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.

The latest version of the documentation can be found online:
http://www.ootii.com/Unity/MotionController/MCReadMe.pdf

I'm also on the forums throughout the day:
http://forum.unity3d.com/threads/229900-Motion-Controller

Tim
tim@ootii.com

Documentation

For the most up-to-date documentation, please head to our website.

This Guide:

http://www.ootii.com/unity/motioncontroller/MCReadMe.pdf

Motion Builder's Guide:

http://www.ootii.com/unity/motioncontroller/MCMotionBuilder.pdf

Overview

Character Controllers are critical in most games as they are the connection between the player/AI and the avatar that's in the game. At the core, character controllers manage input, control movement, and play animations. The Motion Controller does this and more.

The Motion Controller is a character controller that's built to be a framework upon which **any type of motion** (jumping, climbing, sneaking, etc.) can be built. The challenge is that every game is different. You may not need climbing and I may not care about swimming. However, we both need an efficient way to manage the motions our PCs and NPCs have.

This character controller not only provides the framework, but includes core features most character controllers neglect. Things like support for **moving and rotating platforms**, **applying physics** based forces, and cleaning up **animation data**.

As it is, the Motion Controller isn't meant to be the end-all-be-all for every game. It's meant to be a foundation from which you, me, and others can build motions and plug them in to our unique creation.

Motions

Motions are a mix of animation and code that work together to create a specific actions.

Mecanim is an awesome tool for blending and managing animations, but with complex actions like climbing it can fall short. These kinds of motions need to define things: what can be climbed, when should the climb start, and how to move out of the climb.

The Motion Controller is a **framework** for handling these kinds of motions. See 'Understanding Motions' for more information.

Components

The Motion Controller package includes several components that help create a full experience. Most of these components can be used by you outside of the controller itself:

Motion Controller – Framework through which motions are assigned, managed, and run.

Follow Camera Rig – A basic third-person camera that always stays behind the character. Note that this is **NOT** the advanced Adventure Camera & Rig that you can find on the Unity Asset store.

Using the Adventure Camera & Rig with the Motion Controller is a great way to get an advanced experience.

Mecanim Animator – This controller still uses the Animator to manage animations. However, each motion represents a distinct sets of animation nodes that are typically disconnected from other sets.

Note: All models and animations come from Unity's Raw Mocap data for Mecanim. These assets can be easily replaced with your custom ones.

Debug Logger – A class designed to write debug information to the console, screen, and out to files. Included as a bonus just for you! ☺

Object Pool – A super-fast object caching class used to keep your game from having to reallocate objects over and over. Use it with any C# class you want.

Profiler – A class designed to measure the time it takes for scripts to run. This is a great tool for looking at how long processes take.

Features

The Motion Controller supports the following features:

- Extendable motion framework
- Customizable gravity, grounding, and sliding
- Ability to modify root motion data at run-time
- Support for moving and rotating platforms
- Support for applied physics forces
- Simulated input for NPC AI to control basic motions
- Inspector for assigning and customizing motions
- Walking, running, and sneaking
- Customizable falling, jumping, climbing, and sliding
- Support for prefabs
- Out-of-the-box support for Windows Xbox 360 gamepad and keyboard/mouse

Setup

While the document looks long, the reality is that the steps are pretty simple and they are things you're probably doing 100 times for each game. I'm just detailing them here for everyone.

Basic Steps:

1. Setup the Xbox controller (optional)
2. Setup the camera
3. Setup the character controller
    a. Add the Motion Controller to the character controller
    b. Select the motions you want.

In most cases, you'll want to start your game with a fresh project and new scene. This means you will be adding your own avatar (that uses Mecanim's 'Humanoid' rig). The steps below are assuming that you're starting with a new scene and adding everything for the first time.

If you're starting with an existing scene or even with a demo that is included in this package, you'll be modifying existing elements.

If you do get stuck or have a questions, please don't hesitate to email support@ootii.com.

Getting The Demo Running

1. Create a new project or open an existing one

Follow the standard practice for creating or opening a scene. Nothing special here.

2. Add this package to your project

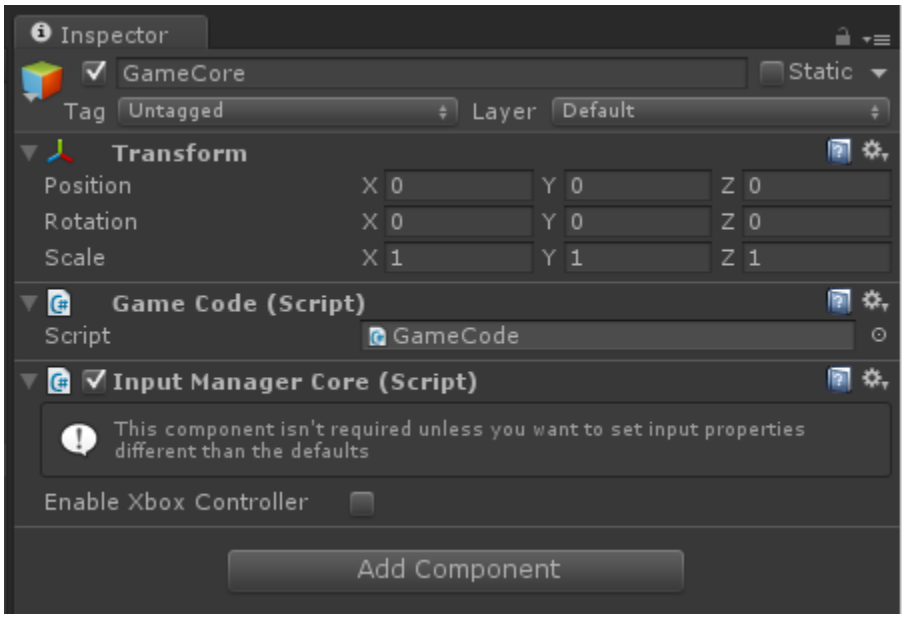
This package contains the controller, Animator, and other code you'll need to get up and running. You can keep all these assets in the folders they are currently located or move them as needed.

3. Setup additional Inputs

The ootii InputManager class in the package supports the keyboard, mouse, and the Windows version of the Xbox Controller. By default, the Xbox controller is disabled.

If you don't want the Xbox controller enabled, move on to step 4.

If you wish to enable the Xbox controller, add the InputManagerCore script (`ootii/Framework_v1/Code/Input`) to a game object in your scene and set the 'Enable Xbox Controller' flag there. This will keep you from having to modify code and keep the value during updates.



You'll also need to ensure Unity's Input Manager is setup to handle the requests. Otherwise, you'll get errors when you run.

You can setup the Unity Input Manager manually (to save existing settings you may have) or automatically. See below.

1. Automatic

If you don't want to spend the time to add the entries manually, you can use the included settings file.

Simply unzip the `InputManager.asset` file from the zip file found in the `ootii\MotionController\Extras` folder.

Replace the existing `InputManager.asset` file found in your `ProjectSettings` folder with the file you're unzipping.

