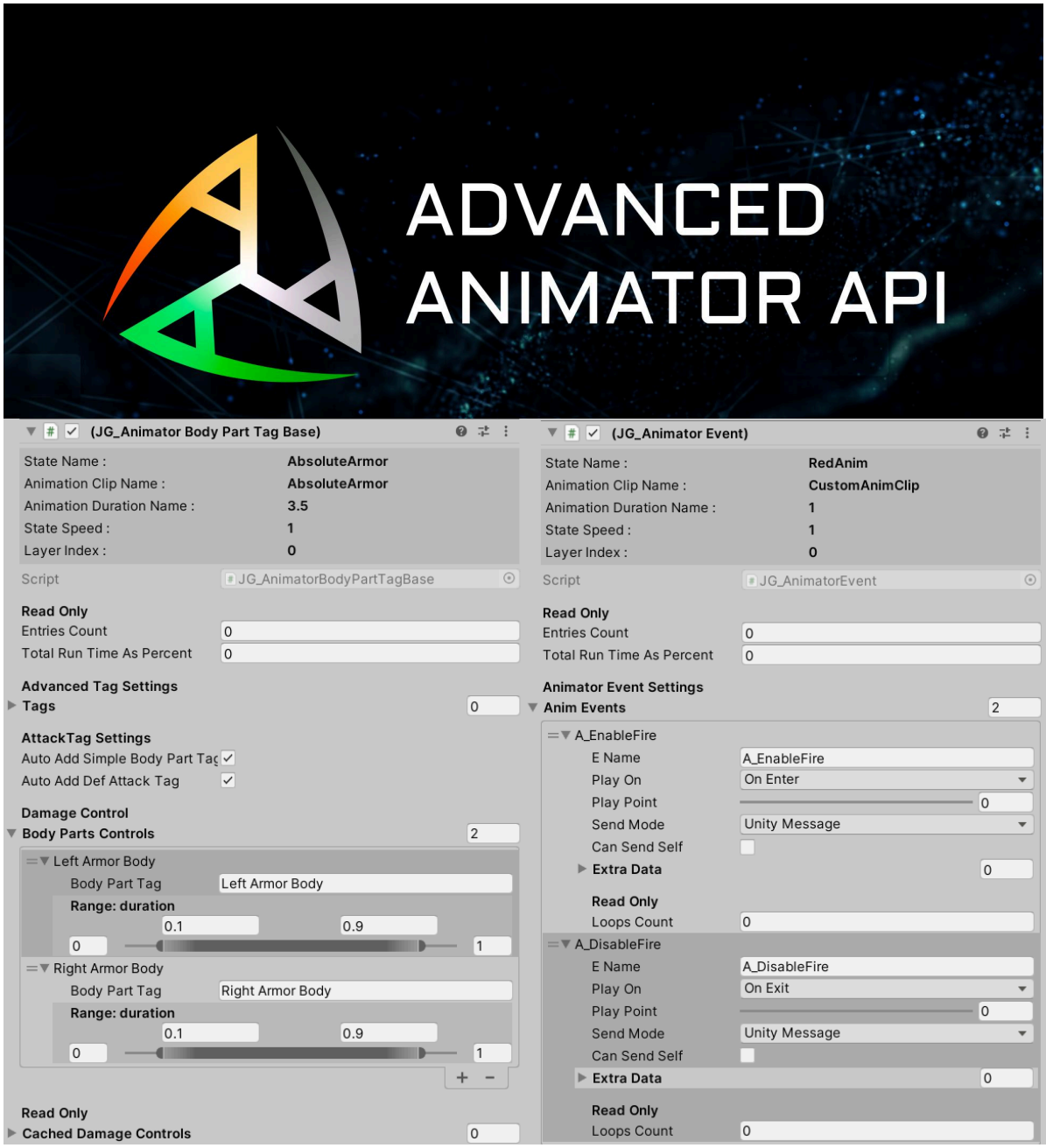
AAA ADVANCED ANIMATOR API

User Manual

Introduction

Congratulations On your purchase of AAA: Advanced Animator API, a whole new world of easy to use animator functions, with improved animator events, custom multiple tags per animator state, and easy to access states from the data of the animator state, and even have the animator automatically control game objects, they will make using the animator in your project a breeze to use, and lots of fun.



About This Package:

What Are The Main Contents?

A large collection of extensions for unity's animator that make working with unity animator far easier, more flexible with many easily extensible behaviors, such as advanced custom animator events, tags, body parts/objects control, and accessing states data like duration or time.

Why Do I Want To Use This Package?

Unity animator is powerful, but the functions that are provided by unity's default api are rather limited, for example, you can not get the currently running state in a specific layer, or the duration of the current animation clip, or the name of the clip, or how far did the current running animator state progress through, or if it's a looping animation, we can't find out how long it had looped, or what about animation events? They're great, but you're bound to scripts that exist on the animator component itself, and if you change the animation, then you'd have to recreate your events all over again, which can be time consuming, and very error prone which could literally break the game.

This package solves all of these problems, and more, I did not design this package to be a highly specialized hyper complex asset that only works in a certain type of game, that is difficult or time consuming to set up.

Instead my main goal was just a few scripts you can throw around that are stable and easy to use, integrate into your existing code, reuse and extend when you need to, that solve all of these problems.

What Kind Of Games Can I Use This For?

Simply put, if your game uses the animator, if you code in C# in your project, and interact with the animator, then this asset is for you, regardless of the game type you're building, there are so many moments where your life would become far simpler when dealing with the animator in code, it can be used in fully fledged games, or even simple game jam games, this asset is easy to use, and it's main purpose is to save time.

What Are The Core Features?

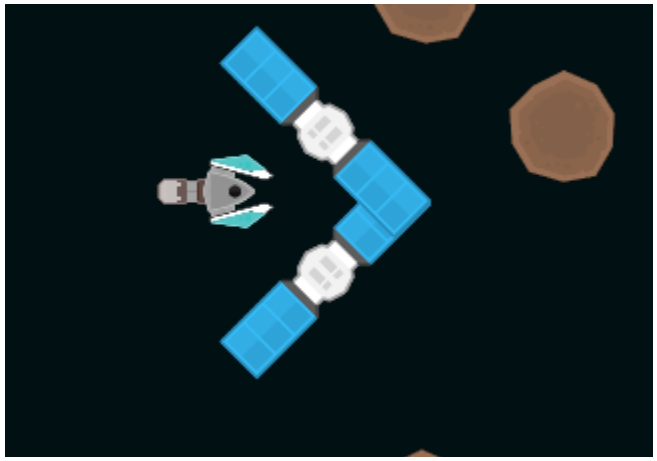


Fig 4: Control Game Objects Through Animators

Fig 5: Multiple Custom Tags Per Animation

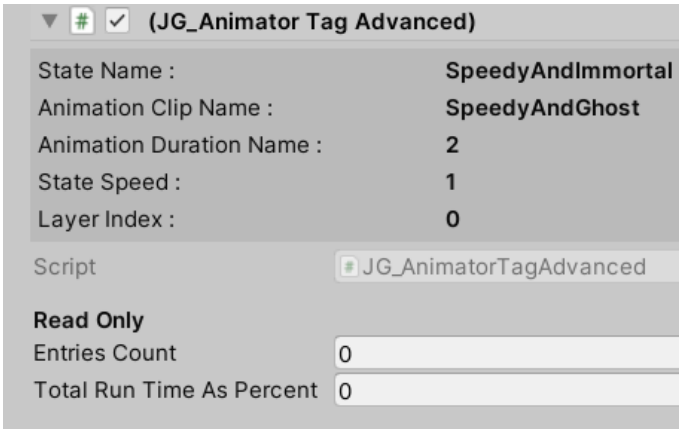


Fig 5: Easy To Access Information From Any Animator State

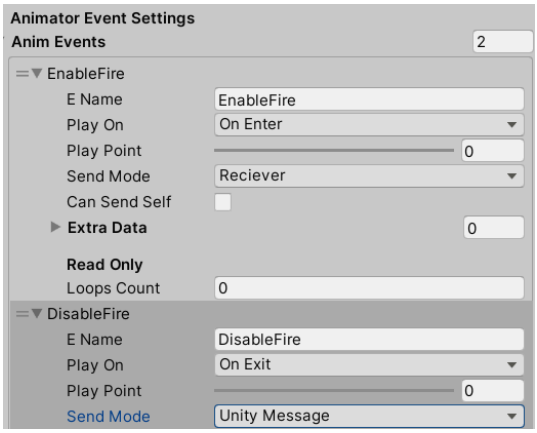


Fig 6: Advanced Animator Events That HasTwo Modes (Inspector And Code Based)

Custom Animator Events: typically unity animation events are tied to the clip itself, if you decide to change the animation clip then you'd need to recreate the animation events on the new clip, of course, all of that in the past, with the Custom Animator Events you can keep all flags, events and messages, and change the clips with no worries.

1. You can send events at any point in time, whether it's at the when you enter the animator state, when you exit or when it reaches a point in time.
2. You can send UnityMessages (functions to scripts) you can also delegate them to different game objects
3. You can send a message to an AnimatorEventRecievers which UnityEvents, similar to OnClick in the buttons.

Connect Game Objects With Animator States: a versatile BodyPartsController is included, that allows you to control specific game objects that game enabled and disabled based on where you are in the animation and configure it to multiple body parts at same time without writing a single line of code

Define Custom Animator Tags:

1. Give an entire animator state, a tag,
2. Give portions of animator states some specific tags

Information Accessing: You can access information from any animator state at any moment of time such as:

1. Is the state running or not
2. how long has it been running
3. Duration of the state
4. The speed that the state is running
5. The layer index in which the state exists in
6. The name of the animation clip

Fully Documented And Source Included: the asset is completely documented with a manual to read and utilize, the source code is also included into the asset and finally, the source code is also fully documented

A Bonus: on top of the core content of the asset, there are also over 120 modular reusable components for you to use into your project and explore, handling things from separating triggers from collider game objects, decal stickers, or positional arrangement scripts like arranging game objects automatically in an arc, a debugger system, and an Automatic Y Sorter for your isometric 2D games, and many more.

Quick Start

Demo Scenes

Advanced Animator Api comes with a variety of examples, showcasing the api abilities and the possible applications.

