MECANIM CALLBACKER

USER MANUAL

Holy Shovel Soft (2018)

Table of content

1. About

To first page

Mecanim Callbacker is a complex, professional and flexible solution for tracking any events happening in the Unity Mecanim Animator as well as creating custom animation state-based curves, events and procedures." This is a pretty accurate definition of what **Mecanim Callbacker** is. Accurate definition, but not the most straighforward.

Primary goal of this asset is to help **Unity** developers when working with Mecanim Animator. Quite often big animation controllers cause hardships when writing code, controlling them, and we have to either simplify controller itself, or to load code into the so-called "if-else hell". In particular, by separating the design of animated objects and code that controls those objects, and by giving access to the wide selection of callbacks, our asset helps to avoid both of those situations. Also, this asset lets you to majorly simplify and speed up almost all stages of prototyping that relate to animating objects in **Unity**.

This asset helps you to:

- Get precise information about what's going on inside Animator at a given moment and to perform only code that is necessary for precisely that state of the object.
- Create curves and events, that are tied to states, not animations, which can be affected by parameters of Animator itself.
- Create procedures, code of which is completely depended on the current state of the animated object, and then use those procedures in any part of the code.
- Get information not only about the entities of the asset itself, but also "raw" data of Animator on object level with larger amounts of information, than those provided by standard StateMachineBehaviour.

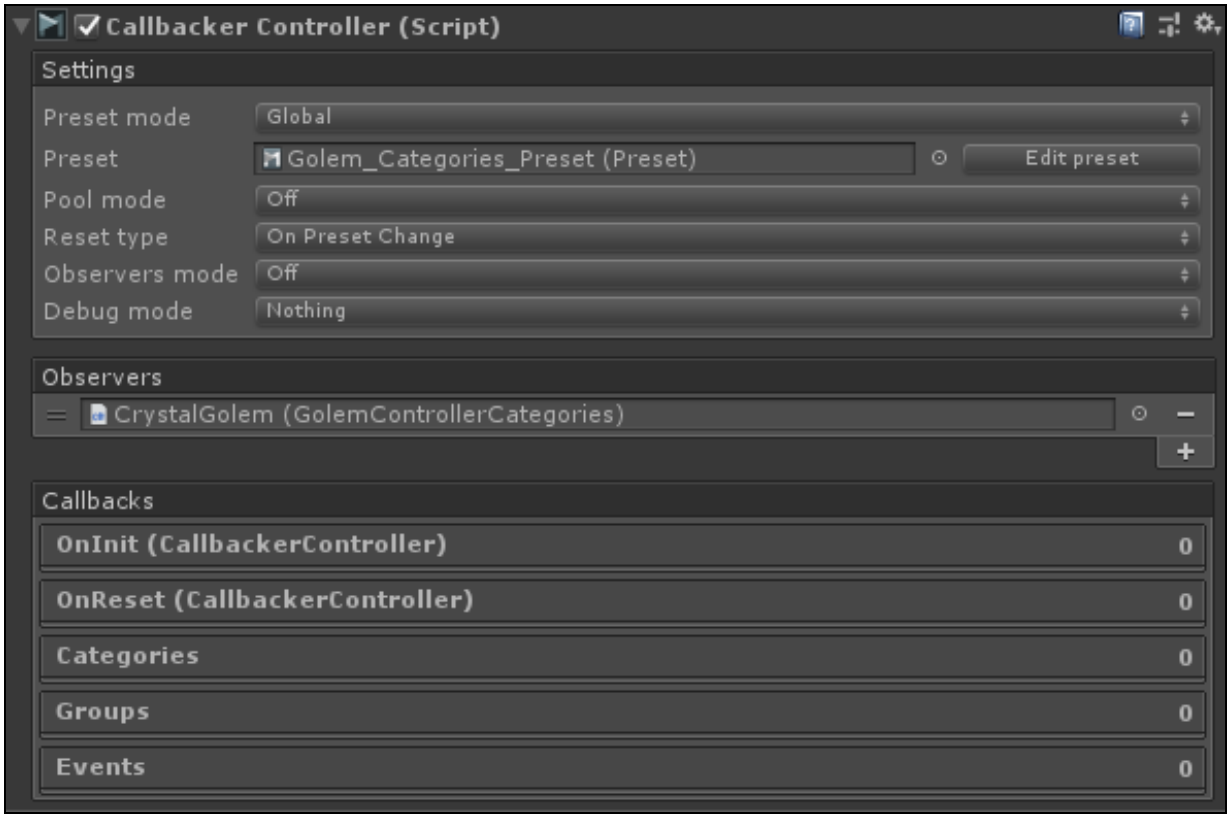
> An importanant part is the fact that this asset DOESN'T replace Mecanim, neither it changes its functions. This asset does not work with Legacy Animations, nor it is not analagous to Locomotion System. In many cases this asset speeds up creation of code and development of systems working with Mecanim, but does not change approach to creation of animation FSM.

You can learn in further detail about all elements in the following sections.

1.1. Controllers

To first page

Primary and most important component of the asset is the CallbackerConrtoller. It's located in the components menu: **Holy Shovel Soft > Mecanim Callbacker > Callbacker Controller**.



Primary functions of this component:

- Processing of all events, happening in Mecanim Animator.
- Processing of all event observers, registered in this controller.

The most of often scenario of using this component is the following pipeline:

1. To register event observer in the controller in one way or another.
2. Wait for controller initialization.
3. Create/get necessary entities and register existing callbacks.
4. When reseting, clear out links to cached entities.

Now, to talk more in detail about settings.

Preset mode

This setting defines which preset will be used for controllers work. (You can read more about presets themselves and their work in the next section)

When selecting option **Off**, the preset won't be used at all. This presumes that all entities are created in runtime, using code.

Local offers saving the preset "inside" the component without creating assets within the project. Local preset can be used in the scenario when this controller is attached to highly specialized object and nowhere else will such a set of entities be used.

Unlike local preset, **Global** option lets you use presets as saved within the project and to reuse it on different objects, if needed.

> In the case of switching from **Local** to **Global** or back, options will be offered to clear out all current settings, as well as options for convertion of the preset.

Pool mode

This option defines which way the major mass of internal objects, required for controllers work, will be instantiated. In common cases, it's nothing you need to worry about and set the option to **Off**, as the objects are only created during initalization and creation of entities from code, which usually happens at scene start. Usually when exiting the scene, these objects are collected into the Garbage Collector. But if your game logic dictates that objects with this controller are often destroyed and created, then to lessen GC calls the good option would be to turn on one of the following options.