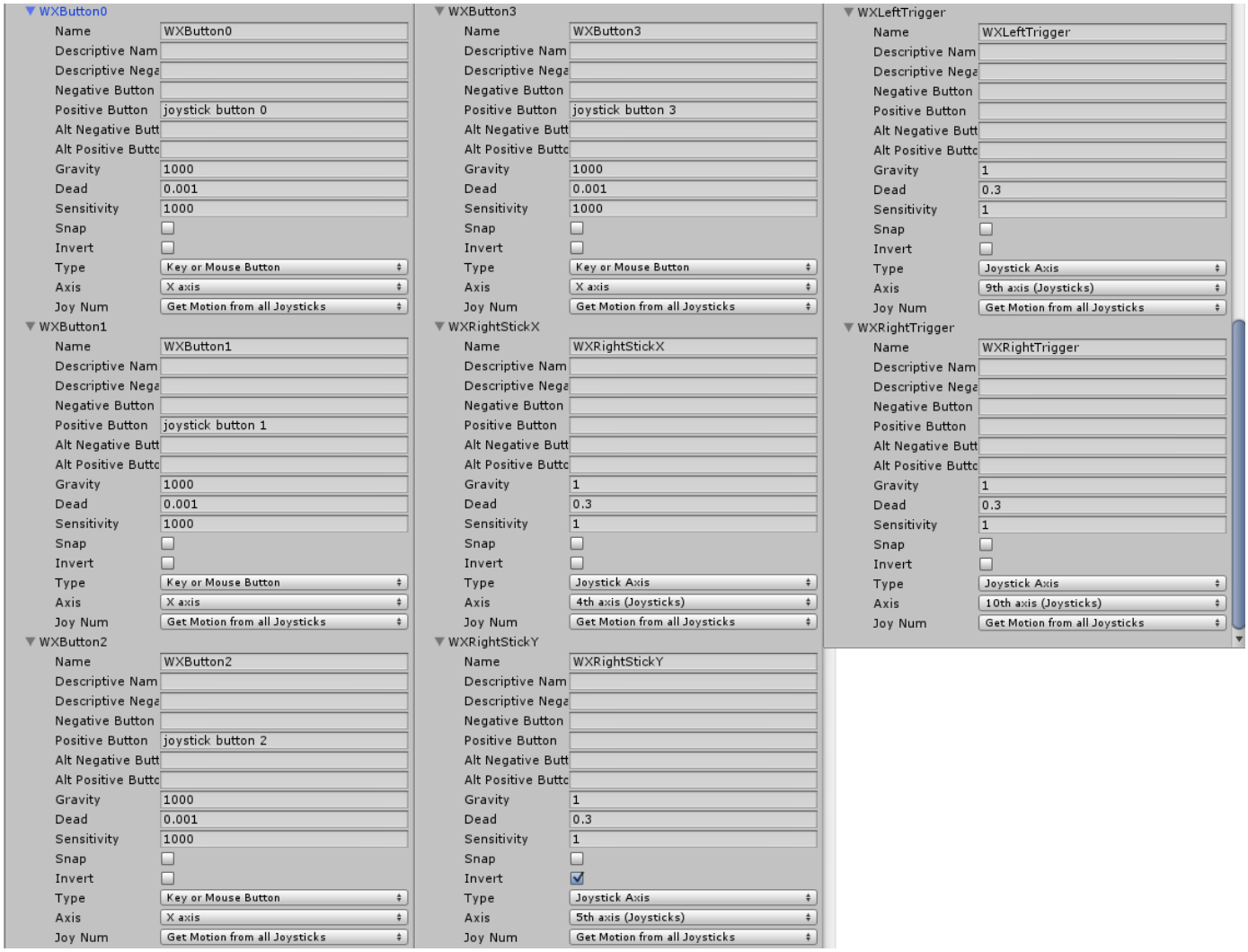
Note: Any mappings you've created prior to the copy WILL BE LOST.

2. Manual

In the Unity InputManager (Edit | Project Settings | Input), add the following entries. These will allow the code to support the Xbox 360 controller on Windows.

First, set the 'Size' property to 23. That will create 8 'Jump' values at the bottom of the list. Modify these values as needed.

These settings are for Windows:



For Xbox controller settings for Mac, see the following page and replace the 'Axis' and 'Positive Button' values with the values for Macs: http://wiki.unity3d.com/index.php?title=Xbox360Controller

4. Play the 'Demo' Scene

At this point, you can load and run the 'Demo' scene found in the `ootii/MotionController/Demos/Scenes` folder.

5. Play the 'AdvDemo' Scene (optional with the Adventure Camera & Rig asset)

If you own the Adventure Camera & Rig, you can also run the 'AdvDemo' scene. However, you'll have to get the latest `AdventureRig.cs` file from the Unity Asset Store (version 7) and put that in the `ootii/Framework_v1/Code/Cameras` folder of this project first.

The AdventureRig.cs file is the only file you need from the Adventure Camera & Rig package.

In fact, if you run with the Motion Controller package and the Adventure Camera & Rig package together you will get 'conflict' errors.

When you do this and open the 'AdvDemo', you may get an error that says a script was not found.

If so, click on 'AdvCameraRig' in the hierarchy, click the Unity selection circle to the right of the 'Script' property, and select 'Adventure Rig' from the list of scripts. Now you're ready to run.

Creating A New Scene

1. Follow steps #1 - #3 from 'Getting The Demo Running'

2. Create a new scene

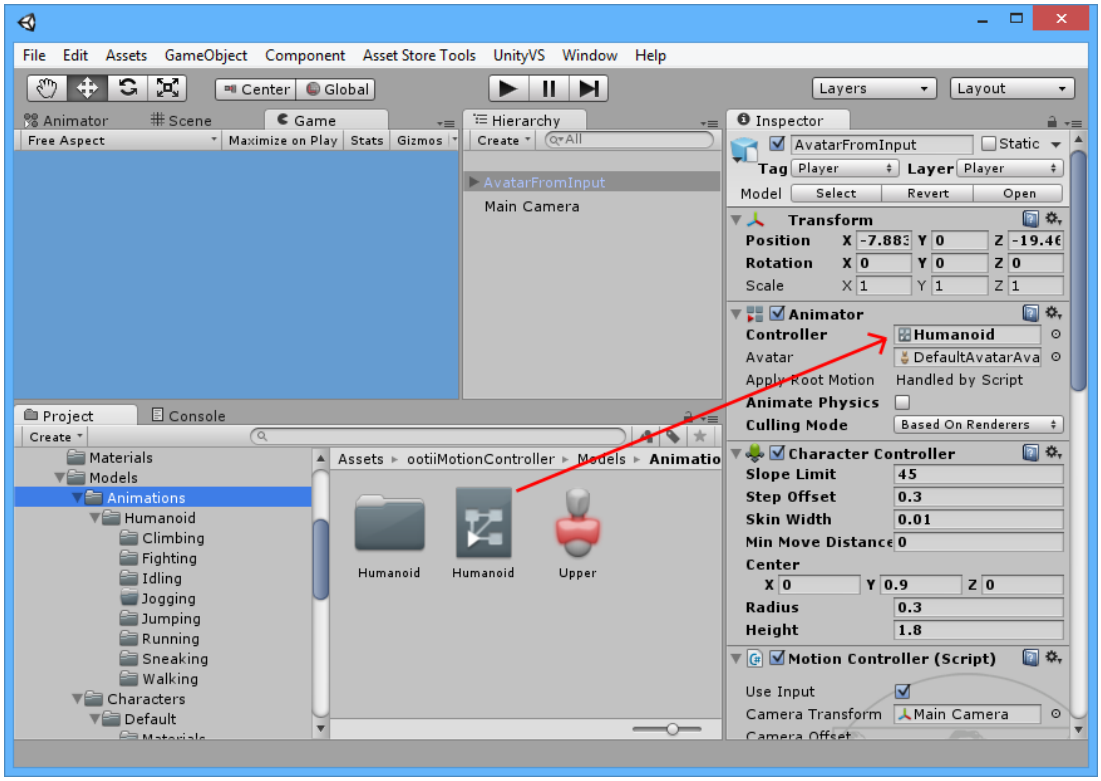
3. Add an avatar to the scene

In this package, you'll find the sample Unity character in the following folder: `Models/Characters/Default`. Drag this avatar into your scene or add your own.

If you add your own, ensure that it has a 'Humanoid' Rig defined. If you need help, you can follow Unity's instructions: http://docs.unity3d.com/Documentation/Manual/AvatarCreationandSetup.html

3.a. Associate the 'Humanoid' Animator to the controller

Under `Models/Animations`, you'll find a Mecanim Animator called 'Humanoid'. Drag this to the 'Controller' slot of the 'Animator' component that was created when you added your avatar.



Note: All models and animations come from Unity's Raw Mocap data for Mecanim. These assets can be easily replaced with your custom ones.

3.b. Add a Unity Character Controller component to the avatar

See the settings in the picture for step #4 below.

Change the 'Skin Width' to be 0.01.

In code, I use the value of 0.075 (or lower) to determine if the avatar is grounded. If the skin width is too large (i.e. 0.08) Unity's Character Controller may force the character to float too far above the ground and become all jittery.

> If you encounter problems when jumping, try changing the 'Skin Width' for your unique avatar model.

> Ensure the Unity character controller capsule height, radius, and center match your character. If they don't, you may find yourself falling through the floor!

