Manual

for “Graphical State Machine” Asset

1. Introduction

This asset is an editor extension for the Unity3D editor. It allows you to create finite state machines using a graphical state machine editor.

Every state provides four callbacks for managing your machine's functionality. Those can be applied by dragging and dropping *GameObjects* from your scene to your states. When you start your machine, the selected objects will be searched once and registered in order to guarantee the best performance.

There is no coding required to change active states except sending single strings (triggers).

2. Getting Started

2.1 Installation

Click **Download** on the Asset Store page of this asset. A popup will show where you need to insert your E-mail-address and your password of your Unity3D-Account if you haven't done it yet.

After downloading, the button will change to **Import**. Click here and select all elements in the next popup. Click **Import** again.

2.2 Setting up the editor

After importing the asset bundle, you will be able to open the *Graphical State Machine Editor (GSM Editor)* using **Window > Graphical State Machine Editor**. You can dock or maximize this window like any other Unity-Window.

If you already have a state machine file, you can double-click it and the window will open as well.

2.3 Creating Machines

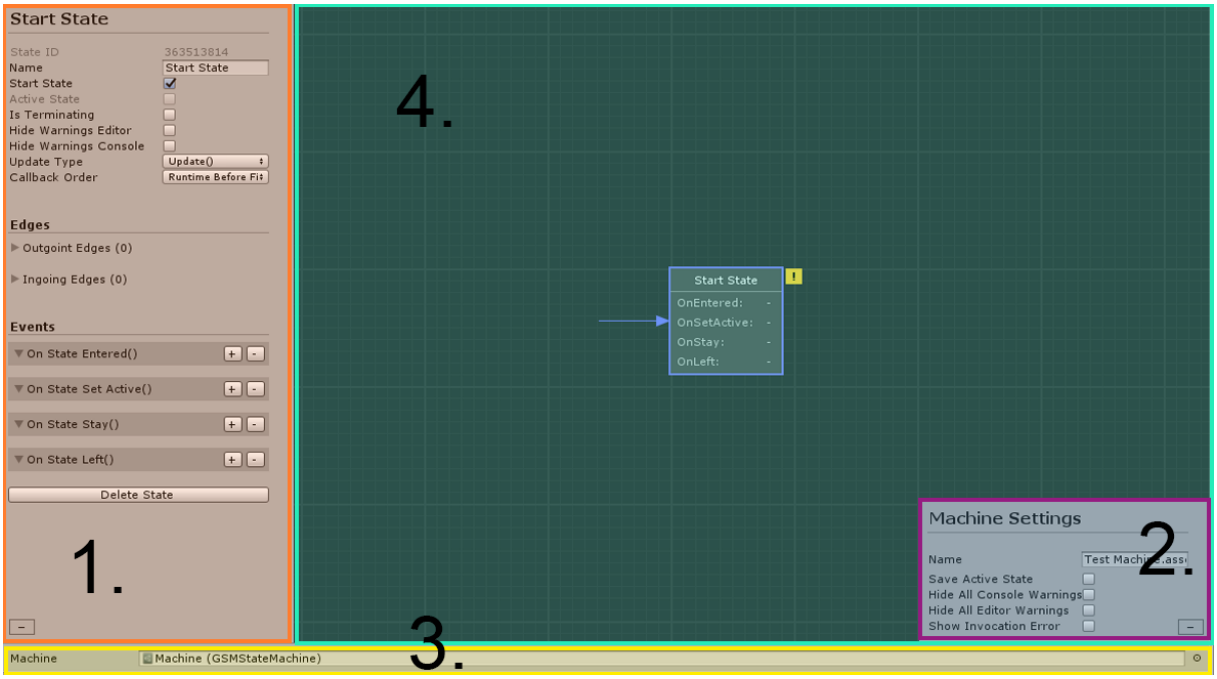
One state machine will be given by one state machine file. To create a machine file, click right anywhere in your *Assets folder* and choose **Create > Graphical State Machine**. A new file will be created which you can name however you want. To modify this machine, double-click the file or drag and drop it into the **Machine** slot on the bottom of the state machine editor.

3. Creating Machines

3.1 Using the editor

After opening the GSM Editor (see 2.2 Setting up the editor) drag and drop a machine file to the slot Machine located at the bottom of the Editor (see 2.3 Creating Machines) or double-click any machine file. You should be able to see a state called *Start State* in the middle of the screen. If not, right-click on the dark area and click **Focus Start State**.

