RPG Character Mecanim Animation Pack ReadMe

Last Updated: **Jan 13, 2022**

TLDR Setup Video: https://youtu.be/ruufqlXrCzU

It is highly recommended to watch Unity's Animation Tutorial Videos before using this asset if you're not familiar with Mecanim or Animation Controllers: http://unity3d.com/learn/tutorials/topics/animation

Controller Overview

The RPG Character Mecanim Animation Pack controller includes several Unity components which control the character's position in the world, and its Mecanim animator. These are listed below and detailed further here.

Core Components

There are just two required components.

- `RPGCharacterController` - This is the main entry point for triggering animations. It is required by every other controller component.
- `RPGCharacterAnimatorEvents` - This component contains placeholder methods for animation events triggered by the animator. It is automatically attached to the game object with the Animator component at runtime.

Default Components

These are not required for the core controller to operate properly, but they implement important animations (like walking and running) and are highly recommended. They are included by default on the RPG-Character prefab.

- `RPGCharacterMovementController` - This component takes care of physics and drives the character in various movement states using SuperCharacterController.
- `RPGCharacterWeaponController` - This component handles all of the animations for wielding and sheathing weapons.

- `SuperCharacterController` - This component handles ground detection and surface interaction.

Demo Components

These scripts are optional.

- `RPGCharacterInputController` - This is a simple implementation of gamepad input and can be used wholesale or as a reference for defining your own input scheme.
- `RPGCharacterInputSystemController` - Alternative InputSystem control scheme for using Unity's new InputSystem. Contained within the InputSystem package in the project folder, and requires the InputSystem Package to be installed via the Package Manager.
- `RPGCharacterNavigationController` - This is a simple component which allows the character to move around using UnityEngine's NavMeshAgent.
- `GUIControls` - This is an example for the demo scene and not intended for your game, but it is a great reference for how to trigger nearly any action.
- `IKHands` - Automatically attached the left hand to two-handed weapons using IK.
- `PerfectLookAt` - This makes the character look at the selected target.

Deprecated Components

The following components were included in previous versions of this package but have been removed.

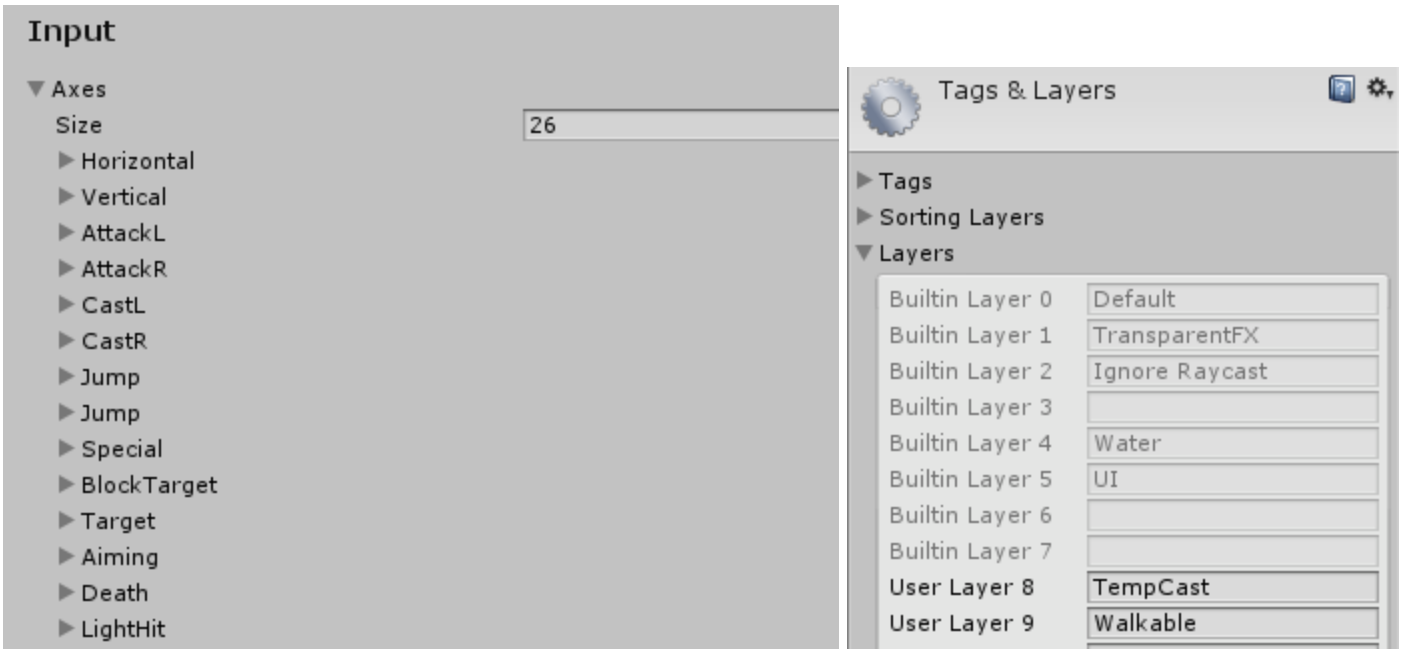
- `AnimatorParentMove` - This has been rolled into `RPGCharacterAnimatorEvents`.