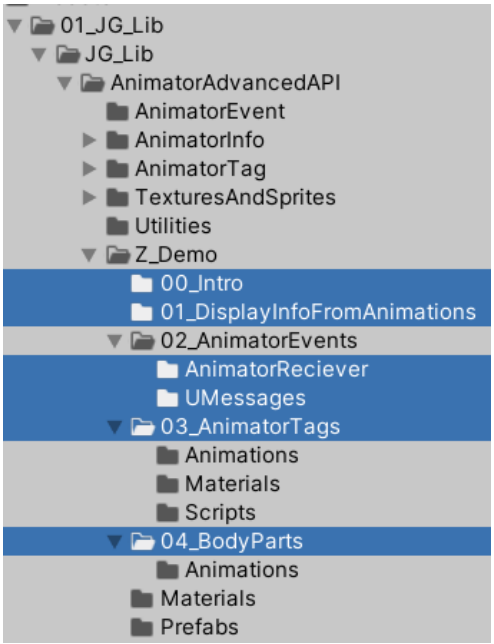


Fig 1: Folder Locations Of Demo Scenes In The Project

00_Intro

This is the most bare minimum demo for the animator, where it modifies the animator states, by adding JG_AnimatorStateInfo component on them to collect information about the different states of the animator

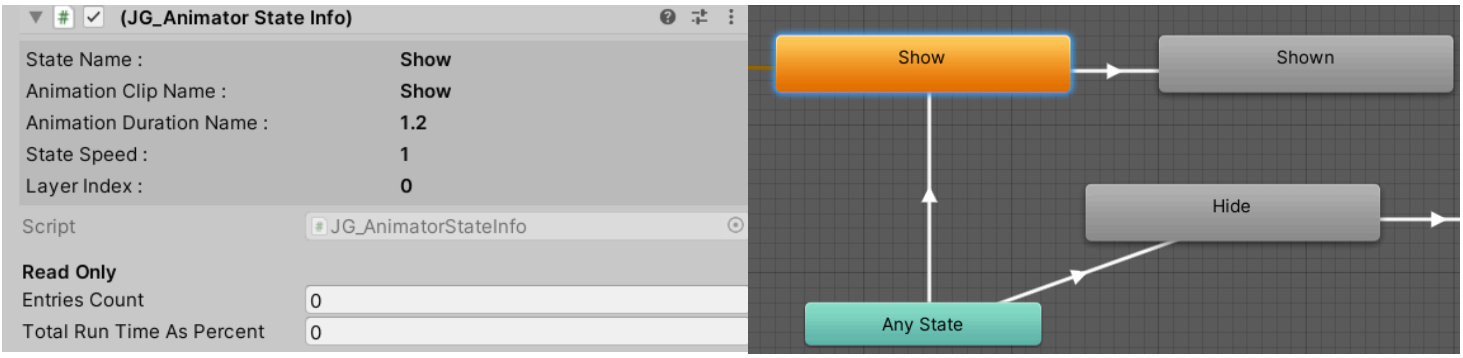


Fig 2: Most Basic Example

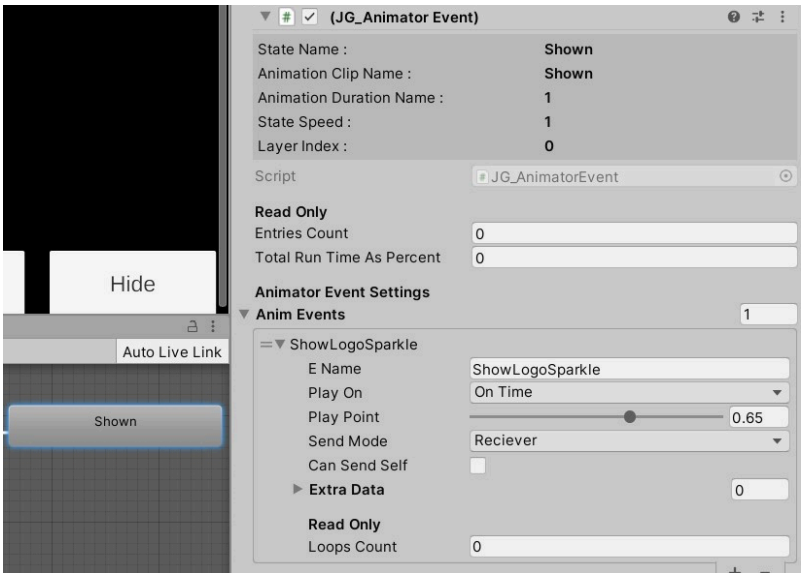
01_DisplayInfoFromAnimations

We expand on the previous demo, by displaying the information that we added to the animator states on the UI, while the game is running.

Fig 3: Displaying animator state information while game is running on UI components

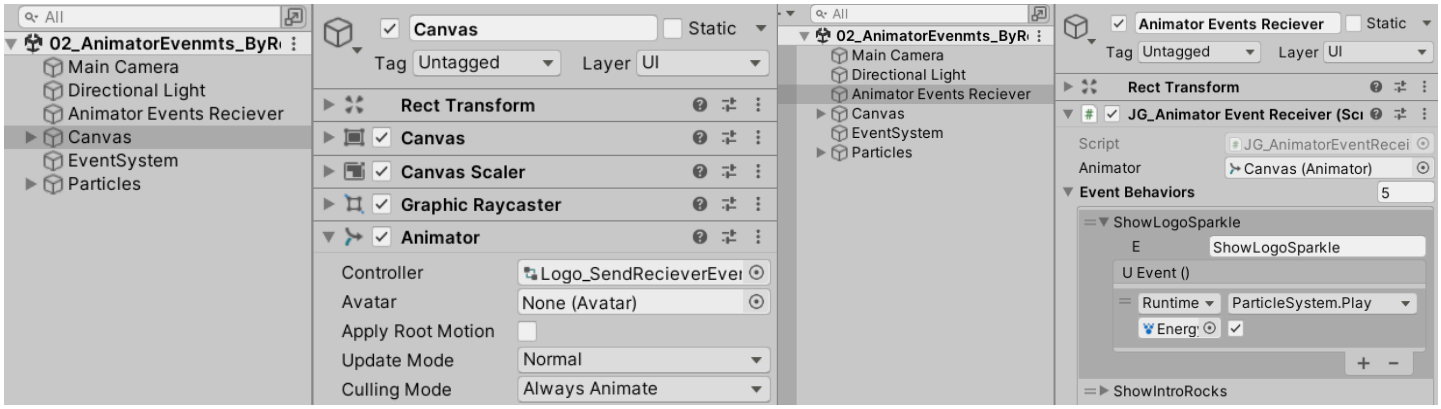
02_AnimatorEvents

Animator events, they're different from **animation** event, you can use them to send **Unity Messages** or call them through the inspector from a component named **JG_AnimatorEventsReciever**, a major benefit of using those over **animation events** is that even if you change the animation clip, you do not have to remap and re-add your animation events, they will still stay, because they are attached to the animator state itself.



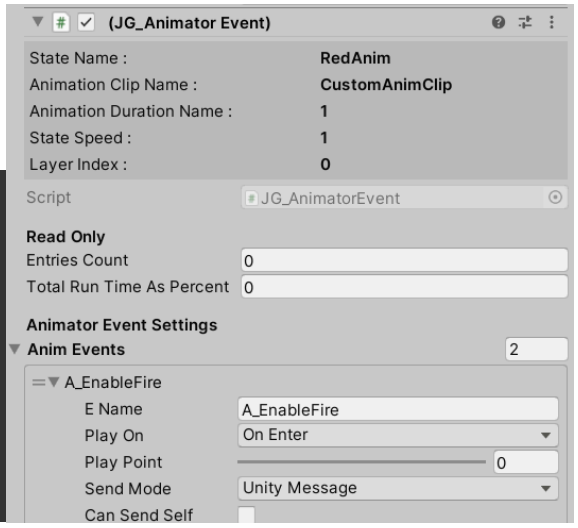
AnimatorReciever Located At **01_JG_Lib/JG_Lib/AnimatorAdvancedAPI/Z_Demo/02_AnimatorEvents/AnimatorReciever/**

In this demo we call **AnimatorEvents** from the animator and respond to them, through the inspector, there are 2 key game objects, the "**Canvas**" which contains the animator, which in turn contains the animator states, and we **send** the events from the corresponding animator states, and the "**Animator Events Reciever**" Gameobject.



UMessages

UMessages stands for Unity Messages, in other words, you specify the function you want to call by writing its name in the Animator event, and then in a script attached to the same game object that has the animator, then the function that you wrote its name will get called.




```
public void Demo_02_UnityMessages : Monobehaviour
{
//This component must be attached to the
//animator, or plugged in a JG_AnimMessagesDelegator
    private void A_EnableFire()
    {
        //Enable fire behavior
    }
}
```

03_AnimatorTags

Say you have a large game, with many different abilities, where for as long as the ability is running an animation is active, some abilities woulds give a speed boost, while others would give you want to tag certain parts, you could define all of that in code, or, you could simply create a system where you play the animation, and the animation just decides what special powers the abilities give, which is what animator tags stand for, so you can give special tags to certain pieces of animations.

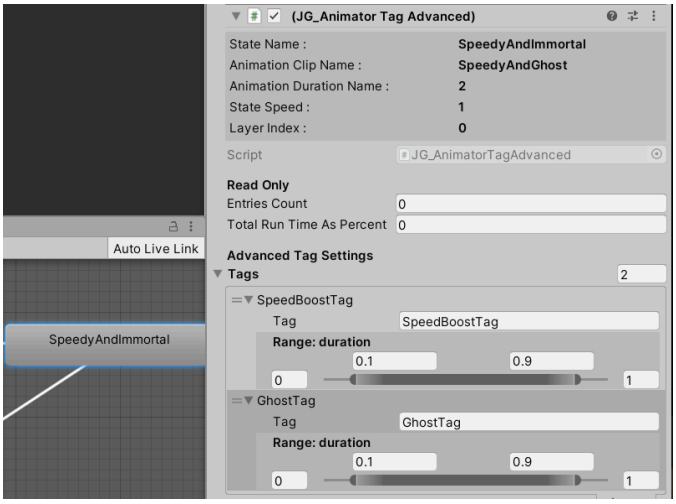


Fig 3: Animator Tags

Example, we defined the animator state **“SpeedyAndImmortal”** above, gave it the tag, **“SpeedBoostTag”** and **“GhostTag”**, now in the code base, based on tag as follows:

```
void Update()
{
      if (animator.IsAnimatorTagActive(ref collector, "SpeedBoostTag"))
      {
           mover.maxMoveSpeed = defaultMaxMoveSpeed * speedyMaxSpeedMulti;
      }
      if (animator.IsAnimatorTagActive(ref collector, "GhostTag"))
      {
          health.isImmortal = true;
          IsGhost = true;
      }
}
```

04_BodyParts

Similarly to **Animator Tags**, you may want to enable/disable game objects during attack or different abilities, for that, all you need is to do modify the animator state by adding **JG_AnimatorBodyPartTagBasic**, and attach **JG_BodyPartMono** to the target game object,where tag names in **JG_BodyPartMono** must match ones defined in the animator state, and finally, you just need to attach JG_BodyPartsController on game the root game object, and then game objects will be enabled/disabled automatically, based on the current animator state.

