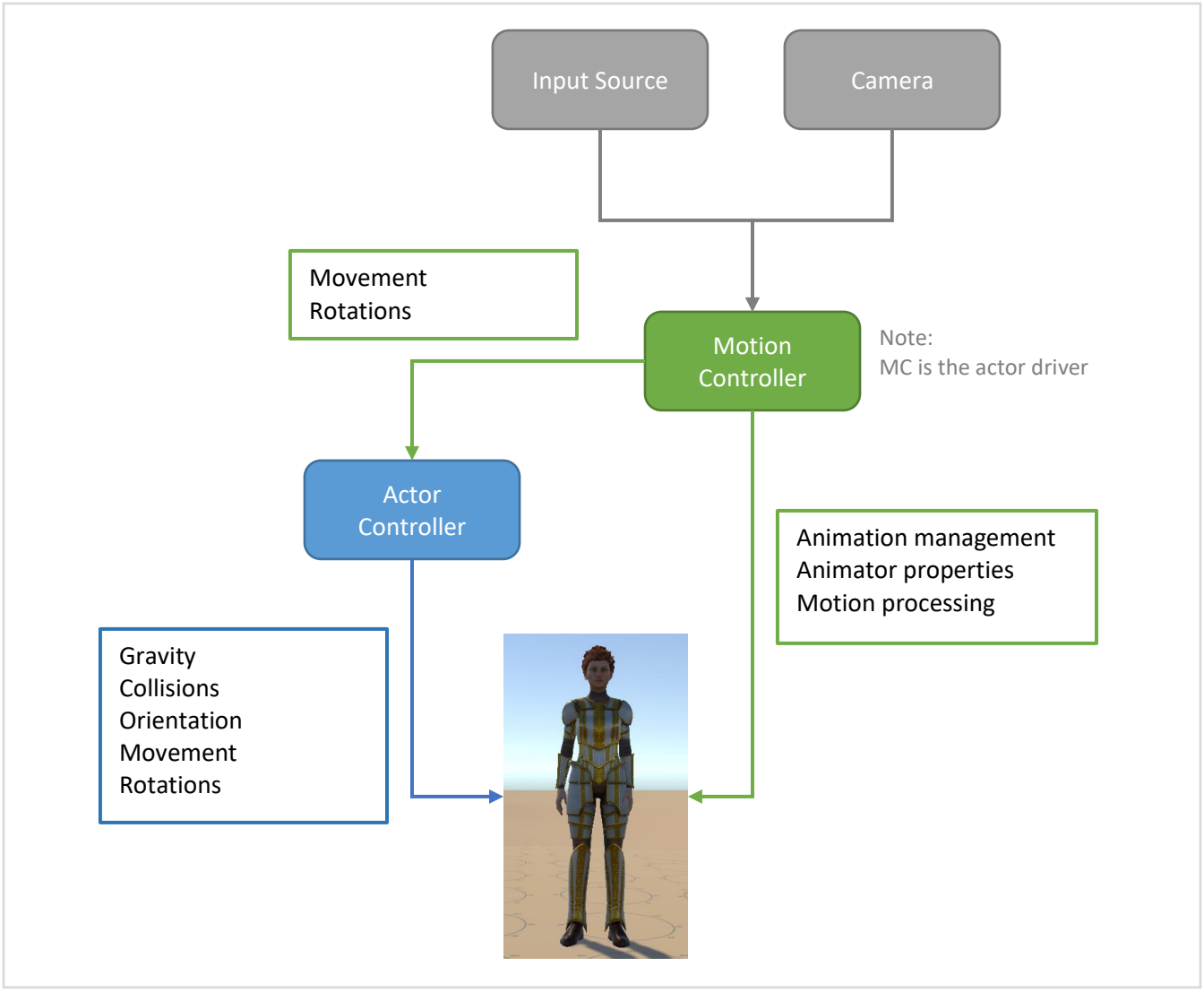
IMPORTANT

View the latest User Guide by clicking here.

The Motion Controller uses the Actor Controller to actually move the character.
So, please review the Actor Controller Guide as well!



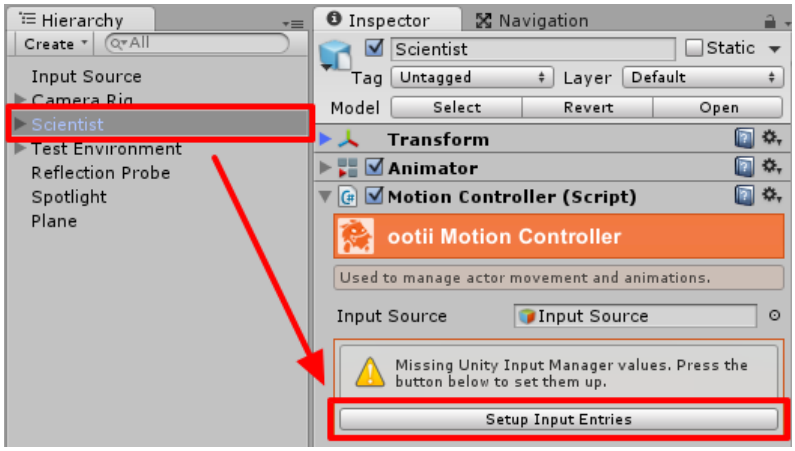
Demo Scene Quick Start

1. **Import** the Motion Controller package into a new project.

2. Open a demo scene.

Assets\ootii_Demos\MotionController\ Scenes

3. Select the scientist and press **Setup Input Entries**.



4. Press **play**.

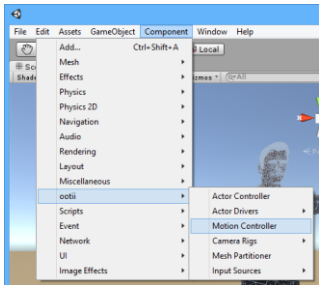
By default, the demo uses the MMO movement style:

W key	= move forward
S key	= move backwards
A key	= rotate left
D key	= rotate right
Q key	= strafe left
E key	= strafe right
Left Shift	= run
Space bar	= jump/climb
RMB down	= rotate character
LMB down	= rotate camera

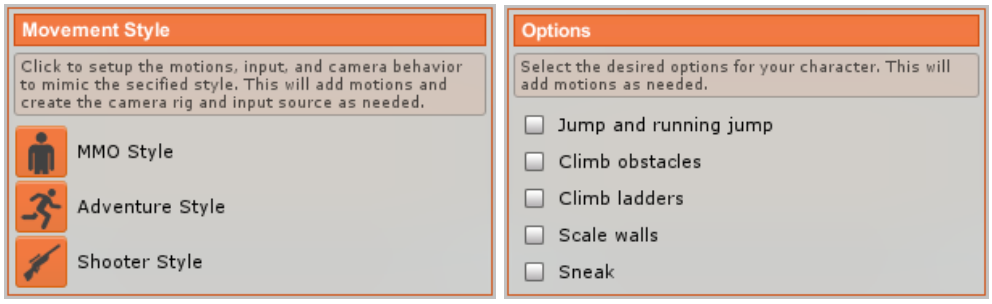
Custom Scene Quick Start

1. **Import** the Motion Controller package into your project.

2. Select your character and add a **Motion Controller**.



3. Select a **movement Style** and options.



When the movement style icon is clicked, the Motion Controller will automatically:

A. Add an input source to the scene (if there isn't one)
B. Add a camera rig to support the movement style (if there isn't one)
C. Move the "main camera" into the camera rig
D. Add a "Humanoid" Mecanim controller (if there isn't one)
E. Add the appropriate motions

4. Press **play**.

Note: Never nest the camera under the character it creates odd behavior.

Foreword

Thank you for purchasing the Motion Controller!

I'm an independent developer and your feedback and support really means a lot to me. Please don't ever hesitate to contact me if you have a question, suggestion, or concern.

I'm also on the forums throughout the day:

http://forum.unity3d.com/threads/motion-controller.229900

Tim

tim@ootii.com

Contents

Overview

The Motion Controller is an animation framework and character controller for any character and any game. It is an “input based” controller that puts user input and responsiveness above physics.

Unlike other third person humanoid controllers on the Asset Store, the Motion Controller is a flexible component-based framework that can support any type of character from humans to cats and aliens to race cars. It was built from the ground up to be extensible and allows you to create your own motions, use your own animations, and even share motions with others.

Out of the box, the Motion Controller includes the Actor Controller; an advanced character controller that supports gravity, moving platforms, custom character shapes, and walking on walls. It also includes several prebuilt motions that you can use for your humanoid characters, allowing them to run, jump, climb ladders, scale wall, and more.

Features

The Motion Controller supports the following features:

- Walk on walls, ceilings, etc.
- Move and rotate on platforms
- Move super-fast
- Create custom body shapes
- Setup humanoids in less than 15 seconds
- Supports Nav Mesh Agents
- Supports root-motion
- Supports any input solution
- Includes code (C#)

Introduction

The Motion Controller is composed of a couple different pieces. Understanding how these pieces fit together will make using and extending the framework much easier.

Input Based

Character controllers typically come in two flavors: input-based and physics-based.

Input based controllers take input from users and code and turn that into precise movement. They make for more precise and responsive character controllers. They can simulate the reaction to physical forces, but aren't as accurate as physics based controllers.

Physics based controllers use physics forces to move the character and tend to be better for physics based simulations (ie billiard balls in a game of pool). However, they tend not to be as responsive to user input and movement isn't as precise.

The Motion Controller (and Actor Controller) are input based controllers.

Actor Controller

This is the actual "character controller" that handles movement, sticking to moving platforms, collisions, gravity, walking on walls, etc.

Basically, the Actor Controller knows 'how' to do these things, but he doesn't know when to do them or why. That's where motions come in.

For more about the Actor Controller, check out the User Guide:
http://www.ootii.com/Unity/ActorController/ACGuide.pdf

Input Sources

If everyone used the same input solution, we wouldn't need this. However, some people use Unity's native input solution, some people like Easy Input, and some people use other assets found on the Asset Store.

That means we have to have a generic way of getting input so motions can be coded once and grab user input from any solution. That's what an Input Source is; a generic way of getting input. By wrapping your preferred input solution in an "input source", the MC and motions themselves can use the input source to tap into your chosen input solution.

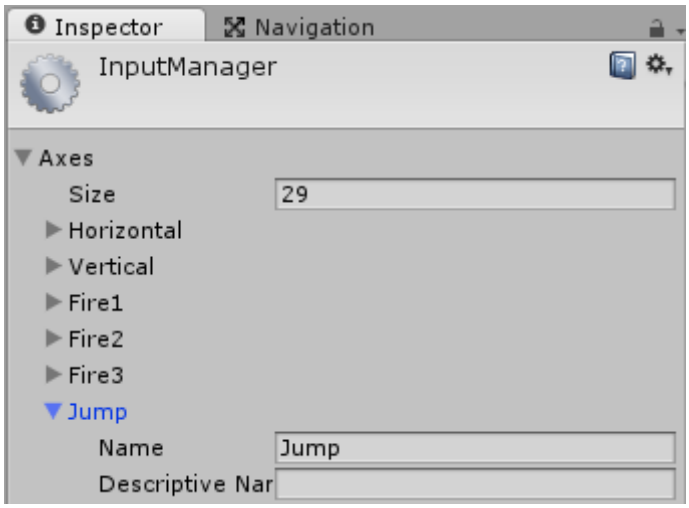
The Motion Controller includes the 'Unity Input Source' which knows how to grab user input using Unity's native solution. Easy Input includes an 'Easy Input Source' and you can create other 'Input Sources' to work with your favorite solution as needed.

