State Machine Zilch API

Manual

Thomas Komair

tkomair@digipen.edu

V 0.1

Contents

Code Files

- SMStateMachine.z
  - Contains the definition for the State Machine component.
- SMState.z
  - Contains the definition for the State component. This component does nothing and should only be used to derive new states to add to the game objects.
- AnimState.z:
  - Contains the definition for the AnimState component. This component derives from State and adds the animation switching mechanism by default. As the State component, it should only be used to derive new states to add to the game objects.
- SuperState.z:
  - Contains the definition for the SuperState component. This component derives from State and acts as a state machine. It is used to create nested state machines for more complex behaviors. As the state component it should only be used to derive new states to add to the game object.

State Component

Code

```
class State : ZilchComponent
{
    // Dependencies guarantee that the StateMachine
    // component is initialzied before the states.
    [Dependency] var StateMachine : StateMachine;

    // Timer that keeps track of the time in the state
    var TimeInState : Real = 0.0;

    // Game object to act on - this is taken from the state machine
    var Actor : Cog = null;


    // Override these functions to customize the behavior
    [Virtual] function Enter(){}
    [Virtual] function Exit(){}
    [Virtual] function Update(event : UpdateEvent){}

    // ---------------------------------------------------------------
    // ---------------------------------------------------------------
    // INTERNAL USE - DO NOT MODIFY THIS CODE

    [Virtual]function StateInitialize(init : CogInitializer)
    {
        // Debug
        Console.WriteLine("State `typeid(this).Name`::Initialize");

        // add the state to the owner state machine.
        this.Owner.StateMachine.AddState(typeid(this).Name, this);

        // Get the game object we act on from the owner state machine
        this.Actor = this.Owner.StateMachine.Actor;
    }
    [Virtual]function InternalEnter()
    {
        this.TimeInState = 0.0;
        this.Enter();
    }
    [Virtual]function InternalExit()
    {
        this.Exit();
    }
    [Virtual]function InternalUpdate(event : UpdateEvent)
    {
        this.TimeInState += event.Dt;
        this.Update(event);
    }
}
```

Description

The component contains the following variables and functions

Table: State

Variable	Description
State Machine [Dependency]	Guarantees that objects have a State Machine before adding any states.
TimeInState	Timer to keep track of the time
Actor	Game object on which the state machine is acting. This might be different from "this.Owner" in the case you are using nested state machines. Otherwise, this.Actor is the same as this.Owner. To be safe, always use this.Actor to act on the object. This variable is initialized by the state machine, and you shouldn't override it.
Function	**Description**
Enter	Virtual function. Meant to be overridden by the derived class. This will be called when a state machine changes its currently active state to this state.
Exit	Virtual function. Meant to be overridden by the derived class. This will be called when a state machine change state and this state is the currently active state.
Update	Virtual function. Meant to be overridden by the derived class. This will be called every frame update on the currently active state.
InternalEnter	Internal Use Only. Resets the timer and calls Enter (described above).
InternalExit	Internal Use Only. Calls Exit (described above). Although it doesn't do much, it is needed in order to override its behavior for other state types such as AnimState and SuperState.
InternalUpdate	Internal Use Only. Updates the timer and calls Update(described above).
StateInitialize	Internal Use Only. Finds the state machine actor and adds itself to the owner object state machine.

Creating a new State

To create a new state, create a new Zilch Component as you would always.

```
1  class Test : ZilchComponent
2  {
3      function Initialize(init : CogInitializer)
4      {
5          //Zero.Connect(this.Space, Events.LogicUpdate, this.OnLogicUpdate);
6      }
7
8      function OnLogicUpdate(event : UpdateEvent)
9      {
10     }
11 }
12
```

Replace “ZilchCompnent” to “State” to derive from the State class instead.

```
class Test : State
{

    function Initialize(init : CogInitializer)
    {
        // YOU MUST CALL THIS FUNCTION NO MATTER WHAT BEFORE
        // WRITING ANY OTHER CODE
        this.StateInitialize(init);

        //
        // WRITE YOUR INITIALIZATION CODE HERE.
        //
    }

    //
    // OVERRIDDEN STATE FUNCTIONS
    //

    [Override]function Enter()
    {
        // WRITE YOUR CODE HERE
    }
    [Override]function Exit()
    {
        // WRITE YOUR CODE HERE
    }
    [Override]function Update(event : UpdateEvent)
    {
        // WRITE YOUR CODE HERE
    }
}
```

Note that you don’t have to override all three functions if you don’t plan on using them, the state machine will work even if you don’t override any of the 3 states function (although doing so is pointless.)

