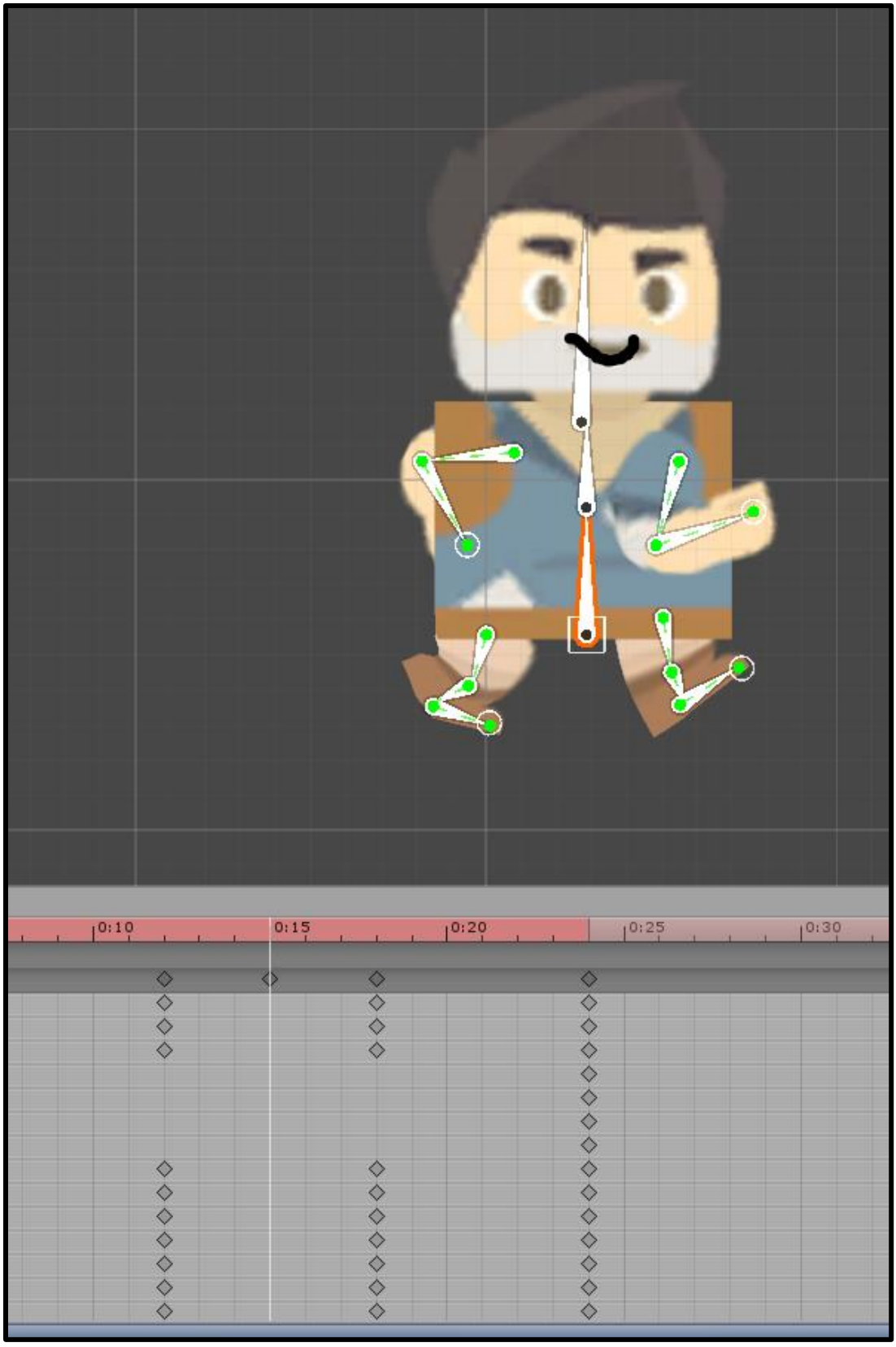


And finally, we have our final dynamic pose where we try and make it look like the first pose. A good place to start is to copy the first frame and paste it at the end. Drag the character forward on the X axis into the correct place. Now move the limbs into roughly the same position as the first pose.

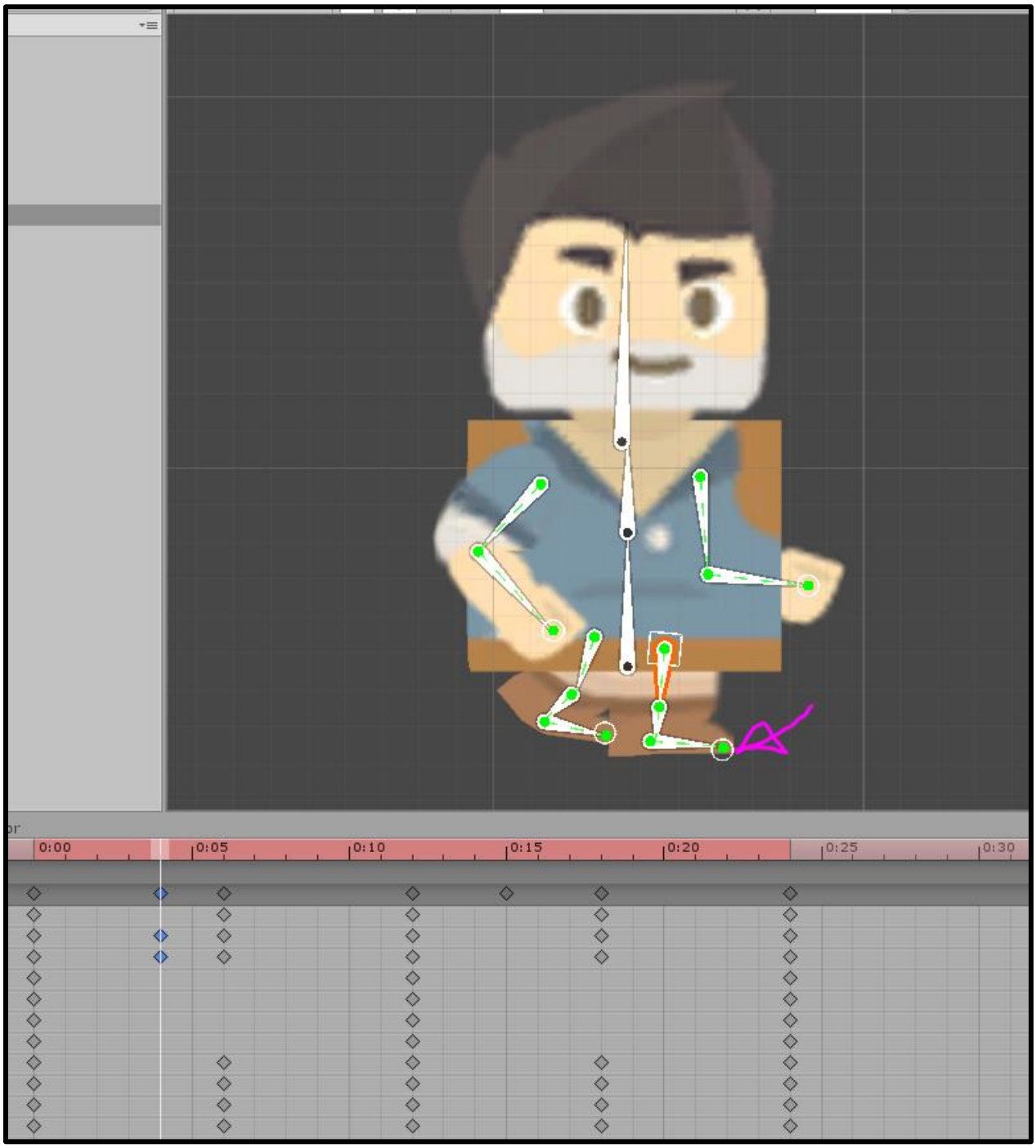


Now, we need to go back through the animation and do two things. First, we need to correct any limbs that are bent in an unnatural manner.

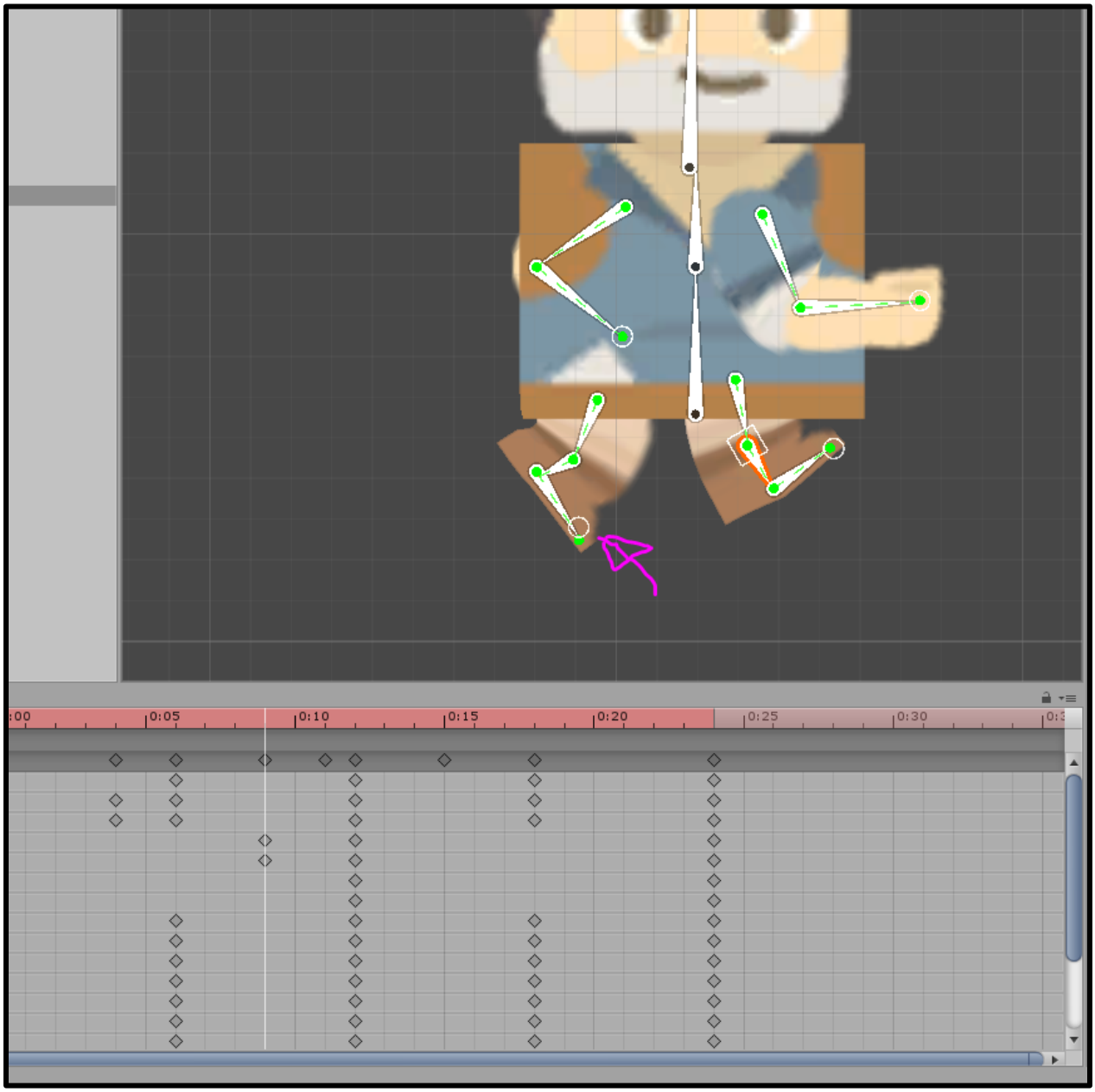




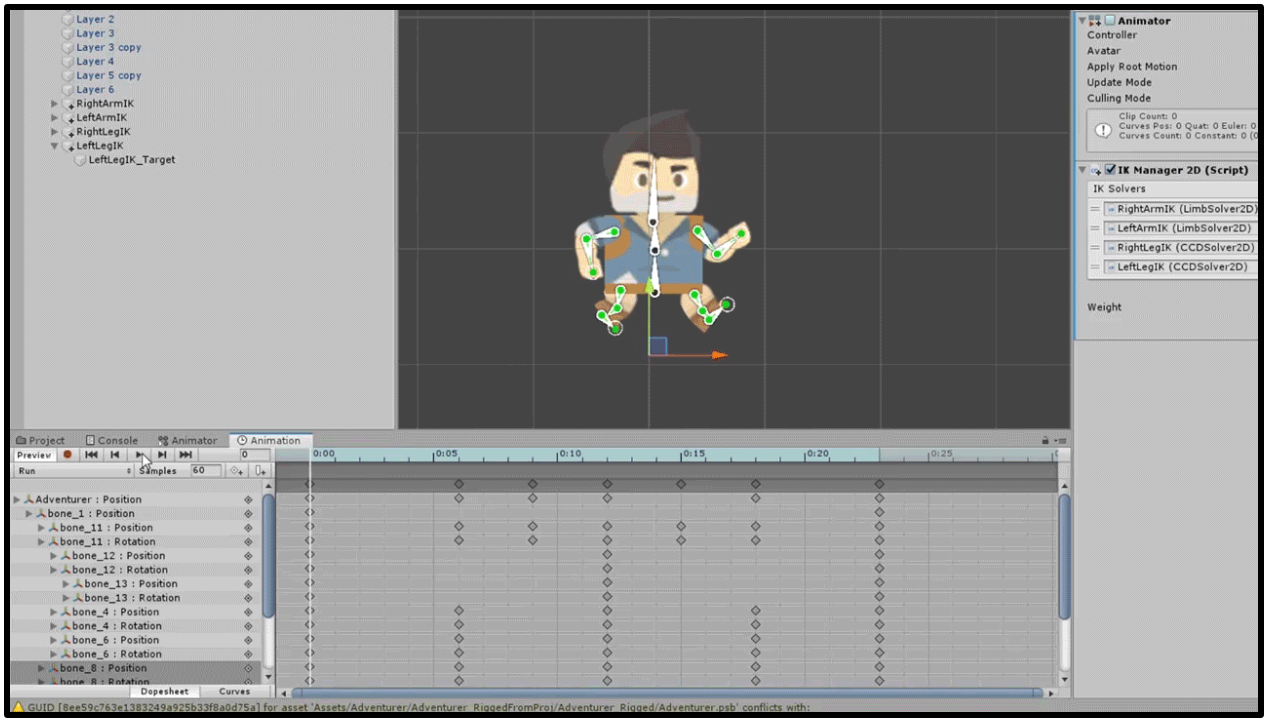
Second, we need to add a pose unique to this run cycle called a "Contact Pose" where we can visibly see the character make contact with the ground. This pose occurs a few frames after the dynamic pose and a few frames before the passing pose. It is simply the forward leg making early contact with the ground.



We need one of these before every passing pose. Also, while we're here, let's add a little bit more toe movement after a passing each passing pose to make it look like the character has pushed off from the ground.



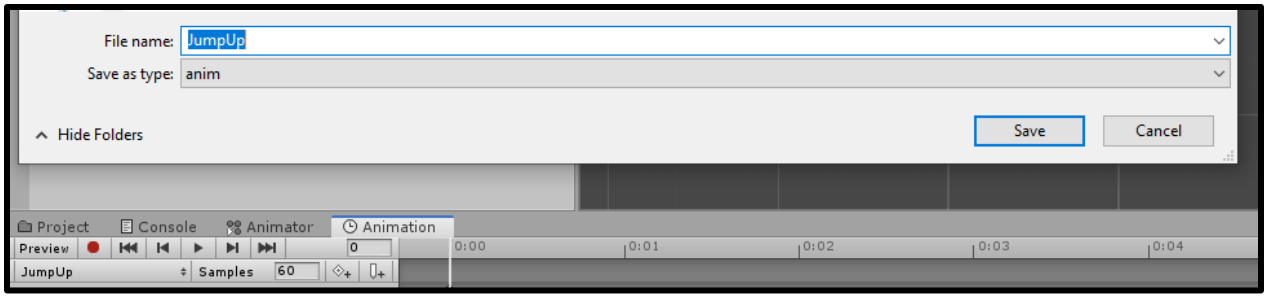
The run animation is probably the most fiddly so be patient and try and get your animation to look as realistic as possible. Go through and try to correct any foot sliding as well.



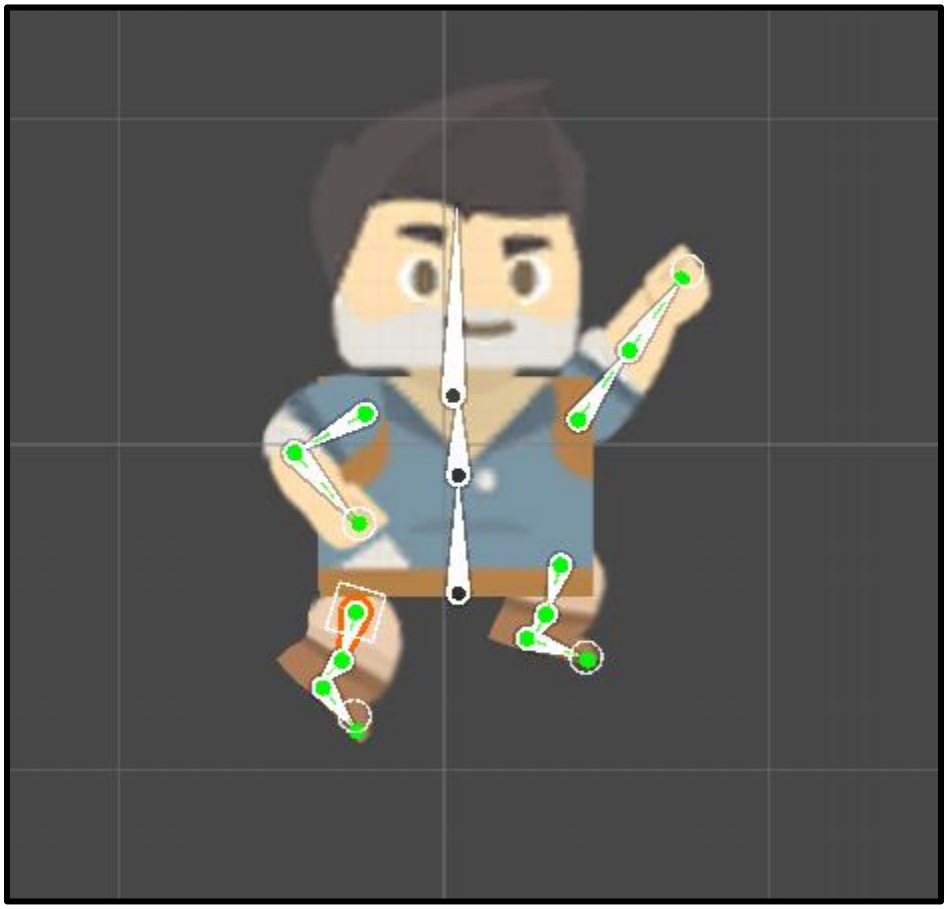
The Jump Animations

A 2D character wouldn't be complete without a jump mechanic. But, animating a jump is completely different than animating a run or walk cycle. In those cases, we needed only one animation clip. In the case of a jump animation, we actually need three. Each animation is just a single frame of a single pose. One animation clip is the "jump up" pose where the character has leaped off the ground. The second animation clip is the "mid-air" pose where the character stops moving upward and gravity begins to pull the character back to the ground. The final animation is the "landed" pose where the character has impacted the ground. The reason we do the jump mechanic this way is so that we can have more control over the power of the jump. If we animated a jump mechanic, it locks us into using that exact jump height. What if we wanted to have jump power-ups? We would have to animate a whole new action with an accentuated jump height. With this method, however, we can have the character jump up with a force of 50 or 500 and it makes no difference, all the animations play in their proper place. The Unity Animator does all the hard work for us in this case.

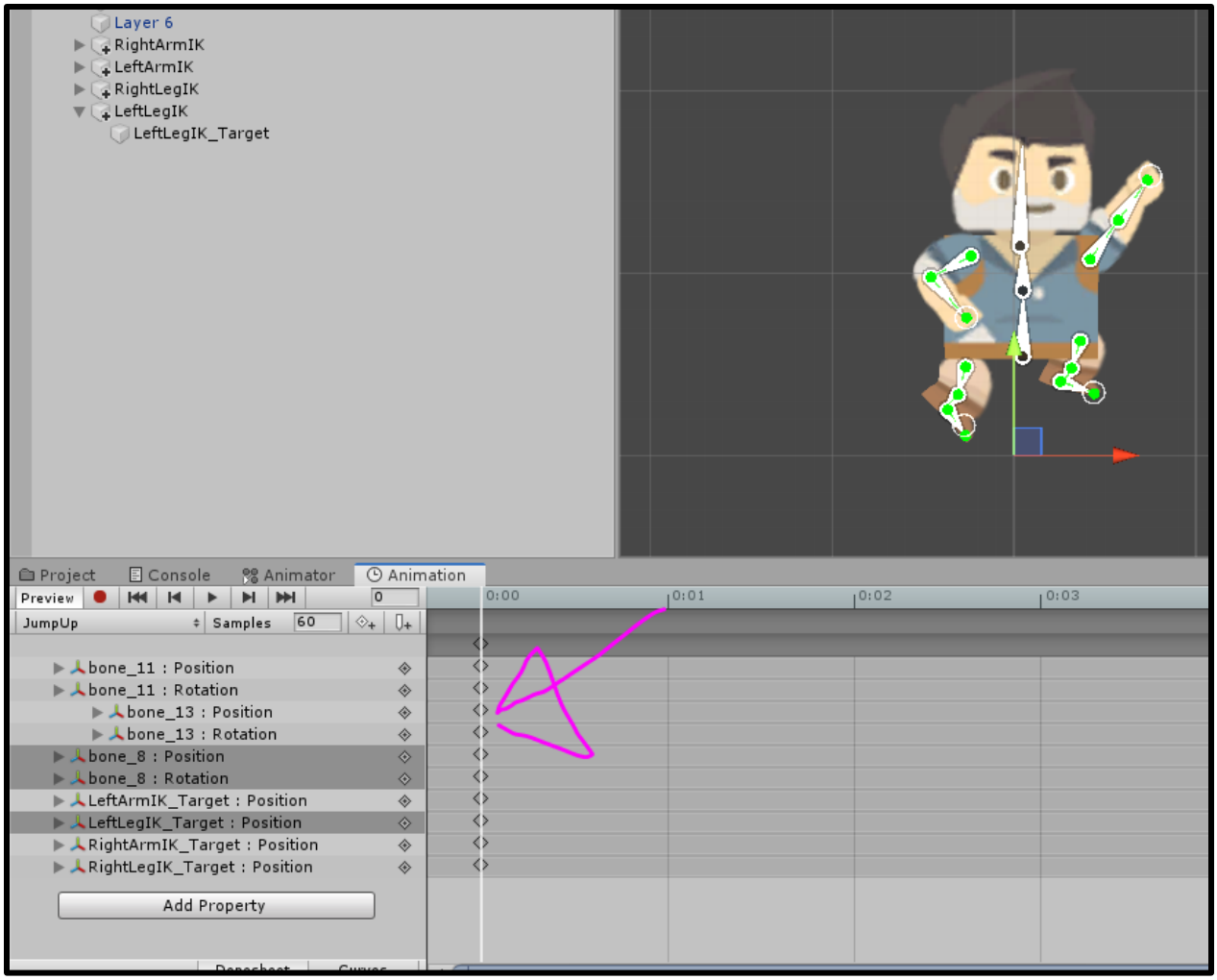
Create a new animation clip called "JumpUp."



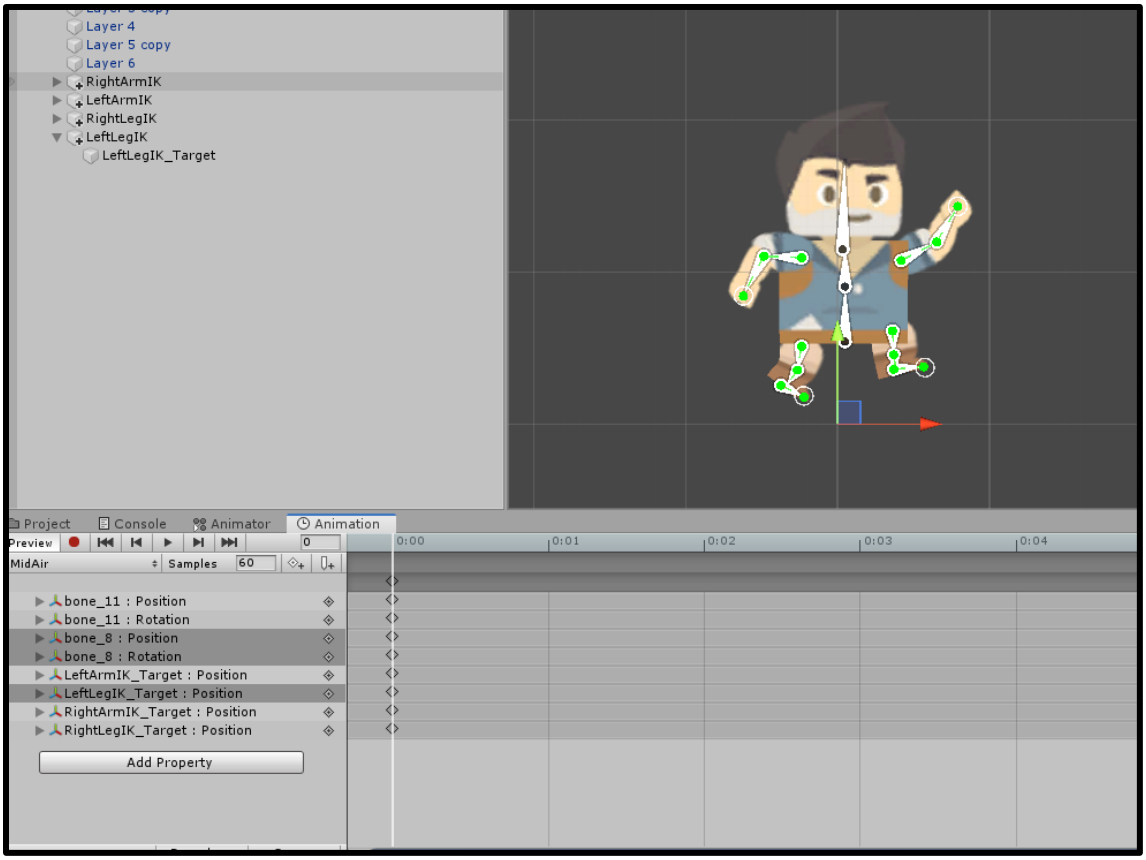
Hit record and simply drag the limbs into a pose that looks like this:



And that's it for that pose! Just make sure it only exists on one single frame. There shouldn't be more than one frame in this animation clip.

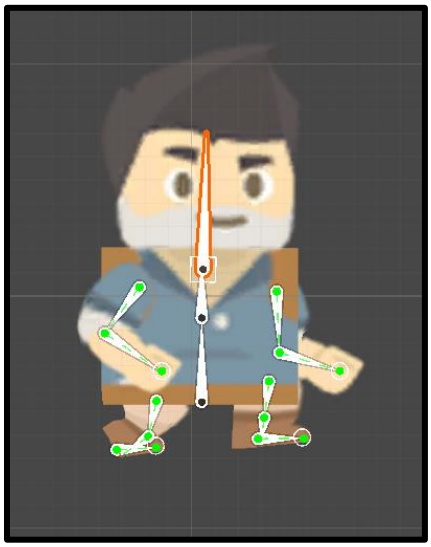


Next, create a new animation clip called "MidAir" and position the limbs like this:



The character has stopped accelerating upwards and is now falling back to the ground. This pose doesn't look very natural but it really adds a realistic flair to your jump mechanic.

Finally, create a new animation called "Landed" and make it look something like this:



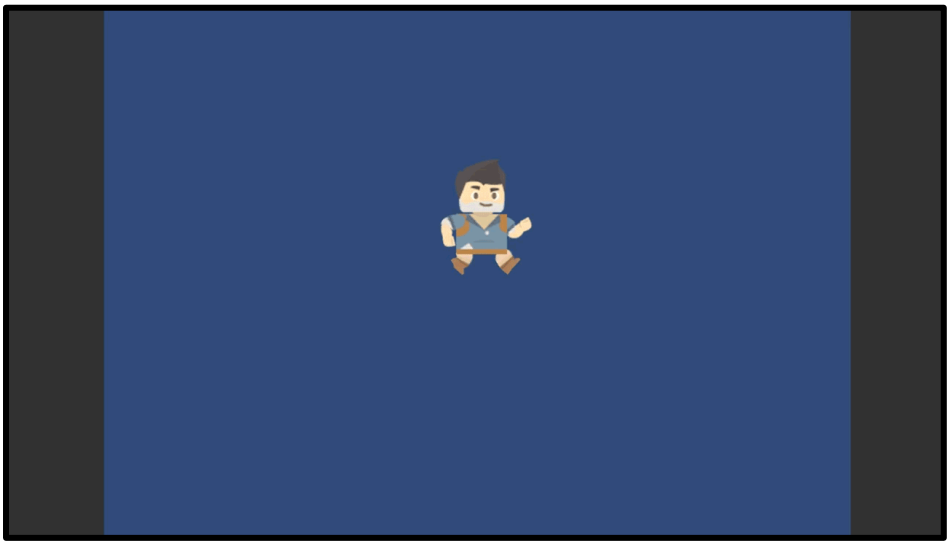
Here, the character has fully impacted the ground after the jump. We now have three animations which will constitute the core of our jump mechanic. All of these poses will be used by the Animator. In the Animator, we will create interactions which will transition and play these animations automatically. The topic of the Unity Animator will be treated in a separate tutorial.

A brief word on Root Motion

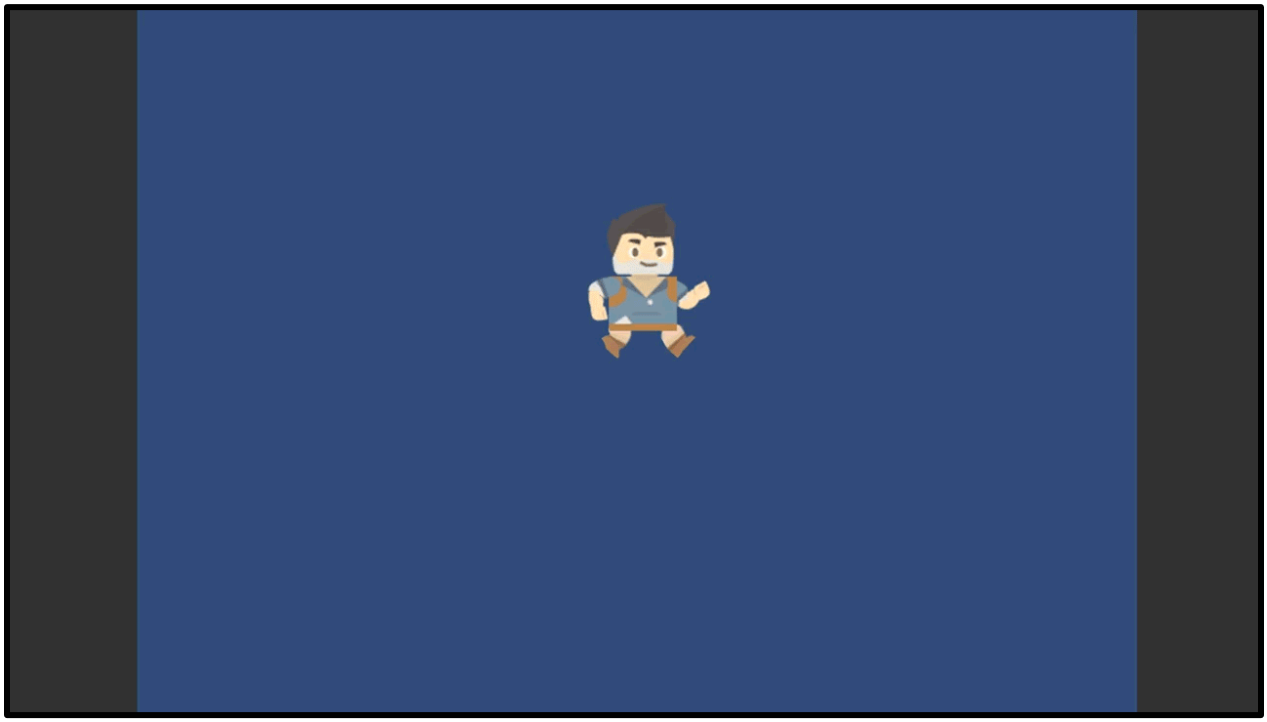
When we animated our character to move forward in the run and walk actions, you'll notice that we built the forward motion into our animation. This is one of two methods of moving the character forward. One of the methods is to animate the character running or walking in place and then script the character's forward motion. This method is good if you, as a developer, prefer to have your forward motion controlled by a few lines of code and if you just like having that kind of control when you are scripting your character. However, we have chosen the other method and that is to animate the character's forward motion. The main difference, in this case, is that we, as animators, now have control over the forward motion. The result of this control is that we can make the animation *much* more realistic. We can add realistic bobbing motion to the character's run and walk cycle which further enhances the illusion and we can prevent the foot sliding that often takes place if the forward motion were scripted and not animated.

This method has one main drawback, however, and that is that our forward motion is not additive. Whenever we hit play on our animation, the character runs forward and then jolts back to its first position. This is why we need a special Unity Animator function called "Root Motion." This takes our forward motion and applies it to the character in an additive way. The forward motion will be applied relative to the character's position. This allows us to create a realistic run and walk animation without having the constraints of a normal animation.

With Root Motion:



Without Root Motion:



Conclusion

This tutorial was mainly focused on animating a 2D character and it dove deeper into the actual structure of an animation. Knowing these things will not only allow you to animate 2D characters but also 3D characters as the principles are the same for each medium. Knowing how to create a convincing run or walk cycle is the first step to creating the illusion of life.

Using the Unity Animator for 2D Characters

Introduction

What is it that an animator does? What summary can we give that would accurately describe the operations an animator performs? Is an animator an artist illusionist? Is he a virtual puppet master? What about video game animators? Are they much different from a pen-and-pencil animator?

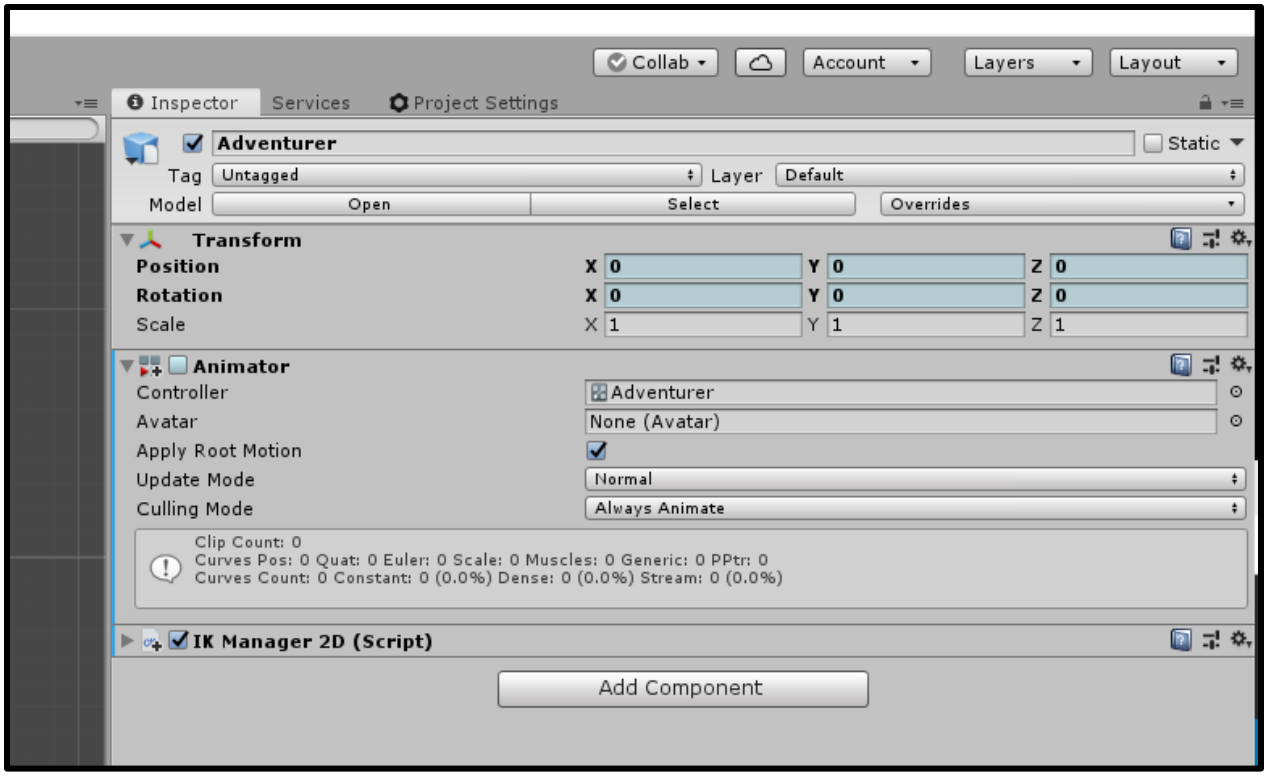
I would say that when it comes to video games, animators are both an illusionist and a puppet master. An illusionist that crafts a realistic movement, and a puppet master that orchestrates this movement. In this tutorial, we will take an animated character and be its puppet master. We will take the animated motions of this character and dictate to it when it will run, jump, or walk. We will create a system of machines that will intelligently transition between actions and take keyboard input from the user. At the end of this tutorial, we will have a complete 2D character that will behave in such a way you would almost expect it to declare, "There are no strings on me!"