Global - creates all objects within the limits of one pool, common for all controllers, which will exist throughtout the entire game session. This option is useful not only if you often recreate objects with this controller, but also if the average number of concurrent objects is roughly similar during the game session. In this case the number of entities in the pool won't constantly increase and at a particular stage the pool would be able to service all future controllers.

Local - creates local pool for each controller, which will be used only by that controller. This options doesn't help to avoid unnecessary Garbage Collecting during constant recreation of objects with this controller, but will help during it's constatnt turning off / on / calling Reset method. As every reset with consequent turning on calls initialization of all objects anew, this option helps to keep the number of created objects on the "previous level" without their regular reset via GC.

> It's important to understand, that when using pools, all entities that were created or got from the controller, when resetting the controller, can be reused. Hence, when resetting the controller it's necessary to clear out all cached entities.

Reset type

(You can read more about resetting and initialization in the section dedicated to Event observers)

This setting defines which way and when controller reset will be called

On Preset Change - reset will be called if during game sessions preset has changed. **preset mode** is defined in Global state.

On Animator Change - reset will be called if during the game session AnimatorController in Mecanim Animator will change.

On Both Change - as is clear from the name, the controller reset will be called if either preset or AnimatorController changes.

By Script - in this case controller won't be reset automatically. Reset could be initiatied by calling method Reset. This method works only when this option is chosen.

Observers mode

This setting lets you run automatic registration of Components, which implement interface of IEventObserver. In the case of **Off** search won't be performed. **Self** will look through all Components of this object. **Self and Childs** will look through all Components of this object and that object's children.

Debug mode

This option works only in the editor of Development Build (with some restrictions of output of information). After choosing one or several of given options, you can receive information about processes inside the component. By selecting **Event Fire**, for example, you will see logging of calls of all events, created within this controller.

Observers

This massive lets you register components of event observers, that were inherited from **MonoCallbackerObserver**. This component itself implements all interfaces, available to **IEventObserver**

Callbacks

List of all callbacks available to this controller in the type of **UnityEvent**. Equivalent to analogous callbacks available from the code, hence to subscribe to events in most cases you only need to register your methods in this controller. Also these **UnityEvent**s can be very useful during prototyping stage.

1.2. Event observers

To first page

All work with **Callbacker Controller** usually happens within classes that implement interface IEventObserver. To be more accurate, you need to implement either of two (or both, depending on the situtation) interfaces:

> If registration of implementation of interface in the controller is done by hand, then class doesn't have to be inherited from **MonoBehaviour**. Also, the abstract class **MonoCallbackerObserver** already implements all necessary interfaces and by inheriting from it, you will get all functions, available to observers. Also, all components available to this type can be added to the list of automatically registered observers in the controllers interface.

1. **ICallbackerEventObserver** - this interface lets you to track events of the initialization of the controller and its reset. This interface is the primary one for working with controller. It is assumed that at controllers initialization, implementation of this interface will create and/or get all necessary entities from controller for it to work, and will free them up and at reset.

```
//Sample
public void OnAttached(CallbackerController controller)
{
    var categories = controller.Categories;
    var canWalkCategory = categories["Can Walk"];
    var canAttackCategory = categories["Can Attack"];
    canWalkCategory.BaseEvents.OnTick += OnCanWalk;
    canWalkCategory.BaseEvents.OnEnd += OnCanWalkEnd;
    canAttackCategory.BaseEvents.OnTick += OnCanAttack;
    canAttackCategory.BaseEvents.OnEnd += OnCanAttackEnd;
}

public void OnDetached(CallbackerController controller)
{
    var categories = controller.Categories;
    var canWalkCategory = categories["Can Walk"];
    var canAttackCategory = categories["Can Attack"];
    canWalkCategory.BaseEvents.OnTick -= OnCanWalk;
    canWalkCategory.BaseEvents.OnEnd -= OnCanWalkEnd;
    canAttackCategory.BaseEvents.OnTick -= OnCanAttack;
    canAttackCategory.BaseEvents.OnEnd -= OnCanAttackEnd;
}
```

2. **IAnimatorEventObserver** - unlike the first interface, this interface helps to track "raw" events happening in Mecanim Animator. In most cases we recommend to use **ICallbackerEventObserver** and entities of the controller, but sometimes it could be useful to get unprocessed data.

```
//Sample
public void  OnStateTick(CallbackerController controller,
Animator targetAnimator, AnimatorStateInfo stateInfo, int layerIndex)
{
    if (stateInfo.IsName("Walk"))
    {
        Debug.Log("We walk right now!");
    }
}
```

You can add and delete observers in runtime, for this controller has two methods: **AddObserver** and **RemoveObserver**. Due to the pecularities of processing, **IAnimatorEventObserver** is added and deleted not immediately after calling those methods, but during the next iteration of processing.

A bit more about OnAttach/OnDettach

While methods of **IAnimatorEventObserver** interface are pretty straightforward - an event happens in animator, observer calls corresponding method; then correct understating of **ICallbackerEventObserver** can affect how well and comfortably you can use **MecanimCallbacker**. This observer doesn't give information about events inside the animation, but rather tells when you can begin to subscribe to events of controllers entities, and when you need to unsubscribe and empty cached entities and data (when using pools).

OnAttach is called when controller began to work and is correctly initialized, if this observer was registered in it. If observer was added to already initialized controlled, then this method will be called during the next iteration of the event processing.

