Milestone 1 – Character Animation

This is an *INDIVIDUAL* assignment.

Use Unity version: <SEE SYLLABUS FOR REQUIRED VERSION NUMBER>

Late Policy

See Canvas Course site for late policy.

Description

For this assignment, you will be working with a provided Unity project that demonstrates four different types of controllable characters. The project is located at:

https://github.gatech.edu/IMTC/CS4455_M1_Support

Check the project out from git and load it into Unity. Follow the Walkthrough below. Finally, submit your project and answers according to the submission requirements.

Walkthrough

If you run the project (see link above), you will see four different characters. There are two humans and two Minions. You can toggle between controlling each character with the T key on the keyboard. If you have a compatible gamepad, you can use the thumbstick to control the character in analog fashion. If you use a keyboard, you can use WASD or the arrows to move. In addition, you can hit numerical keys 1-9 and 0 to change max speeds. 0 is full speed (default). 1 is slowest (10% of full). The keys in-between change the max speed in increments of 10%. Also, Q and E can be used for 50% turn rates. These special keyboard controls allow you to try out analog-like character control. This special input code can be found in Project:Assets/Scripts/CharacterControl/CharacterInputController.

HUD

First, update the HUD to show your name. Edit the Name game object under the "FramerateCounter – NAME TEXT IS HERE".

Skeletal Animated Character without Root Motion

With Hierarchy GameObject named SomeDude_NoRootMotion, carefully observe the animation of the character. In particular, pay attention to the feet relative to the ground. Also, try adjusting the max speed as described above.

Now do the same for SomeDude_RootMotion. You will probably notice that the root motion version looks a lot better, especially with different speeds. The version without root motion moves at a constant rate, unlike a real human. The root motion animation allows translations and rotations to slightly speed up or slow down, coinciding with natural movement. This gives the feet of the model a much more planted and realistic look. The feet sliding problem of SomeDude_NoRootMotion is especially bad for the turning animations. It's hard to programmatically get the movements of the Game Object just right to match what is happening during the turn animation.

In SomeDude_NoRootMotion's control script, try disabling the script code that sends the rotation input to the animator and test it out (it is a simple passing of a parameter to the animator that

you will comment out). You will probably find that this looks better. Without the subtle root motion corrections, the leaning into the turn is very hard to get to look good using programmatic control. In this case, simply not using turn animations is often better. You'll just have the walking forward animation and turn it in place.

One downside to root motion is that sometimes character control can be less responsive because one needs to wait for animations to accomplish the desired movement rather than just immediately executing a translation or rotation. Motion-captured root motion rotation animations tend to have a more obvious sluggish feel to them than walking/running. It's quite common to use root motion for translation but use programmatic rotation for snappy turning. In fact, a lot of Unity demos you find online use this hybrid approach as a compromise. One could similarly modify SomeDude_RootMotion to not pass the rotation parameter to the Animator, but code to programmatically rotate would need to be added (do not perform these steps for this for the assignment).

Root Motion Skeletal Animated Character

Now let's look into making SomeDude_RootMotion more capable by adding some run animations. Since he can only move as fast as the animations allow, run animations will really help him out. In the Animator view of SimpleAnimatorController, drill down into the "Blend Tree – Forward" state. Select the blendtree and observe the "2D Freeform Directional" blendtree in the Inspector. Here you can add Run, RunLeft, and RunRight animations that are found under ModelsAndAnimations->Runs. Adjust the 2D blend positions as appropriate to create a good blend map. It is recommended that you pull the walk animations closer to the blend map's origin and place the runs on the unit circle or closer. (You may find you need to warp the circular layout to more of a half oval with squished width to get the best results when using a thumbstick.) You can test the new blendtree by simply playing the game inside the Unity Editor. However, you can also hit play in the Inspector view of the blendtree and drag the little red dot around the blend map. The latter approach makes blendtree refinements much quicker. Once you get everything dialed in, run the game and test things out a bit.

Next up, let's see if we can add an extra degree of control over SomeDude_RootMotion's animations. Since you probably don't have the 3D animation software or motion capture hardware to revise the animations, you'll want to know about other methods of tweaking root motion. You can certainly make adjustments to the blendtree as you just did above. Also, you can adjust animation Mecanim state Transitions via ExitTime, TransitionDuration and other settings. For instance, you can disable ExitTime for immediate animation transitions to speed up responsiveness. Also, the TransitionDuration can be shortened for the same purpose or increased to make cleaner and more realistic looking animations due to the blending.

