Documentation

Terminology

Action State

Definition:
An action state is the minimal state that describes a character's action. It includes locomotion actions like moving, jumping, dashing, or any combat action like swinging a sword or firing a gun. It is the basic unit that adds up to the entirety of an action system in PAT. Action State by itself has very limited functionality, it is mostly a logic system that shows which state a character is in (what it is doing), and what it can do from the current state.

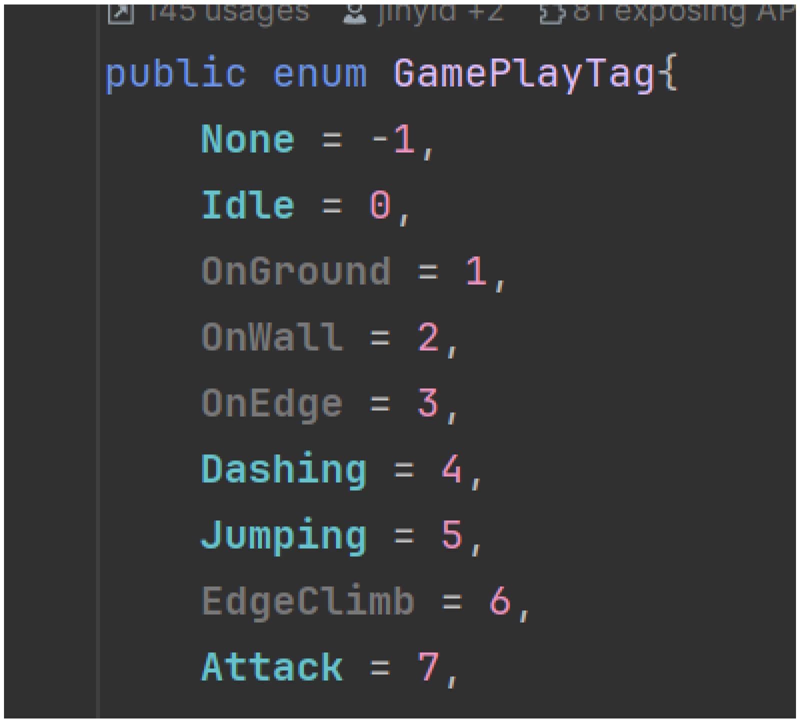
State Modifier

Definition:
State modifiers are attached to Action States. They are triggered during the state and grants different functionality to Action States. For example, controlling animations, activating hitboxes, modifying movement speed, etc. Most modifier scripts are named as xxx Mod, where Mod stands for MODifier. Each Modifier has a Begin and an End Timing Attribute, that defines when will this mod get activated, as well as the duration of it.

Tag

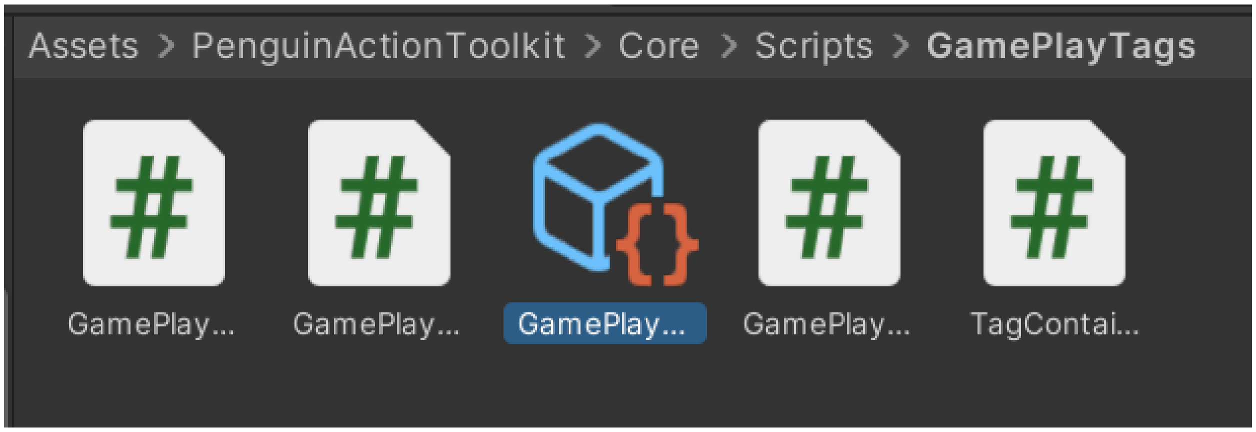
Definition:
Tag system (not to be confused with Unity Tags) is essential to PAT. Action state's transition logic is defined by Tags, and that creates a dynamic state machine system. Beyond that, some other components also use Tags for communication purposes. For example, you can make a character invincible by granting it invincible Tag and Hurtbox will react to it. Attributes are also using Tags to identify themselves.

All tags are stored inside a script named GamePlayTag, you may edit it on your own need. It is a huge enum. Every Tag is represented by a name and a number.

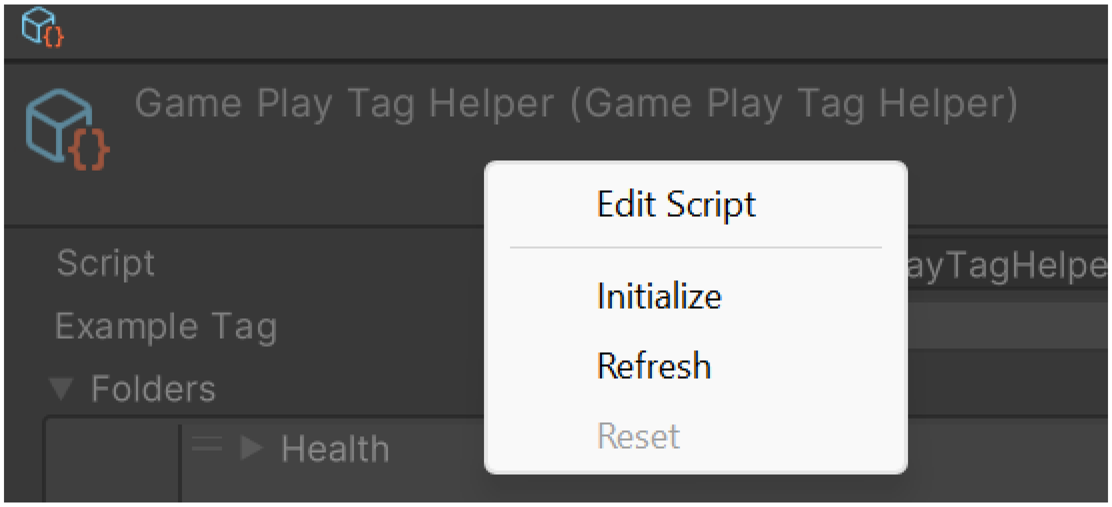


```
145 usages   jinyid +2   81 exposing AP
public enum GamePlayTag{
    None = -1,
    Idle = 0,
    OnGround = 1,
    OnWall = 2,
    OnEdge = 3,
    Dashing = 4,
    Jumping = 5,
    EdgeClimb = 6,
    Attack = 7,
```

You may also find the scriptable object named GamePlayTagHelper for non-script view.


Assets > PenguinActionToolkit > Core > Scripts > GamePlayTags
GamePlay...
GamePlay...
GamePlay...
GamePlay...
TagContai...

You have to right click in the inspector and choose “refresh” for your changes to be reflected. Any new tags added in the script will be put into “Other”. You need to manually copy & paste it into the folder you like.



