

5

Invite Your Characters to Unity

This chapter is all about taking your modeled, rigged, and animated characters from your 3D modeling software and importing them into Unity.

In this chapter, we'll cover the following topics:

- Exporting character models from a 3D software package
- Configuring generic and humanoid animation types
- Creating a character avatar
- Getting your character ready for the Mecanim

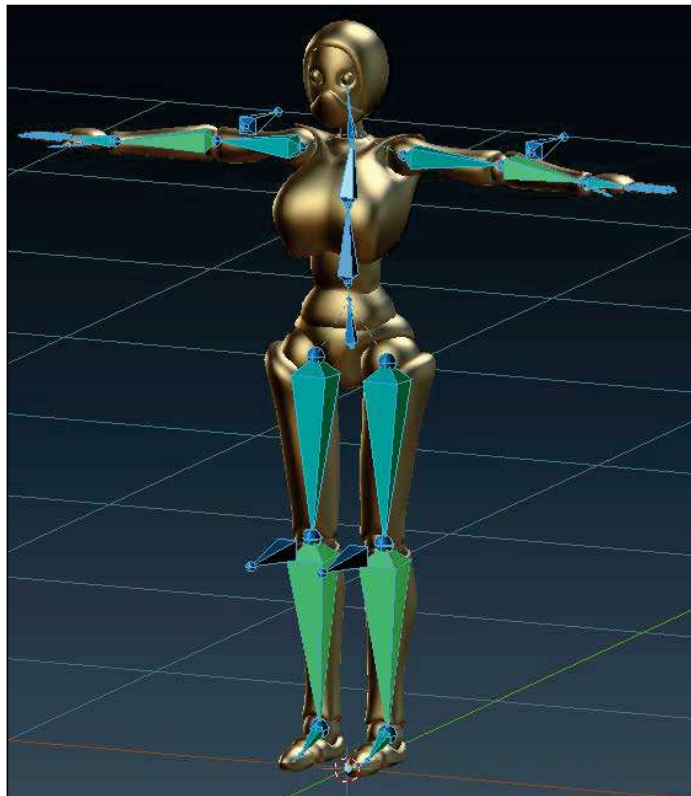
By the end of this chapter, you will learn about the character import process, its benefits, and how to deal with any issues. The example in this book features a specific 3D modeling software package; however, even though the steps might differ, the general process is the same, no matter what you choose to go with.

Get your character ready

I have to tell you – the way you go about rigging and animating your character can, and most likely will, vary from the example shown here. It may depend on the software you are using, your needs, or your devotion to certain standards. That being said, most paths will allow you to bake animations and export them in the `.fbx` format. You don't have to do that though! If the Unity supports the native file extension of your software of choice (at the moment Unity recognizes the 3D Max, Maya, and Blender file formats) you should be able to import it as is and use it in Unity (I am personally not familiar with every software program there is for animation so I can't say for sure that you won't encounter any problems along the way). However, by baking animations and exporting them in more or less generic formats, we will be able to get on the same page regardless of where we've started from.

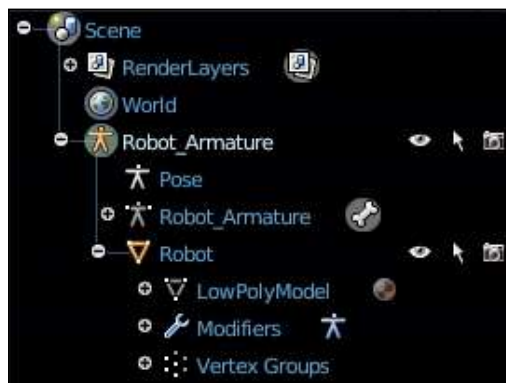
Exporting from Blender

Our character is fully modeled, rigged, skinned, and animated in Blender.



Model overview

This file contains all of the information about our character, there are no references of any kind, and all animations are done on a single timeline.