Another approach is to add some tweaks to the playback of animations. You will now implement some simple features that allow your character to move and turn faster while mostly preserving root motion. With SomeDude_RootMotion, add three public member variable floats to the RootMotionControlScript: animationSpeed, rootMovementSpeed, and rootTurnSpeed (default values of 1f). In FixedUpdate() of the same script, you can set the Animator Component's speed to the animationSpeed scalar. Refer to Unity's online documentation of the "speed" property of the Animator component. This will cause the entire Mecanim animation state machine and blendtrees to play faster or slower as animationSpeed is adjusted. A setting of 1f is regular speed, 0.5f is half speed and 2f is twice as fast.

Now modify OnAnimatorMove() in the same script to scale the difference in position and rotation from the previous frame (this.transform) as compared to the new candidate root motion pose. It is recommended you accomplish this with Vector3.LerpUnclamped() and Quaternion.LerpUnclamped() using your two new scalars, rootMovementSpeed and

rootTurnSpeed. You can refer to Unity's online API documentation if you aren't familiar with Lerp (linear interpolation/extrapolation).

Next, play the game in the editor. Slowly increase the three new scalars via the Inspector view of SomeDude_RootMotion. You should be able to make the character much quicker and more responsive while minimizing foot sliding. Keep in mind that Inspector values changed during game play will be reverted once the game is stopped. So, make note of the scalar values you found that worked best. Once the game is stopped, reenter the new values and save your scene. Also, make sure that the simulated analog controls still work after making the animation speed tweaks!

Minions – Custom Game Objects and Animations

Next up are the minion characters. Both of the minions follow the same pattern as the humans with programmatic movement and root motion control. The trade-offs between the two types of movement aren't nearly as noticeable since these particular minions don't have feet (lost in a workplace accident). One thing about the minions is that their model was made within Unity using primitive shapes. Mecanim is able to animate nearly any attribute of any GameObject. Creating new interactive characters like the minion example is really quite easy. The trick is to leverage the Hierarchy relationships between the GameObjects that make up a character and animate the relative poses. You can even use Mecanim to animate things that aren't characters like elevators, drawbridges, etc.

Check out the Minion animations in the Animation view (the files are in Hierarchy:Assets\Animations). In particular, check out Minion_NoRootMotion's "Minion Forward No Root". You can scrub through the timeline to see the animation progress through the keyframes. This animation simply moves the minion up and down and rotates slightly around the Z axis. If you look at the curves tab of the Animation, you can see that the interpolation between keyframes is not simply linear but follows a curve (defined by control tangents). The ability to control the tangents for keyframe interpolation is great for making nice animations.

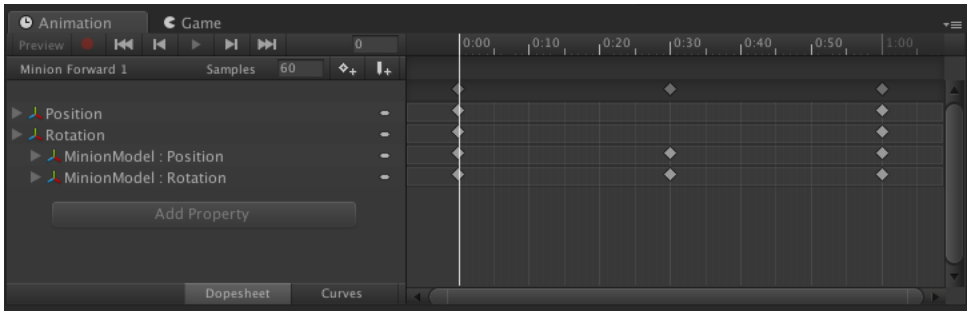
By the way, the view controls of the Curves tabs of Animation is a bit cumbersome to navigate. You can zoom with a mouse scroll wheel or two-finger touchpad zoom. However, this zooms both axis equally. Hold down either SHIFT or COMMAND (maybe ALT on Windows) to zoom the axis independently.

Next, check out Minion_RootMotion's "Minion Forward" and "Minion Forward Turn Left" animations. "Minion Forward" only defines a simple Z direction translation in a straight line. "Minion Forward Turn Left" rotates while moving forward and to the left along a curve. You should check out the Curves view to see what's really going on. Setting the tangents lets us get away with only two keyframes to pull off a turn animation. You could of course add more keyframes and rely less on the tangents though.