3.c. Set the Unity tags

Set the 'Tag' property to 'Player'. This is just a good standard.

4. Assign the Motion Controller

To do this, drag the 'Motion Controller' component onto the avatar you added to the scene in step #3. You'll find the Motion Controller under `ootii/MotionController/Code/AI/Controllers`.

4.a. Use Input

The first important setting is the 'Use Input' flag. If you have multiple characters in the scene, they can all use a motion controller to manage their animations and movements. However, the one you want the player to control should have 'Use Input' checked.

4.b. Motions

As you can see by the image below, the Motion Layers and Motions will be blank. You'll need to add the motions that fit your character.

For a detailed explanation of the settings, see the 'Motion Controller Settings' section below. Follow these settings to add motions you want your avatar to have.

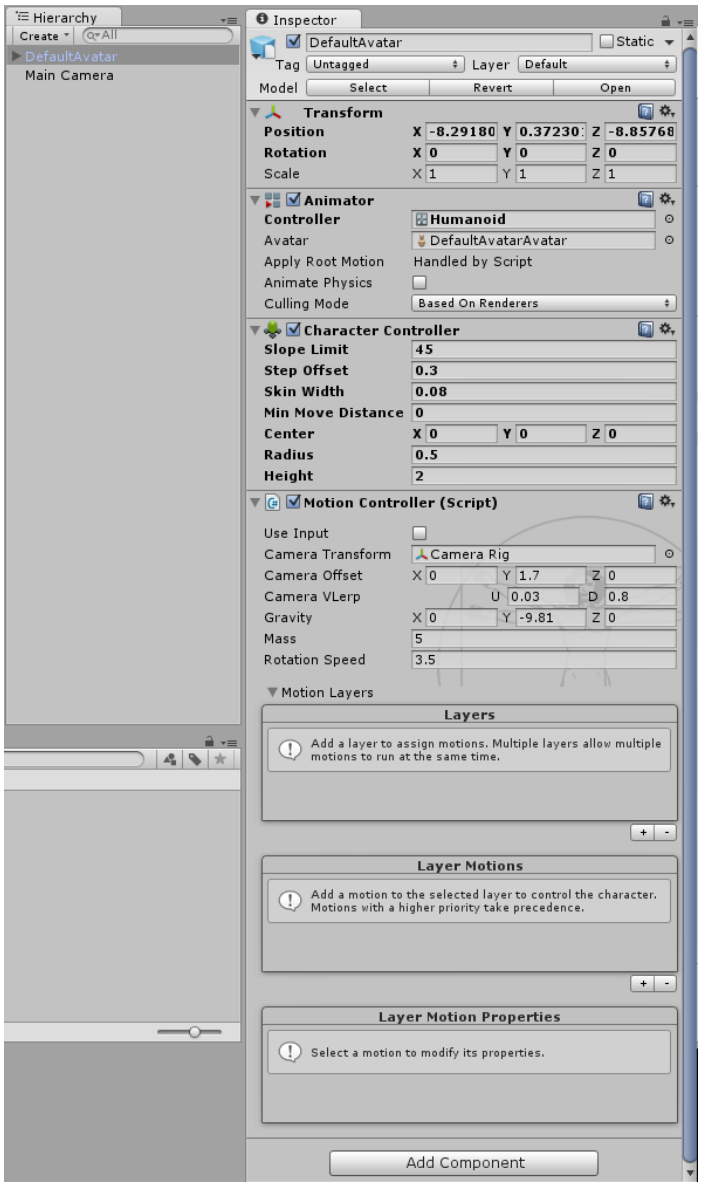
4.c. Motion Example

To start with, you can create a new layer by pressing '+' under the Layers section. Name it something like 'Locomotion'.

Then, add two Layer Motions by pressing the '+' symbol under that section…twice.

In the first drop-down that appears, select 'Casual Idle'. This is the default animation and state for the character.

In the second drop-down, select 'SimpleForward'. With this, your character can walk and run forward.



5. Create the Camera Rig

We're going to place the existing 'Main Camera' into a rig. I like to think of the existing camera as the lens and this new object as the dolly or rig that moves the camera lens.

5.a. Create a GameObject for the rig

Simply add an 'Empty' game object to your scene using the menu item: GameObject | Create Empty.

Rename this to something like ‘Camera Rig’ and drag the existing ‘Main Camera’ onto it. Now the ‘Main Camera’ is a child of the rig.

Ensure the Transform of the ‘Main Camera’ is cleared. Meaning 0 for positions, 0 for rotations, and 1 for scales.

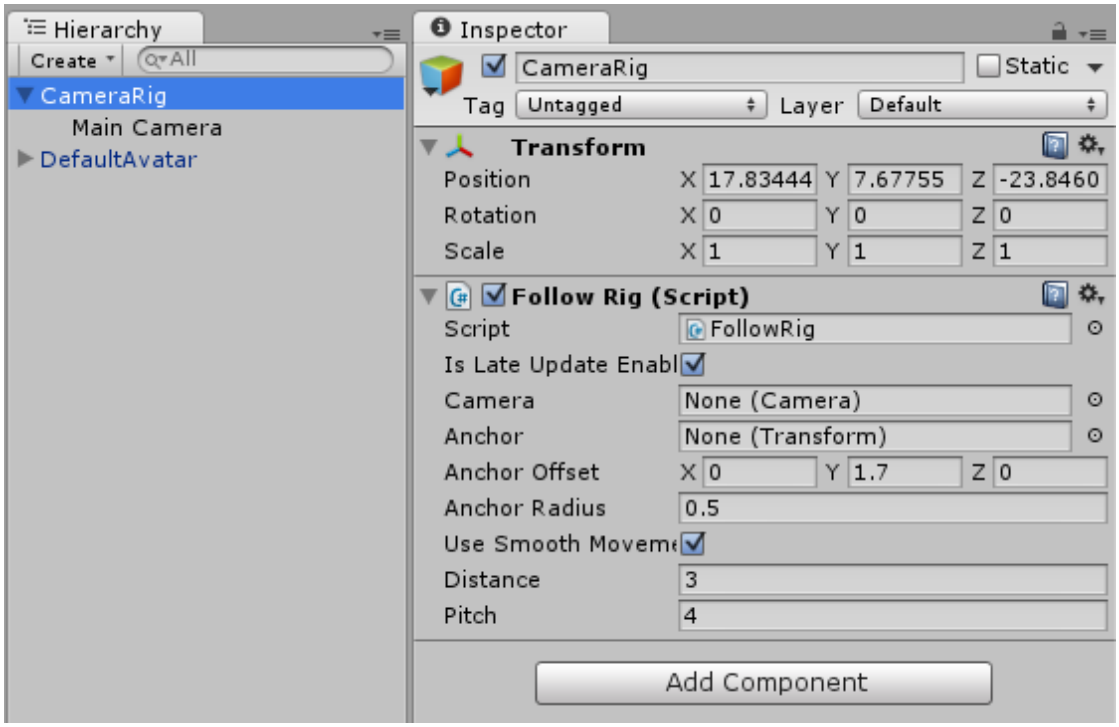
5.b. Add the component

Drag the ‘Follow Rig’ component from the `ootii/Framework_v1/Code/Cameras` folder onto the ‘Camera Rig’ you just created.

5.c. Ensure ‘Is Late Update Enabled’ is NOT checked

We won’t use the camera’s LateUpdate() instead we want the camera to process after the controller. So, the controller will call the camera’s logic when ready.