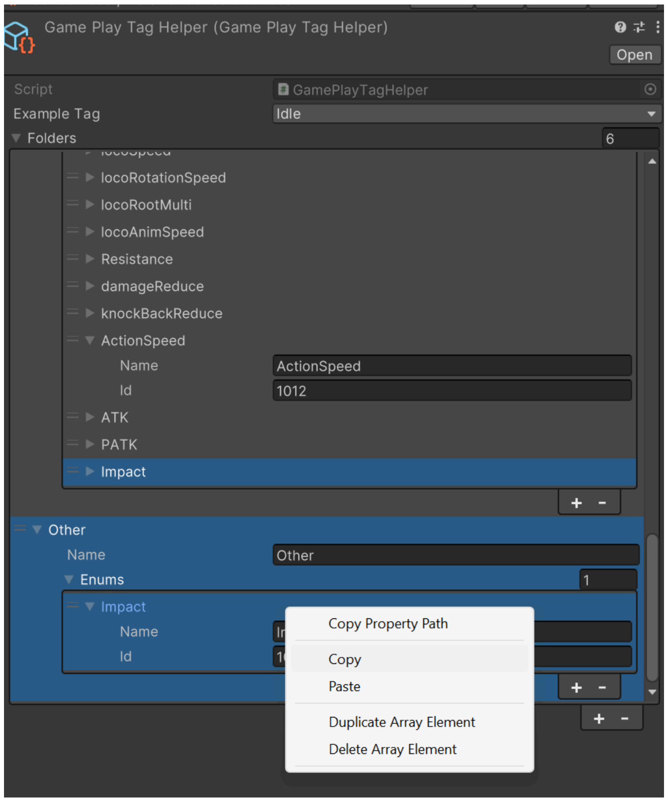
Game Play Tag Helper (Game Play Tag Helper)

Script

Example Tag

Folders

Health



Game Play Tag Helper (Game Play Tag Helper)

Script GamePlayTagHelper

Example Tag Idle

Folders 6

locoRotationSpeed
locoRootMulti
locoAnimSpeed
Resistance
damageReduce
knockBackReduce
ActionSpeed

Name ActionSpeed

Id 1012

ATK
PATK
Impact

Other

Name Other

Enums 1

Impact

Name

Id

Tags are toggled on the Action State component.

Inspector Details

Action State

(All following description assumes the described action state named as **State A**)

Input Tag:
Signals to trigger the State A. For player controlled characters, it is decided by the Input Units on Player Component. [PICTURE HERE] You can assign tags to player input, and everytime the character receives an input, it will try to trigger states using that specific tag.

Can Repeat:
If checked, State A can re-enter itself if other conditions are satisfied (details below). Otherwise, re-entering the same state is prohibited by default.

Auto Exit Time:
State A will auto exit after this given amount of time. By default it will enter Idle state. Or you may assign a specific Next State. If set to 0, it will not try to auto exit.

Tags:

Main Tags:
The Tags a character will hold if it is in State A

Require Tags:
After receiving the corresponding Input Tag, State A will check if the character is currently holding the Required Tags before entering. If not, nothing will happen.

The relationship among outer Elements is logical disjunction (or). As long as one element has all of its requirements fulfilled, this is considered to be true and the State can be entered.

The relationship among inner Elements is logical conjunction (and). All requirement tags must be present for it to be considered true, or the no requirement is checked.

Prohibit Tags:
If another State tries to enter when character is in State A, but at least one of its Main Tags is listed in State A's Prohibit Tags, then it cannot be entered

Self-Prohibit Tags:
If the character is in another State and is currently holding any Tag listed in State A's Self-Prohibit Tags, State A cannot be entered.

Special:
Permit State:
Regardless of the prohibit / require relationship of tags, any State listed here is allowed to enter during State A.

Next State:
Force character to enter the Next State after the exit of State A instead of idle. Notice that this only applies when State A is exiting by itself, not interrupted by the other state

Modifiers (Mods)

Mod Name:
A name for you to track what this modifier is for.

Timing Attribute:

Controls when this modifier will be activated.

Mode:
By Time in State:
An Action State will count the time after it is entered. Modifiers in this mode will be triggered on Begin Time, and stop functioning after End Time.

By Animation Event:
You may add Trigger Animation Events to any animation clip through Unity. You must input an index to it in the inspector. Modifiers in this mode will be triggered when the animation attached to the current state has passed Begin Index, and will end on End Index. Please align indexes in increasing order for proper functioning. Note that Action States must have an animation for modifiers using this mode.

For both Modes, a Begin Time / Index of -1 means the modifier will be triggered as soon as the State is entered. An End Time / Index of -1 means the modifier will not be turned off until the State is exited.

Events: apart from the functionality of modifiers themselves, you can add additional functionality by using the begin & end unity event exposed in the inspector. And a vanilla state modifier is solely for that. But be aware that Unity events are not easy to maintain and debug, please don't use that for your core logic.

Some Common Modifiers in package

Grant Tag Mod:
When the character is in an Action State, it will hold all tags listed in the State's Main Tag. Through this mod, it allows an Action State to have extra Tags for further State transition.

Animation Montage Mod:
Tries to play an animation state in the model's animator, and uses crossfade to create transitions.
Please note that this only works with animator that structures in PAT's template
If you need your own way to implement the animator, you might want to have your own version of Animation Modifier

Info:

State Name:
Name of the State that contains the desired Animation state inside the Animator.

Fade Time:
The duration of crossfade when the animation start

Fade out Time:
Fade out duration, and the target is a state called "Empty" in the same layer. This is helpful because the new state might be playing animation in a different layer

Layer:
The layer index of the animation state in the animator (an increasing integer starts from 0).
[Picture]

Keep Play on Exit:
The animator will not try to fade to "Empty" automatically when state end

Exit on Montage End:
Action State will auto exit if the Unity Animation Play Time reaches 1, helpful for many actions like attack, roll

Move Set:

Auto-Collect:
Collect all the states attached to its child objects at run time, so you don't need to do that in inspector

Creating new Mods

To write your own State Modifiers, you need to extend the State Modifier in your new script.

```
public class AnimationMontageMod: StateModifier
```

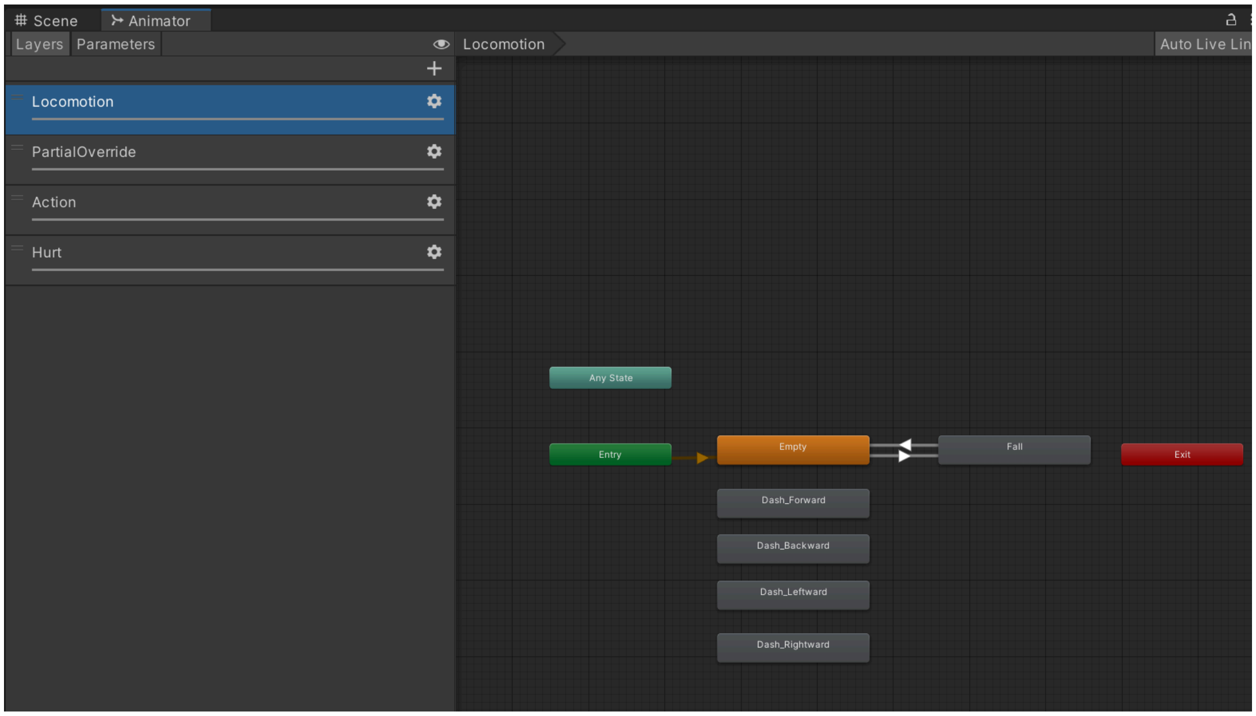
Typically, there are two functions you need to override: BeginEvent & EndEvent. BeginEvent is what happens when the State Modifier is triggered, while EndEvent is when the State Modifier is exited.