The editor is split into four parts. The inspector (1.) which allows you to modify states and edges. The Machine Settings (2.) where you can set a name and change options for your machine. The Machine Reference (3.) to put your machine in and the machine editor (4.) to create states and edges.



3.1.1 The inspector (1.)

Shows information about a selected state or edge and allows you to modify your selection. You can close and open the inspector by clicking the small button in the bottom left corner. You can read more about the inspector's functionalities in the section *3.2 States.*

3.1.1 Machine Settings (2.)

You can modify the machine's name and three options here:

- **Save Active State**
  When checked, the first state which will be active on game start is the last state which was active when you closed the game the last time.
  Imagine having two states *A* and *B*. You run your game the first time and start with state *A*. Then you set state *B* active and shut down the game. If **Save Active State** is enabled and you start the game again, state *B* will be set active. If not, state *A* will be set active because it is the start state.

- **Hide All Console Warnings**
  When starting the machine, it will be tested if there are some unreachable states beside some other tests. If you check this box, there will be no warnings in the console when starting your machine.

- **Hide All Editor Warnings**
  Having a state which cannot be left or reached may be a mistake, so the editor warns you using a yellow box containing an exclamation mark. If you do not want to see the warnings, you can check this box to hide all editor warnings of the current machine.

- **Show Invocation Error**
  If you select this check box, there will be an error if an event could not be called, e.g. because of a missing function.

3.1.1 Machine Reference (3.)

This is the slot in which you put the machine file. The currently edited machine file is always the file which is contained by the machine reference slot. Setting the reference to *None* will not delete the file.

3.1.1 Machine Editor (4.)

The machine editor is the place where you add and remove states and edges.

Press, hold and move the middle mouse button to move the view. Clicking here with the right mouse button gives you some options:

- **Add State**
  Adds a new state at the clicked position *3.2 States*
- **Focus Machine**
  Calculates the midpoint of your machine and moves the view so the midpoint is in the middle of the editor

- **Focus Start State**
  Moves the view so the start state is in the middle of the editor

- **Focus Active State**
  Moves the view so the active state is in the middle of the editor. This option is only enabled if the machine is running or if you save the active state.

- **Focus State**
  Lists you all states in the current machine. You can focus each one of them.

3.2 States

3.2.1 Creating States

To create a state, right-click somewhere in the machine editor and click **Add State**. The new state will be created at the point, you clicked and will take a new unique name. There will be no edges connected to this state and no callbacks registered.

3.2.2 Duplicate States

Right-click a state, you want to duplicate and choose **Duplicate**. The created state will have the same name with *Copy* at the end. If you copy the start state, the duplicated state will not become start state as well because you can only have one start state per machine.

All callbacks including its parameters will be copied.

3.2.3 Delete States

When deleting a state, all connected edges (ingoing and outgoing) and callbacks will be deleted too.

There are multiple ways to delete a state.

Right-click a state and click **Delete**

Select a state by left-clicking it and press the Delete-Key on your keyboard.

After selecting a state there will be a **Delete State** button on the bottom of the inspector (you may scroll down to reach it)

3.2.4 Modify States

To modify a state, you need to select it first. You can do this by left-clicking it or choosing **Edit** after right-clicking it. The inspector on the left side will show an interface giving you different opportunities to modify the state.

- **State ID**
  the unique ID of the state. You cannot (and should not) change this. It can be used to set the active state by code (not recommended).

- **Name**
  Name of the state. Make sure to have unique state names. Having two states with the same name may cause errors.

- **Start State**
  If checked, the state will be start state. The previous one cannot be start state anymore. You cannot uncheck this except by selecting a different state as start state. This option can also be changed by right-clicking the state.

- **Active State**
  Sets the state as active state. You can only change this option if you enabled **Save Active State** in your machine settings or if the machine is running. This option can also be changed by right-clicking the state.

- **Is Terminating**
  When a terminating state is set active, the machine will be stopped after executing **OnStateSetActive()**. Even if you have **Save Active State** enabled, the machine will start at start state next time (otherwise it would be stopped when started).
  This option can also be changed by right-clicking the state.