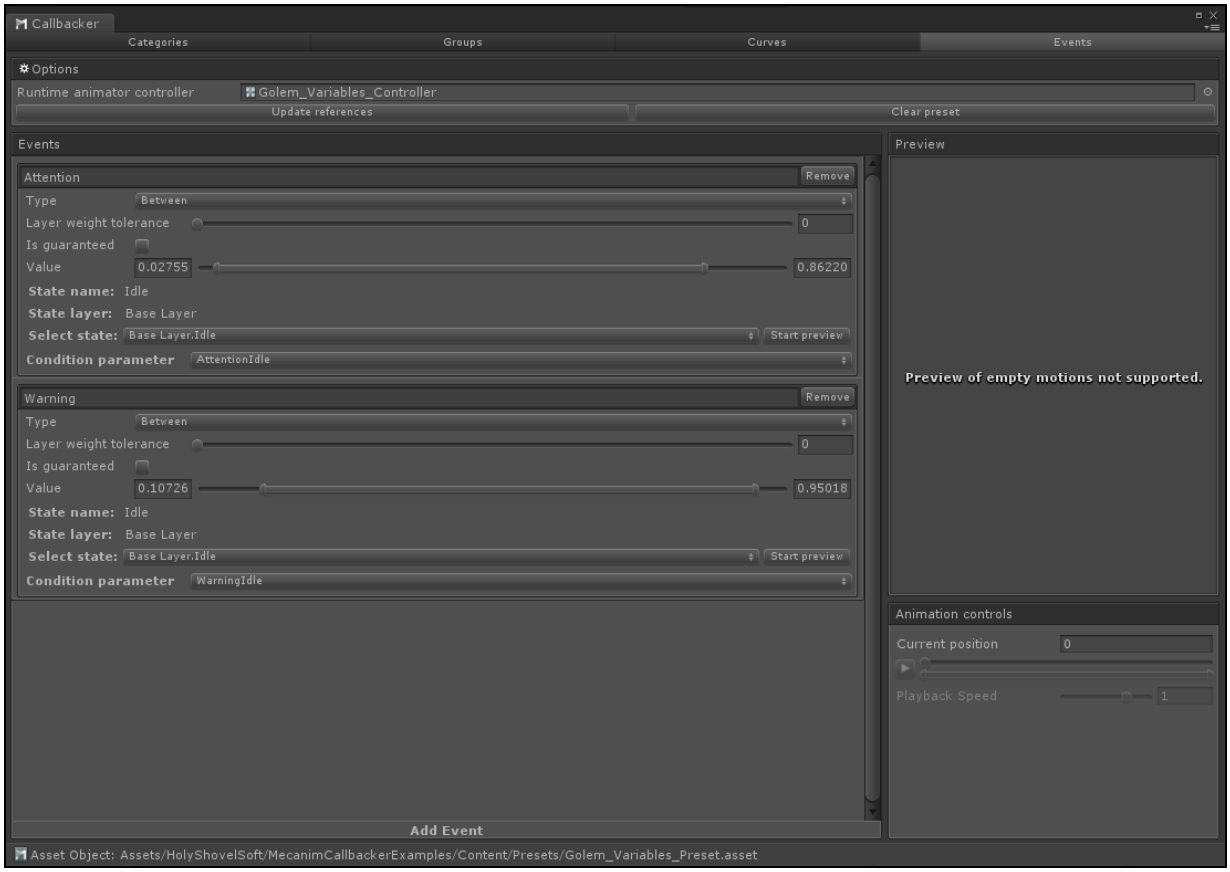
OnDetach will be called if observer was deleted from working controller, or if controller was disabled/destroyed/reset.

Methods of **IAnimatorEventObserver** are pretty straightforward - event happens in animator, observers call corresponding method. Correct understanding of **ICallbackerEventObserver**, however, is paramount to how correctly and easily you can use **MecanimCallbacker**. This observer doesn't give information about events inside animator, but rather, notifies when you can start subscribing to events of controller entities, and when you need to unsubscribe and clean cached entities and data (when using pools). So: OnAttach is called when controller started its work and is correctly initialized, if at that moment observer was registered in it. If observer was added to an already initialized controller, then this method will be called during the next iteration of event parsing. OnDetach will be called if observer was deleted from a working controller, or controller was disable/destroyed/reset.

1.3. Presets

To first page

Common Info



Presets are objects, made for creating and holding entities which controller will be working with in editor. Logic is not contained in the preset, it's only a container for data. You can see it as parallel to this: **Animator** animates an object based on the data from **AnimatorController**, and CallbackerController tracks events and performs actions based on the data held in **Presets**

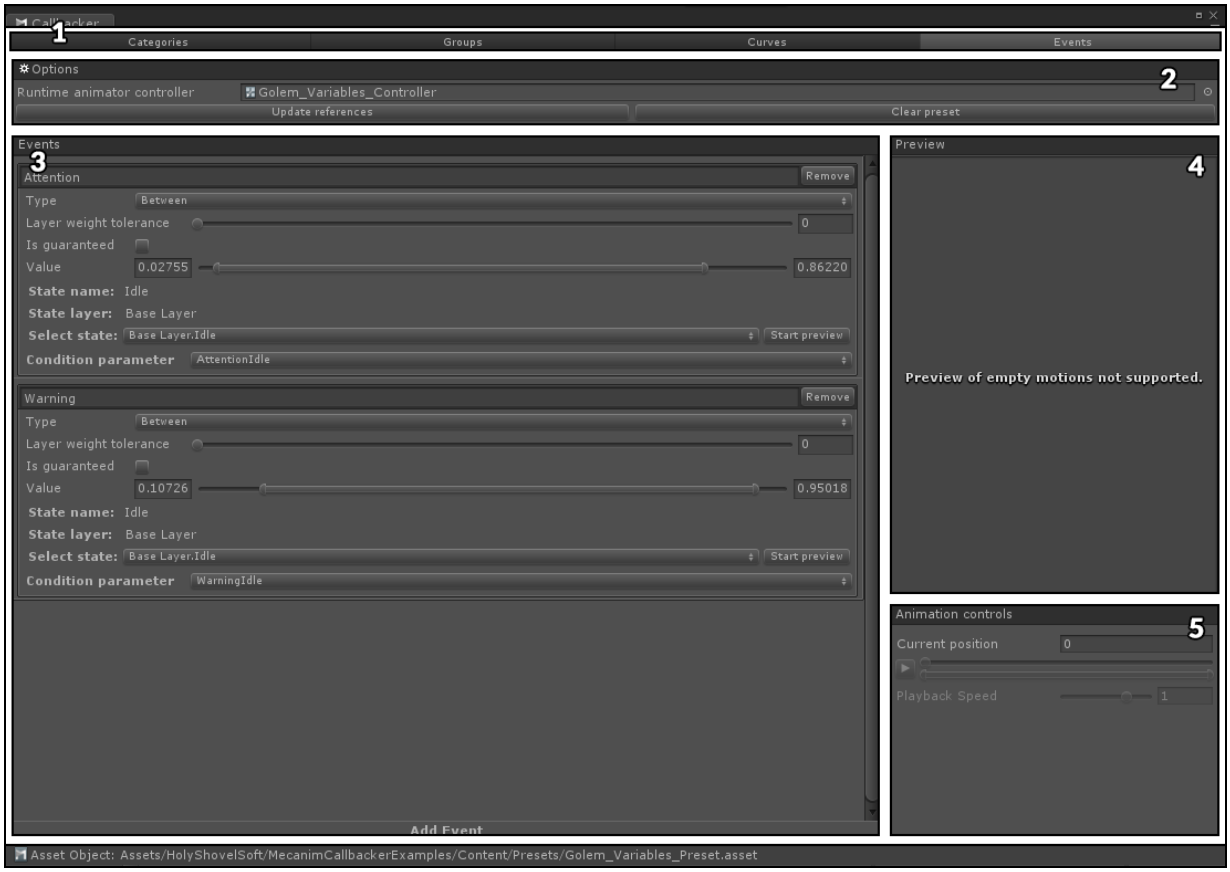
There are three types of presets, two of which are saved in the project as assets: **Preset** and **PresetWrapper**, as well as a common C# object, serialized "inside" controller: **LocalPreset**.