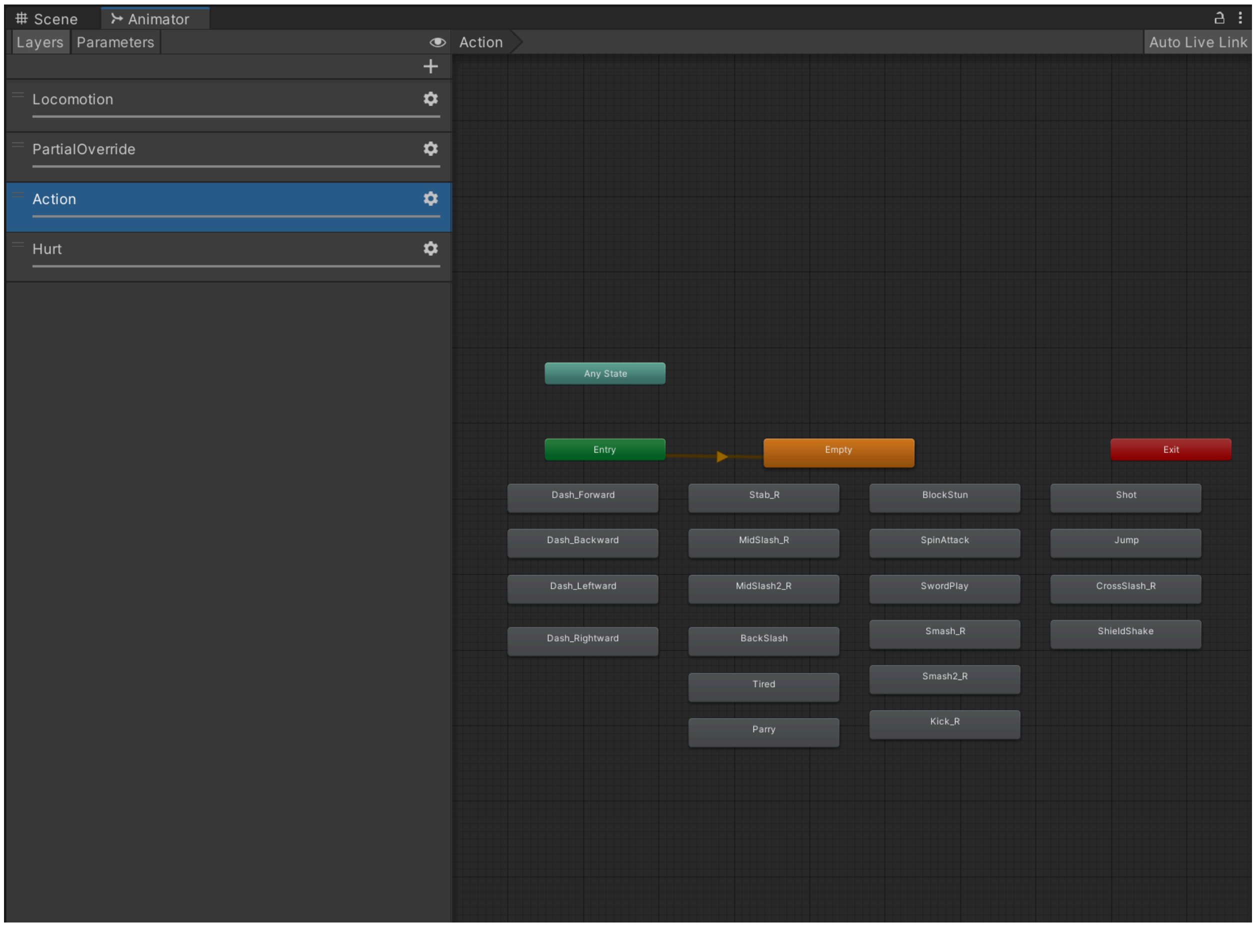
```
public override void BeginEvent()
{
    base.BeginEvent();
}

public override void EndEvent()
{
    base.EndEvent();
}
```

If you want to use a State Modifier to decide whether the character should enter the action state (ex. We did that in UseAttributeMod), you might also override Validate().

Animation

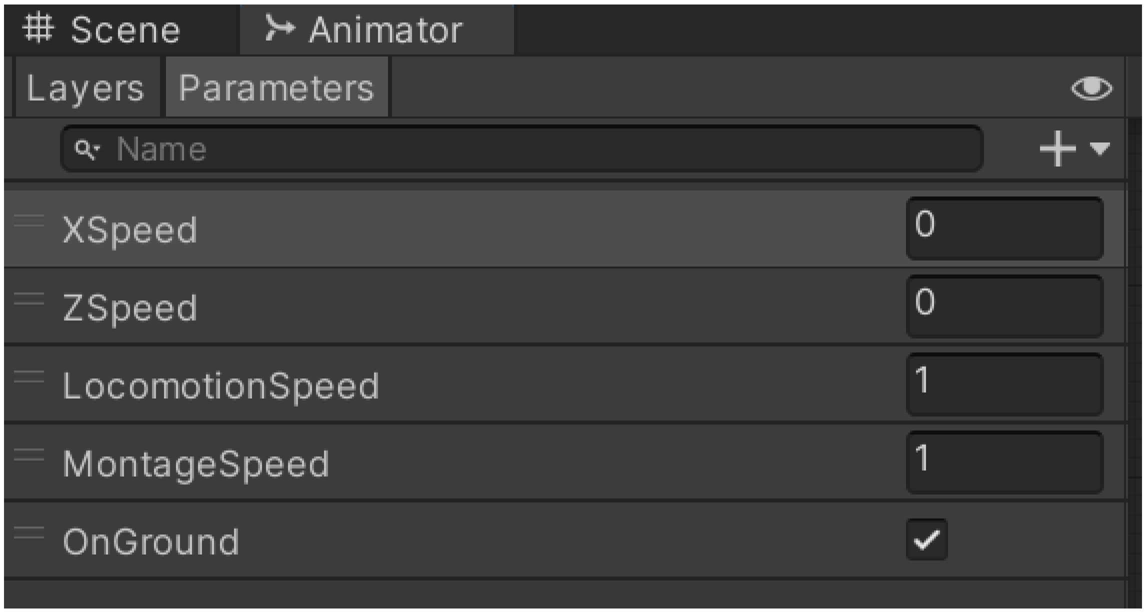
In PAT, the transitions among different animation states is controlled by Animation Montage Modifiers attached to each Action State. It keeps the animator clean and understandable. Your animator should look like the pictures below:





