

Rapid State Machine Guide

- Lightning fast
- Scalable
- Easy

R S M

Rapid State Machine


Overview	4
Where to start	4
Quick Start	4
In-Depth Guide	6
How does a State Machine work?	6
State Machine gameobject structure	6
State Machine Component	7
State Summary	8
State Behaviour	10
State Behaviour Methods	10
States	11
Transitions	12
Conditions/Triggers	15
Adding additional objects and components	18
What is the TransitionCondition return type?	19
State Behaviour Trackers	19

Writing Custom State Components	19
Debugging	21
Visual Studio Integration and Other IDE's	22
VSManager.Trace()	23
Platformer Example Breakdown	23
Overview	23
ExampleCharacter Script	23
Jump	24
Fall	25
Run	25
Die	25
Idle	26
Advanced Platformer Mechanics	26
StateAnimation	26
StateEvents	27
ObjectSpawner	27
Advanced Uses	27
Inheritance within state behaviour scripts	27
State Machines with alternate state behaviour scripts	27
Optimising Conditions	28
Writing your own default conditions	28
FAQ	28
Best Practices	29
Project Contents	29
File Structure	29
Core State Machine Scripts	30
Core Editor Scripts	30
Secondary Scripts	31
Example Platformer	31
Guides	32

Overview

Rapid State Machine is a tool that allows you to create your own state machines. With powerful custom editors and integration with Visual Studio, it allows for extremely fast development.

Where to start

- [Platformer Example Breakdown](#) - This is the best place to begin learning how to use the rapid state machine. It demonstrates a working state machine with detailed explanations.
- [Quick Start](#) - This section of the guide provides a step-by-step on how to do everything but doesn't cover why or go into an extra depth.
- [In-Depth](#) - Gives detailed explanations on exactly how best to use RSM, complete with diagrams. This is the best place to master RSM.
-  [State Machine Diagrams](#) - Is a slideshow of all the diagrams from the In-Depth section. It can be used to learn the basics of the custom inspectors.

Quick Start

Setup

1. Ensure you are using Unity version x or later REPLACE X
2. Import the package from the unity asset store

Creating a new State Machine

In the hierarchy:

1. Right click
2. Select state machine
3. Select the state behaviour gameobject
4. Add a new script with your characters name as the class name i.e. Player, Slime, etc
5. Open the script. Use the RSM namespace and make the script inherit from MonoBehaviour and implement IStateBehaviour

Creating a new State

With a state machine selected:

1. Add States by pressing the "Add State" button in the state machine component, rename the states as needed
2. Double click one of states to open it

Creating a new To Transition

With a generic state selected:

1. Add transitions by pressing the "Add To Transition" in the Generic State component
2. Select a state to transition to by pressing the new dropdown labelled "Missing"

Creating a new Condition

With a state selected that has a transition:

1. Add a condition by pressing the "Add Condition" button within the desired transition
2. Select a condition to check by pressing the new dropdown labelled "Missing"

Adding a new Any Transition

With a state selected:

1. Add an any transition by pressing the "Add Any Transition" button at the bottom.
2. Select a condition as a requirement for the any transition
3. (Optional) Press the "Add Exclude" button to disable any transitions from a specific state

Defining state behaviours

In your state behaviour script:

1. Copy a state behaviour method example from the cheat sheet and rename accordingly.
2. So long as the name matches the state in your state machine, it will execute the code.
This is case sensitive.

Creating custom conditions

In your state behaviour script:

1. Copy the condition method example from the cheat sheet and rename accordingly
2. After compiling, your condition will now appear as an option in the dropdown for conditions

Creating custom Triggers

In your state behaviour script:

1. Copy the trigger method example from the cheat sheet and rename accordingly
2. After compiling, your trigger will now appear as an option in the dropdown for conditions

Changing the priority of ToTransitions

With a state selected that has at least two transitions:

1. Right click a transition, select a “Move” option to reorder the transitions. Transitions at the top will take priority.

Changing the priority of AnyTransitions

With a state machine selected that has at least two states:

1. Right click a state summary, select a “Move” option to reorder the states. States at the top will take priority when passing any transitions.

In-Depth Guide

How does a State Machine work?

A state machine is a system which contains a set of states. The system can only ever be in one state at a time. Each state contains rules which determine which other states it can transition to. Additionally, each state can have behaviour attached to it which executes when the state is entered, exited or on every tick.

State Machine gameobject structure

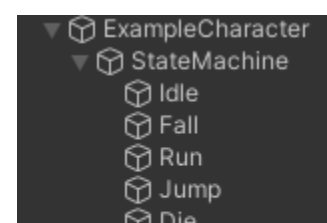
RSM requires a specific structure for its gameobjects. The parent gameobject should have a state behaviour component(which is a script that implements IStateBehaviour). The StateMachine component should be on another gameobject which is a child of the state behaviour gameobject. State gameobjects are created using the custom inspector and are children of the StateMachine gameobject.

All of this will be set up for you when you create a new state machine by right-clicking in the hierarchy and selecting state machine. Whilst this is done automatically, it's important to make sure you never change the structure of these gameobjects.



