IMPORTANT

The Sword and Shield Motion pack requires the following:

Motion Controller v2.8 or higher

Mixamo's free Pro Sword and Shield Pack (using Y Bot)

Importing and running without these assets will generate errors!

SWORD &SHIELD MOTION PACK

10/05/2018

Demo Quick Start

This quick start is simply to get the demo up and running in a self-contained project.

1. Start a new Unity 2017 or higher Project.

2. Download and import the Motion Controller asset.

3. Download and import the Sword and Shield Motion Pack asset.

4. Download Mixamo's Pro Sword and Shield Pack using "Y Bot" (see this flow).

5. Unzip Pro Sword and Shield animations to project's Mixamo folder.
...\Assets\ootii\Assets\MotionControllerPacks\SwordShield\Content\Animations\Mixamo

6. Unzip AnimationMeta.zip from pack's "Extras" folder to project's Mixamo folder (see steps).
...\Assets\ootii\Assts\MotionControllerPacks\SwordShield\Content\Animations\Mixamo

7. Let Unity import the animations and meta data. Then, close and re-open Unity.

8. Open the "demo_SSMP" demo scene.
...\Assets\ootii_Demos\MotionControllerPacks\SwordShield\Scenes\

9. Setup input entries on the YBotPlayer and the Packs tab.

DEFAULT CONTROLS

WASD	= Move
Mouse	= Rotate
LShift	= Run
1	= Equip the sword
LMB	= Attack
Alt	= Block
t	= Lock/unlock target

All Packs Demo

You will need to check Create Mecanim on the Packs view. The states are NOT created automatically like some of the other demos.

Click on "Enemy" and scroll to the bottom of the inspector. You can enable features based on the packs you own.

10. Press play.

SWORD &SHIELD MOTION PACK

10/05/2018

Custom Quick Start

This quick start assumes you've worked with the Motion Controller and that you have a Motion Controller enabled character in your scene already.

1. Open your MC enabled project and scene.

2. Download and import the Sword and Shield Motion Pack asset.

3. Download Mixamo's Pro Sword and Shield Pack using "Y Bot" (see this flow).

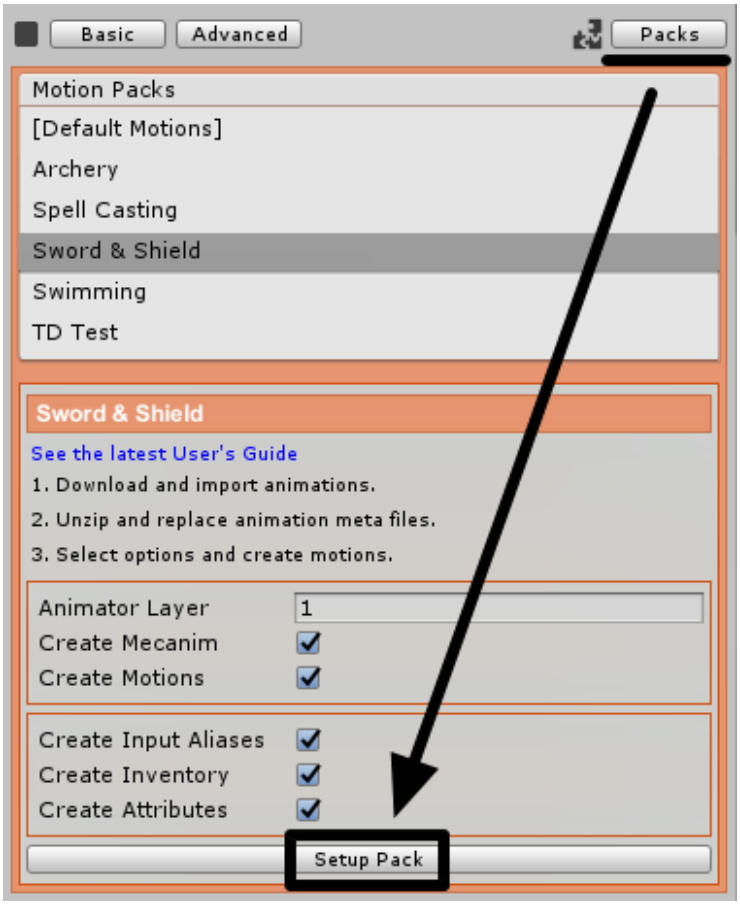
4. Unzip Pro Sword and Shield animations to project's Mixamo folder.
...\Assets\ootii\Assets\MotionControllerPacks\SwordShield\Content\Animations\Mixamo

5. Unzip AnimationMeta.zip from pack's "Extras" folder to project's Mixamo folder (see steps).
...\Assets\ootii\Assets\MotionControllerPacks\SwordShield\Content\Animations\Mixamo

6. Let Unity import the animations and meta data. Then, close and re-open Unity.

7. Open the scene.

8. Setup motion pack on the Packs tab.



DEFAULT CONTROLS

WASD	= Move
Mouse	= Rotate
LShift	= Run
1	= Equip the sword
LMB	= Attack
Alt	= Block
t	= Lock/unlock target

9. Motions are added to the 'Advanced' tab.

Sword & Shield Motion Pack

10/05/2018

Foreword

Thank you for purchasing the Sword and Shield Motion Pack!

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

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

Tim
tim@ootii.com

Contents

Sword & Shield Motion Pack

10/05/2018

Overview

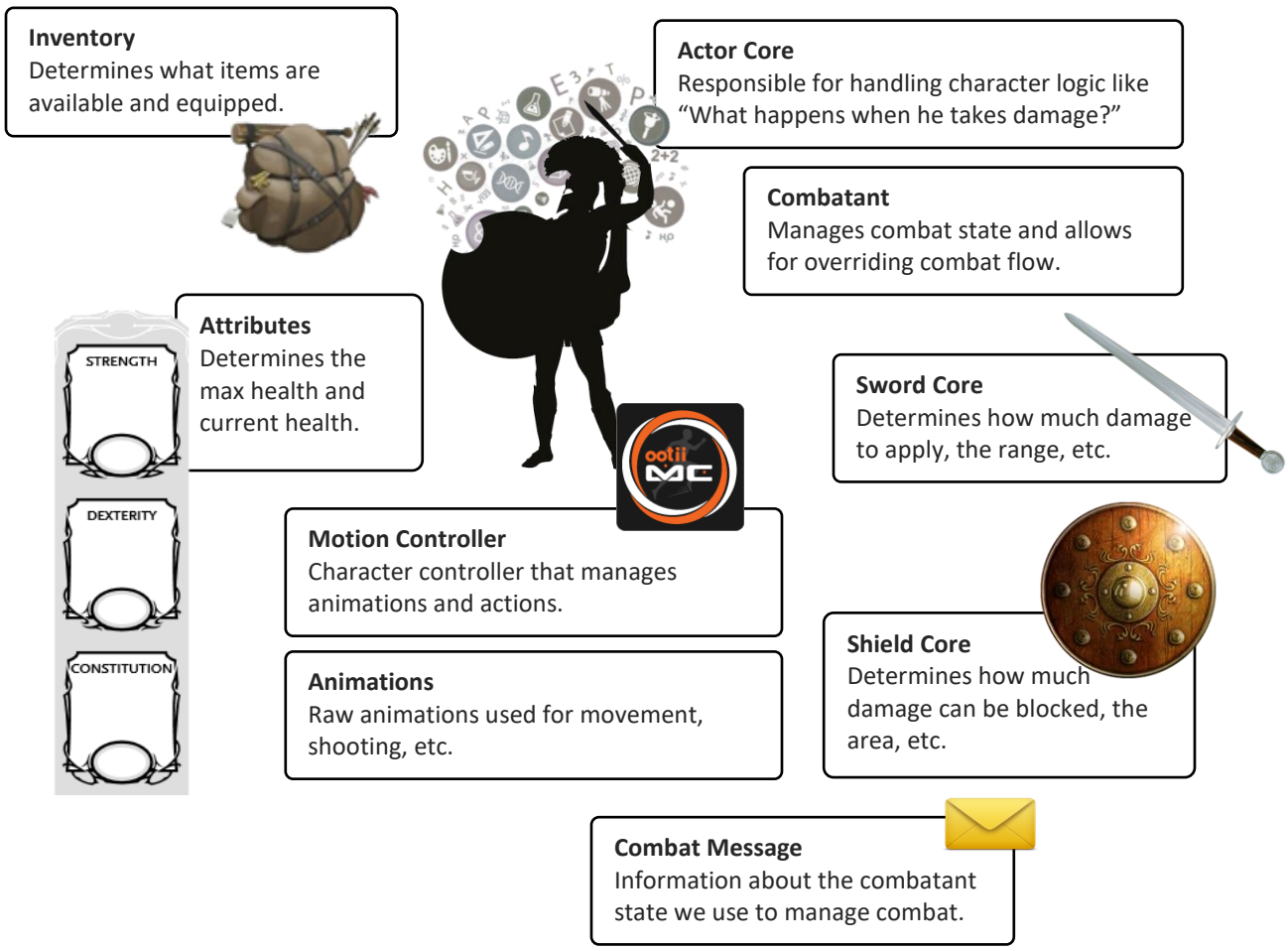
This asset is a bit different from my other assets as it's an add-on pack to the Motion Controller. So, the Motion Controller is required for this asset to work. It also requires Mixamo's free Pro Sword and Shield Pack animations. When those are combined with this asset, your character gains the ability to equip, attack, block, etc.

I've tried to create this asset in a very modular way. This way, you can use it as-is or customize it to work with your game. For example, you can use my basic inventory system or replace it with something like Inventory Pro. In addition, you can use my basic sword or create a shield with special effects. There's a lot of things you can tweak to get exactly what you want.

If you think this add-on is missing a key feature or option, let me know.

Key Components

In order to support different attribute systems, different inventory systems, different arrow types, etc., I created this add-on with multiple components. This is a quick overview and I'll go into them in more detail:



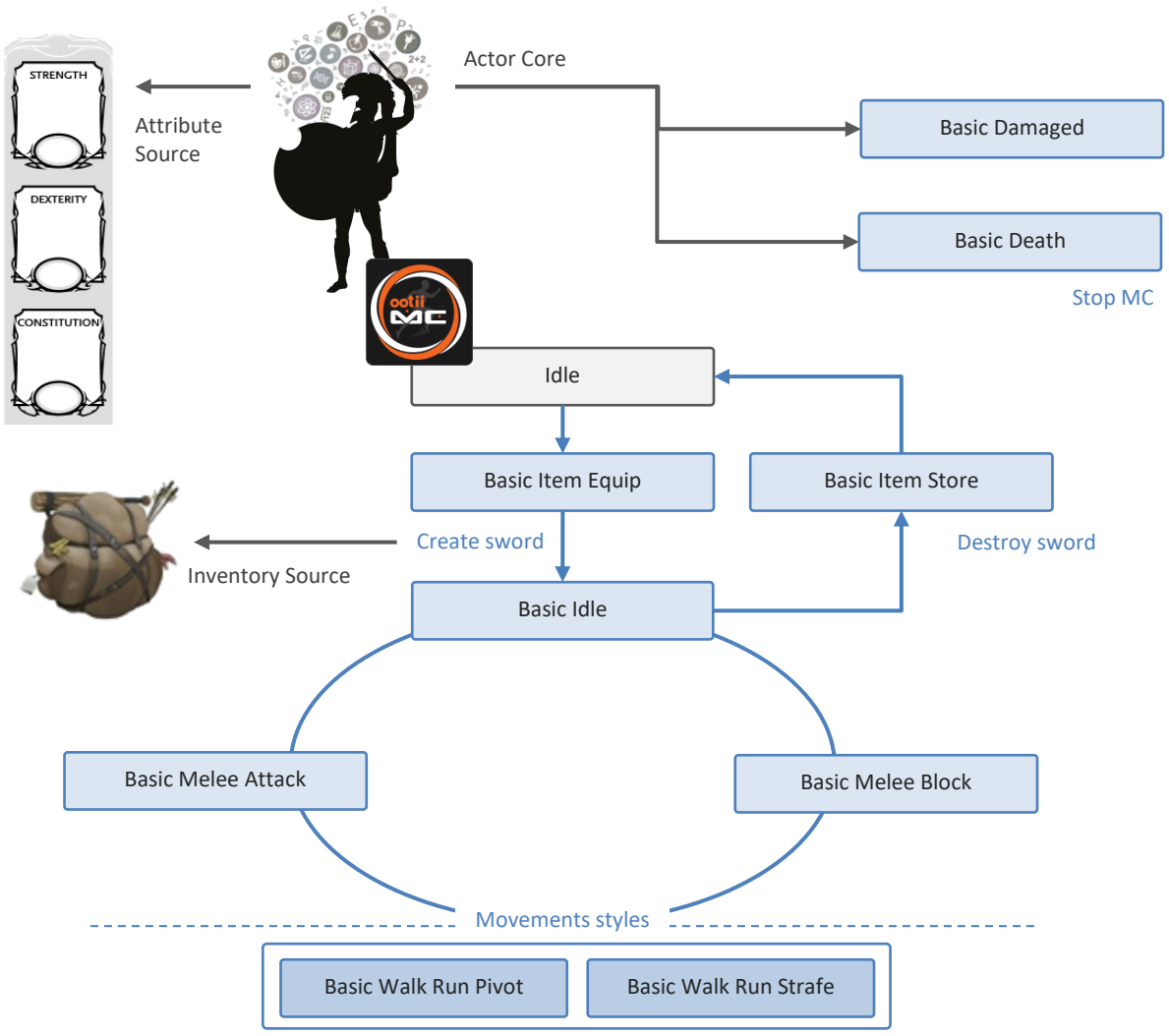
Other than the Motion Controller, each component can be replaced to fit your game's specific needs.

Sword &Shield Motion Pack

10/05/2018

Motion Flow

Just like the motions that come with the Motion Controller, the Sword and Shield Pack is composed of several motions. These motions (blue rectangles below) work together to create the full range of capabilities.