Setup

Pre-Installation

If you are not using Unity's InputSystem, you must first ensure that the layers and inputs are correctly defined.

There is an included **InputManager.preset** and **LayerManager.preset** which contains all the settings that you can load in: https://docs.unity3d.com/Manual/Presets.html



InputSystem

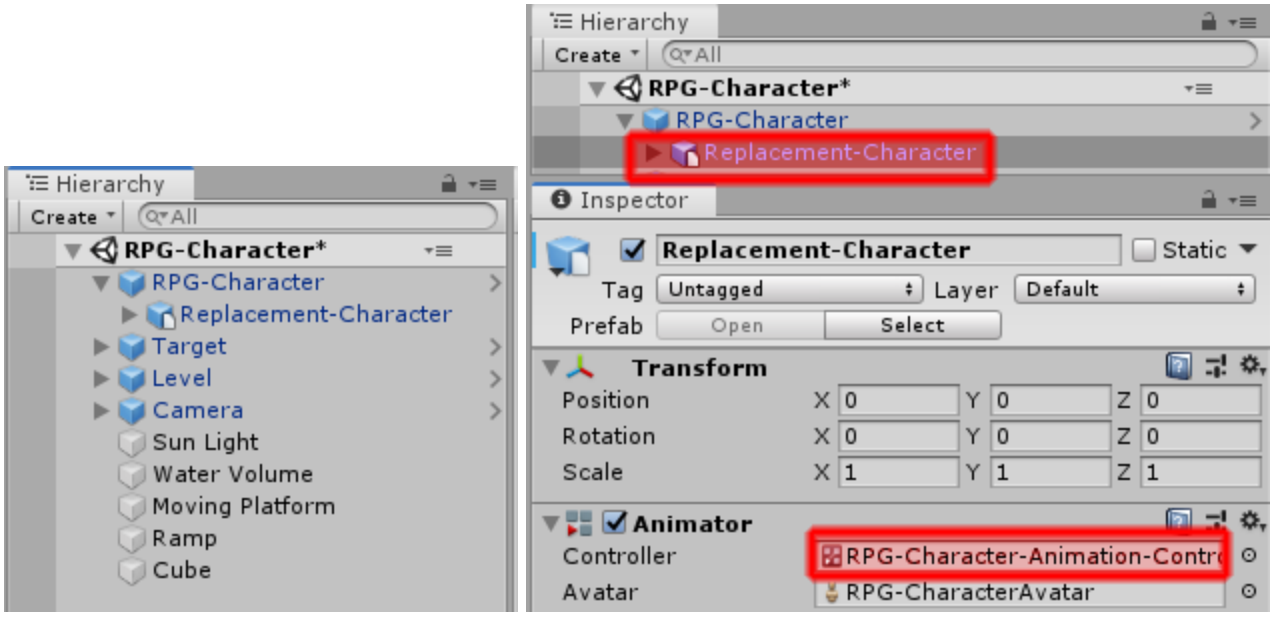
If you wish to use the RPG Character Mecanim Animation Pack with Unity's InputSystem, extract the 'InputSystem - Requires InputSystem Package' package in the project folder, and then replace the `RPGCharacterInputController` script on the prefab with the `RPGCharacterInputSystemController`.