Since Minion_RootMotion's animation is pretty boring, add a hop and a twist to the movement of "Minion Forward" similar to "Minion Forward No Root". You'll probably want to preserve the original "Minion Forward" in case you mess up. To make a duplicate animation, select the animation in the Project View and execute Edit->Duplicate and rename the new file to something appropriate. Use the Animation view to edit the animation and add couple bouncing "steps." Do the same with the left and right forward turn animations. Just makes sure all follow the same pattern. For instance, if the first hop leans to the left and the next leans to the right, you should replicate that in all three animations.

When adding new keyframes for the hopping, be sure to modify the minion model transform and not the root game object position (otherwise you'll mess up the capsule collider orientation). Also, you can add partial keyframes (only keyframing the parameters that are changing). If you

add complete keyframes for all parameters you might end up messing up the curves for the root motion of the game object.



Example partial keyframe shown in the figure above (halfway across timeline).

Now test it out in the Mecanim Animator, "MinionAnimatorController." Change the forward blendtree's referenced animations to your new ones with the hop. If the animations match well enough the hops should smoothly blend for any ratio of forward and turning.

The other thing the minion needs are some squeaky footsteps like his brother (according to official Minion lore, there are no girl minions because minions are too stupid to be female). To accomplish this, first check out the installed components on the Minion_NoRootMotion in the Inspector. You should see that he has a MinionFootstepEmitter attached. Add that same script to Minion_RootMotion. Also, look at the actual script code. You'll notice that it implements a method, ExecuteFootstep(). When called, this method generates an event that is consumed by the AudioEventManager, which plays the sound.

You may be wondering, where is ExecuteFootstep() actually being called? Unity supports a feature called animation events. These are events that you can embed within an animation. When the event is triggered, Unity will call a callback method of your choice. In the Animation view of "Minion Forward" and with the Dopesheet tab selected, click the "Add Event" button (looks like a white pencil with a plus next to it). The new event will show up just below the timeline as a little pencil shape. Drag it left/right to wherever on the timeline the footstep should occur.

With the animation event selected (little pencil turns blue), select the ExecuteFootstep() callback in the Inspector view Function setting and send it to the MinionFootstepEmitter as the receiving Object. Add any more animation events you need for other footsteps (probably at least two). If your footstep occurs at the very beginning of the animation, you don't need one at the very end. This is because of the looping nature of the animation (beginning and end are essentially the same moment). Implementing footstep animation events only in "Minion Forward" is fine for now. But in your future work you might want all your animations to support the event. Be sure to check out the *Tips* section below for discussion of advanced footstep implementation.

Be sure to explore the rest of the code. We will look more closely at the event system that is present in EventManager, AudioEventManager, etc., in future lectures and milestones. Consider which character control methods you like best. Make a backup of the work you have done so far and play around with modifications and extensions to the existing characters. You can also try making entirely new ones based on the patterns that you have learned. Check out the *Tips* section below for how to create or import new humanoid models.

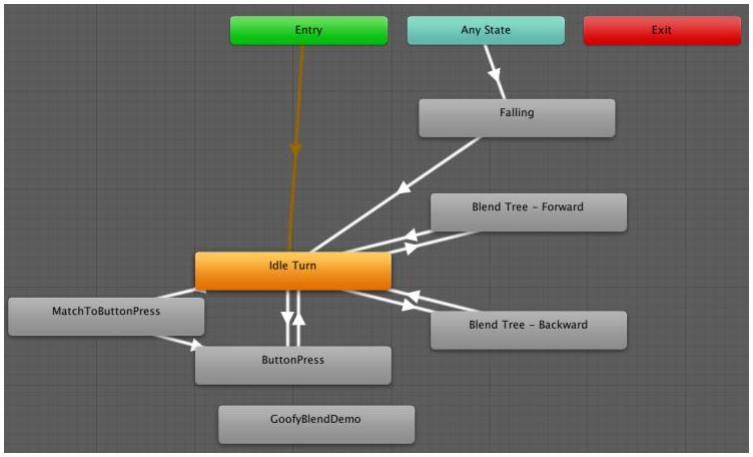
Character Positioning: Match Target and Inverse Kinematics

Now we are going to go back to SomeDude_RootMotion. You may have noticed a stick with a red ball on it just ahead of the character. You are now going to leverage Unity's Inverse Kinematic (IK) features so that the character can press the red ball as if it is a button.

