INSTINCT AI

v 1.0

KUPIO

Instinct AI User Manual

For Version 1.0

https://kup.io

Documentation first release November 2017

Table of Contents

1. Introduction

Thank you for purchasing InstinctAI!

InstinctAI is a fully-featured, code-oriented behaviour tree editor for Unity. With it you can create behaviour tree functionality directly as `MonoBehaviour` scripts that can be attached to your game objects.

Integrated into your behaviour tree is a utility graph node that allows your AI to make better reasoned decisions about any given situation by evaluating all possible options.

InstinctAI uses code-generation to create performant, runtime-free behaviours. To get the most out of it, we recommending reading through this manual in order to become familiar with the workflow and the concepts involved.

We created the editor to be easy to use, but if you do need help then please refer to the feedback section at the end of this manual. Help is just an email away.

1.1. Core Concepts

Before getting started, we will introduce the basic concepts involved in creating behaviours using InstinctAI.

Behaviour trees

Behaviour trees are a means of visually describing complex AI behaviours in a way that is easy to see, understand and alter. Behaviours are composed of a tree of actions and conditions that is navigated at runtime in an intuitive manner.

Actions

When you want your game character to do something, you will place an action in your behaviour tree. InstinctAI ships with a library of common actions, but you will often find is best to create specific actions that suit your own requirements. InstinctAI makes this as easy as possible using nothing more than simple methods in `MonoBehaviour` classes,

Code generation

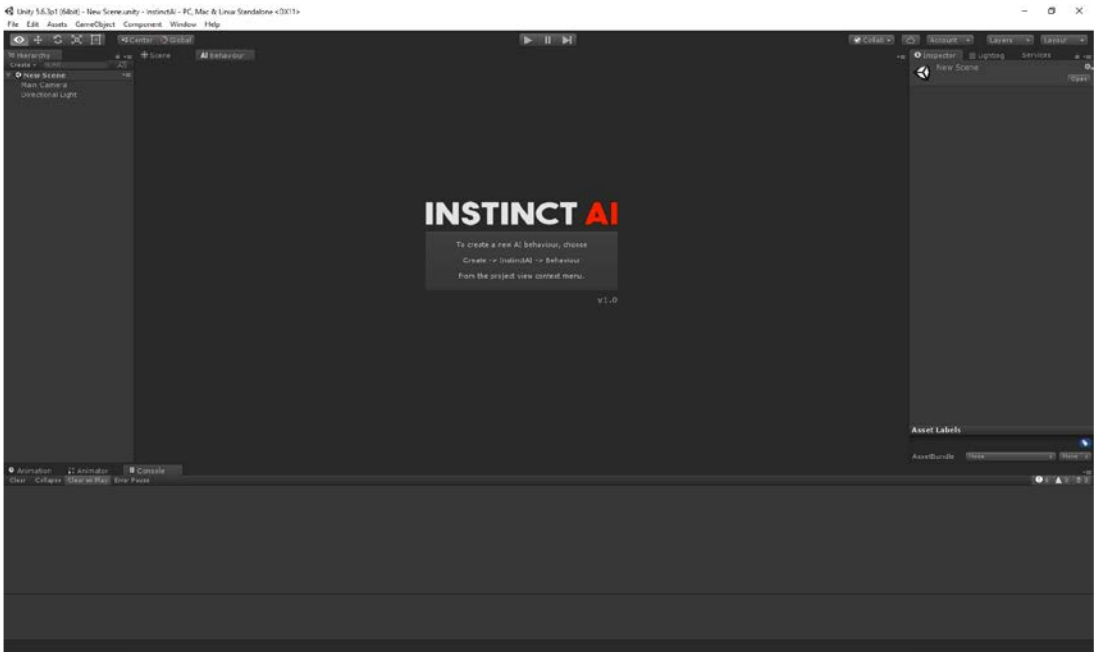
When you are editing a tree using the InstinctAI editor, you are visually editing a generated C# class. This works differently to other behaviour tree libraries, but by doing so, it allows you to generate compact, performant and runtime-free code that is nothing more than a regular `MonoBehaviour` that can be attached to any `GameObject` you want.

