

# **Ultimate Character Controller**

# Table of Contents

<b>Ultimate Character Controller</b> .....	2
<b>Getting Started</b> .....	3
<b>Version 2.2 Update Guide</b> .....	3
<b>Importing</b> .....	10
<b>Demo Scene</b> .....	12
<b>Setup</b> .....	13
<b>Project Update</b> .....	13
<b>Scene Setup</b> .....	14
<b>Character Creation</b> .....	15
<b>Item Type Creation</b> .....	18
<b>Item Creation</b> .....	20
<b>Organization</b> .....	28
<b>Component Overview</b> .....	29
<b>Editors</b> .....	31
<b>Setup</b> .....	31
<b>Character</b> .....	33
<b>Item Types</b> .....	36
<b>Item</b> .....	39
<b>Object</b> .....	42
<b>Integrations</b> .....	43
<b>State Configuration</b> .....	44
<b>Character</b> .....	45
<b>Movement Types</b> .....	53
<b>Included Movement Types</b> .....	54
<b>First Person Combat</b> .....	55
<b>First Person Free Look</b> .....	55
<b>Third Person Adventure</b> .....	55
<b>Third Person Combat</b> .....	56
<b>Third Person Four Legged</b> .....	56
<b>Third Person Pseudo3D (2.5D)</b> .....	56
<b>Third Person RPG</b> .....	58
<b>Third Person Top Down</b> .....	59
<b>Abilities</b> .....	60
<b>Included Abilities</b> .....	65
<b>Align To Ground</b> .....	65
<b>Align To Gravity Zone</b> .....	68
<b>Damage Visualization</b> .....	69
<b>Detect Ground Ability Base</b> .....	71
<b>Detect Object Ability Base</b> .....	72
<b>Drive</b> .....	73
<b>Interact</b> .....	75

<b>Pickup Item</b>	78
<b>Ride</b>	79
<b>Die</b>	80
<b>Fall</b>	82
<b>First Person Lean</b>	82
<b>Follow Pseduo3D (2.5D) Path</b>	83
<b>Generic</b>	84
<b>Height Change</b>	85
<b>Idle</b>	86
<b>Item Equip Verifier</b>	87
<b>Jump</b>	87
<b>Move Towards</b>	89
<b>Move With Object</b>	89
<b>NavMeshAgent Movement</b>	90
<b>Quick Start</b>	91
<b>Quick Stop</b>	91
<b>Quick Turn</b>	92
<b>Ragdoll</b>	93
<b>Restrict Position</b>	94
<b>Restrict Rotation</b>	94
<b>Revive</b>	95
<b>Rideable</b>	97
<b>Stop Movement Animation</b>	98
<b>Slide</b>	98
<b>Speed Change</b>	99
<b>Target Orbit</b>	101
<b>Item Abilities</b>	101
<b>Aim</b>	103
<b>Block</b>	104
<b>Drop</b>	105
<b>Item Set</b>	106
<b>Equip Next</b>	106
<b>Equip Previous</b>	106
<b>Equip Scroll</b>	106
<b>Equip Unequip</b>	107
<b>Toggle Equip</b>	109
<b>Reload</b>	109
<b>Third Person Item Pullback</b>	110
<b>Use</b>	111
<b>In Air Melee Use</b>	112
<b>Melee Counter Attack</b>	113
<b>Ability Starter</b>	114
<b>Ability Start Location</b>	115

<b>Animator Motion</b>	118
<b>New Ability</b>	119
<b>Effects</b>	127
<b>Included Effects</b>	129
<b>Boss Stomp</b>	129
<b>Play Audio Clip</b>	129
<b>Shake</b>	130
<b>Generic Character</b>	131
<b>Time</b>	132
<b>Minimum Component Setup</b>	133
<b>Camera</b>	134
<b>View Types</b>	138
<b>Included View Types</b>	142
<b>First Person</b>	142
<b>Combat</b>	147
<b>Free Look</b>	147
<b>First Person Transform Look</b>	148
<b>Third Person</b>	149
<b>Adventure</b>	150
<b>Combat</b>	151
<b>RPG</b>	151
<b>Third Person Look At</b>	151
<b>Third Person Pseudo3D (2.5D)</b>	152
<b>Third Person Top Down</b>	153
<b>Transition</b>	154
<b>Aim Assist</b>	155
<b>Lightweight Render Pipeline</b>	156
<b>Object Fader</b>	157
<b>Post Processing</b>	160
<b>Split Screen</b>	160
<b>Universal Render Pipeline</b>	162
<b>Items</b>	163
<b>Perspective</b>	167
<b>First Person</b>	168
<b>Third Person</b>	172
<b>Actions</b>	175
<b>Usable</b>	176
<b>Flashlight</b>	177
<b>Magic</b>	178
<b>Melee Weapon</b>	182
<b>Shootable Weapon</b>	189
<b>Throwable Item</b>	199
<b>Grenade Item</b>	201

<b>Shield</b>	202
<b>Dual Wielding</b>	203
<b>Runtime Pickup</b>	204
<b>Animator Audio State Set</b>	207
<b>Inventory</b>	209
<b>Item Types</b>	214
<b>Item Sets</b>	214
<b>Slots</b>	224
<b>Animation</b>	225
<b>Springs</b>	225
<b>Animator</b>	227
<b>Animator Controller</b>	227
<b>Animator Parameters</b>	230
<b>Default Animator Values</b>	232
<b>First Person Arms</b>	234
<b>Replacing Animations</b>	234
<b>Animation Event Trigger</b>	235
<b>Input</b>	236
<b>Virtual Controls</b>	239
<b>Attributes</b>	241
<b>Health</b>	243
<b>Inverse Kinematics (IK)</b>	248
<b>Objects</b>	255
<b>Explosions</b>	255
<b>Magic Particle</b>	256
<b>Object Pickup</b>	257
<b>Health Pickup</b>	257
<b>Item Pickup</b>	258
<b>Trajectory Object</b>	259
<b>Grenade</b>	263
<b>Magic Projectile</b>	264
<b>Projectile</b>	264
<b>Shell</b>	266
<b>Moving Platforms</b>	267
<b>Layer Manager</b>	270
<b>Surface System</b>	270
<b>Surface Manager</b>	272
<b>Surface Impacts</b>	274
<b>Surface Types</b>	275
<b>Surface Effects</b>	276
<b>Surface Identifiers</b>	278
<b>Decal Manager</b>	279
<b>Character Foot Effects</b>	279

<b>Advanced Surface System Topics</b>	280
<b>Spawn System</b>	282
<b>Respawner</b>	282
<b>Spawn Points</b>	284
<b>Audio</b>	286
<b>State System</b>	288
<b>Presets</b>	290
<b>Programming Concepts</b>	291
<b>Events</b>	291
<b>Scheduler</b>	292
<b>Object Pool</b>	293
<b>Artificial Intelligence (AI)</b>	294
<b>Multiplayer</b>	299
<b>Virtual Reality (VR)</b>	299
<b>Integrations</b>	299
<b>Adventure Creator</b>	299
<b>A* Pathfinding Project</b>	299
<b>Behavior Designer - Behavior Trees for Everyone</b>	300
<b>Cinemachine</b>	300
<b>Control Freak</b>	301
<b>DestroyIt</b>	302
<b>Dialogue System</b>	302
<b>Easy Touch</b>	302
<b>Final IK</b>	303
<b>InControl</b>	304
<b>Love/Hate</b>	305
<b>Playmaker</b>	305
<b>Quest Machine</b>	305
<b>Rewired</b>	306
<b>UMA</b>	306
<b>Videos</b>	310

# Ultimate Character Controller

Thank you for your purchase of the most powerful and flexible character controller on the Asset Store. While the title says Ultimate Character Controller, this documentation is applicable towards any of our character controllers, including the Ultimate Character Controller, UFPS, Third Person Controller, First Person Controller, etc. All of these controllers share the same code base and follow the same structure.

The Ultimate Character Controller is the result of two previously independent character controllers on the Asset Store: UFPS and the Third Person Controller. We acquired UFPS from VisionPunk in January 2016 and would like to thank VisionPunk for giving us the opportunity to expand on UFPS.

When we initially started on version 2.0 we had a choice: patch UFPS and Third Person Controller so they can work together or start from an empty project and take the best aspects of UFPS and the Third Person Controller. While we knew that the latter would take more time we thought the end result would be worth it.

We hope that you enjoy our character controller and would love to see what you create with it. Please join us on the [forums](#) or [discord](#) and post your project on [opsive.com/showcase](http://opsive.com/showcase) – we'd love to see what you're working on!

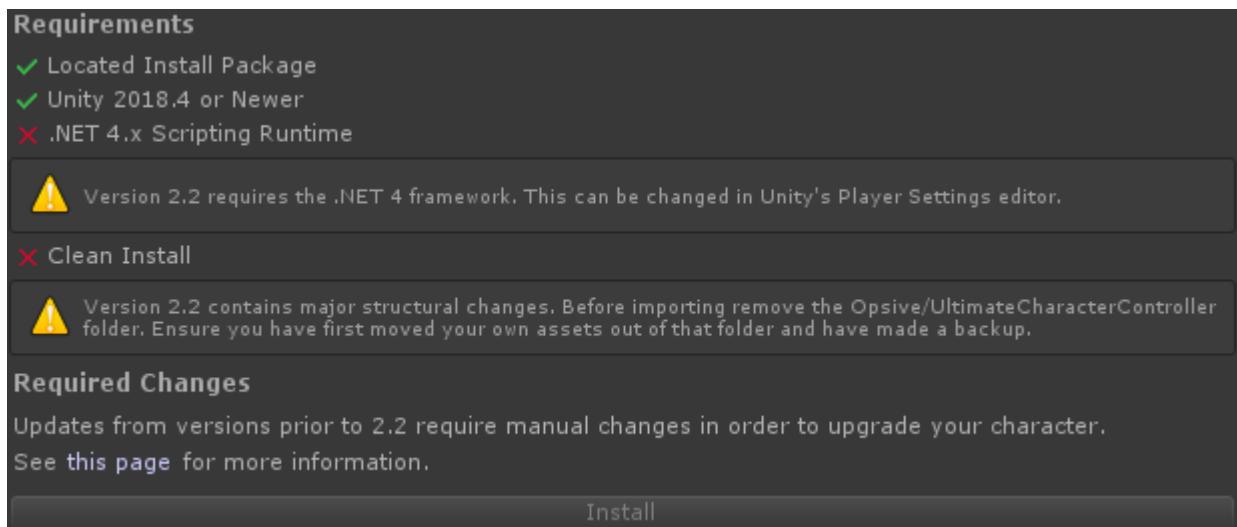
# Getting Started

## Version 2.2 Update Guide

The inventory classes within version 2.2 has been refactored in order to support our upcoming inventory system. While the inventory system is a standalone asset that does not require the character controller, the character controller is sharing a few classes with the inventory system in order to make a seamless integration. As a result of these changes, there are few steps required in order to update to version 2.2 from a prior version.

### Import

After importing version 2.2 into your project a dialogue will appear which checks for the basic requirements:



The following checks are performed:

- *Located Install Package*: When version 2.2 is imported both the installer and a separate package is imported. This separate package contains the character controller assets.
- *Unity 2018.4 or Newer*: Unity 2018.4 is now the minimum version. If you are running a prior version you'll need to first [install 2018.4 or newer](#).
- *.NET 4.x Scripting Runtime*: The scripting runtime has been updated and now uses .NET 4. Version 3 of the framework has been deprecated and can be changed within the [Unity Player Settings](#).



- *Clean Install*: Version 2.2 moves/removes many files and Unity Packages do not deal well with these type of changes. As a result the entire Opsive/UltimateCharacterController directory should be removed before version 2.2 is installed. **Ensure you have first moved your own assets out of that folder and have a backup.**

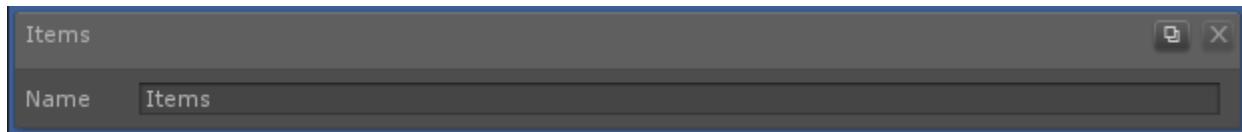
Assuming all of these requirements are satisfied you can click *Install* to import version 2.2.

## 2.2 Setup

After version 2.2 has been installed the following changes are required:

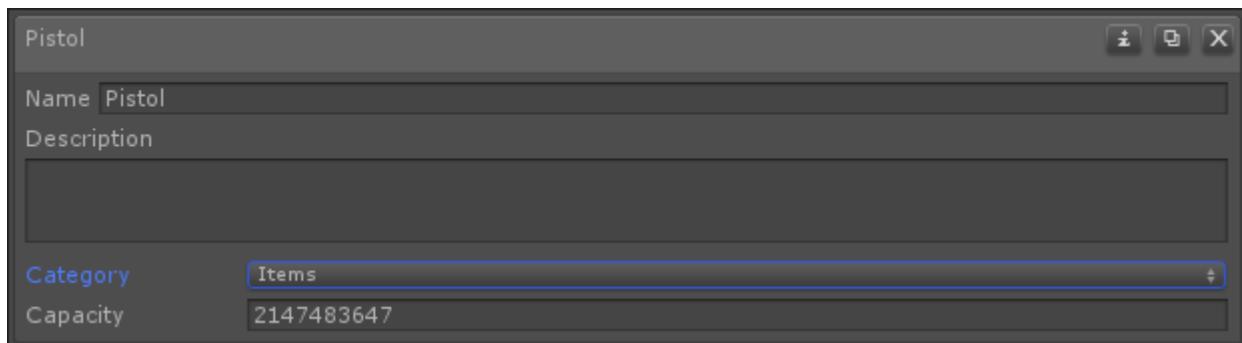
### Inventory Category

The class for the inventory categories has changed for version 2.2. Any created categories should be created within the [Item Type Manager](#).



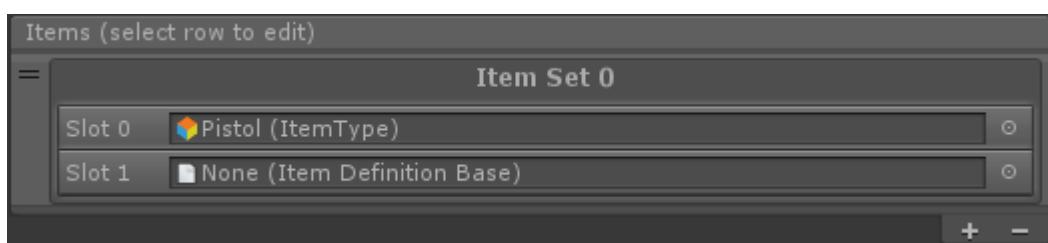
### Item Type Category Assignment

After the category has been created it should be reassigned to the Item Type.



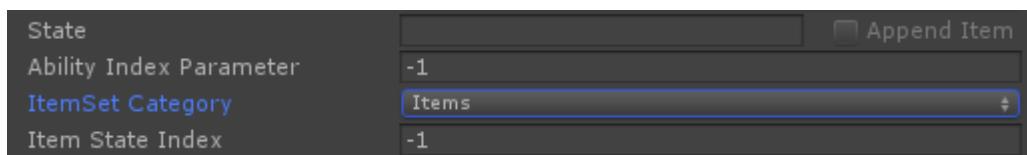
### Item Set Manager

Assign the created categories to the [Item Set Manager](#).



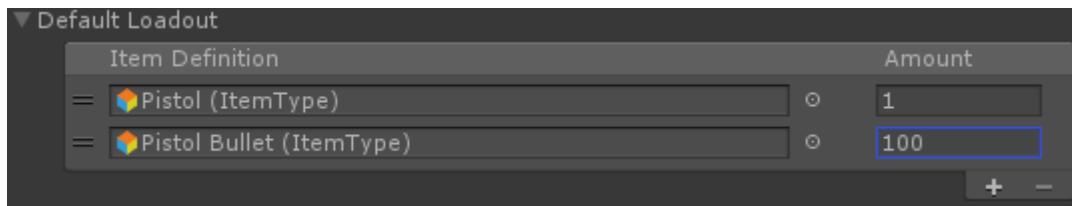
### Item Ability Category Assignment

The new categories should be assigned to the *ItemSet Category* field of the [Item Set Item Abilities](#). This will need to be done for each Item Ability: Equip Unequip, Toggle Equip, Equip Next, Equip Previous, and Equip Scroll.



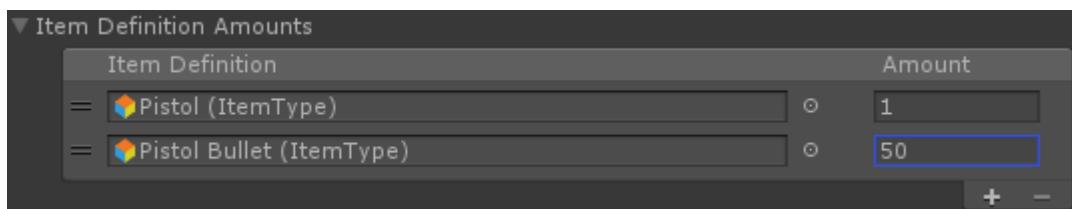
## Default Loadout

The Item Type Count class has been renamed to Item Definition Amount. As a result the *Amount* within the Inventory's *Default Loadout* must be updated.



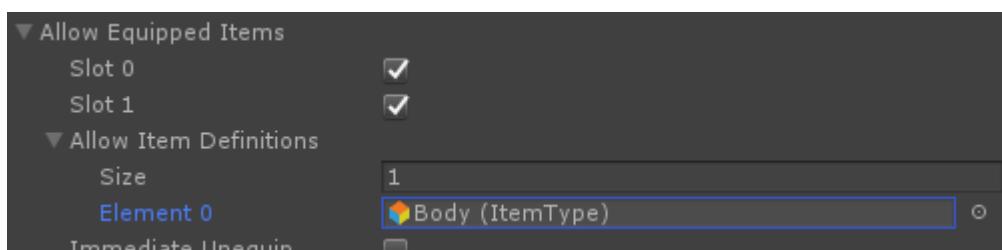
## Item Pickup

Similar to the *Default Loadout*, the *Amount* field within the *Item Definition Amounts* on the Item Pickup field must be updated.



## Ability Allow Item Definitions

The *Allow Item Definitions* under the Ability's General tab must be updated with the Item Types.



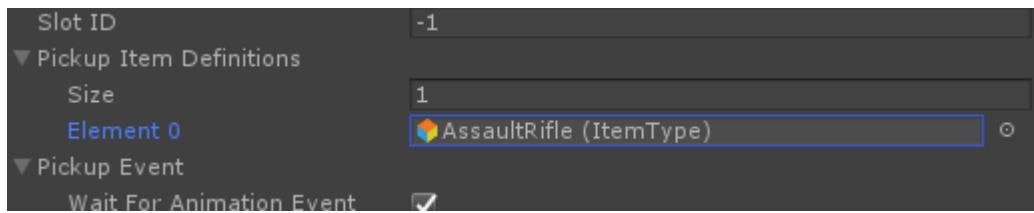
## Append Item

The *Append Item* field needs to be updated. This field appends the Item Definition's name to the state name and is most commonly used for the Aim ability.



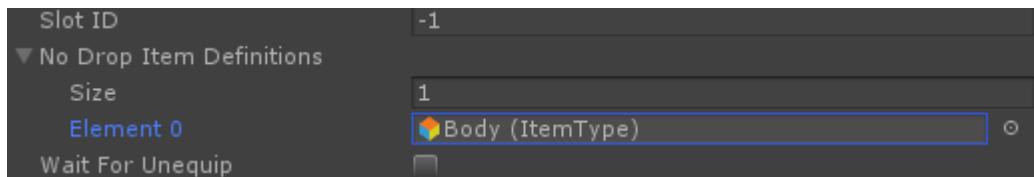
## Pickup Item

If you're using the Pickup Item ability the *Pickup Item Definitions* need to be updated.



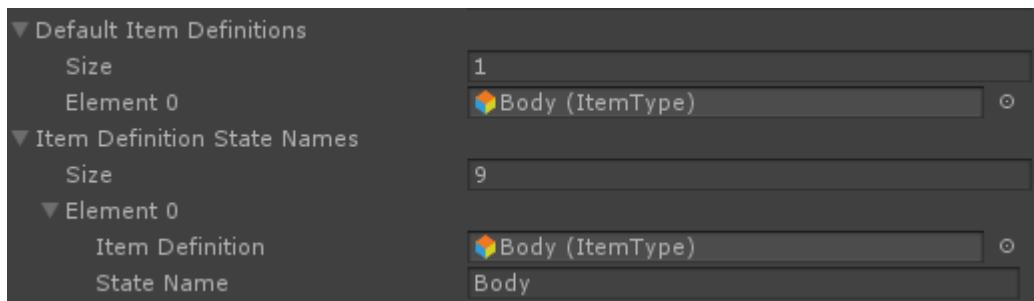
## Drop Item

If you're using the Drop Item ability the *No Drop Item Definitions* need to be updated.



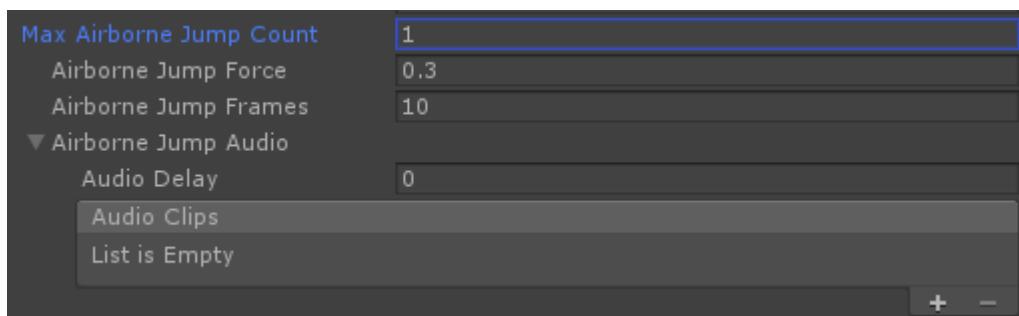
## Slot Equip

If you're using the Slot Equip ability with the VR Add-On the *Default Item Definitions* and *Item Definition State Names* arrays need to be updated.



## Repeated Jump

If you're using a Repeated Jump for the Jump ability the field has been renamed to *Airborne Jump*. These values will need to be updated with the new name.



## Namespaces

Many common classes are shared between the character controller and inventory system. These files are in a new "Shared" namespace. The follow commonly used classes were changed:

### Opsive.Shared.Events

- Event Handler

## **Opsive.Shared.Game**

- Game Object Extensions (responsible for GetCachedComponent)
- Object Pool
- Scheduler

## **Opsive.Shared.Inventory**

- IItemIdentifier (formally Item Type)
- ItemDefinitionBase

## **Opsive.Shared.Utility**

- Generic Object Pool (used to be within Object Pool)
- Type Utility (for GetType)

## **API**

The API underwent changes during the version 2.2 update. Most of the changes were for the new inventory system structure.

### **Inventory**

ItemTypeCount has been renamed to ItemDefinitionAmount, with Amount being an int instead of a float.

AddItem has a new parameter:

```
/// <summary>
/// Adds the item to the inventory. This does not add the actual
ItemIdentifier - PickupItem does that.
/// </summary>
/// <param name="item">The Item to add.</param>
/// <param name="immediateEquip">Can the item be equipped
immediately?</param>
/// <param name="forcePickup">Should the item be force
equipped?</param>
void AddItem(Item item, bool immediateEquip, bool forceEquip)
```

PickupItemType has been renamed to Pickup with updated parameters:

```
/// <summary>
/// Adds the specified amount of the ItemIdentifier to the inventory.
/// </summary>
/// <param name="itemIdentifier">The ItemIdentifier to add.</param>
/// <param name="amount">The amount of ItemIdentifier to add.</param>
/// <param name="slotID">The slot ID that picked up the item. A -1
value will indicate no specified slot.</param>
/// <param name="immediatePickup">Should the item be picked up
immediately?</param>
/// <param name="forceEquip">Should the item be force
```

```
equipped?</param>
/// <returns>True if the ItemIdentifier was picked up.</returns>
bool Pickup(IItemIdentifier itemIdentifier, int amount, int slotID,
bool immediatePickup, bool forceEquip)
```

GetItem has been renamed to GetActiveItem:

```
/// <summary>
/// Returns the active item in the specified slot.
/// </summary>
/// <param name="slotID">The ID of the slot.</param>
/// <returns>The active item which occupies the specified slot. Can be
null.</returns>
Item GetActiveItem(int slotID)
```

GetItem now uses an IItemIdentifier instead of ItemType:

```
/// <summary>
/// Returns the item that corresponds to the specified ItemIdentifier.
/// </summary>
/// <param name="slotID">The ID of the slot which the item belongs
to.</param>
/// <param name="itemIdentifier">The ItemIdentifier of the
item.</param>
/// <returns>The item which occupies the specified slot. Can be
null.</returns>
Item GetItem(int slotID, IItemIdentifier itemIdentifier)
```

EquipItem has an updated parameter:

```
/// <summary>
/// Equips the ItemIdentifier in the specified slot.
/// </summary>
/// <param name="itemIdentifier">The ItemIdentifier to equip.</param>
/// <param name="slotID">The ID of the slot.</param>
/// <param name="immediateEquip">Is the item being equipped
immediately? Immediate equips will occur from the default loadout or
quickly switching to the item.</param>
void EquipItem(IItemIdentifier itemIdentifier, int slotID, bool
immediateEquip)
```

UnequipItem has an updated parameter:

```
/// <summary>
/// Unequips the specified ItemIdentifier in the specified slot.
/// </summary>
/// <param name="itemIdentifier">The ItemIdentifier to unequip. If the
ItemIdentifier isn't currently equipped then no changes will be
made.</param>
/// <param name="slotID">The ID of the slot.</param>
```

```
void UnequipItem(IItemIdentifier itemIdentifier, int slotID)
```

GetItemTypeCount has been renamed to GetItemIdentifierAmount with an updated parameter:

```
/// <summary>
/// Returns the amount of the specified ItemIdentifier.
/// </summary>
/// <param name="itemIdentifier">The ItemIdentifier to get the amount of.</param>
/// <returns>The amount of the specified ItemIdentifier.</returns>
int GetItemIdentifierAmount(IItemIdentifier itemIdentifier)
```

UseItem has been renamed to AdjustItemIdentifierAmount with updated parameters:

```
/// <summary>
/// Adjusts the amount of the specified ItemIdentifier.
/// </summary>
/// <param name="itemIdentifier">The ItemIdentifier to adjust.</param>
/// <param name="amount">The amount of ItemIdentifier to adjust.</param>
void AdjustItemIdentifierAmount(IItemIdentifier itemIdentifier, int amount)
```

RemoveItem has an updated parameter:

```
/// <summary>
/// Removes the ItemIdentifier from the inventory.
/// </summary>
/// <param name="itemIdentifier">The ItemIdentifier to remove.</param>
/// <param name="slotID">The ID of the slot.</param>
/// <param name="drop">Should the item be dropped when removed?</param>
void RemoveItem(IItemIdentifier itemIdentifier, int slotID, bool drop)
```

## Ultimate Character Locomotion

AddSubCollider(s) has been renamed to AddIgnoredCollider(s):

```
/// <summary>
/// Adds an element to the ignored collider array.
/// </summary>
/// <param name="collider">The collider which should be added to the array.</param>
void AddIgnoredCollider(Collider collider)

/// <summary>
/// Adds an array to the ignored collider array.
/// </summary>
/// <param name="colliders">The colliders which should be added to the
```

```
array.</param>
void AddIgnoredColliders(Collider[] colliders)
```

RemoveSubCollider(s) has been renamed to RemoveIgnoredCollider(s):

```
/// <summary>
/// Removes the specified collider from the ignored collider array.
/// </summary>
/// <param name="collider">The collider which should be removed from
the array.</param>
void RemoveIgnoredCollider(Collider collider)

/// <summary>
/// Removes the specified colliders from the ignored collider array.
/// </summary>
/// <param name="colliders">The colliders that should be
removed.</param>
void RemoveIgnoredColliders(Collider[] colliders)
```

#### Generic Object Pool

ObjectPool.Get and ObjectPool.Return have been moved to GenericObjectPool.Get and GenericObjectPool.Return. These are located within the Opsive.Shared.Utility namespace.

## Importing

### Import Errors

If you receive any errors upon import it is likely because of a [namespace](#) conflict. Lets say that you have an existing project and in that project you have a class named Health:

```
using UnityEngine;

public class Health : MonoBehaviour
{
    /// <summary>
    /// Damages the object.
    /// </summary>
    public void Damage()
    {
        // My implementation.
    }
}
```

If you have this class in your project and import the Ultimate Character Controller you'll receive errors similar to:

```
: error CS1501: No overload for method `Damage' takes `4' arguments
```

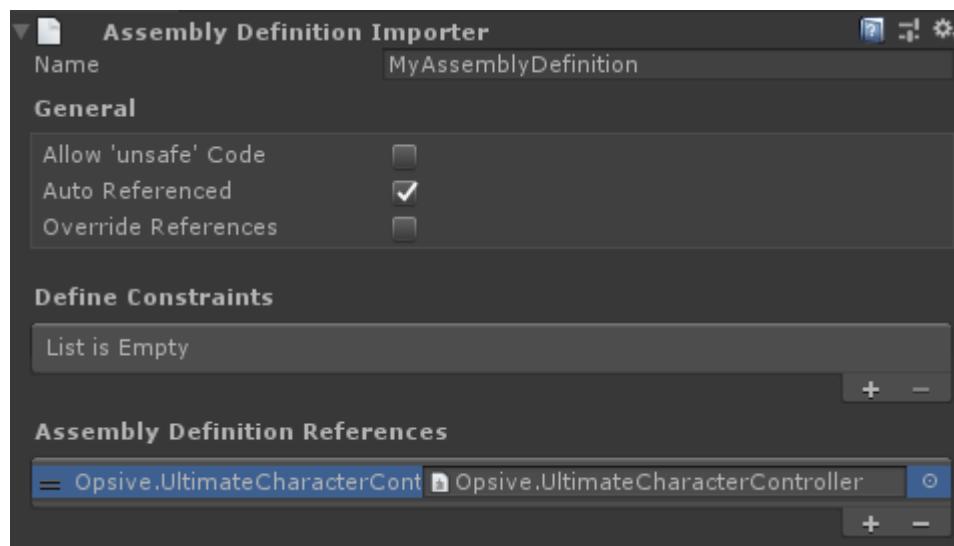
The reason this error exists is because the original Health component in your project is corrupting the global namespace and the compiler which doesn't know which Health component to use. The fix for this is to ensure all classes are in a namespace. You can fix this error by adding my original Health component to its own namespace:

```
using UnityEngine;

namespace MyProject
{
    public class Health : MonoBehaviour
    {
        /// <summary>
        /// Damages the object.
        /// </summary>
        public void Damage()
        {
            // My implementation.
        }
    }
}
```

## Assembly Definitions

The Ultimate Character Controller uses its own [Assembly Definition](#) files to decrease overall project compilation time. If you'd like to access an Ultimate Character Controller object through scripting ensure you have first added the Opsive.UltimateCharacterController Assembly Definition to your own Assembly Definition file.



## Demo Scene

The Ultimate Character Controller Demo scene will not work if you only have the First Person Controller or the Third Person Controller imported. You should instead use the First/Third Person Controller Demo scene. The Ultimate Character Controller Demo scene

requires both the First Person Controller and the Third Person Controller to be imported.

## Lightmapping

In order to reduce the download size the demo scene does not contain any [lightmapping](#). When the scene first opens it will appear dark and this can be corrected by [baking the scene](#). If the lights are not baked before hitting play they will automatically be disabled.

## Import Time

The Ultimate Character Controller contains many animations/textures which increase the amount of time that it takes to import the asset. If you don't care about the demo scene content and want an extremely fast import time you can deselect the Opsive/UltimateCharacterController/Demo folder and it will not bring in any assets used within the demo scene. Note that this will not bring in any animator controllers or animations so if you want these make sure you import the Demo/Animations and Demo/Animator folders.

## Animation Import Warnings

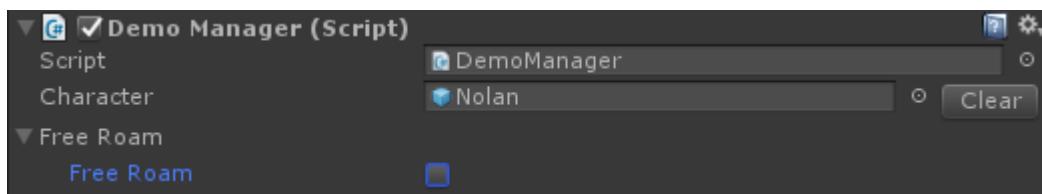
When you import the Ultimate Character Controller into a fresh project you should receive no errors but there will be some warnings related to the animations, such as:

File 'AimWalkFwd' has animation import warnings. See Import Messages in Animation Import Settings for more details.

The reason this warning exists is because the animations were authored in Blender and Unity doesn't like the way Blender exported the animation. Unfortunately there's not a way around this but this is a one time warning and does not affect the animations when they are playing.

## Demo Scene

The included demo scene provides a great base for testing out the features included within the controller. When you start the demo scene the character will be spawned in the basement hallway. If you'd like to prevent the character from spawning in this area you can deselect *Free Roam* on the Demo Manager:



### Ultimate Character Controller Demo Scene

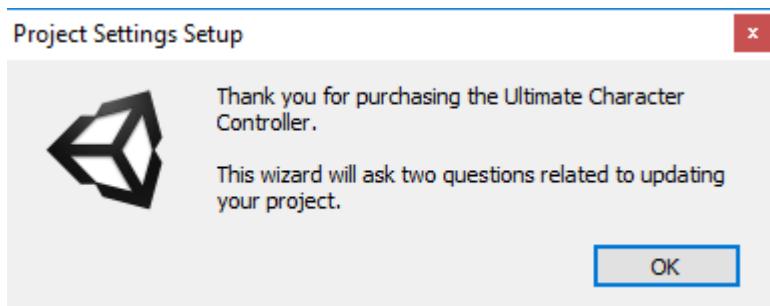
If you've imported the First Person Controller or the Third Person Controller you'll notice that there are two demo scenes included. The Ultimate Character Controller demo scene

requires both the First Person Controller and the Third Person Controller in order for it to work. If you do not have both assets imported you should use the single perspective demo scene.

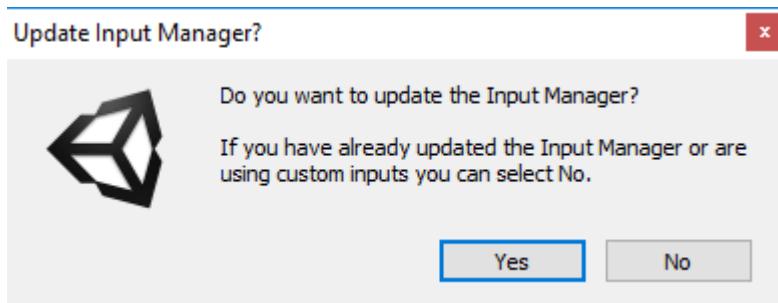
## Setup

# Project Update

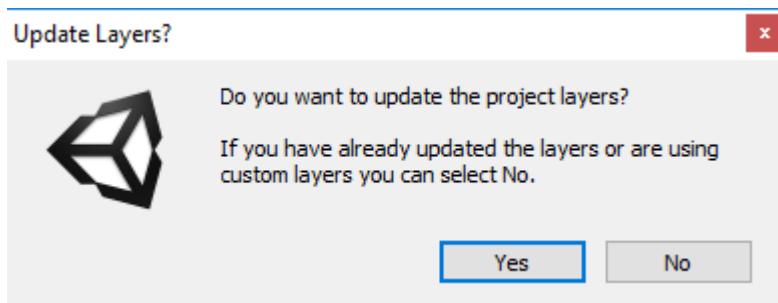
The Ultimate Character Controller is an Editor Extension so when you import from the Asset Store it does not bring the project settings with it. After the Ultimate Character Controller is done importing the following dialog will appear:



The next dialogue relates to updating the input button mappings. By default, the Ultimate Character Controller uses Unity's input system which relies on the [Input Manager](#). The Input Manager requires the button mappings (Jump, Crouch, Fire, etc.) to be created ahead of time so this dialog will add the required button mappings. It will not override a button mapping that already exists.

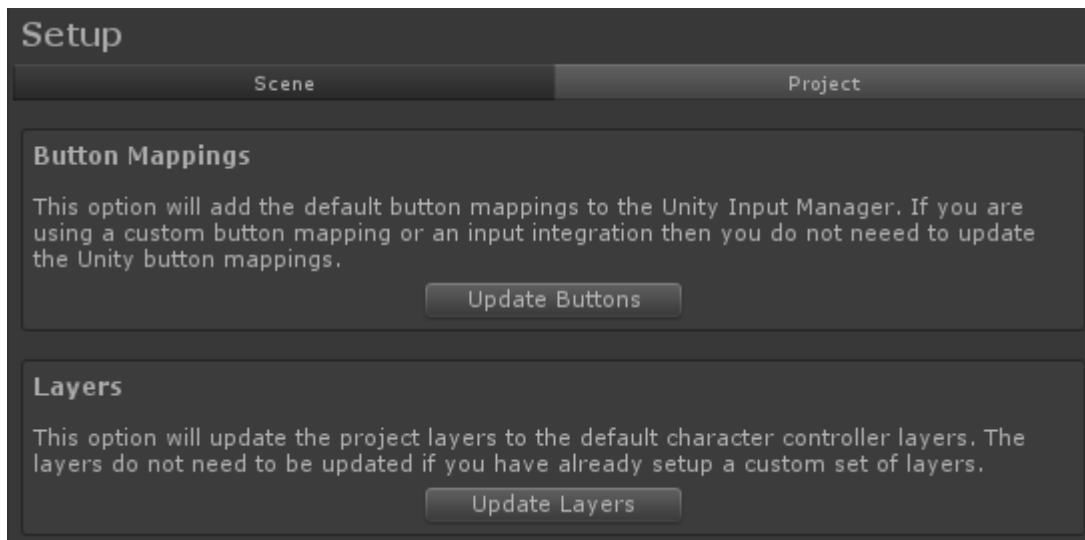


The next (and last) dialog that will appear relates to updating Unity's [layer settings](#). This is recommended so the correct default layer names appear. These layers can later be remapped within the character's [Layer Manager](#).



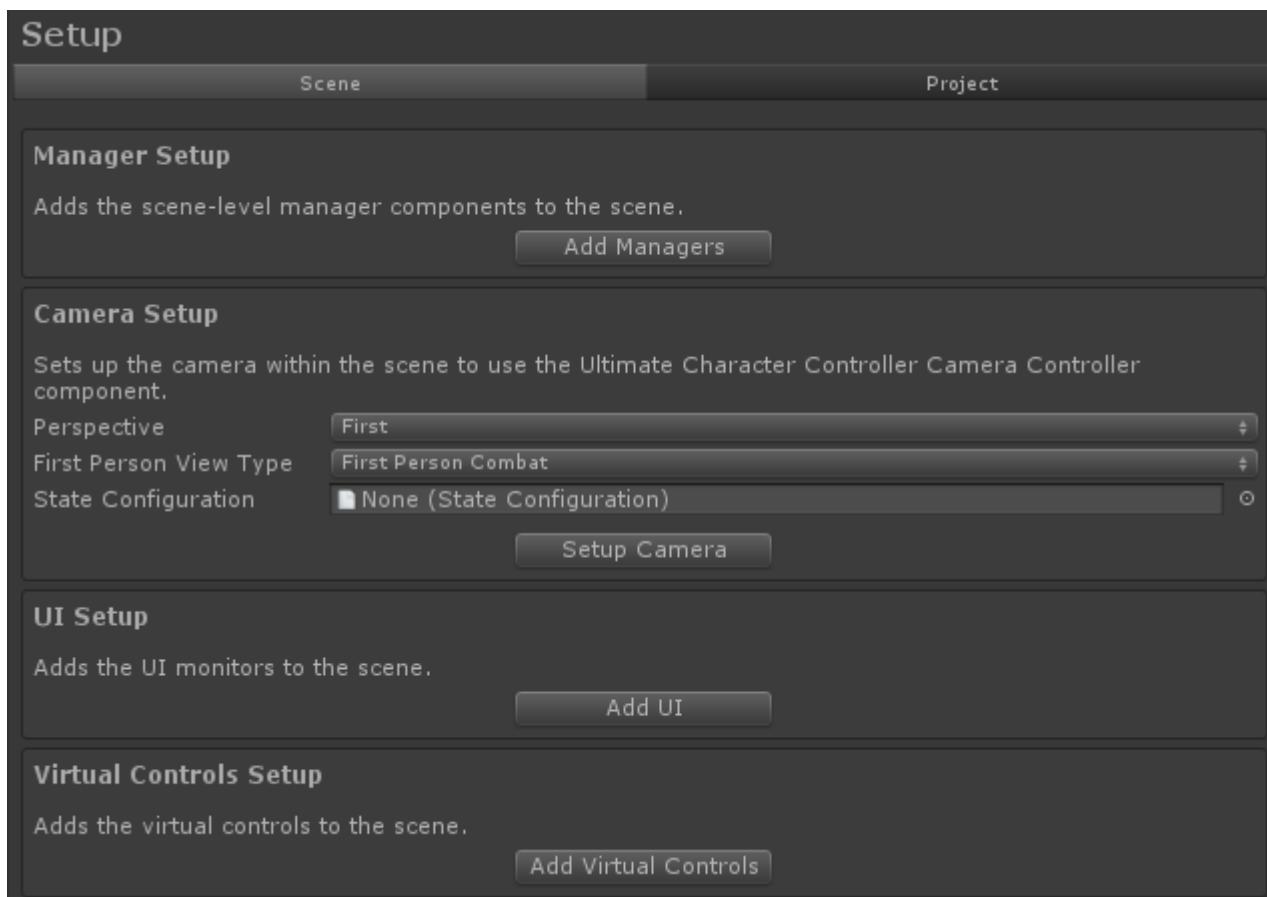
If you would like to perform the button or layer update later on you can do so by selecting the Tools -> Opsive -> Ultimate Character Controller -> Main Manager menu option, then

selecting the Setup button followed by the Project tab.



## Scene Setup

Before you create your character the scene needs to first be setup to work with the Ultimate Character Controller. Setting up the scene involves adding the scene-level manager components and adding the Camera Controller. The scene-level manager components include objects such as the Object Pool and Surface Manager which are a singleton component necessary for the Ultimate Character Controller to function. These objects can easily be created within the Setup Manager (Tools -> Ultimate Character Controller -> Main Manager, then select Setup):



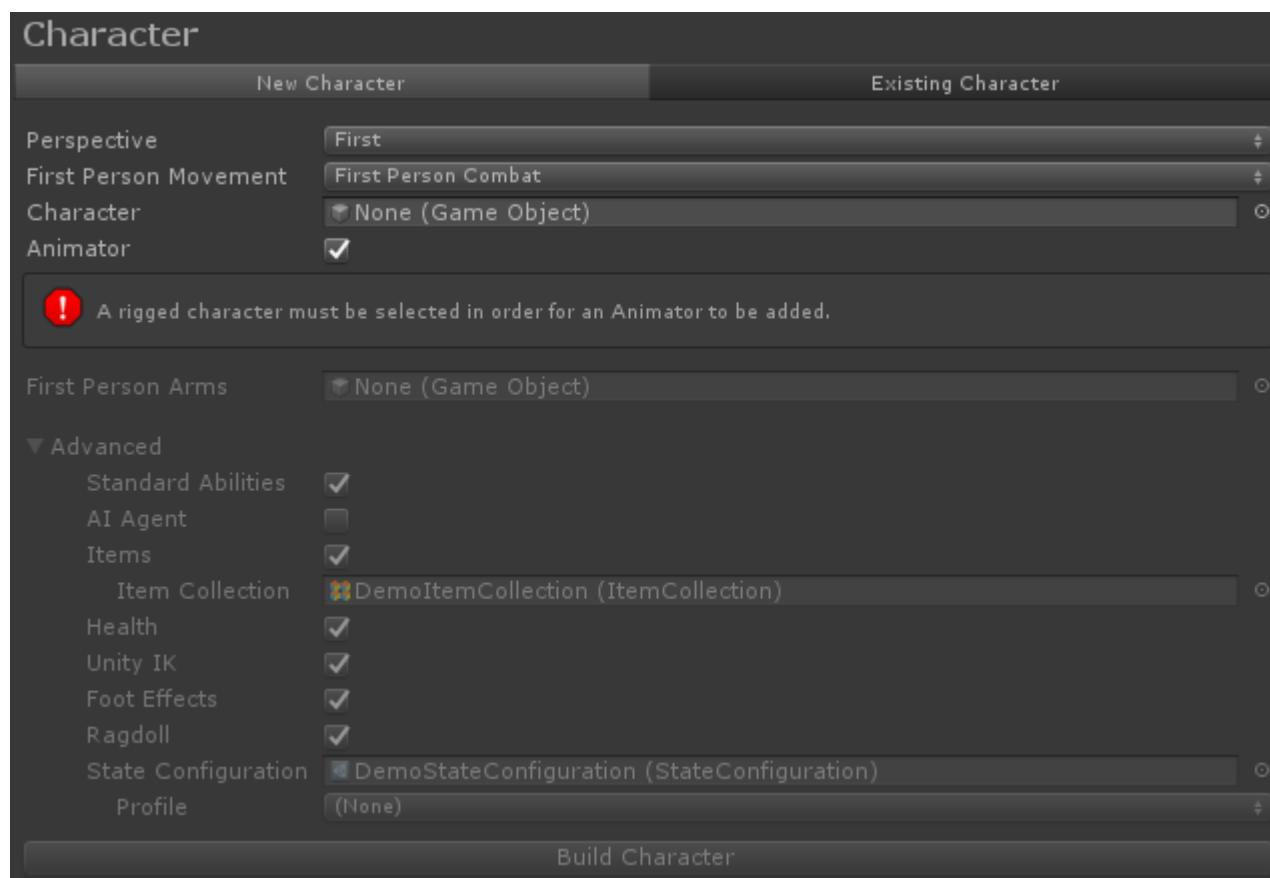
To get started click the "Add Managers" button. This will add the "Game" GameObject to

the scene with all of the necessary scene-level components. The Camera Controller can then be setup by clicking the “Setup Camera” button. Make sure you first select the perspective and view type that you’d like to use.

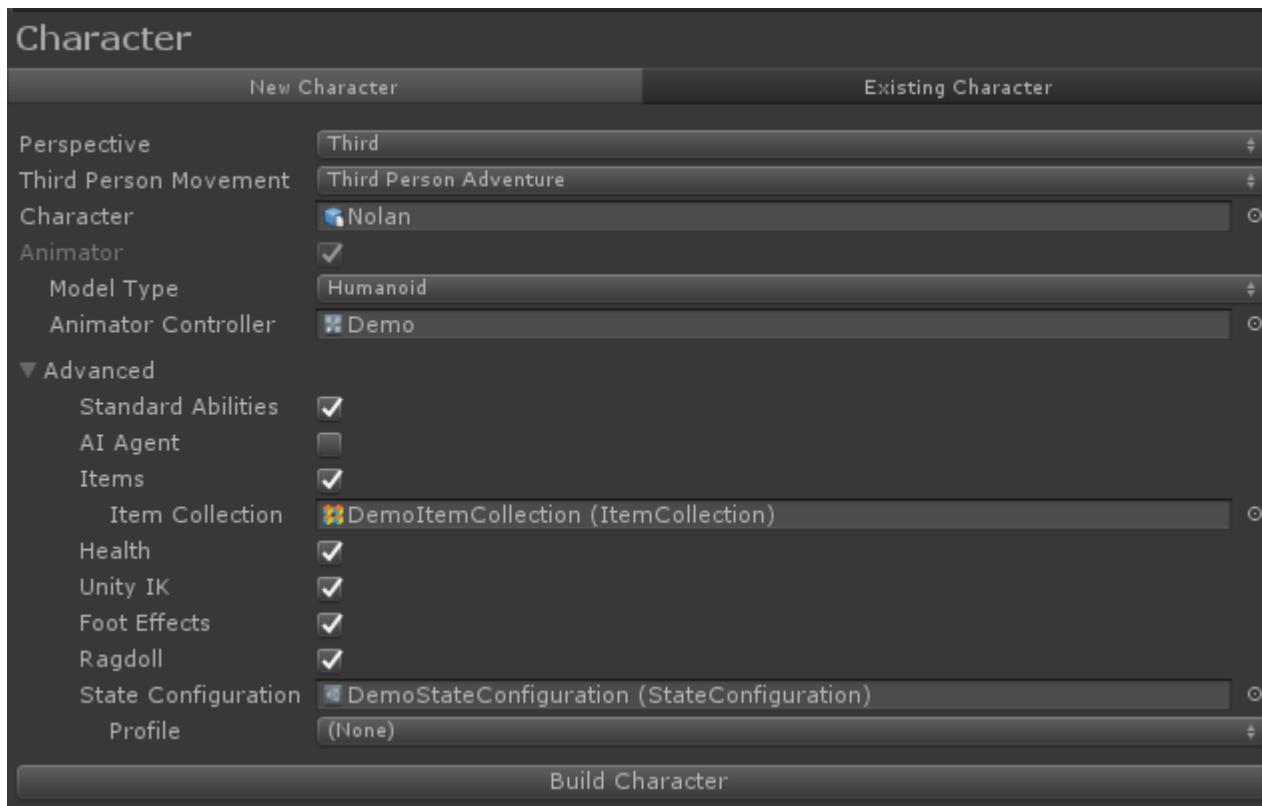
The Setup Manager can also setup the default UI and virtual controls but this is not required to get started. If you do add the UI or virtual controls you can later customize it to fit the style of your game.

## Character Creation

The character can be created using the Character Manager from the Tools -> Ultimate Character Controller -> Character Manager menu option. This will open an editor window that acts as the central hub for any object creation/updating related editor scripts.



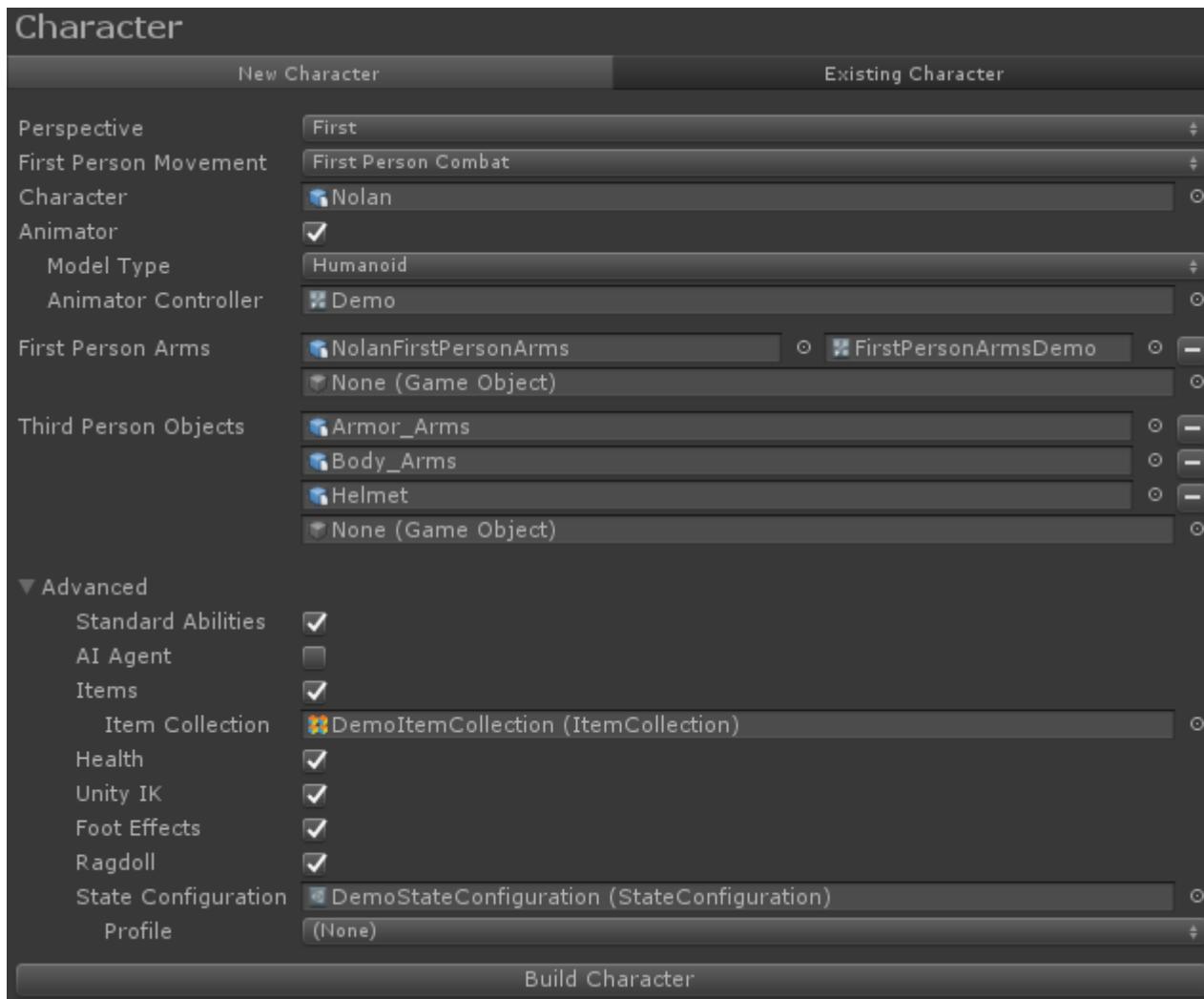
At the moment most of the options are disabled because a character hasn't been selected. Drag the Nolan model (in the Opsive/Ultimate Character Controller/Demo/Models folder) into your scene and assign it to the *Character* field.



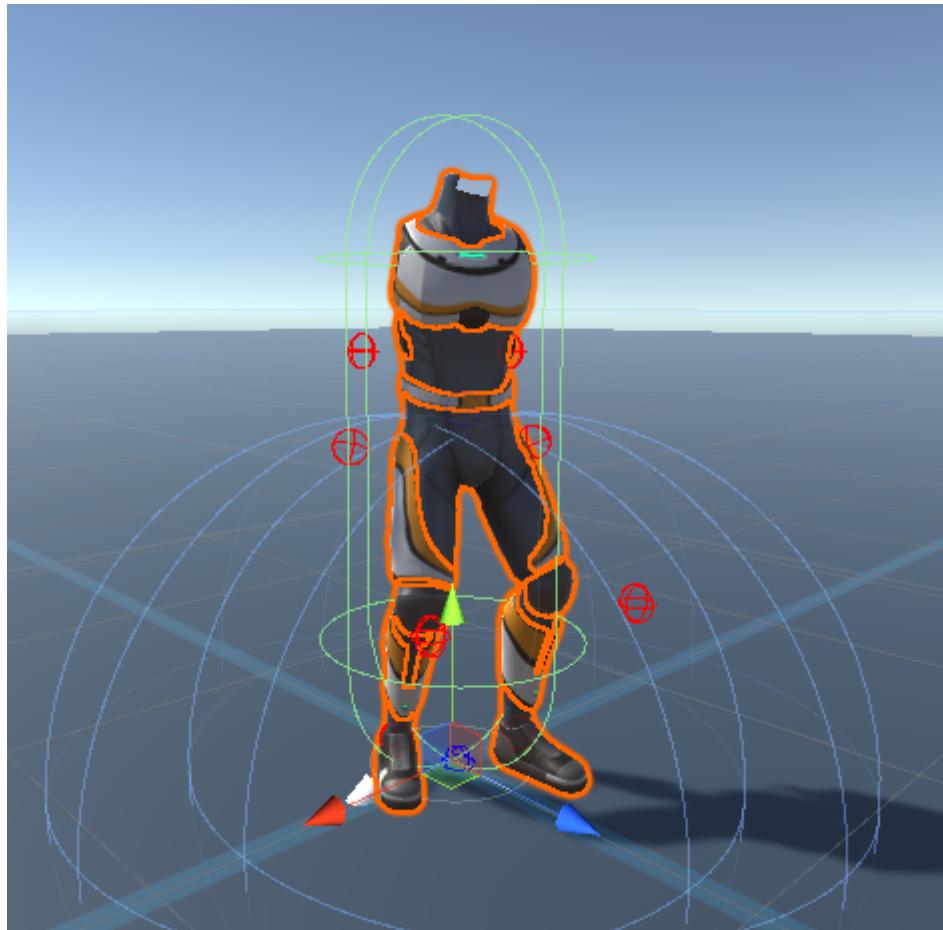
Once the character has been selected you can press the Build Character button and all of the required components and setup will be added.

If you are creating a first person or both character there are two more steps required:

- When the first person camera renders it will render any objects in the *First Person Arms* field last. This will allow the arms to always be in front of the objects and will [prevent clipping](#). This field can be left blank and instead added when creating the [item](#).
- Unless the character model was specifically designed for a first person view, the model is going to have objects that should be hidden when the first person view is active such as the character's head and arms. Any object that should be hidden when the first person view is active should be specified under the *Third Person Objects* field. This field will automatically resize when a new object has been added. If your character model does not have the head/arms separated and you don't have access to the original source file you can separate the objects with a tool such as the [FPS Mesh Tool](#).



If the Build Character button is pressed all of the components will be added to the character and it can be played within the game. Here's a look at the character in first person view with the objects hidden specified by the *Third Person Objects* field.



More details on each Character Builder option can be found on [this page](#).

## Item Type Creation

Item Types are used to identify a particular item that the character can carry. Item Types can also represent the ammo in an assault rifle or the number of grenades that are remaining. Item Types are part of an Item Collection which is a grouping of Item Types. Item Types can be managed within the Item Type Manager:

## Item Types

The screenshot shows the 'Item Types' manager window. At the top, there are tabs for 'Item Types' and 'Categories'. Below the tabs, there's a search bar with a magnifying glass icon and a clear button. A 'Create' button is located at the top right. The main area is titled 'Item Types' and contains a list of items with their names and edit/delete buttons.

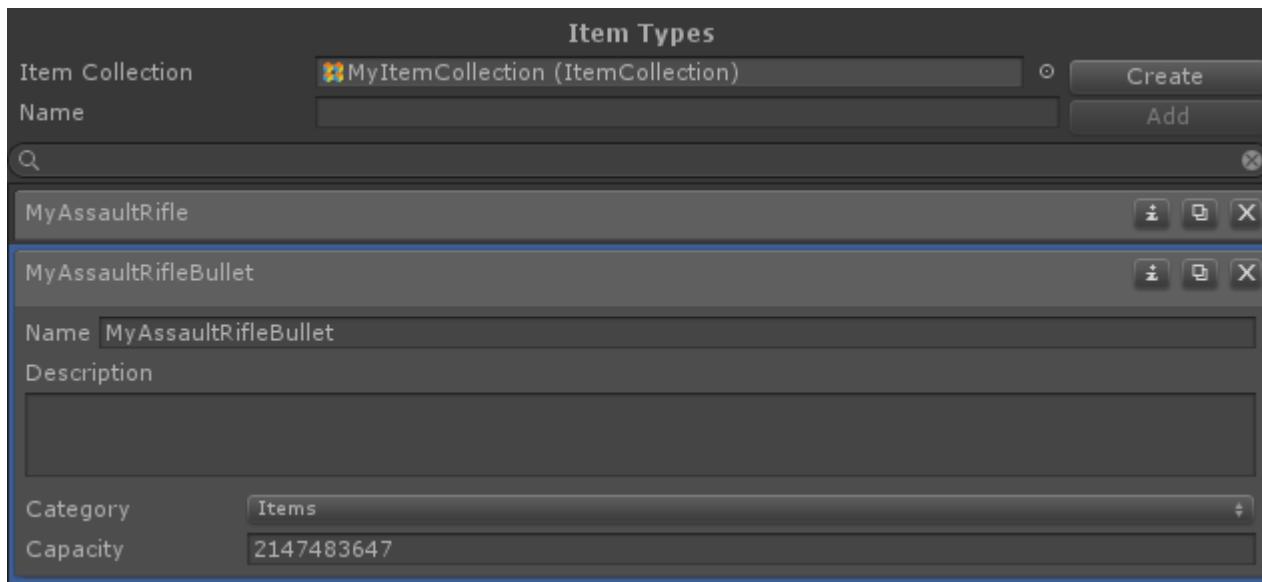
Name	Action Buttons
AssaultRifle	edit, delete
AssaultRifleBullet	edit, delete
Sword	edit, delete
Pistol	edit, delete
PistolBullet	edit, delete
FragGrenade	edit, delete
ThrownFragGrenade	edit, delete
Bow	edit, delete
BowArrow	edit, delete
Body	edit, delete
Shotgun	edit, delete
ShotgunBullet	edit, delete
RocketLauncher	edit, delete
Rocket	edit, delete

Let's create a new Item Type that will be used by the weapon that we will create in the next section. The first step that should be performed is that the "Create" button should be selected to create a new Item Collection. An Item Collection stores the ItemTypes and by creating a new Item Collection you'll ensure that any Item Type customizations don't get overwritten by any updates. Next, under the *Name* field at the top of the Item Type manager type in "MyAssaultRifle" and press "Add". A new ItemType of MyAssaultRifle will appear in the list. The category of the new ItemType should be set to "Items". This category is used by the [Item Set Manager](#).

The screenshot shows the 'Item Types' manager window with a new item type being created. The 'Create' button is highlighted. The 'Name' field contains 'MyAssaultRifle'. The 'Category' dropdown is set to 'Items'. The 'Capacity' field is set to '1'. The 'Description' field is empty.

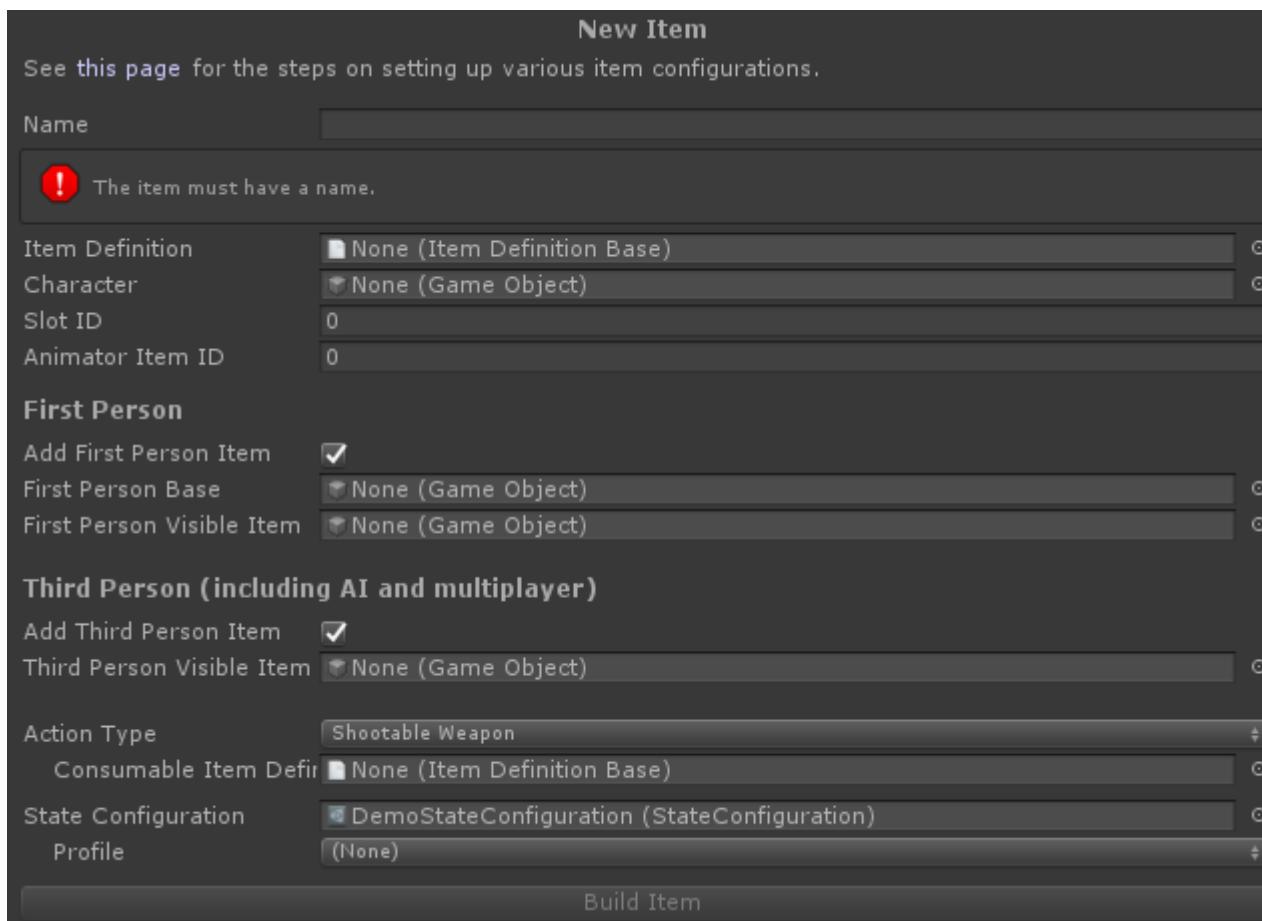
Name	Description	Category	Capacity
MyAssaultRifle		Items	1

The assault rifle will need to fire bullets so create a second Item Type called "MyAssaultRifleBullet":



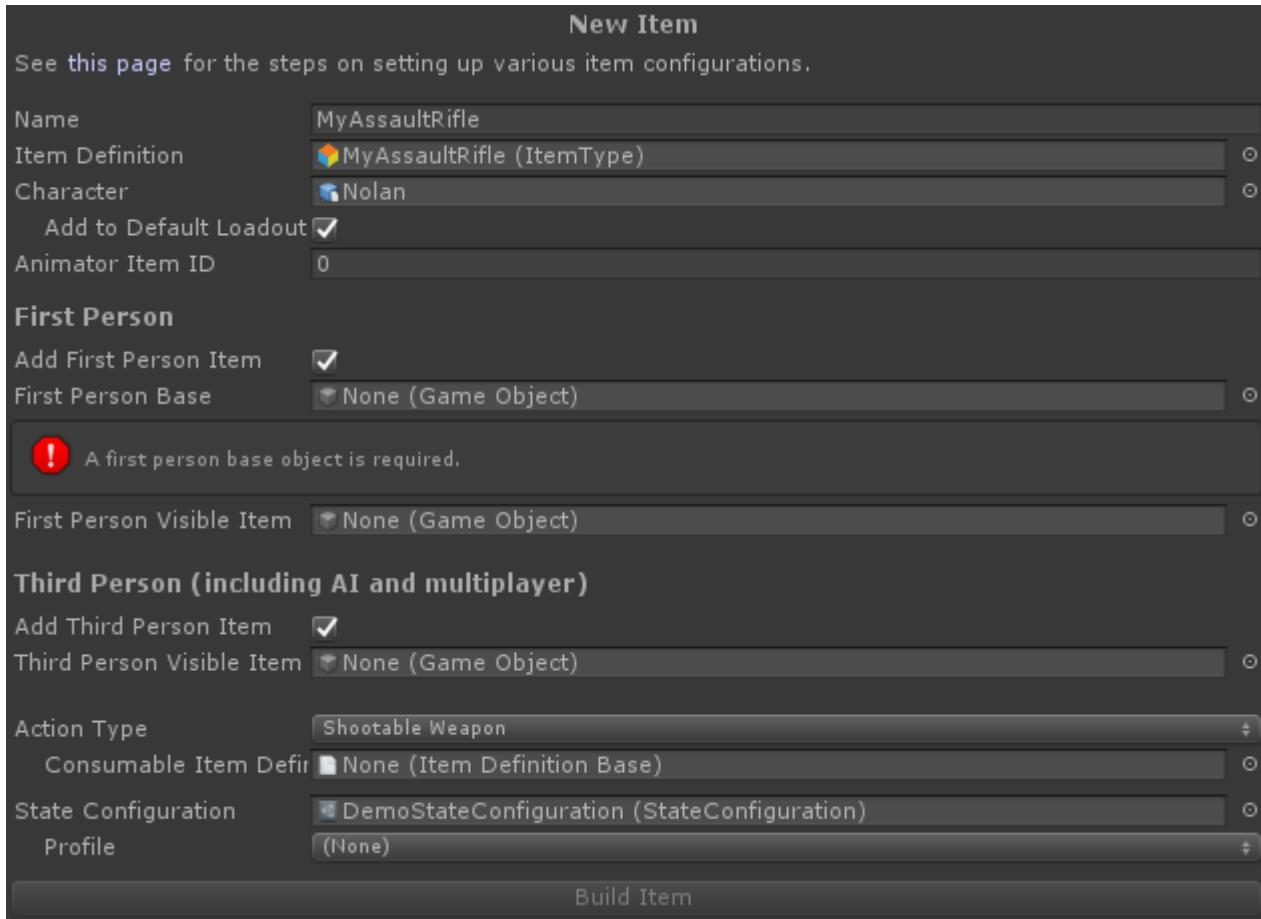
## Item Creation

Items are objects that the character can carry, ranging from a flashlight to a rocket launcher. Items can be created from the Item Manager which can be opened from the Tools -> Ultimate Character Controller -> Items Manager menu option.



Start to fill in the following fields:

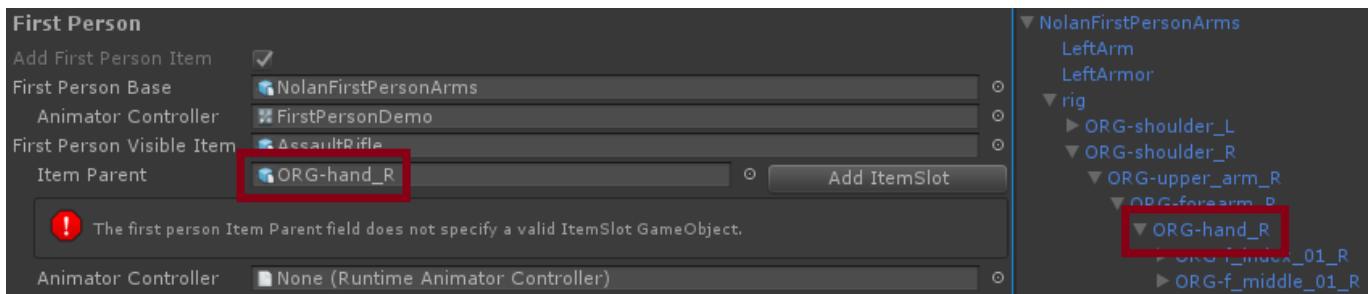
- *Name:* MyAssaultRifle
- *Item Definition:* MyAssaultRifle. This is the ItemType that was created on [this page](#).
- *Character:* Nolan. This is the same character that was created on [this page](#).
- *Animator Item ID:* 1. This number matches the ItemID parameter within the Animator Controller. Default values are located on [this page](#).



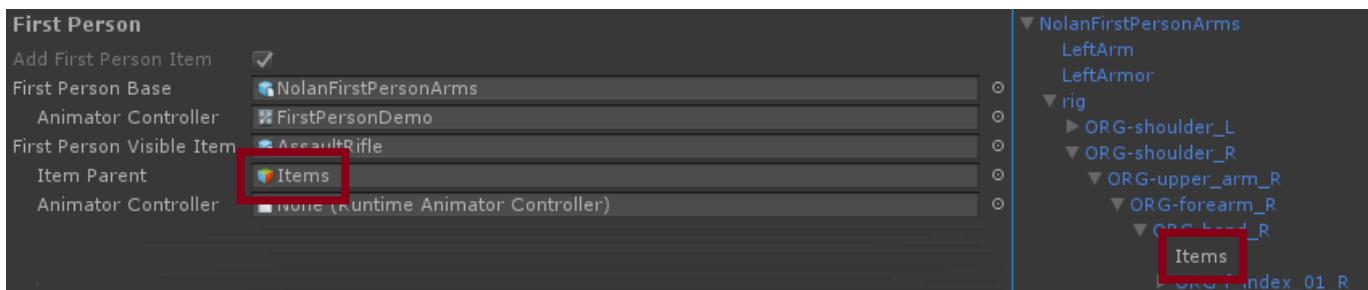
The next step depends on if you are creating a first or third person item. If your character is going to be an AI agent or part of a multiplayer game the *Third Person Visible Item* field should be specified even if your game is a first person game.

For an item used in the first person perspective the following fields should be filled out:

- *First Person Base:* NolanFirstPersonArms. This object is rendered on the screen holding the object. This is normally the character's arms. This should be a separate object so it can be activated independently of the base character model. Items can use existing First Person Objects or create a new object can be used for each item. In most cases it is easier to use an existing object for multiple items. This object should be dragged into the scene so it can be modified before the item is built.
- *First Person Base Animator Controller:* FirstPersonArmsDemo. This is the Animator Controller that the First Person Object uses. This Animator Controller should have the same parameters as the Animator Controller used by the base character. This field will not be visible if the *First Person Object* has already been setup by the Item Manager.
- *First Person Visible Item:* AssaultRifle. This is the actual item model.
- *Item Parent:* ORG-hand\_R. This is our character's right hand, a child of the *First Person Object*. This is the object that the *First Person Visible Item* will be parented to.
- *Animator Controller:* AssaultRifle. This is the Animator Controller that is added to the item and will show the assault rifle being used or reloaded.



Before we can continue the *Visible Item* must be parented to an *ItemSlot* so the inventory knows which item to equip/unequip. Once the Add ItemSlot button is pressed a new “Items” GameObject will be created and the *First Person Visible Item* will use that new GameObject for the *Item Parent*. Ensure your *First Person Base* object is within the scene in order to be able to add the *ItemSlot*.



The remaining first person field, *First Person Visible Item Animator Controller*, specifies the Animator Controller that the visible item should use. This Animator Controller will animate the assault rifle (such as when it is fired).

Third person items are parented to the actual character model so they are easier to setup:

- *Third Person Visible Item*: AssaultRifle. This is the actual item model.
- *Third Person Hand*: Right. This is the object that the *Third Person Object* is parented to. If the character is not humanoid then the object needs to be manually specified.
- *Third Person Animator Controller*: AssaultRifle. This is the Animator Controller that is added to the item and will show the assault rifle being used or reloaded. It is optional but adds a nice polish to the third person view.

The last section that needs to be specified relates to the action types. [Actions](#) perform the actual interaction with the item, such as firing a bullet or slashing a sword. Multiple actions can be added to a single item but for this example we just want the item to shoot so the following values can be specified:

- *Action Type*: Shootable Weapon. The assault rifle should be able to shoot bullets.
- *Consumable Item Definition*: MyAssaultRifleBullet. This is the same ItemType that was previously created.

The *State Configuration* field can be left blank. Once you have filled in all of the field values the builder should look like:

**New Item**

See this page for the steps on setting up various item configurations.

Name	MyAssaultRifle
Item Definition	 MyAssaultRifle (ItemType)
Character	 Nolan
Add to Default Loadout	<input checked="" type="checkbox"/>
Animator Item ID	1
<b>First Person</b>	
Add First Person Item	<input checked="" type="checkbox"/>
First Person Base	 NolanFirstPersonArms
First Person Visible Item	 AssaultRifle
Item Parent	 Items
Animator Controller	 AssaultRifle
<b>Third Person (including AI and multiplayer)</b>	
Add Third Person Item	<input checked="" type="checkbox"/>
Third Person Visible Item	 AssaultRifle
Hand	Right
Animator Controller	 AssaultRifle
Action Type	Shootable Weapon
Consumable Item Defin	 AssaultRifleBullet (ItemType)
State Configuration	 DemoStateConfiguration (StateConfiguration)
Profile	(None)
<b>Build Item</b>	

Go ahead and click “Build Item”. You’ll notice that the assault rifle has now been added to the character, but it is positioned incorrectly in both the first and third person character’s hand.



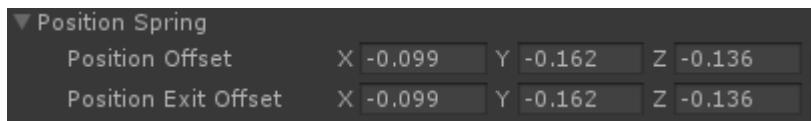
The item position can be adjusted by pressing play within Unity and adjusting the position and rotation of the item's transform component. Once you stop playing you can then paste the values back into the Transform component. Here are the values that we are using for the assault rifle:

Transform			
Position	X -0.05400822	Y 0.279001	Z 0.02200288
Rotation	X -86.33	Y -139.499	Z -130.545
Scale	X 1	Y 1	Z 1

This transform is set on the actual Assault Rifle object located underneath the character's rig or arms hierarchy. The first person assault rifle is located at: Nolan/First Person Objects/NolanFirstPersonArms/rig/ORG-shoulder\_R/ORG-upper\_arm\_R/ORG-

forearm\_R/ORG-hand\_R/Items/MyAssaultRifle. The third person assault rifle has a similar path: Nolan/rig/root/ORG-hips/ORG-spine/ORG-chest/ORG-shoulder\_R/ORG-upper\_arm\_R/ORG-forearm\_R/ORG-hand\_R/Items/MyAssaultRifle.

After you hit play the assault rifle will be positioned correctly, but it will be in the incorrect location for the first person view. This is because the new assault rifle item controls the location of the first person arms (the *First Person Object* value from the Item Builder) and the default location probably won't fit your new item. This can be adjusted within the First Person Perspective Item component of the MyAssaultRifle object under Nolan/Items/MyAssaultRifle. Most of the properties that you edit for the assault rifle will be located on this object. In this case we want to adjust the *Position Offset* and *Position Exist Offset* to:



Now when you hit play the assault rifle should be positioned correct for both first and third person! Further details on the Item Manager can be found on this page.