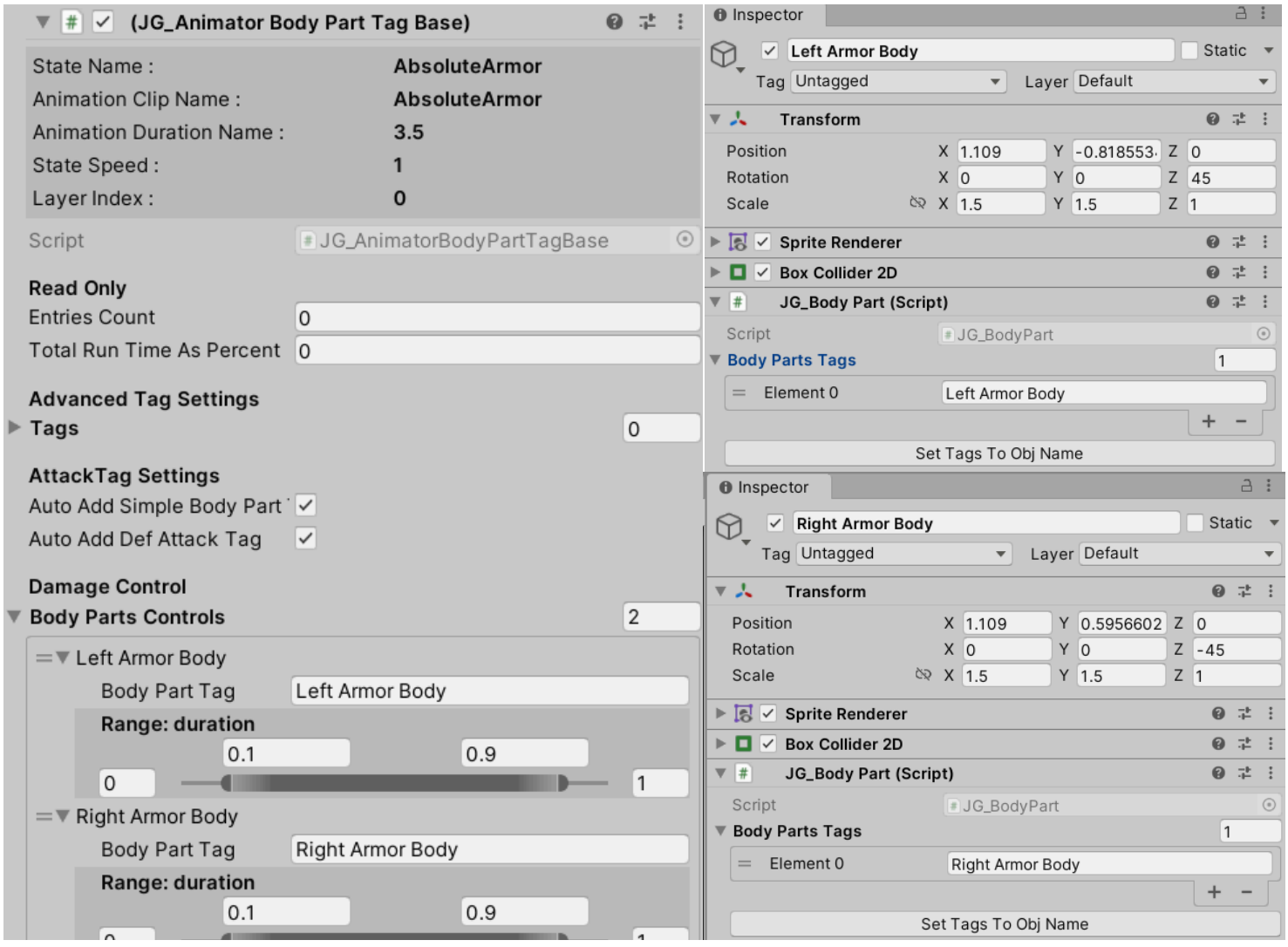
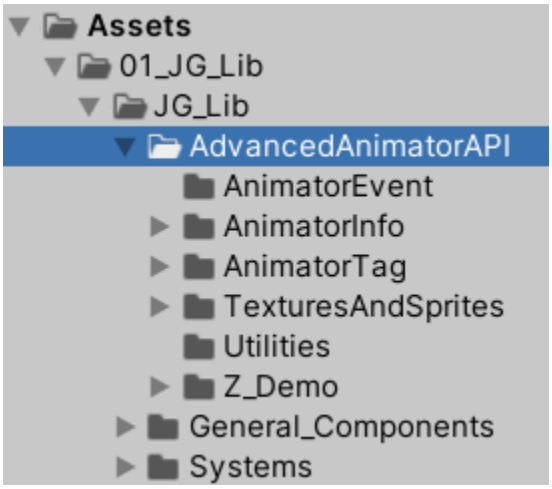


Fig 3: Animator Body Parts

Package Structure

The package is structured as follows:



To keep everything organized and easy to access, all package assets are contained in one root folder named **01_JG_Lib**, this ensures that your project files are separated from the asset files, so it can be very easily imported into any unity project.

AdvancedAnimatorAPI

This is the main directory that contains the core of the asset, which has all **Components** and **AnimatorStateMachines** scripts that are required to use all of the features.

AnimatorInfo

Fig 7: Animator Info Directory

This sub-directory contains one single script:

JG_AnimatorStateInfo: this script is the basis of the entire asset, and most other classes for this asset inherit from it, you can attach it to animator states, and it will make the information related to the animator state easily accessible.

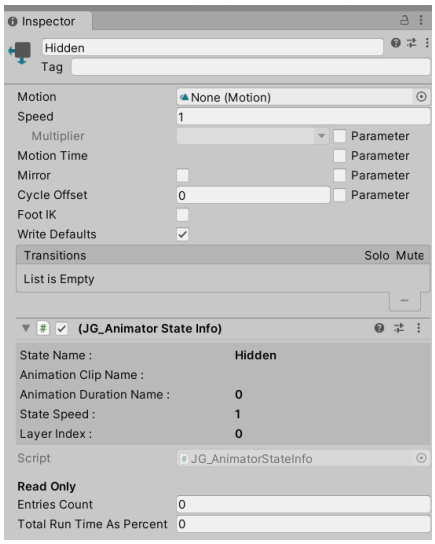


Fig 8: JG_AnimatorStateInfo attached to the animator state named **Hidden**

Then inside of any script you can do as follows:

```
animator.GetAnimatorStateInfo(ref infoCollector,"Hidden");//how to get information
of a specific state
activeInfo.GetStateName;//Get the name of the animator state
activeInfo.GetClipName;//Get the name of the clip
activeInfo.GetLayerIndex;//Get the layer index in which the animator state lives in
activeInfo.GetStateSpeed;//Get the speed of the aniamtor state
activeInfo.GetAnimDur;//Get duration of the animation
activeInfo.GetTotalRunTimeAsPercent;//Get how long the animator state has been
running, 2.5 means it ran twice and a half
activeInfo.GetTotalRunTimeInSeconds;//Get how long aniamtor state has been running
in seconds
activeInfo.GetRunTimeAsPercent;//Get how long animator state has been running, from
0 to 1, 0.5 means state is in middle
activeInfo.GetRunTimeInSeconds;//Get how long animator state has been running in
seconds
```

AnimatorEvent

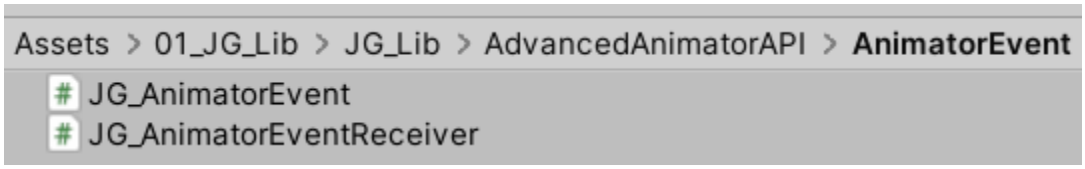


Fig 9: Animator Event Directory

This sub-directory contains two main scripts

JG_AnimatorEvent: is the script that gets attached to animator states, so you can customize and call any event from these states.

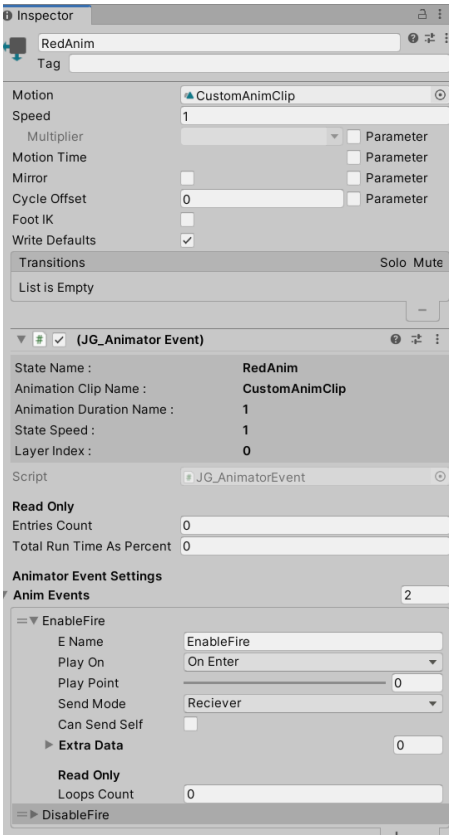


Fig 10: Animator Event Attached To An Animator State Named Red Anim

It contains a field named “**Anim Events**” which holds a reference to all events that get triggered by this animator state.

AnimatorEvent Variables:

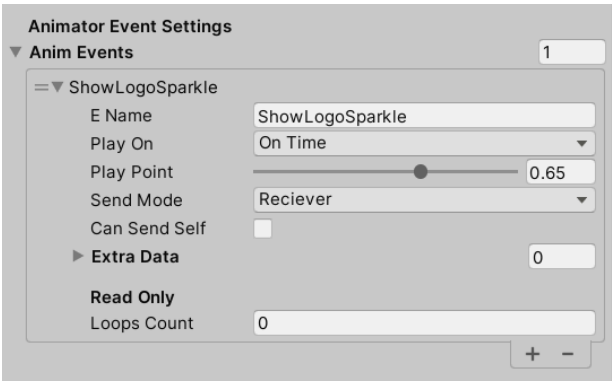


Fig 11: Animator Event Variables In Inspector

E Name: is the event name of the event

Play On: is the moment we want to play the event, **On Time** means it gets played on a specific point in time (from 0 to 1) and is determined by **Play Point** variable, there are also two types which are **On Enter**, which gets called the moment we enter the animator state, and **On Exit** which is called when we leave the animator state.

Play Point: is only used if we are using **On Time** mode for the **Play On** variable, and it represents the point that the event will be played at.

Send Mode: there are 3 supported send modes, which are **Unity Message** which will send a unity message to the game object that contains the animator (which can be delegated to another game object using **JG_AnimMessagesDelegator** component and a receiver mode, which uses our custom inspector named **JG_AnimatorEventReciever** (in same directory).

Can Send Self: is only used in **Unity Message**, and **Both** modes, however even when using Both, the receiver will not receive that data being sent, if this variable is true, then for the unity function call, we will send the Animator Behavior itself, as the first element of the list.

Extra Data: is also used only in **Unity Message** and **Both** Modes, and even when using the "**Both**" mode, then the data will only be sent through the Unity Message, note if **Can Send Self** is set to true, then the "**Self**" which is the behavior, will get added as the first element in the list.

```
void ShowLogoSparkle(List<object> objs) {
//access data here, if you sent Extra Data, or canSendSelf is true
}
void ShowLogoSparkle(){//if canSendSelf is false and Extra Data is empty}
```

JG_AnimatorEventReciever

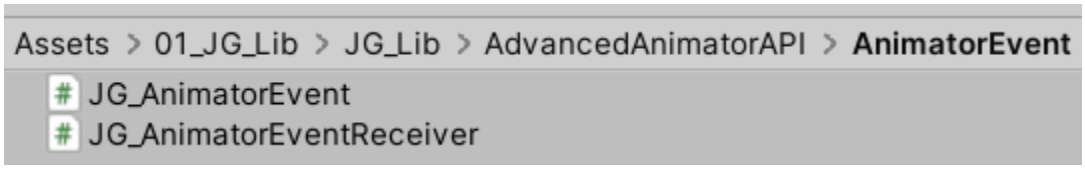


Fig 11: Animator Event Directory

JG_AniamtorEventReceiver: The second script in the AnimatorEvent Directory, now if we used any animator event that utilizes the "**Both**" send mode, and the "**Receiver**" send mode, then for these events, we can use this script.