Click here to learn more about creating Input sources.

Action Aliases

Throughout the motions, I use 'action aliases' to check if input has occurred and if it's time to activate a motion. An action alias is just a friendly name that's given to a key press or input condition that allows us ask about it without understanding how it works.

Let's look at an example…



The Jump motion has an action alias property whose default value is "Jump". When the motion looks to see if it's time to jump, it makes a call like this:

```
bool lShouldActivate = InputSource.IsJustPressed("Jump");
```

The motion doesn't really care how "Jump" was processed. It just cares that a value of `true` or `false` comes back.

This is exactly what Unity does with its input system (look at the picture to the left).

Motions

Motions are about animation flow. They don't replace the Mecanim Animator, but work with it to control when animations start, when they transition, and when they end. As such, motions typically have three components: animations, code, and settings. They work together to create the full experience.

Think about climbing. You can have a climbing animation, but how do you determine 'what' you can climb? What about climbing ladders that don't have a fixed height… how do you know when to stop? A motion would include the ladder climbing animations, the code to answer these questions, and setting to allow the developer to set information.

Priorities

Every motion is given a priority. We use this priority to determine what motion overrides another… the higher the priority, the more important the motion.

So, if two motions could both be activated, the one with the highest priority wins.

If two motions could both be activated and have the same priority, the one lowest on the list wins.

Motion Layers

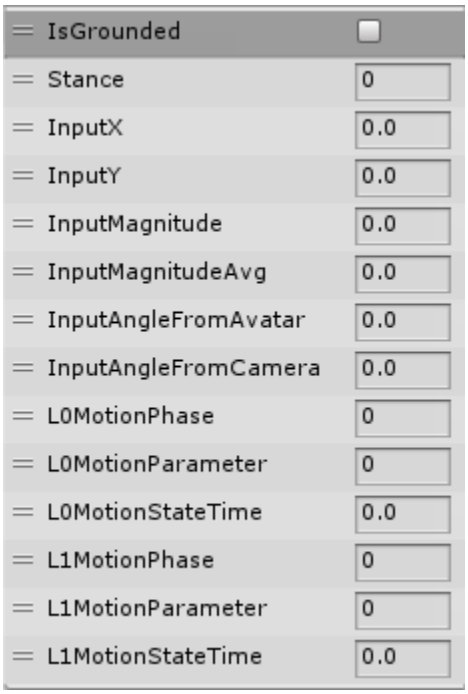
Like Mecanim Animator Layers, we can group motions into layers. This way we can have multiple motions running at once in order to affect different parts of the body. Typically, I work in one or two layers, but you can create up to 10 without having to change any of my code.

To learn more about layers, check out the Punch question in the FAQ.

Mecanim Animator

Under the hood, the MC controls animations by using Mecanim's Animator and Animator Parameters. It simply sets the appropriate parameters and allows the Mecanim state machines to control the animation flow in typical Unity fashion.

To do this, the following parameters are required in your animator:



IsGrounded – Determines if the character is on the ground

Stance – Developer defined integer defining the character state

InputX/InputY – Value of the "movement" input keys/gamepad

InputMagnitude – Value of the movement input

InputMagnitude – Averaged value of the movement input

InputAngleFromAvatar – Angle difference between the input and the character's forward

InputAngleFromCamera – Angle difference between the input and the camera's forward

For each animator layer in your controller, a set of these parameters should exist. They would be labeled as "L0…", "L1…", "L2…", etc.

L<index>MotionPhase – Unique key that activates animator states

L<index>MotionParameter – Additional information to help with the animator states flow.

L<index>MotionStateTime – Normalized time in the current state. The value will always be 0 to 1 (inclusive). If the state loops, the value goes back to 0.

Humanoid Animator Controller

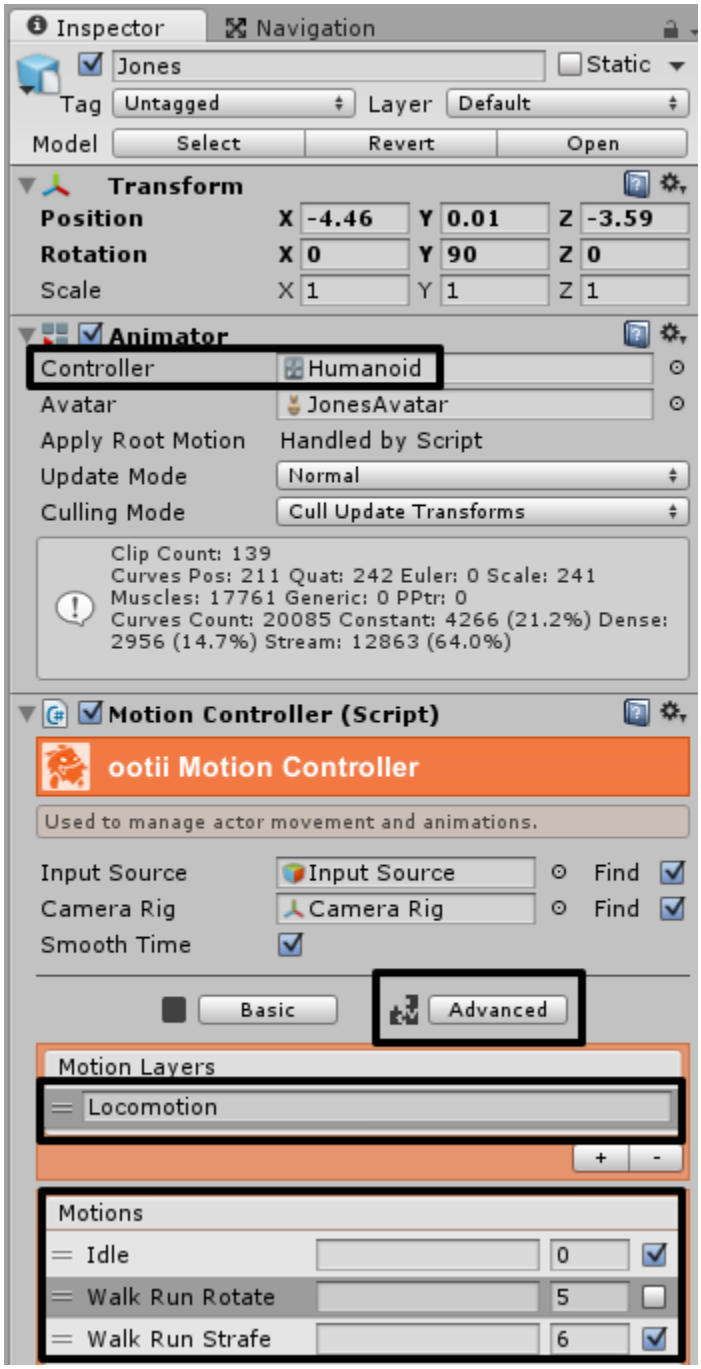
The MC includes a pre-built animator controller for demos and for you to copy. It can be found here:

Assets\ootii\MotionController\Content\Animations\Humanoid\Humanoid.controller

When customizing the animator controller, it's best that you copy this controller and use it with your characters. This way I won't overwrite it during updates.

Initial Setup

Following the Quick Setup on page 3, your character's inspector would look something like this:



The Animator's Controller property should reference the MC's "**Humanoid**" controller. This is the Mecanim controller that contains the animations for walking, running, jumping, climbing, etc.

On the **Advanced** tab, a "**Locomotion**" layer will exist that contains a number of **motions**. These motions are what cause your character to run, jump, climb, etc.

You'll learn later that you can add motions to the list to enable new abilities and even create your own motions.

Custom Input Sources

Remember, the input source that I've included is optional. You can use any input solution you want by creating an input source.

An input source is just a class that implements the IInputSource interface. By implementing the interface, your class promises to implements specific functions like "IsPressed" and "MovementX". It's these functions and properties that the Motion Controller and motions will use to tap into your input solution.

This video will walk you through input sources in more detail:

https://www.youtube.com/watch?v=2v0ZMyvgP4Y

Quick Coding

For example, the **UnityInputSource.cs** file is an input source that comes with the MC and that is used to tap into Unity's native input solution. You'll find it here in the following folder:

Assets\ootii\Framework_v1\Code\Input

Inside that class, you'll find functions like this:

```
public virtual bool IsJustPressed(KeyCode rKey)
{
    if (!_IsEnabled) { return false; }
    return UnityEngine.Input.GetKeyDown(rKey);   // <-------------
}
```

To create your own input source, you can copy UnityInputSource.cs, rename it to something like YourAwesomeInputSource.cs, and change the class name. Now, you can just change the function contents as needed for your input source.

In the previous code, we're just tapping into Unity's "UnityEngine.Input.GetKeyDown" function. In your input source, you'll code the function contents using your input solution. Maybe it would look like this:

```
public virtual bool IsJustPressed(KeyCode rKey)
{
    if (!_IsEnabled) { return false; }
    return YourAwesomeInputSolution.IsKeyPressedThisFrame(rKey);    // <-------------
}
```

As you can imagine, there are a lot of different input solutions on the Asset Store. So, I can't buy and implement them all. Hopefully, the video and this section will help you create the input source you need. If you're willing to share, I'm happy to put it on the Vault.

IInputSource Interface

For developers creating their own Input Sources, remember to implement the IInputSource interface. Once you do this, you can fill in the properties and functions based on the input solution you're using.

Collisions & Triggers

The Motion Controller uses the Actor Controller as its underlying character controller. That means it's the Actor Controller that handles collisions.

To learn more, check out the Actor Controller's Guide.

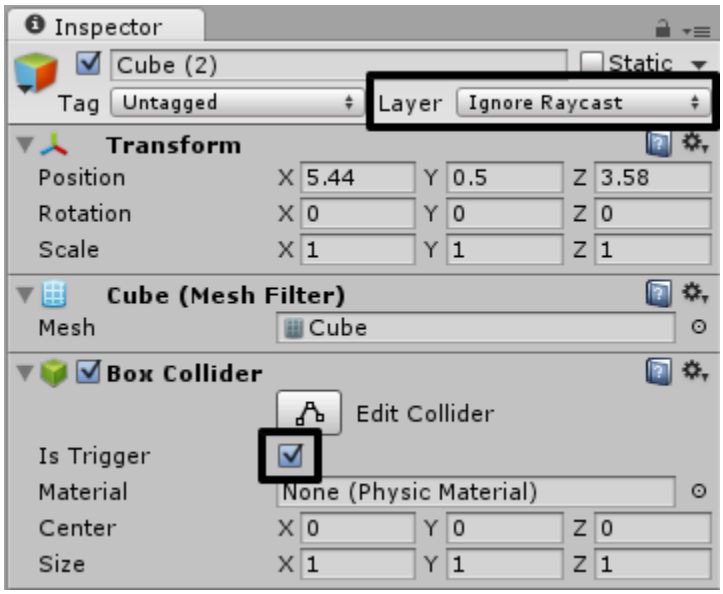
Triggers

The AC (and in-turn the MC) does use Unity's physics engine (PhysX). However, it doesn't have any Unity colliders by default. Instead, I use "Body Shapes" to support different size characters and characters that aren't basic capsules.