## Configurations

The images below show some other common item configurations.

### Sword

Similar to the assault rifle, except no Animator Controller is specified.

**New Item**

See this page for the steps on setting up various item configurations.

Name	Sword
Item Definition	Sword (ItemType)
Character	Nolan
Add to Default Loadout	<input checked="" type="checkbox"/>
Animator Item ID	22
<b>First Person</b>	
Add First Person Item	<input checked="" type="checkbox"/>
First Person Base	NolanFirstPersonArms
First Person Visible Item	Sword
Item Parent	Items
Animator Controller	None (Runtime Animator Controller)
<b>Third Person (including AI and multiplayer)</b>	
Add Third Person Item	<input checked="" type="checkbox"/>
Third Person Visible Item	Sword
Hand	Right
Animator Controller	None (Runtime Animator Controller)
Action Type	Melee Weapon
Consumable Item Defin	None (Item Definition Base)
State Configuration	DemoStateConfiguration (StateConfiguration)
Profile	(None)

**Build Item**

## Body

The body configuration does not include any visible items.

**New Item**

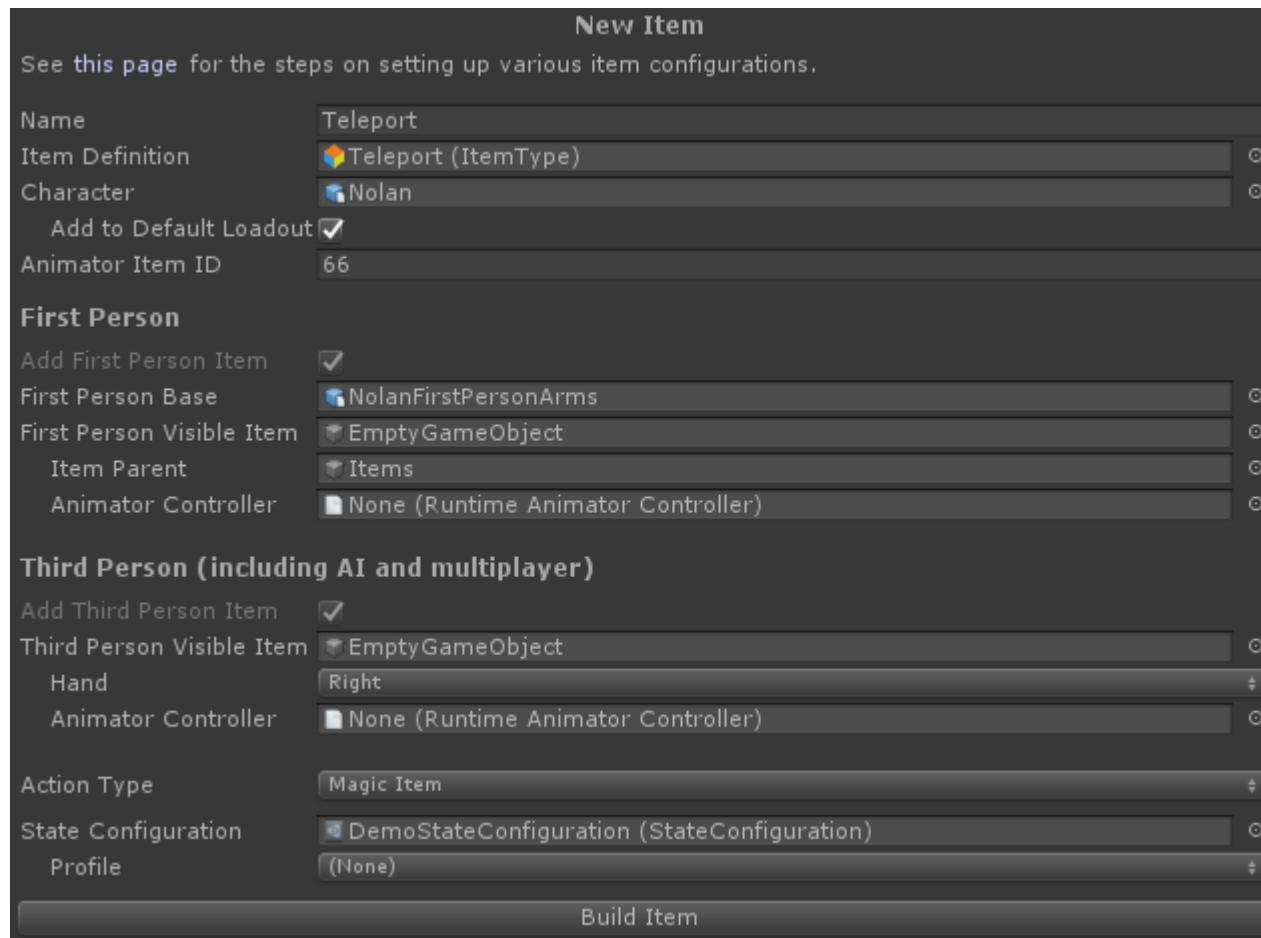
See this page for the steps on setting up various item configurations.

Name	Body
Item Definition	Body (ItemType)
Character	Nolan
Add to Default Loadout	<input checked="" type="checkbox"/>
Animator Item ID	21
<b>First Person</b>	
Add First Person Item	<input checked="" type="checkbox"/>
First Person Base	NolanFirstPersonArms
First Person Visible Item	None (Game Object)
<b>Third Person (including AI and multiplayer)</b>	
Add Third Person Item	<input checked="" type="checkbox"/>
Third Person Visible Item	None (Game Object)
Action Type	Melee Weapon
Consumable Item Defin	None (Item Definition Base)
State Configuration	DemoStateConfiguration (StateConfiguration)
Profile	(None)

**Build Item**

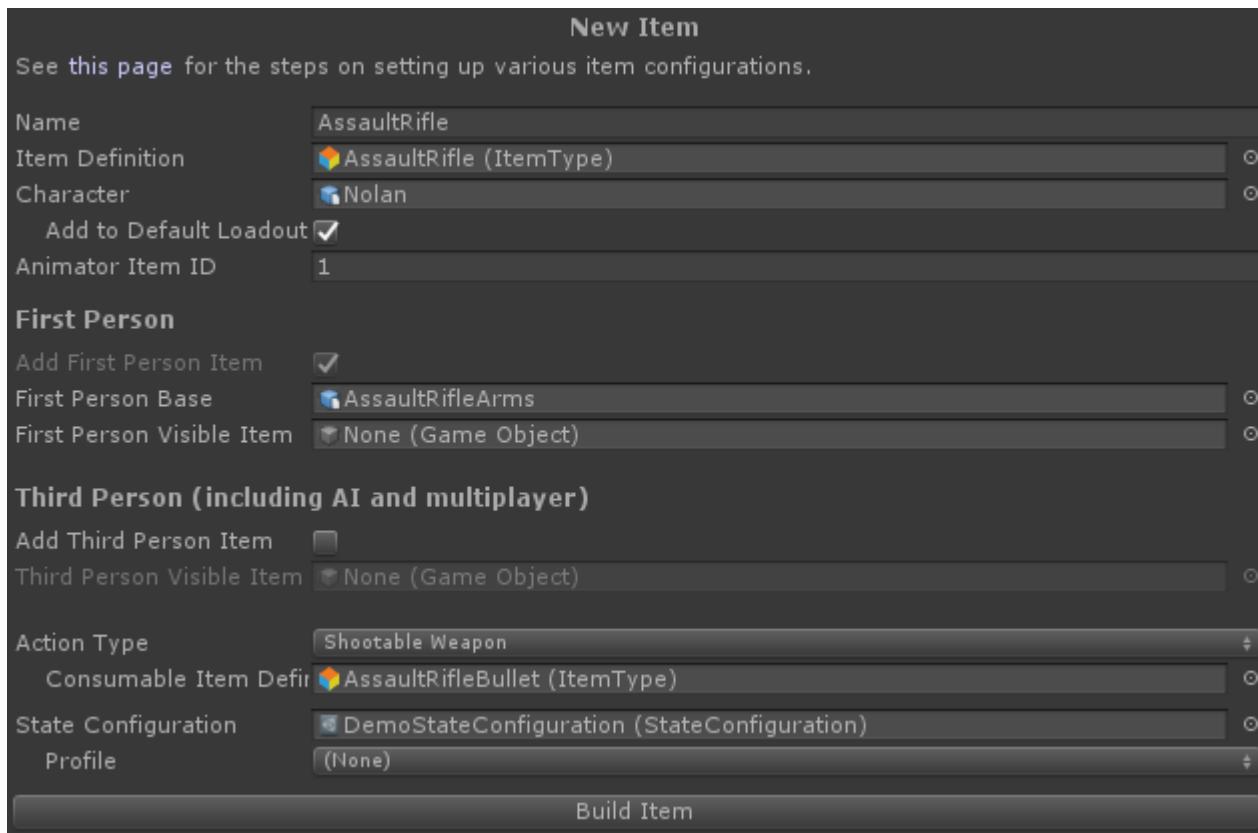
## Magic

The magic configuration is similar to the sword, except the visible items are empty GameObjects.



### Empty First Person Visible Item

For first person items the visible item can be attached to the base object. In these circumstances no first person visible item needs to be specified.



## Organization

The Ultimate Character Controller is the parent asset for a set of child assets. At their core each asset shares base functionality which is then extended based on the imported asset. The Ultimate Character Controller supports switching between first and third person perspective in addition to supporting a wide variety of items: shooting, melee, throwable, and static. If you have the First Person Controller or the Third Person Controller then only one perspective is supported while all of the items are supported. UFPS, UTPS, UFP, UTP only support one perspective and a subset of items. If at any point you'd like to add support for a different perspective or more items in your game you can import the asset which adds that functionality.

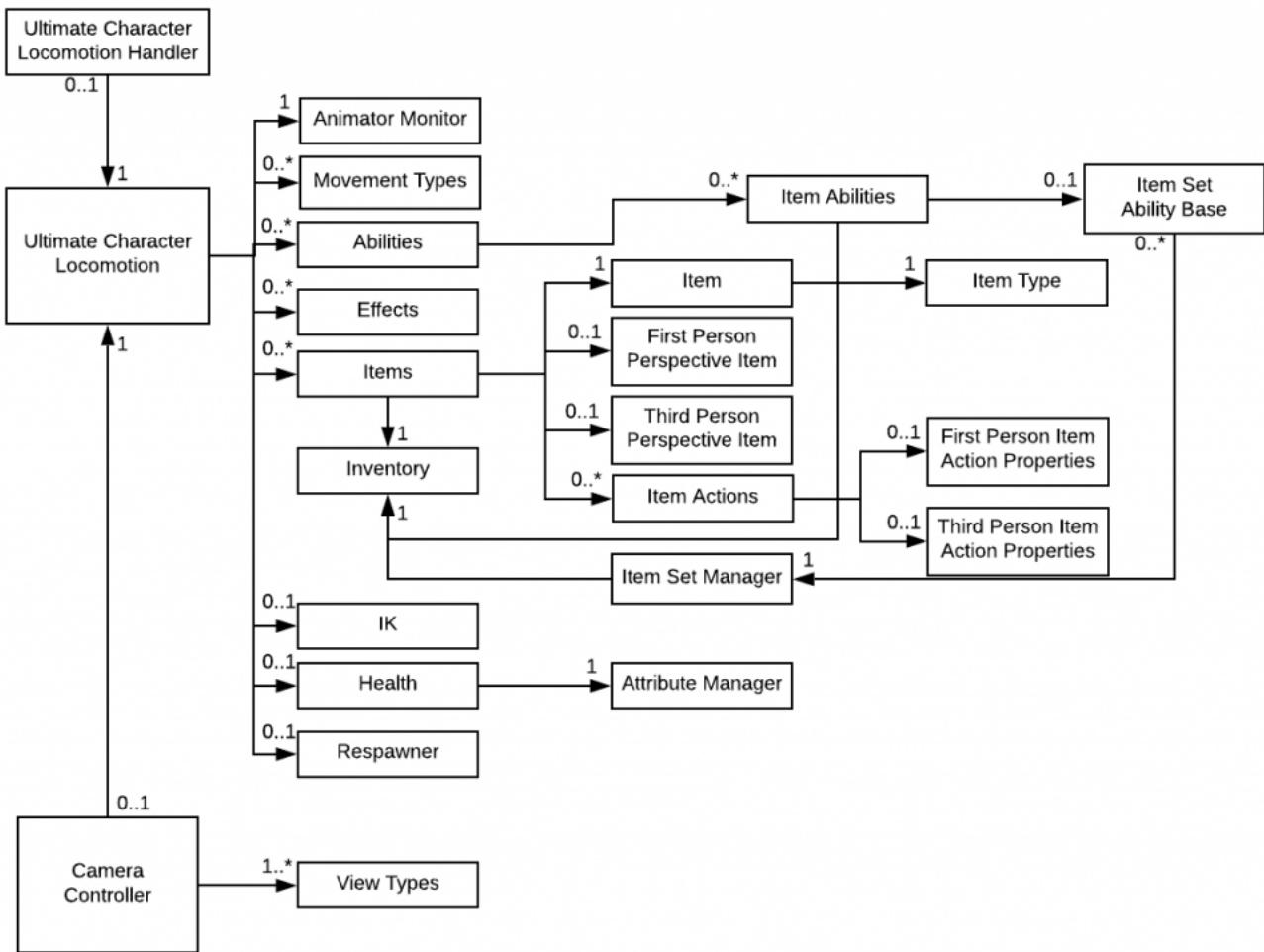
At a high level the following functionality is included in each asset:

	UFPS	UTPS	UFP	UTP	First Person Controller	Third Person Controller	Ultimate Character Controller
First Person Perspective	□	□	□	□	□	□	□
Third Person Perspective	□	□	□	□	□	□	□
Shootable Weapons	□	□	□	□	□	□	□
Melee Weapons	□	□	□	□	□	□	□
Shield	□	□	□	□	□	□	□
Throwable Items	□	□	□	□	□	□	□
AI Ready	□	□	□	□	□	□	□
Multiplayer Ready	□	□	□	□	□	□	□

## Scripting API

The API for the character controller can be found using the [Object Browser](#) of your IDE. We are using the Object Browser for API documentation because it is always up to date for the current version that you are using. It also is interactive so it's easy to navigate through the API.

# Component Overview



The main component responsible for moving the character is the **Ultimate Character Locomotion**. At a low level this component is responsible for collision detection, slopes, stairs, moving platforms, root motion, etc. Higher level this component is also responsible for the movement types, abilities, and effects system. The **Ultimate Character Locomotion** component specifies its own gravity allowing for situations like Super Mario Galaxy where the character walks around a spherical planet. The component also has its own time system allowing for the character to slow down while the rest of the world plays at a normal speed.

Movement Types, Abilities, and Effects offer an infinite amount of flexibility with the **Ultimate Character Locomotion**. Movement Types define how the character moves. Example Movement Types are a First Person Combat Movement Type which moves the character in a first person view or the Top Down Movement Type which allows for movement while the

camera is looking down on the character.

The Ability system is an extremely powerful system and allows for the character to perform entirely new functionality without having to make any changes to the Ultimate Character Locomotion component. Abilities can be extremely simple (such as one which plays a random idle animation) or extremely complex (such as climbing over a mountain). The Effect system can be thought of as a lightweight ability and they allow for any effects related to the character such as shaking the camera.

The Ultimate Character Locomotion component receives input from the Ultimate Character Locomotion Handler. If the character is an AI agent or a remote player on the network then the handler component disables itself so the character can move according to the AI or remote player input.

The Ultimate Character Locomotion component is a deterministic kinematic controller. This makes it especially appropriate to use over the network in which case client-side predication and server reconciliation can be used to ensure ultra-smooth movement. If character animations are necessary the Animator Monitor component is used which interacts with Unity's Animator component.

Any object that the character can interact with (equip, hold, shoot, etc.) are called Items. When switching between first and third person perspectives the item's model is changed. The component responsible for managing this model is called First Person Perspective Item or Third Person Perspective Item depending on the perspective. The First Person Visible Item component is responsible for the movement of the first person item using the powerful spring system. The spring system allows for procedurally generated motion and is responsible for the bobs, sways, and shakes in a first person view.

The components responsible for interaction with items are called Item Actions. Examples include the Shootable Weapon component, the Melee Weapon component, or the Shield component. Multiple Item Action components can be added to a single item allowing for a single item to be able to shoot and melee, or shoot a bullet and shoot a projectile, for example.

Items are identified by their Item Definitions. Item Types inherit Item Definitions and are used by the character controller. Item Definitions allow the controller to be integrated with the Ultimate Inventory System. Item Types are a representation of Items and are used by the Inventory and Item Set Manager components. The Inventory is an optional component which stores the amount of Item Types the character has. If no Inventory is added to the character then an unlimited number of Item Types can be used. The Item Set Manager is responsible for ensuring the correct item is equipped. Multiple categories can be specified which allows for cases such as Halo where the grenades can be switched independently from the main weapons.

The Attribute Manager is a versatile component that can be added to the character which describes a set of values that change over time. The Health component uses the Attribute Manager in order to determine the amount of health or shield the character has remaining. The Attribute Manager can be used to describe any character property including stamina, hunger, thirst, etc.

As the character is moving around in the world the Character IK component will position the limbs to prevent clipping with other objects. The Character IK component goes along with the Character Foot Effects component which uses the Surface System to spawn the correct effects for the feet. The Surface System is a generic effect spawning system which can spawn effects (decals, particles, audio, etc.) based on the type of impact. For example, if the character shoots a wood crate some wood particles may fall off of the crate, or if grass is hit then some grass blades may fly into the air. The Surface System goes along with the Decal Manager which is responsible for placing any decals, whether that is bullet holes or foot step effects.

Object Pooling is used to reduce the number of allocations necessary which increases performance. If an event needs to occur in the future the Scheduler is responsible for invoking that event. There is a preset system which allows component values to be changed at runtime based on the current state. For example, the Zoom state will decrease the camera's field of view.

The camera is controlled by the Camera Controller and the Camera Controller is responsible for managing the different View Types. The View Type describes how the camera should move and can work with the spring system for movements such as bobs, sways, or recoils. In a third person perspective the Object Fader component is added to the camera's GameObject which allows for materials to fade based on the camera position (to prevent the camera from looking inside the character). The Aim Assist component allows the camera to target a specified GameObject.

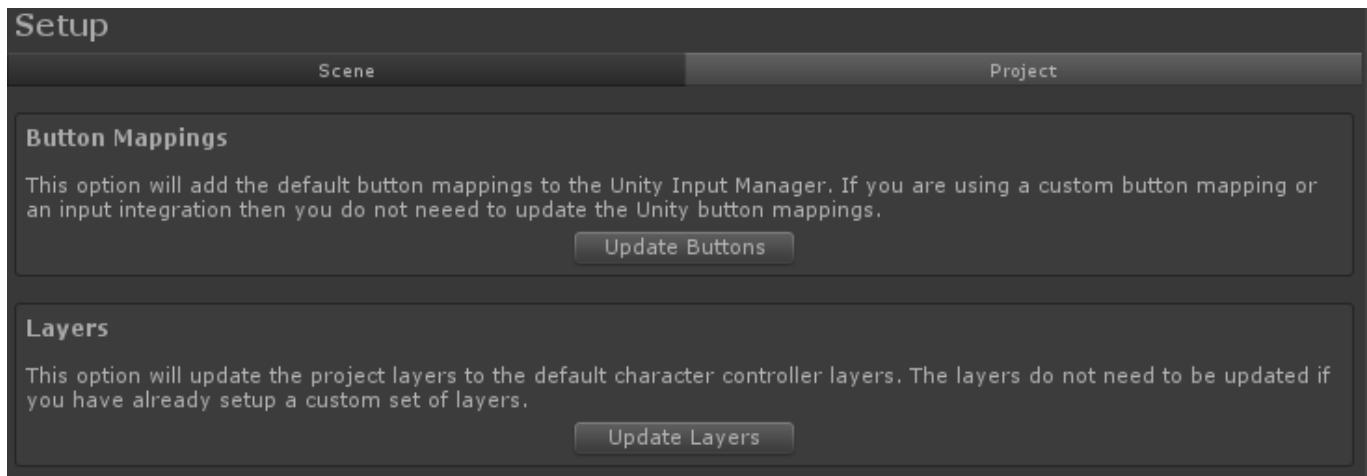
## Editors

### Setup

The Setup Manager contains options for setting up your project and scene. These options will only need to be run once when you are creating a new project or a new scene. The Setup Manager can be accessed via Tools -> Opsive -> Ultimate Character Controller -> Main Manager.

### Project

When the Ultimate Character Controller is first imported a dialog will appear if you want to update the input manager and the layers. If you select no to either of these options (or would like to start fresh with these settings) then you can access them again through the Project tab of the Setup Manager.



## Scene

The scene tab contains the options for each scene.

### Manager Setup

The Ultimate Character Controller uses a set of scene level managers for various tasks such as the object pool or surface manager. These managers should be added to the scene initially and will be placed on the “Game” GameObject within the scene.

### Camera Setup

If the character is controlled by a player (versus being an AI or networked character) then it will most likely have a camera following it. The Camera Setup button will add the camera and necessary components to the scene. If a camera is already within the scene and has the “MainCamera” tag then that camera will be used.

The *State Configuration* field allows for the camera to be preconfigured with already-defined values. This is useful if you have a set of default camera settings that you like to use and want to apply it to a new camera. See the [State Configuration](#) page for more information.

### UI Setup

The UI Setup button will create a [Canvas](#) if there isn’t already one within the scene and instantiate the Monitors prefab from the Opsive/UltimateCharacterController/Demo/UI directory. This prefab gives a good starting point for you to start customizing your UI.

### Virtual Controls Setup

If you are on a mobile platform then you will likely want to control the character via virtual controls. If you click the Add Virtual Controls button then the manager will instantiate the VirtualControls prefab from the Opsive/UltimateCharacterController/Demo/UI directory. This prefab contains an example of using a virtual joystick, touchpad, and button on the screen.

# Character

The Character Manager will setup a character model to work with the Ultimate Character Controller. Both humanoid and generic models are supported. The Character Manager can be accessed from the Tools -> Opsive -> Ultimate Character Controller -> Character Manager menu option. When the manager is opened there are two tabs: New Character and Existing Character.

## New Character

The New Character tab will create a new character so it can work with the Ultimate Character Controller. The [Quick Start](#) guide can be used for quickly setting up a new character. This page will go more in depth for each option.

### Perspective

The Perspective field allows you to select a first, third, or both perspective. In order for the perspective to be selected you must have the corresponding asset imported. The both option requires both a first person perspective and third person perspective asset to be imported. See the [Component Organization](#) page for more details.

### First/Third Person Movement

Specifies the default Movement Type for the character. More [Movement Types](#) can be added after the character has been created.

### Start Perspective

If the *Perspective* field is Both, this field specifies which perspective the character should start in. When the character is built the correct movement type will be activated based on the start perspective. The activated movement type should match the active view type on the camera controller.

### Character

A reference to the character model. This field can be left empty if you're using a first person perspective and don't want to have a body/legs.

### Animator

This field is required for third person characters, AI characters, or characters that are going to be used on the network. If your character is a humanoid then it can use [animation retargeting](#) and use the Animator Controller included with the controller. [Generic characters](#) require their own Animator Controller to be setup.

### First Person Arms

Visible if using a first person perspective. When the first person items are created they are drawn first by the camera. This field specifics the arms that should be drawn. Multiple items

can specify the same set of arms, or each item can use their own set of arms.

### **Third Person Objects**

Visible if using a first person perspective. This field allows you to specify any object that should be hidden when the first person view is active. In most cases this will include the character's head and arms. This field will automatically resize when a new object has been added. If your character model does not have the head/arms separated and you don't have access to the original source file you can separate the objects with a tool such as the [FPS Mesh Tool](#).

### **Standard Abilities**

If enabled the character will have the Jump, Fall, Move Towards, Item Equip Verifier, Speed Change, and Height Change abilities added to it.

### **AI Agent**

Allows the character to be specified that it will be used as an AI agent. AI agents do not need certain components that check for player input such as the Ultimate Character Locomotion Handler component. With this option enabled it will also not add the Unity Input component because the AI will be controlling the input rather than an external input source such as a keyboard or controller.

This option does not actually add any AI to the character, it just sets up the character to be able to be used with an AI solution (such as Behavior Designer).

### **NavMeshAgent**

If *AI Agent* is selected, the *NavMeshAgent* field will add the NavMeshAgent Movement ability to the character if it is enabled. This allows your character to be used with Unity's NavMesh.

### **Items**

Will the character carry items? If this option is enabled then the item related components such as the Inventory will be added.

### **Item Collection**

Specifies the location of the Item Collection asset if the *Items* field is enabled.

### **Health**

This option should be enabled if the character has any attributes such as health, shield, or stamina.

### **Unity IK**

Should the character use Unity's [Inverse Kinematics system](#)? Unity's IK system does not

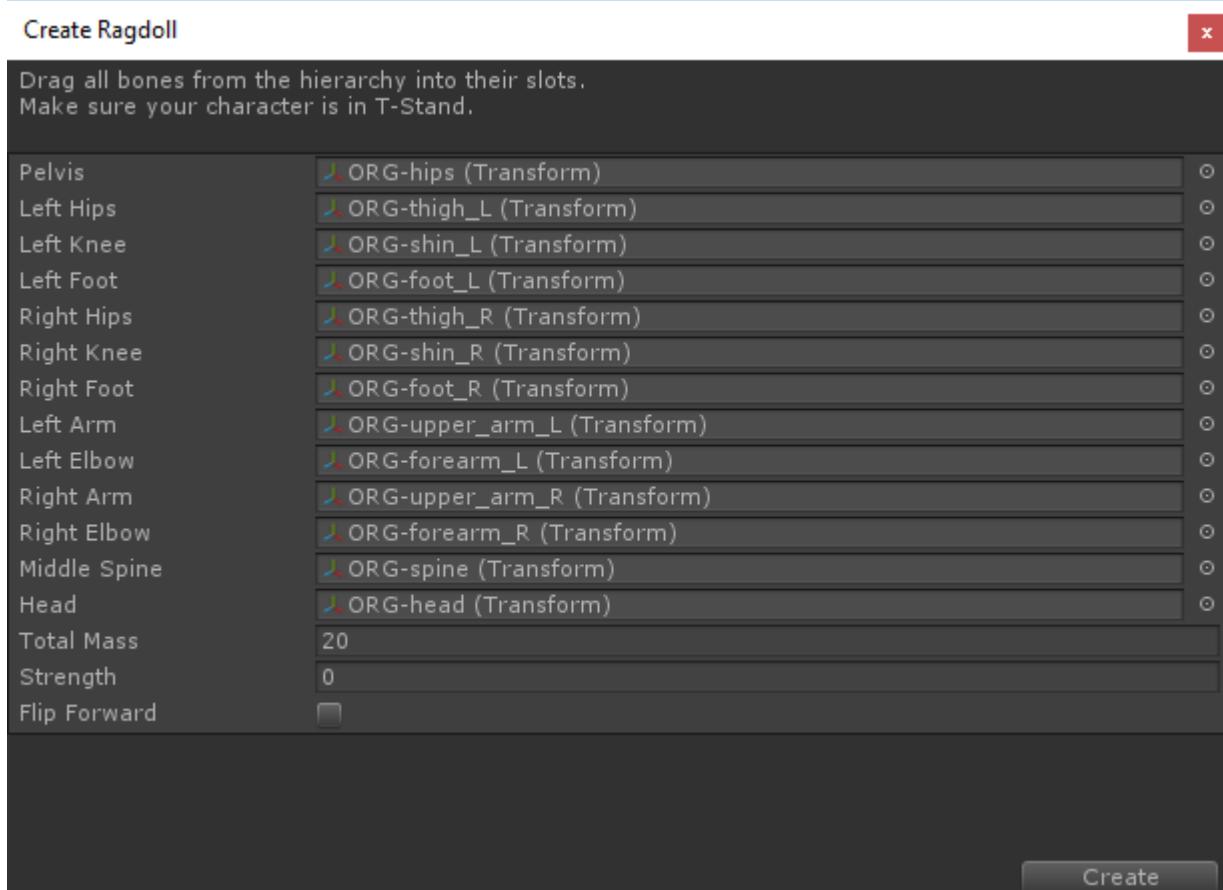
support generics so your character must be a humanoid to use it.

### Foot Effects

Specifies if the character should contain any foot effects such as footprints. This uses the [Surface System](#).

### Ragdoll

If the Ragdoll toggle is enabled then Unity's Ragdoll Creator will open after clicking Build Character. If your character is a humanoid then the values will automatically be filled in.



### State Configuration

Allows for the character to be preconfigured with already defined values. This is useful if you have a set of default character settings that you like to use and want to apply the same values to a new character. See the [State Configuration](#) page for more information.

### Existing Character

The Existing Character options are a subset of the New Character options. These options allow you to easily modify a character which has already been created. To get started you must input the existing character in the *Character* field. An error will appear within the window if your character isn't an Ultimate Character Controller character.

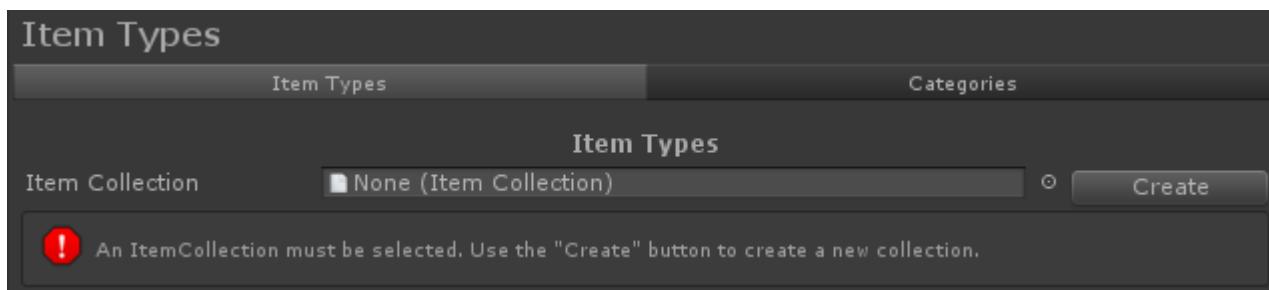
After you have made your selections you can click the Update Character button. This will apply the changes to the character.

# Item Types

The Item Type Manager allows for the creation of new Item Types and Categories. [Item Types](#) are used by the Inventory system to identify a specific Item. Categories are used to group Item Types within the [Item Set Manager](#). The Item Type Manager can be accessed from the Tools -> Opsive -> Ultimate Character Controller -> Item Type Manager menu option. This window is separated into two tabs: Item Types and Categories.

## Item Collection

When an Item Type or Category is created it is added to an Item Collection. An Item Collection is a storage container for the Item Types and Categories. When you open the Item Type Manager for the first time it will show the Item Collection from the included demo scene but you can change this to a new Item Collection for your game:



The same Item Collection should be used throughout all of your project. If your game contains multiple characters (such as AI agents) they should all use the same Item Collection. We recommend that you **create a new Item Collection specifically for your project**. This will prevent any Ultimate Character Controller updates from conflicting with any changes that you have made.

## Item Types

Item Types are a representation of an Item. A listing of all of the Item Types from the specified Item Collection will appear within the manager:

## Item Types

The screenshot shows a software interface for managing item types. At the top, there are two tabs: "Item Types" (selected) and "Categories". Below the tabs, there is a search bar with a magnifying glass icon and a clear button (X). A list of item types is displayed in a table format:

Item Collection	Name	Actions
DemoItemCollection (ItemCollection)	AssaultRifle	☰ ↻ X
	AssaultRifleBullet	☰ ↻ X
	Sword	☰ ↻ X
	Pistol	☰ ↻ X
	PistolBullet	☰ ↻ X
	FragGrenade	☰ ↻ X
	ThrownFragGrenade	☰ ↻ X
	Bow	☰ ↻ X
	BowArrow	☰ ↻ X
	Body	☰ ↻ X
	Shotgun	☰ ↻ X
	ShotgunBullet	☰ ↻ X
	RocketLauncher	☰ ↻ X
	Rocket	☰ ↻ X

Each row contains three buttons in the "Actions" column: a location icon, a duplicate icon, and a delete icon.

New Item Types can be added by specifying their name within the Name field. The field directly beneath that allows for you to search for existing Item Types. Each Item Type row contains three buttons:

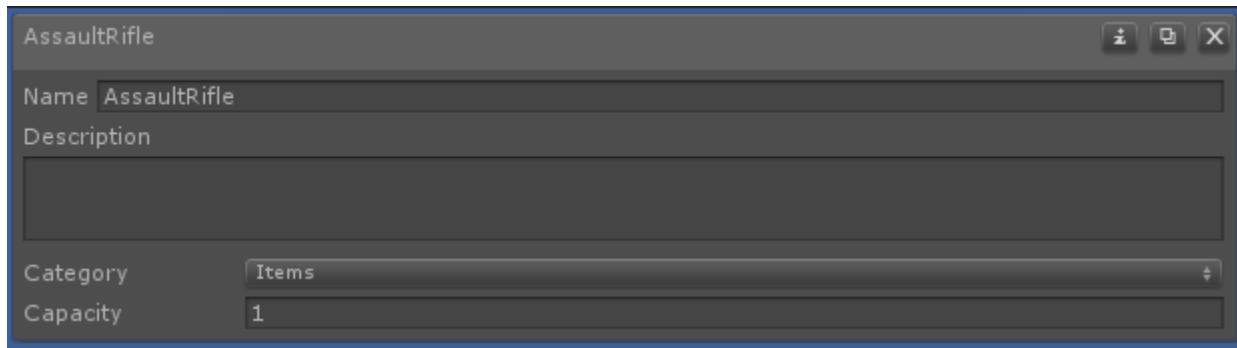


The leftmost button will identify the location of the actual Item Type asset. This asset will be a child of the Item Collection.

The center button will duplicate the current Item Type.

The rightmost button will delete the current Item Type.

The Item Type row will expand if you click on it revealing more options:



### Name

The name of the Item Type. This name must be a unique name within the Item Collection.

### Description

A friendly description of what the Item Type is used for.

### Category

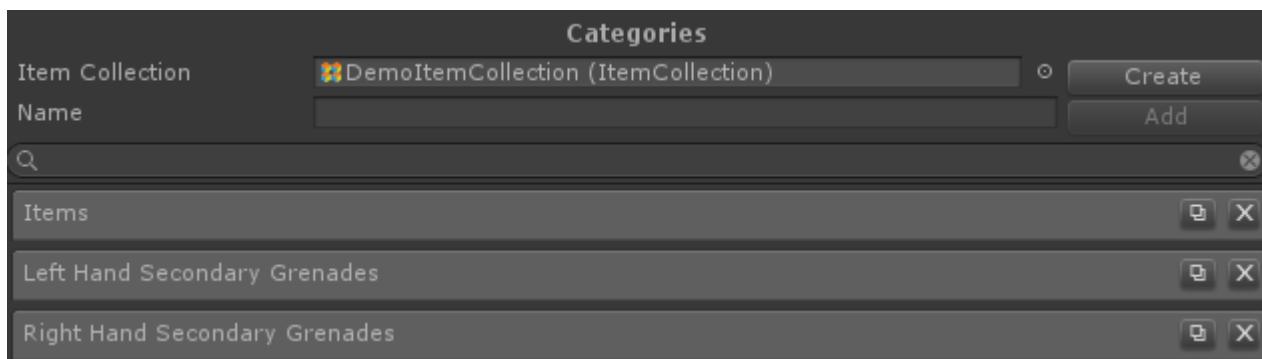
The Category that the Item Type belongs to. A single Item Type can belong to any number of Categories. If the Item Type is consumable (such as an assault rifle bullet) then it does not need to belong to a category. Categories are used by the Item Set Manager.

### Capacity

The maximum number of the Item Type that the inventory can carry.

## Categories

Categories define a grouping of Item Types and are primarily used by the Item Set Manager. Similar to Item Types, Categories must belong to an Item Collection. Once an Item Collection is defined you'll be able to create new categories, as well as see a listing of the existing categories.



Each Category row contains two buttons:



The left button will duplicate the current Category.

The right button will delete the current Category.

Clicking on the Category row will allow you to edit the category name.

#### **Name**

The name of the Category. This name must be a unique name within the Item Collection.

## **Item**

The Item Manager will setup an Item for the Ultimate Character Controller. The Item Manager can be accessed from the Tools -> Opsive -> Ultimate Character Controller -> Item Manager menu option. When the manager is opened there are two tabs: New Item and Existing Item.

### **New Item**

The New Item tab allows new items to be created using the Item Manager. The [Quick Start](#) guide can be used for quickly setting up a new item. This page will give more details on each option.

#### **Name**

Specifies the name of the item. It is recommended that this be a unique name though it is not required.

#### **Item Definition**

The Item Definition that the Item should use. Item Definitions are the parent class of Item Types which are used by the character controller. Item Types must first be created within the [Item Type Manager](#) and must be unique for each slot. As an example, if you are dual wielding pistols each pistol can use the same Pistol Item Type, but each pistol must be in a different slot.

#### **Character**

Specifies the character that the Item should be added to. This field should be empty if the item will be added at [runtime](#).

#### **Add To Default Loadout**

If a character is specified the Item Type can automatically be added to the Inventory's Default Loadout.

#### **Slot ID**

The ID of the slot that the Item should occupy. The Item will be parented to the Item Slot component for the corresponding perspective. The *Slot ID* must match for both first and third person perspective.

## **Animator Item ID**

The ID of the Item within the Animator Controller. This ID is used by the *SlotXItemID* parameter within the Animator Controller and it must be unique for each item. A listing of default Item IDs can be found on [this page](#).

## **Add First Person Item**

Should the first person item perspective be added?

## **First Person Base**

A reference to the base object that should be used by the first person item. This will usually be the character's separated arms. A single object can be used for multiple Items. If a *First Person Visible Item* is specified this object should be within the scene so the Item Slot can be specified.

## **First Person Base Animator Controller**

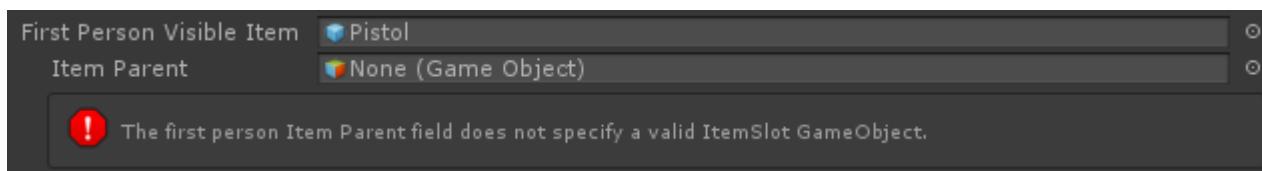
A reference to the Animator Controller used by the *First Person Base* field. This Animator Controller will only be active when the Item is equipped.

## **First Person Visible Item**

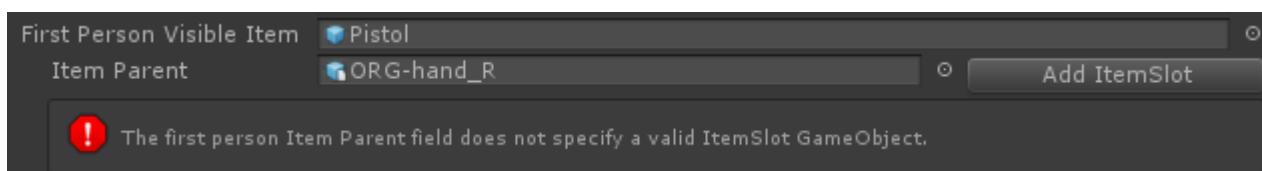
Specifies the Item object that is actually visible and rendered to the screen, such as the assault rifle or sword. This field should be left blank if you are adding an item that is part of the character's body such as a fist for punching.

## **First Person Visible Item Parent**

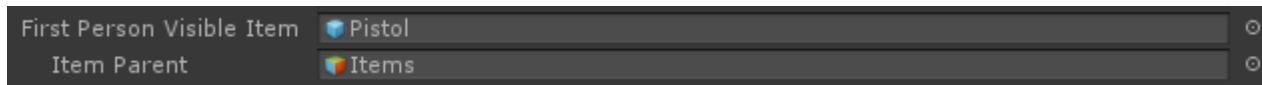
Specifies the object that the *First Person Visible Item* should be parented to. If the *First Person Object* has not been setup to be used by items before then when you first specify the *First Person Visible Item* you'll get an error saying the Item Parent needs to be specified:



In this example the pistol should be parented to the right hand so we'll specify that for the parent:



The Item Parent must be a child of the *First Person Object*. As soon as the Add ItemSlot button is pressed the error should go away:



### First Person Visible Item Animator Controller

Specifies the Animator Controller that should be used by the *First Person Visible Item*.

### Add Third Person Item

Should the third person item perspective be added?

### Third Person Visible Item

Specifies the third person item object. This is the object that will be visible and rendered to the screen, such as the assault rifle or sword.

### Third Person Animator Controller

Specifies the Animator Controller that should be used by the *Third Person Visible Item*.

### Action Type

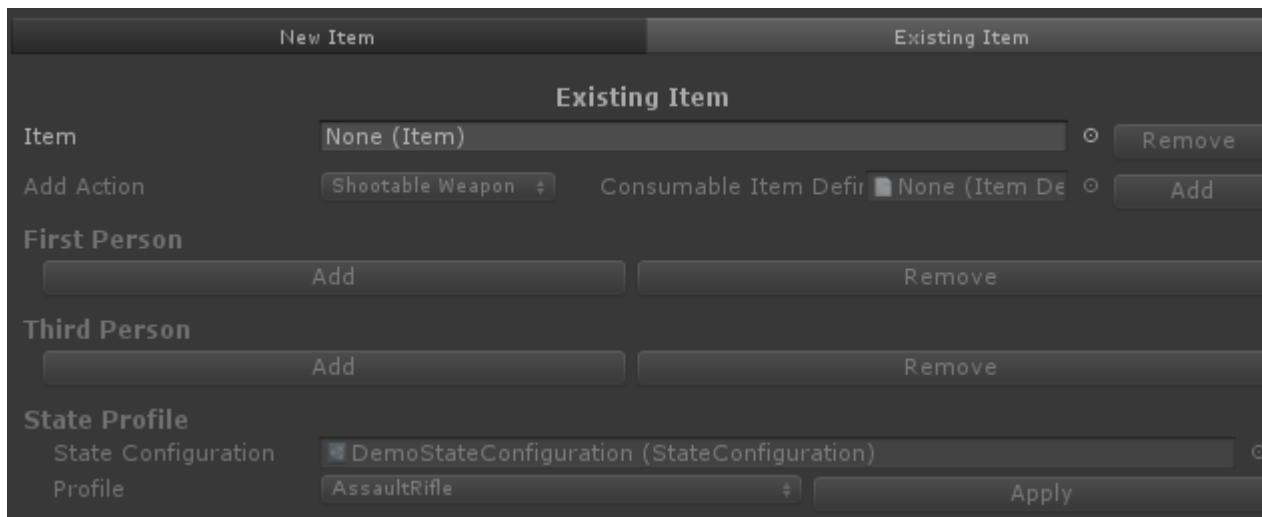
A drop down field which allows you to specify which [Item Action](#) should be added. More Item Actions can be added to the Item through the Existing Item tab.

### State Configuration

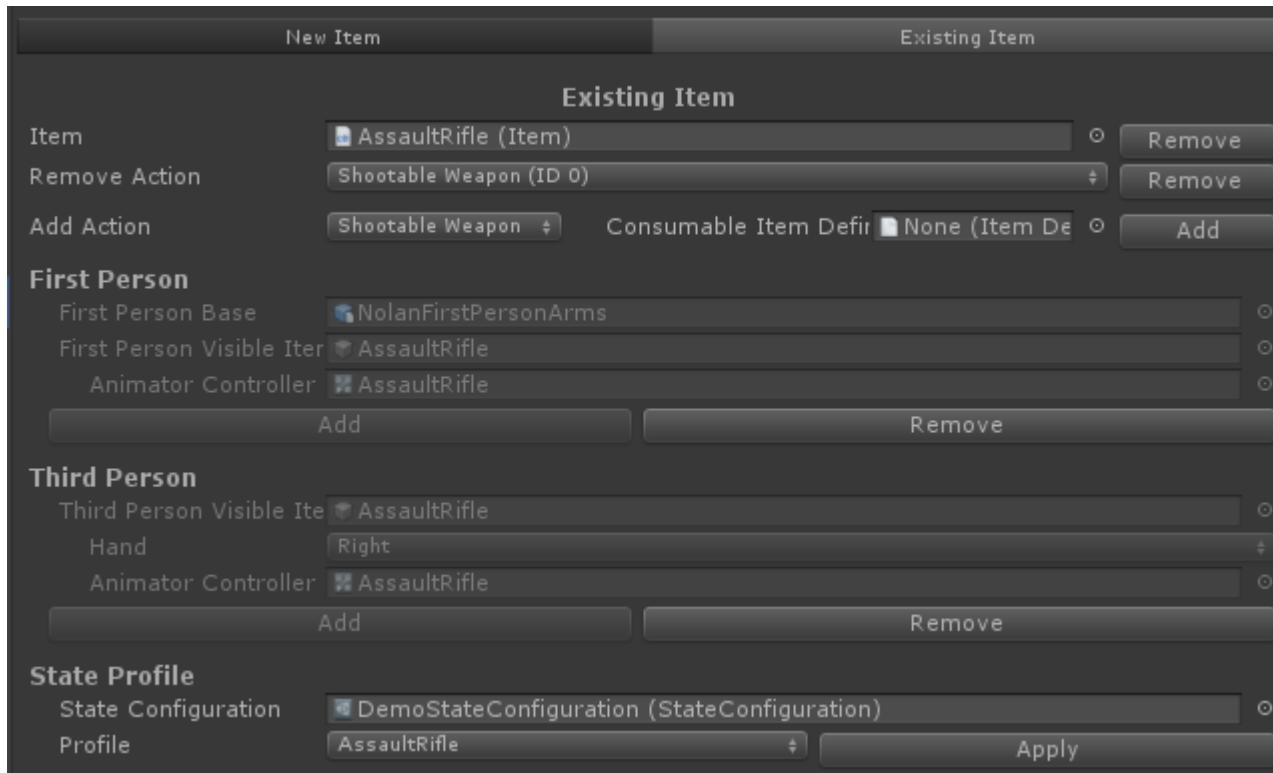
Allows for the item to be preconfigured with already defined values. This is useful if you have a specific type of item that has already been created and you'd like to apply the same values to the new item. See the [State Configuration](#) page for more information.

## Existing Item

The Existing Item tab allows you to modify Items that have already been created. When the tab is first opened the *Item* field is the only active field:



Once an Item has been specified the values will be populated:



Most of the fields are similar to the New Item fields but the following are new:

#### **Remove Item**

This will remove the entire Item from the character. This cannot be undone so ensure you want to delete the item.

#### **Add Action**

Adds a new Item Action to the Item. If the action uses a Consumable Item Definition then that Item Definition can be specified in this window.

#### **Remove Action**

Removes an existing Item Action already on the item. The Action ID is specified allowing you to identify which Item Action is being removed.

#### **Remove First Person**

Removes the objects related to the first person perspective.

#### **Remove Third Person**

Removes the objects related to the third person perspective.

# **Object**

The Object Manager allows you to conveniently create the small objects such as Item Pickups or Shells. The Object Manager can be accessed from the Tools -> Opsive -> Ultimate Character Controller -> Object Manager menu option.

The Object Manager supports the creation of the following object types:

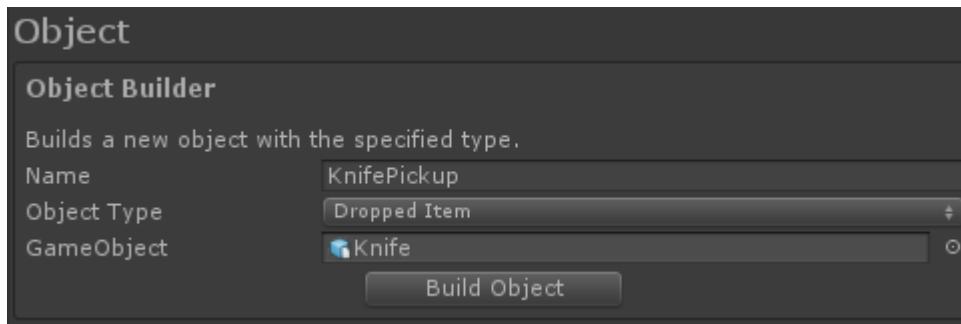
- [Item Pickup](#)
- [Dropped Item](#)
- [Health Pickup](#)
- [Projectile](#)
- [Muzzle Flash](#)
- [Smoke](#)
- [Shell](#)
- [Melee Trail](#)
- [Grenade](#)
- [Explosion](#)
- [Magic Projectile](#)
- Particle

Each of these object types have a similar workflow: enter the object name, select the object type, and select which GameObject it should use. The Object Manager will then save the created object out to a prefab in the specified location. After the object has been created it may need to be further edited so make sure you take a look at the documentation for that object type.

As an example we will create a dropped item prefab for the knife item:

1. Open the Object Manager
2. Specify the *Name*, for this example we will use “KnifePickup”.
3. Specify the type of object to create, which is the Dropped Item.
4. Specify the GameObject that should be used. We will use the pistol model because that is the object that should be dropped.
5. Select the “Build Object” button. A project path will need to be specified with the location that the prefab should be saved.

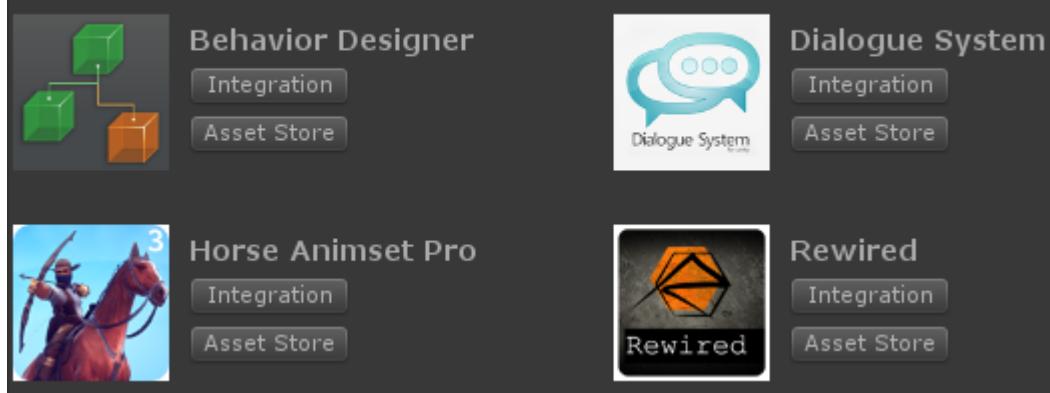
The completed editor window looks like:



## Integrations

The Integrations Manager will list all of the integrations that are available for the Ultimate Character Controller. The manager will fetch the most recent list of integrations and populate the window after it has finished loading. The Integrations Manager can be accessed from the Tools -> Opsive -> Ultimate Character Controller -> Main Manager and then clicking on the Integrations menu.

## Integrations

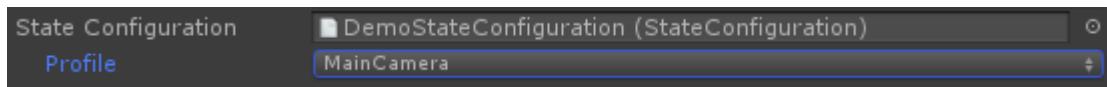


The integration package can be downloaded by clicking on the Integration button next to the asset name. You can view the asset on the Asset Store by clicking on the Asset Store link.

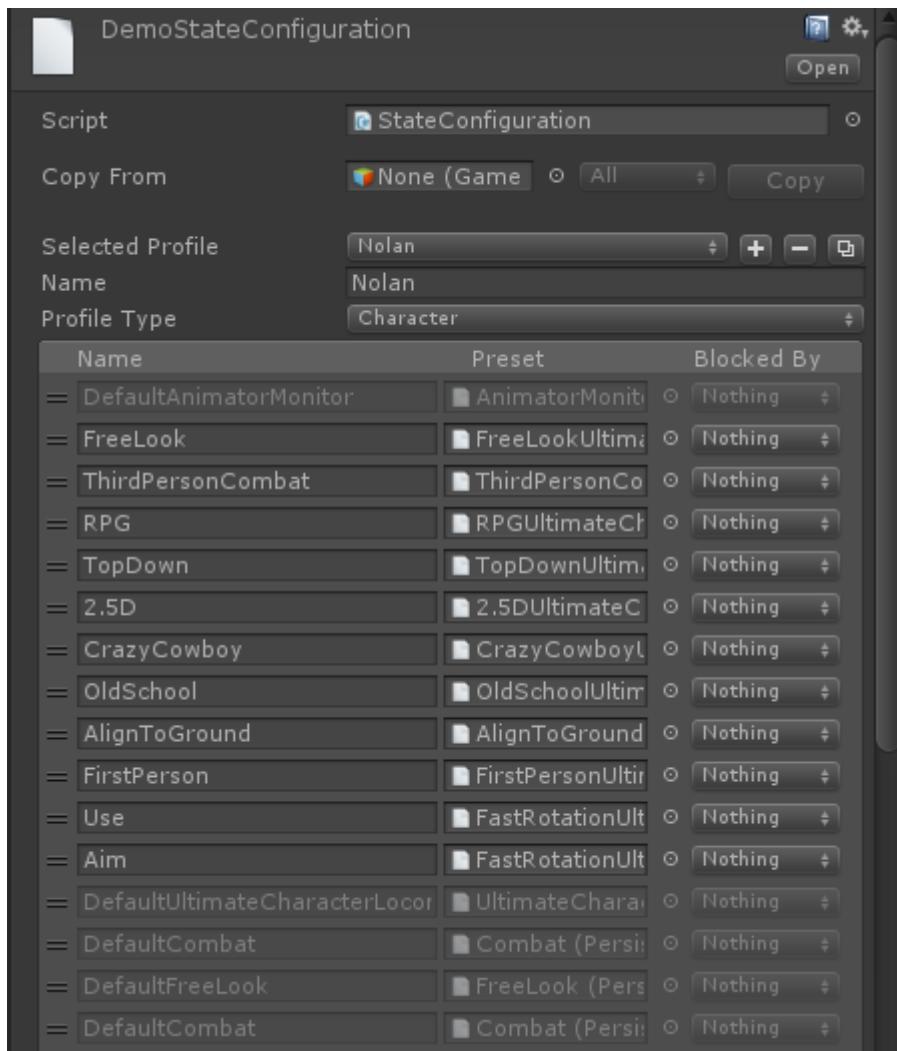
If you are an Asset Store publisher and have added integration with the Ultimate Character Controller please [get in contact](#) with us so we can add your asset to this list.

## State Configuration

The *State Configuration* field within the Camera, Character, and Item builders allow you to use a set of values that have already been setup for that object. When a *State Configuration* asset file is specified you are able to select which profile you'd like to apply to your object.



When the object is built all of the values from that profile will be applied to your object. This includes the default values for a component as well as any states that have been added to the object. All of the profiles contained within the state configuration asset are visible in the inspector if you select the state configuration asset file:



New profiles can be created by dragging the object that you'd like to create into the *Copy From* field and clicking the “Copy” button. The popup field immediately to the left of the *Copy From* field allows you to select if you'd like to copy all of the default values and states, just the default values, or just the states.

The section beneath the *Copy From* section allows you to view all of the existing profiles and their states. The states use the [preset system](#) similar to how it is used at runtime. Clicking on a preset will allow you to modify the values for that preset. A gray preset represents the default values for that object and the state name cannot be edited. New State Configuration assets can be created under the Create -> Ultimate Character Controller -> State Configuration menu option.

## Video

# Character

The Ultimate Character Controller uses a kinematic, deterministic character controller. This means that the movements can be predicted and they can also be replayed. This is extremely useful in a networked environment where the server needs to ensure the position and rotation are valid on the client. The Character Locomotion component is responsible for the core movement and for the following:

- Movement
- Collision Detection
- Root Motion
- Wall Bouncing
- Wall Gliding
- Slopes
- Stairs
- Push Rigidbodies
- Dynamic Gravity
- Variable Time Scale
- Capsule Collider and Sphere Collider support
- Moving Platforms

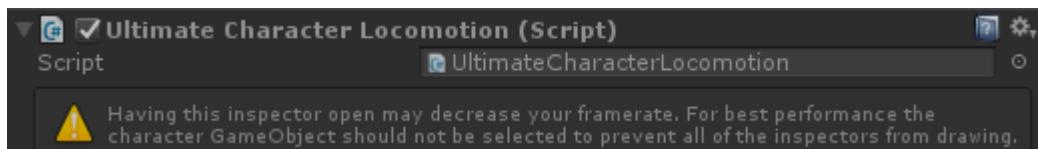
The Character Locomotion component isn't directly added to your character though - the Ultimate Character Locomotion component is instead added. The Ultimate Character Locomotion component inherits Character Locomotion and adds support for the following:

- Movement Types
- Abilities
- Effects
- Animator Knowledge

New characters should be built using the [Character Manager](#). The Character Manager will add all of the necessary components to your character, including the Ultimate Character Locomotion component.

## Editor Performance

If you have your character selected while the game is active you'll see this warning at the top of the Ultimate Character Locomotion inspector:



This warning appears because the character inspector components draw *a lot* of content to the window. The Ultimate Character Controller does prevent content from being drawn if it's not seen but you may still notice a framerate drop when the character is selected in the editor. Rest assured that this is specific to the editor so it does not affect runtime performance at all.

## API

The Ultimate Character Locomotion component works with the Deterministic Object Manager to smoothly move the object. Because of this the standard GameObject.SetActive and Transform.SetPosition/Rotation methods should not be used. Equivalent methods have been added to the Ultimate Character Locomotion component and should be used instead.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Character;
```

```

public class MyComponent : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller
character.")]
    [SerializeField] private GameObject m_Character;

    /// <summary>
    /// Set the position and deactivate the character.
    /// </summary>
    private void Start()
    {
        var characterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        if (characterLocomotion != null) {
            characterLocomotion.SetPositionAndRotation(Vector3.zero,
Quaternion.identity);
            characterLocomotion.SetActive(false);
        }
    }
}

```

The character can persist between scenes by marking the character GameObject as [DontDestroyOnLoad](#). When the character is marked as DontDestroyOnLoad the camera and UI should also be marked with DontDestroyOnLoad. This will allow all of the references to persist between scene loads.

```

private void Start()
{
    // Find the character. If the character does not exist then
    instantiate a new character with DontDestroyOnLoad.
    var character =
FindObjectOfType<Character.UltimateCharacterLocomotion>();
    if (character == null) {
        character =
Instantiate(m_Character).GetComponent<Character.UltimateCharacterLocom
otion>();
        DontDestroyOnLoad(character.gameObject);
    }
    // Find the camera. If the camera does not exist then instantiate
    a new camera with DontDestroyOnLoad.
    var camera = FindObjectOfType<Camera.CameraController>();
    if (camera == null) {
        camera =
Instantiate(m_Camera).GetComponent<Camera.CameraController>();
        // The camera needs to be aware of the instantiated character.
        camera.Character = character.gameObject;
        DontDestroyOnLoad(camera.gameObject);
    }
    // Find the active canvas. If the canvas does not exist then

```

```
instantiate a new canvas with DontDestroyOnLoad.  
    var canvasScaler =  
FindObjectOfType<UnityEngine.UI.CanvasScaler>();  
    if (canvasScaler == null) {  
        DontDestroyOnLoad(Instantiate(m_Canvas));  
    }  
}
```

## Inspected Fields

### Movement Type

The [Movement Type](#) controls the direction that the character rotates as well as the value of the inputs. Multiple Movement Types can be added to a character but only a single Movement Type is active at a time. The Movement Type can be changed with the [state system](#) or by calling *SetMovementType* on the controller.

#### First Person Movement Type

The name of the active first person movement type.

#### Third Person Movement Type

The name of the active third person movement type.

#### First Person State Name

The name of the state that should be activated when the character is in a first person perspective.

#### Third Person State Name

The name of the state that should be activated when the character is in a third person perspective.

#### Update Location

Specifies the location that the object should be updated:

- *Update*: The object will be updated within Unity's Update loop.
- *FixedUpdate*: The object will be updated within Unity's FixedUpdate loop.

#### Use Root Motion Position

Should [root motion](#) be used to move the character?

#### Root Motion Speed Multiplier

If using root motion, applies a multiplier to the root motion delta position while on the ground.

## **Root Motion Air Force Multiplier**

If using root motion, applies a multiplier to the root motion delta position while in the air.

## **Use Root Motion Rotation**

Should [root motion](#) be used to rotate the character?

## **Root Motion Rotation Multiplier**

If using root motion, applies a multiplier to the root motion delta rotation.

## **Motor Rotation Speed**

The rate at which the character can rotate. Only used by non-root motion characters.

## **Mass**

The mass of the character. The mass is used to determine how much force to apply when colliding with a Rigidbody.

## **Skin Width**

Specifies the width of the characters skin, which is used for ground detection. The larger the value the less distance between the character and ground is required for the character to be considered grounded.

## **Slope Limit**

The maximum slope angle that the character can traverse (in degrees).

## **Max Step height**

The maximum object height that the character can step on top of. This value should be less than the radius of any of the character's colliders. The reason for this is to ensure your character has a smooth movement when stepping. If the *Max Step Height* was greater than the radius of the character's colliders then there would be a large vertical jump as the controller moves the character up in the step in a single frame.

## **Motor Acceleration**

The rate at which the character's motor force accelerates while on the ground. Only used by non-root motion characters.

## **Motor Damping**

The rate at which the character's motor force decelerates while on the ground. Only used by non-root motion characters.

### **Motor Airborne Acceleration**

The rate at which the character's motor force accelerates while in the air. Only used by non-root motion characters.

### **Motor Airborne Damping**

The rate at which the character's motor force decelerates while in the air. Only used by non-root motion characters.

### **Motor Backwards Multiplier**

A multiplier which is applied to the motor while moving backwards.

### **Previous Acceleration Influence**

A (0-1) value specifying the amount of influence the previous acceleration direction has on the current velocity.

### **Adjust Motor Force On Slope**

Should the motor force be adjusted while on a slope?

### **Motor Slope Force Up**

If adjusting the motor force on a slope, the force multiplier when on an upward slope.

### **Motor Slope Force Down**

If adjusting the motor force on a slope, the force multiplier when on a downward slope.

### **External Force Damping**

The rate at which the character's external force decelerates.

### **External Force Air Damping**

The rate at which the character's external force decelerates while in the air.

### **Stick To Ground**

Should the character stick to the ground?

### **Stickiness**

If the character is sticking to the ground, specifies how sticky the ground is. A higher value means the ground is more sticky.

### **Time Scale**

The local time scale of the character. The [time scale](#) of the character can be modified

independently of the global time scale.

#### **Smoothed Bones**

An array of bones that should be smoothed by the Kinematic Object Manager.

#### **Moving State Name**

The name of the state that should be activated when the character is moving.

#### **Airborne State Name**

The name of the state that should be activated when the character is airborne.

#### **Use Gravity**

Should gravity be applied?

#### **Gravity Direction**

The normalized direction of the gravity force.

#### **Gravity Magnitude**

The amount of gravity force to apply.

#### **Detect Horizontal Collisions**

Should the character detect horizontal collisions? Disabling this will reduce the number of casts that need to be performed and increase performance.

#### **Detect Vertical Collisions**

Should the character detect vertical collisions? Disabling this will reduce the number of casts that need to be performed and increase performance.

#### **Collider Layer Mask**

The layers that can act as colliders for the character.

#### **Max Collision Count**

The maximum number of colliders that the character can detect.

#### **Max Soft Force Frames**

The maximum number of frames that the soft force can be distributed by.

#### **Rotation Collision Check Count**

The maximum number of collision checks that should be performed when rotating.

## **Max Overlap Iterations**

The maximum number of iterations to detect collision overlaps.

## **Wall Glide Curve**

A curve specifying the amount to move when gliding along a wall. The x variable represents the dot product between the character look direction and wall normal. An x value of 0 means the character is looking directly at the wall. An x value of 1 indicates the character is looking parallel to the wall.

## **Wall Bounce Modifier**

A multiplier to apply when hitting a wall. Allows for the character to bounce off of a wall.

## **Stick To Moving Platform**

Should the character stick to the moving platform? If false the character will inherit the moving platform's momentum when the platform stops quickly.

## **Moving Platform Separation Velocity**

The velocity magnitude required for the character to separate from the moving platform due to a sudden moving platform stop.

## **Min Horizontal Moving Platform Stick Speed**

The maximum speed of the platform that the character should stick when the platform collides with the character from a horizontal position.

## **Moving Platform Force Damping**

The rate at which the character's moving platform force decelerates when the character is no longer on the platform.

## **Yaw Multiplier**

Specifies how much to multiply the yaw parameter by when turning in place.

## **Abilities**

[Abilities](#) allow the controller to be extended without having to change the core controller code. Multiple abilities can be active at the same time and it is strongly recommended that you create new abilities specific to your game as it will add new unique functionality.

## **Item Abilities**

[Item Abilities](#) allow the controller to interact with any [items](#) that the character is carrying.

## Effects

[Effects](#) can be considered lightweight abilities and they are used to provide effects that affect the character or camera, such as shaking the camera during an earthquake or playing a one off audio file.

# Movement Types

Movement Types define the target rotation of the character as well as what horizontal and forward inputs should be passed to the character. It is then up to the character controller to act on this target rotation and input vector to actually move the character. Just because a value is passed to the character controller doesn't mean the controller will act on it. For example, if the movement type says the character should move forward but there is a wall in front of the character then the character will not move forward.

## Create New Movement Types

In most cases you don't need to create a new Movement Type but if you do it's a fairly easy process. New Movement Types can be created by inheriting the `MovementType` class and implementing one property and two methods:

```
/// <summary>
/// Property that returns true if the Movement Type is a first person
movement type.
/// </summary>
/// <returns>True if the movement type is a first person movement
type.</returns>
bool FirstPersonPerspective

/// <summary>
/// Returns the delta yaw rotation of the character.
/// </summary>
/// <param name="characterHorizontalMovement">The character's
horizontal movement.</param>
/// <param name="characterForwardMovement">The character's forward
movement.</param>
/// <param name="cameraHorizontalMovement">The camera's horizontal
movement.</param>
/// <param name="cameraVerticalMovement">The camera's vertical
movement.</param>
/// <returns>The delta yaw rotation of the character.</returns>
float GetDeltaYawRotation(float characterHorizontalMovement, float
characterForwardMovement, float cameraHorizontalMovement, float
cameraVerticalMovement)

/// <summary>
/// Gets the controller's input vector relative to the movement type.
/// </summary>
/// <param name="inputVector">The current input vector.</param>
```

```
/// <returns>The updated input vector.</returns>
Vector2 GetInputVector(Vector2 inputVector)
```

### GetDeltaYawRotation

The GetDeltaYawRotation method returns the difference between the character's current yaw rotation and the target yaw rotation. This rotation can depend on the camera's rotation or it can be completely independent of the camera.

### GetInputVector

Returns the vector that will be used to actually move the character. The input vector is a Vector2 with the x value representing the horizontal movement and the y value representing forward movement. A lot of Movement Types just return the inputted vector but in some cases (such as Top Down or 2.5D) the input vector should be adjusted based on the camera's rotation.

### Example

The following Movement Type is an example of a complete movement type that prevents the character from rotating and also does not modify the input vector. This means when the character moves they will move relative to the character's rotation.

```
using Opsive.UltimateCharacterController.Character.MovementTypes;

public class MyMovementType : MovementType
{
    public override bool FirstPersonPerspective { get { return false; } }

    public override float GetDeltaYawRotation(float characterHorizontalMovement, float characterForwardMovement, float cameraHorizontalMovement, float cameraVerticalMovement)
    {
        return 0;
    }

    public override Vector2 GetInputVector(Vector2 inputVector)
    {
        return inputVector;
    }
}
```

## Included Movement Types

# First Person Combat

The First Person Combat movement type will move the character relative to the camera's direction. The character can strafe and move backwards as is typical with a first person camera. The [First Person Combat](#) view type should be used in conjunction with this movement type.

## **Input      Character Results**

Forward	The character will move forward without changing rotation.
Left	The character will strafe left without changing rotation.
Right	The character will strafe right without changing rotation.
Backwards	The character will move backwards without changing rotation.

# First Person Free Look

The First Person Free Look movement type will allow the character to rotate and move independently of the camera. This movement type is great when you want to allow the character to observe the world without having to completely change directions. The [First Person Free Look](#) view type should be used in conjunction with this movement type.

## **Input      Character Results**

Forward	The character will move forward without changing rotation.
Left	The character will strafe left without changing rotation.
Right	The character will strafe right without changing rotation.
Backwards	The character will move backwards without changing rotation.

# Third Person Adventure

The Third Person Adventure movement type will move the character relative to the input in relation to the camera. The character will always play the forward animation and cannot strafe or play a backwards animation. The [Third Person Adventure](#) view type should be used in conjunction with this movement type.

## **Input      Character Results**

Forward	The character will move forward without changing rotation.
Left	The character will play the forward animation while rotating to face the left direction of the camera.
Right	The character will play the forward animation while rotating to face the right direction of the camera.
Backwards	The character will play the forward animation while rotating to face the backwards direction of the camera. The character will end up walking towards the camera in this case.

# Third Person Combat

The Third Person Combat movement type will move the character relative to the camera's direction. The character can strafe and move backwards and will not change rotation. The [Third Person Combat](#) view type should be used in conjunction with this movement type.

## Input      Character Results

Forward	The character will move forward without changing rotation.
Left	The character will strafe left without changing rotation.
Right	The character will strafe right without changing rotation.
Backwards	The character will move backwards without changing rotation.

# Third Person Four Legged

The Third Person Four Legged movement type allows the character to move as if they are on four legs. This is most useful for generic characters. The character cannot strafe and will rotate when moving left or right.

## Input      Character Results

Forward	The character will move forward without changing rotation.
Left	The character will turn left while changing both position and rotation.
Right	The character will turn right while changing both position and rotation.
Backwards	The character will move backwards without changing rotation.

# Third Person Pseudo3D (2.5D)

The Pseudo3D (2.5D) movement type allows the character to move relative to a 2.5D camera. This Movement Type has many options that can be configured to fine tune the movement such as if the character should be able to move along the local z axis. An optional Path can be used so the character will follow a [cubic bezier curve](#). The [Pseudo3D \(2.5D\)](#) view type should be used in conjunction with this movement type.

## Input      Character Results

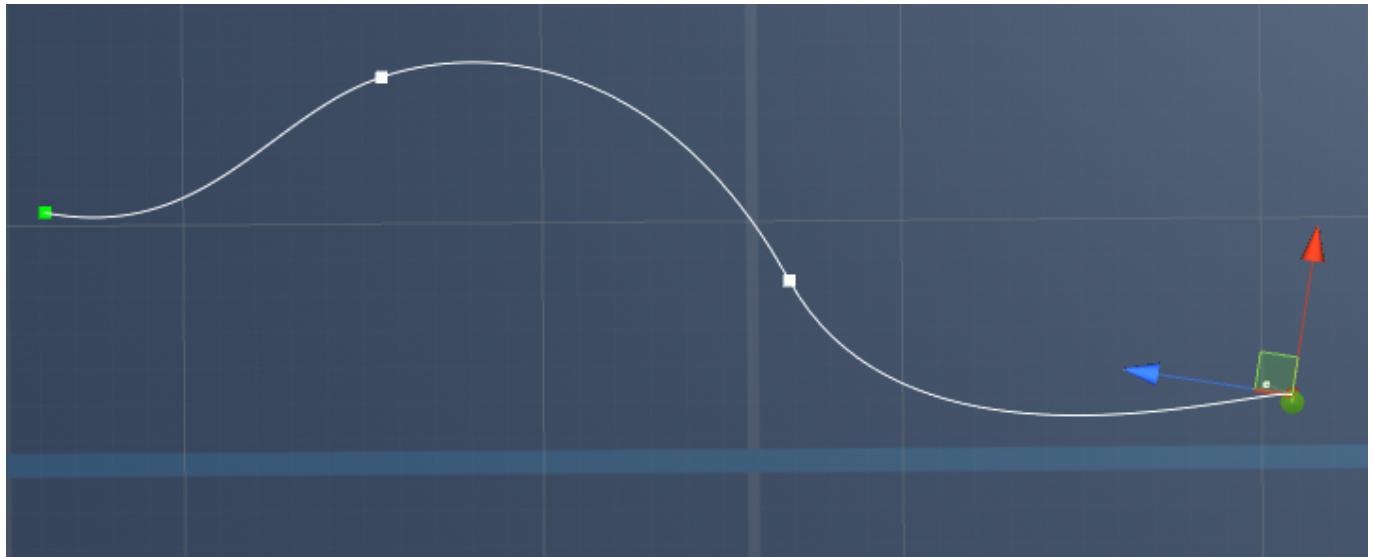
Forward	If <i>Allow Depth Movement</i> is enabled the character will move in the forward direction relative to the camera (away from the camera, into the scene). If <i>Allow Depth Movement</i> is disabled then the character will not move.
Left	The character will move to the relative left direction of the camera. If <i>Look In Move Direction</i> is enabled then the character will rotate in this direction as well.
Right	The character will move to the relative right direction of the camera. If <i>Look In Move Direction</i> is enabled then the character will rotate in this direction as well.
Backwards	If <i>Allow Depth Movement</i> is enabled the character will move in the backwards direction relative to the camera (into the camera). If <i>Allow Depth Movement</i> is disabled then the character will not move.

## Setup

A 2.5D character can be setup by performing the following:

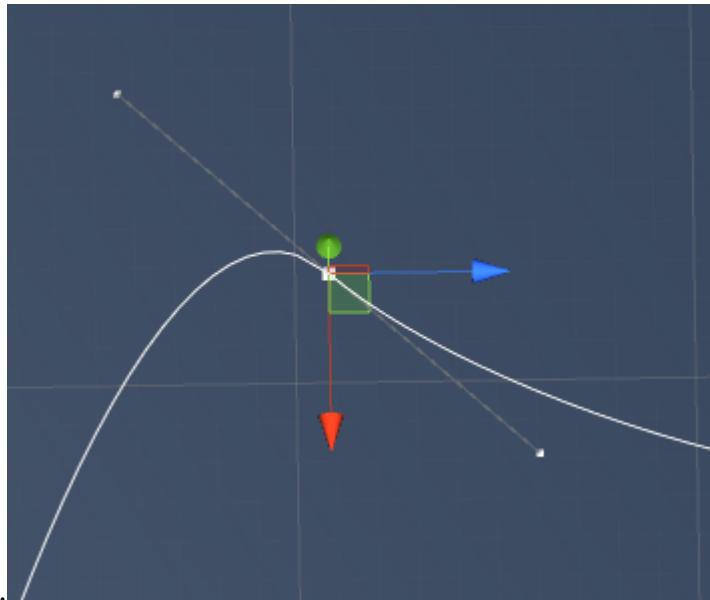
1. Create a new character using the [Character Manager](#). The perspective should be Third Person and have a Movement Type value of Pseudo3D.
2. Ensure your [camera is setup](#) to use the Pseudo3D View Type.
3. Assign the character to the camera's *Character* field.
4. Optionally setup the path that the character should follow (below).

## Path



A curve can define the orientation that the camera faces as the character traverses the scene. This path can be setup by doing the following:

1. Add a new GameObject to your scene that will be the container for the curve. Add the Path component to this GameObject.
2. Click on the "Add Curve Segment" button under the path component. A new curve segment will be added to the scene.
3. Adjust the endpoint positions of the curve. A new segment can be added by clicking on the "Add Curve Segment" button again.
4. The start curve point will be colored green, and the end curve point will be colored red. Points in the middle of the curve will be white.
5. Curve points can be adjusted by clicking on them. Endpoints have a single tangent associated with them which can adjust the curve's slope. Midpoints have two tangents associated with it and is also used for adjusting the curve's slope. The tangent can be adjusted by first selecting an midpoint or endpoint and then selecting the smaller point



connected to it.

6. Assign the newly created path to the Path field of the 2.5D Movement Type.
7. Add the [Follow 2.5D Path](#) ability to your character.

## Inspected Fields

### Allow Depth Movement

Can the character move along the depth axis (the z axis relative to the camera)?

### Look In Move Direction

Should the character look in the direction of movement?

### Look Rotate Buffer

A small buffer used to prevent the character from quickly switching directions when the mouse is near the character's origin.

### Path

The path that the character should orient towards. If null then the character will be oriented towards the look source direction.

## Third Person RPG

The Third Person RPG movement type uses a control scheme similar to the standard setup of the RPG genre. The [RPG](#) view type should be used in conjunction with this movement type. The Ultimate Character Locomotion Handler component should have the *Horizontal Input* value set to the “Alt Horizontal” input mapping.

<b>Input</b>	<b>Character Results</b>
--------------	--------------------------

Forward    The character will move forward without changing rotation.

Left        The character will strafe left without changing rotation. The default mapping is the Q button.

Right	The character will strafe right without changing rotation. The default mapping is the E button.
Backwards	The character will move backwards without changing rotation.
Rotate	The character will turn left or right depending on the axis input values. The default mapping is the middle mouse button.
Turn	The character will turn left or right depending on the button input values. The default mapping is the A and D button.
AutoMove	The character will automatically move in the forward direction. The default mapping is the F button.

## Inspected Fields

### Rotate Input Name

The name of the rotate input mapping.

### Turn Input Name

The name of the turn input mapping.

### Turn Multiplier

The amount to multiply the turn value by.

### Auto Move Input Name

The name of the auto move input mapping.

## Third Person Top Down

The Top Down movement type allows the character to move relative to a top down camera. If *Look In Move Direction* is enabled then the character will always face in the direction that they are moving, otherwise the character will face the direction of the mouse. The [Top Down](#) view type should be used in conjunction with this movement type.

### Input      Character Results

Forward	The character will move up relative to the camera's direction if <i>Relative Camera Movement</i> is enabled, otherwise the character will move in the up direction relative to the world axis.
Left	The character will move left relative to the camera's direction if <i>Relative Camera Movement</i> is enabled, otherwise the character will move in the left direction relative to the world axis.
Right	The character will move right relative to the camera's direction if <i>Relative Camera Movement</i> is enabled, otherwise the character will move in the right direction relative to the world axis.
Backwards	The character will move down relative to the camera's direction if <i>Relative Camera Movement</i> is enabled, otherwise the character will move in the down direction relative to the world axis.

## Inspected Fields

### Relative Camera Movement

Should the character move relative to the camera's direction?

### Look In Move Direction

Should the character look in the direction of movement?

## Abilities

The ability system is an extremely versatile system which allows you to add new functionality to your character without having to modify the core character locomotion system. Any “extra” character functionality within the Ultimate Character Controller is implemented using the ability system - this includes jumping, restricting the character’s rotation, or even firing a weapon. Unlike [effects](#), abilities can use the Animator and they are also synchronized across the network.

New abilities can be created by following [this example](#). The ability system is designed to be extensible so it is highly recommended that you create new abilities specific for your game.

### Priority

Abilities use a priority system when added to the Ultimate Character Locomotion component. The higher the ability is in the list, the higher priority it is. Consider this setup:

Ability	Enabled
= Interact	<input checked="" type="checkbox"/>
= Die	<input checked="" type="checkbox"/>

With this setup the [Interact](#) ability has a higher priority than the [Die](#) ability. The result of this is that the die animation would not be able to play if the character gets killed while interacting with an object. The Die ability should be above the Interact ability in order for the die animation to play correctly while the character is interacting with an object.

Ability	Enabled
= Die	<input checked="" type="checkbox"/>
= Interact	<input checked="" type="checkbox"/>

If an ability should be able to be active at the same time as another ability - such as the Speed Change ability, then the ability can override the IsConcurrent property and return true. The priority system is not used with concurrent abilities.

### API

Abilities can be started/stopped manually from any script by getting a reference to the

Ultimate Character Locomotion component and then calling the TryStartAbility or TryStopAbility method on that component. These methods require a reference to the ability that will be started/stopped and that reference can be retrieved with GetAbility.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Character;
using Opsive.UltimateCharacterController.Character.Abilities;

public class MyObject : MonoBehaviour
{
    [Tooltip("The character that should start and stop the jump
ability.")]
    [SerializeField] protected GameObject m_Character;

    /// <summary>
    /// Starts and stops the jump ability.
    /// </summary>
    private void Start()
    {
        var characterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        var jumpAbility = characterLocomotion.GetAbility<Jump>();
        // Tries to start the jump ability. There are many cases where
the ability may not start,
        // such as if it doesn't have a high enough priority or if
CanStartAbility returns false.
        characterLocomotion.TryStartAbility(jumpAbility);

        // Stop the jump ability if it is active.
        if (jumpAbility.IsActive) {
            characterLocomotion.TryStopAbility(jumpAbility);
        }
    }
}
```

## Events

When the ability starts or stops the “OnCharacterAbilityActive” event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. The following example will subscribe to this event from a new component:

```
using UnityEngine;
using Opsive.Shared.Events;
using Opsive.UltimateCharacterController.Character.Abilities;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Initialize the default values.
    /// </summary>
```

```

/// </summary>
public void Awake()
{
    EventHandler.RegisterEvent<Ability, bool>(gameObject,
"OnCharacterAbilityActive", OnAbilityActive);
}

/// <summary>
/// The specified ability has started or stopped.
/// </summary>
/// <param name="ability">The ability that has been started or
stopped.</param>
/// <param name="activated">Was the ability activated?</param>
private void OnAbilityActive(Ability ability, bool activated)
{
    Debug.Log(ability + " activated: " + activated);
}

/// <summary>
/// The GameObject has been destroyed.
/// </summary>
public void OnDestroy()
{
    EventHandler.UnregisterEvent<Ability, bool>(gameObject,
"OnCharacterAbilityActive", OnAbilityActive);
}
}

```

## Inspected Fields

### Enabled

Can the ability be activated?