The file is named `Robot.fbx` and can be found inside the `Chapter 5` folder of the complimentary files.

Apart from the geometry and the skeleton, it also contains a pair of IK handles for legs and arms. In the end, we have a very basic character ready to be used for video games. You can try to challenge yourself and bring a high-poly model with a fancy rig and a complex set of animations, but that will only equate to doing extra tweaking in Unity further down the line and there are certain limitations posed by Unity that will encourage you to keep it as optimal as possible.


Let's talk about the skeleton. In order for your character to be working in Unity, you don't have to go out of your way and force a certain bone topology. However, if you are building a humanoid, it should have a bone topology similar to humanoids—no three legged/ four armed creatures can actually be considered humanoids by Unity standards, but can still be imported and used.

The skeleton structure of our character is not a template by any stretch of the imagination and you don't have to follow it; work with what you feel comfortable with and I'll show you how to communicate your preference to Unity later in this chapter.

Exporting as FBX

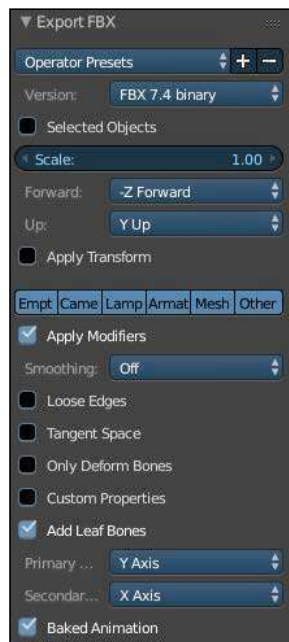
In this section we are going to look into how to export the character into FBX using Blender, a `.blend` file can be found in the same folder as the `Robot.fbx`. This character is ready to be exported. There are a few general things to consider before we get it into Unity:

- Character stands solidly on the ground at 0,0,0 coordinates
- Character is scaled properly
- Check your T-pose to make sure that the palms are facing the ground
- Check your normals to make sure they are facing the right direction

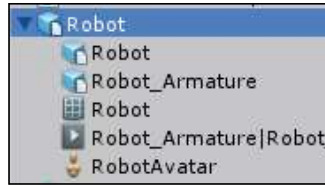
[ 1 Unity unit equals to 1 meter; for better optimization and smoother import make sure to scale your character in the 3D app, accordingly.]

Unity supports the `.blend` file extension; therefore, we can simply import it directly or export it to the `.fbx` file format in the following manner:

1. In Blender, navigate to **File | Export | FBX (Import-Export:FBX format add-on needs to be enabled)**.
2. In the options, check the **Baked Animation** box.
3. Hit **Export**.



Your file can now be safely imported into Unity.

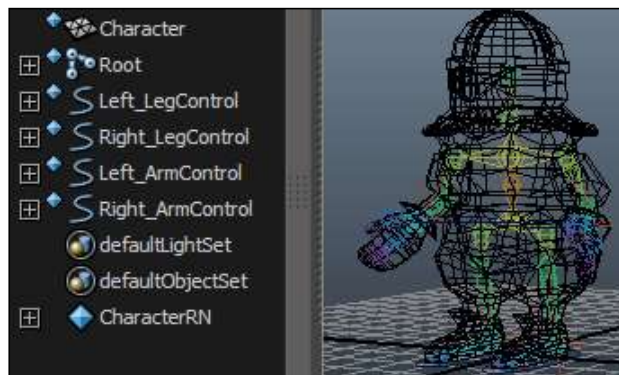


You don't have to get rid of curve controls after baking animations for exporting; they won't cause any problems and won't be visible in the game.

Importing referenced animations

For those of you who use referenced animations – Unity's got you covered.