In the example, we can see the ExampleCharacter gameobject, the StateMachine gameobject and 5 state gameobjects. The state gameobjects are children of the StateMachine gameobject and the StateMachine gameobject is a child of the ExampleCharacter gameObject. ExampleCharacter has a component on it which implements IStateBehaviour, StateMachine has a StateMachine component, and the state

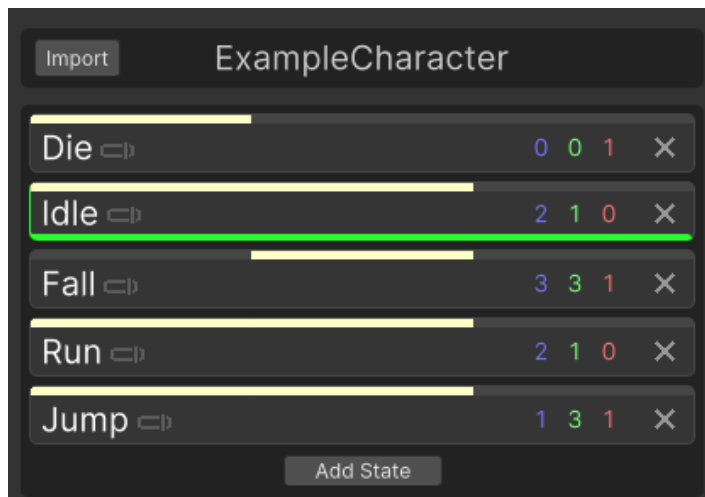
gameobjects all have a GenericState component.



Finally, it's likely you'll need to add other gameobjects to your state machine. These should be added as children of the state behaviour gameObject. In the example, you can see there is a Sprite gameobject and HealthControlelr gameobject which are placed as children of the ExampleCharacter gameobject.

State Machine Component

The StateMachine component is the core of this package. It holds data for each state as well as keeping track of the current state and any transitions that occur. It also has a custom inspector which provides an overview of the states within as well as several tools for manipulating them.



Here is the State Machine inspector from the example character. Firstly, it shows the name of the connected state behaviour script. Next, below the name, it displays some more in-depth information about each state that the state machine holds.

State Machine Custom Inspector

The "Add State" button will create a new state within this state machine. This will create a new child gameobject underneath the StateMachine gameobject since the states are stored as separate gameobjects.



The name of the attached State Behaviour

This section displays a list of all the states in this state machine. These are represented using an element called State Summary

State Summary

State summaries are an element that the state machine inspector uses to represent states. This element contains a lot of different features that let you quickly see exactly what a state is doing.

State Summary Element

This is a state summary that is representing the Die state. Double clicking this will select the state.

This label displays the name of the state that is being represented.

This button allows you to rename a state. The name of a state is what ties it to its state behaviours so it's important to ensure the name of your states match you state behaviour methods.

This green bar outline the Idle state summary is marking the Idle state as the current state. This will update in realtime and when in editor, it shows the first state that will be entered on play.

These three numbers display the number of different transitions in and out of this state. The blue represents transitions to this state, green represents transitions away from this state, and red represents any transitions.

This "x" button will open a menu that lets you delete the selected state. You can choose whether to delete just the state or the state and all transitions to the state. The former will leave transitions in other states which will now go to "Missing". It's important to note that deleting the state and its transitions cannot be undone with ctrl+z.

Import ExampleCharacter

Die	0	0	1	X
Idle	2	1	0	X
Fall	3	3	1	X
Run	2	1	0	X
Jump	1	3	1	X

Add State

This is a state behaviour tracker that is showing the behaviours for the Jump state. This is a very powerful part of RSM, see more in the State behaviour Tracker section.

Note: this inspector can be found by selecting any gameobject with a StateMachine component

There are also several less obvious features for the state summaries:

- Double clicking any state summary will select that state gameobject and show its generic state custom inspector. This means you can double click the Run state summary and immediately see the generic state inspector for the Run state.
- Right clicking any state summary will allow you to Set a state as the current state. This can be used in editor mode to set the default state or in runtime. It should be noted that this is meant to be a developer tool and so it doesn't respect transitions, exceptions or run any of the enter and exit behaviour that comes with states, it simply sets this state and the current state ignoring all transitions.
- Right clicking any state summary will allow you to select options that change the order that the state summaries are displayed in. This has one practical application which is the priority of any transitions. State summaries that appear higher in the list have a higher priority when querying their any transitions. For example, in the image above, the die state appears higher than the fall state so even if both of their any transitions are returning true, die will take priority and the state machine will move to the Die state.

Aside from any transitions, the order is essentially cosmetic so any states that don't use any transitions can be ordered however you like.

State Behaviour

IStateBehaviour is an marker interface with no method which you can implement. It is used by the state machine to help identify which script it should be using for its behaviours. Any script implementing IStateBehaviour can be used by StateMachine so long as it's placed on the parent gameobject.

A script that implements IStateBehaviour is where you define all the custom behaviour of your state machine. This includes what each state does, what conditions it uses and what triggers it uses. This custom behaviour is defined using state behaviour methods and conditions/triggers are defined using condition behaviour methods. See more about these in the state behaviour method and condition method sections.

Additionally, this script can be used to set/call certain conditions/triggers. If you're used to creating a character class that holds all the behaviour of a character, this can be considered the replacement.

Creating a new state behaviour is as simple as creating a new script with whatever name you like. Opening the script and adding ", IStateBehaviour" after "MonoBehaviour". You'll also need to add using RSM; at the top of your script for IStateBehaviour to not cause an error.

```
using RSM;  
Unity Script | 0 references  
public class YourClass : MonoBehaviour, IStateBehaviour
```

It's recommended to write state behaviour and condition methods at the bottom of a state behaviour script. It's also recommended to use regions to separate them from the rest of the state behaviour code. The advantage of this is that all of the code that can't be accessed quickly is at the top of the script. The state behaviour and condition methods can be opened using the custom inspector so it doesn't need to be near the top of the state behaviour script.

State Behaviour Methods

State behaviour methods are methods within your state behaviour script that are used by you states. There are three kinds; enter, tick, and exit.

An enter state behaviour method is called when its state is entered. An exit, when its

```
0 references  
public void EnterAttack()  
{  
    VSManger.Trace();  
    //Executes once every time the attack state is entered  
}  
0 references  
public void Attack()  
{  
    VSManger.Trace();  
    //Executes every tick whilst in the attack state  
}  
0 references  
public void ExitAttack()  
{  
    VSManger.Trace();  
    //Executes once every time the attack state is exited  
}
```

state is exited and tick is called every tick when the state is the current state.