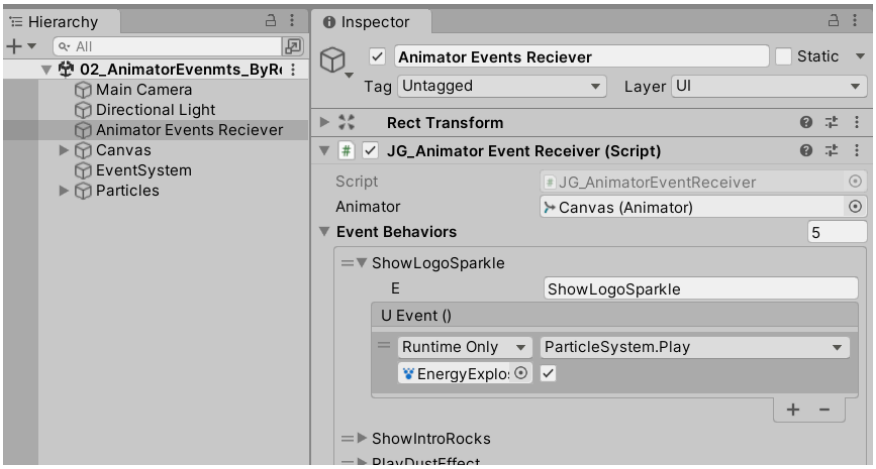


Fig 12: JG_AnimatorEventReciever Component On A GameObject

You can attach this component to any game object, and all you need to do is plug the exposed "Animator" variable, and then fill whatever events you need.

Animator: the animator that you want to connect to, and listen to.

E: the name of the event you want to listen to

U Event: is the unity event that will be called based on the animator event, think of it like On Click from the UI Button but instead of it being called when you click on the ui component button, it gets called on the Animator Event, that you defined previously in the animator, and that you wrote it's name here, then you can do any actions inside of it like playing particles or calling other functions.

AnimatorTag

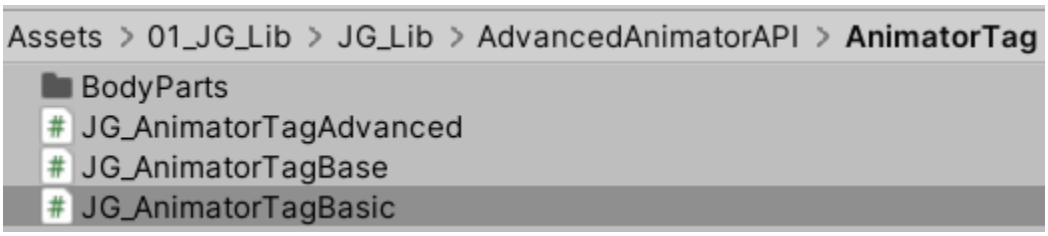


Fig 12: AnimatorTag Directory

There are three main scripts in this directory, and a subdirectory named BodyParts.

JG_AnimatorTagBase:

This script is an abstract class that all animator tags inherit from.

JG_AnimatorTagBasic Variables:

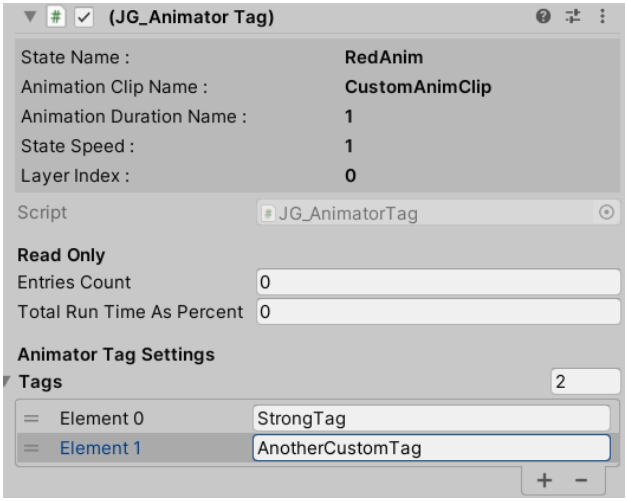


Fig 13: JG_AnimatorTag

This is the simplest animator tag, you simply attach it to some animator state, and the animator state will use the tag, you can define them in the "**Tags**" list, in the code you can detect whether the current running animations have a specific tag as follows:

```
if (animator.IsAnimatorTagActive(ref collector, "SpeedBoostTag"))
{
   mover.maxMoveSpeed = defaultMaxMoveSpeed * speedyMaxSpeedMulti;
   mover.acceleration = defaultAcceleration * speedyAccelMulti;
}
```

JG_AnimatorTagAdvanced Variables:

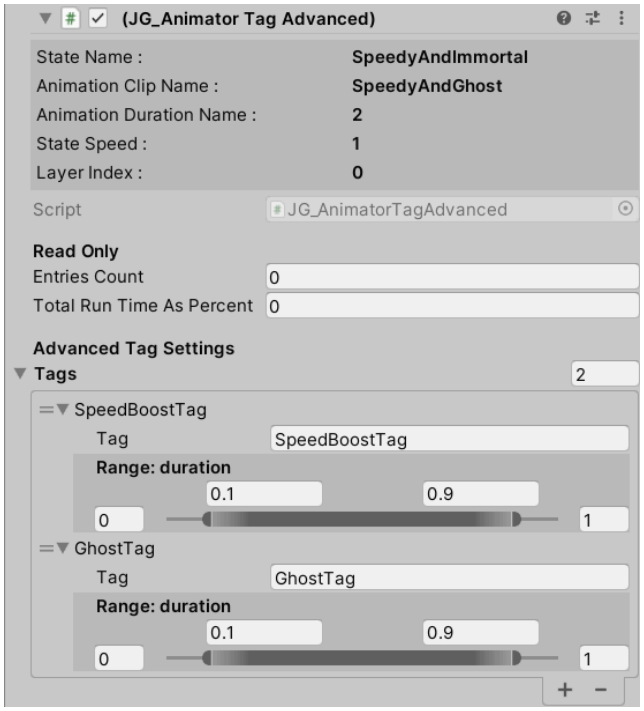


Fig 14: JG_AnimatorTagAdvanced

This is exactly the same as **JG_AnimatorTag** but with the difference that the "**Tags**" that you define in this script are time based (as percent), where the tag will only be active for the specified duration, example SpeedBoostTag will run while the animation is between 0.1 to 0.9, if the animator state is at 0.95 for example, then even though we are in the animator state, it will register has if it's not running.

And we can access these tags through code, in fact we do not need to change the code at all compared to the JG_AnimatorTag, we can in fact reuse exactly the same code to detect whether a tag is running or not.

```
if (animator.IsAnimatorTagActive(ref collector, "SpeedBoostTag"))
{
   mover.maxMoveSpeed = defaultMaxMoveSpeed * speedyMaxSpeedMulti;
   mover.acceleration = defaultAcceleration * speedyAccelMulti;
}
```

BodyParts

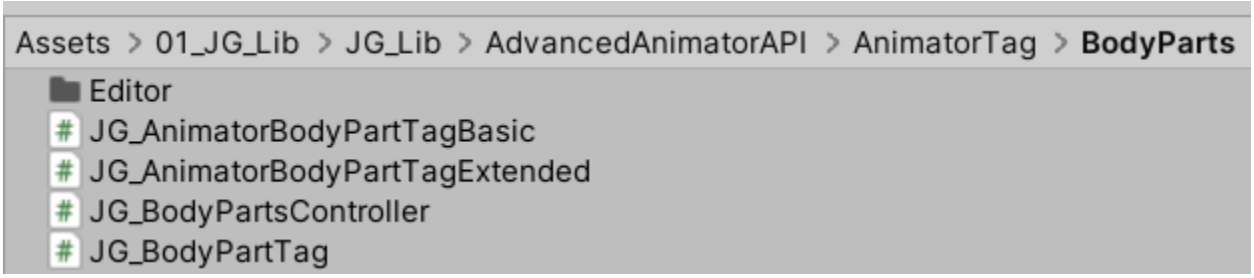


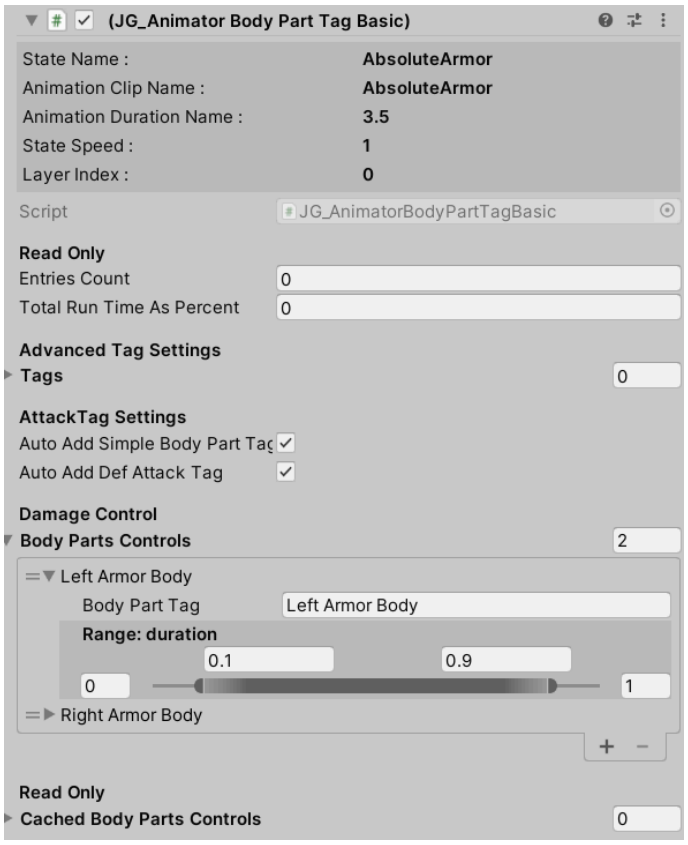
Fig 15: BodyParts Directory

This sub-directory contains four core scripts, two StateMachineBehaviors and two Monobehaviors. Before we discuss them, let us recall why we use the BodyParts system? The answer is simply when we want to control the activity of game objects, in the animator, you basically decide on a game object you want to control the activity of through animator, then you attach **JG_BodyPartMono** to the game object then in the animator state you add **JG_BodyPartTagBasic,** and finally on the animator

JG_AnimatorBodyPartTagBasic

This script behave very similarly to **JG_AnimatorTagAdvanced** as it gets added to the animator state, it extra property over the **JG_AnimatorTagAdvanced**:

Body Parts Controls: in here you define a custom tag, that corresponds to a game object, and you define it's activity duration, just like you did define for tags previously, so from 0.1 to 0.9, the game object tagged with our special tag, with string of “Left Armor Body” will be activate for that duration.