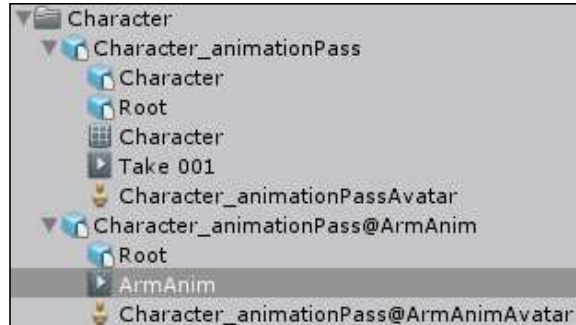
Here is another example of a file that is referencing a different file that contains geometry and rig, but no animations. This reference file, on the other hand, contains our arm animation and a reference, nothing more. This is a different character and this time in Maya.



Here is the process of getting this animation into Unity assuming that you've already imported your referenced model in the .fbx format:

1. Bake animation by navigating to **Edit | Keys | Bake simulation**.
2. Select character's entire skeleton (every node).
3. Export that in .fbx format by going **File | Export Selected** (you'll only need your skeleton, nothing else).
4. Name the file `Character_animationPass@ArmAnim.fbx`.

5. Import that into Unity, in the same folder as the referenced model.



Unity will automatically assign the referenced model to the animation and will allow us to use it with our character.

The key here is the file referencing the model, and the correct naming convention for the animation file, which should be as follows: `referencename@animationname.fbx`. In this case, our original file was named **Character_animationPass** and then we named our animation **ArmAnim**, which you will see in the hierarchy window in Unity.

Rinse and repeat for other animations if that is your usual working pipeline. The major benefit of this approach is being able to add as many animations as you like after importing the model into Unity.

We've covered the basics behind the character export from one of the 3D applications, but there are others with their unique nuances. To find out how to export from a specific 3D app, visit the Unity official documentation, where you can read about importing objects from a specific app at:

<http://docs.unity3d.com/Manual/HOWTO-importObject.html>.

If you are using Blender and Rigify for character rigging, the following article on how to import a Rigify rig into Unity should interest you:

<http://docs.unity3d.com/Manual/BlenderAndRigify.html>.

Configuring a character in Unity

We will continue working with the character that was imported with embedded animations.

With the character imported, we go right to the **Rig** tab of **Import Settings** to explore the remaining two animation types: **Generic** and **Humanoid**. As mentioned in previous chapters, there are two animation types that are required in order to use Mecanim, a powerful animation control tool introduced in the 4th version of Unity. I can't stress enough, how awesome Mecanim is; this system allows you to significantly improve your development pipeline and reduce the amount of code to control animations, but more about that in the next chapter. Right now we need to figure out how to set up the model to be used by this system.

Generic and humanoid – what's the difference?

This is exactly how it sounds—Generic can be used for everything, from a dragon to a toaster, whereas, Humanoid can only be used on the characters that have the humanoid bone topology.

Generic Animation Type

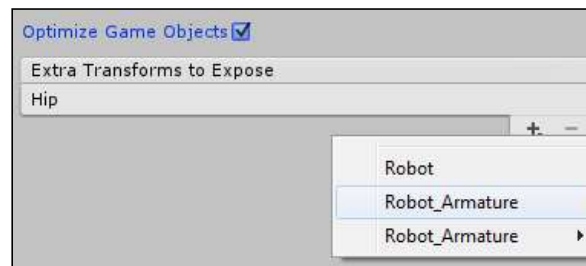
Generic is the easier type and you might end up using and relying on it most of the time especially when the humanoid doesn't work for you. So let's get the easier part out of the way and look at the following:



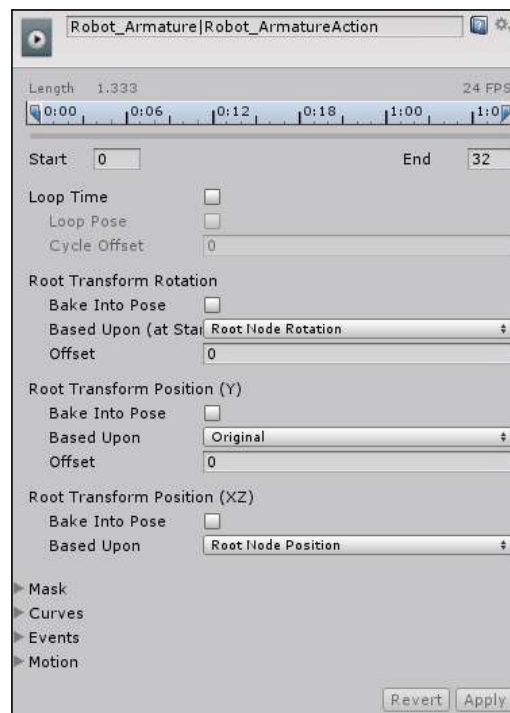
- **Avatar Definition: Generic Animation Type** doesn't allow us to make full use of **Character Avatar**, so leave it at the default; as **Create From This Model**.
- **Root node:** This is the node that contains animation translation; by selecting the node from the drop-down menu, you will enable the **Root Motion** parameters in the **Animation** tab. For now set **Root node** to **Robot** from the drop down menu.

- **Optimize Game Objects:** By default, Unity creates an empty GameObject for every transform in your character, checking this box allows us to prevent this from happening and increase the overall performance, since Unity doesn't have to deal with those extra transforms.

Thankfully, you can still create some of these transforms on demand if you need to reference them through the code. Select the transform from the hierarchy by clicking on the + sign of the **Extra Transforms to Expose** list which appears when you've selected the **Optimize Game Objects** option.



Now let's move to the **Animation** tab and look at how the **Clip** options changed since we last used them for **Legacy Animation**.



As you can see, there is a significant increase in the number of options, as follows:

- **Loop Time:** Checking this option will make our clip play through till the end and then restart from the beginning. This also enables the following options:
 - **Loop Pose:** This makes your animation loop seamlessly. However, this only works well if the starting pose matches the end pose; it will take the difference and will make a blend throughout the clips length to make them match.
 - **Cycle Offset:** This is the offset to the cycle of a looping animation.
- **Root Transform Rotation, Root Transform Position (Y), and Root Transform Position (XZ):** These serve a very similar purpose—they prevent the GameObject from being rotated or translated along the respected axis by the AnimationClip. In other words, if you don't want your GameObject to be moved by the animation, check **Bake Into Pose** in the required category, to prevent it. They will only appear if you've specified the **Root Node** in the **Rig** tab.
- **Based Upon:** You can choose the GameObjects rotation or position to be based on the **Root Node** specified in the **Rig** tab, or the way you set it up on exporting, by choosing **Original**.
- **Offset:** This allows you to add the offset to the rotation or translation of the GameObject if you chose **Root Transform Rotation** or **Root Transform Position (Y)** to be based on **Root Node** (the **Original** values will be taken from the model).
- **Mask:** This is very simple to understand and use. Let's say you need to remove a motion from the **Neck** transform, and its children, in one of your animations. To do that you, need to go under the **transform** menu, and uncheck the **Neck** transform. If you run the animation now, you will notice that the **Neck** transform, and its children, aren't moving. We will talk about the application of Masks in the next chapter.
- **Definition:** This allows you to choose to, either create the mask from this specific model, or copy it from another mask.

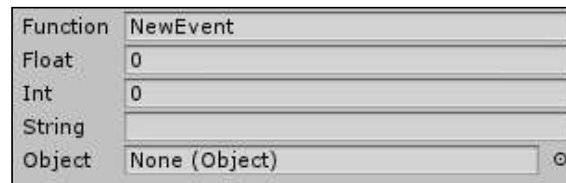
To create a custom avatar mask for our character, do the following:

1. Click on **Create** from the drop down menu and select **Avatar Mask**.
2. Go under the **Transform** drop down menu of the **Avatar Mask**.
3. Drag the **RobotAvatar** generated by our character model (it can be found inside the imported **Robot** model).
4. Click on the **Import skeleton** button.