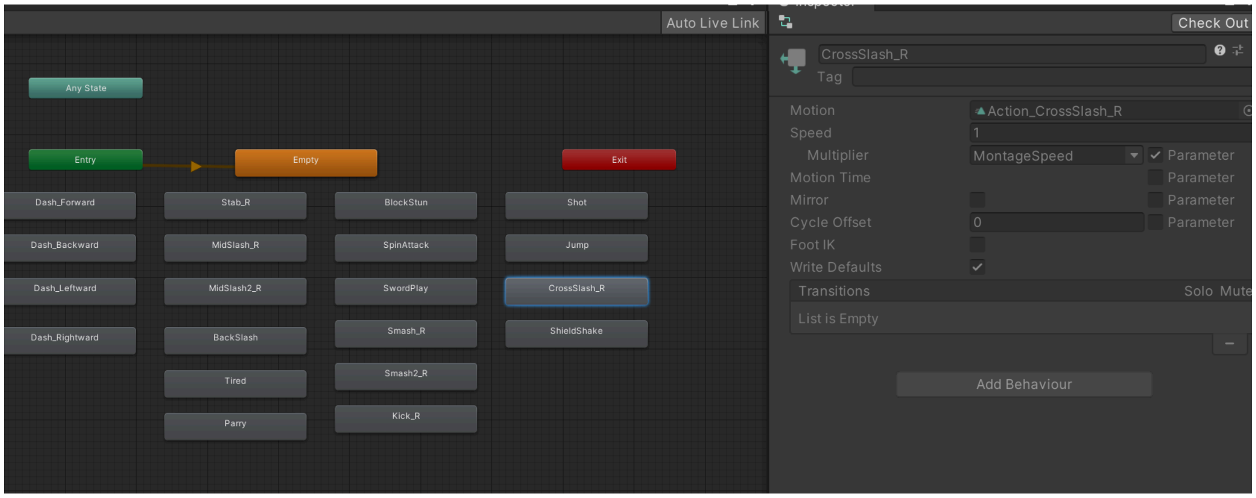
We recommend you categorize animation clips under different layers. Each layer has an index, starting from 0.

On the parameter page, you may want to introduce parameters like MontageSpeed, which allows you to use MontageSpeedMod to control the speed of your animation.



XSpeed	0
ZSpeed	0
LocomotionSpeed	1
MontageSpeed	1
OnGround	✓

You may choose the animation state and put in your speed multiplier.



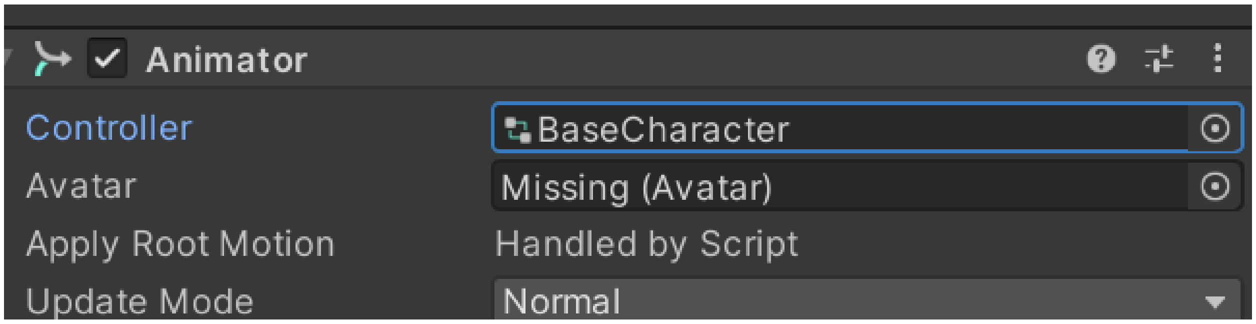
CrossSlash_R
Tag

Motion	Action_CrossSlash_R	
Speed	1	
Multiplier	MontageSpeed	✓ Parameter
Motion Time		Parameter
Mirror		Parameter
Cycle Offset	0	Parameter
Foot IK		
Write Defaults	✓	

Transitions — Solo Mute
List is Empty

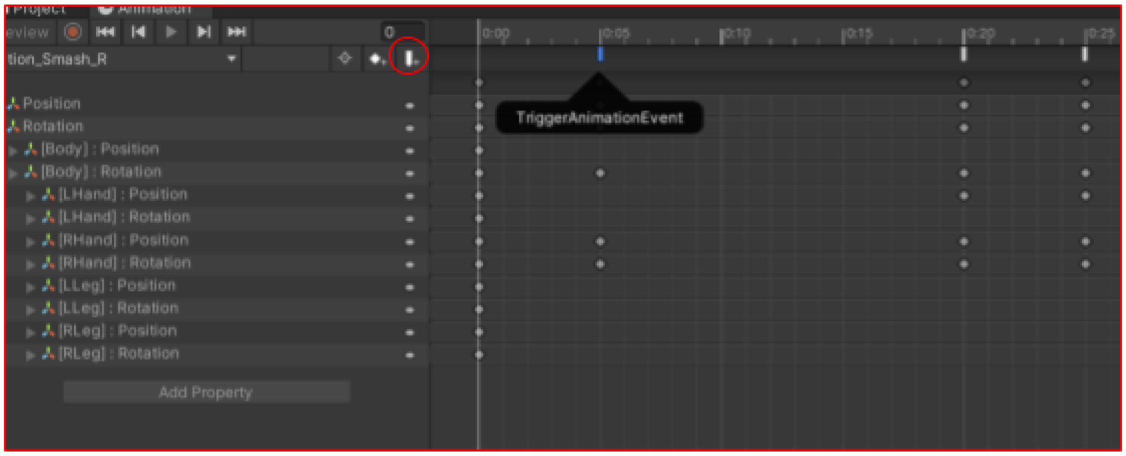
Trigger Animation Event:

If you are trying to control your state modifiers by animation events, you will need to add Model Handler to where your animator is at. We recommend you open the animation by clicking on the animator you attached to your character. If you directly click on the animation clip, the inspector view will be more confusing due to Unity's own issue.

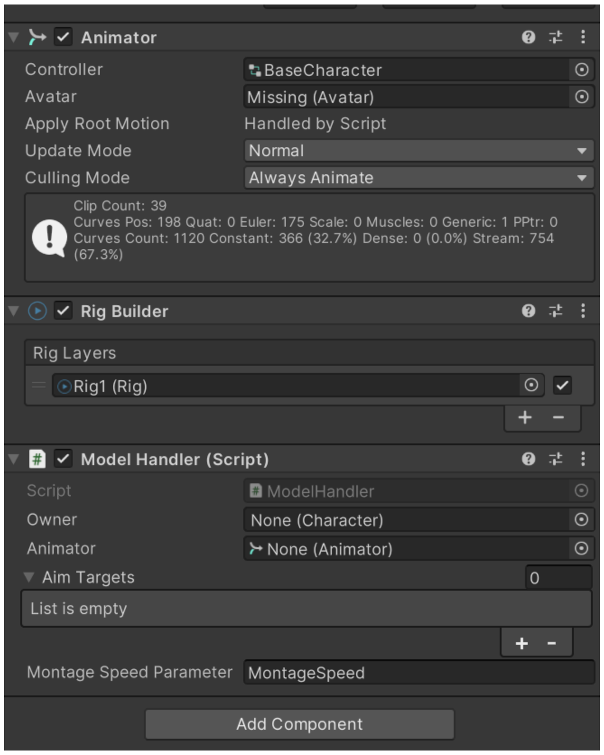

Animator

Controller	BaseCharacter
Avatar	Missing (Avatar)
Apply Root Motion	Handled by Script
Update Mode	Normal

After selecting an Animation Clip, you may use the circled button to add a trigger animation event to an animation clip. This is used for modifiers to track their begin and end timing.