State Machine Component

Code

```
class StateMachine : ZilchComponent
{
    // The name of the initial state of the state machine.
    // if specified, this will be the first state that the state machine
    // changes to when running for the first time and/or when reset.
    [Property]  var InitStateName : String = null;

    // Container for all the states.
    var StateMap : HashMap[String, State] = new HashMap[String, State]();

    // Variable to keep track of the current, previous and next states
    var CurrentState : State = null;
    var NextState : State = null;
    var PreviousState : State = null;

    // Game object to act on - found automatically on initialize
    var Actor : Cog = null;


    // Follows the hierarchy from the bottom up (child->parent->grandparent,etc...)
    // until it finds the first object that has a state machine component. The root.
    function GetRootOwner() : Cog;

    // Returns the root object's state machine component.
    function GetRootStateMachine() : StateMachine;

    // Searches for and return the state machine component of the object
    // whose name matches the specified one. The search starts at the current
    // object and follows the top-down ordering (parent->child->grandchildren etc...)
    function FindStateMachineFromObj(smName : String) : StateMachine;

    // This will navigate the whole object hierarchy (i.e. starting from the actor object)
    // to try and find the state machine of the object of the given name.
    function FindStateMachineFromRoot(smName : String) : StateMachine;

    // ChangeState. Does exactly what it's name suggest. Note that the state change
    // will only take effect once the state machine is updated. Therefore, only the last
    // call to ChangeState before the state machine is updated will determine the new state.
    function ChangeState(nextState : State);

    // This is a helper function that searches for the state by name and changes to it.
    // this can be useful when you don't have access to the state component directly.
    function ChangeState(nextStateName : String);

    // This will reset the state machine. Similarly
    function ResetStatMachine();

}
```

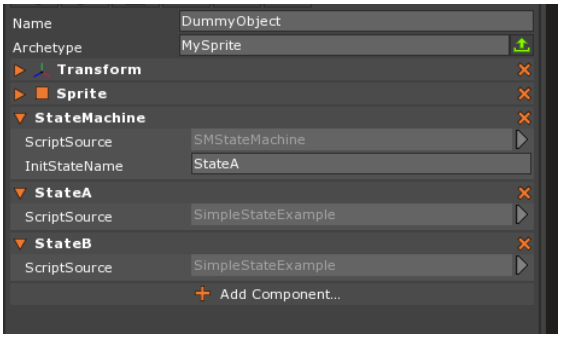
Note: The state machine functions have not been expanded in order to keep the size of this document short. Please refer to the zilch script for the full implementation of the state machine component.

Description

The StateMachine component acts as a *container* for states. Specifically, it uses a HashMap[String, State] to be able to store each state and easily retrieve it by name. (Make sure to read the code snippet on HashMap to learn more about the HashMap container.)

Figure: Example HashMap for a simple state machine

Key : String	Value : State
“StateA”	this.Owner.SM_Walk
“StateB”	this.Owner.SM_Jump



- These are components on the game object.
- They all derive from State.
- **IMPORTANT:** The SM class takes the name of the component automatically using typeid() function

The StateMachine component is unique. Unlike the State component (see above), it is not meant to be derived from and instead it is meant to be unique.

The StateMachine component is designed to support the implementation of simple finite state machines as well as nested state machines and as such, most of the functions above are useful when working with nested state machines. A brief description of these functions is given in the comment above the function.

For simple state machines, you would only really need the ChangeState function, which does basically just this, it changes states. Technically, the ChangeState function simply sets the next state to the given state:

```
function ChangeState(nextState : State)
{
    if(nextState == this.CurrentState)
        return;

    if(nextState != null)
    {
        this.NextState = nextState;
    }
}
```

The state change actually happens in the UpdateStates function, which is called when the LogicUpdate event is triggered. **IMPORTANT:** Note that if multiple calls to ChangeState are made before the StateMachine updates, only the last call will actually take effect.

When a state is changed, the state machine calls the InternalExit function on the current active state (CurrentState) and InternalEnter on the next state (NextState). Additionally, if there are no sate changes, UpdateStates calls InternalUpdate.

You can also reset the state machine if you so desire but you would probably need it on rare occasions. ResetStateMachine will internally call ChangeState by passing the initial state as an argument, and setting CurrentState and NextState to null. Note that Enter and Exit will not be called if the StateMachine is reset.

