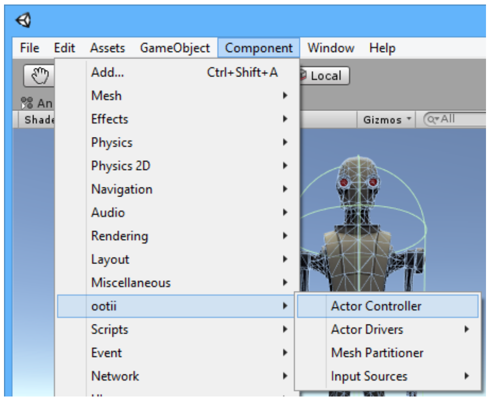
ACTOR CONTROLLER

9/15/2016

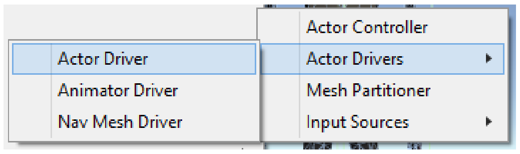
Quick Start

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

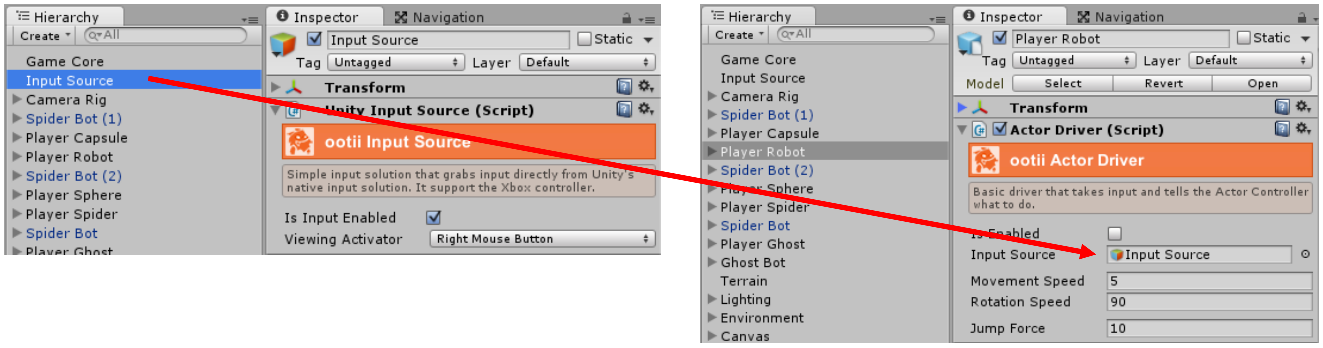
2. Select your character and add an **Actor Controller**.



3. Add an **Actor Driver** to your character.



4. Add an **Input Source** to the scene and assign it to your Actor Driver.



Input Source — Static — Tag: Untagged — Layer: Default

Simple input solution that grabs input directly from Unity's native input solution. It support the Xbox controller.

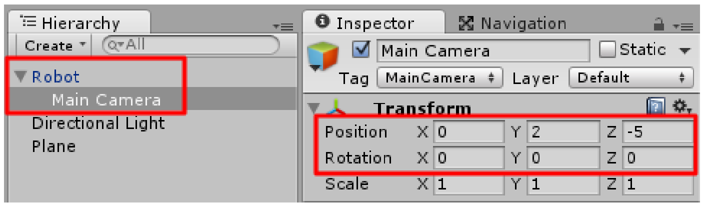
Is Input Enabled ✓
Viewing Activator: Right Mouse Button

Player Robot — Static — Tag: Untagged — Layer: Default — Model: Select, Revert, Open

Basic driver that takes input and tells the Actor Controller what to do.

Is Enabled ☐
Input Source: Input Source
Movement Speed: 5
Rotation Speed: 90
Jump Force: 10

5. Parent the **camera** to your character (optional).



Main Camera — Static — Tag: MainCamera — Layer: Default

	X	Y	Z
Position	0	2	-5
Rotation	0	0	0
Scale	1	1	1

Foreword

Thank you for purchasing the Actor 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/actor-controller-an-advanced-character-controller.360976

Tim
tim@ootii.com

Overview

The Actor Controller is a replacement for Unity's standard character controller and provides advanced features.

While you can use it alone in any Unity 5 solution, it is also the foundation for the Motion Controller.

Features

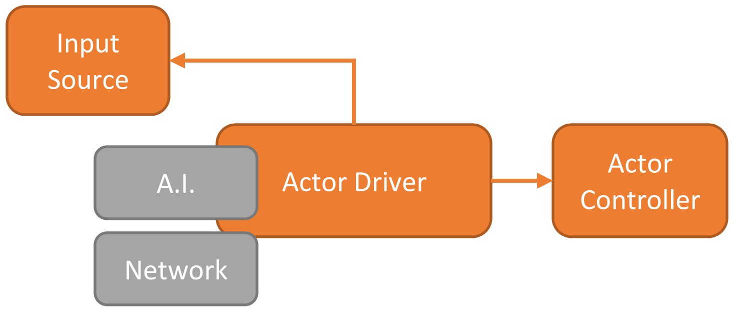
The Actor Controller supports the following features:

- Walk on walls, ceilings, etc.
- Move and rotate on platforms
- Move super-fast
- Slide on steep slopes
- Orient character to ground slope
- Create custom body shapes
- React to external forces
- Supports Nav Mesh Agents
- Supports root-motion
- Supports any input solution
- Zero Garbage Collection *
- Includes code (C#)

* Zero garbage collection when running in release mode. While in debug, some unity physics calls create minor amounts of garbage.

The Basics

In order to keep the Actor Controller as flexible as possible, I've created a clear distinction between the Actor Controller, the Actor Driver, and the Input Source. This allows you to use different components with the Actor Controller: including assets from other developers.



Each update, the **Actor Driver** gathers input from an **Input Source** or other creates its own and tells the actor what to do.

The **Actor Controller moves and rotates** while respecting the environment.

Actor Controller

Controls full-body character movement and rotation in response to the environment and external forces.

Actor Driver

Determines how the Actor Controller moves based on user input, AI, etc. The Actor Drivers I've included are usable, but can also be expanded on. In fact, you can create your own driver as needed.

Note: If you own the Motion Controller, the Motion Controller IS the driver.

Input Source

Used to gather user input from the keyboard, mouse, gamepads, etc. By splitting this out, you could use the basic Unity Input Source provided or create an input source that uses a 3rd party input solution.

With this approach, you can use any input solution you want. You can also change how your actor moves (ie walking vs. flying) by changing how the Actor Driver directs the Actor Controller.

This also means that different characters in the same scene could be controlled in different ways or even by different players.

Custom Input Sources

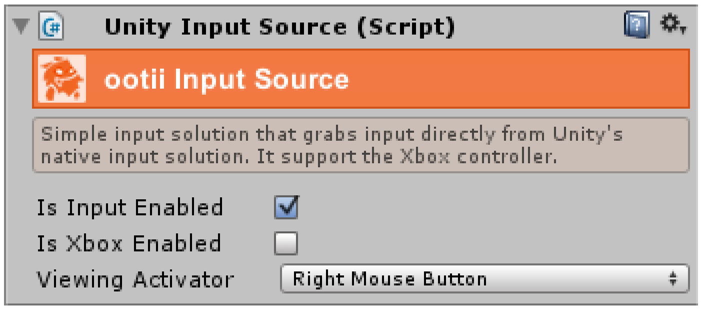
Remember, the input source that I've included is optional. You can use any input solution you want by creating an input source.

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

Unity Input Source – Assets/ootii/Framework_v1/Code/Input

This is a default input source that just uses Unity's native Input Manager solution to read input. It supports basic movement and viewing with the keyboard, mouse, and Xbox controller.



Simple input solution that grabs input directly from Unity's native input solution. It support the Xbox controller.

Is Input Enabled ✓
Is Xbox Enabled ☐
Viewing Activator: Right Mouse Button

To enable the Xbox controller, just check the box. The appropriate values will be added to Unity's Input Manager list (if needed).

The 'Viewing Activator' property allows you to determine how rotation/viewing is activated. For example, when set to 'Right Mouse Button', actors will only rotate when the right mouse button is held down.

In the end, it's the driver's responsibility for honoring the input source as needed.

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.

Custom Actor Drivers

Remember, the drivers that I've included are optional. These drivers handle basic movement cases and work for a variety of situations. However, you can also create your own driver to control the actor however you want.

In the end, the drivers are really calling functions in the Actor Controller:

```
ActorController.Move()
```

```
ActorController.RelativeMove()
```

```
ActorController.Rotate()
```

There are several other functions you can use when creating your own driver, but the ones above are the basics.

Another useful function for applying forces is:

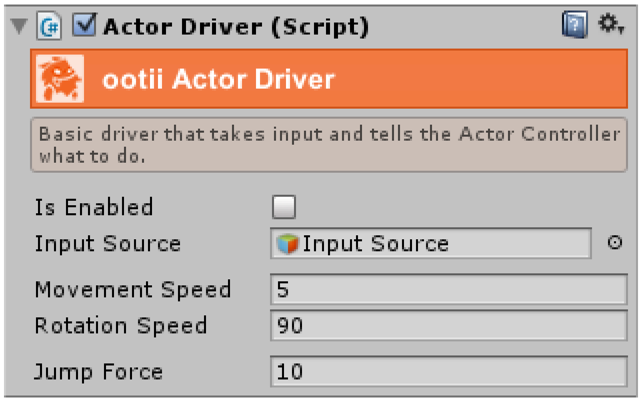
```
ActorController.AddImpulse()
```

This can add a force for things like a jump. However, you should NOT use AddImpulse() to move the character normally as the AC is not a physics-based controller, but an input-based controller.