Requirements

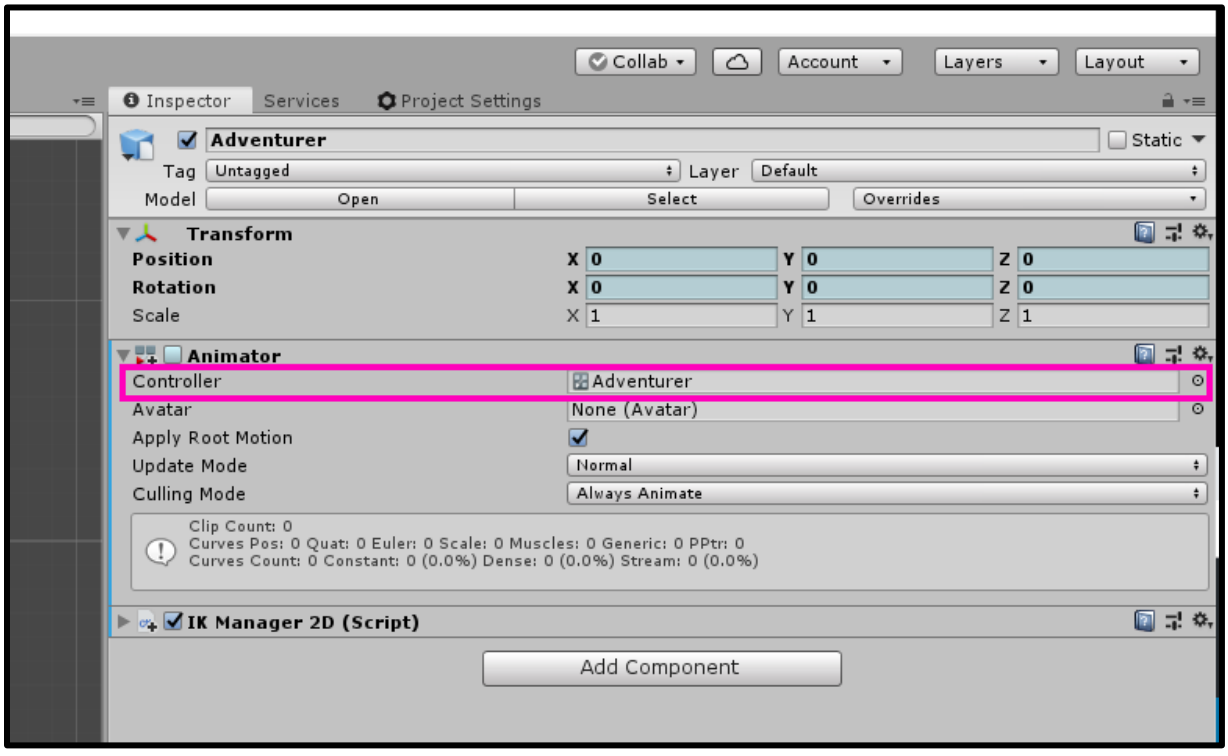
This tutorial uses a 2D sprite that was rigged in this tutorial (Rigging a 2D Character in Unity) and animated in this tutorial (Animating a 2D Character in Unity). If you are interested in sprite rigging or 2D animation, have a look at those two tutorials. Both are an in-depth view of rigging and animating a character. Or, if you like, you could download the project files from the animation tutorial and jump right into this one. Here is a link to the project files that will get you started in this tutorial here. This project also requires the "2D Animation," "2D Sprite Skinning," and "2D Inverse Kinematics" packages so make sure you have those downloaded and imported into your project before continuing (for instructions on downloading and importing the packages, refer to the first part of this tutorial: Rigging a 2D Character in Unity). Also, some familiarity with C# would be quite helpful. If, however, you aren't confident in your C# coding skills, follow along with this tutorial anyway. You will still be able to complete this project even if you have limited knowledge of C#.

The Animator Component

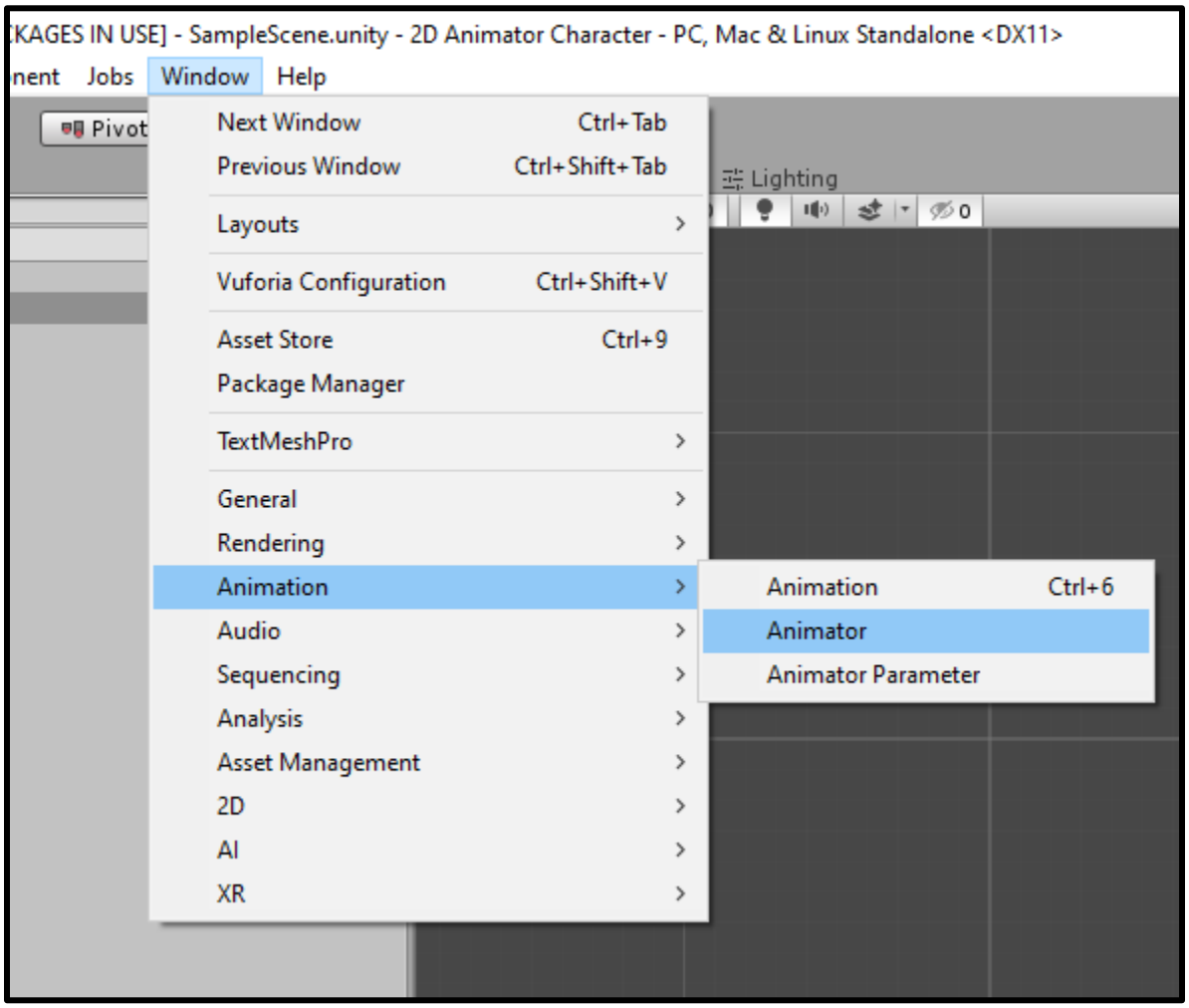
Let's start by examining the Animator component. Click on the Adventurer character and have a look at the Animator component.

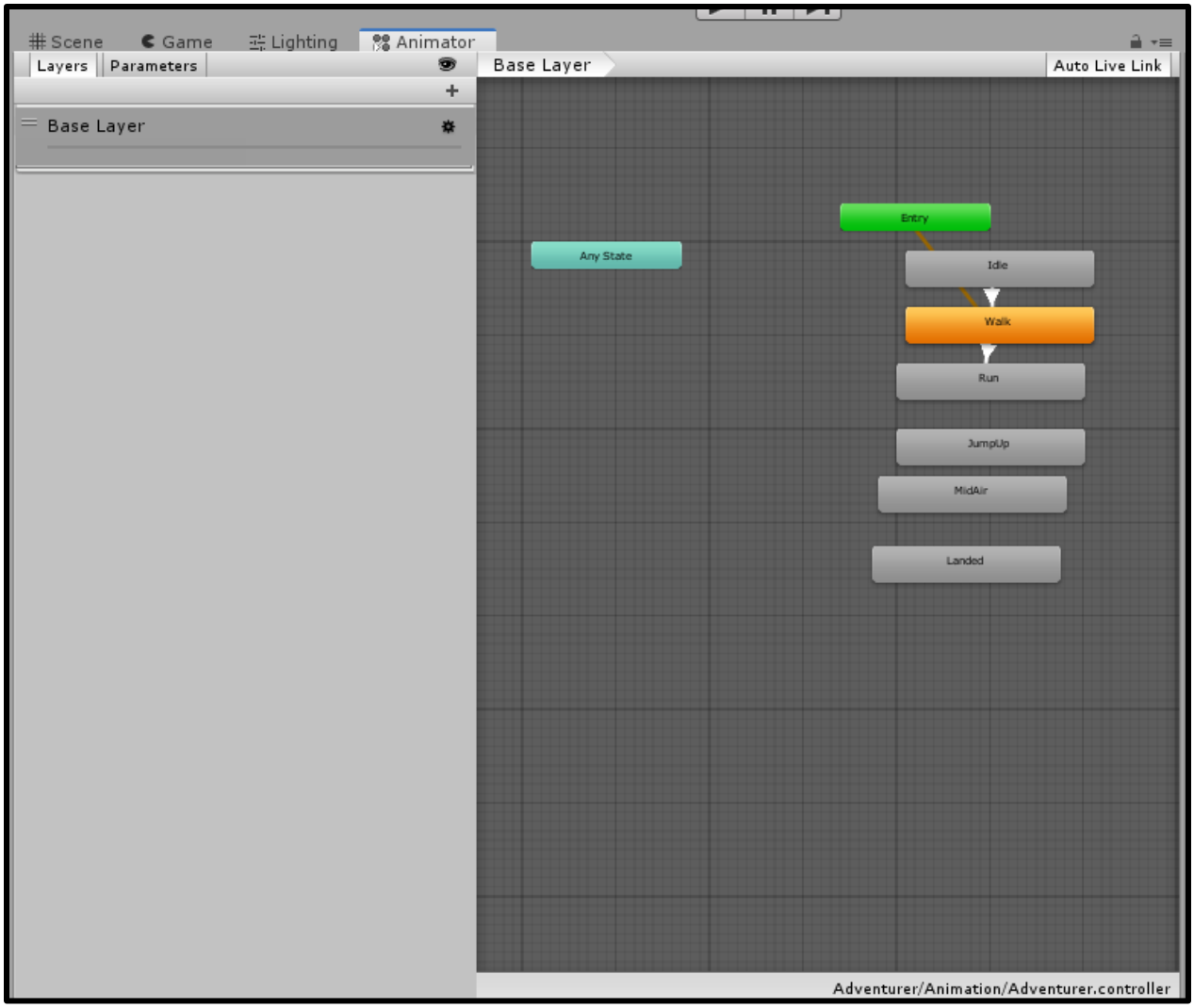


The "Controller" field is where we assign an animator controller.

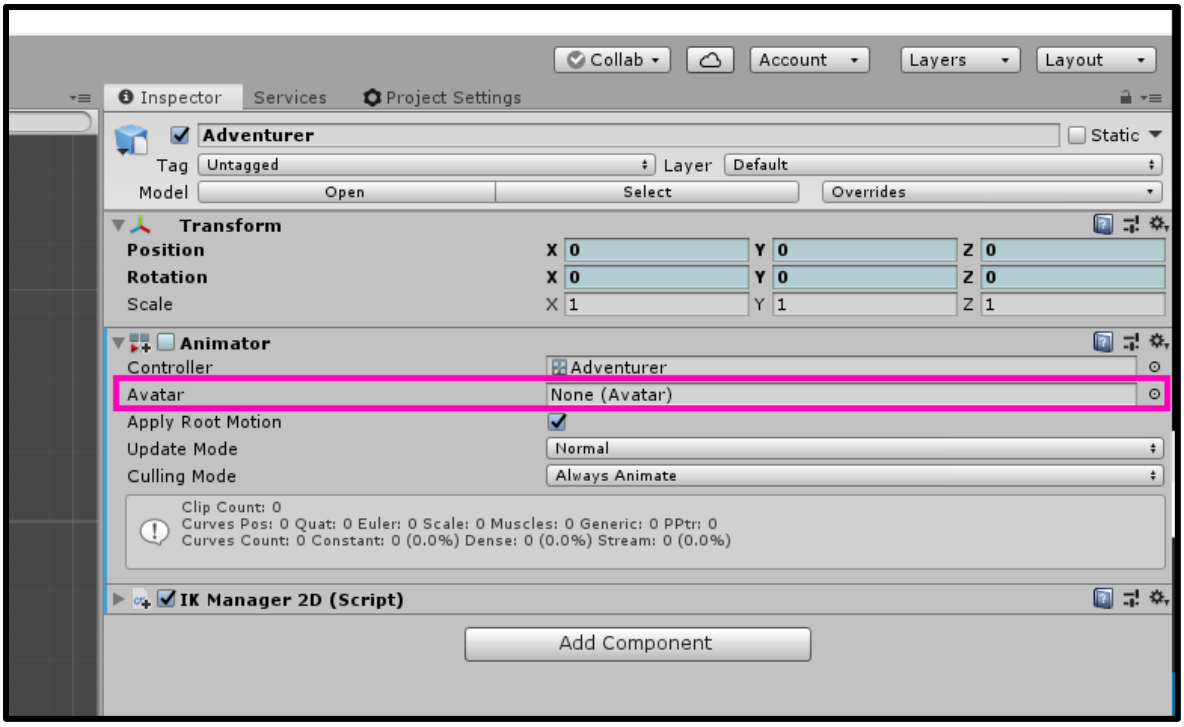


An "Animator Controller" allows us to create interactions between animations. It is where all the running, jumping, and walking actions come together into one unified site. We already have one assigned to this field and you can go to Window -> Animation -> Animator to see what an Animator Controller looks like.

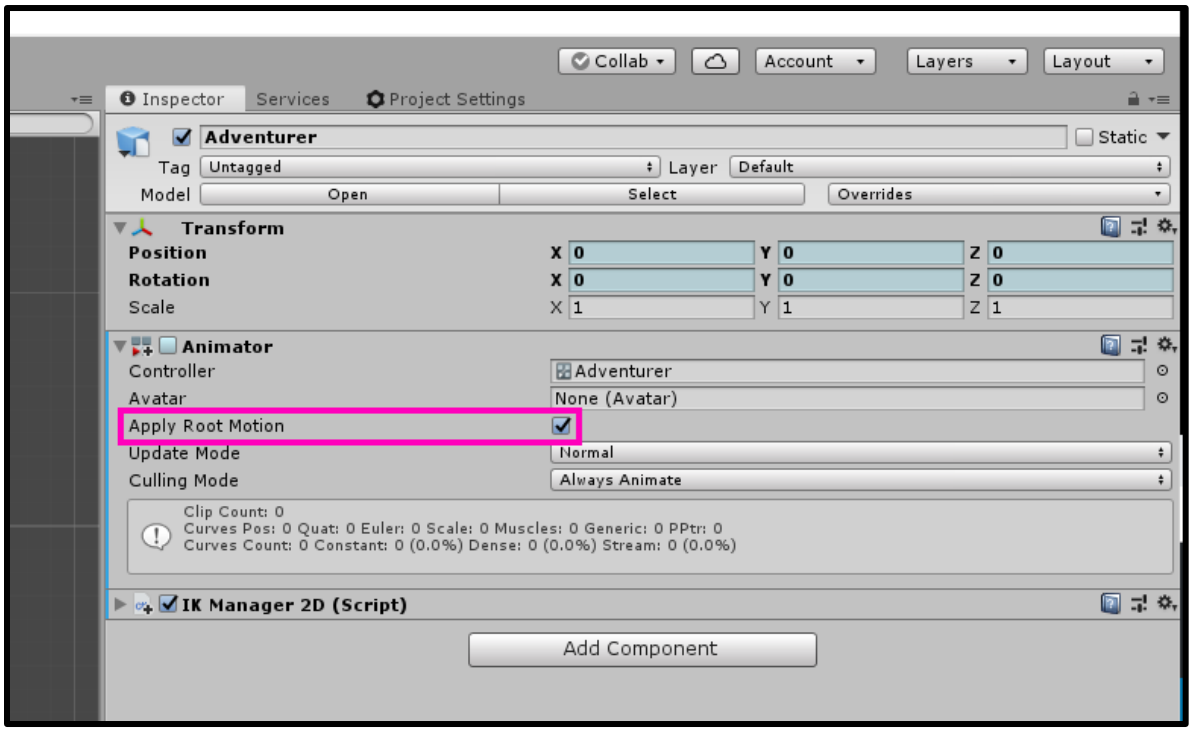




As you can see, all of our animations are stored here ready for us to do structure interactions between them. For now, however, let's go back to the Animator component and continue examining it.

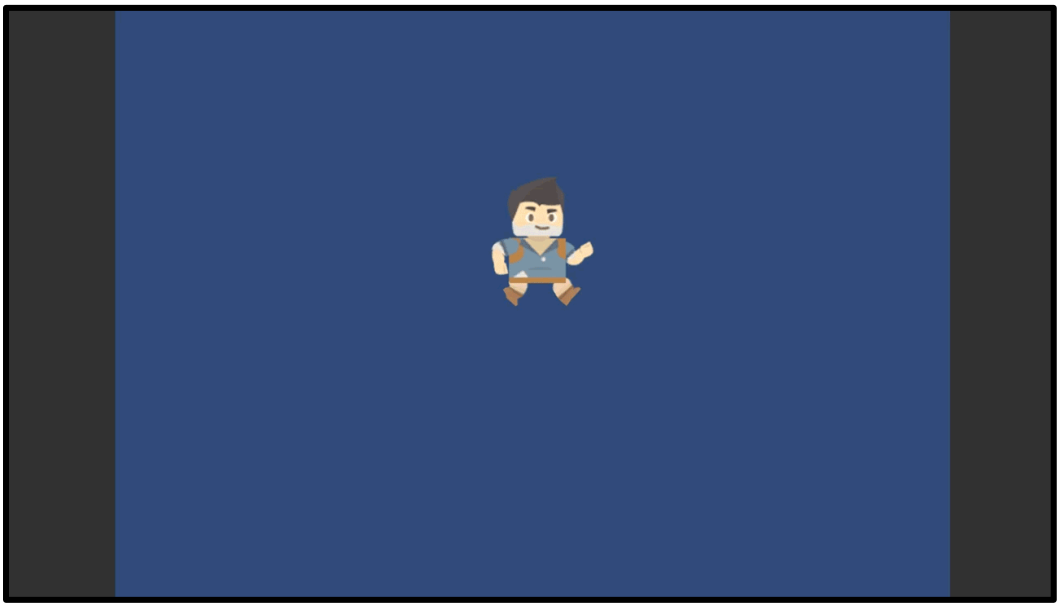


The "Avatar" field is fairly self-explanatory. This is where we would assign the Avatar property. You'll notice this field is empty. This is because an Avatar is primarily for 3D humanoids as opposed to 2D. Since we're operating in 2D, we have no need for an Avatar.

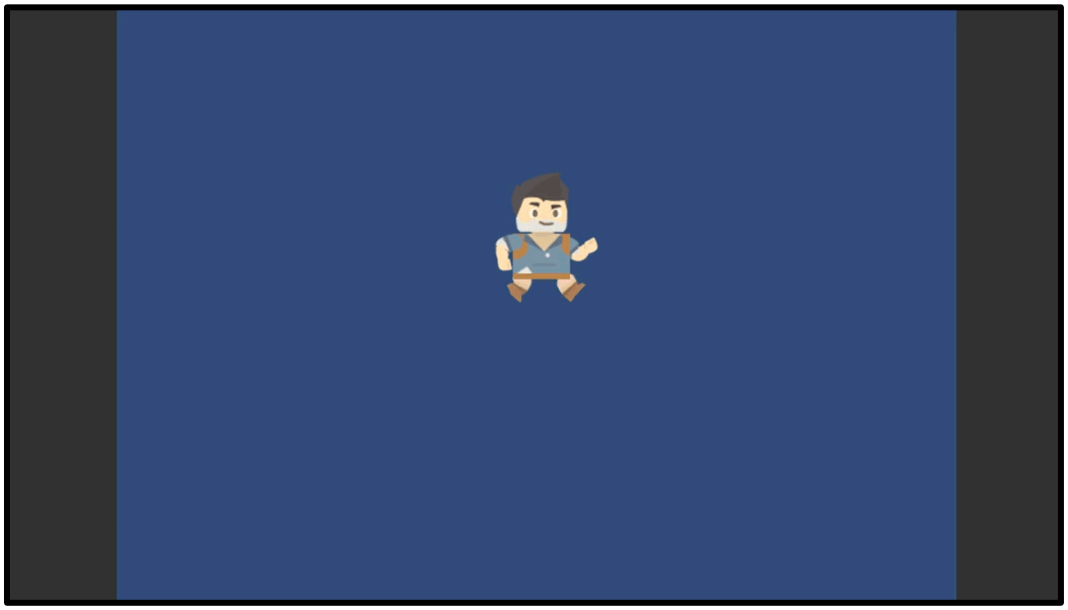


"Apply Root Motion" will determine whether or not our character will move in an additive way. With this enabled, our animation will be what drives the locomotion of our object not the scripting of the game object. This obviously contributes to a more realistic character animation but it requires the animator to actually animate the character moving forward. All of our walk and run animations have the forward motion animated into the clip so we want this enabled.

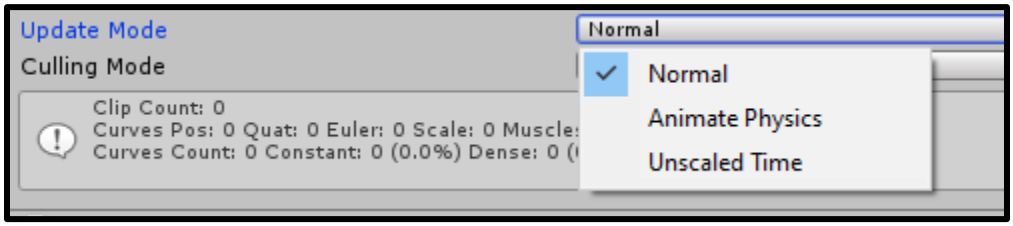
With Root Motion enabled:



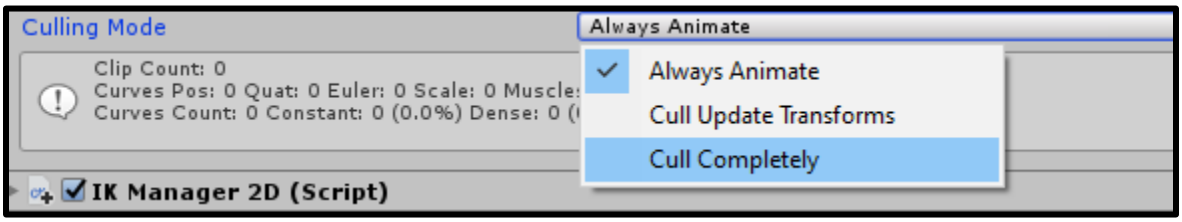
Without Root Motion:



"Update Mode" determines what frame rate the Animator should use.



"Normal" simply uses the frame rate that the Update() method uses. This is the one we're going to be using as it is will match the current frame rate of the game. "Animate Physics" uses the frame rate that the FixedUpdate() uses. It is best for animations with a lot of physics interactions as this operates on a completely different frame rate than "Normal". And "Unscaled Time" simply runs all animations at 100% speed. This is best for UI animations that aren't trying to be realistic and have no physics interactions.

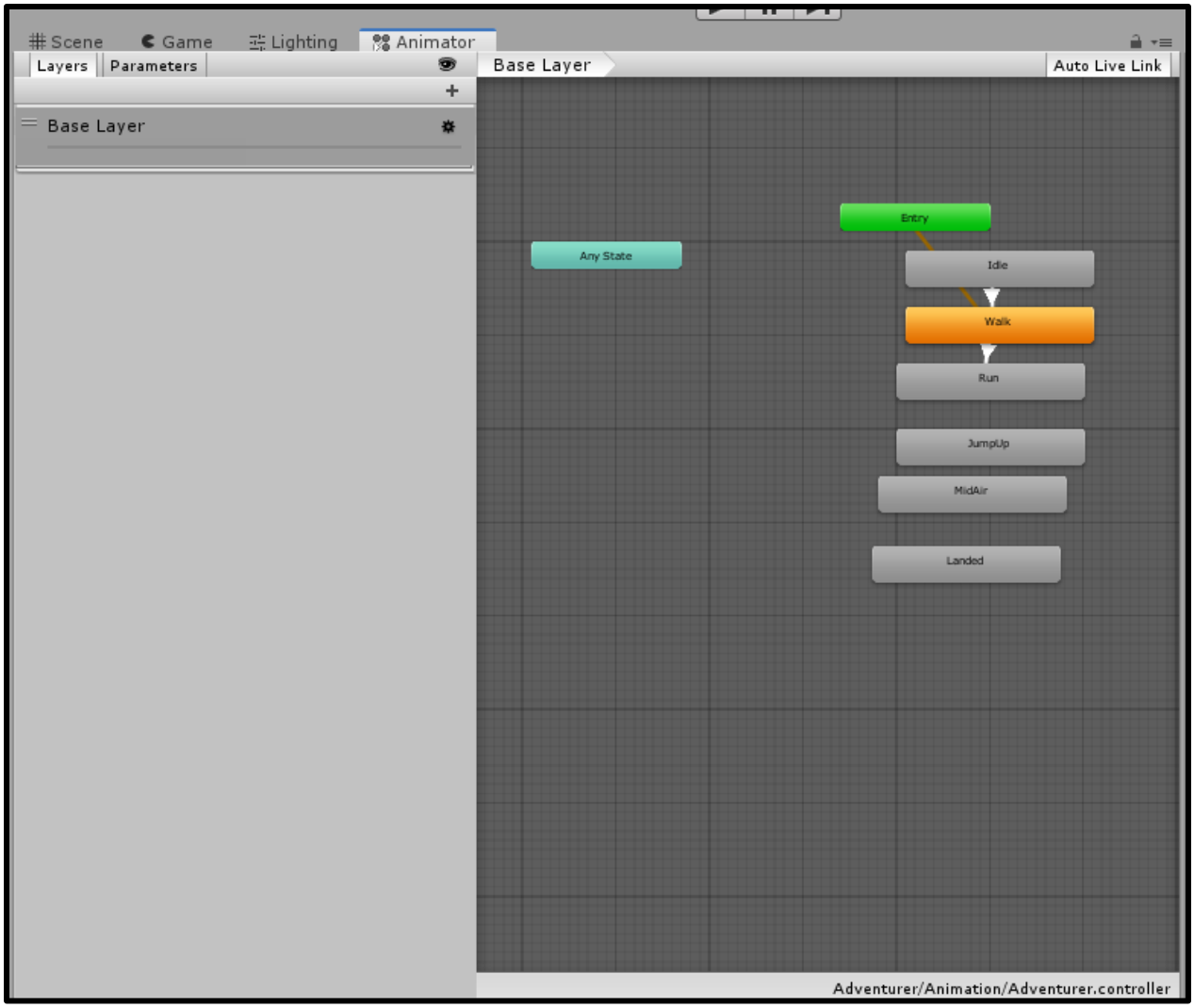


"Culling Mode" determines how the animations will behave when they are outside of the view of the camera. "Always Animate" will keep the animation running. This is obviously the most memory intensive option but for a 2D game, it will not make much difference. "Cull Update Transform" will pause the animation but will continue to update the transform. And "Cull Completely" will disable everything. For the sake of this tutorial, I'm going to set it to Always Animate.

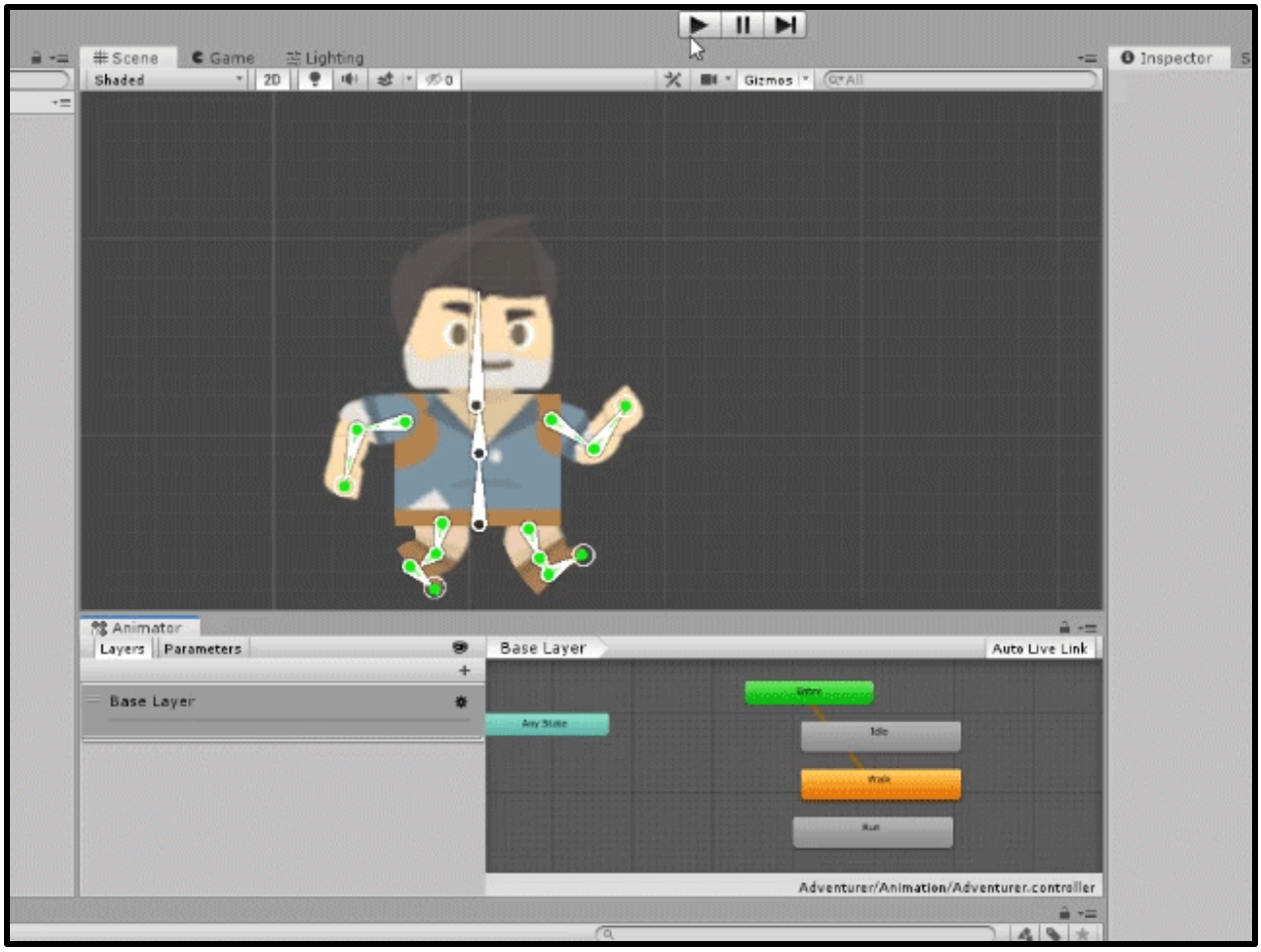
Now that we have some fair knowledge of the Animator Component, let's start working in the Animator Controller!

Creating the Running Mechanic

If you haven't already, go to Window -> Animator and dock the animator tab somewhere in your workspace. I chose to put it directly underneath the scene view. You'll notice that all of our animations show up here as grey rectangles.



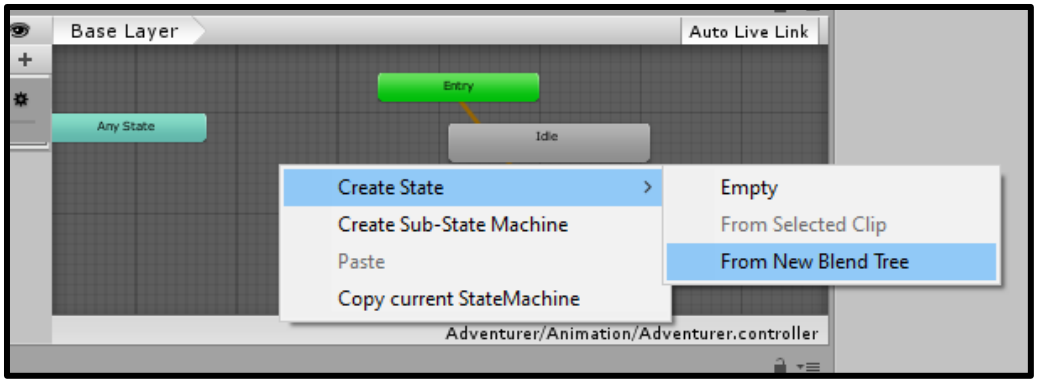
These are called "states" in Unity. You'll notice that one of them is orange. This is known as the "Default State" and it is this is what will run as soon as we play the game.



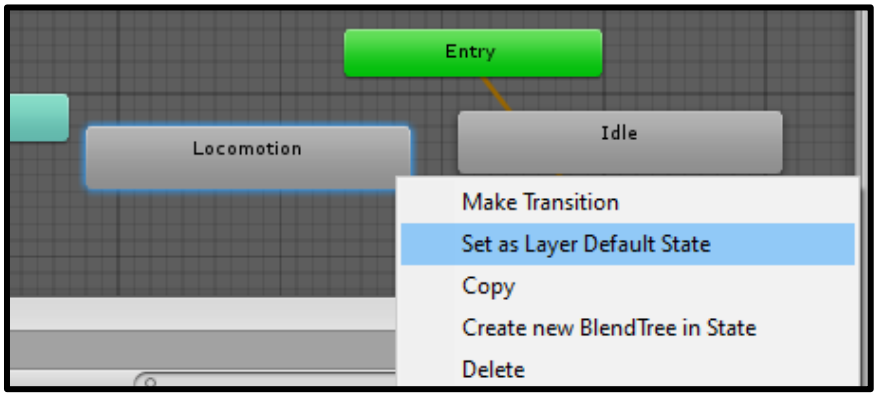
The first action we need to make for our character is the ability to run and walk. So let's think about how we're going to do this. We have two animations, run and walk. We need a way to have both of these played consecutively. When the player presses the left and right arrow keys, the character needs to go from an idle state to a running state without it being a jerky transition from standing to running. We can prevent a jerky transition by having the character blend between the idle, walking, and running animations.

The way we would do this in Unity is by creating what is known as a "Blend Tree." A Blend Tree will take a series of animations and blend between them based on input from a parameter. This explanation can be a little confusing so let's see what it looks like in practice.

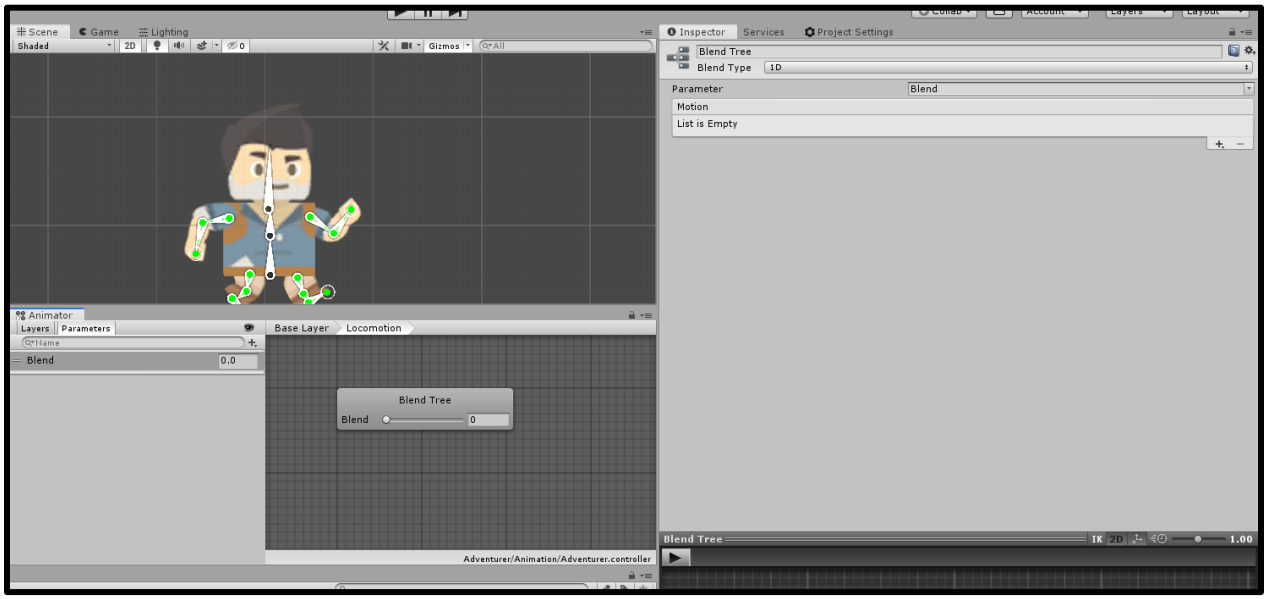
Right-click in empty space on your Animator and select "Create State -> From New Blend Tree."