That means the MC won't raise trigger events by itself. To have the character raise trigger events, you need to add a Unity collider to your character as you normally would. This just follows Unity's standard procedure.

Colliding with Trigger Volumes

The MC (and AC) uses Unity raycasts to manage collisions. However, Unity's raycasts **DO** collide with trigger volumes. That means your character will collide with colliders even if the colliders have "Is Trigger" checked.



To prevent your character from colliding with trigger volumes, simply set their layer to Unity's "Ignore Raycast" layer.

User Interface

I've broken the user interface into two sections: basic and advanced. For humanoid characters, the basic view provides most of what you need. However, if you're creating your own motions, want to tweak values, or have motions for non-humanoids, you'll want to use the advanced view.

Basic View

You can short-cut most of the setup process just by clicking one of the movement style buttons. By doing that, the Motion Controller will create the input source, setup cameras, and add motions as needed.

Movement Styles

To support different third person style gameplay, I've included different "Walk Run" motions that mimic popular games / game genres.

Walk Run Rotate – MMO Style (think World of Warcraft)

Walk Run Pivot – Adventure Style (think Tomb Raider or Shadows of Mordor)

Walk Run Strafe – Shooter Style

If one of these motions doesn't fit your needs, you can use these motions as a reference and create a custom motion that does. That's the power of the Motion Controller.

Options

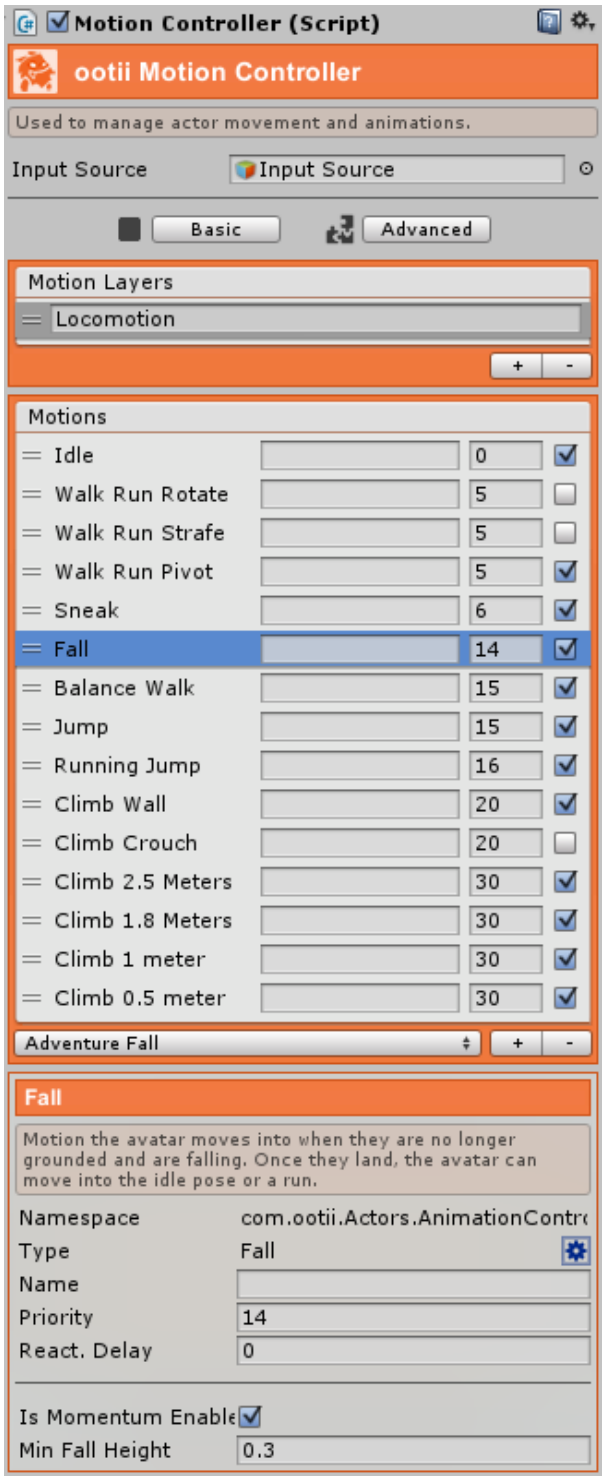
The options section provides a quick way to add the prebuilt motions I've included.

Just check the motions you want and they will be added with the default settings.

With some motions, you'll need to specify what Unity layers identify the objects that can be acted on.

Advanced View

Use this view to customize motion properties, enable motions you've created, or use animations with the Simple Motion feature.



Motion Layers

Add, remove, and name layers as needed. Each layer you add can hold a collection of motions.

The number of layers should match the number of layers you use in the Mecanim Animator.

To learn more about layers, check out the Punch question in the FAQ.

Motions

When you select a Motion Layer, you'll be presented with the list of motions it contains.

In the list, you'll see:

- Motion Type
- Motion Name (text field)
- Motion Priority (number field)
- Enabled flag (checkbox)

You can also use the icon on the left of the row to rearrange the order. This is helpful when motions have the same priority value. Remember… the lowest one wins.

Motion Details

When you select a motion, the detail section under the list will open up.

The top part (that has the description) is generic stuff. Here you can add a friendly name (which can also be searched for) and change the priority.

The Reactivation Delay is the number of seconds that have to wait before the motion can activate again. 0 means there is no delay.

The bottom part (under the line) are motion specific properties. All of them have tooltips to help. Just hover over the property name and it will describe what the property does.

Motion Controller Events

To help support integration with your code, the Motion Controller exposes three events that you can tap into:

MotionActivated

This function will fire when a motion's "Activate" function is called. It includes the layer the motion is on, the motion itself, and the old motion that will be deactivated.

MotionUpdated

This function will fire after a motion's "Update" function is called. Since a motion can deactivate itself, you may see this called once AFTER the MotionDeactivated event is called. That's expected as the Update event called Deactivated and then finished.

MotionDeactivated

This event will fire when the motion's "Deactivate" function is called. It includes the layer the motion is on and the motion itself.

The MotionDeactivated event can fire before or after the MotionActivated event based on how the current motion is being deactivated.

Example

The following is a simple example of how you can tap into the events:

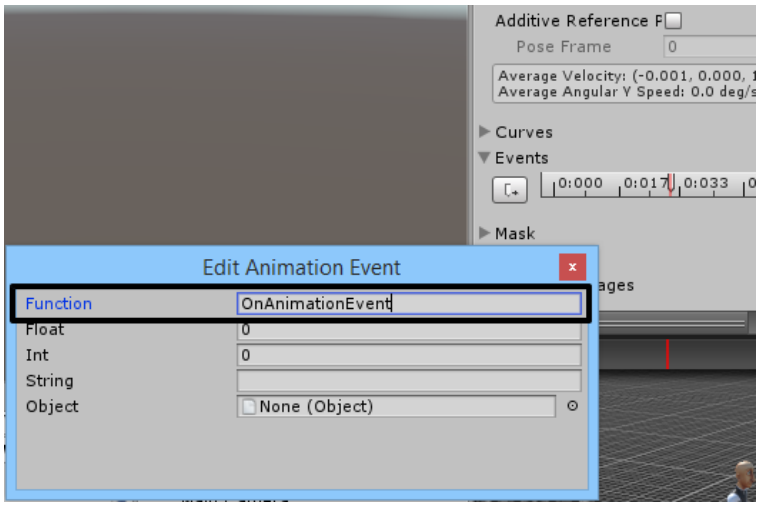
```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
{
    public MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
        mMotionController.MotionActivated += MotionActivated;
        mMotionController.MotionUpdated += MotionUpdated;
        mMotionController.MotionDeactivated += MotionDeactivated;
    }

    protected void MotionActivated(int rLayer, MotionControllerMotion rNewMotion, MotionControllerMotion rOldMotion)
    {
        Debug.Log(string.Format("Activated m-new:{0} l:{1}", rNewMotion.GetType().Name, rLayer));
    }

    protected void MotionUpdated(float rDeltaTime, int rUpdateCount, int rLayer, MotionControllerMotion rMotion)
    {
        Debug.Log(string.Format("Updated m:{0} l:{1} dt:{2:f2}", rMotion.GetType().Name, rLayer, rDeltaTime));
    }

    protected void MotionDeactivated(int rLayer, MotionControllerMotion rMotion)
    {
        Debug.Log(string.Format("Deactivated m:{0} l:{1}", rMotion.GetType().Name, rLayer));
    }
}
```

Animation Event

Unity allows you to raise events at specific times when animations run. To learn how to do this, head over to Unity's Animation Events on Imported Clips page.

Editor

The key to using this feature with the Motion Controller is to set the `Function` property to be "OnAnimationEvent":



You can also fill out the Float, Int, String, and Object parameters if you want.

At run-time, the Motion Controller will catch the events and bubble them up to the active motion.

Code

To respond to the event, you just need to add the OnAnimatorEvent function to your custom motion:

```
/// <summary>
/// Raised by the animation when an event occurs
/// </summary>
public override void OnAnimationEvent(AnimationEvent rEvent)
{
}
```

You can than interrogate the AnimationEvent parameter and grab the values that you set in the editor window.

With this approach, you're now inside your motion when the event fires. So, you can access motion variables, access the Motion Controller or Actor Controller, change the animation state, etc. What you do with the event is totally up to you.

Multiplayer Networking

While I haven't done any multiplayer work or networking myself, other users have successfully used the Motion Controller with UNet, Bolt, Photon, and other multiplayer solutions.

You'll need to integrate the MC with your specific networking solution as you see fit. However, I've added some components to help with UNet. You'll find them under 'Assets/ootii/Framework_v1/Code/Networking'.

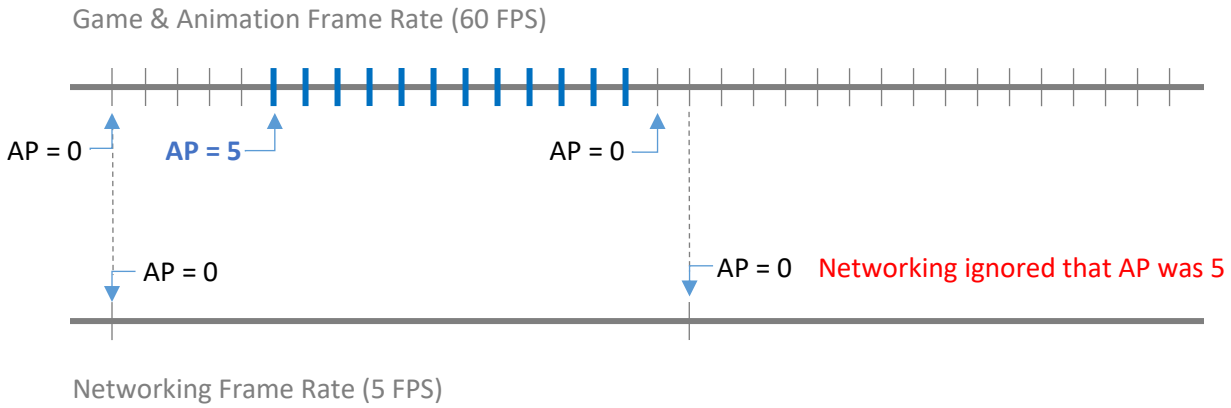
I've also created a video using UNet that you may find helpful:

https://www.youtube.com/watch?v=7V2bYcOBca0&list=PLw7Y7lUmqrILegveIWo5mYkGIv1Z1pQr5&index=18

Fast Changing Animator Parameters

Based on user feedback, we've found that some networking solutions cannot handle "fast changing animator parameters".

