Manaspark

ONE : Logic State Machines for Unity3D

Contents

1. Introduction

ONE relies on Unity's Animator system and makes it super easy to add logic to Animator states and to construct state-machines.

You can build two types of state-machines:

Controller State Machines

- for controlling high-level game logic

Entity State Machines

- for controlling the behaviour of single entities

The logical operations are defined in *StateMachineBehaviours* that get attached to states like components to MonoBehaviours. This tool is built on these components and every action that we define with it, is called a *State Action*.

Basic Info:

Controller type:

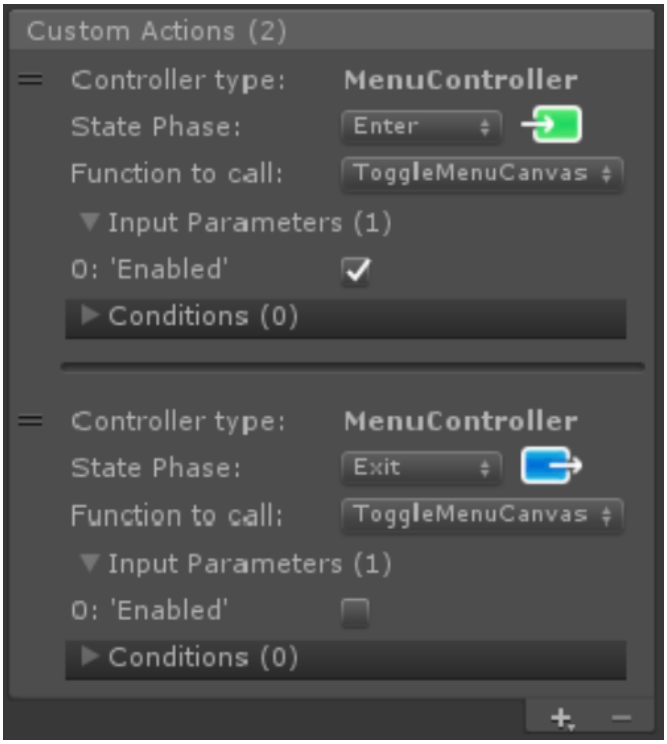
- The referenced class where you choose methods and events from.

State Phase:

- The timing for the execution

Function to call:

- Well, you guessed it, the function from the referenced class, that gets called.



2. Installation

Download the package from the Asset Store and import it in your project:

https://assetstore.unity.com/packages/slug/122543

3. Integration

3.1 Creating a State Machine

The first thing you have to do to create a State Machine is to create an *Animator Controller* in your project window.

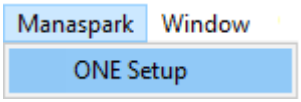
Project > Right click > Create > Animator Controller

The Animator Controller runs on a State Machine and usually it is used to combine animation clips on states in a flow-chart. But we can also attach logical components to these states and either choose to run this State Machine only with logical components or even in combination with animations.

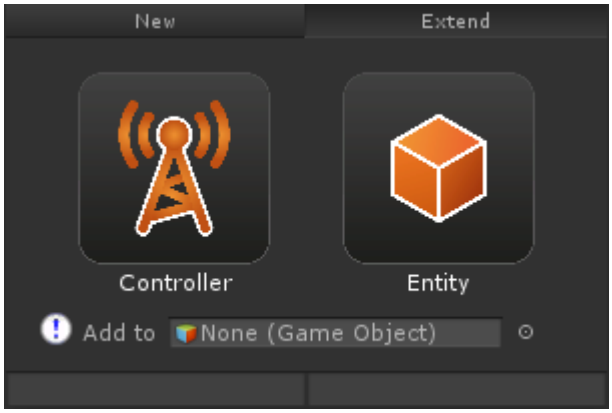
3.2 Plug the State Machine in a GameObject

Animator Controllers run on Animator components, so we'll need one on the hosting GameObject. Besides the Animator component we also need additional components for this State Machine system. So, to support you in this setup process, there is a little Setup Guide:

Unity head menu band > Manaspark > ONE Setup



With the help of this guide, you can either create a new GameObject or extend an already existing one, to run as a Controller or Entity State Machine.