If you run the game and select SomeDude_RootMotion, you can press the "Fire1" button to execute a button pressing animation. But it only works if you get SomeDude_RootMotion perfectly placed on the blue marker and facing the button. On a keyboard "Fire1" should be the Left-Control key. Otherwise, you can check Unity's InputManager setting to see what mapping is configured. This distance/angle constraint is already implemented for you. Many games will only allow a character to interact with something if they are close enough. This constraint is implemented via a reference to a buttonPressStandingSpot object (the blue marker in the scene) and conditional evaluation of buttonDistance and buttonAngleDegrees in FixedUpdate() of RootMotionController.cs.

The first thing we want is to make the button press action a little more forgiving regarding where SomeDude_RootMotion is relative to the button. To do this we will use Unity's Animator.MatchTarget() method.

Take a look at SomeDude_RootMotion's Animator Controller state machine. You will see that there are two transition paths through the state machine that lead to the ButtonPress state. Either a transition directly from "Idle Turn" to "ButtonPress" or "Idle Turn" -> "MatchToButtonPress" -> "ButtonPress". The reason for this is that we want SomeDude to immediately press the button if he is perfectly aligned but if not then he will move into alignment first via "MatchToButtonPress."



The "MatchToButtonPress" animation state is simply the character taking a couple steps forward (without root motion). It is unlikely that this animation would be sufficient to get SomeDude exactly where he needs to be to press the button (if root motion was on). However, through the use of Animator.MatchTarget() we can augment the animation to align with the standing spot. The fact that SomeDude is sliding and turning into alignment will be somewhat hidden by the steps he will be taking.

First uncomment the two lines below "TODO UNCOMMENT THESE LINES FOR TARGET MATCHING".

Next, replace the comment "TODO HANDLE BUTTON MATCH TARGET HERE" with the following:

```
// get info about current animation
```

```
var animState = anim.GetCurrentAnimatorStateInfo(0);


// If the transition to button press has been initiated then we want
// to correct the character position to the correct place
if ( animState.IsName("MatchToButtonPress")
   && !anim.IsInTransition(0) && !anim.isMatchingTarget )
{

}
```

The code you just pasted queries for the current animation state. This allows us to know when a particular animation is playing. Additionally, we check to make sure we aren't in a transition between animations and that we aren't already performing Animator.MatchTarget().

Next, add the following inside the if-clause you just placed:

```
if (buttonPressStandingSpot != null)
{
  Debug.Log("Target matching correction started");


  initalMatchTargetsAnimTime = animState.normalizedTime;


  var t = buttonPressStandingSpot.transform;
anim.MatchTarget(t.position, t.rotation, AvatarTarget.Root,
            new MatchTargetWeightMask(new Vector3(1f, 0f, 1f),
                                      1f),
            initalMatchTargetsAnimTime,
            exitMatchTargetsAnimTime);
}
```

The call to MatchTarget() allows us to reference a current animation to serve as a time line. Over the progression of this timeline we will override root motion and interpolate towards a desired pose. The specifics of the pose are determined by which part of the character is used as a reference point and which dimensions are used via a MatchTargetWeightMask. The time period of the interpolation is controlled by a normalized time reference for the start and stop. Check out the following link for more details:
https://docs.unity3d.com/ScriptReference/Animator.MatchTarget.html

Now, try it out. You should be able to walk SomeDude_RootMotion close to the marker and when you press "Fire1" you will see him turn and take a couple steps ending with alignment with the button followed by a press animation. The MatchTarget() call doesn't work perfectly. What you just implemented is a minimal solution. Ideally, you would include more animations and possibly

some AI control to get the character closer to the button before using MatchTarget(). However, even this minimal solution is a big improvement in the quality of interaction.

Now let's focus on the actual button press. Note that the button press animation always moves the hand to exactly the same spot according to the keyframes of the animation. But we want the hand to go to where the red ball is. In order to do this, we will override the animation with IK to go to the ball.

IK works in the opposite direction of forward kinematics. In this case, we specify the desired pose of one of the extremities of the character (hands, feet, head) and IK will determine a valid pose configuration of all bone rotations up the chain. For instance, if a hand is set to be positioned at a particular point, a pose will be determined for the wrist, elbow, and shoulder.

In order to get IK working, the first thing you need to do is enable IK on the appropriate Animation Layer of the character's Animator. In this case, there is only one layer, the "Base Layer." Under the Layers tab of the Animator, click the gear and check IK pass (note that it may already be checked).

Next up, we will add a callback to the RootMotionControlScript called OnAnimatorIK(). It should have a void return type and "*int layerIndex*" as the parameter. Within this method, you will implement the logic necessary for the character to be able to look at the button and to reach out and touch the button.