See the Component API guide for more information.

Replace Character Model

Simply drag in your character model underneath the main RPG-Character prefab, and then set the Controller property of the Animator component to the RPG-Character-Animator-Controller in the Animation Controller folder.

The `RPGCharacterAnimatorEvents` script is automatically attached to your character's Animator component at runtime.

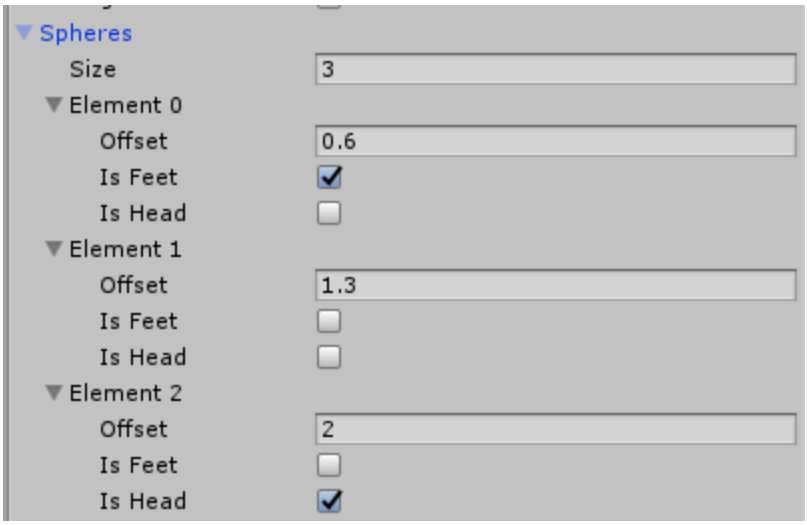


Set Target

The `RPGCharacterController` script needs a target object for purposes of targeting/strafing.

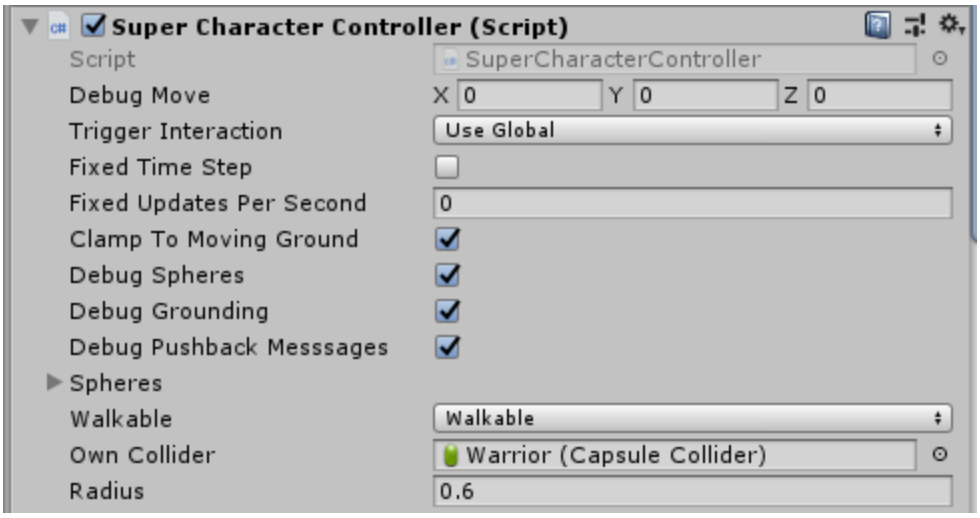
Adjust Collider and Super Character Controller Spheres

If needed, adjust the Capsule Collider for your character, and also adjust the `SuperCharacterController` Sphere's objects to proper size and position.