For example, let's assume we have an animator parameter named "AP":



In the example above, you can see that the slower networking frame rate totally missed that the animator parameter (AP) was once set to 5. That means the remote client can't use the animator parameter to enter a Mecanim transition.

We would expect that networking solutions would cache the animator parameter changes between the ticks and then send the parameter... but, some don't.

In my BasicActorNetworkSync component, you'll see how I do it. This works for UNet and you'll need to apply similar logic to your networking solution if they don't' support fast changing animator parameters.

General Multiplayer Networking

For more information on multiplayer networking, please see Unity's documentation:

https://unity3d.com/learn/tutorials/topics/multiplayer-networking

Prebuilt Motions

All prebuilt motions use the 'Humanoid' Animator Controller that is included in the package. This controller contains animations that are specific to humanoid characters.

Idle

Idle is the default for all motions. When nothing is happening, the Idle motion kicks in.

It's a bit more complicate than you'd expect since rotations can happen while the character isn't doing anything and this motion supports all the movement styles.

Walk Run Rotate

This motion closely allows for an 'MMO style' control and is responsible for walking, running, strafing, and rotating while the character is moving.

Using World of Warcraft as a model, the controls are as follows:

W key	= move forward
S key	= move backwards
A key	= rotate left
D key	= rotate right
Q key	= strafe left
E key	= strafe right
Space bar	= jump/climb
Left Shift	= run
RMB down	= rotate character
LMB down	= rotate camera

Walk Run Pivot

This motion closely allows for an 'Adventure style' control and is responsible for walking, running, pivoting, and rotating while the character is moving.

This approach supports pivoting when input causes an immediate reversal of direction. It also allows the character to pivot while standing still. To do this, simply tap the WASD keys.

Using Shadows of Mordor as a model, the controls are as follows:

W keys	= rotates and moves forward (away from the player)
S key	= rotates and moves forward (toward the player)
A key	= rotates and moves left (relative to the camera)
D key	= rotates and moves right (relative to the camera)
Space bar	= jump/climb
Left Shift	= run
RMB down	= rotate character

Walk Run Strafe

This motion closely allows for a 'Shooter style' control and is responsible for walking, running, strafing, and rotating while the character is moving.

This approach supports strafing using the AD keys and always keeps the camera behind the character.

The controls are as follows:

W key	= move forward
S key	= move backwards
A key	= strafe left
D key	= strafe right
Space bar	= jump/climb
Left Shift	= run
RMB down	= rotate character and camera

Jump

The Jump motion provides a physics based jump that launches the character in the air. Unlike a static animation, the height is determined by the 'Impulse' property and the character will animate correctly no matter how long they are in the air.

The character will also recover gracefully if they jump onto an object at any point within the motion.

Impulse

The amount of instantaneous force to apply. The character's mass (set in the Actor Controller) is used to determine the final height the jump can reach.

Convert To Hip Base

Typically a person's jump arc is based on their center of mass (hips). However, to support jumping onto objects, we use the feet as the root of the character. This options uses math to offset the root position based on the distance between the hips and the feet. It provides a cleaner look in most animations.

Hip Bone

If 'Convert To Hip Base' is used, this is the name of the hip bone in the character's hierarchy. If no value is set, we'll attempt to find the bone automatically.

Running Jump

The Running Jump motion is also a physics based jump, but has a different look. It mimics more a running long jump than the original Jump motion.

Fall

The Fall motion uses a subset of the Jump motions animations to simulate falling.

When the ground distance exceeds the 'Min Fall Height' property, the fall motion will kick in. The actual speed of the fall and physics is controlled by the Actor Controller.

Balance Walk

This motion mimics a balance beam or tight-rope walk. The animations look like the character is wobbling left and right as they move forward and backwards.

Activating the motion can be done based on Beam Width properties or by specifying a layer. If the widths are used, I'll attempt to find the beam. However, sometimes edges and corners of other objects may trigger the motion. When a layer is used, the layer will need to be specified.

Slide

Slide is a motion that will trigger when the character is on a slope. When activated, it will have the character go into a 'balancing' pose as they rotate towards the direction of the slope.

Sneak

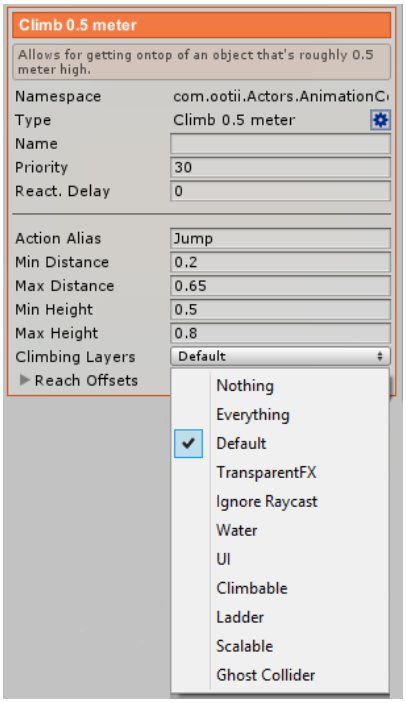
Sneak is similar to the Walk Run Strafe motion, but the animations look more like someone is stalking or sneaking.

Climb 0.5 Meters, Climb 1 Meter, Climb 1.8 Meters, Climb 2.5 Meters

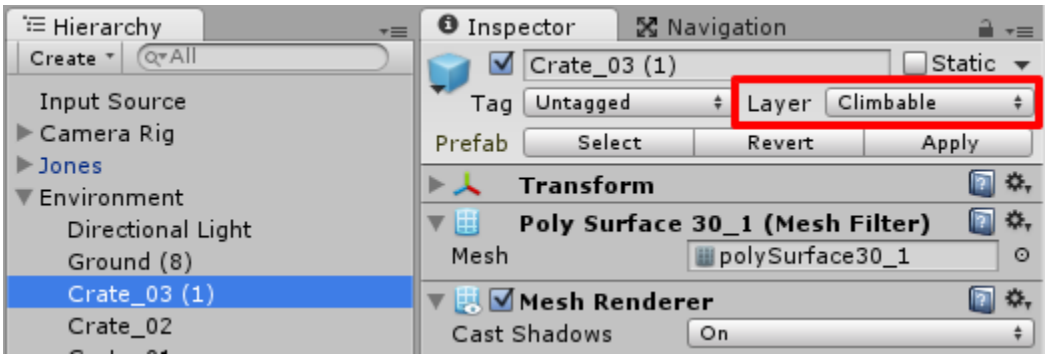
These motions allow the character to step/climb onto obstacles based on the height. Each of the climbs is a complete animation that takes the character from the bottom to the top in one cycle.

Climbing Layers

The 'Climbing Layers' property is used to determine what can actually be climbed.



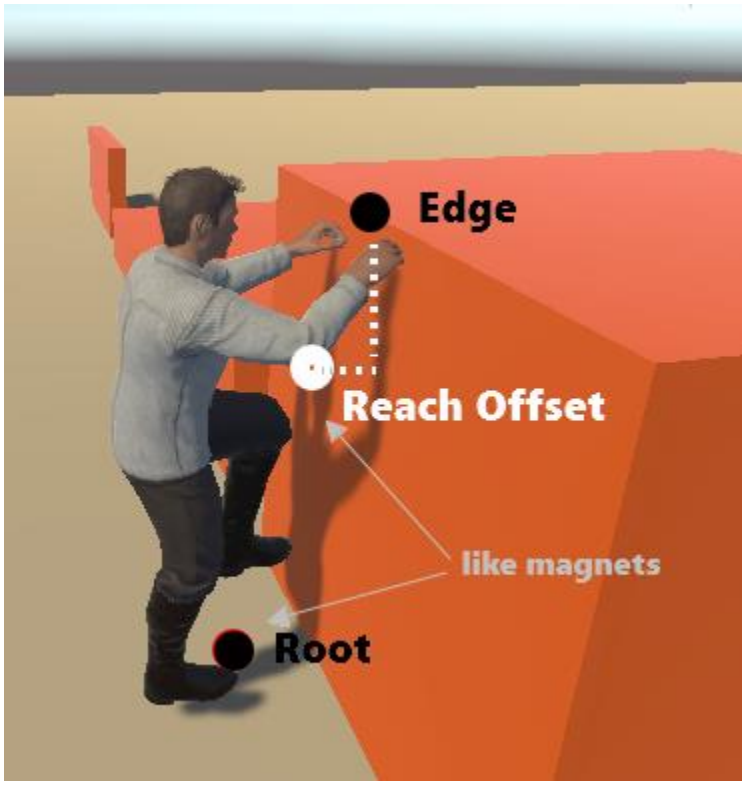
When the motion does a test to see if the obstacle is able to be climbed, it compares the obstacle's 'Layer' value to this property. If they match, the climb test can continue. If not, the climb is aborted.



Reach Offsets

When the player attempts to climb, there's no guarantee they will start from the exact same spot every time. We also don't know the exact height of the object being climbed. So, we use "reach offsets" to help pull the character to a specific point during the climb.

We can actually have several reach offsets during one climb. This way, we can pull and tug the character while allowing the animation's root motion to do the majority of the work.



Using reach offsets, we can ensure the character doesn't float off the obstacle or get too close to it.

If your character is a different size, you can also use the reach offsets to ensure they line up nicely with the obstacle they are climbing.

Think of them like magnets. The edge is the point we found while looking for something to climb and the reach offset is the offset from that edge. The root is the character's root. As the character animates, its root moves due to root-motion. We then add some extra movement to pull the reach offset and root together.

Since each of the climb motions has a different animation, each one has different reach offsets.

Climb 0.5 meters, Climb 1 meter Reach Offset Properties

End actor up – Height offset from the edge the character will end at.

End edge normal – Forward offset (from the edge normal) we will end at.

Climb 1.8 meters Reach Offset Properties

Start actor – Position offset from the edge, based on the character's rotation, that he will start at.

Start edge – Position offset from the edge, based on the edge, that he will start at.

Mid edge – Position offset from the edge, based on the edge.

End edge – Position offset from the edge, based on the edge, that he will end at.

Climb 2.5 meters Reach Offset Properties

Stand start actor – Position offset from the edge, based on the character's rotation, that he will start at.