Once you do this, the inspector will look something like this:



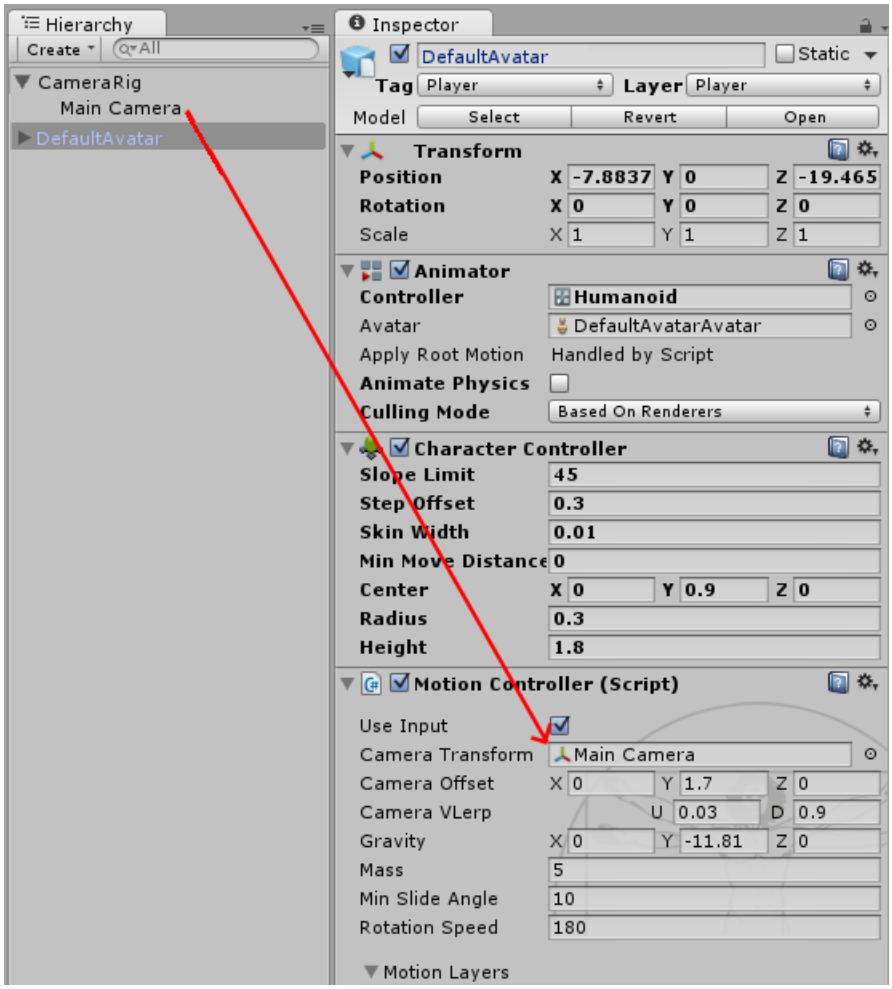
For a detailed explanation of the settings, see the ‘Follow Rig Settings’ section below.

6. Connect the Follow Rig and Motion Controller

In the setting inspectors, you'll see some properties that need to be filled. Basically, we want to connect the camera to the controller and the controller to the camera.

6.a. Tell the avatar about the camera

In any Avatar that has a Motion Controller, set the 'Camera Transform' of the Motion Controller to the Main Camera that is the child of CameraRig you added in step #5.

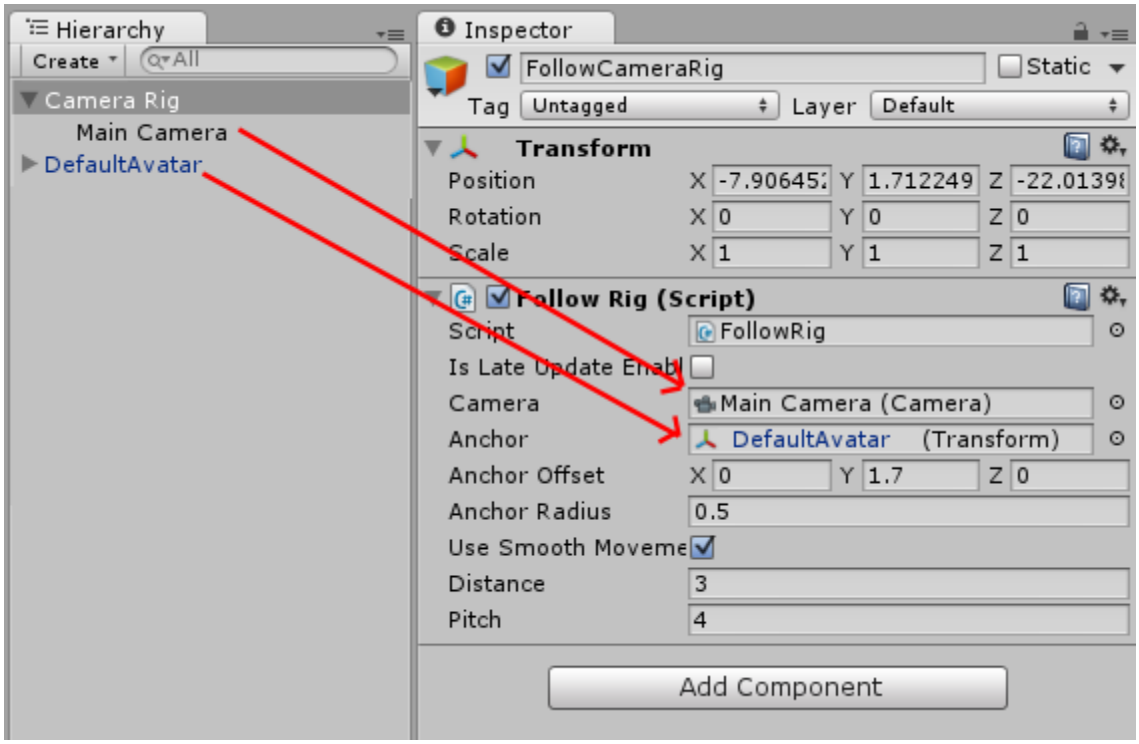


6.b. Tell the camera rig about the avatar that is controlling it

First, tell the camera rig that you created in step #5 about the camera it is managing.

The camera rig will attempt to follow a specific Transform. I call this the 'Anchor'. By using a transform as the anchor, you could attach the camera to something other than the character controller and the camera would follow that.

In most cases, the camera anchor will be the transform of the avatar you created in step #3. So, tell the camera about the avatar's transform.



7. Create a floor

This is really specific to your game.

You can use a cube that is stretched, a Unity terrain, a model, etc. Whatever you plan on using for your terrain, place it so the avatar doesn't fall through the world. Again, what you use depends on the type of game you're making.

Make sure the floor is under your new avatar. ☺

8. Add screen logging (optional)

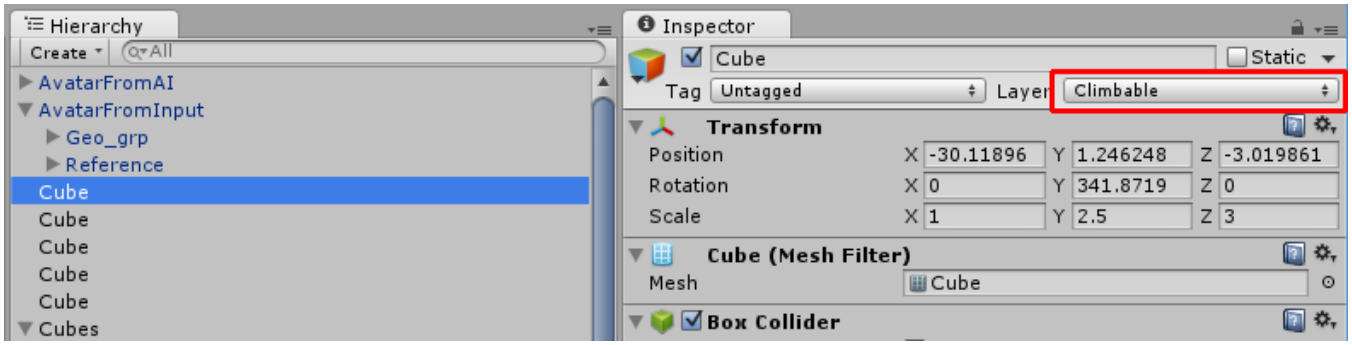
Included in the Motion Controller is the Debug Logger. Use this to write debug info to the console, to the screen, or to a file. To get more information, check out our online documentation:
http://www.ootii.com/UnityDebugLogger.cshtml

To setup the debug logger, simply add the 'Log' script directly to the camera lens (the 'Main Camera' attached to your camera rig).