When you have selected the tab "New", you can see in the bottom line how many plugged State Machines already exist in the current scene.
There can always be only one Controller State Machine in the scene, so this guide will warn you, if there are more.

Same with the "Extend" tab: there can always be only one Animator component on a GameObject and only one extension component for each State Machine type.

3.3 Exposing custom code

Using events and methods from custom code in this State Machine system is the central feature of this asset.
To use this feature correctly, you have to mind some rules of usage.

3.3.1 MonoBehaviours

Only classes that derive from *MonoBehaviour* (incl. child classes) can be exposed.

3.3.2 Scene objects

To inspect a class that is derived from MonoBehaviour, it must at least be attached to one GameObject that exists in the currently opened scene. Otherwise it cannot be found.

If you want to work on a State Machine that runs e.g. on a Prefab, you have to drag the Prefab in the current scene. After modifying the State Machine, you can delete the GameObject if you want – already set methods and events on states will stay.

3.3.3 Marking with Interfaces

All MonoBehaviours in the current scene get filtered in order to not overwhelm the user. You have to add a special interface to the class in order to mark it for this tool.

To do so, the class must include the namespace:
„ManasparkAssets.LogicStateMachine“

```
using ManasparkAssets.LogicStateMachine;
```

[Controller State Machine] Implement the interface „`IActionController`“

[Entity State Machine] Implement the interface „`IEntityAgent`“

example:

```
…
using ManasparkAssets.LogicStateMachine;

public class MyControllerClass : MonoBehaviour, IActionController { … }
```

3.3.4 Allowed Methods

When you assing a method to a State Action, only certain methods are allowed.

a) They must not have input parameters with keywords like *ref, out, in, …*

b) They must not have any overload. (Otherwise an error is thrown)

c) Further, only the following types of input parameters can be directly set in the Inspector:

The following parameter types are currently supported:

- `int`
- `bool`
- `float`
- `string`
- `Vector2`
- `Vector3`
- `Color`
- `Gradient`
- `Material`
- `Texture`
- `Sprite`
- `AudioClip`
- `GameObject`
- `ScriptableObject`

d) Access Modifiers play an important role.

 **[Controller State Machine]:**

- Simple actions & else-statements: ***public static*** *
- Conditions: ***public static*** with a ***bool*** return value

 **[Entity State Machine]:**

- Simple actions & else-statements: only ***public*** *
- Conditions: only ***public*** with a ***bool*** return value

* The return value has no influence, as it does not get received.

3.3.5 Allowed Events

When the timing of a State Action is set to “On Event”  , you have the choice between Basic and Advanced events, which differ in their integration.

- Always make sure to use the keyword „`event`“! -

a) Basic Events

The delegate type for these events is already defined: `StateActionEvent`

 **[Controller State Machine]:**

```
public static event StateActionEvent EventName;
```

 **[Entity State Machine]:**

```
public event StateActionEvent EventName;
```

b) Advanced Events

First: define a new delegate with any input arguments you need.

e.g. `public delegate void` AdvancedEventDelegate(`Collider` _col);

Then create the event:

 **[Controller State Machine]:**

```
public static event AdvancedEventDelegate EventName;
```

 **[Entity State Machine]:**

```
public event AdvancedEventDelegate EventName;
```

Only methods with matching input parameters can be subscribed to Advanced Events!

4. State Machine types

As shown before, there are two different types of State Machines related to this tool. In this section we'll learn some details about their usage.

4.1 Controller State Machine 

You can create State Actions for Controller State Machines by attaching a *ControllerStateActions* component to a state.

Controller State Machines should only exist once in your scene, because they rely on static references. Their intention is to address classes in your scene, that have a managing role, like e.g. a Game Manager, UI Controller, Audio Manager, etc... Preferably components, that are unique in the current scene, too.

Controller State Machines are used to control the logic and flow of your game/app on the highest level.