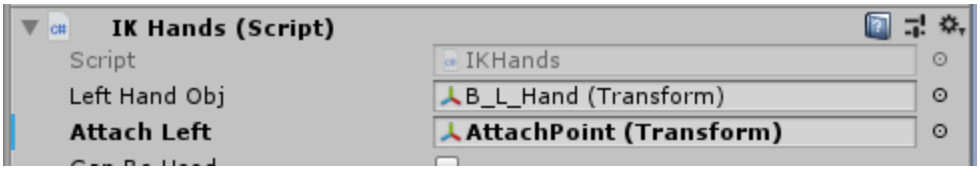
Set Super Character Controller Script Walkable Layer and Own Collider.

This will be set to Walkable by default, so be aware of that for your game's environment colliders.



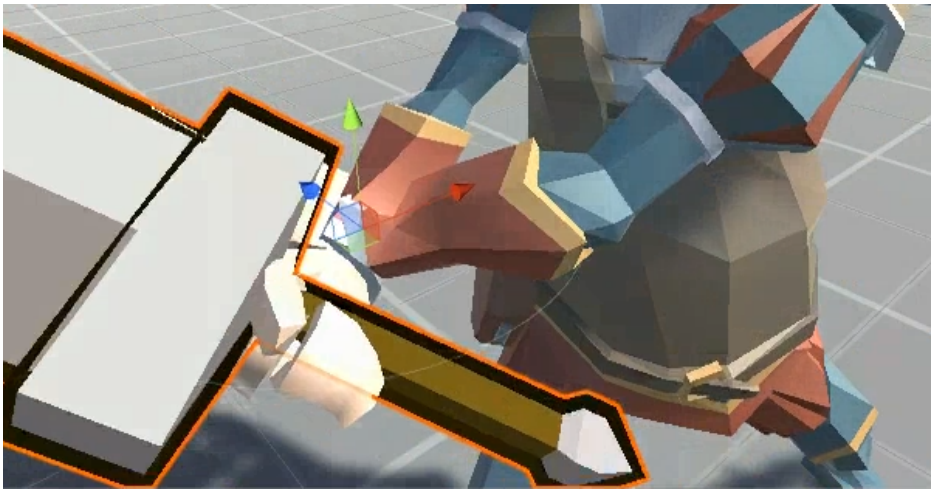
Set IK hand

If you're using `IKHands` script then you need to add it to your character models object with the Animator component, and set your character's left hand joint as the script's **Left Hand Obj**. The **Attach Left** parameter will be automatically connected at runtime.



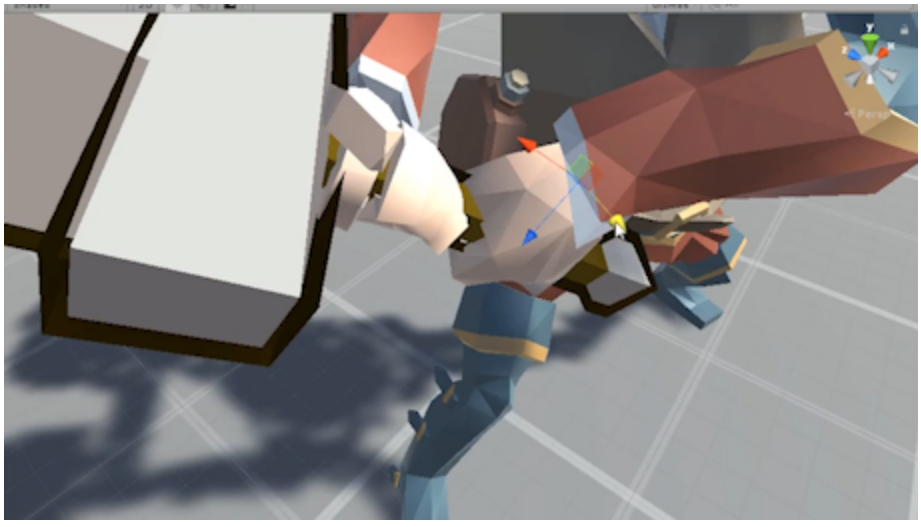
Position/Scale Weapons in Hand.