Stand start edge – Position offset from the edge, based on the edge, that he will start at.

Stand mid edge – Position offset from the edge, based on the edge.

Stand end edge – Position offset from the edge, based on the edge, that he will end at.

Leap start actor – Position offset from the edge, based on the character's rotation, that he will start at.

Leap mid actor – Position offset from the edge, based on the character's rotation.

Leap mid actor – Position offset from the edge, based on the character's rotation.

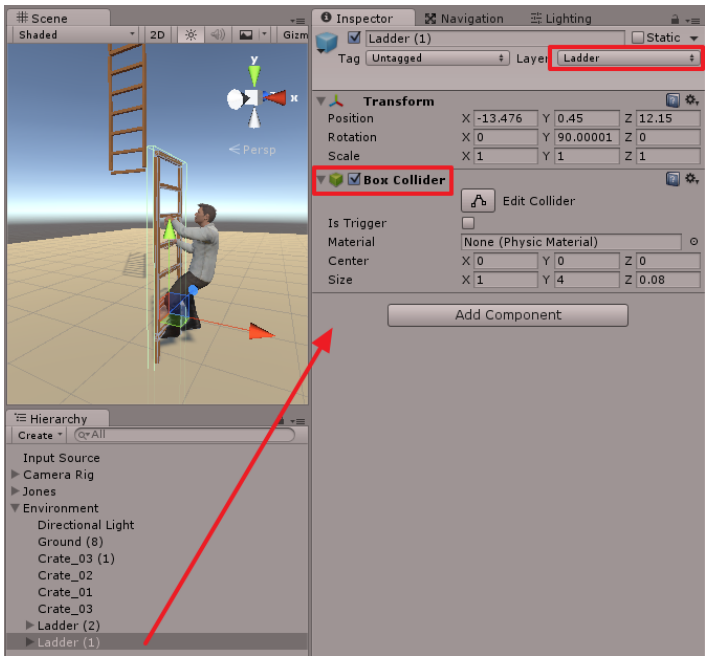
Leap end edge – Position offset from the edge, based on the edge, that he will end at.

Climb Wall, Climb Ladder

These motions are very similar. The only real differences are the animations themselves. Climb Wall looks like spider-man climbing a wall while Climb Ladder looks like someone…well…on a ladder.

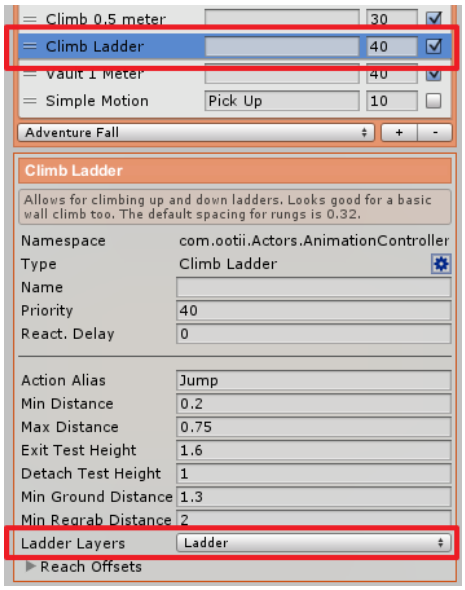
Both allow you to start from the ground, from the top, or mid jump.

Both use the layer property to determine what can be climbed. Typically, we'd use two different layers; one for to flag walls and one to flag ladders.



Setting up the ladder or climbable surface is just a matter of adding a box colliders and setting the layer.

The thing with ladders is that the rungs and mesh don't actually matter. They are just there for looks. What we really do is have a parent object that is the ladder. It's this object that we put the box collider on and set the layer.



With the ladder setup, it's just a matter of making sure the "Climb Ladder" motion is added to your character and the "Ladder Layers" matches the layer you set for your ladder.

Both support the same reach offsets. However in this case, the 'edge' isn't always the 'edge'. When starting on the ground, it's the hit our forward ray captured at 'min height'.

Reach Offset Properties

Bottom start edge forward – Position offset from the edge, based on the hit's normal, that he will start at.

Top start actor up – Position offset from the edge, based on the actor up, that he will start at.

Top start edge forward – Position offset from the edge, based on the edge normal, that he will start at.

Top mid actor up – Position offset from the edge, based on the actor's up.

Top mid edge forward – Position offset from the edge, based on the edge normal.

Top end actor up – Position offset from the edge, based on the actor's up, that he will end at.

Top end edge forward – Position offset from the edge, based on the edge normal, that he will end at.

Climb Crouch

The climb crouch allows the character to hand onto the edge then pull up to the top of a wall or shimmy left and right. In order to get into the hanging-crouch, the character would either jump up to the ledge of catch it during a fall.

Like the other climbs, this motion uses the Climbing Layers property to determine what can be climbed.

Vault 1 Meter

Similar to the Climb 1 Meter motion, but this motion has the character vault over the thinner object. Think of something like fence. It has height, but no depth. So, if the character is moving forward, they would really leap over it.

Like the climbs, it has a Climb Layers and Reach Offset properties.

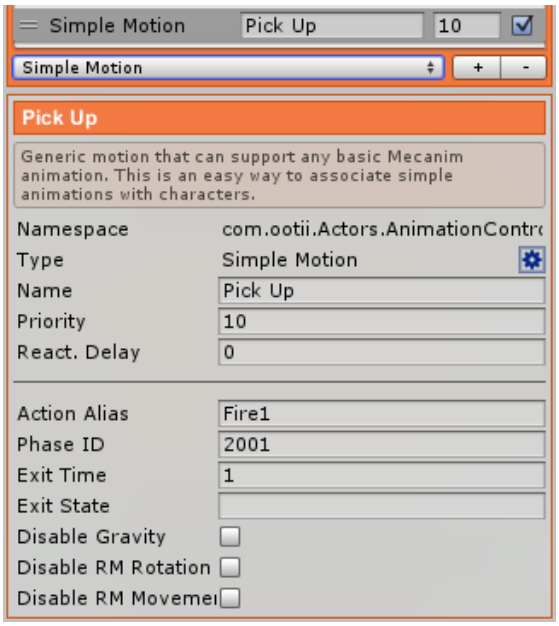
Reach Offset Properties

Start actor – Offset from the edge relative to the actor rotation.

Start edge – Offset from the edge relative to the edge rotation.

Simple Motion

This is a powerful motion that allows you to hook up your own animations and create motions without writing code. You can have multiple Simple Motions setup and each of them can point to a different animation or set of animations.



There three core parts to each Simple Motion:

1. Action Alias

Similar to other action aliases, this is the friendly name of the input we'll test to determine when to activate the motion.

In the example to the left, 'Fire1' is Unity's alias for the left mouse button.

2. Phase ID

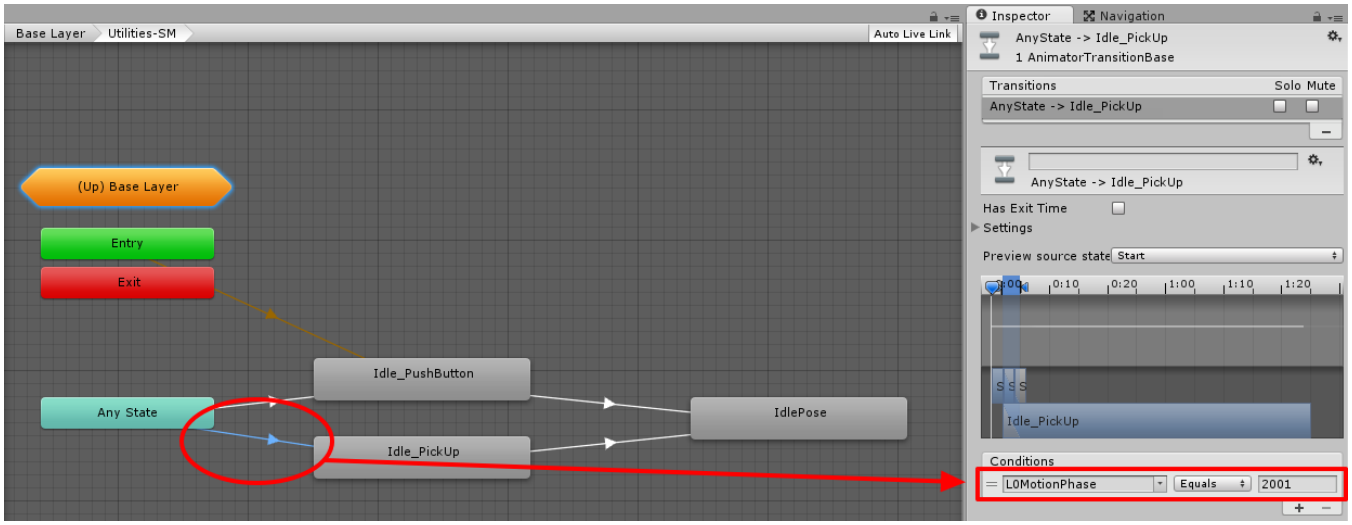
The phase ID is the unique identifier that is used as Mecanim transition condition. It will tell the Mecanim Animator when to start playing the animation tied to this motion..

3. Exit Time (and optional Exit State)

By default, the motion will deactivate when the animation reaches the end (a normalized time of '1'). However, you can change this time in order to have the motion exit early.

If needed, you can also enter the full path of the state that represents the end of the animation set. If no state path is entered, it's assumed the enter state is the exit state.

Pick Up Example



In this example, you can see the we have a transition that goes from 'Any State' to our animation we want to trigger. The condition matches what we set for the Phase ID.

Because no Exit State was set, when Idle_PickUp finishes (a normalized time of '1') the motion will deactivate.