4.1 Entity State Machines 

You can create State Actions for Entity State Machines by attaching a *EntityStateActions* component to a state.

Entity State Machines are the solution to control non-unique elements on a high-level and can be compared to AI Behaviour Sheets.

The references in this type of State Machine are not static. They are bound to the Components which are attached to the same GameObject as the Animator, where the Entity State Machine is running on.

You can easily change the behaviour of a GameObject by swapping the Animator Controller that represents the Entity State Machine.

[Important note]
When designing the Actions on an Entity State Machine, you can choose methods and events from any accessible MonoBehaviour in the current scene.
BUT, when the Animator runs the Entity State Machine, you must make sure, that every referenced MonoBehaviour is attached to the same GameObject as the Animator, as soon as one of its methods or events gets executed! Otherwise you will run into errors, because the referenced Component could not be found.

Note:
It is also possible to attach both components, for Controller and Entity State Machines, to a state and to e.g. address a static manager from an Entity State Machine.

5. Extension Components

Both Controller and Entity State Machines are using Extension Components that get attached to the GameObject besides the Animator. They hold information about their State Machines for reasons of debugging and diagnostics and they have built-in methods to support the user on controlling the State Machine.

[Controller State Machine]: LogicStateMachineController

[Entity State Machine]: EntityAnimatorExtension

5.1 Built-in methods

These methods can be called from your own scripts or be executed as a State Action to let the State Machine control itself.

The following methods can be found in both types of Extension Component.

- `void SetAnimatorTrigger(string ParameterName, float Delay)`

  (`string`) `ParameterName:` Name of the Animator trigger parameter

  (`float`) `Delay:` A delay > 0 will set the trigger in a delayed Coroutine

  Info Sets the named *Trigger* transition parameter in the Animator. This call can be delayed.

- `void SetAnimatorBool(string ParameterName, bool BoolValue float Delay)`

  (`string`) `ParameterName:` Name of the Animator bool parameter

  (`bool`) `BoolValue:` The new parameter value

  (`float`) `Delay:` A delay > 0 will set the trigger in a delayed Coroutine

  Info Sets the named Bool transition parameter in the Animator. This call can be delayed.

- `void SetAnimatorInteger(string ParameterName, int IntValue float Delay)`

  (`string`) `ParameterName:` Name of the Animator int parameter

  (`bool`) `IntValue:` The new parameter value

  (`float`) `Delay:` A delay > 0 will set the trigger in a delayed Coroutine

  Info Sets the named Integer transition parameter in the Animator. This call can be delayed.

- `void SetAnimatorFloat(string ParameterName, float FloatValue, float Delay)`

  (`string`) `ParameterName:` Name of the Animator float parameter

  (`bool`) `FloatValue:` The new parameter value

  (`float`) `Delay:` A delay > 0 will set the trigger in a delayed Coroutine

  Info Sets the named Float transition parameter in the Animator. This call can be delayed.

- `void AbortDelayedCalls()`

  Info Stops all Coroutines, that have been started due to delayed manipulations of Animator parameters.

5.2 Diagnostics field

Both Extension Components have a Diagnostics field, which can give you valuable information about your State Machine, so it's important to check it frequently.

For example, it can tell you, if there are any transitions without conditions, what would make them impassable. Or if there was a delay activated in the transition settings, it would cause unwanted behaviour.