The name of a state behaviour methods is what ties it to a state. The Idle state will try to call EnterIdle() when it's entered, Idle() every tick while it's the current state, and ExitIdle() when it's exited. You can check to see if a state is correctly tied to a state behaviour method using a state behaviour tracker.

When creating new state behaviour methods, create them exactly as you would any normal method that returns void. The name is what makes it special. The first line in a SBM should always be VSMManager.Trace() if you want state behaviour trackers to work correctly.

States

States are used to define each possible mode a state machine can be in at any one time. They're stored as GenericState components on gameobjects. Each state can be connected to up to three state behaviour methods, can contain any amount of to transitions, and one any transition.

Generic State Custom Inspector

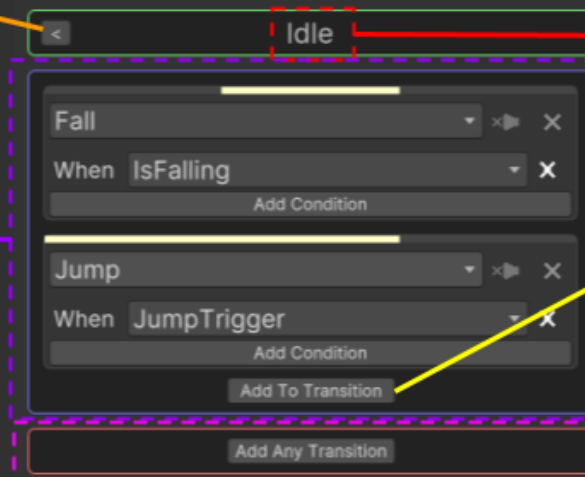
The "<" button will return to the parent state machine of this state.

This section displays a list of the to transitions in this state. These are represented using an element called Transition View.

Displays the name of the currently selected state.

The "Add To Transition" button will create a new transition inside this state.

This section is for the any transition of this state. Pressing the "Add Any Transition" button will display more options for the any transition.



All states should have names, the name is very important for adding behaviour. It's recommended to use verbs when naming states as existing words like Idle, run and jump tend to have the same amount of scope as we'd like for our states (for example moveLeft is too specific). The name of a state is what ties it to its state behaviour methods so it's important to keep them consistent.

States are responsible for their own to transitions which move away from themselves. For example, an Idle state would be responsible for a transition from Idle to Run. This consistency allows for easier debugging when searching for a specific transition. So long as you know what the previous state was, you know where to find the transition. The one exception to this is any transitions. Each state can have one any transition which moves from any other state to this state.

Since each state is stored on its own gameobject, it's easy to add additional components to that object which are only active when that state is active. These are called state components.

Transitions

Transitions are what determine how states move between one another. Each state can have as many transitions as you'd like. These transitions can all be edited via the GenericState custom inspector.

Each transition is made up of an origin state, a target state as well as any number of conditions. The conditions determine when a transition should occur; only when all conditions are true will a transition pass.

Finally, there are two types of transition, a To transition and an Any Transition. The biggest difference between the two is how the origin and target states are chosen.

To Transitions

To transitions are displayed in the To transition section of the GenericState inspector. Each To transition can have its target state chosen individually. Conversely, the origin state is defined by the state that owns the transition. If the transition is defined in the Jump state, a to transition will move from the jump state to another state. To transition can also be muted which means the transition will never occur. This can be helpful for debugging.

To Transition Element

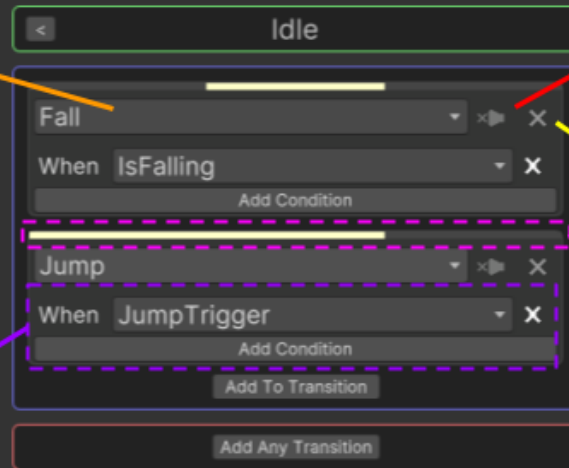
This dropdown displays the name of the state that this transition will move to. Clicking it will allow you to select any of your other states to move to instead.

This section displays a list of the conditions within this transition. All conditions need to pass for the transition to occur. The conditions are represented using a Condition View element.

This mute button is a toggle that mutes this transition. This means the transition will not be checked or performed. This can be useful whilst debugging.

The "X" button will delete this transition.

This is a state behaviour tracker that is showing the behaviours for the state that this transition moves to (in this case Jump). This is a very powerful part of RSM, see more in the State behaviour Tracker section.



Note: this inspector can be found by selecting any gameobject with a GenericState component

Any Transitions

Any transitions have a few more limitations than To Transition. Each state can only have one any transition and the state that defines an any transition will always be the target state of that transition. The origin state for an any transition is undefined meaning it can occur from any state to this state. The results in a behaviour similar to if we had a to transition in every state moving to this state. The last feature of any transitions is "Excludes". You can choose to exclude any number of states meaning they won't be counted as possible origin states. This results in a transition from any state (except the ones you exclude) to this state.

Any Condition Element

The state that the any transition will move to.