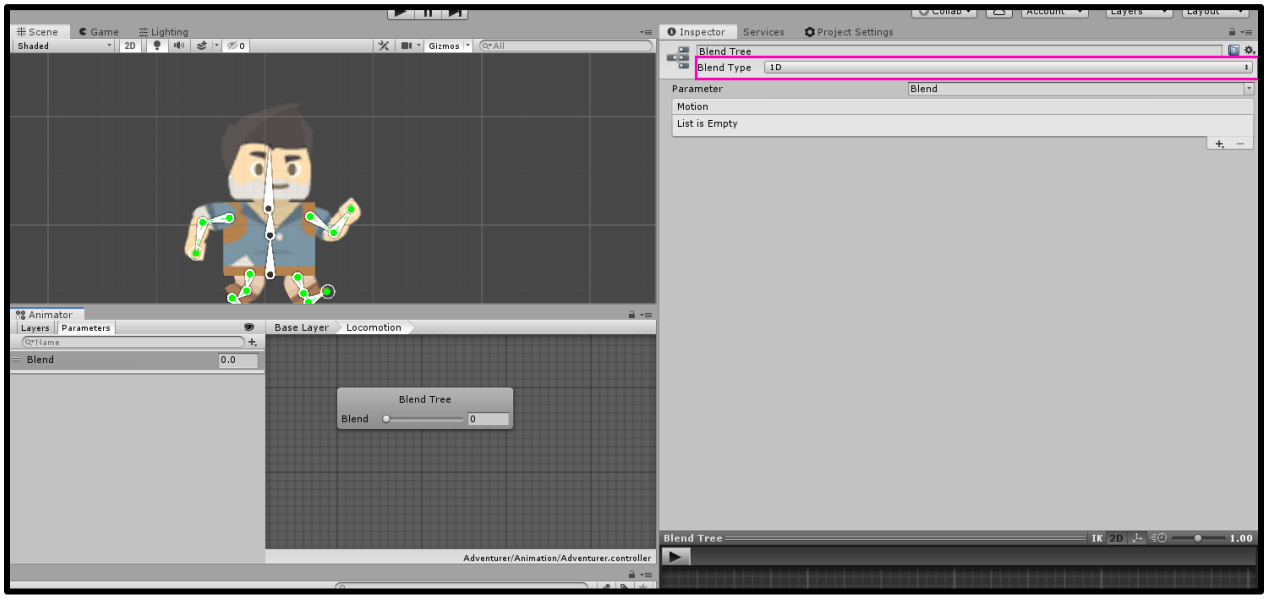
Name it "Locomotion." Right-click on the blend tree and select "Set as Default Layer State."



This means that as soon as we hit play, the blend tree will immediately start playing. Now double click on the blend tree to open it.

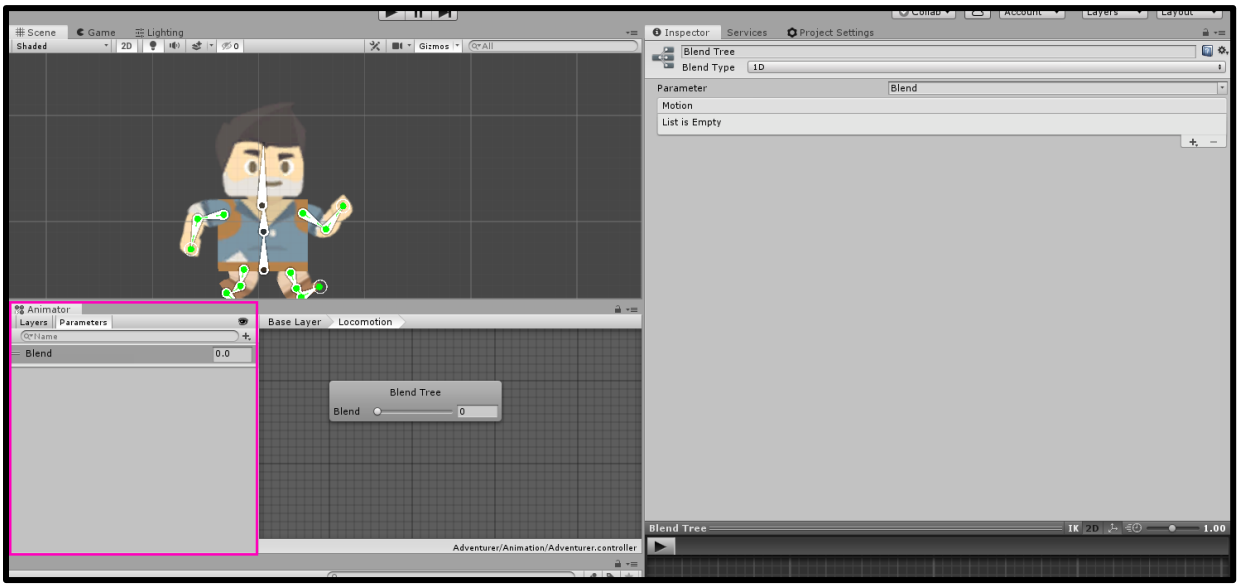


A couple of things we should note. First, the "Blend Type" is set to 1D.



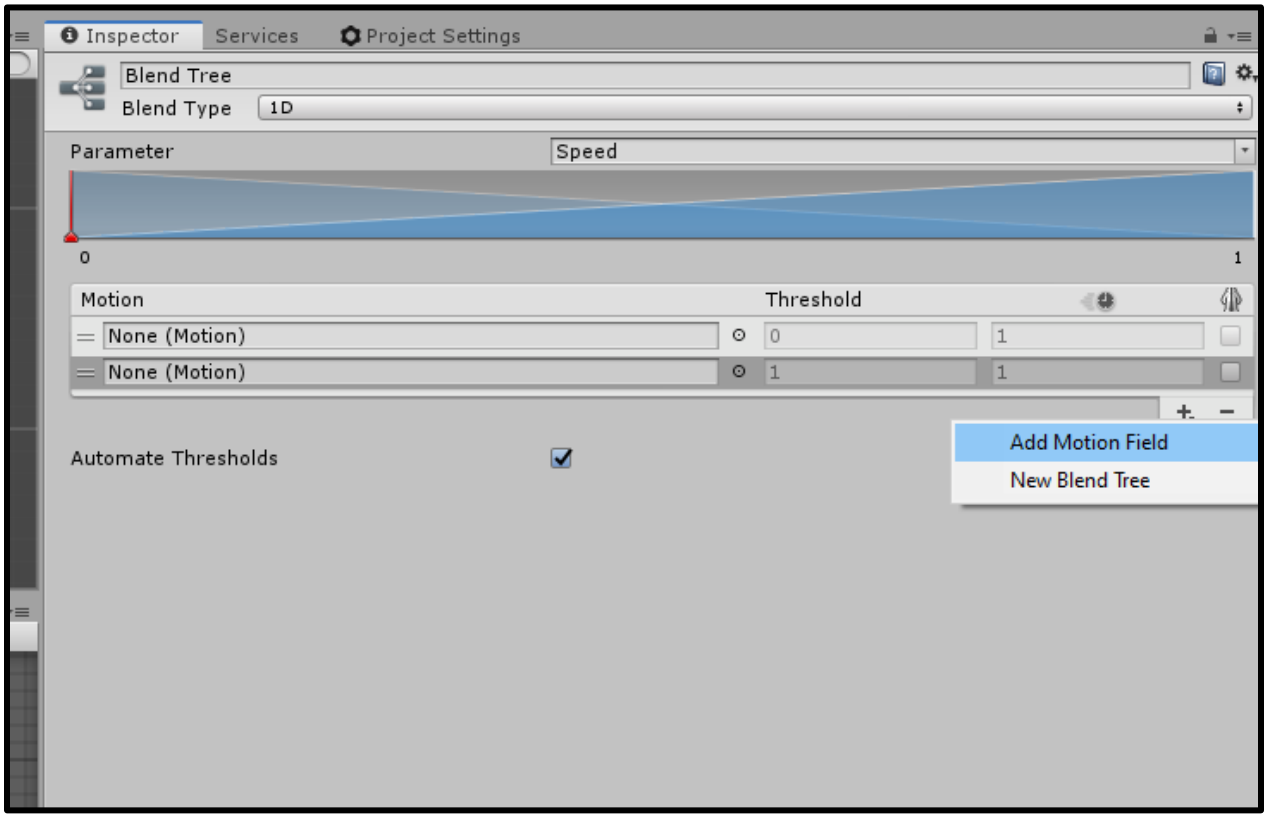
This means the blend tree will only take input from one parameter. The other types ("2D Simple Directional", "2D Freeform Directional", "2D Freeform Cartesian", and "Direct") use more than one. In this tutorial, we're only going to be using 1D but have a look at this Game Dev Academy tutorial (Unity Animator - Comprehensive Guide) if you'd like a more in-depth look at this portion of the blend tree.

Second, if you click over to the "Parameters" panel, you'll see it's created a new parameter called "Blend."

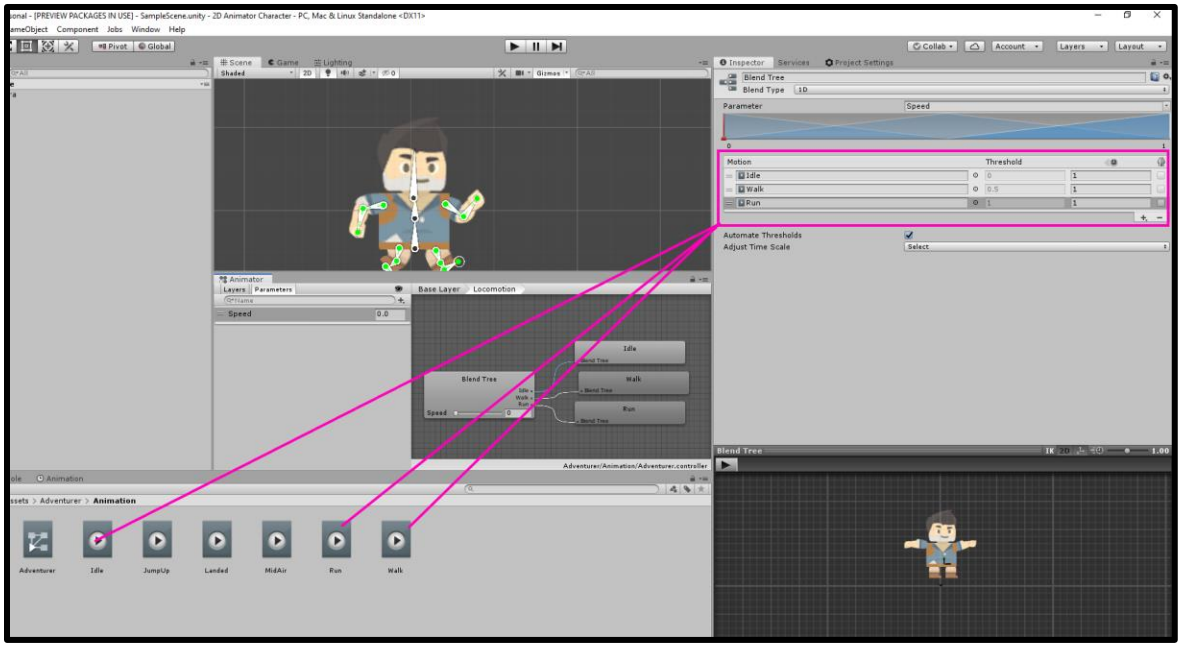


There are different types of parameters ("float", "int", "bool", and "trigger") and they all behave sort of like variables in the Unity animator. They also can be accessed through scripts which we will be doing a little bit later. A blend tree requires that we have a parameter so it has automatically created one for us. This is what will determine when a certain animation will play. Let's name it "Speed" since speed is what will determine whether our character is running or walking.

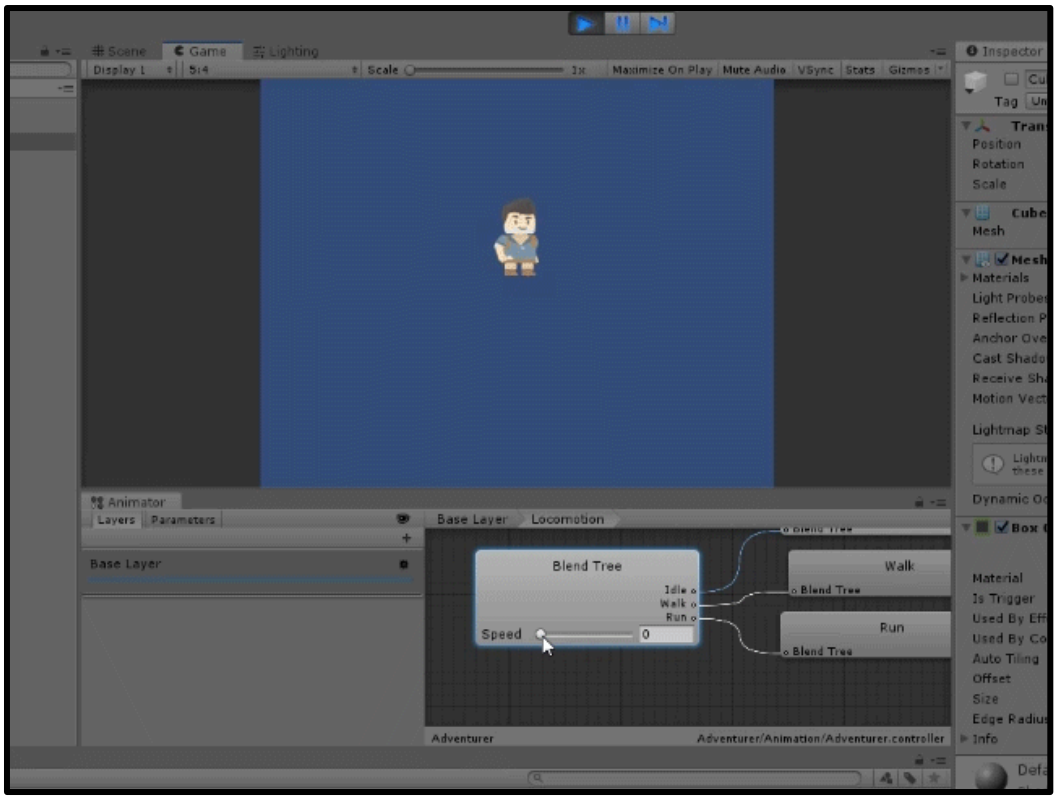
Now, let's construct the body of this blend tree. Hit the plus icon in the inspector and add three new motion fields.



Drag our Idle, Walk, and Run motions into these fields.



Because "Automate Thresholds" is checked, we can use the slide on the lower part of the blend tree square to see when exactly our character will be running. Hit play and drag the slider around.

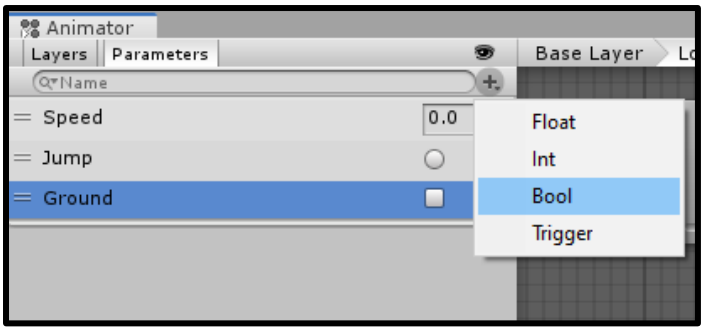


As you can see, our character will run or walk based on the value of the Speed parameter. You can customize the thresholds by unchecking "automate thresholds" and changing the values. We usually leave one as the maximum speed value so don't go past that. I like the current thresholds (idle at 0, walking at 0.5, and running at 1) so I'm just going to leave "automate thresholds" checked.

Creating the Jump mechanic

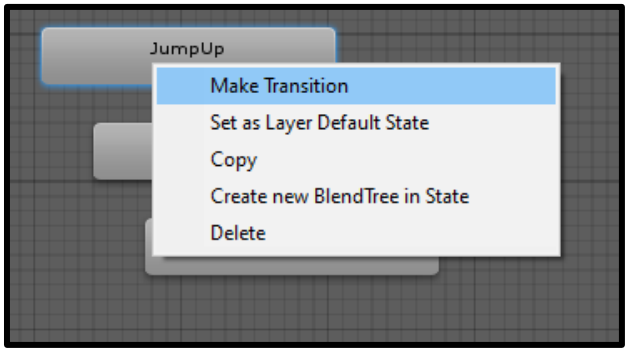
The final mechanic for our character! This has a bit of a different structure than the locomotion blend tree because, if you will remember, we made three poses for our jump mechanic rather than three animations. So this mechanic is going to make use of something called "Transitions." The name is fairly self-explanatory, however, we're going to look at the various ways we can tweak transitions to give us that jumping look.

First, create a new trigger parameter called "Jump" and a boolean parameter called "Grounded."

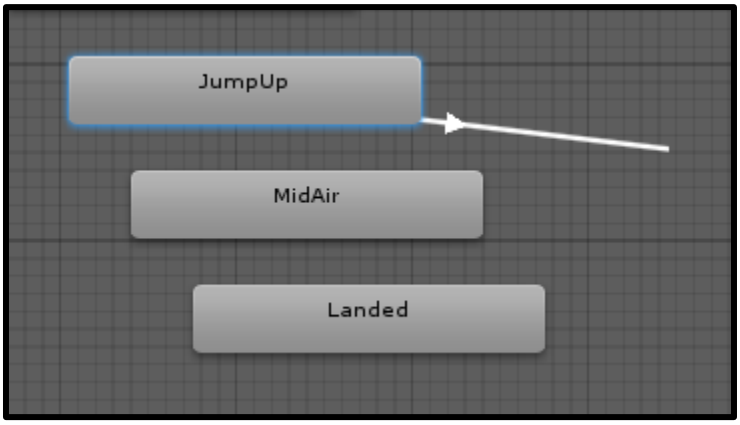


[should be "Grounded" instead of "Ground"]

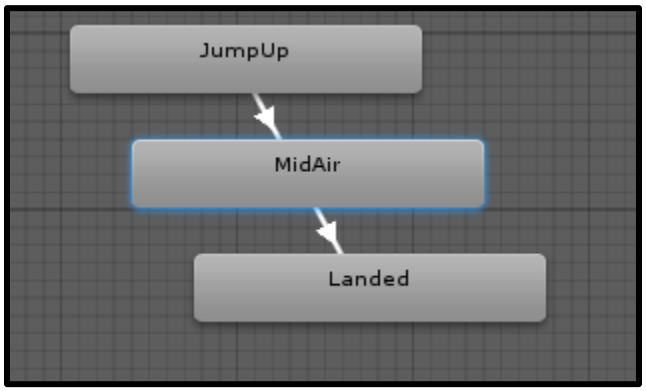
Jump is what we will use to determine if the player has pressed the jump button and Grounded is what we will use to determine if the player is on the ground. Next, locate your three jumping poses ("JumpUp," "MidAir," and "Landed") and place them near each other. Now, right-click on "JumpUp" and select "Make a Transition."



You'll notice a white arrow has come from the JumpUp state and is now tracking to your cursor.



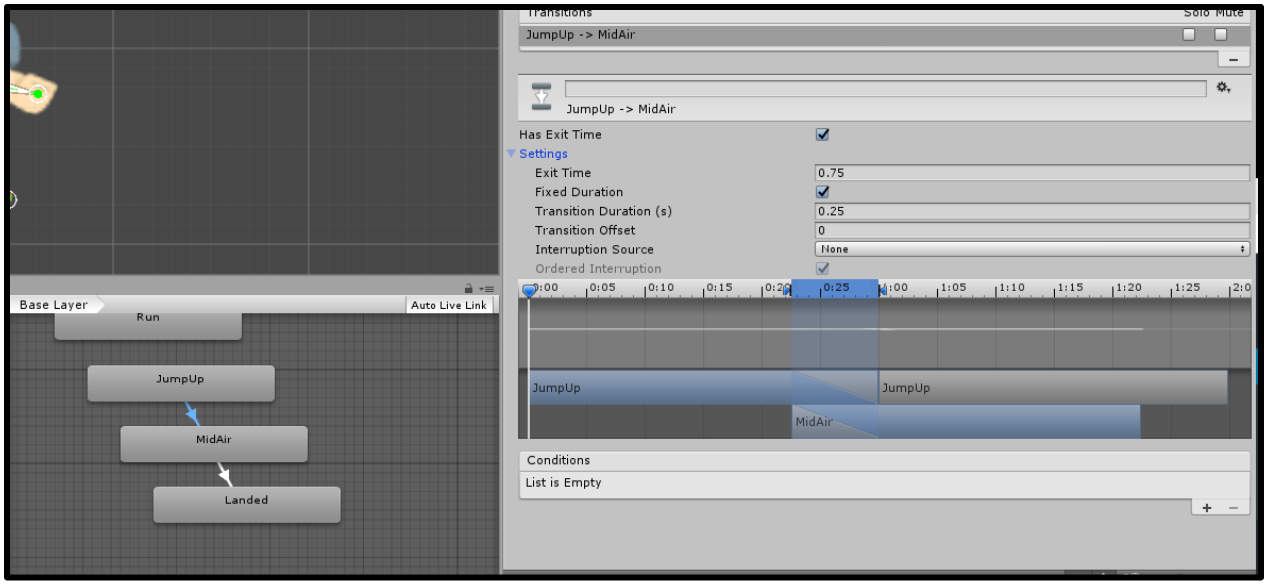
Left-click on the "MidAir" state (I use the term "state," "pose," and "animation" interchangeably in this section). You have now created a transition! Create another transition from MidAir to Landed and that will complete the majority of our mechanic.



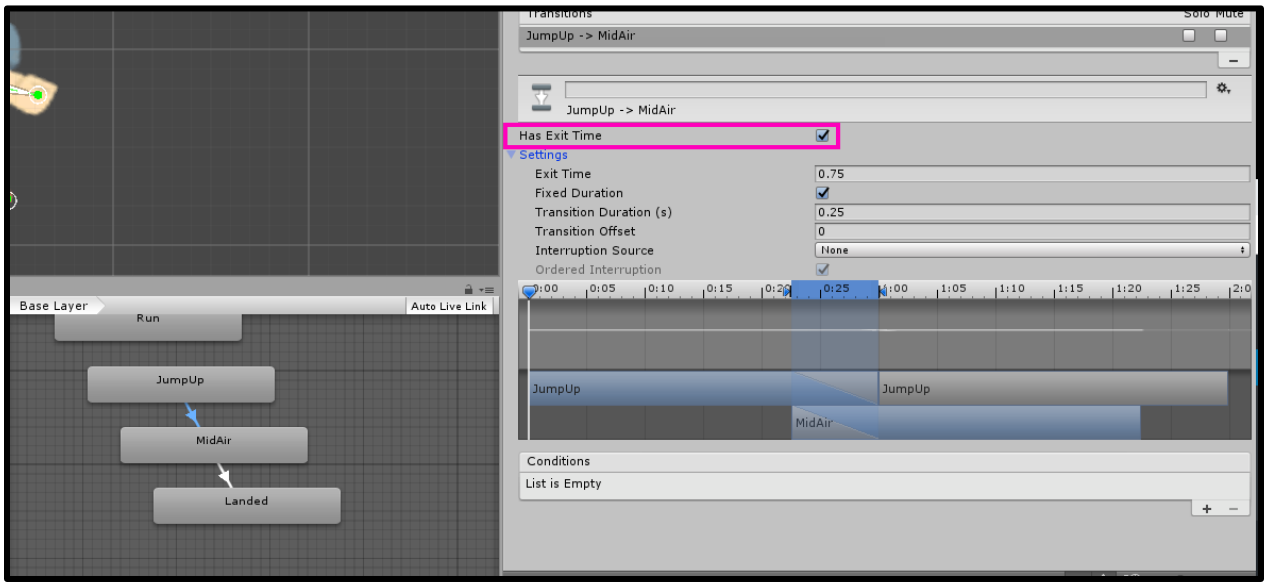
Now, we just need to customize each transition.

Transitions

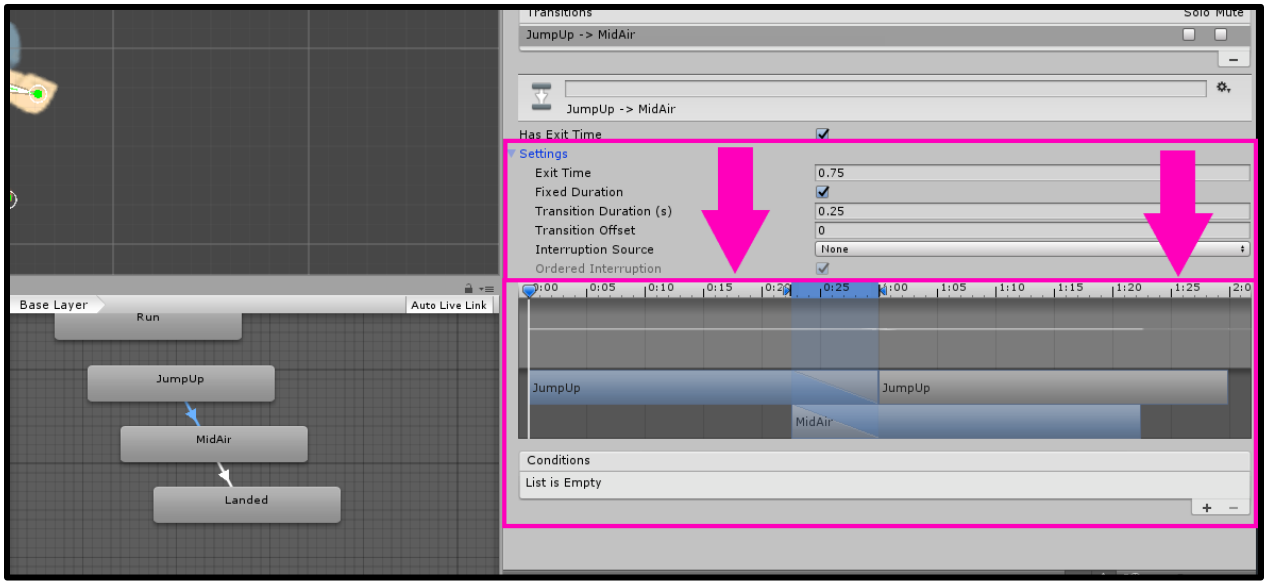
Select the JumpUp to MidAir transition.



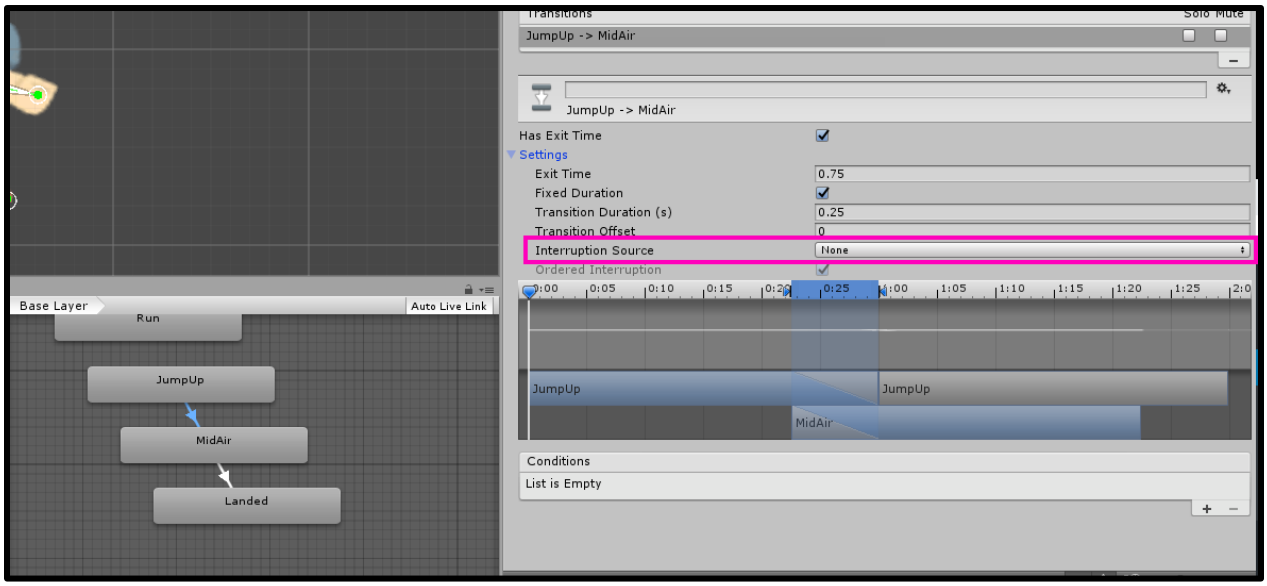
Notice all the settings that appear in the inspector.



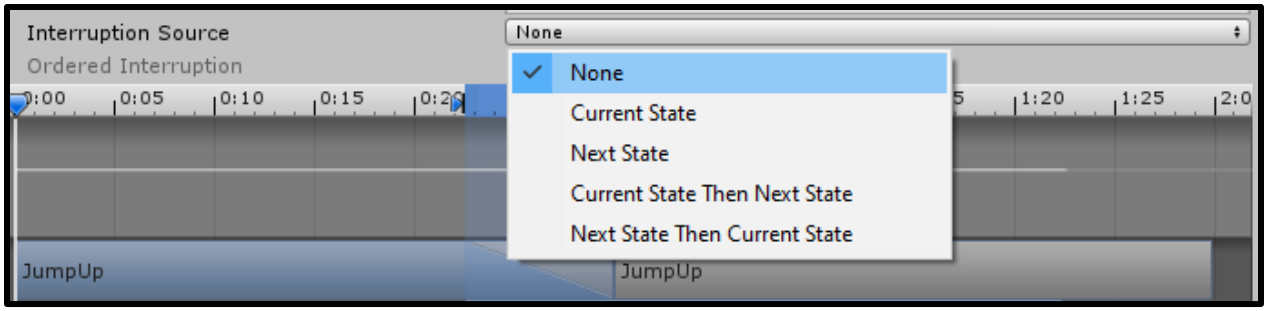
"Has Exit Time" will determine if the animation will play to the end or transition immediately. The other settings under the "settings" drop-down are all fairly self-explanatory and can be configured in the timeline we see in this window.



This timeline will determine how long the transition will be and where in the clip it will transition. The only setting here that cannot be customized through the timeline is the "Interruption Source."

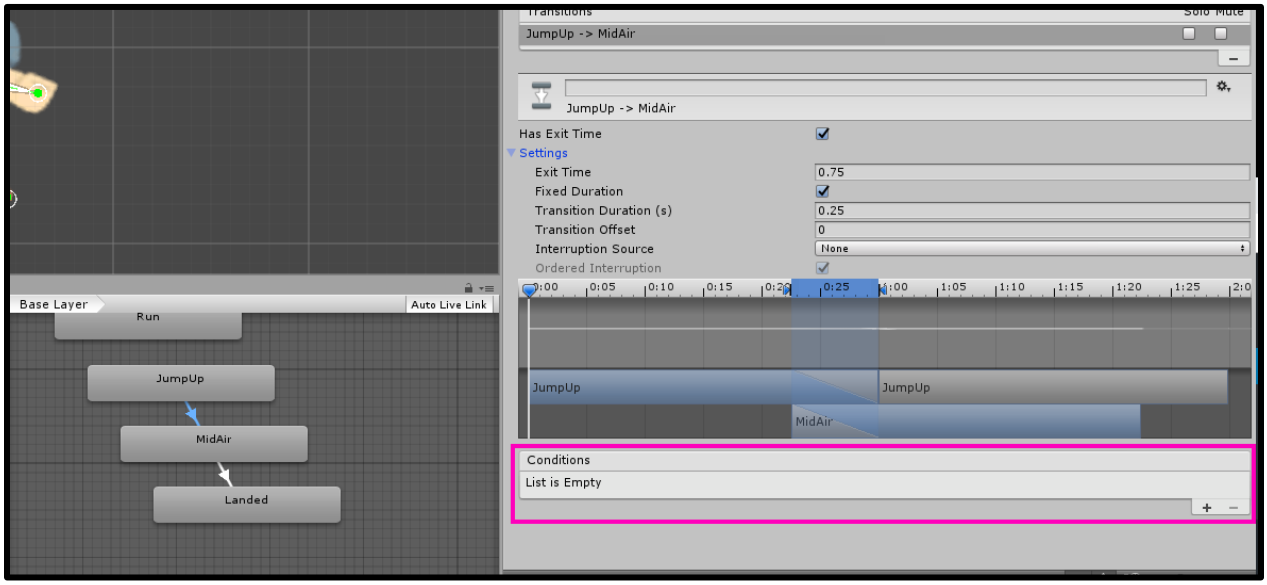


Right now it is set to "None" which means the transition essentially cannot be interrupted (there are exceptions to this which we will look at later). This is the setting we want for the Jump Mechanic but it's worth having a look at the other options in case you will have need of this in your other projects. Click on this setting and notice all the other options.



This determines what transitions will play if an interruption to the current transition is detected. All of the options seem kind of ambiguous but if you stare at them for a little while, they start to explain themselves. For example, if it is set to "Current State Then Next State," when the transition is interrupted, it will play the transitions that are on the current state and then play the transitions on the next state. The converse is true for "Next State Then Current State." If this transition is interrupted, it will immediately play the transitions on the next state and then play the transitions on the current state. This seems really complicated but I hope you can see how useful this could be. Imagine you had a die sequence. If the character is hit by a fatal bullet, you would want the normal hit animation to play and then you would want the die animation to play (likely in that order). And you would want all of this to interrupt whatever transition was currently playing. Interruption Source seems complicated but I hope you see how useful it could be.

The final setting we need to look at is the "Conditions" box.

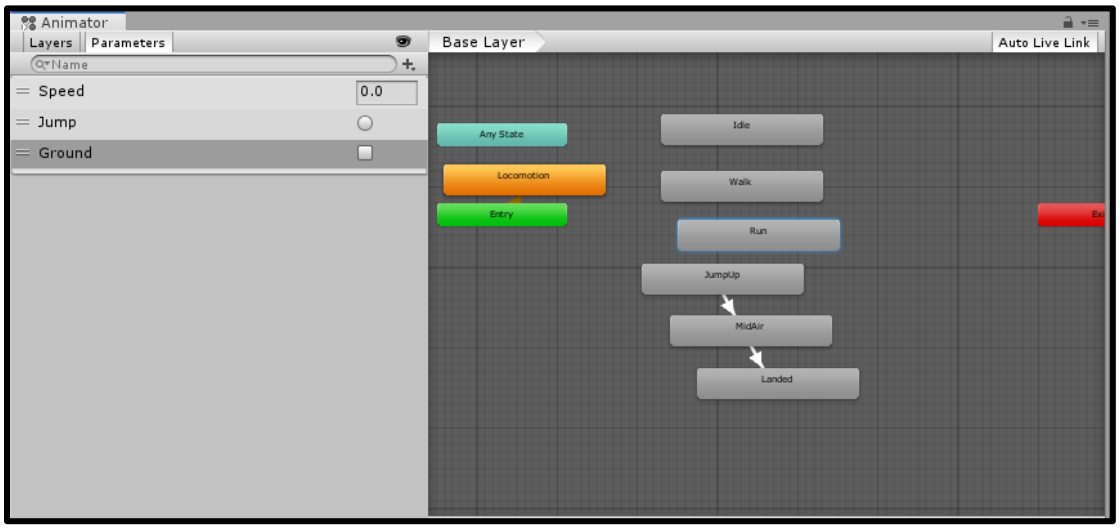


Here we can tell the animator to check certain parameters in a "true-false" scenario. If ***all*** the conditions are met, then the transition will play. If this is empty, it will simply play the transition after the Exit Time has elapsed. We will be using the conditions box to determine if the "Jump"

trigger has been activated and whether the player has landed. At the moment, however, there is some important clean up we must do in order to complete our Jump Mechanic.

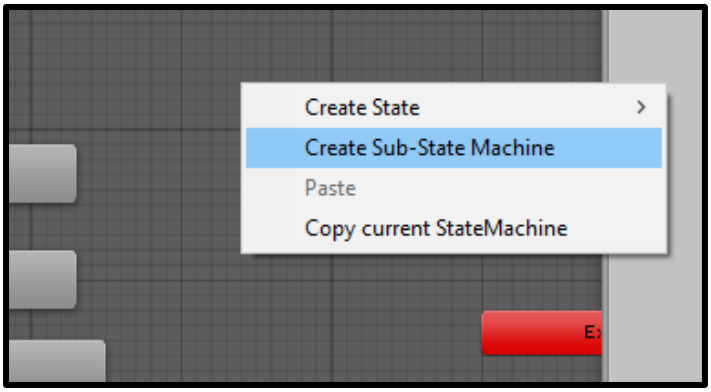
Sub-State Machines

Zoom out and have a complete look at your Animator Controller.

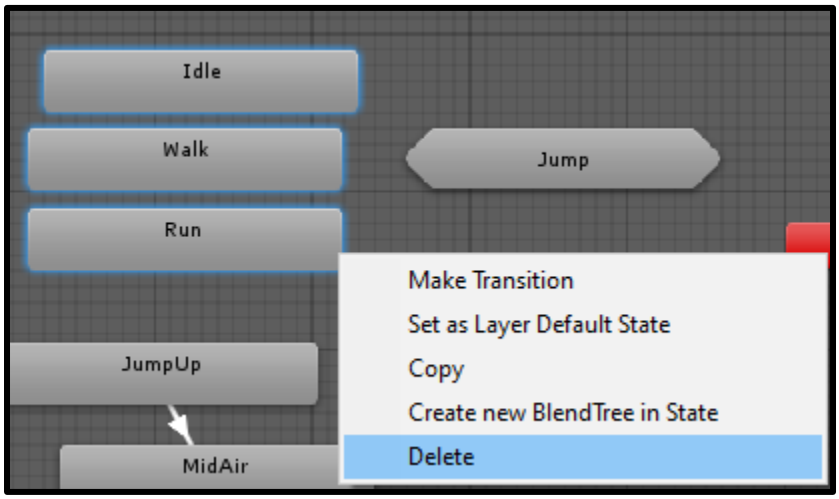


It looks very cluttered. There are half a dozen animation states, transitions going all over the place, and just a general lack of order. This leads to a very inefficient workflow because we have to implement a few more transitions into this to complete the Jump Mechanic. This is where we need to introduce a tool called a "Sub State Machine." A Sub State Machine is essentially an instance of the Animator Controller that exists within the larger Animator Controller. This means we can put all our Jump poses in the Sub State Machine and have them behave in the exact same way they are now. This will tidy our workspace and increase our Controller's efficiency.