JG_BodyPartMono

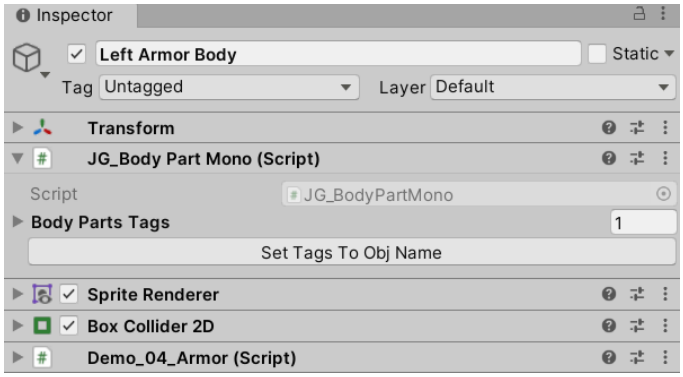


Fig 15: JG_BodyPartMono Attached To A Gameobject

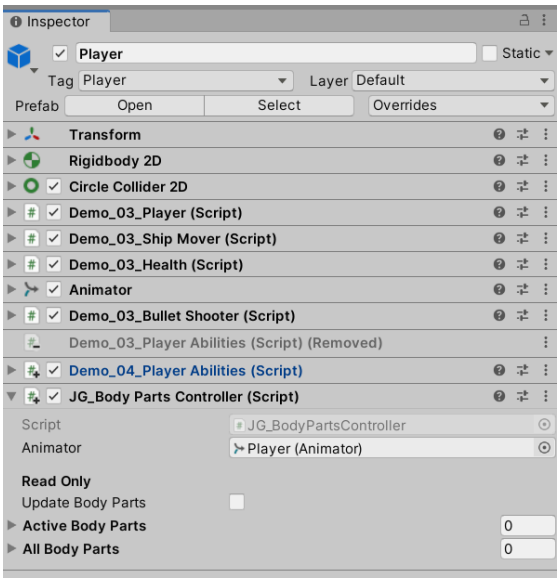
Previously in **JG_AnimatorBodyPartBasic** we defined a custom string, that corresponds to the game object, now, to attach the game object, and make it visible to the system, we need to tag it with our special tag which is a component called **JG_BodyPartMono**.

So we attach **JG_BodyPartMono** to any game object that is a child of a root game object that has the component named **JG_BodyPartsController**, and then we assign the proper string (it must exactly match the string defined in **JG_AnimatorBodyPartBasic**) and that would be all we need to do.

JG_BodyPartsController

You attach this to the root game object (typically the same game object that has the animator component), then you make sure that the animator is plugged.

And finally, this controller will ensure that game objects will respond properly to the **JG_AnimatorBodyPartBasic** behaviors that you attached to any animator state



JG_AnimatorBodyPartExtended:

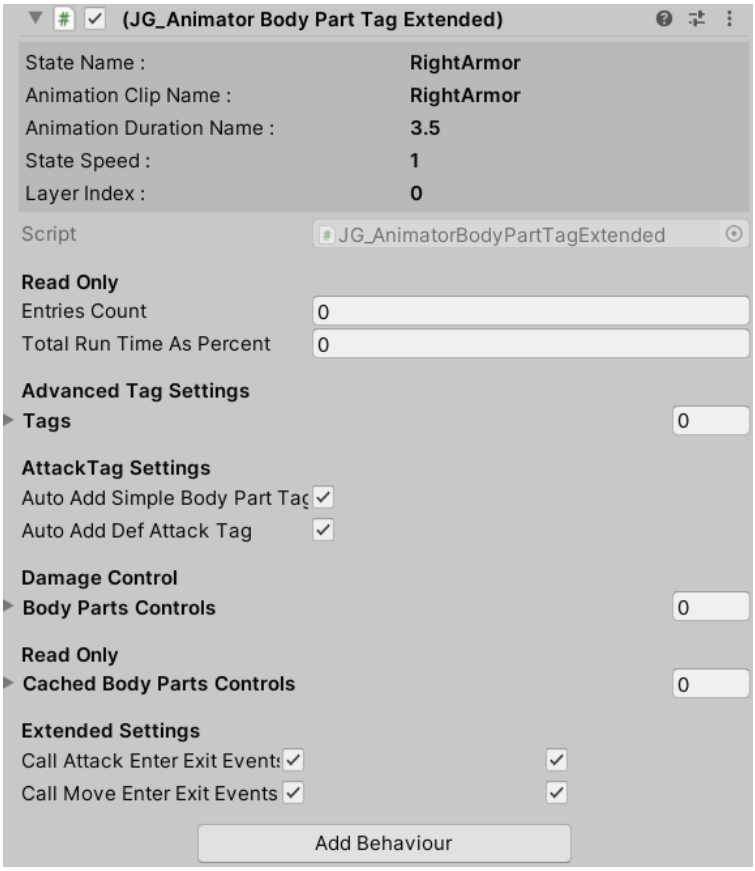


Fig 16: JG_AnimatorBodyPartTagExtended Attached To An Animator State

A simple example where we extend the **JG_AnimatorBodyPartTagBasic** so that it can also send two custom messages, and it also utilizes some of our extra **General_Components**, mainly the **DoubleBool**, it can serve as an example to show how to extend the scripts if needed.

Utilities

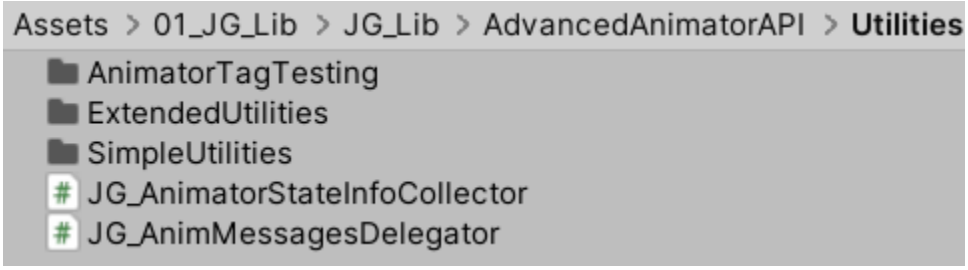


Fig 17: Utilities Sub-directory

JG_AnimatorStateInfoCollector:

This is a helper script that gets attached automatically to the animator when you use the Extension methods, it is responsible for collecting information from the animator, it holds a reference to both the running data, and all data contained within the animator.

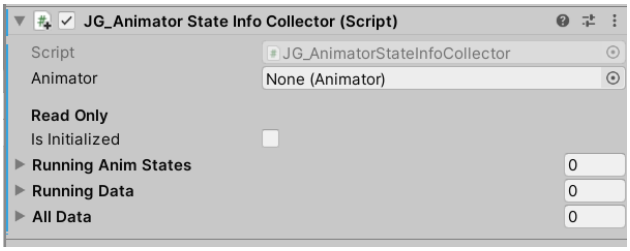


Fig 18: JG_AnimatorStateInfoCollector Attached To A Game Object

Note: you do not need to add this component manually if you use the extensions, but this is just a picture on how it looks, if you use extensions it gets automatically added, and configured properly.

JG_AnimMessagesDelegator:

When using the JG_AnimatorEvent, you can choose a mode of "**Unity Message**" or a mode named "**Both**", for events/messages that are sent this way, they are by default sent to the game game object that has the animator, if you want to delegate these function calls to another game object, simply attach this component **JG_AnimMessagesDelegator** to the same object that has the animator, and then fill the target.

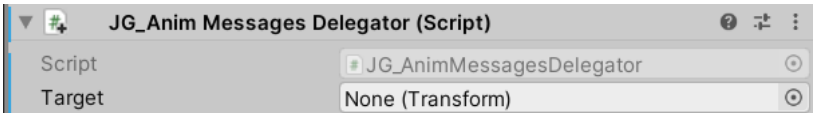


Fig 19: JG_AnimMessagesDelegator Attached To A Game Object

ExtendedUtilities

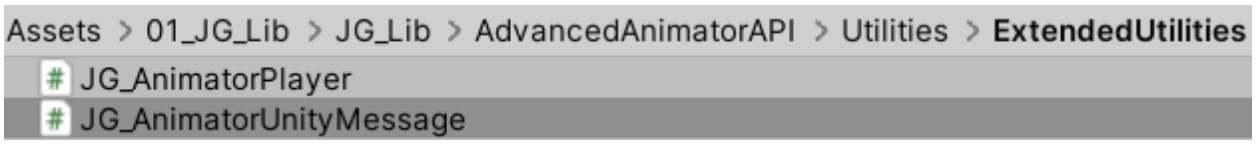


Fig 20: ExtendedUtilities Subdirectory

This directory contains extra helper scripts that extend **JG_AnimatorStateInfo** class

JG_AnimatorPlayer:

A simple abstract class for a StateMachineBehavior, it basically controls the flow of the script such that, an abstract function called “PlayerBehavior(Animator animator)” gets called on one of three places: OnEnter,OnExit,OnTime.

JG_AnimatorUnityMessage:

A simple script that extends **JG_AnimatorPlayer** and is used to serve as an example of how to use the **JG_AnimatorPlayer** and extend it, it has only one job which is sending a **Unity Message**, the JG_AnimatorEvent is a far more advanced version of this script.

SimpleUtilities:

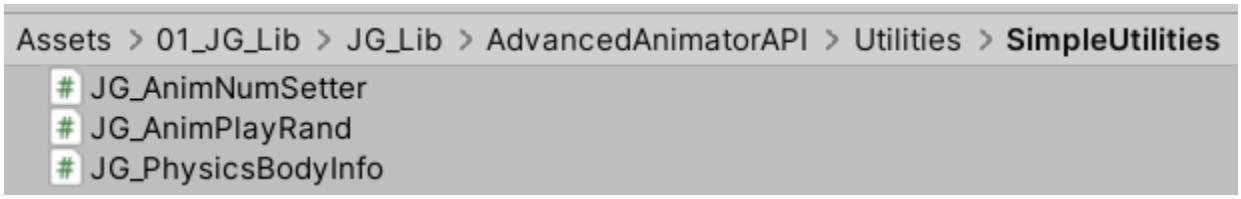
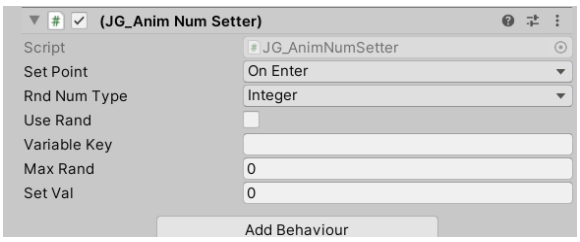


Fig 21: SimpleUtilities Subdirectory

This subdirectory contains scripts that do not inherit from JG_AnimatorStateInfo.

JG_AnimNumSetter:

This script can be added to any animator state, and it will change the number or float, based on inputs.



JG_AnimPlayRand:

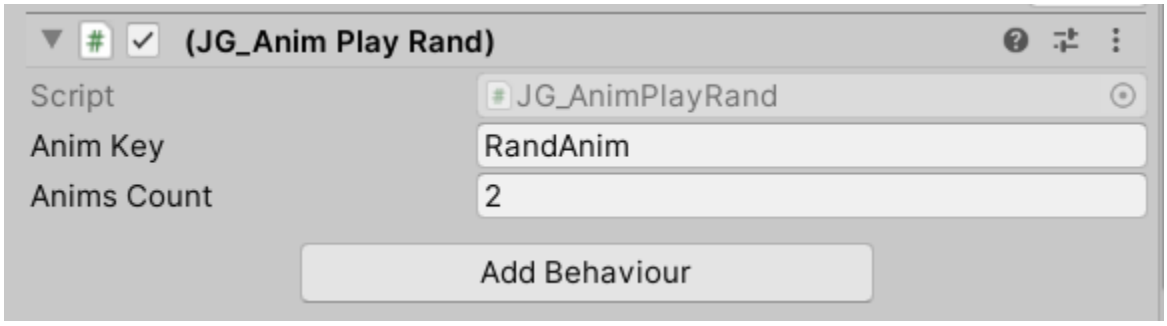


Fig 22: JG_AnimPlayRand assigned to an animator state

This script will assign the animator integer value of "**Anim Key**" to a random value between "**0"** and "**Anims Count**" variable. (returned value will be between 0, and animsCount-1) meaning it's exclusive.