If you click onto the events, you may find this in the inspector. You should choose TriggerAnimationEvent from Model Handler. You need to have a Model Handler on the same object as the Animator. After that, you can set Int which is the index we use for the modifier's timing.It should be in ascending order on the timeline


Animator
Controller: BaseCharacter
Avatar: Missing (Avatar)
Apply Root Motion: Handled by Script
Update Mode: Normal
Culling Mode: Always Animate

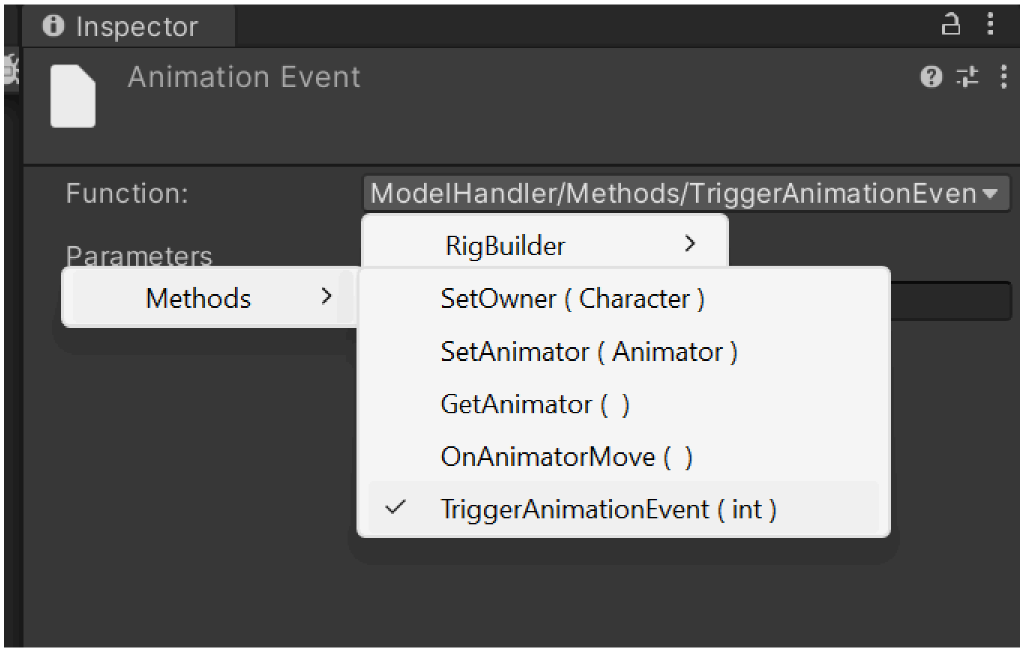
```
Clip Count: 39
Curves Pos: 198 Quat: 0 Euler: 175 Scale: 0 Muscles: 0 Generic: 1 PPtr: 0
Curves Count: 1120 Constant: 366 (32.7%) Dense: 0 (0.0%) Stream: 754
(67.3%)
```

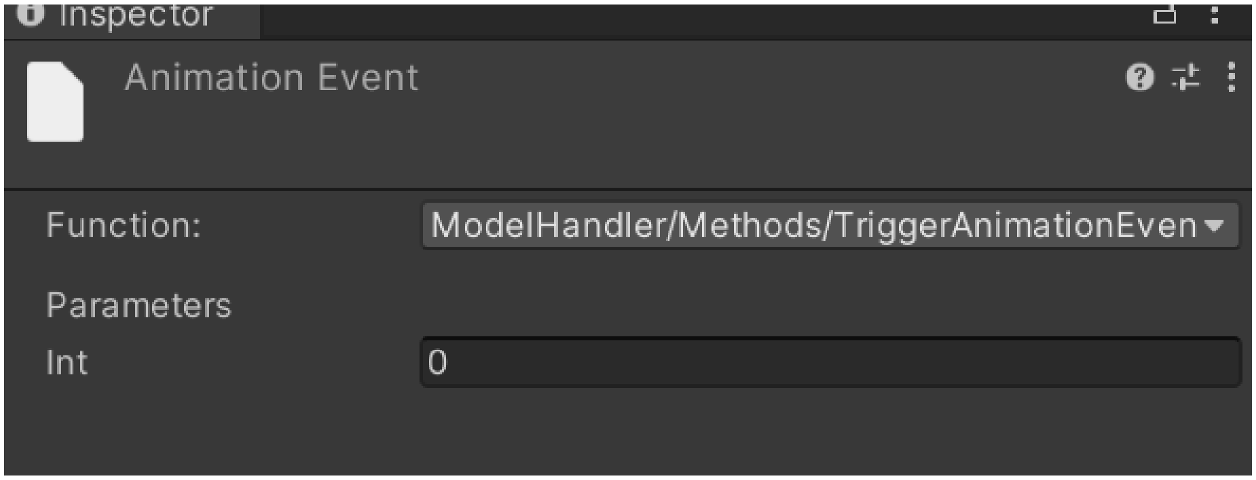
Rig Builder
Rig Layers
Rig1 (Rig)

Model Handler (Script)
Script: ModelHandler
Owner: None (Character)
Animator: None (Animator)
Aim Targets: 0
List is empty
Montage Speed Parameter: MontageSpeed

Add Component

.


Inspector
Animation Event
Function: ModelHandler/Methods/TriggerAnimationEven
Parameters
Methods
RigBuilder
SetOwner (Character)
SetAnimator (Animator)
GetAnimator ( )
OnAnimatorMove ( )
✓ TriggerAnimationEvent (int)



Inspector

Animation Event

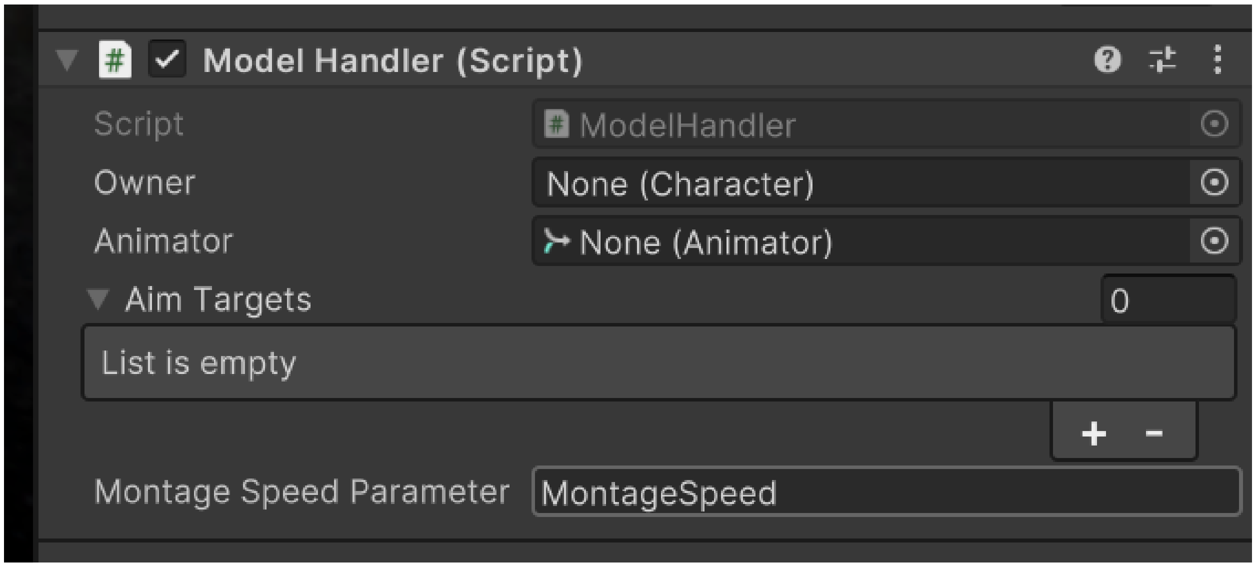
Function: ModelHandler/Methods/TriggerAnimationEven