### Start Type

Specifies how the ability can be started:

- *Automatic*: The ability will try to be started every update.
- *Manual*: The ability must be started with `UltimateCharacterLocomotion.TryStartAbility`.
- *Button Down*: The ability will start when the specified button is down.
- *Button Down Continuous*: The ability will continuously check for a button down to determine if the ability should start.
- *Double Press*: The ability will start when the specified button is pressed twice.
- *Long Press*: The ability will start when the specified button has been pressed for more than the specified duration.
- *Tap*: The ability will start when the specified button is quickly tapped.
- *Axis*: The ability will start when the specified axis is a non-zero value.
- *Custom*: The ability will start after a user defined starter has indicated that the ability

should start.

#### **Stop Type**

Specifies how the ability can be stopped:

- *Automatic*: The ability will try to be stopped every update.
- *Manual*: The ability must be started with `UltimateCharacterLocomotion.TryStopAbility`.
- *Button Up*: The ability will stop when the specified button is up.
- *Button Down*: The ability will stop when the specified button is down.
- *Button Toggle*: The ability will stop when the same button has been pressed again after the ability has started.
- *Long Press*: The ability will stop when the specified button has been pressed for more than the specified duration.
- *Axis*: The ability stop when the specified axis is a non-zero value.

#### **Input Names**

The button name(s) that can start or stop the ability.

#### **Long Press Duration**

Specifies how long the button should be pressed down until the ability starts/stops. Only used when the ability has a start/stop type of LongPress.

#### **Wait For Long Press Release**

Should the long press wait to be activated until the button has been released?

#### **Attribute Modifier**

Allows you to specify an attribute that the ability should start when active. One use for this is applied to the Speed Change ability to give the character stamina.

#### **State**

Specifies the name of the state that the ability should use when active.

#### **Append Item Definition Name**

Should the Item Definition name be appended to the name of the state name?

#### **Ability Index Parameter**

Specifies the value to set the Ability Index parameter to (-1 will not set the parameter).

#### **Start Audio Clip Set**

A set of AudioClips that can be played when the ability is started.

## **Stop Audio Clip Set**

A set of AudioClips that can be played when the ability is stopped.

## **Use Gravity**

Should the character use gravity while the ability is active?

## **Use Root Motion Position**

Can the character use root motion for positioning?

## **Use Root Motion Rotation**

Can the character use root motion for rotation?

## **Allow Positional Input**

Does the ability allow positional input?

## **Allow Rotational Input**

Does the ability allow rotational input?

## **Detect Horizontal Collisions**

Should the character detect horizontal collisions while the ability is active?

## **Detect Vertical Collisions**

Should the character detect vertical collisions while the ability is active?

## **Animator Motion**

A reference to the AnimatorMotion that the ability uses.

## **Allow Equipped Slots**

Specifies which slots can have the item equipped when the ability is active.

## **Allow Item Definitions**

An array of Item Definitions that are allowed to be equipped when the ability starts. Any Item Definition can be equipped if no Item Definitions are specified.

## **Immediate Unequip**

Should the items be unequipped immediately?

### **Reequip Slots**

Should the ability equip the slots that were unequipped when the ability started?

### **Ability Indicator Text**

The text that should be shown by the ability indicator when the ability can start.

### **Ability Indicator Icon**

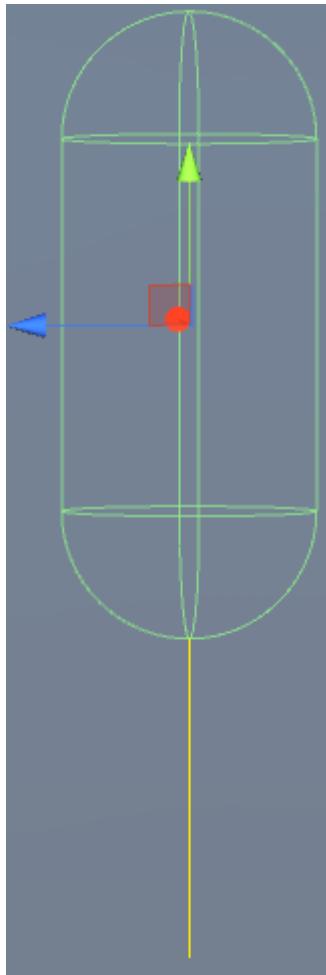
The sprite that should be drawn by the ability indicator when the ability can start.

## **Included Abilities**

### **Align To Ground**

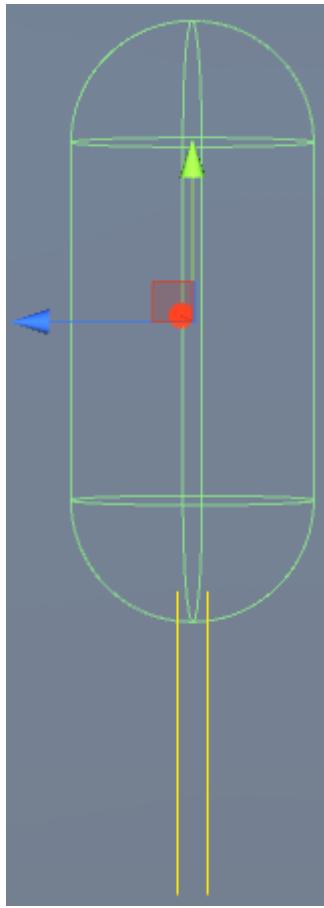
The Align To Ground ability will orient the character to the direction of the ground's normal. A cast will be performed downward to detect the rotation that the character should orient towards. The *Distance* determines the length of the cast. *Depth Offset* allows for multiple raycasts to be used which allows the result to be averaged. This is most useful for long generic characters (such as a horse) but it can also be used by humanoids to smooth the result.

If the *Depth Offset* is zero a cast in the shape of all of the colliders added to the character will be performed in the downward direction:



In this screenshot the cast is represented by the yellow line. Note that the cast is represented by a line here but a capsule cast is actually being performed because a Capsule Collider has been added to the character. The length of the cast is determined by *Distance*.

If the *Depth Offset* is not zero then two raycasts will be performed:



Similar to the last screenshot the casts are represented by the yellow line. In this situation a raycast will always be performed with the specified offset value. The character orientation will be determined by averaging the two raycast normal results.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Align To Ground ability. The ability is concurrent so the ability can be positioned anywhere within the ability list.

## Inspected Fields

### Distance

The distance from the ground that the character should align itself to.

### Depth Offset

The depth offset when checking the ground normal.

### Normalize Direction

Should the direction from the align to ground depth offset be normalized? This is useful for generic characters whose length is long.

## **Rotation Speed**

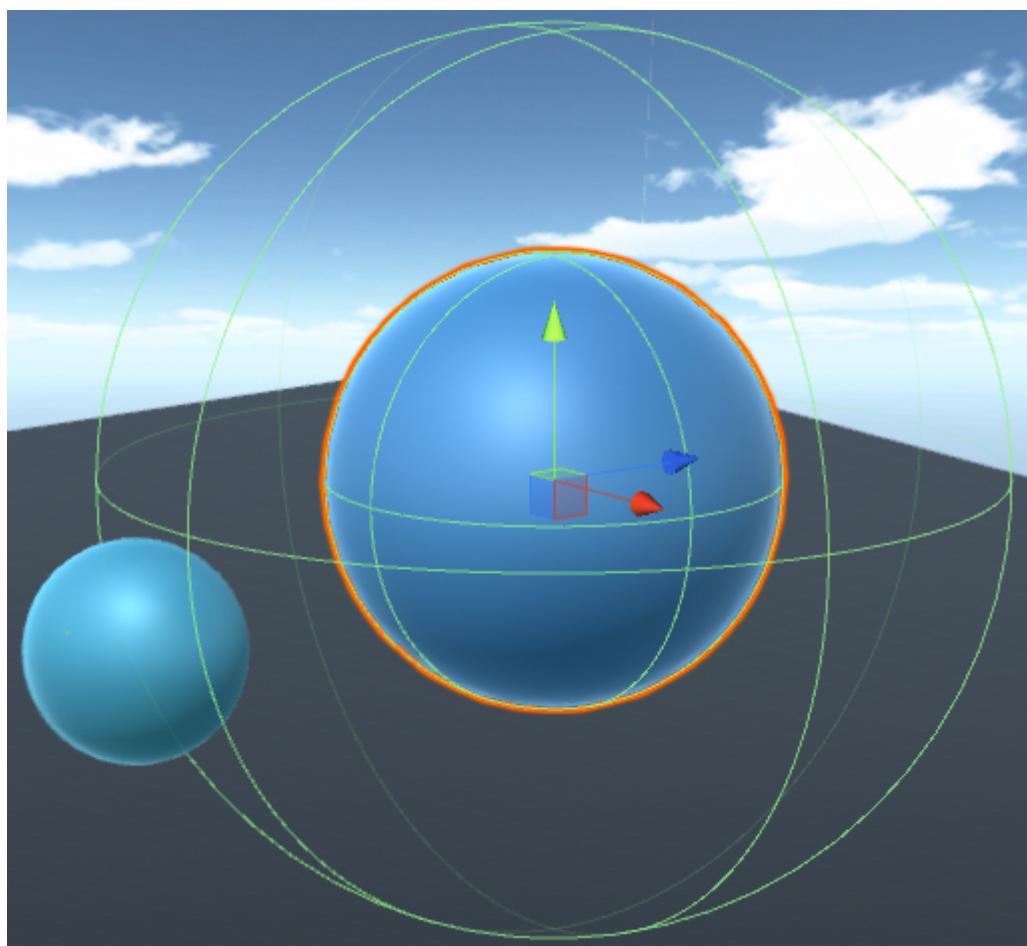
Specifies the speed that the character can rotate to align to the ground.

## **Stop Gravity Direction**

The direction of gravit that should be set when the ability stops. Set to Vector3.zero to disable.

# **Align To Gravity Zone**

The Align To Gravity Zone ability will orient the character to the direction of the gravity zones. Gravity Zones are triggers which influence the gravity direction of the character. The demo scene uses a Gravity Zone on each exterior planet:



When the ability is active and the character is within a Gravity Zone the character will reorient according to the Gravity Zone's influence. The character can be within multiple Gravity Zones and the character will be oriented towards the average of all of the Gravity Zone. Each Gravity Zone can have its own influence multiplier which in the image above it allows the larger planet to influence the character more than the smaller planet.

The Gravity Zone is and abstract class that can be inherited to return a custom gravity direction. The demo planets use a spherical Gravity Zone whose gravity direction is determined by the pivot position of the object.

## **Setup**

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Align To Gravity Zone ability. The ability is concurrent so the ability can be positioned anywhere within the ability list.
3. Add a Gravity Zone to the object that should influence the character’s gravity. The ability will automatically start when the character is within a Gravity Zone.

## **Inspected Fields**

### **Rotation Speed**

Specifies the speed that the character can rotate to align to the ground.

### **Stop Gravity Direction**

The direction of gravity that should be set when the ability stops. Set to Vector3.zero to disable.

# **Damage Visualization**

The Damage Visualization ability will play a damage animation when the character takes damage from an external source (such as a bullet or a sword) and not an internal source (such as taking fall damage). The ability includes four take damage animations but can easily be extended for other damage types.

## **Setup**

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Damage Visualization ability. This ability should be positioned near the top of the list so it will override any abilities beneath it.

## **Adding New Damage Visualization Animations**

There are two steps required in order to add new damage visualization animations:

1. The ability needs to detect which animation should be played.
2. The Animator Controller needs to have the new animations added to it.

### **Ability**

When adding new functionality we recommend subclassing the abilities so it’s easier to update when a new version of the Ultimate Character Controller is released. For this situation you’ll want to subclass the Damage Visualization ability:

```
using Opsive.UltimateCharacterController.Character.Abilities;
public class MyDamageVisualizationAbility : DamageVisualization
{
```

```
}
```

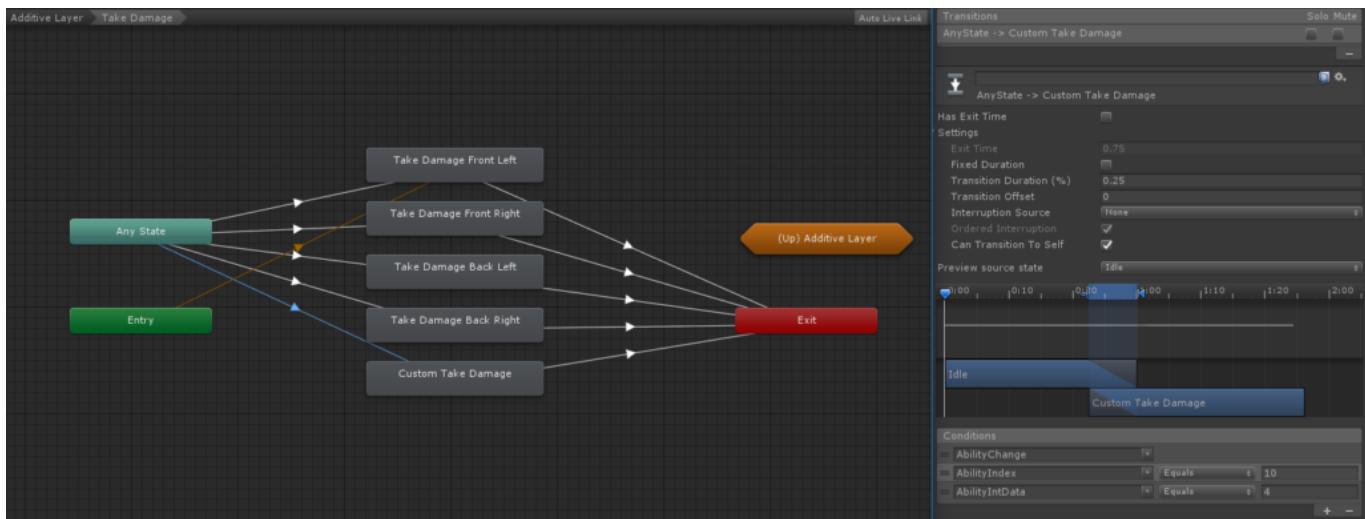
When the ability starts it will call GetDamageTypeIndex to determine the AbilityIntData Animator parameter value. Your custom damage visualization ability should override this method and return a new value. In this example we will return a value of 4 if the character takes a lot of damage.

```
Opsive.UltimateCharacterController.Character.Abilities;

public class MyDamageVisualizationAbility : DamageVisualization
{
    /// <summary>
    /// Returns the value that the AbilityIntData parameter should be
    set to.
    /// </summary>
    /// <param name="amount">The amount of damage taken.</param>
    /// <param name="position">The position of the damage.</param>
    /// <param name="force">The amount of force applied to the
    character.</param>
    /// <param name="attacker">The GameObject that damaged the
    character.</param>
    /// <returns>The value that the AbilityIntData parameter should be
    set to. A value of -1 will prevent the ability from
    starting.</returns>
    protected override int GetDamageTypeIndex(float amount, Vector3
    position, Vector3 force, GameObject attacker)
    {
        if (amount > 20) {
            return 4;
        }
        return base.GetReviveTypeIndex();
    }
}
```

## Animator

Now that the custom ability has been created it is time to modify the Animator Controller so the damage animation with a type of 4 can play. This can be done by creating a [new state](#) within the Additive Layer -> Damage Visualization substate and transitioning to it when the AbilityIntData parameter value is equal to 4:



## Inspected Fields

### Min Damage Amount

The minimum amount of damage required for the ability to start.

### Damage Visualization Complete Event

Specifies if the ability should wait for the `OnAnimatorDamageVisualizationComplete` animation event or wait the specified amount of time before interacting with the item.

# Detect Ground Ability Base

Abstract class which determines if the ground object is a valid object. Similar to the [Detect Object Ability Base](#) ability, the Ground Ability Base ability can use an *Object ID* or *Layer Mask* to determine if the ground is a valid object. If the *Object ID* is used the ground object should have the Object Identifier component with the same ID specified by the ability, similar to the Detect Object Ability Base.

## Inspected Fields

### Object ID

The unique ID value of the Object Identifier component. A value of -1 indicates that this ID should not be used.

### Layer Mask

The layer mask of the ground object.

### Ground Normal Sensitivity

The character is no longer over the ground if the dot product between the character's up direction and the ground normal is less than the sensitivity.

## Angle Threshold

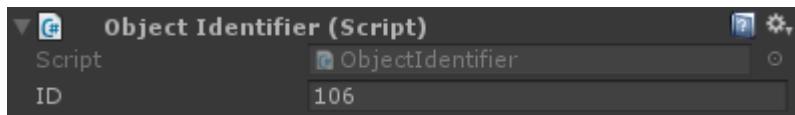
The maximum angle that the character can be relative to the forward direction of the object

# Detect Object Ability Base

The Detect Object Ability Base ability is an abstract class designed for any ability that needs another object to start. This includes abilities such as picking up an object, vaulting, climbing, interacting, etc. This ability doesn't do anything besides detect when an object can be interacted with by the character.

## Object Identifier

If the ability is performing a cast it can filter the found object based on the layer mask or an object identifier value. If an object identifier is used the target object should contain the Object Identifier component with an *ID* that matches the ability's *Object ID* value. In the screenshot below the ability's *Object ID* value would be set to 106.



## Inspected Fields

### Object Detection

The mask which specifies how the ability should detect other objects.

- *Trigger*: Use a trigger to detect if the character is near an object
- *Charactercast*: Use the character controllers to do a cast in order to detect if the character is near an object.
- *Raycast*: Use a raycast to detect if the character is near an object.
- *Spherecast*: Use a spherecast to detect if the character is near an object.
- *Customcast*: The ability will perform its own custom cast.

Multiple values can be selected, though in most cases just the Trigger or Charactercast options will be sufficient.

### Detect Layers

The LayerMask of the object or trigger that should be detected.

### Use Look Position

Should the detection method use the Look Source position? If false the character position will be used.

### Use Look Direction

Should the detection method use the Look Source direction? If false the character direction

will be used.

#### **Angle Threshold**

The maximum angle that the character can be relative to the forward direction of the object.

#### **Object ID**

The unique ID value of the Object Identifier component. A value of -1 indicates that this ID should not be used.

#### **Cast Distance**

The distance of the cast. Used if the Object Detection Mode is set to anything other than a Trigger detection mode.

#### **Cast Frame Interval**

The number of frames that should elapse before another cast is performed. A value of 0 will allow the cast to occur every frame.

#### **Cast Offset**

The offset applied to the raycast or spherecast.

#### **Trigger Interaction**

Specifies if the cast should interact with triggers.

#### **Spherecast Radius**

The radius of the spherecast.

## **Drive**

The drive ability allows the character to drive any vehicle that implements the IDriveSource interface. This ability does not actually move the vehicle. This ability will move the character to the driving location and then allow the vehicle to have control over the input.

## **Setup**

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Drive ability. The Drive ability should be positioned toward the top of the ability list.
3. Add a component to the vehicle that implements the IDriveSource interface.
4. Add the [Ability Start Location](#) indicating where the character can enter the vehicle.
5. Set the *Driver Location* on the IDriveSource component. This is the location that the character should be positioned when the character is driving the vehicle.
6. Ensure the *Update Location* on the [Ultimate Character Locomotion](#) component is

correct. The character must update within the same loop (Update or FixedUpdate) as the vehicle. The [state system](#) can be used to change the *Update Location* at runtime.

## Drive Source Interface

The IDriveSource interface allows any vehicle able to be driven by the Drive ability. This interface includes information about the vehicle such as the GameObject, Transform, and the Transform that the character should be positioned to when they are driving the vehicle.

The *AnimatorID* property of the IDriveSource interface should return a unique animator value for that vehicle. A single Drive ability can be used for any number of vehicles, and the Animator can determine which drive animation to play based on the *AnimatorID* from the IDriveSource. The drive animation state will be added onto the current vehicle's Animator ID.

The IDriveSource contains methods that indicate when the character started to enter or exit the vehicle, as well as when the character completed the vehicle entrance or exit. In the demo scene the vehicle is enabled after the character has entered vehicle, and it is disabled when the character starts to exit the vehicle.

## Animator ID

As mentioned in the previous section, a single Drive ability can be used for any number of vehicles. The vehicle *AnimatorID* value should be in an increment of 10, such as 0, 10, 20, 30, ... 110, 120, ... 1500, 1510, etc. This will allow the character's Animator to determine which vehicle the character entered and what animation state they should be in. The following are valid animation states:

- *Enter*: The character is entering the vehicle.
- *Drive*: The character is driving the vehicle.
- *Exit*: The character is exiting the vehicle.

*Enter* has a value of 0, *Drive* has a value of 1, and *Exit* has a value of 2. This value is added to the vehicle's *AnimatorID*. As an example, if your vehicle has an *AnimatorID* value of 70, when the character is entering the vehicle the character's [AbilityIntData](#) Animator parameter will have a value of 70 ( $70 + 0$ ). When the character is driving the *AbilityIntData* will be 71 ( $70 + 1$ ), and when the character is exiting the *AbilityIntData* value will be 72 ( $70 + 2$ ).

## Inspected Fields

### Teleport Enter Exit

Should the character teleport for the enter and exit animations?

### Can Use Items

Can the Drive ability use items?

## Move Speed

The speed at which the character moves towards seat position.

## Rotation Speed

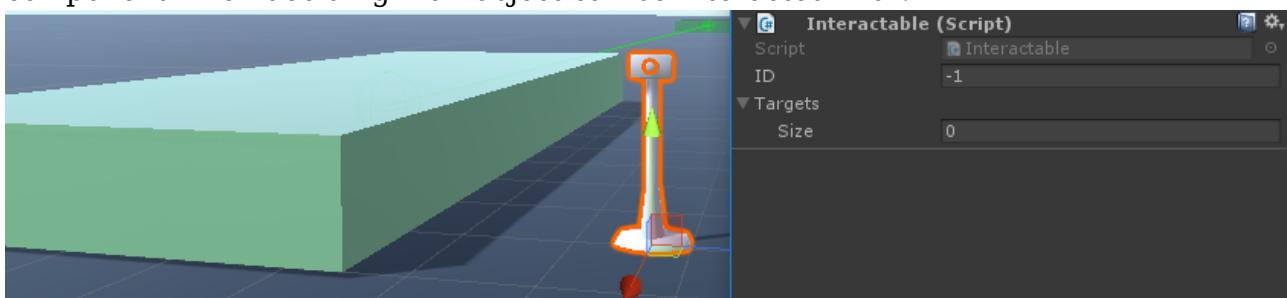
The speed at which the character rotates towards seat position.

# Interact

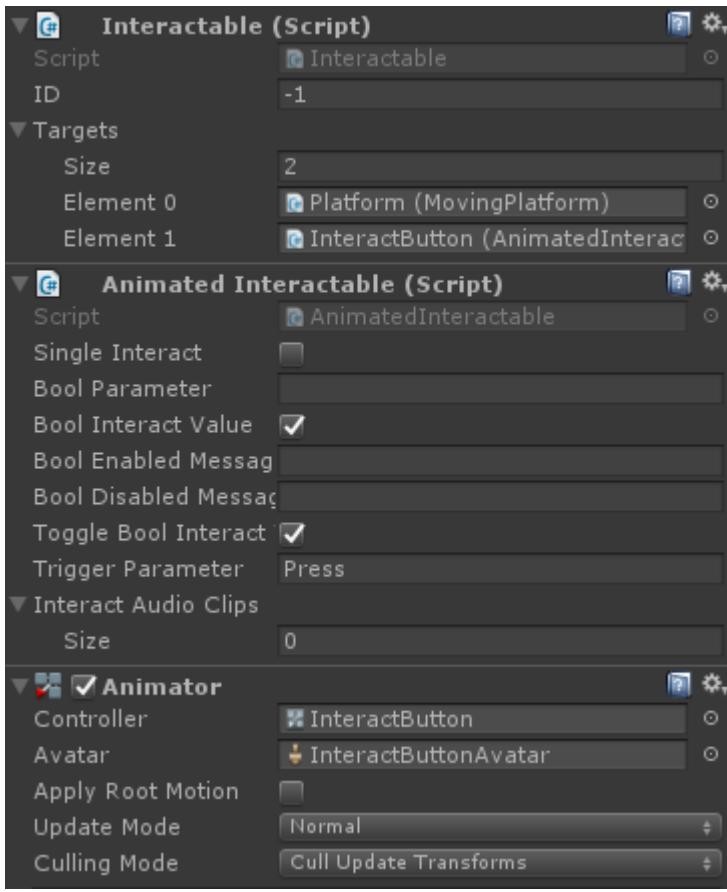
The Interact ability is a versatile ability that allows the character to interface with another object within the scene. Examples include opening a door or pressing a button. This ability is designed to be used with any other object though so is not limited to just those two examples. The Interact ability is a child of the [Detect Object Ability Base](#) so will inherit any properties from it.

## Setup

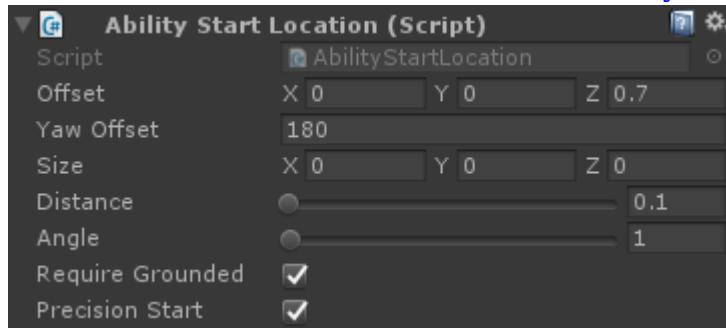
1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Interact ability.
3. Decide which object the ability should interact with. In this example we are going to use a moving platform: when the character interacts with the button it’ll start the moving platform. A [moving platform](#) is already setup for this example. Ensure the Moving Platform has *Enable On Interact* enabled.
4. Add the Interactable component to the button. The Interactable component is responsible for performing the actual interaction. The Interact ability will look for this component when deciding if an object can be interacted with.



5. The Interactable component now needs to know what object it should interact with. Set the *Targets* field to 2 so two targets can be specified. The first target will be the moving platform, and the second will be the button itself so the button press animation can be played.
6. The moving platform should be specified as the first element within the Targets array. The second element should be the button and for that we need to add a new component: the Animated Interactable component. This component will play an animation when the object is interacted with. An Animator should also be added to the button so the Animated Interactable component can play the button press animation.
7. Taking a look at the InteractButton Animator Controller we’ll see that the button press animation is started when the Press parameter is triggered. With this setup the Animated Interactable component should specify:
  - Trigger Parameter: Press

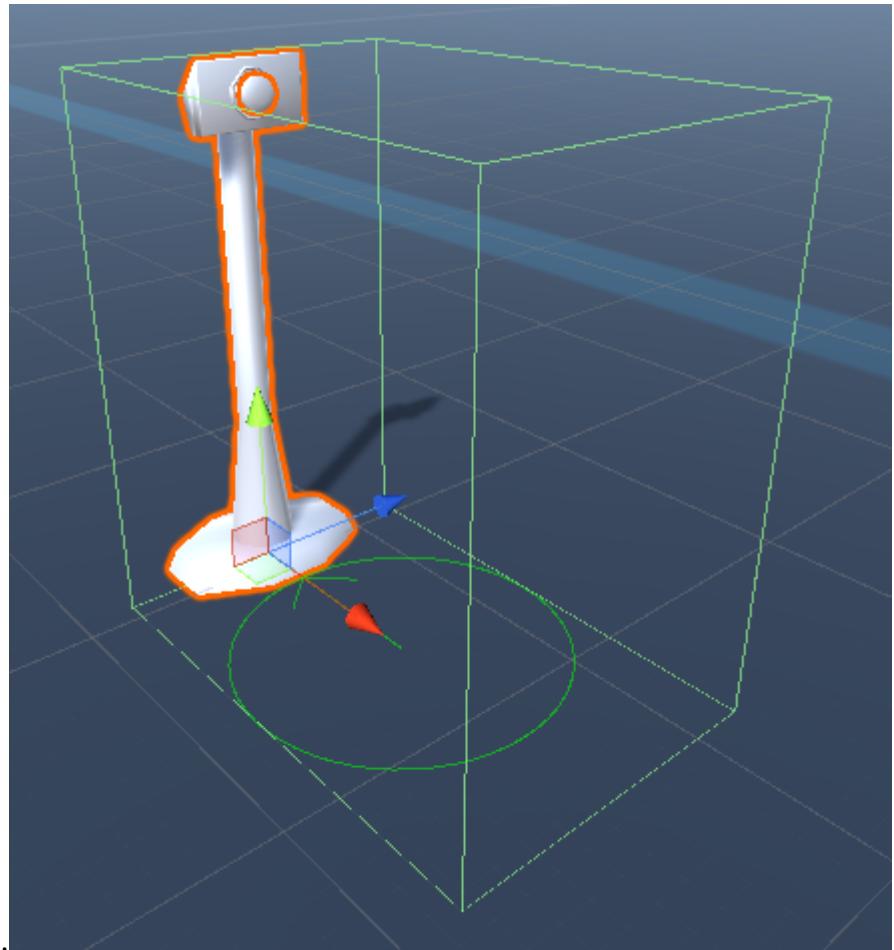


8. Now both the moving platform and the button press Interactable Targets will respond to the interact, but the ability needs to know where the character should be positioned to actually press the button. For this we can add the [Ability Start Location](#) component



to the button.

9. For this example we want the button to be able to be interacted with when the character is within a trigger. Under the *Object Detection* field of the Interact ability on the character we'll set a value of Trigger. A trigger should now be added to the button



that the ability will detect.

10. The interact object is now ready to go! When the character enters the trigger the Interact ability will be able to start. When the ability is started it'll interact with the Interactable component after the *Interact Event* has triggered. The ability will then complete after the *Interact Complete Event* has triggered.

## API

New objects can be interacted with by implementing the IInteractableTarget interface. With the IInteractableTarget the Interact/Interactable scripts do not need to be modified in order to support a new interactable object. The IInteractableTarget interface contains two methods that must be implemented:

```
/// <summary>
/// Can the target be interacted with?
/// </summary>
/// <returns>True if the target can be interacted with.</returns>
bool CanInteract();

/// <summary>
/// Interact with the target.
/// </summary>
void Interact();
```

*CanInteract* returns a bool indicating if the object can be interacted with. An object may not be able to be interacted with if it has already been interacted with (such as a chest being opened) or is currently being interacted with (such as a moving platform that is in the process of moving). *Interact* will indicate that the interaction should start.

## Inspected Fields

### Interactable ID

The ID of the Interactable. A value of -1 indicates no ID. This value is used when the character can interact with multiple objects. For example, in the demo scene the button has an Interactable ID of 1 and the chest has an ID of 2.

### Ability Int Data Value

The value of the AbilityIntData animator parameter.

### Interact Event

Specifies if the ability should wait for the OnAnimatorInteract animation event or wait the specified amount of time before interacting with the item.

### Interact Complete Event

Specifies if the ability should wait for the OnAnimatorInteractComplete animation event or wait the specified amount of time before stopping the ability.

# Pickup Item

The Pickup Item ability will play an animation which picks up the item. This animation could be picking up the item from the ground or on a table. This ability is a child of the [Detect Object Ability Base](#) so will inherit any properties from it.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Pickup Item ability. This ability should be positioned near the top of the list so another ability will not prevent the character from picking up an item.

## Inspected Fields

### Slot ID

The slot ID of the item that can be picked up. A value of -1 indicates any slot.

### Pickup Item Definitions

Specifies a list of Item Definitions that should be picked up. If the list is empty any Item Definition will trigger the animation.

### Pickup Event

Specifies if the ability should wait for the OnAnimatorPickupItem animation event or wait the specified amount of time before picking up the item.

## Pickup Complete Event

Specifies if the ability should wait for the OnAnimatorPickupComplete animation event or wait the specified amount of time before stopping the ability.

# Ride

The Ride ability works with the [Rideable ability](#) to allow an Ultimate Character Controller character to ride on top of another Ultimate Character Controller character. The classic example of this is a humanoid character riding a generic horse - they both use the same underlying locomotion controller but use different Movement Types and abilities to allow one character to ride on top of another.

To keep the animations synchronized it is important that for any active abilities on the Ride character the Rideable ability should also have those abilities. For example, if the Speed Change ability has been added to the Rideable character with a *Speed Multiplier* of 3 then the Speed Change ability on the Ride character should also have a *Speed Multiplier* of 3.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component on the character that you want to be able to ride on top of another character.
2. Add the Ride ability. This ability should be positioned near the top of the list so it will override any abilities beneath it.
3. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component on the character that you want to be able to be ridden by another character.
4. Add the Rideable ability. This ability should be positioned near the bottom of the list so other abilities will be able to control the animator.
5. Setup the [Ability Start Locations](#) on the Rideable character. These start locations are the locations that the ride character can mount onto the rideable character from. The Ride ability supports start locations on the left and right side of the Rideable character.
6. Set the *Mount Parent* on the Rideable ability. This is the object that the Ride character will be parented to after mounting.

## Inspected Fields

### Mount Event

Specifies if the ability should wait for the OnAnimatorMount animation event or wait for the specified duration before mounting to the rideable object.

### Dismount Event

Specifies if the ability should wait for the OnAnimatorDismount animation event or wait for the specified duration before dismounting from the rideable object.

## **Reequip Item After Mount**

After the character mounts should the ability reequip the item that the character had before mounting?

## **Move To Ride Offset Speed**

The speed to move the character into the ride position. This is necessary because of bug [1064826](#).

## **Parent Ride Offset**

The local position to move the character towards while riding. This is necessary because of bug [1064826](#).

# **Die**

The die ability will play a death animation when the character dies. The death animation is determined by the hit location that causes the character to die. If for example an arrow kills the player after hitting them in the chest the death animation should play a falling backwards animation. The die ability includes a forward and backwards death animation but the ability can easily be extended for other death types.

## **Setup**

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Die ability. This ability should be positioned near the top of the list so it will override any abilities beneath it.
3. No extra setup is required – when the character dies the ability will automatically start.

## **Adding New Death Animations**

There are two steps required in order to new death animations:

1. The ability needs to detect which animation should be played.
2. The Animator Controller needs to have the new animations added to it.

## **Ability**

When adding new functionality we recommend subclassing the abilities so it’s easier to update when a new version of the Ultimate Character Controller is released. For this situation you’ll want to subclass the Die ability:

```
using Opsive.UltimateCharacterController.Character.Abilities;
public class MyDieAbility : Die
{
}
```

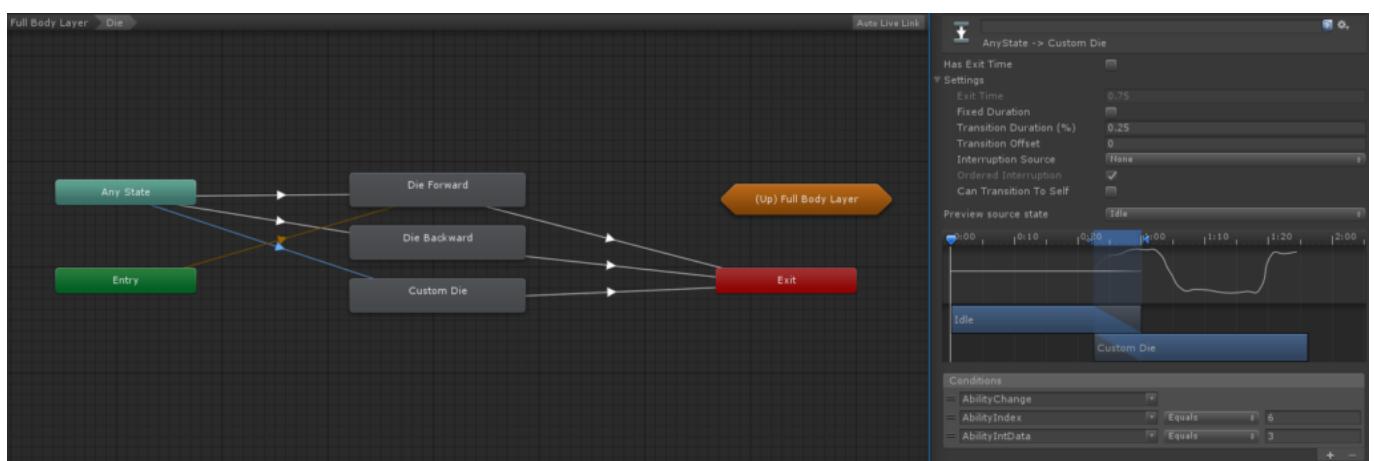
When the ability starts it will call GetDeathTypeIndex to determine the AbilityIntData Animator parameter value. Your custom die ability should override this method and return a new value. In this example we will return a value of 3 if the force is greater than 10:

```
using Opsive.UltimateCharacterController.Character.Abilities;

public class MyDieAbility : Die
{
    /// <summary>
    /// Returns the value that the AbilityIntData parameter should be
    set to.
    /// </summary>
    /// <param name="position">The position of the force.</param>
    /// <param name="force">The amount of force which killed the
    character.</param>
    /// <param name="attacker">The GameObject that killed the
    character.</param>
    /// <returns>The value that the AbilityIntData parameter should be
    set to.</returns>
    protected override int GetDeathTypeIndex(Vector3 position, Vector3
    force, GameObject attacker)
    {
        if (force.magnitude > 10) {
            return 3;
        }
        return base.GetDeathTypeIndex();
    }
}
```

## Animator

Now that the custom ability has been created it is time to modify the Animator Controller so the death animation with a type of 3 can play. This can be done by creating a [new state](#) within the Full Body Layer -> Die substate and transitioning to it when the AbilityIntData parameter value is equal to 3:



## Inspected Fields

### Camera Rotational Force

The amount of force to add to the camera when the character dies. This value will be multiplied by the magnitude of the force that killed the character.

# Fall

The fall ability will activate when the character is falling in the air. This includes after the jump ability is complete or the character is shot into the air from a moving platform.

### Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Fall ability. The Fall ability should be placed directly under the Jump ability within the ability list.
3. No extra setup is required – the Fall ability will automatically start when in the air and the jump ability isn’t active.

## Inspected Fields

### Min Fall Height

The minimum height between the ground and the character that the ability can start. Set to 0 to start at any height.

### Land Surface Impact

A reference to the Surface impact that should trigger when the character hits the ground. This impact is only triggered if the vertical velocity is less than the Min Surface Impact Velocity.

### Min Surface Impact Velocity

The minimum velocity required for the Surface Impact to play.

### Land Event

Specifies if the ability should wait for the OnAnimatorFallComplete animation event or wait the specified amount of time before ending the fall. If you’d like the Fall ability to end immediately after landing instead of playing a fall end animation the *Wait for Animation Event* toggle should be deselected and the *Duration* should be 0.

# First Person Lean

**Note:** This ability is only available with the first person perspective.

The Lean ability will rotate the first person camera to mimic the character leaning left or right. This ability does not change the actual rotation of the character. This allows the character to peak around a corner without exposing their body. The ability can use a collider to prevent the camera from intersecting with other objects as it leans.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the First Person Lean ability
3. A Capsule Collider will automatically be added under the Character/Colliders Game Object. This Capsule Collider should be adjusted to account for the amount of collision detection that the ability does. When another object enters this collider the Lean ability will reduce the amount that it leans.

## Inspected Fields

### Distance

The distance that the camera should lean.

### Tilt

The amount of tilt to apply while leaning (in degrees).

### Item Tilt Multiplier

A tilt multiplier applied to the items.

### Collider

An optional collider that can be used for collision detection and hit points.

### Collider Offset Multiplier

Optionally modify the distance that the collider leans.

### Max Collision Count

The maximum number of collisions that can be detected by the collider.

## Follow Psedu3D (2.5D) Path

**Note: This ability is only available with the third person perspective.**

The Follow 2.5D Path ability will ensure the 2.5D character is bound to the path specified within the [2.5D Movement Type](#). When the Movement Type determines the character rotation using the path object it will get the rotation at the current character position. Because the path may curve between the current location and the destination position the character's position needs to be modified to ensure it stays in the same relative position

along the path.

## Setup

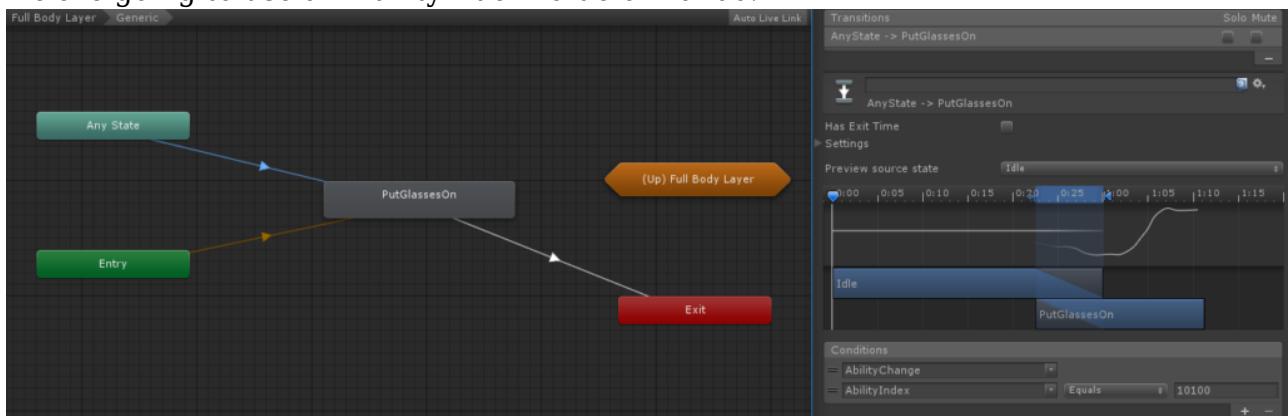
1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Follow Pseudo3D Path ability. This ability runs concurrently so it can be positioned anywhere within the ability list but it is recommended that this ability be placed at the bottom.

# Generic

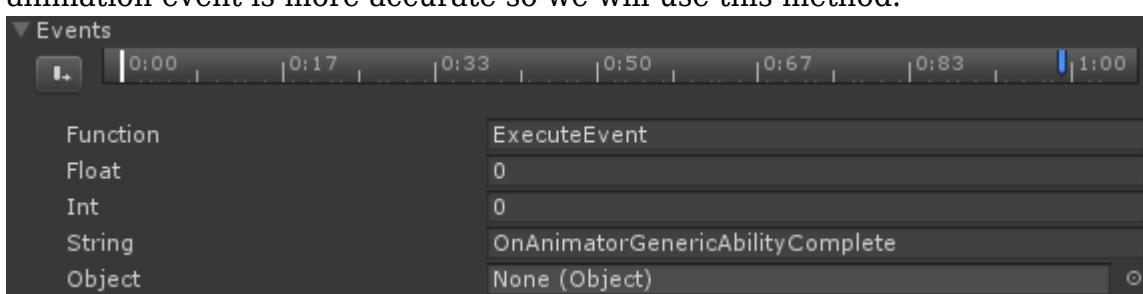
The Generic ability is an extremely versatile ability that allows you to add any animation without scripting a new ability. The ability will end after a specified duration or the OnAnimatorGenericAbilityComplete event is sent.

## Setup

1. Decide on the animations that you want to use. For this example we are going to import the [Adventure - Sample Game](#) asset and use the PutGlassesOn animation.
2. Add the animation to the Animator Controller. Create a new substate within the Full Body Layer called Generic and then create a new state that uses the PlayerPutGlassesOn motion.
3. Setup the transitions to your new state. By default the Generic ability uses an AbilityIndex of 10000 so it doesn’t get in the way of other abilities. For this example we are going to use an AbilityIndex value of 10100.



4. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
5. Add the Generic ability.
6. Set the AbilityIndex value to the same value that was set for the transition. This value should be set to 10100 in this example.
7. Decide if the ability should end based upon an [animation event](#) or a timer. An animation event is more accurate so we will use this method.



8. Decide when the ability should play. By default the ability will start when the action button is pressed.

## Inspected Fields

### Stop Event

Specifies if the ability should stop when the OnAnimatorGenericAbilityComplete event is received or wait the specified amount of time before ending the ability.

# Height Change

The Height Change ability allows the character to toggle between height changes. This ability will most commonly be used for crouching but can be used for other stances such as crawling. When the ability activates it will set the Height Animator parameter. A Height value of 0 indicates that the ability is not active. This parameter can persist when other abilities are active so those abilities know that the character is in a height change stance (this will for example allow the jump animation to start from a crouched state rather than standing up).

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Height Change ability. We recommend placing this ability near the bottom of the list so it doesn’t override any other more important abilities.

## Inspected Fields

### Concurrent Ability

Is the ability a concurrent ability? Concurrent abilities can stay activated while other abilities are activated.

### Height

Specifies the value to set the Height Animator parameter value to.

### Capsule Collider Height Adjustment

The amount to adjust the height of the CapsuleCollider by when active. This is only used if the character does not have an Animator.

### Allow Speed Change

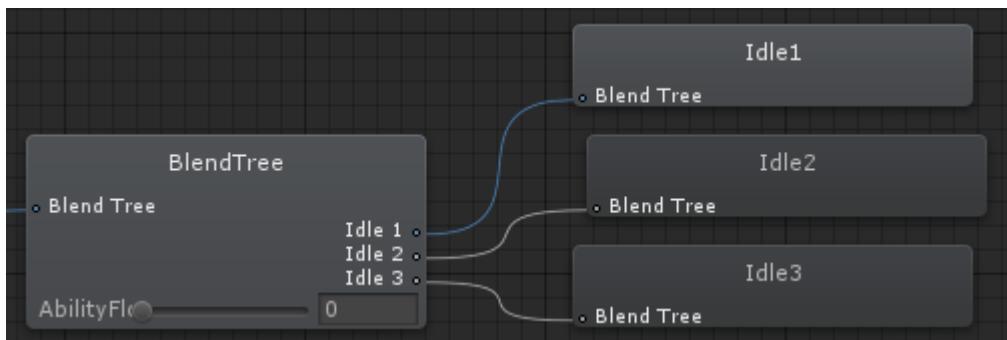
Can the SpeedChange ability run while the HeightChange ability is active?

# Idle

The idle ability will play a random animation based off of the AbilityFloatData Animator parameter. A delay can be set so unique idle animations will only play after a specified amount of time.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Idle ability. This ability should be placed at the bottom of the list so it won’t take priority over any other ability.
3. Ensure the Animator uses the AbilityFloatData while the character is in the idle state. The demo Animator Controller contains 3 idle animations within the Base Layer -> Idle -> Idle Blend Tree.



4. Set the *Max Ability Float Data Value* to the number of idle animations that your Animator contains.

## Inspected Fields

### Start Delay

Specifies how long the ability should wait until it is started. This prevents a random idle ability from starting immediately unless it should.

### Max Ability Float Data Value

The maximum AbilityFloatData Animator parameter. This number will match the number of idle animations that the Animator Controller contains.

### Random Value

Should a random int between 0 and *Max Ability Float Data Value* be used? If false the AbilityFloatData will be increased sequentially.

### Min/Max Duration

The minimum and maximum amount of time that the current AbilityFloatData value should be set until the ability chooses a new value.

# Item Equip Verifier

The Item Equip Verifier ability will ensure the character only has the necessary items equipped according to the starting abilities *AllowEquippedSlotsMask*. As an example the Interact ability will first unequip all items so the character can interact with the object. When the Interact ability is complete the Item Equip Verifier ability will then start again to equip the original items. This ability will be started manually by the controller and should not be triggered by the user.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Item Equip Verifier ability. The ability will be able to start no matter what position it is in within the ability list.

# Jump

The jump ability adds a force to the character in the vertical direction. The jump ability is only active when the character has a positive vertical velocity – when the character has a negative vertical velocity the fall ability will activate. The ability has the option of continuously adding forces for as long as the jump button is held down. It also allows the character to perform a double (or triple, or quadruple) jump by adding a vertical force each time the jump button is pressed and the ability is active.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Jump ability. This ability should be located above the fall ability.
3. No extra setup is required – when the Jump button is pressed the jump ability will start.

## Inspected Fields

### Require Grounded

Does the character need to be on the ground in order to jump?

### Min Ceiling Jump Height

Prevents the jump ability from starting if there is an object above the character within the specified distance. Set to -1 to disable.

### Force

The amount of force that should be applied when the character jumps. This force is added only once when the ability starts. Set to -1 to allow for the character to jump at any time after being airborne.

## **Sideways Force Multiplier**

A multiplier applied to the force while moving sideways. This multiplier is only applied to the initial jump force.

## **Backwards Force Multiplier**

A multiplier applied to the force while moving backwards. This multiplier is only applied to the initial jump force.

## **Frames**

The number of frames that the force is applied in. The larger the value the less of an immediate effect there will be on the initial jump force.

## **Force Damping**

Determines how quickly the jump force wears off.

## **Jump Event**

Specifies if the ability should wait for the `OnAnimatorStartJump` animation event or wait the specified amount of time before starting to jump. If you'd like the Jump ability apply a force immediately without playing a starting jump animation then the *Wait for Animation Event* toggle should be deselected and the *Duration* should be 0.

## **Jump Surface Impact**

The Surface Impact triggered when the character jumps.

## **Force Hold**

The amount of force to add per frame if the jump button is being held down continuously. This is a common feature for providing increased jump control in platform games.

## **Force Damping Hold**

Determines how quickly the jump hold force wears off.

## **Max Repeated Jump Count**

Specifies the number of times the character can perform a repeated jump (double jump, triple jump, etc). Set to -1 to allow an infinite number of repeated jumps.

## **Repeated Jump Force**

The amount of force applied when the character performs a repeated jump.

## **Repeated Jump Frames**

The number of frames that the repeated jump force is applied in. The larger the value the | 93

less of an immediate effect there will be on the initial jump force.

#### **Upward Velocity Stop Threshold**

A vertical velocity value below the specified amount will stop the ability.

#### **Recurrence Delay**

The number of seconds that the jump ability has to wait after it can start again (includes repeated jumps).

## **Move Towards**

The Move Towards ability will move the character to a specified location. This is most useful in situations where the character must be in a precise location before another ability can start. As an example the Interact ability may require the ability to be positioned in front of a door before that door can be opened, or a Climb ability may require the character to be positioned in front of the ladder before the character can mount on that ladder. This ability will be started manually by the controller and should not be triggered by the user.

The Move Towards ability works with the [Ability Start Location](#) component in order to determine the location that the character should move towards.

### **Setup**

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Move Towards ability. The ability will be able to start no matter what position it is in within the ability list.
3. On the Ultimate Character Locomotion component create a [state](#) which sets the *Motor Rotation Speed* to a larger value. In the demo scene the *Motor Rotation Speed* is set to 100.

### **Inspected Fields**

#### **Input Multiplier**

The multiplier to apply to the input vector. Allows the character to move towards the destination faster.

## **Move With Object**

The Ultimate Character Controller is moved during the FixedUpdate loop by the Kinematic Object Manager. The Kinematic Object Manager ensures the character (and other kinematic objects) move at a smooth rate while updating at a fixed timestep. If you want to have your object moved by the Kinematic Object Manager you can implement the IKinematicObject interface but this isn't always possible, especially when integrating the controller with other

assets. The Move With Object ability will allow your character to move with the target object even if that target object is updated outside of the Kinematic Object Manager loop. The target object must be updated within the Fixed Update loop otherwise there will be noticeable jitter.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Move With Object ability. The ability will be able to start no matter what position it is in within the ability list.
3. Ensure the object that the character should follow updates within the FixedUpdate loop. In the demo scene the SimplePlatform component is an example of an object that can move the character while executing outside of the Kinematic Object Manager loop.
4. Add the Kinematic Object component to the object that the character should follow. In the demo this is the same object that the SimplePlatform component was added to.
5. Ensure the script that updates within FixedUpdate executes before the Kinematic Object component. In most cases nothing needs to be changed but if the script updates after the Kinematic Object component then there will be jitter. This can be set in the [Script Execution Order](#) window.

= Opsive.UltimateCharacterController.UI.PersistentItemMonitor	700	-
= Opsive.UltimateCharacterController.UI.SlotItemMonitor	700	-
= Opsive.UltimateCharacterController.Game.KinematicObject	9000	-
= Opsive.UltimateCharacterController.Demo.DemoManager	10000	-

6. The Move With Object ability will start when it has a *Target* field, and it will stop when the *Target* field is set to null.

## Inspected Fields

### Target

The object that the character should move with.

# NavMeshAgent Movement

The NavMeshAgentMovement ability will move the character along a [Unity NavMesh](#). This ability will read the velocity from the NavMeshAgent and translate that into inputs that the Ultimate Character Controller can understand. This ability should be positioned higher than the Speed Change ability within the ability list.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the NavMeshAgent Movement ability. The NavMeshAgent component will automatically be added if it has not been already.
3. Ensure the ability is positioned above the Speed Change ability. This will allow the Speed Change to change the NavMeshAgent’s speed.
4. Ensure your scene has an [active navigation mesh](#).

## Speed

If you use [root motion](#) and adjust the speed on the NavMeshAgent component you'll find that the speed parameter doesn't change the character's speed. This is because the animation is controlling the character's speed rather than the NavMeshAgent. In order to change the speed of the character you should use the [Speed Change](#) ability which will then allow the correct animation to play based on the desired speed.

# Quick Start

The Quick Start ability allows the character to play a starting animation when the character initially starts to move. This ability will monitor the character's input values and will automatically activate when the character starts to move.

## Setup

1. Select the + button in the ability list under the "Abilities" foldout of the Ultimate Character Locomotion component.
2. Add the Quick Start ability. This ability should be positioned towards the bottom of the ability list so it does not prevent any other ability from starting.

## Inspector Fields

### Max Input Count

The number of inputs to store when determining if the ability can start. The smaller the value the less sensitive the ability is when determining if it should start.

### Speed Change Threshold

The input value which differentiates between a walk and a run. When the ability starts it will use this value to determine if a starting walk or run animation should be played.

# Quick Stop

The Quick Stop ability allows the character to play a sudden stop animation. This ability will monitor the character's input values and will automatically activate when the character stops.

## Setup

1. Select the + button in the ability list under the "Abilities" foldout of the Ultimate Character Locomotion component.
2. Add the Quick Stop ability. This ability should be positioned towards the bottom of the ability list so it does not prevent any other ability from starting.

## Inspector Fields

### **Max Input Count**

The number of inputs to store when determining if the ability can start. The smaller the value the less sensitive the ability is when determining if it should start.

### **Speed Change Threshold**

The input value which differentiates between a walk and a run. When the ability starts it will use this value to determine if a walk or run stopping animation should be played.

### **Stop Threshold**

The threshold that indicates when the character has stopped. The ability will start sooner with a larger value.

### **Required Start Success Count**

The number of times CanStartAbility must return true before the ability actually starts. This will prevent the ability from starting too soon to ensure the character is actually starting versus just doing a Quick Turn.

## **Quick Turn**

The Quick Turn ability allows the character to play an explicit animation when the character turns. This turn must be a 180 degree turn – the ability does not activate when taking 90 degree turns. This ability will monitor the character's input values and will automatically activate when the character should perform a 180 degree turn. This ability should be used with the [Adventure](#) view type.

### **Setup**

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Quick Turn ability. This ability should be positioned towards the bottom of the ability list so it does not prevent any other ability from starting.

### **Inspected Fields**

#### **Max Input Count**

The number of inputs to store when determining if the ability can start. The smaller the value the less sensitive the ability is when determining if it should start.

#### **Min Input Sqr Magnitude**

The minimum value of the input vector required for the ability to start. Note that this is the squared value of the input for better performance.

## **Speed Change Threshold**

The input value which differentiates between a walk and a run. When the ability starts it will use this value to determine if a starting walk or run animation should be played.

# **Ragdoll**

The ragdoll ability will enable or disable a ragdoll created with the Unity [Ragdoll Wizard](#). The ability can automatically start when the character dies or can be started by the environment/player.

## **Setup**

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Ragdoll ability. This ability should be placed near the top of the ability list so it will take priority over the abilities beneath it.
3. If you have not already created the ragdoll through the Ragdoll Wizard then select the Add Ragdoll Colliders button under the Editor foldout. Create the ragdoll using that wizard.
4. Ensure the Start/Stop Type values are correct and the *Start On Death* field is the correct value.

## **Inspected Fields**

### **Start On Death**

Should the ability start when the character dies?

### **Start Delay**

Specifies the delay after the ability starts that the character should turn into a ragdoll.

### **Ragdoll Layer**

The layer that the colliders should switch to when the ragdoll is active.

### **Inactive Ragdoll Layer**

The layer that the colliders should switch to when the ragdoll is inactive.

### **Camera Rotational Force**

The amount of force to add to the camera when the ability starts. This value will be multiplied by the magnitude of the force that killed the character if the ability starts when the character died.

# Restrict Position

The restrict position ability will restrict the character to a specified bounds. When the character hits the edge of that bounds they will stop similar to if they hit a wall.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Restrict Position ability. This ability can run concurrently with other abilities so the location within the list does not matter.
3. Select the type of restriction that you’d like to apply. The min/max position fields can then be adjusted based upon this restriction type.

## Inspected Fields

### Restriction

Specifies how to restrict the character’s position:

- *RestrictX*: Restricts the local X position.
- *RestrictZ*: Restricts the local Z position.
- *RestrictXZ*: Restricts the local X and Z position.

### Min/Max X Position

Specifies the minimum and maximum local X position that the character can move to if restricting the X position.

### Min/Max Z Position

Specifies the minimum and maximum local Y position that the character can move to if restricting the Y position.

# Restrict Rotation

The restrict rotation ability will restrict the character to the specified rotation.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Restrict Rotation ability. This ability can run concurrently with other abilities so the location within the list does not matter.
3. Input the angle of restriction that you’d like to apply to the character.

## Inspected Fields

## **Restriction**

The number of degrees that the character can rotate between. This restriction is applied to the local up axis. If for example you'd only like the character to move forward, left, right, or backwards in a grid pattern then this value should be set to 90. A value of 180 will allow the character to only rotate in the forward and backwards direction.

## **Offset**

Any offset that should be applied to the local y rotation. This doesn't add or remove any restriction but it does allow the character to orient towards a different starting direction.

## **Relative Look Source Rotation**

Should the local y rotation of the look source be applied to the rotation?

## **Look Source Offset**

Any offset that should be applied to the look source y rotation.

# **Revive**

The Revive ability will play a getting back up animation transitioning from the character laying on the ground. The ability can start when the character dies or be triggered by the player/environment. If the ability starts when the character dies then a delay is used which will prevent the ability from playing until the timer has elapsed. The revive ability includes a forward and backwards revive animation but the ability can easily be extended for other revive types.

## **Setup**

1. Select the + button in the ability list under the "Abilities" foldout of the Ultimate Character Locomotion component.
2. Add the Revive ability. This ability should be positioned near the top of the list so it will override any abilities beneath it.
3. Determine if the ability should start when the character dies. This is enabled by default with the *Start On Death* toggle.

## **Adding New Revive Animations**

There are two steps required in order to add new revive animations:

1. The ability needs to detect which animation should be played.
2. The Animator Controller needs to have the new animations added to it.

## **Ability**

When adding new functionality we recommend subclassing the abilities so it's easier to update when a new version of the Ultimate Character Controller is released. For this situation you'll want to subclass the Revive ability:

```

using Opsive.UltimateCharacterController.Character.Abilities;
public class MyReviveAbility : Revive
{
}

```

When the ability starts it will call GetReviveTypeIndex to determine the AbilityIntData Animator parameter value. Your custom revive ability should override this method and return a new value. In this example we will return a value of 3 if the attacker has the tag Human.

```

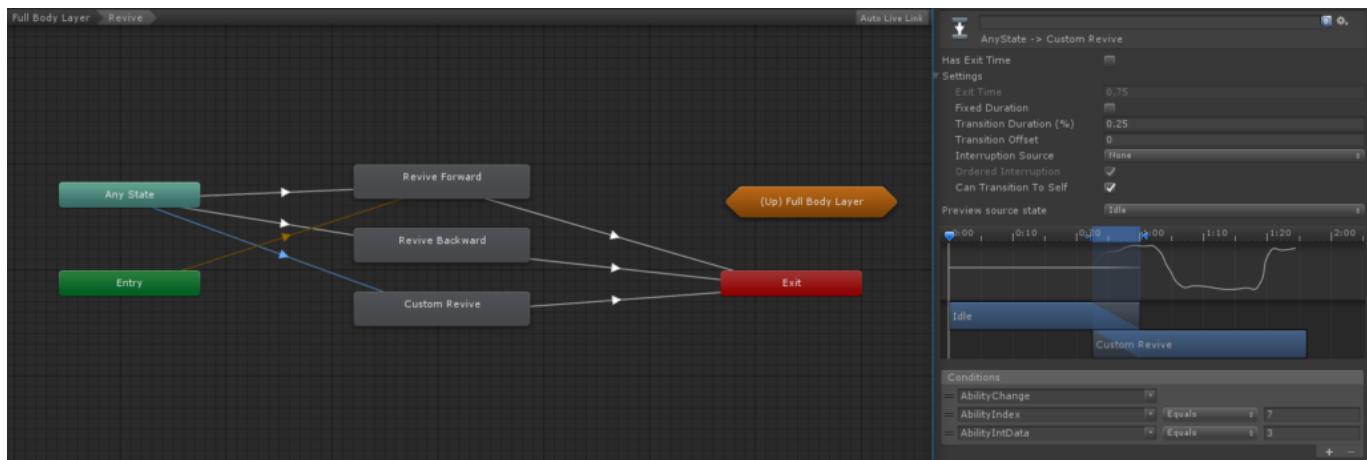
Opsive.UltimateCharacterController.Character.Abilities;

public class MyReviveAbility : Revive
{
    /// <summary>
    /// Returns the value that the AbilityIntData parameter should be
    set to.
    /// </summary>
    /// <param name="position">The position of the force.</param>
    /// <param name="force">The amount of force which killed the
    character.</param>
    /// <param name="attacker">The GameObject that killed the
    character.</param>
    /// <returns>The value that the AbilityIntData parameter should be
    set to.</returns>
    protected override int GetReviveTypeIndex(Vector3 position,
    Vector3 force, GameObject attacker)
    {
        if (string.Compare(attacker.tag, "Human") == 0) {
            return 3;
        }
        return base.GetReviveTypeIndex();
    }
}

```

## Animator

Now that the custom ability has been created it's time to modify the Animator Controller so the revive animation with a type of 3 can play. This can be done by creating a [new state](#) within the Full Body Layer -> Revive substate and transitioning to it when the AbilityIntData parameter value is equal to 3:



## Inspected Fields

### Start On Death

Should the ability start when the character dies?

### Death Start Delay

Specifies the number of seconds after the character dies that the ability should start.

## Rideable

The Rideable ability works with the [Ride ability](#) to allow another Ultimate Character Controller ride on top of it. The classic example of this is a humanoid character riding a generic horse - they both use the same underlying locomotion controller but use different Movement Types and abilities to allow one character to ride on top of another.

To keep the animations synchronized it is important that for any active abilities on the Ride character the Rideable ability should also have those abilities. For example, if the Speed Change ability has been added to the Rideable character with a *Speed Multiplier* of 3 then the Speed Change ability on the Ride character should also have a *Speed Multiplier* of 3.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component on the character that you want to be able to ride on top of another character.
2. Add the Ride ability. This ability should be positioned near the top of the list so it will override any abilities beneath it.
3. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component on the character that you want to be able to be ridden by another character.
4. Add the Rideable ability. This ability should be positioned near the bottom of the list so other abilities will be able to control the animator.
5. Setup the [Ability Start Locations](#) on the Rideable character. These start locations are the locations that the ride character can mount onto the rideable character from. The Ride ability supports start locations on the left and right side of the Rideable

character.

6. Set the *Mount Parent* on the Rideable ability. This is the object that the Ride character will be parented to after mounting.

## Inspected Fields

### Mount Parent

The location that the character should be parented to after mounting on the object.

# Stop Movement Animation

The Stop Movement Animation ability prevents the movement animation from playing when the character would run into a solid object. The move direction is predicted if the character is using root motion because with root motion the movement is applied after the animation plays.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Stop Movement Animation ability. This ability should be positioned towards the bottom of the ability list so it can override the input values.

## Inspector Fields

### Collision Check Distance

The distance that is checked to determine if there is a collision.

### Wall Glide Curve Threshold

The maximum Character Locomotion Wall Glide Curve value that the ability should start with. The higher the threshold to more the ability will ignore the Wall Glide Curve value.

# Slide

The Slide ability will apply a force to the character if the character is on a steep slope. If the slide ability is not added to the character then the character will not slide off of slopes that are too steep for the character to slide on. This ability does not affect the animations.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Slide ability. This ability can run concurrently with other abilities so the location within the list does not matter.

## Inspected Fields

### Min/Max Slide Limit

The minimum and maximum steepness (in degrees) that the character can slide.

### Multiplier

Multiplier of the ground's slide value. The slide value is determined by  $(1 - Dynamic Friction)$  of the ground's physic material.

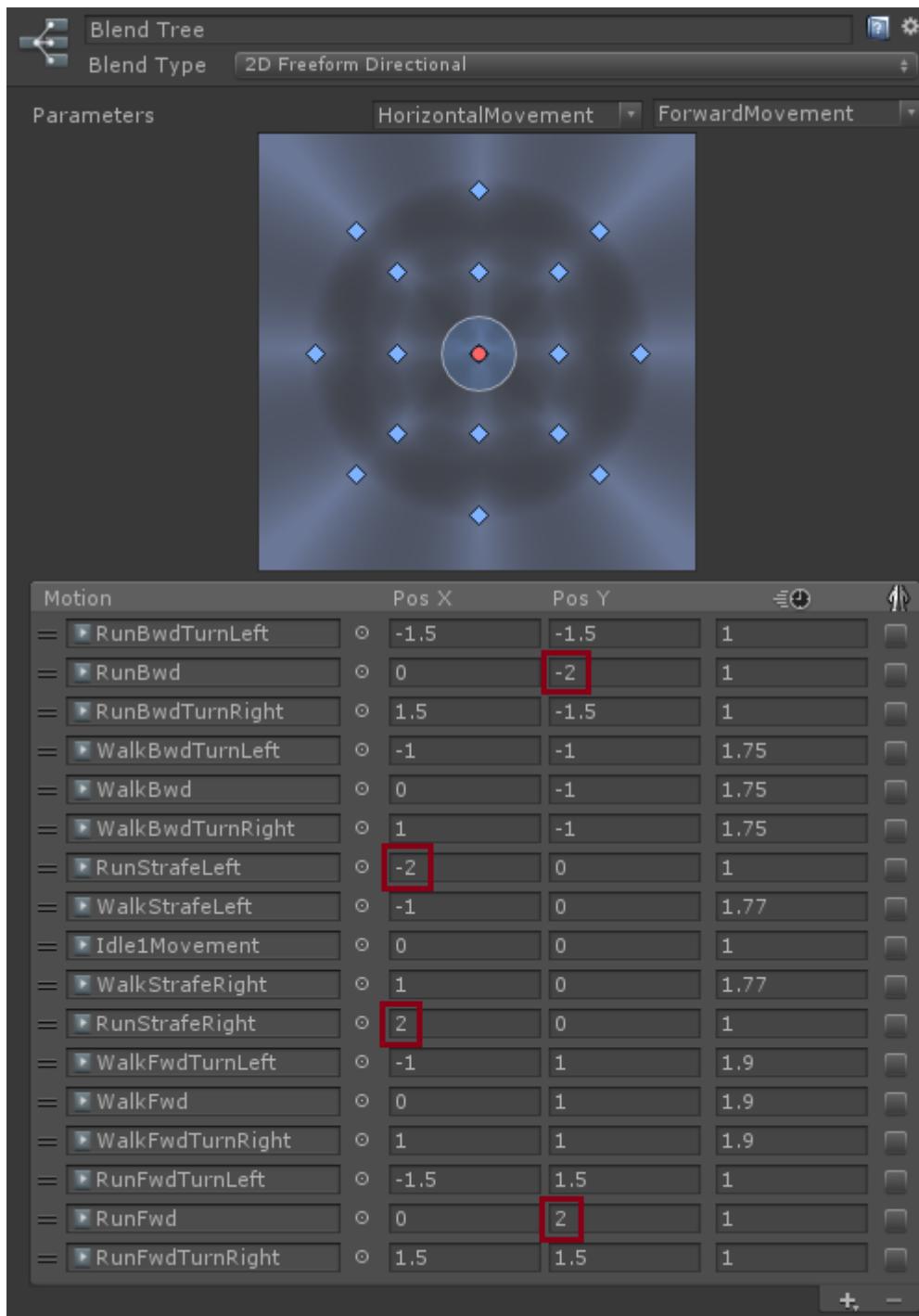
### Max Slide Speed

The maximum speed that the character can slide.

## Speed Change

The Speed Change ability allows the character to change speeds. This ability will most commonly be used for running but can be used for other speeds such as sneaking. When the ability activates it will set the Speed Animator parameter. A Speed value of 0 indicates that the ability is not active. This parameter can persist when other abilities are active so those abilities know that the character has a different speed.

When the character uses [root motion](#) the Speed Change ability doesn't actually change the character's speed - it is still up to the animation to change the character's speed. When the Speed Change ability is active it will multiply the Horizontal and Forward Movement Animator parameters which will then allow the Animator to know that it should play an animation that has a different speed. For an example take a look at the Movement blend tree within the Base Layer -> Movement substate. Notice for any of the running animations it blends based on a value of 2:



With this setup when the Horizontal or Forward Movement parameter has a value of -2 or 2 then the character will play the running animation. This can also be used in reverse - if you'd like the character to sneak then the *Speed Change Multiplier* should be set to 0.5 so a sneaking animation can be added to the blend tree when the Horizontal or Forward Movement parameters have a value of 0.5.

## Setup

1. Select the + button in the ability list under the “Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Speed Change ability. It doesn’t matter where this ability is placed within the list because it is a concurrent ability.

## Inspected Fields

### Speed Change Multiplier

The speed multiplier when the ability is active. This value will affect the Character Locomotion's Input Vector value.

### Min/Max Speed Change Value

The minimum and maximum value that the SpeedChangeMultiplier can change the InputVector value to.

### Speed Parameter

Specifies the value to set the Speed Animator parameter to.

### Require Movement

Does the ability require movement in order to stay active?

## Target Orbit

The Target Orbit ability will orbit around the target when the character moves. The ability does not move the character, rather it ensures the character is correctly rotating around the target in a constant direction. This ability can be used with the camera's [Aim Assist](#) component.

## Setup

1. Select the + button in the ability list under the "Abilities" foldout of the Ultimate Character Locomotion component.
2. Add the Target Orbit ability. This ability should be positioned towards the bottom of the ability list so it gets updated last.

## Inspector Fields

### Use Aim Assist Target

Should the ability use the aim assist target?

### Target

Specifies the target transform if the aim assist target is not used.

## Item Abilities

Item Abilities are a special type of [Ability](#) which allow for interaction with the [Items](#) that the character has. While the Item Ability is a subclass of the Ability class it has some special properties to it:

- Item abilities are added to a separate list compared to standard abilities within the Ultimate Character Locomotion.
- By default, multiple item abilities can be active at the same type. This allows the Aim item ability to still be active while the Use item ability is active.
- Item abilities do not use a priority system to determine which abilities can be active. They do use the priority system when determining which ability should set the animator parameter.

Item abilities have no restrictions in what they can or cannot do - for example they can still control the character's rotation if required. New item abilities can be created with similar steps as the [New Ability](#) topic.

## Inspected Fields

### Item State Index

Specifies the index of the Item State parameter within the Animator. A value of -1 indicates that the parameter is not used for this ability.

## Events

When the ability starts or stops the "OnCharacterItemAbilityActive" event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. The following example will subscribe to this event from a new component:

```
using UnityEngine;
using Opsive.Shared.Events;
using Opsive.UltimateCharacterController.Character.Abilities.Items;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Initialize the default values.
    /// </summary>
    public void Awake()
    {
        EventHandler.RegisterEvent<ItemAbility, bool>(gameObject,
"OnCharacterItemAbilityActive", OnItemAbilityActive);
    }

    /// <summary>
    /// The specified item ability has started or stopped.
    /// </summary>
    /// <param name="itemAbility">The item ability that has been
    started or stopped.</param>
    /// <param name="activated">Was the ability activated?</param>
    /// </summary>
    private void OnItemAbilityActive(ItemAbility itemAbility, bool
activated)
{
```

```

        Debug.Log(ability + " activated: " + activated);
    }

/// <summary>
/// The GameObject has been destroyed.
/// </summary>
public void OnDestroy()
{
    EventHandler.UnregisterEvent<ItemAbility, bool>(gameObject,
"OnCharacterItemAbilityActive", OnItemAbilityActive);
}
}

```

## Aim

The Aim item ability will place the character in an aiming state. Generally while the character is in a first person perspective you don't have to press a button in order to aim so the ability can automatically activate while in a first person perspective.

### Always Aim

If you'd like your character to always aim set the *Start Type* to Automatic and the *Stop Type* to Manual. This will keep the Aim ability active all of the time.