- **Hide Warnings Editor**
  If you have checked this box, there will be no warnings shown inside the machine editor. Note, that you can also **Hide All Editor Warnings** in the machine settings. If this option is activated there will be no warnings, even if you uncheck **Hide Warnings Editor** for one state. This option can also be changed by right-clicking the state.

- **Hide Warnings Console**
  If you have checked this box, there will be no warnings shown in the console when starting the machine. Note, that you can also **Hide All Console Warnings** in the machine settings. If this option is activated there will be no warnings, even if you uncheck **Hide Warnings Console** for one state. This option can also be changed by right-clicking the state.

- **Update Type**
  You can set this to either Update(), FixedUpdate() or LateUpdate(). Depending on your choice, the **OnStateStay()**-Event will be called. Therefore, if you select

Update(), it will be called each Update()-call (see *3.2.6 Events*).

- **Outgoing Edges**
  Shows a list of outgoing edges. You can either select the edge or the target state of any edge. You can also modify the trigger of the edge. For more information, see *3.3 Edges.*

- **Ingoing Edges**
  Shows a list of ingoing edges. There are the same options for this as for the outgoing edges.

- **Events**
  see *3.2.6 Events.*

3.2.5 Editor Warnings

A warning inside the editor will be indicated by a yellow rectangle containing a black exclamation mark. Hovering the cursor over it will show its message. You can hide editor warnings by either checking **Hide Warnings Editor** on the state itself or by checking **Hide All Editor Warnings** in the machine settings.

These are the possible warnings:

- **No ingoing edge**
  Having no ingoing edge leads to not being able to set the state active except by code, which you should not do.

- **No outgoing edge**
  States without outgoing edges cannot be left. This can be used to set the machine as finished. But you should stop the machine by code instead.

- **Duplicate names**
  Having two states with the same name may cause problems. Setting the active state by code when having two states with the same name sets the one with a lower ID active.

- Blablablabla <3

3.2.6 Events

Every state contains four events: OnStateEntered(), OnStateSetActive(), OnStateStay(), OnStateLeft(). Each event can hold any number of callbacks.

To add a callback, click the plus-button beside the event's name. If you have selected a callback, the new callback will be a copy of the selected one.

To remove a callback, select it by clicking it with the left mouse button (it will turn blue) and clicking the minus-button afterwards.

To apply a function to a callback, drag and drop a *GameObject* from your scene to the object field of a callback. In the method field, select the component your method is laying on and select your method afterwards.

Your method must have either no, or one parameter. Allowed parameter types are string, float, bool or int.

You can change the parameter in the parameter field.

- **OnStateEntered()**
  Gets called when the state is set active. When executing the function, the state is already active.

- **OnStateSetActive()**
  Always getting called right after **OnStateEntered()**. The difference is, that this event is also called on the first active when starting the machine. You should consider using this event instead of **OnStateEntered()** when having **Save Active State** enabled.

- **OnStateStay()**
  This is called once per *Update()*-call when state is active. This is where you should implement your game loop. You can determine if you want to call it once per *Update()*, *LateUpdate()* or *FixedUpdate()*.

- **OnStateLeft()**
  Getting called when the state is left to set another state active. When calling, the state will still be active.

Sometimes, the order in which the callbacks of one event get executed is important. All callbacks will be executed from top down. To change the order, just drag and drop a callback to its target position to swap it. Another way to change the order is right-clicking a callback and choosing either **Move Up** or **Move Down**

3.2.6 Applying Events at Runtime

You can apply events to all four state-events as well as the edge event at runtime. Those events will not be saved in the machine file and therefore be deleted when the game stops.

This can be used to apply callbacks of dynamically instantiated prefabs which did not exist when creating the machine.

Here is a quick example:

```
var newObject = Instantiate(prefab);

machine.GetStateByName("Game Running")
      .RegisterRuntimeCallback(
            GraphicalStateMachine.RuntimeCallbackType.OnStateStay,
            newObject.GetComponent<MyComponent>().MyCoolUpdateFunction);
```

This creates a new GameObject called `newObject`. It has a Component `MyComponent` where you want `MyCoolUpdateFunction()` to be called each Update-call while the state "Game Running" is active.

When the machine is stopped, all runtime callbacks will be deleted. If you do not intend this to happen, you may need to do different things depending on your way to stop:

If you stop your machine by code using `machine.Stop()` just call `machine.Stop(false)`. The bool determines whether the callbacks should be deleted or not.