2. Setting up

The asset is very easy to set up and use. Simply download the asset from the asset store into your project and you're good to go. InstinctAI is an editor extension, so once it is added to your project, you will see new options in your editor for creating and editing Behaviour trees.

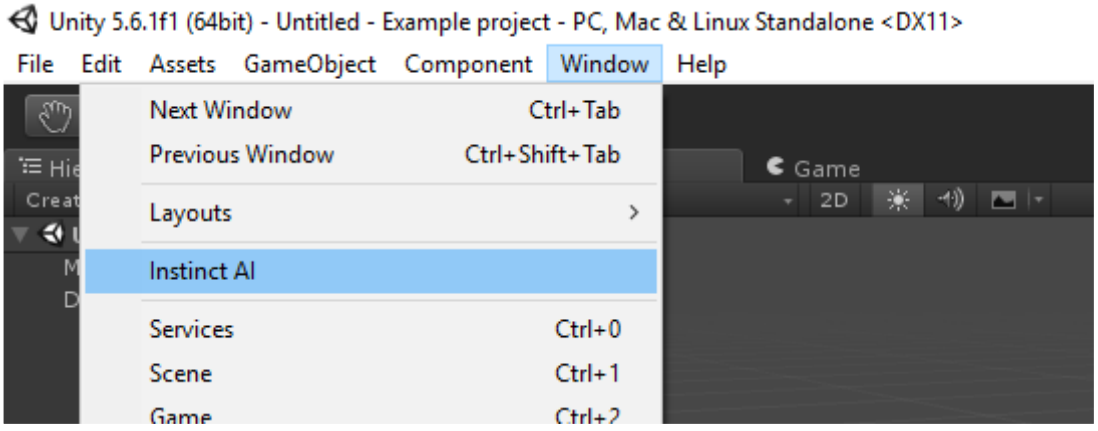
Also included in the package are a number of examples that demonstrate different parts of the product. For more information on the examples provided, please see the Examples section in this document.

3. Interface and Basics



If you need to open the InstinctAI editor window, you can open it from the Window menu.

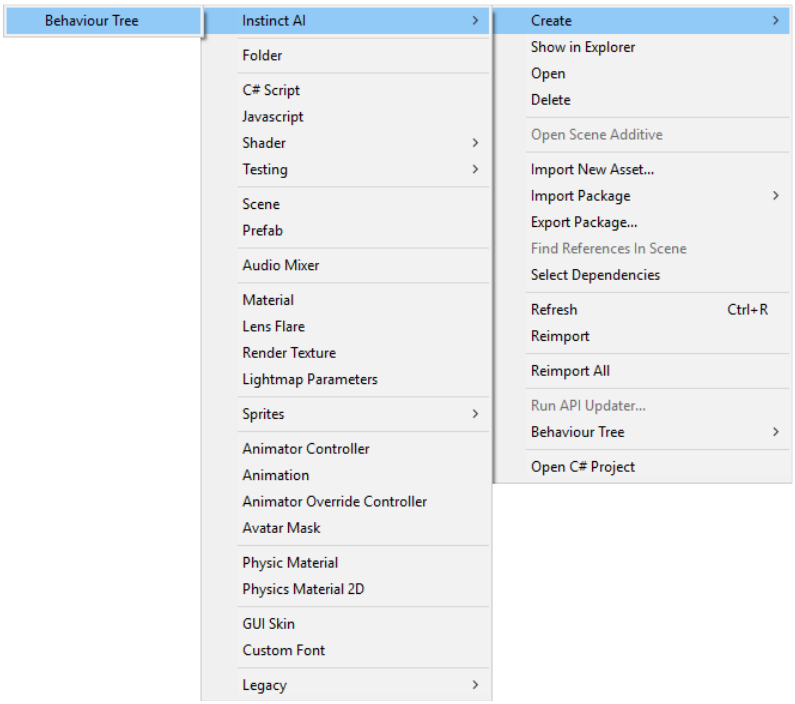
> *Note*
> InstinctAI supports both the light (Personal) and dark (Professional) skins in the Unity editor.