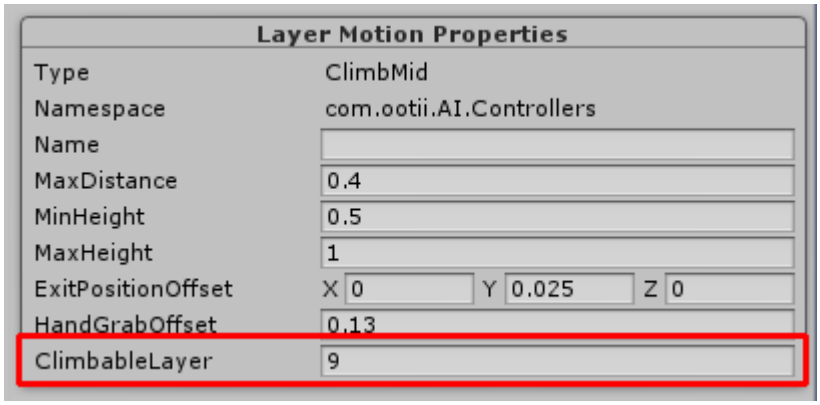
You can find the 'Log' under `ootii/Framework_v1/Code/Utilities/Debug`.

9. Climbables (optional)

In order to keep the character from trying to climb everything he comes in contact with, I've set it up so that object that can be climbed need to be on a 'Climbable' layer in Unity. Create this layer as 'User Layer 9' and then the avatar will be able to climb it.



If you choose a different layer than 'User Layer 9', ensure you change the climb layer in the motions:



10. Run the game

That's the meat and bones of it. With that in place, you can now run the game using a keyboard and mouse or Xbox 360 controller for Windows.

By default:

WASD or **Left-Stick** moves the avatar

Mouse or **Right-Stick** rotates the view

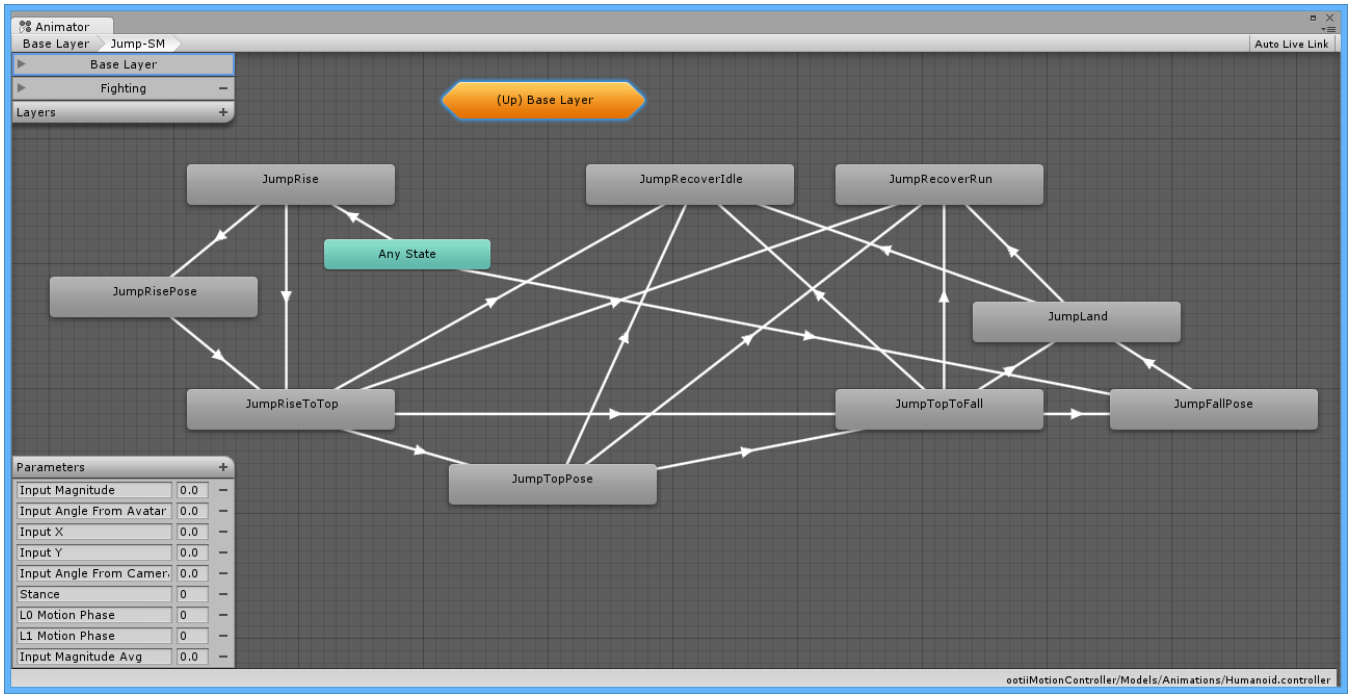
Understanding Motions

Motions don't replace the Mecanim Animator, but work with it to control the animation flow. As such, motions typically have three components: animations, code, and settings. They work together to create the full effect.

Think about a dynamic jump. I'm not talking about a simple jump animation that's always the same height, but a jump based on physics. A jump where different characters with different attributes could have different jump heights.

Animations

There needs to be animations that represent the launch, the rise, the pause at the top, the fall, the land, and then the recover-to-idle or recover-to-run. These animations are setup in Mecanim:

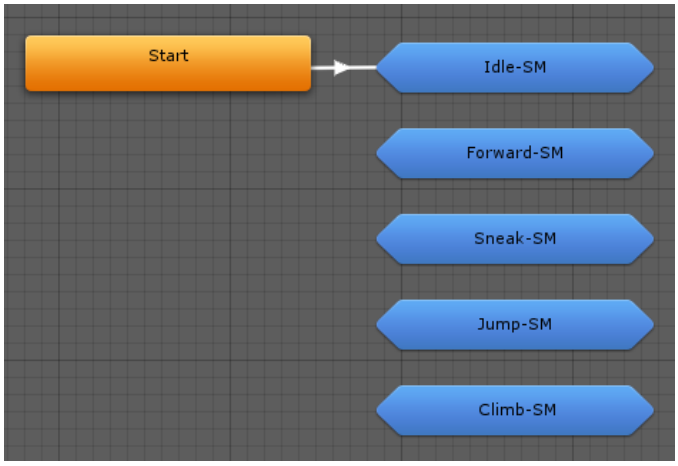


The goal with each motion is to keep its animations contained in a single state machine.

In this way, motions can be added without impacting other motions.

We leverage Mecanim's blending to smoothly transition between motions. In some case, we'll have specific transition animations, but those are handled in the state machines themselves.

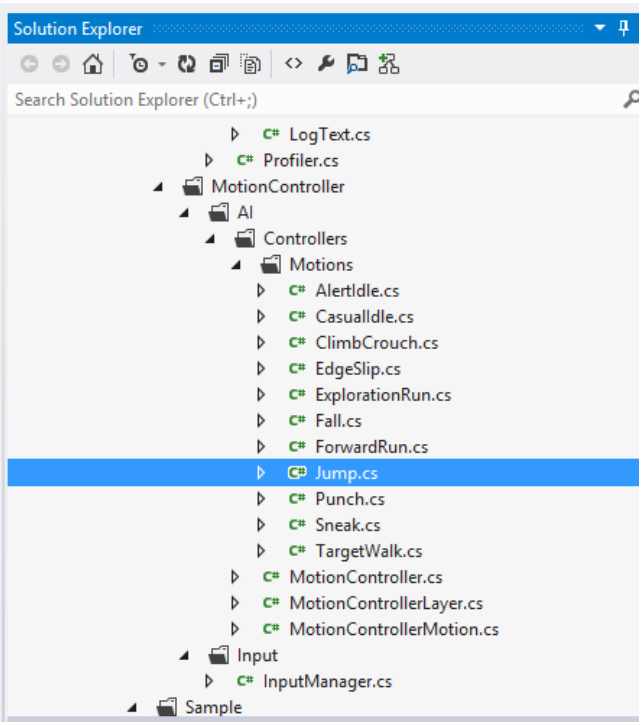
We make heavy use of the 'Any State' to transition into the motion's animations.



Code

In a physics based jump, we need to know when to jump, when we reach the peak, when we land, and how to recover. During the jump we could resize the character's collider, transition to a climb, or quit the jump early because the player jumped onto a platform.