If you stop your machine by the Unity-inspector or by reaching a terminating state, just uncheck the checkbox **Clear Callbacks on Stop** on your `StateMachineProcessor`.

3.3 Edges

An edge is used to create directional transitions between states. Each edge holds a string (trigger) which allows the origin state to use the edge and set the target state active.

If the currently active state *A* has an outgoing edge with the trigger *trigger* targeting state *B* and you send the string "*trigger*" using `machine.SendTrigger("trigger")`, *A* will be left and *B* will become active.

3.3.1 Creating Edges

To create an edge, right-click a state from which you want an edge to emerge and select **Make Edge**. When moving the cursor now, an arrow starting from the selected state will target your cursor. To cancel, press escape or right-click. When clicking another state, the edge will set down. Multiple edges from the same state to the same target are not allowed. Selfloops are allowed but in most cases not needed.

3.3.2 Deleting Edges

Select an edge by clicking the rectangle in the middle of the edge. If you now press delete on your keyboard, the edge will be deleted. Both states (target and origin) will stay.

Another way to delete an edge is by first selecting it and then clicking **Delete Edge** in the inspector.

3.3.3 Modifying Edges

Select an edge by clicking the rectangle in the middle of the edge. The inspector will show an interface in order to modify the edge.

- **Trigger**
  The string you need to send to use the edge.

- **Origin**
  By clicking this button, you will select the origin state of the edge

- **Target**
  By clicking this button, you will select the target state of the edge

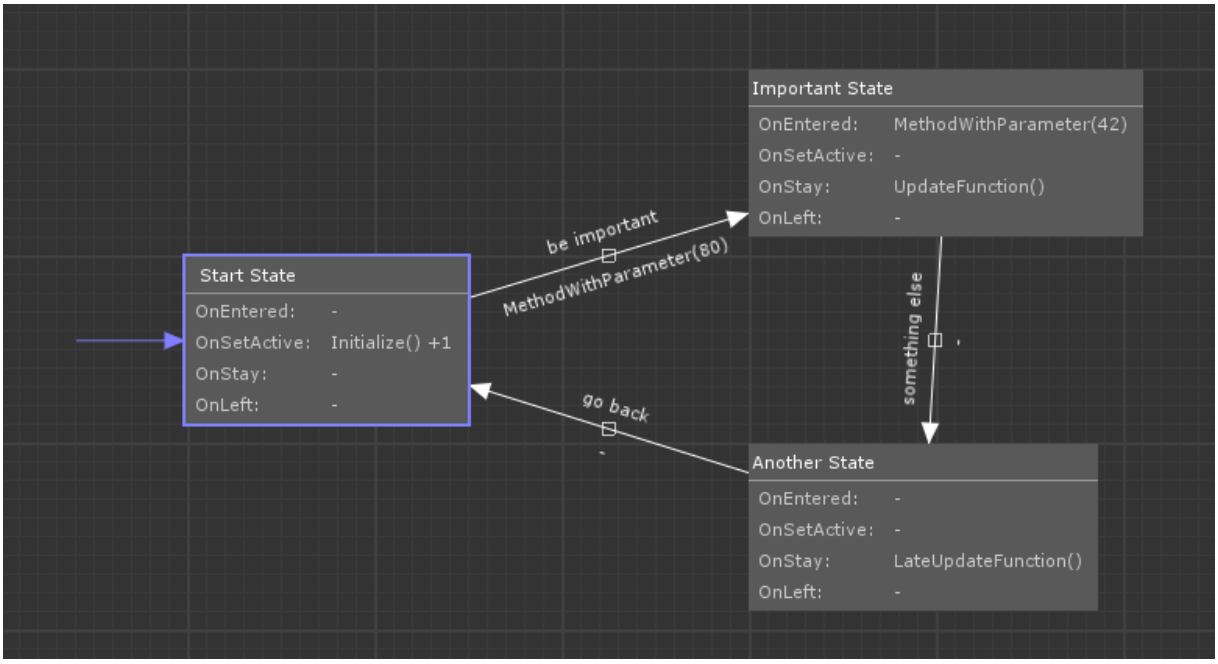
- **On Edge Passed()**
  This is the event which is called when using the edge right after **OnStateLeft()** on the origin state and before setting the target state active. For more information about how to use events see *3.2.6 Events.*

3.3.4 Edge Errors

If an edge has a terminating state as origin, you will notice a warning.