All the other functions (FindStateMachine, GetRootOwner, etc…) are used in the context of nested state machines (see below). In the case of a simple state machine, these function will return the either this.Owner o this.Owner.StateMachine depending on the return value.

Utility States

At this point, you're probably wondering why there are two versions of the state functions:

- InternalEnter/Enter
- IntenalExit/Exit
- InternalUpdate/Update

All of them are virtual, so they can be overridden. So what's the point? Well, when developing state machines, you will often find that even though some of the code is different, a lot of it is exactly the same. For example, most of your states will actually act as simple transitions, designed to wait for a special type of event to happen to trigger a state change, or a certain amount of time to pass.

By dividing the state functions into 2 versions, we can place repetitive code in internal functions and override the normal functions for state-specific customization. These would serve as "utility states".

Finally, by using the fact that States are components, we can add property variables to the utility states therefore reducing the tedious labor of creating interesting behavior.

Below you will find some example of utility states. The AnimState makes the work of changing animations between states automatic and the SuperState shows how to extend the State to support nested state machines.

AnimState:

CODE

```
class AnimState : State
{

    // Sprite Animation Data
    [Property] var StateAnimEnabled : Boolean = false;
    [Property] var StateAnimForceReset : Boolean = false;
    [Property] var StateAnim : SpriteSource = null;
    [Property] var StateAnimStartFrame : Integer = 0;
    [Property] var StateAnimSpeed : Real = 1.0;
    [Property] var StateAnimFlipX : Boolean = false;
    [Property] var StateAnimFlipY : Boolean = false;
    [Property] var StateAnimLoop : Boolean = false;

    // Internal enter is changed from its original
    [Override]function InternalEnter()
    {
        // Change state animation BEFORE
        // so that custom State OnEnter can override the
        // modifications.
        this.StateAnimChange();

        // Call base class internal enter.
        (this as State)~>InternalEnter();
    }
```

```
function StateAnimChange()
    {
        if(this.Actor.Sprite != null && this.StateAnimEnabled && this.StateAnim != null)
        {
            // Change to new anim or reset the animation if necessary
            if(this.Actor.Sprite.SpriteSource != this.StateAnim || this.StateAnimForceReset)
                this.Actor.Sprite.SpriteSource = this.StateAnim;

            // Change state data
            this.Actor.Sprite.StartFrame = this.StateAnimStartFrame;
            this.Actor.Sprite.AnimationSpeed = this.StateAnimSpeed;
            this.Actor.Sprite.FlipX = this.StateAnimFlipX;
            this.Actor.Sprite.FlipY = this.StateAnimFlipY;
            this.Actor.Sprite.SpriteSource.Looping = this.StateAnimLoop;
        }
    }
}
```

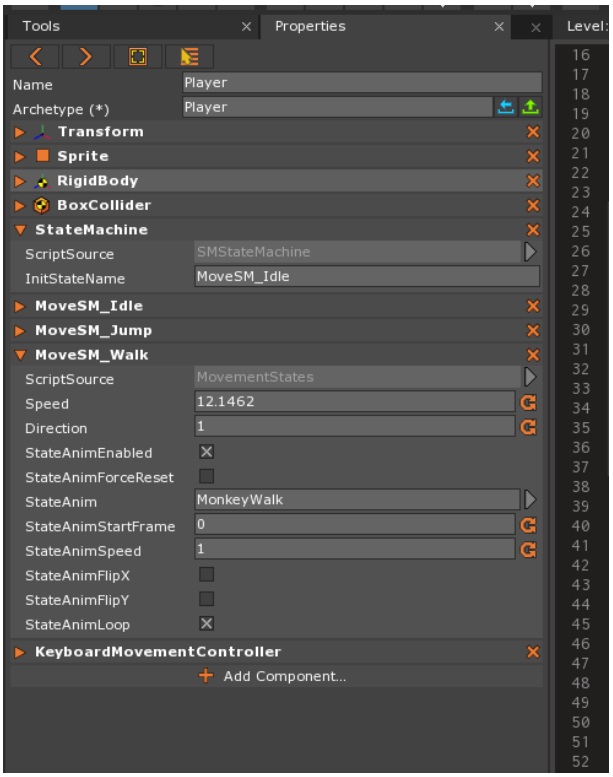
Description

The component contains the following variables and functions. Please remember that because AnimState ***derives*** from State, it also has all of the functions and variables described in the table in the State section.