- **LocalPreset** - is saved with the scene, can't be used on more than one game object, doesn't need to be saved on disk as asset. It's useful when creating unique characters (bosses, traders, etc) or objects.
- **Preset** - gives you functions analagous to **LocalPreset**, with the difference being that it needs to be saved on disk and can be used in several controllers. If you don't know what type to use, use this one.
- **PresetWrapper** - "wrapping" for **Presets** or another **PresetWrapper**, which lets you alter already existing entities or add new ones. To understand when to use one, lets look at an example: we have AnimatorController for a character Human with animation Jump and event, which triggers when character touches ground. But we also have OverrideAnimatorController for character Orc, and in that animation Jump is redefined, and our event doesn't match the timing of the new animation. In this case we create **PresetWrapper** and change the event. Everything else remains "as in parent".

Absolutely all presets contain data about entities, but not entities themselves. But presets also can't be changed or created during runtime. It's done this way with the goal of easier creation of entities from code, because classes that are correctly saved by **Unity** Serialization are not always comfortable to use, and vise-versa, that which is easy to use in runtime, won't be always correctly saved into the asset/object. Because these are not entities, controller, during initialization, converts them into entities and preset is not used afterwards in any way. It's one more reason why there's no poing to change or create them in runtime. We can talk more about every entity in separate sections.

Editor

To edit presets, you can use intuitive and fully functional editor window, that can be called by double clicking on the asset, button in the controller inspector, or asset inspector. The window is roughly divided into five parts.



1. Selection of the type of edited data.
2. Preset options. Every preset type has its own. Only two buttons are common in this section: **Update references** and **Clear preset**. First button launched forced scan of all objects, that this preset is dependent on. This command is useful if some assets have changed, but editor doesn't see the change. Second button completely clears the preset of all entities.
3. Primary work section. In this section you can see already added and set up data in this preset, and to add/edit/delete it.
4. Animation preview section. Some types of data have ability to choose state from **AnimatorController**. With the help of this window, you can see in real time that state with different speed or direction of playthrough.
5. Section for controlling states in animation preview window. In this section we have Play button, as well as settings for speed/animation segment/current position. Also, if the previewed state is BlendTree, then all parameters used in that will be listed here.

Options of the different preset types.

- **Preset**

  - **Runtime Animator Controller** - controller, that will be used as source of states and parameters for setting up of entities (can be used as **AnimatorController** as well as **Override Controllers**)
- **PresetWrapper**

  - **Use parent animator controller** - defines whether **Animator Controller** from parent will be used. If this option is turned off, then an option to redefine link to **Animator Controller** will be available.
  - **Runtime Animator Controller** - same as above.
  - **Parent preset** - link to parent **Preset** or **PresetWrapper**.
- **LocalPreset**

  - **Source of local preset** - component, containing this preset.
  - **Runtime Animator Controller** - **Animator Controller** that is used on this **GameObject**.

> Additionally, at the bottom of the window you can see which particular asset/object is edited at this moment, and by clicking this the ping of that asset/object will be performed.

1.4. Categories and Groups

To first page

Common Info

Lets begin with that **Categories** and **Groups** are united in one section due to their similarities, but they are different things and should be used for different goals. Both of them are meant for tracking events, happening with **AnimatorController** states, uniting in them one, several or even all states.

Primary (and only) difference between **Categories** and **Groups** is how events contained in them are processed. **Categories** processes all its events as if it was one entity. For easier understanding you can consider them analogous to **SubStateMachine** from **AnimatorController**. Meanwhile, **Groups** processes events of states separately for every state. Both **Categories** and **Groups** offer following events:

- **OnStart** - this event triggers when **AnimatorController** enters state, contained in the entity. In the case with **Category** it will be called only if on start of state no other state from the list is active.
- **OnTick** - this event will be called every Update or LateUpdate (depending on the settings of **Animator** component) if any state from **Category** is active (in case with **Group** it will be called for every active state).
- **OnEnd** - will be called upon exiting the last active at that moment state in **Category** and for every exit ouf of state in **Group**.
- **OnLoop** (only for **Groups**) - will be called every cycle of active state in the list.

As well as regular events, these entities also offer **Transitions** events. These events can be tracked on all transitions which leave or arrive to any state from the list. Or you can control direction, as well as states themselves, relation to transition.

- **OnStart** - called on start of the transition.
- **OnTick** - called every Update or LateUpdate analogous to usual **OnTick**.
- **OnEnd** - will be called on transition end..