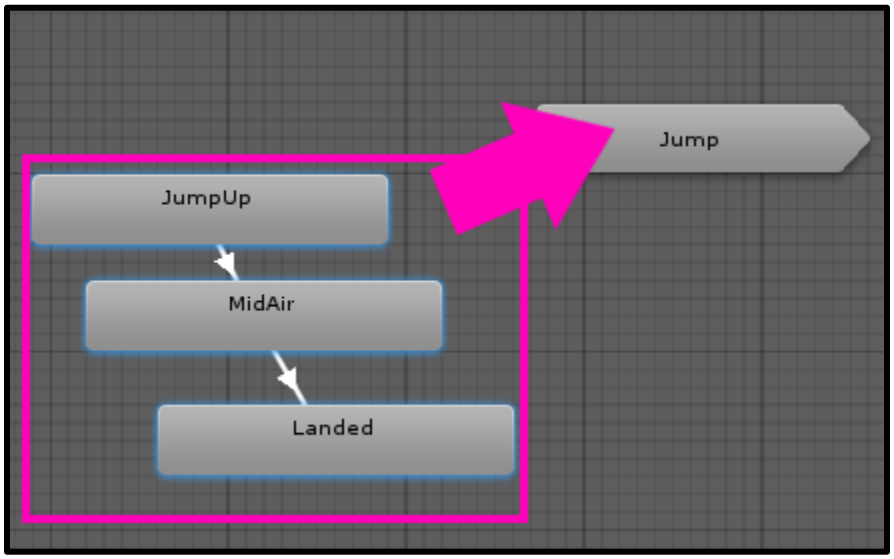
Right-click in empty space and select "Create Sub State Machine."



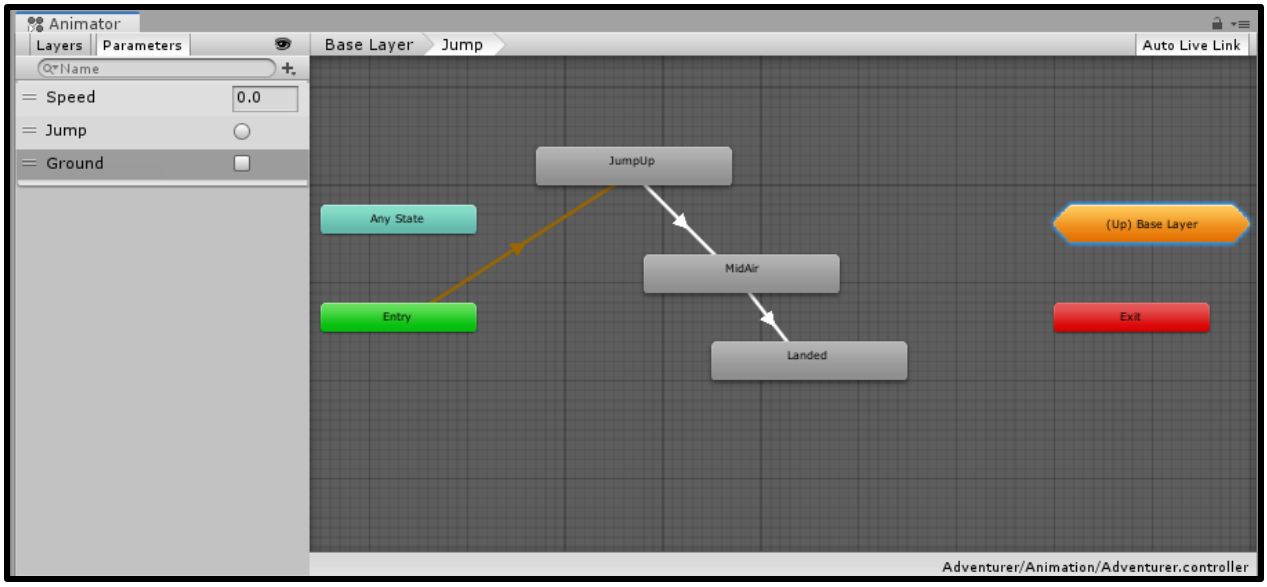
Name this "Jump." Delete the Idle, Walk, and Run animations if they're still in the animator.



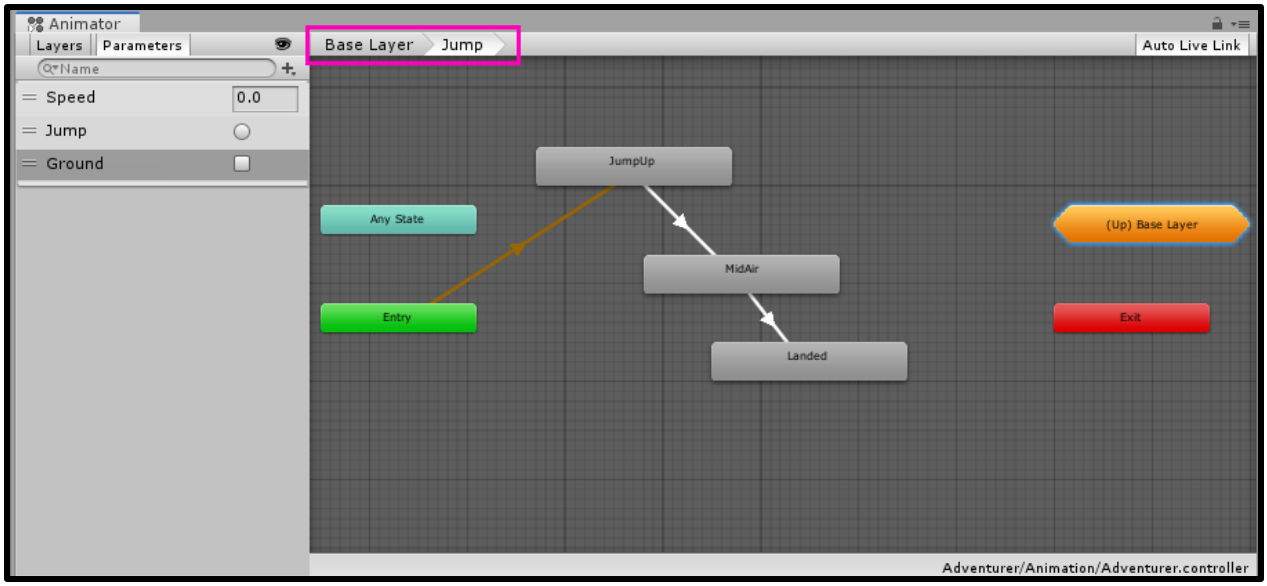
Now, select all three of our jump poses and drag them on top of the sub-state machine to place them inside of it.



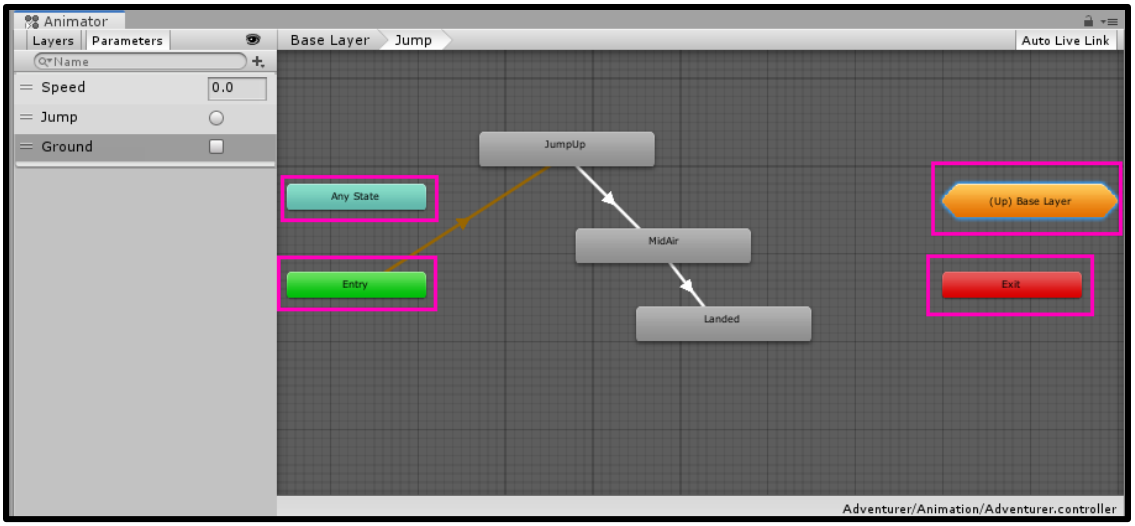
Double click on the sub-state machine to open it up.



You can go back to the Base Layer by using the bread-crumb in the top left corner.



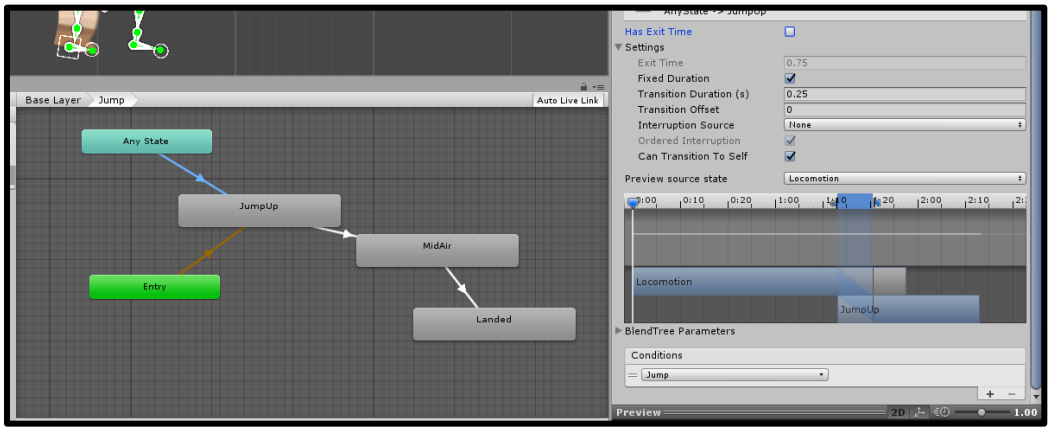
You'll notice the presence of four states. "Any State," "Entry," "Exit," and "(Up) Base Layer."



"Any State", "Entry", and "Exit" exist in the base layer; Up Base Layer is the only new state we get inside of this sub-state machine. Any State means if the condition of the transition coming from it is true, then it will transition no matter what animation is currently playing. This is the only state that will interrupt Interruption Source when it is set to "None." Entry is simply whenever the sub-state machine has been transitioned into. In the Base Layer, this is what is called right as soon as you hit play in the editor. If you were to transition into this sub-state machine, it would go through this state. "Up Base Layer" and "Exit" function in similar ways, however, in Up Base Layer you can specify which state you'd like to transition into while the Exit state simply goes to the Entry state on the base layer.

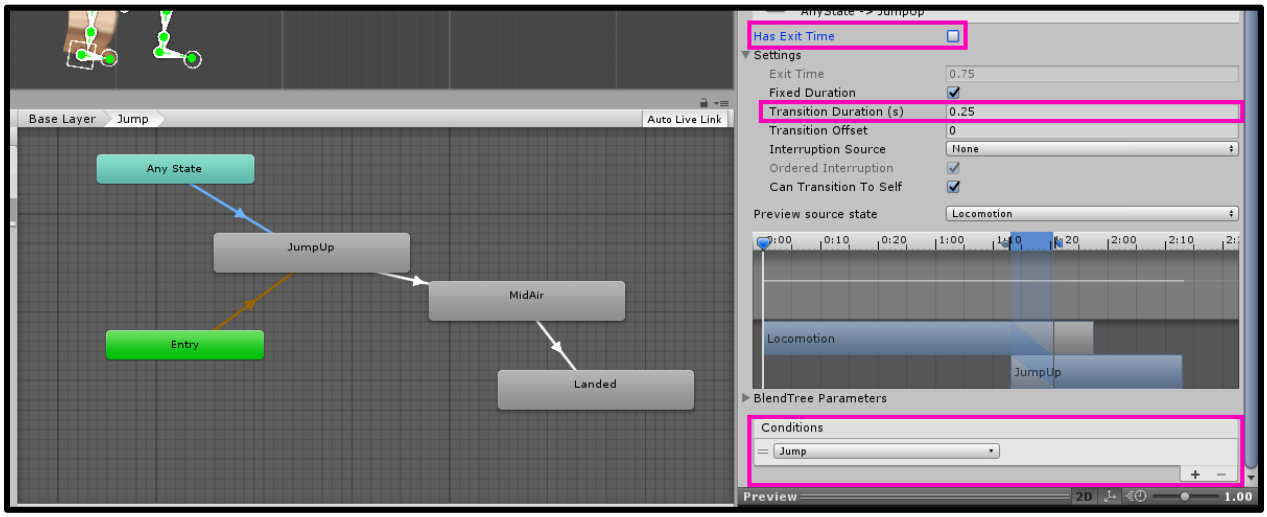
Configuring the Jump Sub-State Machine

On paper, the functions of these states may be confusing but if you take them at face value, they make intuitive sense. For our Jump Mechanic, we want to transition from the "Any State" to our JumpUp pose. Right-click on the Any State state and make a transition. This transition needs to have a condition. This condition needs to be the "Jump" trigger parameter.

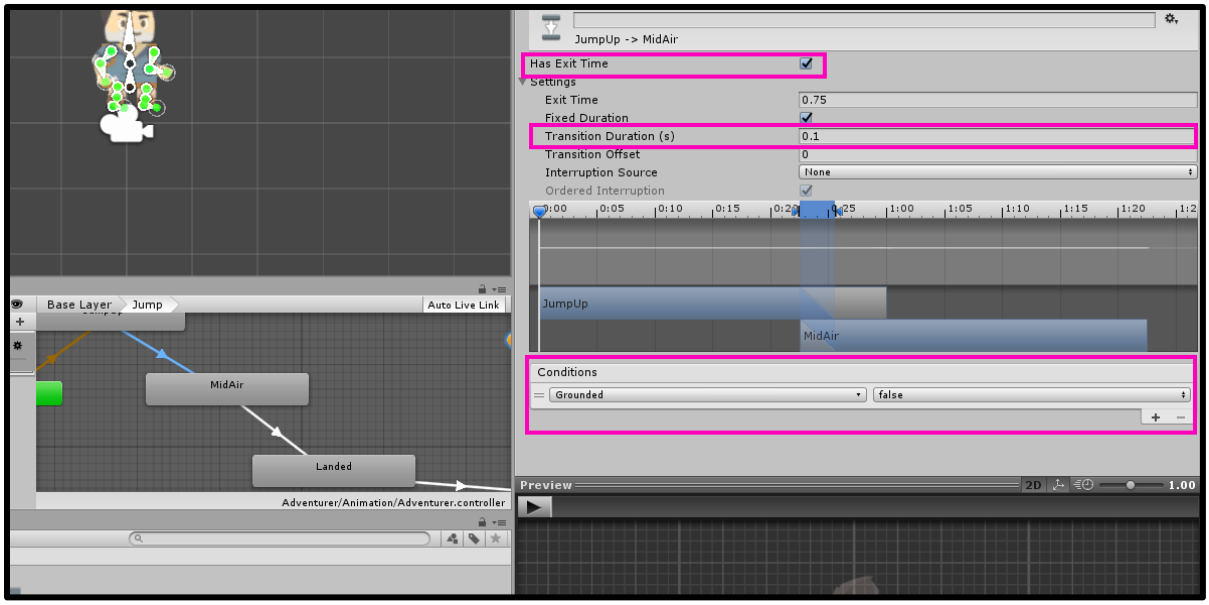


Just like that. Now, whenever this trigger is switched, the character will jump no matter what action is currently running.

The settings for this transition are simple. We need to disable "Has Exit Time" since we do not want the previous animation to run to completion and we want the character to jump right away. Next, we need to specify a duration. It should be something very short since, as I've stated before, we want there to be no delay with this jump. I've found a value of "0.25" to work well.

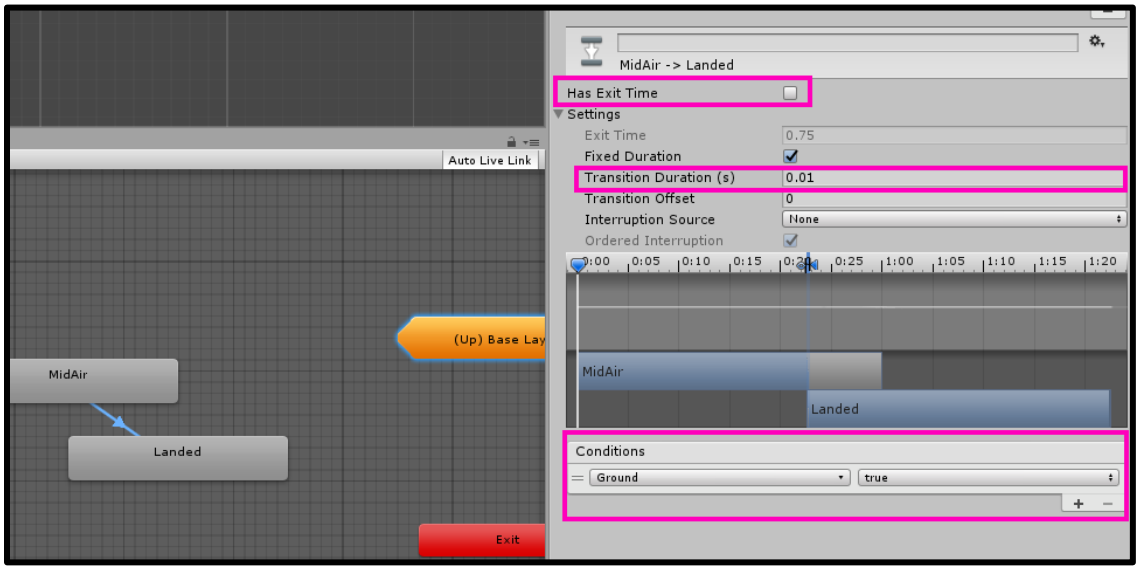


For this next transition (the Jump to Mid Air transition) we make use of the "Grounded" boolean parameter. Since this parameter is what we will use to determine whether the character is touching the ground or not, we want the "Mid Air" pose to play while the character is in the air. Create a condition for the Grounded parameter when it is set to "false." This is the only condition we need so let's set the other settings. It doesn't matter much if there is an exit time on these animations since they are so short. Simply leave it to whatever it is by default. Now we need to set a duration. We want the transition to be very quick as we transition from Jump to Mid Air. Set it to "0.1" as this gives us just enough transition while not slowing down the mechanic.



We are done with this transition! We just have to configure two more.

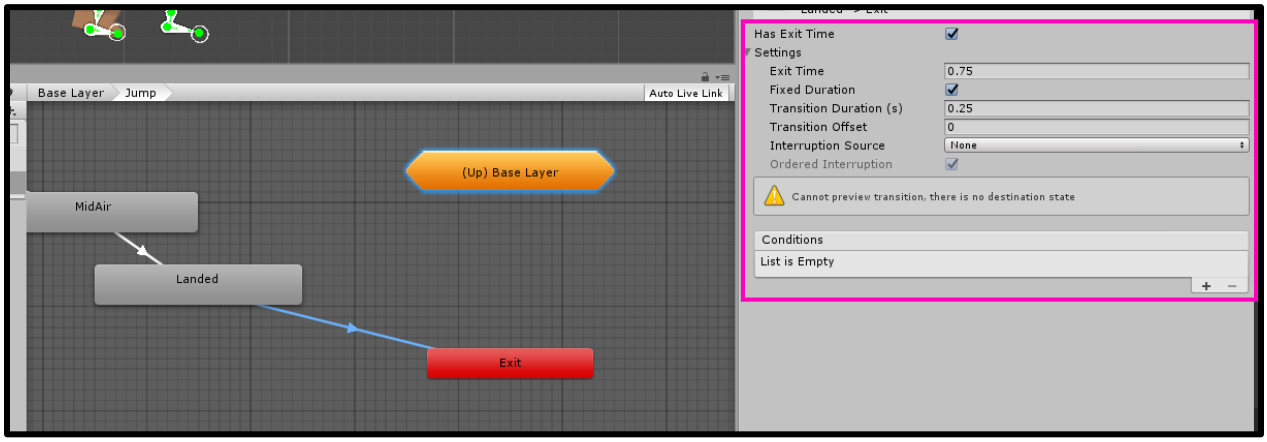
The Mid Air to Landed transition is similar to the previous transition in that it needs to be extremely quick. This is the moment where the character impacts the ground and it would look awkward if there was a delayed reaction. Set the duration to be almost nothing at "0.01". For this transition, it might be a good idea to disable exit time in case leaving this checked would contribute to a few moments of delay. And finally, we need to have a condition. Just like the previous transition was when the character was in the air, this one will be when the character impacts the ground. Therefore, we need a condition for the Grounded parameter when it is set to "true."



And with that, our transition configured! Now on to the final one.

This final transition is when the character "picks himself up" after impacting the ground. This transition should have no conditions and we can leave exit time enabled since we're not worrying about delayed reactions here. As for the duration length, I set mine to 0.25 but I encourage you to tweak this value. A larger value will get a "slow getup" look while a shorter value will have a "quick pick up."

The final aspect we need to configure with this transition is where it will transition to. We have two options, we could transition into the Up Base Layer state or the Exit state. We could get the same result for each option but one has some apparent benefits that the other doesn't possess. The best choice here is to transition into the Exit state for two reasons; the first being that it increases the scalability of the Animator Controller. If we used Up Base Layer, we would have to specify which state to transition to. This isn't a problem for us now since we only have one other state (the Locomotion state) but what if we had more than one? We would have to prioritize which state to transition into after we are finished jumping which is something that shouldn't be done when it comes to creating an efficient Animator Controller. The second reason is that it makes our animator controller look very tidy. Have a look at the Base Layer and notice how clean it looks having only one transition on the Locomotion state. Ultimately, transitioning to the Exit state is the best choice for tidiness and scalability.

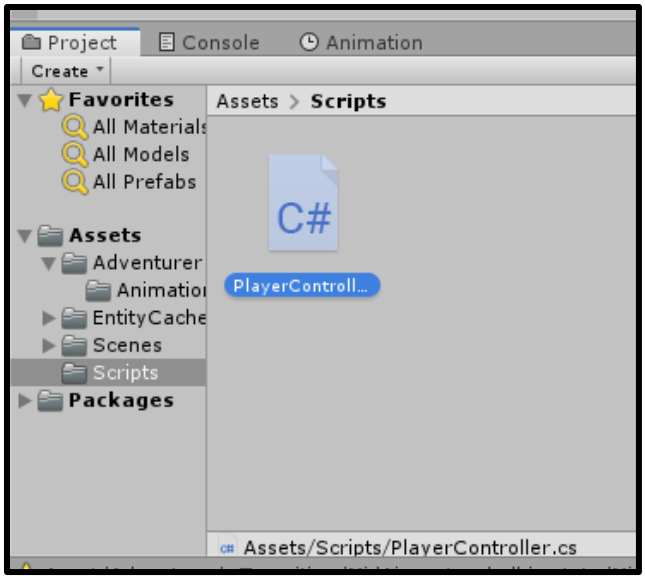


Alright! That is all for our Jumping Mechanic! It is now complete

Scripting our Character

If you were to hit play, you could control the character by fiddling with the values in the parameters panel but that makes for a very lousy game. Instead, we need to be able to modify these values through some form of input such as the keyboard.

Create a new folder called "Scripts" and create a new C# script called "PlayerController."

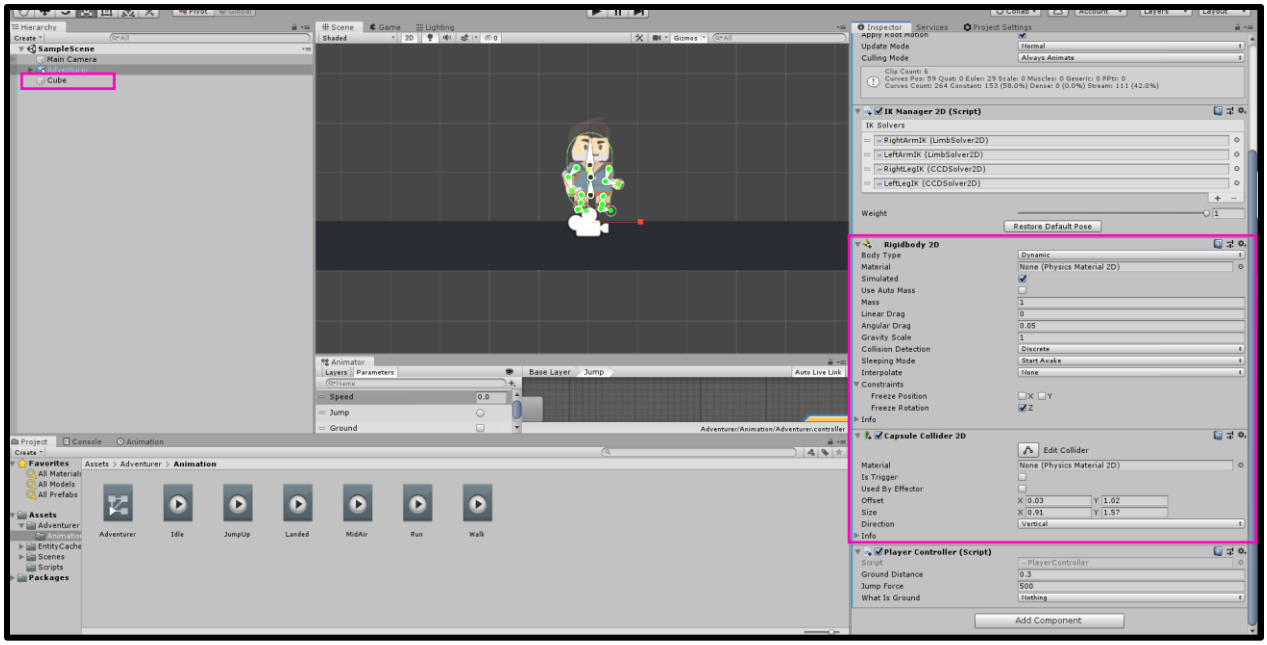


Here is the code that will go into it:

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

public class PlayerController : MonoBehaviour
{
    private Animator thisAnim;
    private Rigidbody2D rigid;
    public float groundDistance = 0.3f;
    public float JumpForce = 500;
    public LayerMask whatIsGround;

    // Use this for initialization
    void Start()
    {
        thisAnim = GetComponent<Animator>();
        rigid = GetComponent<Rigidbody2D>();
    }

```

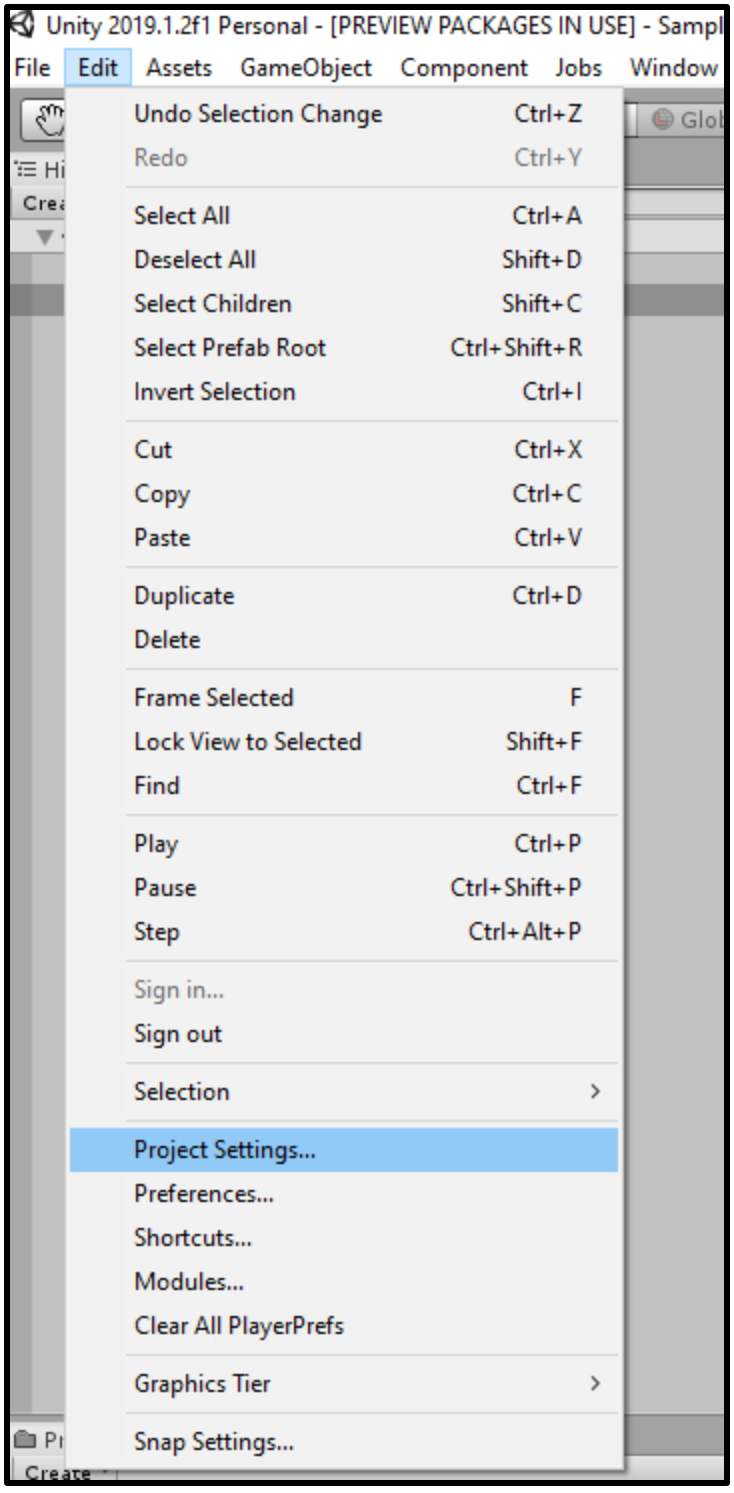
```
    // Update is called once per frame
    void Update()
    {
        var h = Input.GetAxis("Horizontal");

        thisAnim.SetFloat("Speed", Mathf.Abs(h));

        if (h < 0.0)
        {
            transform.localScale = new Vector3(-1, 1, 1);
        }
        else if (h > 0.0)
        {
            transform.localScale = new Vector3(1, 1, 1);
        }

        if (Input.GetButtonDown("Jump"))
        {
            rigid.AddForce(Vector3.up * JumpForce);
            thisAnim.SetTrigger("Jump");
        }
        if (Physics2D.Raycast(transform.position + (Vector3.up * 0.1f),
            Vector3.down, groundDistance, whatIsGround))
        {
            thisAnim.SetBool("Grounded", true);
            thisAnim.applyRootMotion = true;
        }
        else
        {
            thisAnim.SetBool("Grounded", false);
        }