Parameters

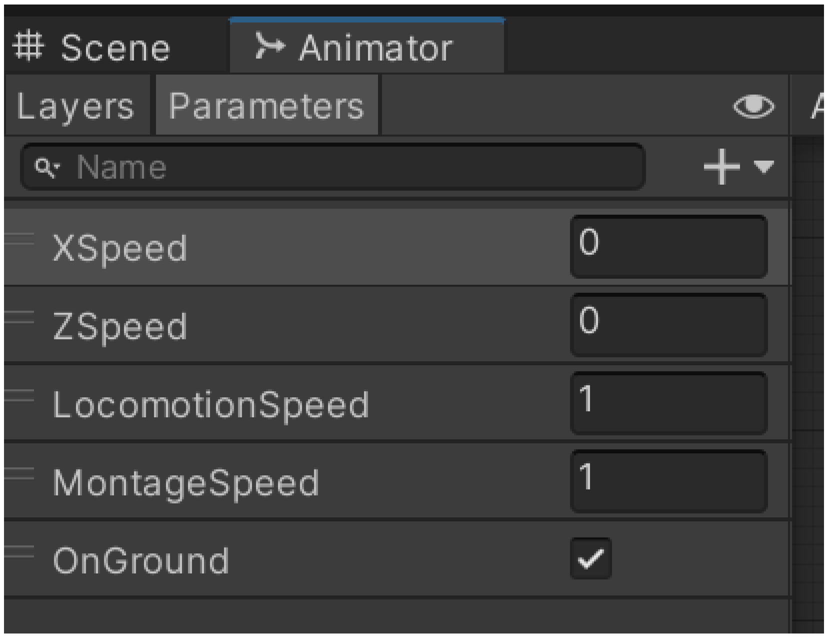
Int 0

Animation speed within each action state:

We have a mod named Montage Speed Mod. You need to have a Model Handler on the same object with Animator (or manually drag in your Animator). The Montage Speed Parameter is a string that should match the speed parameter inside your animator.

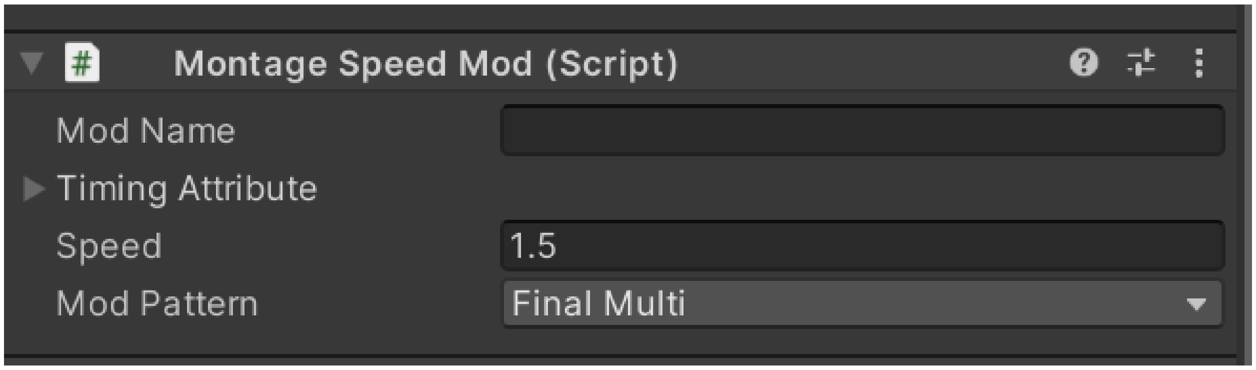

Model Handler (Script)

Script ModelHandler
Owner None (Character)
Animator None (Animator)
Aim Targets 0
List is empty
Montage Speed Parameter MontageSpeed



XSpeed 0
ZSpeed 0
LocomotionSpeed 1
MontageSpeed 1
OnGround ✓

Under a specific Action State, you can add a Montage Speed Mod


Montage Speed Mod (Script)

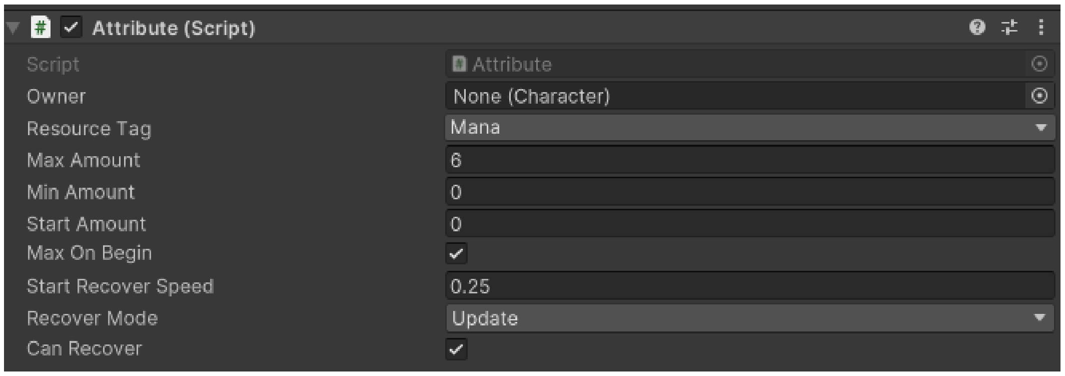
Mod Name
Timing Attribute
Speed 1.5
Mod Pattern Final Multi

Speed is the value that will replace montage speed when the modifier is applied. We don't recommend changing the Mod Pattern unless you are certain about it.

Attribute

Attribute Definition

An Attribute is a MonoBehaviour that holds a value. Health, Posture, Mana, and Stamina are all examples of Attributes. Since Attributes can be influenced by Effects in an easy and stable way without tightly coupling your code, we encourage you to use Attributes for any values you want to change dynamically during gameplay.


Attribute (Script)

Script	Attribute
Owner	None (Character)
Resource Tag	Mana
Max Amount	6
Min Amount	0
Start Amount	0
Max On Begin	✓
Start Recover Speed	0.25
Recover Mode	Update
Can Recover	✓

Resource Tag:
The identifier for the Attribute. Effects and Use Attribute Mod use this to find their target Attribute.

Max Amount:
The maximum value the Attribute can hold.

Min Amount:
The minimum value the Attribute can hold.

Start Amount:
The starting value of the Attribute.

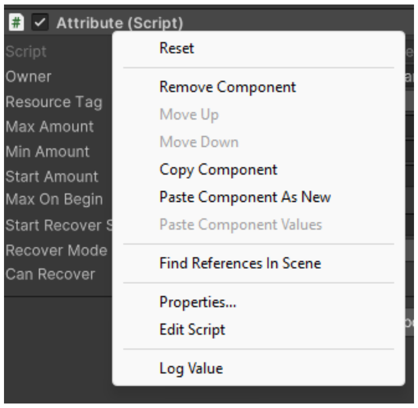
Max On Begin:
Determines whether the base value is set to the maximum when initialized.

Start Recover Speed:
The rate at which the base value recovers or decreases. This can also be modified by an Effect.

Recover Mode:
The method of recovery for the Attribute.

Can Recover:
This should ideally be removed. The recommended way to disable recovery is by using an override Effect Mod Value to set the Recover Speed to 0.