### Events

When the item ability starts aiming it will send the "OnAimAbilityStart" event using the built-in [Event System](#). This event has two parameters: the first indicates if the ability started, and the second indicates if the ability is being started from input. The input start parameter will be false when the ability is automatically starting while in a first person perspective. You can register and use this event with:

```

using UnityEngine;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    public void Awake()
    {
        EventHandler.RegisterEvent<bool, bool>(gameObject,
"OnAimAbilityStart", OnAim);
    }

/// <summary>
/// The Aim ability has started or stopped.
/// </summary>
/// <param name="start">Has the Aim ability started?</param>
/// <param name="inputStart">Was the ability started from
input?</param>

```

```

private void OnAim(bool aim, bool inputStart)
{
    // Ignore the event if the aim ability wasn't explicitly
    started.
    if (!inputStart) {
        return;
    }
    Debug.Log("Aim ability start: " + aim);
}

/// <summary>
/// The GameObject has been destroyed.
/// </summary>
public void OnDestroy()
{
    EventHandler.UnregisterEvent<bool, bool>(gameObject,
"OnAimAbilityStart", OnAim);
}
}

```

## Inspected Fields

### Stop Speed Change

When the Aim ability is activated should it stop the speed change ability?

### Activate In First Person

Should the ability activate when the first person perspective is enabled?

### Rotate Towards Look Source Target

Should the ability rotate the character to face the look source target?

## Block

The Block ability will play a blocking animation when another object comes into contact with the [Shield Item Action](#). When the ability starts it will play a block or a parry animation based on the type of item that was hit. If the hit item has the [Melee Weapon](#) added to it then a parry animation will play, otherwise a blocking animation will play.

A unique block/parry animation can play based on the animation that the opponent character is playing. The blocking animation may be different if the opponent is slashing down with a sword versus slashing to the side with a katana. When the Block ability starts the animator's [Item Substate Index](#) will contain a unique ID which indicates which block animation should be played. This ID is up to a nine digit number that has the following format:

AAABBBCCC

**AAA:** First three digits indicate the Item ID of the melee weapon that the opponent is attacking with. The ID is an integer so any leading zeros will be truncated.

**BBB:** Middle three digits indicate the substate index of the use state that the opponent is running.

**CCC:** Last three digits indicate the substate index for the current ability. This is retrieved from the *Impact Animator Audio State Set* on the [Shield](#).

For an example lets say that the opponent has a sword which has an [Item ID](#) of 22. The opponent's Use ability is playing an animation with an Item Substate Index of 3. Finally, the *Impact Animator Audio State Set* on the shield returns a value of 1. The resulting ID of this sequence of values will be 22003001.

## Inspected Fields

### Slot ID

The slot that should be used. -1 will block all of the slots.

### Block Item State Index

The Animator's Item State Index when the character blocks.

### Parry Item State Index

The Animator's Item State Index when the character parries.

## Drop

The Drop item ability will drop the currently equipped item. If the item has a drop prefab setup then that prefab will be instantiated and the item will fall to the ground.

## Inspected Fields

### Slot ID

The slot that should be dropped. -1 will drop all of the slots.

### No Drop Item Definitions

The Item Definitions that cannot be dropped.

### Wait For Unequip

Should the item wait to be dropped until it is unequipped?

### Drop Event

Specifies if the item should be dropped when the OnAnimatorDropItem event is received or wait the specified amount of time before dropping the item.

# Item Set

The Item Set item ability is an abstract ability base class for common [Item Set](#) operations. These operations include equip and unequip operations such as switching to the next item or toggling the equip state of an item.

## Inspected Fields

### Item Set Category ID

The category that the ability should respond to. By specifying a category ID it allows multiple Item Set abilities to be added each affecting only a single Item Set category. This for example allows you to switch primary weapons with one button mapping and then switch grenade types with another button mapping.

## Equip Next

The Equip Next item ability will equip the next available [Item Set](#) within the specified category. If the current Item Set category is at index 1 and the Equip Next ability starts it'll then try to equip the Item Set at category index 2. If this index cannot be equipped (such as if the inventory doesn't contain the item) then it'll try index 3. The ability will try every index in linear order until it finds a valid Item Set or reaches the starting index again. The Equip Next ability doesn't actually do the equip or unequip - it instead relies on the [Equip Unequip](#) ability to do the actual work.

## Equip Previous

The Equip Previous item ability will equip the previous available [Item Set](#) within the specified category. If the current Item Set category is at index 3 and the Equip Previous ability starts it'll then try to equip the Item Set at category index 2. If this index cannot be equipped (such as if the inventory doesn't contain the item) then it'll try index 1. The ability will try every index in linear order until it finds a valid Item Set or reaches the starting index again. The Equip Previous ability doesn't actually do the equip or unequip - it instead relies on the [Equip Unequip](#) ability to do the actual work.

## Equip Scroll

The Equip Scroll item ability will switch through the available [Item Sets](#) within the specified category. This ability is generally used with a mouse scrollwheel so when the wheel is scrolled forward the next item set equips, and when scrolled backward the previous item set equips. The Equip Scroll ability doesn't actually do the equip or unequip - it instead relies on the [Equip Unequip](#) ability to do the actual work.

## Inspected Fields

## Scroll Sensitivity

The sensitivity for switching between items. The higher the value the faster the scroll wheel has to scroll in order to switch items.

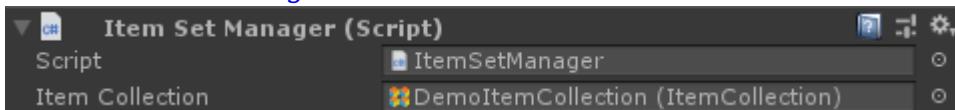
# Equip Unequip

The Equip Unequip item ability is responsible for actually doing the item equip or unequip. When any other equip item ability are activated the actual equip/unequip is done by the Equip Unequip ability. Instead of equipping or unequipping a specific slot the ability instead works by [Item Set](#). By working with ItemSets it ensures the wrong combination of weapons aren't equipped. The ability can be started manually by calling the StartEquipUnequip(ItemSetIndex) method.

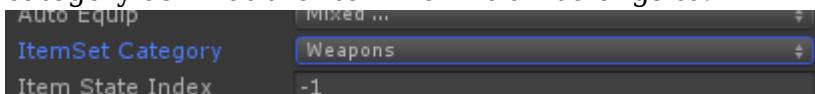
## Equip Troubleshooting

If your item isn't equipping ensure the following is set:

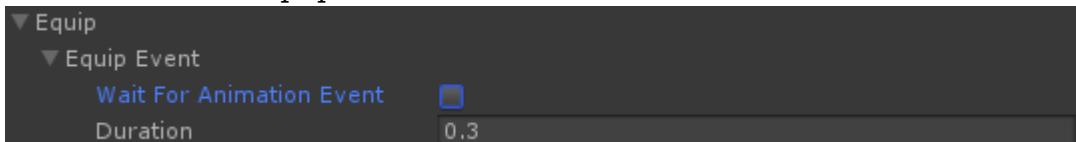
- The [Item Set Manager](#) contains a reference to the Item Collection.



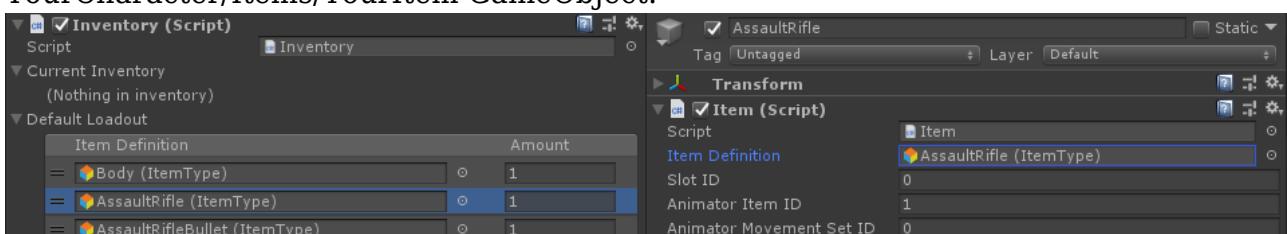
- The Equip Unequip ability exists and has an Item Set Category set to the same category as what the Item Definition belongs to.



- The [Equip Event](#) on the Item component is set correctly. If it is waiting for an animation event ensure the Animator executes the OnAnimatorItemEquip event. As a test to ensure this is not the cause you can disable the Wait for Animation Event toggle so it will instead equip after a set duration.



- Similar to above, ensure the Equip Complete Event is set correctly. If Wait for Animation Event is enabled then the ability will wait for the OnAnimatorItemEquipComplete event.
- The Item Definition is specified within the Default Loadout of the [Inventory](#) (this only applies if you want your item equipped when the character spawns).
- The Item Definition that is specified within the Default Loadout matches the Item Definition specified on your Item component. The [Item](#) component is located under the YourCharacter/Items/YourItem GameObject.



- The First/Third Person Perspective Item component specifies a Visible Object. The [First/Third Perspective Item](#) component is located under the

YourCharacter/Items/YourItem GameObject.

## API

In order to equip or unequip an item the StartEquipUnequip method can be started. The item set index parameter is the corresponding index within the Item Set Manager that should be equipped.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Character;
using Opsive.UltimateCharacterController.Character.Abilities.Items;

public class MyAIAgent : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller character.")]
    [SerializeField] protected GameObject m_Character;

    /// <summary>
    /// Equips the item.
    /// </summary>
    private void Start ()
    {
        var characterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        if (characterLocomotion != null) {
            // Equip a specific index within the ItemSetManager with
            // the EquipUnequip ability.
            var equipUnequip =
characterLocomotion.GetAbility<EquipUnequip>();
            if (equipUnequip != null) {
                // Equip the ItemSet at index 2 within the
                ItemSetManager.
                equipUnequip.StartEquipUnequip(2);
            }
        }
    }
}
```

## Inspected Fields

### Auto Equip

Mask which specifies when to auto equip a new item.

- *Always*: Always equip a picked up item.
- *Unequipped*: Equip the item if there are no items equipped.
- *OutOfUsableItem*: Equip the item if the current item has no more usable Item Definitions left.
- *NotPreset*: Equip the item if the item hasn't been added to the inventory already.

- *FirstTime*: Equip the item the first time the item has been added.

### **Equip Item State Index**

The Item State Index while equipping. This index will be set within the animator so an equip animation can play.

### **Unequip Item State Index**

The Item State Index while unequipping. This index will be set within the animator so an unequip animation can play.

### **Aim Item Substate Index Addition**

The value to add to the Item Substate Index when the character is aiming.

## **Toggle Equip**

The Toggle Equip item ability will equip or unequip the current [Item Set](#). The Toggle Equip ability works with the [Equip Unequip](#) ability to do the actual equip or unequip. While unequipping the Toggle Equip will equip the default Item Set. If no default Item Set is specified then no item set will be equipped.

### **Inspected Fields**

#### **Toggle Default Item Set On Start**

Should the default Item Set be toggled upon start? This is useful when you'd like to ensure at least one Item Set is active at all times.

## **Reload**

The Reload item ability will reload the item. There are two parts to a reload:

- The first part will take the reload amount from the inventory and add it to the item.
- The second part will wait a small amount of time after the first part to ensure the reload animation is complete before ending the ability.

### **Inspected Fields**

#### **Slot ID**

The slot that should be reloaded. If a value of -1 is specified then the ability will reload all of the equipped items.

#### **Action ID**

The ID of the ItemAction component that can be reloaded.

# Third Person Item Pullback

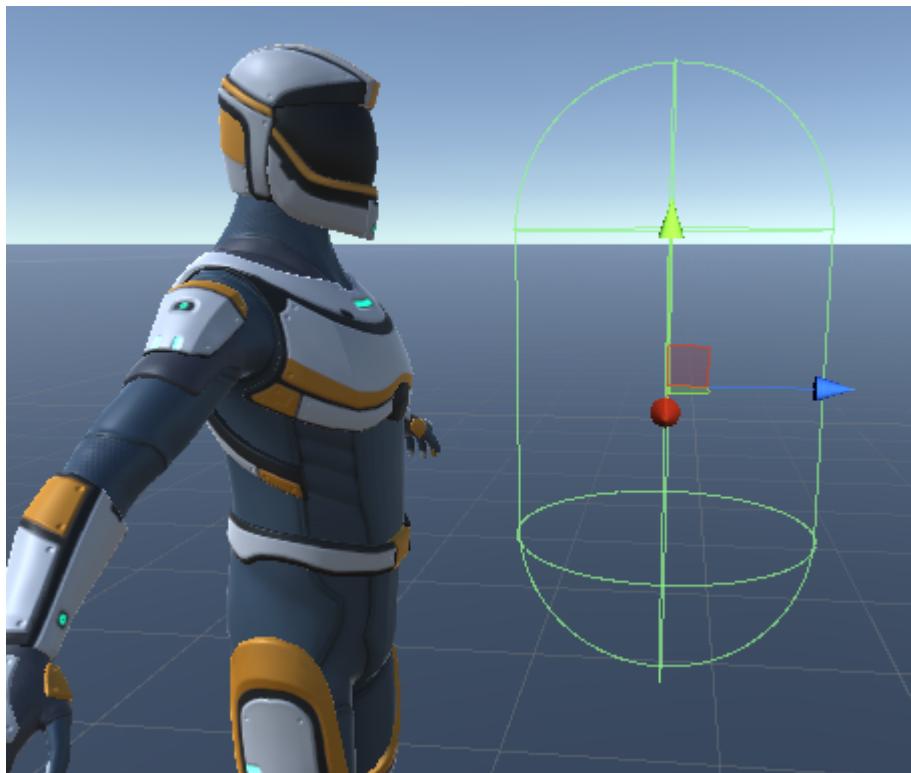
When the third person character is aiming with an object directly in front of them there's a chance that the weapon will clip the object.



The Item Pullback ability solves this by preventing the aim and use abilities from activating when an object would clip the weapon. The ability detects when the item would overlap by placing a Capsule Collider in front of the player and doing a cast to determine if anything overlaps with it.

## Setup

1. Select the + button in the ability list under the “Item Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the Third Person Item Pullback item ability.
3. The ability will automatically add an Item Pullback Collider. This collider is located under the “Character/Colliders” GameObject and should be positioned to be in the area that the character should pull back the item in.



## Inspected Fields

### **Collider**

The collider used to detect when the character is near an object and should pull back the items.

### **Collision Layers**

The layers that the collider can collide with.

### **Max Collision Count**

The maximum number of collisions that should be detected by the collider.

## Use

The Use item ability will play the use animation for the item (shooting, melee slash, grenade throw, etc). This ability is setup so multiple Use item abilities can be added to a single character depending on what type of use the character should perform. As an example an assault rifle can be fired with the left mouse button, but it can also be used as a blunt object in a melee attack. This is done by adding two different Use abilities to the character and specifying a different [Action ID](#).

## Inspected Fields

### **Slot ID**

The slot that should be used. If a value of -1 is specified then the ability will use all of the equipped items.

## Action ID

The ID of the Item Action component that can be used.

## Rotate Towards Look Source Target

Should the ability rotate the character to face the look source target?

# In Air Melee Use

The In Air Melee Use ability is a subclass of the [Use ability](#) and will allow the character to perform a melee attack while the character is in the air. The ability requires the character to have a melee weapon with a [Melee Weapon](#) component that plays an attack animation while the character is in the air.

## Setup

1. Select the + button in the item ability list under the “Item Abilities” foldout of the Ultimate Character Locomotion component.
2. Add the In Air Melee Use ability.
3. Add a new Melee Weapon component with a unique Action ID that should transition to the correct melee attack animation when the ability is active. As an example the Sword item has a Melee Weapon component with an *Action ID* of 2, and under the Use Animator Audio states it has an *Item Substate Index* of 10.
4. Assign the *Action ID* specified within step 3 to the *Action ID* field of the In Air Melee Use ability.

## Inspected Fields

### Upward Force

The amount of force that should be applied at the start of the ability.

### Upward Force Frames

The number of frames that the upward force is applied in.

### Downward Force

The amount of force that should be applied when the character starts to fall.

### Downward Force Frames

The number of frames that the downward force is applied in.

### On Grounded Substate Index

The value of the ItemSubstateIndex parameter when the character has becomes grounded after performing the use.

### **Ground Impact**

The impact effect that should be played when the character is grounded.

### **Ground Position Camera Recoil**

The amount of positional force to add to the camera when the character is grounded.

### **Ground Rotation Camera Recoil**

The amount of rotational recoil to add to the camera when the character is grounded.

## **Melee Counter Attack**

The Melee Counter Attack ability is a subclass of the [Use ability](#) and will allow the character to perform a melee counter attack after the character has blocked the opponent's attack. The ability will start after the following requirements are met:

- The character blocked a melee attack. The [Block ability](#) is required in order to block an attack.
- The character has a melee weapon.
- The opponent is within the *Attack Distance*.
- The ability is started within the *Counter Attack Timeframe* after an attack has been blocked.

The Animator's [Item Substate Index](#) value will be a unique ID that indicates which counter attack animation should be played. This ID is up to a nine digit number that has the following format:

AAABBBCCC

AAA: First three digits indicate the Item ID of the melee weapon that was blocked by the Block ability. The ID is an integer so any leading zeros will be truncated.

BBB: Middle three digits indicate the substate index of the use state that was blocked.

CCC: Last three digits indicate the substate index for the current ability. This is retrieved from the *Use Animator Audio State Set* on the [Usable Item](#).

For an example lets say that the opponent has a sword which has an [Item ID](#) of 22. The opponent's Use ability is playing an animation with an Item Substate Index of 1. Finally, the *Use Animator Audio State Set* on the melee weapon returns a value of 2. The resulting ID of this sequence of values will be 22001002.

When the Melee Counter Attack ability plays it will notify the opponent's Counter Attack Response ability if it has been added to the opponent character. The Counter Attack Response ability allows the opponent to play a full body animation in response to the counter attack.

## **Setup**

1. Select the + button in the item ability list under the "Item Abilities" foldout of the

Ultimate Character Locomotion component.

2. Add the Melee Counter Attack ability.
3. Add a new Melee Weapon component with a unique Action ID that should transition to the correct melee attack animation when the ability is active. As an example the Sword item has a Melee Weapon component with an *Action ID* of 4.
4. Assign the *Action ID* specified within step 3 to the *Action ID* field of the Melee Counter Attack ability.
5. Add the Counter Attack Response ability to any characters that can be counter attacked. This ability should be towards the top of the ability list so it has priority over other abilities.

## Inspected Fields

### Attack Distance

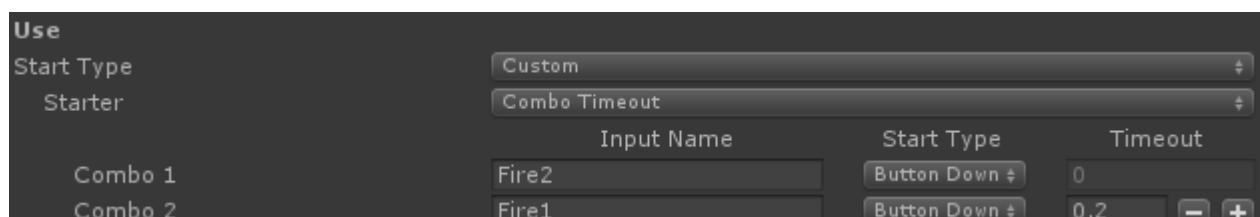
The maximum distance away from the opponent that the counter attack can start.

### Counter Attack Time Frame

The counter attack can start if the character blocked an attack within the specified amount of time.

## Ability Starter

The Ability Starter is an abstract class that allows you to decide when the ability should start. It can be thought of as being similar to Ability.CanStartAbility with an Automatic start type, except it's generic so you don't have to code the conditions within a specific ability. When the Ability Start Type of Custom is selected a dropdown will appear that allows you to select a class that inherits the AbilityStarter class.



Combo Timeout is a class that implements the Ability Starter class. In this example the ability will start after the Fire2 then Fire1 inputs have been detected with a Button Down. If Fire1 is not detected within 0.2 seconds from the Fire2 input then the combo will reset. This is a relatively simple example but the Ability Starter gives you full control over when you can start your ability.

```
/// <summary>
/// Can the starter start the ability?
/// </summary>
/// <param name="playerInput">A reference to the input
component.</param>
/// <returns>True if the starter can start the ability.</returns>
public abstract bool CanInputStartAbility(PlayerInput playerInput);
```

```

/// <summary>
/// The ability has started.
/// </summary>
public virtual void AbilityStarted()

/// <summary>
/// The ability has stopped running.
/// </summary>
public virtual void AbilityStopped()

/// <summary>
/// The object has been destroyed.
/// </summary>
public virtual void OnDestroy()

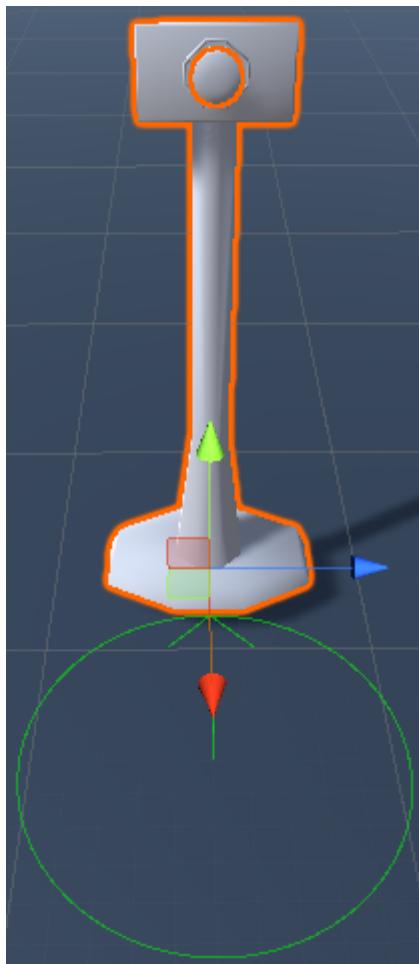
```

## Ability Start Location

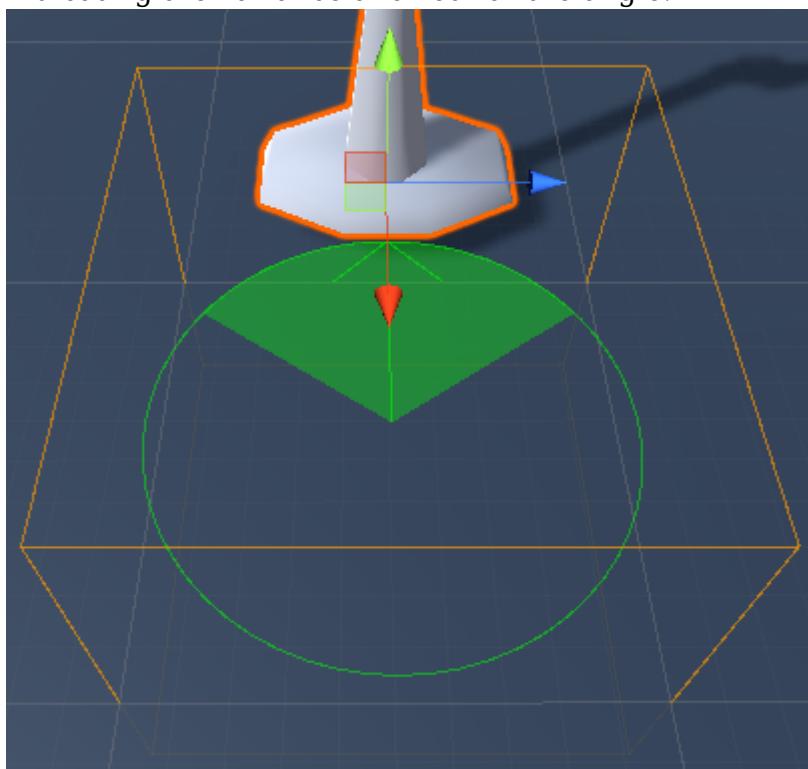
The Ability Start Location is an extremely useful component which determines where the character should move towards when starting an ability. It allows for both a positional and rotational threshold so the character doesn't have to land on the precise point in order for the ability to start. The [Move Towards](#) ability will do the actual movement of the character to the closest Ability Start Location.

### Setup

1. Select the object that is being interacted with (the button, ladder, door, etc) and add the Ability Start Location component. In the case of the [Interact](#) ability the object selected should be the same object that the Animated Interactable component is attached to.
2. Adjust the *Offset* and *Rotation Offset* fields so the gizmo in the editor will be oriented in the position and direction that the character should start the ability in. As an example for the button press interact ability the gizmo should be positioned where the character should stop and start reaching out their hand to press the button.



3. If the position or rotation is more variable for when the ability starts then the *Size* and *Angle* fields should be adjusted. Adjusting the *Size* will show an orange line indicating the location that is considered to be valid. A lighter green color will fill in the circle indicating the variance allowed for the angle.



## Ability Setup

The Ability class has a virtual method which allows the ability to return an array of Ability Start Location objects:

```
/// <summary>
/// Returns the possible AbilityStartLocations that the character can
move towards.
/// </summary>
/// <returns>The possible AbilityStartLocations that the character can
move towards.</returns>
AbilityStartLocation[] GetStartLocations()
```

By returning the possible start locations it indicates to the controller that the ability requires the character to be in a specific location before the ability can start. As an example the [Interact](#) ability will return the AbilityStartLocation on the GameObject that can be interacted with:

```
public override AbilityStartLocation[] GetStartLocations()
{
    return
m_Interactable.gameObject.GetCachedComponents<AbilityStartLocation>();
}
```

## Inspected Fields

### Offset

The offset relative to the transform that the character should move towards.

### Rotation Offset

The rotation offset relative to the transform that the character should rotate towards.

### Size

The size of the area that the character can start the ability at. A zero value indicates that the character must land on the exact offset.

### Distance

The ability can start when the distance between the start location and character is less than the specified value.

### Angle

The ability can start when the angle threshold between the start location and character is less than the specified value.

### Require Grounded

Is the character required to be on the ground?

### Precision Start

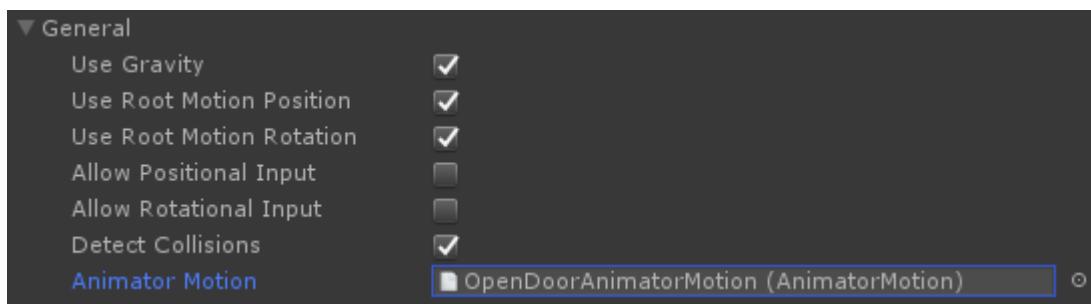
Should the ability wait to start until all transitions are complete?

# Animator Motion

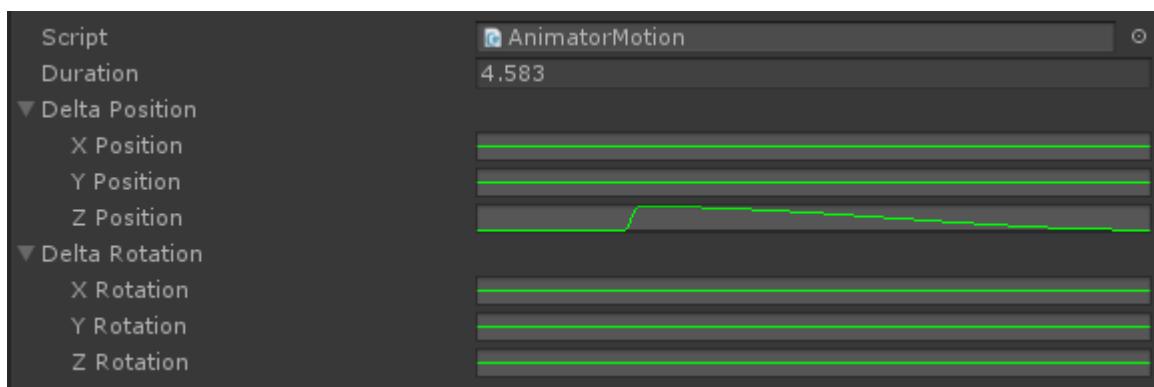
[Root motion](#) is a great way to ensure your character's movements match their animations. However, root motion isn't always available such as in the cases when the animation wasn't authored for it or the character is in a first person perspective without a full body layer. In cases like this the Animation Motion object is perfect for adding movement that matches the animation without having to actually use root motion.

If you have the option of using root motion then it is recommended that you use root motion. You'll have more accurate results while also being quicker to author. Animator Motion is useful when you don't have the option of using root motion.

A new Animator Motion object can be created under the Create -> Ultimate Character Locomotion -> Animator Motion menu. This new object can then be specified within the Animator Motion field of the ability:



When the Animator Motion is specified the ability will use this object to determine how much the character should move/rotate when the ability is active. The Animator Motion component specifies the delta X, Y, and Z position as well as the delta X, Y, and Z Euler rotation:



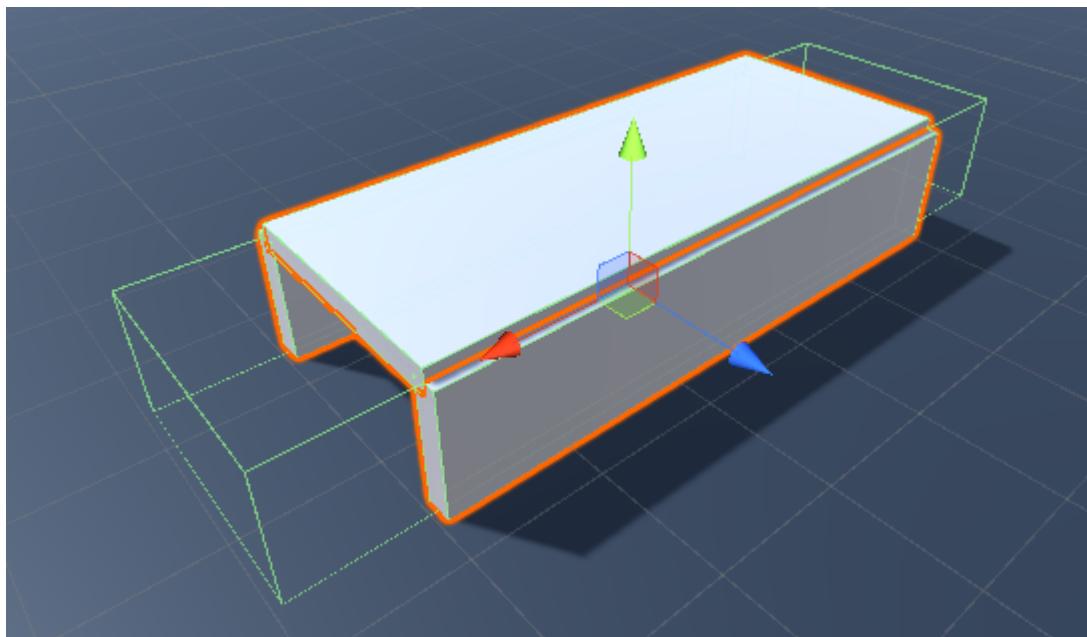
The x-axis on the curve represents the elapsed time and the y-axis represents the delta value. In the above image only the Z-axis has a delta value beginning around the 1 second mark. This means that when the ability starts the character will stay still for 1 second and then start moving along the relative Z-axis. The character will keep moving until the end of the Animator Motion duration which is 4.583 seconds after the start of the ability. No delta rotations have been specified with this Animator Motion object.

# New Ability

The ability system is designed to make it as easy as possible to add new functionality to the controller. New abilities will likely be created often so the system does as much as it can for you while also giving you complete flexibility. For this example we are going to create a crawling ability. The goals of this ability are to:

- Place the character in a crawling state when they approach a tunnel.
- Adjust the rotation torque to prevent the character from rotating as quickly.
- Prevent any items from being used when the ability is active.
- Stop the ability when the character reaches the end of a tunnel.

To get started create the object that the character will actually be crawling under within the scene. We are going to keep it basic by creating a tunnel using three scaled cubes and a trigger that designates when the character should crawl.



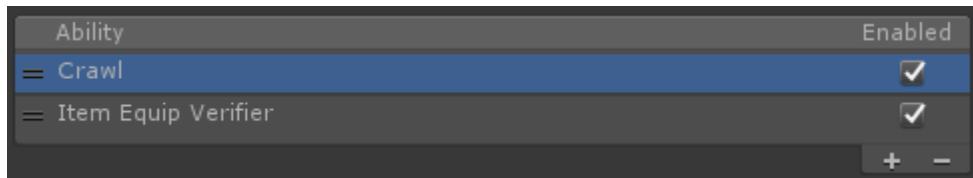
Now that the basic tunnel structure is setup let's create a new ability and derive it from the [Detect Object Ability Base](#) class. By deriving the new ability from this class it'll allow us to detect when the character gets close to a tunnel. If your ability does not need to detect an object (such as jump) then it can derive from the Ability class.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Character.Abilities;

public class Crawl : DetectObjectAbilityBase
{

}
```

With this minimal code the ability is now able to be added to the Ultimate Character Locomotion ability list. The Item Equip Verifier ability should also be added so any items can be unequipped when the ability starts.



The following ability values need to be modified:

- *Ability Index Parameter*: 101.

This value indicates the value that the Ability Index value within the Animator changes to when the ability is active. This value should be unique among all of the abilities that your character uses.

- *Object ID*: 101.

This value indicates the unique value of the object that the Detect Object Ability Base requires in order for the ability to start.

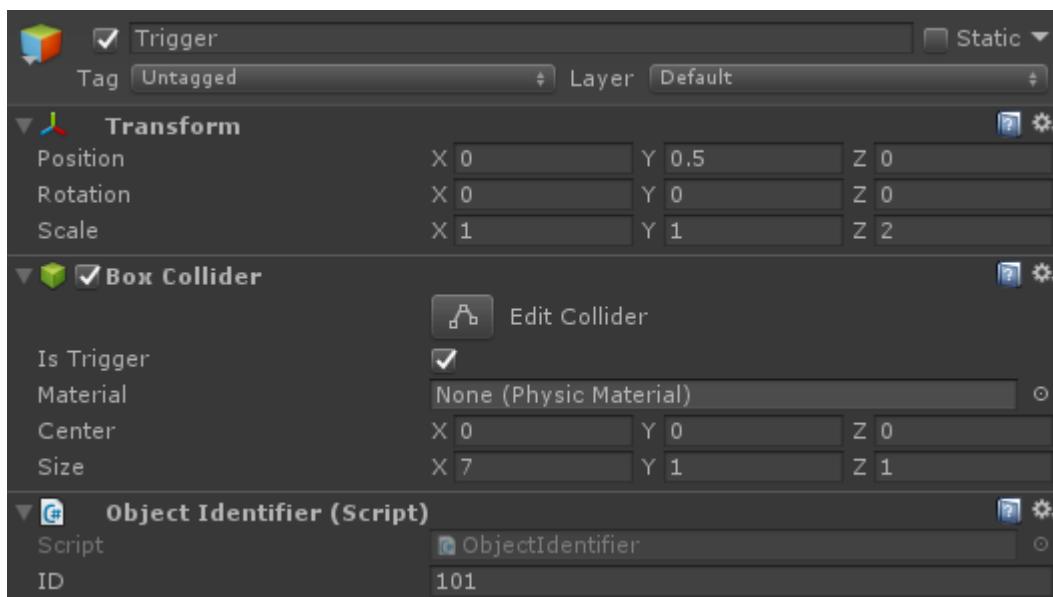
- *Object Detection*: Trigger

The ability can start if the character enters a trigger.

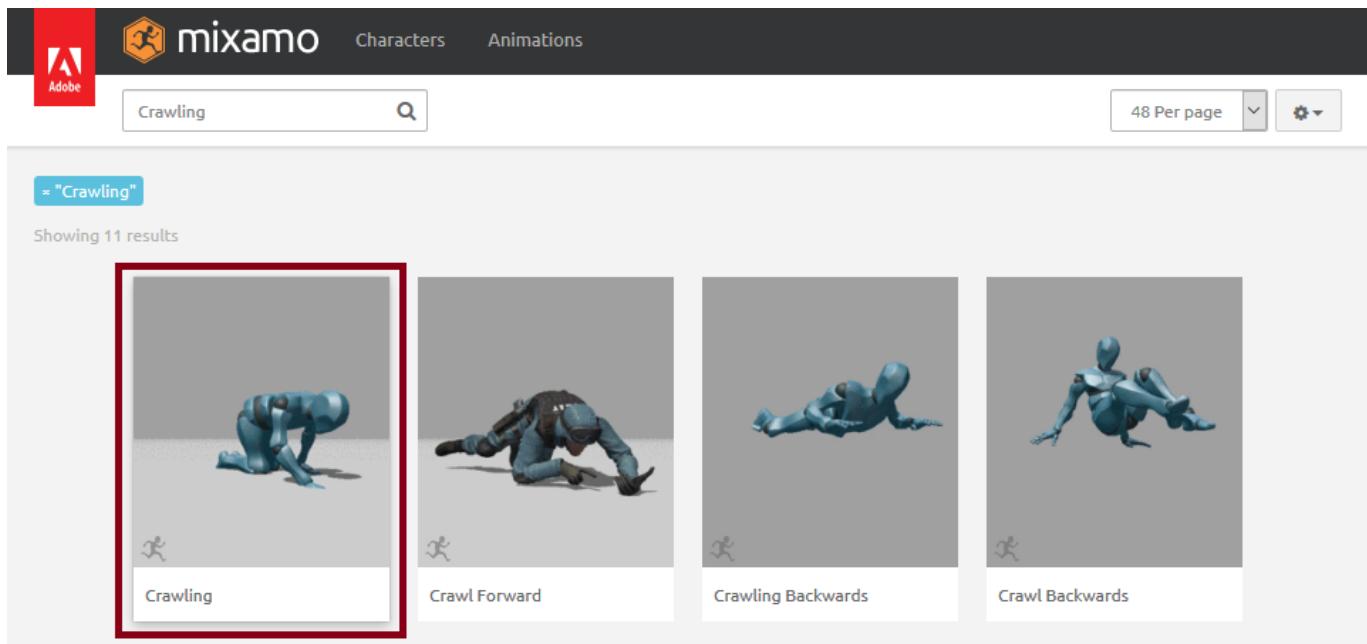
- *Allow Equipped Slots*: All slots should be deselected.

No items should be equipped when the ability is active. Reequip Slots should remain enabled so after the ability has completed any unequipped items will be reequipped.

An *Object ID* of 101 is used which indicates the object that the ability requires in order for it to start. To satisfy this requirement add the Object Identifier component to the tunnel's trigger and set the *ID* value to 101.



The next step is to add the animations to the Animator. For this do a search on mixamo.com for [Crawling](#) and download the first animation:



After you've imported the crawling animation perform the following on the fbx:

- The *Animation Type* should be set to Humanoid within the [Rig](#) tab.
- A new Idle animation clip should be created that spans frames 0 - 1 within the [Animations](#) tab.
- For both the Idle and Crawling clips the following should be performed:
  - Enable *Loop Time* and *Loop Pose* on the clip.
  - Enable *Bake Into Pose* for the Root Transform Rotation. The character controller will control the character's rotation instead of the animation.
- The Idle clip should have the following performed:
  - Enable *Bake Into Pose* for the Root Transform Position (Y and XZ). When the Idle animation plays the character should not move.

**Clips**

	Start	End
Idle	0.0	1.0
Crawling	0.0	108.0

**Idle**

Length 0.017 60 FPS

Start 0 End 1

Loop Time  Loop Pose  Cycle Offset 0 loop match

Root Transform Rotation

Bake Into Pose  Based Upon Original Offset 0 loop match

Root Transform Position (Y)

Bake Into Pose  Based Upon (at Start) Original Offset 0 loop match

Root Transform Position (XZ)

Bake Into Pose  Based Upon (at Start) Center of Mass loop match

**Clips**

	Start	End
Idle	0.0	1.0
Crawling	0.0	108.0

**Crawling**

Length 1.800 60 FPS

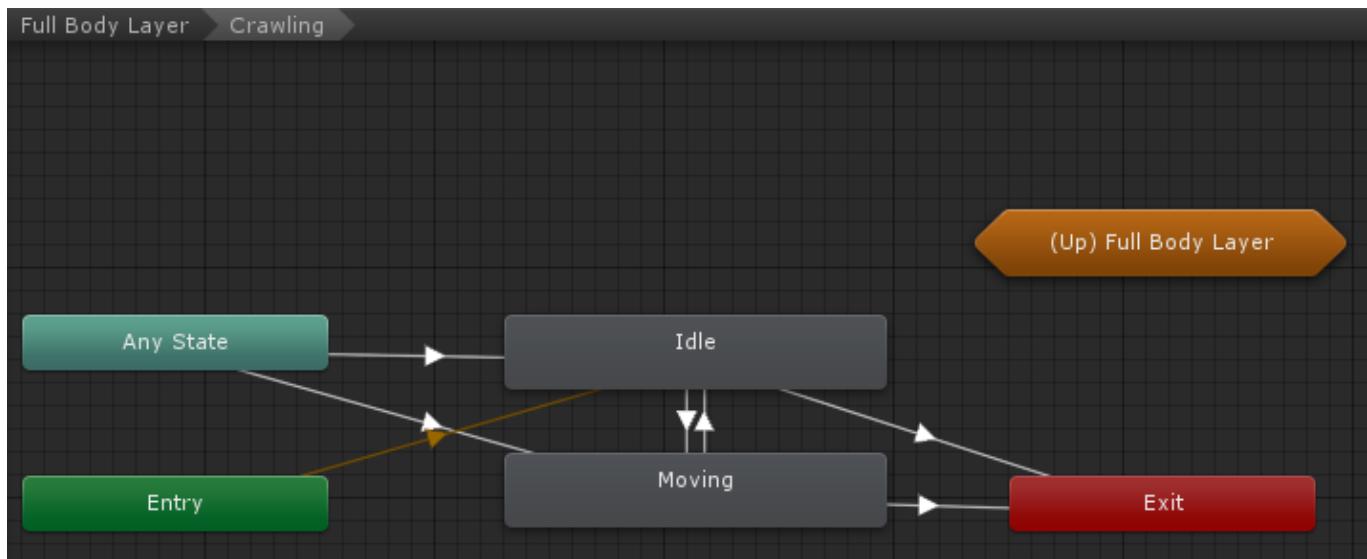
Start 0 End 108

Loop Time  Loop Pose  Cycle Offset 0 loop match

Root Transform Rotation

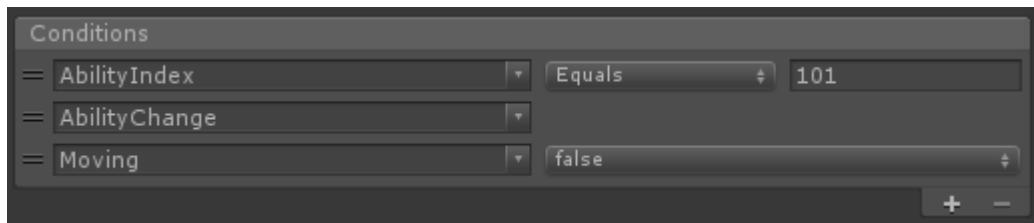
Bake Into Pose  Based Upon Original Offset 0 loop match

A new Crawling substate on the Full Body layer should be created within the Animator Controller. The Full Body layer is used because when the Crawl ability is active only the crawling animation can play. No items will be equipped during this time. Two states should be added to this substate: Idle and Crawl.



The transition from Any State to Idle and Moving should contain three conditions:

- *AbilityIndex*: Equals 101.  
The 101 value was specified when the ability was added to the Ultimate Character Locomotion component.
- *AbilityChange*  
This parameter is triggered when the ability is enabled or disabled.
- *Moving*: True (for the Moving state), False (for the Idle state).  
This parameter indicates whether or not the character is moving.



The transition between Idle and Moving only contains one condition:

- *Moving*: True (Idle -> Moving state), False (Moving -> Idle state)

The final transition to the Exit node contains one condition:

- *AbilityIndex*: NotEqual 101  
This condition will be true when the ability is no longer active.

After the Animator has been setup hit play within Unity and the ability should activate when the character enters the trigger. One of the first things that you'll notice is that the ability does not deactivate when the character leaves the trigger so this will be the first thing that we add to the Crawl ability.

Abilities, similar to MonoBehaviours, can implement the `OnTriggerExit` method and it is here that we'll want to stop the ability as soon as the character leaves the trigger:

```

/// <summary>
/// The character has exited a trigger.
/// </summary>

```

```

    /// <param name="other">The trigger collider that the character
    exited.</param>
    public override void OnTriggerExit(Collider other)
    {
        // The detected object will be set when the ability starts and
        // contains a reference to the object that allowed the ability to start.
        if (other.gameObject == m_DetectedObject) {
            StopAbility();
        }

        base.OnTriggerExit(other);
    }

```

The character should now stop the Crawl ability when the character leaves the trigger.

The next requirement of this ability is that the character turns slower while the ability is active. In order to achieve this we could set the rotation speed on the Ultimate Character Locomotion component but that doesn't make for as good of an example so for this we are going to do it via code. If you look at the API for the Ability class you'll see two methods that look promising:

### **UpdateRotation**

Update the character's rotation values.

### **ApplyRotation**

Verify the rotation values. Called immediately before the rotation is applied.

UpdateRotation is generally used for adding new rotations, while ApplyRotation is used to verify that the rotations are valid. Since we are not adding any new rotation (only restricting the existing rotation) our changes should go within ApplyRotation:

```

    [Tooltip("The maximum number of degrees the character can
rotate.")]
    [SerializeField] protected float m_MaxRotationAngle = 0.5f;

    /// <summary>
    /// Verify the rotation values. Called immediately before the
    rotation is applied.
    /// </summary>
    public override void ApplyRotation()
    {
        var angle = Quaternion.Angle(Quaternion.identity,
m_CharacterLocomotion.Torque);
        if (angle > m_MaxRotationAngle) {
            m_CharacterLocomotion.Torque =
Quaternion.Slerp(Quaternion.identity, m_CharacterLocomotion.Torque,
m_MaxRotationAngle / angle);
        }
    }

```

```
}
```

The Torque value from the CharacterLocomotion component indicates the amount that the character is going to rotate. If this value is greater than the max rotation angle then the slerp method will limit the rotation.

The UpdateRotation and ApplyRotation methods are used for the rotation and there are similar methods for position:

#### **UpdatePosition**

Update the character's position values.

#### **ApplyPosition**

Verify the position values. Called immediately before the position is applied.

The *ControllerLocomotion.MoveDirection* property contains the amount that the character is going to move.

The last requirement for this ability is that the items shouldn't be able to be interacted with while the ability is active. The ShouldBlockAbilityStart method can be used for this:

```
/// <summary>
/// Called when another ability is attempting to start and the
current ability is active.
/// Returns true or false depending on if the new ability should
be blocked from starting.
/// </summary>
/// <param name="startingAbility">The ability that is
starting.</param>
/// <returns>True if the ability should be blocked.</returns>
public override bool ShouldBlockAbilityStart(Ability
startingAbility)
{
    return startingAbility is ItemAbility;
}
```

ShouldBlockAbilityStart is called when another ability tries to start. Any active ability can prevent the ability from starting and in this case the Crawl ability will stop the ability if it is an ItemAbility (Equip, Unequip, Use, etc). With this change the Crawl ability satisfies all of our requirements and can now be used.

As a next step we recommend taking a look at the Ability API and existing abilities included within the controller - the ability system is extremely powerful and makes it convenient for adding new functionality without having to change the core classes. The complete Crawl ability is pasted below:

```
using UnityEngine;
using Opsive.UltimateCharacterController.Character.Abilities;
```

```

using Opsive.UltimateCharacterController.Character.Abilities.Items;

/// <summary>
/// Places the character in a crawling state.
/// </summary>
public class Crawl : DetectObjectAbilityBase
{
    [Tooltip("The maximum number of degrees the character can
rotate.")]
    [SerializeField] protected float m_MaxRotationAngle = 0.5f;

    /// <summary>
    /// Verify the rotation values. Called immediately before the
rotation is applied.
    /// </summary>
    public override void ApplyRotation()
    {
        var angle = Quaternion.Angle(Quaternion.identity,
m_CharacterLocomotion.Torque);
        if (angle > m_MaxRotationAngle) {
            m_CharacterLocomotion.Torque =
Quaternion.Slerp(Quaternion.identity, m_CharacterLocomotion.Torque,
m_MaxRotationAngle / angle);
        }
    }

    /// <summary>
    /// Called when another ability is attempting to start and the
current ability is active.
    /// Returns true or false depending on if the new ability should
be blocked from starting.
    /// </summary>
    /// <param name="startingAbility">The ability that is
starting.</param>
    /// <returns>True if the ability should be blocked.</returns>
    public override bool ShouldBlockAbilityStart(Ability
startingAbility)
    {
        return startingAbility is ItemAbility;
    }

    /// <summary>
    /// The character has exited a trigger.
    /// </summary>
    /// <param name="other">The trigger collider that the character
exited.</param>
    public override void OnTriggerExit(Collider other)
    {
        // The detected object will be set when the ability starts and
    }
}

```

```

contains a reference to the object that allowed the ability to start.
    if (other.gameObject == m_DetectedObject) {
        StopAbility();
    }

    base.OnTriggerExit(other);
}
}

```

## Effects

Effects can be thought of as lightweight abilities. Effects allow for extra camera/item movements that are applied to the character. Examples of an effect include an earthquake shake or boss stomp. Effects do not affect the Animator and are not synchronized over the network. For anything more involved an [ability](#) should be used instead.

### API

Effects can be started/stopped manually from any script by getting a reference to the Ultimate Character Locomotion component and then calling the TryStartEffect or TryStopEffect method on that component. These methods require a reference to the effect that will be started/stopped and that reference can be retrieved with GetEffect.

```

using UnityEngine;
using Opsive.UltimateCharacterController.Character;
using Opsive.UltimateCharacterController.Character.Effects;

public class MyObject : MonoBehaviour
{
    [Tooltip("The character that should start and stop the earthquake effect.")]
    [SerializeField] protected GameObject m_Character;

    /// <summary>
    /// Starts and stops the earthquake effect.
    /// </summary>
    private void Start()
    {
        var characterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        var earthquakeEffect =
characterLocomotion.GetEffect<Earthquake>();
        // Tries to start the earthquake effect. There are many cases
        // where the effect will not start,
        // such as if it isn't enabled or if CanStartEffect returns
        false.
        characterLocomotion.TryStartEffect(earthquakeEffect );

        // Stop the effect if it is active.
    }
}

```

```

        if (earthquakeEffect.IsActive) {
            characterLocomotion.TryStopEffect(earthquakeEffect);
        }
    }
}

```

## New Effect

Effects can be created relatively easily by overriding a couple of methods. The basic flow for the effect life cycle is:

### **EffectStarted**

The controller has started the effect.

### **Update**

The effect is active and the controller calls the Update method to update the effect.

### **EffectStopped**

The controller has stopped the effect.

## Example

The following effect will shake the camera for 3 seconds, after which the StopEffect method is called to stop the effect. This example uses the [Scheduler](#) to stop the effect after the event has elapsed.

```

using Opsive.UltimateCharacterController.Character.Effects;
using Opsive.UltimateCharacterController.Game;
using Opsive.UltimateCharacterController.Motion;

public class MyEffect : Effect
{
    protected override void EffectStarted()
    {
        base.EffectStarted();

        Scheduler.Schedule(3, StopEffect);
    }

    public override void Update()
    {
        m_CameraController.AddPositionalForce(SmoothRandom.GetVector3Centered(
1));
        m_CameraController.AddRotationalForce(SmoothRandom.GetVector3Centered(
1));
    }
}

```

# Included Effects

## Boss Stomp

The Boss Stomp effect will move the camera downward similar to how a large boss would shake the camera as they are stomping around on the ground. The effect can be repeated a set number of times and then will stop.

### Setup

1. Select the + button in the effect list under the “Effects” foldout of the Ultimate Character Locomotion component.
2. Add the Boss Stomp effect.
3. Specify the direction, strength, and the number of times the effect should stomp. A *RepeatCount* value of -1 will play the effect until the effect is stopped or disabled.

### Inspected Fields

#### **Positional Stomp Direction**

The direction to apply the positional force.

#### **Positional Strength**

The strength of the positional boss stomp.

#### **Rotational Stomp Direction**

The direction to apply the rotational force.

#### **Rotational Strength**

The strength of the rotational boss stomp.

#### **Repeat Count**

The number of times the stomp effect should play. Set to -1 to play the effect until the effect is stopped or disabled.

#### **Repeat Delay**

The delay until the stomp plays again.

## Play Audio Clip

The Play Audio Clip Effect will play the audio clip specified within the Audio Clip Set. The effect will end after the clip has finished playing.

## **Setup**

1. Select the + button in the effect list under the “Effects” foldout of the Ultimate Character Locomotion component.
2. Add the Play Audio Clip effect.
3. Specify the clips that you want to be played within the Audio Clips list. At least one clip is required for the ability to start.

## **Inspected Fields**

### **Audio Clip Set**

A set of AudioClips that can be played when the effect is started.

# **Shake**

The shake effect can shake the camera, item, or character based on a force magnitude. The effect will stop after the specified duration.

## **Setup**

1. Select the + button in the effect list under the “Effects” foldout of the Ultimate Character Locomotion component.
2. Add the Shake effect.
3. No values need to immediately be modified – the default should be a good starting point.

## **Inspected Fields**

### **Target**

Specifies which objects to apply the shaking force to.

### **Force**

The amount of force to apply to the shake.

### **Smooth Horizontal Force**

Should a smooth horizontal force be added? If false a random force between 0 and Force.X will be used.

### **Vertical Force Probability**

Specifies the probability that a vertical force will be applied.

### **Fade Out Duration**

The amount of time that it takes for the effect to fade out.

### **Positional Factor**

Exaggerates or reduces the positional force imposed.

### **Rotational Factor**

Exaggerates or reduces the rotational force imposed.

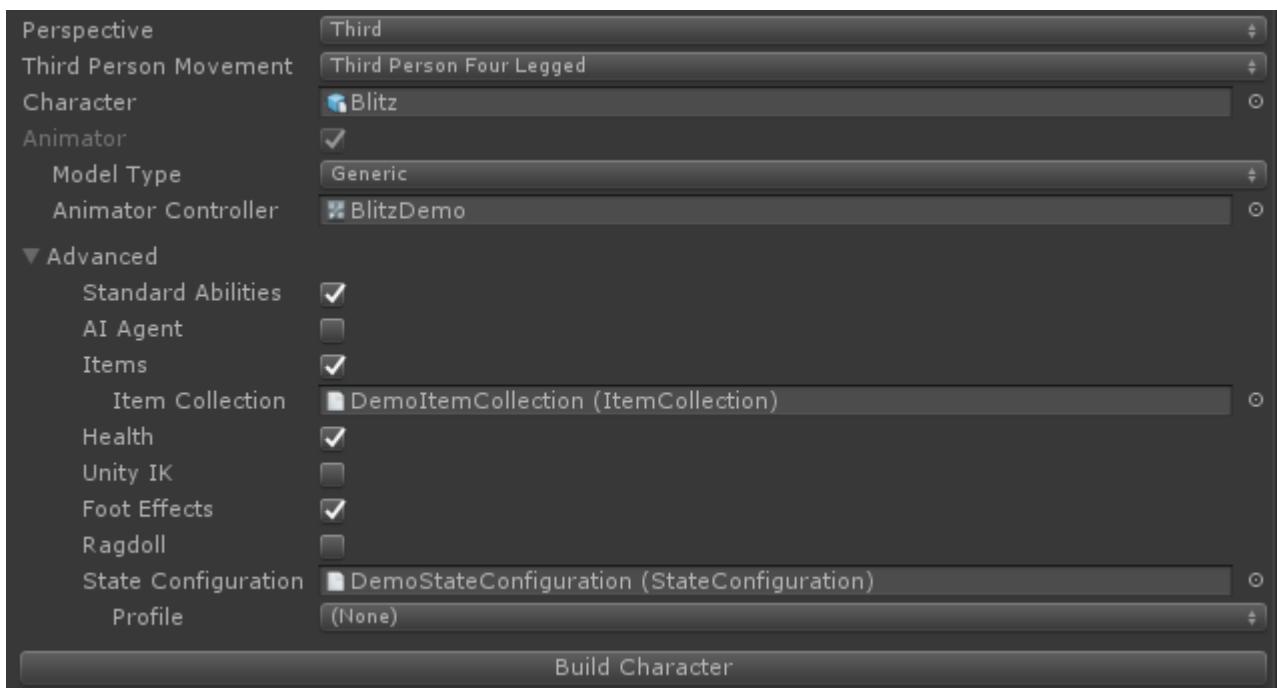
### **Duration**

The number of seconds that the effect will last.

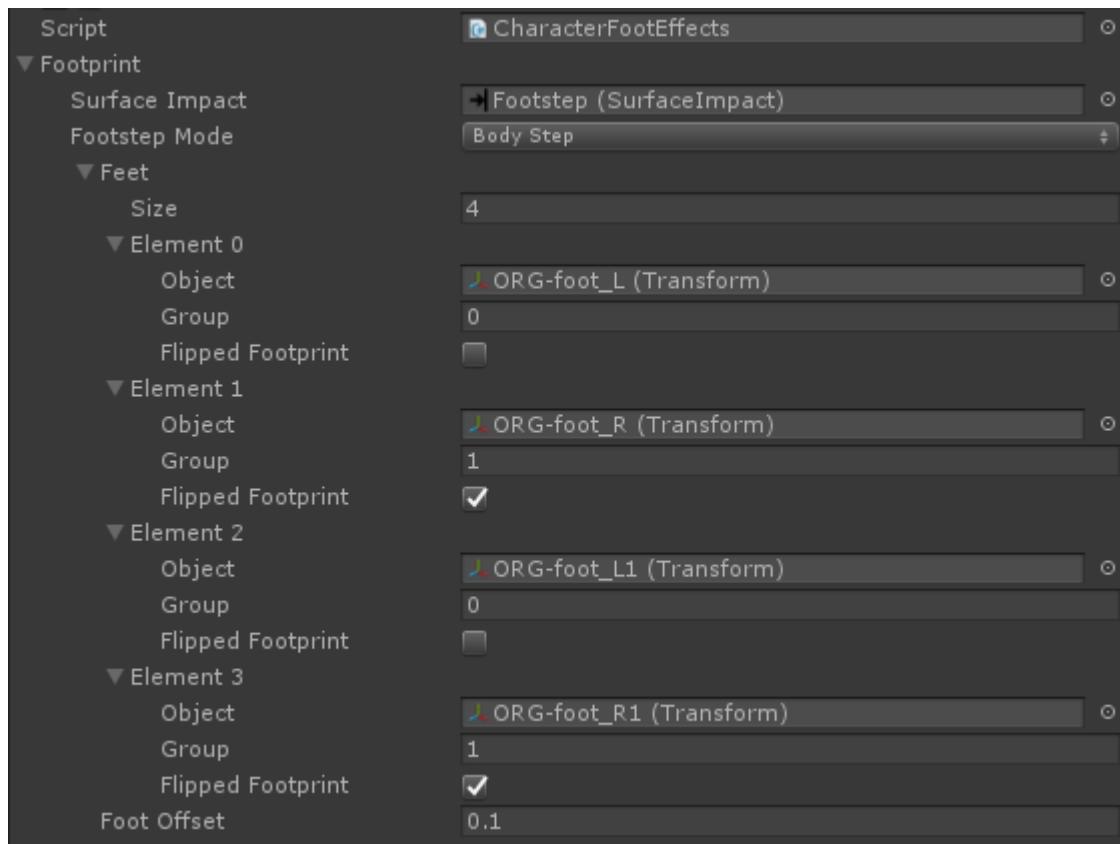
## **Generic Character**

The Ultimate Character Controller supports both humanoids and generic characters. Generic characters do not support [bone retargeting](#) so it does require more setup compared to humanoid characters. Generic characters can be created by performing the following steps:

1. Open the [Character Manager](#) and create your generic character.
  - If you don't have an Animator Controller already created for the Ultimate Character Controller then a new Animator Controller should be created and specified within the Animator Controller field.
  - The CharacterIK and Ragdoll components cannot be added when creating the character. These components use Unity features which rely on humanoid bones in order to function properly.



2. Build your character. If *Foot Effects* is enabled then the foot transform needs to be specified within the [Character Foot Effects](#) component.



# Time

There are two ways to manipulate time:

- Change the *Time Scale* within the Unity [Time Manager](#). This changes the time scale for all objects within the scene.
- Change the *Time Scale* within the Ultimate Character Locomotion. This will change the time scale for only the character. A negative value cannot be set.

When the time scale is changed on the character any objects spawned from the character will also have the same time scale. As an example if you throw a grenade when the character has a 0.5 time scale then the grenade will take twice as long as normal to move.

## Events

When the character's time scale changes the “OnChangeTimeScale” event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. No event will be sent when the time scale is changed through the Unity Time Manager. The following example will subscribe to this event from a new component:

```
using UnityEngine;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Initialize the default values.
    /// </summary>
```

```

public void Awake()
{
    EventHandler.RegisterEvent<float>(gameObject,
"OnCharacterChangeTimeScale", OnChangeTimeScale);
}

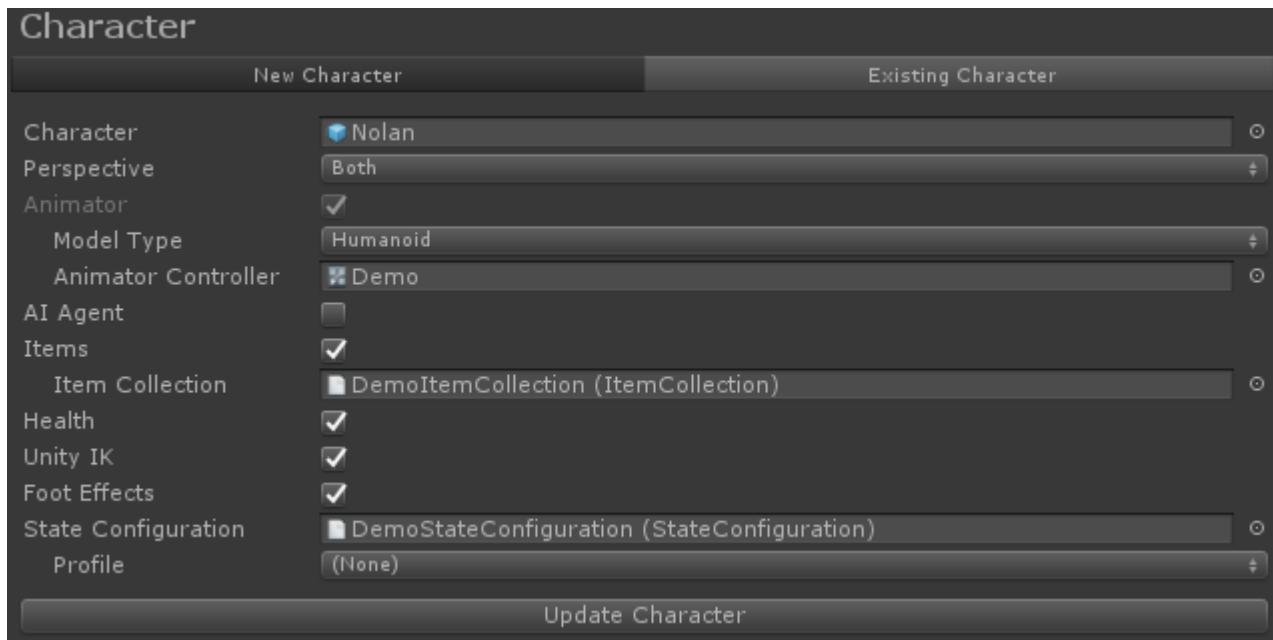
/// <summary>
/// The character's local timescale has changed.
/// </summary>
/// <param name="timeScale">The new timescale.</param>
private void OnChangeTimeScale(float timeScale)
{
    Debug.Log("New time scale: " + timeScale);
}

/// <summary>
/// The GameObject has been destroyed.
/// </summary>
public void OnDestroy()
{
    EventHandler.UnregisterEvent<float>(gameObject,
"OnCharacterChangeTimeScale", OnChangeTimeScale);
}
}

```

## Minimum Component Setup

The Ultimate Character Controller is designed to be extremely modular so if you don't want to use a certain piece of functionality then you can just not include that component. For example, if your character doesn't need the health system then the Character Health component doesn't need to be added. The [Character Manager](#) makes it easy to add/remove various components so it is recommended that the components be added through this editor interface.



The minimum set of components required are:

- Rigidbody (kinematic)
- Ultimate Character Locomotion
- Layer Manager
- A CapsuleCollider or SphereCollider

If your character is controlled by the player (rather than being AI or networked controlled) then the following components should be added:

- Ultimate Character Locomotion Handler
- Player Input (Unity Input component or equivalent input integration component)

If your character is a third person character (including AI or networked) then your character also needs:

- Animator
- Animator Monitor

## Camera

The Camera Controller component is responsible for moving the camera to the correct rotation and position. When the character is attached to a camera (such as for a player-controlled, non AI character) then the Camera Controller is also responsible for determining which direction the character should look or use their item. As a result of this the Camera Controller is required for any player-controlled character.

The Camera Controller component doesn't determine the actual position or rotation that the camera should move towards. It instead uses [View Types](#) to determine those values for it. View types can be thought of as [Abilities](#) but for the camera. View types allow for complete flexibility in terms of how the camera behaves without having to change the core Camera Controller component. View types can also be used to integrate with any external camera controllers, such as [Cinemachine](#).

When the Camera Controller starts it will first try to attach to the character specified by the *Character* field if *Init Character On Awake* is enabled. If this option is enabled but there isn't a character specified then it'll show a warning and search for the GameObject that has the Player tag. If there still is no character found then the Camera Controller will disable itself until a Character is assigned.

While the Camera Controller does provide the information so the character knows where to look/use an item, the Camera Controller should not be used for artificial intelligence (AI) or networked characters. See the corresponding [AI](#) and [networking](#) pages for information on how these characters should be setup.

## API

When the camera initializes it will either assign the character automatically based on the Player tag or use the character set within the inspector. If you'd like to change which character the camera follows you can set the *Character* property. If this value is set to null then the camera will disable itself. The camera can also change perspective with the *SetPerspective* method if the camera has a first and third person view type added to it. View types can be changed with the *SetViewType* method.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Camera;
using Opsive.UltimateCharacterController.Utility;

public class MyObject : MonoBehaviour
{
    [Tooltip("The character that should be assigned to the camera.")]
    protected GameObject m_Character;

    /// <summary>
    /// Sets a third person perspective on the Camera Controller.
    /// </summary>
    private void Start()
    {
        var camera = UnityEngineUtility.FindCamera(null);
        if (camera == null) {
            return;
        }

        var cameraController =
camera.GetComponent<CameraController>();
        cameraController.Character = m_Character;
        cameraController.SetPerspective(false); // false indicates the
third person perspective.

        // Switch to the third person Combat View Type.
        var viewType = "Opsive.UltimateCharacterController.ThirdPersonController.Camera.ViewT
ypes.Combat";
```

```
cameraController.SetViewType(UnityEngineUtility.GetType(viewTypeName),  
false);  
}  
}
```

## Events

The Camera Controller exposes two events using the built-in [Event System](#): OnChangeViewTypes and OnChangePerspectives. Corresponding [Unity events](#) are also exposed for these two events.

### OnChangeViewTypes

The “OnCameraChangeViewTypes” event will be execute when the Camera Controller activates or deactivates a view type. This event can be subscribed to with:

```
EventHandler.RegisterEvent<ViewType, float>(gameObject,  
"OnCameraChangeViewTypes", OnChangeViewTypes);
```

The *gameObject* variable in this example is referring to the Camera’s GameObject. This event includes the view type that was changed as well as if the view type was activated or deactivated.

```
/// <summary>  
/// The view type has changed.  
/// </summary>  
/// <param name="viewType">The ViewType that was activated or  
deactivated.</param>  
/// <param name="activate">Should the current view type be  
activated?</param>  
private void OnChangeViewType(ViewType viewType, bool activate)  
{  
    Debug.Log("The ViewType " + viewType.GetType().Name + " was "  
+ (activate ? "activated" : "deactivated") + ".");  
}
```

### OnChangePerspectives

The “OnCameraChangePerspectives” event will be executed when the Camera Controller switches from a first to third perspective or from a third to first perspective. This event can be subscribed with:

```
EventHandler.RegisterEvent<bool>(gameObject,  
"OnCameraChangePerspectives", OnChangePerspectives);
```

The *gameObject* variable in this example is referring to the Camera’s GameObject. The only parameter sent with this event includes a bool indicating if the camera is now in a first person perspective (true) or a third person perspective (false).

```
/// <summary>
```

```

/// The camera perspective between first and third person has changed.
/// </summary>
/// <param name="firstPersonPerspective">Is the camera in a first
person perspective?</param>
private void OnChangePerspectives(bool firstPersonPerspective)
{
    Debug.Log("The camera is now in a " + (firstPersonPerspective
? "first" : "third") + " person perspective.");
}

```

## OnZoom

The “OnCameraZoom” event will be executed when the Camera Controller starts or stop zooming. This event can be subscribed to with:

```
EventHandler.RegisterEvent<bool>(gameObject, "OnCameraZoom", OnZoom);
```

The *gameObject* variable in this example is referring to the Camera’s GameObject. The only parameter sent with this event includes a bool indicating if the camera is zoomed (true) or not zoomed (false).

```

/// <summary>
/// The camera has toggled zoom state.
/// </summary>
/// <param name="zoomed">Is the camera zoomed?</param>
private void OnZoom(bool zoomed)
{
    Debug.Log("The camera is now in a " + (zoomed ? "zoomed" :
"not zoomed") + " state.");
}

```

## Inspected Fields

### Init Character On Awake

Should the character be initialized on awake?

### Character

The character that the camera should follow.

### Anchor

The transform of the object to attach the camera relative to. This for example allows the camera to be attached to the character’s head instead of the base pivot position. Note that if the camera is attached to a body part that moves there will be a lot more motion when the character moves.

### Auto Anchor

Should the anchor be assigned automatically based on the humanoid bone?

#### **Auto Anchor Bone**

The bone that the anchor will be assigned to if AutoAnchor is enabled.

#### **Anchor Offset**

The offset between the anchor and the camera.

#### **First Person View Type**

The active first person ViewType. This will only be shown if both the First Person Controller and Third Person Controller is imported.

#### **Third Person View Type**

The active third person ViewType. This will only be shown if both the First Person Controller and Third Person Controller is imported.

#### **Can Change Perspectives**

Can the camera change perspectives?

#### **Can Zoom**

Can the camera zoom?

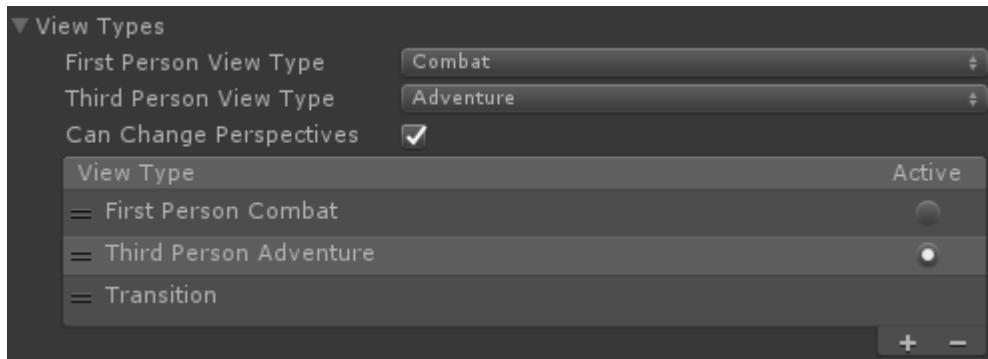
#### **Zoom State**

The state that should be activated when zoomed.

## **View Types**

View Types define the rotation and the position that the camera should orient itself towards. The camera uses the horizontal and vertical inputs (Mouse X/Mouse Y by default) from the PlayerInput component in order to determine which direction it should move. The default view type will rotate the camera, move the character, and then position the camera. This order allows the character to correctly orient itself towards the camera direction.

View Types can be selected within the ViewType Reorderable List on the Camera Controller component. View Types can be added and removed through this list as well.



## API

View Types can be retrieved by using the `GetViewType` method on the Camera Controller component. The `ActiveViewType` property will contain a reference to the View Type that is currently being used by the Camera Controller.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Camera;
using Opsive.UltimateCharacterController.Utility;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Retrieves the combat view type.
    /// </summary>
    private void Start()
    {
        var camera = UnityEngineUtility.FindCamera(null);
        if (camera == null) {
            return;
        }

        var combatViewType =
camera.GetComponent<CameraController>().GetViewType<Opsive.UltimateCharacterController.FirstPersonController.Camera.ViewTypes.Combat>();
        if (combatViewType != null) {
            Debug.Log("Found the Combat view type.");
        }
    }
}
```

## Create a New View Type

It is rare that you will need to create a new View Type but if you do it is pretty straight forward if you do need to do so. New View Types can be created by inheriting the `ViewType` class and implementing three methods:

```
/// <summary>
/// Rotates the camera according to the horizontal and vertical
movement values.
```

```

/// </summary>
/// <param name="horizontalMovement">-1 to 1 value specifying the
amount of horizontal movement.</param>
/// <param name="verticalMovement">-1 to 1 value specifying the amount
of vertical movement.</param>
/// <param name="immediateUpdate">Should the camera be updated
immediately?</param>
/// <returns>The updated rotation.</returns>
Quaternion Rotate(float horizontalMovement, float verticalMovement,
bool immediateUpdate);

/// <summary>
/// Moves the camera according to the current pitch and yaw values.
/// </summary>
/// <param name="immediateUpdate">Should the camera be updated
immediately?</param>
/// <returns>The updated position.</returns>
Vector3 Move(bool immediateUpdate);

/// <summary>
/// Returns the direction that the character is looking.
/// </summary>
/// <param name="lookPosition">The position that the character is
looking from.</param>
/// <param name="characterLookDirection">Is the character look
direction being retrieved?</param>
/// <param name="layerMask">The LayerMask value of the objects that
the look direction can hit.</param>
/// <param name="useRecoil">Should recoil be included in the look
direction?</param>
/// <returns>The direction that the character is looking.</returns>
Vector3 LookDirection(Vector3 lookPosition, bool
characterLookDirection, int layerMask, bool useRecoil);

```

## **Rotate**

The Rotate method will do the actual rotation for the camera. A quaternion is returned which the Camera Controller will then use to set the rotation of the camera.

## **Move**

The Move method will position the camera. A Vector3 is returned which the Camera Controller will then use to set the position of the camera.

## **LookDirection**

Specifies the direction that the character should look. This method is called by the character/item classes when it need to retrieve the a directional rotation, such as determining which direction the head should look for IK or determining which direction a

weapon should fire.

The following properties must also be implemented:

```
public abstract bool FirstPersonPerspective { get; }
public abstract float Pitch { get; }
public abstract float Yaw { get; }
public abstract Quaternion CharacterRotation { get; }
public abstract float LookDirectionDistance { get; }
```

### **FirstPersonPerspective**

Is the View Type used when the character is in a first person perspective?

### **Pitch**

Returns the pitch of the camera. This property is used when switching View Types.

### **Yaw**

Returns the yaw of the camera. This property is used when switching View Types.

### **CharacterRotation**

Returns the rotation of the character. This property is used when switching View Types.

### **LookDirectionDistance**

Specifies the max distance that the character can look ahead from their current position. This is mostly used by the View Type within the LookDirection method.

### **Example**

The following View Type is an example of a complete view type that will keep the camera in the same position/rotation. It can be used as a stationary camera.

```
using Opsive.UltimateCharacterController.Camera.ViewTypes;

public class MyViewType : ViewType
{
    private float m_Pitch;
    private float m_Yaw;
    private Quaternion m_CharacterRotation;

    public override bool FirstPersonPerspective { get { return false; } }
    public override float Pitch { get { return m_Pitch; } }
    public override float Yaw { get { return m_Yaw; } }
    public override Quaternion CharacterRotation { get { return m_CharacterRotation; } }
```

```

        public override float LookDirectionDistance { get { return 100; } }

        public override void ChangeViewType(bool activate, float pitch,
float yaw, Quaternion characterRotation)
{
    if (activate) {
        m_Pitch = pitch;
        m_Yaw = yaw;
        m_CharacterRotation = characterRotation;
    }
}

public override Quaternion Rotate(float horizontalMovement, float
verticalMovement, bool immediateUpdate)
{
    return m_Transform.rotation;
}

public override Vector3 Move(bool immediateUpdate)
{
    return m_Transform.position;
}

public override Vector3 LookDirection(Vector3 lookPosition, bool
characterLookDirection, int layerMask, bool useRecoil)
{
    return m_CharacterTransform.forward;
}
}

```

## Included View Types

### First Person

The First Person view type is an abstract view type that will move the camera in the direction of the input mapped to the camera. The view type uses the [spring system](#) to allow for bobs, sways, shakes, and reaction to external forces. If the attached character has a humanoid then the view type will follow along with the local delta changes of the neck. This will prevent the camera from looking on the inside of the character model.

#### First Person Camera

With a single camera in a first person view it's possible for the character's hands and items to clip the object in front of them. Consider the following screenshot which shows a character standing in front of a wall with the assault rifle equipped.



The common solution for this is to add a second camera that just renders the hands and items (the Overlay layer in the Ultimate Character Locomotion). With this solution the objects will no longer clip the wall:



When a first person view type is added to the Camera Controller it will automatically add the first person camera if the camera does not already exist. This camera will be added as a child to the main camera and have the correct values set for the camera's render settings.

A second camera is not needed if you are using the lightweight render pipeline – see [this page](#) for more details.

## Head Bob

When the character is idle or moving the First Person View Type will play a bobbing animation based on the *Shake Speed* and *Shake Amplitude*. If you notice bobbing only while the character is moving this is likely caused by the *Smooth Head Offset Steps* field. See the

details below for a in-depth description of this field.