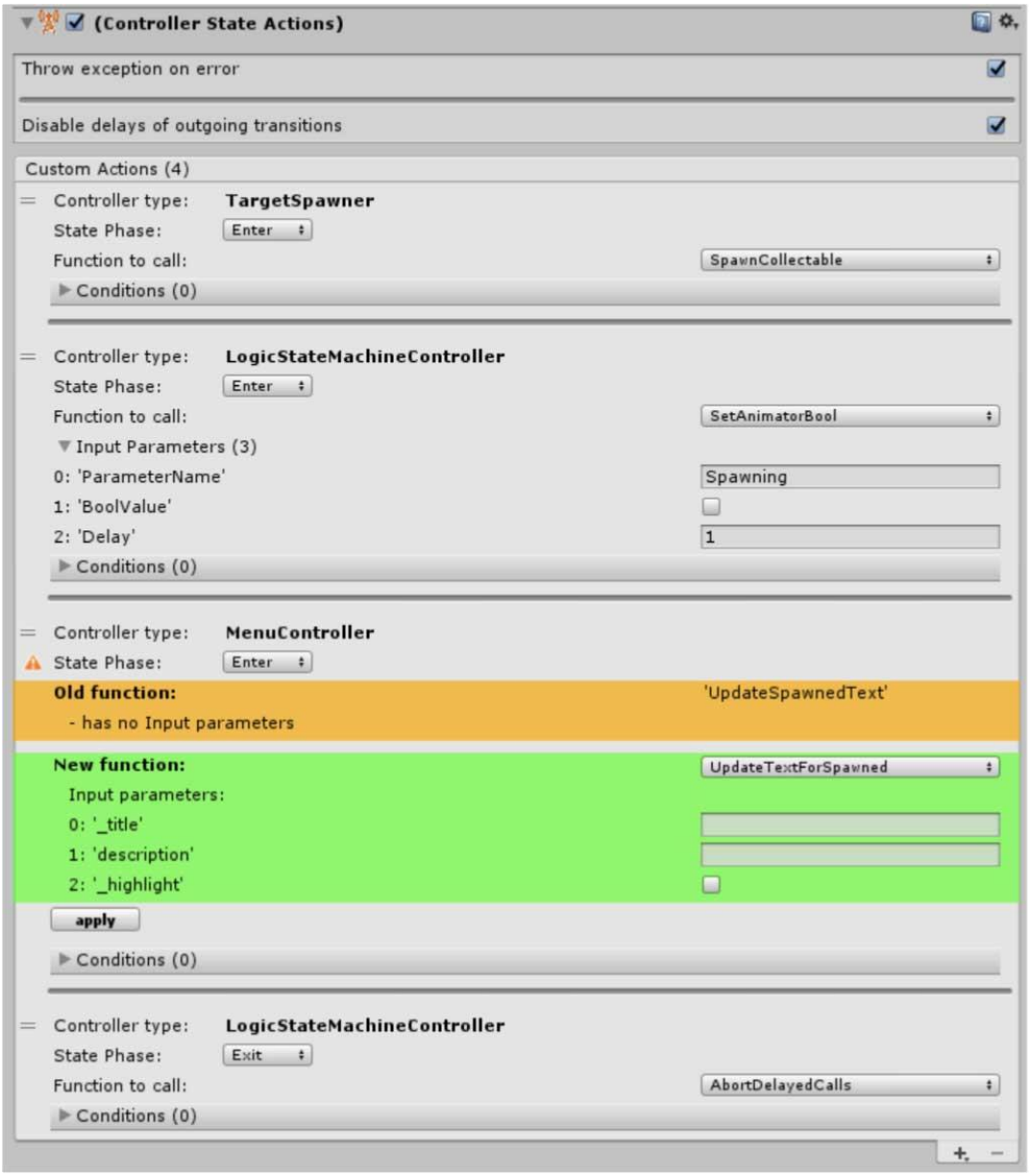
These and other errors get displayed in the Diagnostics field.
It is a good practice to lock this view in the Inspector, while you are working on your State Machine

6. Change Management

It is often the case that you change and refactor your code during development. This can break State Actions, for example when you rename a method or change its input parameters. When something like this happens, don't worry!

In the example below, the method "UpdateSpawnedText()" has been renamed and got some new input parameters. In a nutshell, this method does not exist anymore.
Now, in the Change Management you can still see the old selection for your State Action (marked in orange). The green section is where you can choose a new method to replace the broken reference. In the example below the renamed method "UpdateTextFromSpawned(…)" with its new input parameters has been selected.

Now just click on "apply" and everything is fine!