Included in the package are several Actor Drivers that you can use:

Actor Driver – Assets/ootii/ActorController/Code/Actors/CharacterControllers

This is the default driver. It takes input from the keyboard, mouse, and Xbox controller and turns that into movement and rotation that is relative to the character's forward direction. It then calls the Actor Controller functions to actually move the actor.



Basic driver that takes input and tells the Actor Controller what to do.

Is Enabled	☐
Input Source	Input Source
Movement Speed	5
Rotation Speed	90
Jump Force	10

Animator Driver – Assets/ootii/ActorController/Code/Actors/CharacterControllers/Drivers

Inherits from Actor Driver, but looks for a Unity Animator that is attached to the game object. If found, it will query for root-motion data and use that to move and rotate the character. If no root-motion data is found, input will be used to control the character.

Nav Mesh Driver – Assets/ootii/ActorController/Code/Actors/CharacterControllers/Drivers

Inherits from Animator Driver, but uses a Nav Mesh Agent to move the actor to a specific target. If an Animator is found, it will query for root-motion data and use that to move and rotate the character. If no root-motion data is found, speed will be set on the component.

Sphere Actor Driver – Assets/ootii/_Demos/ActorController/Code

Inherits from Actor Driver. When an "inner" sphere is found it will use this the actor's body and rotate it based on the direction the character is moving. This give the impression that the sphere is rolling.

Spider Actor Driver – Assets/ootii/_Demos/ActorController/Code

Inherits from Animator Driver. Similar to the other drivers, but when jump is pressed (and the actor is facing a wall), it will jump onto the wall so it can climb.

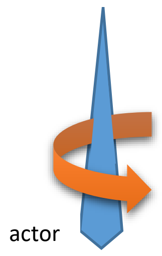
Coding an Actor Driver

Coding an Actor Driver is really just a matter of reading the input and then telling the Actor Controller what to do.

To code the AC, you need to understand that rotations are broken up into two pieces; yaw and tilt.

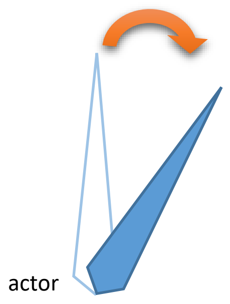
Yaw by rotating around the up axis.

Both the base and top stay in the same position, but the actor spins relative to its current up axis.



Tilt by rotating around the forward and right axis.

The base is rooted, but this actor's top moves around.



Typically, you'd use the following function of the Actor Controller:

Move(Vector3)	Moves the actor based on world space.
RelativeMove(Vector3)	Moves the actor based on local space.
SetPosition(Vector3)	Forces the absolution position value.
Rotate(Quaternion)	Rotates the actor around its "up" axis. This rotation is typically called 'yaw'.
Rotate(Quaternion, Quaternion)	Rotates yaw and tilt.
SetRotation(Quaternion)	Forces the absolute rotation value.

Simple Driver Code

```
using UnityEngine;
using com.ootii.Actors;
using com.ootii.Helpers;
using com.ootii.Input;

public class SimpleDriver : MonoBehaviour
{
    public GameObject _InputSourceOwner = null;
    public float MovementSpeed = 5f;
    public float RotationSpeed = 120f;

    protected IInputSource mInputSource = null;
    protected ActorController mActorController = null;

    void Start()
    {
        mActorController = gameObject.GetComponent<ActorController>();
        mInputSource = InterfaceHelper.GetComponent<IInputSource>(_InputSourceOwner);
    }

    void Update()
    {
        // Rotate based on the mouse
        if (mInputSource.IsViewingActivated)
        {
            Quaternion lRotation = Quaternion.Euler(0f, mInputSource.ViewX, 0f);
            mActorController.Rotate(lRotation);
        }

        // Move based on WASD
        Vector3 lMovement = new Vector3(mInputSource.MovementX, 0f, mInputSource.MovementY);
        mActorController.RelativeMove(lMovement * MovementSpeed * Time.deltaTime);
    }
}
```

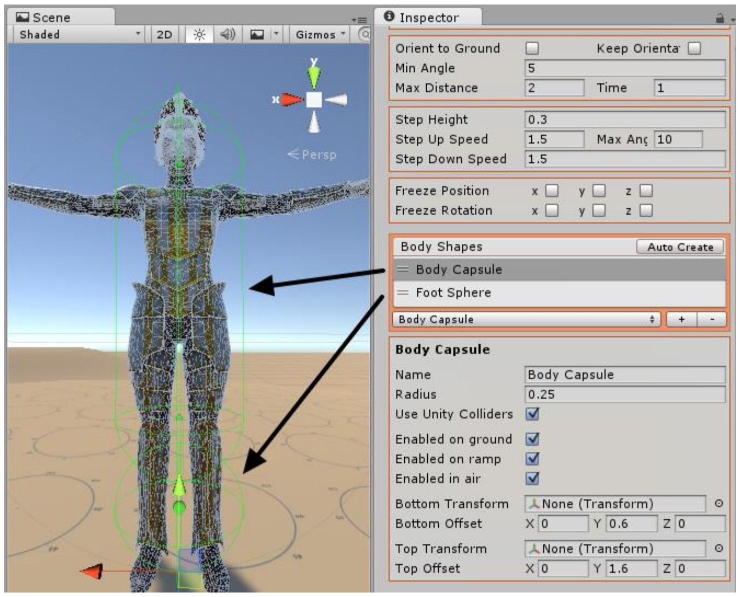
If you wanted your character to tilt, you could change the rotation code inside the Update() function to look something like this:

```
            Quaternion lRotation = Quaternion.Euler(0f, mInputSource.ViewX, 0f);

            float lTiltAngle = (mInputSource.IsPressed(KeyCode.E) ? 1f : 0f);
            lTiltAngle = lTiltAngle + (mInputSource.IsPressed(KeyCode.Q) ? -1f : 0f);
            Quaternion lTilt = Quaternion.Euler(0f, 0f, lTiltAngle * 5f);

            mActorController.Rotate(lRotation, lTilt);
```

Notice that the last line uses the second version of the Rotate() function. This one includes the ability to tilt.

Body Shapes and Colliders

Body shapes are used for collision detection by the Actor Controller. By using spheres and capsules, we can represent the shape of a human as well as other non-simple characters. We also have the ability to change these shapes during run-time in order to match the character's pose.



```
Orient to Ground  [ ]        Keep Orienta [ ]
Min Angle         5
Max Distance      2          Time  1

Step Height       0.3
Step Up Speed     1.5        Max Ang 10
Step Down Speed   1.5

Freeze Position   x [ ]  y [ ]  z [ ]
Freeze Rotation   x [ ]  y [ ]  z [ ]

Body Shapes
  Body Capsule
  Foot Sphere
Body Capsule

Body Capsule
Name               Body Capsule
Radius             0.25
Use Unity Colliders [✓]
Enabled on ground  [✓]
Enabled on ramp    [✓]
Enabled in air     [✓]
Bottom Transform   None (Transform)
Bottom Offset      X 0   Y 0.6   Z 0
Top Transform      None (Transform)
Top Offset         X 0   Y 1.6   Z 0
```

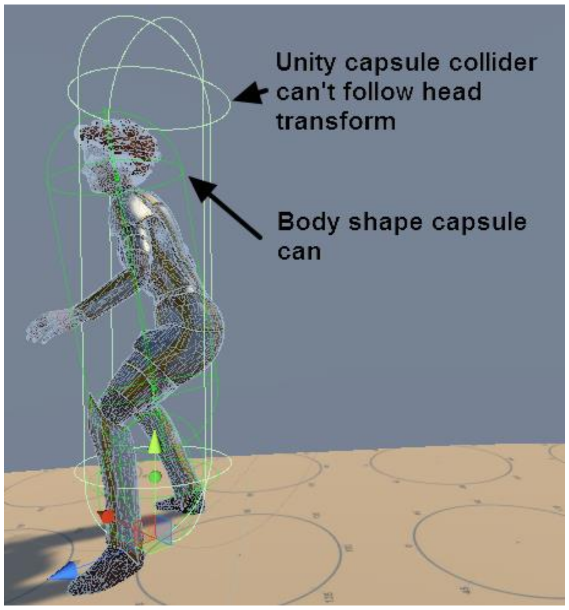
The default setup has two shapes:

Body Capsule – This capsule is similar to a traditional character controller, but is raised off the ground. This allows the actor to get right up to the edge of a step… which is great for foot IK.

Foot Sphere – This sphere is only active while in the air. This keeps our feet from entering things as we jump over objects. However, it doesn't block us when we move close to a ledge or step.

It's important to note that body shapes are NOT colliders. So, external raycasts or rigid-bodies won't react to them. If you want an external raycasts to hit your character, you can add traditional Unity colliders to your actor as you normally would or check the 'Use Unity Colliders' checkbox.