More commonly, you would simply double-click a behaviour tree in your project.

3.1. Creating new Behaviour Trees

Since the tree editor is conceptually a visual code editor, behaviour trees must be created in with the code of your project. You can do this from the context menu in your project view; normally within one of your own scripts folders.



Once you name your new tree, two additional C# script files will be created alongside it.

These files are one `MonoBehaviour` class, split across two script files. In C#, these are known as partial classes, which are a method of creating one class using multiple files. In this case the generated portion of the class is in one file, and the other file can be edited to add your own functionality. If you open the non-generated file, you will find an empty class waiting for you to add your own code.

```
//%GenSrc:1:Ln27fDSJLkCUXveI4Puutg
//%CodeKey:puAME+MUh0eLxKBiGHiytQ
/*
 * This code was generated by InstinctAI
 *
 *         DO NOT MODIFY BY HAND
 *     THIS FILE WILL BE REGENERATED
 *
 *      DO NOT REMOVE/EDIT COMMENTS
 *
 * If you do need to modify this file, note that it is a partial class.
 * See https://msdn.microsoft.com/en-us/library/wa80x488.aspx
 * You can edit the file Animation.cs
 */

namespace instinctai.usr.behaviours
{
    using UnityEngine;

    public partial class MyNewBehaviour : MonoBehaviour
    {

    }
}
```

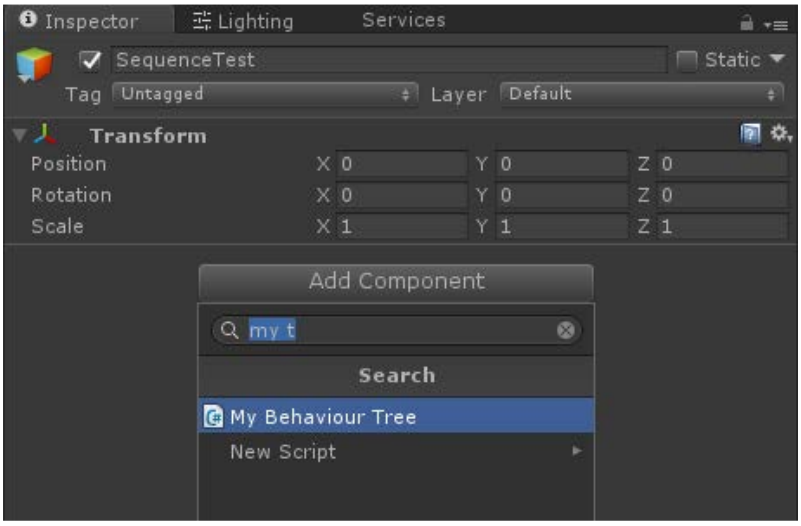
Notice the top of the file, where there are two generated comments. It is recommended that you do not remove this. Certain operations will look for these comments to verify associations with your behaviour tree, so without the comments the tools will not work properly.

It is safe to edit the rest of the file prologue so long as the first two lines remain at the top of the file.

In the other file (The generated file) it is not safe to change anything you find there. The simple reason is that any changes you do make will be overwritten whenever the tree changes.

3.1.1. Attaching a Behaviour Tree to a GameObject

Since behaviour trees are simply MonoBehaviour classes, they can be added to a GameObject in the same way you would add any other component.



You can also attach it by dragging the class file onto the object; either the class file or the generated portion will work.

Dragging the tree object itself however will not work.