The interface shows a transition configuration for 'Jump'. At the top, a green box highlights the 'Jump' label and a back arrow. Below, a blue box highlights the 'When' conditions section, which includes two rows: 'When IsFalling' and 'When Delay 0.2', each with an 'Add Condition' button. At the bottom, a red box highlights the 'When JumpTrigger' condition, which has an 'Add Condition' button. To the right of this, a dashed pink box highlights the 'excluded states' section, which lists 'Jump', 'Fall', and 'Die' with 'x' icons and an 'Add Exclude' button. A pink arrow points from the text box on the right to the 'excluded states' section.

This section displays the Conditions that must be met for the any transition to occur.

This section displays the "excluded states". This is a list of your other states that the any transition won't occur from. In this case, Jump can transition from any states except Jump, Fall, and Die when JumpTrigger is true.

To Transitions vs Any Transitions

The two types of transitions can often achieve the same results so it can be difficult to understand when to use one type over the other.

In general, to transitions are suited for more specific situations where a transition need specific requirements whereas an any transition is better for broad transitions that have the same requirements regardless of the origin state.

Since each state can only have one any transition, adding an if/else type of behaviour isn't possible without to transitions: If you want behaviour that move to one state under one condition but a different state under another then a to transition is required.

If you have a state that always has the same entry requirement (for example a fall state that occurs when the character isn't on the ground), then an any transition is the better choice. This allows us to create one transition that handles the interaction. Additionally, when adding more

states in the future, it's not required to add transitions to Fall in these states since any transition automatically assume that it can move from any states except the excluded ones.

A combination of the two types will achieve the best results.

Conditions/Triggers

Condition Element

This dropdown displays the name this condition. Clicking it will allow you to select any of your other conditions or a default condition.

The "X" button will delete this condition.

The "When" text can be clicked to invert a condition. When "When" is displayed, the condition to the right must return true. When "NOT" is displayed, it must return false to pass.

"Delay" is a default condition which has a parameter. This input field allows you to input the length of the delay. Hovering these input fields will display a tooltip explaining them.

Conditions

Conditions are what determine when a transition can occur. A transition can have any number of conditions and all of them must return true before the transition will be allowed to pass.

Each condition is visible and editable from within the GenericState inspector. They're owned by transitions so each condition will be displayed on the transition that it's used by.

When displayed in the inspector, conditions can have a couple of different editable properties. All conditions can be inverted meaning when the state machine checks if a transition can occur, all of the transitions conditions must be true except for the inverted transitions which must be false instead.

Additionally, some conditions have extra input fields that allow adjustments for that condition. (see more: [Delay Condition](#), [Cooldown Condition](#))

Finally, there needs to be a way to set the rules for a condition; this is done in the state behaviour script. (see more [Condition Method](#), [trigger method](#))

Conditions vs Triggers

Triggers are another type of condition. The two are very similar and are indistinguishable within the GenericState inspector however, there are a few key differences.

Both types of condition act like bools, when the state machine checks them, they're either true or false. Normal conditions achieve this by using a rule, for example health > 0 would be a fitting rule for an IsAlive condition. Any time we check the IsAlive condition, the health is compared to zero. Conversely, triggers work more like a light switch. You define rules for a trigger, instead you set whether the trigger is true or false within other parts of your script like Update() using triggers methods: Trigger() and CancelTrigger(). Additionally, any time a transition that uses a trigger passes, all triggers are set to false.

Due to these differences, regular conditions are better used for continuous checks like IsAlive or IsGrounded whereas Triggers are superior for snap events like collisions or input.

The last major difference between the two is how they're defined in the state behaviour script.

Condition Method

A condition method is what defines the rule for a condition. It's written like this:

```
0 references
public TransitionCondition YourCondition()
{
    VSManager.Trace();
    return new TransitionCondition(yourBool);
}
```

(a copyable version is available inside the [Cheat Sheet](#)).

The condition method should always be a simple method. The TransitionCondition return type is what the GenericState inspector uses to find your condition methods for use in the condition dropdown.

VSManager.Trace(); is what allows the state behaviour tracker to work and should always be the first line of a condition.

Lastly, the `return new TransitionCondition(yourBool);` line is where you define your rules. You can replace `yourBool` with anything that returns a bool value. This could be something complicated like a raycast, more simple like just passing in an existing bool or even just `"true"`.

Trigger Method

A trigger method is where we define a variable for our new trigger and a method for the GenericState inspector to access our trigger through. It's written like this:

```
private TransitionCondition conditionTrigger;
0 references
public TransitionCondition Trigger()
{
    VSManager.Trace();
    if (conditionTrigger == null) conditionTrigger = new TransitionCondition(false, true);
    return conditionTrigger;
}
```

(a copyable version is available inside the [Cheat Sheet](#)).

First we define a variable of type `TransitionCondition` with the name of our trigger. Then write a method which returns `TransitionCondition` and has a similar name. The name of the method is what will display in the GenericState inspector.

The trigger method will check to see if the variable is null and if it isn't, it'll create a new `TransitionCondition` with trigger set to true.

Finally, the trigger method just returns the `conditionTrigger` variable.

Now within this state behaviour script, writing `conditionTrigger.Trigger()` will set the trigger to true. `conditionTrigger.CancelTrigger()` will set the trigger to false. If you want to set a trigger from outside of the state behaviour script, you'll need to use a [Condition Trigger](#) component

Delay Condition

The delay condition is a default conditions meaning it already has its rules defined. This condition checks how long the state machine has been in the current state and compares it to a value. If the time spent in the state is \geq the value, it returns true.

The value to compare against can be set in the extra input field that appear next to any delay conditions in the GenericState inspector. This allows for the quick and easy creation of several types of behaviour such as transitional states (an intermediary state that moves to another after a delay), charging states for weapons or spells, or even more complex mechanics like coyote-time.

Inverting the delay condition also provides some interesting utility. It will now only return true when the time spent in the state is $<$ the value. This could be used to create mechanics like

combo attacks where you only want to allow moving to the next attack if it's within a certain time of the previous attack.