Accessing the MC through Code

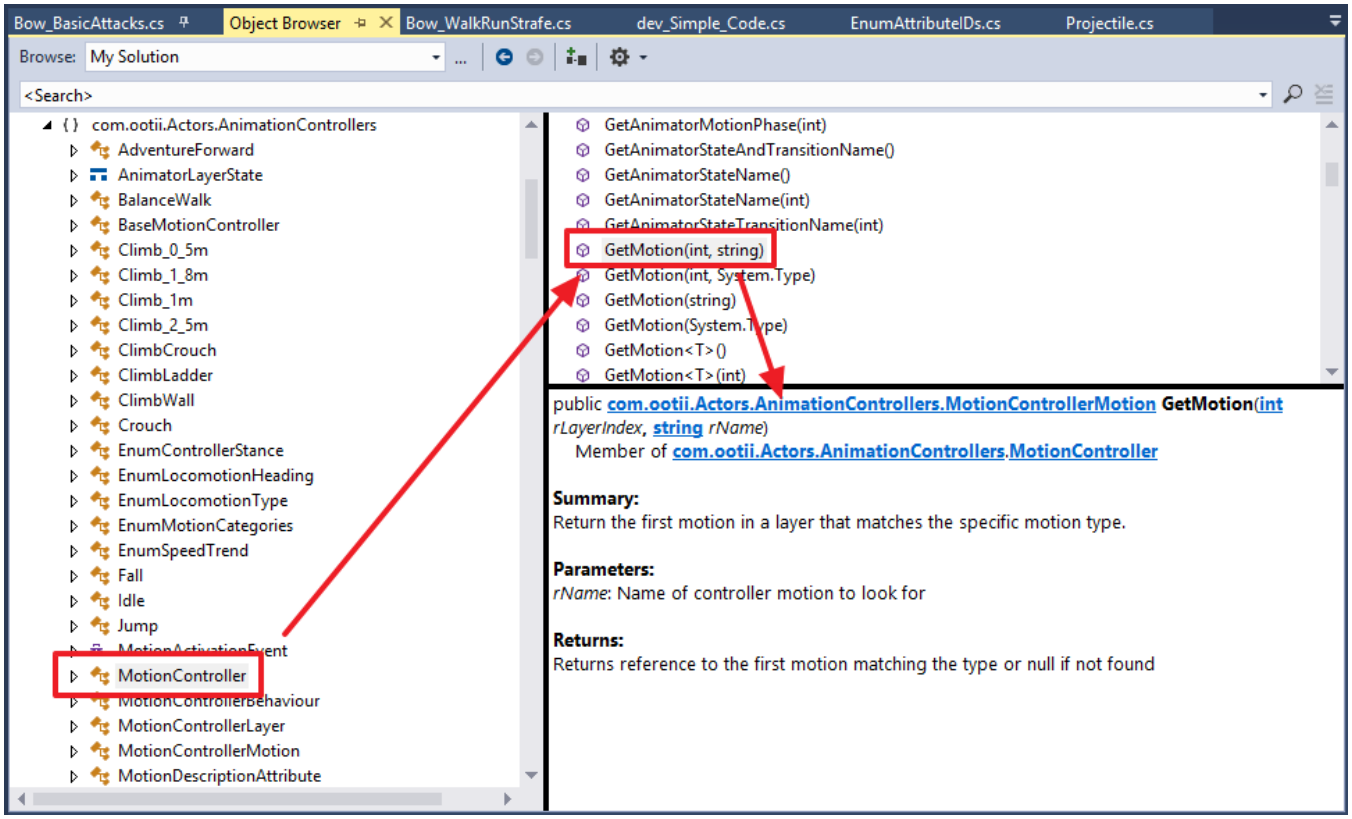
Start() function

Currently, the Motion Controller initializes the motions during its Start() function. Since Unity doesn't guarantee the order that Start() functions are called, if you try to access a motion in the Start() function of your MonoBehaviour the results will be unpredictable.

While I'm looking to move my initialization code to Awake(), it's best to NOT get motions or do motion specific work in the Start() function of your class.

Object Browser

Pretty much everything you can do to the MC through the editor, you can do through code. The best resource is to actually your IDE's object browser. That will allow you to see all the functions that are available on the MC as well as comments I've made.



Another good resource is to look at the MotionControllerEditor.cs file. This file contains most of the logic for the MC inspector. So, it shows how to call functions based on how I do it in the editor.

Tapping Into Events

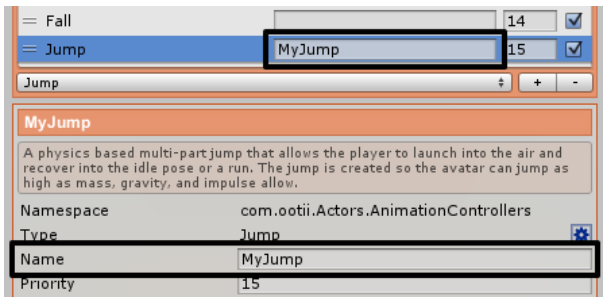
The MC allows you to setup events to catch when motions are activated, updated, or deactivated. Use these to extend the capabilities of the motions without modifying the motion itself.

```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
{
    public MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
        mMotionController.MotionActivated += MotionActivated;
        mMotionController.MotionUpdated += MotionUpdated;
        mMotionController.MotionDeactivated += MotionDeactivated;
    }

    protected void MotionActivated(int rLayer, MotionControllerMotion rNewMotion, MotionControllerMotion
rOldMotion)
    {
        Debug.Log(string.Format("Activated m-new:{0} l:{1}", rNewMotion.GetType().Name, rLayer));
    }

    protected void MotionUpdated(float rDeltaTime, int rUpdateCount, int rLayer, MotionControllerMotion rMotion)
    {
        Debug.Log(string.Format("Updated m:{0} l:{1} dt:{2:f2}", rMotion.GetType().Name, rLayer, rDeltaTime));
    }

    protected void MotionDeactivated(int rLayer, MotionControllerMotion rMotion)
    {
        Debug.Log(string.Format("Deactivated m:{0} l:{1}", rMotion.GetType().Name, rLayer));
    }
}
```

Getting Motions by Name or Type

You can access motions by name or by type. Doing it by name allows you to have multiple motions of the same type, but setup different properties for each. Just ensure the name field in the motion property is filled out.

The numeric parameter in the GetMotion() function is the index of the layer the motion is on, starting at index 0.

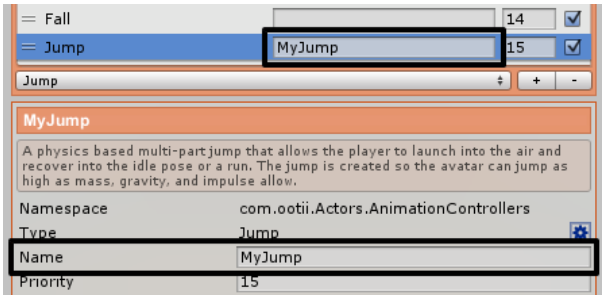



```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
```

```
{
    protected MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
    }

    void Update()
    {
        MotionControllerMotion lMotion = mMotionController.GetMotion(0, "MyJump");
        MotionControllerMotion lMotion2 = mMotionController.GetMotion<Jump>(0);
    }
}
```

Enabling Motions by Name

You can access motions by name or by type. Doing it by name allows you to have multiple motions of the same type, but setup different properties for each. Just ensure the name field in the motion property is filled out.




```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
{
    protected MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
    }

    void Update()
    {
        MotionControllerMotion lMotion = mMotionController.GetMotion(0, "MyJump");
        lMotion.IsEnabled = true;
    }
}
```

Accessing Motion Specific Properties

You can access motions by name or by type. Once you have the motion (using GetMotion), you can simply cast the motion to the type you're interested in. Then, get or set the properties as you would any other object.

```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
{
    protected MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
    }

    void Update()
    {
        Jump lJumpMotion = mMotionController.GetMotion(0, "Jump") as Jump;
        float lJumpImpulse = lJumpMotion.Impulse;
    }
}
```

Adding Motions at Runtime

You can add and remove motions at runtime by creating motions as needed. Each motion will go on a layer, so that needs to be defined first.

```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
{
    protected MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
    }

    void Update()
    {
        if (UnityEngine.Input.GetKeyDown(KeyCode.Insert))
        {
            Jump lNewJump = new Jump();

            MotionControllerLayer lLayer = mMotionController.MotionLayers[0];
            lLayer.AddMotion(lNewJump);
        }
    }
}
```

Activating Motions

To activate your motion, just grab the motion you're interested in and tell the MC to activate it.

```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
```

```
{
    protected MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
    }

    void Update()
    {
        if (UnityEngine.Input.GetKeyDown(KeyCode.Space))
        {
            MotionControllerMotion lMotion = mMotionController.GetMotion(0, "MyJump");
            mMotionController.ActivateMotion(lMotion);
        }
    }
}
```

Moving

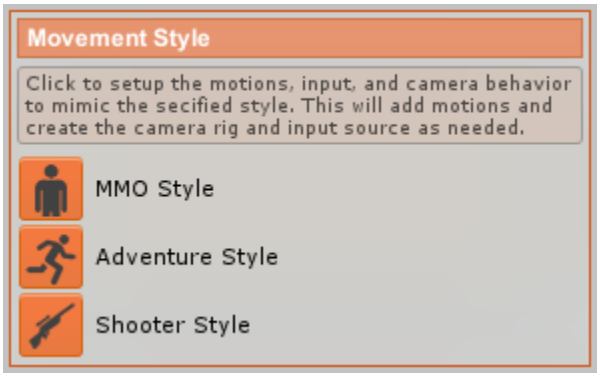
Having an NPC walk to a target typically requires some AI to determine which target, how close you are to the target, and how quickly the NPC should move. I prefer Node Canvas, but you can use any AI solution.

Once you have the basic information, you can control a character by using the SetTargetPosition() function.

```
using UnityEngine;
using com.ootii.Actors.AnimationControllers;

public class dev_Simple_Code : MonoBehaviour
{
    public Vector3 Target = new Vector3(0f, 0f, 10f);
    protected MotionController mMotionController = null;

    void Start()
    {
        mMotionController = gameObject.GetComponent<MotionController>();
    }

    void Update()
    {
        if (UnityEngine.Input.GetKeyDown(KeyCode.Space))
        {
            mMotionController.SetTargetPosition(Target, 1f);
        }

        float lDistance = Vector3.Distance(transform.position, Target);
        if (lDistance < 0.5f)
        {
            mMotionController.ClearTarget();
        }
    }
}
```

Setting Movement Styles at Runtime

Each movement style that I show in the editor is really just a collection of motions that I enable and disable.



If you want to expose this capability to your players, you really just want to mimic what I do at edit time. That means adding motions to the MC, disabling motions, and enabling motions.

In addition, you can change the camera type as I do with the 'Shooter Style'.

To see what I'm doing, open the MotionControllerEditor.cs file and look for the following functions:

EnableMMOStyle()
EnableAdventureStyle()
EnableShooterStyle()

Each of these functions does a couple of things:

1. Sets up the Input Source if it isn't already
2. Determines if some features (like the AdventureRig) are available
3. Sets up the camera if it isn't already
4. Adds a motion layer if needed
5. Creates (or enables) motions as needed
6. Disables motions as needed

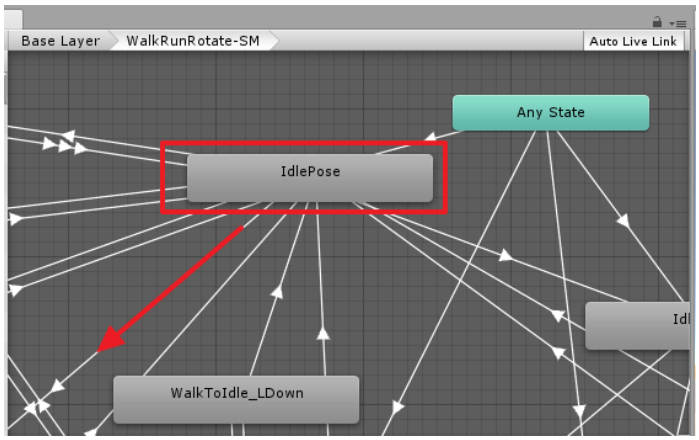
At runtime, really it's #5 and #6 that you'll deal with. I've created some helper functions; CreateMotion() and DisableMotion(). You may want to look at them and do the same thing in your runtime code.

FAQ

How can I add additional idle animations?

The first thing we need to do is talk about “poses” vs. “animations”. Think of a pose as a really short animation; one that’s really only a frame or two.