3.2. Regenerating Code

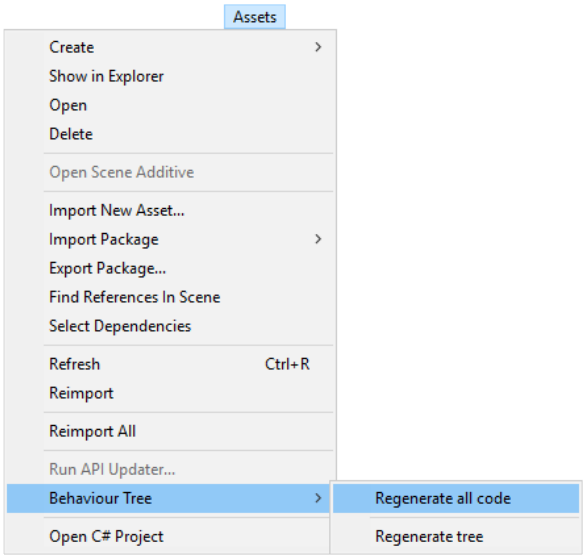
When editing a behaviour tree, the underlying code is not updated automatically. There are, however, a number of different ways to regenerate the code as you work.

You can see when a tree is out of sync by the regenerate code button in the corner of the editor window.

When you make a change to a tree that would alter the underlying code, the button turns amber to indicate that the code is out of date. Clicking the button will regenerate the underlying code and the button turns green.

Alternatively, the code will be regenerated when you simply save your project, so ctrl-s will have the same result.

If you need to regenerate code in all behaviour trees across your entire project, then there is a menu option that will do that.



This may be useful in cases where a package update would yield improvements in the generated code.

3.2.1. Errors whilst regenerating code

In some cases, the code generation will fail. It is entirely possible to construct invalid trees that cannot be converted into code. When this happens, an error will be logged to the Unity console that will tell you were the error occurs in your tree.

The code in the generated file will remain unmodified.

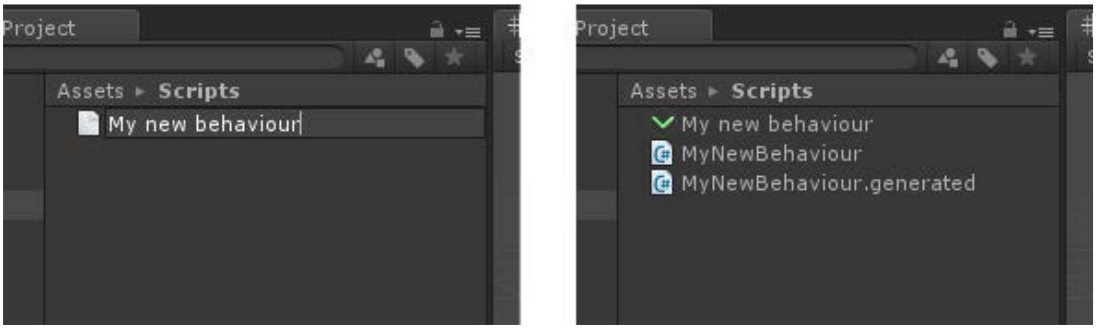
Information on the ways in which trees may be malformed can be found in the node reference section.

3.3. Renaming Files

Since the classes have a generated portion, you need to take care when renaming classes, or changing the namespace. Renaming the individual files yourself can lead to problems. Fortunately, InstinctAI has support for making this as easy as possible.

To rename a class, simply rename the tree file to the name you would like.

Once you do, the class files and the class definition will also be changed automatically.



> *Note*
> It's a good idea to save any pending changes to your script files before attempting to rename them.

Normally, InstinctAI will only ever re-write the generated portion of the class and will leave your user edited part alone. When changing class names, this is one of the few times InstinctAI will modify your own part of the class.

For this reason, it is recommended that you make a backup first if you are renaming files in your behaviour trees.

See the section on code generation options for related topics, such as namespaces.