There's a second delay condition variant called "delay between". It has two input fields which allow you to enter maximum and minimum delay values. Each time the delay passes, the delay is randomised between the two. This can be used to provide some variance within transitions and can often make character seem more natural.

Cooldown Condition

Like the delay condition, the cooldown condition is a default condition. It also has an extra input field but unlike the delay condition, the cooldown condition compares its value against the last time the parent transition passed. It will only return true if the last pass time is \geq the value.

`v IS THIS TRUE? V REPLACE`

This can be used to quickly add cooldowns for transitions. It should be noted that these won't apply to a state as a whole so if there's more than one transition into a Shoot state, each transition would have it's own cooldown. For this sort of situation, it'd be better to define your own cooldown as a regular condition.

Inverting a cooldown condition will make it only return true when that transition has been used within the last x seconds.

Finally, there's a second variant of cooldown condition called "Cooldown Between". It has two input fields which allow you to enter maximum and minimum delay values. Each time the cooldown condition passes, the value is randomised between the two input values.

Adding additional objects and components

When using state machines, you'll often need to add additional elements. For example, a character will need sprites or models as well as colliders and many more. In unity, these will be created using components which could easily be added to the state behaviour gameobject but this is **not** recommended.

It's best to keep state machine related elements separate from non-state machine elements as much as possible. As such, the only reasonable place to add additional elements to a state machine is as a child of the state behaviour gameobject.

There are some exceptions to this

State components which perform additional behaviour when in a state should be placed on the relevant state gameobject. By placing state components this way, you can also piggy-back off of the hierarchy to easily get other components in the hierarchy (e.g. an animation controller that automates some of the drag/drop of using the StateAnimation).

Any components that handle movement should be placed on the state behaviour gameobject itself as that is the parent transform of the collective gameobject.

What is the TransitionCondition return type?

TransitionCondition is a class that mimics a bool but adds a tiny bit of extra behaviour to allow for triggers. This works using an implicit operator which allows all TransitionConditions to return a bool. If the TransitionCondition is for a condition, it'll just return the `isTrue` bool. If it's for a trigger, it'll return the `IsTrue` bool and if `isTrue` was true, then `isTrue` is set to false.

The other important thing to note about TransitionConditions is that they're what the state machine uses to tell with methods within your state behaviour script are condition/triggers. As such, all condition and trigger must use TransitionCondition as a return type.

State Behaviour Trackers

State behaviour trackers are a key part of RSM and what makes RSM so fast to use. It allows you to quickly access any state behaviour methods or condition methods inside a state behaviour script regardless of its size or complexity.

The screenshot shows the 'State Behaviour Tracker' inspector for a component named 'ExampleCharacter'. The tracker is divided into three segments: 'enter' (left), 'tick' (middle), and 'exit' (right). The 'Idle' state is highlighted in yellow, indicating it is the current state. The 'Fall' state is highlighted in green, indicating it is the next state. The 'Run' and 'Jump' states are highlighted in blue, indicating they are future states. The 'Die' state is highlighted in red, indicating it is a terminal state. The 'Add State' button is at the bottom.

State Behaviour Tracker

Import ExampleCharacter

State	Enter	Tick	Exit
Die	0	0	1
Idle	2	1	0
Fall	3	3	1
Run	2	1	0
Jump	1	3	1

Add State

Annotations:

- Red box:** This is a State Behaviour Tracker. It's divided into three segments; the left is for enter, the middle is for tick, and the right is for exit.
- Orange box:** When a StateBehaviour contains state behaviour methods for a state, they'll glow this yellow colour. This means the ExampleCharacter has state behaviour methods for EnterIdle() and Idle() but not ExitIdle().
- Blue box:** Clicking a segment of the state behaviour tracker that is glowing yellow will open that state behaviour method. In this case, when clicked, Fall() will be opened in your IDE. This works using the `VSManager.Trace()` method so all state behaviour methods must have this as their first line for use with the state behaviour tracker.

Note: this inspector can be found by selecting any gameobject with a StateMachine component

State behaviour trackers can be used to tell at a glance if a state is correctly tied to its state behaviour methods. Additionally, when yellow, these bars can be clicked to take you to the line of code where the state behaviour method is defined (see more: [VSManager.Trace\(\)](#)).

Writing Custom State Components

State components are extremely powerful as they allow you to easily add behaviour that execute when entering or exiting a state. This allows a much more scalable coding process which can significantly reduce repetition. As such, it's important to be able to write your own state components to better fit your needs.

When writing a state component, the easiest way to add behaviour to entering or exiting a state is to use the existing events inside the state machine. State components should always be placed on state gameobjects which should always be children of a state machine. This means we can easily get the StateMachine component using `transform.parent.GetComponent<StateMachine>();`

There are two ways to add behaviour to entering and exiting states when writing a state component. The first is simpler and involves subscribing to the `enterEvent` or `exitEvent` inside the State class. Since state components should be placed on state gameobjects alongside `GenericStates`, it's easy to get the state and subscribe to either of these `UnityEvents`. In the example below, `MyMethod()` will be executed whenever the state that this component is on is entered.

```
State state;
Unity Message | 0 references
public void OnEnable()
{
    state = GetComponent<State>();
    state.enterEvent.AddListener(MyMethod);
}

Unity Message | 0 references
public void OnDisable()
{
    state.enterEvent.RemoveListener(MyMethod);
}
```

The other method is slightly more complex but allows for more intricate behaviour. There are two events inside the StateMachine. `OnStateChange` which occurs before `currentState` is set to the new state and `AfterStateChange` which occurs after `currentState` is updated. These events are `StateChangeEvents` which means they contain information about the two states that the state machine is transitioning between. The `StateAnimation` component uses the `OnStateChange` event to play an animation when entering a state. Knowing what the previous state was as well allows the `StateAnimation` component to optionally play a different animation when coming from certain states.