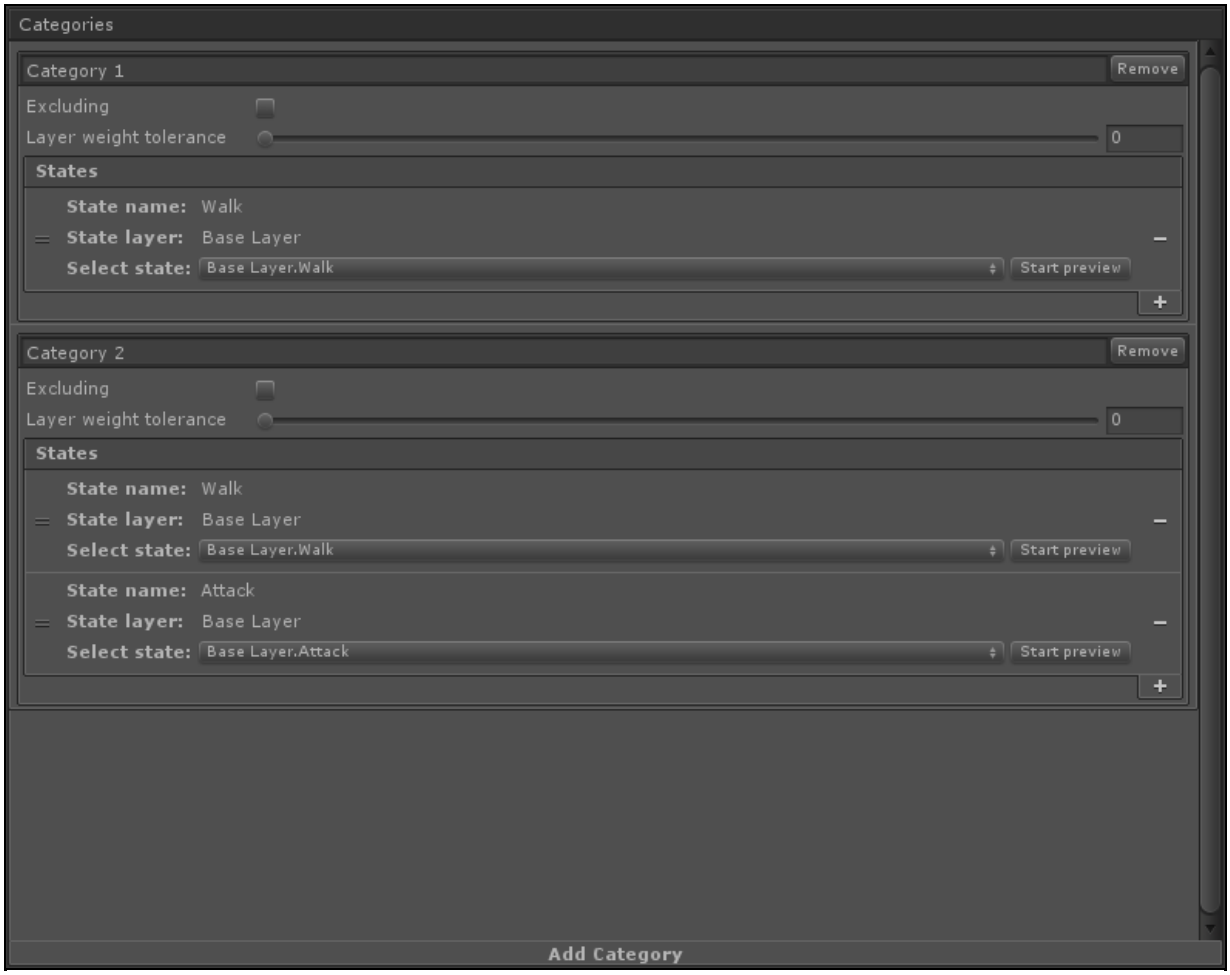
> Important detail is the fact that in case with **Category Transitions** events can't be processed if both states (original and goal) are within the given **Category**, as for the entity itself no transition occurs. When using **Groups**, this limitation does not exist.

The the least important option is Excluding - entities with this option track events not for all states listed within, but all states not listed.

> For exameple, if regular **Group** lists following states: "walk", "jump" and "idle", and animator also has "attack and die", then in regular **Group** events will relate to "walk", "jump" and "idle", but in Excluding **Group**, events will relate to "attack" and "die".

Editor

Editors are absolutely analagous for both types, so lets have a look at an example of **Categories** editor.



This editor lets you add, delete and edit **Categories** or **Groups**. For **PresetWrapper** button for adding new entities also lets you to create clean or full copy of entities from parent preset. Name of every entity is editable; the button for deleting entity is located to the right of the name. These control elements are identical for all entities, hence onwards we will look only at unique control elements.

Primary elements for entity control:

- **Excluding** - whether this entity is excluding or not. beginning with which threshhold state is considered active, in the case the weight of the layer of this entity is other than 1
- **Layer weight tolerance** - beginning with which threshhold state is considered active, in the case the weight of the layer of this entity is other than 1.
- **States** - list of states included in this entity, you also have ability to preview states.

1.5. Curves

To first page

Common Info

Curves serve only one goal: to superpose given value to a given state for the duration of the entire state. Just as standard curves in **Unity**, which could be used in animation clips. Unlike them, **Curves** in **MecanimCallbacker** are more flexible and complex, allowing you to solve more complex tasks.

Every **Curve** by itself contains several curves (called **SubCurve**), each of which is installed for a given state. This lets you attach a curve to several states at once. For every **SubCurve** you can also install several **Modifiers**. These are elements that change the value of the curve depending on their settings. All **Modifiers** are applied consequetively, changing the value and passing it onto the next. They work the following way: every **Modifier** has two states: first one contains data, that will replace previously calculated value, second contans the amount of influence of the first curve on the calculated value.

Modifiers have several types:

1. **Parameter** - uses float parameter from **AnimatorController** to calculate the amount of influence. This parameter points at which point the curve of influence amount is going to be requested.
2. **Transition from this** - works in the case if **AnimatorController** is in the state of transition from the state, to which this **SubCurve** is attached to. If this is not the case, it's added automatically with linear interpolation.
3. **Transition to this**- works in the case if **AnimatorController** is in the state of transition to the state, to which this **SubCurve** is attached to. If this is not the case, it's added automatically with linear interpolation.

They also have several types of work function:

1. **Override** - Modificator replaces the previously calculated value.
2. **Multiply** - Modificator is multiplied with previously calculated value.
3. **Additive** - Modificator is added to previously calculated value.
4. **Min** - minimal value between Modificator and previously calculated value is chosen.
5. **Max** - maximal value between Modificator and previously calculated value is chosen.

Because several layers can be active at once, and hence the curve can be calculated based on several active **SubCurves**, a setting exists as to how to calculate the final value between layers. For every layer, just as with Modificators, you can set up the curve of influence (depends on layer weight) and calculation mode. Just as with Modificators, calculations occur consequitively.