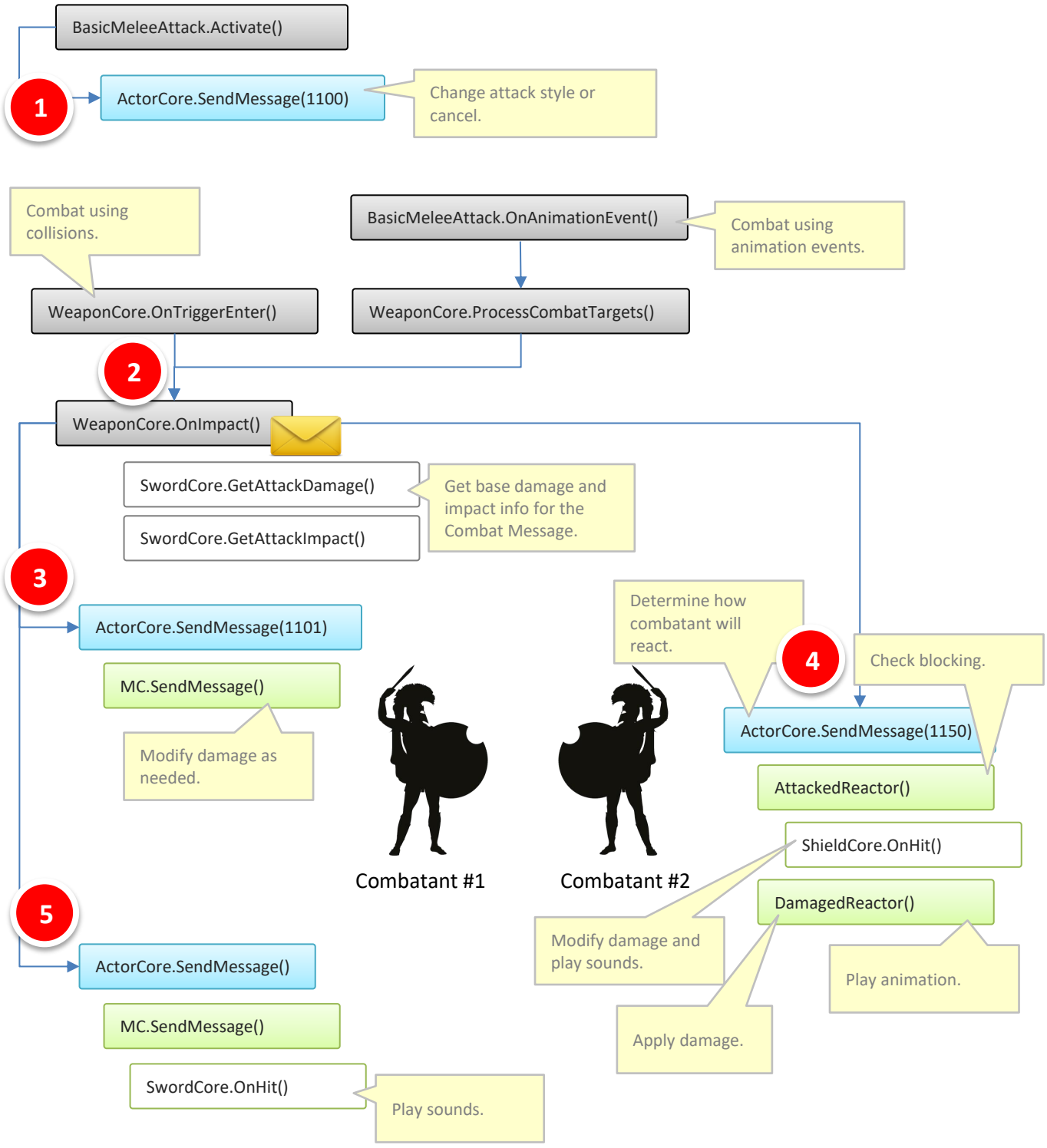
If you don't need a sword and shield motion, want to change how a motion works, or want to add an additional motion... you do it just like any other motion. These motions are just custom motions that take advantage of Mixamo's sword and shield animations.

I've created this pack based on how I think sword and shields should work. In fact, it's very close to the popular Dark Souls games. As you can imagine, there are lots of ways we could do it and your game may be different than mine. If you want the motions to act differently, you are welcome to change these motions or shoot me an email and there may be a feature or option I can add.

For determining which sword or shield to create, we use an "Inventory Source". To manage health, we use an "Attribute Source".

Combat Flow

I use a “Combat Message” to pass around data between combatants. As the combatants perform actions, they update the message. This allows for an act-react style of combat.



Combat Mechanics

This section details how the combat mechanics work. With the component based approach I've taken, you can modify these mechanics for your unique game.

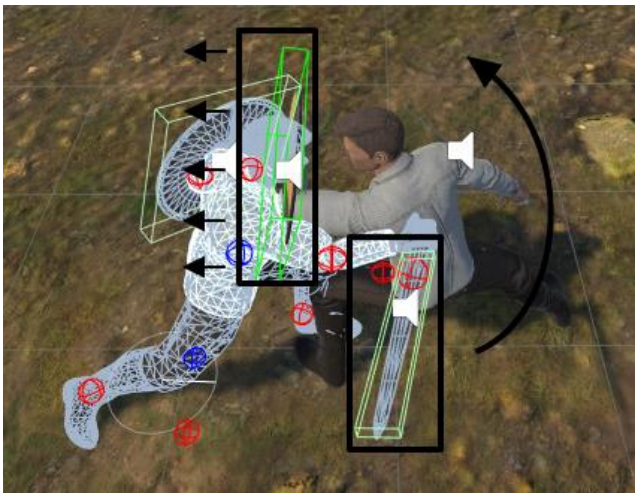
Out-of-the-box, the Sword and Shield Motion Pack approaches combat in one of two ways:

Colliders

By putting colliders on the swords and shields, we'll use Unity's collision system to know when objects hit. With this approach, we'll only react to combat events when there is a true collision.

This approach is more realistic, but can cause issues when objects can't be reached. For example, let's say there's a small box on the ground. You may never be able to hit the box because the animations keep swinging too high.

When using the collider approach, there's some things to take into consideration. For example, what happens when the combatants are too close together and the sword collider doesn't hit the shield?



In this picture for example, you can see the two characters are so close that the sword is behind the enemy's shield.

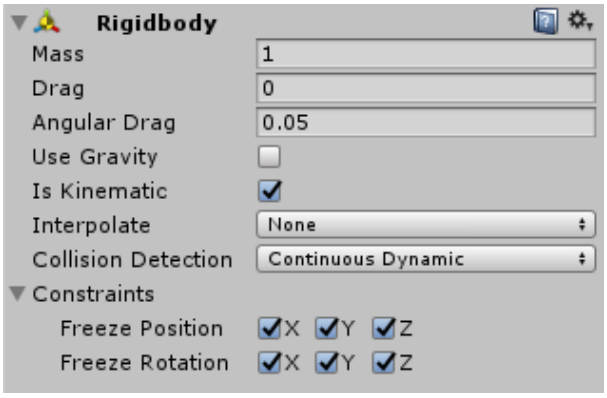
While I do extra logic to compensate for this, it's best to make sure the AC's body shapes prevent characters from getting too close.

I've found a 'Body Capsule' radius of 0.4f and a bottom-offset of (0, 0.8f, 0f) works pretty well.

The downfall to this approach is that we are at the mercy of Unity's collision system. If it doesn't fire the OnTriggerEnter event, a hit won't be reported. I've seen this happen.

Weapons

When setting up the weapon prefab, ensure that it has both a collider and a Rigidbody. The weapon Rigidbody needs to be setup with the following properties:

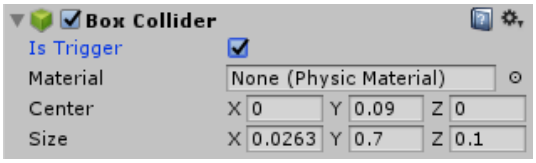


Use Gravity = false
Is Kinematic = true
Collision Detection = Continuous Dynamic
Freeze Position = true (all)
Freeze Rotation = true (all)

Is Kinematic is important as the animation will be controlling the position and rotation of the sword.

Collision Detection is important to capture the collision of the fast swinging sword.

Since Unity's OnCollisionX functions won't fire unless one of the colliders has a non-kinematic rigidbody attached, we use the OnTriggerX functions instead. So, the weapon's collider should be set to "Is Trigger":

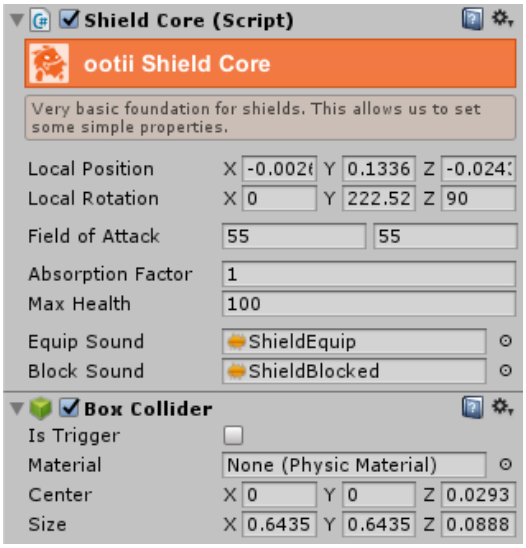


The Sword and Shield logic will actually apply a physics force to objects the sword collides with. So, you will still get physics-based effects when the swords hits another rigidbody.

However, if you need you can simply add another collider to the weapon and have that second one not be a trigger. I don't recommend this, but you can.

Shields

For shields, simply use a standard collider:



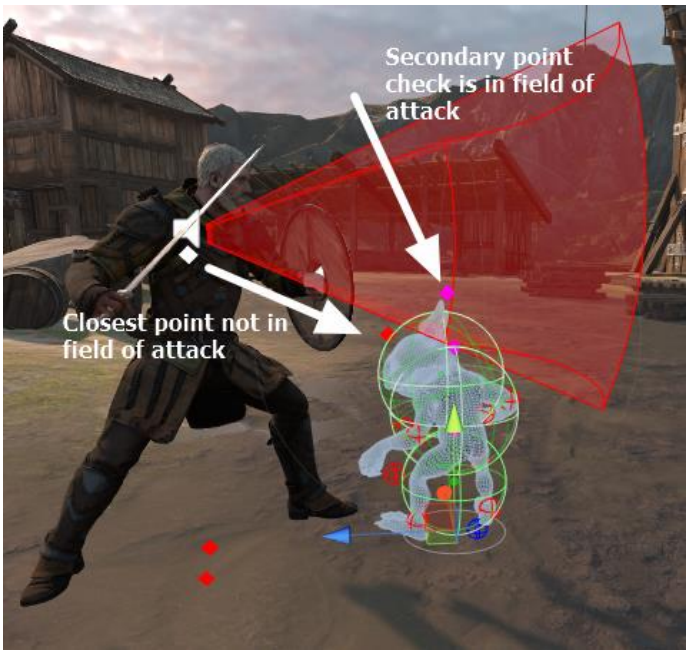
Field of Attack

This approach estimates combat events by testing if closet-points are in specific angles. The attack style properties and shield properties are used to define these angles and test are done at specific times. Neither the sword nor shield would use colliders or rigidbodies.

This approach is typically used in games and doesn't have the overhead of relying on the physics engine. However, it's a lot more general in where an attack can get it.

Test Angles

When using this approach, there may be times when it looks like there should be a hit… but there isn't. That's because I'm using the "closest point" to test angles. Here's an example:



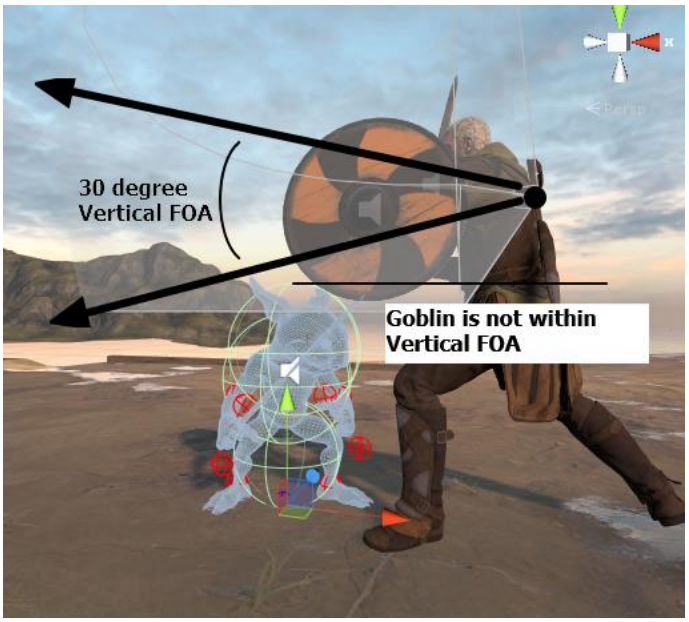
I've determined that the closest point to the human's combat origin (white dot) is the red dot on the goblin's forehead. You can see that it is just under the field of attack… so, no hit.

But the top of the goblin's head really is in the field of attack… just barely.

So, I do a secondary check (the magenta dot) just in case. In this example, the secondary check is in the field of attack. So, we do have a hit.

The point is that I'm not testing to see if the collider sphere is within the "frustum area", but the "closest point" is within the angles.

Here's another example:



In this example, the back slash has a vertical FOA of 30-degrees.

Because the goblin is short and close, he doesn't fall within the vertical FOA. So, no hit. If you want him to hit, you'd simply increase the vertical FOV.

Choosing an Approach

If your swords and shields have colliders on them, I'll automatically use the 'Colliders' approach. If no colliders are found, I'll use the Field of Attack approach.

Whichever approach you use, be consistent across all characters.

Attack Styles

The BasicMeleeAttack motion supports several styles of attacks. Each style is associated with a separate animation, but it doesn't have to be. The idea behind 'Attack Styles' is that they allow us to provide attack specific information during combat.

Pressing the 'Reset' button will add the default attack styles, but you can add or remove them as needed.

Each attack style has the following properties:

Standard Properties

Name: Friendly name for the attack style. It should be unique to other attack styles for this character.

Parameter ID: This is the L0MotionParameterID used by the animator sub-state machine to trigger the specific animation.

Is Blockable: Determines if this attack style can be blocked by a shield.

Damage Modifier: Multiplier to use with the weapon's base damage.

Field of Attack Approach

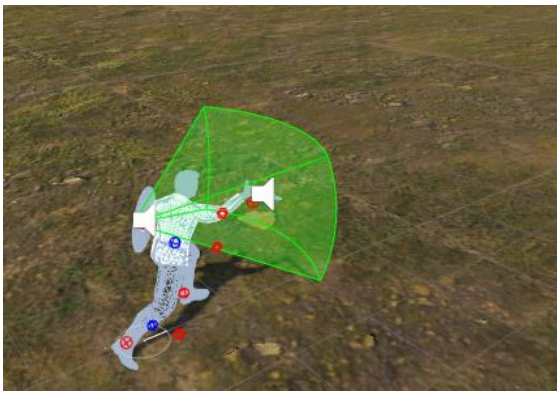
The following properties really only matter when using the 'Field of Attack' combat approach mentioned earlier.

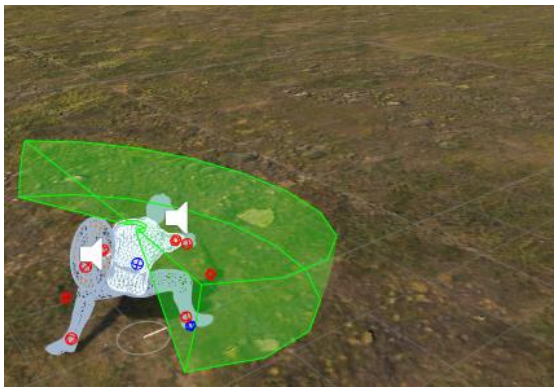
Attack Forward: Center direction of the attack.

Field of Attack: Horizontal and vertical angles the attack occurs in.

Range: Min and max range of the attack. The final attack will also take into account the combatant's reach.

Attack Style Examples





Left is the Forward Slash and right is the Spin Slash.