    }
}
```

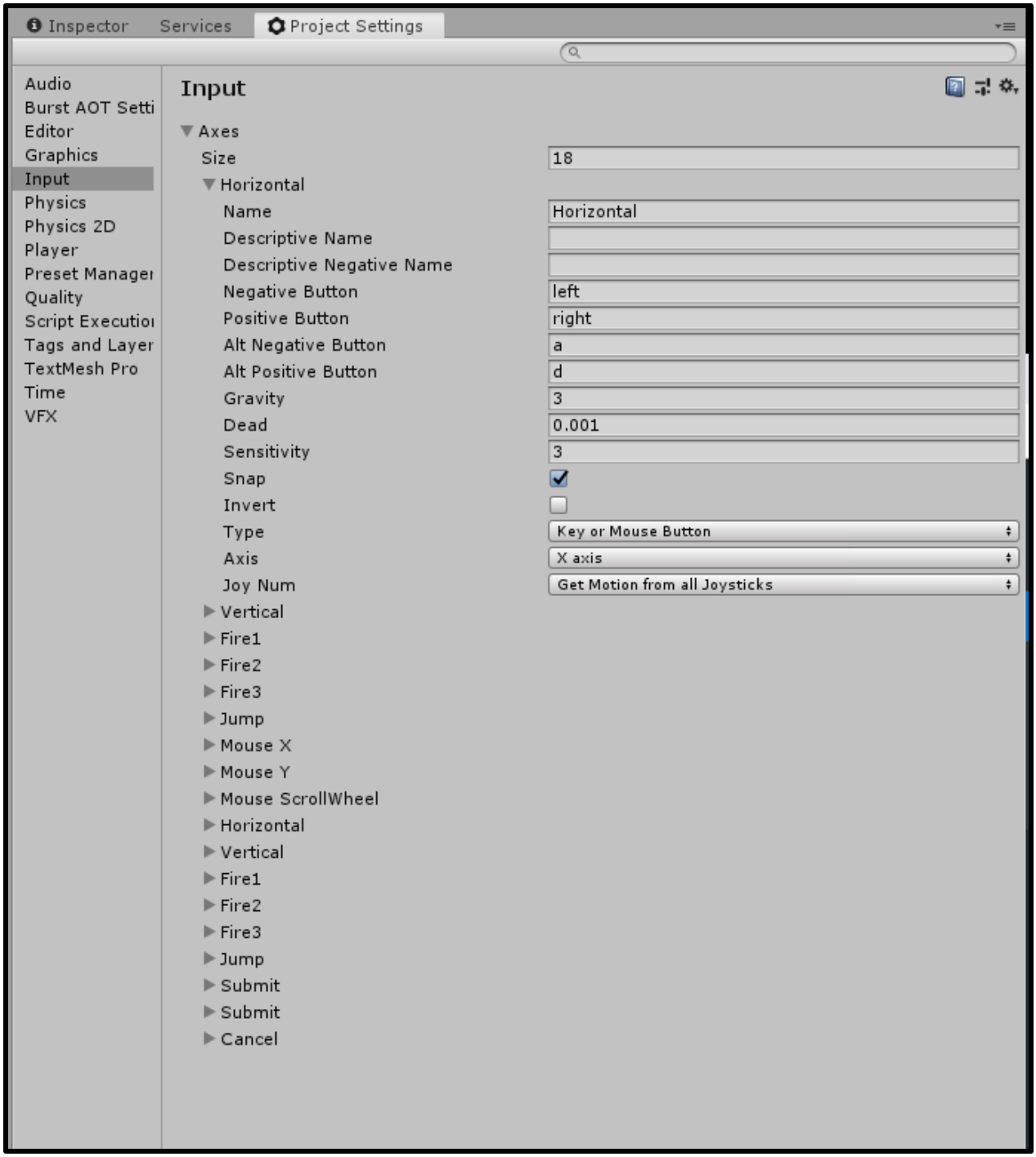
This script requires the character to have a 2D Rigidbody component and a collider of some sort. I set up my scene to look like this so I could test out my character:



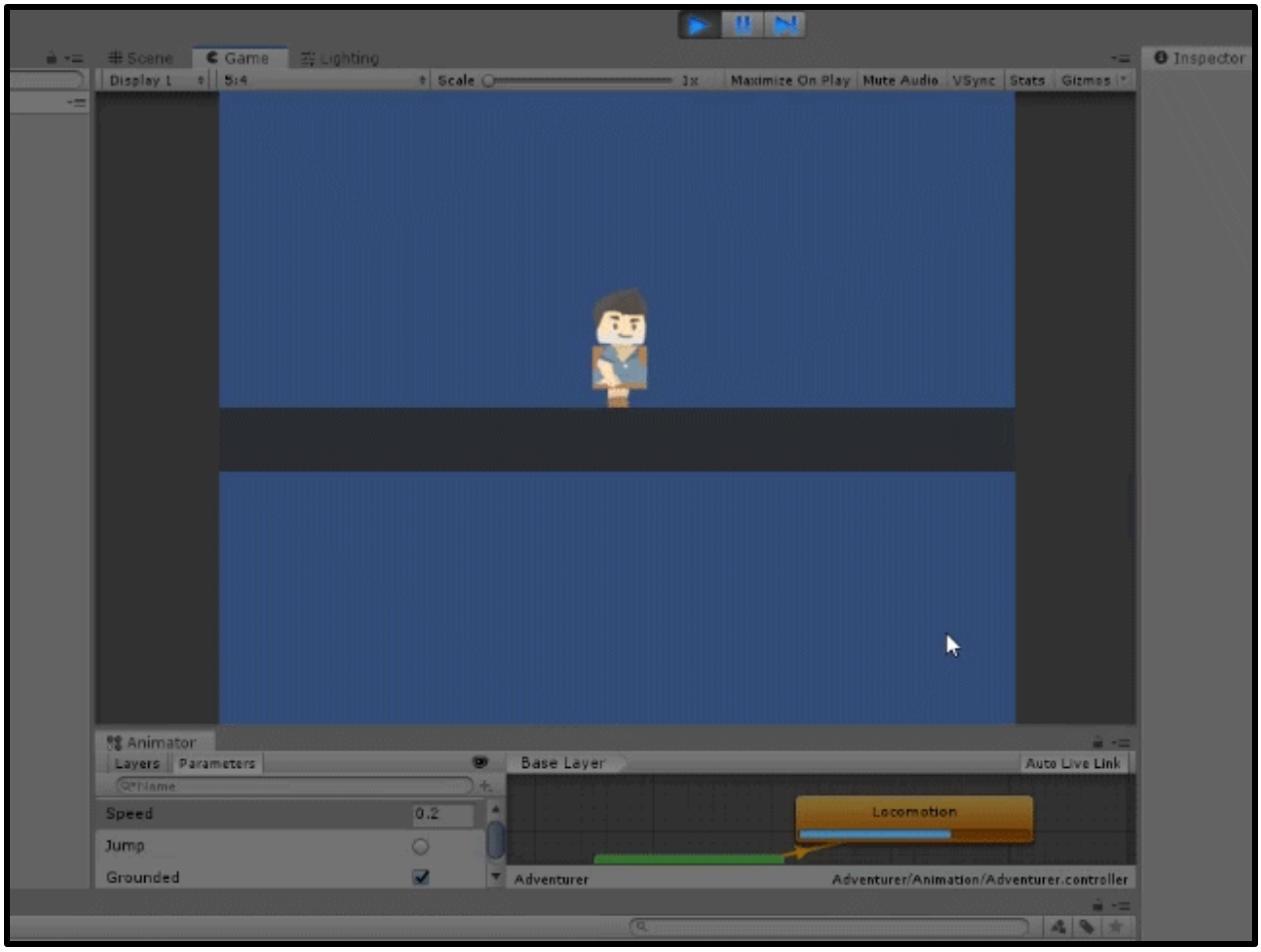
With this script, it is important to note how we access the parameters from the animator controller. We use a variable of type "Animator" and access a parameter by saying ".SetBool(*Name of Parameter*, *value*)." Two other important features are how we take input and how we assign the value of the "Grounded" boolean. Assigning this boolean was the most mystifying for me when I was learning how to work the Unity Animator. What we're doing is sending a raycast (a raycast is simply a ray shot in a specified direction that can collect data about what it hits) in the downwards direction and thereby determining if the character is touching the ground or not.

The way we are taking input is by saying "Input.GetAxis("*specify name of the axis*)." This is a good way to take input since it is cross-platform, meaning, we wouldn't have to change this value if you were to export it as a console game or as a PC game. Go to Edit -> Project Settings -> Input to see exactly what is going on in with this method.

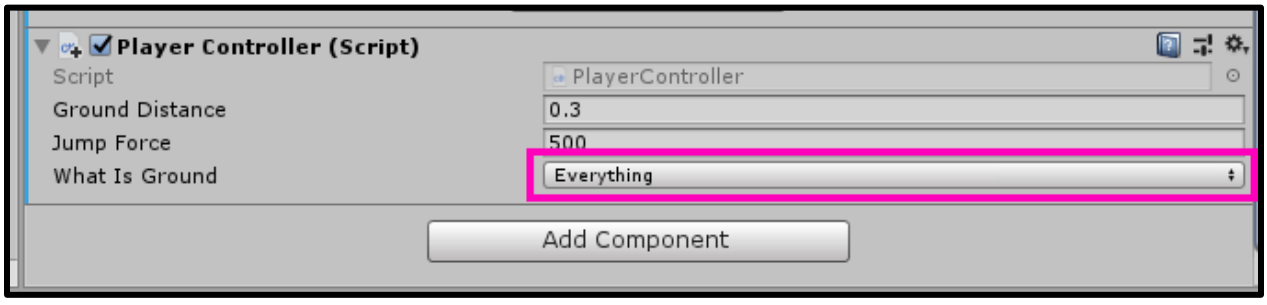




For a desktop game, the "Horizontal" input is the left and right arrow keys. Therefore, if you hit play, you can now operate our character through the keyboard. Pretty cool, isn't it?



If you attempt to jump in your game and notice that the character always seems to be stuck in a jump pose, check the "What is Ground" variable on your PlayerController script. It needs to be set to "Everything" in order for it to work.



Conclusion

This character was rigged, animated, and scripted all from scratch. If you were able to complete all three of our animation tutorials, I, with the power vested in me as a game developer, do declare you an official video game "Puppet Master." It is no small achievement to have completed all of that. I am sure you will find these skills used over and over again in your own video game projects. That's it for me.

Keep making great games!

Cinemachine and the Timeline Editor for 2D Game Development

Introduction

In mid-2017 Unity Technologies announced a new system of cameras that would allow the creator to make complex shots and compositions without a single line of code. The only problem was that it was restricted to 3D. 2D was left out of this enormous innovation. Until late 2017 and early 2018 when Unity technologies released Unity 2017.3. This new version brought this suite of cameras, called Cinemachine, to 2D. The category of games the benefited the most from this update is 2D Platformer games. Now creators can properly track a character without a single line of code. And the best part is that it is completely customizable so that you can get the exact look you are after. Unity Technologies also released a new tool called the Timeline Editor. This allows the creator to manipulate game objects and cameras together as if you were editing a movie. With these tools, anyone can create beautiful compositions without a single line of code, for 3D and 2D. In this tutorial, we will be creating a simple 2D composition with these tools and, even though this composition is simple, it will cover several techniques that will really help you with any future projects.

Assets and requirements

You can get all of the assets to be used in this tutorial from this pack here. This tutorial is designed to be beginner friendly. However, if you don't quite understand something or would like more information then I suggest you take a look at these other tutorials on the Game Dev Academy:

- Cinemachine and the Timeline Editor Part 1
- Cinemachine and the Timeline Editor Part 2

Planning our project

The project we are making served as the introduction to a game I made. Since 2D platformers benefited the most I decided to make one.

This is what we will be recreating in this tutorial. However, nothing is set in stone. If you have a completely different idea then do that one instead. With that said, you should have a pretty good idea as to what is going on in this composition but here is the breakdown just in case:

Camera:

- 1 Shot of the inside of the building
- 1 Shot of the outside
- 1 Shot where the camera zooms out and we see the inside and outside

Actions:

A guy ***runs*** up and ***throws*** a bomb which makes the door ***explode***. Then the guy ***runs*** inside and starts ***typing***.

So that's our plan! Notice the words in italics in the "Actions" section, these are the things that we are going to animate for our character. Although, instead of the guy throwing a bomb, I think we should make the guy punch the door. The explosion wasn't really necessary and it will just increase our project size. So our revised plan looks like this:

Actions:

A guy ***runs*** up and *punches* the door which destroys the door. Then the guy ***runs*** inside and starts ***typing***.

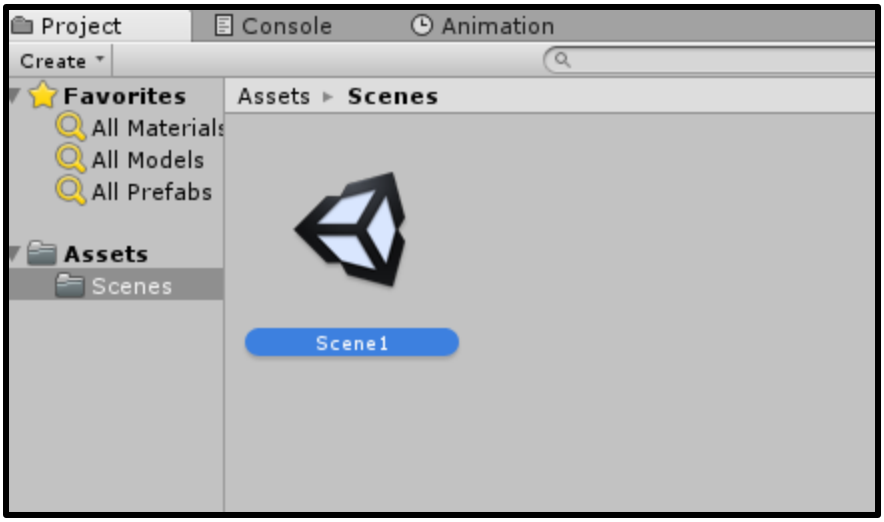
It is very important that you construct a plan like this whenever you go to make your own compositions. It will save a lot of time and brainpower.

There also is a green block that bounces around but this is an extra part that we won't be doing. However, if at the end of this tutorial you like the green block feel free to add it in yourself.