The reason I use body shapes instead of colliders is that the normal Unity capsule collider isn't very flexible. You can't rotate it arbitrarily or tie it to different transforms. That means its shape won't automatically change as your character animates. For example:

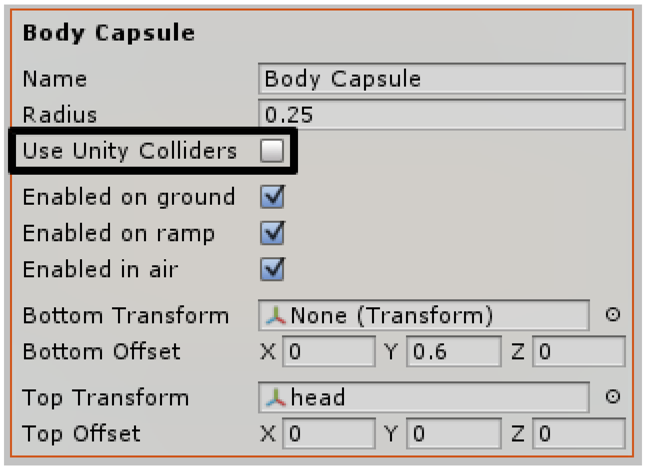


Notice how the Unity capsule collider can't resize automatically to match the head transform. It's also stuck in the cardinal directions.

However, the body shape capsule will resize and rotate with the transforms it's tied to. So, your character's shape will change automatically.

Unity Colliders

To help compensate for the Unity capsule collider restrictions, there is an option on each body shape that will create and manage Unity colliders using the body shapes.

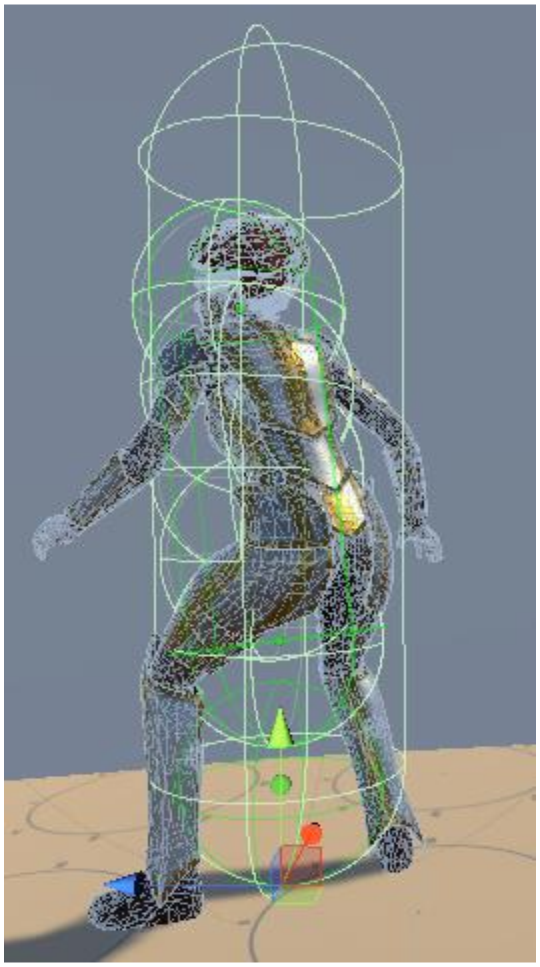


Body Capsule

Name	Body Capsule
Radius	0.25
Use Unity Colliders	☐
Enabled on ground	✓
Enabled on ramp	✓
Enabled in air	✓
Bottom Transform	None (Transform)
Bottom Offset	X 0 Y 0.6 Z 0
Top Transform	head
Top Offset	X 0 Y 0 Z 0

Simply check the “Use Unity Colliders” checkbox and colliders will be created at run-time to match the body shapes.

Spheres are easy. However, since we can’t really use Unity capsule colliders, spheres are used instead.

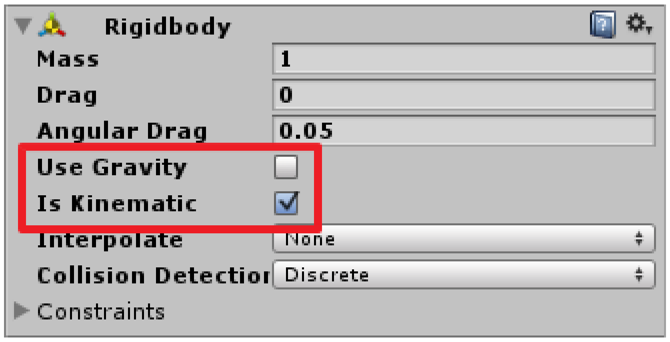


It’s a little cluttered, but notice how we use multiple sphere to approximate the body shape capsule’s position, size, and rotation.

As the character’s head moves, the spheres will move to keep the capsule’s shape.

Rigidbody

Adding a Rigidbody component to your character is fine. However, the rigidbody will attempt to control your character. So, that means the rigidbody and AC are competing. To fix this, disable gravity on the rigidbody and check the "Is Kinematic" check box. This will stop the rigidbody from trying to control your character.



Rigidbody	
Mass	1
Drag	0
Angular Drag	0.05
Use Gravity	☐
Is Kinematic	✓
Interpolate	None
Collision Detection	Discrete

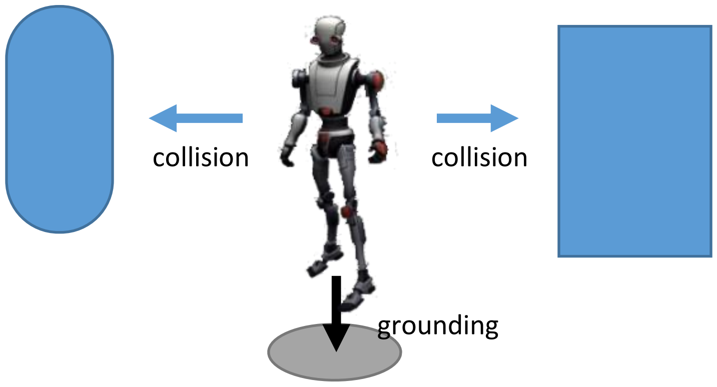
Collisions vs. Grounding

"Colliding" and "Ground" are two different things.

Colliding means collision detection has determined you've bumped into another object… say a wall.

Grounding means we've tested the ground directly under the actor and they can stand on it.

So, you can completely disable collisions and your character will still be able to walk on the ground, move up slopes, etc. There is a huge performance boost when you disable collisions. This is great for NPCs that may be on rails or have limited movement.



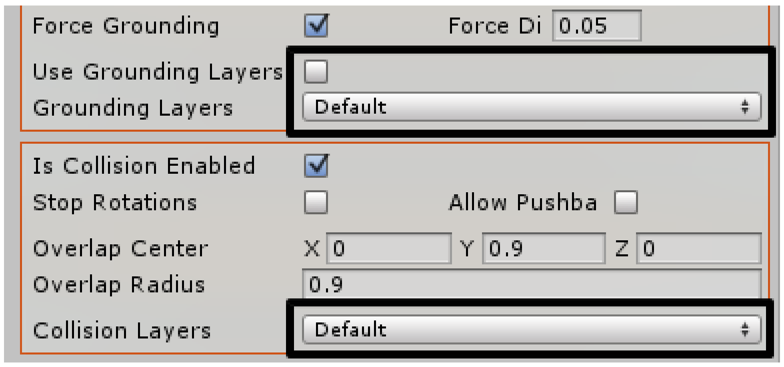
Colliders

Actor Controller is able to collide against standard unity colliders. The fastest colliders are sphere, box, and plane. However, it can also detect collisions against mesh colliders. However, like Unity's Character Controller, there's a performance impact when colliding against mesh colliders.

With Unity 5.3, the AC is now using Unity's non-allocating physics calls. This is great for mobile as collisions no longer create garbage that can cause performance blips. However, these calls cause collisions with back-faces of planes (and probably other shapes).

Collision Layers

The Actor Controller uses standard Unity's layers to determine what to collide with.



Force Grounding ✓ Force Di 0.05
Use Grounding Layers ☐
Grounding Layers Default
Is Collision Enabled ✓
Stop Rotations ☐ Allow Pushba ☐
Overlap Center X 0 Y 0.9 Z 0
Overlap Radius 0.9
Collision Layers Default

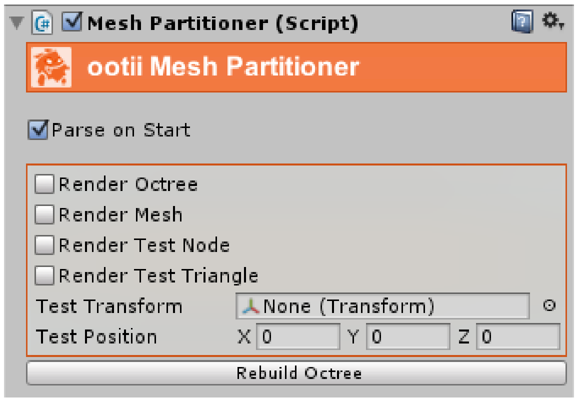
Simply set the layers in the collision section to collide with objects and props that are on the same layer. If an object is on a layer that isn't included in the Collision Layers, the character will simply go through the object.