This is recommended to finalize during runtime so you can see how the weapon is positioned in the character's deformed hand, and then copy the transform and re-paste it back when the game is not running.

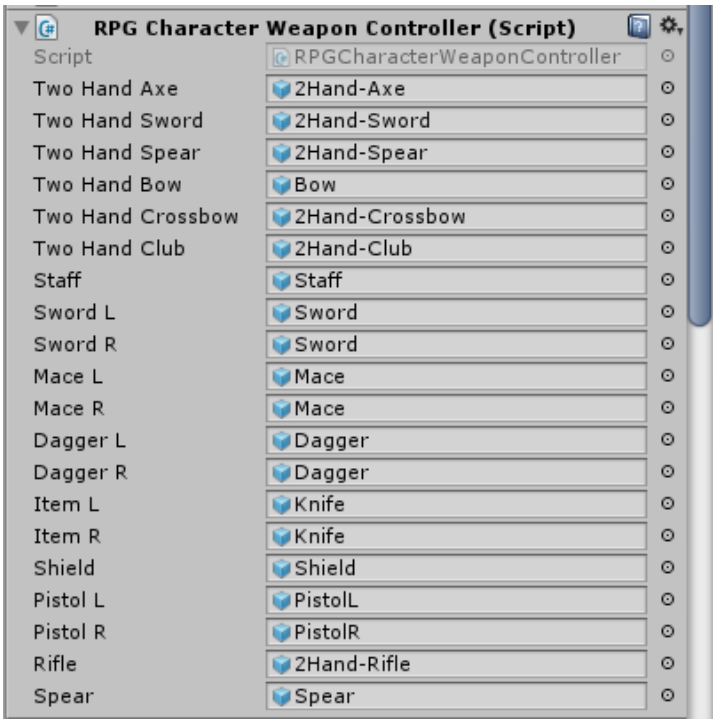


Position/Rotate Attachment Points.

The same process is followed for placing the weapons in the characters hands. Position the **Attach** point, which will be the first child of the weapon game object, and then copy the transform when the game is not running.