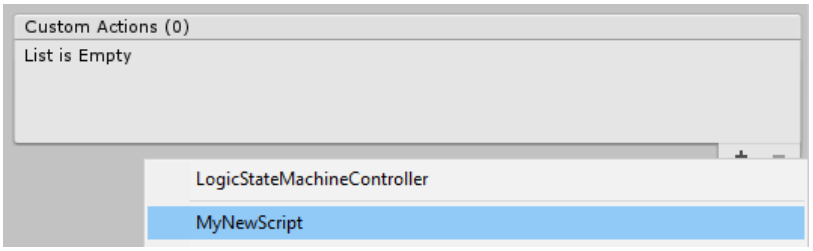


7. Usage

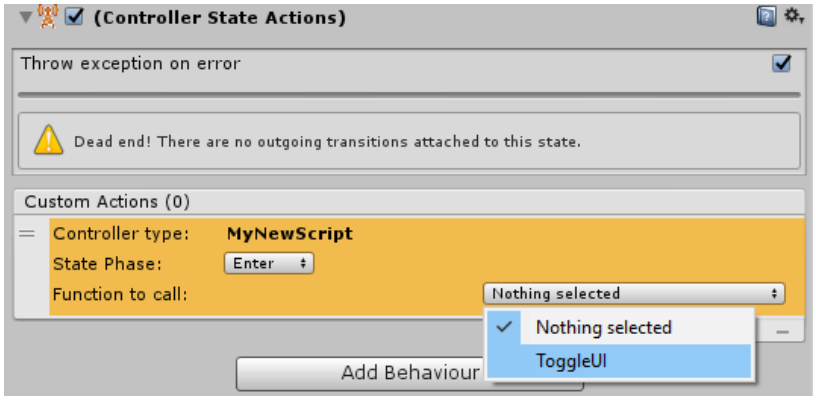
The basics are simple. Select a state on your Animator Controller and either add *ControllerStateActions* or *EntityStateActions* as Behaviour. The difference is explained in chapter 4.

If you have already exposed your code as described in chapter 3, you are good to go.

- Add a new class reference:



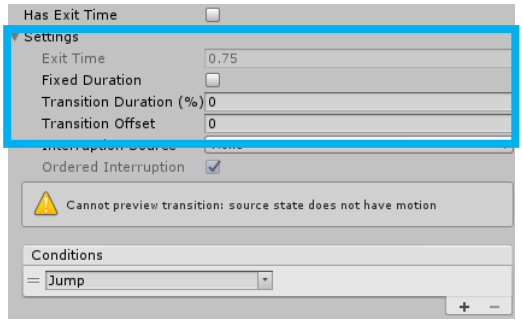
- Add a new State Action:



Then you can set the timings, parameters, conditions, etc…

Note on transitions between states:

In order to work properly and avoid misbehaviour, always make sure, that the settings for the transitions between two states are like the ones shown in the example on the right. Usually these are set automatically, but only on outgoing transitions from states, that have a Controller or Entity Behaviour attached.



7.1 Conditions & Else-Statements

In addition to the main method of a State Action, you can add a condition and also an else-statement.

Both, conditions and else-statements, have to be exposed as described in chapter 3.

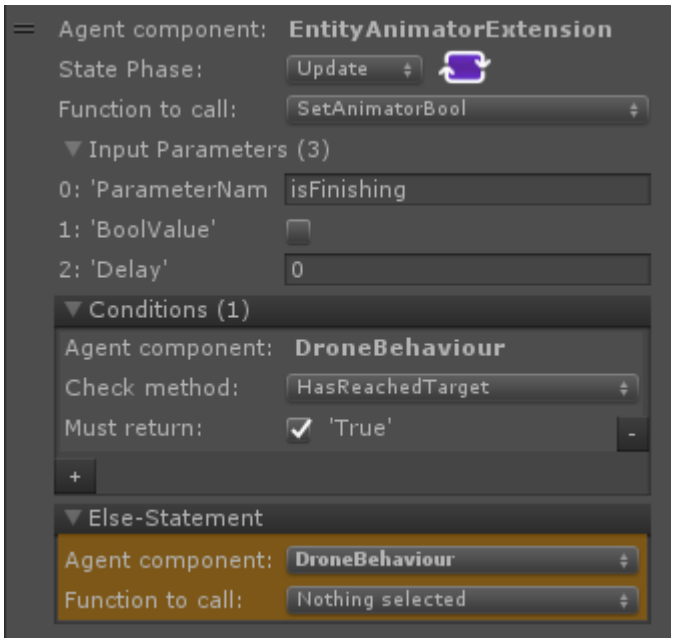
A condition, in this case, is a method with a bool return value. When it is time to execute the whole State Action, this method is the first that gets executed.