## Inspected Fields

### Look Direction Distance

The distance that the character should look ahead.

### Look Offset

The offset between the anchor and the camera.

### Look Down Offset

Amount to adjust the camera position by when the character is looking down. This will prevent the camera from clipping the character's body mesh.

### Culling Mask

The culling mask of the camera.

### Field Of View

The field of view of the main camera.

### Field Of View Damping

The damping time of the field of view angle when changed.

### First Person Position Offset

Specifies the position offset from the camera that the first person objects should render.

### First Person Rotation Offset

Specifies the rotation offset from the camera that the first person objects should render.

### Object Overlay Render Type

Specifies how the overlay objects are rendered. This field is only shown if the lightweight or universal render pipelines are imported.

- *Second Camera*: Use a second stacked camera to ensure the overlay objects do no clip with any other objects.
- *Render Pipeline*: Use the LWRP/URP render pipeline to ensure the overlay objects do no clip with any other objects.
- *None*: No special rendering for the overlay objects.

### Use First Person Camera

Should the first person camera be used? The first person camera may not want to be used if

the first person objects (arms/items) should intersect with scene objects, such as for an interact animation. This field is shown if the lightweight and universal render pipelines are not imported.

#### **First Person Camera**

A reference to the first person camera. This camera will render the items/character hands to prevent clipping with overlapping objects.

#### **First Person Culling Mask**

The culling mask of the first person camera.

#### **Synchronize Field Of View**

Should the first person camera's field of view be synchronized with the main camera?

#### **First Person Field Of View**

Specifies the field of view for the first person camera.

#### **First Person Field Of View Damping**

The damping time of the field of view angle when changed.

#### **Position Lower Vertical Limit**

A vertical limit intended to prevent the camera from intersecting with the character.

#### **Position Fall Impact**

Determines how much the camera will be pushed down when the player falls onto a surface.

#### **Position Fall Impact Softness**

The number of frames that the fall impact force should be applied.

#### **Rotation Strafe Roll**

Rotates the camera depending on the sideways local velocity of the character, resulting in the camera leaning into or away from its sideways movement direction.

#### **Rotation Fall Impact**

Determines how much the camera will roll when the player falls onto a surface.

#### **Rotation Fall Impact Softness**

The number of frames that the fall impact force should be applied.

### **Position Spring**

The positional spring used for regular movement.

### **Rotation Spring**

The rotational spring used for regular movement.

### **Secondary Position Spring**

The positional spring which returns to equilibrium after a small amount of time (for recoil).

### **Secondary Rotation Spring**

The rotational spring which returns to equilibrium after a small amount of time(for recoil).

### **Min Pitch Limit**

The minimum pitch angle(in degrees).

### **Max Pitch Limit**

The maximum pitch angle(in degrees)

### **Bob Positional Rate**

The rate that the camera changes its position while the character is moving.

### **Bob Positional Amplitude**

The strength of the positional camera bob.Determines how far the camera swings in each respective direction.

### **Bob Roll Rate**

The rate that the camera changes its roll rotation value while the character is moving.

### **Bob Roll Amplitude**

The strength of the roll within the camera bob. Determines how far the camera tilts from left to right.

### **Bob Input Velocity Scale**

This tweaking feature is useful if the bob motion gets out of hand after changing character velocity.

### **Bob Max Input Velocity**

A cap on the velocity value from the bob function, preventing the camera from flipping out when the character travels at excessive speeds.

### **Bob Min Through Vertical Offset**

A trough should only occur when the bob vertical offset is less than the specified value.

### **Bob Trough Force**

The amount of force to add when the bob has reached its lowest point. This can be used to add a shaking effect to the camera to mimick a giant walking.

### **Bob Require Ground Contact**

Determines whether the bob should stay in effect only when the character is on the ground.

### **Shake Speed**

The speed that the camera should shake.

### **Shake Amplitude**

The strength of the shake. Determines how much the camera will tilt.

### **Smooth Head Offset Steps**

If the character is a humanoid the First Person View Type will follow the character's head position. This prevents the camera from seeing inside the character's model when the character has a different vertical height (such as if the character is crouching). This head tracking will also give a slight bob as the character is moving. You can disable the head tracking completely by setting the *Smooth Head Offset Steps* field to 0. This field specifies the number of head positions the camera remembers when determining the head position. The larger the value to more positions will be remembered which gives a smoother result.

### **Collision Radius**

The radius of the camera's collision sphere to prevent it from clipping with other objects.

## **Combat**

The Combat view type extends the [First Person](#) view type by allowing the camera to change yaw values. This view type will move the camera in a manner common to most first person games. The [First Person Combat](#) movement type should be used in conjunction with this view type.

## **Free Look**

The Free Look view type extends the [First Person](#) view type by allowing the camera to move independently of the character. With the Free Look view type weapon will fire in the direction that it is currently facing rather than in the camera's look direction. The [First Person Free Look](#) movement type should be used in conjunction with this view type.

## Inspected Fields

### Min Yaw Limit

The minimum yaw angle (in degrees).

### Max Yaw Limit

The maximum yaw angle (in degrees).

### Yaw Limit Lerp Speed

The speed in which the camera should rotate towards the yaw limit when out of bounds.

## First Person Transform Look

The First Person Transform Look ViewType is a first person view type that will move and rotate according to the specified Transform. This view type is most useful for when the first person character dies and the camera should look from the perspective of the character (such as the head).

## Inspected Fields

### Move Target

The object that determines the position of the camera.

### Rotation Target

The object that determines the rotation of the camera.

### Offset

The offset relative to the move target.

### Collision Radius

The radius of the camera's collision sphere to prevent it from clipping with other objects

### Move Speed

The speed at which the camera should move.

### Rotation Lerp Speed

The speed at which the view type should rotate towards the target rotation.

# Third Person

The Third Person view type is an abstract view type that will move the camera in the direction of the input mapped to the camera. The view type will orbit around the character while always have the character in view, making sure no objects obstruct the view. The [spring system](#) can also be used for things such as recoil or camera shakes.

## Inspected Fields

### **Look Direction Distance**

The distance that the character should look ahead.

### **Look Offset**

The offset between the anchor and the camera.

### **Look Offset Smoothing**

The amount of smoothing to apply to the look offset. Can be zero.

### **Forward Axis**

The forward axis that the camera should adjust towards.

### **Field Of View**

The field of view of the main camera.

### **Field Of View Damping**

The damping time of the field of view angle when changed.

### **Collision Radius**

The radius of the camera's collision sphere to prevent it from clipping with other objects.

### **Position Smoothing**

The amount of smoothing to apply to the position. Can be zero.

### **Obstruction Position Smoothing**

The amount of smoothing to apply to the position when an object is obstructing the target position. Can be zero.

### **Min Pitch Limit**

The minimum pitch angle (in degrees).

### **Max Pitch Limit**

The maximum pitch angle (in degrees).

### **Position Spring**

The positional spring used for regular movement.

### **Rotation Spring**

The rotational spring used for regular movement.

### **Secondary Position Spring**

The positional spring which returns to equilibrium after a small amount of time (for recoil).

### **Secondary Rotation Spring**

The rotational spring which returns to equilibrium after a small amount of time (for recoil).

### **Step Zoom Input Name**

The name of the step zoom input mapping.

### **Step Zoom Sensitivity**

Specifies how quickly the camera zooms when step zooming.

### **Min Step Zoom**

The minimum distance that the step zoom can zoom.

### **Max Step Zoom**

The maximum distance that the step zoom can zoom.

## **Adventure**

The Adventure view type extends the Third Person view type by allow the camera's yaw angle to rotate freely. A minimum and maximum yaw value can be specified. The [Third Person Adventure](#) movement type should be used in conjunction with this view type.

### **Inspected Fields**

#### **Min Yaw Limit**

The minimum yaw angle (in degrees).

## **Max Yaw Limit**

The maximum yaw angle (in degrees).

# **Combat**

The Combat view type extends the Third Person view type while keeping the camera rotated to the same local yaw value as the character. This in effect will always have the character rotated in the same forward direction as the camera. The [Third Person Combat](#) movement type should be used in conjunction with this view type.

# **RPG**

The RPG view type extends the Third Person view type for a control scheme similar to the standard setup of the RPG genre. This movement type allows for free yaw movement when the *Camera Free Movement* button is down, but will otherwise follow behind the character. The [RPG](#) movement type should be used in conjunction with this view type.

## **Inspected Fields**

### **Yaw Snap Damping**

The damping of the yaw angle when it snaps back to behind the character as the character moves.

### **Camera Free Movement Input Name**

The name of the camera free movement input damping.

# **Third Person Look At**

The Third Person Look At view type will orient the camera so it is looking at the target. If no target is specified the camera will look towards the character's pivot position. This view type is useful for the death camera showing the character's ragdoll reacting to the physics forces or the death animation playing.

## **Inspected Fields**

### **Target**

The object to look at. If this value is null then the character's transform will be used.

### **Offset**

The offset relative to the character.

### **Min Look Distance**

The minimum distance away from the target that the camera should move towards.

### **Max Look Distance**

The maximum distance away from the target that the camera should move towards.

### **Move Speed**

The speed at which the camera should move.

### **Collision Radius**

The radius of the camera's collision sphere to prevent it from clipping with other objects.

### **Rotational Lerp Speed**

The speed at which the view type should rotate towards the target rotation.

### **Rotation Spring**

The spring used for applying a rotation to the camera.

## **Third Person Pseudo3D (2.5D)**

The Pseudo3D view type places the camera in a 2.5D view, allowing the camera to look at the character from the side. The [Pseudo3D \(2.5D\)](#) movement type should be used in conjunction with this view type. If the 2.5D movement type is active and references a path then the camera will follow that path.

### **Inspected Fields**

#### **Look Direction Distance**

The distance that the character should look ahead.

#### **Forward Axis**

The forward axis that the camera should adjust towards.

#### **View Distance**

The distance to position the camera away from the anchor.

#### **Vertical Dead Zone**

The camera will readjust the position/rotation if the character moves outside of this vertical dead zone.

### **Move Smoothing**

The amount of smoothing to apply to the movement. Can be zero.

### **Rotation Smoothing**

The amount of smoothing to apply to the rotation. Can be zero.

### **Depth Look Direction**

Should the look direction account for depth offsets? This is only used when the mouse is visible.

## **Third Person Top Down**

The Top Down view type allows the camera to be placed in a top down perspective with the character in view. The [Top Down](#) movement type should be used in conjunction with this view type.

### **Inspected Fields**

#### **Look Direction Distance**

The distance that the character should look ahead.

#### **Forward Axis**

The forward axis that the camera should adjust towards.

#### **Up Axis**

The up axis that the camera should adjust towards.

#### **Rotation Speed**

The speed at which the camera rotates to face the character.

#### **Min Pitch Limit**

The minimum pitch angle (in degrees).

#### **Max Pitch Limit**

The maximum pitch angle (in degrees).

#### **Collision Radius**

The radius of the camera's collision sphere to prevent it from clipping with other objects.

### **View Distance**

The distance to position the camera away from the anchor.

### **View Step**

The number of degrees to adjust if the anchor is obstructed by an object.

### **Move Smoothing**

The amount of smoothing to apply to the movement. Can be zero.

### **Vertical Look Direction**

Should the look direction account for vertical offsets? This is only used when the mouse is visible.

## **Transition**

The Transition view type will transition the camera from one view type to another. This view type should not explicitly be set but will instead be activated when changing from one view type to another. The view type has different transition durations depending on which perspective the camera is transitioning from/to. If a value of 0 is specified for any of these durations then the camera will instantly be transition to the new view type.

### **Inspected Fields**

#### **First to Third Transition Duration**

The amount of time to transition from a third person to first person person.

#### **Start First Person Camera Offset**

The starting offset when transition from a first person perspective to third person.

#### **Third to First Transition Duration**

The amount of time to transition from a third person to first person perspective.

#### **End First Person Camera Offset**

The ending offset when transition from a third person perspective to first person.

#### **Third to Third Transition Duration**

The amount of time to transition from a third person to another third person perspective

# Aim Assist

The Aim Assist component allows for the camera and character to automatically face the specified target. A crosshairs with the Crosshairs Monitor is required. If Aim Assist is active the crosshairs will notify that it has found a target and that the character/camera should look in the target's direction.

## Setup

1. Add the Aim Assist and the Aim Assist Handler components to your camera.
2. Ensure the Crosshairs Monitor has been added to your crosshairs. The Aim Assist component will target any object that uses the [Character's Enemy Layer](#).
3. Set the desired values of the fields on the Aim Assist Handler and Aim Handler. The value that will most likely require modification is the *Influence* field of the Aim Assist component.
4. The Aim Assist component has the option to target humanoid bones through the *Target Humanoid Bone* field. If the target is not a humanoid the *Aim Assist Offset* component can be added to the target GameObject to apply an offset to the look direction.

## Inspected Fields

### Assist Aim

Should the component assist with the aiming?

### Require Active Aim

Does the Aim ability need to be active in order for the component to assist with aiming?

### Max Distance

The maximum distance that the target can be away from the character in order to start influence the aim.

### Influence

The amount of influence the aim assist has on the camera rotation. The x value represents the angle delta between the current camera rotation and the camera rotation. The y value represents the amount of influence at that angle. A y value of 1 indicates complete influence while a value of 0 indicates no influence.

### Target Offset

Specifies an offset to apply to the target.

### Target Humanoid Bone

If the target is a humanoid should a bone from the humanoid be targeted?

### **Humanoid Bone Target**

Specifies which bone to target if targeting a humanoid bone.

### **Break Force**

The magnitude required in order to break the current target lock.

### **Switch Target Radius**

If trying to switch targets, specifies the radius that the nearby targets should be in.

### **Switch Target Rotation Speed**

If switching targets, specifies the speed at which the camera rotates to face the new target.

### **Max Switch Target Colliders**

The maximum number of colliders that should be considered within the target switch.

## **Lightweight Render Pipeline**

When the first person arms and items are drawn to the screen they are drawn using a second camera (called camera stacking). This prevents the [objects from clipping](#) with walls that are in front of the character. This works well for the standard pipeline, but the lightweight render pipeline does not support camera stacking so another solution needs to be used.

Beginning in Unity 2019.2 you can change the [camera's Renderer Type](#) to change the order that the objects are rendered. The Ultimate Character Controller uses this method to ensure the first person arms and items do not clip with other objects.

## **Downloads**

[Ultimate Character Controller](#)

[First Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

## **Setup**

1. [Setup your project](#) to use the lightweight render pipeline.
2. **Import the lightweight render pipeline integration package.**
3. Change the camera's *Render Type* to Custom and set the *Renderer Data* to OverlayForwardRendererData:



4. On the Camera Controller select the First Person View Type and change the *Overlay Render Type* to Render Pipeline:



5. To prevent duplicate shadows ensure *Cast Shadows* is set to Off for all of the [Renderers](#) on the first person objects. For example, with the example character you'll want to ensure *Cast Shadows* is set to Off for all of the objects underneath the Nolan/First Person Objects GameObject.
6. On your character's Perspective Monitor the *Invisible Material* should be set to the InvisibleShadowCastorLWRP material included in the download.
7. If you are using the [Object Fader](#) component set the *Color Property Name* to “\_BaseColor”.

## Object Fader

The Object Fader component has the option of fading either the character or any obstructing objects. In order for the object to be faded it must be using a material which uses a shader that supports transparency.



## Character Fade

If character fade is enabled then the character will fade if the camera gets too close to the character. This option is most useful with a third person view type. The character's materials must be using a shader that supports transparency. Any material that is on the character which supports transparency will fade if the character fading option is enabled.

### Cache Character Materials

Should the character materials be cached at the start? If false the material values will be saved each time the character starts to fade. This option should generally be enabled unless your character swaps out materials.

### Start Fade Distance

The distance between the character and the camera that the character should start to fade.

### End Fade Distance

The distance between the character and camera that the character materials should be completely invisible.

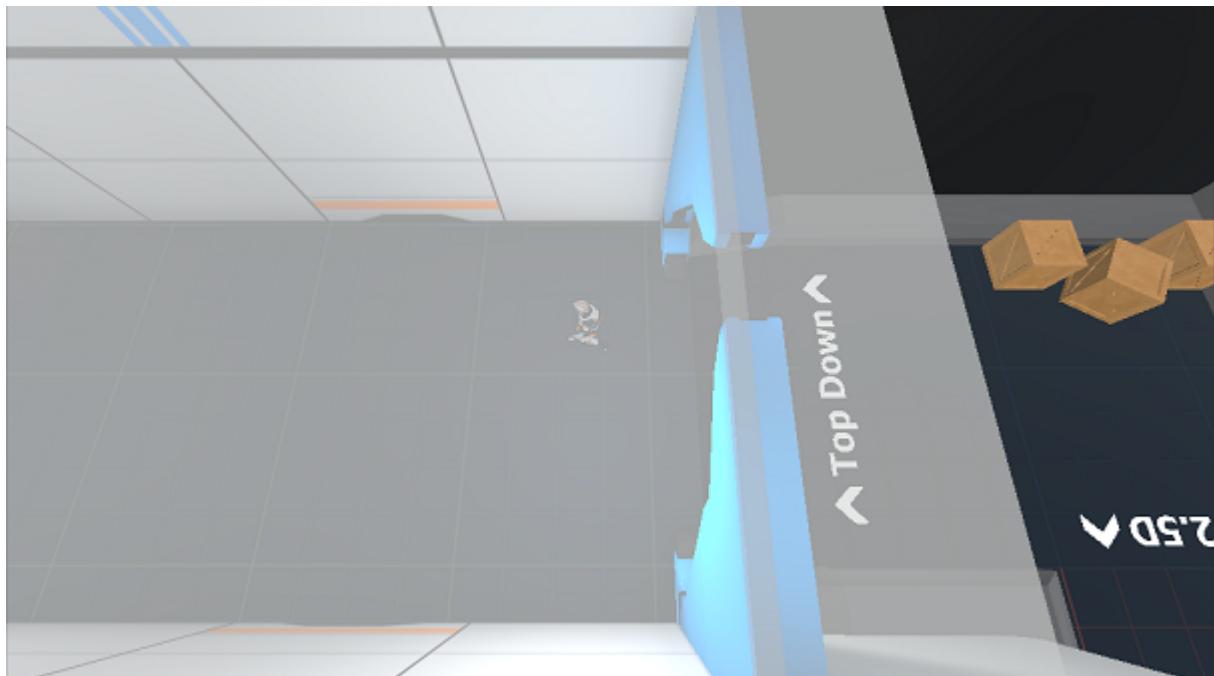
### Character Fade State Change Cooldown

Prevents the character change from updating for the specified number of seconds after a state change.

### Transform Offset

The offset to apply to the character when detecting if the character should fade/is considered obstructed.

## Obstructing Objects Fade



If Obstructing Objects fade is enabled then any objects between the character and the camera will be faded. This option is most useful with a top down or 2.5D view type. The obstructing object's materials must be using a shader that supports transparency.

#### **Collision Radius**

The radius of the camera's collision sphere to prevent it from clipping with other objects. If a top down view type is being used it is recommend that this value is greater than or equal to the collision radius for the top down view type.

#### **Fade Speed**

Specifies the speed at which the obstructing material can fade.

#### **Fade Color**

The color that the obstructed object will fade to.

#### **Auto Set Mode**

Should the material mode be set automatically when an object is obstructing the view? If you don't want any obstructing object to be able to be faded then this option should be set to false and any objects that you want to be able to fade should use a transparent shader.

#### **Disable Collider**

Should the obstructing object's collider be disabled when the material is faded? This is useful so physics raycasts won't detect a faded object.

#### **Max Obstructing Collider Count**

The maximum number of obstructing colliders that can be faded at one time.

### **Max Obstructing Material Count**

The maximum number of obstructing materials that can be faded at one time. This value should be greater than the collider count.

### **Transform Offset**

The offset to apply to the character when detecting if the character should fade/is considered obstructed.

## **Post Processing**

Post processing effects are an awesome addition to any project and will give your project that next-gen look. In third person view the post processing effects can be assigned to the character's camera (the camera with the Camera Controller component). While in first person view there are some special considerations:

- Post processing effects should be assigned to one of the two cameras available which follow your character.
- Post processing effects assigned to the First Person Camera (beneath the character's camera) will affect the scene AND the first person arms/items.
- Post processing effects assigned to the character camera will only affect the scene and NOT the first person arms/items.

## **Split Screen**

When in a first person perspective the third person materials are swapped for an invisible shadow castor material. This then allows the separate first person arms object to render without seeing duplicate arms (or the character's head from getting in the way of the camera). This works well for single player games, but with split screen games or mirrors the material needs to be swapped multiple times so other players don't see a character without any arms or head.

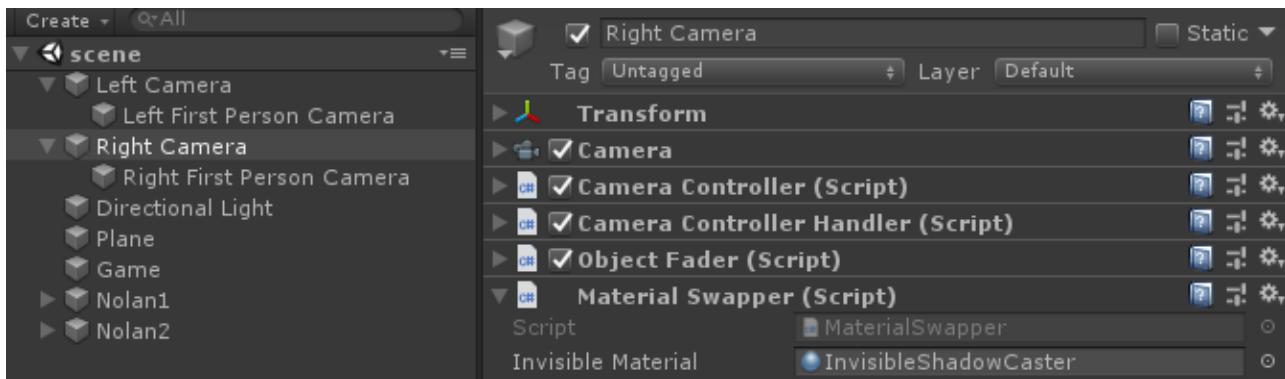


The Split Screen Camera component will do this material swap allowing for multiple cameras on the same screen.



## Setup

1. Add a camera using the [Scene Setup Manager](#).
2. Duplicate that camera. Ensure only one [Audio Listener](#) exists within the scene.
3. Create a new character using the [Character Manager](#).
4. Create a second character using the Character Manager.
5. Add the Material Swapper component to the GameObjects that contain the CameraController.
6. If you are not using a Scriptable Render Pipeline add the Material Swapper component to the First Person Camera that is a child of the main camera. Your setup should look similar to:

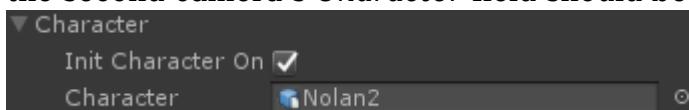


The Material Swapper component exists on the four Camera GameObjects.

7. Adjust the Camera's *Viewport Rect* so the cameras do not render over each other. In the above scene the left viewport was adjusted so it has a width of 0.5. The right camera has a X value of 0.5 and a width of 0.5.



8. On the Camera Controller component set the *Character* field to the first character, and the second camera's *Character* field should be set to the second character.



9. Assign the player-specific inputs to the character. The [Rewired integration](#) makes it easy to have multiple inputs attached to a single instance.

# Universal Render Pipeline

When the first person arms and items are drawn to the screen they are drawn using a second camera (called camera stacking). This prevents the [objects from clipping](#) with walls that are in front of the character. This works well for the standard pipeline, but the universal render pipeline offers a better and more efficient solution.

The [camera's Renderer](#) can be changed which allows you to specify the order that the objects are rendered. The Ultimate Character Controller uses this method to ensure the first person arms and items do not clip with other objects.

## Downloads

[Ultimate Character Controller](#)

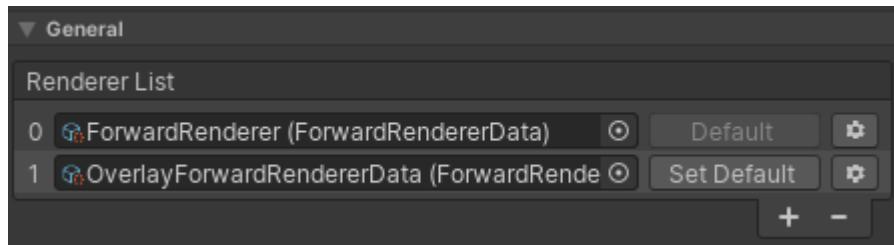
[First Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

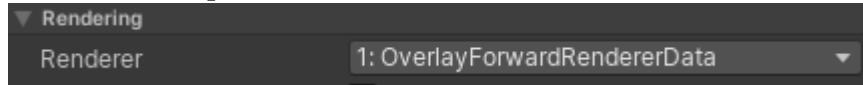
[UFPM: Ultimate First Person Melee](#)

## Setup

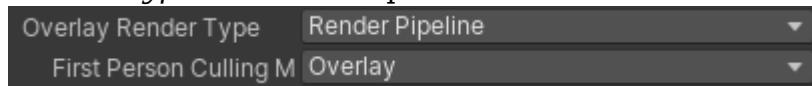
1. [Setup your project](#) to use the universal render pipeline.
2. **Import the universal render pipeline integration package.**
3. Add the OverlayForwardRendererData reference to universal render pipeline's setting file.



4. Change the camera's *Renderer* to the *OverlayForwardRendererData* that was specified in the last step:



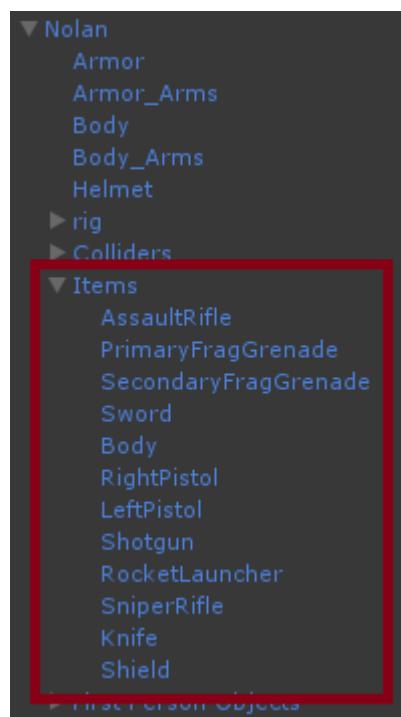
5. On the Camera Controller select the First Person View Type and change the *Overlay Render Type* to Render Pipeline:



6. To prevent duplicate shadows ensure *Cast Shadows* is set to Off for all of the [Renderers](#) on the first person objects. For example, with the example character you'll want to ensure *Cast Shadows* is set to Off for all of the objects underneath the Nolan/First Person Objects GameObject.
7. On your character's Perspective Monitor the *Invisible Material* should be set to the InvisibleShadowCastorUniversalRP material included in the download.
8. If you are using the [Object Fader](#) component set the *Color Property Name* to “BaseColor”.

## Items

The item system is a generic implementation allowing for a wide variety of uses. Items are any object that the character can carry, such as an assault rifle, sword, bow, grenade, shield, flashlight, or book. When a [new item is built](#) it is placed under the Item GameObject which is a child of the character.



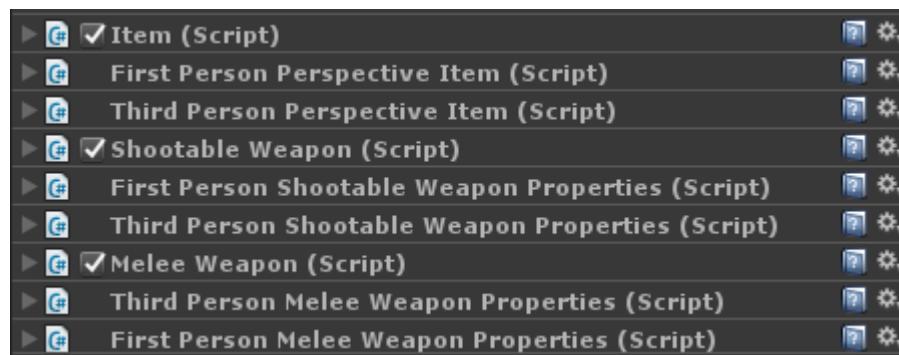
An item is identified by the Item Definition ScriptableObject. This Item Definition must be unique for each [slot](#) and is purely a representation of the item GameObject that should be interacted with. Item Definitions allow you to easily reference which item should be picked up within the ItemPickup component, or which item should be used by the Use ability. Each item GameObject is required to have the Item component and this component implements the logic for more general item-related functions such as initialization, being picked up, equipped, dropped, etc.

The object that is actually held by the character and rendered on the screen is managed by the [Item Perspective](#) component. A [First Person Item Perspective](#) and a [Third Person Item Perspective](#) component should be added to the item GameObject depending on which perspective the item will be viewed with. The Item Perspective component is responsible for managing the item rendered to the screen. The First Person Item Perspective component makes extensive use of the [spring system](#) to move the item in a realistic way while in a first person perspective.

When an ability is activated that performs an action on the item (such as a melee slash or grenade throw) it calls the corresponding [Item Action](#) component to perform the actual action. Multiple Item Action components can be added to the same item GameObject which will allow for the item to perform multiple functions. A classic example of this is an assault rifle that can shoot or be used as a blunt object and melee attack. Another example could be a light sword attack or a heavy sword attack.

The Item Perspective Properties is the last type of component that you'll see added to an Item GameObject and this component is responsible for storing values specific to the corresponding perspective for the Item Action component. As an example the Shootable Weapon component will use the First Person Perspective Properties component to point to the location of the muzzle flash for the first person assault rifle. A Third Person Perspective Properties component will point to the location of the muzzle flash for the third person assault rifle.

With all of these components added to the Item GameObject you'll see a setup similar to the image below. In this image the item is setup for both a first and third person perspective, and it also can be used as a shootable weapon or a melee weapon.



## Inspected Fields

### Item Definition

A reference to the object used to identify the item.

**Slot ID**

Specifies the inventory slot/spawn location of the item.

**Animator Item ID**

Unique ID used for item identification within the animator.

**Animator Movement Set ID**

The movement set ID used for within the animator.

**Dominant Item**

Does the item control the movement and the UI shown?

**Unique Item Set**

Does the item belong to a unique [Item Set](#)?

**Allow Camera Zoom**

Can the camera zoom when the item is equipped?

**Drop Prefab**

The GameObject that is dropped when the item is removed from the character.

**Equip Event**

Specifies if the item should wait for the OnAnimatorItemEquip animation event or wait for the specified duration before equipping. This field uses an [Animation Event Trigger](#).

**Equip Complete Event**

Specifies if the item should wait for the OnAnimatorItemEquipComplete animation event or wait for the specified duration before stopping the equip ability. This field uses an [Animation Event Trigger](#).

**Equip Animator Audio State Set**

Specifies the animator and audio state from an equip. This field uses an [Animator Audio State Set](#).

**Unequip Event**

Specifies if the item should wait for the OnAnimatorItemUnequip animation event or wait for the specified duration before unequipping. This field uses an [Animation Event Trigger](#).

### **Unequip Complete Event**

Specifies if the item should wait for the OnAnimatorItemUnequipComplete animation event or wait for the specified duration before stopping the unequip ability. This field uses an [Animation Event Trigger](#).

### **Unequip Animator Audio State Set**

Specifies the animator and audio state from an unequip. This field uses an [Animator Audio State Set](#).

### **UI Monitor ID**

The ID of the UI Monitor that the item should use.

### **Icon**

The sprite representing the icon.

### **Show Crosshairs On Aim**

Should the crosshairs be shown when the item aims?

### **Center Crosshairs**

The sprite used for the center crosshairs image.

### **Quadrant Offset**

The offset of the quadrant crosshairs sprites.

### **Max Quadrant Spread**

The max spread of the quadrant crosshairs sprites caused by a recoil or reload.

### **Quadrant Spread Damping**

The amount of damping to apply to the spread offset.

### **Left Crosshairs**

The sprite used for the left crosshairs image.

### **Top Crosshairs**

The sprite used for the top crosshairs image.

### **Right Crosshairs**

The sprite used for the right crosshairs image.

## **Bottom Crosshairs**

The sprite used for the bottom crosshairs image.

## **Show Full Screen UI**

Should the item's full screen UI be shown?

## **Events**

Four [Unity events](#) are exposed by the item:

- Pickup Item
- Equip Item
- Unequip Item
- Drop Item

These events can be assigned within the item's inspector.

# **Perspective**

A Perspective Item represents the object that is actually rendered in the game. The Perspective Item class is an abstract class and is implemented by the [First Person Perspective Item](#) and the [Third Person Perspective Item](#) depending on which perspective is being rendered. The main responsibility of the perspective components are to spawn and to determine the location that the item should be rendered at. While both the first and third person components can use the spring system, the springs are more prevalent in a first person perspective because the component needs to handle bobs, sways, falls, etc. With a third person perspective the item is more restricted to the parent object (such as the hand).

## **Inspected Fields**

### **Object**

The GameObject of the object rendered. Can be a prefab or a GameObject that is a child of the character.

### **Local Spawn Position**

If Object is a prefab, specifies the local position of the spawned object.

### **Local Spawn Rotation**

If Object is a prefab, specifies the local rotation of the spawned object.

# First Person

The First Person Perspective Item will position and rotate the first person objects according to [spring forces](#). This can be used in conjunction with regular animations allowing for more complex motions such as reloading. Similar to the [first person camera](#), the perspective item allows for bobs, sways, shakes, and reaction to external forces.

## Inspected Fields

### First Person Base Object ID

The ID of the First Person Base Object that the item should be spawned under. This is used when the item is [picked up at runtime](#).

### Visible Item

The GameObject of the visible item. This is the actual item object such as the assault rifle or sword. The Object from the Perspective Item parent class contains a reference to the arms that the visible item is spawned as a child of.

### VR Hand Parent

Should the pivot be positioned as the immediate parent of the VisibleItem? This is useful for VR.

### Additional Control Objects

Any additional objects that the item should control the location of. This is useful for dual wielding where the item should update the location of another base object.

### Addition Control Object Base IDs

Any additional object IDs that the item should control the location of. This is useful for dual wielding where the item should update the location of another base object.

### Position Spring

The positional spring used for regular movement (bob, sway, etc).

### Position Offset

An offset relative to the parent pivot Transform. This position is where the item “wants to be”. It will try to go back to this position after any forces are applied.

### Position Exist Offset

An offset relative to the parent pivot Transform. This is the desired position when the item is moving off screen (in the case of unequipping).

## **Position Fall Impact**

Determines how much the item will be pushed down when the player falls onto a surface.

## **Position Fall Impact Softness**

The number of frames over which to even out each the fall impact force.

## **Position Fall Retract**

Makes the weapon pull backward while falling.

## **Position Move Slide**

Sliding moves the item in different directions depending on the character movement direction

X slides the item sideways when strafing.

Y slides the item down when strafing.

Z slides the item forward or backward when walking.

## **Position Platform Slide**

Sliding which moves the item in different directions depending on the moving paltform's velocity."

## **Position Input Velocity Scale**

A tweak parameter that can be used if the spring motion goes out of hand after changing character velocity. Use this slider to scale back the spring motion without having to adjust lots of values.

## **Position Max Input Velocity**

A cap on the velocity value being fed into the item swaying method, preventing the item from flipping out when the character travels at excessive speeds (such as when affected by a jump pad or a speed boost). This also affects vertical sway.

## **Rotation Spring**

The rotational spring used for the regular movement (bob, sway, etc).

## **Rotation Offset**

An offset relative to the parent pivot Transform. This rotation is where the item "wants to be". It will try to go back to this rotation after any forces are applied.

## **Rotation Exit Offset**

An offset relative to the parent pivot Transform. This is the target rotation when the item is moving off screen (in the case of unequipping).

## **Rotation Fall Impact**

Determines how much the item will roll when the player falls onto a surface.

## **Rotation Fall Impact Softness**

The number of frames over which to even out each the fall impact force.

## **Rotation Look Sway**

This setting determines how much the item sways (rotates) in reaction to input movements. Horizontal and vertical movements will sway the weapon spring around the Y and X vectors, respectively. Rotation around the Z vector is hooked to horizontal movement, which is very useful for swaying long melee items such as swords or clubs.

## **Rotation Strafe Sway**

Rotation strafe sway rotates the item in different directions depending on character movement direction.

X rotates the item up when strafing (it can only rotate up).

Y rotates the item sideways when strafing.

Z twists the item around the forward vector when strafing.

## **Rotation Vertical Sway**

This setting rotates the item in response to vertical motion (e.g.falling or walking in stairs). Rotations will have opposing direction when falling versus rising. However, the weapon will only rotate around the Z axis while moving downwards / falling.

## **Rotation Platform Sway**

Sliding which rotates the item in different directions depending on the moving paltform's velocity.

## **Rotation Ground Sway Multiplier**

This parameter reduces the effect of item vertical sway when moving on the ground. At a value of 1 the item behaves as if the player behaves normally. A value of 0 will disable vertical sway altogether when the character is grounded.

## **Rotation Input Velocity Scale**

A tweak parameter that can be used to temporarily alter the impact of input motion on the item rotation spring, for example in a special player state.

## **Rotation Max Input Velocity**

A cap on the velocity value being fed into the item swaying method, preventing the item from flipping out when extreme mouse sensitivities are being used.

### **Pivot Position Spring**

The positional spring used for the item pivot Transform.

### **Pivot Position Offset**

An offset relative to the parent First Person Object Transform.

### **Pivot Rotation Spring**

The rotational spring used for the item pivot Transform.

### **Pivot Rotation Offset**

An offset relative to the parent First Person Object Transform.

### **Shake Speed**

Determines the shaking speed of the item.

### **Shake Amplitude**

The strength of the angular item shake around the X, Y and Z vectors.

### **Bob Positional Rate**

The rate that the item changes its position while the character is moving.Tip: y should be (x\*2) for a nice classic curve of motion.

### **Bob Positional Amplitude**

The strength of the positional item bob. Determines how far the item swings in each respective direction.Tip: make x and y negative to invert the curve.

### **Bob Rotational Rate**

The rate that the item changes its rotation value while the character is moving.

### **Bob Rotational Amplitude**

The strength of the rotation within the item bob. Determines how far the item tilts in each respective direction

### **Bob Input Velocity Scale**

A tweak parameter that can be used if the bob motion goes out of hand after changing player velocity.

### **Bob Max Input Velocity**

A cap on the velocity value being fed into the bob function, preventing the item from

excessive movement when the character travels at high speeds.

#### **Bob Require Ground Contact**

Determines whether the bob should stay in effect only when the character is on the ground.

#### **Step Min Velocity**

Sets the minimum squared controller velocity at which footstep impacts will occur on the item. The system will be disabled if is zero (default).

#### **Step Softness**

The number of frames over which to even out each footstep impact. A higher number will make the footstep softer (more like regular bob). A lower number will be more 'snappy'

#### **Step Position Force**

A vector relative to the item determining the amount of force it will receive upon each footstep. Note that this parameter is very sensitive. A typical value is less than 0.01.

#### **Step Rotation Force**

Determines the amount of angular force the item will receive upon each footstep. Note that this parameter is very sensitive. A typical value is less than 0.01.

#### **Step Force Scale**

Scales the impact force. It can be used for tweaking the overall force when you are satisfied with the axes' internal relations.

#### **Step Position Balance**

Simulates shifting weight to the left or right foot, by alternating the positional footstep force every other step. Use this to reduce or enhance the effect of limping.

#### **Step Rotation Balance**

Simulates shifting weight to the left or right foot, by alternating the angular footstep force every other step. Use this to reduce or enhance the effect of limping.

## **Third Person**

The main responsibility of the Third Person Perspective Item is the correctly spawn/parent item objects in the hierarchy. Unlike the First Person Perspective Item, the Third Person Perspective Item doesn't need to position the rendered item every frame because the item will be parented to the character's bone and doesn't require as much dynamic movement.

## IK Positioning

All of the animations were made using the rig provided by the character included in the demo scene, Nolan. If the humanoid character that you are using has slightly different proportions compared to the demo rig then the animations won't *precisely* match like they do with the demo character. This is just how the [retargeting](#) feature works within Unity and for the best results you should use animations that match your character's rig.

You can also use IK to correct some of the animation issues by using the *Non Dominant Hand IK Target* field on the Third Person Perspective Item component. This field will use IK to reposition the non-dominant hand of the character. The non-dominant hand is the hand that the item is not parented to. The *Non Dominant Hand IK Hint* field specifies the location of the hint bone. With Unity's IK implementation this is the [elbow location](#). This feature only works with humanoid characters.

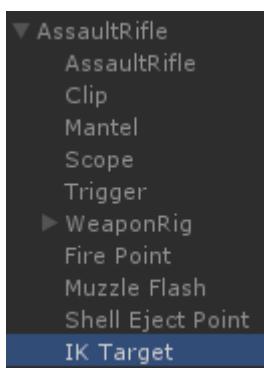
As an example lets use the Ethan character from the Standard Assets. If we pull this character in and attach an assault rifle you'll notice that the left hand is moving through the handle:



This can be corrected by assigning a transform to the *Non Dominant Hand IK Target* field that will use IK to reposition the hand.



In this example a new GameObject was created that is a child of the Assault Rifle.



## Holstering

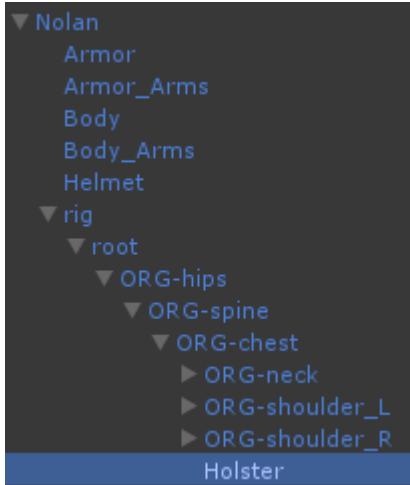
When the item is unequipped you may want the item to be holstered instead of disappearing. This can be done by setting the holster target field within the Third Person Item Perspective component.



This holster target can be set one of two ways:

- Assign the *Holster Target* field directly within the Third Person Perspective Item component. This works best when the item is not added at runtime. The target

Transform should be a child of the character's rig.



- If the *Holster Target* field is null then the *Holster Target ID* field can be used. If this ID is not -1 then when the item is added it will look for a holster target based on the ID of the Object Identifier component. The Object Identifier component should be added to a GameObject that is the child of the character's rig, similar to above.

## Inspected Fields

### Use Parent Humanoid Bone

Should the Object object be spawned based on the character's humanoid bone?

### Parent Humanoid Bone

If using the humanoid bone, specifies which bone to use.

### Non Dominant Hand IK Target

The location of the non-dominant hand which should be placed by the IK implementation.

### Non Dominant Hand IK Target Hint

The location of the non-dominant hand hint which should be placed by the IK implementation.

### Holster Target

The transform that the item should be holstered to when unequipped.

### Holster ID

The ID of the HolsterIdentifier component that the item should be holstered to when unequipped. This id will be used when holster target is null and the the ID is -1.

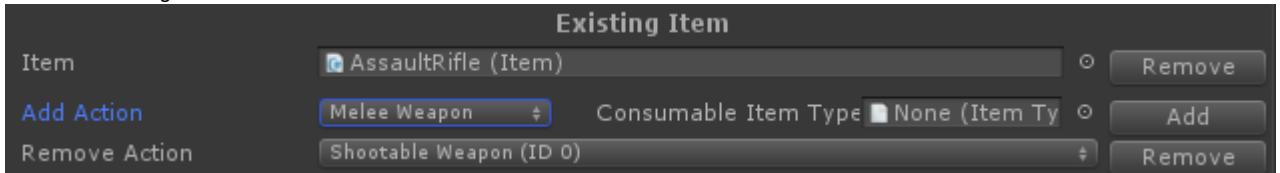
## Actions

The item and perspective components will handle displaying and positioning the item, but the item won't be able to do anything with just those components. The Item Action

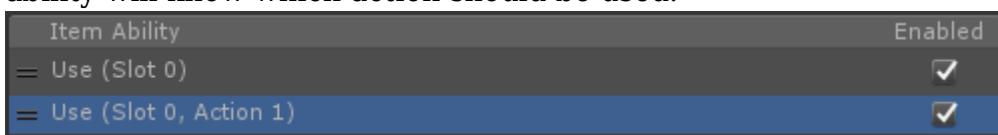
component will allow the item to perform an action, such as firing a bullet, melee slashing, or being thrown. Any number of item actions can be added to the item - the *Item Action ID* allows the item ability to determine which action to perform the interaction on. New actions can be added through the [Item Manager](#).

## Multiple Item Actions Setup

1. Add all of the actions that you'd like your item to be able to perform through the Existing Items tab of the Item Manager. In this example we are using the assault rifle and adding the melee weapon action so the assault rifle can both shoot and be used as a melee object.



2. Make a note of the *Action ID* of the new action that was just added to the item. This ID will be increased sequentially from the previous ID. The assault rifle previously only contained a single action (the Shootable Weapon) so the ID of the Melee Weapon will be 1.
3. Add any [Item Abilities](#) that should use the newly added action. For the Melee Weapon the [Use](#) ability should be added. An *Action ID* of 1 should be specified so the Use ability will know which action should be used.



## Usable

A Usable Item is any item that can be triggered by the [Use](#) ability. The Usable Item will determine common use functionality such as the rate that the item is used or if the item uses an animation event to indicate that it is done being used.

### Inspected Fields

#### Use Rate

The amount of time that must elapse before the item can be used again.

#### Can Equip Empty Item

Specifies if the inventory can equip an item that doesn't have any consumable items left.

#### Face Target

Should the character rotate to face the target during use?

#### Stop Use Ability Delay

The amount of extra time it takes for the ability to stop after use. This is useful for

preventing the item from immediately going back to the idle state after hip firing.

#### **Use Event**

Specifies if the item should wait for the OnAnimatorItemUse animation event or wait for the specified duration before being used. This field uses an [Animation Event Trigger](#).

#### **Use Complete Event**

Specifies if the item should wait for the OnAnimatorItemUseComplete animation event or wait for the specified duration before completing the use. This field uses an [Animation Event Trigger](#).

#### **Force Root Motion Position**

Does the item require root motion position during use?

#### **Force Root Motion Rotation**

Does the item require root motion rotation during use?

#### **Attribute Modifier**

Optionally specify an attribute that the item should affect when in use.

#### **Play Audio On Start Use**

Should the audio play when the item starts to be used? If false it will be played when the item is used. Melee Weapons will generally have a false value.

#### **Use Animator Audio State Set**

Specifies the animator and audio state that should be triggered when the item is used. This field uses an [Animator Audio State Set](#).

## **Flashlight**

The Flashlight Item Action allows the character to turn on or off a flashlight. When the flashlight is used the light GameObject is toggled between an active and inactive state. An optional battery attribute can disable the flashlight after it is empty.

### **Inspected Fields**

#### **Battery Modifier**

The battery attribute that should be modified when the flashlight is active.

# Magic

The Magic Item Action is an extremely flexible Item Action which allows the item to perform a wide variety of magic and spell related actions. The Magic Item relies on sub-actions to perform the actual work: BeginActions, CastActions, ImpactActions, and EndActions. Using the sub-actions the Magic Item can spawn particles, objects, surface effects, audio, etc. New actions can be created by implementing the base class.

## Begin/End Actions

Begin Actions starting playing at the start of the item use until the Use event is sent. When the Use event occurs the Cast Action will trigger. For *Single Use Type* Magic Items the End Actions will start playing immediately after the Use event occurs. *Continuous Use Type* Magic Items will start playing the End Action when the Magic Item is starting to be stopped. Both actions will stop when the item is no longer being used.

Begin and End Actions implement the same base class so the same object can be used for either the start or end of the Magic Item use. These actions allow for starting or ending effects such as starting to fade the character before a teleport or spawn a local particle system that should occur after the actual cast.

### Fade Material

Fades the materials on the character.

### Play Audio Clip

Plays an audio clip.

### Spawn Particle

Spawns a particle.

## Cast Actions

Cast Actions are the primary sub-action within the Magic Item. Cast Actions occur when the Use event is sent and is the main visual that you see for the Magic Item. Common cast actions include spawning a particle, but other cast actions can spawn any arbitrary object or start an [effect](#) on the character.

When a Cast Action collides with an object it can start an Impact Action. This allows for effects to be played at the impact location.

### Physics Cast

Uses the physics system to determine the impacted object.

### Play Audio Clip

Plays an audio clip when the cast is performed.

## **Spawn Object**

Spawns an object when the cast is performed.

## **Spawn Particle**

Spawns a particle when the cast is performed.

## **Spawn Projectile**

Spawns a projectile when the cast is performed.

## **Start Effect**

Starts an effect on the character.

## **Target Impact**

Immediately calls impact on the object at the target position.

## **Teleport**

Teleports the character.

## **Impact Actions**

Impact Actions play an effect at the location determined by the Cast Action. Frequent Impact Actions will be to spawn a particle or change an [Attribute](#) value.

### **Add Force**

Adds a force to the impacted object.

### **Add Torque**

Adds a torque to the impacted object.

### **Damage**

Damages the impacted object.

### **Heal**

Heals the impacted object.

### **Modify Attribute**

Modifies the specified attribute on the impacted object.

## Play Audio Clip

Plays an AudioClip on the impacted object.

## Ricochet

The Ricochet action will cast a new CastAction for nearby objects, creating a ricochet effect.

## Spawn Particle

Spawns a ParticleSystem upon impact.

## Spawn Surface Effect

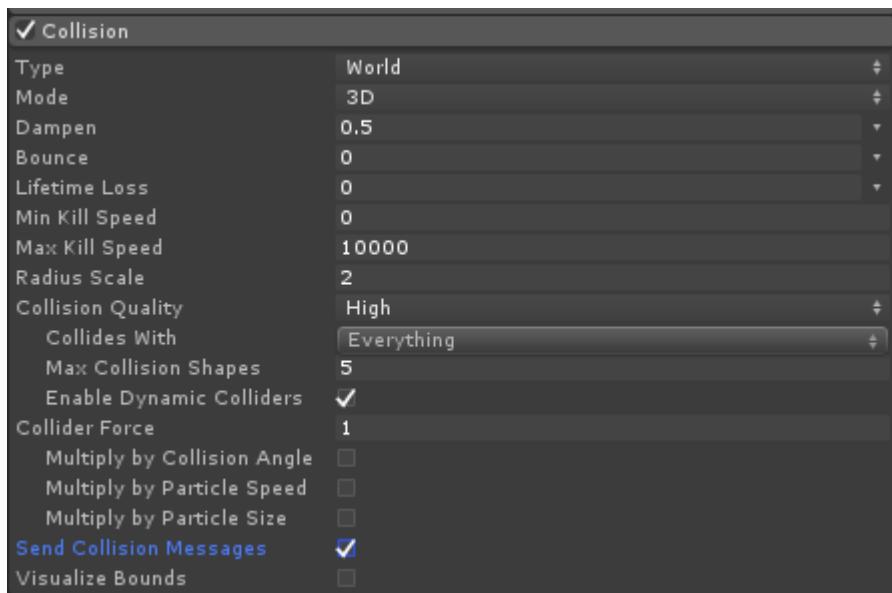
Spawns a SurfaceEffect upon impact. The SurfaceImpact object is specified on the MagicItem.

## Collision

When a Cast Action effect plays it can collide with other objects to play an Impact Action. This collision can occur with one of the following methods:

### Particle Collision

Unity's Particle System has the option of sending [collision messages](#) when the particle collides with another object. In order for the collision message to be received the Magic Particle component must be added to the Particle System that can cause the collision.



## Projectile

The [Magic Projectile](#) component will trigger an impact when it collides with another object. The Magic Particle component uses the [Trajectory Object](#) component to do its actual movement.

## **Raycast**

If the Physics Cast Cast Action is used then a raycast will be performed when the Cast Action is run. This raycast will execute the Impact Actions immediately.

## **Inspected Fields**

### **Require Grounded**

Is the character required to be on the ground?

### **Direction**

The direction of the cast.

- *None*: The cast has no movement.
- *Forward*: The cast should move in the forward direction.
- *Target*: The cast should move towards a target.
- *Indicate*: The cast should move towards an indicated position.

### **Use Look Source**

Should the look source be used when determining the cast direction?

### **Max Distance**

The maximum distance of the movement cast direction.

### **Radius**

The radius of the movement cast direction.

### **Detect Layers**

The layers that the movement directions can collide with.

### **Max Angle**

The maximum angle that the target object can be compared to the character's forward direction.

### **Max Collision Count**

The maximum number of colliders that can be detected by the target cast.

### **Target Count**

The number of objects that a single cast should cast.

## **Surface Indicator**

The transform used to indicate the surface. Can be null.

## **Surface Indicator Offset**

The offset when positioning the surface indicator.

## **Use Type**

Specifies how often the cast is used.

- *Single*: The cast should occur once per use.
- *Continuous*: The cast should occur ever use update.

## **Min Continuous Use Duration**

The minimum duration of the continuous use type. If a value of -1 is set then the item will be stopped when the stop is requested.

## **Continuous Cast**

Should the continuous use type cast every update?

## **Use Amount**

The amount of the ItemIdentifier that should be used each cast.

## **Interrupt Source**

Specifies when the cast should be interrupted.

- *Movement*: The cast should be interrupted when the character moves.
- *Damage*: The cast should be interrupted when the character takes damage.

## **Surface Impact**

The SurfaceImpact of the cast.

## **Can Stop Substate Index Addition**

The value to add to the Item Substate Index when the item can stop.

# **Melee Weapon**

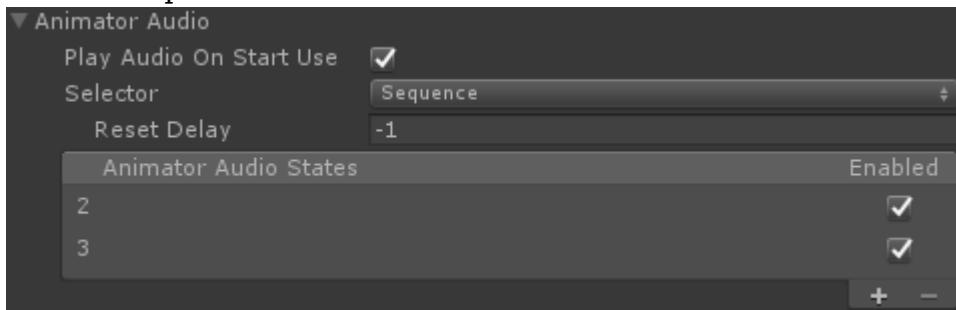
The Melee Weapon Item Action allows the character to attack using a melee weapon such as a sword or a baseball bat. Using the [Animator Audio State Set](#) from the parent [Usable](#) component the melee weapon can chain animations together allowing for combos. The

First/Third Person Melee Weapon Properties component will also be attached to the item and this component contains the properties that are specific for that perspective.

## Attack Troubleshooting

If your melee weapon gets stuck after being used the first time ensure you have checked the following:

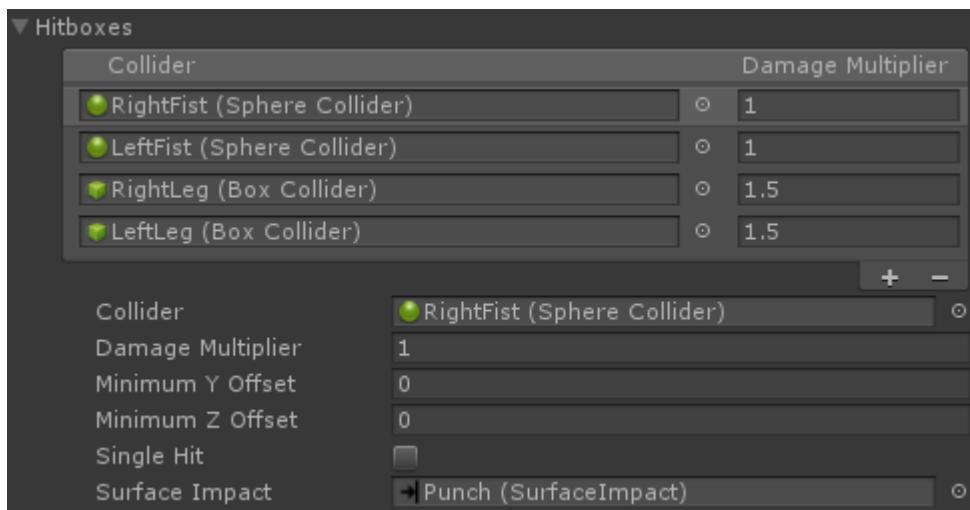
- The correct Use [Animator Audio State Set](#) elements are set. These options are available under the Use foldout. The demo animator controller has two different attack types so two different states must be set. The animator will get stuck if there is only one state specified.



- If only one Use Animator Audio State Set element is specified, *Allow Attack Combos* should be deselected under the Melee foldout.
- Ensure the correct [Animation Event Triggers](#) are set. New animation events may need to be added with new attack animations.

## Hitboxes

The First/Third Person Melee Weapon Perspective contains a “Hitboxes” foldout and within this foldout you can specify the hitboxes that the weapon should use. For most melee weapons you’ll only have one hitbox - a collision box representing the area that the area of the melee weapon that can impact with other objects. In the case of the body item there are four hitboxes: two for the character’s hands and two for the character’s feet. This allows the character to punch or kick using the same item. Hitboxes can use a SphereCollider, BoxCollider, or CapsuleCollider.



In this screenshot the right fist is selected. A *Damage Multiplier* can be specified which will multiply the Melee Weapon’s *Damage Amount* by the specified value. This allows certain colliders to inflict more damage compared to others, such as a kick inflicting more damage.

than a punch.

The *Minimum Y Offset* and *Minimum Z Offset* fields specify the relative offset that the collider position needs to be above in order to inflict damage. This is useful for the leg collider as it prevents a hit from occurring when the foot is on the ground. *Single Hit* should be enabled if the collider should only do damage once per melee attack. The *Surface Impact* field allows you to specify a [Surface Impact](#) for when the melee weapon collides with another object.

You'll see a colored gizmo rendered in the scene view when the item is selected in the hierarchy:

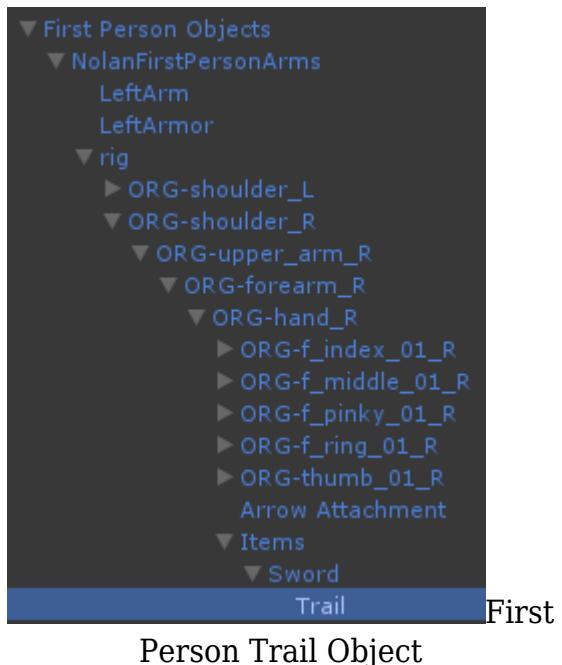


The color will interpolate between green, yellow, and red. The lower the *Damage Multiplier* value the more green the hitbox will be. The higher the *Damage Multiplier* the more red the hitbox will be.

## Melee Trail

The melee trail object can dynamically draw a smooth curve indicating the previous position of the melee object. This is most often used when the Melee Weapon is attacking, but it can also always be shown (this is set with the *Trail Visibility* field). A new melee trail object should be created with the [Object Manager](#):

1. Open the [Object Manager](#) and create a new object of type Melee Trail.
2. Assign the new trail object to the Melee Weapon's *Trail* field.
3. Create a child GameObject that specifies the location that the trail should spawn at.  
This object should be created for each perspective and should be a child of the perspective's visible object.



First Person Trail Object

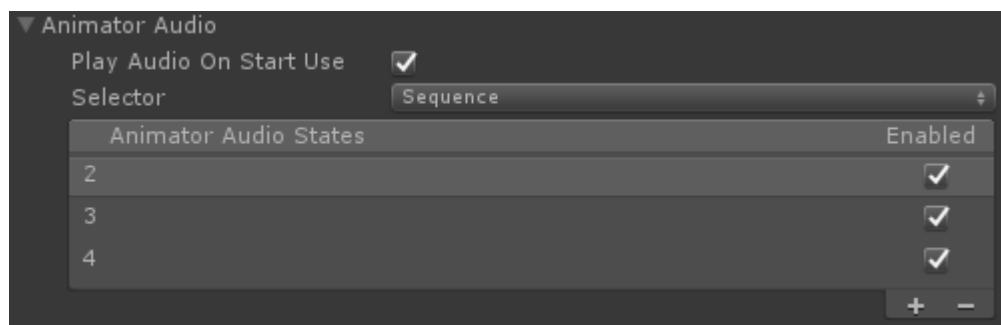


Third Person Trail Object

4. Specify the newly created GameObject under the *Trail Location* field of the First Person and Third Person Melee Weapon Properties.
5. Specify when the trail should show with the *Trail Visibility* field on the Melee Item.

## Combos

If *Allow Attack Combos* is enabled the weapon can transition to the next attack state before the current attack state has completed. A new combo can be started after the *Use Event* has been triggered and before the *Use Complete Event* has been triggered. Lets say that you have the following states listed for your Melee Weapon:



Lets say that the state with an Item Substate Index of 2 is currently playing and the *Use Event* is triggered after 1 second. The *Use Complete Event* is then triggered 0.5 seconds after the *Use Event*. With this setup the following flow will occur:

- The state is started at a time of 0 seconds.
- Another attack input is received at 0.6 seconds, the current state is not interrupted because the *Use Event* has not triggered.
- The *Use Event* is triggered at 1 second.
- Another attack input is received at 1.1 seconds, the current state transitions to the

next state (with an Item Substate Index of 3) because at this point the *Use Event* has triggered.

- Another attack input is received at 1.5 seconds, the current state is not interrupted because the *Use Event* has not triggered.
- The *Use Event* is triggered at 2.1 seconds.
- The *Use Complete Event* is triggered at 2.6 seconds.

## Impact Callback

When the melee weapon collides with another object the “OnObjectImpact” event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. This event allows you to add new functionality when the impact occurs without having to change the class at all. The following example will subscribe to this event from a new component:

```
using UnityEngine;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Initialize the default values.
    /// </summary>
    public void Awake()
    {
        EventHandler.RegisterEvent<float, Vector3, Vector3,
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);
    }

    /// <summary>
    /// The object has been impacted with another object.
    /// </summary>
    /// <param name="amount">The amount of damage taken.</param>
    /// <param name="position">The position of the damage.</param>
    /// <param name="forceDirection">The direction that the object
took damage from.</param>
    /// <param name="attacker">The GameObject that did the
damage.</param>
    /// <param name="attackerObject">The object that did the
damage.</param>
    /// <param name="hitCollider">The Collider that was hit.</param>
    private void OnImpact(float amount, Vector3 position, Vector3
forceDirection, GameObject attacker, object attackerObject, Collider
hitCollider)
    {
        Debug.Log(name + " impacted by " + attacker + " on collider "
+ hitCollider + ".");
    }

    /// <summary>
```

```
/// The GameObject has been destroyed.  
/// </summary>  
public void OnDestroy()  
{  
    EventHandler.UnregisterEvent<float, Vector3, Vector3,  
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);  
}  
}
```

## Inspected Fields

### Require In Air Melee Ability In Air

Does the weapon require the In Air Melee Use Item Ability in order to be used while in the air?

### Require Counter Attack Ability

Does the weapon require the Melee Counter Attack Item Ability in order to be used?

### Consumable Item Definition

The Item Definition that is consumed by the item (can be null). This Item Definition should be specified if the melee weapon use is finite.

### Aim Item Substate Index Addition

The value to add to the Item Substate Index when the character is aiming.

### Max Collision Count

The maximum number of collision points which the melee weapon can make contact with.

### Forward Shield Sensitivity

The sensitivity amount for how much the character must be looking at the hit object in order to detect if the shield should be used (-1 is the most sensitive and 1 is least).

### Single Hit

When the weapon attacks should only one hit be registered per use?

### Multi Hit Frame Count

If multiple hits can be registered, specifies the minimum frame count between each hit.

### Can Hit Delay

The delay after the weapon has been used when a hit can be valid.

### **Allow Attack Combos**

Can the next use state play between the ItemUsed and ItemUseComplete events?

### **Impact Layers**

A LayerMask of the layers that can be hit by the weapon.

### **Trigger Interaction**

Specifies if the melee weapon can detect triggers.

### **Damage Amount**

The amount of damage done to the object hit.

### **Impact Force**

The amount of force to apply to the object hit.

### **Impact Force Frames**

The number of frames to add the impact force to.

### **Impact State Name**

The name of the state to activate upon impact.

### **Impact State Disable Timer**

The number of seconds until the impact state is disabled. A value of -1 will require the state to be disabled manually.

### **Surface Impact**

The Surface Impact triggered when the weapon hits an object.

### **Apply Recoil**

Should recoil be applied when the weapon hits an object?"

### **Recoil Animator Audio State Set**

Specifies the animator and audio state from a recoil.

### **Trail**

A reference to the trail prefab that should be spawned.

## **Trail Visibility**

Specifies when the melee weapon trail should be shown.

- *Attack*: The trail is only visible while attacking.
- *Always*: The trail is always visible.

## **Trail Spawn Delay**

The delay until the trail should be spawned after it is visible.

## **Attack Stop Trail Event**

Specifies if the item should wait for the OnAnimatorStopTrail animation event or wait for the specified duration before stopping the trail during an attack.

## **On Impact Event**

Unity event invoked when the weapon hits another object.

# **Shootable Weapon**

The Shootable Weapon Item Action allows the character to attack using a ranged object such as an assault rifle or a bow and arrow. The Shootable Weapon can attack using a raycast (hitscan) or a projectile. The weapon can also reload when the object no longer has any ammo. The First/Third Person Shootable Weapon Properties component will also be attached to the weapon and it contains properties that are specific for that perspective.

## **Firing Troubleshooting**

If you are unable to fire a new Shootable Weapon ensure you have checked the following:

- The inventory has ammo. An easy way to ensure this is true is to add your Consumable Item Definition to the Default Loadout of the inventory.
- The use event is being fired. The weapon won't try to be fired until the used event is triggered and this can be specified with the *Use Event Animation Event Trigger* on the shootable weapon.
- The weapon is facing in the fire direction. The weapon won't fire until after it is facing in the direction that it should be fired. This sensitivity can be adjusted within the *Look Sensitivity* field of the First/Third Person Shootable Weapon Properties component. If the sensitivity is set to -1 then the weapon will always fire no matter which direction the weapon is facing.

## **Muzzle Flash**

When the weapon fires a muzzle flash can be shown. This muzzle flash uses a mesh that will fade with time. A light can also be attached which will also reduce intensity as time elapses. A muzzle flash can be created with the [Object Manager](#).

1. Open the [Object Manager](#) and create a new object of type Muzzle Flash.
2. Assign the new object to the Shootable Weapon's *Muzzle Flash* field.
3. When a Shootable Weapon is created the "Muzzle Flash" GameObject will be created by default on the perspective's visible object. This GameObject should be specified for the *Muzzle Flash Location* field of the First Person and Third Person Shootable Weapon Properties.

## Smoke

When the weapon fires a smoke particle system can be spawned from the weapon. This particle system will be disabled when it has completed. Smoke can be created with the [Object Manager](#):

1. Open the [Object Manager](#) and create a new object of type Smoke.
2. Assign the new object to the Shootable Weapon's *Smoke* field.
3. Create a new GameObject as a child of the perspective's visible object. This GameObject should be at the same level in the hierarchy as the *Fire Point*. This new GameObject for the *Smoke Location* field of the First Person and Third Person Shootable Weapon Properties.

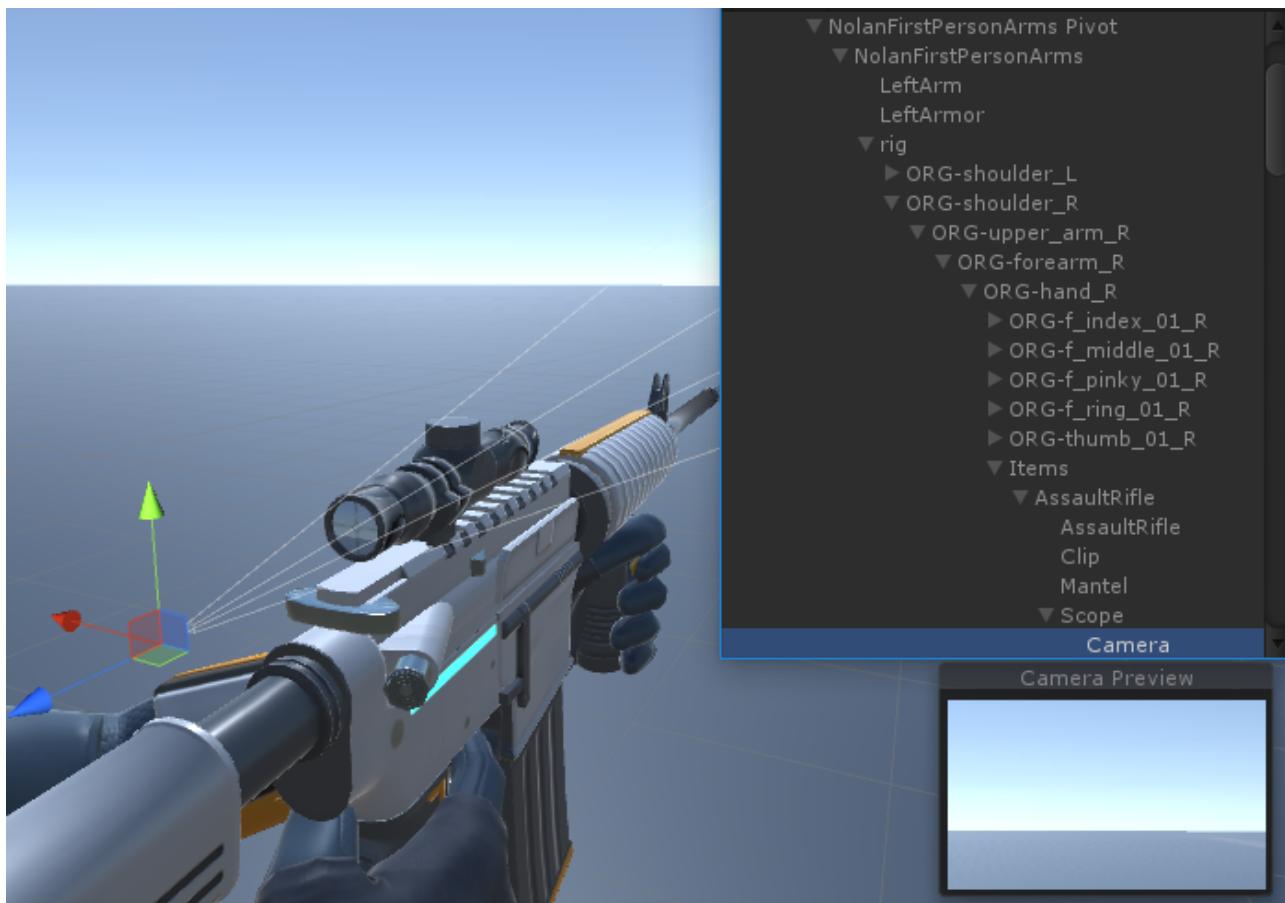
## Reloadable Clip

When the weapon is reloaded the clip that is attached to the weapon can be removed during the reload animation. If the *Reloadable Clip Attachment* Transform is specified within the First/Third Person Shootable Weapon Properties then the clip will be reparented to the object specified when the *Reload Detach Event* triggers. The clip will then be parented back to the original transform when the *Reload Attach Event* is triggered.

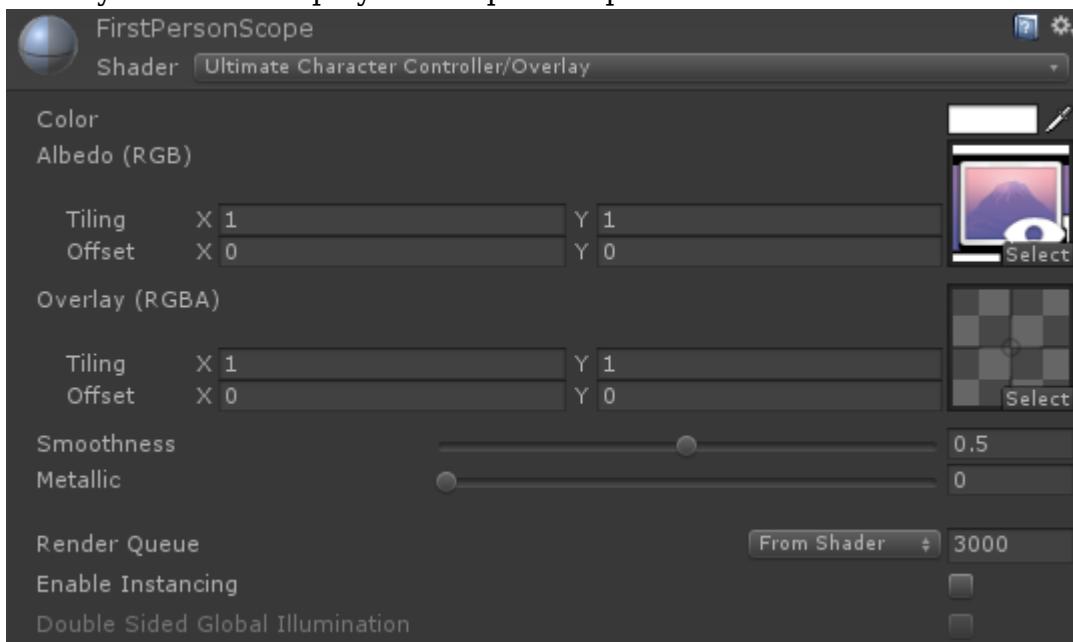
## Scope

A scope can optionally be specified which uses a separate camera to show a zoomed in view of the target. This scope can be setup by doing the following:

1. Add the scope camera to your item. This camera should be added to both the first and third person item under the rig. It **should not** be added to the item object that contains the Shootable Weapon component under the character/Items parent.



2. Reference this camera in the *Scope Camera* field of the First/Third Person Shootable Weapon Properties component.
3. Decide when you want the camera to be active. By default the camera will be active but you can only activate it when the character is aiming by toggling the *Disable Scope Camera On No Aim* field within the Shootable Weapon component.
4. Create a new [Render Texture](#) asset and point the scope camera's Target Texture field to this new render texture.
5. Create a new material that uses the render texture. For the assault rifle we created the material below (*Albedo* specifies the render texture). This material uses a custom overlay shader to display the scope on top of the texture.

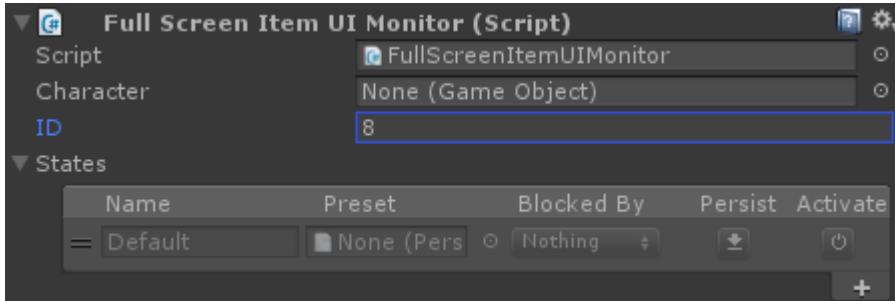


6. Apply the material to the object that you'd like to use as the scope. The scope is now setup.

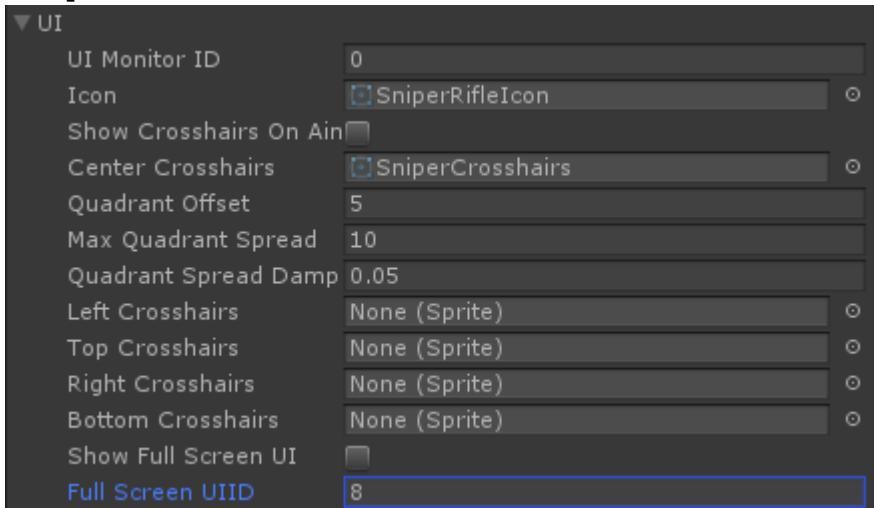
## Full Screen UI

When the sniper rifle aims a full screen UI appears that blocks the game view besides the center portion. The full screen UI can be setup by performing the following steps:

1. Add the Full Screen UI Monitor component to the UI object that you'd like to enable when the weapon should aim. In the ID field for this component you should specify the unique ID of the FullScreenUI object. In our case we will just be using this UI object for the sniper rifle so for the ID we used the Animator ID of the sniper, a value of 8.