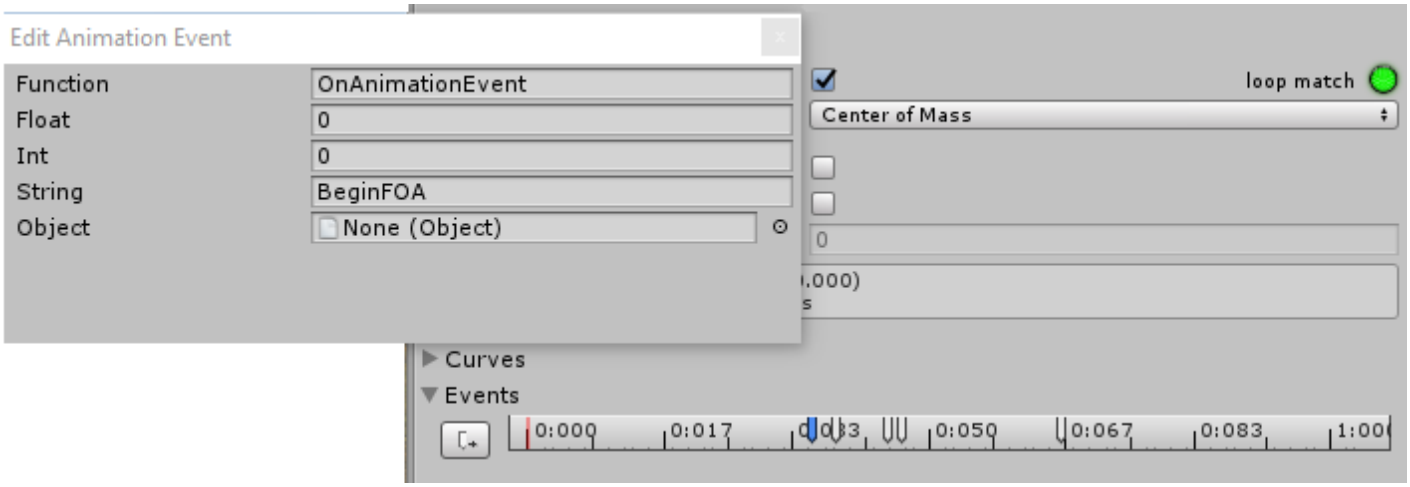
Animation Events

In both approaches, I use animation events to determine when an attack is actually valid. This way, we don't get attack events while a sword is being pulled back. Each attack animation can have the following events defined:

BeginFOA – Begins the time when attack events can occur.

EndFOA – Ends the time when attack events can occur.

Hit – Specific time to test for contact. This is useful if the BasicMeleeAttack motion is not set to 'continuous'. In this case, the only time we'll check for a combat collision is when a 'Hit' animation event is raised.



Chaining Attacks

The Sword and Shield Motion Pack contains a very basic way to create combos or chain attacks.

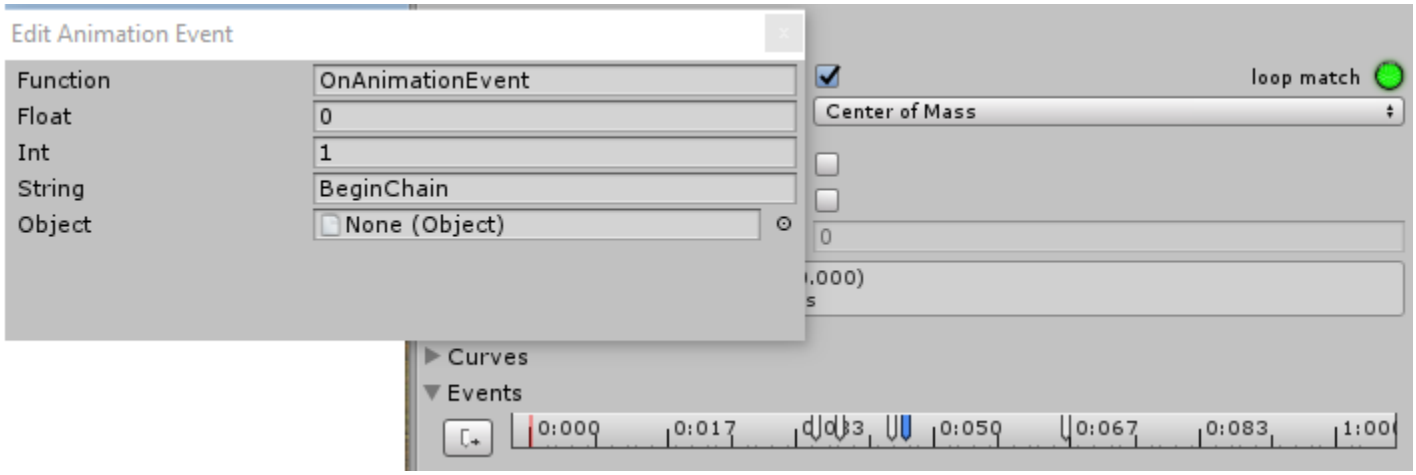
During each of the attack animations, two animation events (similar to above) can be added that define the time when another attack can immediately occur. If the player times a second attack within this window, it will immediately occur.

Each attack animation can have the following events defined:

BeginChain – Begins the time when an attack can immediately happen.

EndChain – Ends the time when an attack can immediately happen.

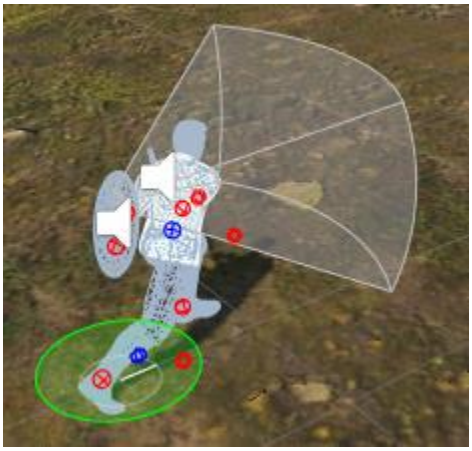
Within the animation events, you can also specify which attack will occur next by using the 'Int' field.



Increasing the amount of time a player has to chain attacks is just a matter of spreading out the animation events.

Debugging

To help with debugging, a green circle will appear at the base of the character when we're within the 'BeginChain' and 'EndChain' windows.



Blocking

I chose for blocking to work a certain way. As always, you can modify the code to work a different way. The idea is that shields can absorb some damage, but not always all. I also wanted players to be able to break a block if they cause enough damage.

Shields are able to absorb a percentage of the incoming attack. When that happens, the amount absorbed is removed from the damage the character takes. In this way, the shield can absorb 100% or only 50% and allow attacks to cause some damage.

Shields store the amount of damage that they absorb (from melee weapons only). When that amount is greater than the shield's max health, the block motion ends. The player (or NPC) would then have to re-activate the block motion or risk being hit for full damaged.

Note that ranged weapon damage is never added to the stored amount even though it is absorbed.

When the character comes out of a block and then goes into a block, the current amount of absorbed damage starts at 0 again.

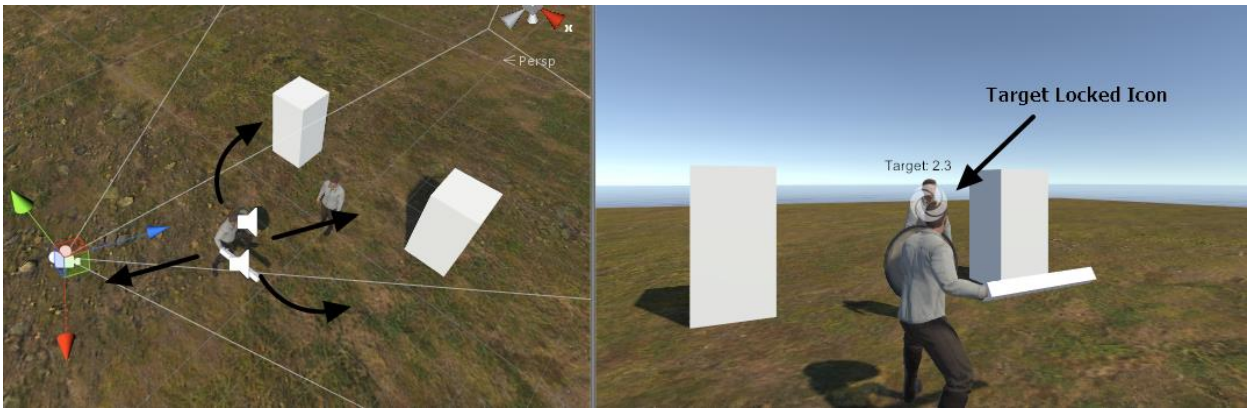
Debugging

When debug is enabled, I'll show the amount of health the shield has vs. its max health.

When the field-of-attack for the shield is green, that means it is ready to take a hit. When gray, the motion is active, but the character isn't ready to block a hit with the shield.

Targeting

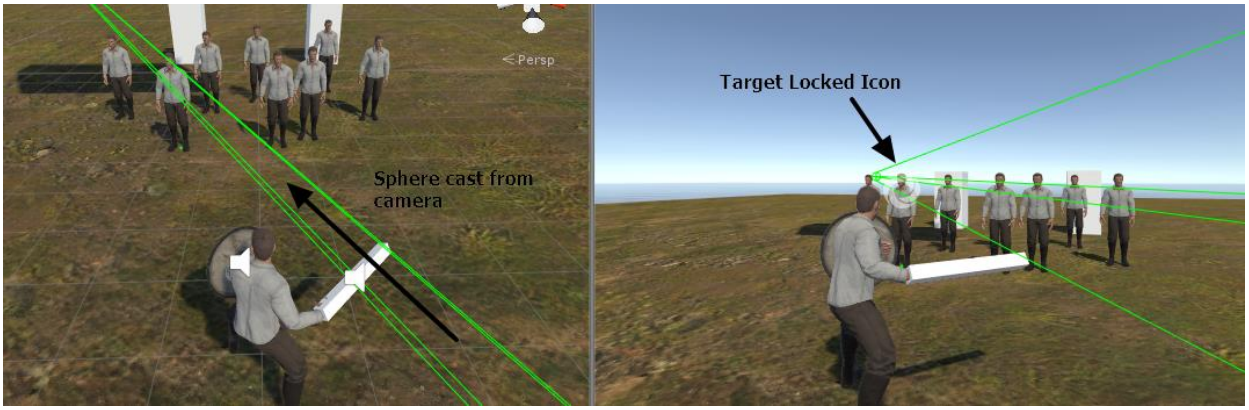
The Combatant is able to target an enemy. When that happens, the camera and character will lock onto the target and movement will be focused on strafing around the enemy.



Choosing a Target

When choosing the target, the combatant goes through a couple of steps:

1. A sphere is cast from the camera and along the camera direction. The first object hit becomes the target.



2. If no target is hit, a sphere is cast from the player and along his forward direction. The first object hit becomes the target.

3. If no target is hit, a proximity check is made against all targets within the area.

Note: In order to increase performance, some short cuts are taken in processing the potential targets that are within the area. If we select a target in step #3, it may not be the closest target.

Custom Targeting Logic

To customize the targeting logic, simply create a new MonoBehaviour and inherit from Combatant. Then, override the FindTarget() function.

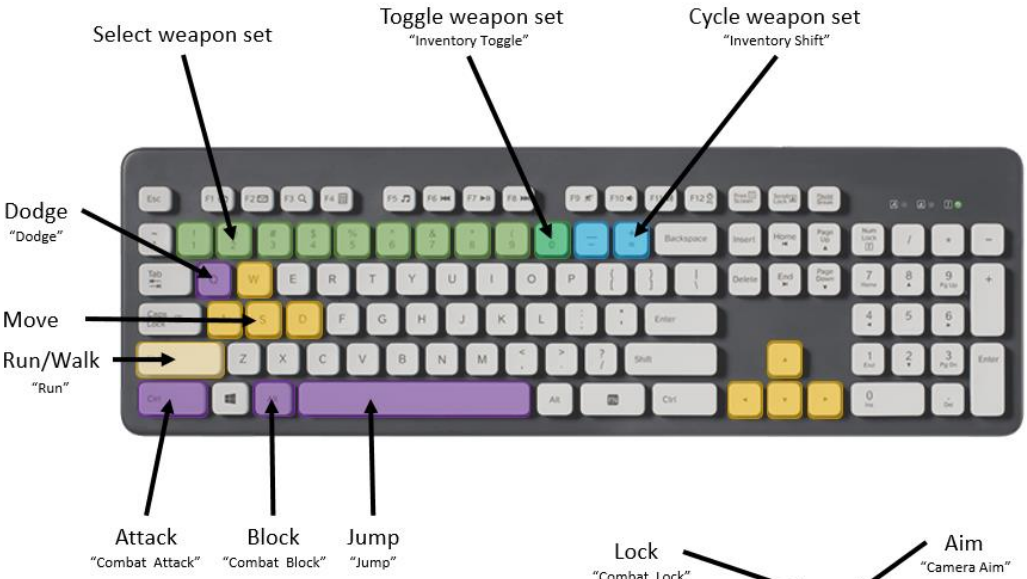
Sword & Shield Motion Pack

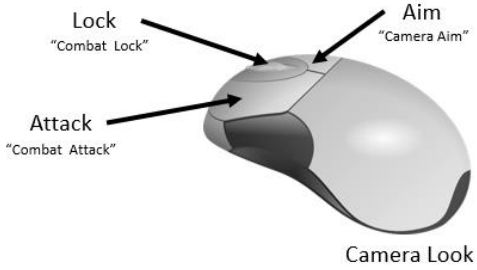
10/05/2018

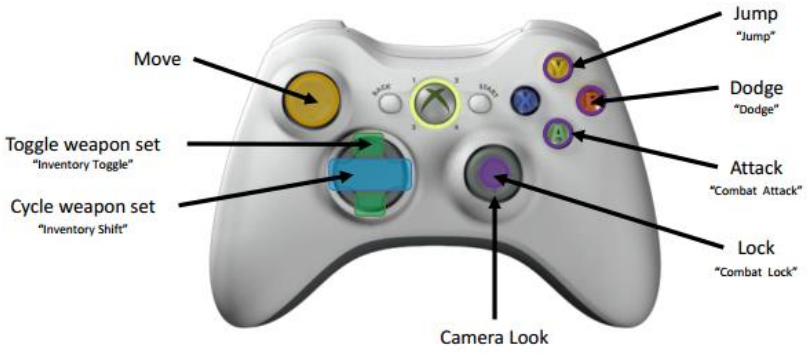
Input

As always, you have complete control over how the motions work. However, I am standardizing controls for all the motion packs. This way, there's a starting point that you can change. (Note: Not all inputs are needed with SSMP)

For the latest input setup, go here: http://www.ootii.com/Unity/MotionControllerPacks/SwordShield/InputMap.pdf





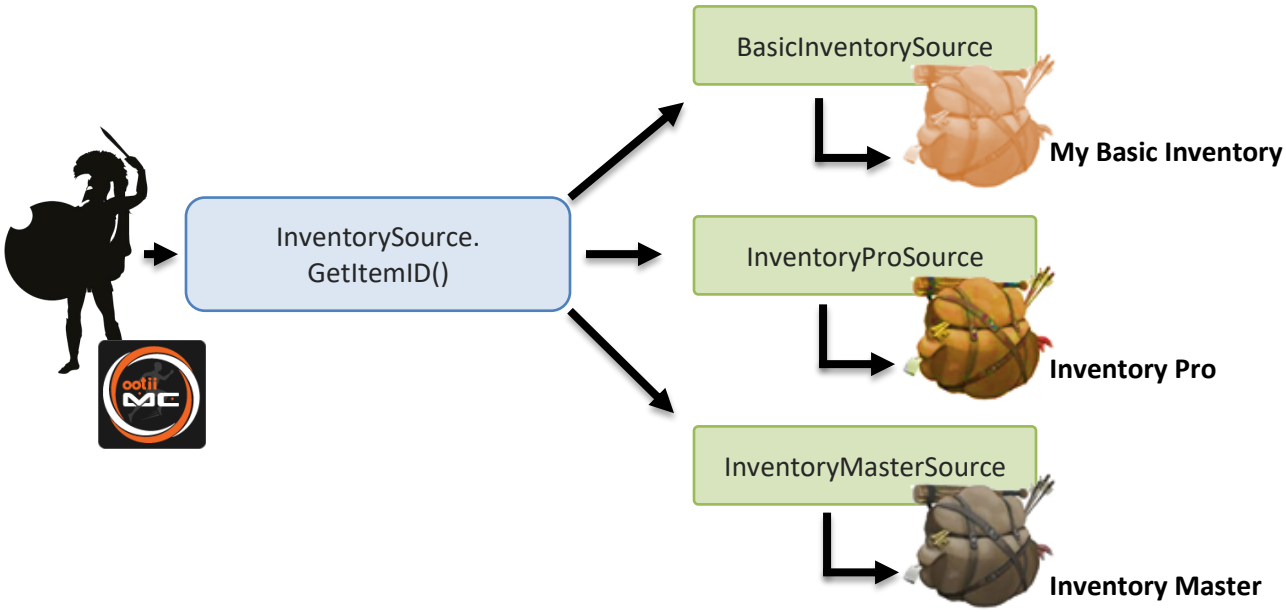


"Sources" Introduction

There's lots of inventory solutions on the asset store. Some developers use Inventory Pro, some use Inventory Master, and some create their own solutions. So, I have to create a way to support them all.