This should load all the transforms from our character, into the mask, and allow us to configure them here, to be assigned to multiple objects with the same bone structure.



Events are improved and expanded from the Legacy Animation; now you can trigger any function on any `GameObject` just by filling in the blanks and specifying the frame on the timeline.



This covers the **Animation** tab for the **Generic Animation Type**; we will omit talking about the **Curves** and **Motion** parameters as they aren't required for this example.

Humanoid Animation Type

Imagine yourself in a situation where you have multiple humanoid characters that require the same animation – sitting, walking, running, and so on; or there is a specific animation that you would like to reuse on multiple humanoids. Usually, you would be required to create each animation for every single character, taking into consideration their body proportions. With **Humanoid Animation Type**, that is not the case, you can cut out a lot of steps in your usual animation pipeline by referencing animations using **CharacterAvatars**, and tweak them to match and tailor to any character using a muscle system. Let's take a look at how this is done.

Character avatar

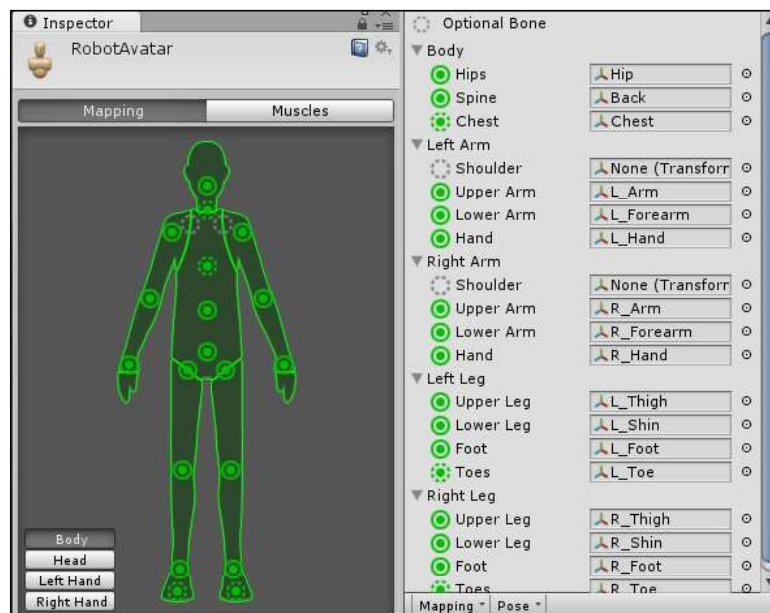
First things first, we need to switch our **Animation Type** of the imported Robot from **Generic** to **Humanoid**, and hit **Apply**.



Avatar Definition, using the default **Create From This Model** parameter, will generate a Robot Avatar for this model automatically, by trying to map the Robot's skeleton onto the humanoid topology. The key factors in successful mapping are hierarchy and naming (bone direction and ratio also contribute, however, not as significantly as those two). The algorithm will search for a bone called **Hip**, check if it's a root bone, and which bones are attached to it. So make sure that you're using proper naming conventions to get the best out of this process.

If the process is successful you'll see a check mark next to the **Configure...** button, but regardless, let's click on it and see what has actually happened.

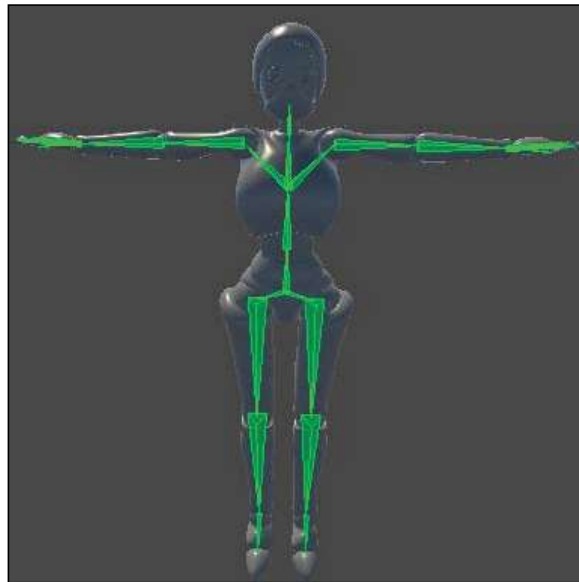
Unity will open a new scene with our character in the center and ask you to save the current scene; it would be wise to do so.