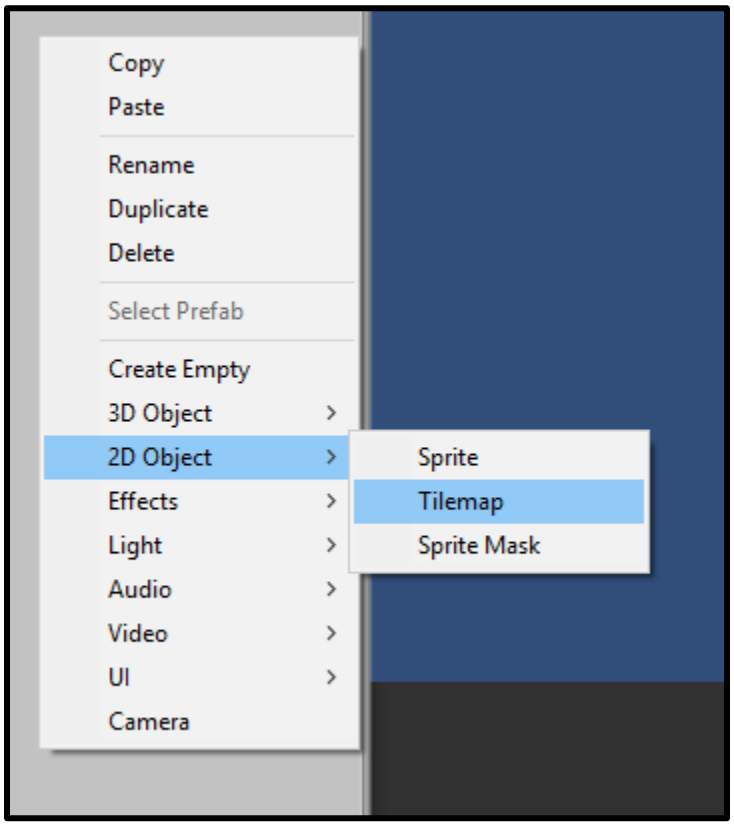
Now that we have our plan it is time to start creating!

Setting up our project

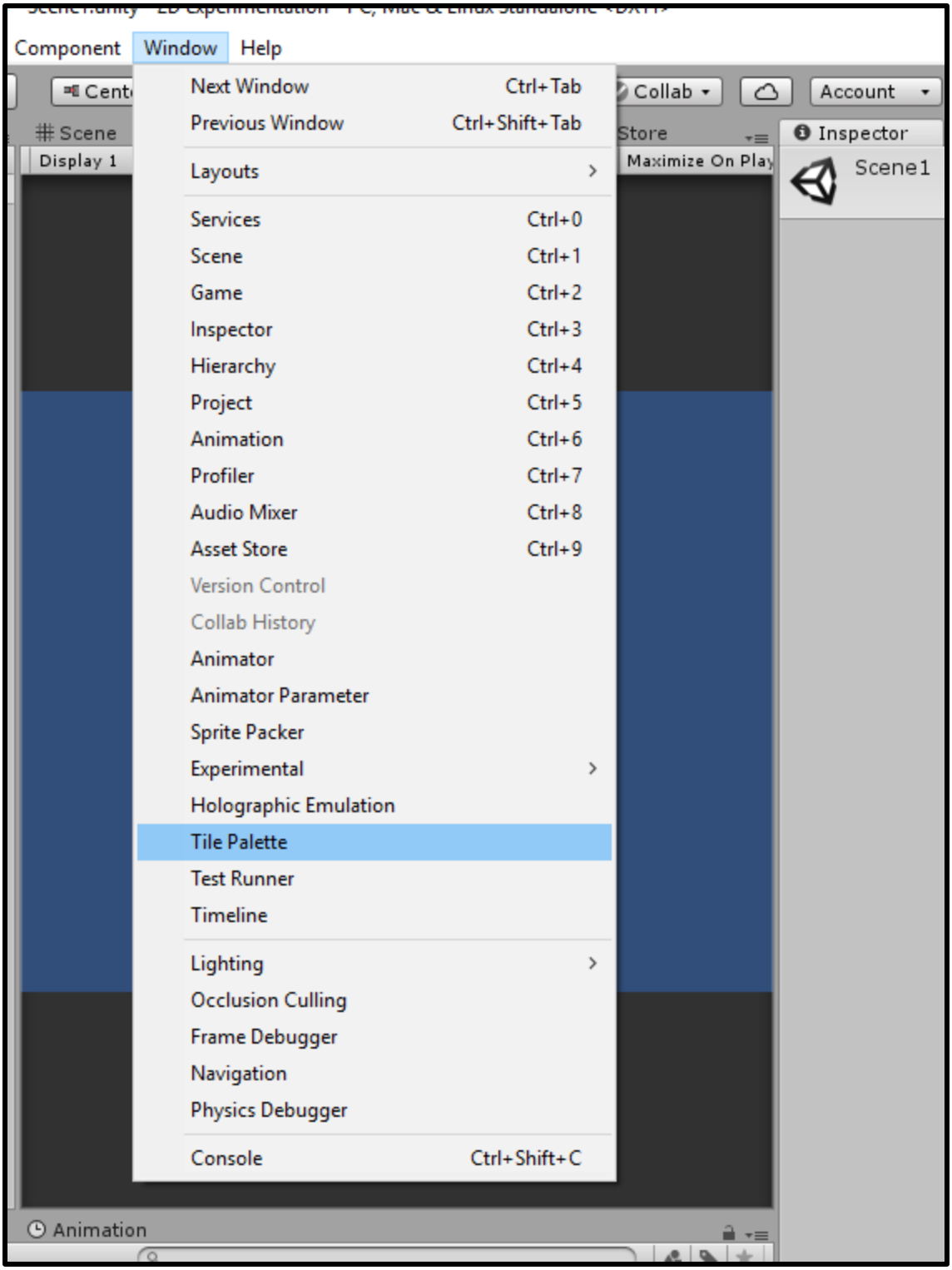
Another thing Unity Technologies introduced in 2017.3 was a new way to build levels. 2D platformers benefited once again with the new Tilemap Editor. The specifics should be left for another tutorial but basically, it allows you to create tile maps inside Unity without having to import them from a third party program. To get started using the Tilemap Editor we need to create a new project and then import the assets. Then create a new folder called "Scenes" and create a new scene called "Scene1" in that folder.



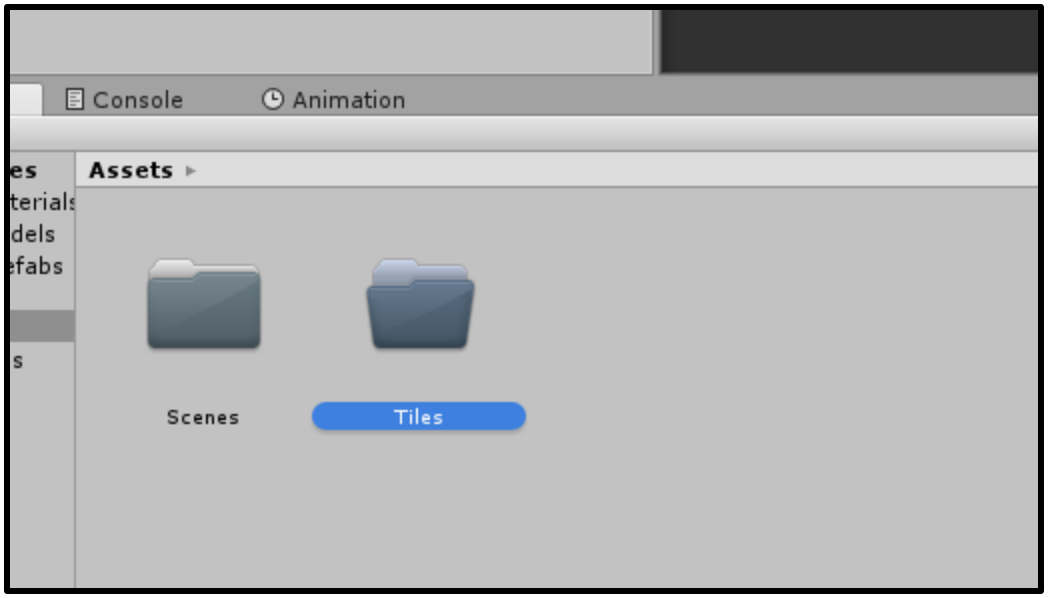
Open "Scene1" and create a new Tilemap by right-clicking in the hierarchy and going to "2D -> Tilemap".



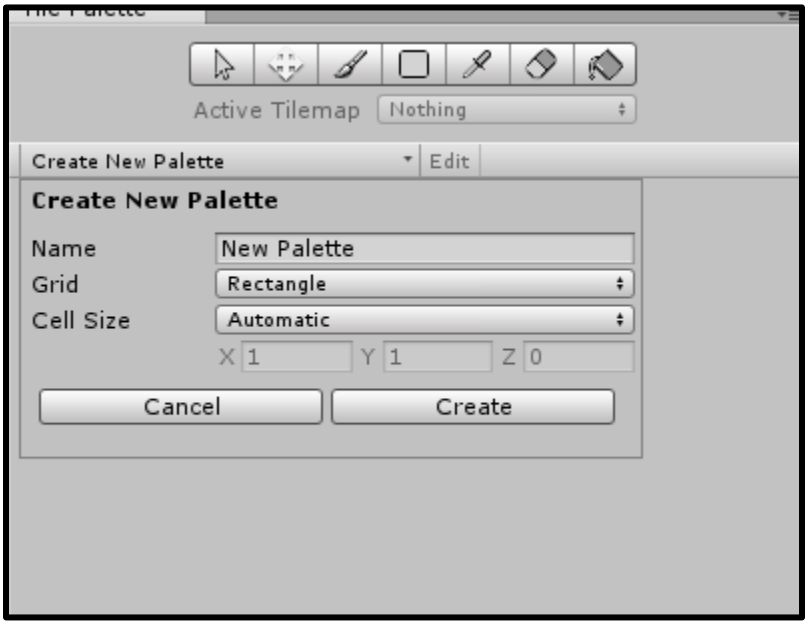
Then go to Window and bring up the Tilemap Palette and place it somewhere in your workspace.



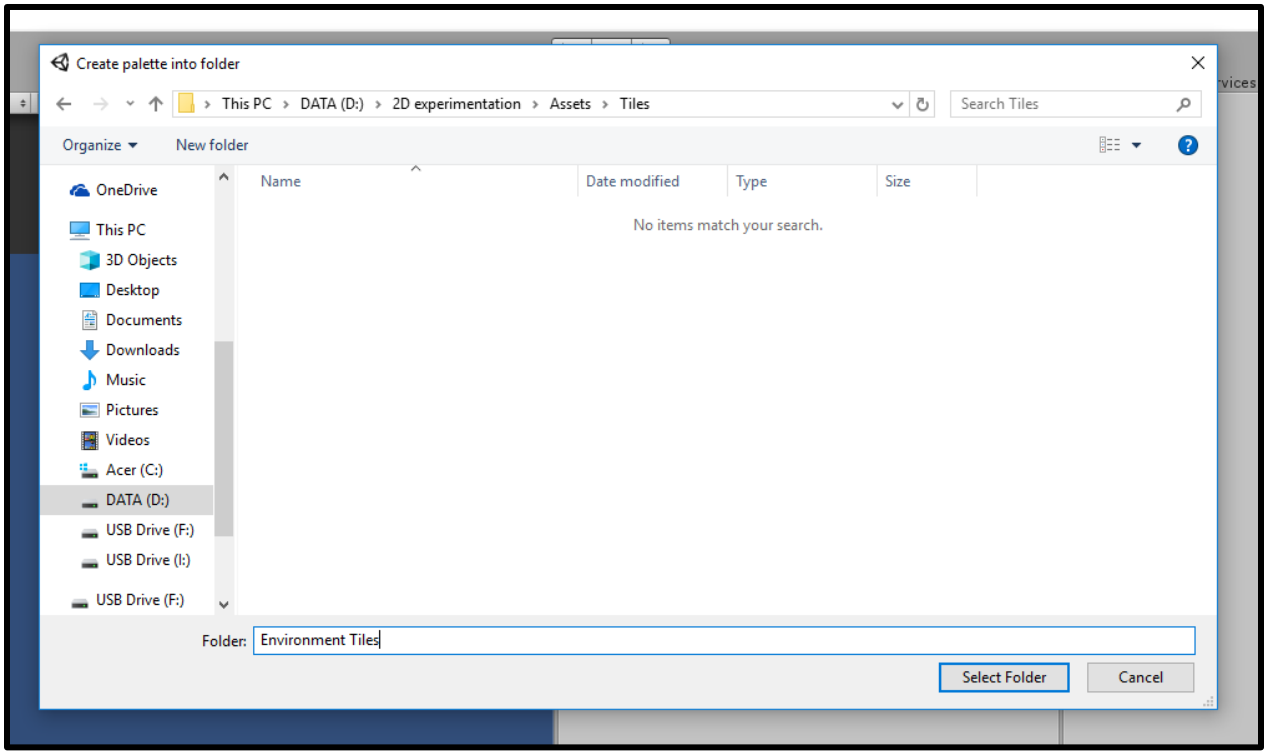
This is where we will use tiles to "paint" our world. Create a new folder called "Tiles".



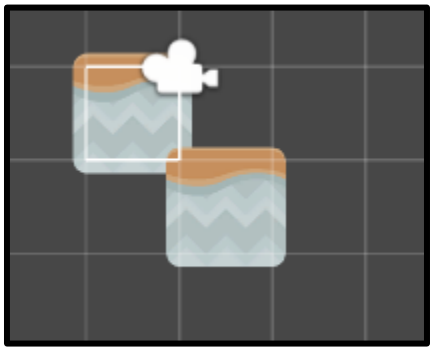
Then go to your Tilemap Palette and click "create a new palette".



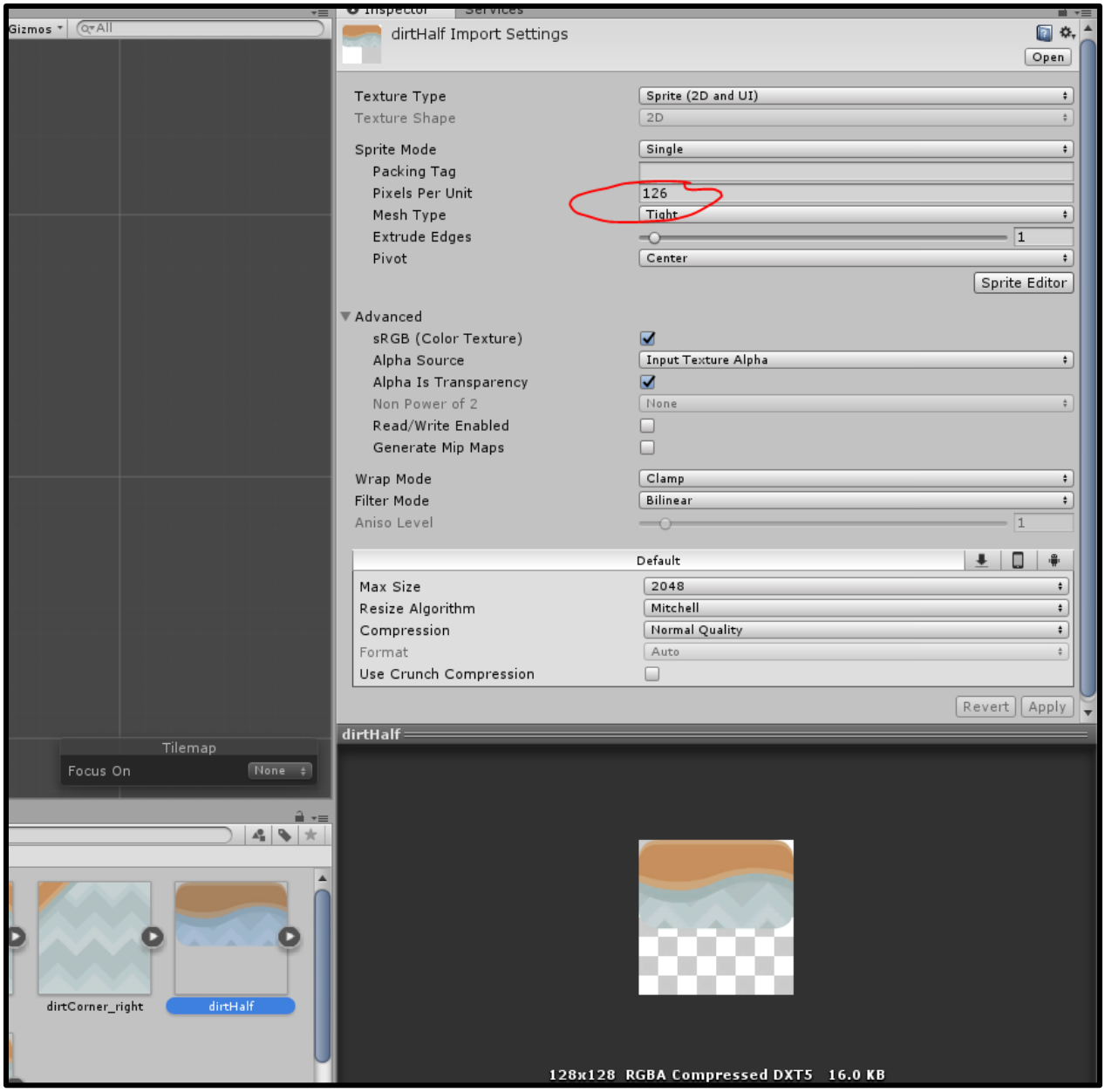
It gives us a couple options, all of them should be pretty self-explanatory. "Grid" specifies the shape of each tile, right now "rectangle" is the only one option. "Cell size" is the size of each tile, let's leave it set to "automatic" for now. Give our palette a sensible title, something like "Environment Tiles". Click "Create" and then save it to our "Tiles" folder.



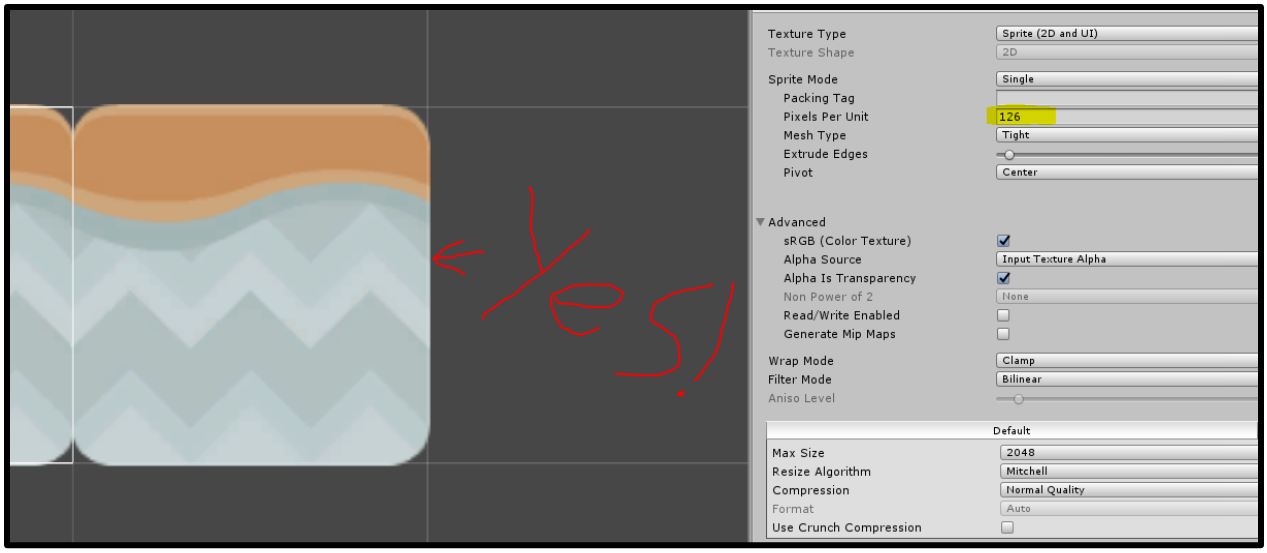
Now we need import our tiles into our newly created palette. This is where things can get kind of sketchy. It is hard to make the tile images match the tilemap's cell size.



Some tile images will fit just fine while others will not. The best way to go about fixing this is to change the "Pixels Per Unit" in the import settings.



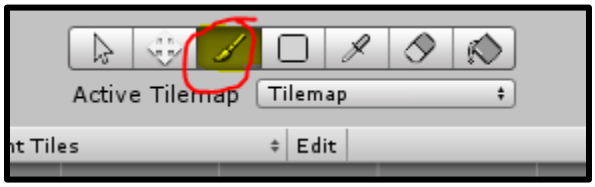
Increasing this value makes the resultant image smaller and vice versa. There isn't any other way to find the correct value except by experimentation. Fortunately for you, I have found a value that fixes this problem fairly well:



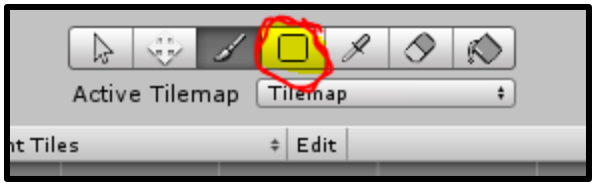
This is where you need to think about what you want your building to look like. This way you can pick the tiles you are going to need and then apply the "Pixels Per Unit" change to those few without having to go through each tile image. Once you have a good idea as to which tiles you need, go ahead and select them all (shift-click if they are all in a group or control-click if they are separate) and then input the correct "Pixel Per Unit" value. Then click apply. It is now applied to each one of your selected tiles. Now drag each one of your selected tiles into your Tile Palette window. It will want you to specify where to save each tile. This can be kind of tedious but just try to get through it as quickly as possible. Give each tile a sensible name and store it in our "Tiles" folder. Okay! Now our project is set up! it is now time to start creating!

Creating the environment

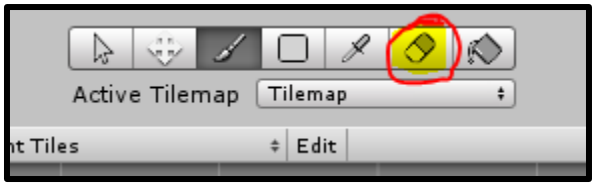
Now is when we start creating the building and the environment around it. I am going to start by giving you a quick run through of the tools in the Tile Palette editor:



This one allows you to paint one tile at a time.

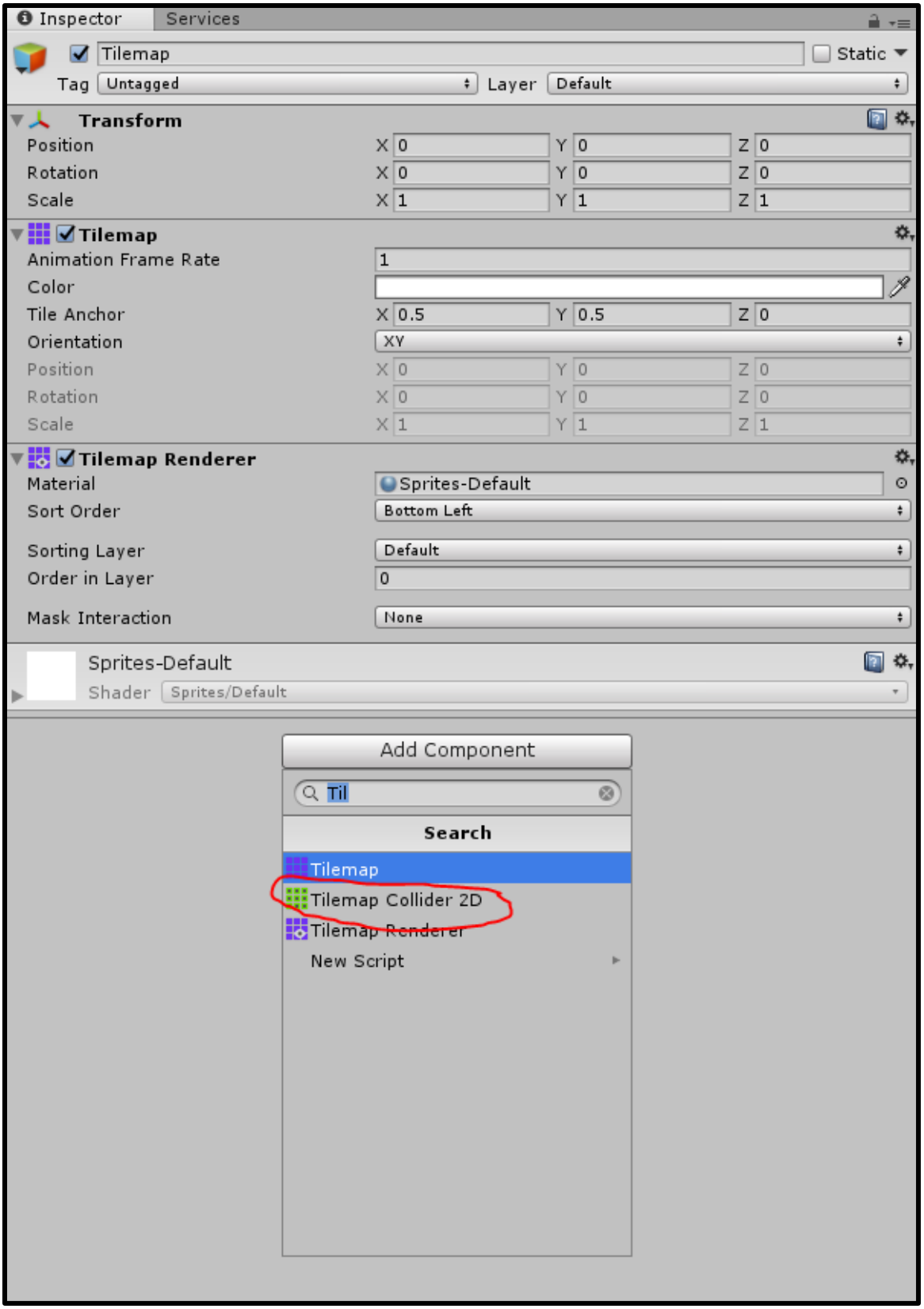


This one allows you to select an area and fill that area with the selected tile.

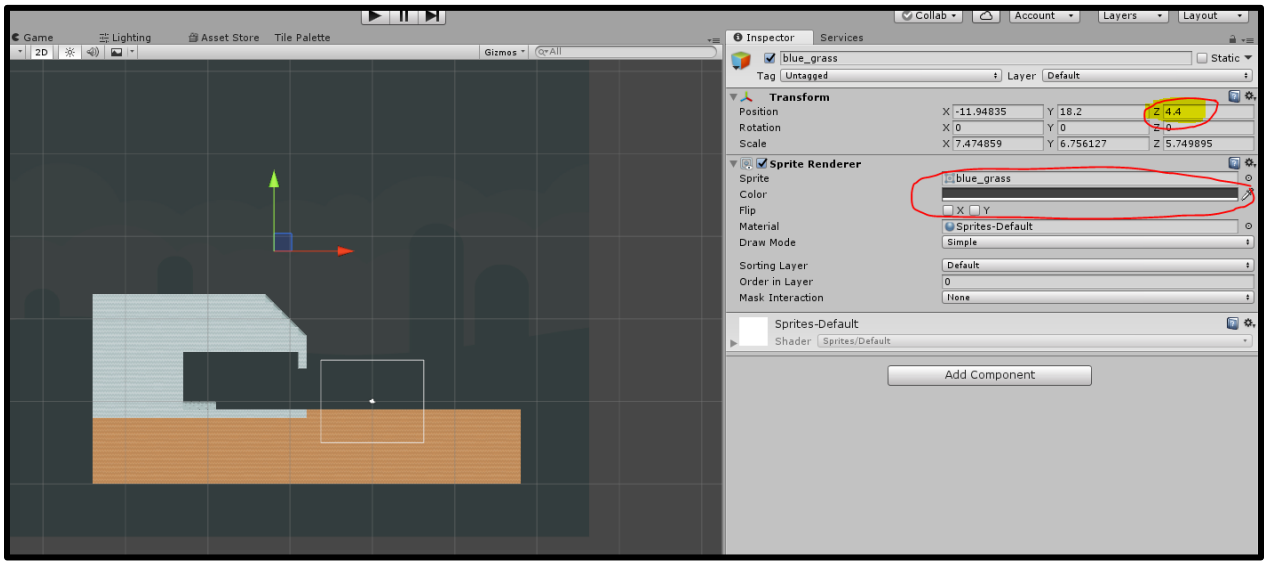


This one will erase any tile in the scene view, a shortcut is shift-click (which removes any type of tile) or control-click (which removes only the selected type of tile).

Now that you know the tools its time to get creative! Start constructing any building you have in mind! The only requirement is that it has some sort of table (for the computers). Once that is done, open up the "Grid" object and click on "Tilemap". Then add a "Tilemap Collider" component to "Tilemap".

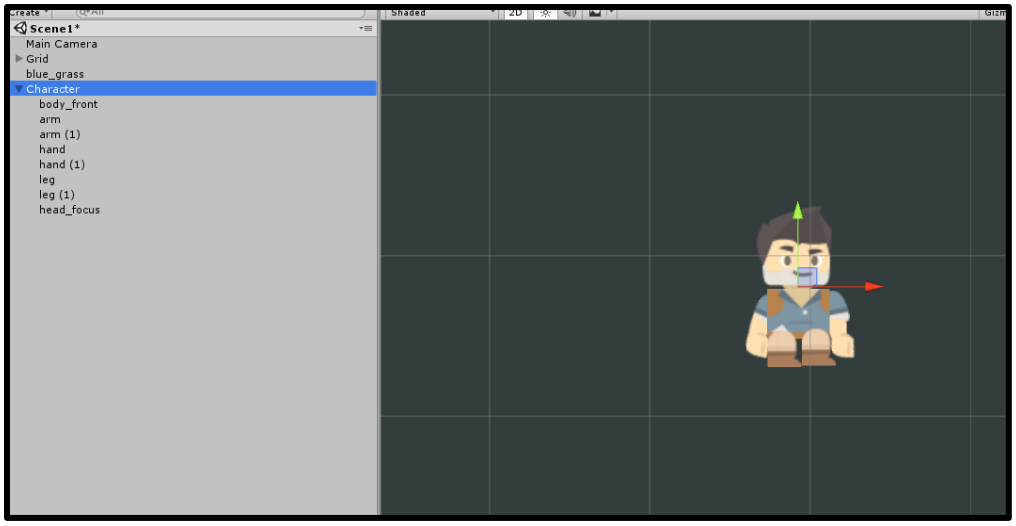


This gives each tile a collider which makes it behave like a proper environment when it comes to physics. Now navigate to the "items" folder in the asset pack and pick some items to add to your building. These can be dragged directly into your scene, no changes to the import settings are necessary. Use the "boxCrate" for the door. The final thing we need is a background. Pick an image in the "Backgrounds" folder from the asset pack. We are going for a night scene so pick one that is dark or can be darkened easily. The z-depth must be less than any of the other elements. Alright! We are done! This what my scene looks like:



Constructing the Character

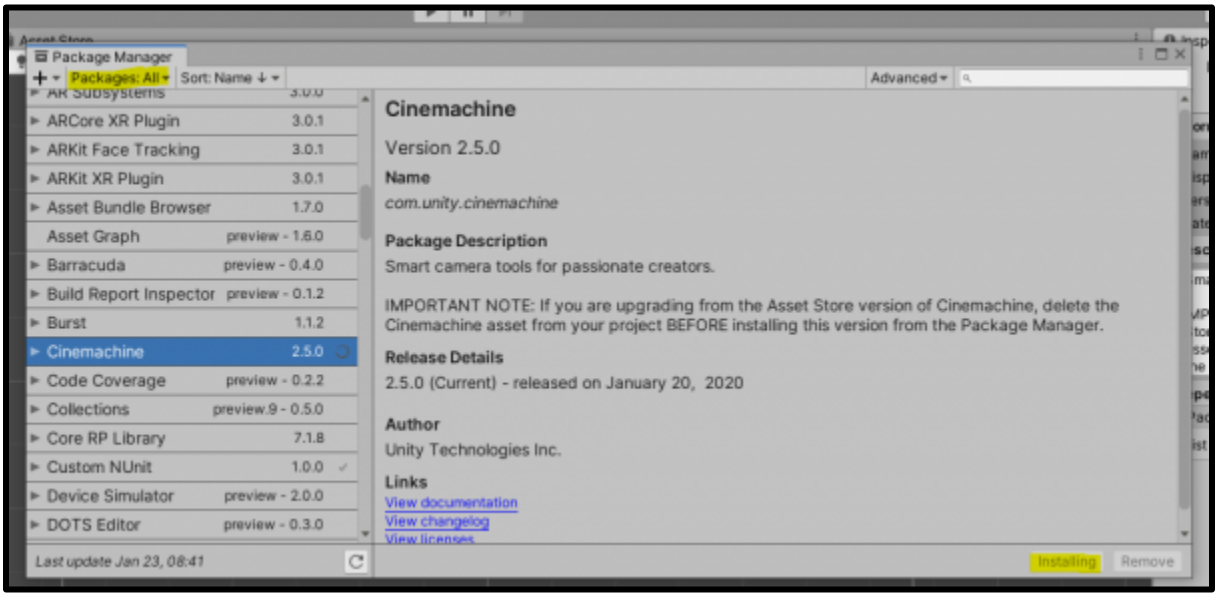
In the "Characters" folder you will find a couple of folders labeled things like "Player" and "Soldier", these are the actual characters we are going to be using. Have a look at them and pick the character that you want to go in your composition. Once you have found one you like, open the "limbs" folder. We are going to use these to put together a character. Before we do that though we need to create an empty game object called "Character". Now we can drag all of the limbs into this new game object. No change to the import settings is necessary so just start constructing! You should end up with something like this:



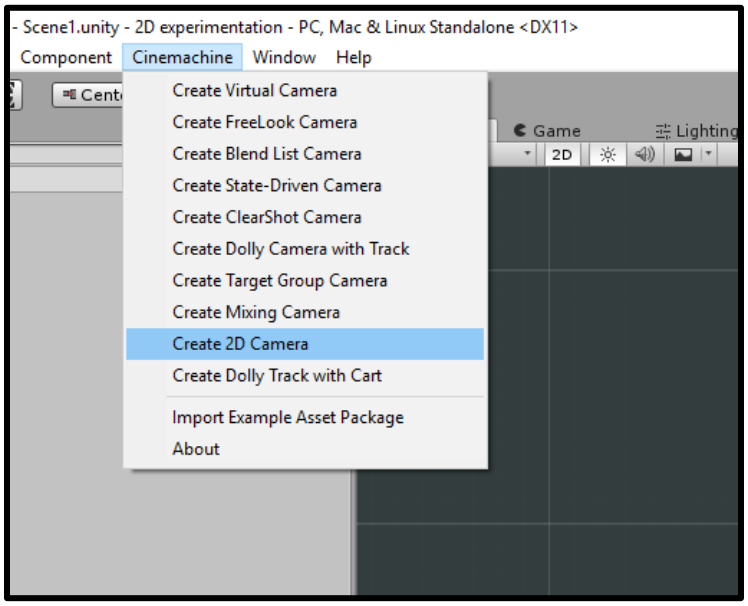
I decided to double the size of my character just because my building is pretty big.

Setting up the cameras

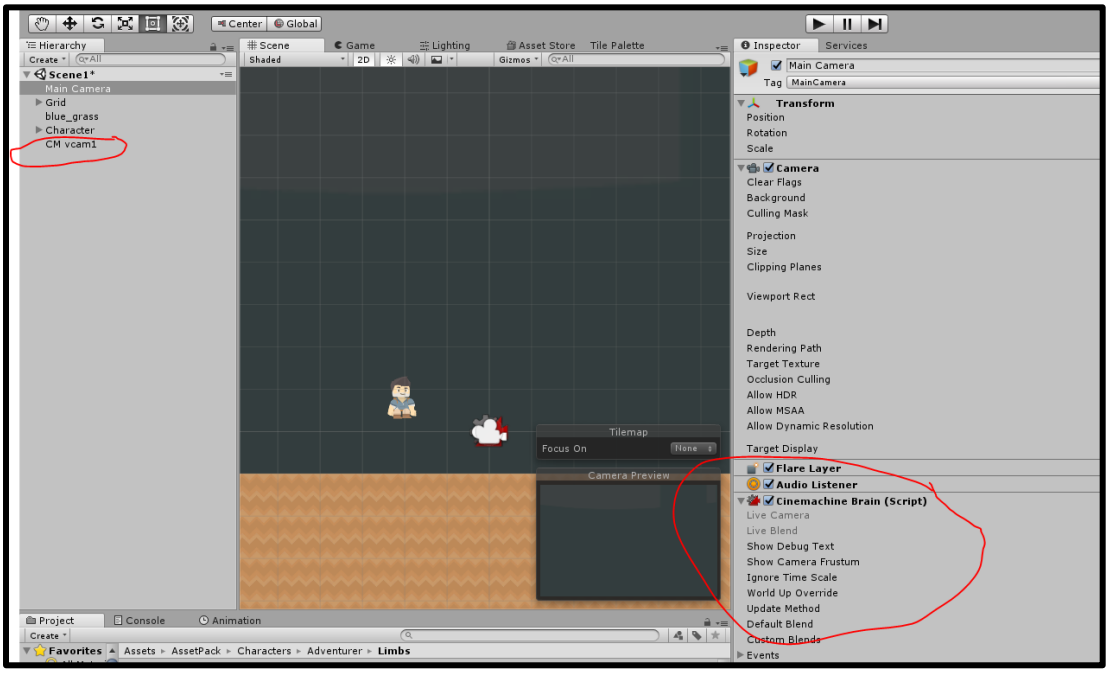
Let's start the execution of our plan by setting up the cameras. This means that we have one last bit of setup to do and that is to import Cinemachine. Go to Window -> Package Manager and set the package view to ‘all.’ Scroll down to find the Cinemachine package. Download and import it.



Now if you notice it has created a new button in the top toolbar called "Cinemachine". Click on this and select "Create 2D Camera".

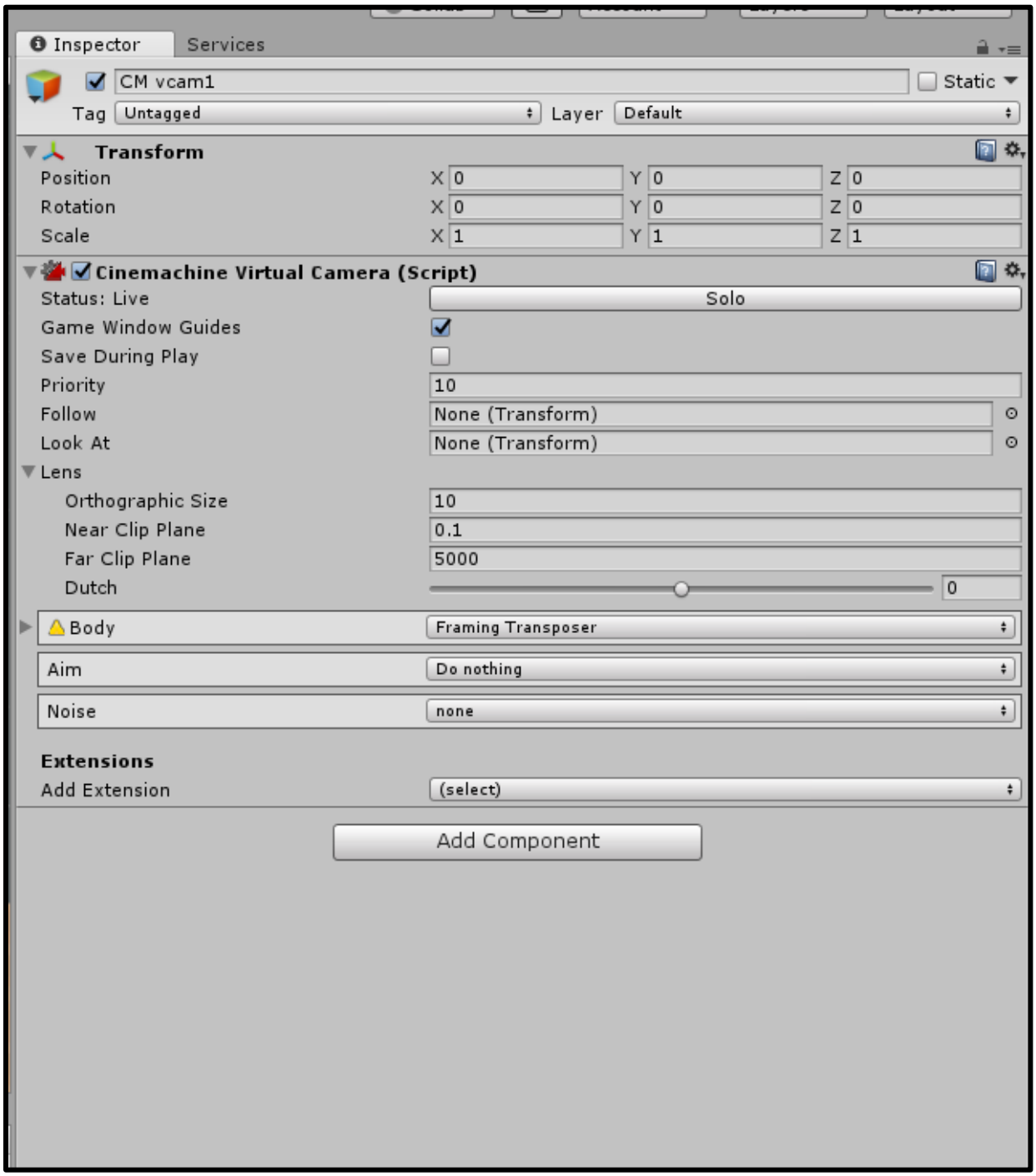


Unity has now done a couple things. The first noticeable thing is that it has created something called a "Virtual Camera". The second thing Unity has done created a "Cinemachine Brain" component on the "Main Camera".

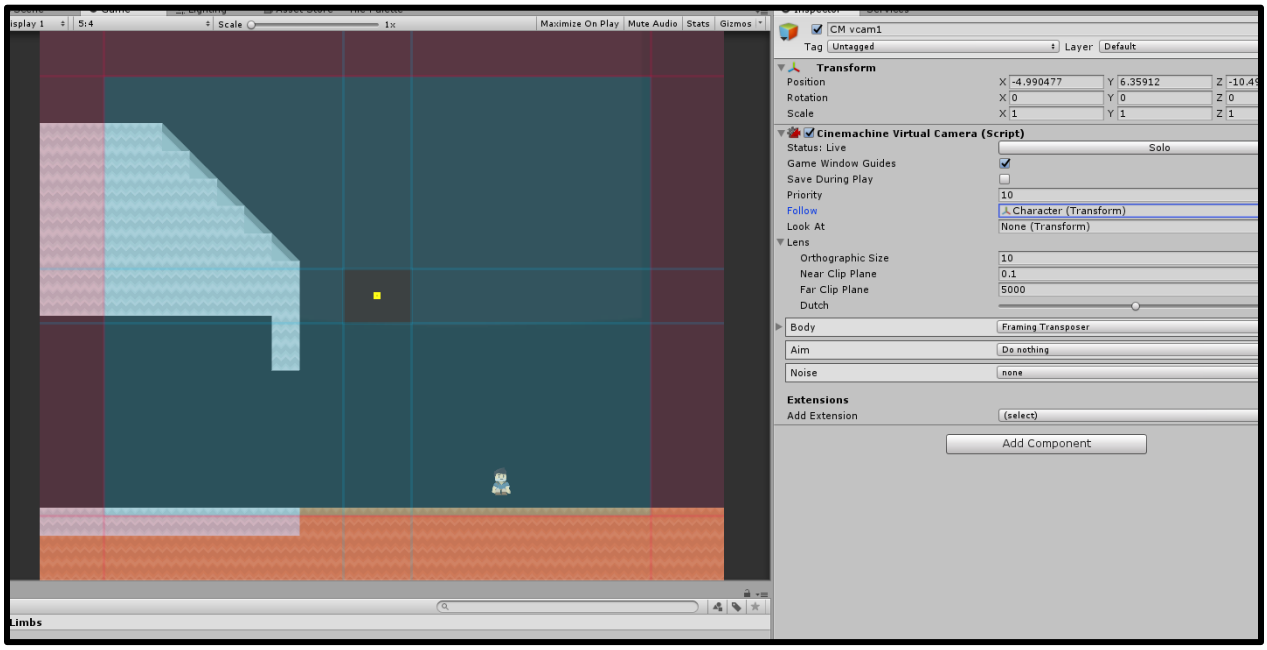


This is the way Cinemachine works, there is only one actual Camera (which is determined by the Cinemachine Brain component) and several Virtual Cameras. The actual Camera will then

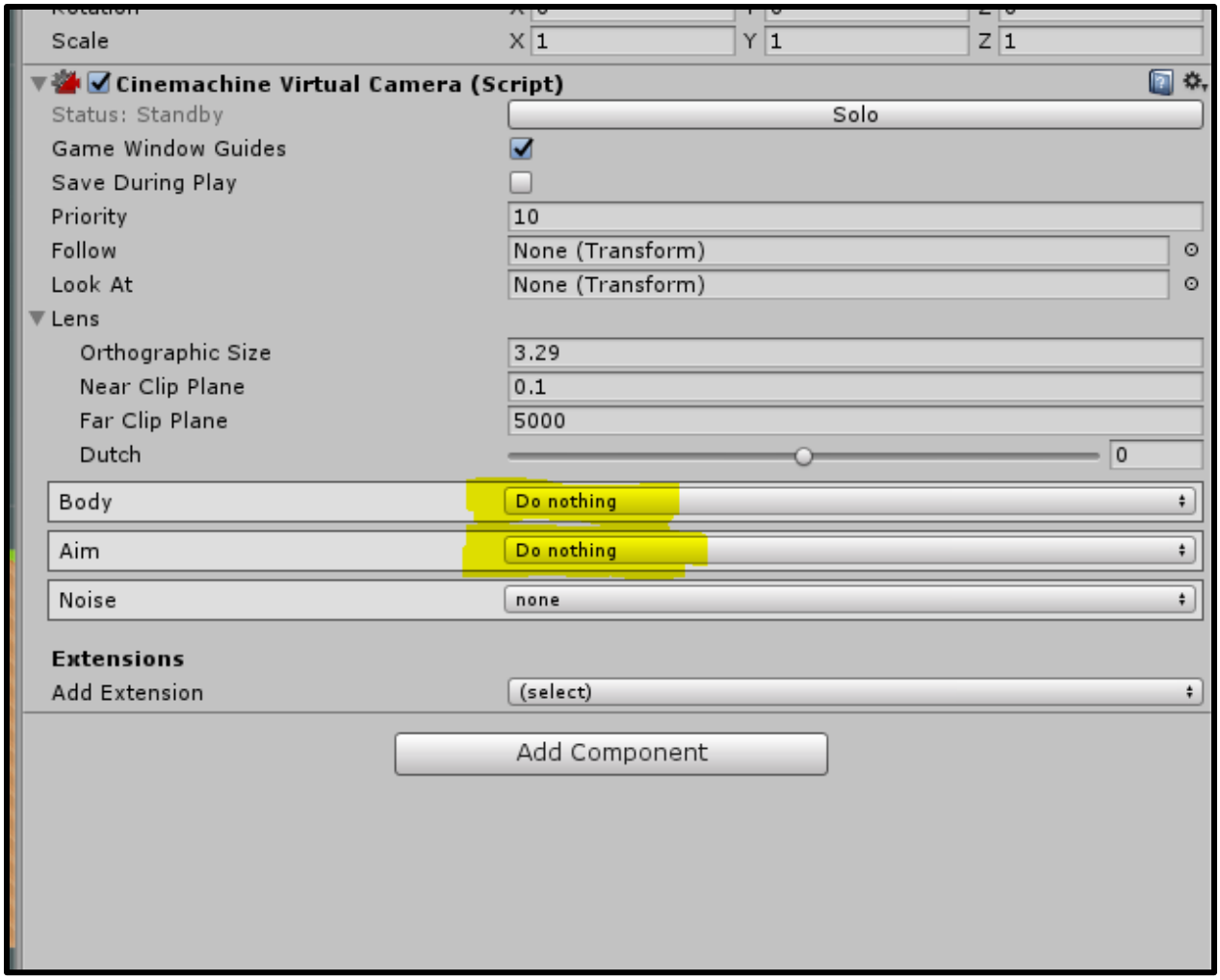
interpolate from one Virtual Camera to another. And we can decide when to transition from one Virtual Camera to another by using either a script or the Timeline Editor (which is what we will be doing in this tutorial). Let's take a look at the Cinemachine Virtual Camera component.



The first option is "Game Window Guides". With this enabled, you won't see anything in the game view until you assign something to the "Follow" field. Then you will see blue and red areas.



This is what allows the creator to tweak the look of the camera without leaving Play Mode. If the camera is following a game object it will always try to keep the game object in the middle of the guide, the part without any color, which is known as the "Dead Zone". If the object crosses into the blue area, known as the "Soft Zone", the camera will slowly interpolate in an effort to get the object back into the clear area. The red zone is known as the "Bias". A game object will never cross over into the Bias, the camera will always move to make sure the object stays in the Dead or Soft zones. You can change the size or lengths of these zones by just dragging them in the game view. You can also change the angle of the camera by clicking and dragging in the Dead Zone. This greatly benefits 2D platformers since we can now create a custom camera tracking action without doing any code. The next two fields are "Follow" and "LookAt". These should be pretty self-explanatory, the camera will follow whatever is in the Follow field and just Look At whatever is in the Look At field. Under "Lens" there really is only one setting that we can use for a 2D game and that is "Orthographic Size". If you reduce this number you'll notice that our camera zooms in. Keep this in mind as we start setting up cameras. The last few options are the "Body" and "Aim" fields. These give specifics as to how you want the camera to track or look at an object. We aren't going to be using the Look at Field so just keep "Aim" set to "Do Nothing". There aren't a lot of settings that will work with a 2D game so just leave "Body" set to "Framing Transposer", this will allow us to use the guides to tweak our camera.

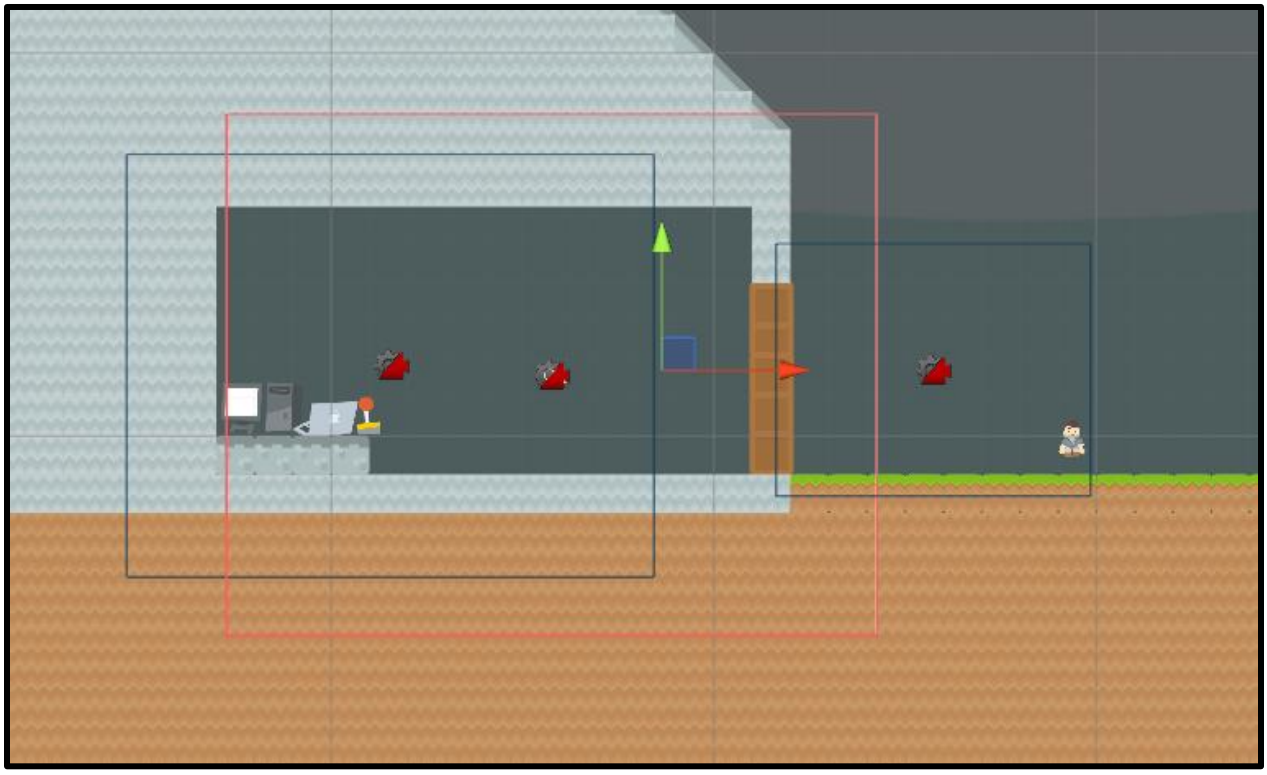


I encourage you to explore these other settings and see if there are any settings that you would like to use. Now that we have seen how the Cinemachine Virtual camera works, let's set up our cameras. For my composition, my cameras weren't that fancy so if you would like to add something more sophisticated feel free to do that but here is my camera plan:

Camera:

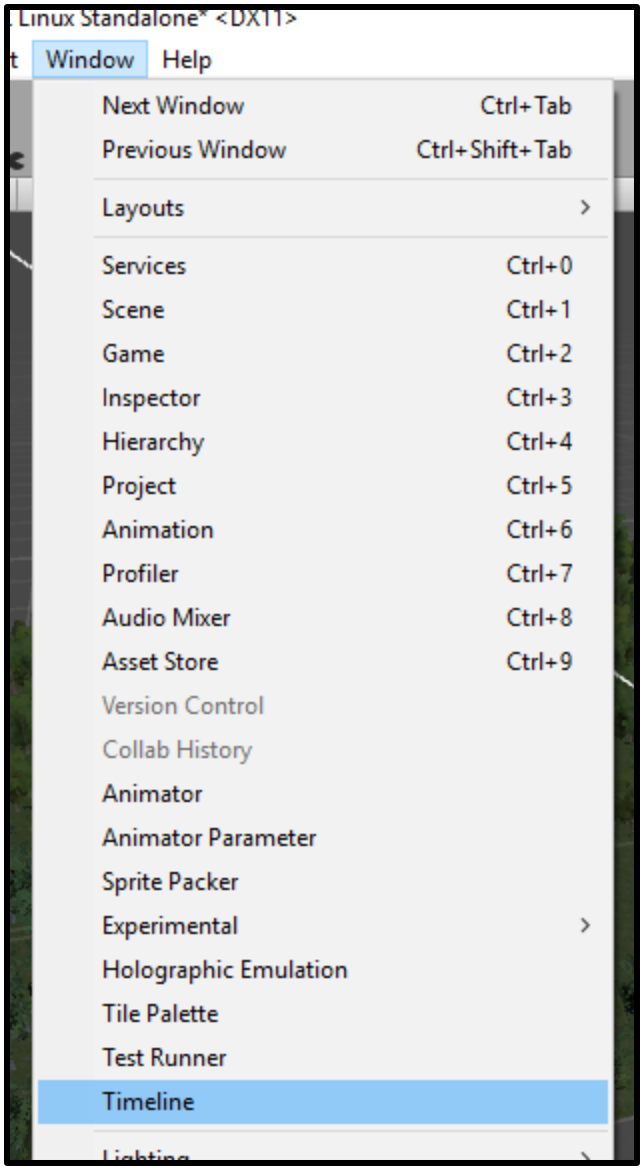
- 1 Shot of the inside of the building
- 1 Shot of the outside
- 1 Shot where the camera zooms out and we see the inside and outside

What we need here are three Virtual Cameras. One placed on the inside, one placed on the outside, and one placed where we see both the inside and the outside. Like this:

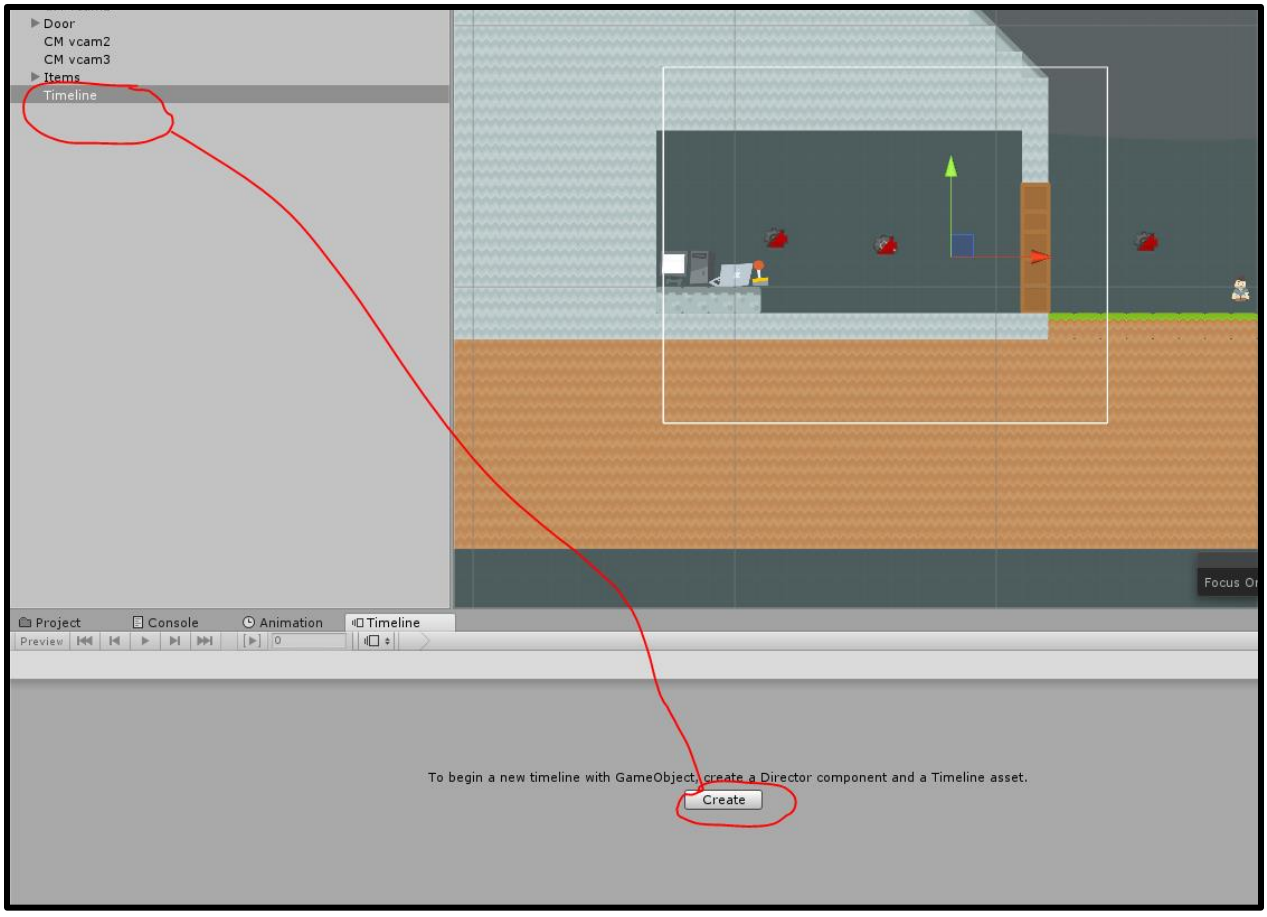


The Timeline Editor

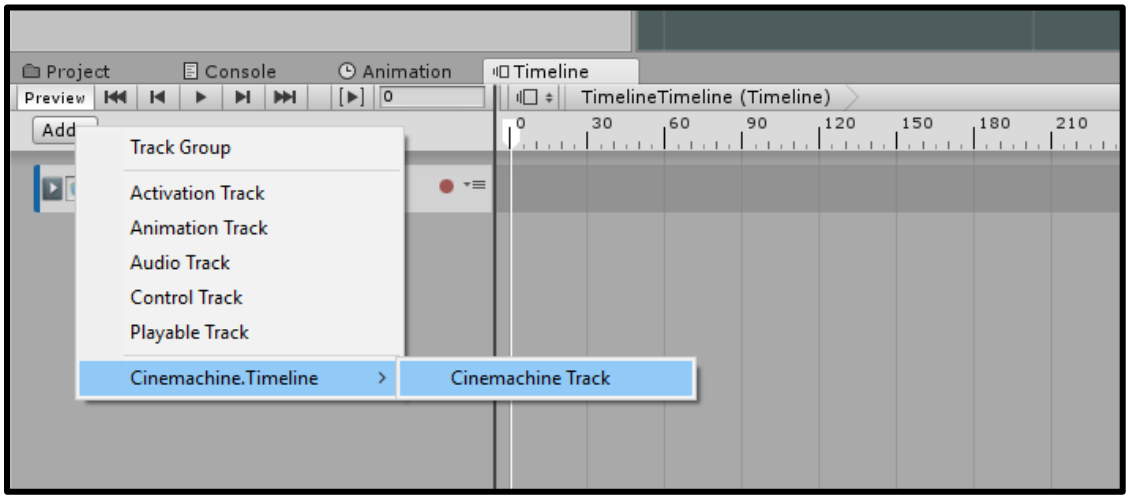
We have our cameras set up, now we just need to tell the camera when to interpolate. In this case, we won't be doing any scripting so that means that we need the Timeline Editor. To bring up the editor go to Window -> Timeline Editor…



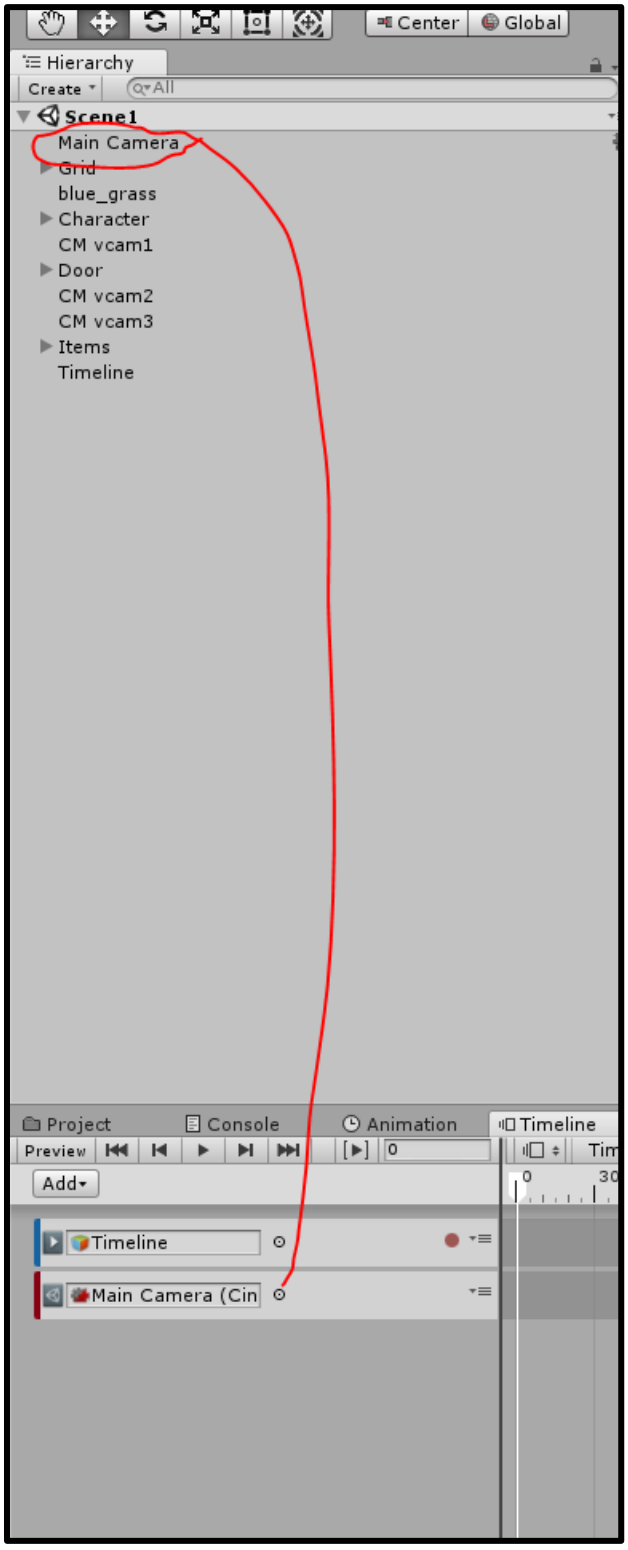
… and then place it in a good spot in your workspace. Now create an empty game object called "Timeline", then click the "Create" button in the Timeline Editor.



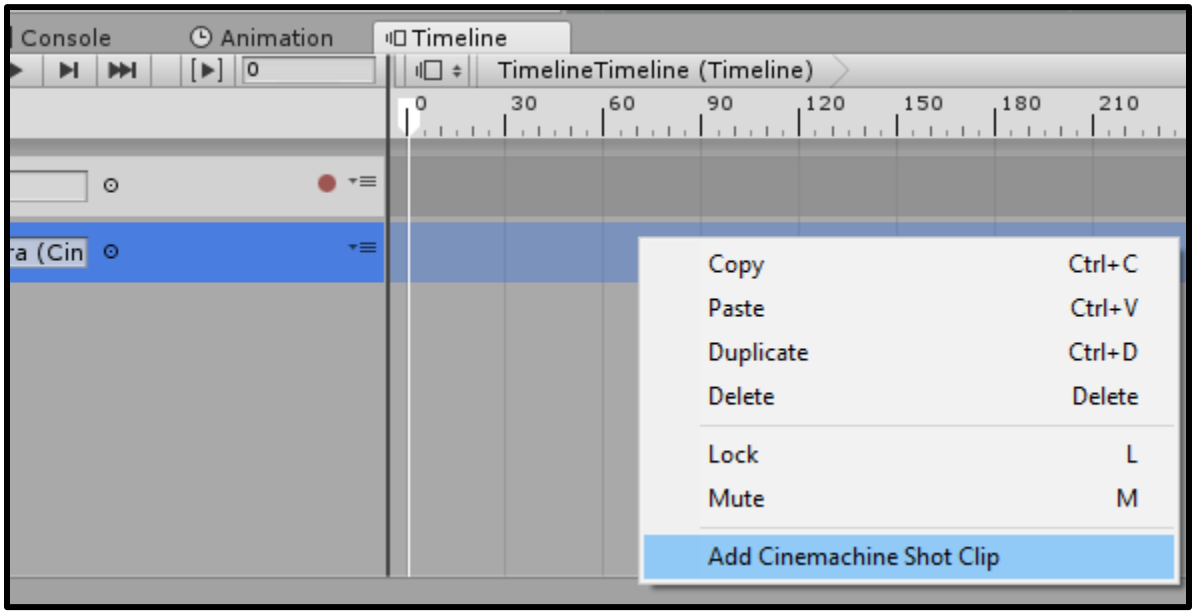
Since we are only going to have one Timeline we can just name it "Timeline" and save it in the root folder. Now click "Add" and go down to "Cinemachine.Timeline" and click "Cinemachine Track".



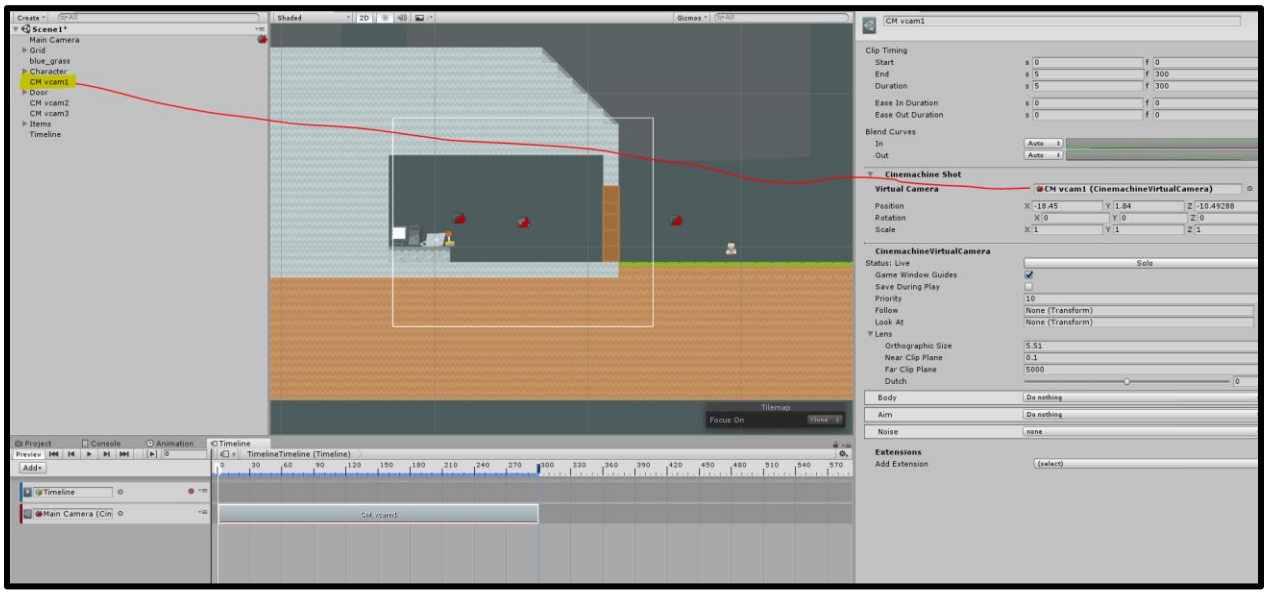
Then drag the main camera into the newly created track.



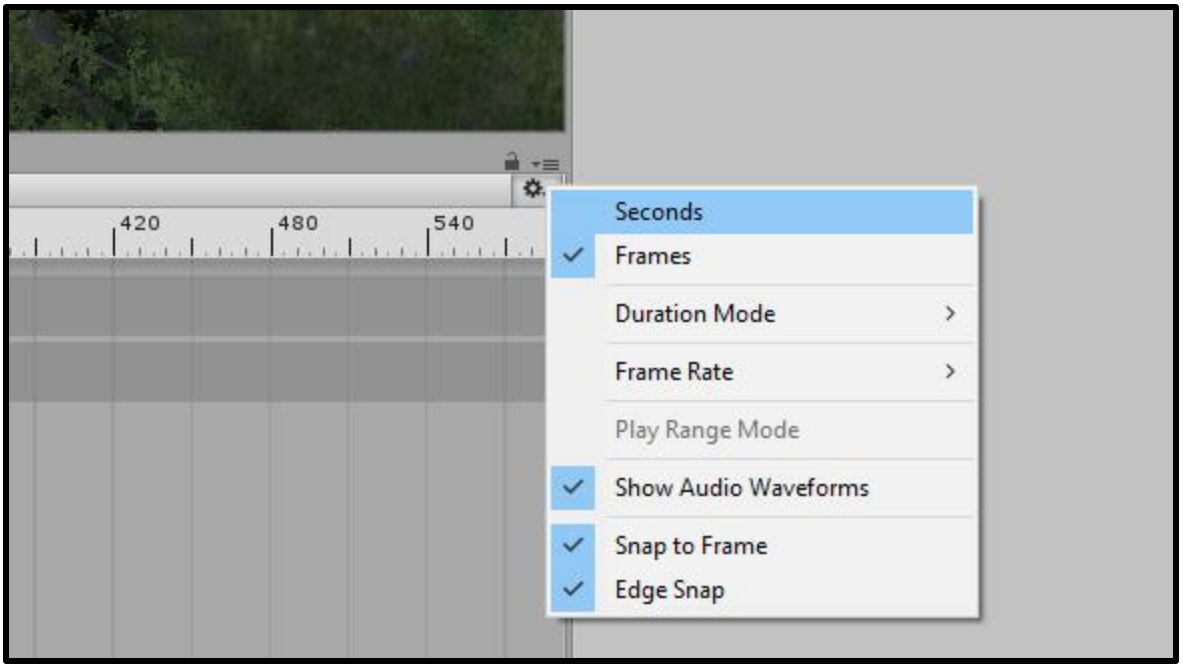
Now if you right-click on our newly created track you can create a new "Cinemachine Shot Clip".



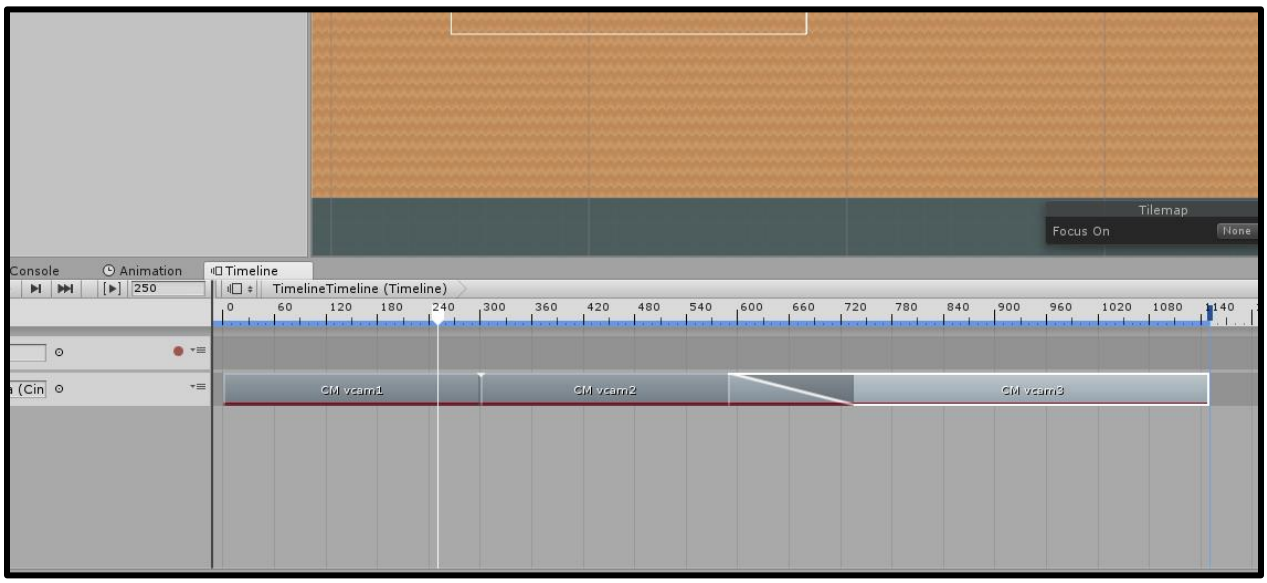
This will tell the main camera how long to stay at this Virtual Camera before going to another one. You need to first assign a camera to this track and then you can start changing the duration.