When a terminating state is set active, the machine will stop. Therefore, the state does not need any outgoing edges.

3.4 Example Machine



The start state of this machine is *Start State*, indicated by the arrow, targeting it while not having an origin. This state has a blue border as it is currently selected.

When setting this active, Initialize() and one more function will be called. This happens, when starting the machine and when setting the state active by trigger.

If you send the trigger *be important* which you can see on top of the edge between *Start State* and *Important State*, the edge will be used and MethodWithParameter(80) will be executed. After that, *Important State* will be active and MethodWithParameter(42) gets executed.

Each time, Update() is called, UpdateFunction() is called as well while *Important State* is active. When sending trigger *something else*, the state will be changed to *Another State* and LateUpdateFunction() gets called each LateUpdate() call instead of UpdateFunction() each Update() call.

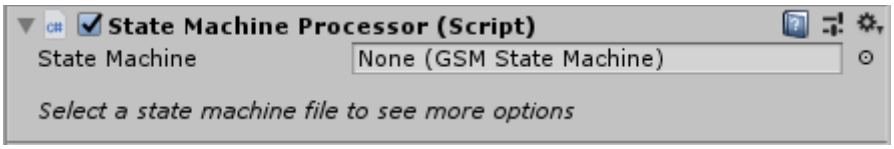
Note that the names of the functions make no difference. You can determine the update type in the inspector when selecting a state.

4. Using Machines in Games

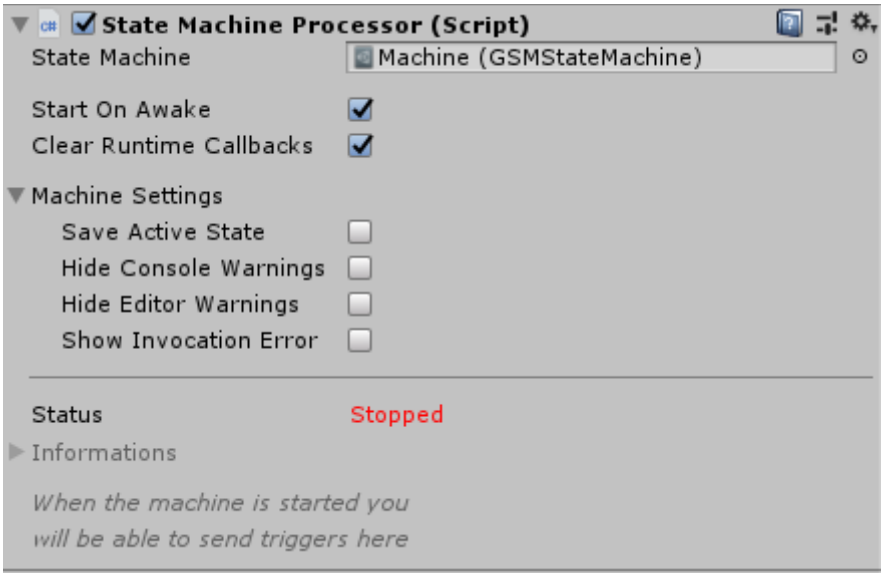
4.1 Setting up State Machine Processor

You can use any number of machines in each scene and even every machine in any number of scenes.

To use a machine, add the component **StateMachineProcessor** to any *GameObject* in your scene.



To specify which machine to use, drag the machine file into the **State Machine** Slot.