Robot Avatar shows us a humanoid body with bones represented by lined (mandatory) and dotted (optional) circles. As you can see, all the mandatory bones were successfully matched, leaving a few optional bones unchecked.

We can check the rest of our bones by switching between body parts at the right bottom corner (**Body**, **Head**, **LeftHand**, and **RightHand**).

Here is where it gets interesting. Take a look at the **Scene** window in our character model (hitting the **Configure...** button takes us to a different scene with our character).



All the bones that were mapped on the avatar successfully, now appear green. Any additional bones that don't follow the Unity topology standards will be marked as grey. In order to map the skeleton on the template, Unity will exclude any additional bones and, as a result, it will not be animated. A very common example of this is, if you are using the three bone structure for the torso such as the lower back, spine, and chest, then the spine will be grayed out; however, unlike avatar mask, it doesn't exclude its children, therefore, chest and its children will animate just fine. This is a bit unfortunate and there are some things that you need to keep in mind when creating a skeleton if you wish it to be recognized as a humanoid by Unity.

Correct topology

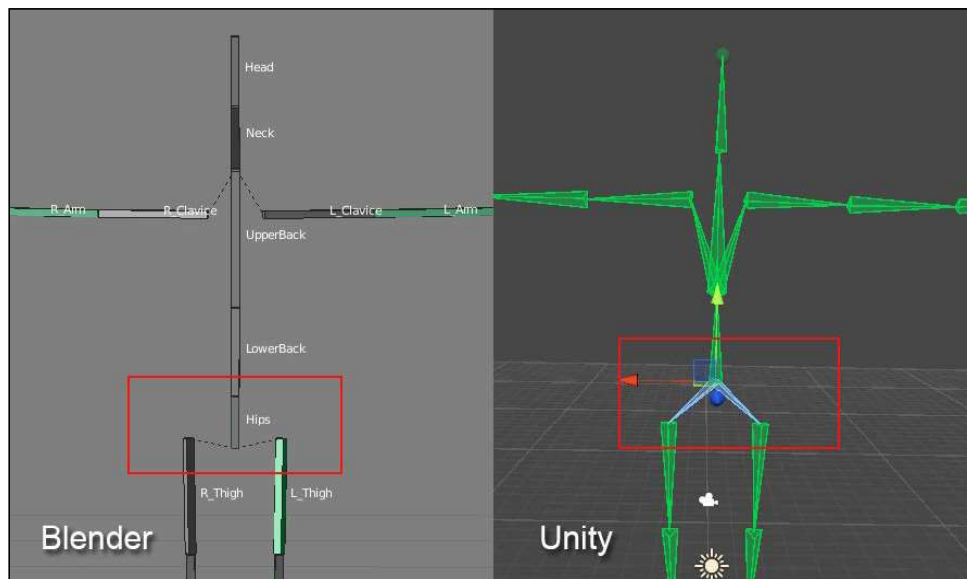
By topology, I mean that the skeleton must have certain bones that follow a strict hierarchy. Now, that doesn't mean that your character must have a certain skeleton or bust, you may or may not have certain bones, but there are those that define us as humanoids and they must exist. Here is the structure skeleton we must respect in order for our character to be recognized as a humanoid by Unity standards:

- Hips | Upper Leg | Lower Leg | Foot | Toes
- Hips | Spine | Chest | Neck | Head
- Chest | Shoulder | Arm | Forearm | Hand
- Hand | Proximal | Intermediate | Distal

The sad part here is that if you have any extra bones that are not included in the list, there is no way to add them to the avatar, the list is fixed. Keep in mind, however, that the humanoid animation type was designed as a compromise based on numerous humanoid rig standards. If you are using a different standard or have more bones to increase control and precision, or using the humanoid rig screws in your animation, then you can always switch to **Generic** without a need to re-rig your character.