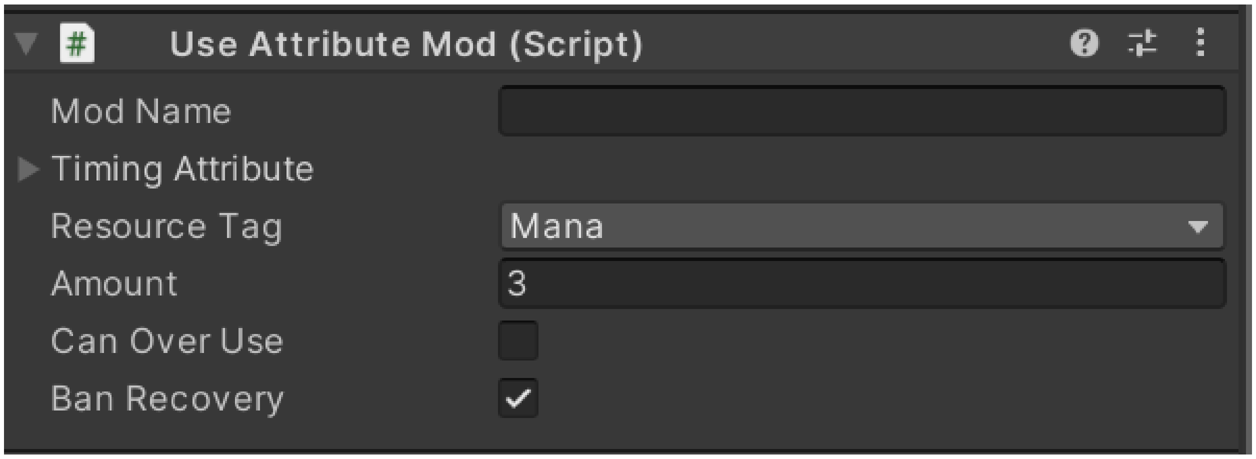
To get the current value of an Effect, right-click on the component and select 'Log Value.' The value will be printed in the console. We plan to make this more visible in the future.



Attributes can be dynamically added to a character. Some special Attribute classes, like Health and Knockback, have different behaviors. However, we may change how Knockback is handled in the future.

Using Attributes

You may attach UseAttributeMod to an Action State and choose the attribute through Resource Tag. Typically, the current attribute amount needs to be equal or greater than the required Amount for this action state to be triggered.


Use Attribute Mod (Script)

Mod Name:
Timing Attribute
Resource Tag: Mana
Amount: 3
Can Over Use: ☐
Ban Recovery: ✓

Resource Tag:
Choose the resource tag that is bound to the attribute.

Amount:
How much a character needs to cost for entering this state. This amount will be subtracted from the current amount.

Can Over Use:
Even if the current attribute amount is less than the required amount, as long as it is greater than 0, it can still be triggered, and will set the current amount to 0.

Ban Recovery:
If checked, the specified attribute will not start its recovery while the Agent is still in this Action State.

Creating a New Attribute:

You may create a child object under your character object, and it will try to add itself to the character’s attribute list by runtime.

Attribute Recovery

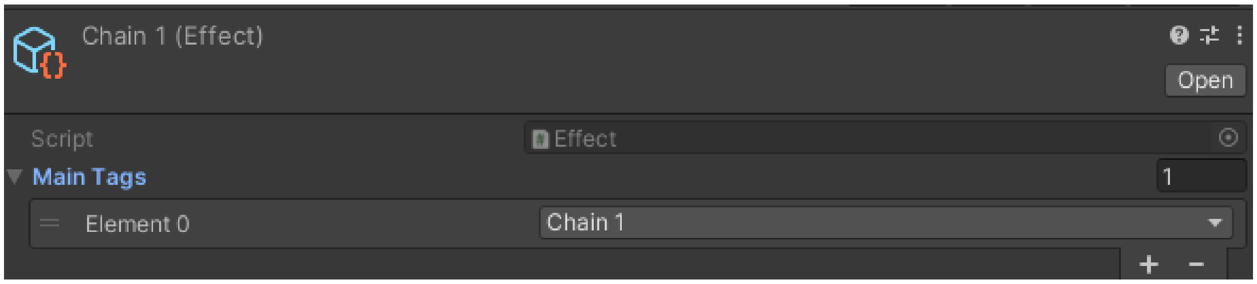
1. You may set auto recovery through the attribute component.
2. For other ways of recovery, you probably need to write an Effect for it.

Effect

Effect Definition

An Effect is a ScriptableObject that can be applied to a Character. In PAT, Effects control various logic such as damage, knockback, and more.

By default, an Effect has main tags. Once applied, it grants the tags it holds to the character, and these tags are removed when the Effect is removed.



Chain 1 (Effect)

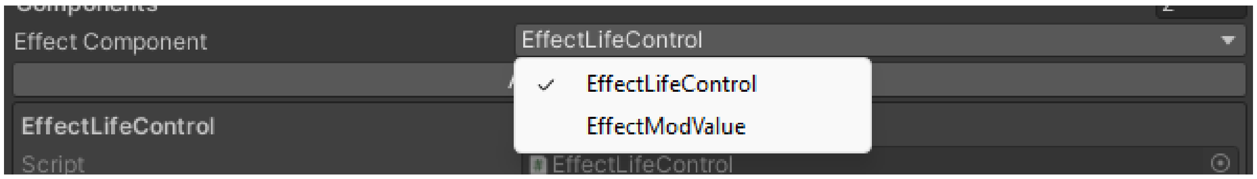
Open

Script: Effect

Main Tags: 1

Element 0: Chain 1

Effects also have components that modify their behavior in different ways. You can add an Effect Component by selecting a class from the dropdown and then clicking the 'Add Component' button.



Effect Component: EffectLifeControl

EffectLifeControl

Script: EffectLifeControl

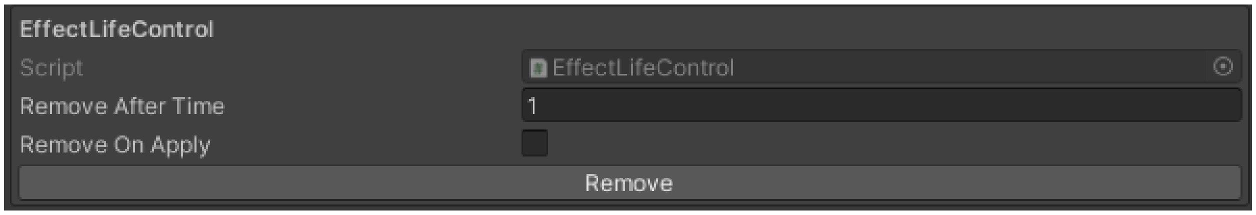
Effect Component

Currently, there are two types of Effect components implemented in PAT: Effect Life Control and Effect Mod Value.

This section introduces how these components help establish Effect logic, and you can create your own components by inheriting from the Effect Component class.

Effect Life Control:

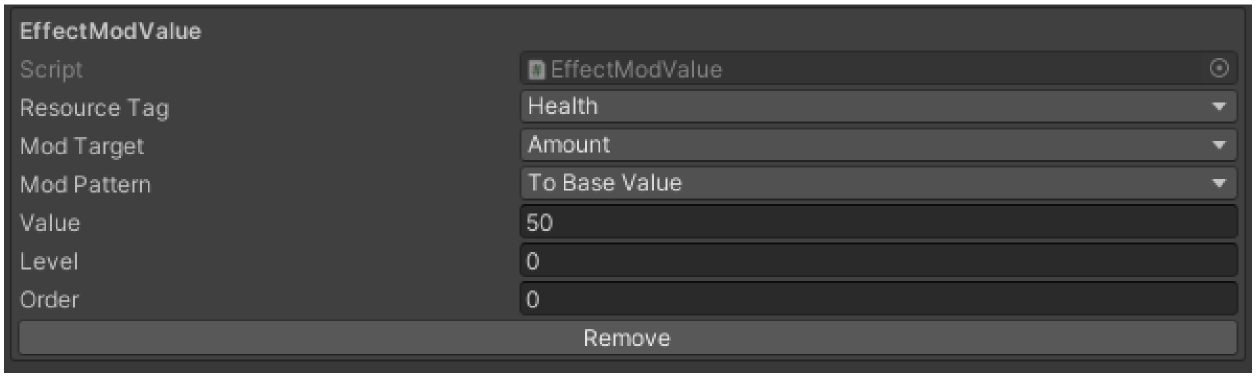
This component controls how long the Effect will remain on the character on its own. It's a useful tool when you don't want to manage the Effects' lifespan manually.