JG_PhysicsBodyInfo:

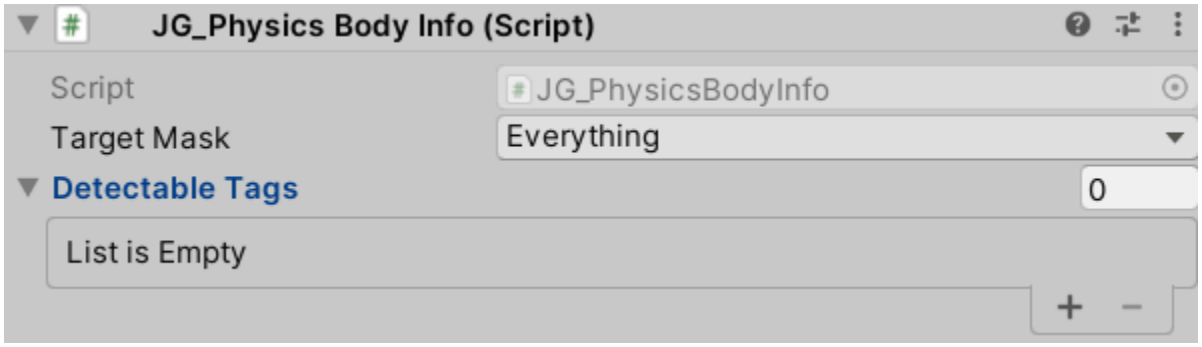


Fig 23: JG_Phyics Body Info Component

Is a simple script that gets attached to a monobehaviour, and it will expose a **TargetMask** and **DetectableTags**, it's sort of an information container for typical physics based filters.

Direct Usage Examples

Accessing Animator Info From Code:

In many situations while you are building your own game, you might want to access certain information from a specific animator state, or you might want to get the currently running animator state, and find out maybe the duration of the animation, or for how long it has been running, or the animator state speed, advanced animator api allows you to access all of that and more.

First you start by defining a reference to keep track of the animator state info collector as follows (it is just a simple member variable).

```
        JG_AnimatorStateInfoCollector infoCollector = null;
```

Then from any function where you want to access the information you can just write the following line to get the currently running animator state info:

```
        activeInfo = animator.GetRunningAnimatorStateInfo(ref infoCollector);
```

Note that, you need to use the JG_Lib.Utility and JG_Lib to be able to access this new extension.

Let us now define 2 more variables, the animator itself, and something to hold the active currently active state, so now we have a total of three variables as follows:

```
        [SerializeField] Animator animator = null;
        JG_AnimatorStateInfo activeInfo = null;
        JG_AnimatorStateInfoCollector infoCollector = null;
```

Then let us say we want to find the currently running animator state and update some text every frame, you can do so in the Update function, we can easily do so as follows:

```
void Update()
{
    //Only get the newest one when needed(in other words, when there is no info, or
info was exitted)
    if (activeInfo == null || !activeInfo.isEntered)
         activeInfo = animator.GetRunningAnimatorStateInfo(ref infoCollector,0);
```

Note: zero that is plugged as second parameter is an optional parameter that means we get the running animator state in the first layer of the animator.

Now, we can expand our function so that, it updates some texts every frame, first we need to expose the texts as follows:

```
[SerializeField] TextMeshProUGUI stateName = null;
[SerializeField] TextMeshProUGUI clipName = null;
[SerializeField] TextMeshProUGUI animDur = null;
[SerializeField] TextMeshProUGUI animRunTimeAsPercent = null;
[SerializeField] TextMeshProUGUI animSpeed = null;
```

Now back to the update function, we can fill these these texts with the currently running animator state as follows:

```
stateName.text = "State Name : " + activeInfo.GetStateName;
clipName.text = "Clip Name " + activeInfo.GetClipName;
animSpeed.text = "State Speed " + activeInfo.GetStateSpeed;
animDur.text = "Anim Dur " + activeInfo.GetAnimDur;
animRunTimeAsPercent.text = "Run Time (No Loop) Percent " +
                            activeInfo.GetRunTimeAsPercent.ToString("0.00");
```

There are many more properties to check, such as "**isEntered**" which we have already used in the Update function, to which we use to check whether the "**JG_AnimatorStateInfo**" instance is entered or not, because as we move between animator states, we might leave the state, so using the isEntered we can know whether it is running/entered or not.

The "**GetRunTimeAsPercent**" property will return a value between 0 and 1, where 0.99 means the animation is almost done, and a value of 0.01 means the animation has just started and so on, the full script is in our Z_Demos directory under the name of **Demo_01_InfoDisplayer**.

Note: in the update function that we used in this example, we only update the state when needed, but do keep in mind, the entire state of the animator is cached automatically, so even if you query it every frame, there will not be an impact on performance, because you are accessing them from the cache rather than getting the value directly, so essentially it is as fast as Dictionaries and Lists are, and obviously if you have an animator with 1000 animator states, then things might get slower.

Unity Messages As Events Example:

Unity offers us Events in the animations, but, if you were to change the clip, then you'd need to recreate the animation events on the new clip as well, which is time consuming, and easy to miss, however advanced animator api allows you to call functions the same way you'd call them from animation events with the key difference that they're tied to the animator state, not the animation clip, meaning even if you change animation clips, the functions you call through the animator state, will remain, allowing for vastly improved workflow, and as a bonus, you can optionally call events and respond to them through the inspector as well, here is a simple setup from one of the demos.

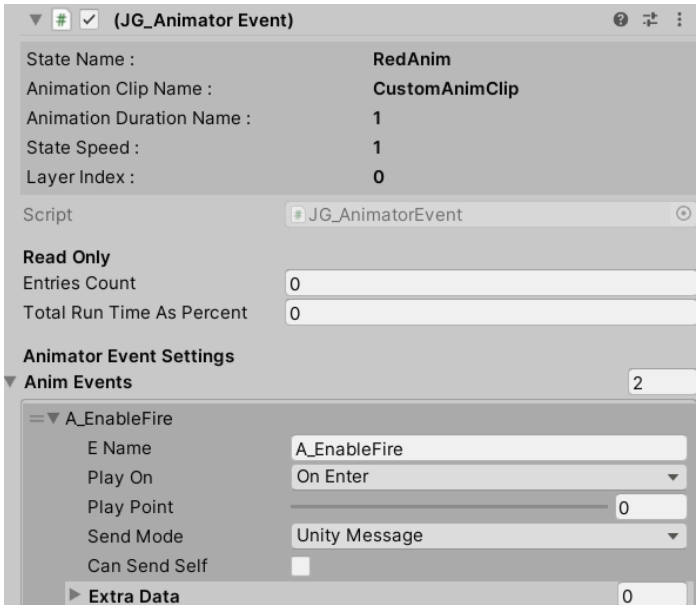


Fig 24: JG_AnimatorEvent used to send a unity message **A_EnableFire**

Now in the code, we attach any monobehaviour on the animator, and ensure that it had defined a function named "**A_EnableFire()**" as follows

```
//Functions called from RedAnim Animator State
private void A_EnableFire()
{
    fire1.Play(true);
}
```

For example, in the **Demo_02_Messages.cs** script from our demos directory, we have defined this function, which simply plays a particle system.

Animator Events Using Inspector Example:

But for simple calls, like playing audio, playing particle systems, or anything that is a simple oneshot, it might be easier and quicker to just call it through the inspector, and with advanced animator api, we offer you a full solution for that.

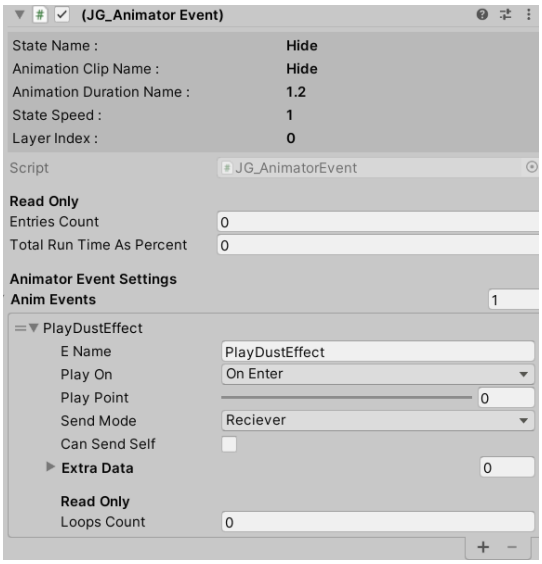


Fig 25: JG_AnimatorEvent used to send a message named “PlaydustEffect” to our custom receiver

Then from any game object in the scene, we need to attach the component named **JG_AniamtorEventReciever** to it, then plug the animator, and then call any functions we want similar to **On Click**, from unity’s buttons.

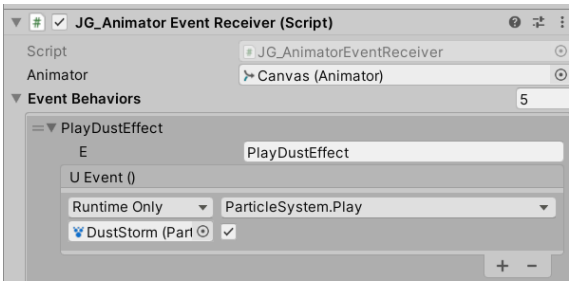


Fig 26: JG_AniamtorEventReciever attached to a game object, and responds to **PlayDustEffect**

For this entire example, simply head over to **02_AnimatorEvenmts_ByReciever** scene in our demos.

Simple Tags Example:

Let us start by creating a simple abilities system, utilizing tags, so if we press 1 or 2 or 3, then we will play a specific animation, and if the animation is tagged with a certain tag, we will change the player to respond maybe we will make him take no damage if he has immortality tag, or we will will make it so nothing can collide with the player if he has the ghost tag and so on, and that will be our end goal.

```
using UnityEngine;
using System.Collections.Generic;
using JG_Lib;
```

First we assign the proper name spaces., we need the JG_Lib namespace, so we can use the Extension methods that are provided by the package, you can also use them, using the static class Extensions,

```
        [SerializeField] KeyCode ability1 = KeyCode.Alpha1;
        [SerializeField] KeyCode ability2 = KeyCode.Alpha2;
        [SerializeField] KeyCode ability3 = KeyCode.Alpha3;

        [SerializeField] Animator animator = null;

        [SerializeField] float speedyMaxSpeedMulti = 1.5f;
        [SerializeField] float speedyAccelMulti = 1.5f;
        [SerializeField] List<Collider2D> cols = new List<Collider2D>();

        Demo_03_ShipMover mover = null;
        Demo_03_Health health = null;

        float defaultMaxMoveSpeed;
        float defaultAcceleration;

        public bool IsGhost { get; private set; } = false;

        JG_AnimatorStateInfoCollector collector = null;
```

After that we we defined some variables

All variables of type **KeyCode** are used to define the keyboard button that we must press to activate the ability.

Animator: variable is the animator that we will play the animations on.

speedyMaxSpeedMulti: is the multiplayer that we will apply on the movement speed when we are in an ability that increases the speed.

speedyMaxAccelMulti: is the multiplayer that we will apply on the acceleration when we have an ability that increases the speed.

cols: is a list of colliders in the ship

mover: is a component responsible for moving the ship.

health: is the component responsible for the health of the character

defaultMaxMoveSpeed: is the move speed at the start of the game.

defaultAcceleration: is the acceleration of the ship at the start of the game.

