CardboardCore StateMachine

Goal of this package

CardboardCore's StateMachine is made to be a very simple and quick to set up and use. Without too much overhead like creating traditional transitions. And, if written properly, re-usable states between different StateMachines.

How to use

States

To create a new State, create a new class which derives from `State`. Create the required `OnEnter` and `OnExit` overridden methods. And shove in some code that makes sense to you. The `OnEnter` method will automatically be called whenever this specific state is activated. The `OnExit` method will automatically be called whenever this specific state is deactivated.

The concept of States is keeping your code dependencies in check. Which means, sooner or later, you'll want to transition into another state. This is done by calling either `owningStateMachine.ToNextState()` or `owningStateMachine.ToState<MyNewState>()`. To learn which kind of transition to use. Please refer to **Static and FreeFlow** part of this document below.

StateMachines

To create a new StateMachine, simply create a new class which derives from `StateMachine`. Create/generate (depending on your IDE/approach) a constructor where the `base(bool enableDebugging)` is set accordingly.

In the constructor of the earlier mentioned StateMachine, add `SetInitialState<MyFirstState>()`. From here add new transitions using either `AddStaticTransition<FromState, ToState>()` or `AddFreeFlowTransition<FromState, ToState>()`. To learn which kind of transition to use. Please refer to **Static and FreeFlow** part of this document below.

Static and FreeFlow

This package offers support for both static and free form state transitions.

Static

A static flow means that you could either force your state to always transition into a specific "next" state. This "next" state is defined by the StateMachine's transitions. To create such a transition, simply call `StateMachine.AddStaticTransition` . This will create a transition that'll be called whenever `owningStateMachine.ToNextState()` is called from a specific state. Please refer to the example code for more detail.

FreeFlow

Next to the static approach, there's a more free-flow approach. To create such a transition, simply call `StateMachine.AddFreeFlowTransition` from the StateMachine you just created. To handle this transition, you'd need to call `owningStateMachine.ToState<YourSpecificState>()` from a State.

Combining

Every transition has to go "From" a specific state -> "To" a specific state: `Transition<FromState, ToState>` . This means that both `owningStateMachine.ToNextState()` and `owningStateMachine.ToState<YourSpecificState>()` require transitions to be in place.
Both Static and FreeFlow transitions can be combined within a single StateMachine. Please refer to "FreeFlowStateMachine" in the Demo code folder.

Improve your workflow

Both StateMachines and States are pure C# code. One can easily create MonoBehaviours and wrap them around their pure C# code.

Another solution would be to use Dependency Injection (DI). CardboardCore offers such a package which works complimentary with this package, which can be found here: https://assetstore.unity.com/packages/slug/204241.

Why DI?

Getting pure C# code in MonoBehaviours is quite simple. But getting references to MonoBehaviours, or any other Unity-related components, is quite tricky to do.
With CardboardCore's DI it's very easy to get a reference to any kind of Unity Object in pure C# code. Lowering the barrier between MonoBehaviours and C#.

Get if for free!

I'm willing to give you a free voucher if you've already paid for this package. Simply send your invoice number, order ID or product code for me to verify and I'll send you back your free voucher for CardboardCore's Dependency Injection. Details where to send can be found on the earlier shared Asset Store Page. Please note that one voucher is eligible for one DI package.