Wrong topology example

Allow me to illustrate an example how a perfectly fine bone structure created in Blender can go wrong when viewed in Unity.



If you look at the image above you'll see that the **Hips** bone is the only one grayed out by Unity. That is happening because **Hips** is a child of the **LowerBack** bone. This is where Unity gets confused; it automatically assigns the topmost bone in the hierarchy to be **Hips**, searches down the hierarchy for two bones that represent **UpperLegs** and finds **R_Thigh** and **L_Thigh** (since they are children of the **LowerBack** child bone, they meet the requirements and will work just fine). But with our **Hips** bone from Blender, Unity will simply ignore it, as well as the animation data associated with it, as if it doesn't exist.

If you aren't planning on using animation referencing, you may simply ignore this issue and switch to **Generic Animation Type**, everything is going to work just fine there. However, if you do plan to rely on the animation referencing, then the only way to make it work, will be to go back to Blender, and reparent bones so that **LowerBack** is a child of the **Hips** bone.

Muscles

Before we go into this topic, there is one thing that is vital to proper muscle work, and that is the T-pose. Make sure that your character was modeled and rigged in the T-pose; this is very important. If, for any reason, that's not the case with your character, you can follow these steps:

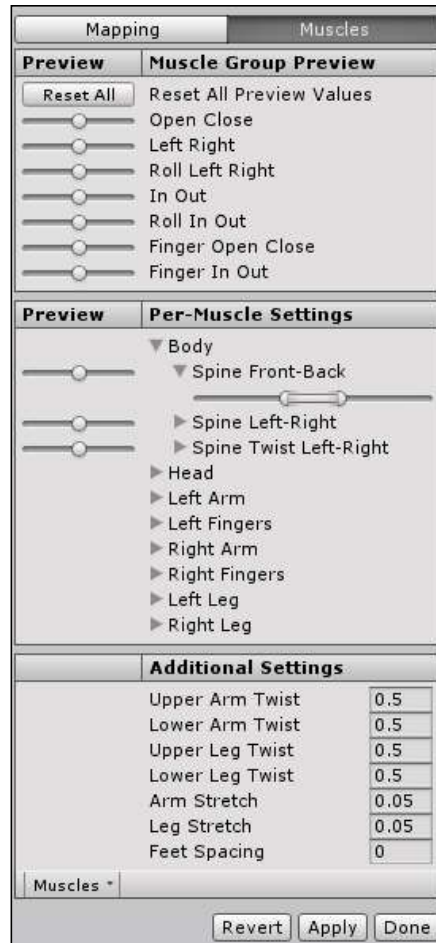
1. Find the **Pose** drop down menu at the bottom of the **Inspector** window
2. Select **Enforce T-Pose**

With a bit of luck, this will help (note that it's OK to have your character animated out of it, but the actual rig needs to be in the T-Pose).

The actual muscles in Unity are deceptively easy to use. Think of them as restrains that you might have already applied during the rigging process. Unity tries to apply its own settings of how the body parts should bend and twist.

Let's talk about the benefits of it. Imagine that you have an animation that is to be applied on two characters, one of which is naked and the other one is geared in full plate medieval armor. Simply referencing animation won't work, you would have a naked character looking stiff, or geared with body parts penetrating his own armor. Muscles allow you to configure their avatars individually, applying different restrains and making sure that the animations look better on both of them. Don't expect this system to create miracles and make a full plated guy's backflip look equally pretty, as it is ridiculous (unless you've designed a specific armor set).