2. On the sniper rifle's Item component set the *Full Screen UI ID* to the same ID value that was used in step 1.



3. When the *Show Full Screen UI* toggle is enabled the Full Screen UI Monitor component will activate. This can be set using the [state system](#) so when the Aim ability is activated it enables the toggle.
4. If you are adding a sniper ensure you have added a view type state that changes the camera state (such as offset or field of view) when the sniper aims. In the demo scene this state is called AimSniperRifle.

## Impact Callback

When the hitscan or projectile collides with another object the "OnObjectImpact" event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. This event allows you to add new functionality when the impact occurs without having to change the class at all. The following example will subscribe to this event from a new component:

```
using UnityEngine;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    /// <summary>
```

```

/// Initialize the default values.
/// </summary>
public void Awake()
{
    EventHandler.RegisterEvent<float, Vector3, Vector3,
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);
}

/// <summary>
/// The object has been impacted with another object.
/// </summary>
/// <param name="amount">The amount of damage taken.</param>
/// <param name="position">The position of the damage.</param>
/// <param name="forceDirection">The direction that the object
took damage from.</param>
/// <param name="attacker">The GameObject that did the
damage.</param>
/// <param name="attackerObject">The object that did the
damage.</param>
/// <param name="hitCollider">The Collider that was hit.</param>
private void OnImpact(float amount, Vector3 position, Vector3
forceDirection, GameObject attacker, object attackerObject, Collider
hitCollider)
{
    Debug.Log(name + " impacted by " + attacker + " on collider "
+ hitCollider + ".");
}

/// <summary>
/// The GameObject has been destroyed.
/// </summary>
public void OnDestroy()
{
    EventHandler.UnregisterEvent<float, Vector3, Vector3,
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);
}
}

```

## Inspected Fields

### Consumable ItemDefinition

The Item Definition that is consumed by the item. If this is null the weapon will have unlimited ammo.

### Fire Mode

The mode in which the weapon fires multiple shots.

- *SemiAuto*: Fire discrete shots, don't continue to fire until the player fires again. | 198

- *FullAuto*: Keep firing until the ammo runs out or the player stops firing.
- *Burst*: Keep firing until the burst rate is zero.

### **Fire Type**

Specifies when the weapon should be fired.

- *Instant*: Fire the shot immediately.
- *ChargeAndFire*: Wait for the Used callback and then fire.
- *ChargeAndHold*: Fire as soon as the Use button is released.

### **Min Charge Length**

If using a charge FireType, the minimum amount of time that the weapon must be charged for in order to be fired.

### **Full Charge Length**

If using a charge FireType, the amount of time that the weapon must be charged for in order to be fully fired.”

### **Charge Item Substate Parameter Value**

The Item Substate parameter value when charging.

### **Min Charge Strength**

If using a charge FireType, the minimum amount strength that the weapon will use if it isn't fully charged.

### **Charge Audio Clip Set**

A set of AudioClips that can be played when the weapon is charging.

### **Fire Count**

The number of rounds to fire in a single shot.

### **Burst Count**

If using the Burst FireMode, specifies the number of bursts the weapon can fire.

### **Burst Delay**

If using the Burst FireMode, specifies the delay before the next burst can occur.

### **Spread**

The random spread of the bullets once they are fired.

## **Fire In Look Source Direction**

Should the weapon fire in the LookSource direction? If false the weapon will be fired in the direction of the weapon.

## **Projectile**

Optionally specify a projectile that the weapon should use. If no projectile is specified then the weapon will fire a raycast in order to determine if it hit any objects.

## **Projectile Fire Velocity Magnitude**

The magnitude of the projectile velocity when fired. The direction is determined by the fire direction.

## **Projectile Visibility**

Specifies when the projectile should become visible.

- *OnFire*: The projectile is only visible when the weapon is being fired.
- *OnAim*: The projectile is visible when the character is aiming.
- *Always*: The projectile is always visible when the item is equipped.

## **Projectile Start Layer**

The layer that the projectile should occupy when initially spawned.

## **Projectile Fired Layer**

The layer that the projectile object should change to after being fired.

## **Layer Change Delay**

The amount of time after the object has been fired to change the layer.

## **Projectile Enable Delay After Other Use**

The amount of time after another item has been used that the projectile should be enabled again.

## **Dry Fire Item Substate Parameter Value**

The Item Substate parameter value when the weapon tries to fire but is out of ammo.

## **Dry Fire Audio Clip Set**

A set of AudioClips that can be played when the weapon is out of ammo.

## **Hitscan Fire Range**

The maximum distance in which the hitscan fire can reach.

### **Max Hitscan Collision Count**

The maximum number of objects the hitscan cast can collide with.

### **Impact Layers**

A LayerMask of the layers that can be hit when fired at.

### **Hitscan Trigger Interaction**

Specifies if the hitscan can detect triggers.

### **Damage Amount**

The amount of damage to apply to the hit object.

### **Impact Force**

The amount of much force to apply to the hit object.

### **Impact Force Frames**

The number of frames to add the impact force to.

### **Impact State Name**

The name of the state to activate upon impact.

### **Impact State Disable Timer**

The number of seconds until the impact state is disabled. A value of -1 will require the state to be disabled manually.

### **Surface Impact**

The Surface Impact triggered when the weapon hits an object.

### **Clip Size**

The number of rounds in the clip.

### **Auto Reload**

Specifies when the item should be automatically reloaded.

### **Reload Type**

Specifies how the clip should be reloaded.

- *Full*: Reload the entire clip.
- *Single*: Reload a single bullet.

### **Reload Can Camera Zoom**

Can the camera zoom during a reload?

### **Reload Crosshairs Spread**

Should the crosshairs spread during a recoil?

### **Reload Animator Audio State Set**

Specifies the animator and audio state from a reload.

### **Reload Event**

Specifies if the item should wait for the OnAnimatorItemReload animation event or wait for the specified duration before reloading.

### **Reload Complete Event**

Specifies if the item should wait for the OnAnimatorItemReloadComplete animation event or wait for the specified duration before completing the reload.

### **Reload Complete Audio Clip Set**

The clip that should be played after the item has finished reloading.

### **Reload Detach Attach Clip**

Should the weapon clip be detached and attached when reloaded?

### **Reload Detach Event**

Specifies if the item should wait for the OnAnimatorItemReloadDetachClip animation event or wait for the specified duration before detaching the clip from the weapon.

### **Reload Show Projectile Event**

Specifies if the item should wait for the OnAnimatorItemReloadShowProjectile animation event or wait for the specified duration before showing the projectile.

### **Reload Attach Projectile Event**

Specifies if the item should wait for the OnAnimatorItemReloadAttachProjectile animation event or wait for the specified duration before parenting the projectile to the fire point.

### **Reload Drop Clip**

The prefab that is dropped when the character is reloading.

### **Reload Clip Layer Change Delay**

The amount of time after the clip has been removed to change the layer.

### **Reload Clip Target Layer**

The layer that the clip object should change to after being reloaded.

### **Reload Drop Clip Event**

Specifies if the item should wait for the OnAnimatorItemReloadDropClip animation event or wait for the specified duration before dropping the clip from the weapon.

### **Reload Attach Event**

Specifies if the item should wait for the OnAnimatorItemReloadAttachClip animation event or wait for the specified duration before attaching the clip back to the weapon.

### **Position Recoil**

The amount of positional recoil to add to the item.

### **Rotation Recoil**

The amount of rotational recoil to add to the item.

### **Position Camera Recoil**

The amount of positional recoil to add to the camera.

### **Rotation Camera Recoil**

The amount of rotational recoil to add to the camera.

### **Camera Recoil Accumulation**

The percent of the recoil to accumulate to the camera's rest value.

### **Localize Recoil Force**

Is the recoil force localized to the direct parent?

### **Muzzle Flash**

A reference to the muzzle flash prefab.

### **Pool Muzzle Flash**

Should the muzzle flash be pooled? If false a single muzzle flash object will be used.

## **Shell**

A reference to the shell prefab.

## **Shell Velocity**

The velocity that the shell should eject at.

## **Shell Torque**

The torque that the projectile should initialize with.

## **Shell Eject Delay**

Eject the shell after the specified delay.

## **Smoke**

A reference to the smoke prefab.

## **Smoke Spawn Delay**

Spawn the smoke after the specified delay.

## **Tracer**

Optionally specify a tracer that should appear when the hitscan weapon is fired.

## **Tracer Spawn Delay**

Spawn the tracer after the specified delay.

## **Disable Scope Camera On No Aim**

Should the camera's scope camera be disabled when the character isn't aiming?

## **On Hitscan Impact Event**

Unity event invoked when the hitscan hits another object.

# **Throwable Item**

The Throwable Item Item Action allows the character to throw an object such as a grenade or baseball. The [Grenade Item](#) is a subclass that allows the pin from a grenade to be removed before the object is thrown.

The GameObject that the Throwable Item attaches to is not the actual object that is thrown - the *Thrown Object* field specifies the object that should actually be thrown instead. The Throwable Item also allows the character to [show a trajectory](#) of the path that the object will follow when it is thrown. The First/Third Person Thrower Properties

component will also be attached to the item and this component contains the properties that are specific for that perspective.

## Inspected Fields

### Thrown Object

The object that is thrown.

### Consumable Item Definition

The Item Definition that is consumed by the item.

### Disable Visible Object

Should the visible object be disabled?

### Activate ThrowableObject Event

Specifies if the item should wait for the OnAnimatorActivateThrowableObject animation event or wait for the specified duration before activating the throwable object.

### Throw On Stop Use

Should the object be thrown when the stop use method is called?

### Velocity

The starting velocity of the thrown object.

### Start Layer

The layer that the item should occupy when initially spawned.

### Thrown Layer

The layer that the thrown object should change to after being thrown.

### Layer Change Delay

The amount of time after the object has been thrown to change the layer.

### Damage Amount

The amount of damage applied to the object hit by the thrown object.

### Impact Layers

The layers that the thrown object can collide with.

## **Surface Impact**

The Surface Impact triggered when the object hits another object.

### **Impact Force**

The amount of force to apply to the object hit.

### **Impact Force Frames**

The number of frames to add the impact force to.

### **Impact State Name**

The name of the state to activate upon impact.

### **Impact State Disable Timer**

The number of seconds until the impact state is disabled. A value of -1 will require the state to be disabled manually.

### **Reequip Event**

Specifies if the item should wait for the OnAnimatorReequipThrowableItem animation event or wait for the specified duration before reequipping.

### **Reequip Item Substate Parameter Value**

The value of the Item Substate Animator parameter when the item is being reequipped.

### **Show Trajectory On Aim**

Should the item's trajectory be shown when the character aims?"

### **Trajectory Offset**

The offset of the trajectory visualization relative to the trajectory transform set on the Throwaway Item Properties.

# **Grenade Item**

The Grenade Item extends the Throwaway Item to allow a pin to be removed just before the object is thrown. The grenade will also start to cook when the item is used, allowing for the grenade to explode while still in the character's hand if they don't throw it fast enough.

## **Inspected Fields**

### **Animate Pin Removal**

Is the pin removal animated?

## **Remove Pin Event**

Specifies if the item should wait for the OnAnimatorItemRemovePin animation event or wait for the specified duration before removing the pin from the object?

# **Shield**

The Shield Item Action will protect the character from taking damage. The shield can be invincible or slowly degrade as it takes more and more damage. The area that will protect the character from damage is defined by the collider that is attached to the first or third person shield object.

## **Inspected Fields**

### **Require Aim**

Does the shield only protect the player when the character is aiming?

### **Absorption Factor**

Determines how much damage the shield absorbs. A value of 1 will absorb all of the damage, a value of 0 will not absorb any of the damage.

### **Absorb Explosions**

Should the shield absorb damage caused by explosions?

### **Apply Impact**

Should an impact be applied when the weapon is hit by another object?

### **Impact Animator Audio State Set**

Specifies the animator and audio state for when the shield is impacted by another object.

### **Impact Complete Event**

Specifies if the item should wait for the OnAnimatorItemImpactComplete animation event or wait for the specified duration before completing the impact.

### **Durability Attribute Name**

The name of the shield's durability attribute. When the durability reaches 0 the shield will not absorb any damage.

### **Drop When Durability Depleted**

Should the item be dropped from the character when the durability is depleted?

# Dual Wielding

The inventory and item system allows for multiple items to be equipped at the same time. As each item is equipped they will occupy a slot within the inventory. Because there isn't a limit to the number of slots that a single character can have there isn't a limit to the number of items that can be equipped at the same time. In most cases the most number of items that a character will have equipped at any one time is two so this page will setup that scenario.

For this example we are going to setup a Sword and Shield that you can dual wield. The initial item setup is the same as a single-wield item. Both items should first be created with the [Item Manager](#). The Sword should specify a *Slot ID* of 0 (for the right hand) within the Item component, and the shield should specify a *Slot ID* of 1 (for the left hand).

Once both items have been created a new [Item Set](#) can be created that uses both items:



When the Item Set is active the [Equip Unequip](#) ability will try to equip the Sword in Slot 0 and the Shield in Slot 1. In most cases this is all that you need to do, however for the first person perspective you have some additional options based on which parent Object you are using.

For the first person perspective all of your items can share the *Object* within the First Person Perspective Item component. If however your item can be equipped with other items it is useful to have the item use its own object so each hand can operate independently. This was the approach that we took for the Sword Shield setup. If you look at the First Person Objects hierarchy you'll see that the Shield is parented to just the Left Arms:



When the Shield is equipped the *NolanFirstPersonLeftArms* object will activate and be controlled by the shield's First Person Perspective Item component. This allows the right arms to operate independently.

# Runtime Pickup

There are two ways an item can be added to the character through the [Item Manager](#):

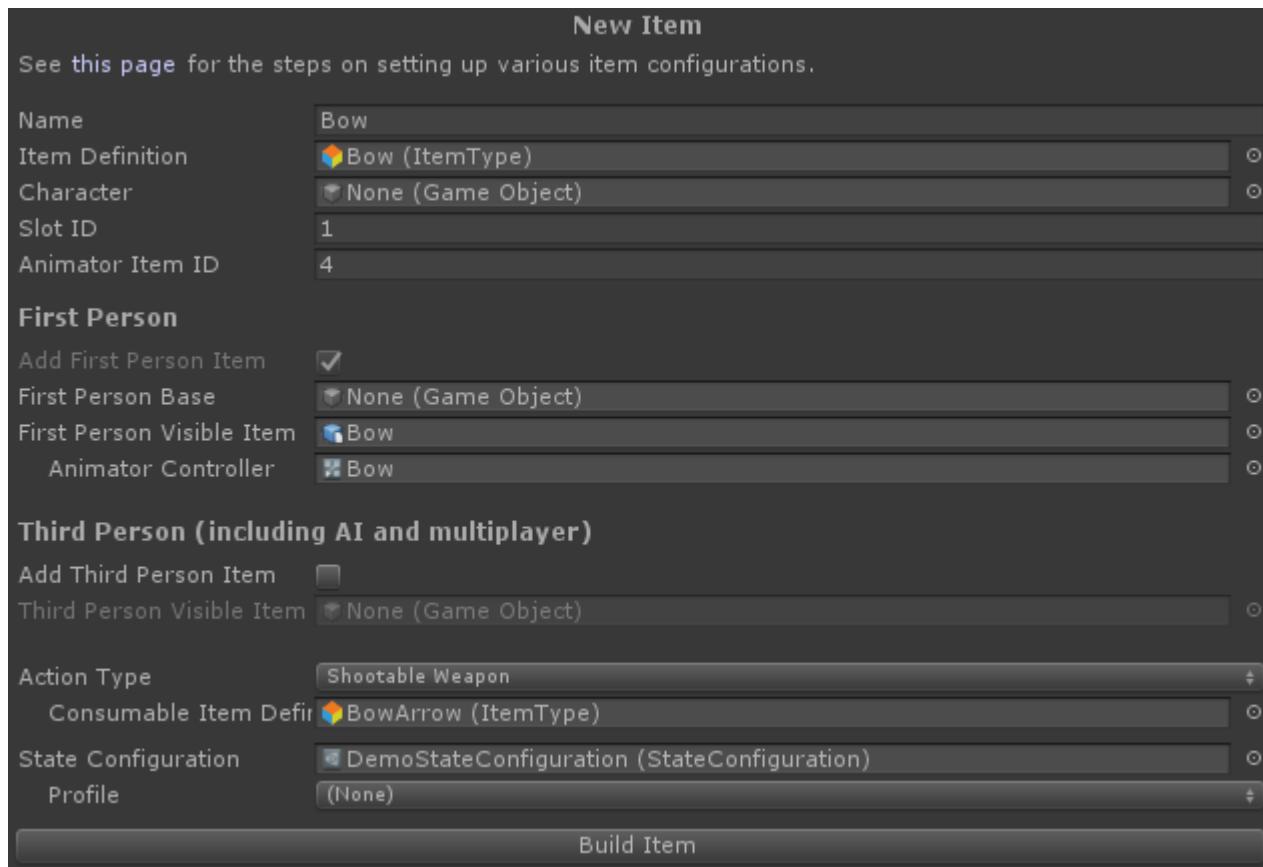
1. Create the item with the Item Manager and assign the item to the character. This is the most common method and it requires all possible items to be added to the character at edit time.
2. Create the item with the Item Manager and don't assign the item to a character, resulting in a prefab of the item. This method is extremely useful for RPGs where there are many items to configure.

This page will walk through the second method. This page is split up into two sections, one for first person items and the other for third person items. If you are using both perspectives then both a first and third person item should be created in step 1.

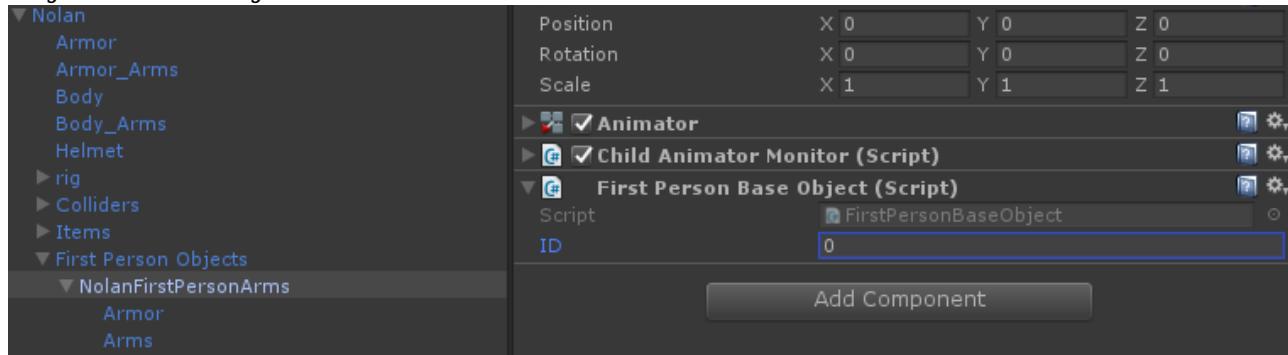
## First Person Perspective

1. Open the Item Manager. Specify the following fields:
  - *Name*: The name of the item.
  - *Item Definition*: The Item Definition that provides a unique id for the item.
  - *Slot ID*: The slot that the item will occupy.
  - *Animator Item ID*: Optional, the Item ID used within the Animator.
  - *Add First Person Item*: True.
  - *First Person Base*: Optional, the arm object that the item should be parented to. If this field is left blank (which is the most common) then the item will search for the parent based off of the FirstPersonBaseObject component ID.
  - *First Person Visible Item*: The object rendered within the game.
  - *First Person Visible Item Animator Controller*: Optional, the Animator Controller for the First Person Visible Item.
  - *Action Type*: The type of Item Action to add to the item.

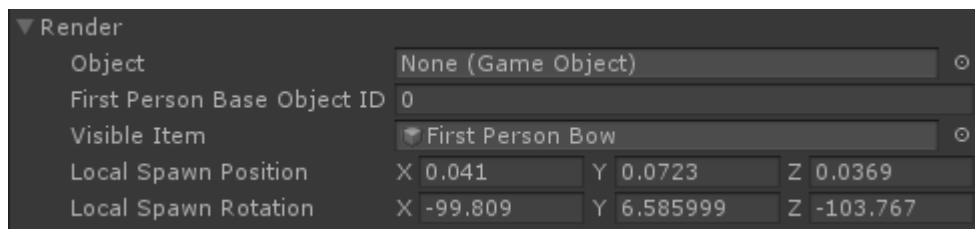
The following screenshot provides a sample setup:



2. Build the item to a new prefab. Select that prefab within the project.
3. Because we left the *First Person Base* field blank we need to specify which object the *Visible Item* will be parented to. We are going to specify a value of 0 for the *First Person Base Base* field of the First Person Perspective Item component because we want to use the first person arms that are already created under the First Person Objects GameObject:



4. The spawn position/rotation now needs to be specified. The best method for determining this value is to pickup the item with the Item Pickup component and then adjust the visible item's transform while in play mode. Save the position/rotation values and then copy them to the *Local Spawn Position* and *Local Spawn Rotation* fields:

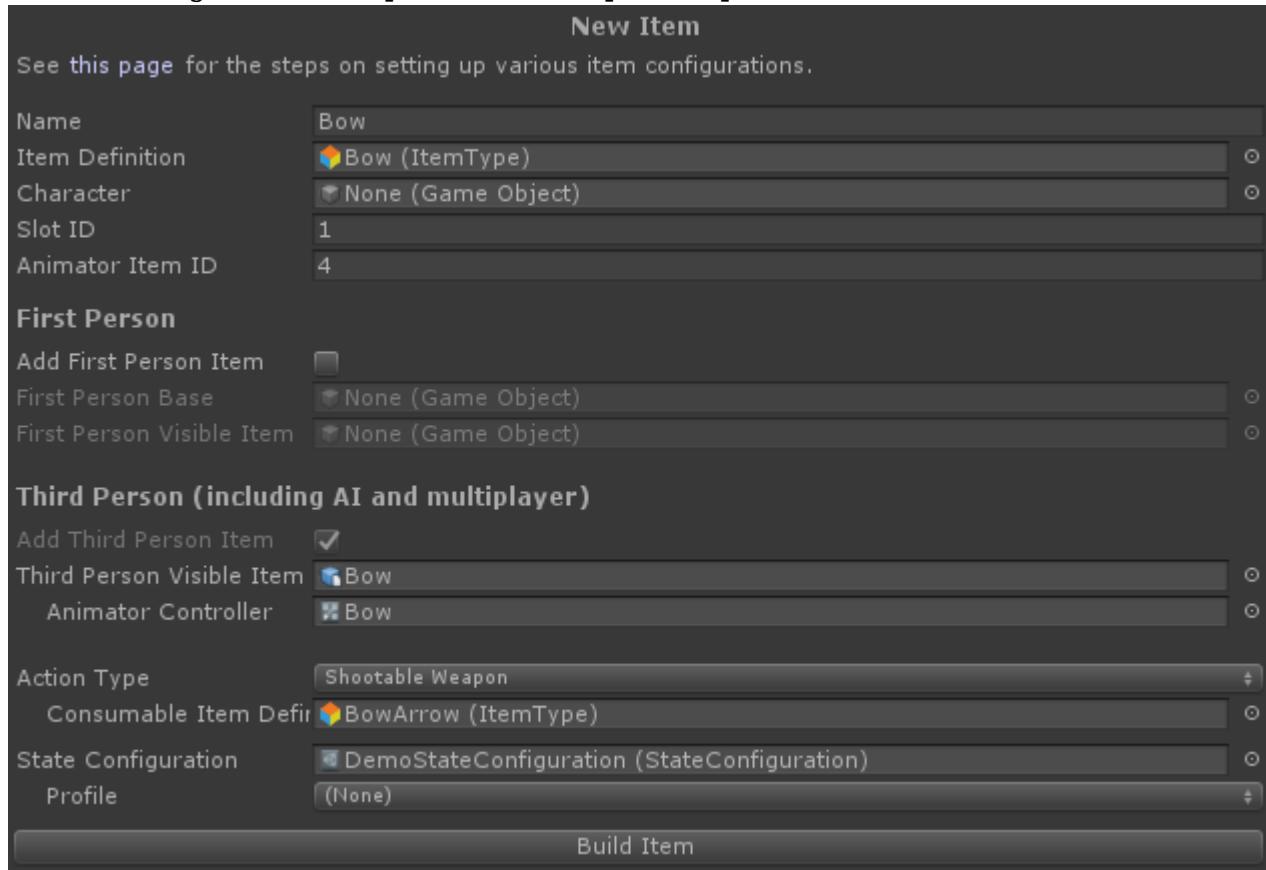


5. Your item has now been setup to be picked up at runtime. This prefab should now be specified within an [Item Pickup](#) GameObject so the item can actually be picked up.

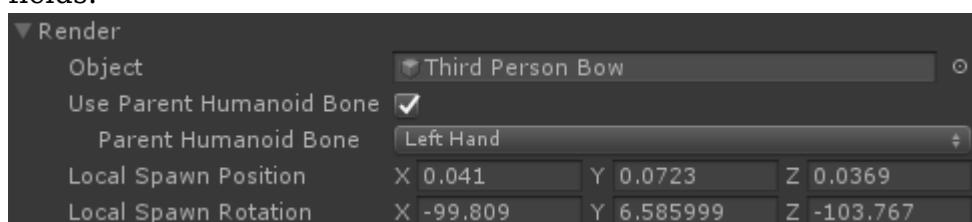
## Third Person Perspective

1. Open the Item Manager. Specify the following fields:
  - *Name*: The name of the item.
  - *Item Definition*: The Item Definition that provides a unique id for the item.
  - *Slot ID*: The slot that the item will occupy.
  - *Animator Item ID*: Optional, the Item ID used within the Animator.
  - *Add Third Person Item*: True.
  - *Third Person Object*: The object rendered within the game.
  - *Action Type*: The type of Item Action to add to the item.

The following screenshot provides a sample setup:



2. Build the item to a new prefab. Select that prefab within the project.
3. The spawn parent needs to be specified. For this item we are going to spawn it in the character's left hand so we ensure *Use Parent Humanoid Bone* is enabled and then the *Left Hand* is specified. If this is false then the item will be spawned based on the *Slot ID*.
4. The spawn position/rotation now needs to be specified. The best method for determining this value is to pickup the item with the Item Pickup component and then adjust the visible item's transform while in play mode. Save the position/rotation values and then copy them to the *Local Spawn Position* and *Local Spawn Rotation* fields:

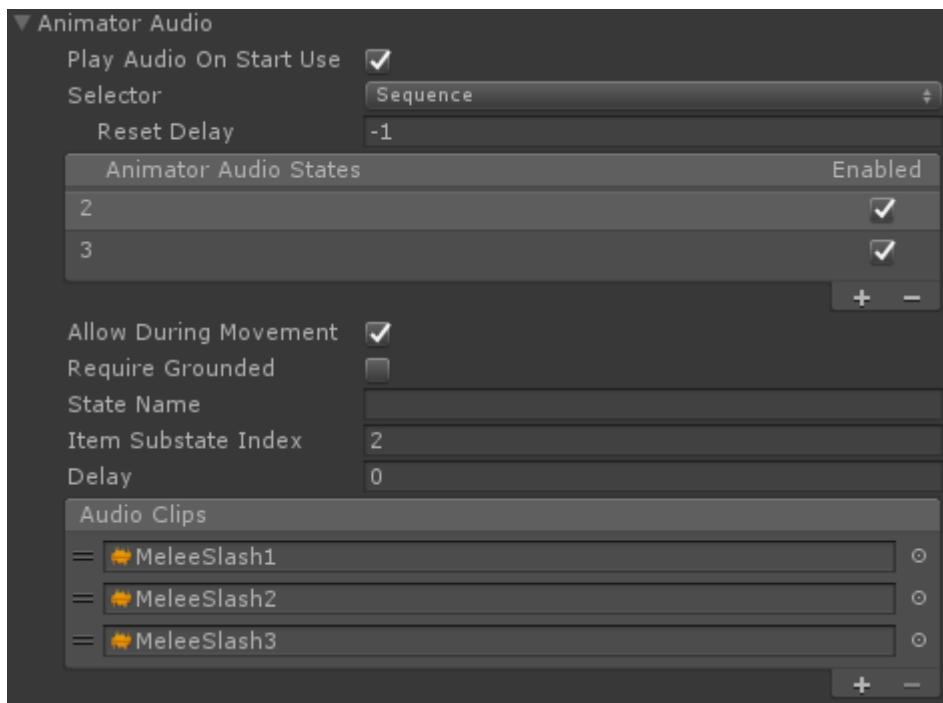


5. Your item has now been setup to be picked up at runtime. This prefab should now be

specified within an [Item Pickup](#) GameObject so the item can actually be picked up.

## Animator Audio State Set

When your item is being used you may not always want the same animation to play. As an example you may want your sword to first slash from the right and then slash from the left. This can be setup using the Animator Audio State Set. Consider the following example:



In this example the item has two different Animator Audio States. The selected state has an Item Substate Index of 1 and three audio clips. When this state is triggered it will change the Animator's Item Substate Index value to 1 and randomly choose one of the audio clips. When the next state is called it'll move onto the second state in the list. This can be changed by adjusting the *Selector* popup that is above the Animator Audio States reorderable list.

The selector popup is a powerful feature which allows the state to be selected based on the current game state. The following selector classes are included:

- *Matching Recoil*: Selects the same state index as what was retrieved by the Use Animator Animator State Selector
- *Random*: Selects the next state randomly.
- *Random Recoil*: Selects the next state randomly for a recoil.
- *Sequence*: Will select the next state sequentially.
- *Sequence Recoil*: Selects the next state sequentially for a recoil.

When the selector is determining what state can be selected it will first ensure that the possible state can be selected. Cases when it may not be able to be selected are if *Allow During Movement* is false but the character is moving or *Require Grounded* is true but the character is not grounded. When the state is selected the state with the specified *State Name* will activate allowing for states to change the character's state. An example use may be that when the state at index 2 plays the melee weapon should cause more damage because it is a heavy attack.

## API

You can also create your own selection objects by extending the Animator Audio State Selector object. The Animator Audio State Selector has the following API:

```
/// <summary>
/// Initializes the selector.
/// </summary>
/// <param name="gameObject">The GameObject that the state belongs to.</param>
/// <param name="characterLocomotion">The character that the state belongs to.</param>
/// <param name="item">The item that the state belongs to.</param>
/// <param name="states">The states which are being selected.</param>
public virtual void Initialize(GameObject gameObject,
UltimateCharacterLocomotion characterLocomotion, Item item,
AnimatorAudioStateSet.AnimatorAudioState[] states);

/// <summary>
/// Starts or stops the state selection. Will activate or deactivate the state with the name specified within the AnimatorAudioState.
/// </summary>
/// <param name="start">Is the object starting?</param>
public virtual void StartStopStateSelection(bool start)

/// <summary>
/// Returns the current state index. -1 indicates this index is not set by the class.
/// </summary>
/// <returns>The current state index.</returns>
public virtual int GetStateIndex()

/// <summary>
/// Moves to the next state.
/// </summary>
public virtual void NextState()

/// <summary>
/// Returns an additional value that should be added to the Item Substate Index.
/// </summary>
/// <returns>An additional value that should be added to the Item Substate Index.</returns>
public virtual int GetAdditionalItemSubstateIndex()

/// <summary>
/// The object has been destroyed.
/// </summary>
public virtual void OnDestroy();
```

# Inventory

The focus of the Ultimate Character Controller is to be an exceptional character controller so the inventory system is less focused on features and more focused on being flexible. With that said, in order to have a functioning item system the inventory system does need to be able to support the basics such as keeping an ammo count or determining if the character has a weapon.

The inventory uses Item Definitions to identify a particular item. Item Definitions are also used for consumables such as assault rifle bullets or amount of battery left. The built-in inventory class uses a concept of Item Types, which are a subclass of Item Definitions. You'll also see Item Identifiers within the asset and these are a reference to the specific item instance. This distinction is made to allow for easier integration with our inventory solution asset. Item Types are used within the character controller and are both Item Definitions and Item Identifiers. Item Types can be created within the [Item Type Manager](#).

## Pickup Script

The script below will add a new ItemType to the inventory:

```
using UnityEngine;
using Opsive.UltimateCharacterController.Inventory;

/// <summary>
/// Picks up the specified ItemType within the Start method.
/// </summary>
public class Pickup : MonoBehaviour
{
    [Tooltip("The character that should pickup the item.")]
    [SerializeField] protected GameObject m_Character;
    [Tooltip("The ItemType that the character should pickup.")]
    [SerializeField] protected ItemType m_ItemType;
    [Tooltip("The number of ItemType that the character should
pickup.")]
    [SerializeField] protected int m_Count = 1;

    /// <summary>
    /// Picks up the ItemType.
    /// </summary>
    public void Start()
    {
        var inventory = m_Character.GetComponent<InventoryBase>();
        if (inventory == null) {
            return;
        }

        // Adds the specified amount of the ItemType to the inventory.
        // m_ItemType: The ItemType to pick up.
        // m_Count: The amount of ItemType to pick up.
```

```

        // -1: The slot ID that picked up the item. A -1 value will
        indicate no specified slot.
        // true: Should the item be picked up immediately? If false
        the EquipUnequip ability may not switch to the newly picked up item.
        // false: Should the item be force equipped?
        inventory.PickupItemType(m_ItemType, m_Count, -1, true,
false);
    }
}

```

## Equip ItemSet Script

The script below will equip a specific ItemSet from the [ItemSetManager](#). The item must first be picked up before it can be equipped.

```

using UnityEngine;
using Opsive.UltimateCharacterController.Character;
using Opsive.UltimateCharacterController.Character.Abilities.Items;

/// <summary>
/// Equips the specified ItemSet within the Start method.
/// </summary>
public class Equip : MonoBehaviour
{
    [Tooltip("The character that should drop the item.")]
    [SerializeField] protected GameObject m_Character;
    [Tooltip("The index of the category that the ItemSet belongs
to.")]
    [SerializeField] protected int m_CategoryIndex;
    [Tooltip("The index of the ItemSet that should be equipped.")]
    [SerializeField] protected int m_ItemSetIndex;

    /// <summary>
    /// Equips the ItemSet.
    /// </summary>
    public void Start()
    {
        var characterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        var equipUnequipAbilities =
characterLocomotion.GetAbilities<EquipUnequip>();
        for (int i = 0; i < equipUnequipAbilities.Length; ++i) {
            if (equipUnequipAbilities[i].ItemSetCategoryIndex ==
m_CategoryIndex) {
                // Starts equipping to the specified ItemSet.
                // ItemSetIndex: The ItemSet to equip/unequip the
items to.
                // ForceEquipUnequip: Should the ability be force
started? This will stop all abilities that would prevent EquipUnequip
            }
        }
    }
}

```

```

from starting.

        // ImmediateEquipUnequip: Should the items be equipped
        // or unequipped immediately?
equipUnequipAbilities[i].StartEquipUnequip(m_ItemSetIndex, true,
false);
        return;
    }
}
}
}

```

## Drop Script

The script below will drop an ItemType from the inventory:

```

using UnityEngine;
using Opsive.UltimateCharacterController.Inventory;

/// <summary>
/// Drops the specified ItemType within the Start method.
/// </summary>
public class Drop: MonoBehaviour
{
    [Tooltip("The character that should drop the item.")]
    [SerializeField] protected GameObject m_Character;
    [Tooltip("The ItemType that the character should drop.")]
    [SerializeField] protected ItemType m_ItemType;

    /// <summary>
    /// Drops the ItemType.
    /// </summary>
    public void Start()
    {
        var inventory = m_Character.GetComponent<InventoryBase>();
        if (inventory == null) {
            return;
        }

        // A single ItemType can exist in multiple slots. Drop all of
        // the items.
        for (int i = 0; i < inventory.SlotCount; ++i) {
            var item = inventory.GetItem(i, m_ItemType);
            if (item != null) {
                // Removes the item from the inventory. The last
                // parameter drops the item.
                // ItemType: The ItemType to remove.
                // SlotID: The ID of the slot.
                // Drop: Should the item be dropped when removed?
                inventory.RemoveItem(m_ItemType, i, true);
            }
        }
    }
}

```

```
        }
    }
}
```

## Inspected Fields

### Current Inventory

The Current Inventory foldout will list every item that is currently in the inventory. This list will only be populated when the game is running. This list is purely for informational purposes as the list cannot be edited.

### Default Loadout

The character will spawn with any items specified within the Default Loadout. New ItemTypes can be added and removed by selecting the plus or minus icon on the bottom right of the list.

### Remove All on Death

Should all of the items be removed when the character dies?

### Load Default Loadout on Respawn

Should the inventory load the Default Loadout after the character has respawned?

## API

Most of the operations performed directly on the inventory will be to react to an event sent by the inventory rather than performing an operation on the inventory. The inventory exposes many events using the built in [Event System](#):

```
// Called when the item is initially added to the inventory (ex. an
// assault rifle has been added to the character at edit time).
OnInventoryAddItem(Item item)

// Called when an ItemIdentifier is picked up (ex. the character finds
// an assault rifle on the ground and picks it up).
OnInventoryPickupItemIdentifier(IItemIdentifier itemIdentifier, int
amount, bool immediatePickup, bool forceEquip)

// Called when an Item is picked up. This event is similar to
// OnInventoryPickupItemIdentifier except it is only called for
// ItemIdentifiers that have an item attached to them.
OnInventoryPickupItem(Item item, int count, bool immediatePickup, bool
forceEquip)

// Called when the item is equipped (ex. the event is called for the
// assault rifle when switching from a knife to an assault rifle).
```

```

OnInventoryEquipItem(Item item, int slotID)

// Called when the ItemIdentifier amount is changed (ex. the event is
// called when an assault rifle being fired).
OnInventoryAdjustItemIdentifierAmount(IItemIdentifier itemIdentifier,
int remaining)

// Called when the item is unequipped (ex. the event is called on the
// assault rifle when switching from an assault rifle to a knife).
OnInventoryUnequipItem(Item item, int slotID)

// Called when the item is removed from the inventory (ex. the event
// is called when the character dies and drops all of their items).
OnInventoryRemoveItem(Item item, int slotID)

```

Corresponding [Unity events](#) exist for each of these events. The following code can be used as an example for subscribing to one of these events:

```

using UnityEngine;
using Opsive.Shared.Inventory;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    public void Awake()
    {
        EventHandler.RegisterEvent<IItemIdentifier, int>(gameObject,
"OnInventoryAdjustItemIdentifierAmount",
OnAdjustItemIdentifierAmount);
    }

    /// <summary>
    /// The specified ItemIdentifier has changed values.
    /// </summary>
    /// <param name="itemIdentifier">The ItemIdentifier that has been
used.</param>
    /// <param name="remaining">The remaining amount of the specified
ItemIdentifier.</param>
    private void OnInventoryAdjustItemIdentifierAmount(IItemIdentifier
itemIdentifier, int remaining)
    {
        Debug.Log("The inventory used " + itemIdentifier + ", " +
remaining + " is remaining.");
    }

    public void OnDestroy()
    {
        EventHandler.UnregisterEvent <IItemIdentifier,
int>(gameObject, "OnInventoryAdjustItemIdentifierAmount",

```

```

        OnAdjustItemIdentifierAmount);
    }
}

```

## Item Types

An Item Type is a ScriptableObject representing an [Item](#). Each Item must have an Item Type and Item Types are required so the character knows which item to interact with (pickup, equip, use, etc). New Item Types can be created through the [Item Type Manager](#). Item Types are primarily used by the Inventory, Item Set Manager, and Item Pickup components.

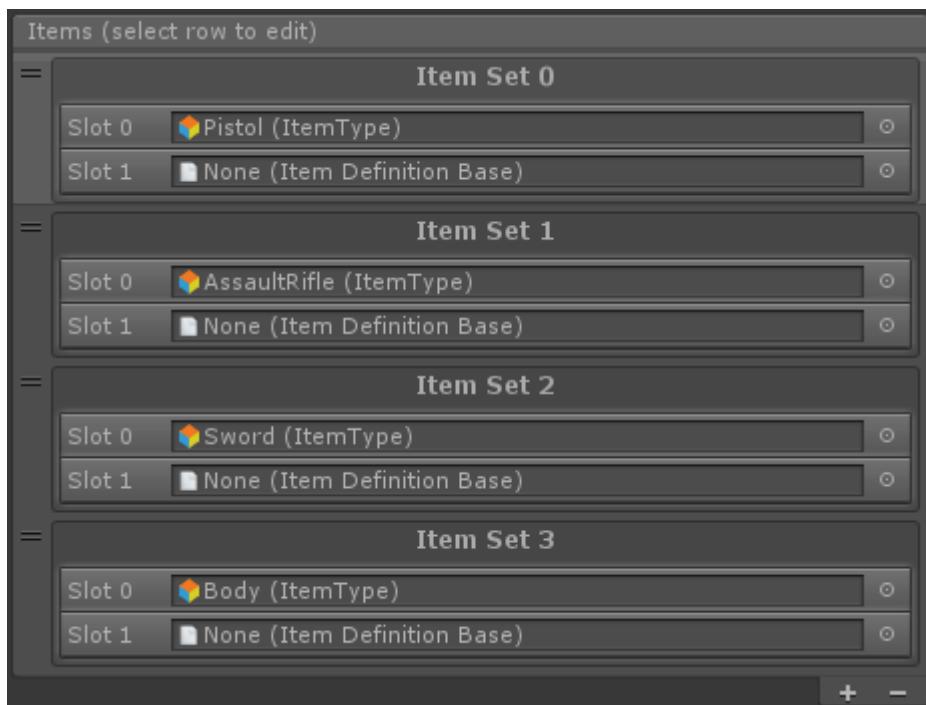
## Item Sets

Item Sets represent a set of Item Definitions that can be equipped at the same time. The Item Set Manager is the component responsible for switching which Item Sets are active. When a new item is added to the character an Item Set will automatically be created with that Item Definition if it isn't already created. More than one Item Sets can be active at the same time through the [Categories](#) defined within the Item Collection.

### Scenarios

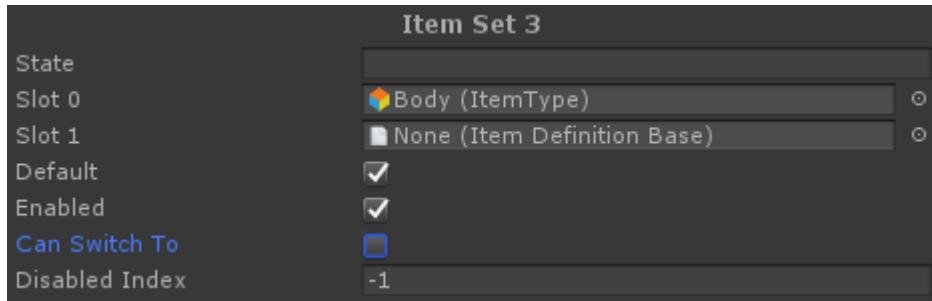
#### Single Category Setup

In this scenario the character will always have a max of one item equipped at a time. While this is the most basic setup of the Item Set Manager it will also be the most common. In a lot of the cases each item will occupy the same slot (such as the right hand slot) but this is not a restriction.



When the game starts the Item Set 0 will first be equipped (assuming the slot is valid - in this scenario the slot will be valid if the character has the pistol). When the [Equip Next](#) ability is activated the Item Set at index 1 will be activated, which contains the assault rifle.

Since the Item Set 1 specifies the assault rifle for slot 0 the pistol will be unequipped. The sword will then be activated after the assault rifle. If the Equip Next is triggered again after Item Set 2 the Item Set Manager will then skip back to the beginning, Item Set 0. This is because Item Set 3 has the *Can Switch To* field deselected:



The *Can Switch To* field indicates whether or not the abilities can switch to the specified Item Set. If the abilities cannot switch to the Item Set then the only way for the Item Set to be activated is it is the *Default* Item Set or if a state activates it via the *Disabled Index*. In this scenario the Item Set 3 is the *Default* Item Set so when the [Toggle Equip](#) ability is activated then the Item Set 3 will be activated. By setting up the Item Set this way it prevents the character from switching to the body Item Set unless no item should be equipped. When the body Item Set is the default Item Set it will allow the character to punch and kick when no item is equipped.



The Item Set doesn't have to contain an Item Definition - if both Slot 0 and 1 had no Item Definitions set then when the Item Set is activated the character wouldn't be able to punch or kick.

### Dual Wield Single Category Setup

This scenario will extend upon the previous scenario by allowing the character to dual wield two pistols and a sword with a shield. No new categories need to be added as the dual wield items can only be equipped with a single Item Definition. In this scenario the Weapons category looks like:

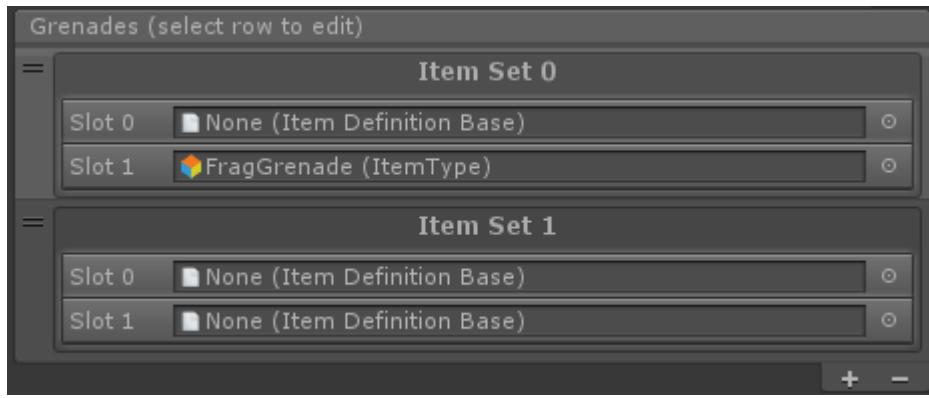
Items (select row to edit)		
<b>Item Set 0</b>		
Slot 0	Pistol (ItemType)	<input type="radio"/>
Slot 1	Pistol (ItemType)	<input type="radio"/>
<b>Item Set 1</b>		
Slot 0	AssaultRifle (ItemType)	<input type="radio"/>
Slot 1	None (Item Definition Base)	<input type="radio"/>
<b>Item Set 2</b>		
Slot 0	Sword (ItemType)	<input type="radio"/>
Slot 1	None (Item Definition Base)	<input type="radio"/>
<b>Item Set 3</b>		
Slot 0	Sword (ItemType)	<input type="radio"/>
Slot 1	Shield (ItemType)	<input type="radio"/>
<b>Item Set 4</b>		
Slot 0	Body (ItemType)	<input type="radio"/>
Slot 1	None (Item Definition Base)	<input type="radio"/>

When the game starts it will first try to activate Item Set 0. If the character does not have 2 pistols then Item Set 0 will not be able to be activated and the next slot will be tested. Assuming the character does have 2 pistols in the inventory then Item Set 0 will activate. Slot 0 corresponds to the right hand and slot 1 corresponds to the left hand. This will allow two pistols to be equipped at the same time (assuming the pistols are setup to [work together](#)).

When the next Item Set is activated it will activate the Item Set 1 which only contains the assault rifle in the right hand. There is no Item Definition specified for slot 1 so the left pistol will be unequipped. The next Item Set only contains the sword which will unequip the assault rifle. Item Set 3 contains a sword and shield combo. Slot 1 will equip the shield and because slot 0 contains the same Item Definition as what was in Item Set 2 the slot will not change.

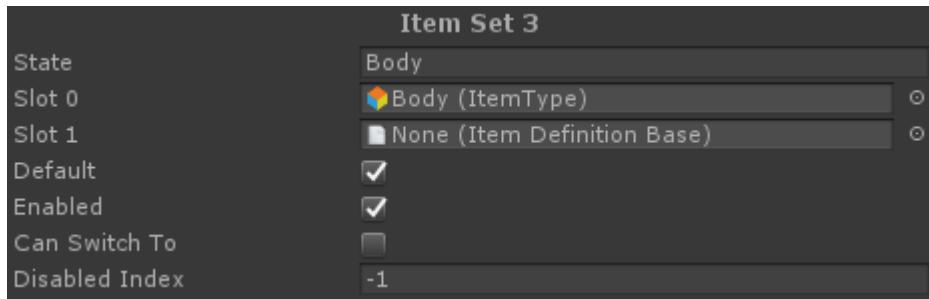
### Double Category Setup

The goal of this scenario is to have a primary weapon but also allow a grenade to be thrown at any time. One way to do this would be to have the grenade act as a dual wield item similar to the last scenario, but in this case we don't want the Item Set to be dependent on the number of grenades the character has so a second category should be created. The Weapons category within the Item Set Manager should look the same as it did within the Single Category scenario, and the following category and Item Set should be added:

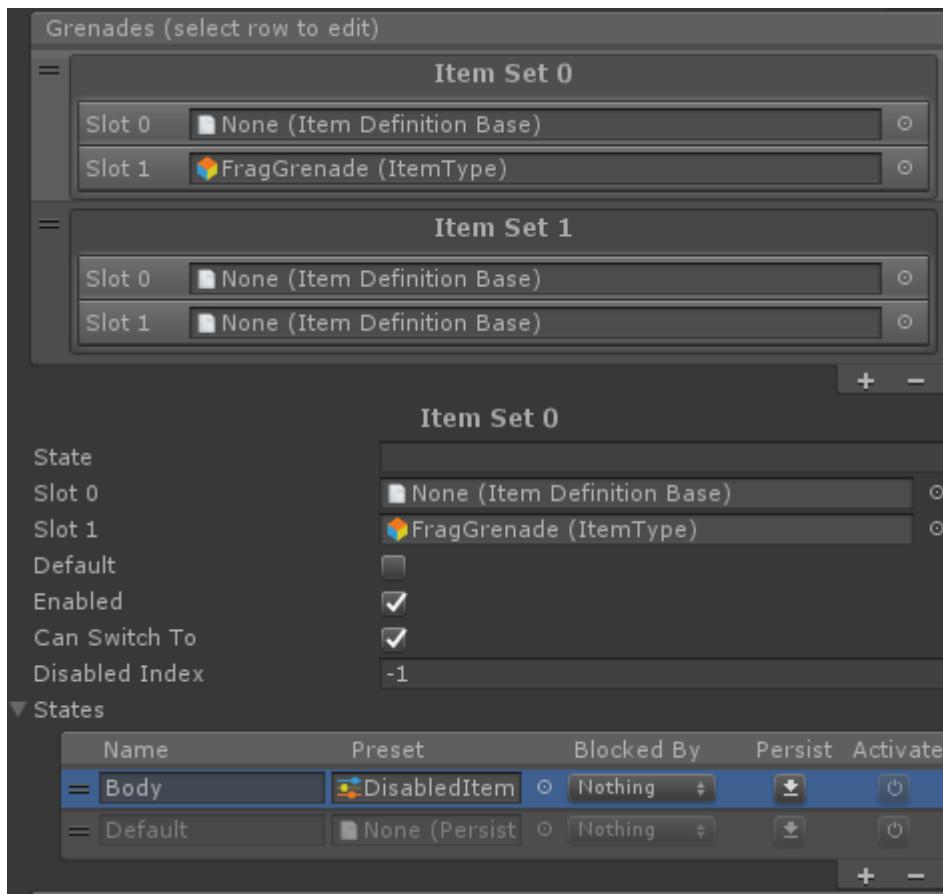


This new category has the frag grenade being equipped within slot 1, the left hand. If you take a look at the Single Category scenario you'll see that no Item Definitions occupy slot 1 within the Weapons category so there are no conflicts. If the character has a grenade the Item Set 0 will be activated and they can throw the grenade with the left hand at any time.

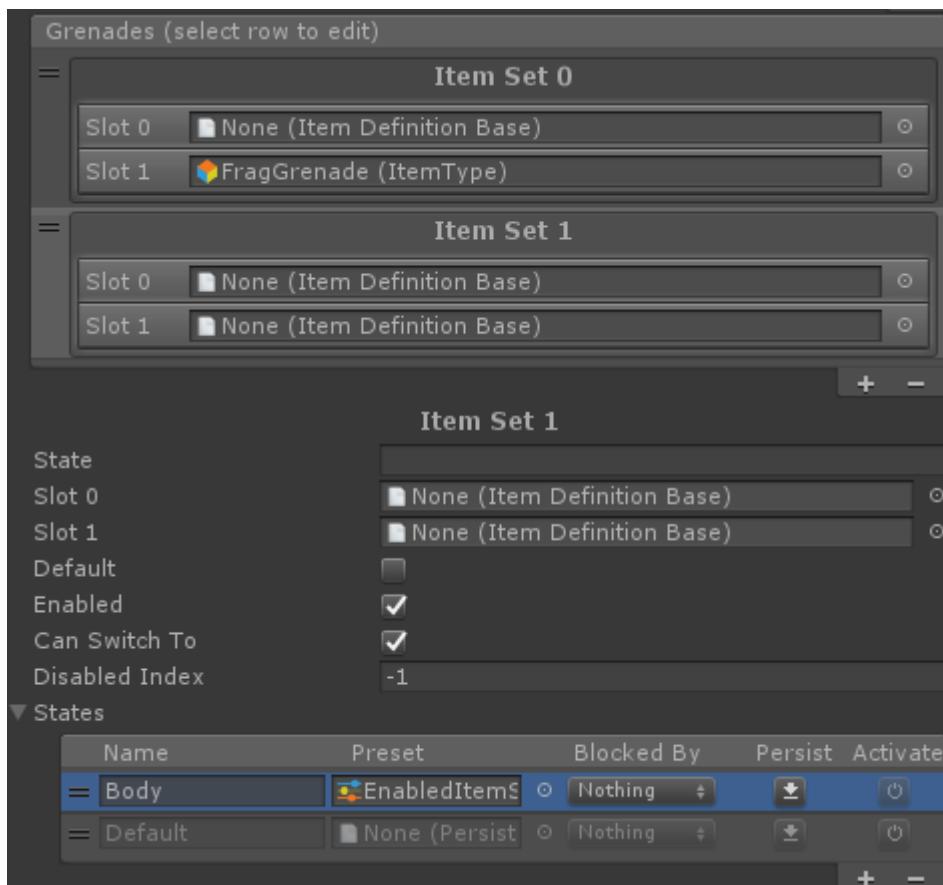
When the body Item Set from the Weapons category is activated we don't want the character to be able to throw a grenade. We can prevent this from happening by first setting a state name when the body Item Set is activated:



When the Body state is activated the Item Set 0 from the Grenades category should disable itself by setting the Enabled property to false within the [state system](#).



With this setup the Body state will be activated when the body Item Set is activated within the Weapons category and the state will disable the *Enabled* property within the Item Set. When the Item Set is disabled it will switch to the Item Set 1 because that Item Set is *Enabled* when the Body state is active:



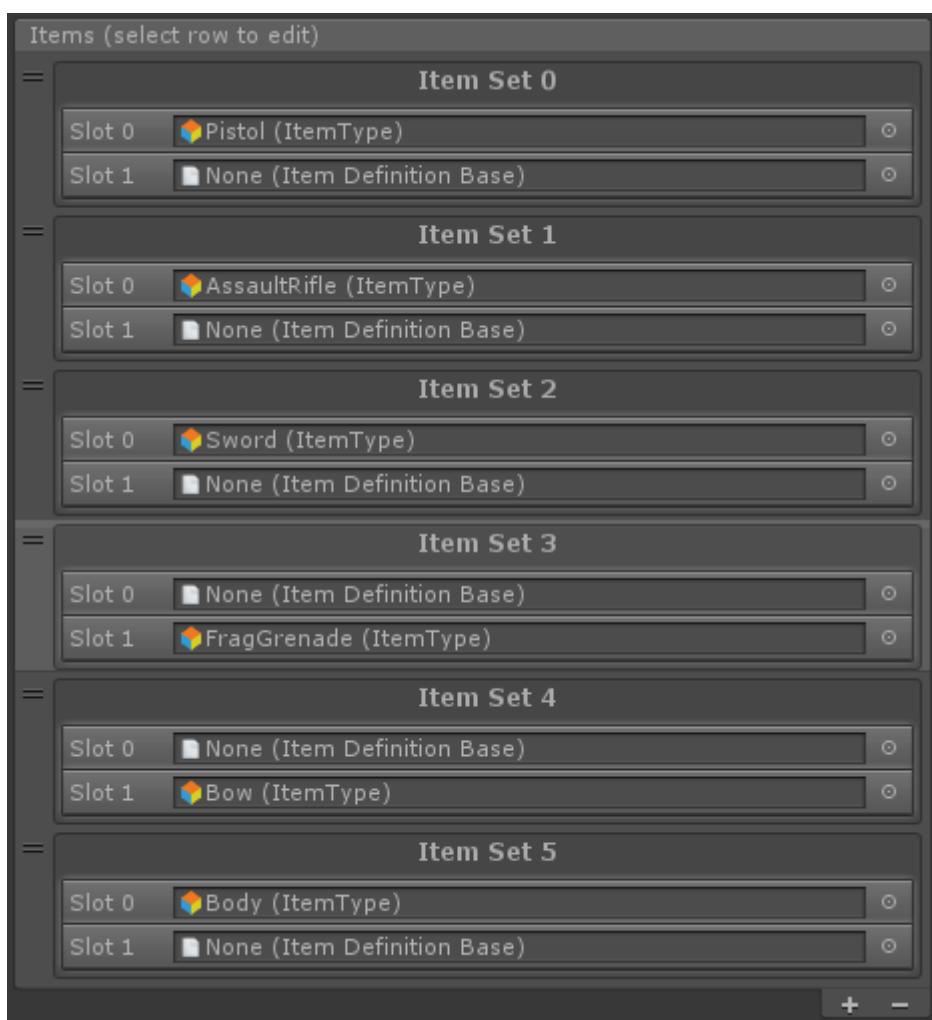
Notice that the *Disabled Index* is set to 0 which means that when the Item Set is deactivated

(the Body state is deactivated) the Item Set will disable and switch to the index specified, which is Item Set 0.

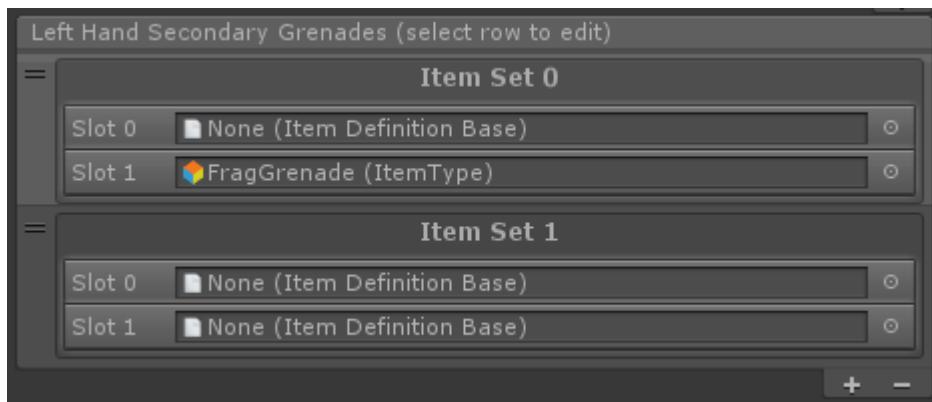
### Triple Category Setup

The Ultimate Character Controller uses a triple category setup. This setup allows the grenade to be equipped as a primary item with the right hand or thrown from either hand as a secondary item. In most cases the grenade will be thrown from the left hand as a secondary item (such as when the character has an assault rifle or sword equipped) but the bow occupies the left hand so the grenade needs to be able to be thrown from the right hand as well.

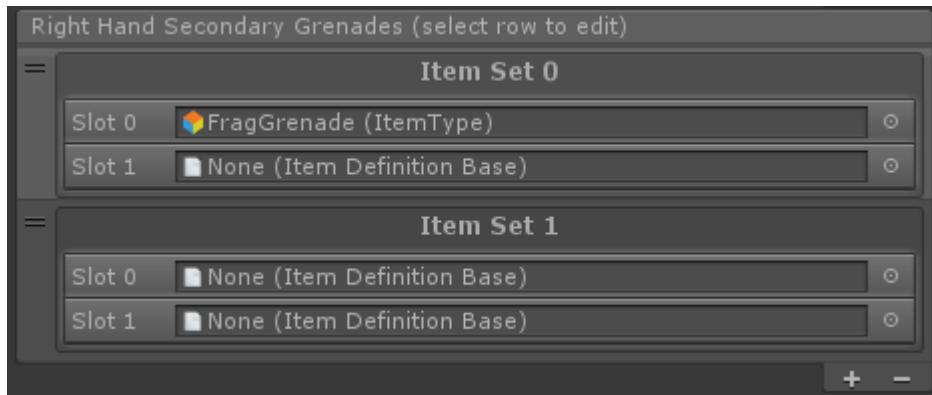
To get started ensure the Left Hand Secondary Grenades and Right Hand Secondary Grenades categories are created and then add a Frag Grenade Item Definition to the Weapons Item Set. To demonstrate the secondary grenade switching slots the Bow Item Definition has also been added to the Weapons category:



The Frag Grenade Item Definition should be added to the Left Hand Secondary Grenades Category along with an empty Item Set:

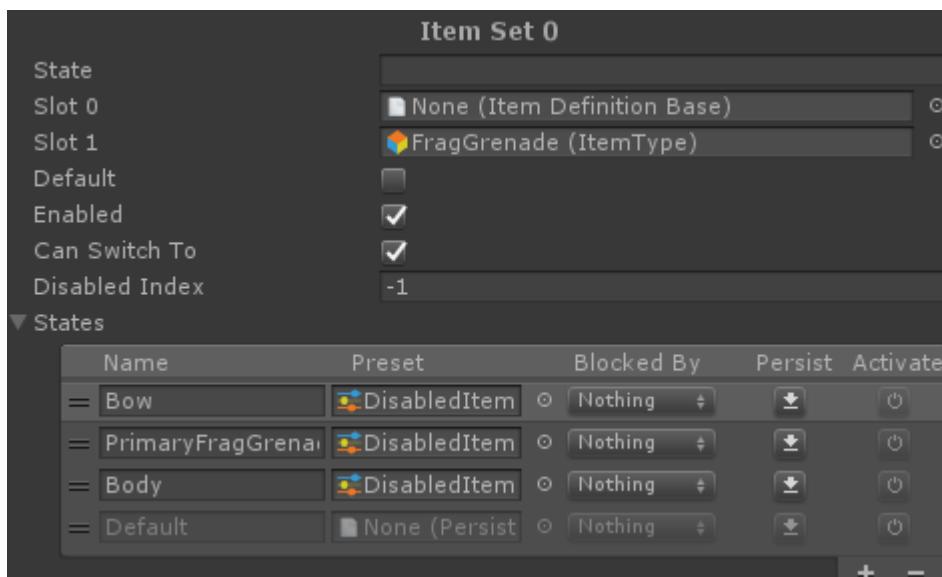


Finally the Frag Grenade Item Definition should be added to the Right Hand Secondary Grenades Category along with an empty Item Set:

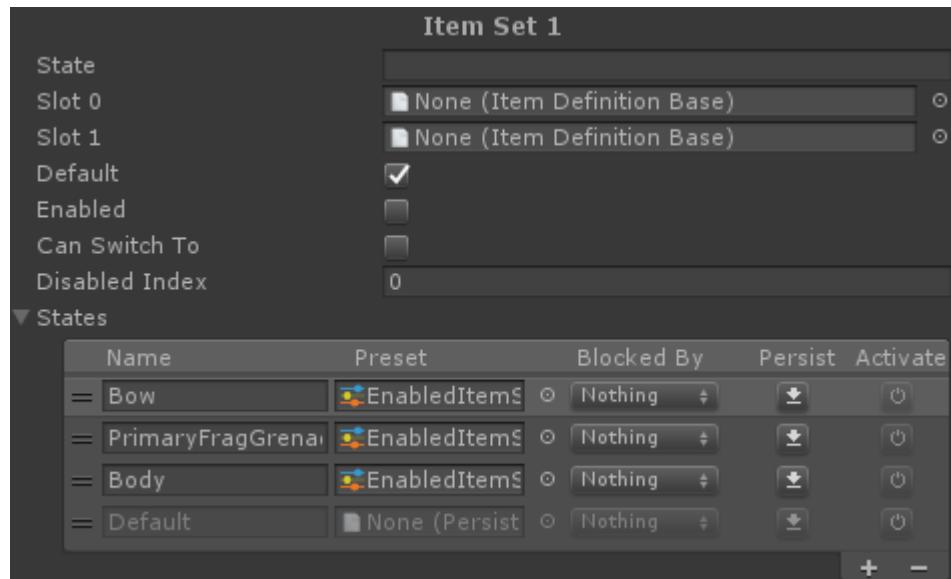


The basic setup is now complete and the Item Definitions need to be setup, similar to how the Item Definitions were setup in the Double Category scenario. In order for the Left and Right Hand categories to know that they need to do something different from the default frag grenade Item Definition (at index 3) should activate the PrimaryFragGrenade state. Similarly, the bow Item Definition (at Item Set 4) should activate the Bow state.

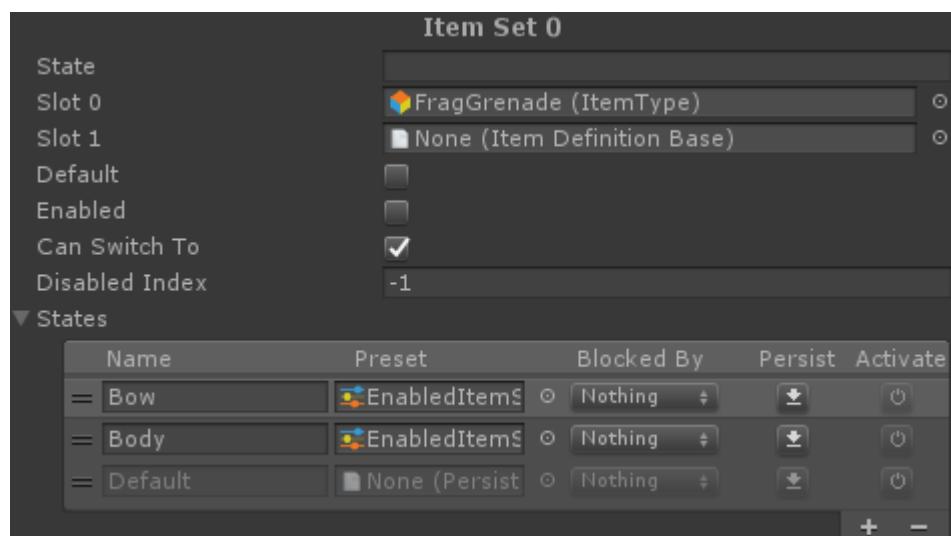
Looking at the Left Hand Secondary Grenades category first, by default the Item Set 0 should be enabled since this Item Set will be activated most of the time. When the PrimaryFragGrenade or Bow (or Body) state is activated this Item Set should deactivate so Item Set 1 can activate. This will prevent the Frag Grenade from being equipped in slot 1 when another item is activated in slot 1:



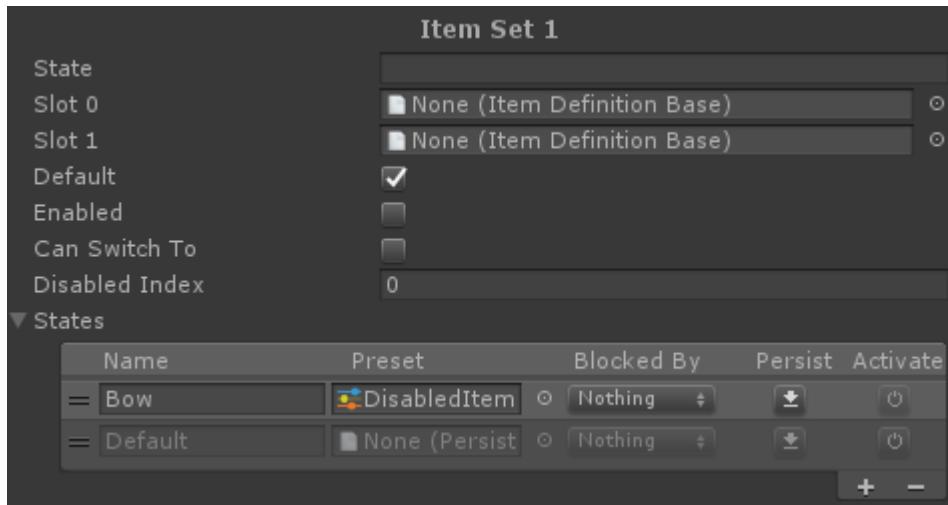
Item Set 1 starts off disabled and will be enabled when the PrimaryFragGrenade, Bow, or Body states are active. When the state is disabled again (meaning none of those states are active) then the category will switch to the *Disabled Index* which is Item Set 0. This is how Item Set 0 reactivates again.



The Right Hand Secondary Grenades Category is almost the reverse of the Left Hand Secondary Grenades Category. Item Set 0, which contains the Frag Grenade Item Definition, should only be activated when the bow is activated. This allows the grenade to still be thrown when the bow is equipped even though the bow occupies slot 1. The Item Set still disables itself when the Body state is activated because no Item Sets should be enabled at that time.

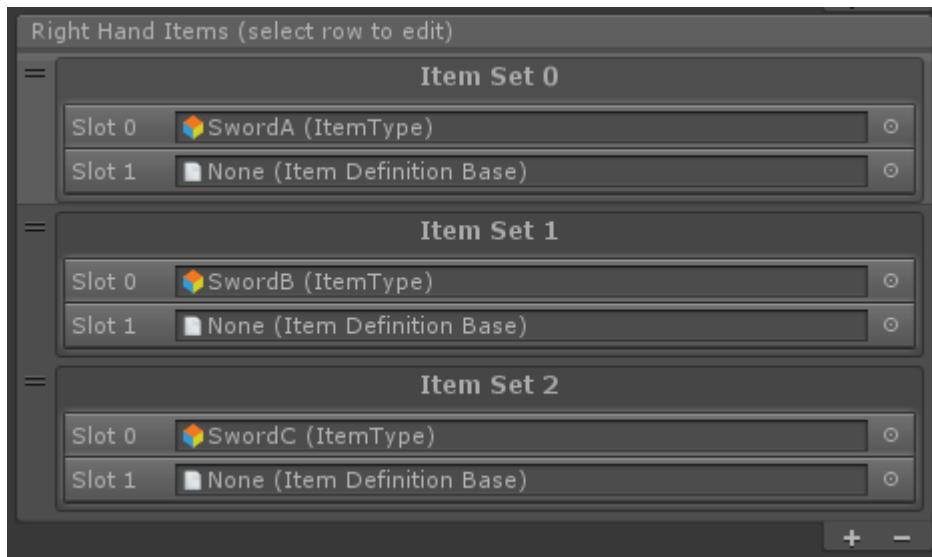


Item Set 1 of the Right Hand Secondary Grenades Category is similar to Item Set 1 of the Left Hand Secondary Grenades Category except it doesn't contain as many states because this Item Set is only activated when the bow is equipped.

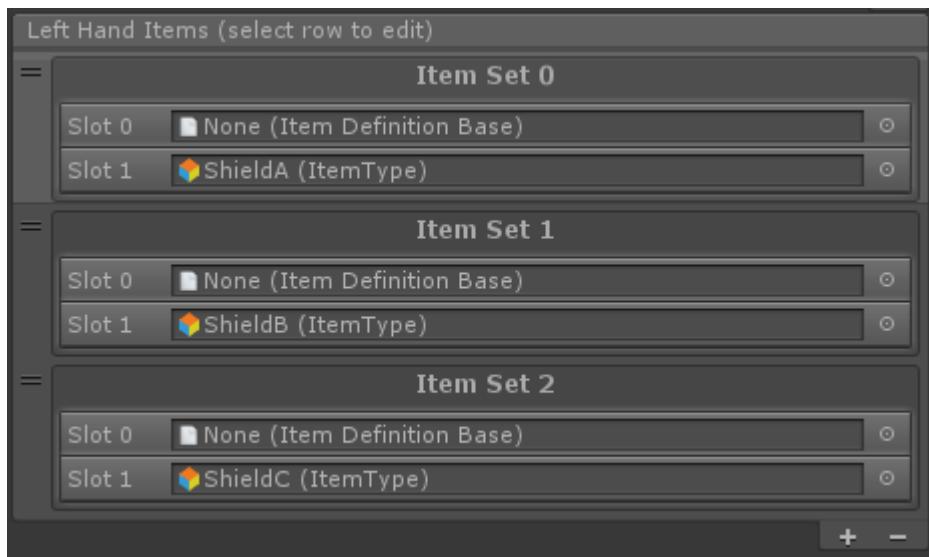


### Sword and Shield Category Setup

You may be working on an RPG which has hundreds of swords or shields and they can all be dual wielded together. Having a single category with every possible combination would not be feasible. For this case you can create two categories: Left Hand Items and Right Hand Items. All of the swords can be equipped in the right hand (slot 0) and all of the shields can be equipped in the left hand (slot 1). With this setup any shield can be equipped with any sword. Here's an outline of what the Right Hand Item Category would look like:



The Left Hand Category would then contain all of the shields:



## Inspected Fields

### State

The state name that should be enabled when the ItemSet is active. This is extremely useful in the case where you want to enable or disable certain properties when a particular item is equipped.

### Slot

The Item Definition for the specified slot index. These slots match up to the same slots that are specified by the ItemSlot component.

### Default

Is the ItemSet the default ItemSet? Default ItemSets will be activated when the Toggle Equip ability is activated or a particular ItemSet has been disabled. Only one Default ItemSet is valid per category.

### Enabled

Is the ItemSet enabled? A disabled ItemSet cannot be activated through the EquipUnequip ability.

### Can Switch To

Can the ItemSet be switched to by the EquipNext/EquipPrevious abilities?

### Disabled Index

Specifies the ItemSet index that should be activated if the current ItemSet is disabled while being active.

# Slots

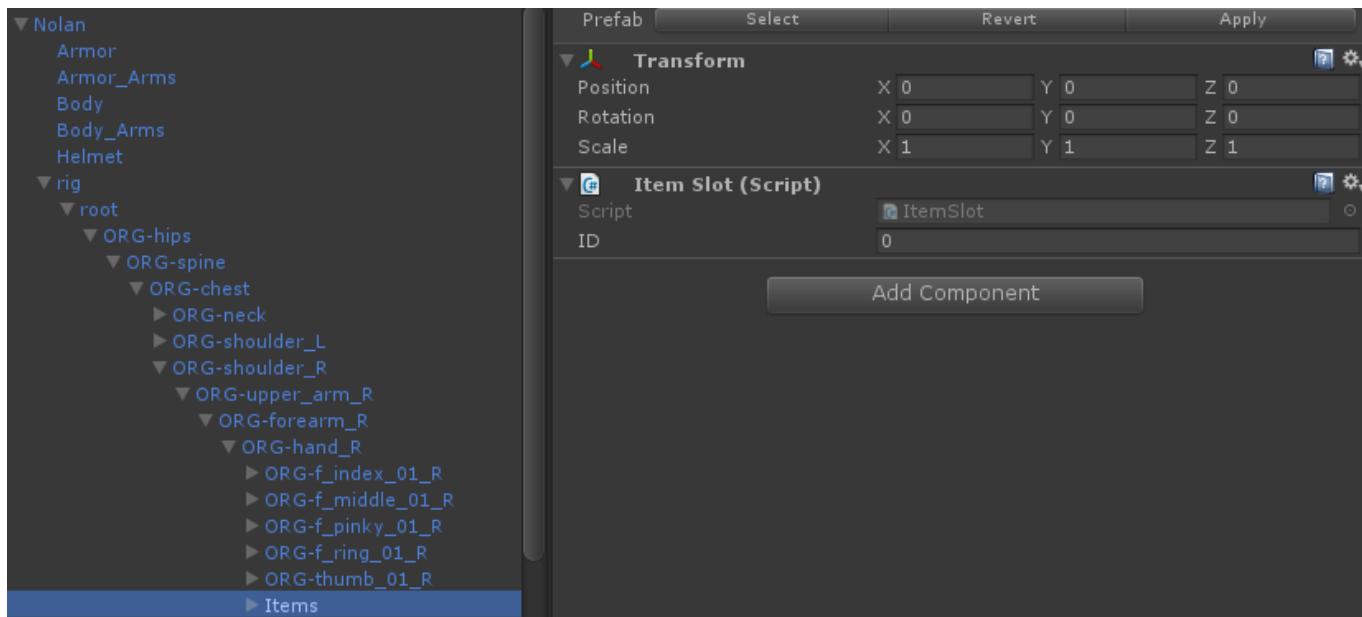
When the [Item Set Manager](#) determines if a particular item can be equipped it does so based off of the slot system. Think of slots like an array element within the inventory. Only one item can occupy each slot, and that item has to be unique so the inventory knows which item should be equipped. From the animator's perspective that unique item is represented by the animator ID.

Consider the example where you want to have dual pistols equipped. In the animator the pistol items are represented by the [unique animator ID](#) of 2, but because you can have multiple pistols they should each occupy a slot within the inventory. When both pistols are equipped the animator will contain the following values:

Slot0ItemID: 2  
Slot1ItemID: 2

This tells the animator that the character has dual pistols.

The demo scene is setup for two slots: slot 0 represents the right hand and slot 1 represents the left hand. The number of slots is determined by the index of the Item Slot component. When you create your character this component is added to the "Items" GameObject underneath the character's rig.



If your character can switch between both first and third person perspective then there should be a matching slot for each perspective. For example, if the assault rifle is in slot 0, the right hand, of the third person perspective then it should also be in slot 0, the right hand, of the first person perspective. Each perspective must have the same number of item slots in the same location.