To enable layers with grounding, you must also check the "Use Grounding Layers" checkbox.

Mesh Colliders

To help minimize the impact, we parse the mesh collider and store its information for future collisions. For small meshes, this can be done on first impact. However, for large meshes you'll want the Actor Controller to pre-process the mesh collider.

To do this, add an ootii Mesh Partitioner component to the game object that has a Mesh Collider object. Then, check 'Parse on Start'.



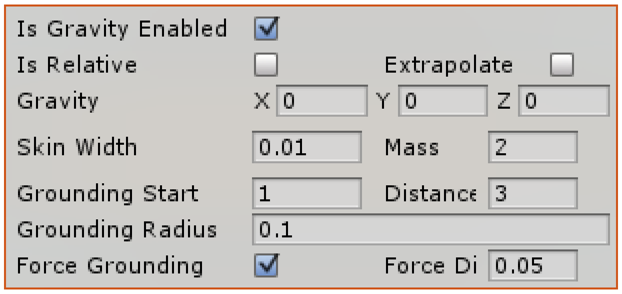
- [✓] Parse on Start
- [ ] Render Octree
- [ ] Render Mesh
- [ ] Render Test Node
- [ ] Render Test Triangle

Test Transform: None (Transform)
Test Position: X 0 Y 0 Z 0

All the remaining options are for visualizing the partitions in the editor and should not be enable.

Actor Controller Advanced Settings

Grounding



Is Gravity Enabled	✓		
Is Relative	☐	Extrapolate	☐
Gravity	X 0	Y 0	Z 0
Skin Width	0.01	Mass	2
Grounding Start	1	Distance	3
Grounding Radius	0.1		
Force Grounding	✓	Force Di	0.05

Is Gravity Enable
Determines if we use gravity or not.

Is Relative
Determines if gravity is based on the world orientation (Vector3.up) or the ground surface's normal.

In order to walk-on-walls, this option must be set.

Extrapolate Physics
The Actor Controller uses the LateUpdate() function for all its processing. However, physics based information (like gravity and forces for jumps) are processed in FixedUpdate() in order to be consistent across frame-rates. These two functions aren't always in synch.

If you notice stuttering while falling, checking this options will smooth that out.

Gravity
Determines the force and direction of gravity. If no value is set, we use Unity's gravity value.

The final gravity direction will be determined by the **Is Relative** flag.

Skin Width
Small amount of room allowed between the body shapes of the actor and objects they collide with or ground to.

Mass
Mass of the actor. We use this value when applying external forces. We use Unity's standard which is roughly 1 unit = 1 cube = ~35 kg.

Grounding Start
When shooting the ray for grounding, this is the height above the character's root that we'll start shooting the ray downward.

Grounding Distance
Determines how far below the character we'll shoot the grounding ray.

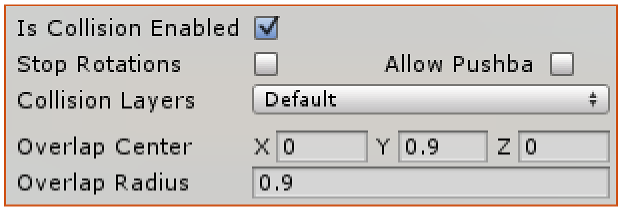
Grounding Radius
In addition to a ray, a sphere is cast downward in order to help prevent falling into gaps. This value should be the radius of the "feet" or bottom of the actor.

Force Grounding

Determines if we'll push the character down to the ground if they are within the **Force Distance** range.

Force Distance
Maximum distance the character can be from the ground and we can still force them too it when **Force Grounding** is checked.

Collisions



Is Collision Enabled ✓
Stop Rotations ☐ Allow Pushba ☐
Collision Layers Default
Overlap Center X 0 Y 0.9 Z 0
Overlap Radius 0.9

Is Collision Enabled
Determines if we'll test for collisions

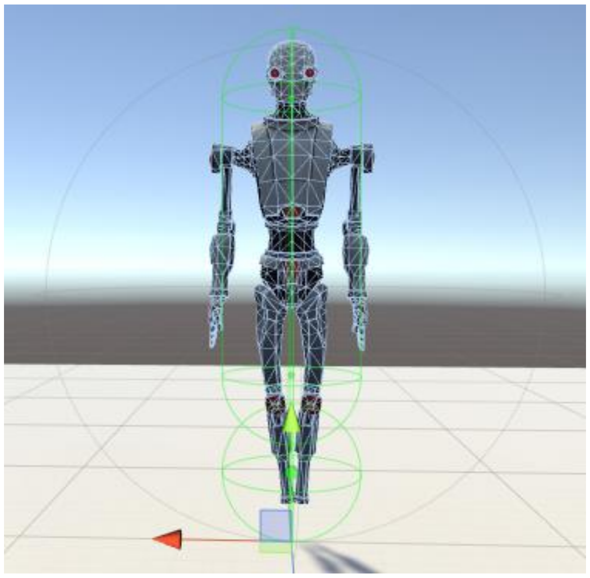
Stop Rotations
Determines if collisions will stop the character from rotating

Allow Pushbacks
Determines external colliders will cause the actor to move.

Assume the actor is just standing idle, but a collider is moving to the character. This object will allow the collider to push the character back in order to keep the character from penetrating it.

However, like the Unity Character Controller, the collider could push the character into another collider. So, care should be taken with this.

There is also a performance penalty for using this option. Use it sparingly.

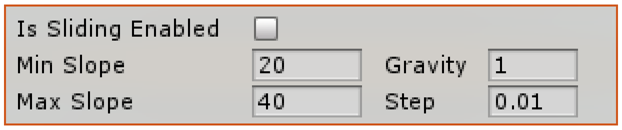


Overlap Center
When performing collision detection, we first see if there are objects within the range of the character. This is the center of that range.

Overlap Radius
When performing collision detection, we first see if there are objects within the range of the character. This is the radius we use.

In the editor, you'll see a light gray sphere representing the overlap circle that will be used for collision tests. Use the "Overlap" settings to ensure the majority of your character is covered.

Sliding



Is Sliding Enabled	☐		
Min Slope	20	Gravity	1
Max Slope	40	Step	0.01

Is Sliding Enabled
Determines if the actor will slide when on a ramp whose angle is greater than the **Min Slope** angle.

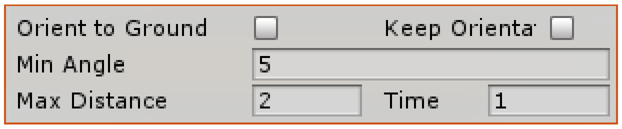
Min Slope
Minimum angle the slope (under the actor) needs to be before the actor starts to slide.

Gravity
When sliding, the percentage of gravity that will be applied to the slide. This helps to fake friction on the surface.

Max Slope
The maximum slope an actor can move up. If a slope is greater than this, the character will simply stop moving.

Step
Small distance used internally to find the closes point to where the slope actually starts. This value typically doesn't need to be changed

Orientation



Orient to Ground	☐	Keep Orienta	☐
Min Angle	5		
Max Distance	2	Time	1

These features are important when a character is mean to walk on walls or ceilings. In order to do this, the character must orient to the surface they are walking on.

Orient to Ground
Determines if the actor will change its "up" vector to match the normal of the surface he is standing on.

In order to walk-on-walls, this option must be set.

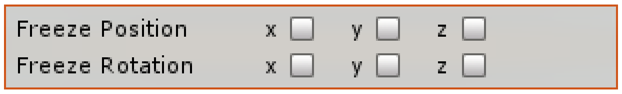
Keep Orientation
Determines if the actor will keep the last orientation it had while jumping. When not checked, the character will return to the natural orientation (Vector3.up) while jumping.

Min Angle
Minimum angle change that will cause an orientation change.

Max Distance
The maximum from the ground before the actor's orientation will start to change to the natural ground (Vector3.up).

Time
The number of seconds it takes to reach the new orientation.

Freezing

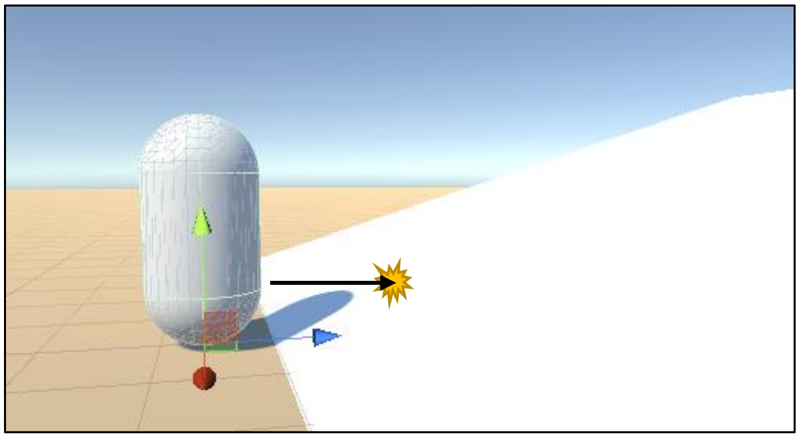

Freeze Position x ☐ y ☐ z ☐
Freeze Rotation x ☐ y ☐ z ☐

Setting these options allow you to limit the movement and rotation of the specified axis.