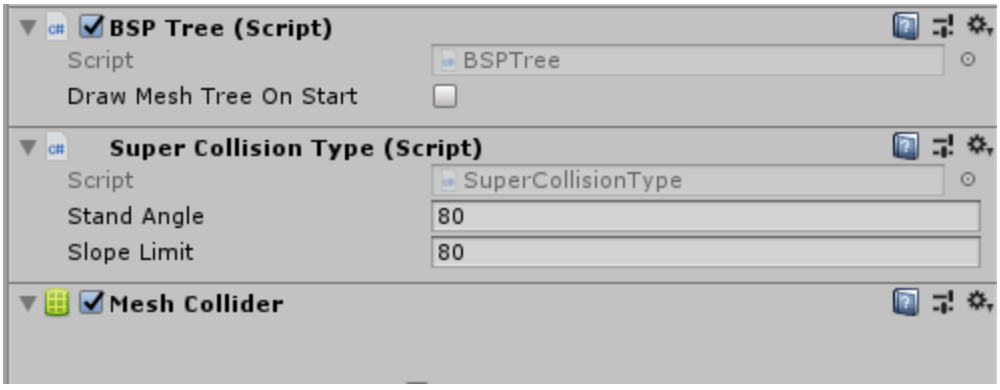


Move and/or Replace Weapons on New Character and Copy Weapon Parameters



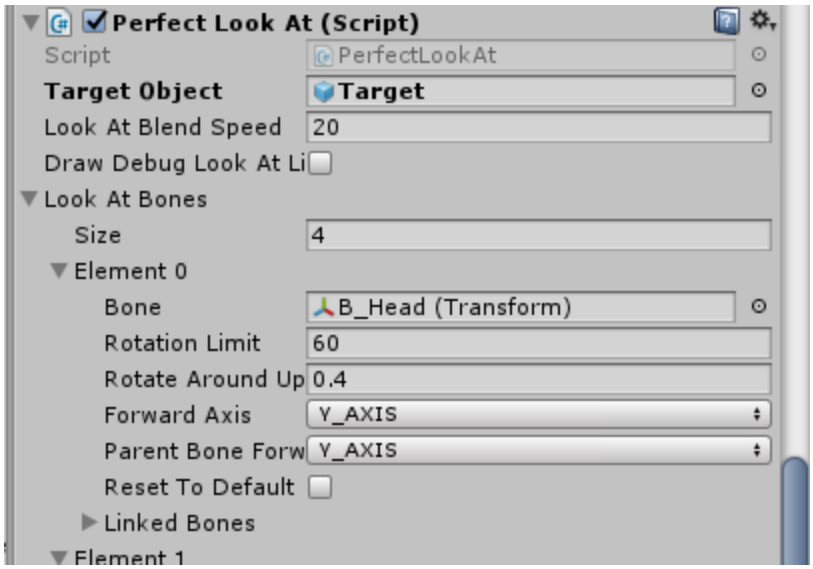
Setup World Colliders

For any objects that you want the character to walk over or collide with, set them as Layer: **Walkable**, and make sure this is set the same in the `SuperCharacterController` script. Objects with primitive colliders on them such as sphere, box, or capsule colliders won't need any additional settings, but any object with a *Mesh Collider* needs the `BSPTree` script attached to it. If you want to control the allowable slope height for the object, you can attach the `SuperCollisionType` script to it as well.



Change LookAt bones and Target.

If you're planning to use the `PerfectLookAt` script then you'll need to set the Bone fields under the Look At Bones properties.



Animation Events

The RPG Character Animations contain Animation Events that need a receiver. In the example setup, the `RPGCharacterController.cs` automatically finds the child with the Animator component, and attaches the `RPGCharacterAnimatorEvents.cs` script, which contains receivers for all the Animation Events.

If your setup doesn't use the `RPGCharacterController.cs`, you'll need the following methods attached to a script on the same object as your character's Animator:

```
public void Hit()
{
}

public void Shoot()
{
}

public void FootR()
{
}

public void FootL()
{
}
```

```
public void Land()
{
}
```

These can be empty methods, but they must be there to receive the Animation Events. If you don't have them you'll see errors like:

```
AnimationEvent 'FootR' on animation 'Unarmed-Run-Forward' has no receiver! Are you missing a component?
```

Actions

Animations in this package are organized around the Actions API. Each action is an entrypoint to one or more animations. Because Unity's Mecanim Animation system is a state machine with predefined transitions, not all animations are available at any given time–you can't shoot a bow while swimming, or climb a ladder in the middle of a jump (...yet).

You can provide additional information to an action by passing a context object. Many actions require no context–nodding the characters head "Yes" doesn't take much–but some actions are complex enough that they require several values every time. Do not fret, every action is documented, and there are many examples in the `GUIControls` and `RPGCharacterInputController` files.

A simple example

Consider the following code snippet which can be used to have the character block attacks.

```
bool blocking = Input.GetKey(KeyCode.B);
RPGCharacterController controller = GetComponent<RPGCharacterController>();

if (blocking && controller.CanStartAction("Block")) {
    controller.StartAction("Block");
}
else if (!blocking && controller.CanEndAction("Block")) {
    controller.EndAction("Block");
}
```

`Input.GetKey` checks if the B key is pressed on that frame and stores it in a boolean value named ***`blocking`***. If the key is pressed, ***`blocking`*** is set to true, otherwise it is set to false. Once we know whether the player intends to block (or stop blocking), we can use actions to get the character into the right state.