Within OnAnimatorIK(), we need to add some logic to recognize when the ButtonPress Animator state is playing. You can accomplish this with:

```
if(anim) {
    AnimatorStateInfo astate = anim.GetCurrentAnimatorStateInfo(0);

    if(astate.IsName("ButtonPress")) {

    }
    else
    {

    }
}
```

Now add a public member variable to RootMotionControlScript for a GameObject named buttonObject.

Next up, we will add some calls to the Animator to solve IK. Add the following to the if-clause from above.

```
float buttonWeight = 1.0f;
```

```
// Set the look target position, if one has been assigned
if(buttonObject != null) {
   anim.SetLookAtWeight(buttonWeight);
   anim.SetLookAtPosition(buttonObject.transform.position);
   anim.SetIKPositionWeight(AvatarIKGoal.RightHand,buttonWeight);
   anim.SetIKPosition(AvatarIKGoal.RightHand,
                buttonObject.transform.position);

}
```

In the else-clause, add the following:

```
anim.SetIKPositionWeight(AvatarIKGoal.RightHand,0);
anim.SetLookAtWeight(0);
```

Now in the Inspector view of SomeDude_RootMotion, connect the public parameter buttonObject to the Scene Hierarcy game object ButtonHandle (child of ButtonParent).

You should now be able to play the game and observe the character touching the button when you press the “Fire1” key. However, you will notice that the hand immediately snaps to the button during the animation and then snaps back down again at the end. This is because we are using a weight of 1.0 for the IK pose versus the current animation pose. What we really want is to smoothly transition from the animation to the IK override.

This can be handled by animating the buttonWeight float value. The Mecanim animation system can be used for animating all sorts of things, including float values such as the buttonWeight. In the Project View (Assets), navigate to and select ModelsAndAnimations/YBot button push/ybot@Button Pushing. In the Inspector, scroll to the Curves section and expand it. You will see that there is a buttonClose curve already created for you. Scrub through the animation preview and you will see that the buttonClose parameter is 1.0 during the time that the hand is pressing the button. Outside of that period of time, buttonClose is a value less than 1.0. This animation curve can be used to smoothly transition the IK weight and therefore create a smooth correction of the animation to get the character to properly click the button.

In the script, change the buttonWeight declaration line to use the buttonClose animation curve value read from the Animator:

```
anim.GetFloat("buttonClose");
```

Now run the game and try out the button press again. It should animate nicely with the hand going to the correct location. Try selecting ButtonParent in the Hierarchy while the game is running and translating it up and down to various heights. The IK should adjust the hand position accordingly as you test out the button press.

You can use IK easily in your own games for situations where corrections to the hands or feet are desirable such as button pressing, grabbing items off the ground, grabbing ledges, placing feet on uneven terrain, looking at points of interest, etc.

You have now completed this milestone. When you are ready to submit your work, double check that you completed everything in the *Modifications Check List*.

Audit Tool

Find the Auditor in the Hierarchy. Go to the child object of the same name. In the Inspector of this child object, set your last name and first initial. Confirm that assignment code is set to "m1".

With this set, you can now build your project. If you run from the build, you will get an audit displayed on screen. Use the audit tool for all individual milestones. Just change the assignment code as you move through the assignments.

Modifications Check List (100 points)

1.) SomeDude_NoRootMotion – disable turn animation by NOT passing turn input parameter to the Animator in the script (only turning programmatically and the character should still be walking during the turn) **10 pts**
2.) SomeDude_RootMotion – add running animations (forward run AND turning runs) to forward blendtree and move the blend ratios around as appropriate so that the player can easily control the character. The character should continue to run while turning at full speed. **20 pts**
3.) SomeDude_RootMotion – add public scalars that allow adjustment of animation speed and root motion scale (translation and rotation). Adjust the scalars slightly faster than default until you are happy with control speed and overall animation quality. The goal is for the character to have the capabilities of a super hero, but still look reasonable natural. **20 pts**
4.) Minion_RootMotion – Replace/modify the forward and turn animations to include some comical hopping steps. **15 pts**
5.) Minion_RootMotion – Add animation events that generate minion squeaky footstep events to your forward animation. **15 pts**
6.) SomeDude_RootMotion – Match Target and Inverse Kinematics addition for button press animation **20 pts**
7.) Also, update the name that appears on the HUD. The placeholder is found under the framerate overlay canvas.
8.) Make sure to provide builds of your project with the auditor configured and generating no errors from your build. You will need to set Project Settings->Player::Product Name to "Last_FirstInit_m1". Follow Assignment Packaging and Submission formatting mentioned below. For Mac build, make sure to select Intel 64-bit + Apple Silicon for Architecture.
9.) Make sure your build demonstrates all of the above as the TAs predominantly use your build to assess runtime deliverables.