EffectLifeControl	
Script	EffectLifeControl
Remove After Time	1
Remove On Apply	☐

Effect Mod Value:

This component influences the values of Attributes in various ways. HP, Mana, Posture, Defense, and similar stats are all Attributes in PAT. Controlling Attributes generally means managing every non-action aspect of the game.



EffectModValue	
Script	EffectModValue
Resource Tag	Health
Mod Target	Amount
Mod Pattern	To Base Value
Value	50
Level	0
Order	0

Resource Tag:
The Effect Mod Value will search for an Attribute with the same tag and influence it.

Mod Target:
Specifies whether it modifies the Attribute's value or the auto-recovery rate of the Attribute.

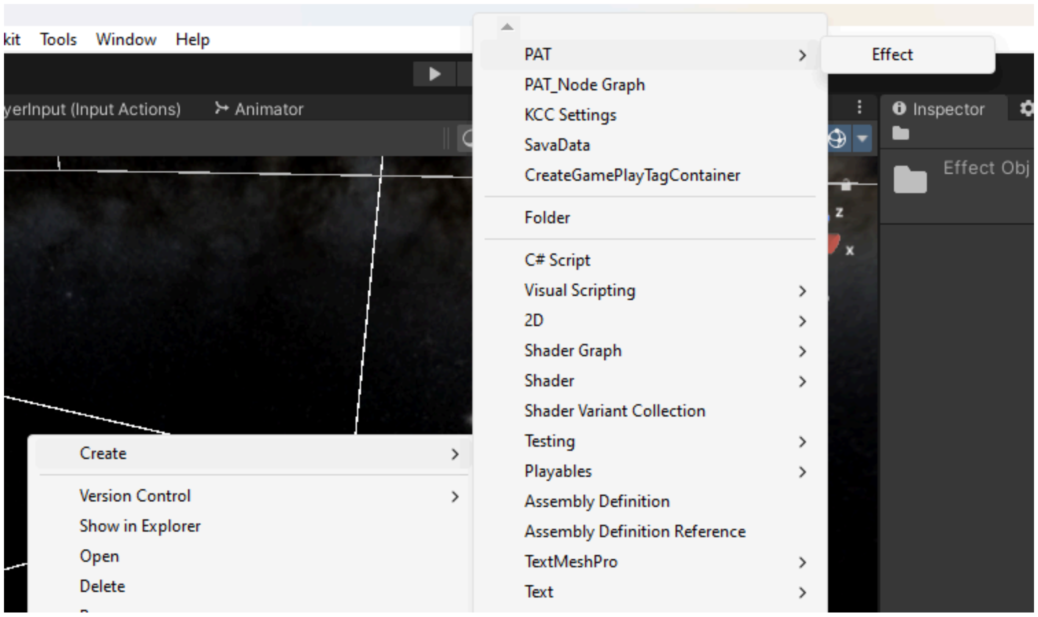
Mod Pattern:

- **To Base Value:** Adds or subtracts from the actual base value, commonly used for damage, mana, or stamina.
- **Add:** Increases or decreases the final value, but will no longer influence it once the Effect is removed. This is useful for buffs like a speed boost.
- **Add Multi:** Multiplies the value. Multiple Add Multis will stack together before the calculation. Similar to Add, it will no longer influence the final value once removed.
- **Final Multi:** Unlike Add Multi, this does not stack. It multiplies the value directly.
- **Override:** The value is directly replaced by the override value and will no longer influence the final value once the Effect is removed. Useful for effects like reducing movement speed to zero.
- **Level:** Not used by default, but you can implement special logic by overriding the Attribute class.

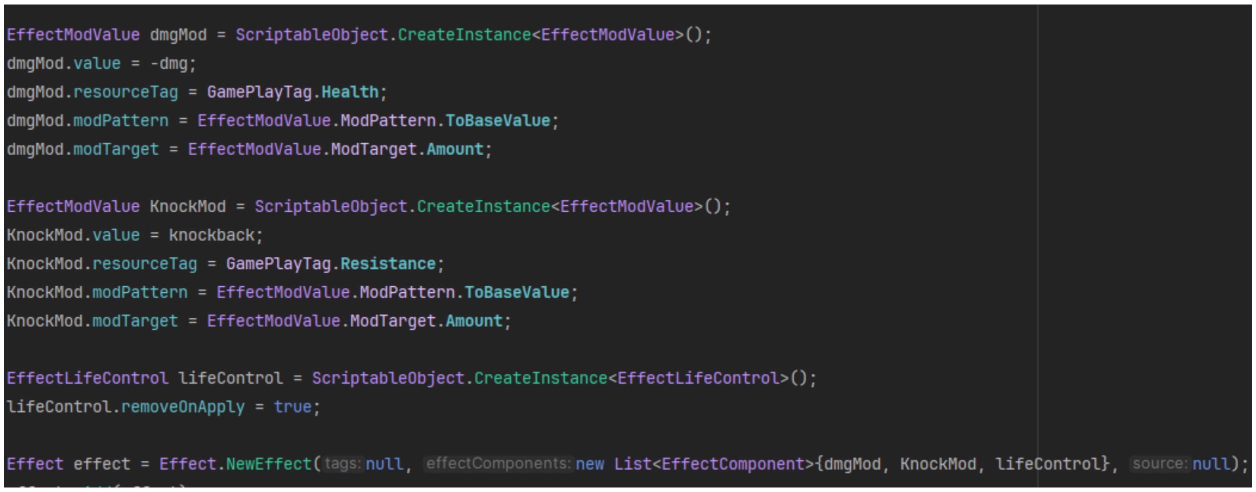
- **Order:** Determines the priority between multiple Effect Mod Values. If both have the same order, the most recently added Effect will take precedence.

Effect Creation

Currently, there are two types of Effect components implemented in PAT: Effect Life Control and Effect Mod Value. To create an Effect, you can either create a ScriptableObject asset in your project or implement your own custom Effect by inheriting from the appropriate class.



Or you create via script:



```
EffectModValue dmgMod = ScriptableObject.CreateInstance<EffectModValue>();
dmgMod.value = -dmg;
dmgMod.resourceTag = GamePlayTag.Health;
dmgMod.modPattern = EffectModValue.ModPattern.ToBaseValue;
dmgMod.modTarget = EffectModValue.ModTarget.Amount;

EffectModValue KnockMod = ScriptableObject.CreateInstance<EffectModValue>();
KnockMod.value = knockback;
KnockMod.resourceTag = GamePlayTag.Resistance;
KnockMod.modPattern = EffectModValue.ModPattern.ToBaseValue;
KnockMod.modTarget = EffectModValue.ModTarget.Amount;

EffectLifeControl lifeControl = ScriptableObject.CreateInstance<EffectLifeControl>();
lifeControl.removeOnApply = true;

Effect effect = Effect.NewEffect( tags: null,  effectComponents: new List<EffectComponent>{dmgMod, KnockMod, lifeControl},  source: null);
```

Using ScriptableObjects is preferred for prototyping and designing game logic. You can find many mods, such as 'Add Effect' or 'Add Effect On Hit,' created this way.

Script generation is preferred for dynamic hit processing. Modifiers like Block Mod and Attack Mod are typically implemented using this method.

Applying Effects

If a character wants to add effect to itself, you should probably use Add Effect Mod.

If a character wants to add effect to other PAT components, you may use Attack With Hitbox Mod or Add Effect On Hit Mod. Or you can use the Add Effect Box, which needs to be attached to a trigger so that it allows designs like traps.

When choosing which Effects to apply, you can drag all the Scriptable Objects you created into the effect lists, or you can find an option called Effect Factory on scripts like Attack With Hitbox Mod.

Effect Factory

Effect Factory is a composited effect for you to create multiple effects at once. We provide you a template called EffectFactory and a damage factory EffectFactory_P applied that template. You may write factories for your own convenience.