FAQ: Tiny Steep Slopes

In designing the AC, I wanted to be able to limit the slopes characters could go up. This way we could have some characters (like monsters) that could go up steeper slopes than say a human.

However, slopes provide an interesting challenge as you can't predict a slope. Instead, you can only react to slopes. Here's what I mean...



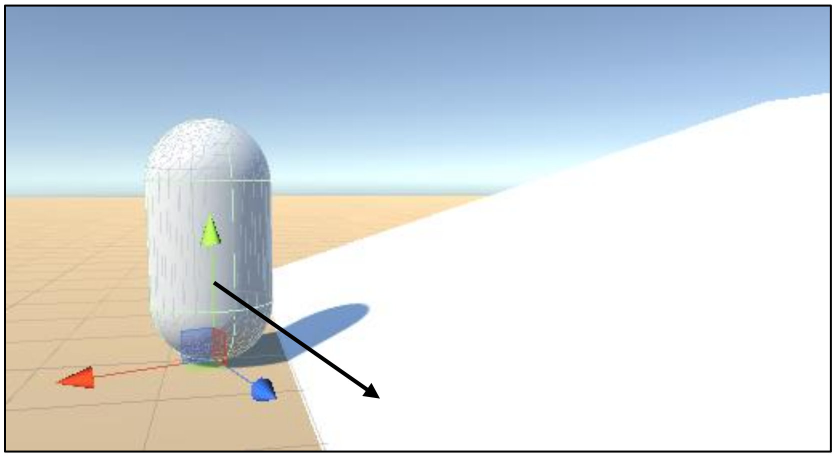
Here we have an actor that is moving directly into a slope. Let's say the slope is too steep and we shouldn't be able to go up it.

In this case, we can shoot a ray ahead of the actor in the direction of movement. When that ray collides with the slope, we can determine the angle and stop.

That's predictive and it works.

But wait... I said you can't predict the slope!

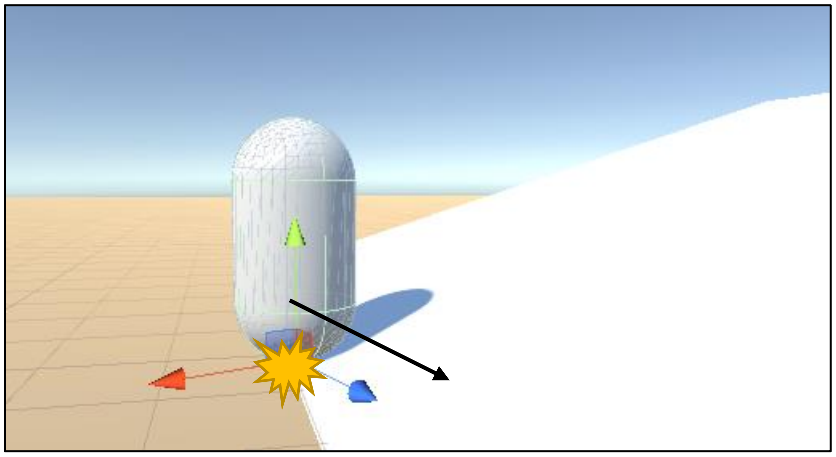
What happens if the character isn't moving directly into the slope? What if he's moving up the slope, but taking a long path.



In this case, we can shoot a ray pretty far forward and never hit the slope. That means we never get an angle to test and we think we're good to move.

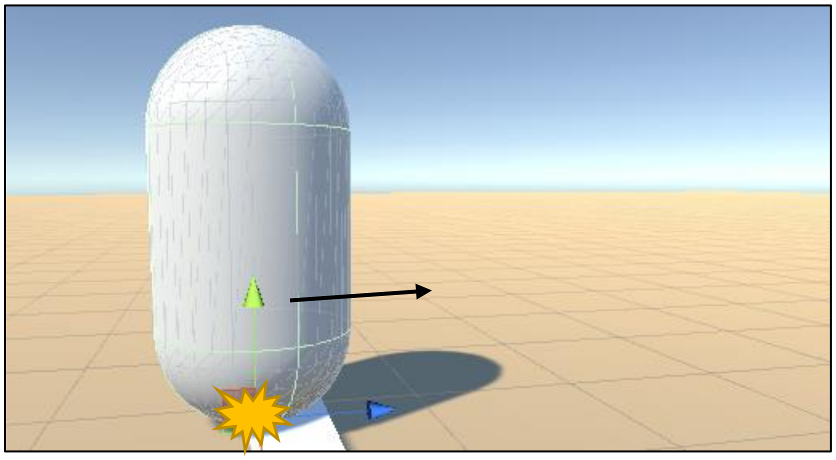
When this happens, we can start doing all sorts of tricks to see if there really is a slope. Unfortunately, each trick adds a tiny performance hit and each trick leaves a hole to be filled by another trick.

So, I've found that the best approach is to test for the slope that is directly under the character. I call this "reactive".



By testing the character's origin, we truly know if we're on a slope regardless if we're moving directly against it or sideways.

The downside is that even the tiniest of slope is tested.



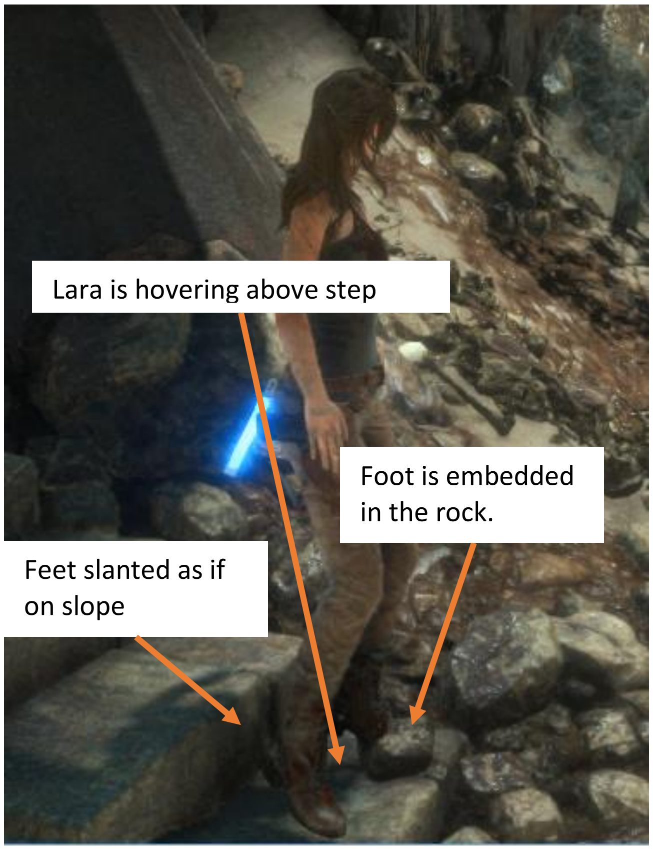
In looking at this picture, we probably should be able to just step right over the slope. After all, the slope is pretty tiny.

However, we can't accurately predict if the area above the "step height" is open if we have the situation I've just talked about.

So, that tiny slope will stop movement.

Does that mean your terrain can never have tiny bumps? Not really. It just means you need to manage it.

Remember that when creating a game, you're not duplicating reality. There simply isn't enough computing power to handle every situation that the real world has to offer. So, even AAA games compromise. The act of creating the game is the act of creating the illusion of reality.


Lara is hovering above step
Foot is embedded in the rock.
Feet slanted as if on slope

Rise of the Tomb Raider (PC 2016)

The goal of this isn't to point out Tomb Raider's flaws... it's one of my favorite games.

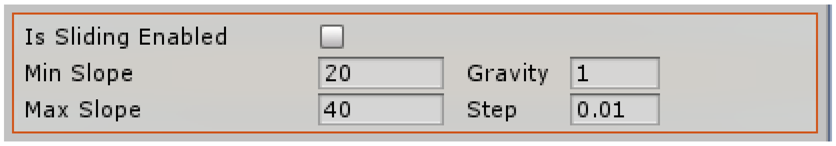
But, it's to show you that even AAA game makers use tricks to balance the playability, the look, and the performance.

As game developers and level designers we do too.

So, what are our options?

Max Slope

Set the Max Slope to something larger that allows you to go up the bump... 85 degrees would be an extreme. Obviously this means your character walks up any slope of that degree or smaller. Depending on your game, that may be a good option.



Is Sliding Enabled	☐		
Min Slope	20	Gravity	1
Max Slope	40	Step	0.01

If you set the Max Slope to 0, internally I default it to 70 degrees.

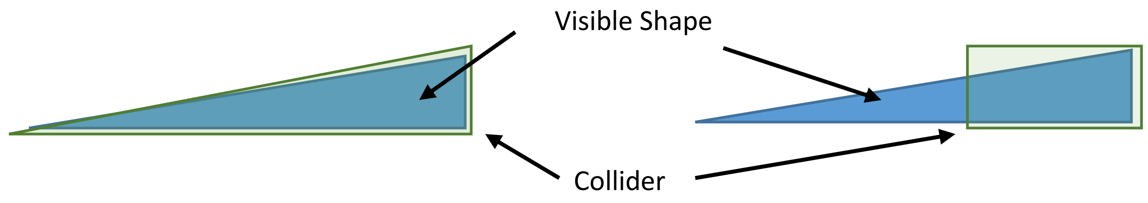
Remove Colliders

Remove the collider from the tiny slope. If you have a tiny slope, does it really matter that the character bumps over it? In the case of a human with two legs, you typically wouldn't even notice it. This is especially valid when you're dealing with terrain.