To do this for inventories, attributes, input, etc., I've come up with the concept of "sources". Basically, a source is a bridge between my asset and every other asset. Just like I did with Input Sources, I've created "Inventory Sources" and "Attribute Sources".

Let's look at how this works:



The motions know there's an "Inventory Source" on the character. It doesn't care if it's a "Basic Inventory Source", an "Inventory Pro Inventory Source", or an "Inventory Master Inventory Source". A motion can than ask the "Input Source" for the sword type that is to be equipped.

Depending on the inventory source type, it asks the actual inventory solution for the sword. The resource path is returned to the motion and it creates the item. In the case of Basic Inventory, I allow the inventory source to create the item and I return the GameObject instead of a path.

This is exactly how Input Sources work and how Attribute Sources work too.

Interfaces

All this is capable because of interfaces.

To learn more about interfaces, here are some resources:
Input Sources and Interfaces
Interfaces in C# (For Beginners)
Unity Interfaces

"Cores" Introduction

A "core" represents the heart-beat of a character or object. Typically, this is the MonoBehaviour whose Update() function controls the object. For this pack, we have three specific cores:

Actor Core – This core will be a component on your characters and has OnDamaged() and OnDeath() functions. It allows the actor to respond to arrow hits and other events.

The Actor Core implements the IActorCore interface. This way, you can use the interface with your own "core" (if you have one).

Sword Core – The Sword Core is a component on the sword prefab and defines things like impact power and damage. In this way, you could have two different swords with unique capabilities.

Shield Core – As a component on the shield prefab, this core is responsible for testing for collisions and determining the area of protection. It also decreases incoming damage when the damage is blocked.

Combatant – The 'Combatant' is a type of core that helps manages the state of combat. It contains things like the target, primary weapon, etc. This is useful as we can pass this object around to see the combatant's state.

Motion Packs

The Motion Controller UI has changed slightly and now includes a button for “Packs”.

When the “Packs” button is pressed, the Motion Packs imported into the project will be listed. Selecting a pack will provide detailed information about the pack.

In this case, we select the “Sword and Shield” pack and the Sword and Shield Pack details are listed.

Each pack may show different options. For the Sword and Shield Pack, I’ll go through each option. When the “Setup Pack” button at the bottom is clicked, the checked options are run.

Create Mecanim States – This option is used to create/recreate the Mecanim sub-state machines that are used by the pack’s motions. This is exactly what we do with normal motions. I just create a short cut here.

Create Input Aliases – When checked, all the required input aliases used by the motions are created in Unity’s Input Manager. This only needs to be done ones for the whole project.

Create Inventory – If you don’t have an inventory solution, this option will create a “Basic Inventory” component and add it to your character.

Create Attributes – If you don’t have an attribute solution for things like health, this option will create a “Basic Attributes” component and add it to your character.

Create Combatant – If you don’t have an Actor Core on the character, one will be created for you. This also creates the Combatant component.

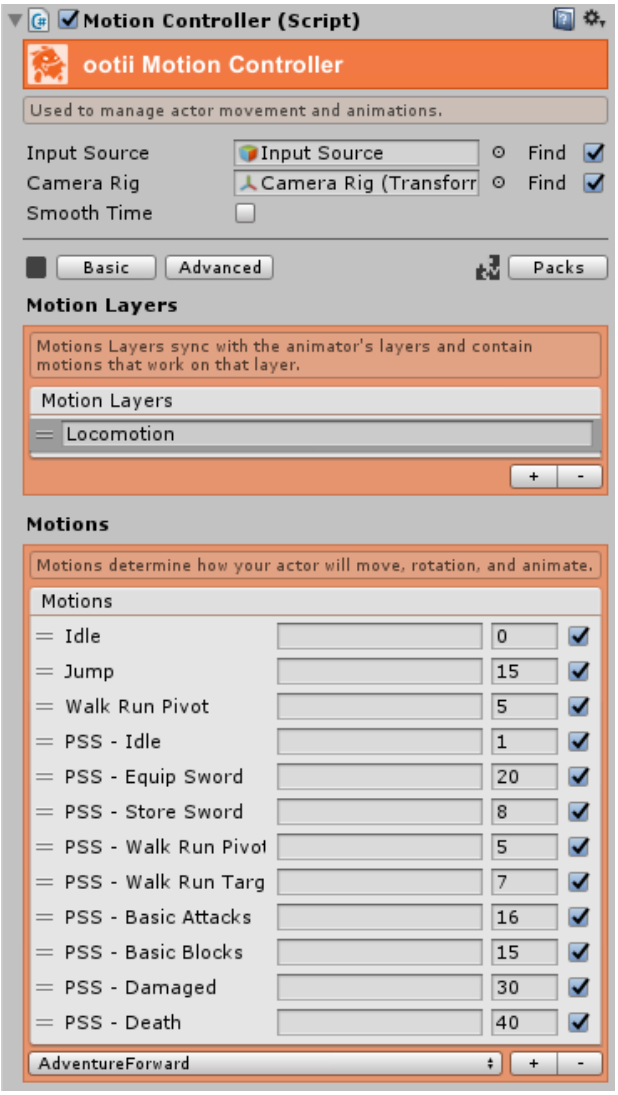
Create Motions – This is the object that actually creates the pack’s motions.

Typically, you’ll just leave all these options checked before pressing “Setup Pack”. However, as you customize your character and include other assets you may no longer need my basic inventory solution.

The following pages show what is created by the pack (assuming all options are checked):

Motion List

With the motions created, if you go back to the “Advanced” tab, you’ll see the new bow motions listed and setup based on the options you’ve checked.

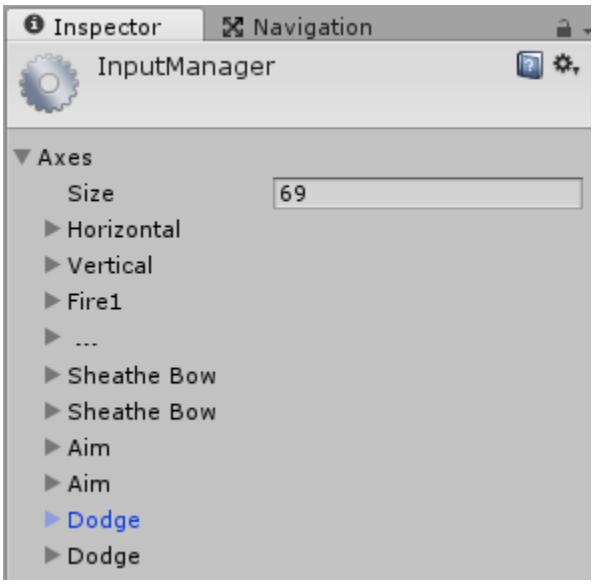


Once setup, you can treat them like any other motion. You can disable them, remove them, activate them using AI, etc. They are motions just like any other motion.

I’ll detail the motions later. I just wanted to show that the motions get created.

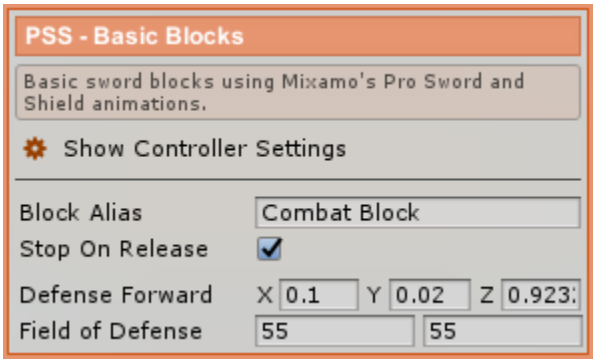
Inputs

For the most part, I use Unity's standard input for moving and controlling the character. However, some new input entries are added to support things like aiming and sheathing the weapon.



For all these extra entries, there is a 'keyboard' entry and an Xbox controller entry.

Following my Motion Controller and Input Source standards, each motion that requires a special key (like BasicMeleeBlock), I allow you to change the input entry. I call these Action Aliases and they trigger the motion. For example, BasicMeleeBlock uses the "Combat Block" entry in Unity Input Manager.



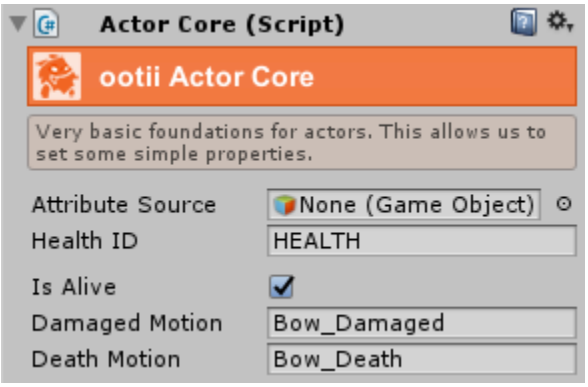
Using this approach, you can change the input entry the motion uses by setting the "**Action Alias**" or changing the key or button that the alias represents.

How you change the key or button depends on the input solution you're using. For standard Unity Input Manager, you'd just select the 'Edit | Project Settings... | Input' menu item in Unity.

Actor Core

As I mentioned, the Actor Core lives on all your characters and represents the decision making and logic part of your character. When a sword hits a character with an Actor Core, it's the Actor Core that receives the damage and reacts appropriately.

Note that you don't need an Actor Core if the object is an object that doesn't get damaged or destroyed.



The Actor Core is pretty basic. It has the following properties:

Attribute Source – The component that contains the actor's attributes.

Health ID – String that is the ID of the attribute that holds our health value.

Is Alive – Simple boolean that determines if the object is still active.

Damaged Motion – Name of the motion to activate when damage is taken.

Death Motion – Name of the motion to activate when the actor takes so much damage that it should die.

Code Summary

Internally, the Actor Core has a couple of key functions:

OnDamaged() – This function is called by the weapon when it impacts an object that has an ActorCore. This is how the weapon tells the character it has been hit and how much damage is done.

In this function, the ActorCore asks the Attribute Source for how much health exists. If the damage exceeds this health, the character dies.

OnDeath() – This function is called by the OnDamaged function if the damage exceeds the character's health.

Once the death animation finishes, the Motion Controller is disabled.

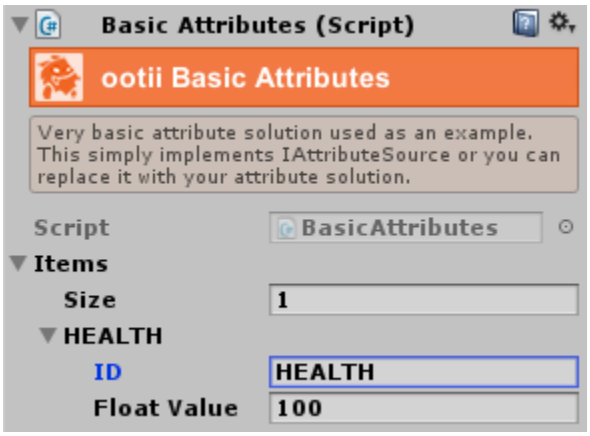
IActorCore Interface

If you have your own "actor heart beat" MonoBehaviour, you can simply add the IActorCore interface to your class and ignore this one. Again, the goal is to be modular and replaceable.

Basic Attributes

While we could have simply stored the health attribute inside the Actor Core, I wanted to make sure we were modular enough to support RPG assets that may be managing the attributes.

If you recall from earlier, I talked about Attribute Sources. A source is simply a bridge that I can use to grab attributes regardless of the solution being used. This “Basic Attributes” component is my very basic version of an RPG attribute asset. It implements the IAttributeSource interface as all Attribute Sources would.



By implementing attributes this way, you can add new attributes to your game. For the add-on, we only care about one attribute:

HEALTH – Current health a character actually has. Remember this is the ID that we used in the Actor Core above.

Code Summary

As an Attribute Source, the following functions are available:

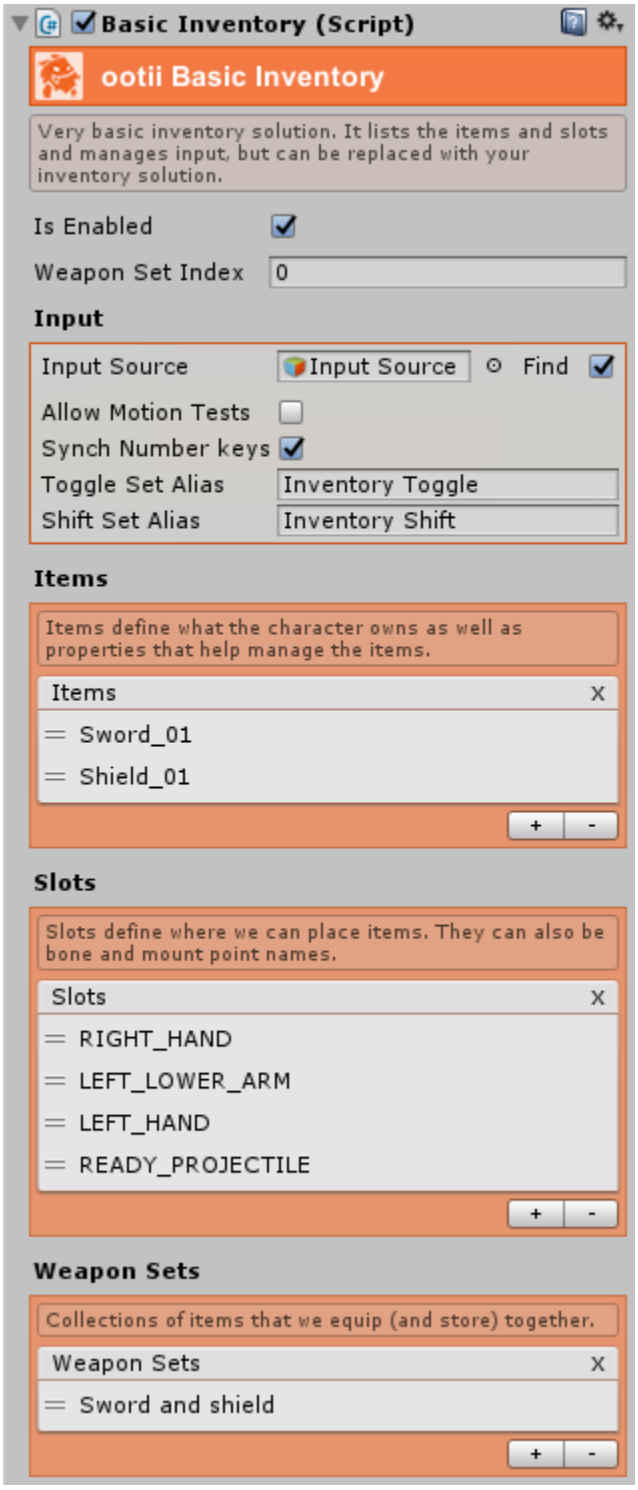
GetAttributeValue() – This function returns the value of the attribute given the name.

SetAttributeValue() – This function sets the value of an attribute given its name.