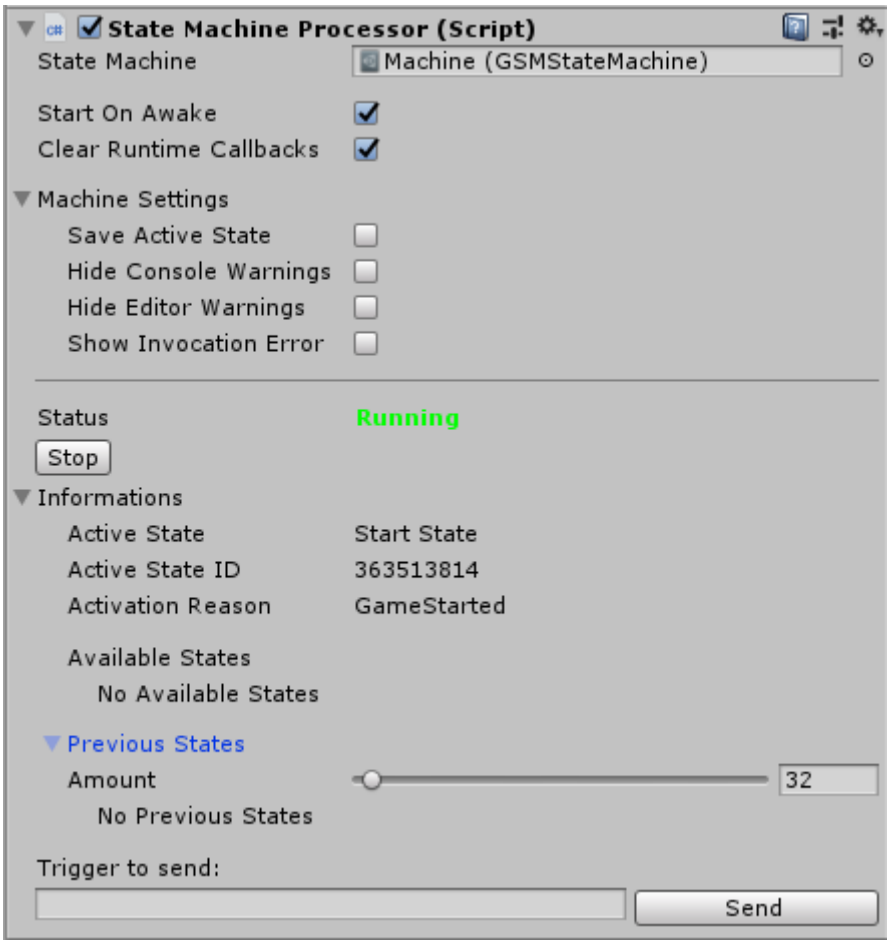


When checking **Start On Awake**, the machine will automatically start running when the scene is loaded. The first active state will be the one which is set as start state. If you have enabled **Save Active State**, the first active state will be the last one which was active when you stopped the machine the last time. If there is no such state, the default start state will be used.

When setting **Clear Callbacks on Stop** to true, all runtime callbacks will be deleted when the machine is stopped.

Under Machine Settings you can also modify all machine settings just like inside the editor (see *3.1.1. Machine Settings*).

If you are in play-mode, you will be able to start and stop machines using the button below *Status.*



When the machine is running, you will be able to send triggers manually by this **StateMachineProcessor**. Just type the trigger you want to send into the text field and press send.

There will also be some information about the status of the machine.

- **Active State**
  The name of the state which is currently active

- **Active State ID**
  The ID of the state which is currently active

- **Activation Reason**
  Reason for this state being active. Those can be:
  - **None**: If there is no active state (even though this should not be the case).
  - **GameStarted**: The machine has just been started and this is the state which was set active at the beginning.
  - **Edge**: The state was reached by an edge using a trigger.
  - **SetByName**: The state was set by name via script.

- **Available States**
  The states which can directly be reached using an edge. You can see the trigger

you need to use to reach the state and the name of the target state.

- **Previous States**
  The machine saves previous states up to an amount which you can specify using the **Size** slider.
  Beneath that you will see the log of all saved previous states.

4.2 Sending Triggers

The `StateMachineProcessor`-class and the `GraphicalStateMachine`-class provide a function called `SendTrigger(string)`. For more information see *5. Code Documentation.*

5. Code Documentation

All classes can be found in namespace `GSM`

5.1 StateMachineProcessor

Is a component for using machines inside scenes (see 4. Using Machines in Games).

Static Properties

Name	Type	Accessors	Description
`RunningMachines`	`List<GraphicalStateMachine>`	get	Returns all currently running machines.

Public Methods

Name	Return	Description
`IsMachineRunning(string)`	`bool`	Checks if a machine is running. Returns true if the given machine is running
`GetMachine(string)`	`GraphicalStateMachine`	Finds a registered machine by its name and returns it if it is running. Returns null if not.

Public Fields

Name	Type	Description
`startMachineOnAwake`	`bool`	If set to true, the machine will automatically be started on awake.