Table: AnimState

Variable	Type	Description
StateAnimEnabled	Boolean	Specifies whether the AnimState should overwrite animation data. If false, the AnimState would act as a regular State component.
StateAnimForceReset	Boolean	Specifies if the sprite animation should be changed even when the sprite is already playing that animation. If false, then the AnimState will change animation if and only if the current animation on the Actor is different than the one in StateAnim.
StateAnim	SpriteSource	The sprite source to take the animation from.
StateAnimStartFrame	Integer	Specifies the start frame of the animation.
StateAnimSpeed	Real	Specifies the speed of the animation for this state
StateAnimFlipX	Boolean	Specifies whether the animation should be flipped on the X (i.e. horizontally) or not
StateAnimFlipY	Boolean	Specifies whether the animation should be flipped along the Y axis or not (i.e. vertically).
StateAnimLoop	Boolean	Specifies whether or not the animation should loop.

Similar to the basic State component, you must create a new component and ***derive*** it from the AnimState class in order to add it to your game object.



SuperState

The SuperState class also derives from State and works exactly in the same way (i.e. you create a new component and derive from this class and ***override*** its Enter, Exit and Update function. The main role of the SuperState class is to act as both a state **AND** a state machine. The following section will explain the motivation behind the design of the super state class, what additional features it aims to provide and how it is implemented.

Motivation:

In games, AI behaviors can be quite complex, and sometimes using a simple state machine might not be enough. Sure, you could probably make almost any behavior work with just a simple state machine, but it will probably mean a really large set of state with lots of duplicate code, and ultimately will be very hard to follow, debug and improve.

Two major improvements on the base StateMachine improvements will be the ability to run *concurrent and nested state machines*.

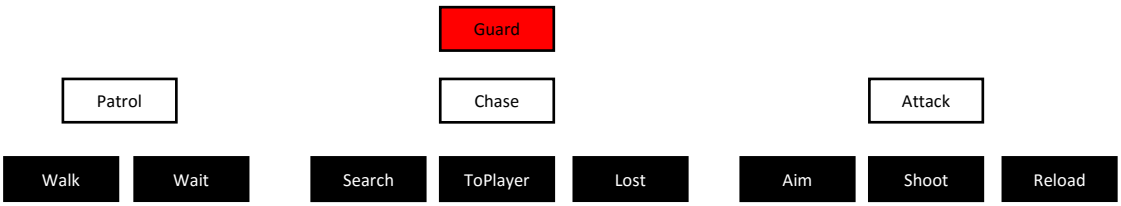
Concurrent state machines are exactly what their name suggests, state machines running in parallel. Remember that one state machine can only be in one state at a time, making it impossible to have two different set of behaviors acting at the same time (for example: movement and targeting).

Nested State Machines are state machines where one or more states have their own sub-states. Nested state machines allow to encapsulate complex states into their own state machines and help with debugging, code maintenance and refining state behaviors. As an added bonus, we will see how we can take advantage of the COG hierarchy to save our behaviors to archetypes making merging smoother.

Implementation:

By design, in the Zero Engine it is not possible to add two components of the same type, they can derive from the same type, but they cannot be of *exactly* the same type. This is why, to use them you have to create a new component and derive from either State, AnimState or SuperState or any other utility state you create for yourself. It's also the same reason why you cannot simply add another StateMachine component on the object to get concurrent state machines.

The solution here would be to leverage the object hierarchy by treating a game object as a sub state machine. Indeed, we could look at our state machines in terms of parent child relationship. For example, in the case of a complex enemy AI, we would have:



Guard is the root state machine, Patrol, Chase and Attack are sub-state machines, and the black boxes are simple states. Here The COG hierarchy equivalent could be:

- Guard
  - Patrol
  - Seek
  - Attack

Guard, Patrol, Seek and Attack are all game objects with a StateMachine component. Guard, being at the top of the hierarchy, is the actor game object. All other states will act on this object.

However, this is not purely speaking a nested state machine. In this occurrence, if we wanted the guard to patrol, seek and attack **at the same time**, we would only need to add to Patrol, Seek, and Attack objects their respective States components. Here, we in fact have an example of ***concurrent state machines.***

For this to truly be a ***nested state machine***, the guard can only be in one of the 8 states listed above (black boxes) at any single time. Here, what we need is to extend the basic set of features of the State component, while at the same time maintaining the integrity of the base State Machine API.