In the example below, `MyMethod` is called whenever the state machine changes states. Therefore, `MyMethod()` immediately returns if the current state isn't the same as the state that

the state component is on. The other important difference here is that MyMethod has to have two parameters of type State.

```
State localState;
StateMachine stateMachine;
☒ Unity Message | 0 references
public void OnEnable()
{
    localState = GetComponent<State>();
    stateMachine = transform.parent.GetComponent<StateMachine>();
    stateMachine.OnStateChange += MyMethod;
}

☒ Unity Message | 0 references
public void OnDisable()
{
    stateMachine.OnStateChange -= MyMethod;
}

2 references
public void MyMethod(State from, State to)
{
    if (to != localState) return;
}
```

Debugging

It's important to be able to easily tell what the state machine is doing. There are four features within RSM that will allow you to understand exactly why the state machine is doing whatever it is doing.

Current State Indicator

This is the simplest of the debugging tools. In the StateMachine inspector, the current state is highlighted with a green outline. This updates in realtime so it's excellent for giving you a quick overview of what's happening in your state machine. The downside here is when transitions are happening too quickly for the human eye, this indicator doesn't provide enough information for you to tell what's happening. Fortunately, the StateMachineDebugger does.

State Machine Debugger Component

This component is placed on the state machine gameobject. It attaches itself to the state machine and the StateMachine OnStateChange event in order to track all the transitions that occur within the state machine.

The component has a custom inspector and when the “Track” button is clicked, it will display the last 10 transitions that have occurred since you began tracking. The custom inspector shows three pieces of information for each transition; what state the transition came from, what state it's going to, as well as how long the state machine was in the from state. This is extremely useful when dealing with rapid sequences of transitions. Normally, they happen faster than you can see however, as they're all listed, you can follow the path that the state machine took through states.

Depending on how you've used the state machine, you might be able to tell exactly which transition was used for each changing of state however, if any of your states contain duplicate transitions (e.g. multiple transitions from Run to Jump) with different conditions then it becomes trickier. In this scenario, if something isn't behaving as expected, it's recommended to use transition muting to narrow down the cause of the problem.

Muting Transitions

When a state contains duplicate transitions with different conditions (e.g. multiple transitions from Run to Jump), it can be difficult to tell which transition is being used. If something isn't behaving as expected then understanding which transition is causing problems is vital. Within the GenericState inspector, each transition has a mute button which can be toggled. By muting all but one of your duplicate transitions relating to the problem, you can check each individually to ensure they're performing as expected.

Since muting is a toggle, ensure that you unmute any transitions when you're done debugging.

State Behaviour Tracker

This element is visible in the StateMachine inspector and GenericState inspector. It can be useful to quickly tell if a state is connected to its state behaviour. (see more: [State Behaviour Tracker](#))

Visual Studio Integration and Other IDE's

Being able to quickly open state behaviour methods and condition methods is core to what makes RSM so fast to use. As such, it's important to ensure that it works with your IDE of choice.

Tested IDE's

In our use, these IDE's work with RSM:

- Visual Studio 2017
- Visual Studio 2022
- Visual Studio Code

These IDE's have been tested and won't work with the default implementation of `VSManager.Trace()` but may work with something custom:

- Sublime Text
- Atom

My IDE hasn't been tested

If your IDE has not yet been tested, it's very easy to run a test yourself before purchasing RSM. Simply execute this code inside Unity and it will open a script, if it opens to the correct line then your IDE works by default. If it doesn't open to that line, see the section below.

My IDE doesn't work with the default implementation of `VSManager.Trace()`

If your IDE doesn't work straight away, that doesn't mean it's impossible to get it working. Line 39 in `VSManager` script is what should be opening your IDE to a specific line, if it doesn't work by default with your IDE then this is the line you'll need to change to something more custom. Googling "open "my IDE" to specific line"

`VSManager.Trace()`

`VSManager.Trace()` allows the custom editor to open Visual Studio and jump to the method that you're inspecting. This is core to what makes this state machine so fast as you can navigate rapidly between state behaviour methods and condition methods even within a very complex character with many methods. Due to this, it should be the first line in all state behaviour methods, condition methods and transition methods. However, it is optional. If you don't wish to use this part of the state machine, it can simply be removed from all these methods.

Platformer Example Breakdown

Overview

The Platformer Example is built around a character controller for a 2D platformer. Despite having very little code, the character has several more advanced platforming mechanics like coyote time. These are all implemented via the RSM custom inspectors.

This example will demonstrate how to use all of the primary functions of RSM as well as some good practices. Additionally, it uses the secondary state components such as the State Animation and State Events.

ExampleCharacter Script

The `ExampleCharacter` script inherits from `MonoBehaviour` and implements `ISateBehaviour`. This makes it the state behaviour script for the example character meaning it's where all of the custom state behaviour methods and conditions are defined.