As you can image, these two functions are used by the Actor Core to get and store the character’s current health. The Actor Core will determine if the character is simply damaged or killed based on the “HEALTH” attribute.

Basic Inventory

While you can use any inventory solution you want, I've included a "Basic Inventory" solution for you. As we've been talking about, any inventory solution you use will need to implement the IInventorySource interface. This way, we know how to retrieve information from it.



My "Basic Inventory" solution is very basic, but it is an Inventory Source. My implementation contains three lists; Items, Slots, and Weapons Sets.

The **Items** list contains all the items that the character has in his inventory and some properties for them.

The **Slots** list represents the usage of items.

So, the "RIGHT_HAND" slot defines what is equipped in the right hand.

The "READY_PROJECTILE" slot determines what arrow is ready to be used.

In the example to the left, the character has a sword ("Sword_01") and a shield ("Shield_01").

The **Weapon Sets** allow you to group items so they can be equipped and stored at the same time. For example, "Sword and shield" includes Sword_01 and Shield_01. You can imagine that another weapon set my be a bow and arrow.

Item

Each item has the following properties:

ID – Simple string that uniquely identifies the item.

Equip Motion – Name of the motion used to equip the item

Store Motion – Name of the motion used to store the item

Instance – Edit-time created instance that is the item

Resource Path – This is a path of a Unity Resource Folder where we can find the prefab that represents the item. We use this value to create the bow and arrows when the time comes.

Local Position – When mounted, the position relative to its parent. When Mount Points is used, this value is ignored.

Local Rotation – When mounted, the rotation relative to its parent. When Mount Points is used, this value is ignored.

To learn more about Unity resources, look here: https://unity3d.com/learn/tutorials/topics/best-practices/resources-folder

Slot

Each slot has the following properties:

ID – Simple string that uniquely identifies the slot.

Item ID – ID of the item that is currently in the slot. An empty value means the slot is empty.

Code Summary

As an Inventory Source, the following functions are available:

EquipItem() – Equips the specified item in the specified slot.

StoreItem() – Stores the item that is in the specified slot.

GetItemID() – Gets the item ID of the item in the specified slot.

GetItemPropertyValue<T>() – Gets the requested property value from the specified item.

These functions are used by the Sword and Shield motions to determine what sword and shield should be created and to equip/store the weapons.

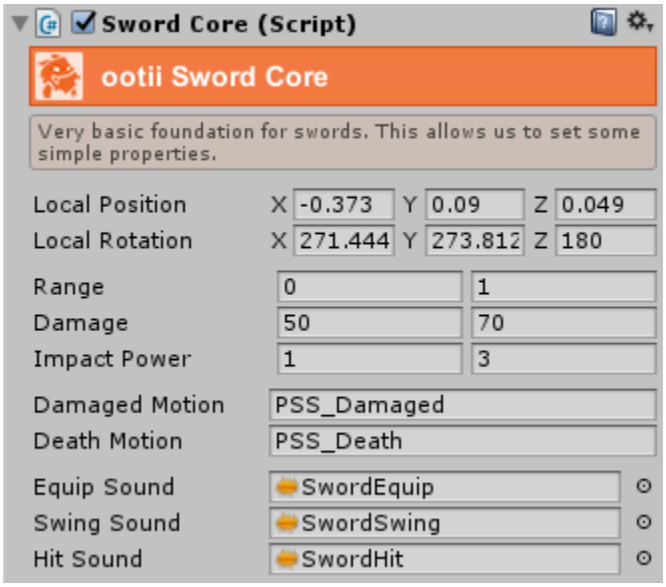
Again, you can use any inventory solution you want. However, that solution needs to implement the IInventorySource interface.

Swords

In the pack, there's a sword included that you can use in your game. You can customize the properties or create your own sword following the guidelines I'll go over.

The sword is made up of three parts: the model, the core, and the prefab. The model and prefab are standard Unity things. We'll use a static sword model, set some properties, and turn that into a prefab.

Sword Core



The Sword Core is a MonoBehaviour that will live on the sword prefab. It contains basic information about how the sword will work:

Local Position & Rotation: When placed in the character's right hand, the position and rotation value used to place it correctly.

Range: The reach of the sword.

Damage: How much damage is applied on a hit. Currently I only use the "min" value (that's the first number).

Impact Power: How much physics impact is applied on a hit. Currently I only use the "min" value (that's the first number).

Damaged and Death Motions: Motions to trigger on the target. The target doesn't have to use this motion, but it can if it doesn't know another one to use.

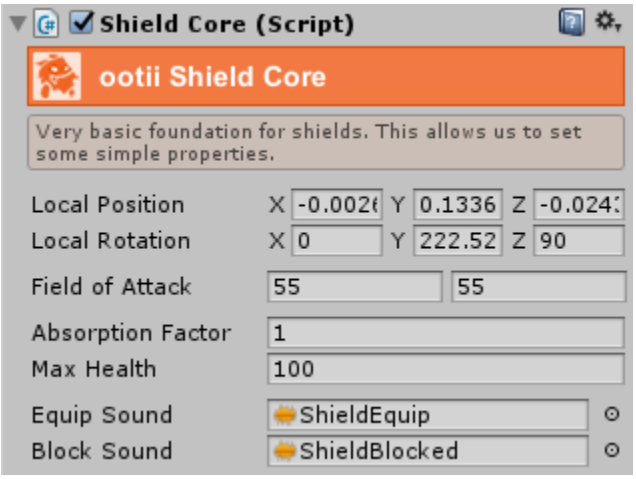
Sounds: Sounds that play on the specific events.

Shields

In the pack, there's a shield included that you can use in your game. You can customize the properties or create your own shield following the guidelines I'll go over.

The shield is made up of three parts: the model, the core, and the prefab. The model and prefab are standard Unity things. We'll use a static shield model, set some properties, and turn that into a prefab.

Shield Core



The Shield Core is a MonoBehaviour that will live on the shield prefab. It contains basic information about how the shield will work:

Local Position & Rotation: When placed in the character's right hand, the position and rotation value used to place it correctly.

Field of Attack: The horizontal and vertical angle that creates a cone. If an attack comes from within that cone, it will be blocked.

Absorption Factor: A value of 0 to 1 (representing 0% to 100%) that says how much the incoming damage is reduced by.

The shield will absorb this damage. The remaining damage is applied to the character.

Max Health: How much damage can be absorbed before the Block motion is disabled.

Sounds: Sounds that play on the specific events.

Positioning Items

If you're using Mount Points, ignore this part.

If you're not using Mount Points, you may need to adjust the Local Position and Local Rotation to fit your character. Since every character model could be different, you may have to modify the position and rotation to match with the hand and arm bones of your character.

In fact, you may need multiple sword prefabs and multiple shield prefabs in order to support different characters.

With each, the easiest way to fill in the Local Position and Local Rotation is to put the sword and shield prefabs into the scene. Then, attach them to the bones of your character manually and position and rotate them to your liking. Then, copy the values from the Transform into the Local Position and Local Rotation properties. I describe this more in step 4 of Custom Swords.

Don't forget to press Unity's "Apply" button and then delete the prefabs from the scene.

Also, I suggest creating a copy of my prefabs to make your changes. Otherwise I'll overwrite you changes during updates.

Motions

Earlier, I gave a quick description of each motion. Here I'll go into more detail about the motion properties.

Basic Idle

Simple idle animation with weapon in hand.

Rotate With Input – Determines if the character rotates as the mouse (right stick) moves.

Rotate With Camera – Determines if the character rotates to match the direction of the camera.

Rotation Speed – Degrees per second to rotate.

Use Pivot Animations – Determines if pivot animations are used while rotating. Since the rotation can be in increments smaller than 90 degrees, the animations may be over-kill. You can use them a desired.

Basic Item Equip

Unsheathe animation that determines the right sword to equip and then equips it.

Action Alias – Input entry that equips the weapon

Inventory Source – Inventory source we'll get the bow resource path from. If left empty, we'll check if the inventory source is on this character.

Slot ID – Slot ID of the inventory source where we'll place the weapon.

Item ID – Item ID of the inventory item that defines the weapon.

Resource Path – If you don't want to use an inventory source, the resource path to the sword prefab to instantiate.

Basic Item Store

Puts the sword away and returns to the normal (non-weapon) idle.

Action Alias – Input entry that stores the sword.

Inventory Source – Inventory source we'll use to say we've un-equipped the sword.

Slot ID – Slot ID of the inventory source we will clear.

Basic Walk Run Pivot

Movement motion with sword in hand. This movement style is more "Adventure" style and if use with an orbiting camera, allows the character to walk towards the camera.

Default to Run – Determines if the character runs or walks by default (Action Alias inverts it).

Run Action Alias – Used to trigger the character to run (or walk) based on the Default to Run option

Walk Speed – When greater than 0, overrides root-motion and creates a constant velocity (units per second).

Run Speed – When greater than 0, overrides root-motion and creates a constant velocity (units per second).

Rotate With Camera – Determines if the character will rotate with the camera as it rotates.

Rotate Action Alias – Key that must be pressed for Rotate with Camera to work.

Rotation Speed – Degrees per second to rotate.

Use 180 Pivot – Determines if the character will pivot 180 degrees when input is 180 degrees from the camera.

Use Tap to Pivot – Determines if we'll allow the character to pivot 90 degrees when idle and a direction is tapped.

Smoothing Samples – Number of values to average to get the actual movement velocity and direction. The more samples, the smoother. However, the more samples the less responsive.

Basic Walk Run Strafe

Movement motion with sword in hand. This movement style is more "Shooter" style and always has the character facing forward. It supports strafing.

Similar to the standard WalkRun... motions, you would only have WalkRunPivot or WalkRunStrafe enable. Not both.

Activation Alias – Input alias that activates this motion (if needed).

Default to Run – Determines if the character runs or walks by default (Action Alias inverts it).

Run Action Alias – Used to trigger the character to run (or walk) based on the Default to Run option.

Walk Speed – When greater than 0, overrides root-motion and creates a constant velocity (units per second).

Run Speed – When greater than 0, overrides root-motion and creates a constant velocity (units per second).

Rotate With Camera – Determines if the character rotates to match the direction of the camera.

Rotate With Input – Determines if the character rotates as the mouse (right stick) moves.

Rotation Speed – Degrees per second to rotate.

Smoothing Samples – Number of values to average to get the actual movement velocity and direction. The more samples, the smoother. However, the more samples the less responsive.

Basic Melee Attack

Motion used to swing the sword.

Attack Alias – Used to trigger the character to swing the sword.

Rotation Speed – Degrees per second to rotate.

Default Attack Style – Index to use when attacking (and not overridden). A value of -1 means randomize the attack style.

Test Continuously – When not using colliders on the weapons, determines if we test for hit only once (based on the animation event) or multiple times until there is a hit.

Attack Styles – See Attack Styles

Basic Melee Block
Uses the shield to block incoming attacks.

Rotate With Camera – Determines if the character rotates to match the direction of the camera.

Rotate With Input – Determines if the character rotates as the mouse (right stick) moves.

Rotation Speed – Degrees per second to rotate.

Block Alias – Input alias use to trigger the block.

Stop On Release – If checked, releasing the Block Alias key releases the block. Otherwise, the Block Alias acts like a toggle.

Defense Forward – Forward direction of the block (relative to the character). This represents the center of the shield.

Field of Defense – Horizontal and vertical angles that create the code of defense. Attacks within the cone are blocked. The center of the cone comes from the Defense Forward.

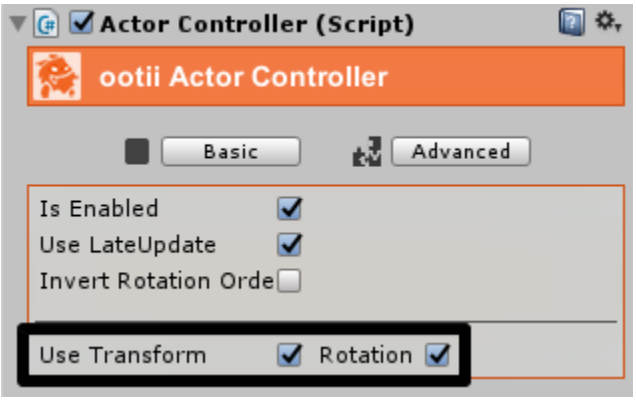
Basic Damaged
Animation used when the actor is hit and his holding a sword.

Basic Death
Animation used when the actor is killed while holding a sword.

NPCs
You can use the MC and this motion pack with NPCs. For the most part, you'll set the NPC up the same way you do your character.

Or course, how you control your NPCs, the AI asset you use, and how they act is completely up to you.

Movement
More than likely, you'll want to check the 'Use Transform' and 'Rotation' options on the Actor Controller.



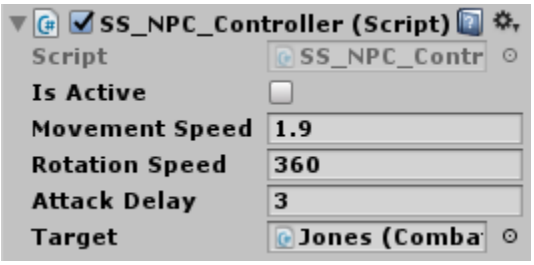
Doing this will allow your AI package to control NPC movement simply by setting the NPC's transform's position and rotation values.

The Motion Controller will then activate the right movement motions as needed.

AI Example

In the asset, I've included a simple code-based AI MonoBehaviour. This isn't meant to be advanced, but an example on how you can code your own NPCs.

You'll find the file here:
Assets\ootii_Demos\MotionControllerPacks\SwordShield\Scenes\SS_NPC_Controller.cs



When added to an NPC, it will equip the sword and shield, chase after the target, and attack.

It's only built for a simple flat ground. So, don't expect it to work on terrain. Again, it's only an example.

So, let's look at some of the code.

Movement

With the Actor Controller's "Use Transform" property set, you can move the character simply be setting the transform's property value. I do this at line 141:

```
transform.position = transform.position + (lToTargetDirection * lSpeed);
```

Equip Sword

Around line 113, you'll see that we just make this call:

```
mBasicInventory.EquipWeaponSet();
```

Block

To activate blocking with the shield, I do this logic around line 148:

```
CombatMessage lMessage = CombatMessage.Allocate();
lMessage.ID = CombatMessage.MSG_COMBATANT_BLOCK;
lMessage.Attacker = null;
lMessage.Defender = gameObject;

mMotionController.SendMessage(lMessage);
```

```
CombatMessage.Release(lMessage);
```

The message ID (MSG_COMBATANT_BLOCK) is what's telling the NPC to block. What's nice about this approach is that if we had other weapons active (say dual swords), those motions could respond to the message just like the sword and shield motions do.

Cancel Block

At line 187, we cancel the block by doing this:

```
CombatMessage lMessage = CombatMessage.Allocate();
lMessage.ID = CombatMessage.MSG_COMBATANT_CANCEL;
lMessage.Attacker = null;
lMessage.Defender = gameObject;

mMotionController.SendMessage(lMessage);
CombatMessage.Release(lMessage);
```

Attack

Similar to blocking, we can use an 'attack' message to tell our NPC to attack (see line 163):

```
CombatMessage lMessage = CombatMessage.Allocate();
lMessage.ID = CombatMessage.MSG_COMBATANT_ATTACK;
lMessage.Attacker = gameObject;
lMessage.Defender = Target.gameObject;

mMotionController.SendMessage(lMessage);
CombatMessage.Release(lMessage);
```

The message ID is key here too. Again, if the character had a different weapon set active, those motions could respond to the attack message just like Sword and Shield motions do.

Customizations

Given the Key Components, Motion Flow, and Combat Flow above, you can customize the system anyway you want.