Calculation of the **Curve** happens following way:

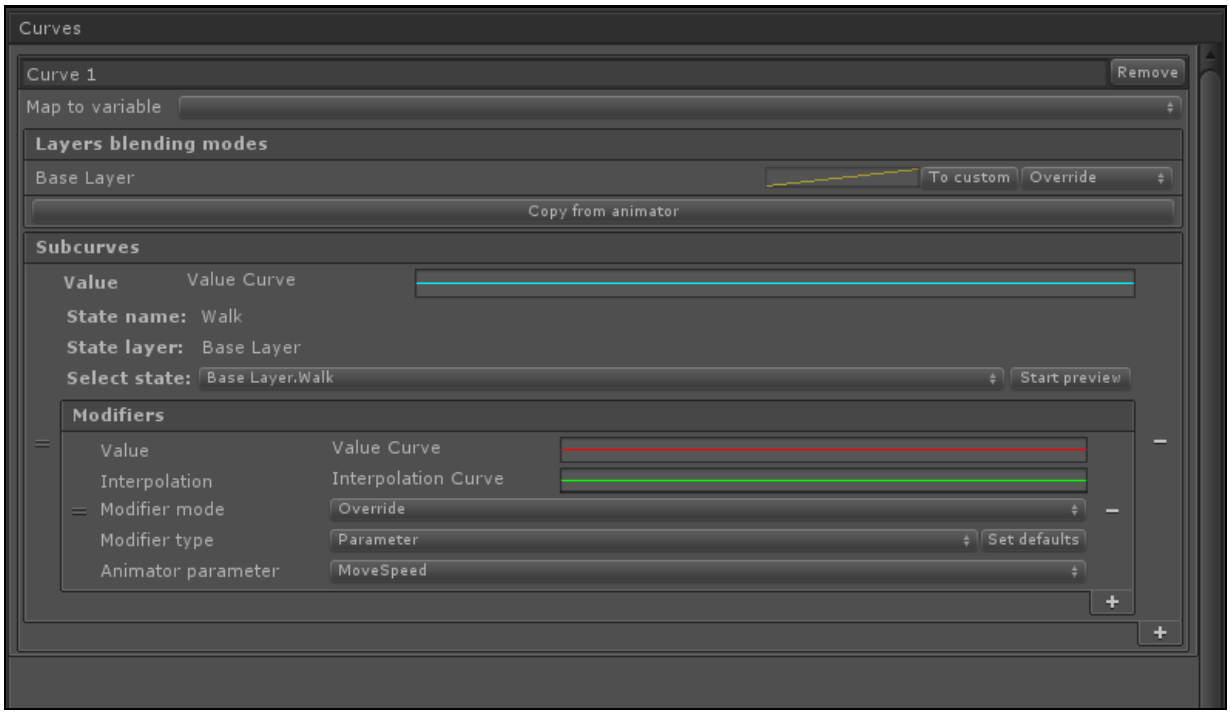
1. All active at that moment **SubCurves** are collected.
2. For each of them, values are calculated consequitevely.
    - For every Modifier, values are consequently calculated without the curve of influence. For example **Multiply** is calculated as: candidateValue = modifierValue * originalValue.
    - Then, based on the value gotten from parameter or **Transition** normalized time (in the case of **Transition to/from this**) we get the amount of influence from corresponding curve.
    - Original value is replaced to a blend of itself and Modificator value as: newValue = lerp(candidateValue, originalValue, interpolationValue).
3. In the case several of them are in one layer - they are added (possible only in the case of transition, and hence in one layer you can only have maximum of two **SubCurve**). In the case **SubCurve** are located in separate layers, their values are calculated analagous to mechanics of **Modifier**.

Also, every curve can be automatically attached to parameter from **AnimatorController**. System will automatically update this parameter every frame according to the current value of the curve.

To get the value of the curve you can use the following two attributes.

- **BaseValue** - returns value without applying **Modifiers**.
- **Value** - returns value after applying all **Modifiers**.

Editor



1. **Layers blending modes** - by-layer setting rules for mixing **SubCurves**. For every layer you can set its own rule, its own curve, as well as copy the rule from **AnimatorController**.
2. **Subcurves** - list of curves based on which value is calculated.
    - **Value** - curve of influence. Value is calculated based on the normalized time state, to which this **Subcurve** is attached.
    - **State** - selection of the observed state, it's possible to preview the state by manual rewinding.
    - **Modifiers** - list of **Modifiers** of this **Subcurve**.
        - **Value** - curve of influence. Value is calculated based on the normalized time state, to which this **Subcurve** is attached.
        - **Interpolation** - curve of influence. Calculated based on **Parameter** or **Transition** normalized time.
        - **Modifier mode** - work function type of this modifier.
        - **Modifier type** - based on what to calculate the value of the curve of influence.
        - **Animation parameter** - parameter, based on which value of the curve of influence is calculated, in the case if **ModifierType** is set to **Parameter**.

1.6. Events

To first page

Common Info

Events, just as **Curves** - are a replacement to the standard **Unity** events, which can be called in animation. Unlike them, these entities are not attached to animation clip, they are attached to the particular state of **AnimationController**.

There are two types of **Events**:

1. **One shot** - standard mechanic. Within limits of normalized time state a point is selected. When state enters that point, this **Event** is triggered.
2. **Between** - when setting up the even you can select not a point of normalized time state, but a period of it. Unlike **One shot**, three events wil be called: entering the period, exiting the period, and every update within the period.

Events can also be "guaranteed" and "not guaranteed". "Not guaranteed" events process only the last triggering of the events. Guaranteed events process all events that should have happened since the last frame. This option can be useful if the state is very short, if lag has occured and eaten up a few cycles of states, and you need **Event** to occur on every of those cycles.

Events offer the following C# events for subscribing to:

1. **OnStart** (only for **Between**) - occurs on entering current normalized time state in the set period.
2. **OnFire** - triggers upon reaching the normalized time state of the set point, or every frame the given time is inside the set period.
3. **OnEnd** (only for **Between**) - occurs on exiting current normalized time state from the set period.

Editor

1. **Type** - type of a given **Event**.
2. **Layer weight tolerance** - weight threshold of state layer, after which event starts to trigger.
3. **Is guaranteed** - whether this event is guaranteed or not.
4. **Value** - point or period at which event should trigger.
5. **State** - state, to which the event is attached to.
6. **Conditional parameter** - boolean paramaters which controls whether given **Event** should trigger.

It's important to understand, that in case of **Between Event** parameters **Layer weight tolerance** and **Conditional parameter** upon changing can cause triggering of **OnStart** and **OnEnd**. For example, if the state is located inside the set period and parameter is change from true to false, then **OnEnd** will be called.

1.7. Variables

To first page

This entity is one without an editor. Essentially, it's wrapping for parameters of **AnimatorController**, but with some bonuses:

1. **IVariableProcessor** - an interface, by implementing which and setting it in **Controller.VariableProcessor** quality you can get control over how variables receive and set values. By default **Controller** simply installs values and receives them with the help of **AnimatorController**. This could be useful, if, for example, it is necessary to synchronize values of parameters across network, log them, send them to a child object, etc.

   > It's important to understand, that upon redefining this quality, the realization of interface in **Mecanim Animator** needs to send and receive date, this will not be done automatically.

2. **Binding** - every variable will have attached get/set delegates, using which the variable will send it's value to user code every frame... or receive values from there and set it as it's new value.