- Contains all methods for basic 2d character controller (recommended to separate these for commercial use)
- Rudimentary ceiling+floor collision
- Update() contains input, triggers, stores horizontal input
- Velocity variable stores a direction, direction is added to characters current position in Apply Movement
- Gravity() reduces velocity.y when called
- AirControl() adjust velocity to match horizontal input with slow acceleration
- GroundControl() adjust velocity to match horizontal input with quick acceleration
- ApplyVelocity() adds velocity to the current position moving the character in that direction adjusted for deltaTime. Called in all tick state behaviour methods where the character will be moving.
- CeilingCollision() extremely basic raycast collision upwards, don't use this commercially.
- EnterIdle() and EnterRun() both reset remainingJumps since we know we're on the ground
- All grounded states will set velocity.y to 0 since the character shouldn't be falling if it's on the ground or rising if it's not jumping. This is done in both enter and tick state behaviours
- Idle() multiplying velocity by 0.98 which means it gets reduced to zero over time for smooth stop.
- Run() calls GroundControl() since we want to move based on input
- CeilingCollision() called in all states that the character could be moving upwards in
- AirControl() called in all airbourne states
- EnterDie() sets velocity to zero to stop the character from moving
- JumpTrigger() is a trigger, .Trigger() and .CancelTrigger() are called in Update() when jump is pressed
- DieTrigger() is a trigger, called from outside this script via a ConditionTrigger
- IsGrounded() is a condition, fires a ray at the floor below and returns true if touching. This is how the state machine knows whether the character is on the floor or not
- IsFalling() is a condition, returns true when velocity.y is < 0 which means the character is falling
- JumpHeld() is a condition, works in tandem with JumpTrigger(), returns true if the jump key is held. Used for variable jump height
- CanJump() condition, checks if the character has used all its jumps yet
- MoveHeld() checks if horizontal input is held or not, used to transition between Idle and Run

Jump

Behaviour

- Enter set velocity.y to jumpStrength

- Tick CeilingCollision(), AirControl(), ApplyVelocity()

When to enter

- When jump is pressed and we're on the ground (implemented in jump any transition)
- Or when the character's just began falling and hasn't used the jump yet (implemented in Fall to Jump transition)

Fall

Behaviour

- Tick CeilingCollision(), AirControl(), Gravity(), ApplyVelocity()

When to enter

- When character isn't on the ground but is current in a ground state (see Fall Any Transition)
- When player releases jump (see Jump to Fall transition 1)
- When character is falling (see Jump to Fall transition 2), this occurs when CeilingCollision() is called which immediately makes the character fall
- After 0.2 seconds (see Jump to Fall transition 3). This is the maximum duration for staying in jump, otherwise player could hold space and rise infinitely

Run

Behaviour

- Enter stop y movement, Reset remainingJumps
- Tick GroundControl(), stop y movement, ApplyVelocity()

When to enter

- When character is in Idle state but move is being held (see Idle to Run transition)
- When character is in Fall state but is now on the ground and move is being held (see Fall to Run transition)

Die

Behaviour

- Enter, stop movement

When to enter

- When HealthController collides with spikes, it calls Trigger() on the ConditionTrigger component which is placed on the Die state gameobject. (see Die any transition, ConditionTrigger component on Die gameObject, and HealthController component)

Idle

Behaviour

- Enter stop y movement, Reset remainingJumps
- Tick stop y movement, reduce horizontal velocity, ApplyVelocity()

When to enter

- When character is in Run state but move is no longer being held (see Run to Idle transition)
- When character is in Fall state but is now on the ground and move is not being held (see Fall to Run transition)

Advanced Platformer Mechanics

Coyote Time

- Allows the character to jump for a moment after falling off a platform, makes it feel better
- Defined using Fall to Jump transition that checks if character has already jumped and if character began falling within 0.2 and jump pressed
- Normally a lot more complex to implement this but required zero code with RSM

Variable Jump Height

- Allows character to jump higher the longer jump button is held
- Gravity not applied in jump state
- Leave jump state when jump button is released (see Jump to Fall transition 1) or we've stayed in jump for set time (see Jump to Fall transition 3)
- Only required the creation of an extra condition: JumpHeld()

Jump Buffer

- If player presses jump just before being on the ground, character will still jump
- Triggers are true when .Trigger() is called until their condition passes
- Only cancel when jump is released so as long as the player holds jump before landing, jump will fire
- Required no extra code

StateAnimation

- State component so only active whilst attached state is the current state
- Plays an animation
- Can play different animations based on previous state
- Powerful for making smooth feeling animation transitions
- Add transitions in Unity Animator from the transition animation to the default animation for that state for smooth results
- Subscribes to state machine OnStateChange event
- You can write your own components that subscribe to these events

StateEvents

- State component so only active whilst attached state is the current state
- Allows for use of UnityEvents for drag and drop functionality
- Write scripts that do anything and drag and drop into state events to make them execute when entering or exiting a state
- Subscribe to UnityEvents enterEvent and exitEvent inside attached state

ObjectSpawner

- Simple script, spawns an object
- In the example, spawns a particle when character dies
- Spawn is triggered by StateEvents component on Die state gameobject

Advanced Uses

Inheritance within state behaviour scripts

- Inheritance works with conditions and triggers
- And probably state behaviour methods
- You can have a slime class and child classes for different colours
- The different colour child slimes can override state behaviour methods or conditions to add variety to their behaviour

State Machines with alternate state behaviour scripts

- Only thing connecting state behaviour scripts to their state machines are names of conditions/triggers and name of state behaviour methods
- So long as two state behaviour methods have all the same conditions, you can simply swap them and one state machine will run either state behaviour
- Pseudo polymorphism

- Useful for characters that have similar overarching states but very different behaviour
- Example enemies that seek out the player, one walks, the other flies
- Inheritance isn't good here since the characters are too different but if the conditions are the same then we can just reuse the same state machine and modify values that affect conditions
- Example, have a CanSeePlayer() condition which has a range and change the range between the two state.

Optimising Conditions

- Conditions are checked every tick
- If there's an expensive condition, you should cache the result and place the result as the rule of the condition instead
- Cheapest conditions first since the moment one fails, the rest won't be checked

Writing your own default conditions

- Create new class that inherits from MonoBehaviour and implements IStateBehaviour. Write your default conditions in this class. Inherit from this in your state behaviour script instead of MonoBehaviour. CHECK THIS
- Hard to do one with inputs, you'll need to add them in the Editor scripts

FAQ

StateMachine says "Missing State Behaviour"

If the StateMachine inspector is displaying "Missing State Behaviour", it means that your state behaviour gameobject doesn't have a state behaviour script. You'll need to create a new script and make that script implement IStateBehaviour. To do this, open the new script and add "IStateBehaviour" after "MonoBehaviour".