> I had someone show me a scene where 1,000's of rocks and pebbles were all over the ground. Every rock and pebble was part of the collider in their hill. The collider was a complex and massive mesh.
>
> So, while the overall slope of the hill was fine, each little pebble caused too steep of a slope and the AC would stutter trying to move up the hill.
>
> This goes to my comment about the "illusion of reality". Having every pebble be part of the collider may be closer to reality, but the performance hit it puts on your game typically isn't worth it.

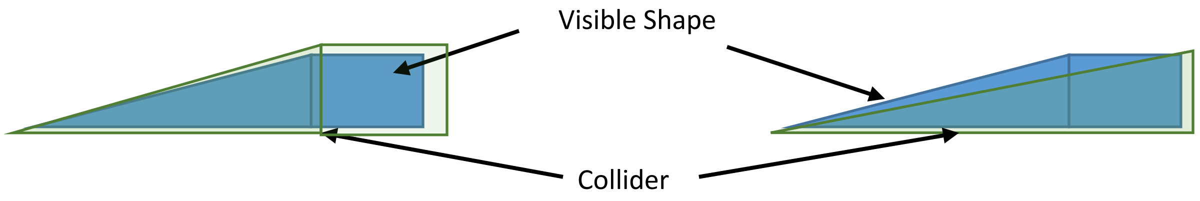
Slopes to Steps

Change the collider from a "slope" to a "step". The AC handles steps and slopes differently. In the case of a step, we'll smoothly pop up. So, your slope can look like a slope, but the collider would actually be a cube with a 90-degree face.



Slopes to Slopes

Maybe you can get away with changing the collider of your steep slope to something not so steep.



Summary

Remember, we're not talking about every slope. We're talking about small steep slopes that exceed the Max Slope property that we've set on the Actor Controller. We're talking about breaking the rule that we set.

Game development is all about compromise.

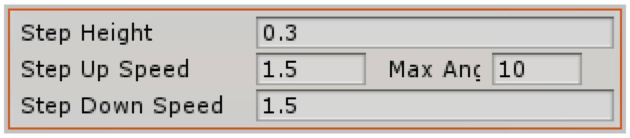
FAQ: Slowly Rising Up Stairs and Slopes

Most character controllers have you “pop” up steps. It works, but it can also be a bit jarring as your character is low one frame and higher the next. This is especially true if you have several steps; you get this pop-pop-pop experience.

The typical solution is to use ramps as the colliders for the steps. This provides a nice smooth way to go up stairs, but your foot IK won’t recognize the individual steps. Instead, if you stop on the stairs your foot IK thinks you’re on a slope and your feet tilt.

Smooth Stepping

So, with the AC, I include a feature called “smooth stepping”. It basically rises you up to the step over time instead of popping you up.



Step Height	0.3		
Step Up Speed	1.5	Max Anç	10
Step Down Speed	1.5		

The “Step Up Speed” determines how quickly (units per second) you’ll move to the higher step.

“Step Down Speed” determines how quickly you’ll move to the lower step.

“Max Angle” is used for slopes. If the ground angle is steeper than this value, we disable smooth stepping.

The Catch

The catch is that sometimes when dealing with a longer set of stairs or with ramps, the character will rise slower than you are moving forward. In this case, the character looks like he’s sunk and slowly rising up.

To fix this, there’s a couple of options:

Disable Smooth Stepping

Disable smooth stepping by setting Step Up Speed and/or Step Down Speed to “0”. This will cause you to pop up and down the steps in a traditional way.

Increase Speeds

The reason this ‘catch’ typically occurs is that your character is moving forward faster than they are moving up/down. So, you can increase the speeds to match the game play. Stepping will be faster and should keep up with your forward movement.

Lower the Max Angle

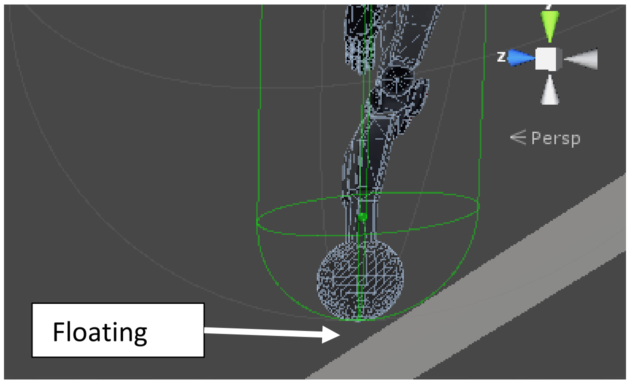
With slopes, you can lower the max angle to disable the smooth stepping on things like terrain. I’ve actually found that “3” is a good number for terrain.

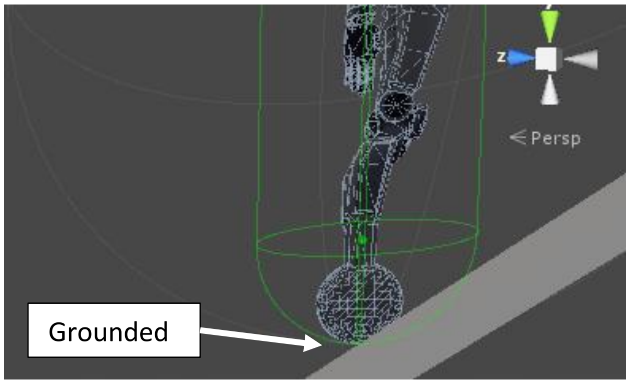
FAQ: Sinking on Steep Slopes

I've had a couple people ask why the character's colliders sink into slopes.

Most characters aren't perfect capsules or spheres. However, we use capsules and spheres to define the collision areas because they are fast for collision detection, provide simplicity, and the curved bottoms can help when moving over bumpy ground. However, steep slopes can be an issue.

This is pretty easy to visualize:



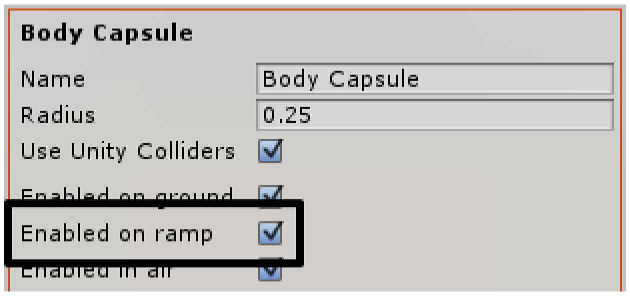


In the first picture, we don't allow sinking. Instead, we simply force the collider to stop on the slope. In this case, the character is no longer "grounded" and actually floating in the air.

Not only will this make it look like your character is floating, but the AC will think the character is floating and movement could be stopped. So, grounding and collisions happen differently as I mention previously in the document.

The other reason for sinking is foot IK. If we strictly adhered to the collider shape, the feet would never reach the ground and we couldn't force the feet to a more natural stance with the environment.

All that said, you do have some control over this. Use the "Enable on ramp" option to determine if the collider should respect ramps or not:



Body Capsule

Name	Body Capsule
Radius	0.25
Use Unity Colliders	✓
Enabled on ground	✓
Enabled on ramp	✓
Enabled in air	✓

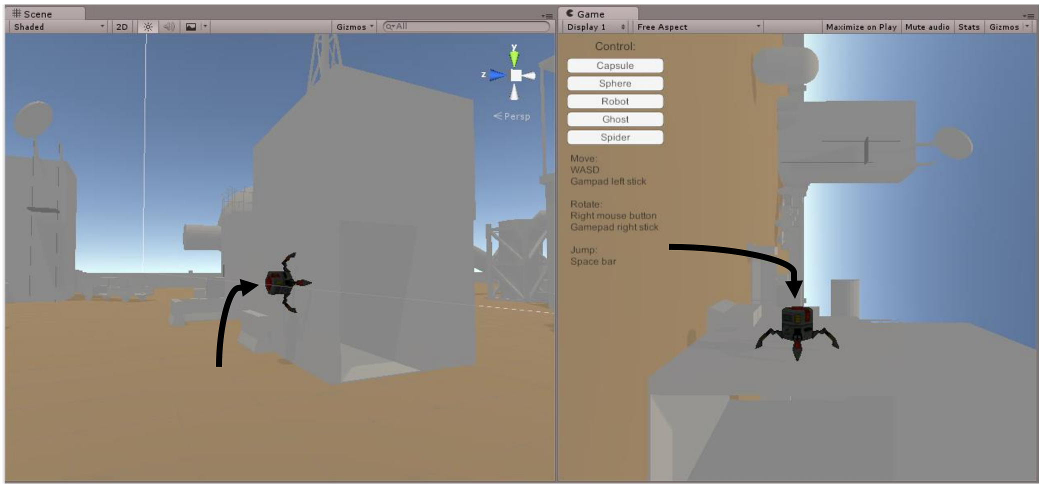
Fair warning: Sometimes I still allow the collider to sink into the ramp... But, only when the code feels it is important.