I use the “IdlePose” state in several places because it creates a great way to determine if a transition is done. For example, in WalkRunRotate I use the IdlePose as a way to transition to different parts of the sub-state machine:



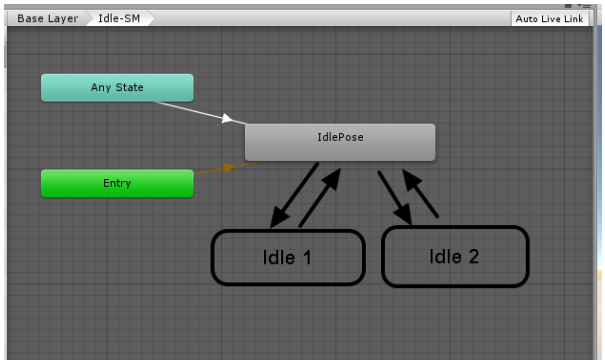
This is important because most of the time, Unity determines the length of a transition based on the length of the current animation.

So, if we have a long idle animation, the transition from “IdlePose” to “IdleToWalk” will take a long time.

The 1-frame IdlePose helps us move in and out really fast.

If you want to add some nice long idles, can do the following:

1. Add some idles off of the main Idle-SM sub-state machine.



Once added, you’ll create transitions that use the L0MotionPhase property to go into which ever idles you want... whenever you want.

Doing it here ensures these longer idles will only trigger when no other action (ie walking) is taking place. You can then randomize which idle plays, choose specific ones based on time, etc.

Inside of the Idle.cs motion, you’ll trigger the transition to your idle by using the following code:

```
mMotionController.SetAnimatorMotionPhase(mMotionLayer.AnimatorLayerIndex, <your L0MotionPhase>, true);
```

2. Create your version of the IdlePose.

This step isn’t required, but if you want you can create a small 1-frame animation that matches your idle. Then, replace all the IdlePose states with your new pose animation.

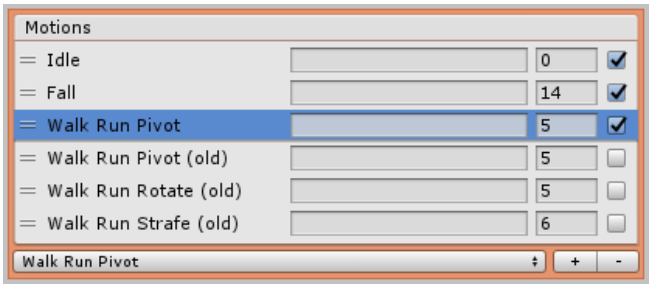
How do I use the new 'WalkRun…' motions with my custom animator?

I didn't remove the old WalkRun… motions. I just added "(old)" to their names. This way, you can still use them if you prefer.

I also can't auto-update your scenes. So, by default you'll be using the older version of the motions in your existing scenes.

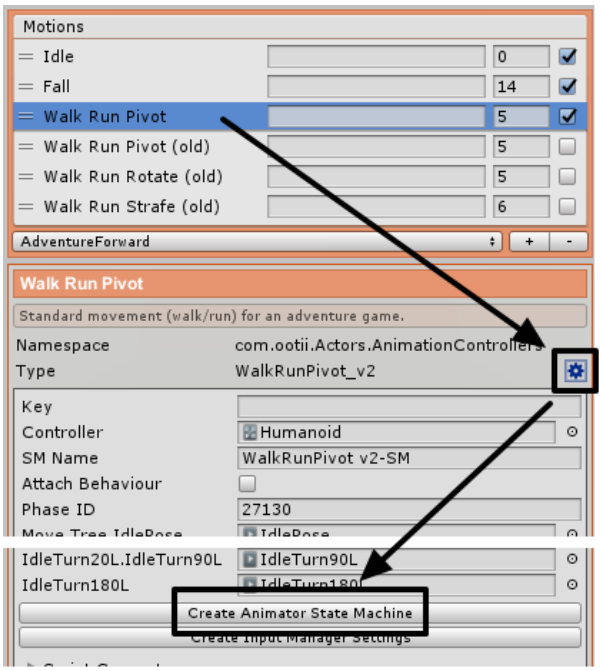
There's a couple of steps to use the new version in your existing scenes. In these steps, I'll use WalkRunPivot as an example, but it works for any of them.

1. Add the new "WalkRunPivot" motion to your scene and disable or remove "WalkRunPivot (old)"



2. If you're using an animator you've modified and not my default 'Humanoid' animator, you'll need to add the new motions' state machines to your animators.

Simply click on the motion, click the blue-gear icon, and press the "Created



That should be it.

Can I use MC with UMA?

Yes. I haven't used UMA in a long time, but a recent post from SecretAnorak (11/1/2016) explains the basic steps:

Hi there,

After a request for help I've asked the brains behind UMA to make a small change to UMA2.1 to make life easier for Motion Controller users. If you download the latest UMA53 Branch you will find the fix which allows the following steps to work:

1. Make sure you have MC and the newest version of UMA53 installed (or 2.1 when it's on the store).
2. Open up one of the UMA examples and make a prefab out of the UMA game object. (the one with the libraries in)
3. Open up the MC_Simple Scene from Motion Controller.
4. Add your UMA Prefab (remove the UMACrowd & UMACustomization objects)
5. Duplicate and deactivate Jones.
6. Re-Assign the camera rigs to your new copy of Jones.
7. Delete all children of the Jones object.
8. Add an UMADynamicAvatar component to Jones
9. Pick an UMA recipe (such as "Hugo") and put it into the Recipe field.
10. Put the MC "Humanoid" animation controller in the Animation Controller field.

Run and have fun,...

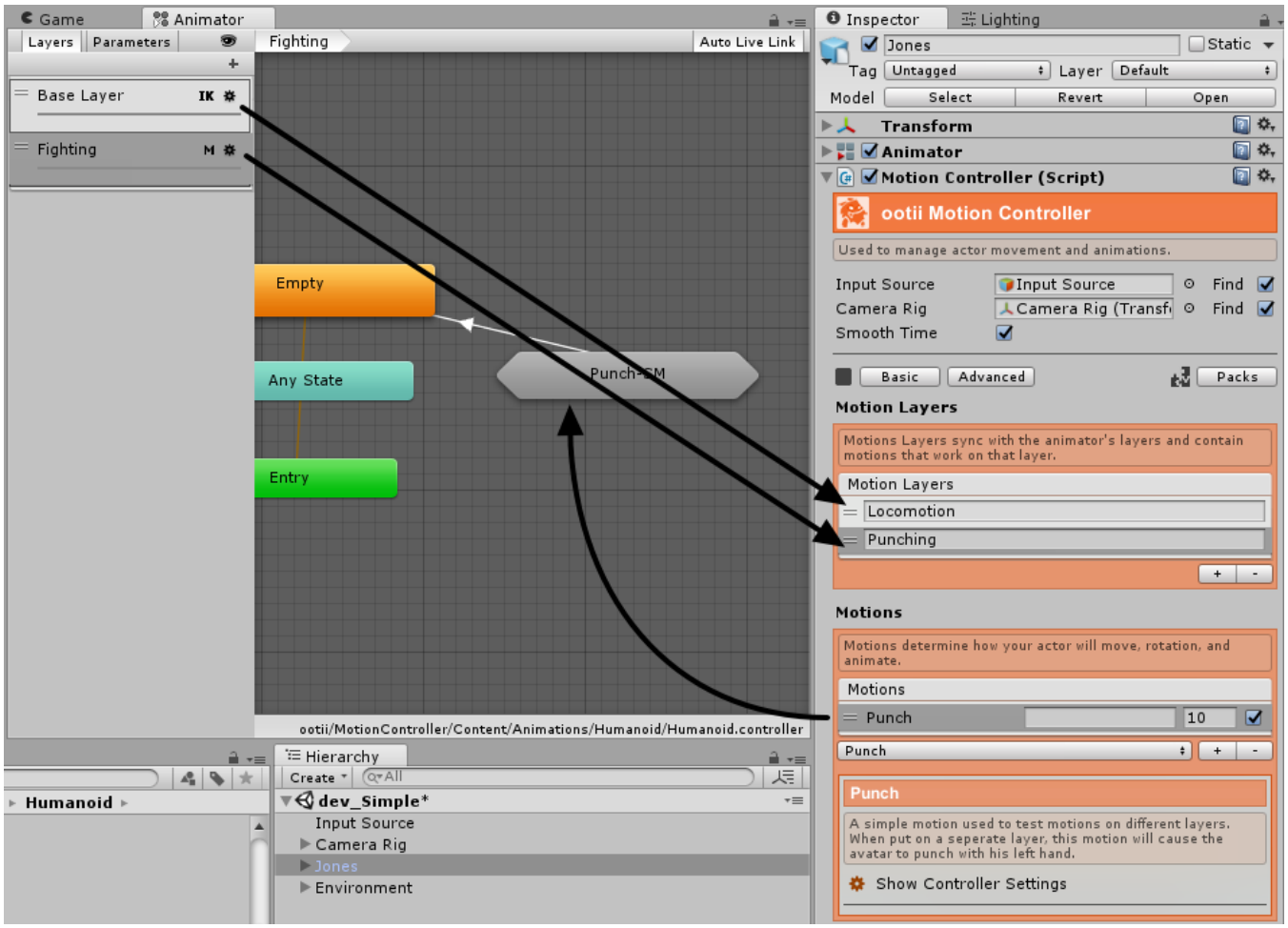
Notice that the actual UMA part is only the last 3 steps . Of course you can add your own customization system on top of this but hopefully this change makes UMA2 far easier to use with MC.

NB: because UMAs can be any height, you will need to recalculate your colliders in the actor controller (hit the "Auto Create" button when running). Also things like the climbing animation rely on specific heights of characters, so you may have to come up with an adaptation to the climbing motion if you are going to allow players to change their height. Try climbing on the box with a troll to see what I mean.

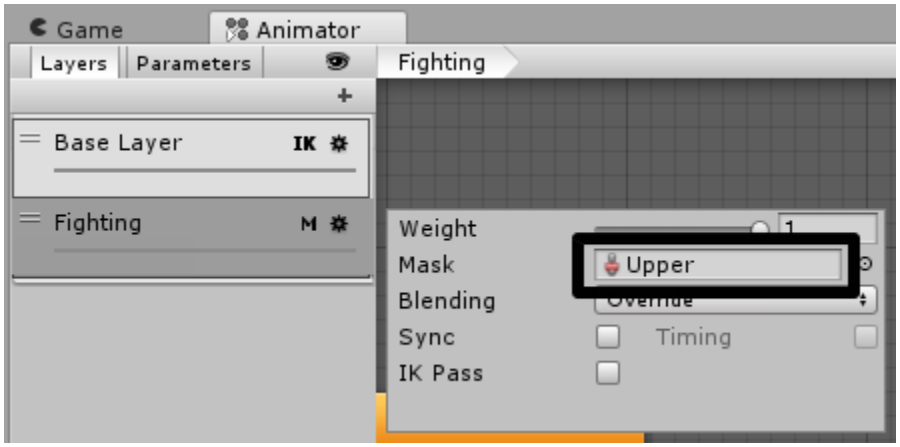
Hope that helps.this layer doesn't affect anything. That's why the weight isn't needed by the MC.