The outcome (return value) of this message is then compared with the value that is displayed next to "*Must return"*. If both have the same value, the main method gets executed, otherwise the else-statement (if there is one)

Example:

Every Update, the condition "HasReachedTarget" gets evaluated and compared to the bool value "True", as it is set next to "Must return".

If "HasReachedTarget" returns true and "Must return" is also set to true (which it is), the main action "SetAnimatorBool" gets executed.
If both values do not match, nothing happens, because there is no Else-Statement selected ("Nothing selected")



With the combination of the timing "Update" and a condition, you can re-create the pattern of an observer-loop.

7.2 Examples and best practice

i). Use multiple layers:

You have the option to use multiple layers in the Animator window. Every layer is run in parallel by the Animator and this feature is supported by this tool.

If you need to control several high-level processes at the same time, try to separate their execution on different layers.

ii). Use Sub-State Machines:

Instead of a new state, you can also create Sub-State Machines in the Animator window. They work like a folder for states as you can double-click on them and then create new states or even further Sub-State Machines inside of them.

This is a great way to keep an organized layout.

Further you can also attach *Controller-* or *EntityStateActions* to a Sub-StateMachine. For example, this enables you to run a StateAction in Update mode as long as your State Machine is in one of the states, which are inside of this Sub-State Machine.

iii). Event-driven State Machine:

With the new feature of events as timings, you can subscribe any of the built-in methods from the Extension Components to an event. This way the State Machine can listen to events and change its state automatically instead of setting a transition parameter somewhere in your code.

iv). Static Event + Entity State Action:

You can create static events on your Manager classes and let Entities subscribe on them with one of their methods. Just add the “IEntityAgent” interface to the class with the static event.

This way, you can e.g. let an NPC know when the “Play” button has been pressed.

v). The power of Advanced Events and Unity Callbacks:

Given the following code in an exposed MonoBehaviour for an Entity 

```
public delegate void AdvancedEventDelegate(Collider _col);

public event AdvancedEventDelegate OnTriggerEnterEvent;

private void OnTriggerEnter(Collider _other)
{
    if(OnTriggerEnterEvent != null)
        OnTriggerEnterEvent(_other);
}
```

Select a matching main method to be subscribed to this Advanced Event and it can receive Collider information! As long as the state with this State Action is active, the invocation of the event will also call the selected main-method. Leaving the state will automatically unsubscribe the main method. The same applies for Basic Events.

8. Support

The State Machine is based on Unity‘s Animation Controller. More information on this topic: *https://docs.unity3d.com/Manual/class-AnimatorController.html*.

For more information on Unity’s *StateMachineBehaviour* classes, take a look in the Unity API: *https://docs.unity3d.com/ScriptReference/StateMachineBehaviour.html*

For detailed tutorials, visit Manaspark’s YouTube channel:

https://www.youtube.com/channel/UCP3Mw1ITJFFYdY5N6mt-wdQ

For support, visit the Unity forum thread:

https://forum.unity.com/threads/one-logic-statemachine-high-level-state-machines-on-custom-code-base.624793/

… or just contact me via email: *max.schaefer@manaspark.studio*

- or via Twitter: @ManasparkStudio

- or via Discord: MaxFromManaspark#0494

I will always be happy to get your feedback!

If you like my tool, then please leave me a review on the Asset Store.

You have created something awesome and have used my tool for it?

Fantastic, please tell me about it or tag me on Twitter! ☺