All actions live in the `RPGCharacterController` component, so we store this component in a variable named **`controller`** using Unity's `GetComponent()` method. `RPGCharacterController` has a few important methods which are used here. All of these methods take a string as their first argument tells the controller which action we care about. In this example, we've passed the string "Block" to all four methods.

- `CanStartAction()` returns true when it's possible for the animator to enter this action state. In this case, the Block action can start only when it hasn't already started. We don't want the character to continually raise their shield in a loop, we want them to do it once and hold the block.
- `StartAction()` actually starts the action, unsurprisingly.
- `CanEndAction()` returns true when it's possible for the animator to exit this action state. Like a light switch, the Block action can be turned off only when it's already on.
- Finally, `EndAction()` exits the action state.

Putting it all together: if the player presses the B key and the controller can start the block, then start blocking. If the player has released the B key and the controller can end the block, then end blocking.

NOTE: Savvy code investigators will notice in `ActionHandler` that `EndAction()` won't start unless `CanStartAction()` also returns true.

A more complex example

Some actions require additional context in order to start. Constructing the context object is an extra step, but triggering the action uses the exact same methods on `RPGCharacterController`. Let's look at the `Attack()` action.

```
// using RPGCharacterAnims.Actions;

bool attack = Input.GetKeyDown(KeyCode.Space);
bool kick = Input.GetKeyDown(KeyCode.Enter);
RPGCharacterController controller = GetComponent<RPGCharacterController>();
AttackContext context;
```

Actions and their context objects are defined in the **`RPGCharacterAnims.Actions`** namespace, so we add a **`using`** statement in order to access the `AttackContext` class. The input logic here is similar to the previous example. We want to attack with a weapon if the player presses the spacebar, and kick if the player presses enter. Unlike blocking, where we care if the key is held down or released, we only want to trigger the action in the frame when the key is pressed, so we use `Input.GetKeyDown()`. We also store the `RPGCharacterController` component in the `controller` variable, just like before. Finally, we create a placeholder **`AttackContext`** object called `context`, which we will use below.

Next up is our code to trigger the attacks.

```
if (controller.HasRightWeapon && attack) {
    context = new AttackContext("Attack", "Right");
    controller.StartAction("Attack", context);
}

if (kick) {
    context = new AttackContext("Kick", "Left", 1);
    controller.StartAction("Attack", context);
```

```
}
```

This animation package supports an unarmed character throwing a punch, but in this example we want to attack only when the character is holding a weapon. We can use the controller's ***HasRightWeapon*** property to see if the character is carrying a weapon in their right hand. If so, we can create a new **AttackContext** object and pass it to `StartAction()`.

AttackContext takes up to three parameters. The first is the attack ***type***, and in this case we are using "Attack" to indicate that this is a regular attack. The second parameter is the ***side***, which is either "None", "Left", "Right", or "Dual". We are attacking with the right hand weapon, so we pass "Right" to the **AttackContext** constructor. The final parameter is the attack ***number***, which is an integer representing the animation number. If the number parameter is omitted as it has been here, the action chooses an animation at random.

The second example is similar, except we pass "Kick" as the type, "Left" as the side, and we specify 1 as the animation number. This context will play the same animation every time.

Things you can do with Action Handlers

All actions live in the Code/Actions directory in this package, and every action implements the **IActionHandler** interface, usually by inheriting from the **BaseActionHandler** class. **IActionHandler** requires that all actions implement the `CanStartAction()`, `StartAction()`, `CanEndAction()` and `EndAction()` methods that we have already seen.

Is this thing on?