A couple of key areas you can expand on:

Combatant

Inheriting from Combatant.cs, implementing ICombatant.cs, or tapping into the associated events is one of the best ways to customize your combat flow. If you refer to the diagram about the flow, you'll see you can override several functions to modify how and when combat happens:

OnAttackActivated – This function is called by the attack motion when the motion is activated. It gives you a chance to modify the attack style that will be used or simply cancel the attack all together.

OnPreAttack – Allows you to respond to the combat message before it is sent to the defender. You could modify the damage, cancel the attack, etc.

OnPostAttack – Allows you to respond to the combat message after it was sent to the defender. This way you could respond to a block, clear attack details, etc.

OnAttacked – This is called on the defender so they can respond to the attack. They could block, take damage, die, etc.

FindTarget – Called in order to choose a target to lock onto. You can override this to change which character gets locked on.

OnTargetLocked – Occurs when the target is locked onto.

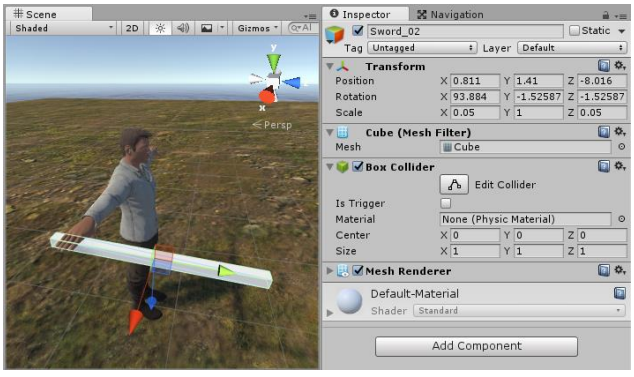
OnTargetUnlocked – Occurs when the target lock is released.

Events

Instead of overriding the Combatant.cs, you can also just tap into events. The names of events are nearly the same as the functions above. For example "onPreAttack" has an event delegate called "PreAttack".

Custom Swords

To create a custom sword:



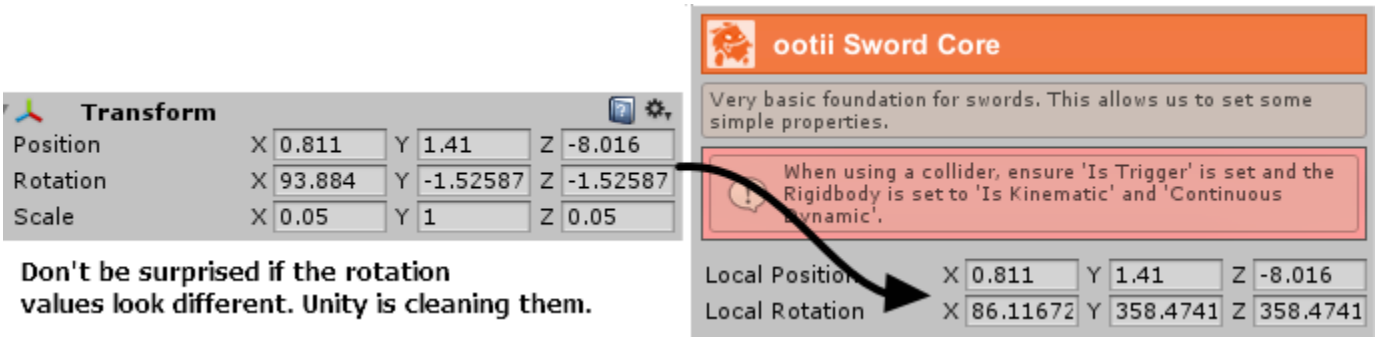
1. Import your custom sword model into Unity

2. Add your custom sword model to the scene.

Parent it to your character's hand. Rotate and position the sword as it would really fit in the hand.

3. Add the SwordCore to your sword.

4. Change the “Local Position” and “Local Rotation” properties of the SwordCore to match the current transform of the sword in the scene. This will allow us to remember how the sword fits.



5. If you are using colliders, ensure your collider is set to ‘Is Trigger’ and the Rigidbody has the following properties:

- Use Gravity = false
- Is Kinematic = true
- Collision Detect = Continuous Dynamic

6. Drag the sword item in the Hierarchy tab to your resources folder in the Project tab to create the new prefab.

7. Delete the sword instance from your scene.

Note that creating custom shields follows the same process, but with the ShieldCore component.

Actor Core, Combatant, Sword Core, Shield Core

Each of these components handle specific functionality. While you can modify the properties I provide, you could also create new MonoBehaviours that inherit from these components and add your own features.

Flaming Sword

For example, let’s say you want a flaming sword and the damage will be multiplied when the flames are on.

You would create a new MonoBehaviour called FlamingSwordCore and inherit from SwordCore. Then, add a Boolean to help determine if the flames are active or not. Finally, override the SwordCore’s GetAttackDamage() function to return the extra damage when the flames are active.

Glowing Shield

Let’s say you want a shield to glow brightly when it’s hit. Maybe you’ll use that to blind the attacker…

You would create a new MonoBehaviour called GlowingShieldCore and inherit from ShieldCore. Then, add a particle effect that represents the glow that will occur. Finally, override the ShieldCore’s OnHit() function to activate the particle effect when the combat message’s ID is equal to MSG_DEFENDER_BLOCKED.

Motions

Another way to customize the combat is to create new motions. You can inherit from my motions and override the Update(), OnAnimationEvent(), or OnMessageReceived() events to do whatever you like. You can modify the combat message, activate new motions, etc.

Sword &Shield Motion Pack

10/05/2018

Frequently Asked Questions

Below is a list of how-to and responses to questions that I get (or expect to get). Hopefully they help provide some quick insight and tutorials.

Pre-Purchase

Can I use this with Inventory Pro, Rewired, or other non-ootii assets?

Yes. I've built this (and the MC) to be very modular. You can simply replace pieces as needed.

Combat

How do I increase the time for a combo or chaining?

Simply move the animation events on the attack animation itself. Currently I use about 0.25 seconds, but you can increase or decrease that time by changing when the 'BeginChain' and 'EndChain' events fire.

Can I use this as a foundation for other weapons?

Yes! In fact, I plan to do exactly that.

I'm going to assume you've made a copy of BasicMeleeAttack.cs and the animator. This way when I update the asset, you don't lose your changes.

Let's say you want to use a 2-handed ax... here are some rough steps that you'll need to follow to do it.

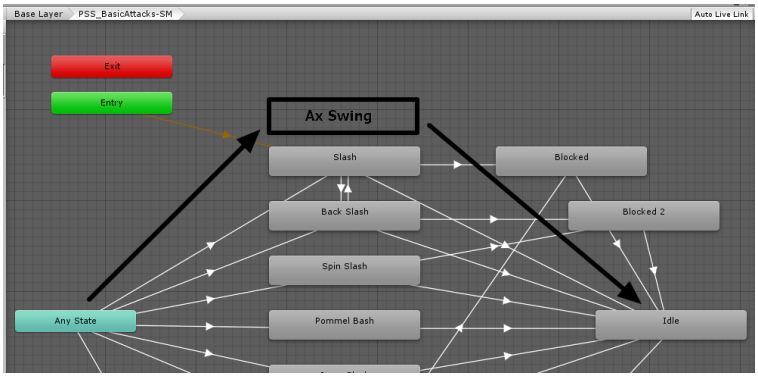
1. Read through this documentation and make sure you understand all the parts. ☺

2. Ensure you have animations that fit your 2-handed ax. Add the animation events to those animations.

3. Ensure you add the Sword Core to your ax's prefab and set the local position and rotation.

4. Setup your attack styles to fit the ax animations. The "Parameter ID" will be important as that's the number that we'll use in the animator to trigger your new attacks.

5. Add animator states. Here you'll have to modify the animator you're using. You'll add the ax animation that matches the attack style you just created.



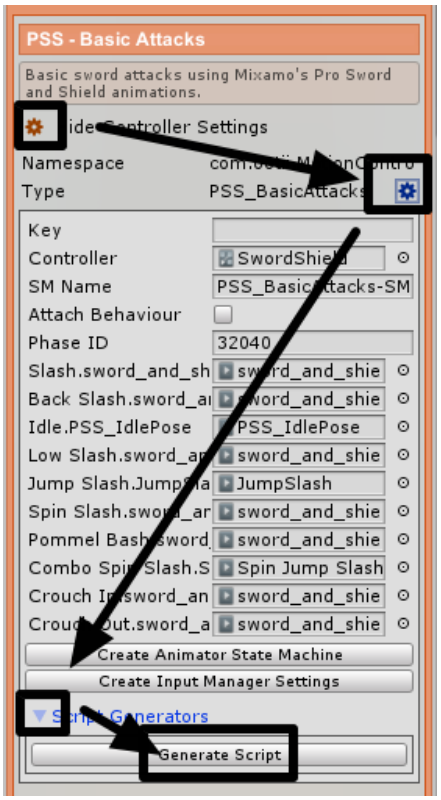
In doing this, you'll add the transition timings to fit your ax animations. To start simple, we'll just play the ax swing and exit to the idle.

The key is that the transition to your ax swing will need to use the "Parameter ID" you used with the attack style.

If you follow what I do with the other attach styles and transitions, it should all make sense.

6. Now, here's the tricky part... Because you modified the animator sub-state machine, you need to re-auto-generate the code for your copy of BasicMeleeAttack. This ensures the motion knows about your changes.

To do this, add your copy of BasicMeleeAttack to the character's motion list and click these buttons...



This will copy code to your clip board.

Now, go into your copy of the motion and search for "#region Auto-Generated".

```
 872
 873     Attack Styles
1197
1198     #endif
1199
1200     #endregion
1201
1202     Auto-Generated
1875         }
1876     }
1877
```

You'll want to delete and replace everything within that region with your new code.

7. The rest of the setup is the same as the sword. Meaning you'll want to create an "item" in the Basic Inventory that represents your Ax. Then, you'll create a Weapon Set that equips the ax.

With that done, you can now equip the ax and use the attack style you created the same way we do with swords.

Obviously this can get more advanced if you have ax-specific movement or jumps. However, this is a good primer for just swapping out different weapon types.

Movement

How do I make the movement speed faster?

If you want to use the root-motion, you can increase the speed of the animations in the Animator (specifically BasicWalkRunPivot-SM and BasicWalkRunStrafe-SM).

If you don't want to use root-motion, just change the "Walk Speed" and "Run Speed" properties of the motions. However, the more you move away from the animation's speed... the more your character will look like they are ice skating.

Camera

Can I use my own camera system?

Absolutely.

This asset doesn't control the camera at all. It does use the camera's direction to help aim and move, but that is simply a transform that I access.

You'll have to play around with the motion properties to get the kind of behavior that you're looking for.

Inventory

How do I create an Inventory Source for Inventory Pro?

The process is the same for any inventory asset, but I'll write these steps for Inventory Pro.

1. Create a new class and call it "InventoryProSource".

2. Use this code as a template and put it in your new class:

```
using UnityEngine;

namespace com.ootii.Actors.Inventory
{
    /// <summary>
    /// Inventory Pro Inventory Source for use with ootii.
    /// </summary>
    public class InventoryProSource : MonoBehaviour, IInventorySource
    {
        /// <summary>
        /// Some motions will use this to determine if they should test
        /// for activation or allow the inventory source to drive activation.
        /// </summary>
        public virtual bool AllowMotionSelfActivation
        {
            get { return true; }
```

```
        }

        /// <summary>
        /// Instantiates the specified item and equips it. We return the instantiated item.
        /// </summary>
        /// <param name="rItemID">String representing the name or ID of the item to equip</param>
        /// <param name="rSlotID">String representing the name or ID of the slot to equip</param>
        /// <param name="rResourcePath">Alternate resource path to override the ItemID's</param>
        /// <returns>GameObject that is the instance or null if it could not be created</returns>
        public virtual GameObject EquipItem(string rItemID, string rSlotID, string rResourcePath = "")
        {
            // Add Inventory Pro specific code;

            return null;
        }

        /// <summary>
        /// Instantiates the specified item and equips it. We return the instantiated item.
        /// </summary>
        /// <param name="rSlotID">String representing the name or ID of the slot to clear</param>
        public virtual void StoreItem(string rSlotID)
        {
            // Add Inventory Pro specific code;
        }

        /// <summary>
        /// Retrieves the item id for the item that is in the specified slot. If no item is slotted, returns an empty
string.
        /// </summary>
        /// <param name="rSlotID">String representing the name or ID of the slot we're checking</param>
        /// <returns>ID of the item that is in the slot or the empty string</returns>
        public virtual string GetItemID(string rSlotID)
        {
            string lItemID = "";

            // Add Inventory Pro specific code here;

            return lItemID;
        }

        /// <summary>
        /// Retrieves a specific item's property value.
        /// </summary>
        /// <typeparam name="T">Type of property being retrieved</typeparam>
        /// <param name="rItemID">String representing the name or ID of the item whose property we want.</param>
        /// <param name="rPropertyID">String representing the name or ID of the property whose value we want.</param>
        /// <returns>Value of the property or the type's default</returns>
        public virtual T GetItemPropertyValue<T>(string rItemID, string rPropertyID)
        {
            T lValue = default(T);

            string lPropertyID = rPropertyID.Replace(" ", string.Empty).ToLower();

            // Resource Path is used to grab the path to the prefab (string)
            if (lPropertyID == BasicInventory.Properties[0])
            {
            }
            // Instance is used to grab the pre-created item (GameObject)
            else if (lPropertyID == BasicInventory.Properties[1])
            {
            }

            return lValue;
        }
    }
}
```

3. Each function needs to be customized to work with Inventory Pro. I don't know Inventory Pro, but I would expect it has the concept of items and slots. Add code that performs the needed function.

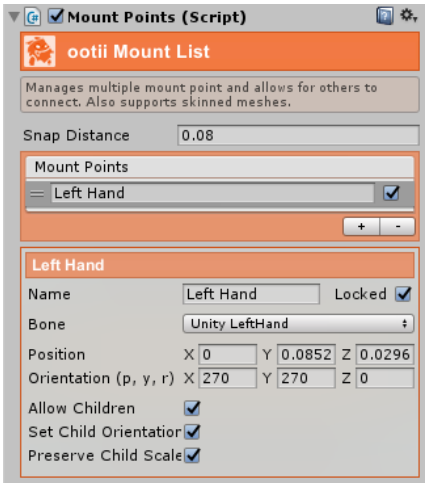
4. Once that's done, you can use Inventory Pro instead of my Basic Inventory solution.

Can I use Mount Points with the Sword and Sheild Motion Pack?

Absolutely.

Since I can't assume users of AMP will have Mount Points, I have to build it like they don't. So, there's a couple of things you need to do to enable Mount Points.