IsGhost: is a property that expresses information about whether a ship is in ghost mode or not.

collector: is the variable to use as ref, by extensions (will be cleared down below).

```
        private void Awake()
        {
            mover = GetComponent<Demo_03_ShipMover>();
            animator = GetComponentInChildren<Animator>();
            health = GetComponent<Demo_03_Health>();
            defaultMaxMoveSpeed = mover.maxMoveSpeed;
            defaultAcceleration = mover.acceleration;
        }
```

In awake function, we initialize all the components, and set the default max move speed and default acceleration as well.

```
void Update()
{
    if (Input.GetKeyDown(ability1))
    {
        animator.SetTrigger("Speedy");
    }
    else if (Input.GetKeyDown(ability2))
    {
        animator.SetTrigger("Ghost");
    }
    else if (Input.GetKeyDown(ability3))
    {
        animator.SetTrigger("SpeedyGhost");
    }
```

Then in the update function, we read the **KeyCode** variables, and play an animation based on each one.

Next, and this is a critical part, for using the tags in code.

```
    if (animator.IsAnimatorTagActive(ref collector, "SpeedBoostTag"))
    {
        mover.maxMoveSpeed = defaultMaxMoveSpeed * speedyMaxSpeedMulti;
        mover.acceleration = defaultAcceleration * speedyAccelMulti;
    }
    else
    {
        mover.maxMoveSpeed = defaultMaxMoveSpeed;
        mover.acceleration = defaultAcceleration;
    }
```

In the line `animator.IsAnimatorTagActive(ref collector, "SpeedBoostTag")` we test the animator, whether a tag is running or not, this is the core of the entire script utilizes the **AdvancedAnimatorApi** in here we essentially check whether there is an animator state, that has the animator tag “SpeedBoostTag”, and if there is, then we get a true, thus we are in animation that modifies the speed, thus we access the mover and update it's speed.

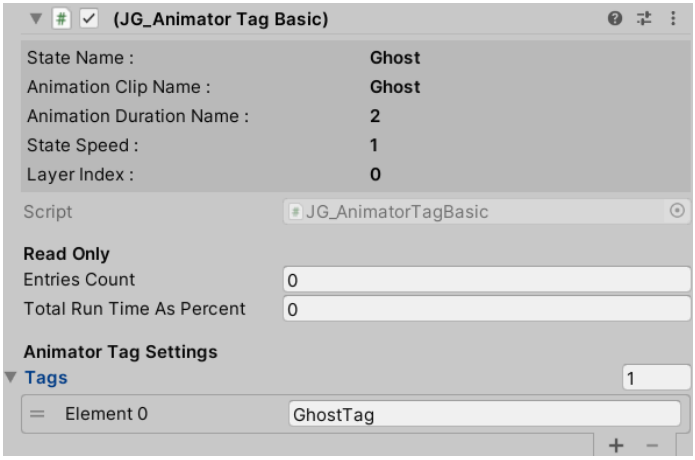
If this is false, then we reset the speed to the default values that it had at the start of the game.

In the same manner, we can expand our abilities script as follows:

```
            if (animator.IsAnimatorTagActive(ref collector, "GhostTag"))
            {
                health.isImmortal = true;
                IsGhost = true;
                foreach (Collider2D c in cols)
                {
                    if (c.enabled)
                        c.enabled = false;
                }
            }
            else
            {
                health.isImmortal = false;
                IsGhost = false;
                foreach (var c in cols)
                {
                    if (!c.enabled)
                        c.enabled = true;
                }
            }
```

Again the only code that is unique to Advanced Animator API is `animator.IsAnimatorTagActive(ref collector, "GhostTag")` where as all of the other lines are some lines unique to the actual ability, now, with this, it does not matter which or what animator state is running, as long as the animator state is tagged with “GhostTag”, then this ability will run, which will essentially turn the character into a ghost that can not take damage, this entire script is in the demo scenes, under the name **Demo_03_PlayerAbilities.cs** which you may use as reference.

And here is an example of what the animator state looks like, you just attach one of the tag behaviors (in the picture case we used **JG_AnimatorTagBasic**, to the animator state itself, and from code, you just deal with it as we did in this example.



Body Parts Controller Example:

Now if you are making some game, that has some sort of melee combat, you may want to activate/deactivate game objects at key moment in the animation, now you can achieve that using the animator events discussed previously, but a more automatic easier to track way is using a combination of three scripts, first **JG_BodyPartMono**, which you attach to the game object it self.

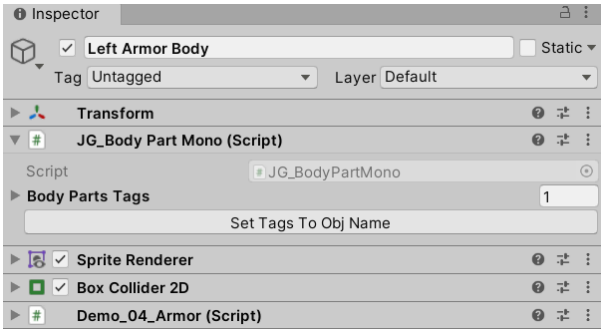


Fig 27: JG_BodyPartMono attached to the game object we want to control

Then the second script is **JG_BodyPartsController**, which must be attached to a game object that is a parent of all other **JG_BodyParts** it is okay if it's not a direct parent but it must be a parent, and preferably, it also must be next to the animator component.

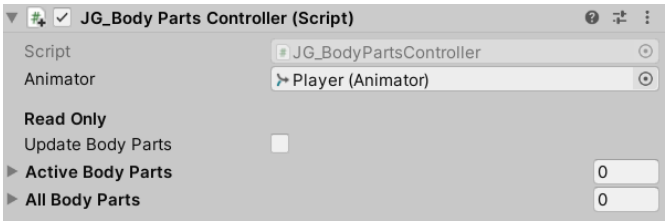
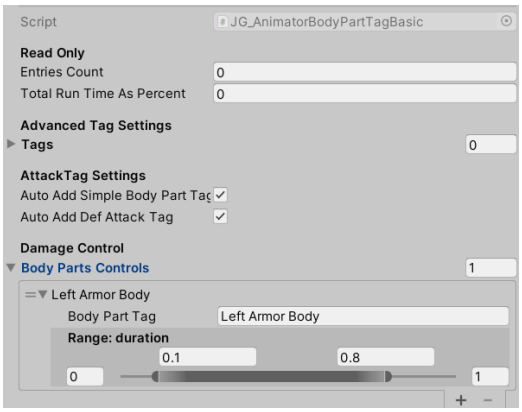


Fig 28: JG_BodyPartsController attached to a common parent for all **JG_BodyPartMono**

And finally for the **JG_BodyPartTagBasic** which gets attached to the animator state itself, and controls the duration/time in which we activate the game object tagged as **"Left Armor Body"**



There is a full blown demo scene, that is a mini twin stick shooter game, under our Z_Demo directory and the scene is named **04_BodyParts**, which you may use to check this in full details.

Send Custom Data With Unity Messages As Events Example:

In the previous example, **Unity Messages As Events Example**, we sent Unity Messages, to call functions, but occasionally you might want to send data along with the function call, You can tick the "**Can Send Self**" and it will the instance of the class itself, and if you want to send more customized data, you will need to expand the JG_Animator behavior that you attached to whatever animator state of your choice

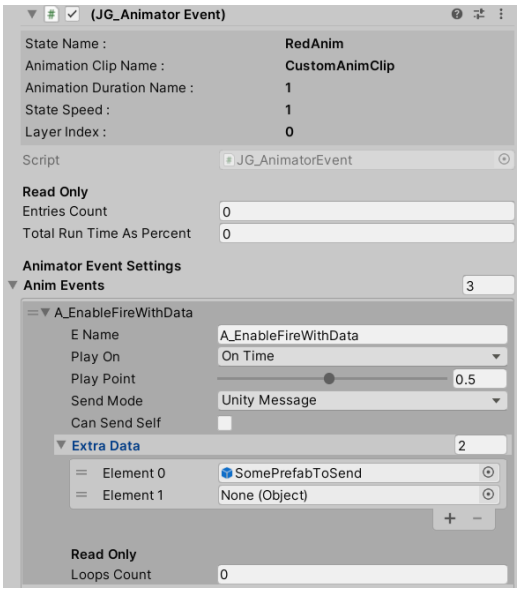


Fig 29: JG_AnimatorEvent with Extra Data Unfolded and expanded

Using this field, you can use it to send any unity based type, such as Scriptable Objects, and prefabs, However, sometimes you just want to send some simple variables, like a float or a string, for that, i created a set of variables that are stored in a scriptable object, so in the event you send the scriptable object, and then access the value, first you need to create the scriptable object, lets say, we want to send a string along side the message, so we need a string variable, and we need some duration, so we also need a float.

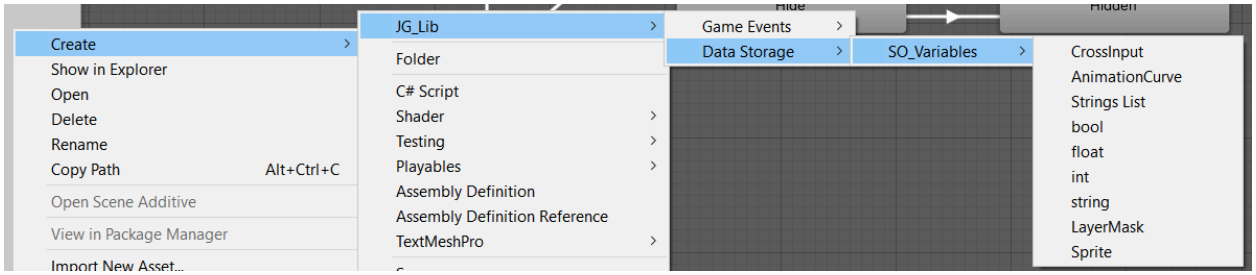


Fig 30: Picture of menu that we can use to create primitive variables stored in scriptable objects

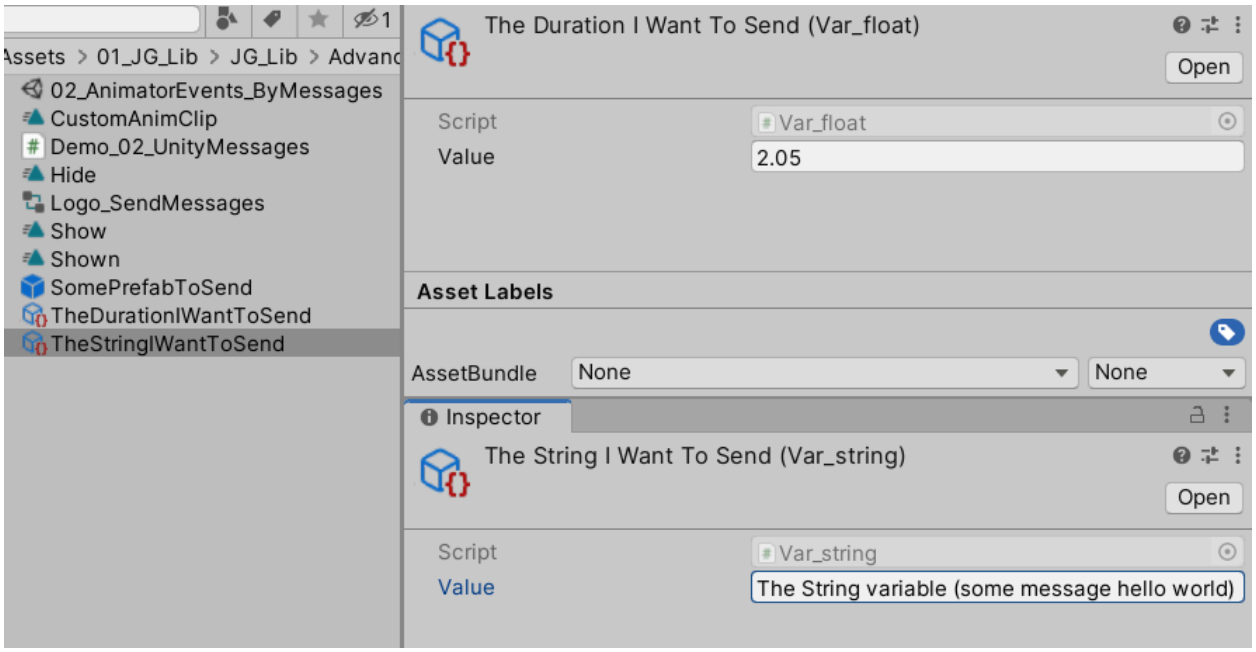


Fig 31: Picture of project files after we created the two new variables (string and float)

Now we created these two variables, now we can easily go back to the inspector of the animator state, and access the animator event, and just attach this new data, which is the two scriptable objects.