Properties

Name	Type	Accessors	Description
`Machine`	`GraphicalStateMachine`	`get`	Reference to the machine
`ActiveState`	`GraphicalState`	`get`	Currently active state. May be null if machine is not running
`IsRunning`	`bool`	`get`	True if the machine is currently running

Public Methods

Name	Return	Description
`SendTrigger(string)`	`bool`	Sends a trigger to the machine. If ActiveState has an outgoing edge with this trigger, the edge will be used to set the target state active.

		Returns true if there was a state change
SendTrigger(string, out GraphicalState)	bool	Sends a trigger to the machine. If ActiveState has an outgoing edge with this trigger, the edge will be used to set the target state active. Gives a reference to the new active state. Returns true if there was a state change
SendTriggerDelayed(string, float)	void	Sends a trigger to the machine after the given amount of time (in seconds). If ActiveState has an outgoing edge with this trigger, the edge will be used to set the target state active. If there was a state change within the delay time, the trigger will be sent anyways.
SetActiveState(string)	GraphicalState	Sets the currently active state and returns it. Throws UnknownStateException if there is no state with such name. Does not work while machine is stopped.
StartMachine()	bool	Starts the machine. Does not work if the machine is already running. Returns true if the machine was started.
StopMachine()	void	Stops the machine if it is running. Clears all runtime callbacks if clearRuntimeCallbacksOnStop is set to true

5.2 GraphicalStateMachine

Represents a state machine. You can control your machine with this class but not modify it.

Public Fields

Name	Type	Description
previousStatesSize	int	Number of states being saved in state history
clearRuntimeCallbacksOnStop	bool	If set to true, all runtime callbacks will be deleted when the machine is stopped

Properties

Name	Type	Accessors	Description
PreviousStatesSize	int	get, set	Number of states which are saved in PreviousStates list
PreviousStates	List<GraphicalState>	get	List of states which were active earlier. The most recent one will be at index 0
ActiveState	GraphicalState	get	Currently active state. Returns null if machine is not running
StartState	GraphicalState	get	Start state which is going to be active when the machine starts
States	List<GraphicalState>	get	List of all states
Edges	List<GraphicalEdge>	get	List of all edges
HasStartState	bool	get	True if there is a start state
Name	string	get	Custom name of the machine
IsRunning	bool	get	True if machine is currently running
SaveActiveState	bool	get	True if you want to save the active state
StateActivationReason	ActivationReason	get	Reason why the currently active state is active. Possibilities are: None, GameStarted, Edge and SetByName
Triggers	List<string>	get	List of all triggers

Public Methods

Name	Type	Description
GetOutgoingEdges(GraphicalState)	List<GraphicalEdge>	Returns a list of edges with start at the given state
GetIngoingEdges(GraphicalState)	List<GraphicalEdge>	Returns a list of edges which target the given state
GetStateByID(int)	GraphicalState	Returns the state with the given id. May be null if there is no such state
GetEdgeByTrigger(string)	GraphicalEdge	Finds an edge with the given trigger. If there are multiple edges with the same trigger it will return the first one found
GetEdgesByTrigger(string)	List<GraphicalEdge>	Returns a list of all edges with the given trigger
GetStateByName(string)	GraphicalState	Returns the state with the given name. If there are multiple states with the same name it will return the first one found.
Validate()	bool	Proofs the machine for some warnings and errors. Returns true if the machine can start.

		Gives a reference to ValidationResults object
SendTrigger(string)	bool	Sends a trigger to the machine. If ActiveState has an outgoing edge with this trigger, the edge will be used to set the target state active. Returns true if there was a state change
SendTrigger(string, out GraphicalState)	bool	Sends a trigger to the machine. If ActiveState has an outgoing edge with this trigger, the edge will be used to set the target state active. Gives a reference to the new active state. Returns true if there was a state change
CleanUpPreviousStates()	void	Removes states from PreviousStates list if there are more than PreviousStatesSize elements
SetActiveState(string)	GraphicalState	Sets the currently active state and returns it. Throws UnknownStateException if there is no state with such name. Does not work while machine is stopped.
Start()	bool	Starts the machine. Does not work if the machine is already running. Returns true if the machine was started.
Stop()	void	Stops the machine if it is running. Removes all registered runtime callbacks
Stop(bool)	void	Stops the machine if it is running. You can determine if you want all registered callbacks to be removed.
ClearRuntimeCallbacks()	void	Removes all runtime callbacks on edges and states