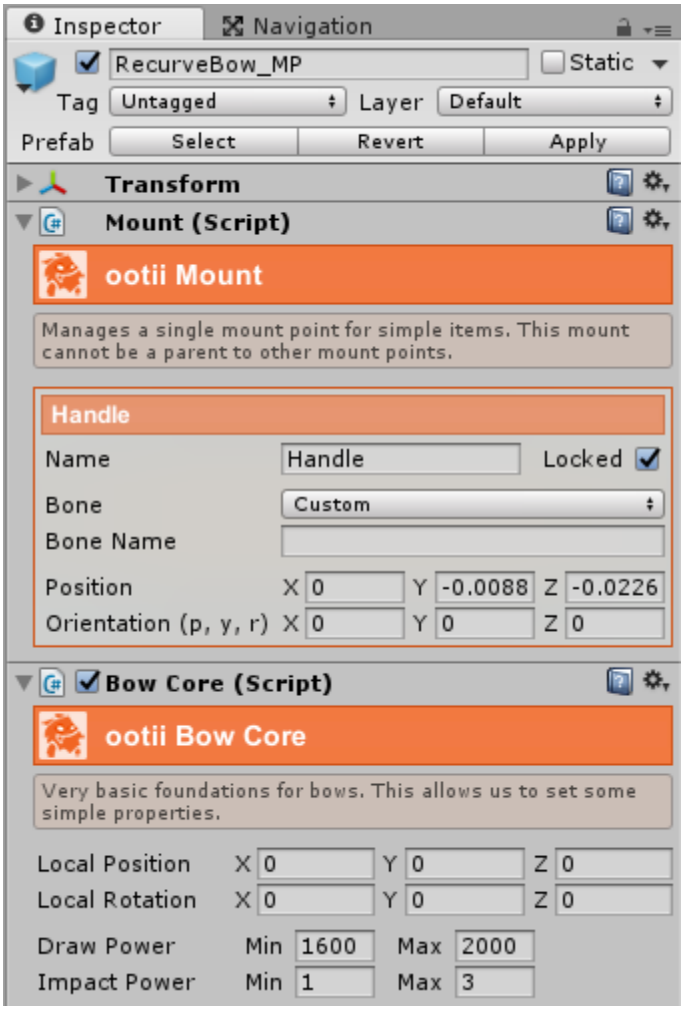
1. Place Mount Points on your character and setup a mount point for the right hand:



I use the name "Right Hand".

2. When setting up your inventory, you'll want to use a sword and shield resource that include a mount point. For example, I've included two prefabs: Sword_01 and Sword_01_MP. The second one includes a mount point.

I use the mount point name of "Handle".



When you do this, you won't be using the 'Local Position' and 'Local Rotation' properties of the Bow Core. Instead, you'll be using Mount Points.

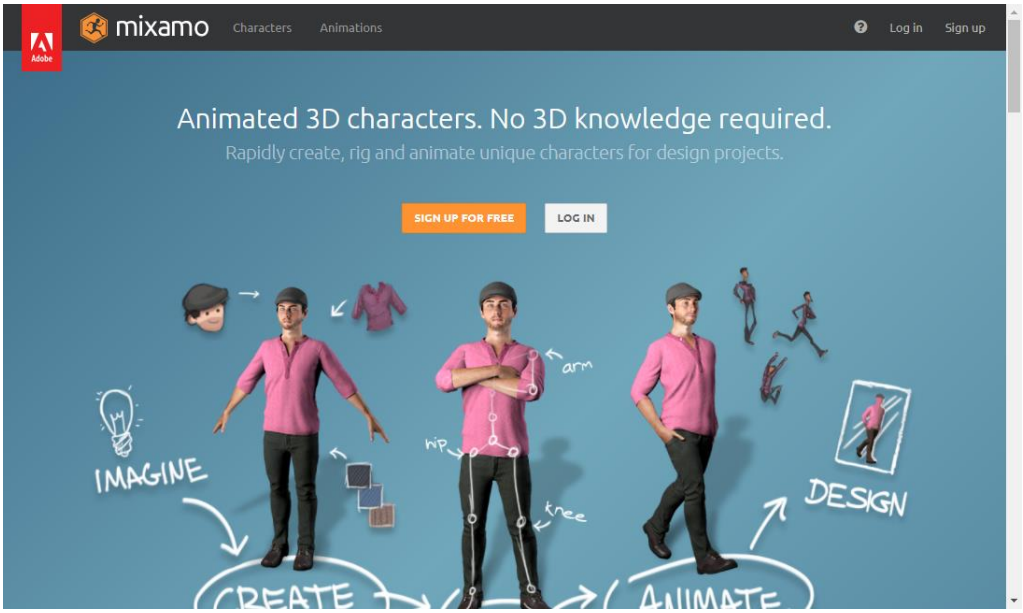
This is better as it handles different swords and different characters in a more generic, flexible, and consistent way. To learn more about Mount Points, check out its Asset Page on the Unity Asset Store.

Mixamo Animation Download

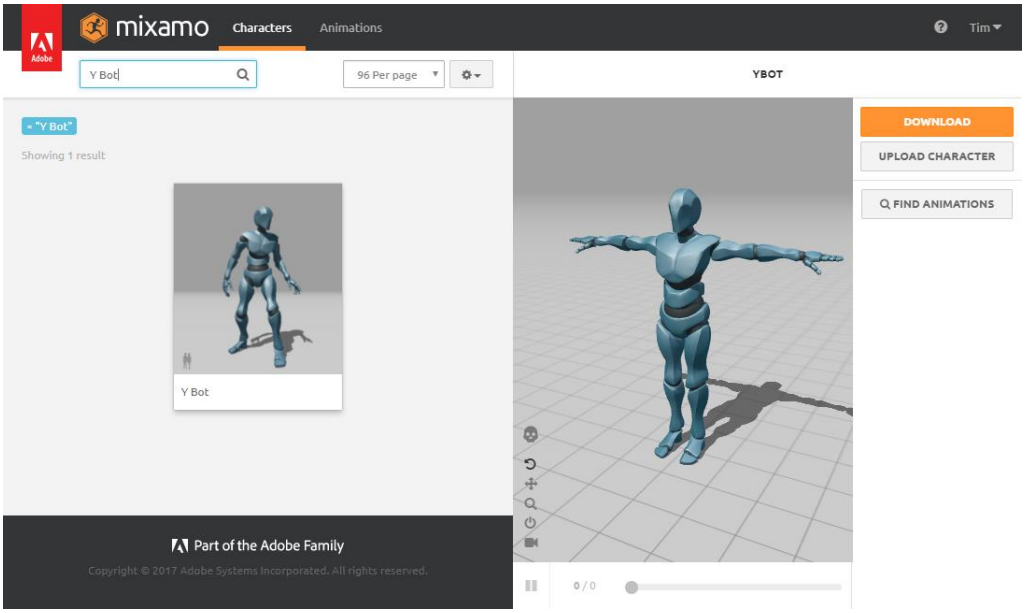
On August 23rd 2017, Mixamo updated their site and changed the flow. In addition, they change file names. This flow represents the updated process for getting the animations.

The following is a step-by-step approach on how to download the Mixamo animations for **Y Bot**.

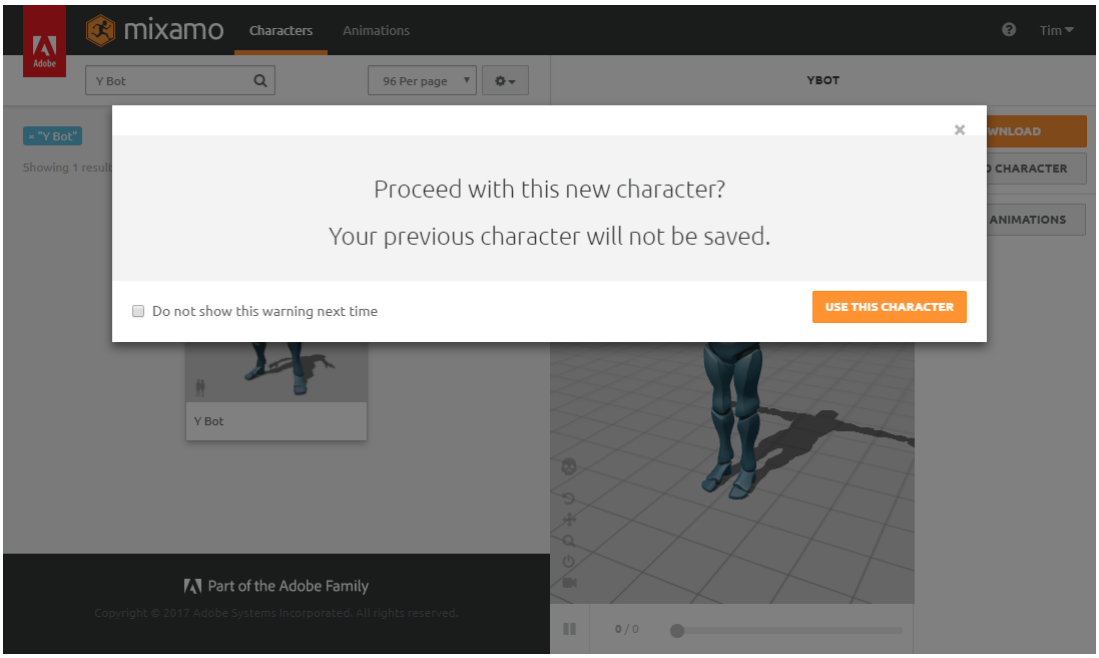
1. Go to www.mixamo.com and login



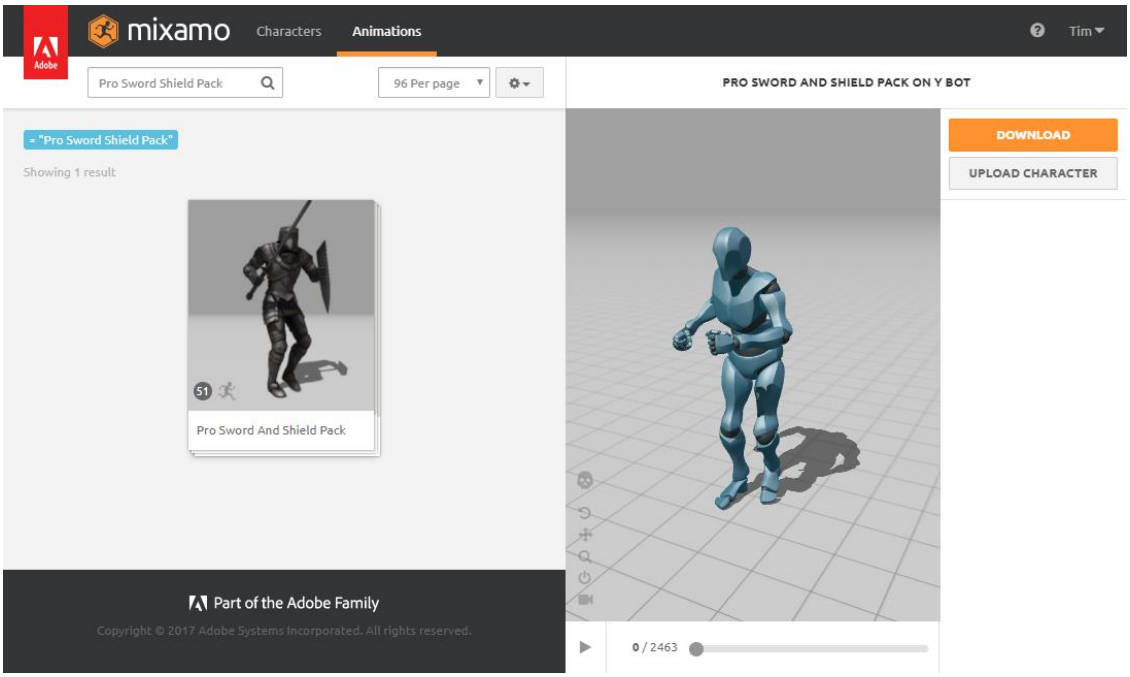
2. Go to the “Characters” link at the top and search for “Y bot”



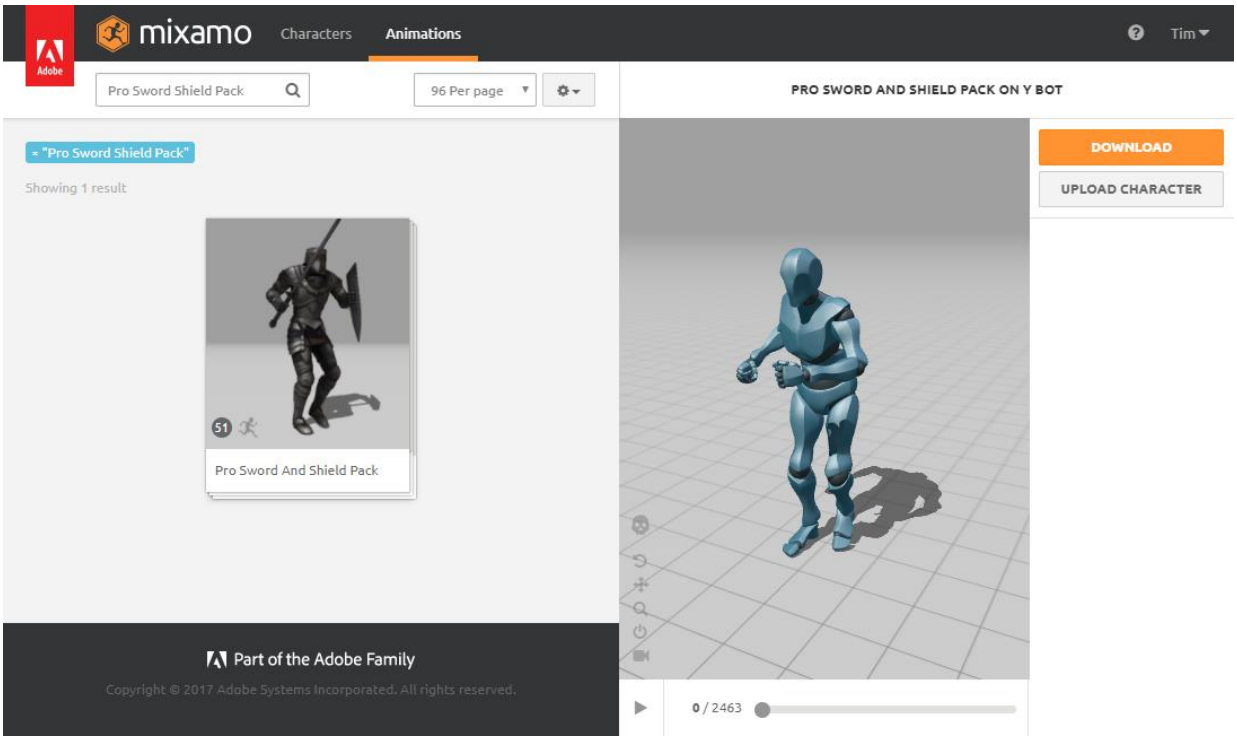
3. Select the “Y Bot” character and press “Use This Character”. This will make the Y Bot your default character for the selected downloads.