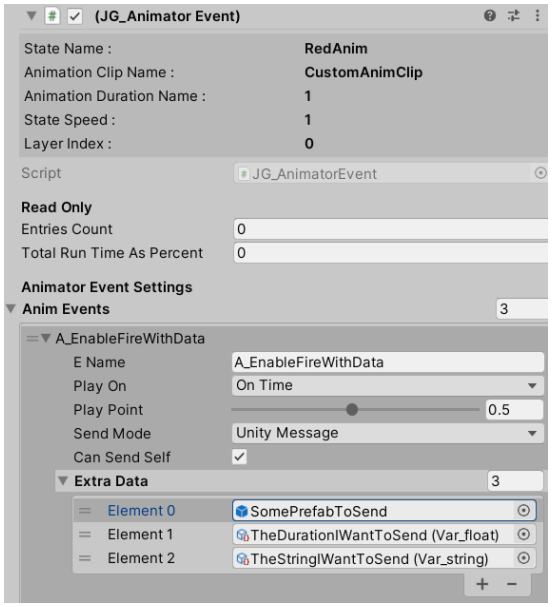


Fig 32: Picture of updated event where CanSendSelf is true, and we send 1 prefab and 2 scriptable objects.

Now finally we want to access this data from the code, from **Fig 32** we can see that we send 4 things, first "**Can Send Self"** is true so we send the state machine behavior itself, then we have the prefab named "SomePrefabToSend" and we have 2 variable "TheDurationIWantToSend" and "TheStringIWantToSend", where the two variables, are two scriptable objects, that contain the primitive data that we want to send, now in our code we define our function as follows:

```
private void A_EnableFireWithData(List<object> objs)
{
    JG_AnimatorStateInfo info = (JG_AnimatorStateInfo)objs[0];
    GameObject thePrefabThatIRecieved = (GameObject)objs[1];
    Var_float theFloatIRecieved = (Var_float)objs[2];
    Var_string theStringWeRecieved = (Var_string)objs[3];
}
```

Now from our code, we are free to use the data however we like, for example, if we want to debug the data, wae can do as follows

```
Debug.Log("Name Of State, That Message Was Called In " + info.GetStateName);
Debug.Log("Name Of Prefab We Received " + thePrefabThatIRecieved.name);
Debug.Log("Duration That We Received As Custom Var " + theFloatIRecieved.Value);
Debug.Log("String We Received As Custom Var " + theStringWeRecieved.Value);
```

And that would be the way, to receive extra data within the animator events, naturally, all of this section is optional, if you want to just call a function, without sending any data, you can easily do so as mentioned in the **Unity Messages As Events Example**, one thing to keep in mind, is that the Inspector based receiver, will not receive any of the of this data, this data is only supported in the Unity Messages format.

Of course as per usual, the entire example is included in the package under the name of **Demo_02_Messages.cs**.

Note: Null data in the list of "**Extra Data**" gets fully ignored, as if the element does not exist, we do not send nulls in the `List<object> objs`.

Send Unity Messages As Events To Different Game Objects Example:

Typically, you send the unity messages to different to the game object that has the animator itself, but if we want to delegate it, so we send the message to a different game object, than the one that holds the animator, then for that case all we need to do is attach **JG_AnimMessagesDelegator** component to the game object that holds the animator, and then plug the "target" transform value in it, and the message will be sent to that transform.

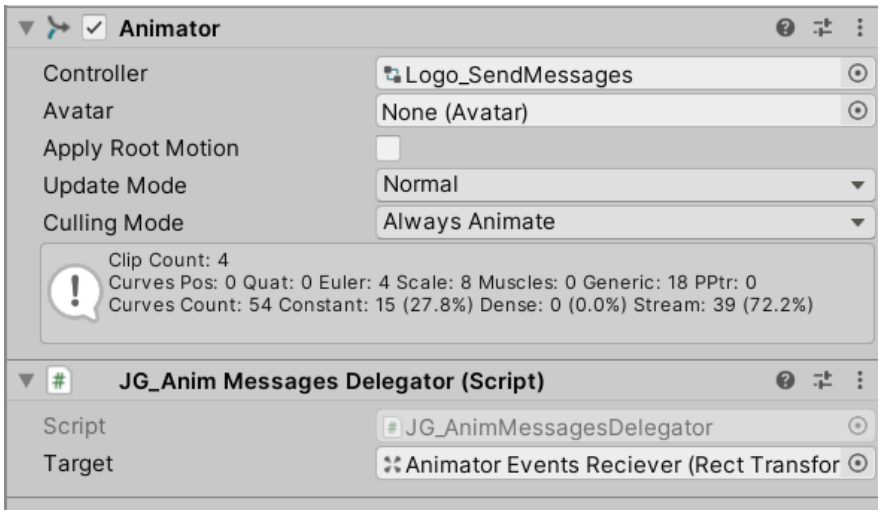


Fig 33: Picture of **JG_AnimMessagesDelegator** component attached next to the animator and pointing towards a game object named Animator Events Receiver.

Now, if we take a look at that plugged game object, we can see that it only contains the script that has these messages functions so it can listen to the animator messages.

Fig 34: Picture of the component on **Animator Events Receiver** game object.

Now that script will handle listening to all animator events despite the fact that it is not attached to the animator, the full example is in **02_AnimatorEvents_ByMessages** scene.

Extra Content

Gameplay Related:

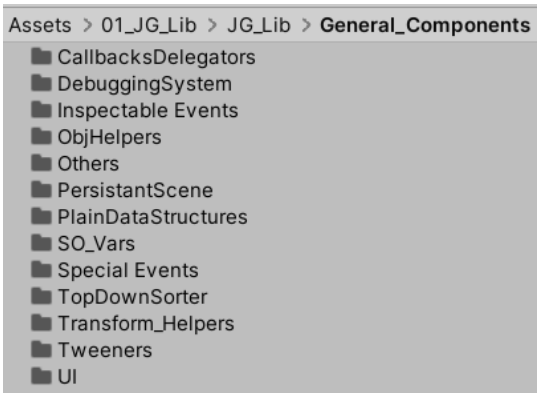


Fig 34: Picture of the directory that contains the extra content

This directory, over a 100 helper scripts that are unrelated to the actual content of **Advanced Animator API** you may use these scripts to help you advance your project, they're modular components that are easy to digest and use, and here is a brief overview on them.

CallbacksDelegators: unity built in functions/events are typically called on specific game objects example OnTriggerEnter is called on functions that has a collider, you can use some of the scripts in this directory to delegate these calls to different game objects through C# Events.

DebuggingSystem: it is self explanatory, this is a full blown debugging system that you can use for your game, you can debug messages based on channels and priority.

InspectableEvents: are game events, that are created as scriptable objects, and you can connect to them by C# code, or by custom scripts, and you can determine a priority for functions that are being called.

ObjHelpers: are a group of smaller properties you can use in other scripts, such as DoubleBool that is a class that wraps two bools, or SqrFloat which is a float that automatically wraps and gets the squared value, and the MinMaxRange class which you have seen used in some of the classes that inherit from JG_AnimatorTag.

Others: is a directory that contains a bunch of scripts that do not fit any of the other categories, like GOSwitcher which switches a game object or GoCacher, which caches a game object, so it can point to another.

PersistantScene: a small system where you can create a "**PersistantScene**" prefab that can hold all your persistent objects.

PlainDataStructures: contain classes that wrap around some data structures, currently it wraps around KeyValuePairs, which should help you create serializable like dictionaries, which can be used in a saving system.

SO_Vars: is the directory that contains the scriptable object based variables that was discussed in the example where we send data with the Unity Messages.

Special Events: are a group of scripts that extend around the **UnityEvent** class, which allows you to have serializable function calls, but with extra functionality, like wrapping it with a C# event or extending its sendable types, so you have things like UnityColliderEvent, which sends a collider.

TopDownSorter: if you are making an isometric or 2.5D game, then this directory is super helpful for you, you can use this which is a simple script, to automatically update the sorting order and priority of any sprite renderer, dynamically, so you won't face any problems with characters overlapping each other, or your player being stuck behind environment and so on.

Transform_Helpers: a collection of scripts that act as utilities for transforms such as TransformMatcher which will make objects follow each other, or ChildrenTagApplier which will apply the tag to all children, and so on.

Tweeners: is a group of scripts that require DOTween to work, and it contains mainly a TimeScaler to help you control the time scale of your game smoothly.

UI: it should be self explanatory, and it's a directory to contain helper scripts for the ui.

Editor Related:

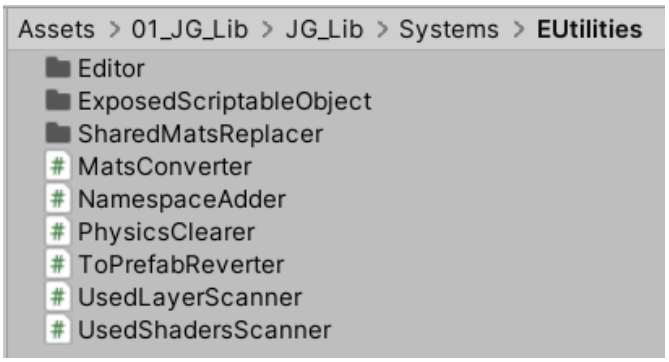


Fig 35: Picture of the directory that contains helper scripts

These scripts are mini editor tools that help you speed up the way you create game, by doing repetitive tasks, such as finding game objects that use specific layer, or finding shaders, or replacing materials or adding a specific namespace to a group of scripts, or reverting a group of prefabs to their prefab state all at once.

Contacts:

Any questions, suggestions or feedback?

Feel Free To Send Me An Email: info@jondobgames.com

Or Join Our Discord: https://discord.com/invite/tCnmcnVpcE

Link On The Asset Store Is : https://assetstore.unity.com/packages/slug/254444

Graphics And Visuals Link:
https://drive.google.com/drive/folders/1oKfsejklOqpIafMDmzrYjCaPUHly5rs_

Full Tutorials Playlist Link:

https://www.youtube.com/playlist?list=PLNVG78sXvnG0rJE6FC15D-uelqZEhlv5A

Please leave a **review** on the asset store for the asset, as it helps us improve the package and shows your support, thank you very much <3.