5.3 GraphicalState

Properties

Name	Type	Accessors	Description
Name	string	get	Name of the state
ID	int	get	ID of the state
OutgoingEdges	List<GraphicalEdges>	get	List of outgoing edges
IngoingEdges	List<GraphicalEdges>	get	List of ingoing edges
Outgrade	int	get	Amount of outgoing edges
Ingrade	int	get	Amount of ingoing edges
IsActive	bool	get	True if the state is currently active
IsTerminating	bool	get	True if the state is terminating. When setting a terminating state active, the machine will stop
UpdateType	StateStayUpdateType	get	Use this variable to determine in which type of update-method the OnStateStay()-event should be triggered
CallbackInvokationOrder	CallbackInvokationOrder	get	Determine the order to invoke the callbacks. Can be set in machine editor
Machine	GraphicalStateMachine	get	The machine where the state is in

Public Methods

Name	Type	Description
HasEdgeIngoing(GraphicalEdge)	bool	Checks if the given edge is an ingoing edge
HasEdgeOutgoing(GraphicalEdge)	bool	Checks if the given edge is an outgoing edge
OnStateStay()	bool	Manually invokes the OnStateStay() event. Returns true if invocation was successful
OnStateLeft()	bool	Manually invokes the OnStateLeft() event. Returns true if invocation was successful
OnStateEntered()	bool	Manually invokes the OnStateEntered() event. Returns true if invocation was successful
OnStateSetActive()	bool	Manually invokes the OnStateSetActive() event. Returns true if invocation was successful

GetNeighbours(bool)	List<GraphicalState>	Returns a list of states which are connected by an edge to this state. You can determine if you want neighbours which are connected by edges in both directions.
GetNeighbours()	List<GraphicalState>	Returns a list of states which are connected by an edge to this state. Will contain only states which are reachable by sending a trigger
InvokeRuntimeCallbacks (RuntimeCallbackType)	bool	Invokes all registered runtime callbacks for the given callback type. You can only use types OnStateEntered, OnStateLeft, OnStateStay and OnStateSetActive here.
RegisterRuntimeCallback (RuntimeCallback)	void	Registers a new runtime callback. You can only use types OnStateEntered, OnStateLeft, OnStateStay and OnStateSetActive here.
RegisterRuntimeCallback (RuntimeCallbackType, RuntimeCallbackDelegate)	void	Registers a new runtime callback. You can only use types OnStateEntered, OnStateLeft, OnStateStay and OnStateSetActive here.
GetRuntimeCallbacks (RuntimeCallbackType)	List<RuntimeCallback>	Returns a list of all RuntimeCallbacks which are invoked at the given event type
GetRuntimeCallbacks()	List<RuntimeCallback>	Returns all RuntimeCallbacks of this state
ClearRuntimeCallbacks()	void	Removes all registered callbacks

5.4 GraphicalEdge

Properties

Name	Type	Accessors	Description
OriginID	int	get	ID of the origin state
TargetID	int	get	ID of the target state
Trigger	string	get	Trigger of the edge
Origin	GraphicalState	get	Origin state
Target	GraphicalState	get	Target state
Machine	GraphicalStateMachine	get	Machine where this edge is in

Public Methods

Name	Type	Description
OnEdgePassed()	bool	Manually invokes the OnEdgePassed() event. Returns true if invoking was successful
RegisterRuntimeCallback (RuntimeCallback)	void	Registers a new callback to event OnEdgePassed
RegisterRuntimeCallback (RuntimeEventDelegate)	void	Registers a new callback to event OnEdgePassed
GetRuntimeCallbacks()	List<RuntimeCallback>	Get all registered callbacks
InvokeRuntimeCallbacks()	bool	Invokes all registered runtime callbacks. Returns true if every invoke was successful
ClearRuntimeCallbacks()	void	Removes all registered callbacks

Help and bug report

If you need help, please read the manual attentively first. If it does not help you, feel free to contact briskled@gmx.de for individual help.

We recommend you to also send positive and negative experiences with the Graphical State Machine Asset to briskled@gmx.de.

If you encounter bugs or wrong documentation, please also send a detailed description to briskled@gmx.de.