The idea would be to derive a new Utility State to be able to *act as a state machine proxy*. In other words, it is a state that internally controls a state machine. For lack of a better word, I called this class the SuperState:

CODE

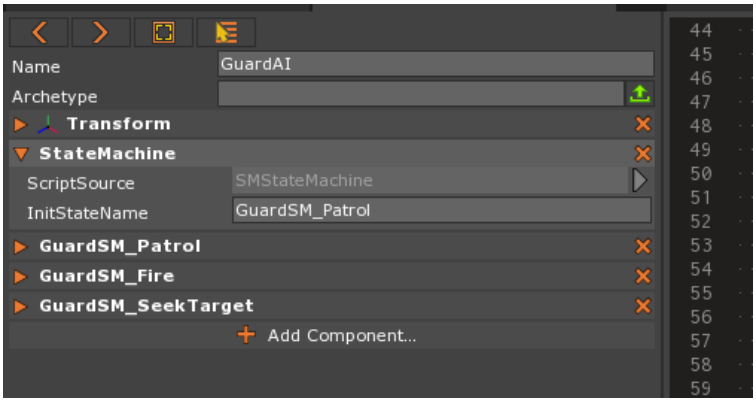
```
class SuperState : State
{
    // The state machine that holds the sub states of the SuperState.
    var SubStates : StateMachine = null;

    // StateInitialize. automatically tries to find the child object
    // whose name matches the name of the class that derives from SuperState.
    // For example, if you derive a SuperState and you call it "Attack", then
    // the child object in the hierarchy must also be called "Attack"
    [Override] function StateInitialize(init : CogInitializer);

    // Internal Enter: Call default InternalEnter to insure that Enter is called,
    // then resets the sub state machine and force Enter to be called on the initial state
    [Override] function InternalEnter()
    {
        // Call internal enter as base state
        (this as State)~>InternalEnter();

        // reset the sub state machine and force Enter to be called on
        // the initial state.
        if(this.SubStates != null)
        {
            this.SubStates.ResetStatMachine();
            this.SubStates.DelayedInitialize(); // force new state to be set
            // call enter on the current sub state
            if(this.SubStates.CurrentState != null)
                this.SubStates.CurrentState.InternalEnter();
        }
    }

    // Internal update: Calls default update and then update the sub-state machine.
    // This is useful to have global logic on top of the sub-states.
    [Override] function InternalUpdate(event : UpdateEvent)
    {

        // call internal update as base state
        (this as State)~>InternalUpdate(event);

        // update the sub state machine
        if(this.SubStates != null)
            this.SubStates.UpdateStates(event);
    }

    // Internal Exit: Calls default internal exit and insure that current state exit is called.
    // Notice that the order is the opposite of InternalEnter.
    [Override] function InternalExit()
    {
        // call enter on the current sub state
        if(this.SubStates != null && this.SubStates.CurrentState != null)
            this.SubStates.CurrentState.InternalExit();

        // call internal exit  as base state
        (this as State)~>InternalExit();
    }
}
```

Note: Not all functions are shown in order to keep the size of this document short. Please refer to the zilch script for the full implementation of the SuperState component.

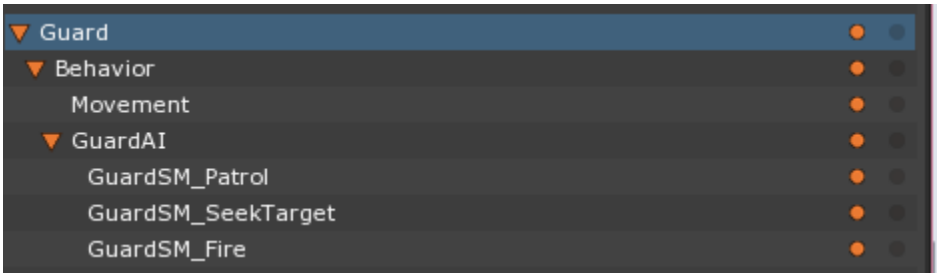
As you can see, the SuperState overrides the Internal State function to act both as a regular state (highlighted: the default Internal functions are still called) AND as a state machine.

Usage

Once again, to create SuperStates, you will have to derive a new class from SuperState that you will attach to a game object:



Then create new objects in the hierarchy that will actually hold the sub-states:



IMPORTANT: The name of the game object must match the name of the class derived from SuperState. In this case GuardSM_Patrol, GuardSM_SeekTarget and GuardSM_Fire.

Finally, add the states onto the sub-state objects.