This is all handled in the code part of the motion.



Each motion has its own script that 'plugs into' the Motion Controller.

The script has specific functions that test to see if it's time for the motion to become active.

In the case of a jump, the test could be for a specific button press. In the case of a fall, it could be for a specific ground distance.

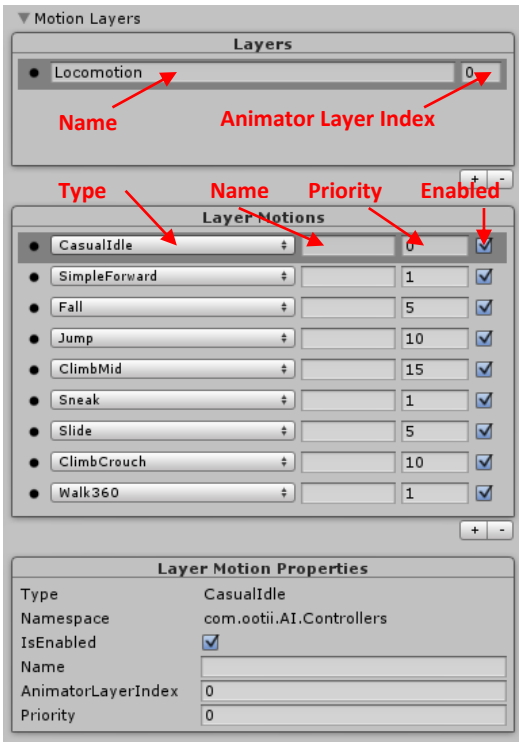
Once the script is activated, it sends a signal to the Mecanim animator and the animation begins moving along.

The active motions then monitor their states and drive the animator through the phases of the motion.

In the end, the code determines when the motion has completed and shuts itself down.

Settings

The last key ingredient is the settings that are applied to the individual character and controller.



In the inspector for the Motion Controller, each character can have multiple motion layers. These pretty much match Mecanim's Animator Layers and allow multiple motions to control different animations at the same time.

Then, we can assign any number of motions to the characters. In the example to the left, our character can idle, run, fall, jump, climb, and sneak.

Removing or adding the support for a new motion is simply a matter of configuring it in the setup.

At the bottom, we can set the properties for the specific motions. Here you can see Jump is selected and we could change the jump impulse to change how high and far the character can jump.

In this implementation, the Jump also allows us to determine how much 'in-air' control the player has.

In the end, there could be multiple types of jumps that are used by different characters or even the same character at different times.

Summary

So a character using the Motion Controller can have any number of motions assigned. The priority of the motion along with the activation test (in code) determines which motions will activate and when.

The goal being that people can create new motions following the Motion Builder's Guide and pass them around. If you need the swim motion, in can be built and integrated easily with the Motion Controller. If you don't need it, don't add it the list of character motions.

Motion Controller Settings

The Motion Controller inspector has two large sections:

General Properties

These are the properties that define how the controller interacts with the camera and the world.

Use Input

Since the Motion Controller can be used with NPCs, this setting is used by the code to determine if the specific character should react to input events.

Camera Transform

References the Follow Camera Rig you setup in step #5. This is a Transform value so you could use another camera if needed.

Camera Offset

The 'anchor' position (relative to the avatar) that is the orbit center of the camera. Typically this is the head of your avatar.

Camera Lerp

As the avatar moves up (say in a jump or climb), how quickly the camera moves up to follow it. This allows us to have a different movement feel for a camera going up and down than side to side.

Gravity

This controller applies gravity itself. Set the gravity values here in order to customize how it affects the controller. This value is meters per second squared.

The reason we allow for this is so that you can allow the player to fly, super jump, etc. On the flip side, it can also help increase the falling speed and make movement feel faster.

Forward Bumper

The forward bumper is used to detect an obstacle ahead of the character. When a character hits the obstacle head on, they stop. This vector determines the position of the forward bumper relative to the character's origin and forward direction.

Forward Bumper Blend

When the character stops because of the bumper, we'll slowly start to move the character if they are rotated away from a head-on impact. Imagine a head-on impact is a '0' angle, this property is the minimum angle (to the left or right) the character must face before movement will start again.

Mass

Used in physics calculations when forces are applied.

Min Slide Angle

The 'Slope Limit' of the Unity Character Controller determines how large of a slop the character can go up. This property determines at what angle the character will slide down the slope when standing idle.

See 'Terrain Tips' for more information on using the two values.

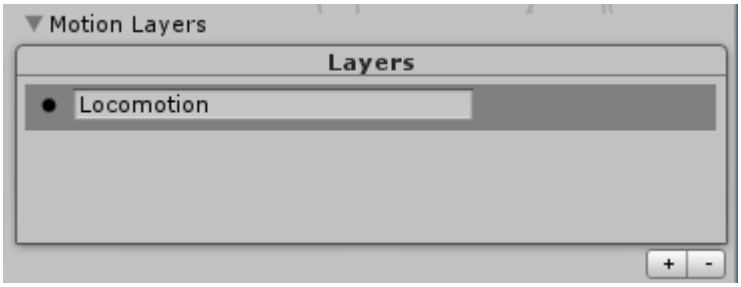
Rotation Speed
Determines how quickly the avatar will pivot. This value is degrees per second.

Motion Layers

Motion layers allow for blending of motions similar to the way the Mecanim Animator supports multiple layers. In fact, you'd typically want the number of layers to match.

On each layer, you'll assign the motions that you want the character to be able to perform. You can create as many motions as you want.

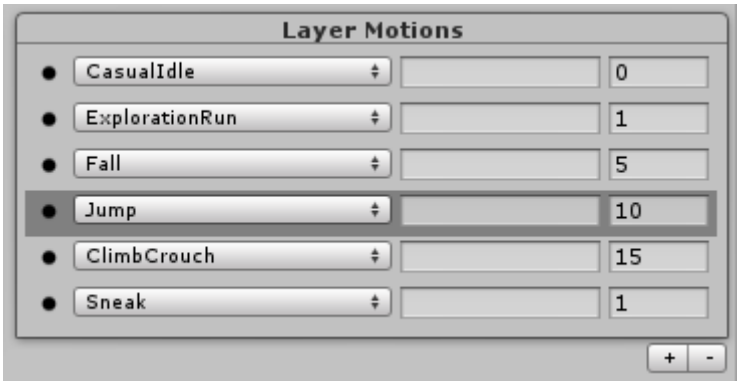
Each motion can then have properties that customize how the motion behaves.



Start by pressing the '+' to create a new layer.

Layers run in parallel just like Animator Layers. So creating a layer per Animator Layer is typically a good idea.

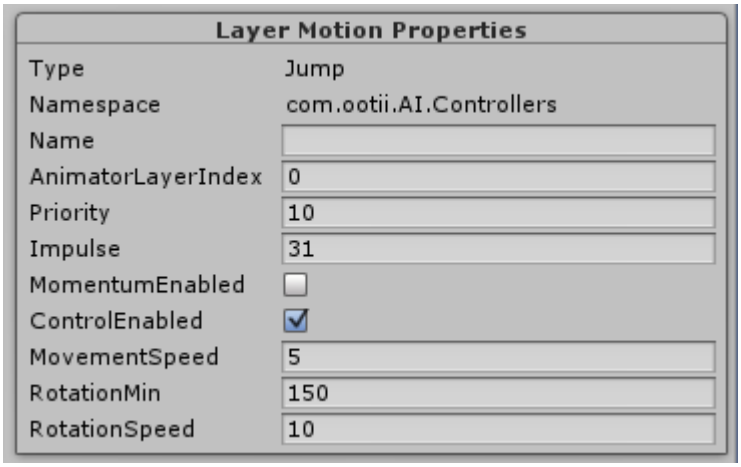
You could have a layer for 'basic locomotion' and then a layer for 'upper body' if you wanted a player to shoot while running or standing still.



Motions are then added to the layer. All motions in the code solution are automatically found and listed in the drop-down.

Select the motion the character is going to have. In this way, characters that don't jump won't need the 'Jump' motion.

The name and priority settings are here for convenience. Note that motions with higher priority trump those with lower priorities.