You can change the timescale to be in seconds and not frames by going to Settings -> Seconds.



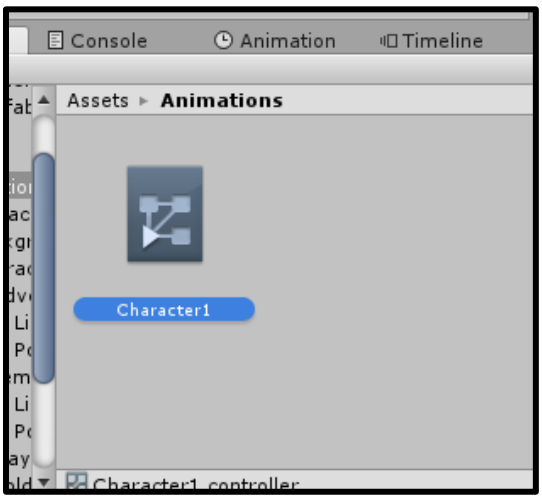
This just makes it easier to tell how long a clip is going to last. Now add all three of our cameras into the Timeline Editor and tweak their lengths to get the look your going after. For the one shot where the camera zooms out, we can actually blend the last two cameras together by dragging them on top of each other. Keep tweaking lengths until you have enough time for each shot. Nothing is set in stone though so if you need to come back and change the lengths feel free to do so.



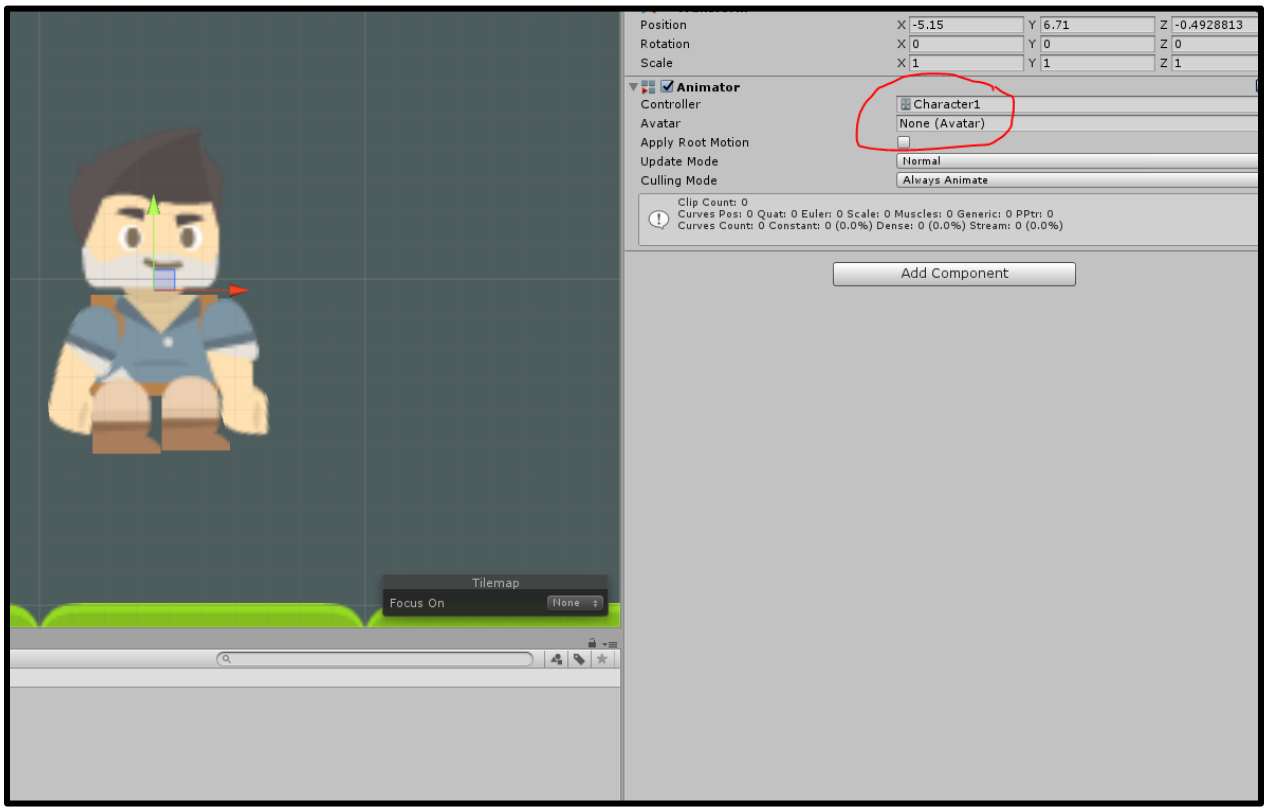
Once you are done you can hit play and watch what happens!

Animating the Character

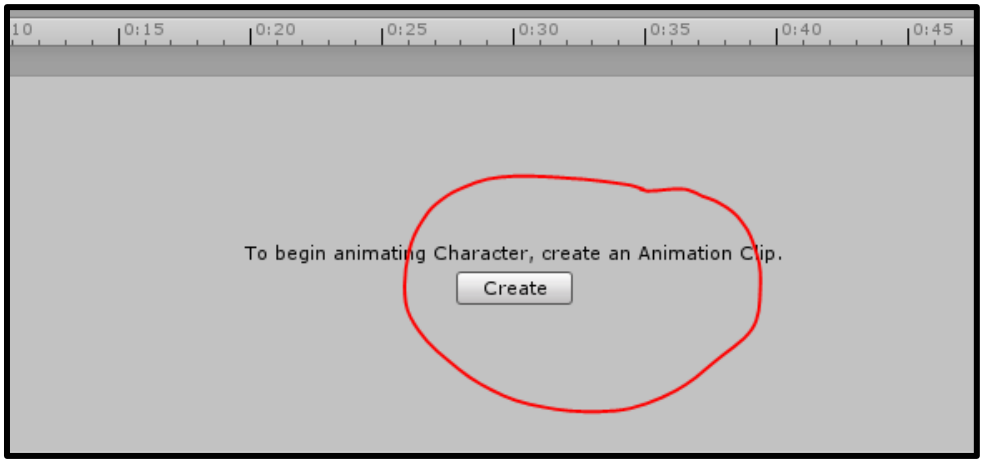
Let's start by animating the actions we are going to need. Create a new folder called "Animations" and create a new "Animator Controller" called "Character1".



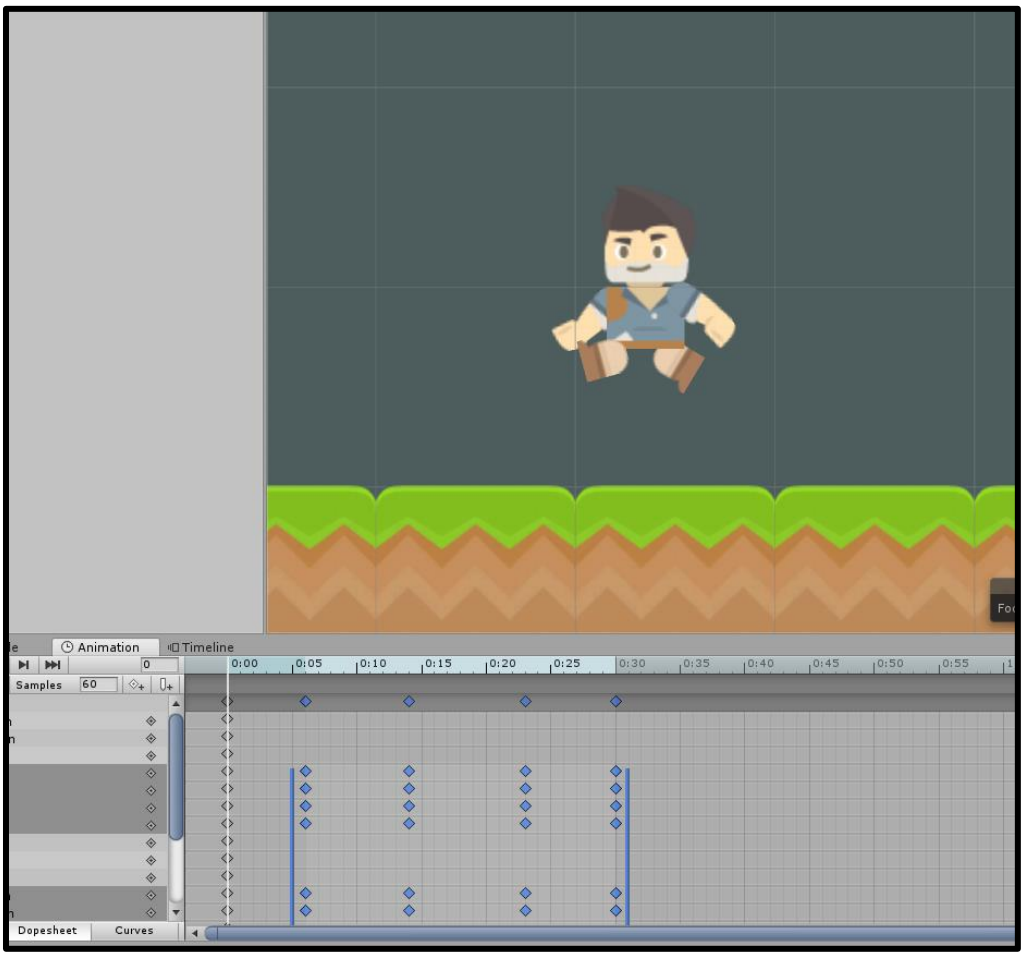
Then go to our Character and add an Animator component and add the newly created Animator Controller into the "Controller" field on the Animator component.

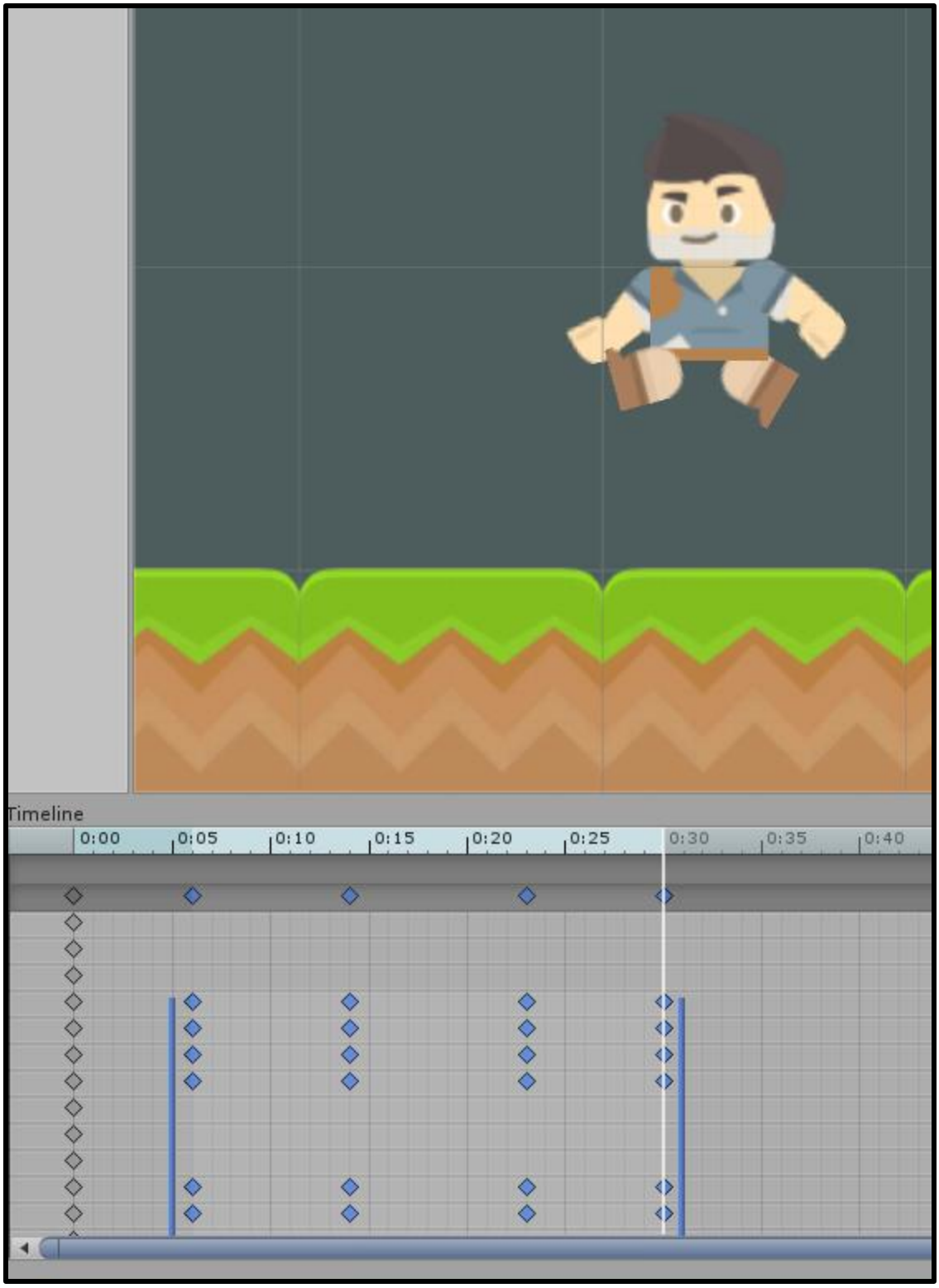


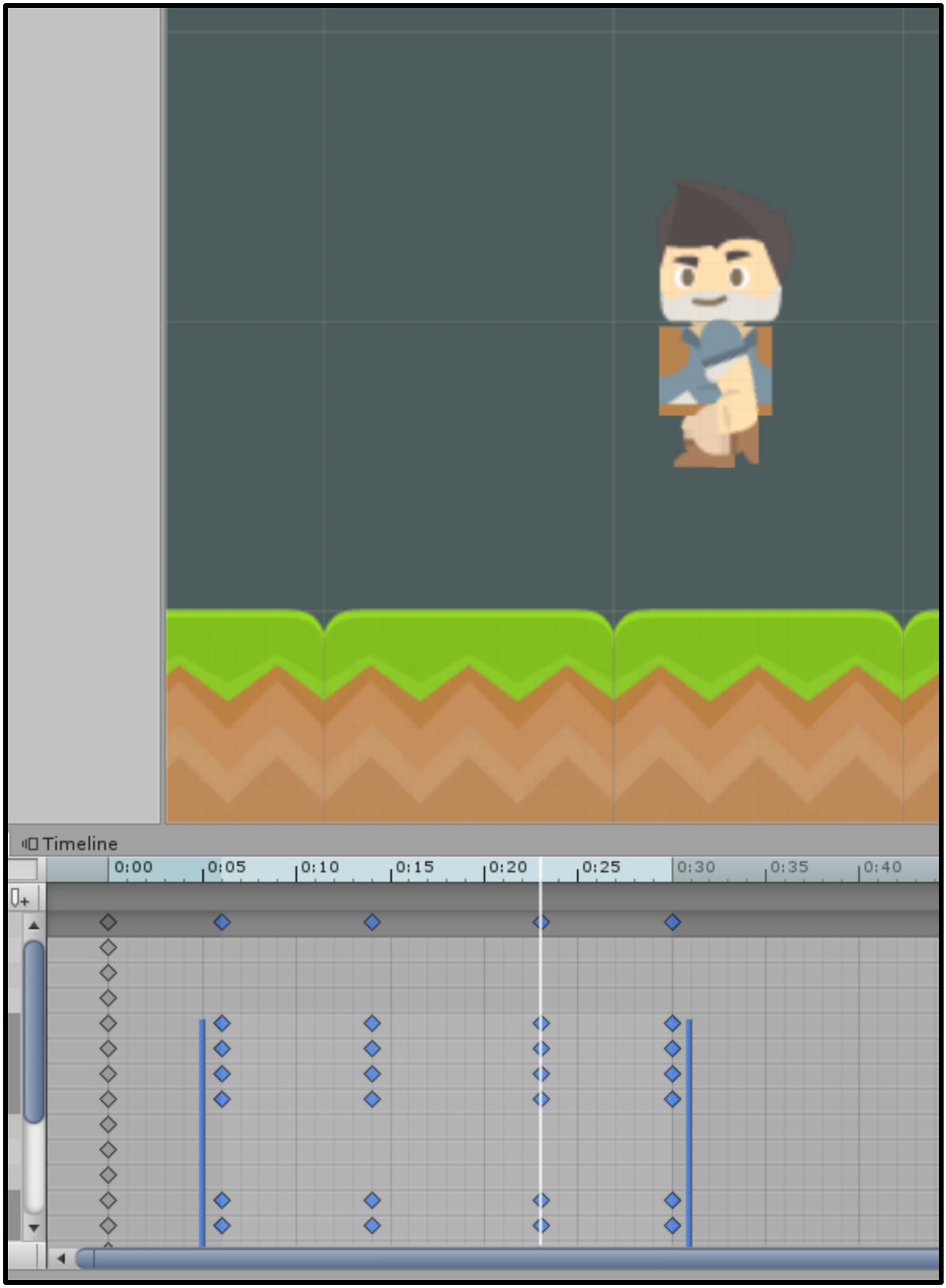
Now go to the Animation window and click create.

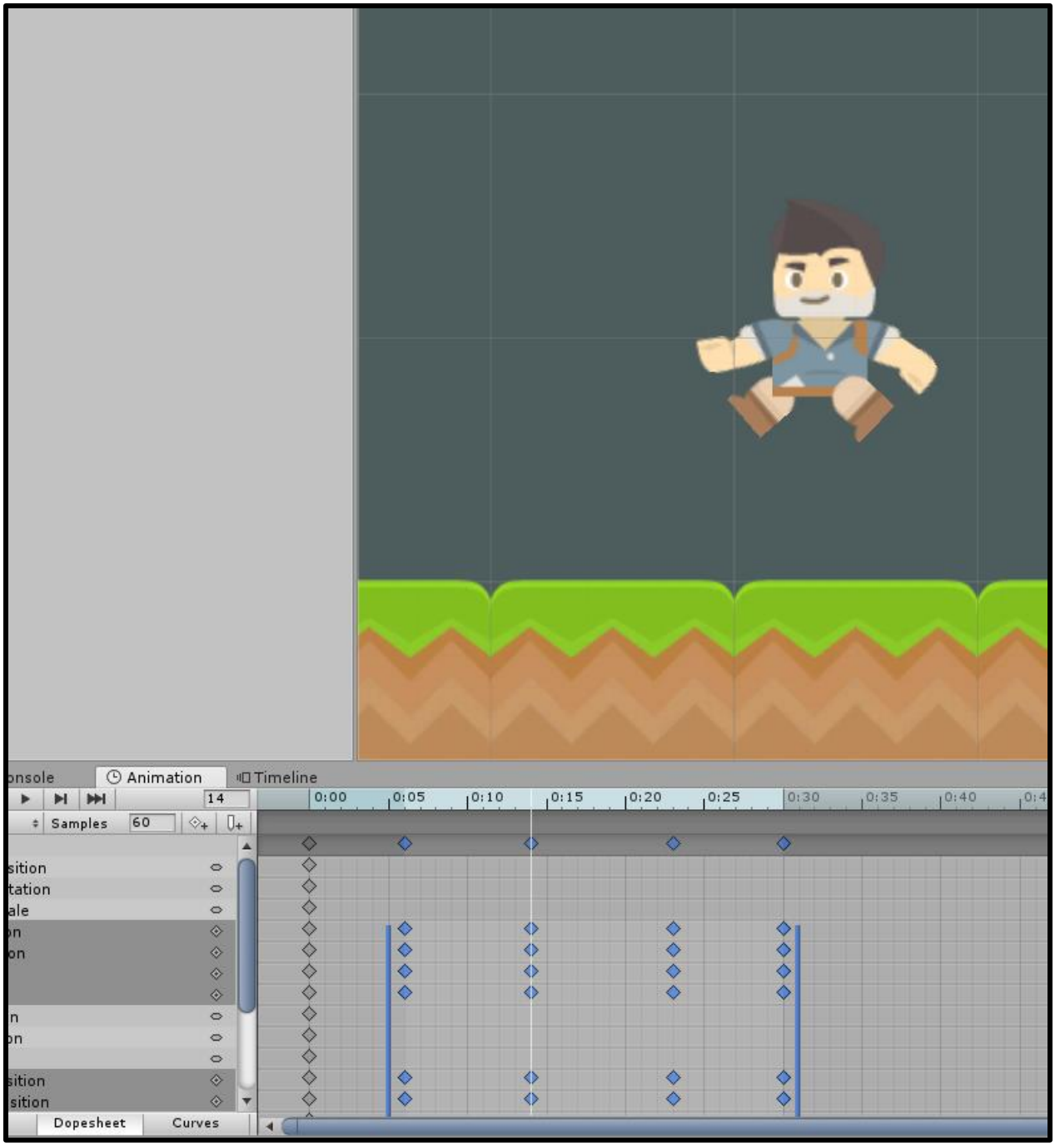


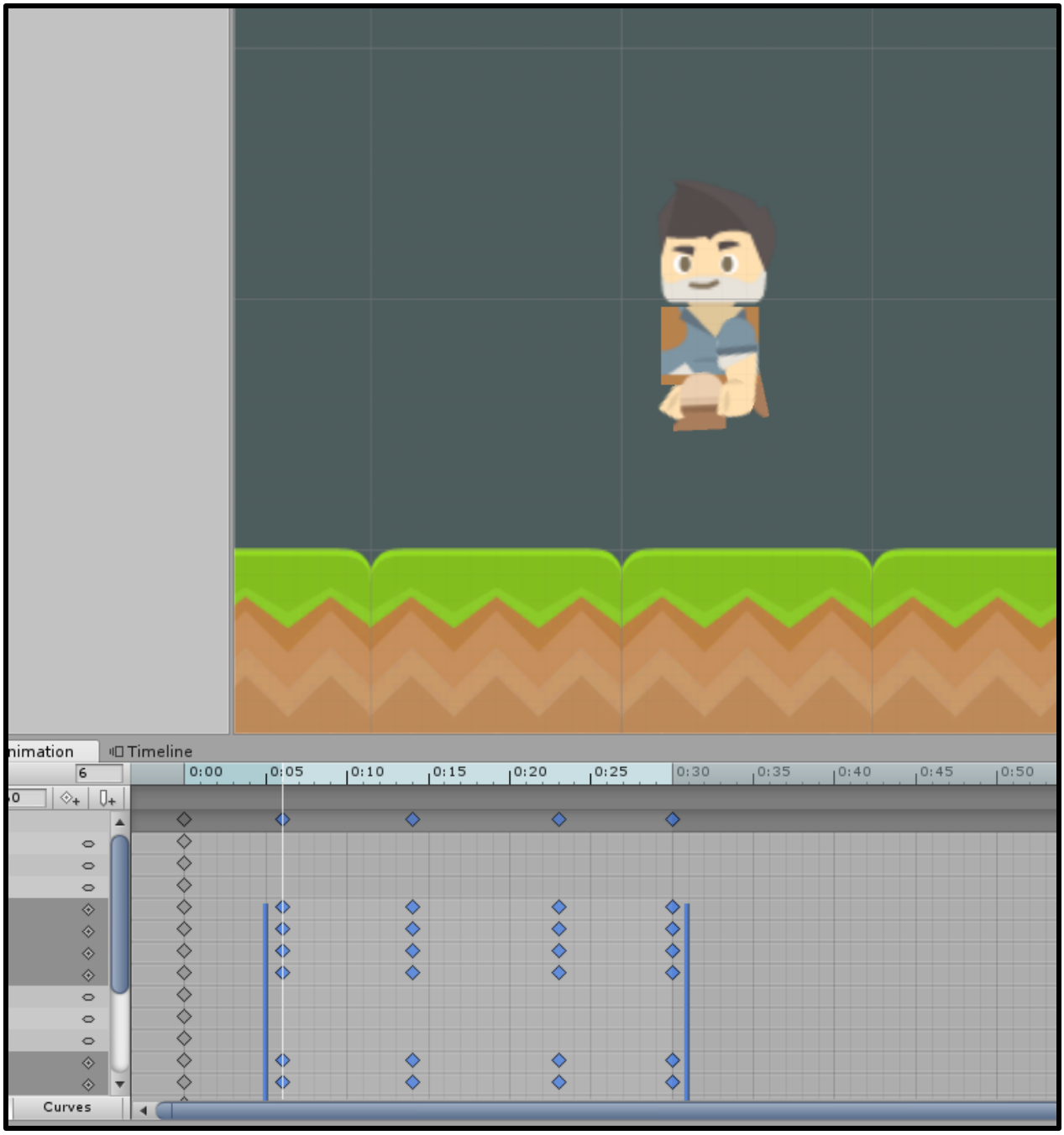
Call it "running" and save it the Animations folder. Animating can be kind of a daunting task but fear not! Here are the poses from my animation:



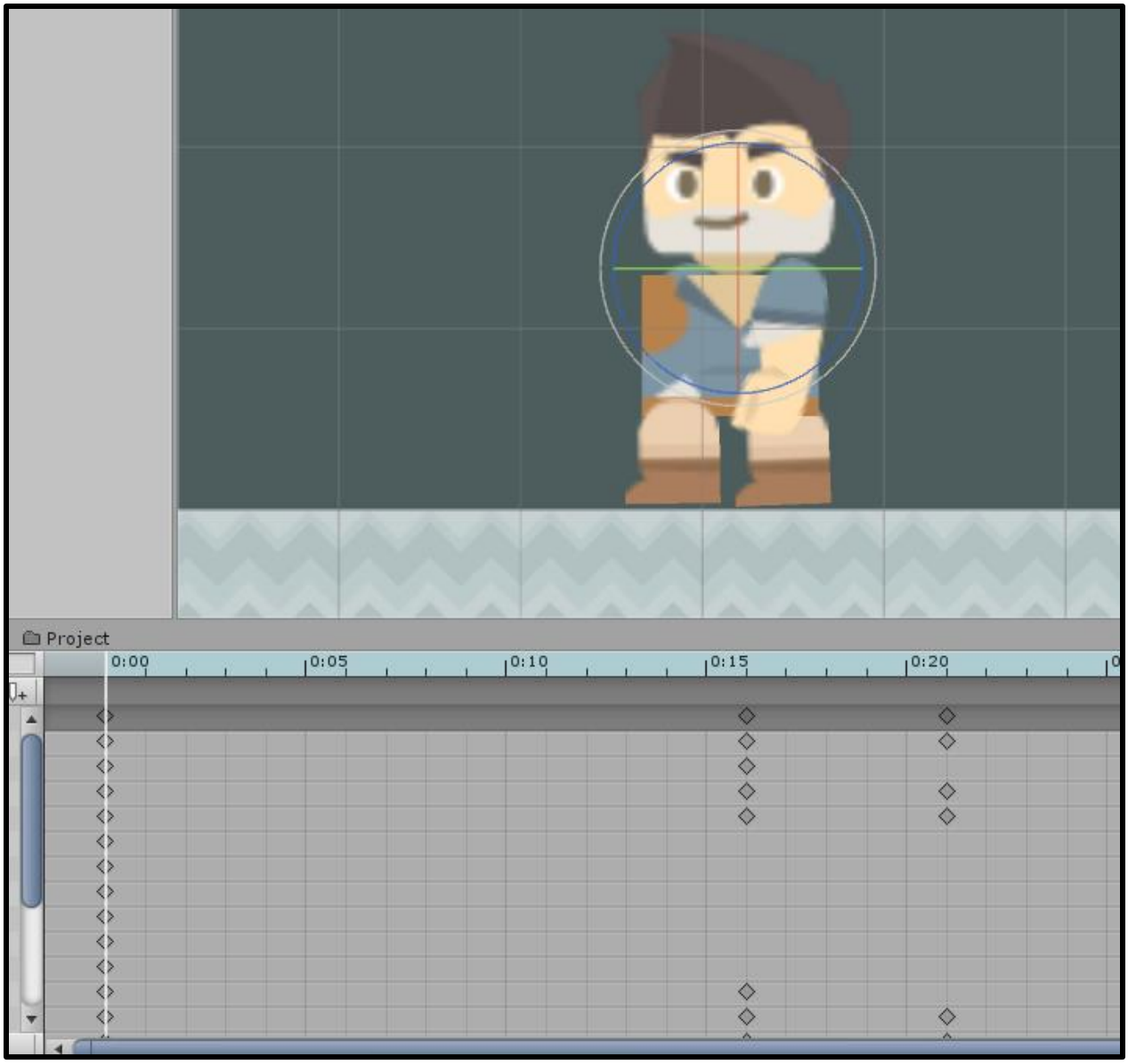


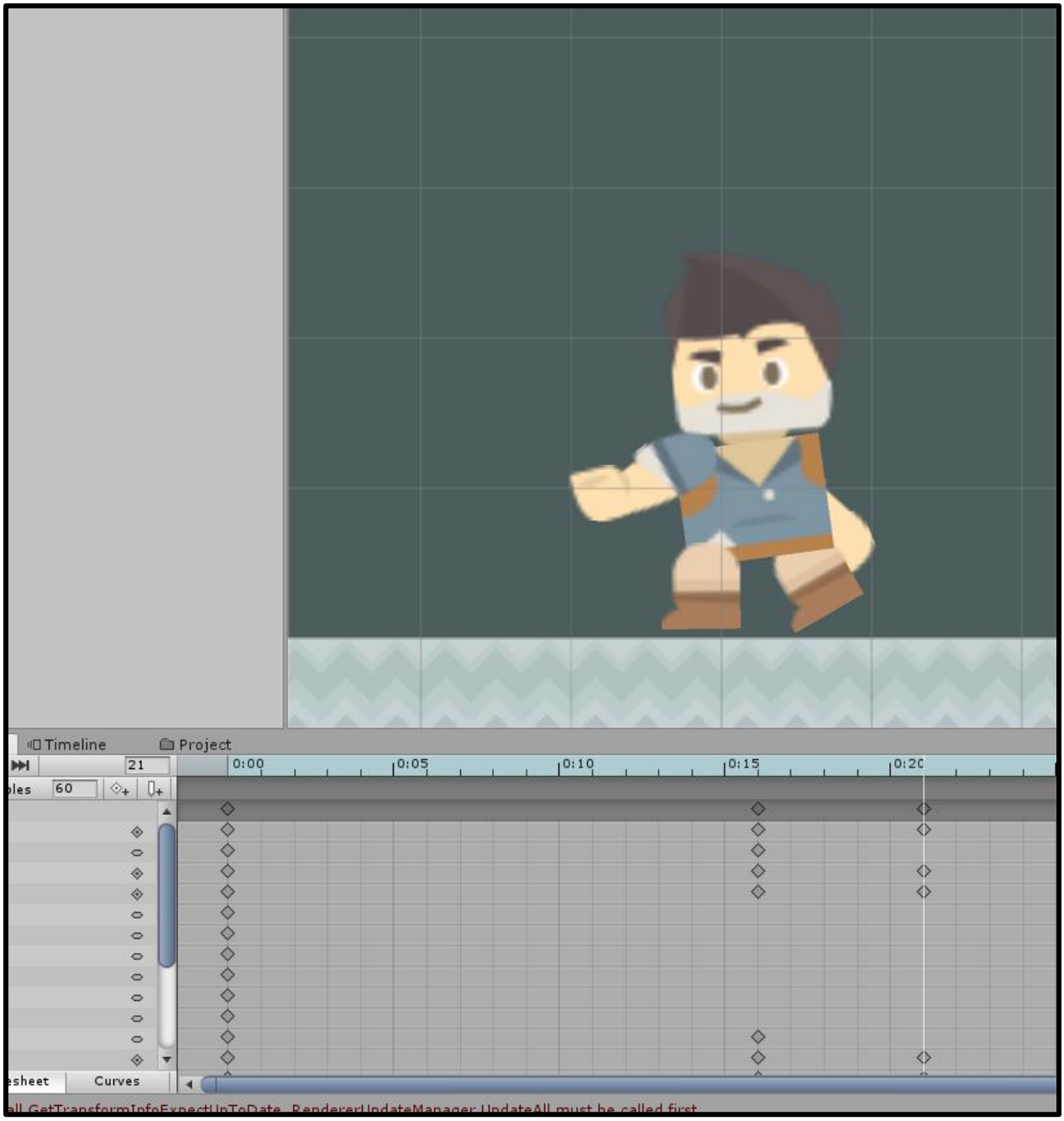


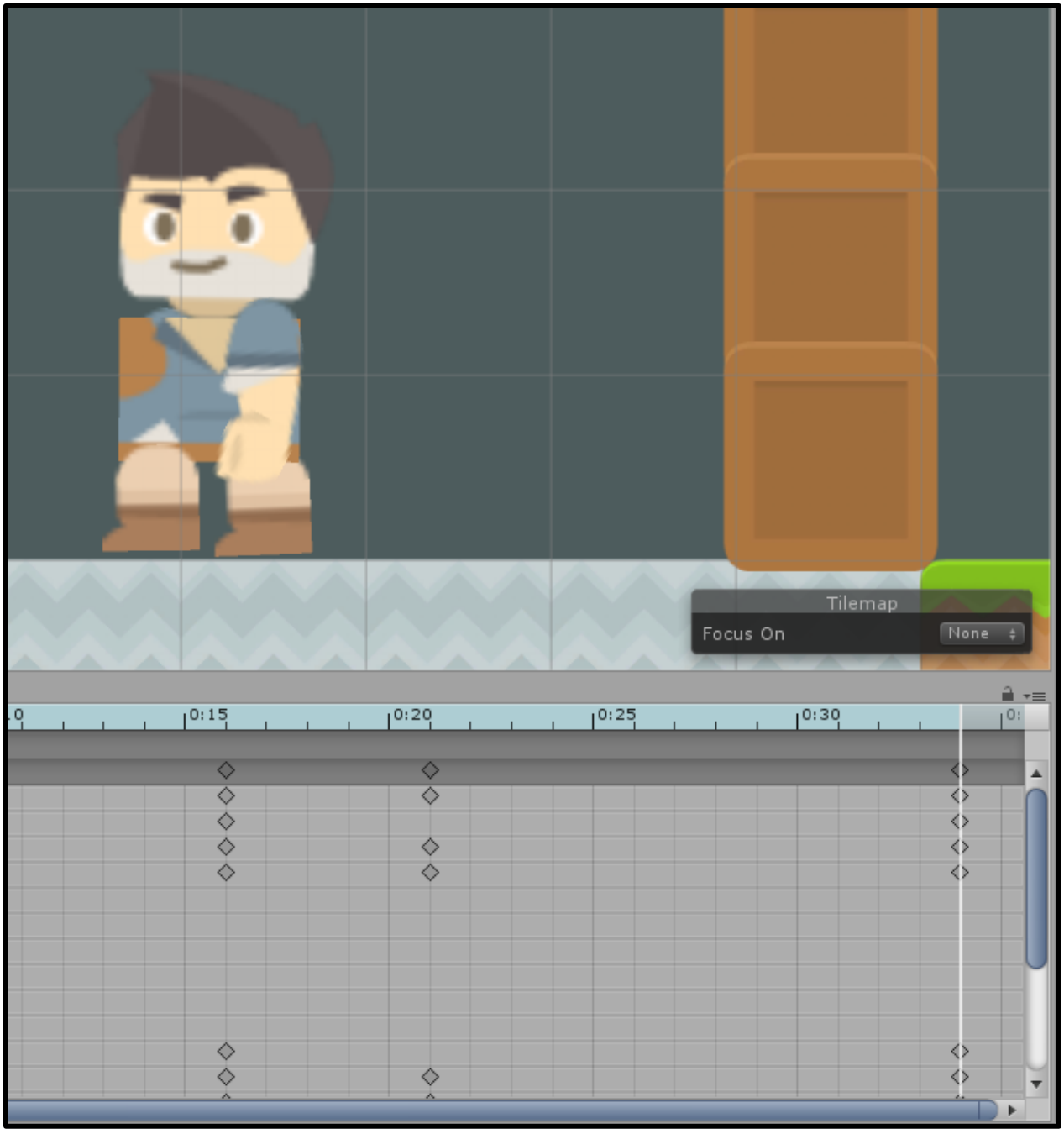


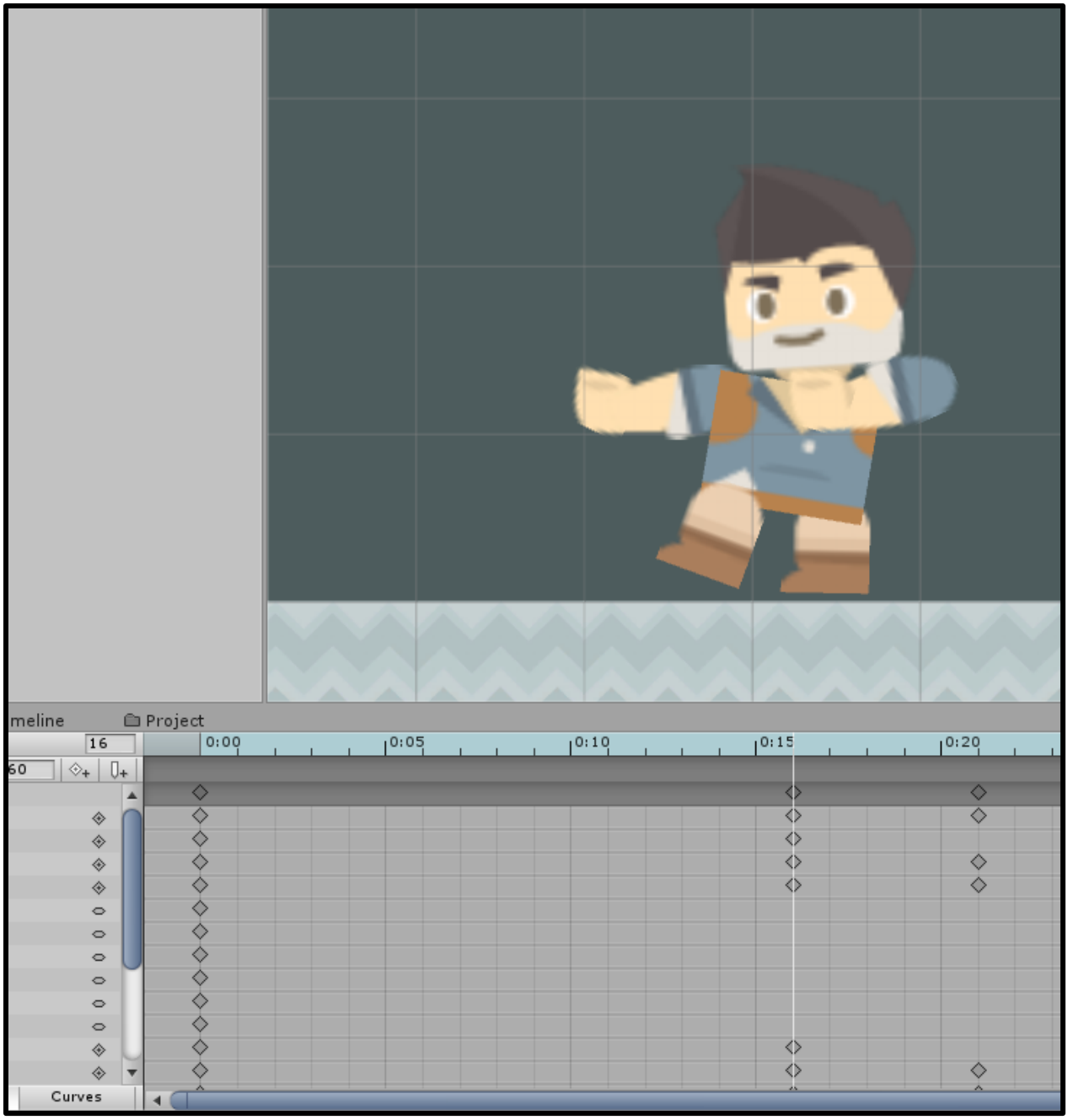


All you have to do is hit record and then move the limbs to match the pose. Each pose should be about 6 frames apart. The character is running in place so don't move it either left or right. Then create a new clip called "Punching". This one can be kind of tricky so here are my poses as well:





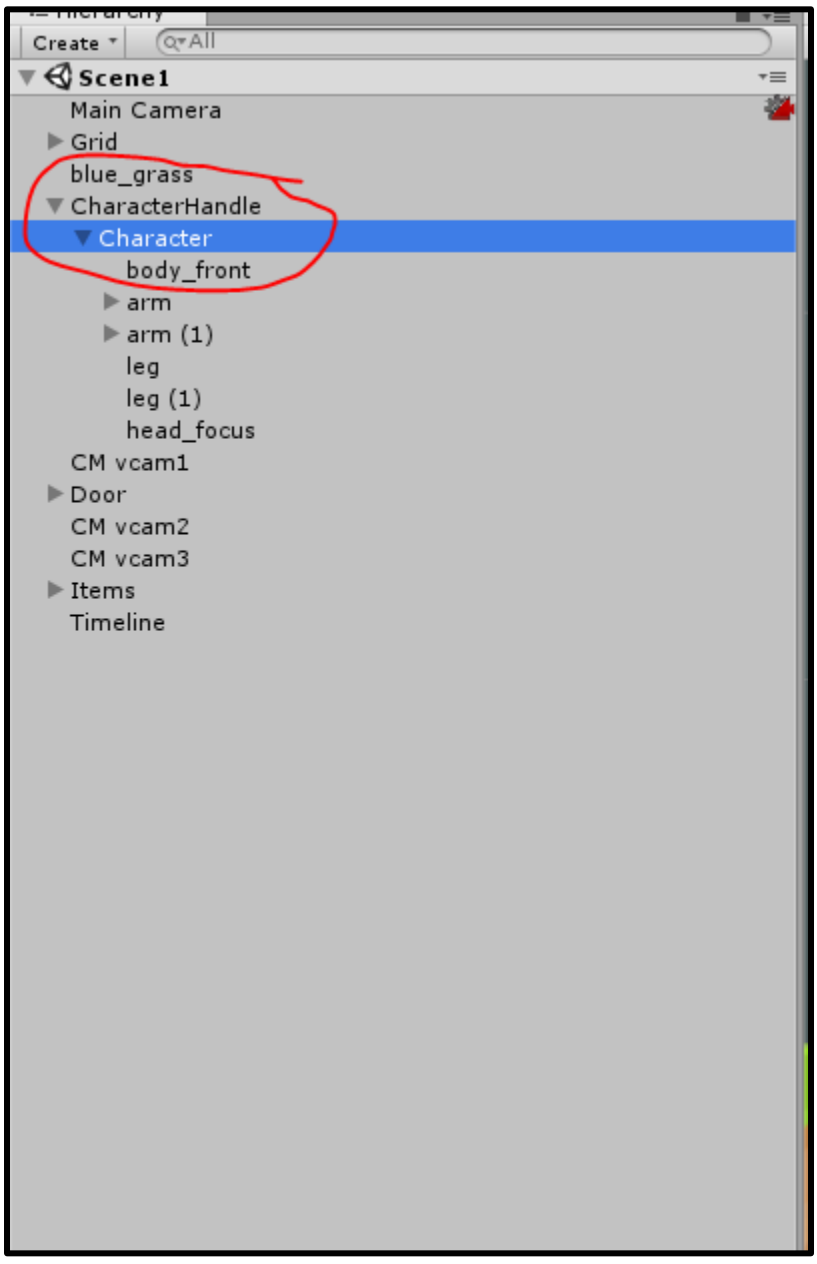




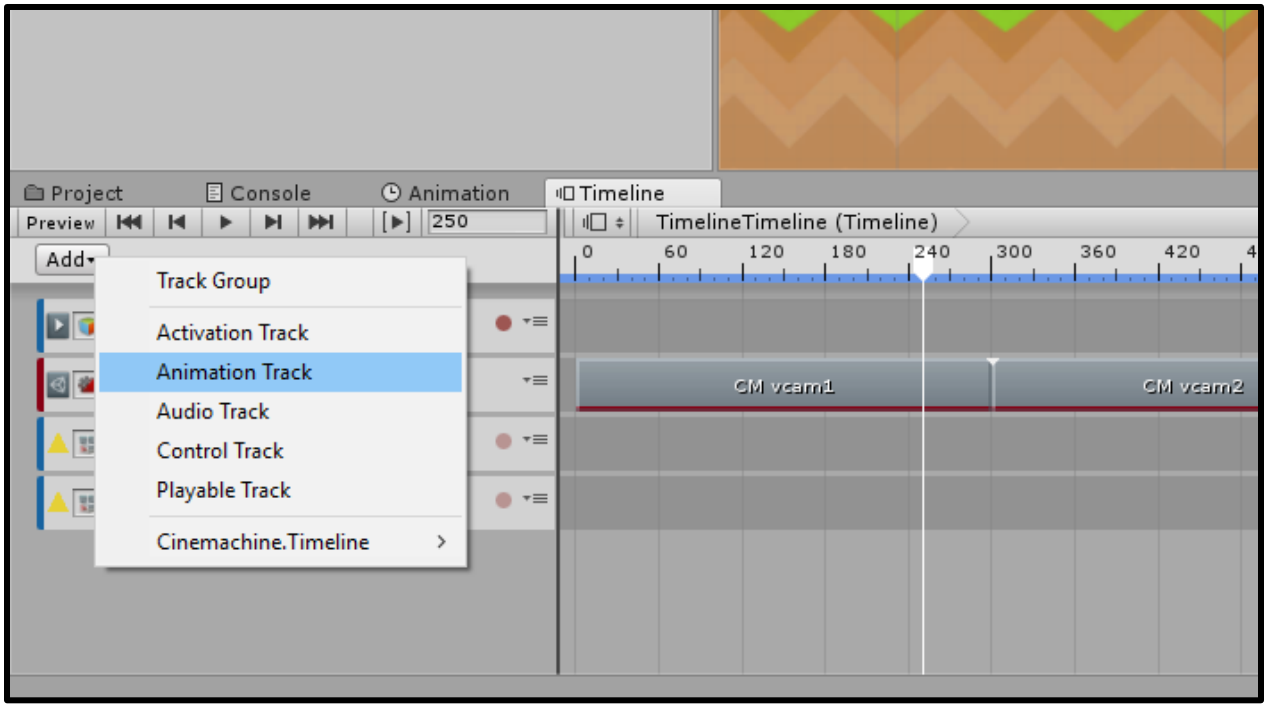
The typing as well shouldn't be too hard. You'll see that the typing is just the arms moving up and down. It is very important that you animate the character in whatever direction it is facing. If you animate him punching in one direction but have him running in the other direction it will look weird. Also, if the record button is greyed out hit save and then try.

Putting it all together

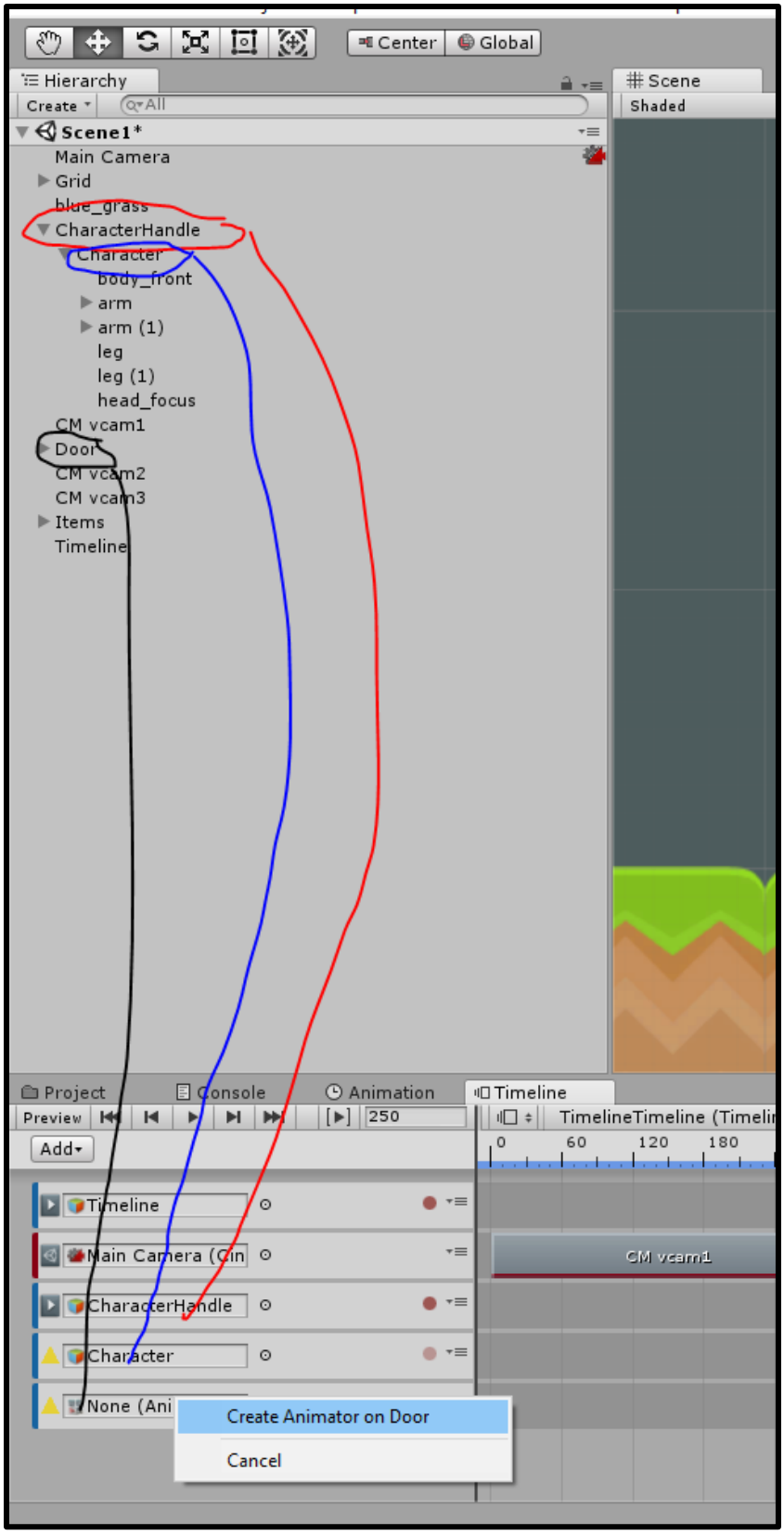
Create an empty game object and call it "Character Handle" and drag our Character into this object.



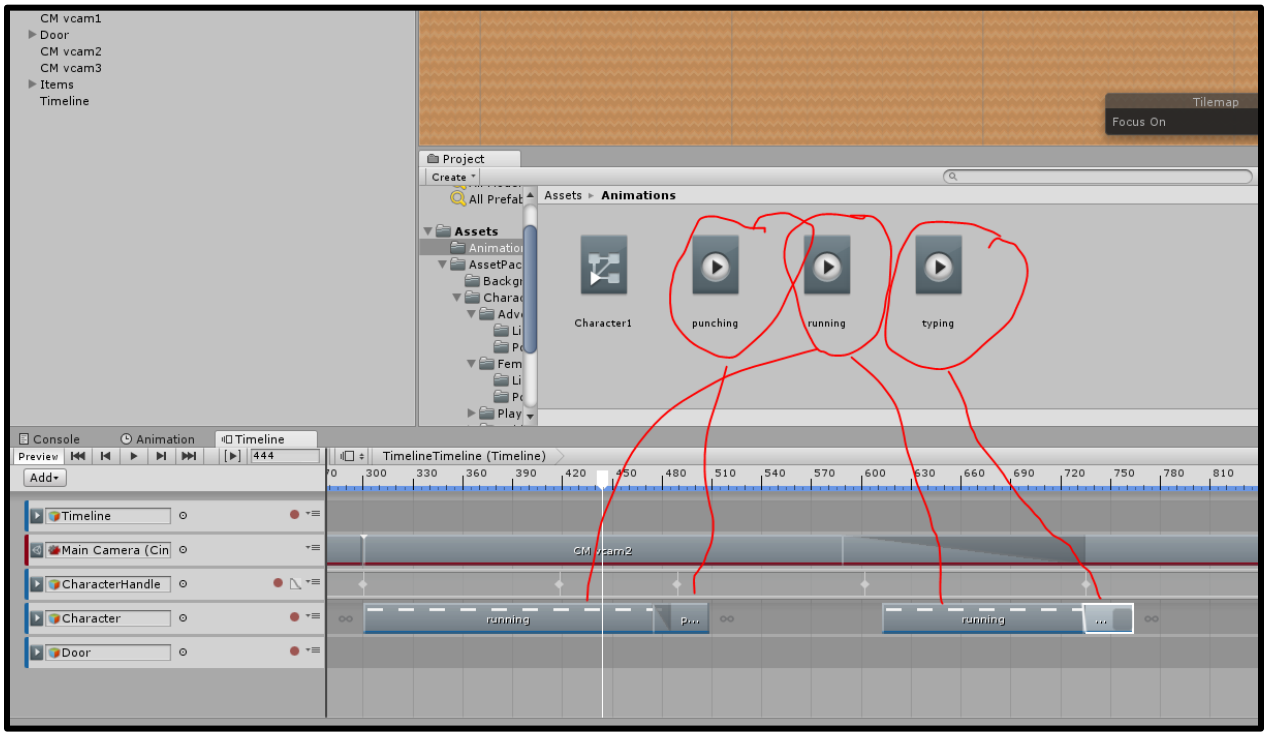
This is where we will be animating the transform of our character. If we did the transforms and the animations on the same object, I have found that it makes weird results. Now create at least three animation tracks in the Timeline Editor.



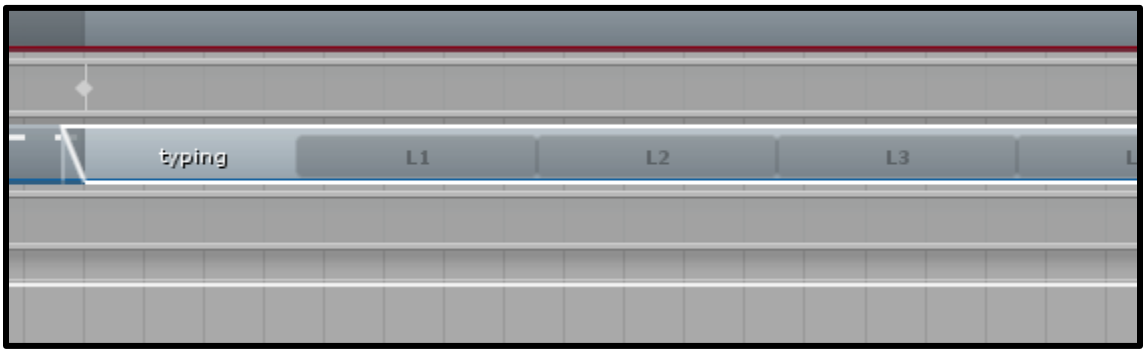
These are for the action animations of our character, the transform animations on our character, and the door getting knocked down (if your door is made of more than one crate then make all of the crates a child of one game object). Assign each track their respective objects. It will ask you if you would like to create an animator component on some of them to which we would reply with a yes.



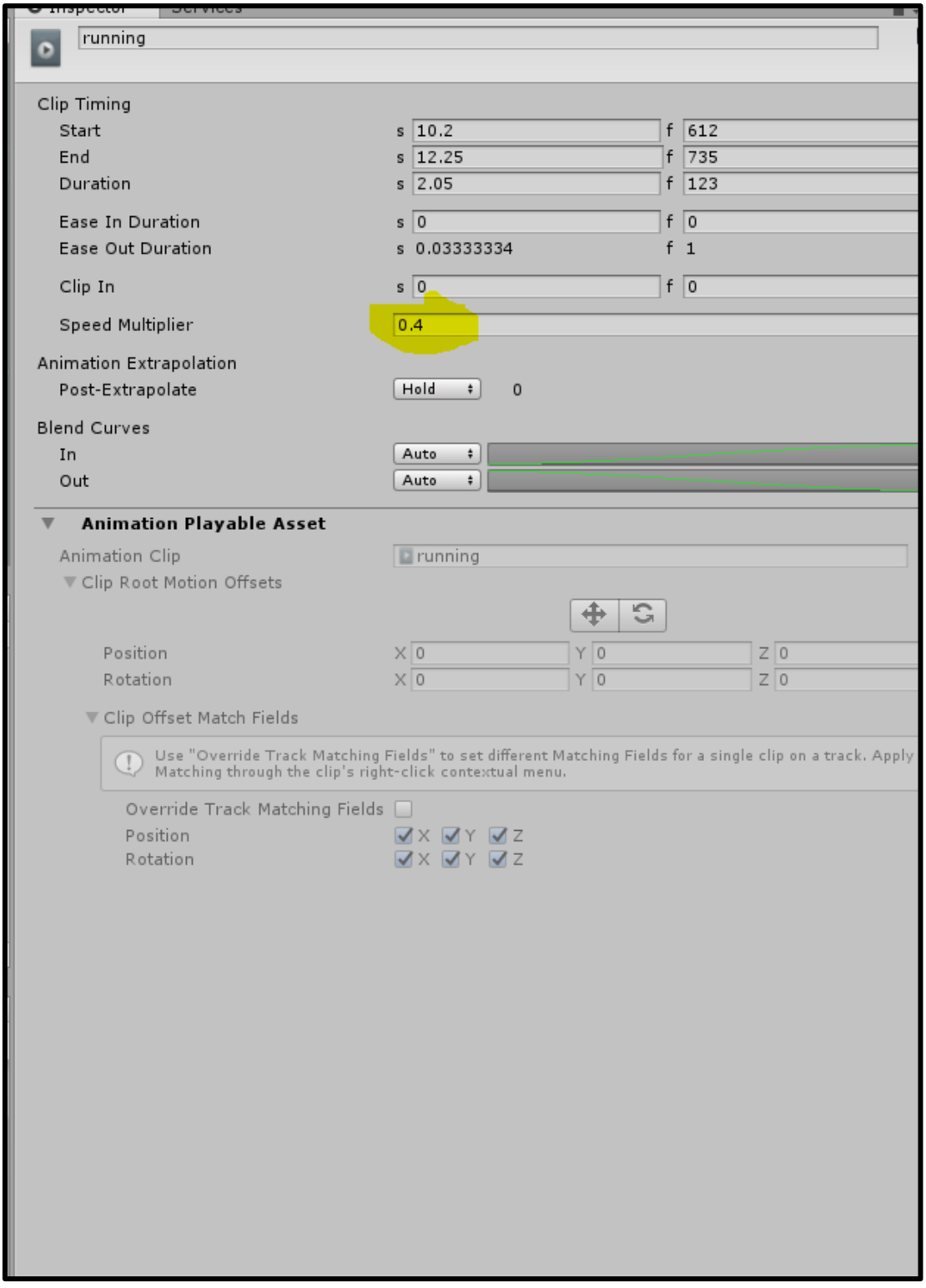
Now animate the character's transform by clicking the record button. On the final shot, the character shouldn't leave the frame. Try to leave enough time for each action, that way the audience won't feel that it is too rushed. When the camera is zooming out, I have found it is best to have the character enter the building. It makes a nice composition with the character on one side of the screen and the computers on the other. Now we need to add our character's animations. The best way to do this is to drag your Project tab above you Timeline Editor. Now go to the Animations folder and drag in the animations into their respective positions (i.e "running" when the character is running).



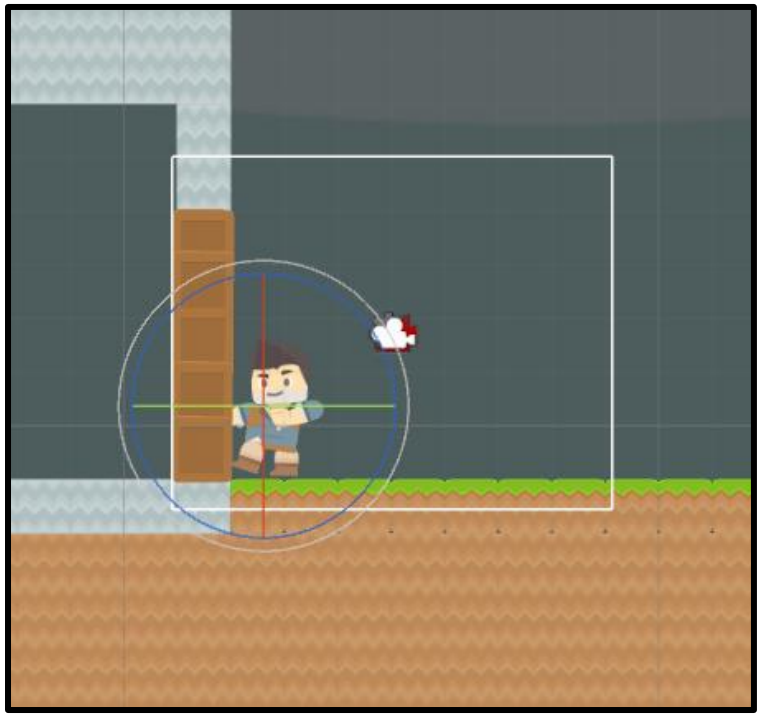
You can also drag these clips on top of each other to blend them. And you can also loop them by dragging them out.

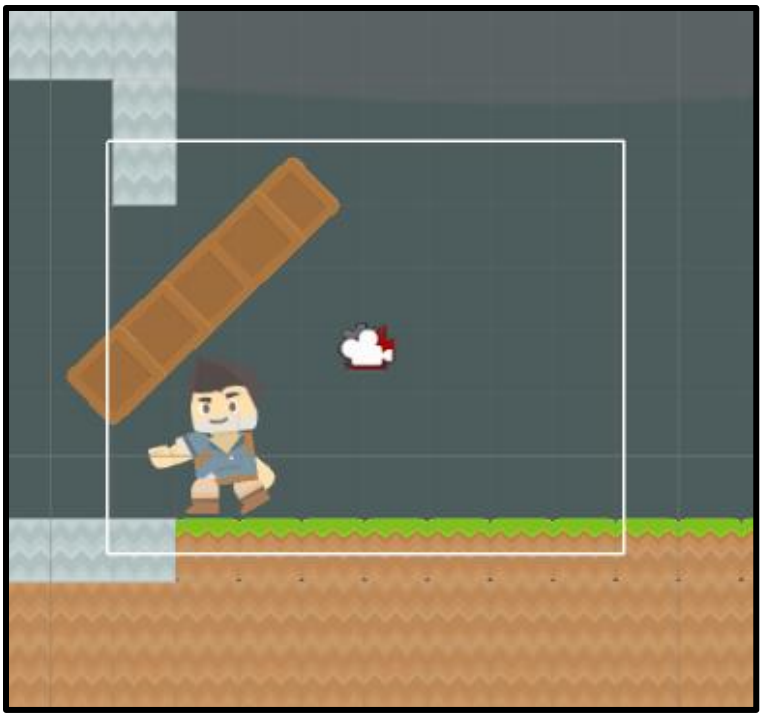


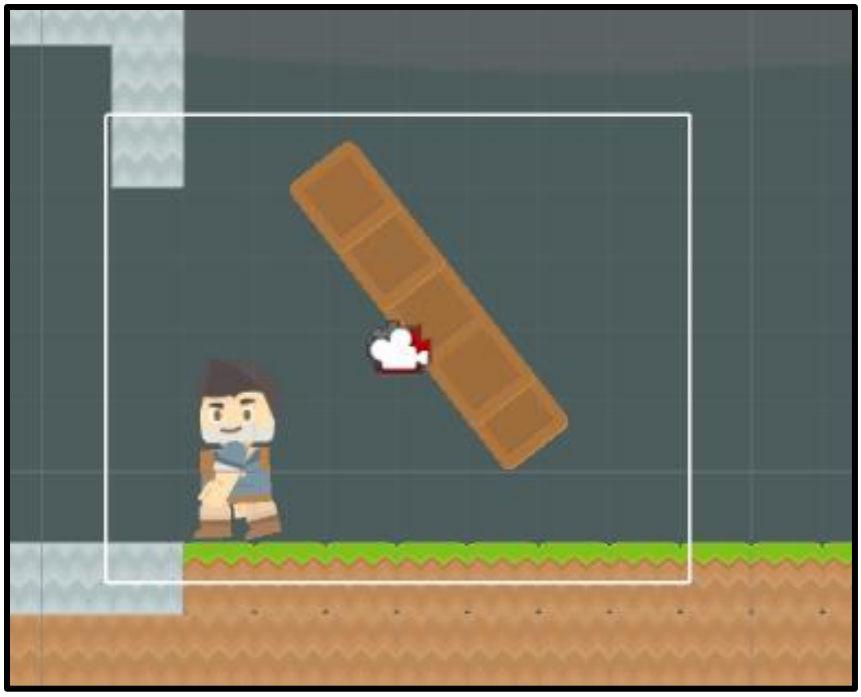
If you animated like I did then you may need to decrease the speed of the "running" clip.

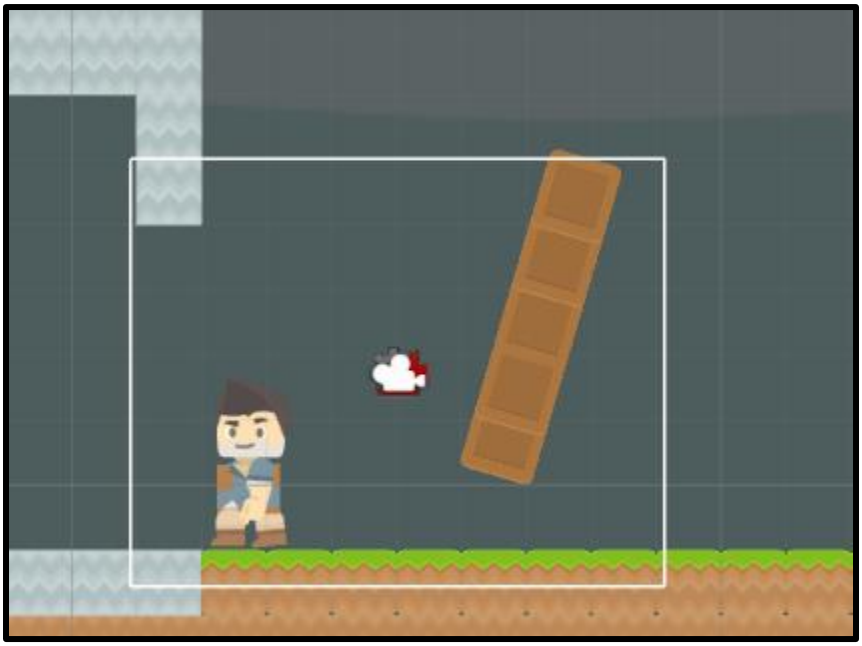


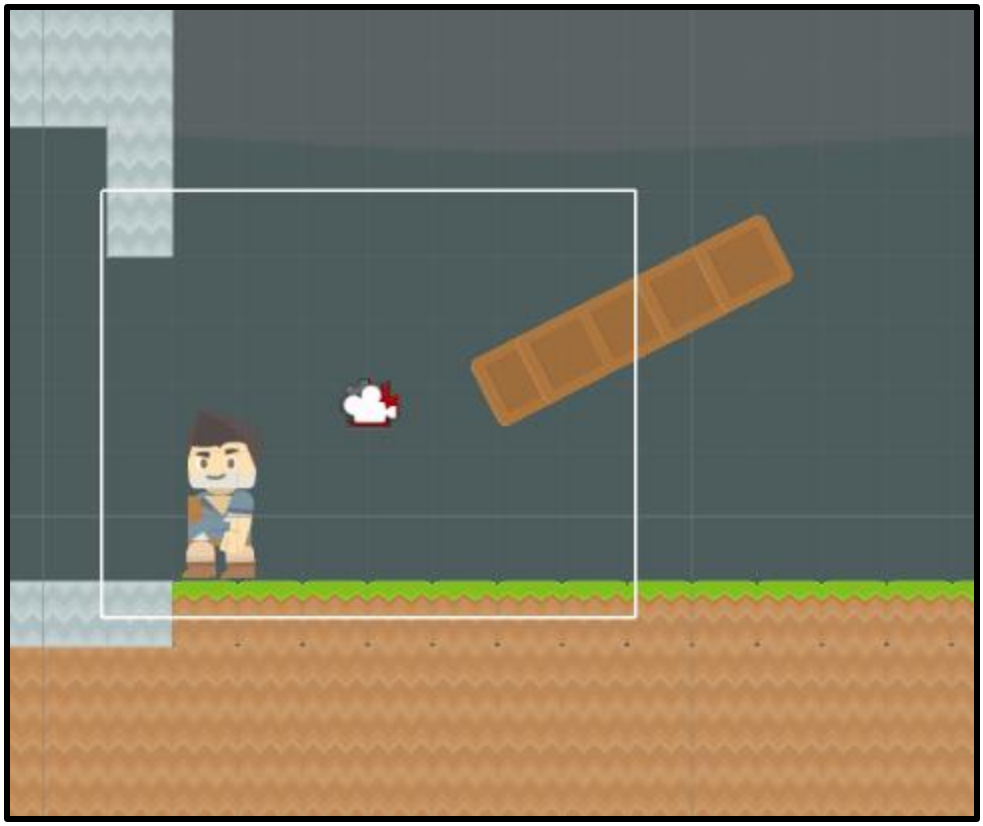
Once you have the character completely done we now only have to animate the door getting knocked down. This can be kind of tricky. If it falls forward it may clutter the inside. But it falling backward may look kind of weird. It all depends on how your building is set up. You can always have the door just fall of the world entirely. It's all up to you. I decided to do this to mine:

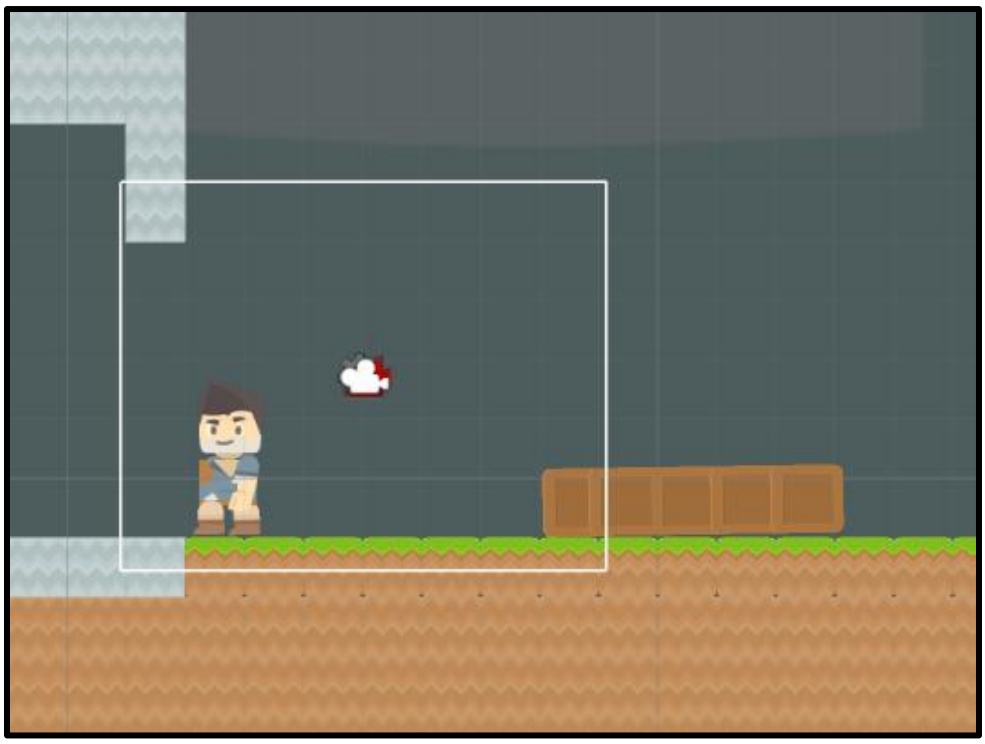












Summary

Congratulations on getting to the end of this tutorial! I hope you have enjoyed it and that it has been helpful. Hopefully, now you understand Cinemachine for 2D well enough that you can start implementing it into your own projects. The example footage came from a game that I made called "Chaos 2D". You can download it now from Amazon, Google Play, and the Apple app store.

Keep making great games!