The controller can handle any number of slots but the animator is only setup to handle two slots so if you are using an animator and want to use more than two slots you'll need to add four new slot parameters. These parameters should be (where X represents the slot index):

- SlotXItemID (int)

- SlotXItemStateIndex (int)
- SlotXItemStateIndexChange (trigger)
- SlotXItemSubstateIndex (int)

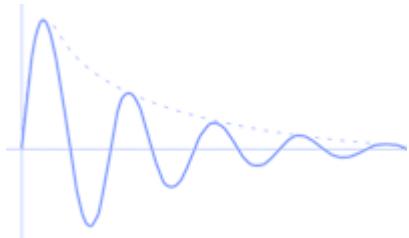
Once these parameters are setup the Animator Monitor component will automatically set the parameters based on the slot count (determined by the max Item Slot ID).

## Animation

The primary method of the character playing an animation is through [Unity's Animator component](#). If you have not used an Animator Controller before it is highly recommended that you first take a look at the Unity Learn tutorial. As you are replacing animations ensure you first check the animation to see if it has any [Animation Events](#) that will also need to be replaced.

Procedural animations can be added to the first person arms and camera through the [spring system](#). The spring system makes it easy to add fluid animations without having to add them through the Animator Controller. If you are using a different rig for your first person arms then you will not be able to use the included first person animations. Unity's Animator component does not support [animation retargeting](#) for generic rigs.

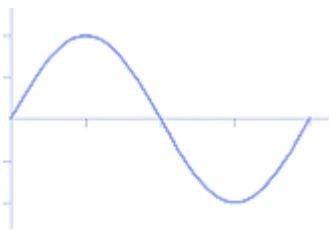
## Springs



The spring system is an extremely powerful system which allows for smooth procedural motion without requiring any premade animations. This system is primarily used for the first person perspective, though the third person perspective also uses it for camera recoil.

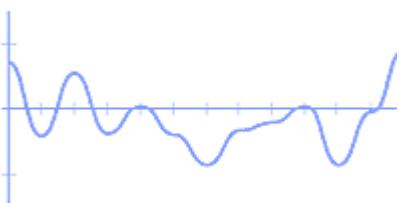
The spring has a Rest State (also known as the “static equilibrium”) where it “wants to be”. If you move it away from the target value using external force, it will immediately and continuously strive back to its target position or angle. The spring Stiffness – or mechanical strength – determines how loosely or rigidly the spring behaves. Finally, the Damping makes spring velocity wear off as it approaches its rest state. Springs can be used to simulate everything from jello (a loose spring operating on object scale), an arrow hitting a wall (a very stiff spring rotating the arrow with the pivot at the head) or an underwater camera (a loose spring making the camera bounce softly as your feet hit the ocean floor).

### Bobs



Bobs are the sinusoidal motion of an object or the camera. Bobbing has been around for ages in games as a means of moving the first person camera up and down while walking. Modern first person cameras don't always have a very pronounced view bob, but bob is still very useful for animating the item model (and camera if used in moderation). If applying bob along both the x and y vector you can get interesting results such as the feeling of being a dinosaur or a huge ogre.

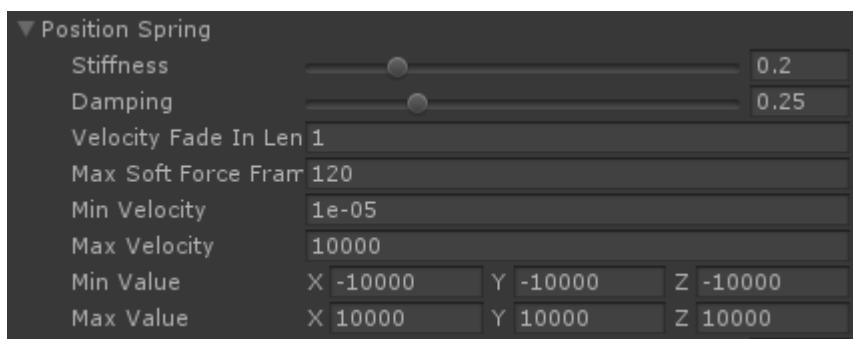
## Noise



Procedural noise has been used in computer graphics and special effects for years as a means of generating smoke, terrains and camera shakes (to name but a few areas). The Ultimate Character Controller uses a standard perlin noise function for applying random turbulence to the first person weapon and camera. Examples include a slight breathing camera motion as you zoom in with a sniper rifle, gentle idle movements for a hand holding a sub machine gun, or the heavily disturbed camera movements of a character that is drunk or poisoned.

## Inspected Fields

Any component that uses a spring can show the spring properties in the inspector.



The following fields can be modified:

### Stiffness

Spring stiffness – or mechanical strength – determines how loosely or rigidly the spring behaviors. A low value will result in a soft, sway motion, and a higher value will result in stronger, stiffer movements.

## Damping

Damping makes the spring velocity wear off as it approaches its rest state. Low values will result in a very loose, sway motion. Higher values will either result in a motion that quickly settles, or a stiff shaking motion. The result depends on the stiffness value.

## Velocity Fade In Length

The amount of time it takes for the velocity to have its full impact. This can prevent the spring from quickly snapping to the rest position when the game is initially started.

## Max Soft Force Frames

The maximum number of frames that a soft force can be spread over. The higher the value the more the spring will ease into the soft force.

## Min Velocity

The minimum value of the velocity.

## Max Velocity

The maximum value of the velocity.

## Min Value

The minimum value of the spring.

## Max Value

The maximum value of the spring.

# Animator

## Animator Controller

### New Animator Controller

The animator controller included works well for the included demo scene but may not be the best setup for your particular game. It is recommended that you start from a completely new animator controller and add your own states/transitions. The character controller doesn't particularly care what structure the animator controller has. The only requirement is that you have the same [parameters](#) as what the demo animator controller uses.

### Structure

The Demo Animator for the third person character is made up of 13 layers. Each layer has a specific role in terms of what it controls for the character's movements which is achieved by

utilizing masks and transitions.

### **Base Layer**

The Base Layer is the primary layer and contains all information related to movement for the character. If no other layer is activated everything would default to this layer, so all animations in this layer contain lower and upper body data.

The character's movement is split into two basic states, Idle and Movement. When not in motion and no other abilities are active, the character defaults to Idle. If the character is not in motion but aiming, the character defaults to Aim Idle. Same thing if the character is in motion, the character could either be in Movement or Aim Movement.

Aim Idle and Aim Movement are then broken down into three different forms of such movement in the default animator: Aim, Melee Aim, and Bow Aim. Each of these aim movement styles correspond to the *Movement Set ID* value which is set on each individual weapon. In this case all shootable weapons (outside of the bow) are going to Aim (*Movement Set ID = 0*), melee weapons are going to Melee Aim (*Movement Set ID = 1*), and the bow is going to Bow Aim (*Movement Set ID = 2*). You can customize your animator to make use of any number of *Movement Set ID* values, or only use one set.

Outside of character movement the Base Layer contains substates for a variety of abilities which include Crouch, Fall, Interact, Jump, Quick Turn, Ride, Start Movement, and Stop Movement. Crouch is an ability that is treated as another form of movement, so it's substate is set up identically to the standing idle and movement substates. The animator transitions to each of these abilities based on the value of the *Ability Index* parameter.

### **Left Hand and Right Hand Layers**

These two layers are masked to control only the left or right hand of the character respectively. These layers are used to override the Base Layer if an item is equipped that does not have a specific idle animation for the upper body. For example, if the Frag Grenade is equipped the right hand should grip the grenade, but all other motion for the idle animation would remain the same as the Base Layer. So in this case, an idle animation is placed under the Frag Grenade substate within the Right Hand Layer to indicate that the right hand fingers should be in the grip animation at all times.

### **Arms Layer**

This layer is masked to control only the arms of the character, it does not utilize chest/torso motion. This layer is used for any item animations that use both arms while the character is in motion. The reason why these can't be on the Upper Body Layer is so the chest is not included when the character is in motion. If the chest were to be included when moving during the equip animation, the walk cycle for the chest would become out of sync with the base layer. Whenever the character is in motion the animator will use this layer for item abilities such as equip/unequip, reload, etc.

### **Upperbody Layer**

This layer is masked to control the entire upper body of the character, including the

chest/torso motion. This layer is used for any item animations that should utilize both arms and the chest rotation.

The shootable weapon substates are identical to the states on the Arms Layer, so the ones on the Upperbody Layer are used to include the chest rotation when the character is not in motion.

The melee weapon substates include the same item actions as the Arms Layer, and in addition also have any attack animations that should include the upper body while in motion. The melee attack animations always include the chest rotation otherwise these would look strange.

Finally, this layer also includes some abilities that utilize both arms and the chest bone such as interact and pickup item.

#### **Left Arm Layer / Right Arm Layer**

These layers are masked to control only the left or right arm of the character and do not include the chest/torso motion. These layers are used for any item actions that can be performed with one arm only or use each arm separately.

For instance, with the lighter melee weapons such as the knife and sword, or the frag grenade, while in motion the left arm does not need to play when performing item actions such as equip/unequip, etc. That is why those are on this layer and not the Arms Layer.

The dual pistols are able to be used independently of each other and therefore have their own individual actions on each layer. You can shoot with the left pistol (Left Arm Layer goes to Attack) while the right pistol remains in idle (Right Arm Layer in Idle).

#### **Left Upperbody Layer / Right Upperbody Layer**

These layers are masked to control the left arm and the chest/torso motion of the character or the right arm and the chest/torso motion of the character. This allows animations that utilize one of the arms and the chest rotation to play.

For example, the secondary Frag Grenade throw occurs while another weapon is equipped, so a secondary throw animation is under the Frag Grenade substate of each layer. This allows the right arm to continue holding the Assault Rifle for instance, while the Left Arm Upperbody Layer then overrides the Arms Layer and plays the Frag Grenade secondary throw. Including the chest rotation with this allows the Frag Grenade throw to feel powerful enough, while not overriding the primary weapon idle.

#### **Additive Layer**

This layer is masked to the upper body with a blending mode of additive to allow the animations on this layer to add to the top of previous layers. For example, with the shootable weapons the attack animations should simply add to the aim animation for each. These attack animations include the recoil from shooting the weapon without being their own overriding animation so that the animation seamlessly blends back into the aim state.

The Damage Visualization ability is also on this layer since the damage visualization animations should add on to the top of any animation state the character may be in. For instance if the character is jumping and gets hit, the jumping animation will continue to play while the damage visualization adds on so the chest rotates to show the character was hit.

#### **Additive Left Arm Layer / Additive Right Arm Layer**

Same function as the Additive Layer except that these layers are used for additive actions being performed on one arm only. In this case, the dual pistol attack is an additive animation that should be applied to the left and right arm independently.

#### **Full Body Layer**

This layer is masked to include the entire avatar, and overrides any previous layer. This layer has the ultimate say! Use cases for this layer are any animation that should immediately take over and play using the full body such as abilities like Die and Revive. In any part of the game the character could die, so this layer allows that to occur immediately with clean transitions.

Other animations that make use of this layer are any full body melee attack animations. For example, when the character attacks with the sword, the attack takes over even if the character is in motion already. For the knife, when the character is not in motion the full body attack layer plays and includes the feet rotating for an added effect. When the character is in motion and attacks with the knife it would play the Upper Body Layer and would not utilize the Full Body Layer.

## **Animator Parameters**

The Demo Animator for the third person and first person character utilizes 22 parameters, which include the following:

#### **Horizontal Movement**

Float value to indicate horizontal direction and speed of character.

#### **Forward Movement**

Float value to indicate forward direction and speed of character.

#### **Pitch**

Float value to indicate viewpoint angle of character.

#### **Yaw**

Float value to indicate movement around the yaw axis of the character.

#### **Speed**

Float value to indicate the character's speed in reference to whether the speed change | 235

ability is active or not.

### **Height**

Float value to indicate the character's height in reference to whether the height change ability is active or not.

### **Moving**

Bool to indicate if the character is in motion or not, checked when moving and unchecked when not moving.

### **Aiming**

Bool to indicate if the character is aiming or not, checked when aiming and unchecked when not aiming. In first person view this is always true.

### **Movement Set ID**

Int value that specifies which movement set is currently active for the primary item equipped. *Movement Set ID* is set on each item to allow for different aiming movement / idle animations.

### **Ability Index**

Int value that specifies which ability is currently active. This is set on each individual ability under the [Ability Index Parameter](#).

### **Ability Int Data**

Int value that relates to the *Ability Index*. This value is utilized to provide more information on a particular ability outside of the main *Ability Index*. For example, the interact ability utilizes the *Ability Int Data* to indicate which particular interact animation should play based on what object the character is interacting with.

### **Ability Float Data**

Float value that relates to the *Ability Index*. This value is utilized to provide more information on a particular ability outside of the main *Ability Index*. For example, the quick turn ability utilizes the *Ability Float Data* to direct the transitions into the quick turn walk or the quick turn run states. Since the character's actual speed decreases to zero during quick turn, the *Ability Float Data* allows the original speed to be captured so as to play the correct animation and transition back into movement after quick turn is complete.

### **Slot 0 Item ID**

Int value that indicates which item is currently equipped in Slot 0. This is set on the [Item component](#) under *Animator Item ID*.

### **Slot 0 Item State Index**

Int value that indicates which Slot 0 [Item Ability](#) is currently active.

### **Slot 0 Item State Index Change**

Bool to indicate if the Slot 0 Item State Index has changed.

### **Slot 0 Item Substate Index**

Int value that is a secondary index related to the *Item State Index*. This value is utilized for item abilities to provide more information outside of the *Item State Index* to allow variations of animations within one active ability. For example, melee attack utilizes the *Item Substate Index* to indicate which particular melee attack animation should play based on the sequence setting.

### **Slot 1 Item ID**

Int value that indicates which item is currently equipped in Slot 1. This is set on the [Item component](#) under *Animator Item ID*.

### **Slot 1 Item State Index**

Int value that indicates which Slot 1 [Item Ability](#) is currently active.

### **Slot 1 Item State Index Change**

Bool to indicate if the Slot1 Item State Index has changed.

### **Slot 1 Item Substate Index**

Int value that is a secondary index related to the *Item State Index*. This value is utilized for item abilities to provide more information outside of the *Item State Index* to allow variations of animations within one active ability. For example, when the secondary frag grenade is equipped the *Slot 1 Item Substate Index* will change to 11 if the character has run out of grenades to throw.

### **LegIndex**

Float value to indicate which leg is moving forward. This is enables smoother transitions back into any movement animation. Right leg is 0, Left leg is 1.

## **Default Animator Values**

When you add a new ability or item ability it will have a default value for the *Ability Index* or *Item State Index*. This value can be changed to one that fits your Animator better but the following are the default *Ability Index* values used:

### **Ability**

### **Ability Index**

Jump	1
Fall	2
Height Change	3
Die	4
Revive	5
Start Movement	6
Stop Movement	7
Quick Turn	8
Interact	9
Damage Visualization	10
Pickup Item	11
Ride	12
Melee Counter Attack Response	13
Drive	14
Generic	10000

The following are the default *Item State Index* values used for each Item Ability:

#### **Item Ability Item State Index**

Aim	1
Use	2
Reload	3
Equip	4
Unequip	5
Drop	6

The following are the default *Item ID* values used for each Item:

<b>Item</b>	<b>Item ID</b>
Assault Rifle	1
Pistol	2
Shotgun	3
Bow	4
Sniper Rifle	5
Rocket Launcher	6
Body	21
Sword	22
Knife	23
Katana	24
Shield	25
Frag Grenade	41
Flashlight	42
Fireball	61

Particle Stream	62
Ricochet	63
Heal	64
Shield Bubble	65
Teleport	66

The following are the default *Movement Set ID* values used for each item stance:

### **Stance Movement Set ID**

Default	0
Melee	1
Bow	2

## **First Person Arms**

The arms use a generic model setup so they require the animations to be created specifically for that rig and cannot use Unity's animation retargetting feature. An easy way to get started though is to use the third person humanoid animations for the first person view. You can do this by using a tool such as [FPS Mesh Tool](#). With the FPS Mesh Tool it can split your mesh up into just the arms and allow you to use the humanoid animations. While these animations won't look perfect for a first person view they will allow you to prototype quickly.

If you are creating your own generic arms you can follow [this page](#) for importing your animations. The table below lists the positives and negatives of using a humanoid rig for your character. We wanted the first person arms to be of the highest quality so we chose the generic arms option for the demo scene.

### **Humanoid Arm Rig Positives**

- Supports animation retargetting
- Supports Unity IK

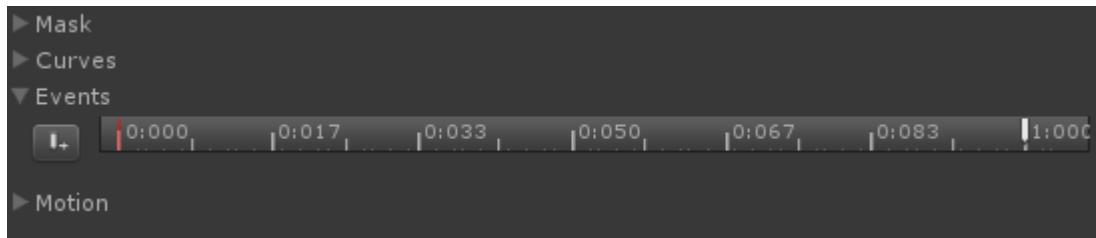
### **Humanoid Arm Rig Negatives**

- Contains unnecessary data within the animation
- Cannot have unique first person animations, for extra detail while the camera is closer to the first person weapon

## **Replacing Animations**

The Ultimate Character Controller uses Unity's Animator Controller to control all of its animations. The Animator Controller makes it extremely easy to replace the built-in animations with your own. If you have never used the Animator Controller before take a look at these [Unity Learn videos](#). These videos will give you an overview of the Animator Controller as well as how to setup your own animations. Most of this work is already taken care of for you with the Ultimate Character Controller, you'll just need to replace the

animation clips with your own. Unity also did a [live training video](#) which goes into depth for how to setup a humanoid character with the Animator Controller.

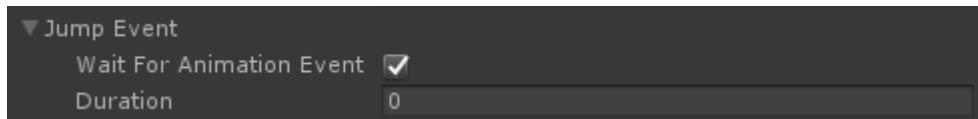


As you are replacing the animations there are two major things to keep in mind:

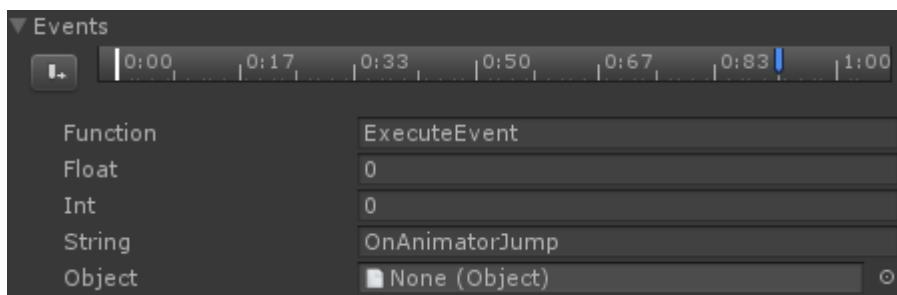
- Any [animation events](#) on the original animation should also be added to the new animation. The [Animation Event Trigger](#) within the inspector makes it easy to use a timer instead of animation events, but if any animations are already setup then they should continue to be used unless you change the value within this trigger.
- Ensure you replace the animations for all of the layers. As an example if you are replacing the walk animation within the base layer you should also ensure the upper body layers also match this new walk animation for each item.

## Animation Event Trigger

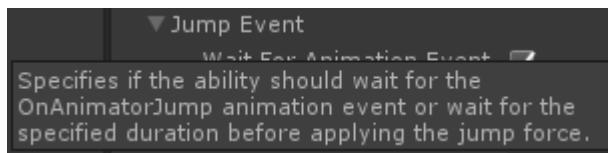
As you are looking through the inspector fields on the Ultimate Character Controller objects you'll see a field which looks similar to this field within the Jump ability:



This field is an Animation Event Trigger and it allows you to specify if the object should wait for an [animation event](#) before the event is triggered or if a timer should be used instead with a specified duration. In this screenshot *Wait For Animation Event* is enabled which means the Jump ability will wait for the "OnAnimatorJump" event. For the animation event ensure the *Function* name is "ExecuteEvent" and the *String* value is the animation event that you want to call.



The animation event that it is looking for can be determined by either hovering over the *Jump Event* field and reading the tooltip or look at the Tooltip attribute for that field within the code.



## Animation Event Debugging

If *Wait For Animation Event* is enabled the animator will send the event. Debugging when this event is sent can be done by enabling *Log Events* on the [Animator Monitor](#) component. In the OnAnimatorJump example the following will be outputted to the console when *Log Events* is enabled:

```
Execute OnAnimatorJump
UnityEngine.Debug:Log(Object)
Execute OnAnimatorJump
UnityEngine.Debug:Log(Object)
Opsive.UltimateCharacterController.Character.AnimatorMonitor:ExecuteEvent(String) (at
Assets/Opsive/UltimateCharacterController/Scripts/Character/AnimatorMonitor.cs:945)
UnityEngine.Animator:Update(Single)
Opsive.UltimateCharacterController.Character.AnimatorMonitor:UpdateAnimatorInternal(Single) (at
Assets/Opsive/UltimateCharacterController/Scripts/Character/AnimatorMonitor.cs:346)
Opsive.UltimateCharacterController.Character.AnimatorMonitor:UpdateAnimator(Single) (at
```

The *Animator.Update* line is highlighted because it is here that we can see that the Animator is responsible for sending the animation event. If *Wait For Animation Event* was not selected then the event would be sent from the Scheduler and the call stack would be different.

If you are having issues with the timing of when events occur a good debug step is to look at this call stack to determine when the event is being triggered. If it is being triggered too soon and it is being triggered from the Animator then you should move the animation event later in the animation. If you are working in a first person perspective and have a full body animation ensure that this event is changed on both the first person and third person Animator.

## Input

The Ultimate Character Controller supports input via keyboard/mouse, controllers, and mobile input with virtual controls. The controller support uses Unity's Input Manager which doesn't have wide controller support - if your game is going to be making use of controllers we highly recommend an asset that is dedicated to controller support such as Rewired or InControl. The Ultimate Controller is integrated with both of these assets.

The Player Input component is base component that the Unity Input and integration components derive from. By using the Player Input component any new input can be added by just swapping out the component and the rest of the code does not need to change. If your character is an AI agent then no input component should be added - the AI implementation is responsible for interacting with the character.

## Events

The "OnEnableGameplayInput" [event](#) can be sent if you'd like to disable (or enable) character and camera input. As an example the following will disable the input on the

character specified by `m_Character`:

```
EventHandler.ExecuteEvent(m_Character, "OnEnableGameplayInput",  
false);
```

The `Opsive.Shared.Events` namespace must be included for the `EventHandler` class to be found.

```
using UnityEngine;  
using Opsive.Shared.Events;  
  
public class MyObject : MonoBehaviour  
{  
    [Tooltip("A reference to the Ultimate Character Controller  
character.")]  
    [SerializeField] private GameObject m_Character;  
  
    /// <summary>  
    /// Disable the input.  
    /// </summary>  
    private void Start()  
    {  
        EventHandler.ExecuteEvent(m_Character,  
"OnEnableGameplayInput", false);  
    }  
}
```

## API

The PlayerInput API follows the same standard as Unity's input class with `GetButton`, `GetButtonDown`, `GetButtonUp`, and `GetAxis` methods. This component also allows for double press or long press detection with `GetDoubleClick` and `GetLongPress`.

As an example the following code will check if the "Jump" button is down:

```
using UnityEngine;  
using Opsive.UltimateCharacterController.Input;  
  
public class MyObject : MonoBehaviour  
{  
    [Tooltip("A reference to the Ultimate Character Controller  
character.")]  
    [SerializeField] private GameObject m_Character;  
  
    /// <summary>  
    /// Initialize a reference to PlayerInput.  
    /// </summary>  
    private void Awake()  
    {  
        m_PlayerInput = m_Character.GetComponent<PlayerInput>();  
    }
```

```

}

/// <summary>
/// Check for the jump button press.
/// </summary>
private void Update()
{
    if (m_PlayerInput.GetButtonDown("Jump")) {
        // Do Jump.
    }
}
}

```

## Inspected Fields

### Horizontal Look Input Name

The name of the horizontal camera input mapping.

### Vertical Look Input Name

The name of the vertical camera input mapping.

### Look Vector Mode

Specifies how the look vector is assigned. If a mouse is being used then the look vector will almost always be based on the mouse movement. If a controller is used then the controller will supply the look input.

- *Smoothed*: Apply a smoothing to the look vector. This smoothing interpolates the input over several frames to reduce jerky player input.
- *Unity Smoothed*: Uses the input's direct value.
- *Raw*: Use the input's direct (raw) value.
- *Manual*: The look vector is assigned manually. This is useful for VR head movement.

### Look Sensitivity

If using look smoothing, specifies how sensitive the mouse is. The higher the value to more sensitive.

### Look Sensitivity Multiplier

If using look smoothing, specifies a multiplier to apply to the LookSensitivity value.

### Smooth Look Steps

If using look smoothing, the amount of history to store of previous look values.

### Smooth Look Weight

If using look smoothing, specifies how much weight each element should have on the total smoothed value (range 0-1). Reducing the weight to 0 is similar to setting the Look Vector Mode to Raw. A value of 1 will cause the result to be a simple average of all of the recently sampled smooth steps (and will feel very laggy).

#### **Smooth Exponent**

If using look smoothing, specifies an exponent to give a smoother feel with smaller inputs.

#### **Look Acceleration Threshold**

If using look smoothing, specifies a maximum acceleration value of the smoothed look value (0 to disable).

#### **Controller Connected Check Rate**

The rate (in seconds) the component checks to determine if a controller is connected. The only way to check if a controller is connected in Unity generates garbage so this value should be set to 0 if you do not plan on supporting controllers.

#### **Connected Controller State**

The state that should be activated when the controller is connected.

#### **Force Input**

Specifies if any input type should be forced. By default the controller will enable the virtual buttons for a mobile platform but this field allows you to force standalone input on a mobile platform. Virtual input can also be forced for a platform that normally does not use virtual controls.

#### **Disable Cursor**

Should the cursor be disabled?

#### **Enable Cursor with Escape**

Should the cursor be enabled when the escape key is pressed?

#### **Prevent Look Vector Changes**

If the cursor is enabled with escape should the look vector be prevented from updating?

## **Virtual Controls**

For more robust virtual controls we recommend an input asset dedicated to mobile input such as [Control Freak](#) or [Easy Touch](#) which the Ultimate Character Controller is integrated with. A basic set of virtual controls are included though which allow for virtual buttons, virtual joysticks, or virtual touchpads. The built-in virtual controls will automatically be enabled on a mobile platform. They can also be force enabled within the Unity Input

component.

All virtual controls must be a child of the Virtual Controls Manager component. This allows the virtual controls to associate themselves with the correct character.

## Setup

1. Open the Setup Manager.
2. Click on the “Add UI” button. This will add the Demo/Prefabs/UI/VirtualControls prefab as a child of the Canvas GameObject.

### **Virtual Button**

The virtual button allows for a virtual control that the player can press.

#### **Button Name**

The name of the button input.

### **Virtual Joystick**

The virtual joystick represents a joystick that stays within the specified radius range. When the press is released the joystick know will snap back to the starting position.

#### **Horizontal Input Name**

The name of the horizontal input axis.

#### **Vertical Input Name**

The name of the vertical input axis.

### **Joystick**

A reference to the joystick knob that moves with the press position.

### **Radius**

The maximum number of pixels that the joystick can move.

### **Deadzone Radius**

The joystick will return a zero value when the radius is within the specified deadzone radius of the center.

### **Virtual Touchpad**

The virtual touchpad that will return an input axis position of the press relative to the starting press position.

## **Horizontal Input Name**

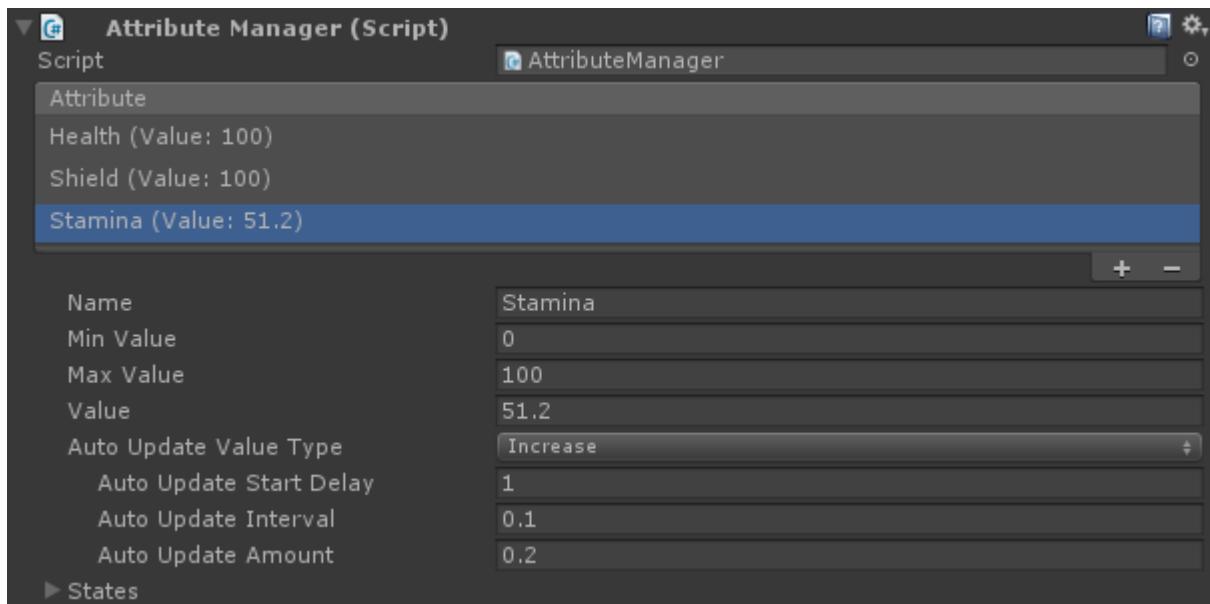
The name of the horizontal input axis.

## **Vertical Input Name**

The name of the vertical input axis.

# **Attributes**

The Attribute Manager is a component added to your character which can describe a set of values that change over time. These values are called attributes and can be used for the character's health, shield, stamina, hunger, etc. Any number of attributes can be added to the Attribute Manager.



In this screenshot the Attribute Manager is visible and there are three attributes added: Health, Shield, and Stamina. The Stamina attribute is selected and the details will be shown for any selected attribute. Attributes can be added and removed by selecting the plus or minus icons on the bottom right of the list.

### **Name**

The name of the attribute. This is used to identify the attribute that should be retrieved from the Attribute Manager and must be unique.

### **Min Value**

The minimum value that the attribute's value can be. This value cannot be greater than the Max Value.

### **Max Value**

The maximum value that the attribute's value can be. This value cannot be less than the Min Value.

## **Value**

The current value of the attribute.

## **Auto Update Value Type**

If the value isn't at the min or max value then it can automatically be updated to reach the min or max value.

- *None*: The attribute will not automatically update.
- *Decrease*: Automatically decrease the value until the *Min Value* is reached.
- *Increase*: Automatically increase the value until the *Max Value* is reached.

Auto updating is useful in situations such a shield or stamina where the value should automatically regenerate to the max value. Imagine if the character is taking damage and the shield's value is being affected by this damage. Once the character is no longer taking any damage the shield should start to regenerate back to its max value.

## **Auto Update Start Delay**

Specifies a time (in seconds) that the attribute should start to be auto updated after the attribute's value is changed.

## **Auto Update Interval**

After the start delay has elapsed the interval specifies how often the value is continuously updated until it reaches the target value.

## **Auto Update Amount**

The amount value specifies the amount to add to the attribute's current value every time the auto update occurs.

## **API**

### **Get Attribute**

The most common operation that you'll perform on the Attribute Manager is to get an attribute and then modify its value. In the following code snippet an attribute will be retrieved and a value of 10 will be added to the Health attribute.

```
var attribute = m_AttributeManager.GetAttribute("Health");
attribute.Value += 10;
```

No checks need to be performed to ensure the value stays within the Min and Max value as the value will automatically be clamped.

### **Value Change**

You can receive notification when an attribute changes value by registering for the "OnAttributeUpdateValue" event. The actual attribute whose value was changed will be sent

as the only parameter of this event.

```
private void OnUpdateValue(Attribute attribute)
{
    Debug.Log("New Attribute Value:" + attribute.Value);
}
```

## Health

The Health component uses the Attribute Manager to determine how much health or shield the object has left. The Health component is a generic component that can work on any object - it doesn't have to only be added to an Ultimate Character Controller character. The Character Health is a subclass of the Health component and this allows the character to take fall damage. When the object dies a set of objects can be spawned or destroyed. This is useful for example if you want to spawn wood shards in the case of a wooden crate.

### API

If you'd like to deal damage to the health component directly you can call the *Damage* method. This method is overloaded which allows for many different damage options. *ImmediateDeath* will kill the object immediately. The object can be healed with the *Heal* method.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Traits;

public class MyObject : MonoBehaviour
{
    [Tooltip("The character that has the health component.")]
    [SerializeField] protected GameObject m_Character;

    /// <summary>
    /// Damages and heals the character.
    /// </summary>
    private void Start()
    {
        var health = m_Character.GetComponent<Health>();
        // Cause 10 damage to the character.
        health.Damage(10);

        // Heal the character by 10.
        health.Heal(10);

        // Kill the character.
        health.ImmediateDeath();
    }
}
```

## Events

The Health component exposes three events: OnHealthDamage, OnHeal, and OnDeath. These events are executed through the [Event System](#) and are also executed with Unity's built-in [event system](#).

### OnHealthDamage

When the Health component takes damage it will execute the "OnHealthDamage" event. This event can then be used by other components to determine that an action needs to be taken. For example, using the event system you can register for the OnHealthDamage event:

```
EventHandler.RegisterEvent<float, Vector3, Vector3, Game0bject, Collider>(game0bject, "OnHealthDamage", OnDamage);
```

And then the OnDamage method will execute when the Health component takes damage:

```
private void OnDamage(float amount, Vector3 position, Vector3 force, Game0bject attacker, Collider hitCollider)
{
    Debug.Log("Object took " + amount + " damage at position " +
position + " with force " + force + " by attacker " + attacker + ".
The collider " + hitCollider + " was hit.");
}
```

With this the full component looks like:

```
using UnityEngine;
using Opsive.Shared.Events;

public class My0bject : MonoBehaviour
{
    public void Awake()
    {
        EventHandler.RegisterEvent<float, Vector3, Vector3, Game0bject, Collider>(game0bject, "OnHealthDamage", OnDamage);
    }

    /// <summary>
    /// The object has taken damage.
    /// </summary>
    /// <param name="amount">The amount of damage taken.</param>
    /// <param name="position">The position of the damage.</param>
    /// <param name="force">The amount of force applied to the object
while taking the damage.</param>
    /// <param name="attacker">The Game0bject that did the
damage.</param>
    /// <param name="hitCollider">The Collider that was hit.</param>
    private void OnDamage(float amount, Vector3 position, Vector3
force, Game0bject attacker, Collider hitCollider)
```

```

{
    Debug.Log("Object took " + amount + " damage at position " +
position + " with force " + force + " by attacker " + attacker + ".
The collider " + hitCollider + " was hit.");
}

public void OnDestroy()
{
    EventHandler.UnregisterEvent<float, Vector3, Vector3,
GameObject, Collider>(gameObject, "OnHealthDamage", OnDamage);
}
}

```

### **OnHealthHeal**

The “OnHeal” event will be executed when the *Heal* method is called. This will increase the object’s health. You can subscribe to the OnHealth with:

```
EventHandler.RegisterEvent<float>(gameObject, "OnHealthHeal", OnHeal);
```

When OnHeal is called it will include the amount of health that the object was healed by.

```

/// <summary>
/// The object has been healed.
/// </summary>
/// <param name="amount">The amount of health that the object was
healed by.</param>
private void OnHeal(float amount)
{
    Debug.Log("The object was healed with " + amount + " health.");
}
```

### **OnDeath**

The Health component is also responsible for sending the “OnDeath” event which you can subscribe to by calling:

```
EventHandler.RegisterEvent<Vector3, Vector3, GameObject>(gameObject,
"OnDeath", OnDeath);
```

When OnDeath is called it will include the force, position, and attacker that caused the death. These parameters will be the same value as those received by “OnHealthDamage”.

```

/// <summary>
/// The object has died.
/// </summary>
/// <param name="position">The position of the force.</param>
/// <param name="force">The amount of force which killed the
object.</param>
/// <param name="attacker">The GameObject that killed the
```

```
object.</param>
private void OnDeath(Vector3 position, Vector3 force, GameObject
attacker)
{
    Debug.Log("The object was killed by " + attacker);
}
```

### **Invincible**

Set to true if the object is invincible and cannot be destroyed.

### **Time Invincible After Spawn**

The amount of time after the object spawns that it should be invincible. This is useful in an intense fire fight and you want to give the player a few moments to get their bearing before they can be damaged.

### **Health Attribute**

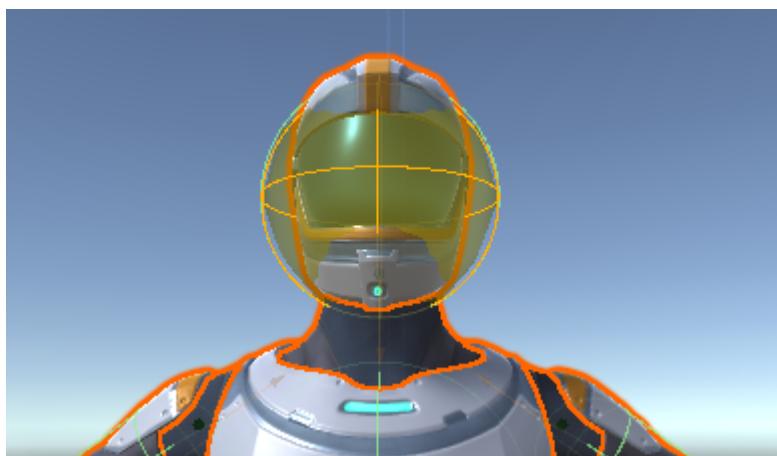
The name of the attribute within the Attribute Manager that should be used for the object's health. By default, the health will not regenerate automatically.

### **Shield Attribute**

The name of the attribute within the Attribute Manager that should be used for the object's shield. In many games the shield can regenerate so that's why the Health component can use a second attribute.

### **Hitboxes**

Maps a collider to a multiplier to adjust the damage based on the collider hit. This can be used for instant kills with a headshot. For an instant headshot kill the collider should be the character's head collider, and the multiplier should be an extremely large value. When the head collider is damaged the multiplier value will increase the amount of damage done and with a large enough value the object will instantly die.



### **Max Hitbox Collision Count**

The maximum number of colliders that can be detected when determining if a hitbox was damaged.

#### **Spawned Objects on Death**

A list of objects that should spawn when the object dies. This can be used for particle effects upon death.

#### **Destroyed Objects on Death**

A list of objects that should be destroyed when the object dies. This can be used to cleanup any objects that the object spawned.

#### **Deactivate on Death**

Set to true if the object should be deactivated when the object is killed.

#### **Death Layer**

The layer that the object should occupy upon death. This can be used to have the physics engine ignore the main CapsuleCollider on the character and instead use the ragdoll colliders.

#### **Take Damage Audio Clip Set**

A set of AudioClips which can be played when the object takes damage.

#### **Heal Audio Clip Set**

A set of AudioClips which can be played when the object is healed.

#### **Death Audio Clip Set**

A set of AudioClips which can be played when the object dies.

#### **Apply Fall Damage**

Added with the Character Health component, optionally determine if the character should take damage after landing from a predetermined height.

#### **Min Fall Damage Height**

The minimum height that the character needs to fall before taking any fall damage.

#### **Min Fall Damage**

The amount of damage to apply to the character when the character falls from the minimum height.

### **Max Fall Damage**

The amount of damage to apply to the character when the character falls from the maximum height.

### **Death Height**

The height at which the character should be killed because they fell from too great of a height.

### **Damage Curve**

Specifies how much fall damage to apply based on the fall height. The x axis (0 - 1) represents the distance fallen and the y axis (0 - 1) represents the amount of damage to apply. A value of 0 on the x axis represents the minimum fall damage height while a value of 1 represents the death height. A value of 0 on the y axis represents the minimum amount of fall damage while a value of 1 represents the maximum fall damage.

## **Inverse Kinematics (IK)**

[Inverse Kinematics](#)(IK) is the process of adjusting the character's bones after the animation frame has completed.IK is extremely useful to position the feet so they are always on the ground, rotating the upper body so the character always looks at the crosshairs, and positioning the hands so they are properly holding the item.The CharacterIK component included with the Ultimate Character Controller uses Unity's IK solution which only works on humanoid rigs.This means that the CharacterIK component will not work with your first person arms because it is a generic rig.

### **Inspected Fields**

#### **Base Layer Index**

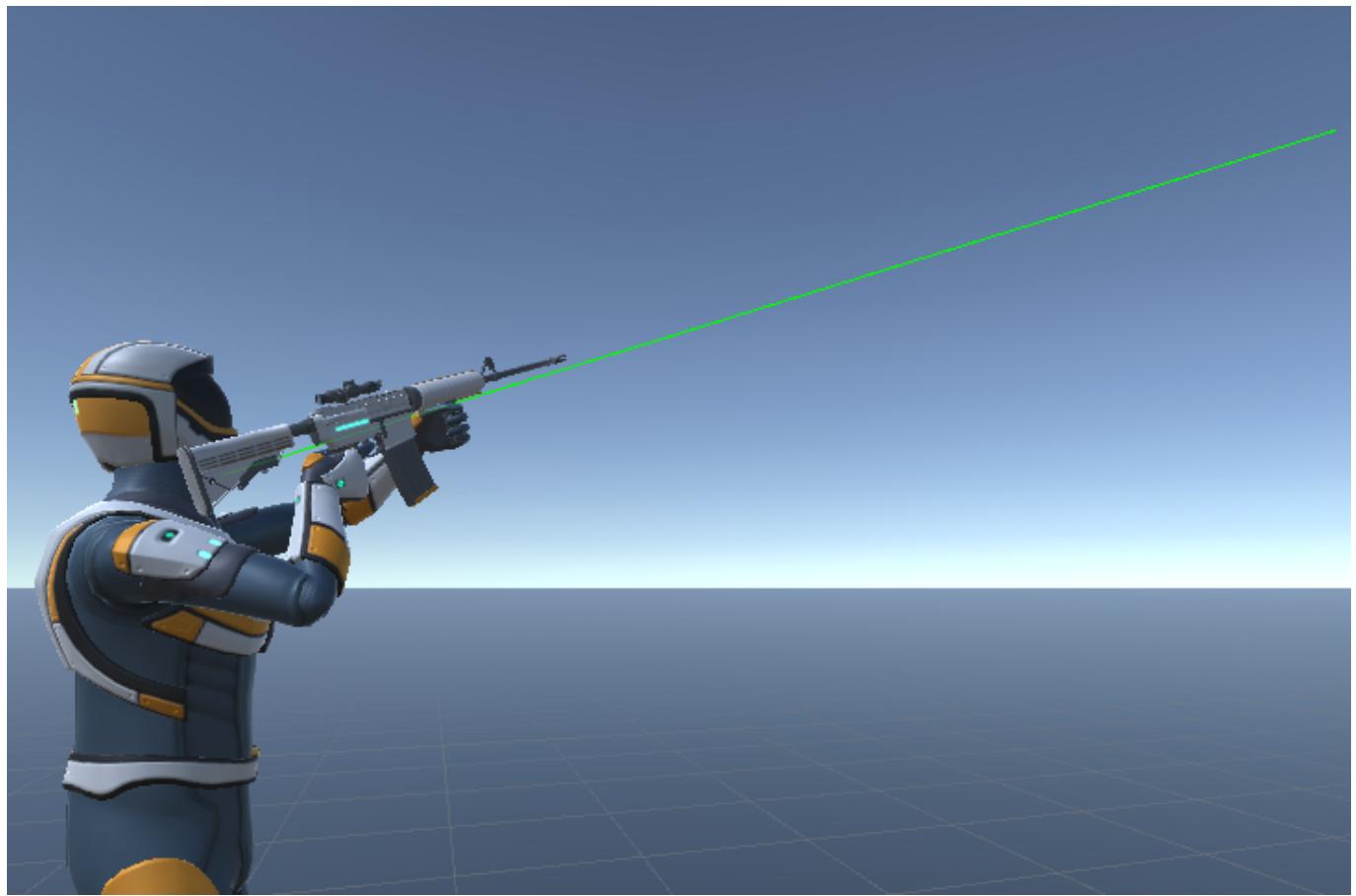
The index of the base layer within the Animator Controller.

#### **Upper Body Layer Index**

The index of the upper body layer within the Animator Controller.

#### **Debug Draw Look Ray**

Draw a debug line to see the direction that the character is facing (editor only). When this field is enabled you'll see a green line similar to the image below.

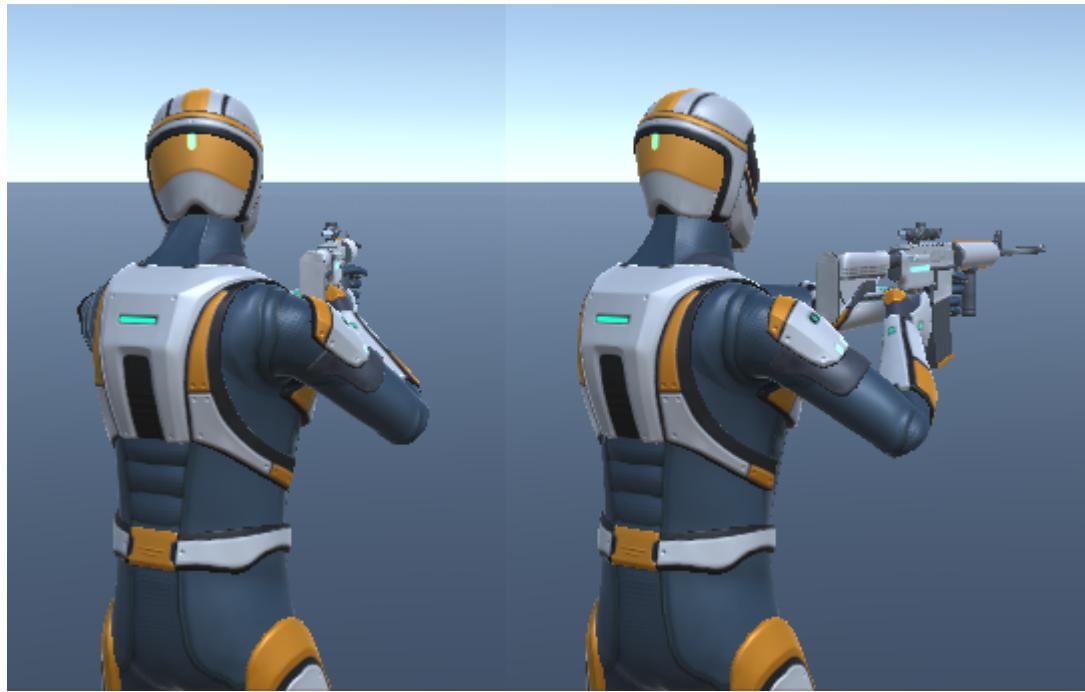


### Layer Mask

The layers that the component should use when determining the objects to test against. This mask allows the IK component to ignore invisible layers and is useful for stairs. For example, at design time you may place an invisible plane to act as the slope so the character smoothly moves up the stairs. IK shouldn't position the character's feet on this invisible plane, rather the feet should be positioned directly on the stairs. By using the Layer Mask field you can limit what objects the IK component detects.

### Look At Offset

Any offset to apply to the look at direction for the body and arms. The offset allows the character to adjust look direction without the camera actually changing angle. The left image below has an offset of  $(0, 0, 0)$  and the right image has an offset of  $(0.5, 0, 0)$ .



#### **Look at Body Weight**

Determines how much weight is applied to the body when looking at the target. A value of 0 indicates that the body should not look at the target while a value of 1 indicates the maximum amount of weight should be applied.

#### **Look at Head Weight**

Determines how much weight is applied to the head when looking at the target. A value of 0 indicates that the head should not look at the target while a value of 1 indicates the maximum amount of weight should be applied. The screenshots below show increasing weight values for the body and head look at weight.



#### **Look at Eyes Weight**

Determines how much weight is applied to the eyes when looking at the target. A value of 0 indicates that the eyes should not look at the target while a value of 1 indicates the maximum amount of weight should be applied.

#### **Look at Clamp Weight**

A value of 0 means the character is completely unrestrained in motion, 1 means the character motion completely clamped (look at becomes impossible) (0-1).

#### **Look at Adjustment Speed**

The speed at which the look at weight should adjust.

#### **Hips Position Adjustment Speed**

The speed at which the hips position should adjust between using IK and not using IK. The hips will change position when the character is standing on uneven ground. As an example, imagine that the character is standing on a set of stairs. The stairs have two sets of colliders: one collider which covers each step and another plane collider at the same slope as the stairs. The character's collider is going to be resting on the plane collider while standing on the stairs and the IK system will be trying to ensure the feet are resting on the stairs collider. In some cases the plane collider may be relatively far above the stair collider so the hip needs to be moved down to allow the character's foot to hit the stair collider.



#### **Foot Offset Adjustment**

The offset of the foot between the foot bone and the base of the foot. This value should be increased if the bottom of your character's foot is intersecting with the ground.

#### **Foot Weight Active Adjustment Speed**

The speed at which the foot weight should adjust to when foot IK is active.

#### **Foot Weight Inactive Adjustment Speed**

The speed at which the foot weight should adjust to when foot IK is inactive.

#### **Upper Arm Weight**

Determines how much weight is applied to the upper arms when looking at the target. A value of 0 indicates that the upper arms should not look at the target while a value of 1 indicates the maximum amount of weight should be applied. In the left screenshot below the upper arm weight has a value of 0 and the right screenshot has a value of 1.



#### **Upper Arm Adjustment Speed**

The speed at which the upper arm rotation should adjust to using IK and not using IK.

#### **Hand Weight**

Determines how much weight is applied to the hands when looking at the target. A value of 0 indicates that the hands should not look at the target while a value of 1 indicates the maximum amount of weight should be applied. In the left screenshot below the weight has a value of 0 and the right screenshot has a weight of 1. The upper arms have a weight of 0 in this screenshot which is why the left hand in the image on the right isn't positioned correctly.



#### **Hand Adjustment Speed**

The speed at which the hand position/rotation should adjust to using IK and not using IK.

#### **Hand Position Offset**

Specifies a local offset to add to the position of the hands.

#### **Left Hand Position Spring**

The left hand positional spring used for IK movement. This can be used for adding positional effects such as recoil.

#### **Left Hand Rotation Spring**

The left hand rotational spring used for IK movement. This can be used for adding rotational effects such as recoil.

#### **Right Hand Position Spring**

The right hand positional spring used for IK movement. This can be used for adding positional effects such as recoil.

#### **Right Hand Rotation Spring**

The right hand rotational spring used for IK movement. This can be used for adding rotational effects such as recoil.

# Objects

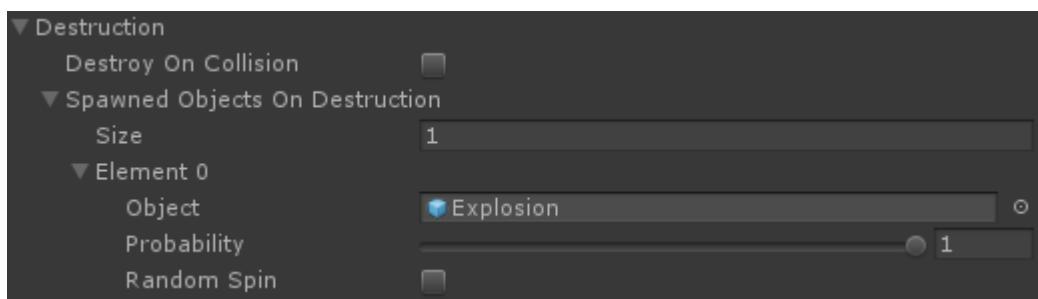
## Explosions

The Explosion component is a relatively simple component which will apply a force and damage to any nearby Rigidbodies or objects with the Health component attached. When it is spawned it can optionally explode immediately or wait for the Explode method to be called.

### Setup

An explosion can be setup by performing the following:

1. Open the [Object Manager](#) and create a new object of type Explosion.
2. Specify the material used by the Particle System.
3. Optionally add an AudioSource if the explosion should play an audio clip when it explodes.
4. Assign the prefab to the object that should cause the explosion. As an example it can be assigned to the Frag Grenade under the Spawned Objects on Destruction foldout:



### Inspected Fields

#### Explode On Enable

Should the object explode when the object is enabled?

#### Radius

Determines how far out the explosion affects other objects. The magnitude of the force and damage will be scaled based on the distance that the object was hit - the further from the explosion the less force/damage the object will receive.

#### Damage Amount

The maximum amount of damage to explosion applies to objects with the Health component.

#### Impact Force

The maximum amount of force the explosion applies to nearby Rigidbody/IForceObject objects.

## **Impact Layers**

The layers that the explosion can affect.

## **Line Of Sight**

Does the explosion require line of sight in order to damage the hit object

## **Lifespan**

The duration of the explosion.

## **Max Collision Count**

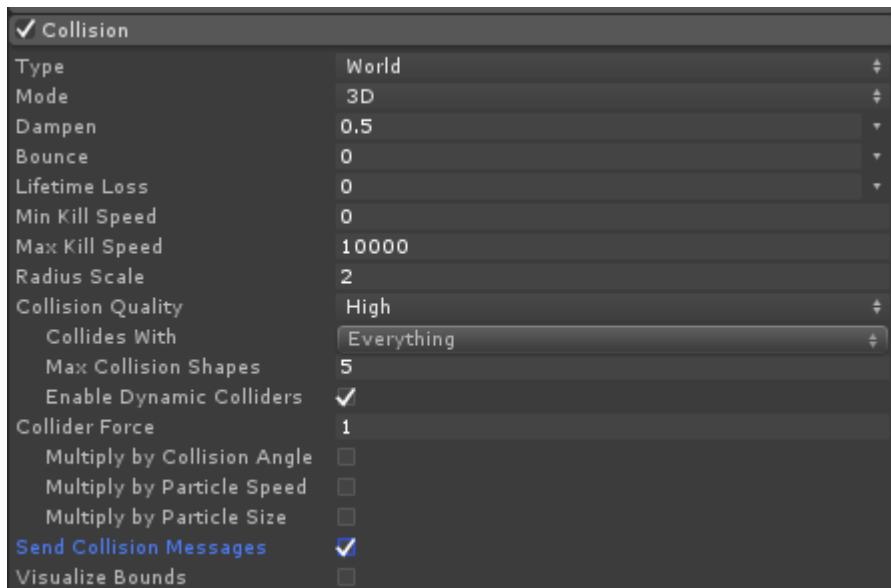
The maximum number of objects that the explosions can detect.

## **Explosion Audio Clip Set**

A set of AudioClips that can be played when the explosion occurs. The AudioClip that is played will randomly be selected out of this list.

# **Magic Particle**

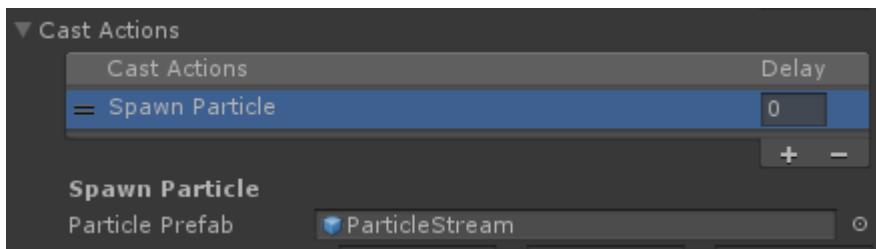
The MagicParticle will perform a Magic Item impact when it collides with an object. In order for this to work correctly the ParticleSystem must have collisions enabled and the [Send Collision Event](#) parameter enabled.



## **Setup**

A new projectile can be created by performing the following:

1. Open the [Object Manager](#) and create a new object of type Magic Particle.
2. Add the Spawn Particle Cast Action to the Magic Item.
3. Assign the prefab to the *Particle Prefab* Cast Action:



# Object Pickup

The Object Pickup component is an abstract class that allows the character to pickup objects such as a health pack or item. The Object Pickup object can be created with the Object Manager.

## Inspected Fields

### Trigger Enable Delay

The amount of time to enable the trigger after the object has been enabled and the rigidbody has stopped moving.

### Pickup On Trigger Enter

Should the item be picked up when the character enters the trigger?

### Rotation Speed

The amount that the object should rotate while waiting to be picked up.

### Pickup Message Text

The text that should be shown by the message monitor when the object is picked up.

### Pickup Message Icon

The sprite that should be drawn by the message monitor when the object is picked up.

### Pickup Audio Clip Set

A set of AudioClips that can be played when the object is picked up.

# Health Pickup

The Health Pickup object inherits the [Object Pickup](#) component and will replenish the character's health if it is low. The shield is replenished first and then any remaining health amount will be used for the standard health. If the character's health is not low then the health pickup will only disappear if *Always Pickup* is enabled.

## Setup

An explosion can be setup by performing the following:

1. Open the [Object Manager](#) and create a new object of type Health Pickup.
2. Specify the amount of health that should be replenished when the object is picked up.
3. Place the prefab in your scene.

## Inspected Fields

### Health Amount

The amount of health to replenish.

### Always Pickup

Should the object be picked up even if the object has full health?

# Item Pickup

The Item Pickup object inherits the [Object Pickup](#) component and contains a list of Item Identifiers and corresponding count that should be added to the inventory. This can be used both for pickups within the scene or as a *Drop Prefab* specified by the Item. *Drop Prefabs* are dropped by the character when the item is removed.

## Setup

An explosion can be setup by performing the following:

1. Open the [Object Manager](#) and create a new object of type Item Pickup or Dropped Item.
2. If the Item Pickup type was selected then the picked up Item Definitions should be specified within the Item Pickup component. A Dropped Item will automatically inherit the Item Definitions that the character is carrying so no Item Definition need to be specified.
3. Add the object to the scene or assign it to the *Drop Prefab* field of the Item component.

## Inspected Fields

### Always Pickup

Should the object be picked up even if the inventory cannot hold any more of the item?

### Item Pickup Set

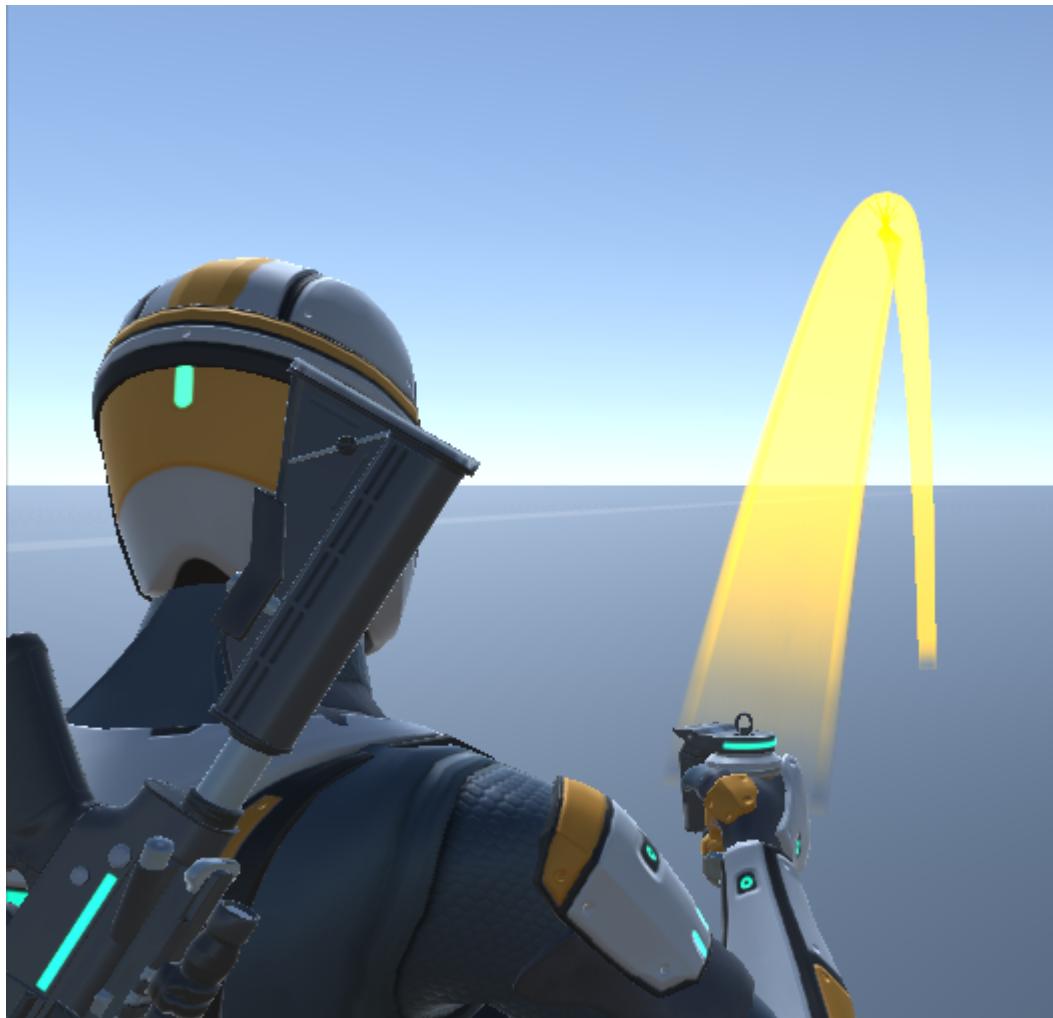
An array of items and ItemSets to pick up.

### Item Definition Amounts

An array of Item Definitions to be picked up.

# Trajectory Object

The Trajectory Object is the base component for any object that is projected from the character. This can include a wide range of objects including rockets, arrows, grenades, or shells. If the Trajectory component is added directly to the character then it can also show a trajectory curve of the direction that the object will follow.



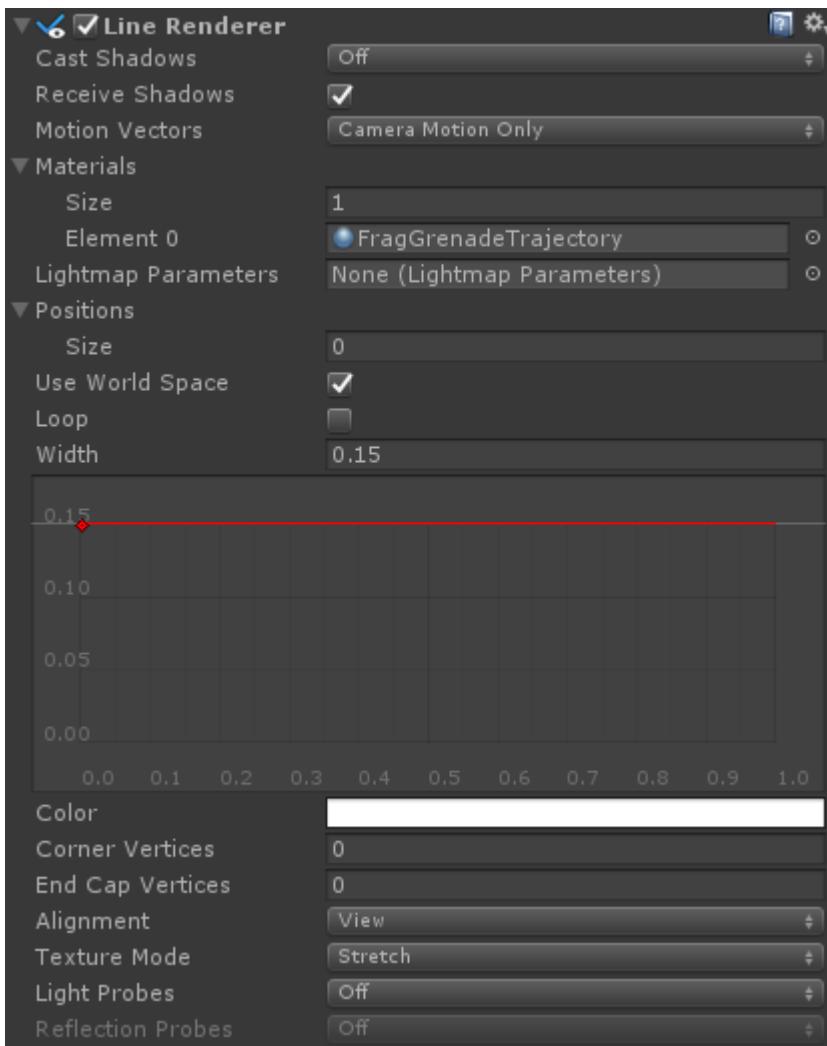
## Visible Trajectory Curve Setup

The Trajectory Object component can optionally display a curve indicating where the object is going to move towards when launched. This is a deterministic curve so the path will be accurate for where the object lands. This curve can also be used within VR to show the location that the first person player is going to teleport to.

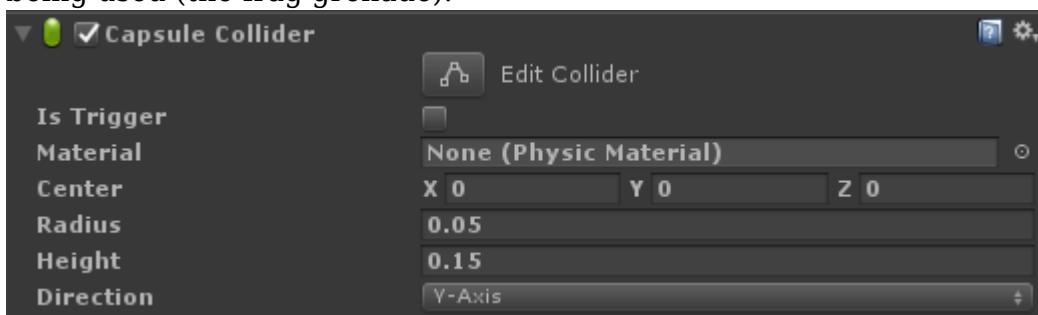
1. Select the GameObject that you want the curve to affect. For our example we selected the Nolan/Items/PrimaryFragGrenade GameObject since this curve should be displayed by the ThrowerItem.
2. Add the following components:
  - Trajectory Object
  - Line Renderer
  - Capsule Collider. This component is optional and can be nothing, a Capsule Collider, or Sphere Collider. This component will tell the Trajectory Object what

type of collisions to use (if no collider is added a standard raycast is used). The Capsule Collider was chosen for this example because the frag grenade uses a Capsule Collider.

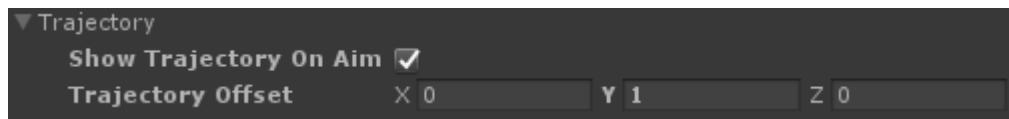
3. The Trajectory Object values should match the values of the Trajectory Object that is being used. For this example all of the default Trajectory Object values can be left alone.
4. The following Line Renderer values should be changed:
  - Cast Shadows: Off
  - Receive Shadows: Disabled
  - Materials: your Line Renderer material
  - Positions: 0
  - Width: 0.15



5. The Capsule Collider values should match the Capsule Collider values on the object being used (the frag grenade):



6. The final step is to enable the trajectory upon aim with the frag grenade. This can be found under the Trajectory foldout of the ThrowableObject component:



## Inspected Fields

### Initialize On Enable

Should the component initialize when enabled?

### Mass

The mass of the object. The mass affects the velocity, the larger the mass the less the velocity.

### Start Velocity Multiplier

A multiplier to apply to the starting velocity. This is useful for simulation when the trajectory curve should account for any velocity that is added to the character when the object is being thrown.

### Gravity Magnitude

The amount of gravity to apply to the object. The gravity direction is determined by the character's up direction.

### Speed

The movement speed of the object. This value is multiplied by the velocity to give the final new position. A larger speed value is most useful if the trajectory curve is visible to reduce the number of total positions.

### Rotation Speed

The rotation speed of the object. This value is slerped between the original rotation and the target rotation.

### Damping

The damping value to apply to the movement. The higher the damping the quicker the object will stop moving.

### Rotation Damping

The damping value to apply to the rotation. The higher the damping the quicker the object will stop rotating.

### Rotate In Move Direction

Should the object rotate in the direction that the object is moving? This is useful for a bow and arrow where the arrow should face in the movement direction.

### **Settle Threshold**

When the velocity and torque have a square magnitude have a square magnitude value less than the specified value then the object will be considered settled. Settled objects do not update anymore and will not move/rotate. Set to 0 if the object should never settle.

### **Sideways Settle Threshold**

Specifies if the collider should settle on its side or upright. The higher the value the more likely the collider will settle on its side. This is only used for CapsuleColliders and BoxColliders. This is useful for bullet shells when you want them to land upright.

### **Start Sideways Velocity Magnitude**

Starts to rotate to the settle rotation when the velocity magnitude is less than the specified values.

### **Impact Layers**

The layers that the object can collide with.

### **Surface Impact**

The identifier that is used when the object collides with another object. If the identifier is null it can be overridden by the object initializing the trajectory object.

### **Force Multiplier**

When a force is applied the multiplier will modify the magnitude of the force.

### **Collision Mode**

Specifies how the object should behave after hitting another collider.

- *Collide*: Collides with the object. Does not bounce.
- *Reflect*: Reflect according to the velocity.
- *Random Reflect*: Reflect in a random direction. This mode will make the object nondeterministic but for visual only objects such as shells this is preferred.
- *Ignore*: Passes through the object. A collision is reported.

### **Reflect Multiplier**

Specifies the multiplier to apply to the reflectvelocity (if the object can reflect).

### **Max Collision Count**

The maximum number of objects that the projectile can collide with at a time.

### **Max Position Count**

The maximum number of positions any single curve amplitude can contain. This is only used

when the curve is being displayed.

# Grenade

Grenades are [Trajectory Objects](#) that can destroy themselves after they collide with an object or a set amount of time. They are thrown by the `ThrowableItem` component and can explode in the character's hand if they aren't thrown fast enough.

## Setup

A new grenade can be created by performing the following:

1. Open the [Object Manager](#) and create a new object of type Grenade.
2. A Capsule Collider will be added to the shell by the Object Manager. A Sphere Collider can also be used if the shell is more spherical.
3. Adjust the Grenade values to match the type of grenade. In most cases only the `Spawned Objects On Destruction` value will need to be adjusted.
4. Assign the prefab to the `Throwable Item` that should throw it. This is assigned to the `Thrown Object` field under the `Throw` foldout:



## Inspected Fields

### Sticky Layers

The layers that the object can stick to.

### Destroy On Collision

Should the grenade be destroyed when it collides with another object? Most grenades will have this option set to false.

### Destruction Delay

The amount of time after a collision that the object should be destroyed.

### Spawned Objects On Destruction

The objects which should spawn when the object is destroyed. Each element within this array has the following options:

- *Object*: the object that should be spawned
- *Probability*: the likelihood that the object will be spawned (0 - 1). A higher value means it is more likely to be spawned.
- *Random Spin*: should a random spin be applied to the object after it has spawned?

### Lifespan

The length of time before the grenade destructs.

## Pin

A reference to the pin that is removed before the character throws the object (can be null).

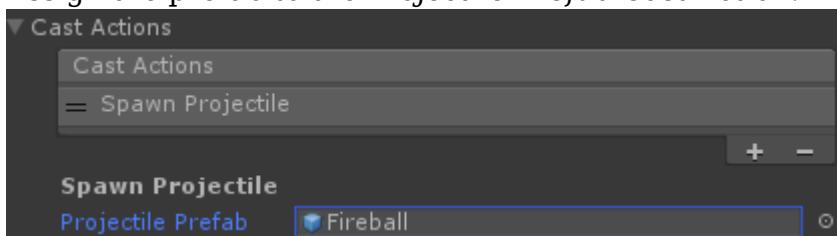
# Magic Projectile

Magic Projects are [Trajectory Objects](#) that can destroy themselves after a set amount of time. Magic Projectiles work with Magic Items to perform the impact actions when the projectile collides with another object.

## Setup

A new projectile can be created by performing the following:

1. Open the [Object Manager](#) and create a new object of type Magic Projectile.
2. Add the Spawn Projectile Cast Action to the Magic Item.
3. Assign the prefab to the *Projectile Prefab* Cast Action:



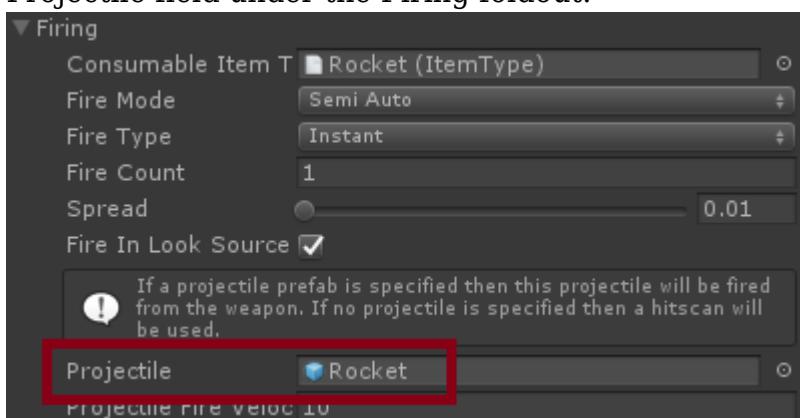
# Projectile

Projects are [Trajectory Objects](#) that can destroy themselves after they collide with an object or a set amount of time. They are used for objects that are fired from the Shootable Weapon component such as rockets or arrows.

## Setup

A new projectile can be created by performing the following:

1. Open the [Object Manager](#) and create a new object of type Projectile.
2. Adjust the Projectile values to match the type of projectile. The most common values that will be adjusted are Destroy On Collision and Spawns Objects On Destruction.
3. Assign the prefab to the Shootable Weapon that should fire it. This is assigned to the Projectile field under the Firing foldout:



## Impact Callback

When the projectile collides with another object the “OnObjectImpact” event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. This event allows you to add new functionality when the impact occurs without having to change the class at all. The following example will subscribe to this event from a new component:

```
using UnityEngine;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Initialize the default values.
    /// </summary>
    public void Awake()
    {
        EventHandler.RegisterEvent<float, Vector3, Vector3,
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);
    }

    /// <summary>
    /// The object has been impacted with another object.
    /// </summary>
    /// <param name="amount">The amount of damage taken.</param>
    /// <param name="position">The position of the damage.</param>
    /// <param name="forceDirection">The direction that the object
took damage from.</param>
    /// <param name="attacker">The GameObject that did the
damage.</param>
    /// <param name="attackerObject">The object that did the
damage.</param>
    /// <param name="hitCollider">The Collider that was hit.</param>
    private void OnImpact(float amount, Vector3 position, Vector3
forceDirection, GameObject attacker, object attackerObject, Collider
hitCollider)
    {
        Debug.Log(name + " impacted by " + attacker + " on collider "
+ hitCollider + ".");
    }