Submission:

If your zipped project is too big for Canvas, you should submit a private cloud hosting (such as GT's Box license) link to a ZIP (or 7zip) file of your Unity project via Canvas. **Please clean the project directory to remove unused assets, intermediate build files, etc., to minimize the file size and make it easier for the TA to understand. Refer to Assignment Packaging and Submission on the Canvas Syllabus for further details.**

The submissions should follow these guidelines:

a) Your name should appear on the HUD of your game when it is running.
b) Follow the *Assignment Packaging and Submission* steps including:
    i. ZIP file: *<lastName_firstInitial>_m<milestone number>.zip*
    ii. Complete Unity Project
    iii. Builds
    iv. Readme file should be in the top-level directory: *<lastName_firstInitial >_m<milestone number>_readme.txt* and should follow base requirements from *Assignment Packaging and Submission*
    v. Size reduction

Submission total: (**up to 20 points deducted** by grader if submission doesn't meet submission format requirements)

Be sure to save a copy of the Unity project in the state that you submitted, in case we have any problems with grading (such as forgetting to submit a file we need). Do not alter or remove your submission from cloud hosting until your grade has been returned.

Useful Resources:

- The following project contains a starting point for this assignment: https://github.gatech.edu/IMTC/CS4455_M1_Support
- Event System Tutorial - https://unity3d.com/learn/tutorials/topics/scripting/events-creating-simple-messaging-system

  Note that the EventSystem code has been improved and is included in the github project above (along with a working demo)
- Check out the 3rd Person Character under Unity's standard assets as a possible source of animations
- Setting up a model in Unity - https://www.youtube.com/watch?v=pbaOGZzth6g
- *Adobe Mixamo* is a great source of models and animations. https://www.mixamo.com

Tips:

These tips cover lots of issues related to making Mecanim humanoid characters. Be sure to refer to these tips as you begin working on your group projects.

General:

- Pay careful attention to any tutorial/instructions you follow related to humanoid models. Often if you miss one little step, you'll have a lot of trouble. This is particularly true for importing models and animations.

Hierarchy Layout:

- There are good reasons to have your model be a child of a game object that contains the character control scripts, collider, etc. For instance, if you do so you can swap models out easily (possibly even at runtime for a morphing character). Unfortunately, this adds a bit of complexity. You can read more about it here:
- https://docs.unity3d.com/Manual/Retargeting.html
- Also, the following thread explains how to pass a child's root motion up to a parent:
- http://answers.unity3d.com/questions/1081677/correct-way-to-handle-root-motion-with-parent-game.html

Animations in General:

- All blended animations must be similar (e.g. start with the same foot and take the same number of steps). Otherwise, blending will give wonky results.
- Some animations you may find have way more steps than you want. Don't assume anything about an animation until you watch it in the inspector preview. Observe the timeline to see when it loops and count the steps taken up to that point.
- You only need a single turn left (or a turn right) animation for your character to turn. Normally you can use the mirror checkbox in your blend tree. So long as you are working with Humanoid animations, Unity is usually smart enough to both mirror and figure out how to blend the footfall correctly. However, if you get a bunny hop (out of sync blend), you can alternatively duplicate the offending animation in the assets (Edit->Duplicate), then set the animation to be mirrored in the animation's inspector settings. Lastly, adjust "Cycle Offset" under the Animations tab such that the correct foot takes a step first.

Adobe's Mixamo:

As of the end of 2018, exporting from Mixamo as regular FBX works slightly better than FBX-for-Unity, but you can get either to work.

I recommend you first make an assets folder for your new model so that you can see all the related import files for that specific model (e.g. make a folder "some_dude" that is empty).

One you add FBX to Unity Assets in the new folder from above, in Import Settings set Rig to Humanoid and Materials to "Use External Materials (Legacy)". Then Apply. With a regular FBX you will probably get an error about incorrect normal map texture naming, but you can click a button to automatically fix it.

The weird bit now is that sometimes the import process doesn't always completely run. In your assets you want an "Import Settings" object with the FBX base name, a "Materials" folder and another folder with the actual texture files (similar name as FBX model). You might try hitting play, adding/deleting model to your scene, hitting play some more, etc. Eventually you should get the full import to run and see the above mentioned assets. Just keep trying until you have all three. But you might get lucky and everything works first try.