As for the actual muscle configuration, it's very intuitive:



The editor is divided into three categories:

- **Muscle Group Preview:** This allows you to test a group of muscles by applying different motions that may or may not be a part of your animation.
- **Per-Muscle Settings:** This allows you to apply restrains on each individual muscle (generated from a humanoid skeleton and mapped on a character's bones). You can dig into hierarchy, specify the range of motion using the slider on the right, and test it with a slider on the left, under a **Preview** column.
- **Additional Settings:** This gives you extra options to play with to make sure that your character animates well.

After all the options are set, just click on **Apply**, then on the **Done** button, and you will return to the previous scene.

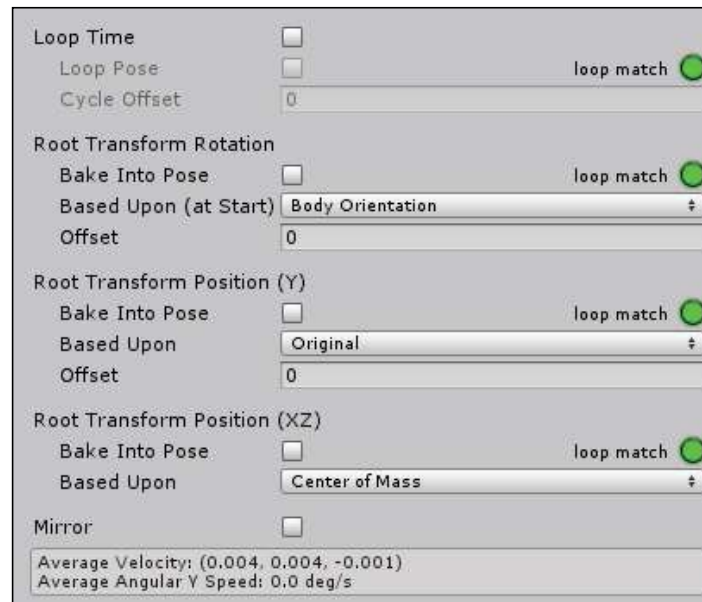
Adjust character muscles

In the next chapter, we will be importing the locomotion animation package from the Unity Asset Store. The Humanoid animation type will help us to reference them onto our character, but to make sure that the character looks fine with a random set of animations; proper set up of the muscle system is required. Try to edit the muscles and utilize the motion sliders to restrict character movement in order to avoid geometry overlapping. In the case of our character, the most problematic area will be the chest that might cause the hands to crush into the geometry while running. The final result should have no overlapping geometry while testing the character with the **Muscle Group Preview** sliders.

Additional options

The Humanoid animation type further extends our options in the **Animation** tab.

Animation clips now have additional indicators for **Looping**, **RootTransforms**, and **Root Rotation** called **loop match**. These indicators evaluate the difference between the first and final frame of the animation and suggest whether the **Loop Pose** option should be used.



The new humanoid animation type also brought a **Humanoid** setting to a **Mask**.

Using the humanoid mask, you can enable or disable the different groups of muscles and IKs to be affected by the animation, and see the result right in the preview window. As of v5.01, Unity supports only **LeftFoot**, **RightFoot**, **LeftHand**, and **RightHand** IK goals for the humanoid animation type.

This concludes this part of the animation pipeline. I hope you've managed to grasp the essentials of how this system works and what to expect from it.

Summary

Getting your characters from the 3D modeling software to Unity could not have been easier. Unity's support of popular file formats and additional compatibility with Blender, Maya, and 3Ds Max creates a very flexible pipeline. Humanoid animation type is an amazing feature that can speed up your animation process if you are willing to adjust to its strict standards.

If you still experience strange behavior with your rig after importing it into Unity, try searching the Unity forums (<http://forum.unity3d.com/>) for solutions, chances are other people have experienced the same problem and managed to find a solution to it.

In the following chapter, we will finally take a good look at this new beast called Mecanim and see how far we can get with all the work we've done up until now.