Once the motions are selected, you can customize them in order to create the behavior you want. For example, one character may jump higher than another.

What properties show depend on the creator of the motion.

With this setup, it's important for you to understand the capabilities and limits of each motion you use. This is especially true with motions created by others.

For example, your jump could work differently than someone else's jump.

Assuming there's demand, as this tool improves I'll create better ways to find and share motions.

Follow Rig Settings

The code is available for you to customize or change the rig as needed. However, if you want to use the camera system out-of-the-box, these settings can help you customize it.

Is Late Update Enabled

Disable this. It tells the camera not to update itself automatically. Instead the camera's LateUpdate function will be called when the controller is ready.

Camera

The camera that is parented to the rig.

Anchor

References the Motion Controller you setup in step #4. In fact, this is really just a Transform and you can use any object's transform.

Anchor Offset

The target point from the Anchor that the camera wants to look at. Typically this is the head or eyes of the anchor.

Anchor Radius

The collider radius of the anchor/or controller. This helps us control how closely the camera can get to the anchor.

Use Smooth Movement

Determines if we use smoothing during linear movement

Distance

Determines how far from the anchor + offset the camera should stay.

Pitch

The value represents the camera's pitch (degrees around the x-axis)

Looking straight forward has a pitch of 0. Looking down, creates a negative pitch (in degrees). Looking up creates a positive pitch (in degrees).

User Input and ootiiInputStub

I know you're going to get tired of me saying this, but every game is different. Not only can the supported devices differ, but the interactions on those devices differ across games.

So, my preference is to abstract the device and device interaction from the resulting action as much as possible. Unity's Input Manager actually does that as well. That said, there are some thing it has a problem with (like allowing players to change settings at run-time) and as I programmer, I like to have more control.

To help this abstraction, I've included two key files:

`ootii/Framework_v1/Code/Input/ootiiInputStub.cs`

> This static class is used inside all of the motions in order to test if specific input occurred. For example, you may call ootiiInputStub.IsJustPressed("Jump") to determine if the user just pressed the jump key.
>
> These object just passes the message onto your current input system. By default, it's the Motion Controller's InputManager.cs file.

`ootii/MotionController/Code/Input/InputManager.cs`

> This is the default handler for input. The ootiiInputStub file passes requests to this class by default.
>
> Now this InputManager isn't nearly as robust as other assets out there, but it gets the job done and gives you an example of how you can abstract the action from the input.

Custom Input Solutions

The reason I did it this way was so that you could implement your own input solution or purchase a more robust one on the Asset Store without having to change the code in all the motions.

You just have to change ootiiInputStub so that it doesn't send the messages to my InputManager. Instead, ootiiInputManager would send messages to the custom input solution of your choice.

Jump Input Action Example

The ootiiInputStub has a function '`IsJustPressed`' and InputManager has a function '`IsJustPressed`'. They both take a string value. In reality, this word doesn't have to have anything to do with animations, motions, or anything else. It's just a word.

Since I want something to trigger my jump motion, it seems like 'Jump' is a good word to use.

From anywhere, I can call '`ootiiInputStub.IsJustPressed("Jump")`' and get a Boolean back. The caller doesn't care how the Boolean was determined and it creates nice abstraction.

Inside ootiiInputStub, I call `InputManager.IsJustPressed("Jump")`. Inside InputManager, what I did was tie the word 'Jump' to a Unity input test. I could test anything, but I've chosen to test if the space-bar is pressed or if the 'A' button on the Xbox controller is pressed.

Remember the Input Manager included is very basic. For a full game, you'll want to allow players to change input setups and other more advanced features. There are plenty of input managers on the asset store or you can build your own.

Now that I can test for a jump, I want the right motion to play…

Input Actions and Motions

Motions are triggered in one of two ways:

1. Automatically, based on the implementation of the motion's '`TestActivate`' function. This function is called every frame by every motion to see if it's time to trigger the motion.

Because these functions are called all the time, you want to keep an eye out for performance bottlenecks.

2. Manually, using the Motion Controller's '`QueueMotion`' function. In this case, you can activate the motion from anywhere by flagging it to run next frame.

Take the Jump motion script for example:
'`ootii/MotionController/Code/AI/Controllers/Motions`'

In this scripts '`TestActivate`' function, you'll find some simple conditions that determine if it's time to jump. I'll walk through some of them:

> `mIsStartable` – An internal flag that determines if the motion is capable of starting again.
>
> `mController.UseInput` – Determines if this is the controller the player is using at the avatar.
>
> `ootiiInputStub.IsJustPressed("Jump")` – Ahhh… now this looks familiar. ☺

So we've tested some conditions and we hit the '`IsJustPressed`' Input Action we talked about earlier. Here is where the motion tests if the Jump action has occurred and the motion should be activated.

Motions to Animation

Now that we know how a motion gets activated, how does that trigger the actual animation?

Animations themselves are created in the Mecanim Animator; no different than what is done without the Motion Controller.

What the Motion Controller does is manage the animations as they flow through the Animator. One of the key tools to do this are the 'Motion Phase' Animator Parameters: 'L0 Motion Phase', 'L1 Motion Phase', etc.

Note that the 'Lx' part represents the Mecanim Animator Layer index. So following the image below: 'L0 Motion Phase' is for the first layer 'Base Layer', 'L1 Motion Phase' is for the second layer 'Fighting', etc.

Using the Motion Phase parameter as a trigger, the code will set the parameter to a value. Then the Animator will use that value as a gate for moving the animation forward.

Jump Animation

Look again at the jump motion script. The '`Activate`' function occurs when the trigger tests (described above) have passed. In this function, the motions prepare themselves to run.

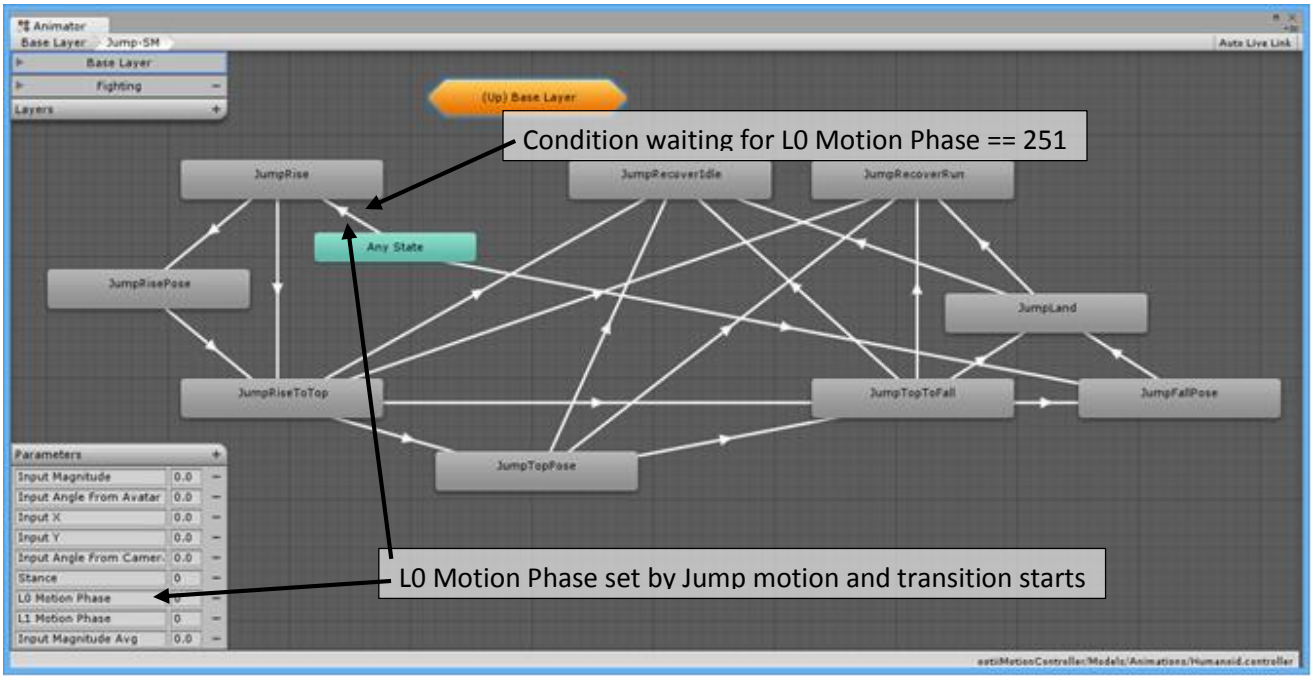
About midway through the jump's `Activate` function, you'll see a line:

```
mController.SetAnimatorMotionPhase(mAnimatorLayerIndex, Jump.PHASE_START_JUMP);
```

This is the call that sets the Motion Phase parameter in the Animator. Each motion could have a set of enumerations that are the parameter values that actually trigger the Animator to animate.