If you are using the regular Mixamo FBX you will now need to go change the materials to be opaque so the model doesn't look bizarre. With FBX for Unity you will need to do the same

thing, but also you will need to manually assign each texture to the materials for Albedo and Normal map. For either case, I think Gloss textures can be placed in Metallic category but you will need to manually adjust Smoothness.

If you have freaky eye lashes or other parts that need transparency then you need to do the trick mentioned below.

More Mixamo:

- The Adobe Fuse application (custom character builder) is no longer an actively supported product. However, we are keeping some troubleshooting tips below in case anyone runs a legacy version of the software.
- Adobe Fuse may work better with the Mixamo server-side auto-rigging if you set your new models leg stance apart. Try doing this if the wrong part of the leg and/or clothing follows the wrong leg bone.
- When using Fuse on OSX, you might have trouble adjusting your character's physical properties. If nothing seems to work, check Fuse -> Preferences and under "App Options" set Maximum Adjustment Shape Value to 100.
- Export your character model as FBX or FBX-for-Unity (see discussion above). It should come with a TPose "animation" that doesn't actually animate.
- When you add the model to your Unity assets, immediately change the rig to Humanoid and the avatar definition to "Create from this Model"
- If your character looks weirdly inside out or partially transparent, you will need to change the character materials from transparent to opaque. If your character has eye lashes, you can make them look better if you duplicate the body material and make it transparent and then assign to the eyelash mesh. This can work for other geometry that needs transparency as well.

  > "This [problem] is actually a transparency issue with the way that Unity handles opacity. We recommend duplicating the body material, naming it 'eyelashes'. Set the body material to have no transparency. Set the eyelashes material to use transparency. Then just assign the new transparent material to the eyelashes alone and leave the non-transparent material on the body. You might need to check that transparency is off for the other textures as well"

- Export each of your Mixamo animations as FBX for Unity.
- When you add the animations to Unity assets, immediately change the rig to Humanoid and the avatar definition to "Create from Other Avatar". Drag the avatar from the original T Pose model to the "Source" avatar field.
- For all animations that are intended to be blended for on-foot locomotion (e.g. walking, running), set the "Loop Time"==true. You will also probably want to set "Root Transform Position (Y)" to "Bake into Pose"==true. This will allow natural falling behavior. Also, see "Falling" section below.
- For straight ahead locomotion you can set "Root Transform Rotation" "Bake into Pose"==true which can sometimes smooth out the chase camera from rocking back and forth.
- You can usually use humanoid animations from another source with a Mixamo model (or any other rigged humanoid model). However, you typically don't want to try to change the avatar on the animations themselves (because the bone names likely won't match). Instead, just try adding the animations to your mecanim animation controller. This will often work with decent results if the models aren't too different in dimensions. You can also just assign an existing AnimatorController to your new Mixamo character's Animator. Use the Mixamo character's avatar under the same Animator settings. This approach will usually give decent results as well.

- You will find that Mixamo animations are often annoyingly all named “Mixamo.com” within the exported FBX file. You can change the name under the Animations tab of the Animation Inspector next to an icon that looks like a play button. This will make setting up blendtrees less painful as you will be able to identify each animation clip by name.

Animation Curves:

Animations with non-zero offsets

(The following may have been patched since the problem was first noted)
If your animation uses a non-zero start offset, the animation curves you author don't match the normalized timeline of the animation preview. The way to deal with this is to write down your non-zero offset so you don't forget, set it to zero (temporarily), then author your animation curve as your scrub through the animation (e.g. looking for footsteps). Lastly, re-enter your original offset value and click "apply."

Using Animation Curves for Footsteps

I personally make a curve named "foot" with a value of 1.0 for left foot on the ground and a value of -1.0 for right foot on the ground for all animations. Then I make sure to add a "foot" parameter to the mecanim state machine so I can get the value out in script. Next, you can call Animator.GetFloat() to query the current "foot" value. I look for the "foot" value to go above/below some threshold like 0.6 to know when to trigger the left/right footstep events.