FAQ: Forcing Orientation

Remember, orientation is based on the ground surface, not gravity. Take a spider walking on the wall... World gravity doesn't change, but its orientation did.

That said, we use the "Is Relative" flag in the gravity section to ensure the spider sticks to the wall by changing its personal gravity. This doesn't cause an orientation change, it just makes sure his gravity is pulling him towards the ground's normal. If the spider is on the wall that means he is pushed towards the wall and now he's "grounded" on the wall.

From time to time, you may want to force your character to have a certain orientation. In fact, I do this in the AC demo (demo_Factor) when the spider jumps to the wall. Check out the SpiderDriver.cs file that is responsible for spider movement.



To do this, you want to use the AC's **SetTargetGroundNormal**() function. This function allows you to force the ground normal that will cause the actor's orientation to change if "Orient To Ground" is checked in the AC settings.

In addition to setting SetTargetGroundNormal, you may want to change the speed it takes for the actor to orient to this direction. You can do that with **OrientToGroundSpeed**.

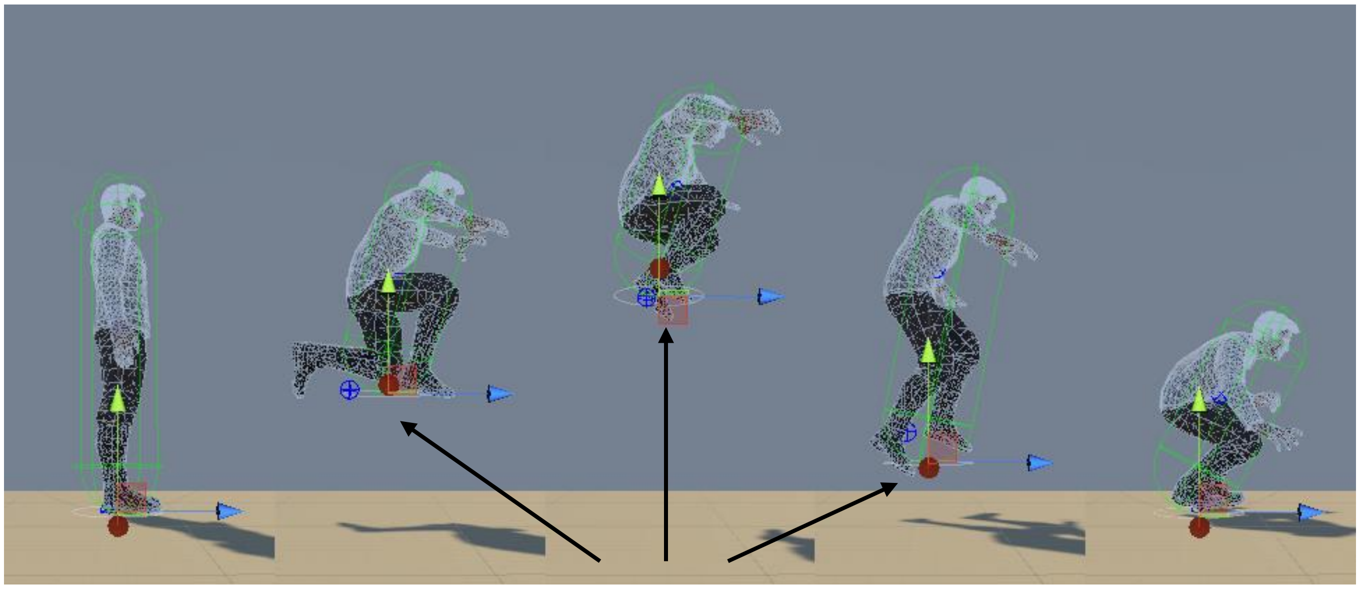
Again, you can see all this happening in SpiderDriver.cs:

- Line 167: Player pressed jump and wall found. Initiating re-orientation
- Line 175: Change the OrientToGroundSpeed based on the player pressing jump
- Line 179: Add impulse so the spider moves up as he changes orientation
- Line 112: SetTargetGroundNormal called and actor starts to rotate
- Line 105: Actor reaches new ground target and SetTargetGroundNormal is cleared

One thing to think about: If your actor were to simply rotate around his origin right where he's at, he'll probably end up embedded in a wall or floor. So, you may need to add some impulse or movement to prevent that. I do that at line 179.

FAQ: Adding Jumping

Jumping can be implemented a couple of different ways. I prefer to use a force that applies lift to the character and then let gravity pull him down.



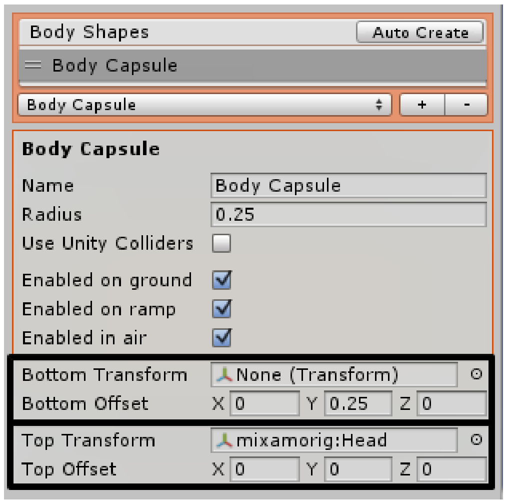
Root moved by physics force

There's a several of advantages to this approach:

1. The AC likes having the character's root remain at his feet; that's the true position of the character. So, having the root move with the jump allows him to jump on-top of objects easily.

2. In game, you can change how high the character jumps just by changing the force of the jump.

3. If you setup your body shape to use the head transform, the body shape will automatically resize to fit the character's pose. See the settings here:



Body Capsule

Name	Body Capsule
Radius	0.25
Use Unity Colliders	☐
Enabled on ground	✓
Enabled on ramp	✓
Enabled in air	✓
Bottom Transform	None (Transform)
Bottom Offset	X 0 Y 0.25 Z 0
Top Transform	mixamorig:Head
Top Offset	X 0 Y 0 Z 0

Notice how the "Bottom Transform" is empty. That means we base it on the root. The "Top Transform" is based on the head bone.

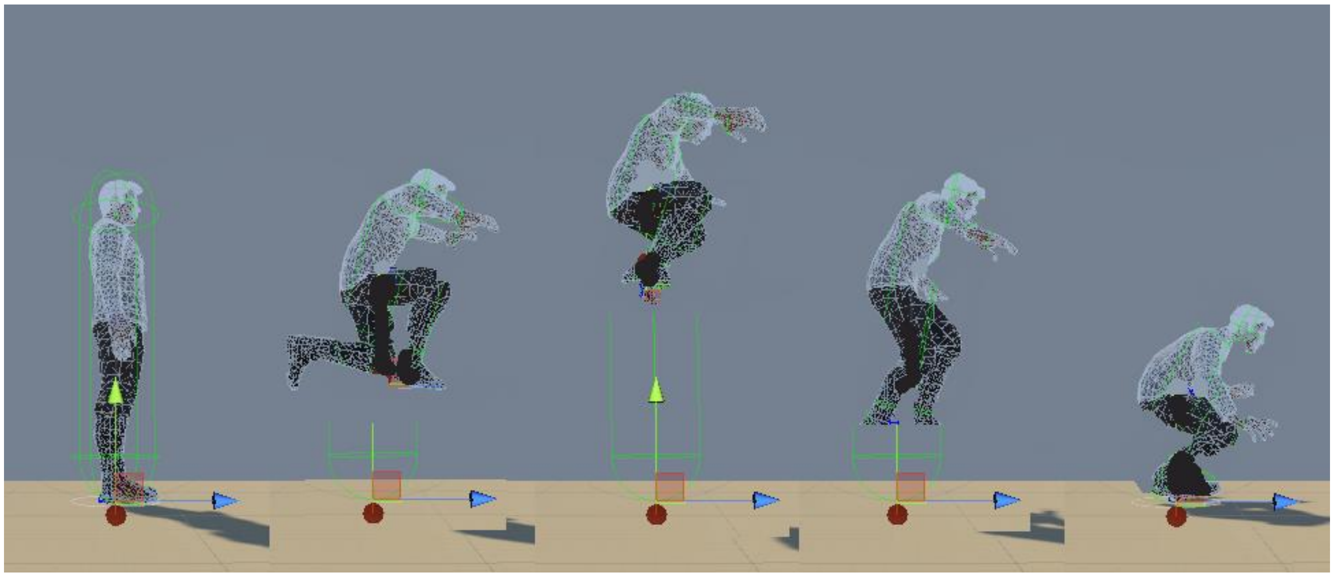
To apply the force, you'd simply make call to the AC's AddImpulse function like this:

```
ActorController mActorController =
gameObject.GetComponent<ActorController>();

mActorController.AddImpulse(transform.up * _JumpForce);
```

If you do go with this approach, typically you'd set it so that your jump animation doesn't have your character rise up. Instead, his feet stay at the root. The raw animation looks a bit wonky by itself, but once you add the impulse, it looks great.