This script should be added to the gameobject named statebehaviour. Now when you select the statemachine gameobject, it should display the name of your new state behaviour script instead.

Why not use nodes?

- Messy when state machine becomes complex
- Or Nested which hides a lot
- List style allows for very large state machines
- Each state is only responsible for transitions out of it and a single any transition in

- Means you know exactly where to look if something isn't working as intended

Best Practices

It's recommended to create regions in your state behaviour script to contain the state behaviour, condition, and trigger methods. These should be organised below other methods and variables in the script as you can navigate to them via the state machine inspector.

Don't try to make a state for each direction of movement, instead have one state for movement that applies movement based on inputs. You can store velocity in a variable outside of the Move state to allow continuous movement between different states.

Don't have nested condition methods or state behaviour methods. For example placing in the return for CanJump() you should never include IsGrounded since it will break behaviour tracking for those conditions. Either take the base boolean that IsGrounded uses and use that in CanJump or use IsGrounded alongside CanJump in the custom inspector.

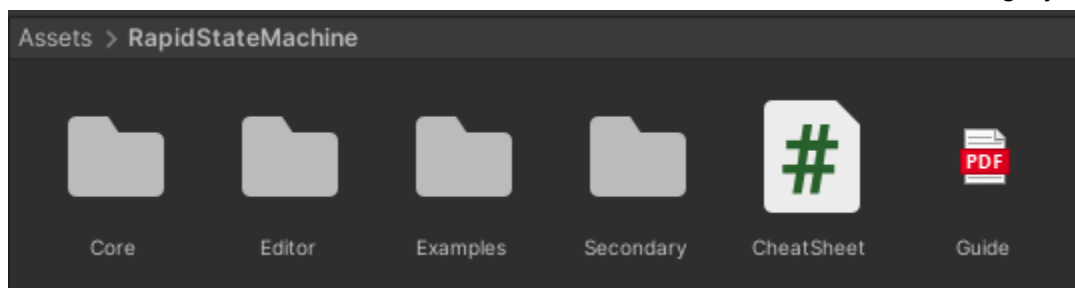
Don't use the condition methods in if statements. Elements of the state machine are determined based on whether conditions return true or false and so checking them outside the scope of the state machine can cause unexpected results. Especially with Triggers.

Separate states with different movement speeds and set the movement speed upon entering the State. For example, place the movement logic inside a MovePlayer() method and call that from Sprint and Walk. Then set movement speed to 1 in EnterWalk() and to 3 in EnterSprint().

Project Contents

File Structure

The structure of the files can be divided into four main folders with a fifth category for guides



Core

Contains all the scripts that make the state machine work. Removing any of these from the project will cause the state machine to no longer function.

Editor

This folder holds the scripts and visual tree assets for the state machine editors. Removing these will cause the state machine and state components to have their default unity inspectors. The default inspector offers significantly reduced functionality.

Examples

Contains the platformer example which has a working 2D platformer controller using the state machine.

Secondary

Contains several scripts and components which aren't mandatory for the state machine to function.

CheatSheet script

A script containing several commented code snippets that you'll have to use regularly when working with this state machine.

Core State Machine Scripts

IStateBehaviour

This is a marker interface with no methods. A script that implements IStateBehaviour can be interacted with by a StateMachine. Therefore, a script that implements IStateBehaviour is where all the custom logic for your state machine should be written. These scripts are called state behaviours.

StateMachine

This is a component and the backbone of the RSM. It's where all of the data for states is stored. The StateMachine component is also responsible for keeping track of the current state, calling the state behaviour methods and checking whether a state is ready to transition.

State

The primary job of the State class is to store data; information about transitions and any transitions.

GenericState

This is a component which inherits from State. Its role is to get and store the state behaviour methods from the state behaviour script that match the name of the gameobject this component is on.

Core Editor Scripts

StateMachineEditor

This editor class is responsible for drawing the state machine custom inspector. It does this using the StateMachineInspector.xml visual tree asset as well as drawing child visual tree assets like StateSummaryInspector.

GenericStateEditor

This editor class is responsible for drawing the generic state custom inspector. This is achieved using the StateInspector.xml visual tree asset and drawing several child elements including TransitionView, StateView, and ConditionView.

Secondary Scripts

State Events

This is a component that goes on state gameobjects. It uses UnityEvents to allow methods to be added to a states enter and exit using drag and drop.

Condition Trigger

A component with a custom inspector that allows you to select a and then call .Trigger() on this to trigger the condition. This allows triggers to be set easily outside of the state behaviour script where they're defined.

StateMachineDebugger

This is a component that goes on the state machine gameobject. It has a custom inspector and records the states that the state machine passes through allowing for easier debugging.

VSManager

Contains methods that open visual studio to a specific line when state behaviour trackers are pressed.

Example Platformer

ExampleCharacter Prefab

This is a ready to go prefab for a simple 2D platformer player character. It has most of the basic functionality like input, simple collision and gravity. Additionally, it contains some more complex mechanics like coyote time which are defined in the custom inspector. This prefab is designed as an example and is not recommended for use in a commercial project.

ExampleCharacter Script

This is the most important script within the example since it contains all the custom state behaviour, conditions and triggers that define how the 2D character behaves.

StateAnimation Script

This is a component that goes on state gameobjects. It subscribes to the events inside the StateMachine and uses those events to play animations outside the scope of the defined states.

HealthController Script

A simple script that calls Trigger in a ConditionTrigger when touching spikes.

ObjectSpawner Script

A simple script containing a single method that instantiates a gameobject.

Guides

Cheat Sheet

This is an example script which contains code snippets that are frequently used when using RSM. It can be found at Assets/RapidStateMachine.