   > **Binding** is used for syncronizing the value of **Curves** with variable, because if the code will set different biding, then connection to the curve will be lost. You can also set more complex connections with not just curves, but other variables using binding through code.

1.8. Procedures

To first page

Just as **Variables** this entity is not represented in the editor, but at the same time is a very useful element of the system. **Mecanim Callbacker** gives many callbacks for working with events... But almost all of them are dependant on Update/FixedUpdate look. What if it's necessary to perform code, that is dependent on what state, **Group**, **Category** or even **Between Event** we are located in? **Procedures** can help us do that!

So, what is **Procedure**? It's a named entity that lets you bind your code and listed above entities (all of them are united by that we can be located in them). This is the usual pipeline for working with **Procedures**:

1. Create/get procedure by calling method **Controller.Procedures.GetProcedure**. This method has versions with the ability to create procedures with parameters (up to five). As long as types of parameters are different, you can use the same name for procedure.

```
//Sample
//Get first procedure
var p1 = controller.Procedures.GetProcedure<float>("proc");
//Get second procedure
var p2 = controller.Procedures.GetProcedure("proc");
//Get third procedure, but its the same that first
var p3 = controller.Procedures.GetProcedure<float>("proc");
```

2. Bind our code to procedure.

```
//Sample
//Get procedure
var p = controller.Procedures.GetProcedure<float>("proc");
var c = controller.Categories["category"];
c.BindProcedure(swapProcedure).OnCall += (sourceController, f) => {
    Debug.Log("Hello there! I got value = " + f);
}
```

3. Call procedure in any part of the code, and only code that's bound to currently active entities will be called.

```
//Sample
//Get procedure
var p = controller.Procedures.GetProcedure<float>("proc");
p.Call(10f);
```

1.9. Attributes

To first page

For easier work with **Animator** and **MecanimCallbacker** data you can use following attributes:

Animator

1. **AnimatorBoolParameterAttribute** - attribute for selecting boolean parameters from target animator.
2. **AnimatorFloatParameterAttribute** - attribute for selecting float parameters from target animator.
3. **AnimatorIntParameterAttribute** - attribute for selecting int parameters from target animator.
4. **AnimatorTriggerParameterAttribute** - attribute for selecting trigger parameters from target animator.
5. **AnimatorStateAttribute** - attribute for selecting states from target animator.

Callbacker

1. **CallbackerCategoryAttribute** - attribute for selecting category from target preset.
2. **CallbackerGroupAttribute** - attribute for selecting group from target preset.
3. **CallbackerCurveAttribute** - attribute for selecting curve from target preset.
4. **CallbackerEventAttribute** - attribute for selecting event from target preset.

О параметрах аттрибутов

1. **sourceSearchMode** is where search occurs for the source (**AnimatorController** или **Preset**). It's necessary to understand, that when using this parameter, the search will happen not for assets themselves, but for components of **Animator** or **CallbackerController**. Afterwards the found component will be used for getting a link to the asset. Possible variants: ThisObject (used by default) - search for the components in the current object. Childs - search for the first matching component in the current object and its children. Parents - search for the first matching component in the current object and its parents.
2. **memberPath** - name of the member (field, property, method) that will be used for getting **AnimatorController** or **Preset**. When using this parameter you can have more flexibility when specifying source of data. This member can't be static and needs to return required type (in the case of methods, it can't have parameters). Also, this member can return not only **AnimatorController** or **Preset**, but **Animator** or **CallbackerController** as well.

```
//Sample
private BasePreset targetPreset;
//We use this property as source of data
public BasePreset TargetPreset { get { return targetPreset; } }
[CallbackerCategory("TargetPreset")]
public string canWalk;
```

These parameters are mutually exclusive. **sourceSearchMode** has priority.

2. FAQ

To first page

In this section we will answer the most common or important questions, regarding to working with the asset.

2.1. How to install or update asset?

To first page

Links for download

1. **Asset Store**
2. **Examples**

Installation

When installing for the first time, asset is installed using standard methods. Examples are imported as ***.unitypackage*** **using command** "Assets/Import Package/Custom Package..."**

Update

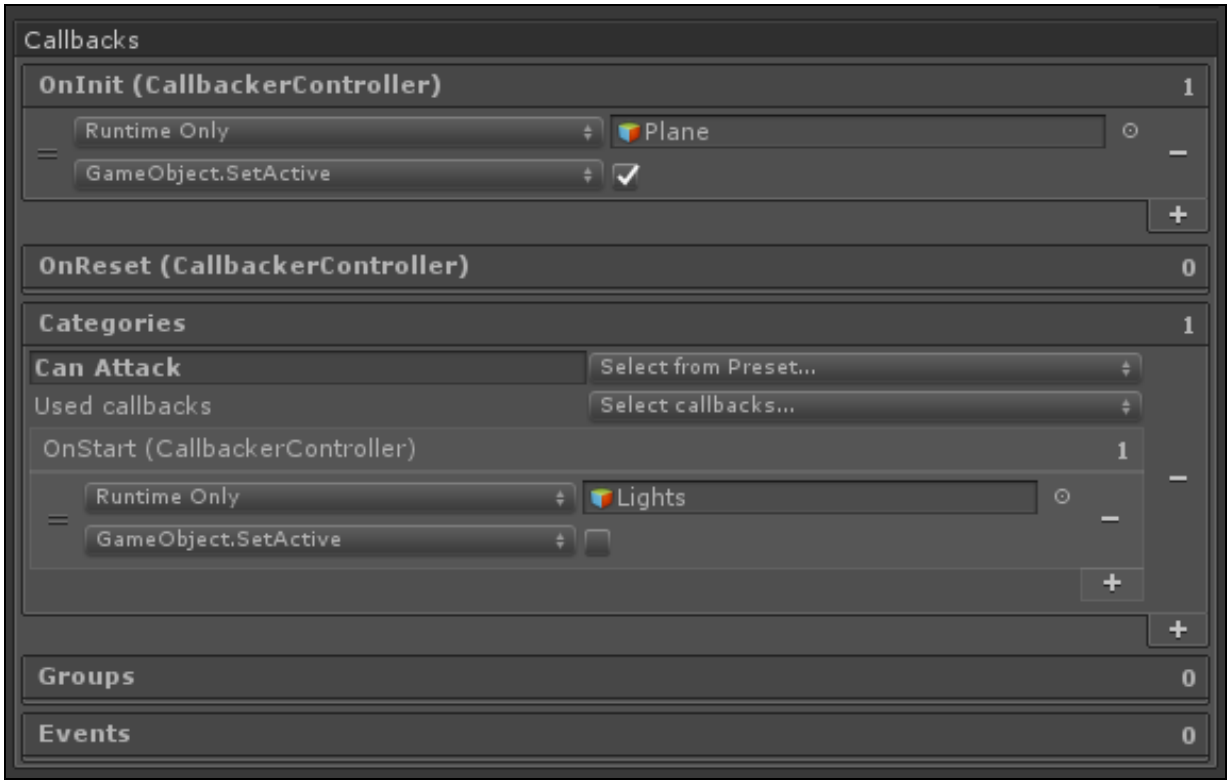
When updating, we recommend the following steps:

1. Always create a backup of your project before updating.
2. Close the project.
3. Delete folders **MecanimCallbacker** and **MecanimCallbackerExamples**.
4. Install updated asset into a clean project using same version of **Unity** as for the main project.
5. Copy folders **MecanimCallbacker** and **MecanimCallbackerExamples** into your main project into the same directory it was stored in before.
6. Enjoy!

2.2. How use callbacks without codding?

To first page

When working with **MecanimCallbacker** it's possible to use callbacks without writing any code. For this, you can use **Callbacks** section.



- **OnInit/OnReset** - are analogues of the calls of methods **OnAttached/OnDettached** of **ICallbackerEventObserver** interface.
- **Categories/Groups/Events** - containers for callbacks of existing entities. To register callback, you need to select an entity (from a drop-down list or by manual entry) and to select a callback type (from a drop-down list)

3. Can I create entities in runtime?

To first page

Yes. Entities can be created in runtime, and procedures can be only created in runtime. This lets you use **CallbackerController** without presets.

Categories

```
//create simple category
var category1 = controller.Categories.FormStates["BaseLayer.Walk", "BaseLayer.Idle"];
//after we can get this category by name
var category1FromName = controller.Categories[category1.Name];
//also we can create category as excluding
var category2 = controller.Categories.Excluding["BaseLayer.Walk", "BaseLayer.Idle"];
```

Groups

```
//create simple group
var group1 = controller.Groups.FromStates["BaseLayer.Walk", "BaseLayer.Idle"];
//after we can get this group by name
var group1FromName = controller.Groups[group1.Name];
//also we can create group as excluding
var group2 = controller.Groups.Excluding["BaseLayer.Walk", "BaseLayer.Idle"];
```

Curves

```
//curve creating more complex than category or group
//first of all we create preset
var curvePreset = new CurvePreset();
//set mapping to variable
curvePreset.mapTo = controller.Variables.Floats["float1"];
//and create som subcurves for it
curvePreset.subcurves = new SubcurvePreset[1];
curvePreset.subcurves[0] = new SubcurvePreset();
//set state name for subcurve
curvePreset.subcurves[0].statePath = "BaseLayer.Idle";
//and AnimationCurve
curvePreset.subcurves[0].valueCurve = AnimationCurve.Linear(0, 1, 1, 1);
//also we can assign modifiers
curvePreset.subcurves[0].modifiers = new CurveModifierPreset[1];
curvePreset.subcurves[0].modifiers[0] = new CurveModifierPreset();
curvePreset.subcurves[0].modifiers[0].parameter = controller.Variables.Floats["float2"];
curvePreset.subcurves[0].modifiers[0].type = CurveModifierType.Parameter;
curvePreset.subcurves[0].modifiers[0].interpolationCurve = AnimationCurve.Linear(0, 1, 1, 1);
curvePreset.subcurves[0].modifiers[0].valueCurve = AnimationCurve.Linear(0, 1, 1, 1);
//after all we can create curve
var curve = controller.Curves.FromPreset(curvePreset);
```

Events

```
//events also creating from preset
var eventPreset = new EventPreset();
//set state name
eventPreset.statePath = "BaseLayer.Idle";
//and type
eventPreset.type = CallbackerEventType.Between;
//set guaranteed flag
eventPreset.isGuaranteed = false;
//assign range (for CallbackerEventType.OneShot use eventPreset.timePoint)
eventPreset.timeRange = new Vector2(0.25f, 0.89f);
//and set condition variable
eventPreset.conditionVariable = controller.Variables.Bools["bool1"];
//after all we can create event
var resultEvent = controller.Events.CreateFromPreset(eventPreset);
```

Procedures

```
//procedures is very simple in create
//this code create or get procedure with some name
var procedure = controller.Procedures.GetProcedure<float, int>("MyProc");
//now we must bind some action to it
controller.Categories["MyCategory"].BindProcedure(procedure).OnCall +=
    (sourceController, param1, param2) =>
    {
        Debug.Log("I'm called with params " + param1 + " " + param2);
    };

//now we can call procedure in any place, and if MyCategory is active - our code will be called
procedure.Call(10f, 15);
```

4. Do you have integration with {X} asset?

To first page

No. At this moment we don't have any integrations with third-party assets. After beta testing we are planning integration with **FlowCanvas** and supporting attributes in **Odin Inspector**. If enough requsts are made, other integrations are open to discussion.

5. Do I need to change the approach to working with Mecanim?

To first page

Mostly "no". **MecanimCallbacker** presupposes work with any **Animator** settings.

6. How I can send bug info, request or question?

To first page

You can use our **contact form**. Also you can write us an email at **support@holyshovelsoft.com**. We are open to your ideas, suggestions, and are ready to answer your questions!

3. Integrations

To first page

Attention!

Integrated assets can become uncompilable after updating main assets due to scripting API change. Always create backups when updating both integrated assets and MecanimCallbacker. In case of third-party asset API change causes something to break, we will try to fix the problem as soon as possible, but we can't guarantee an instant fix

1. FlowCanvas

To first page

Installation

Package with integration scripts can be found here

Content

1. Scripts for integration (HolyShovelSoft/MecanimCallbackerIntegrations/FlowCanvas/Nodes). They are located outside custom asmdef. This is done with purpose of maximal compatibility with majority of Unity project layouts. If necessary, asmdef can be put directly into the folder HolyShovelSoft/MecanimCallbackerIntegrations/FlowCanvas and set up so that in References of a given Assembly were set MecanimCallbacker and FlowCanvas (depending on your project settings it can also be other Assemblies).
2. Remake of "6. Procedures" scene from the asset examples using only FlowCanvas nodes(HolyShovelSoft/MecanimCallbackerIntegrations/FlowCanvas/Examples). Without installed primary asset (MecanimCallbacker) this scene will not function!