Animation Controlled Jump

If you don't want to use the AddImpulse approach, you can have the animation include the up-movement of the jump. In this case, I'm assuming it has no root-motion. The challenge with this approach is that the root of the character is still on the ground while his body is in the air.



Root does not move with animation

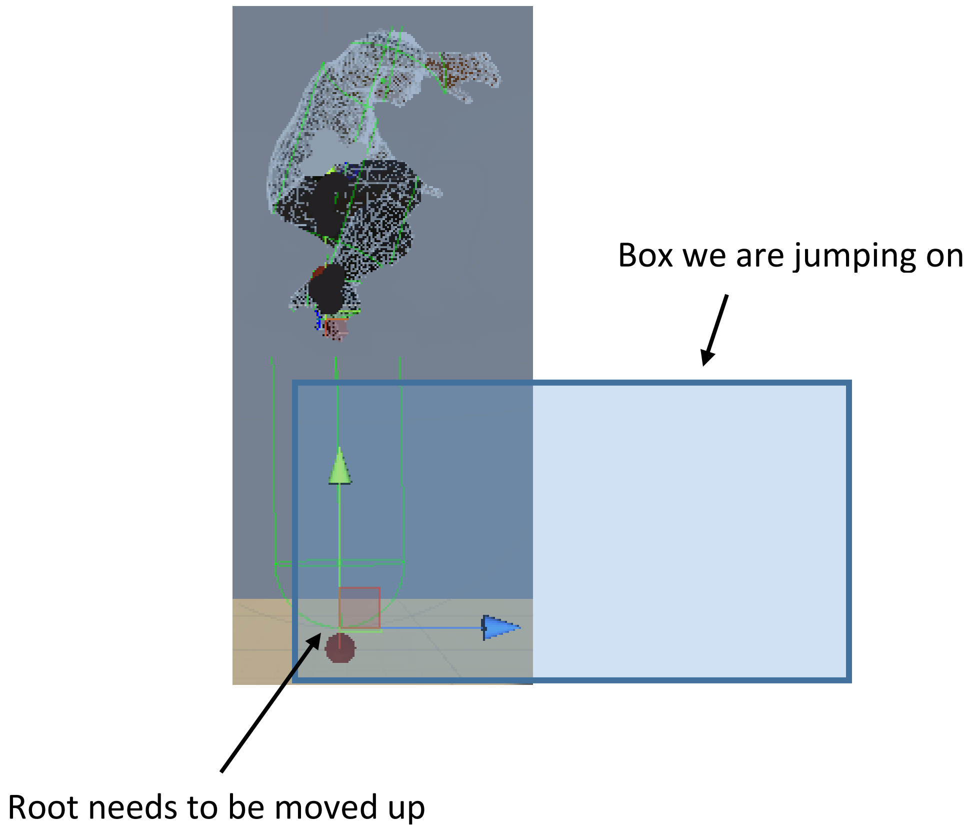
To compensate for this, you'd typically move and/or resize the collider to represent the character's position over the life of the jump.

To do that with the body shapes, you'd use code like this:

```
// Grab the first body shape as a capsule
BodyCapsule lCapsule = mActorController.BodyShapes[0] as BodyCapsule;
if (lCapsule != null)
{
    // Changes the offset of the bottom part of the capsule
    lCapsule.Offset = Vector3.up * 0.25f;

    // Changes the offset of the top part of the capsule
    lCapsule.EndOffset = Vector3.zero;
}
```

Of course, you'll have to change the settings as the jump occurs and then put them back once the jump completes.

Another thing to think about is that since the root is still on the ground, jumping onto objects becomes a bit trickier. The reason is that the root would actually go into the box you're jumping on. For example:

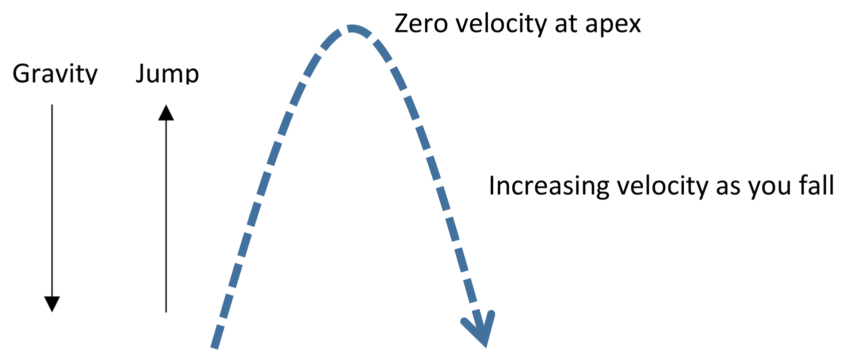


In all these cases, it would be up to your driver to compensate for the jump, root position, and capsule shape.

Double-Jumps

When implementing a double-jump, you'll want to clear the AC's "AccumulatedVelocity" property prior to calling AddImpulse() again. The reason is physics...

When using AddImpluse() for a jump, we're dealing with gravity. Just like in real-life, you have a decreasing velocity as you go up (due to gravity countering the initial jump velocity), 0 velocity at the apex, and increasing velocity as you go down (due to gravity).



I store that accumulation of velocity in the "AccumulatedVelocity" variable. So, as you fall and gravity increases your speed, that is captured here frame over frame.

Since doing AddImpulse() again for a double jump adds velocity, you can get different results depending on your current accumulated velocity. If you're on the falling side of the jump and a lot of gravitational velocity has accumulated, your double-jump may not be enough to cause you to move up again.

By clearing AccumulatedVelocity, you're removing the increasing impact gravity had over time and the accumulation starts over. So, you'll have a new fresh jump from your current position.

To clear AccumulatedVelocity, just do code like this:

```
lActorController.AccumulatedVelocity = Vector3.zero;
```

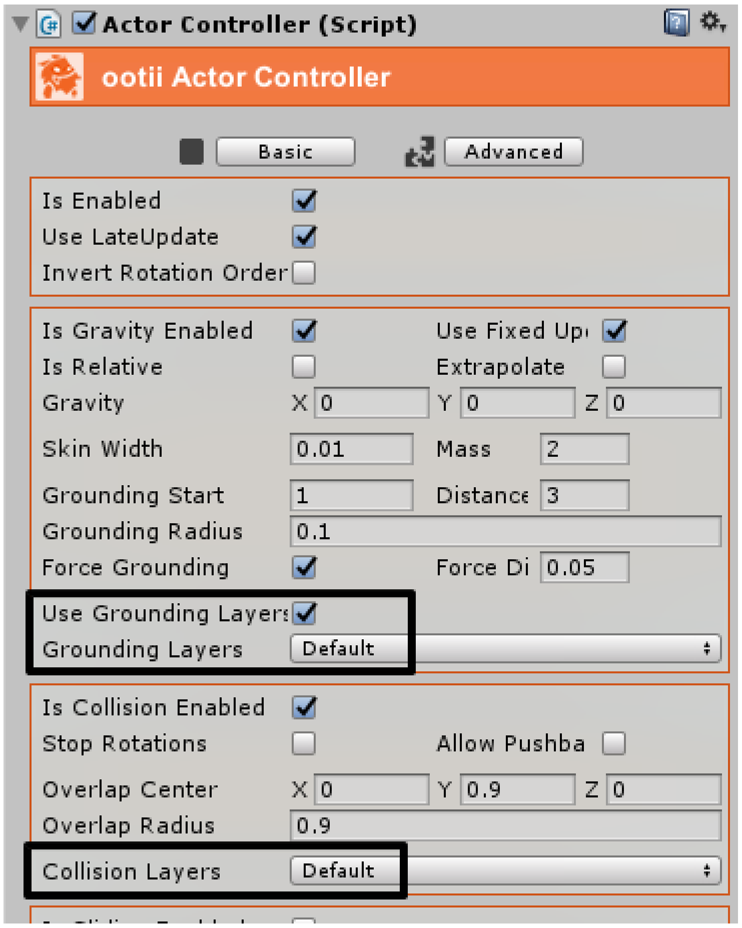
FAQ

How do I stop actor controllers from colliding with each other?

If you remember early on, I talk about the difference between collisions vs. grounding. Even if we make it so two actor's don't collide with each other, we could get some odd results as they go through each other. That's because one wants to be "grounded" on-top of the other.

For example: Say I have two actors; one on Layer #8 and one on Layer #9.

To prevent collisions, you want to enable "Use Grounding Layers" and ensure the "Ground Layers" doesn't include your other character layers. You also want to ensure "Collision Layers" doesn't include your other character layers as well.



Is Enabled ✓
Use LateUpdate ✓
Invert Rotation Order ☐

Is Gravity Enabled ✓ | Use Fixed Up ✓
Is Relative ☐ | Extrapolate ☐
Gravity X 0 Y 0 Z 0
Skin Width 0.01 | Mass 2
Grounding Start 1 | Distance 3
Grounding Radius 0.1
Force Grounding ✓ | Force Di 0.05
Use Grounding Layers ✓
Grounding Layers Default

Is Collision Enabled ✓
Stop Rotations ☐ | Allow Pushba ☐
Overlap Center X 0 Y 0.9 Z 0
Overlap Radius 0.9
Collision Layers Default

Since My "Grounding Layers" and "Collision Layers" for both actors doesn't include those layers... they don't collide.

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