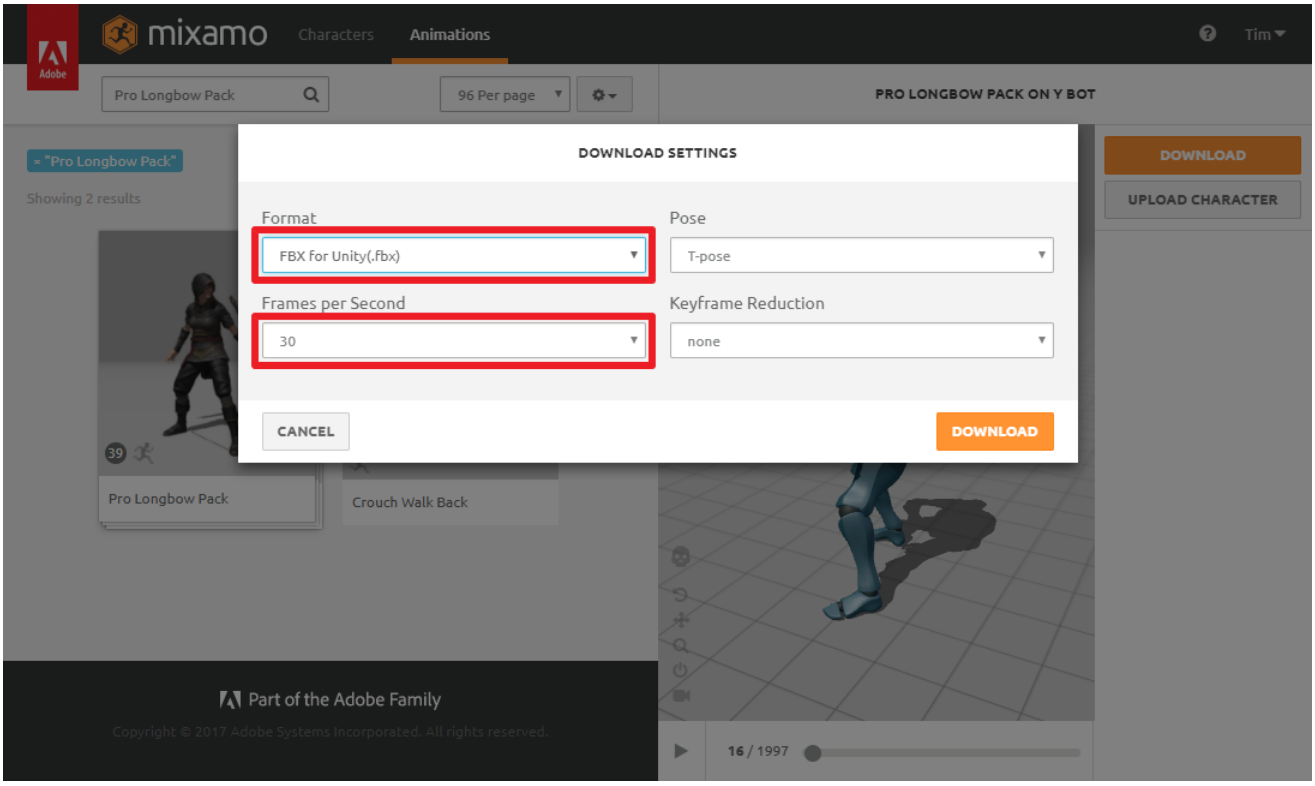
4. Go to the “Animations” link at the top and search for “Pro Sword Shield Pack”



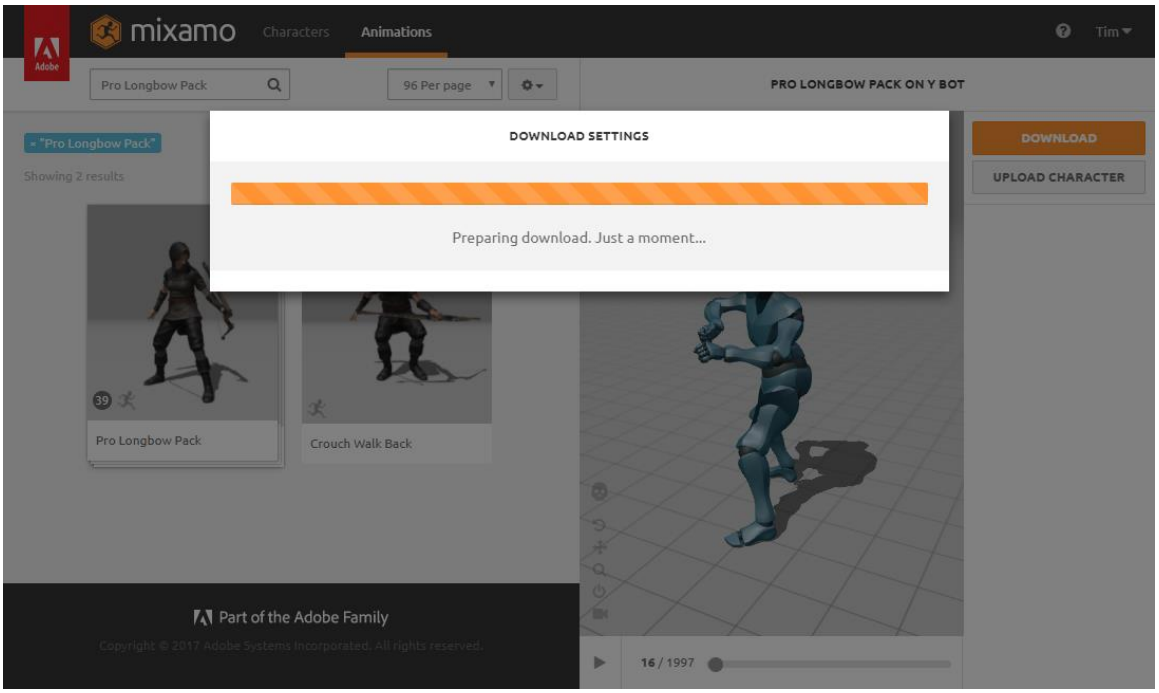
5. Click the Pro Sword And Shield Pack and press the “Download” button at the top right.



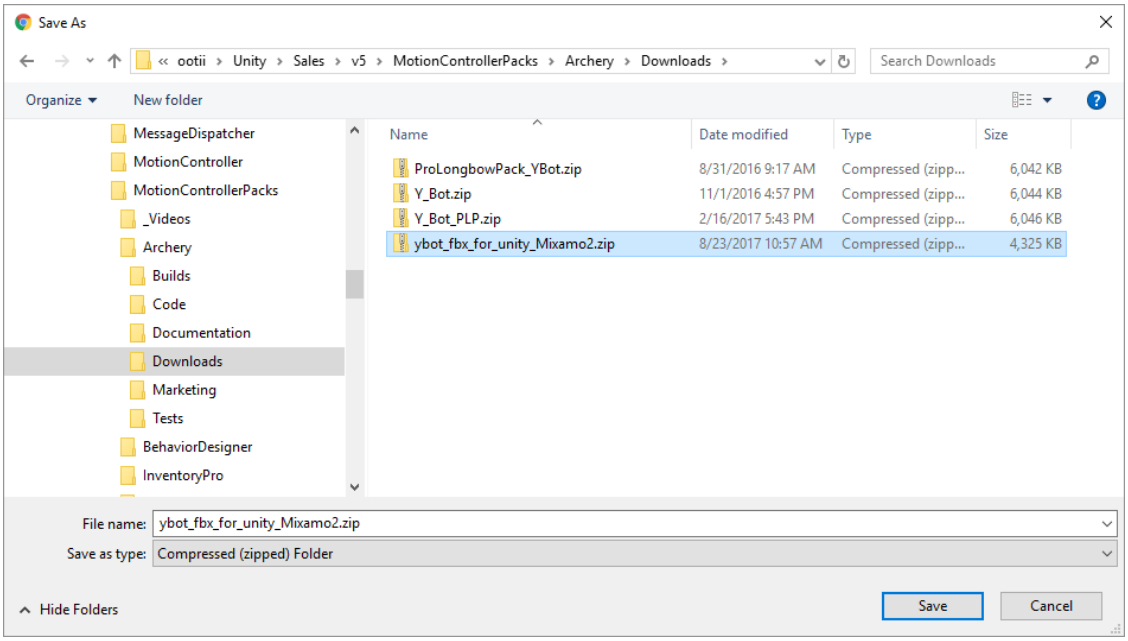
6. Select “FBX for Unity” and “30” Frames per Second. Then, press “Download”.



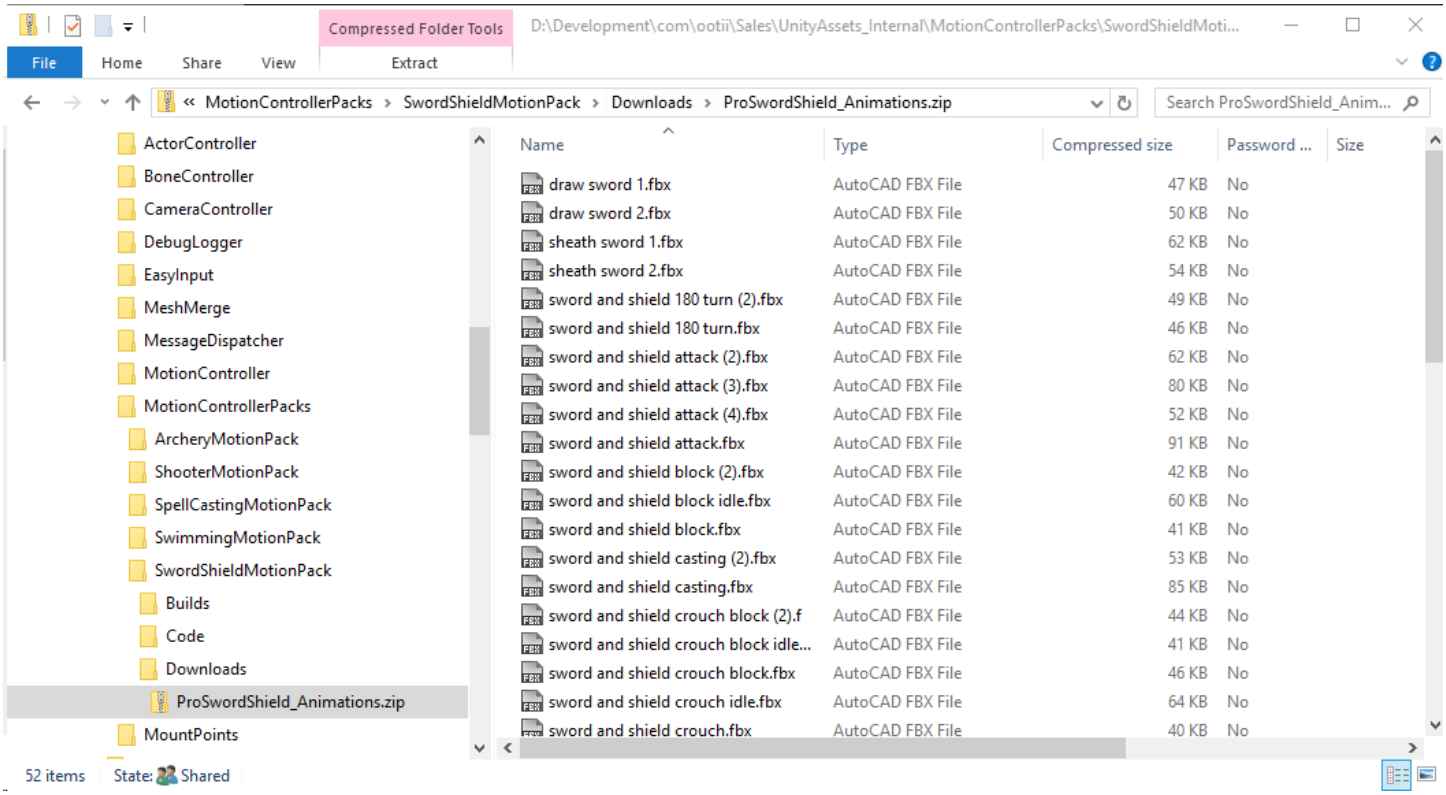
7. Wait for the processing to finish



8. Select where to store the animations



12. Unzip the contents to "...\Assets\ootii\Assets\MotionControllerPacks\SwordShield\Content\Animations\Mixamo"



NOTE: Your animation names may be different. For example, they may not include "Y_Bot@" or underscores. That's fine.

Animation & Meta Files

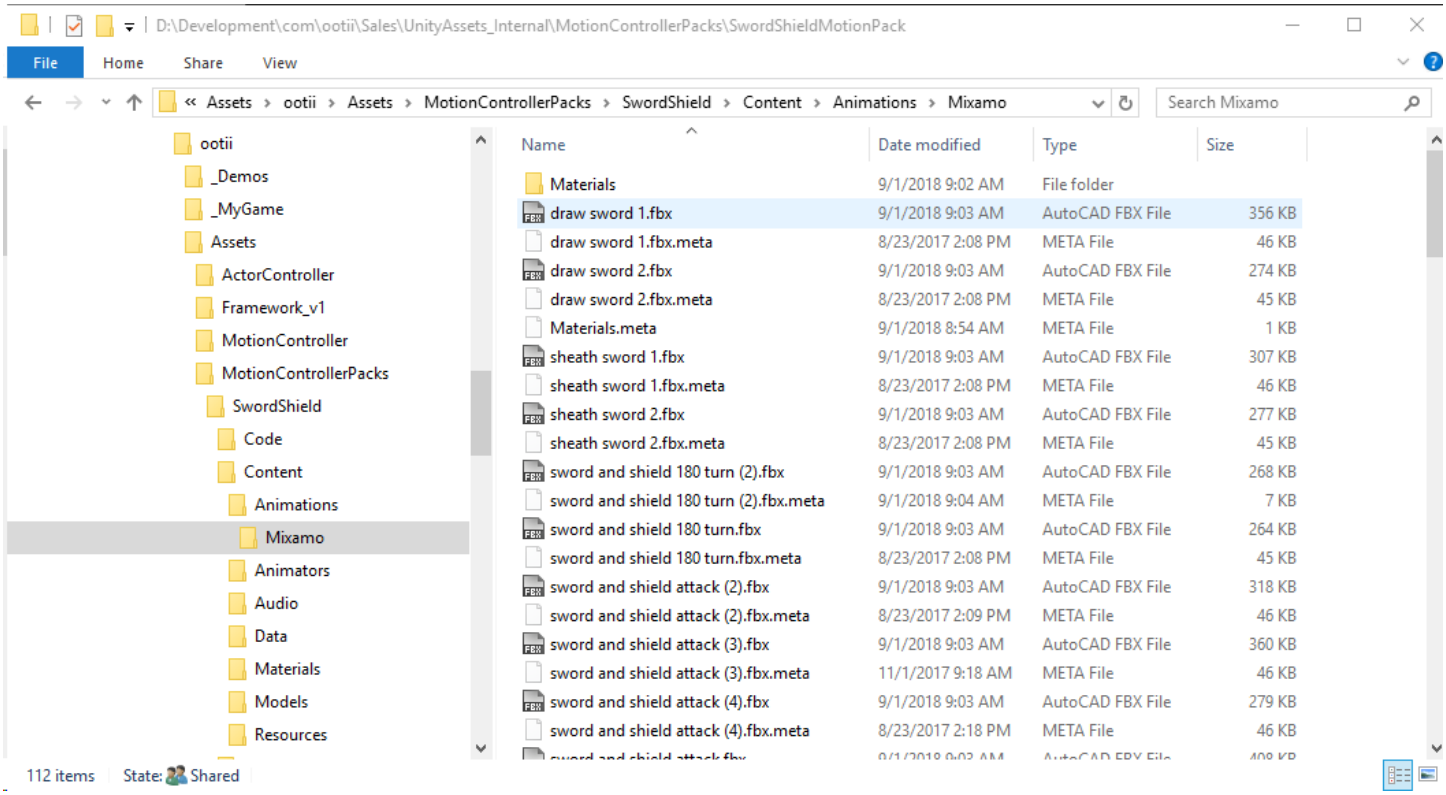
Assets\ootii\Assets\MotionControllerPacks\SwordShield\Content\Animations\Mixamo

When you first import the Sword & Shield Motion Pack, the Mixamo folder will only have a “Materials” folder in it (and the associated Materials.meta file).

Once you download the Mixamo animations, there will be 51 animation files + 1 character file. You’ll place them in that Mixamo folder.

If you click on Unity, it will create a meta file for each file. Instead, you want the meta files found in the Assets\ootii\Assets\MotionControllerPacks\SwordShield\Extras\AnimationMeta.zip file.

Extract, copy, and paste those 46 files into the Mixamo folder that we just put the animations (replace if needed). When you click on Unity again, it will reload the meta files. Your Mixamo folder will look like this:



You may get some warnings about .meta files missing. That’s fine. I include some extra .meta files for older users.

Finally, you will need to close Unity and re-open it. For some reason, Unity needs to do this to update the Animator with these animations.