    /// <summary>
    /// The GameObject has been destroyed.
    /// </summary>
    public void OnDestroy()
    {
        EventHandler.UnregisterEvent<float, Vector3, Vector3,
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);
    }
}
```

}

## Inspected Fields

### Sticky Layers

The layers that the object can stick to.

### Destroy On Collision

Should the projectile be destroyed when it collides with another object? A rocket will want this option enabled but an arrow normally will not.

### Destruction Delay

The amount of time after a collision that the object should be destroyed.

### Spawned Objects On Destruction

The objects which should spawn when the object is destroyed. Each element within this array has the following options:

- *Object*: the object that should be spawned
- *Probability*: the likelihood that the object will be spawned (0 - 1). A higher value means it is more likely to be spawned.
- *Random Spin*: should a random spin be applied to the object after it has spawned?

### Lifespan

The length of time that the projectile should exist before it deactivates if no collision occurs. This will prevent the object from existing forever if for example it is shot into the air and no gravity is applied to it.

## Shell

Shells are a [Trajectory Object](#) that will randomly bounce when they collide with another object. Shells are for visuals only and are not synchronized across the network.

### Setup

A new shell can be created by performing the following:

1. Open the [Object Manager](#) and create a new object of type Shell.
2. A Capsule Collider will be added to the shell by the Object Manager. A Sphere Collider can also be used if the shell is more spherical.
3. Adjust any interested Shell values. In most cases the defaults are a good start.
4. Assign the shell to the Shootable Weapon that should eject it. This is assigned to the Shell field under the Shell foldout:



## Inspected Fields

### Lifespan

Time to live in seconds before the shell is removed.

### Persistence

Chance of shell not being removed after settling on the ground. This allows shells to be removed early for to increase randomness.

# Moving Platforms

The moving platforms are a generic term meaning any object that can be moved/rotated within the scene. This includes elevators, trains, revolving doors, etc. The character can ride on top of or get pushed around by these moving platforms.

## Requirements

In order to stay properly synchronized with the character and camera the object must use the Moving Platform layer:



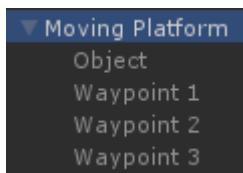
The Moving Platform or Animator Update component must be added to your movable object. The Moving Platform component should be used for any object that does not have an Animator. The Animator Update component is then used with the Animator.

One of these scripts must be attached in order to prevent the character from jittering on a moving platform. The Kinematic Object Manager updates the objects in a specified order to allow for the smooth movement and without one of these scripts attached the Kinematic Object Manager will not know the moving platform exists.

## Moving Platform Component

The Moving Platform component allows the object to move or rotate along a given waypoint path. An outline will show the shape of the object at each waypoint object, and a line will connect the waypoints indicating the path that will be traversed.

## Setup



Moving platforms can be setup by following:

1. Create a new empty GameObject that will act as the container for the actual moving platform object and waypoints.

2. Add your platform that will actually move as a child of the GameObject created in step 1. The Moving Platform component and any colliders should be added to this GameObject. This GameObject should be on the MovingPlatform.
3. Add any number of child GameObjects to the GameObject created in step 1. These GameObjects will serve as the waypoints that the platform will traverse.
4. Each waypoint should be specified in the Waypoints list under the Moving Platform component.

## Inspected Fields

### Update Location

Specifies the location that the object should be updated:

- *Update*: The object will be updated within Unity's Update loop.
- *FixedUpdate*: The object will be updated within Unity's FixedUpdate loop.

### Waypoints

The Waypoints that the moving platform will traverse. If no waypoints are set then the object will not change positions. With each waypoint a delay and state name can be specified along with the platform. The platform will stay at the waypoint for the specified delay and activate specified state when the platform start to traverse towards that waypoint.

### Direction

Specifies the direction that the platform should traverse.

- *Forward*: Moves from the waypoint at the least index to the greatest index within the waypoints array.
- *Backwards*: Moves from the waypoint at the greatest index to the least index within the waypoints array.

### Movement Type

Specifies how the platform traverses through waypoints.

- *Ping Pong*: Moves to the last waypoint and then back the way it came from.
- *Loop*: Moves to the last waypoint and then directly to the first waypoint.
- *Target*: Moves to the specified waypoint index.

### Target Waypoint

Specifies the waypoint index to move towards if using the Target Movement Type.

### Movement Speed

The speed at which the platform should move.

## **Movement Interpolation**

Specifies how the platform should interpolate the movement speed.

- *Ease In Out*: Gently moves into full movement and gently moves out of it at each waypoint.
- *Easy In*: Gently moves into full movement.
- *Ease Out*: Moves into full movement immediately and gently moves out of full movement at each waypoint.
- *Ease Out 2*: Moves into full movement immediately and moves out of full movement according to the movement speed.
- *Slerp*: Uses Vector3.Slerp to move in and out of movement according to the movement speed.
- *Lerp*: Uses Vector3.Lerp to move in and out of movement according to the movement speed.

## **Rotation Interpolation**

Specifies how the platform should interpolate the rotation speed.

- *Sync To Movement*: Rotates according to the movement speed.
- *Ease Out*: uses Quaternion.Lerp to lerp the rotation based on a linear curve.
- *Custom Ease Out*: Uses Quaternion.Lerp to lerp the rotation based on the Rotation Ease Amount value.
- *Custom Rotate*: Rotates according to the rotation speed.

## **Rotation Ease Amount**

Specifies the amount to ease into the target rotation if using the Custom Ease Out Rotation Interpolation mode.

## **Custom Rotation Speed**

Specifies the rotation speed if using the Custom Rotate Rotation Interpolation Mode.

## **Max Rotation Delta Angle**

The maximum angle that the platform can rotation. Set to -1 to have no max angle.

## **Character Trigger State**

The state name that should activate when the character enters the platform trigger. Note that the platform must have a trigger collider in order for the state to be activated.

## **Enable On Interact**

Should the moving platform wait to be enabled until it is interacted with? The [Interact ability](#) can start the interaction.

## **Change Directions On Interact**

Should the moving platform change directions when the Interact ability interacts with the platform? The [Interact ability](#) can start the interaction.

### **Gizmo Color**

The color to draw the editor gizmo in (editor only).

### **Draw Debug Labels**

Should the delay and distance labels be drawn to the scene view (editor only)?

# **Layer Manager**

When the character is created one of the required components is the Character Layer Manager. The Character Layer Manager specifies which layers the character should collide with or ignore. When the [Update Layers](#) button is pressed the controller will add the required layers to elements 26 - 31. If you have existing layers that occupy those elements then you can manually change these elements by modifying LayerManager.cs. Unfortunately because these layers have to be static there isn't a way to modify it without changing the source code. Ensure these layers are updated before the character is built so the correct indices will be used.

### **Enemy Layers**

Layer Mask that specifies the layer that the enemies use.

### **Invisible Layers**

Layer Mask that specifies any layers that are invisible to the character (such as water or invisible planes placed on top of stairs).

### **Solid Layers**

Layer mask that specifies any layers that represent a solid object (such as the ground or a moving platform).

An example use is for the collision detection by the character controller. The controller will use the Solid Layers mask to determine what is considered the ground to prevent the character from falling through the floor.

# **Surface System**

The Surface System is responsible for spawning effects caused by collisions and impacts. It is designed to be dynamic, powerful, and useful in a broad range of possible physics situations.

# Features Overview

## Surface Effect System

Spawns random sounds, prefabs, and decals upon impact/collision with objects and characters. To increase randomness sounds can have varying pitch, prefabs can have spawn probability, and decals can have a random scale.

## Decal Manager

Removes decals if they overlap with wall corners, and ages (weathers) remaining ones. Decals can fade and rotate.

## Character Foot Effects

Handles footstep and footprint effects with four footstep detection modes.

## Texture Fallbacks

Allows you to identify surfaces depending on terrain or object textures. Supports mapping UV regions within a texture.

## Default Fallbacks

Determines the best effect to play in situations where the system doesn't have information about the impact or surface type.

## Logic Overview

A Surface Impact and Surface Type trigger a Surface Effect.

The typical flow for an effect being generated is as follows:

1. A collision is caused by an object (such as the character or a projectile) with an assigned Surface Impact.
2. A Surface Type is found based off of the collision object.
3. A Surface Effect is then retrieved which should be used base off of the Surface Impact - Surface Type combination.
4. That Surface Effect finally spawns the particles, decals, objects, or sounds into the scene.

## Component Overview

The Ultimate Character Controller scene will have a Surface Manager, Decal Manager, and Object Pool component generated when the Setup Scene button is pressed within the Setup Manager. Surface Impact, Surface Type, and Surface Effect are ScriptableObjects created at design time. Surface Identifier and Character Foot Effects are assigned to an object within the scene.

## **Surface Manager**

Contains fallback Surface Types derived from the textures of a target object. Also contains default fallbacks for (potentially missing) impact and surface effects.

## **Decal Manager**

Handles the fading out (weathering) of decals and smart removal of any badly placed decals.

## **Object Pool**

Handles recycling of objects that are frequently destroyed and reused. Saves lots of memory and is good for performance.

## **Surface Impact**

Identifies the collision type for determining the Surface Effect. Typical Surface Impacts are Bullet Hit, Fall Impact, Footstep, etc. If you're coming from version 1 of UFPS this object was previously known as vp\_ImpactEffect.

## **Surface Type**

Identifies the surface of the object that was hit by the Surface Impact.

## **Surface Effect**

A recipe for a group of effects which should be spawned in response to a certain type of collision.

## **Surface Identifier**

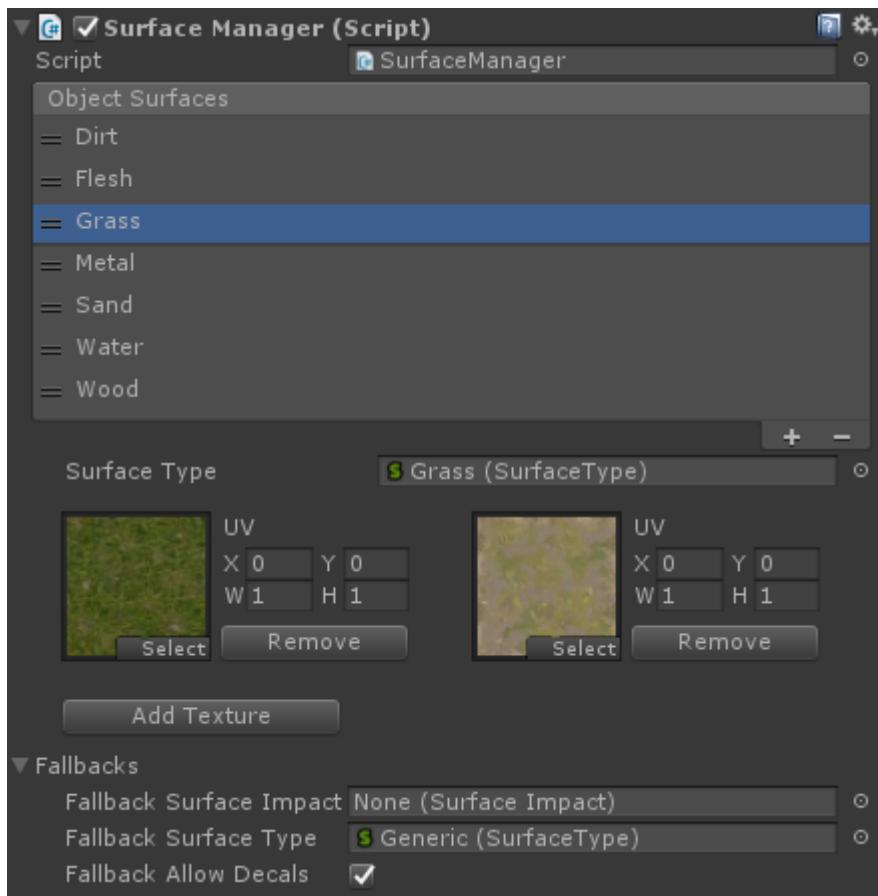
Associates scene colliders with Surface Types. Attaches to any and all scene object which can be static or moving. One per collider.

## **Character Foot Effects**

Attaches to the character and handles the footstep and footprint effects.

# **Surface Manager**

The Surface Manager component can be used to trigger effects depending on the object texture, thus reducing the need for assigning a Surface Identifier to every GameObject in the scene. It also has default fallbacks for (potentially missing) Surface Impact and Surface Types.



The Surface Manager will try to determine the Surface Type of the object hit from various sources including single textures, textures among multi-materials, and terrain textures. It is also possible to define UV regions for textures inside the inspector (for atlas maps).

## Object Surfaces

Click on the plus symbol of the reorderable list in order to add a new Object Surface. The Surface Type should be assigned along with a list of textures that correspond to that Surface Type.

The UV field allows for the surface to be restricted within a rectangular region inside a texture. The Object Surface element higher in the list will be selected if two Object Surfaces share the same texture and having overlapping UVs. If you want to have several surfaces inside a single texture then you should add that texture to each Object Surface Element.

## Fallbacks

These fallbacks will be used when the system cannot determine the Surface Impact or Surface Type involved in an impact. Examples where this can happen include when a projectile does not have its Surface Impact set, or an impacted object does not have either a Surface Identifier or a texture fallback.

Fallbacks are very powerful as they allow you to provide rough behavior for all of the surfaces in your game with all of the effect combinations accounted for with an absolute minimum amount of work.

## Fallback Surface Impact

The Surface Impact will be used when the Surface Impact is unknown (such as if a footstep or a projectile has no Surface Impact). This fallback will apply to all cases when the Surface Impact is unknown, unless a Surface Type has a Surface Effect assigned for its fallback.

## Fallback Surface Type

The Surface Type will be used when the Surface Type is unknown (such as when an object has no Surface Identifier and there are no object or terrain Surface Types). Your fallback surface can be one of the existing surfaces, but it is important that all of the Surface Impacts are hooked up to a Surface Effect, including the Fallback Surface Impact.

## Fallback Allow Decals

When Allow Decals is false (recommended) the default fallbacks will not be able to spawn decals, even if the Surface Effect has them. This will prevent fallback decals that don't fit well with the target surface.

## Stretched Decals

When a decal is added to an object it can be set to the child of that object. Due to Unity engine internals, chiliding a decal to an object with a non-uniform scale (meaning the x, y, and z scale are not identical) will result in decals that are strangely stretched/warped. Because of this objects that are non-uniform will only receive decals if the Allow Non Uniform Decals toggle is true within the Decal Manager.

The warping of the decals may be very slight and acceptable if the object has only slight scale deviations on one or more axes. If the scale deviations are considerable then the decals will start to look very deformed when they are parented. There are three workarounds for this:

1. The target object should be marked as static if it will never move or be destroyed at runtime. This will allow all decals to appear normal on the surface even though they aren't actually parented to the object.
2. The Allow Non Uniform Decals option on the Decal Manager can be deselected to prevent decals from every being parented to a non-uniform object.
3. A Surface Identifier can be added to control if decals should spawn on a per-object basis. The Allow Decals option can be deselected within the Surface Identifier which will prevent decals from being parented to that object.

# Surface Impacts

The Surface Impact Scriptable Object is used to identify different collision types for surface effect purposes. It declares (by filename) a particular type of collision that will be used to trigger effects when sent to a collider that has a Surface Type.

Example Surface Impacts are bullet hit, melee impact, fall impact, footstep, etc. When a bullet hits a rock, the Surface Impact is what makes the Surface Manager spawn sparks,

dust, and a ricochet sound instead of footstep effects. This is achieved by binding the Surface Impact object to a Surface Effect object by an encompassing Surface Type object.

The Surface Impact script doesn't actually do anything. It is just used as an identifier object for a certain type of impact. Of course, at runtime the Surface Impact object plays a very important within the larger Surface System.

## Creating a new Surface Impact

A new Surface Impact can be created from the Assets -> Create -> Ultimate Character Controller -> Surface Impact menu option. This menu is also available when you right click within the project view. However, you may find it easier to just duplicate an existing effect.

After you have created a new Surface Impact it must be assigned in at least two places:

1. The object that is actually using the Surface Impact, such as the bullet, footstep, or ability.
2. A Surface Type object which binds a Surface Impact to a Surface Effect. Alternatively the Surface Impact can be set as the default Fallback on the Surface Manager component.

## Fallback Surface Impact

You can set a scene-wide fallback Surface Impact within the Surface Manager - Fallbacks foldout. This can be used for cases where the Surface Impact is unknown (such as if a Surface Impact wasn't set on a bullet component).

# Surface Types

The Surface Type Scriptable Object is the main surface concept. Surfaces will trigger effects in response to projectiles, abilities, footsteps, and potentially many other things. Every surface has a list of Impact Effects that bind a Surface Impact to a particular Surface Effect. There should be an element representing each type of collision effect response that you want to cover.

## Impact Effects

The Impact Effects array lists all of the Surface Impacts that are used by this Surface Type. Any Surface Impact which should trigger a Surface Effect should be listed here. Elements can be added or removed to the array by increasing or decreasing the size value. Remember that the last element(s) will be removed if you decrease the size value.

## Surface Impact and Surface Effect

Whenever an impact occurs the Surface Manager looks for a particular Surface Type and Surface Impact which match. If the Surface Manager cannot find a Surface Type and Surface Impact that match then the Surface Manager will try to come up with a good fallback. The resulting Surface Effect is then triggered.

## **Creating a new Surface Type**

A new Surface Impact can be created from the Assets -> Create -> Ultimate Character Controller -> Surface Type menu option. This menu is also available when you right click within the project view. However, you may find it easier to just duplicate an existing type.

## **Fallback Surface Type**

When the Impact Type is unknown a fallback Surface Type can be used. This is set within the Surface Manager under the Fallback foldout.

## **Recommended Usage**

It is recommended that most of your scene prefabs have a Surface Identifier component with the Surface Type assigned to them. This is the most performant way for the Ultimate Character Controller to look up a surface. When a Surface Effect needs to be played the Ultimate Character Controller will have the Surface Type instantly if it has been added to a Surface Identifier. This prevents the Surface Manager from having to search for a Surface Effect that is the best match.

# **Surface Effects**

The Surface Effect Scriptable Object is the recipe for a bunch of different effects to be played in response to a certain type of collision. It might trigger when a bullet hits a wall, when a character places a footprint, or when the player falls violently to the ground.

Each Surface Effect contains the object references and logic for a simple one-shot effect. When triggered it can play a random sound from a list, spawn a set of prefabs (according to random probabilities), or select a randomly scaled decal.

It is important to understand that a Surface Effect object is never spawned in the scene. Instead, it is triggered and responds by playing sounds and spawning other prefabs. The Surface Effect object doesn't exist at the scene level, but at the project level.

## **Creating a new Surface Effect**

A new Surface Effect can be created from the Assets -> Create -> Ultimate Character Controller -> Surface Effect menu option. This menu is also available when you right click within the project view. However, you may find it easier to just duplicate an existing type.

## **Inspected Fields**

### **Spawned Objects**

A list of objects that can be spawned when the Surface Effect is triggered. Each element will be spawned based on a probability value.

### **Object**

Represents the prefab that is attempted to be spawned. This is perfect for particle effects

and rubble prefabs!

### Probability

A value of 1 means the prefab will always spawn when the Surface Effect is triggered. A value of 0 means that it will never spawn.

### Random Spin

A boolean indicating if a random spin should be applied to the spawned object. If enabled a random rotation will be applied relative to the object's normal direction.

### Decals

Lists the decals that can be spawned as a result of the effect. Only one decal will be randomly chosen out of the list.

### Prefabs

A list of decal prefabs that can be spawned. All decal prefabs are assumed to have a MeshFilter and MeshRenderer on their main Transform with a class 2-triangle quad(normal aligned with the Z-vector). The overlap detection features of the Decal Manager require this type of decal.

### Min & Max Scale

Multiplies the local XY scale by a random value inside this range when a decal is spawned. The local Z scale is not affected.

### Allowed Decal Edge Overlap

With zero overlap any decals that overlap the corner of a wall in the slightest bit will be removed. When the overlap is at its max value (0.5) the decals are allowed to overlap corners all of the way to their center. This feature only works if the scene has a Decal Manager component with Placement Tests enabled.

### Audio

When the effect triggers one Audio Clip from the list will be chosen and played.

### Audio Clips

A list of Audio Clips that can be played by the Audio Manager when the effect is triggered.

### Min & Max Volume

The sound will be played as-is if both the min and max volume values are equal to 1. Any other values will cause the volume of the clip to be modified by a random value inside the range.

### **Min & Max Pitch**

The sound will be played as-is if both the min and max pitch values are equal to 1. Any other values will cause the pitch of the clip to be modified by a random value inside the range.

### **One Clip Per Frame**

Enable this boolean to avoid excessive sound volume on impact with effects that trigger many times at once (such as shotgun pellets).

### **Random Clip Selection**

Should a random clip be selected out of the Audio Clip list? If false then the clips will be selected sequentially.

### **State**

When an effect is triggered it can optionally enable the state on the object that was hit.

#### **State Name**

The name of the state that should be triggered when the object is hit. This is useful if for example you'd like to make an enemy move in slow motion when being shot by a gun which affects the time scale of objects.

#### **State Disable Timer**

The number of seconds until the state name is disabled. A value of -1 will require the state to be manually disabled at a later point in time.

## **Surface Identifiers**

The Surface Identifier component is used to determine what effect should be triggered from the surface of an object when hit by a Surface Impact. It is recommended that you rely on Surface Identifiers first and foremost when making your game. You should only resort to the more advanced Surface Manager features for special cases like terrains and multiple material objects. This is for both performance reasons and if you make it a habit to place Surface Identifiers on all of your scene prefabs they will "just work" as you'd expect.

### **Surface Type**

The Surface Type determines what the object's surface is made of, and what Surface Effect will trigger when it gets hit by a Surface Effect.

### **Allow Decals**

If enabled then bullet holes and footprints can stick to the surface of the object. It will override any Surface Manager settings for this specific object. This is useful for objects that have colliders that do not perfectly follow the shape of the object and thus would otherwise get bad decal placement.

# Decal Manager

The Decal Manager is responsible for managing the spawned decals. The number of decals can be capped at a limit to prevent too many from being spawned. The decals can then slowly be faded (weathered) for a smooth transition rather than the decal just popping out of existence.

## Decal Limit

The maximum number of decals that can spawn. As new decals appear the older ones are weathered and eventually removed.

## Weathered Decal Limit

Specifies how many of the oldest decals will be gradually faded away a tiny bit each time a new decal is spawned. The oldest of the decals will be almost invisible before it is removed.

Weathering is based on the number of decals in the scene and the order in which the decals were created rather than on game time. Every time a new decal is placed the oldest of the decals will be weathered. If no decals are placed and a decal is 50% weathered then it will stay in the world indefinitely.

## Remove Fadeout Speed

The speed that the decals should fadeout after they have been removed from being weathered.

# Character Foot Effects

The Character Foot Effects component will determine when a new footstep has occurred. The footstep is detected based on the footstep mode and when a footstep occurs the Surface Effect based on the Surface Impact will play.

## Surface Impact

The Surface Impact used to identify the footstep.

## Feet

An array of Transforms specifying the location of each of the character's feet. The Character Manager will automatically fill in this array for humanoid characters

## Footstep Mode

Specifies how the footsteps are detected:

- *Body Step*: Footsteps are determined by the vertical height of the character's foot. When the foot's Transform position is below the vertical Foot Offset value relative to the character's position then a footstep has occurred. This is the most realistic mode

however note that results are very dependent on the quality of your animations and tweaking a good foot offset to go along with it.

- *Trigger*: Footsteps will be placed when the foot Transform enters a trigger. This mode requires the FootstepTrigger component to be added to each foot Transform. When Unity calls the `OnTriggerEnter` method the FootstepTrigger component will notify the CharacterFootstep component that a footprint has occurred. Note that the feet array is not used when the mode is set to the trigger type.
- *Fixed Interval*: Footsteps will be placed according to a timer, and the horizontal foot position will be used to place footprint effects at ground level. Each foot will trigger effects whether touching the ground or not (this is hardly noticeable since feet are usually fairly close to the ground and moving fast). In this mode you'll want to create separate states with custom intervals to differentiate between walking, running, or crouching.
- *Camera Bob*: Footsteps will be placed when the camera bob dips (reaches the lowest point and then ascends again). This mode is most useful for first person mode and requires the camera to have bob motion enabled.

### **Min Velocity**

The minimum velocity (squared) that the character must have in order for a footprint to be detected.

### **Move Direction Frame Count**

If using the BodyStep mode, specifies the number of frames that the foot must be moving in order for it to be checked if it is down.

### **Foot Offset**

Specifies an offset for when a raycast is cast to determine if the character's foot is considered down.

### **Interval**

If using the FixedInterval mode, specifies how often the footsteps occur (in seconds) when the character is moving.

### **Min Bob Interval**

If using the CameraBob mode, specifies the minimum time that must elapse before another footprint occurs.

## **Advanced Surface System Topics**

### **Surface Manager**

#### **Fallback Logic**

- If there is no Surface Impact but there is a Surface Type then the fallback Surface

Impact will be used with the detected surface.

- If there is no Surface Type but there is a Surface Impact then the detected Surface Impact will be used with the fallback Surface Type.
- If there is no Surface Impact or Surface Type then the fallback Surface Impact will be used with the fallback Surface Type.

## Limitations

- For performance reasons the surface system relies heavily on caching. Because of this you cannot:
  - Manipulate the Surface Manager at runtime.
  - Change surface settings at runtime.
  - Manipulate hierarchies of any target object at runtime.
- Due to Unity engine internals, childing a decal to an object with a non-uniform scale (the x, y, and z scales are not identical) will result in decals that are strangely stretched/warped on such objects.
- UV regions on atlas textures do not work with Unity's default primitives (cube, sphere, capsule, etc). It will only work on objects with a Mesh Filter and Mesh Renderer.
- UV regions on atlas textures do not work on static objects.
- Surface detection will not work if the collider is not static and the renderer is static.
- Surface detection will not work with multiple materials in the following scenarios:
  - If the Renderer is static. This is because Unity merges the meshes of all of the static objects into a single mesh. Unfortunately, you must either make the renderer non-static or split the object into separate, single material static objects and assign the Surface Identifier to each.
  - If the GameObject has a Mesh Collider. This is because Unity creates separate meshes for each material under-the-hood when rendering. The Renderer and Collider must share the same logical mesh for Surface Effects to work with Mesh Colliders.

## Troubleshooting

### A level geometry exists whose material / shader does not have a texture. How should it be associated with a surface?

Add the Surface Identifier component to the object.

### My fallback Surface Impact or Surface Type doesn't work.

Ensure there is not a Surface Identifier on the object. This will override all fallback settings.

### A Surface Identifier is on the object but it doesn't work.

1. Verify that you have assigned a Surface Type.
2. If you want to receive decals, verify that Allow Decals is enabled on the Surface Identifier.
3. Verify that there is only one Surface Identifier on the object.
4. Verify that the object has a uniform scale.

### Surface Effects don't work on static, multi-material / atlas map objects

This is a very unfortunate limitation due to Unity architecture. When you mark an object as rendering static then Unity will merge the meshes together in a way that prevents the surface system from deriving material info / texture coordinates for effect spawning. At the time of writing there are three known solutions to the problem:

- Make the object non-static. This may be an acceptable solution if it's not for all level geometry but just a few objects.
- Place invisible, static colliders above floors or walls with a certain texture and assign a Surface Manager to the invisible object.
- Don't use multi-material objects, instead break the objects into separate, static objects.

## Spawn System

### Respawner

The Respawner component will respawn the object when the object dies or is destroyed. The object can either spawn at the current location, the starting location, or can use the spawn point system to determine where to spawn.

#### Inspected Fields

##### Positioning Mode

Specifies the location that the object should spawn:

- *None*: The object will not change locations when spawning.
- *Start Location*: The object will spawn in the same location as the object started in.
- *Spawn Point*: The object will use the Spawn Point system to determine where to spawn.

##### Grouping

The grouping index to use when spawning to a spawn point. The grouping will be ignored if the value is -1. Groupings can be used in the case of having multiple teams. As an example, consider a team deathmatch game where the red team has a grouping value of 0 and the blue team has a value of 1. When a player on the red team goes to spawn only the Spawn Points with a grouping value of 0 will be used to decide if the player can spawn in that location.

##### Min Respawn Time

The minimum amount of time before the object respawns after death or after being disabled.

##### Max Respawn Time

The maximum amount of time before the object respawns after death or being disabled.

## Schedule Respawn On Death

Should a respawn be scheduled when the object dies?

## Schedule Respawn On Disable

Should a respawn be scheduled when the component is disabled?

## Respawn Audio Clip Set

A set of AudioClips which can be played when the object is respawned.

## Events

When the object respawn occurs the “OnRespawn” event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. The following example will subscribe to this event from a new component:

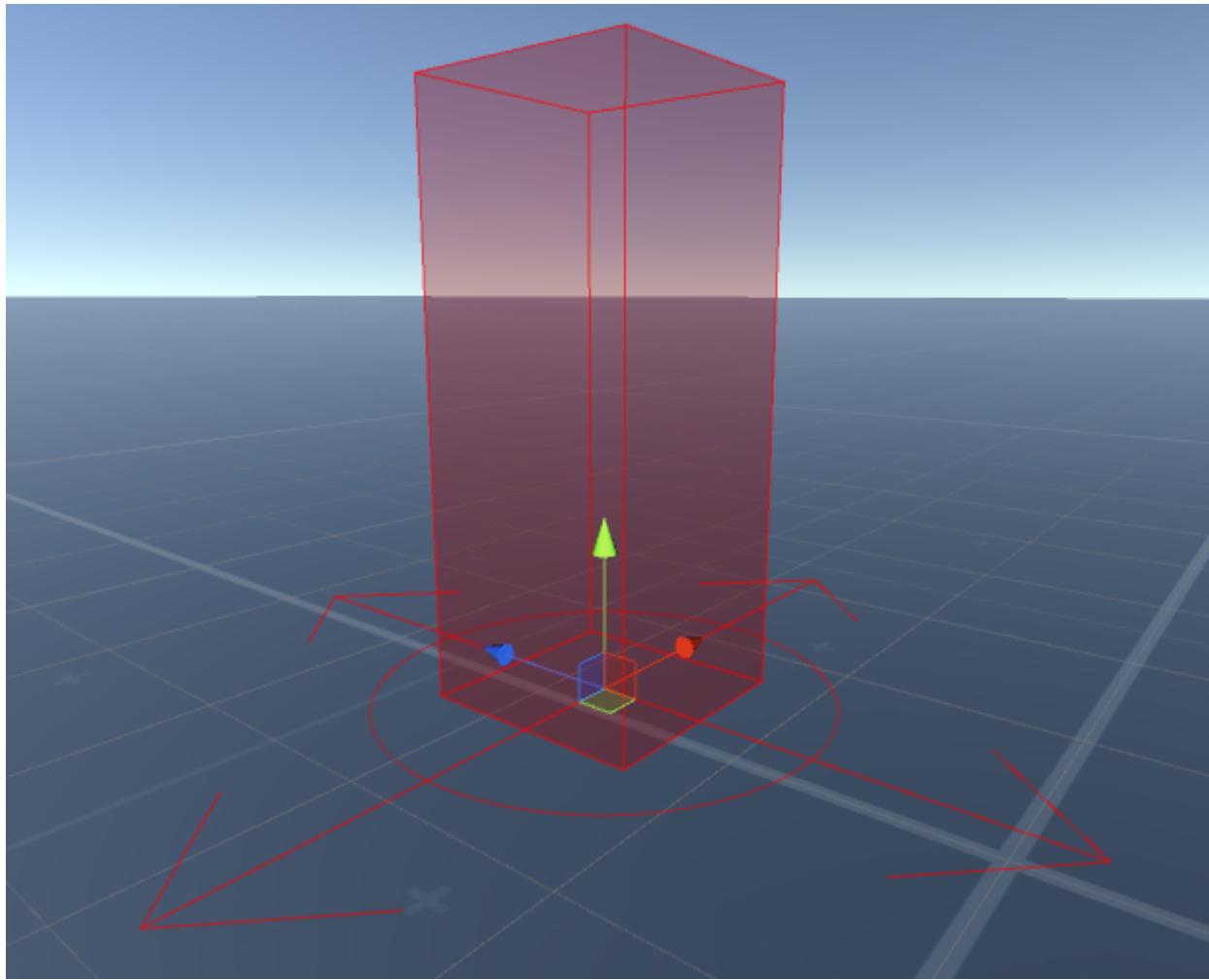
```
using UnityEngine;
using Opsive.Shared.Events;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Initialize the default values.
    /// </summary>
    public void Awake()
    {
        EventHandler.RegisterEvent(gameObject, "OnRespawn",
OnRespawn);
    }

    /// <summary>
    /// The object has respawned.
    /// </summary>
    private void OnRespawn()
    {
        Debug.Log(name + " Respawned.");
    }

    /// <summary>
    /// The GameObject has been destroyed.
    /// </summary>
    public void OnDestroy()
    {
        EventHandler.UnregisterEvent(gameObject, "OnRespawn",
OnRespawn);
    }
}
```

# **Spawn Points**



The Spawn Point component is used to determine the location that the object can spawn. A shape can be defined by the component and this shape is used to determine where within the bounding area the object should spawn.

### Grouping

An index value used to group multiple sets of spawn points. The grouping will be ignored if the value is -1. Groupings can be used in the case of having multiple teams. As an example, consider a team deathmatch game where the red team has a grouping value of 0 and the blue team has a value of 1. When a player on the red team goes to spawn only the Spawn Points with a grouping value of 0 will be used to decide if the player can spawn in that location.

### Shape

Specifies the shape of the spawn point (a sphere or a box).

- *Point*: The spawn point will be determined at the transform position.
- *Sphere*: The spawn point will be determined within a random sphere.
- *Box*: The spawn point will be determined within a box.

### Size

The size of the shape. If the shape is a sphere this size will be the radius, and for a box it is the local x and z extents. The height of the box is determined by the Ground Snap Height.

### **Ground Snap Height**

Specifies the height of the ground check. When the object is spawned a ground check will be used to ensure the character spawns on the ground and not in the air.

### **Random Direction**

Should the character spawn with a random y direction?

### **Check For Obstruction**

Should a check be performed to determine if there are any objects obstructing the spawn point?

### **Obstruction Layers**

Specifies the layers which can obstruct the spawn point. These layers are used for both for ground snapping and the obstruction check.

### **Placement Attempts**

If checking for obstruction, specifies how many times the location should be determined before it is decided that there are no valid spawn locations.

### **Gizmo Color**

The color to draw the editor gizmo in (editor only).

## **Audio**

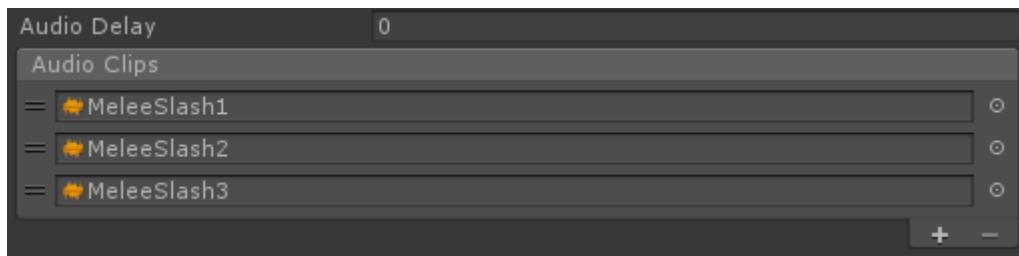
The Audio Manager is a singleton component which manages audio playback. The advantages of having this component rather than playing an audio clip directly on an Audio Source are:

- Allows multiple clips to be played at the same time without one clip stopping another clip from playing.
- Allows clips to continue playing even if the object is disabled.
- Allows for easier integration with other assets.

Your workflow doesn't need to change when you are using the Audio Manager component. If you add an Audio Source to your GameObject the Audio Manager will use the properties from that Audio Source automatically. If a new Audio Source needs to be added to the GameObject it will also use the correct properties.

### **Audio Clip Set**

The Audio Clip Set makes it easy to randomly play a clip out of a set of multiple audio clips:



New clips can be added by selecting the plus button on the bottom right. Once a row is selected it can be removed by selecting the minus button.

When a clip needs to be played the clip will be randomly selected out of this list. A delay can be specified which will prevent the audio from playing for the specified number of seconds.

## API

The AudioManager exists in the Opsive.UltimateCharacterController.Audio namespace and the most common methods that you will use are:

```
// <summary>
// Registers the AudioSources on the Game0bject so they can be played.
// The Register method should be called before any clip
// is played.
// </summary>
Register(GameObject game0bject)

// <summary>
// In some cases you may want the AudioManager to always play on the
// same Audio Source and don't want any other clips to play
// on that Audio Source. Setting the reserve count will reserve the
// number of Audio Sources so they can only be played when
// explicitly called. The reserve index is used by the Equip/Unequip
// clips for the items because both equip and unequip will
// never need to play at the same time.
// </summary>
SetReserveCount(GameObject game0bject, int count)

// <summary>
// Plays the clip on the Game0bject. The AudioManager will find the
// next available Audio Source on the specified Game0bject.
// An optional reserved index can be specified if the clip should play
// on a specific Audio Source.
// </summary>
Play(GameObject game0bject, AudioClip clip, int reservedIndex
(optional))

// <summary>
// Plays the clip on the Game0bject with the specified delay. The
// AudioManager will find the next available Audio Source
// on the specified Game0bject. An optional reserved index can be
// specified if the clip should play on a specific Audio Source.
```

```

// </summary>
PlayDelayed(GameObject gameObject, AudioClip clip, int reservedIndex
(optional))

// <summary>
// Stops playing the clip on the Audio Source at the reserved index.
// </summary>
Stop(GameObject gameObject, int reservedIndex)

```

## State System

The State System is an incredibly powerful system built into the Ultimate Character Controller. Any game will have the character switching between component values as they play the game. For example, lets take a simple case of zooming the camera by changing the field of view. When the Aim ability is active it could manually set the field of view, but what if you want a different zoom for the assault rifle versus the sword? Normally you'd have to manually add a case for the assault rifle (or sword) and add that to the Aim ability so when the ability is activated it correctly switches field of view. As your game grows this would be a very tedious process. The State System simplifies this by moving much of the work into the editor and solving runtime blending automagically.

The backbone of the State System is the Presets System.

### Creating a New State

States can be created by selecting a component which has the “States” foldout with the reorderable list. Clicking on the plus button on the bottom right of the state list will allow you to select a state with a new preset or an already created preset. Selecting one of these options will create the new state. After the state is created it must be named. Names are case sensitive and must be unique within the state list.

Name	Preset	Blocked By	Persist	Activate
= MyState	MyPreset	Nothing		
= Default	None (Per	Nothing		

### State Order

The State System works similarly to the layers of a paint program. The state (layer) at the bottom represents the current inspector values of a component. The state above it represents a preset that will override all - or just some - of the underlying state value. The remaining presets in the list will be recombined when a state is disabled or enabled.

States can be reordered by dragging the two line icon ( ) on the left. Any currently active states at the top will override any active states beneath them.

## Default State

The bottom-most state is called the Default state. It represents the standard component values and cannot be edited, removed, or moved.

## Active States

States that are active will be bold and have “(Active)” appended to the end of the state name. States cannot be edited while the game is running so their fields are disabled in the inspector.

Name	Preset	Blocked By	Persist	Activate
SwordShield	EnabledItemSet	Nothing		
<b>Shield (Active)</b>	EnabledItemSet	Nothing		
DualPistols	EnabledItemSet	Nothing		
Body	EnabledItemSet	Nothing		
PrimaryFragGrenade	EnabledItemSet	Nothing		
Bow	EnabledItemSet	Nothing		
<b>Default (Active)</b>	(Preset)	Nothing		

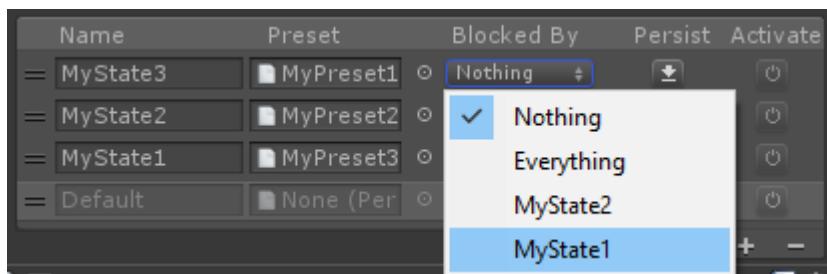
## Preset

The preset column maps a particular preset to a state. Each state (besides the bottom Default state) must have a present assigned to it.

## Blocking

Sometimes you want states to override each other entirely. For example, you may not want the character to aim down the sights while running, and so you want the Run state to override the Zoom state on the camera and weapon. Theoretically you could block out all underlying states by applying a present that contains every possible value for the component in question, however this becomes labor intensive very quickly.

State blocking should be used instead. Blocking allows a state to mute one or more other states on the component while active. The “Blocked By” dropdown will list the available states that can be blocked.



Every state that is selected will be blocked while the current state is active. States cannot block themselves, states that block them, or the Default state.

## Persist

The Persist button will update the preset with the current values of the component. This is 294

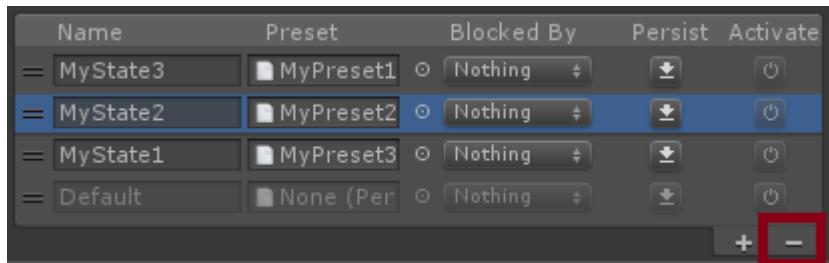
useful when you change values within the component and then would like to apply those values to a particular preset.

### Activate

The Activate button is a helpful button which will activate a particular state manually. Pressing on the Activate button when a state is active will deactivate the state.

### Removing a state

States can be removed by selecting an existing state within the list and then clicking the minus button on the bottom right of the state list.



### API

States can be activated or deactivated programmatically with StateManager.SetState:

```
StateManager.SetState(m_Character, "MyState", true);
```

The first parameter specifies the GameObject that should have the states activated or deactivated on. The second parameter specifies the state name that should be activated or deactivated. The final parameter specifies if the state should be enabled or disabled (true/false).

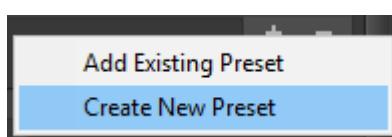
The state manager lives within the Opsive.UltimateCharacterController.StateSystem namespace.

## Presets

The Preset System allows you to take a snapshot of all of the properties in a component and save it to an asset. Presets can be loaded via script at runtime and used to quickly manipulate all of the parameters of a component as needed during gameplay.

### Create Presets

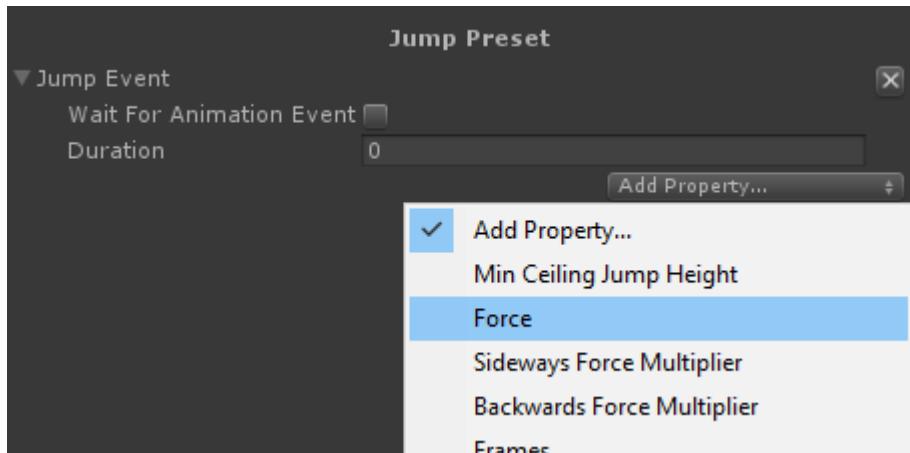
Objects which allow presets will have a reorderable list displayed at the bottom of its inspector under the “States” foldout. New presets can be created by clicking the plus button on the bottom right of the reorderable list. A menu will then appear and “Create New Preset” should be selected:



## Adding Properties

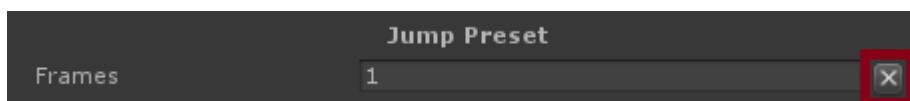
For best performance only [C# properties](#) can be added to presets. The property must have a public get and set attribute in order for it to be visible to the preset system. Presets can store any type of serializable file besides references to objects within a scene. This is a Unity restriction and occurs because project-level files (such as presets) cannot reference objects within a scene.

The property can then be selected within the “Add Property” dropdown after clicking on the preset. In this example a preset for the Jump ability is selected:



## Removing Properties

After a property is added it can be removed by clicking on the “X” button on the right to the property value:



# Programming Concepts

## Events

The EventHander adds a generic event system to the character controller. This system is preferred internally over Unity’s event system because it doesn’t allocate any garbage when executing. It is also easier to extend and events can be bound to a specific object or can be executed globally. As with any robust event system, multiple objects can be subscribed to the same event. The following method definition is used to listen for an event:

```
EventHandler.RegisterEvent(object obj, string name, Action handler)
```

This function is overloaded so events with up to five parameters can also be sent:

```
EventHandler.RegisterEvent<int>(object obj, string name, Action<int> handler)
```

Unregistering from an event is very similar to registering for an event:

```
EventHandler.UnregisterEvent<int>(object obj, string name, Action<int> handler)
```

An event can then be executed with:

```
EventHandler.ExecuteEvent(object obj, string name)
```

If the event is a global event then the first parameter (object obj) is omitted. As an example lets have an object listen for the “OnDeath” event. The entire component looks like:

```
using UnityEngine;
using Opsive.Shared.Events;

public class MyComponent : MonoBehaviour
{
    public void Awake()
    {
        // Register for the event when the component starts. The event
        registers with the local GameObject (the first parameter) so
        // the component must be added to the same GameObject as the
        object that is sending the "OnDeath" event.
        EventHandler.RegisterEvent<Vector3, Vector3,
        GameObject>(gameObject, "OnDeath", OnDeath);
    }

    /// <summary>
    /// Receives the "OnDeath" event.
    /// </summary>
    private void OnDeath(Vector3 position, Vector3 force, GameObject
attacker)
    {
        Debug.Log("The object died");
    }

    public void OnDestroy()
    {
        // Unregister from the event when the component is no longer
        interested in it. In this example the component is interested for the
        lifetime of
        // the component (Awake -> OnDestroy).
        EventHandler.UnregisterEvent<Vector3, Vector3,
        GameObject>(gameObject, "OnDeath", OnDeath);
    }
}
```

## Scheduler

The Scheduler allows for delayed execution of methods. As you are going through the code you'll see that this is used in many ways, from scheduling when a projectile should

deactivate to checking that a controller is connected. The scheduler also accepts up to three parameters if the calling method requires it. When a new method is scheduled to occur in the future a `ScheduledEventBase` object is returned. This object allows you to cancel events before they are scheduled to occur.

An event can be scheduled with:

```
Scheduler.Schedule(float delay, Action action)
```

Where `delay` specifies the number of seconds that the action should execute. If a value of -1 is supplied for the `delay` then the event will be executed every frame. This is extremely useful when you want to update a particular method without having to manually update it within the standard `MonoBehaviour` loop. A great use case of this is the Spring system and it ticking every frame.

When the event in the above example is executed it is executed within Unity's `Update` loop. If instead you'd like the event to be executed within the `FixedUpdate` loop you can schedule the event with the following:

```
Scheduler.ScheduleFixed(float delay, Action action)
```

Abilities and items are generally updated within the `FixedUpdate` loop so they mostly use the `ScheduleFixed` variant, while everything else is updated within `Update` so they use the regular `Schedule` variant.

Scheduled events can later be cancelled with:

```
Scheduler.Cancel(ScheduledEventBase scheduledEvent)
```

`ScheduledEventBase` objects can be checked to determine if they are currently scheduled with the `Active` property. A full example of using the `Scheduler` API is below.

```
// The scheduler class is in the Game namespace.  
using Opsive.UltimateCharacterController.Game;  
  
// Schedule the Explode method to occur in 1.5 seconds. The first  
// parameter of Explode will have a value of true.  
var scheduledEvent = Scheduler.Schedule(1.5f, Explode, true)  
  
// Check the active state of the scheduled event.  
Debug.Log("Is active: " + scheduledEvent.Active);  
  
// Cancel the scheduled event  
Scheduler.Cancel(scheduledEvent);
```

## Object Pool

It is a relatively expensive operation to instantiate and destroy objects. To avoid many instantiations and destructions an object pool is used which will reuse an already-created object. The `ObjectPool` component can pool both `GameObjects` and regular objects derived

from System.Object.

When a new object is instantiated the ObjectPool.Instantiate method is called instead of the GameObject.Instantiate method. An object can then be placed back in the pool with ObjectPool.Destroy.

Consider the following example where a projectile is spawned. Instead of using GameObject.Instantiate/Destroy you want to use the ObjectPool component to improve performance. The following code can be used:

```
// Include the namespace that the ObjectPool component is located in.  
using Opsive.Shared.Game;  
  
// Keep a reference to the object so it can later be destroyed.  
private GameObject m_Projectile;  
  
// Instantiate the projectile.  
m_Projectile = ObjectPool.Instantiate(projectilePrefab);  
  
// The projectile should be destroyed at a later point in time.  
ObjectPool.Destroy(m_Projectile);
```

The GenericObjectPool can also pool non-GameObjects with the Get and Return methods. The following example shows the ActiveInputEvent object being pooled (ActiveInputEvent is derived from System.Object):

```
// Include the namespace that the GenericObjectPool component is  
located in.  
using Opsive.Shared.Utility;  
  
// Keep a reference to the object so it can later be destroyed.  
private ActiveInputEvent m_PooledInputEvent;  
  
// Get a new ActiveInputEvent object.  
m_PooledInputEvent = GenericObjectPool.Get<ActiveInputEvent>();  
  
// The object can be returned back to the pool at a later point in  
time.  
GenericObjectPool.Return(m_PooledInputEvent);
```

## Artificial Intelligence (AI)

The Ultimate Character Controller is focused on being a great character controller so it does not include any built-in AI implementations. AI is an extremely large topic so instead of including an implementation that would likely be replaced past the prototype stage the controller is instead structured so it can work with existing AI implementations. [Behavior Designer](#) is a behavior tree implementation and is [cleanly integrated](#) with the Ultimate Character Controller.

## API

If you are scripting your own AI some common functions that you may need to perform are listed below. Ensure you have setup your character as an [AI agent](#) so the character will not use the camera.

### Damage

An Ultimate Character Controller character can be damaged by getting a reference to the [Health](#) component and then calling the Damage method.

```
using UnityEngine;
using Opsive.UltimateCharacterController.Traits;

public class MyAIAgent : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller
character.")]
    [SerializeField] protected GameObject m_Character;

    /// <summary>
    /// Damages the character.
    /// </summary>
    private void Start ()
    {
        var health = m_Character.GetComponent<Health>();
        if (health != null) {
            health.Damage(50); // Inflict 50 damage on the character.
        }
    }
}
```

### Use Item

An item can be used by starting the Use Item Ability:

```
using UnityEngine;
using Opsive.UltimateCharacterController.Character;
using Opsive.UltimateCharacterController.Character.Abilities.Items;

public class MyAIAgent : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller
character.")]
    [SerializeField] protected GameObject m_Character;

    /// <summary>
    /// Uses the item.
    /// </summary>
    private void Start ()
```

```

    {
        var characterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        if (characterLocomotion != null) {
            // Get the Use ability.
            var useAbility = characterLocomotion.GetAbility<Use>();
            if (useAbility != null) {
                // Try to start the use ability. If the ability is
started it will use the currently equipped item.
                characterLocomotion.TryStartAbility(useAbility);
            }
        }
    }
}

```

## Equipping

In order to equip or unequip an item an item ability should be started. In the example below the ability will equip the next item, and then it will equip a specific item index.

```

using UnityEngine;
using Opsive.UltimateCharacterController.Character;
using Opsive.UltimateCharacterController.Character.Abilities.Items;

public class MyAI : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller
character.")]
    [SerializeField] protected GameObject m_Character;

    /// <summary>
    /// Equips the item.
    /// </summary>
    private void Start ()
    {
        var characterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        if (characterLocomotion != null) {
            // Get the EquipNext ability and start it. The next item
within the ItemSetManager will be equipped.
            var equipNext =
characterLocomotion.GetAbility<EquipNext>();
            if (equipNext != null) {
                characterLocomotion.TryStartAbility(equipNext);
            }

            // Equip a specific index within the ItemSetManager with
the EquipUnequip ability.
            var equipUnequip =

```

```

characterLocomotion.GetAbility<EquipUnequip>();
    if (equipUnequip != null) {
        // Equip the ItemSet at index 2 within the
ItemSetManager.
        equipUnequip.StartEquipUnequip(2);
    }
}
}
}

```

## Move

If you'd like to move your character a certain direction you can call the SetCharacterMovementInput and SetCharacterDeltaRawRotation methods on the KinematicObjectManager. The KinematicObjectManager ensures the character moves smoothly even when the framerate is low so it should be called rather than directly moving the character.

```

using UnityEngine;
using Opsive.UltimateCharacterController.Character;
using Opsive.UltimateCharacterController.Game;

public class MyAI : MonoBehaviour
{
    [Tooltip("A reference to the Ultimate Character Controller
character.")]
    [SerializeField] private GameObject m_Character;

    private UltimateCharacterLocomotion m_CharacterLocomotion;

    /// <summary>
    /// Initializes the default values.
    /// </summary>
    private void Start()
    {
        m_CharacterLocomotion =
m_Character.GetComponent<UltimateCharacterLocomotion>();
        if (m_CharacterLocomotion == null) {
            enabled = false;
        }
    }

    /// <summary>
    /// Move the character in the forward direction with a slight
turn.
    /// </summary>
    private void Update()
    {
KinematicObjectManager.SetCharacterMovementInput(m_CharacterLocomotion
| 302

```

```

.KinematicObjectIndex, 0, 1);
KinematicObjectManager.SetCharacterDeltaYawRotation(m_CharacterLocomotion.KinematicObjectIndex, 1);
}
}

```

## Impact Callback

When a weapon collides with another object the “OnObjectImpact” event will be sent from the built-in [Event System](#). A corresponding [Unity event](#) will also be sent. This event allows you to add new functionality when the impact occurs without having to change the class at all. The following example will subscribe to this event from a new component:

```

using UnityEngine;
using Opsive.UltimateCharacterController.Events;

public class MyObject : MonoBehaviour
{
    /// <summary>
    /// Initialize the default values.
    /// </summary>
    public void Awake()
    {
        EventHandler.RegisterEvent<float, Vector3, Vector3,
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);
    }

    /// <summary>
    /// The object has been impacted with another object.
    /// </summary>
    /// <param name="amount">The amount of damage taken.</param>
    /// <param name="position">The position of the damage.</param>
    /// <param name="forceDirection">The direction that the object
took damage from.</param>
    /// <param name="attacker">The GameObject that did the
damage.</param>
    /// <param name="attackerObject">The object that did the
damage.</param>
    /// <param name="hitCollider">The Collider that was hit.</param>
    private void OnImpact(float amount, Vector3 position, Vector3
forceDirection, GameObject attacker, object attackerObject, Collider
hitCollider)
    {
        Debug.Log(name + " impacted by " + attacker + " on collider "
+ hitCollider + ".");
    }

    /// <summary>
    /// The GameObject has been destroyed.
    /// </summary>
}

```

```
/// </summary>
public void OnDestroy()
{
    EventHandler.UnregisterEvent<float, Vector3, Vector3,
GameObject, object, Collider>(gameObject, "OnObjectImpact", OnImpact);
}
}
```

## Multiplayer

Multiplayer is supported with the [PUN Multiplayer Add-On](#). An abstract networking layer has been added to the controller and this allows for a variety of networking implementations.

## Virtual Reality (VR)

VR is supported with the [VR Add-On](#). An abstract VR layer has been added to the controller and the VR add-on interfaces with that layer.

## Integrations

### Adventure Creator

The Ultimate Character Controller is integrated with [Adventure Creator](#) allowing your character to be managed by Adventure Creator. The Adventure Creator integration is created by ICEBOX Studios and the documentation can be found on [this page](#).

### A\* Pathfinding Project

The Ultimate Character Controller is integrated with the [A\\* Pathfinding Project](#) allowing your AI agent to navigate the world with the A\* Pathfinding Project's pathfinding algorithms. This integration can be downloaded from one of the pages listed below.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Setup

1. Install the integration from above.

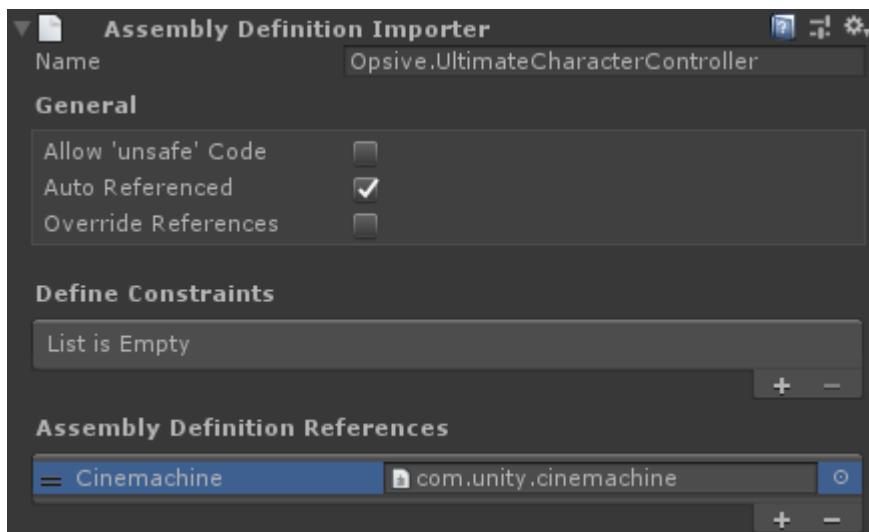
2. Create your character using the [Character Manager](#). AI Agent should be selected so the player-controlled components do not get added.
3. Add the A\* Pathfinding Project components to your character by following the [A\\* documentation](#).
4. Import the A\* integration package from the link below.
5. Add the Astar AI Agent Movement ability to your agent. This ability can be positioned anywhere within the ability list as it is a concurrent ability.
6. Set the destination of the A\* Pathfinding Project component. When the ability has a destination it will move the character according to the path created by the A\* Pathfinding Project.

## Behavior Designer - Behavior Trees for Everyone

See [this page](#) for information on the Behavior Designer integration.

## Cinemachine

The Cinemachine ViewType allows the camera to be controlled by Unity's [Cinemachine](#). Cinemachine version 2.1 or later is required and the com.unity.cinemachine Assembly Definition Reference must be added to the Opsive.UltimateCharacterController [Assembly Definition](#).



## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Setup

1. Ensure you are using Cinemachine 2.1+ and have added the Assembly Definition Reference.
2. Import the Cinemachine integration package.
3. Add the Cinemachine ViewType to the Camera Controller.
4. Create a Cinemachine camera (through the Cinemachine toolbar). Ensure the CinemachineBrain component exists on the camera.
5. If you are using a [FreeLook virtual camera](#) the CinemachineSpringExtension component should be added to that virtual camera so it will respond to spring events.

## Inspected Fields

### **Look Direction Distance**

The distance that the character should look ahead.

### **Field of View**

The field of view of the main camera.

### **Field of View Damping**

The damping time of the field of view angle when changed.

### **Position Spring**

The positional spring used for regular movement.

### **Rotation Spring**

The rotational spring used for regular movement.

### **Secondary Position Spring**

The positional spring which returns to equilibrium after a small amount of time (for recoil).

### **Secondary Rotation Spring**

The rotational spring which returns to equilibrium after a small amount of time (for recoil).

## Control Freak

The Ultimate Character Controller is integrated with [Control Freak](#) allowing you to use Control Freak for your input instead of Unity's input system. This integration can be downloaded from one of the pages listed below.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Setup

1. Install the integration from above.
2. Remove the Unity Input component from your character (if it has been added).
3. Add the Control Freak Input component to your character.
4. Setup the [Control Freak rig](#) to work with your input setup. The “CF2-Opsive-UCC-Rig” prefab is included in the integration package allowing you to quickly get started. If you are using this prefab make sure you drag it into your scene.
5. Remove any Virtual Controls created by the Setup Manager. The Virtual Controls are designed to work with the Unity Input component.

## DestroyIt

The Ultimate Character Controller is integrated with [DestroyIt](#) allowing objects to be destroyed using the DestroyIt system. The DestroyIt integration is created by ModelShark Studio and the documentation can be found on [this page](#).

## Dialogue System

The Ultimate Character Controller is integrated with the [Dialogue System](#) allowing your character to interact in dialogue and quests provided by the Dialogue System. The Dialogue System integration was created by Pixel Crushers and the documentation can be found on [this page](#). You can download the integration package from the links below.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Easy Touch

The Ultimate Character Controller is integrated with [Easy Touch](#) allowing you to use Easy Touch for your input instead of Unity’s input system. This integration can be downloaded from one of the pages listed below.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Setup

1. Install the integration from above.
2. Remove the Unity Input component from your character (if it has been added).
3. Add the Easy Touch Input component to your character.
4. Setup the Easy touch input components to work with your input setup. The “EasyTouch UCC Canvas” prefab is included in the integration package allowing you to quickly get started. If you are using this prefab make sure you drag it into your scene.
5. Remove any Virtual Controls created by the Setup Manager. The Virtual Controls are designed to work with the Unity Input component.

## Final IK

The Ultimate Character Controller is integrated with [Final IK](#) allowing you to use Final IK for IK rather than Unity’s built-in IK system. This integration can be downloaded from one of the pages listed below. After you have imported the integration package Final IK can be used by performing the following steps:

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person](#)

## Setup

1. Install the integration from above.
2. Remove the Character IK component from your character (if it has been added).
3. Add the Final IK Bridge component included in the integration package.
4. Add any of the following Final IK components:
  - Full Body Biped IK: Used for humanoids, allows for finer positioning of the character’s limbs.
  - Look At IK: Allows the character to look in the direction of the look source.
  - Aim IK: Allows the character to aim at a target.

- Grounder FullBody IK: Allows the character's feet to be positioned on the surface.
  - Interaction System: Allows the character to use IK to interact with ability objects.
5. Ensure all of the Final IK components are configured correctly. The [Final IK documentation](#) is a good reference for how to setup the Final IK components.
  6. If the AimIK component has been added the character the Animated Aim Direction Vector3 may need to be adjusted. This value represents the the direction of the animated weapon aiming in character space.

## Interaction System

Abilities can work with the IK system in order to position limbs in the correct locations. An example of this is when the Interact ability positions the hand so it always presses the button. In order for this to work with Final IK you must have the following setup:

- The Final IK Interaction System component is added to the character.
- The object that the character is interacting with has the Ability IK Target and Final IK [Interaction Object](#) components added to it. In the Interact example the button GameObject should have the Interaction Component added.

## InControl

The Ultimate Character Controller is integrated with [InControl](#) allowing you to use InControl for your input instead of Unity's input system. This integration can be downloaded from one of the pages listed below.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Setup

1. Install the integration from above.
2. Remove the Unity Input component from your character (if it has been added).
3. Add the InControl Input to your character.
4. Setup the [InControl input bindings](#) to work with your input setup. These input bindings must implement the IBindings interface included with the integration. A sample set of bindings is also included in the package.
5. Specify the created binding class name within the InControl Input component added in step 2. If you are using the sample bindings the value "Opsive.UltimateCharacterController.Integrations.InControl.SampleBindings" should be specified for the *Bindings Type* field.
6. Add the InControl Manager to your scene.

7. Remove any Virtual Controls created by the Setup Manager. The Virtual Controls are designed to work with the Unity Input component.

## Love/Hate

The Ultimate Character Controller is integrated with [Love/Hate](#) allowing your character to report combat actions as Love/Hate deeds that faction members can witness. The Love/Hate integration was created by Pixel Crushers and the documentation can be found within the README of the integration package. You can download the integration package from the links below.

### Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Playmaker

The Ultimate Character Controller is integrated with [Playmaker](#) allowing you to use Playmaker to control your character. An individual action has not been created for each field/property because the [state system](#) can be used instead. New states can be set with the Set State action. The Playmaker actions can be used using the standard Playmaker workflow and no extra steps are necessary.

If you'd like to see any new actions created please request those actions on the [forum](#).

### Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Quest Machine

The [Quest Machine](#) integration allows your character to go on quests while being managed by Quest Machine. After downloading the integration package the documentation can be found within the Quest Machine/Third Party Support/Opsive UCC Support folder.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Rewired

The Ultimate Character Controller is integrated with [Rewired](#) allowing you to use Rewired for your input instead of Unity's input system. This integration can be downloaded from one of the pages listed below.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

## Setup

1. Install the integration from above.
2. Remove the Unity Input component from your character (if it has been added).
3. Add the Rewired Input component to your character.
4. If you are using Rewired [touch controls](#) the *Enable Touch Controls* field on the Rewired Input component should be enabled.
5. Setup the [Rewired Input Manager](#) to work with your input setup. The "Rewired UCC Input Manager" prefab is included in the integration package allowing you to quickly get started. If you are using this prefab make sure you drag it into your scene.
6. Remove any Virtual Controls created by the Setup Manager. The Virtual Controls are designed to work with the Unity Input component.

## UMA

The Ultimate Character Controller is integrated with [UMA](#) allowing your character to be created dynamically while being controlled by the Ultimate Character Controller. Version 2.8 or later of UMA should be used for this integration.

There are two different options for using the Ultimate Character Controller with UMA. The first option uses UMA's Bone Builder tool which creates all of the bones necessary at edit time so you can then use the [Character Manager](#) like any other character. The second

method adds all of the Ultimate Character Controller components after UMA has created the character. This option is more advanced and is recommended for those with programming experience as you will likely need to do some scripting in order to get the most out of it.

## Downloads

[Ultimate Character Controller](#)

[First Person Controller](#)

[Third Person Controller](#)

[UFPS: Ultimate First Person Shooter](#)

[UTPS: Ultimate Third Person Shooter](#)

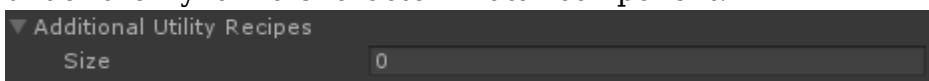
[UFPM: Ultimate First Person Melee](#)

[UFPM: Ultimate Third Person Melee](#)

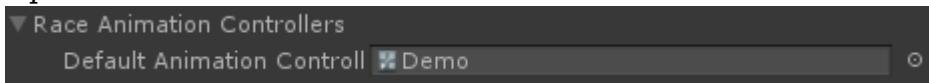
## Bone Builder

UMA's Bone Builder tool creates all of the bones necessary for your character at edit time which allows you to use the Ultimate Character Controller editors like normal. To the Ultimate Character Controller it doesn't even know that UMA was used and no integration components are required.

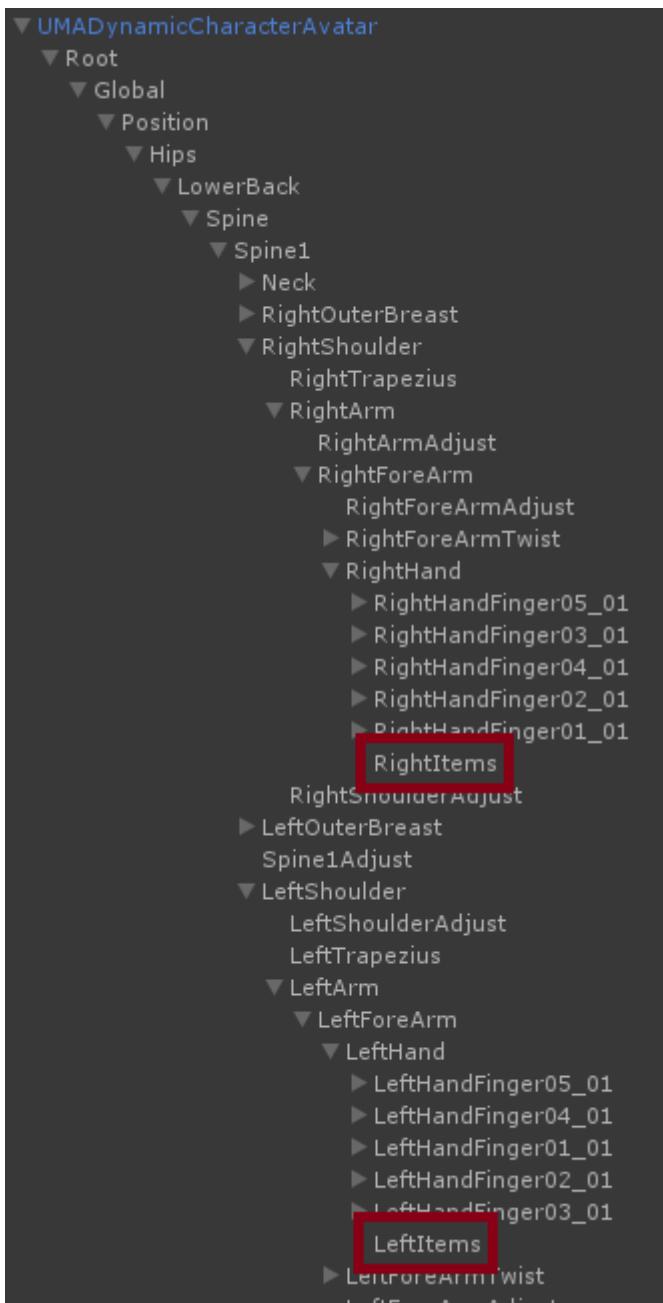
1. Download the UMA integration.
2. Drag your UMA Dynamic Character Avatar prefab into your scene.
3. Open the UMA Bone Builder under the UMA Toolbar.
4. Select the UMA Dynamic Character Avatar in your scene as the *UMA GameObject*.
5. Select the Generate Bones button. UMA Should create all of the bones required for the character.
6. Build your character like normal through the Character and Item Managers.
7. Select the UMADynamicCharacterAvatar GameObject and inspect the Dynamic Character Avatar GameObject. Ensure the *Additional Utility Recipes* array is empty under the Dynamic Character Avatar component.



8. Set the *Default Animation Controller* under the *Race Animation Controllers* dropdown to the animator controller that has been added to your character. It will be the same Animator Controller used by the Character Manager, such as Demo or ThirdPersonControllerDemo. This Animator Controller is located within the Opsive/UltimateCharacterController/Demo/Animator/Characters folder.



9. UMA doesn't like transforms that have the same name. You'll want to rename the "Items" GameObject to a unique value such as "RightItems" or "LeftItems" to ensure UMA doesn't throw any errors.

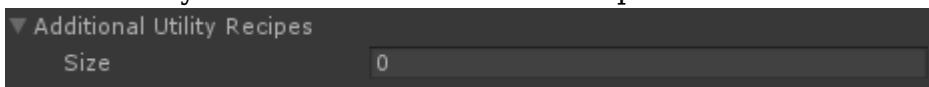


10. Your UMA character is now ready to be used.

## Runtime Creation

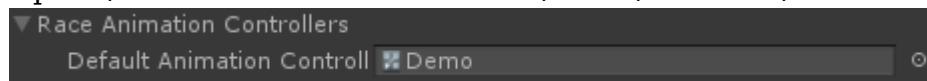
The UMA Character Builder component can add the Ultimate Character Controller components after UMA has created the character at runtime. This method does not require the bones to be added ahead of time using Bone Builder. This option is meant for more advanced uses that have experience with both UMA and the Ultimate Character Controller as there will likely be some scripting required to customize your character.

1. Download the UMA integration.
2. Drag your UMA Dynamic Character Avatar prefab into your scene.
3. Select the UMADynamicCharacterAvatar GameObject and inspect the Dynamic Character Avatar GameObject. Ensure the *Additional Utility Recipes* array is empty under the Dynamic Character Avatar component.

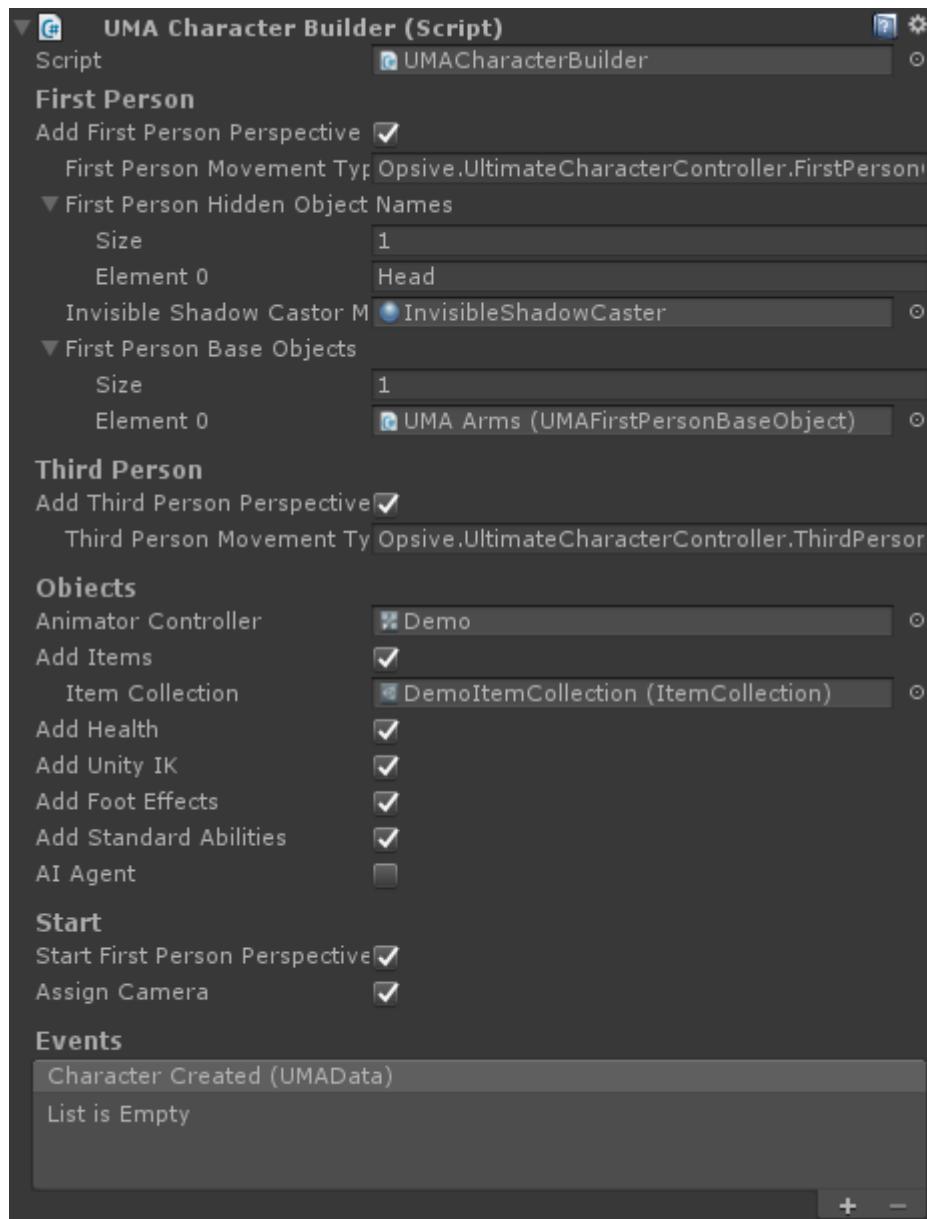


4. Set the *Default Animation Controller* under the *Race Animation Controllers* dropdown to the animator controller that has been added to your character. It will be the same.

Animator Controller used by the Character Manager, such as Demo or ThirdPersonControllerDemo. This Animator Controller is located within the Opsive/UltimateCharacterController/Demo/Animator/Characters folder.



5. Add the UMA Character Builder component to your UMA Dynamic Character Avatar. This component will add all of the Ultimate Character Controller objects and execute after UMA has created the character. If you'd like to execute any code after the character has been created you can do so by adding your code to the *Character Created* event.



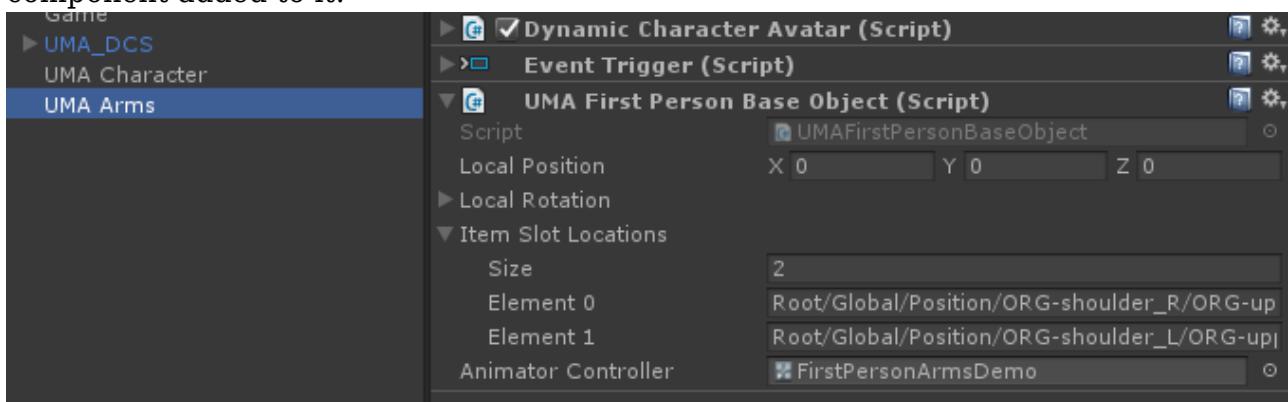
The options within the UMA Character Builder are similar to the options within the [Character Manager](#). There are a few key differences:

- *First/Third Person Movement Type* is specified by class name. For example, “Opsive.UltimateCharacterController.FirstPersonController.Character.MovementTypes.Combat” points to the Combat Movement Type for the first person perspective.
- *First Person Hidden Object Names* uses the transform name to determine which objects should be hidden while in a first person view. In the screenshot the Head GameObject will be hidden. The names are relative to the character’s base

Transform.

- *First Person Base Objects* optionally specify any First Person Base Objects that should be dynamically generated by UMA and added to the character. See step 6 for more details.
- *Assign Camera* specifies if the camera should be assigned after the character has been created.

6. Optionally add the UMA Ability Builder or UMA Item Pickup components to the same GameObject as the UMA Character Builder was added to. These components will likely need to be expanded to completely customize your character and will require scripting experience.
7. If you'd like to use a UMA created character for your first person arms you can do so by specifying the UMA character that you'd like to use within the *First Person Base Objects* array of the UMA Character Builder. This requires another UMA Dynamic Character Avatar to be placed in the scene with the UMA First Person Base Object component added to it:



This UMA character can be a generic object so you can use a generic set of arms with the main character model. A great video which shows how to do this is located [here](#). The UMA First Person Base Object component contains the following parameters:

- *Local Position/Rotation* specifies the local position and rotation that the UMA First Person Base Object should spawn underneath the character's First Person Base Objects Transform.
- *Item Slot Locations* specifies the child Transform path that Item Slot component should be added to.
- *Animator Controller* refers to the Animator Controller that should be added to the GameObject.

8. When Unity enters play mode the character will now first be created by UMA and then able to be controlled by the Ultimate Character Controller.

## Videos