Debugging Animation Curves

For debugging of what the animation curve blend looks like in graph form, you can add a public AnimationCurve to your character controller script and populate it with a running list of "foot" readings. Here are some of the key pieces of code for how I do it:

```
public AnimationCurve footPosBlend = AnimationCurve.Linear (0f, 0f, 1f, 0f);


....


protected void clearFootPosHistory() {


 //not efficient, but just temporarily used for debugging
 footPosBlend.keys = new Keyframe[0];


 //make a nice flat line for keyframes to be inserted into
 footPosBlend.AddKey (new Keyframe (0f, 0f));
 footPosBlend.AddKey (new Keyframe (1f, 0f));


}
```

```
....


//in update()

AnimatorStateInfo currentState = anim.GetCurrentAnimatorStateInfo (0);

footTime = currentState.normalizedTime % 1f; //modulus to find loop start/stop times

//reset the graph when we reach the normalized anim loop end
if (footTime < prevFootTime) {
 clearFootPosHistory ();
}

prevFootTime = footTime;
footPosBlend.AddKey (new Keyframe (footTime, footPos));
```

The above approach to debugging blended animation curves works best if you have a game controller. That way you can click the animation curve in the inspector while the game is playing. This will give a live readout of the graph values and the joystick on the controller lets you move the character around at the same time. Unfortunately with keyboard control the game view must have focus and this causes the animation curve graph window to disappear. You could write a Unity editor extension to overcome this if you like.

Animation Events:

Animation events initially appear easier for footstep implementation, but if all your blended animations call the same callback you will get duplicate calls. Unity is not smart enough to blend and merge callbacks from blended animations. The end result sounds bad with lots of doubled up footstep sounds a few milliseconds apart.

Two approaches we know of for making animation events work:

Filtering:

Make a left and right foot callback and call those appropriately in all your animation events. In the script for the callbacks, keep track of a timestamp of the last time the footstep sound was played. Write a condition that the footstep sound will not be played again until a certain amount of time has elapsed from the previous playback. This will remove most duplicate sounds. (Separating left and right footstep events will make this filtering work better.)

By animation weight:

In your Update(), call IAnimatorControllerPlayable.GetCurrentAnimatorClipInfo() to get all the current animations that are being blended. The returned AnimatorClipInfo's will contain the current weight that the blendtree is using as well as the clip itself. Whichever of the clips has the highest weight should get to specify when the footsteps play. To do this, you will need to make uniquely named footstep callback methods for each clip that is in your blendtree (I don't think there is a way to pass the caller as a param which would otherwise let you make the determination of where the call came from). Each callback will need code that only plays the footstep audio if the calling clip is the currently highest weighted clip. Note that I haven't actually implemented this before and I can't say for certain that IAnimatorControllerPlayable.GetCurrentAnimatorClipInfo() will correctly return all clips being blended by a blendtree animation state. Also, you may still need to use the filtering technique from above in case of sudden changes in blending frame to frame that might result in a footstep being skipped over.

Blendtree:

- A 2D Freeform Directional Blendtree should work well for you for locomotion. I make one for the forward variants of locomotion (and turn in place) and a separate one for backward variants (also with turn in place).
- The inspector is great for debugging blendtrees.
- Previewing a blendtree might work just fine, then look bad when running the game. If so, you probably didn't set "Loop Time"==true as described in Mixamo section above for ALL animations being blended.

Mecanim:

- Be wary of transition settings between states. These can create long delays between animations with the default crossfade settings
- Consider enabling "Foot IK" on your animation states to get better foot placement

Legacy InputManager:

- Read up on Unity's InputManager, in particular be aware of the filtering effects that cause delays in input values changing (e.g. gravity, sensitivity, snap, etc.)
- You can also just use raw input values and handle filtering in your character control script.
- The new input system may have similar issues

Animator Component:

- Make sure "apply root motion" is turned on, or that you are implementing OnAnimatorMove().

Falling:

- A root motion based character will only fall if "Bake into Pose Position Y" is set for the currently blended animations. See following for more details: https://docs.unity3d.com/Manual/RootMotion.html
- Avoid scaling your character's root game object if at all possible. This will throw off gravity if you are using a rigidbody+capsuleCollider. However you can fix this by applying extra downward force to your rigidbody every FixedUpdate(). The 3rd Person Character scripts under Unity Standard Assets has an example of gravity scaling.

Event System:

An example event system is provided in the github project.

Putting your name of the HUD:

How to make a screen space overlay canvas with text:
File Menu: GameObject>UI>Canvas
(defaults to screen space overlay, which you want)
select the canvas in scene hierarchy
File Menu: GameObject>UI>Text
select the text in scene hierarchy
optionally set your view to 2D mode in your scene view
set min/max anchors of text's rect transform to 0,1 (top left corner)
play with font type, size, color until appropriate
drag text box to top left; resize box to fit your name if it's getting cropped
(turn 2d mode back off when you are done)