All actions have an `IsActive()` method which can tell you if the action has started. You can access this method via the controller's `IsActive()` method by passing it the name of the action.

```
RPGCharacterController controller = GetComponent<RPGCharacterController>();
bool blocking = controller.IsActive("Block"); // returns true or false
if (blocking) {
    // do something related to blocking
}
```

Adding event listeners

You can listen for an action to start and end by using `AddStartListener()` and `AddEndListener()`. As an example, the `RPGCharacterWeaponController` component uses listeners to hide all weapons when the character begins swimming.

```
RPGCharacterController controller = GetComponent<RPGCharacterController>();
Actions.IActionHandler swimHandler = controller.GetHandler("Swim");
swimHandler.AddStartListener(HideAllWeapons);
```

Simple actions

Many actions don't require explicit class definitions in order to get work done, either because they have no special requirements for turning on or off, or they don't pass any specialized contextualized data. Some actions are so simple that no work is performed at all! They function simply as flags on the controller.

For these cases we use a `SimpleActionHandler()` object, which takes two methods in its constructor: one for when the action starts and one for when it ends. Here's a handful of simple handlers defined in `RPGCharacterController`.

```
SetHandler("Block", new Actions.SimpleActionHandler(StartBlock, EndBlock));
SetHandler("Relax", new Actions.SimpleActionHandler(() => { }, SetUnarmed));
SetHandler("Sprint", new Actions.SimpleActionHandler(() => { }, () => { }));
```

`Block()` has two methods, `Relax()` has one method and one empty method, and Sprint is simply a flag, with two empty methods.

Instant actions

Sometimes it doesn't make sense for an action to stay "on" after it is started. For example, with the `Dodge()`, `Turn()`, and `Reload()` actions, the character can trigger the action again as soon as the animation is finished. Under the hood, these are simply actions which end immediately after they start. Instant actions can always end (though nothing happens, except for triggering listeners) and they are never considered active.

Nearly all of the actions for movement defined in `RPGCharacterMovementController` are instant actions. This is because the movement controller uses `SuperCharacterController` to handle physics, and the character is always in one of the movement states. This is also why an action like `Sprint()` can simply be a flag. It doesn't have any side effects of its own, but the movement controller checks the Sprint action when deciding what speed the character should move.

Creating your own

Most actions are very simple to implement. Take a look at the `MySlowTime()` action which changes Unity's time scale. The `MySlowTime()` class inherits from the `BaseActionHandler()`, a generic class with a type parameter that defines the type of context. `MySlowTime()` expects a **`float`** when this method starts, so it must inherit from `BaseActionHandler<float>`.

```
using UnityEngine;
using RPGCharacterAnims;
using RPGCharacterAnims.Actions;

public class MySlowTime : BaseActionHandler<float> { }
```

Action classes must implement `CanStartAction()` and `CanEndAction()` which determine when the action can start and end. In this case, `SlowTime()` can start when it's not active, and can end when it is active.

```
public override bool CanStartAction(RPGCharacterController controller)
{
    return !IsActive();
}

public override bool CanEndAction(RPGCharacterController controller)
{
    return IsActive();
}
```

Action classes must also implement `_StartAction()` and `_EndAction()` which are called when the action starts and ends. These methods are wrapped by already implemented `StartAction()` and `EndAction()` (without the underscore) which also takes care of some housekeeping like triggering the event listeners we saw before and setting the ***active*** variable. You can see that the **float** type from BaseActionHandler<float> is reflected here as the type of the context parameter.

```
protected override void _StartAction(RPGCharacterController controller, float context)
{
    Time.timeScale = context;
}

protected override void _EndAction(RPGCharacterController controller)
{
    Time.timeScale = 1f;
}
```

This action can now be used with `RPGCharacterController`. You can set this handler on the controller like this. Notice that the action name “SlowTime” can be different from the class name.

```
// set the SlowTime handler
controller.SetHandler("SlowTime", new MySlowTime());
```

Then, you can start the action as before.

```
// slow time to half speed
controller.StartAction("SlowTime", 0.5f);
```

Questions, comments, requests or suggestions:

Contact.