How do layers work?

Punch is a special motion that I created to show how to use multiple Animator Layers with the Motion Controller. Each layer in the Motion Controller syncs with a layer in the Animator. So, MC Layer 0 works with Animator Layer 0, MC layer 1 works with Animator Layer 1, etc.



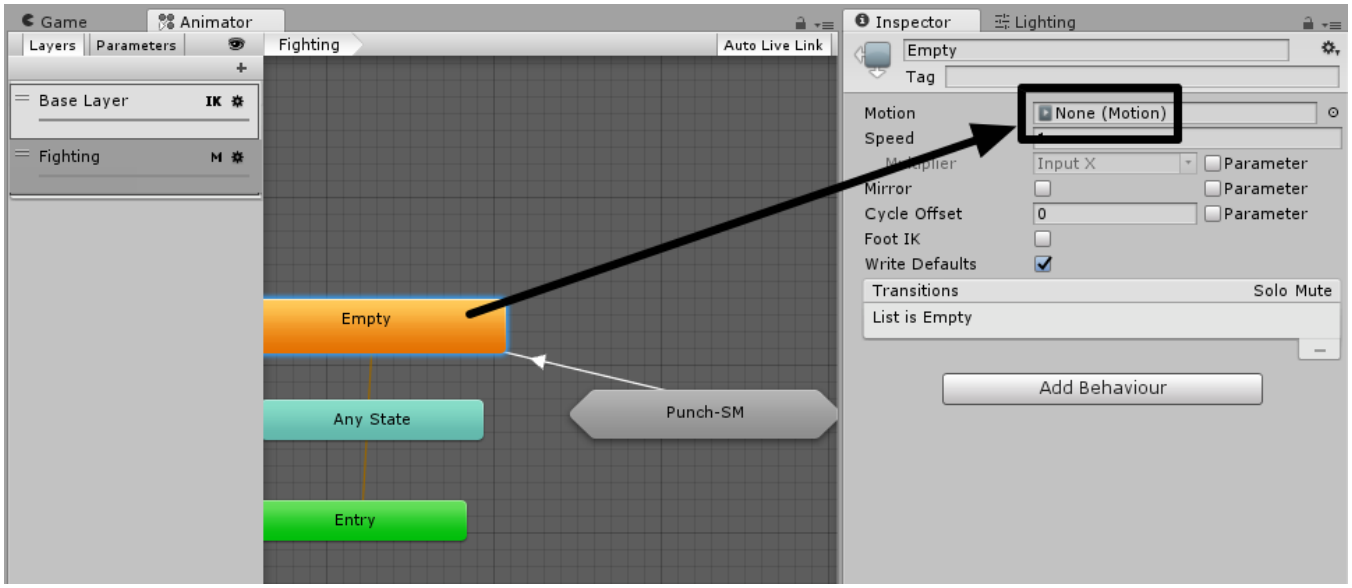
Layers can use Unity's Masks to control which part of the body the animations effect:



In this case, I'm using an "Upper" mask. In this way, the legs would still use the first layer while the upper body is controlled by the second layer. This would allow the character to punch while walking.

Note that I don't do anything with weight. You can create a custom motion to control that, but it's not something I do automatically.

You'll notice that on the second layer the default state doesn't use an animation:

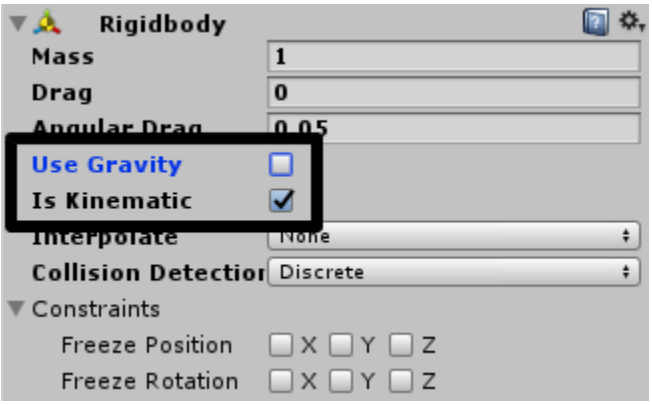


This allows the previous layers' animations to flow through when no animation is occurring on this layer. So, when no punch is active, this layer doesn't affect anything. That's why the weight isn't needed by the MC.

Can I use a Rigidbody with the MC?

You can use a Rigidbody with the MC. However, the MC is an input based character controller. That means the default rigidbody and MC will fight over who controls the character.

To stop this fighting, you need to set the rigidbody to kinematic and remove its use of gravity:



With that done, the MC will be in charge of movement and rotation.

Can the MC push objects around?

Yes. However, there are some steps to follow.

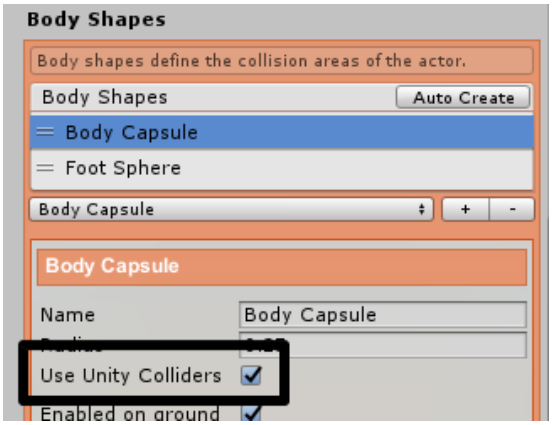
1. Rigidbody Colliders

Unity has rules about what kind of objects can interact with their physics system. Since the MC requires that its rigidbody be kinematic, it will only interact with rigidbody colliders.

You can see the rules here: https://docs.unity3d.com/Manual/CollidersOverview.html

2. Use Colliders with the MC

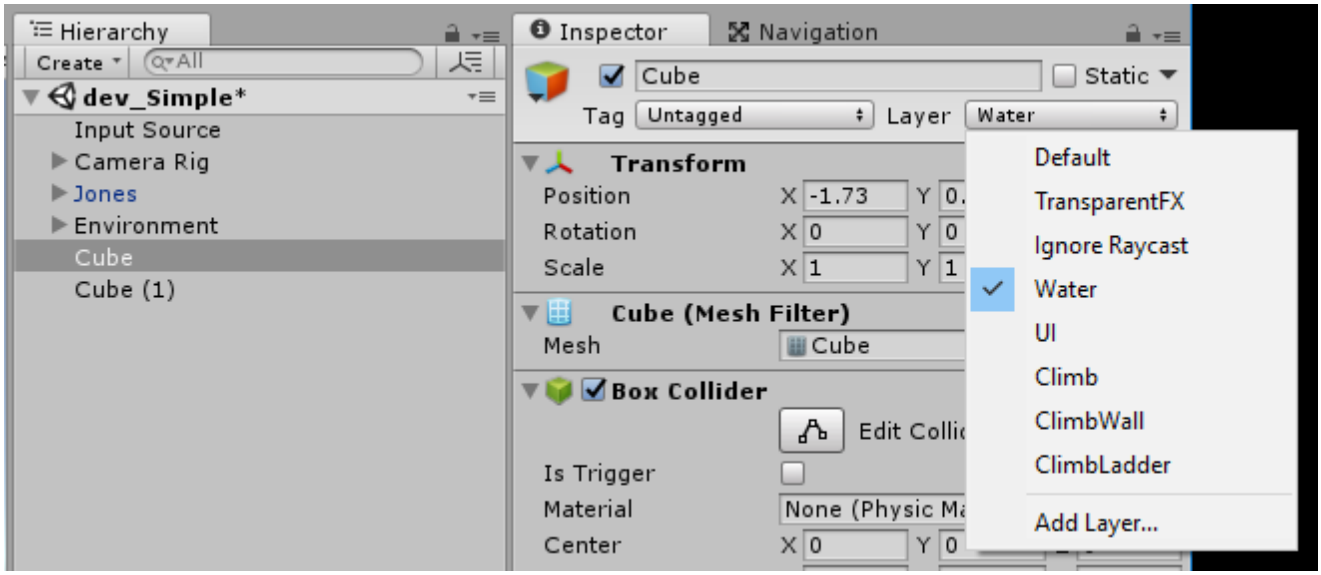
Because we're using Unity's collision system to move the objects, we need to ensure colliders exist on your MC character. You can add them manually or check the "Use Unity Colliders" check box on the Actor Controller's body shapes.



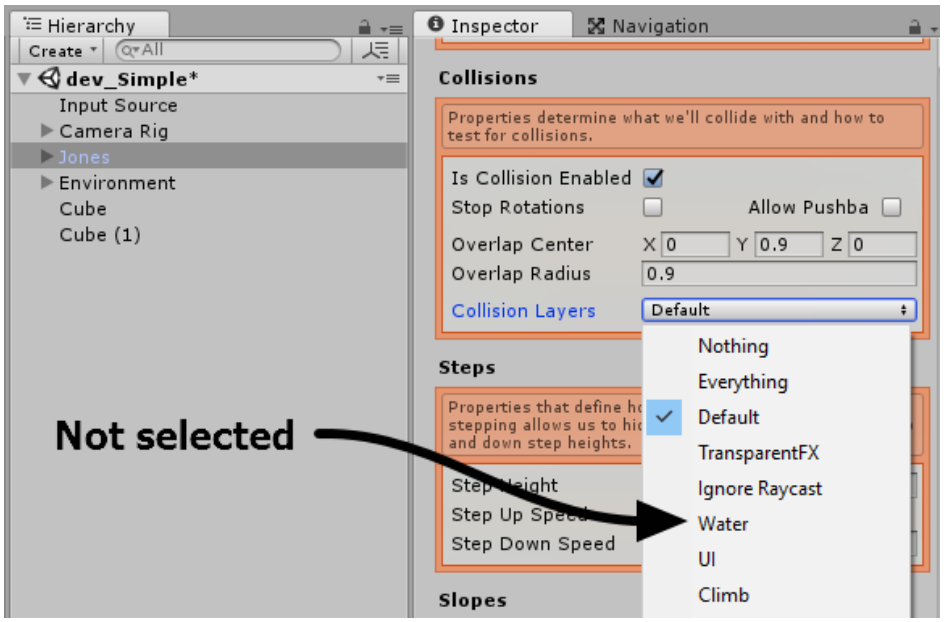
3. Layers

By default, the Motion Controller / Actor Controller will determine collisions with objects and stop movement. So, we need to tell the MC to not determine the collision and allow Unity's physics system to handle them. This will allow the MC to push forward and Unity to move the other objects.

Ensure all your other physics objects are on a separate layer. This is just Unity's standard "Layer" property for GameObjects.



Then, ensure the Actor Controller does not determine collisions with that layer.



Once done, your Motion Controller character can push these objects around. However, he'll stop when pushing against objects he determines he should collide with.

Support

If you have any comments, questions, or issues, please don't hesitate to email me at support@ootii.com. I'll help any way I can.

Thanks!

Tim