In this case, the `Jump.PHASE_START_JUMP` value is set to 251.

> Every motion in the project should use a unique set of enumeration values.
>
> Before others start creating motions and sharing them, I'll have to create a way to claim and register a set of values.
>
> Unfortunately, we can't use strings for transition conditions.



Motion Flow

How the animation and motion flows from here is unique to the motion. Some motions may start an animation, wait for the animation to complete, and then call '`Deactivate`' to shut themselves down. Some motions may manage each state in the Animator's state machine using the Motion Phase parameters. It really depends on the motion.

Adventure Camera & Rig (optional)

The Adventure Camera found on the Asset Store works great with the Motion Controller. It will give your third-person game a AAA game feel.

The only thing you need out of the Adventure Camera & Rig package is the "AdventureRig.cs" or "IndAdventureRig.cs" file found in:

`Assets\ootii\AdventureCamera\Code\Cameras`

To integrate Adventure Camera, load up your Motion Controller project and follow these simple steps:

1. **Get** the latest Adventure Camera & Rig from the Asset Store:

   https://www.assetstore.unity3d.com/#/content/13768

2. **Import** the Adventure Camera & Rig into your Motion Controller Project.

   As soon as you import the Adventure Camera & Rig, you'll get an error about 'InputManager'. Go to the next step…

3. **Delete** the Adventure Camera & Rig's InputManager.cs file.

   The Motion Controller contains a file with the same name. That's because these files are specific to each asset. Just delete the Adventure Camera & Rig's file found in "`Assets\ootii\AdventureCamera\Code\Input`".

4. Add the AdventureRig to the scene by following Motion Controller step #5 (on page 11).

   If you're opening the Motion Controller's "AdvDemo" scene, it's setup to use the Adventure Camera. You just need to click on the "AdvCameraRig" in the hierarchy and ensure the "AdventureRig" script is set as a component.

5. Ensure the camera rig 'Is Late Update Enabled' flag is unchecked.

   The camera needs to wait until the controller has fully updated until it can position itself. Typically, the "LateUpdate" function is fine. However, the controller uses that function to handle moving and rotating platforms.

6. Use the 'Adventure Forward' motion instead of the 'Simple Forward' motion.

   This will enable pivots and allow you to use the controller similar to how Uncharted and Tomb Raider do.

7. If you've added the 'Casual Idle' motion to your list of motions, ensure the 'Rotate With View' option is set to false. The Adventure Camera will handle this.

With the AdventureRig.cs added and modified, you can now add it to the scene and child the "Main Camera" to it per the Adventure Camera & Rig instructions.

That's it.

Easy Input (optional)

The Motion Controller comes with a basic solution for handling the keyboard, mouse, and Xbox controller. However, if you're using a custom input solution you'll need to make changes (see Input).

If you're using Easy Input as your input solution, this section is for you. To integrate Easy Input, load up your Motion Controller project and follow these simple steps:

1. **Get** the latest Easy Input package from the Asset Store:

   https://www.assetstore.unity3d.com/en/#!/content/15296

2. **Import** Easy Input into your Motion Controller Project.

   As soon as you import the package, you'll get errors. Go to the next steps…

3. **Delete** the Motion Controller's InputManager.cs file.

   The Motion Controller contains a file with the same name. That's because these files are specific to each asset. Just delete the Motion Controller's file found in:

   `Assets\ootii\MotionController\Code\Input`

4. **Unzip** the Easy Input projects settings

   In the `Assets\ootii\EasyInput\Extras` folder you'll find a zip file that contains a new InputManager.asset file. Simply unzip this file to your ProjectSettings folder.

   You may need to restart Unity so it reads the new asset file.

5. **Add** MCGameCode to the scene.

   Some motions are going to call specific input aliases. Easy Input has these setup for you. Just add a new GameObject to the scene. Add the MCGameCode script as a component. It can be found in:

   `Assets\ootii\EasyInput\Extras`

With that, you're ready to use Easy Input with the Motion Controller.

If you open MCGameCode.cs, you'll see how input aliases are setup. Use this and the Easy Input guide to create new key bindings.

That's it.

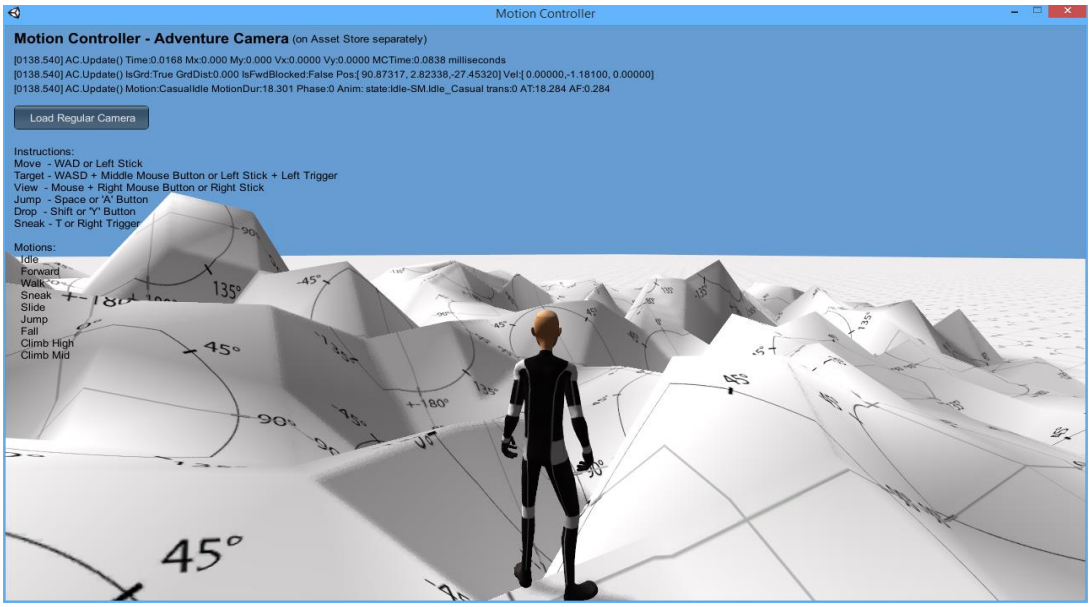
Terrain Considerations

The truth is that there isn't enough processing power even in modern computers to duplicate real-world physics for all objects all of the time. As game developers, we can't possibly model every situation that occurs in the real world.

So, we become masters at hiding seams, blocking pathways, and manipulating terrain to give the world a "free" feel even if it's not.

Why do I say this? Because you have to be prepared that the Motion Controller (or any character controller) isn't going to be able to handle everything perfectly all the time.

The reason you've never seen a game use this for terrain…



Is that there's simply too much opportunity for the character to slide, get stuck, or just create a bad experience.

Level designers for games like Tomb Raider and Uncharted go out of their way to design levels that prevent the player from getting stuck.

While the Motion Controller does a pretty good job at handling this terrain, don't do it.

Some tips:

1. Smooth out concave areas that can trap the collision capsule.
2. Use invisible blocking geometry to prevent the character from getting wedged in.
3. Play your character 1,000 time in all areas to make sure the player can't get stuck.

Some Resources (tools may be different, but techniques are the same):

- http://www.paladinstudios.com/2013/07/05/building-levels-in-unity-part-1-of-3/
- https://developer.valvesoftware.com/wiki/Level_Design_Video_Tutorials
- http://www.worldofleveldesign.com/

Motion Builder's Guide

Check out the Motion Builder's Guide at:

http://www.ootii.com/unity/motioncontroller/MCMotionBuilder.pdf

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