3.4. Navigating the Tree

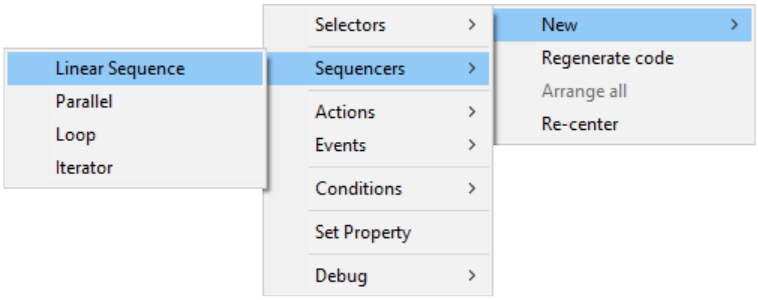
The tree view can be navigated with the middle-mouse button. Click and drag to pan around.

You can also zoom out for a wider view, and back in again. To do this, use the mouse scroll wheel.

If you get lost and need to re-orient the view, simply right-click the editor background to bring up the context menu and select "Re-centre". The view will be snapped back to the centre position of the tree.

3.5. Creating New Nodes

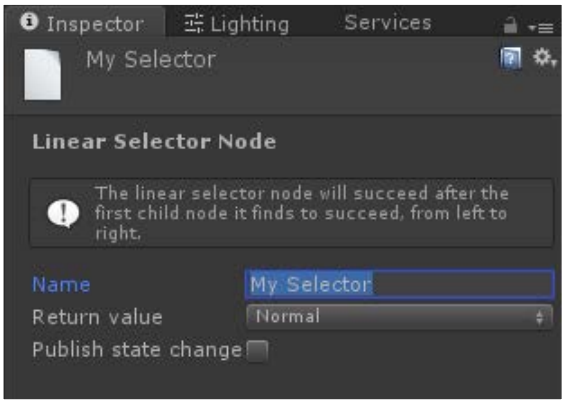
To add a new node, right-click the background of the tree editor and choose the node type from the 'New' sub-menu. The new node will be created under the mouse pointer position.



For details of the different node types, refer to the node reference section.

3.6. The Node Inspector

You can select a node with the left mouse button to view its options in the inspector. Each node type has different types of options, but some options are common to all nodes.

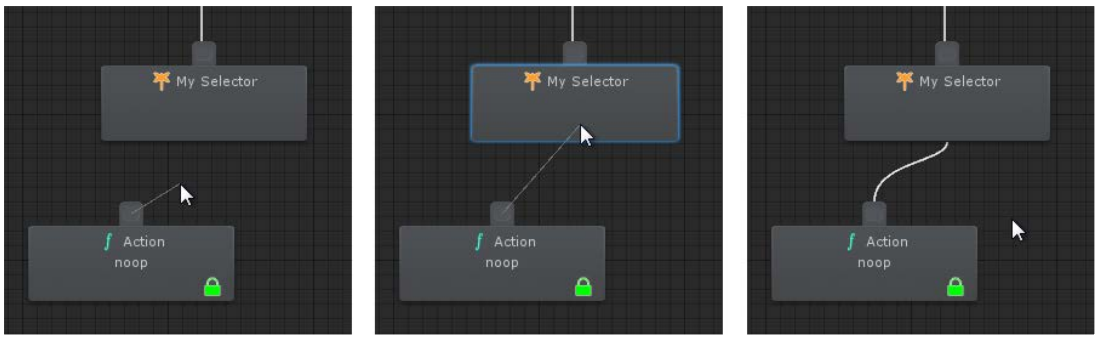


Edit the name in the inspector to label your node in the editor. This is recommended for two reasons. Firstly, it will add clarity to your tree view. Secondly, in some parts of the editor you will need to choose references to nodes that you have created, so in this case the name becomes more important.

Other common options will be explained in later parts of this document.

3.7. Links between nodes

To construct a tree from your nodes, you need to link them together. You do this by dragging a line from the child node handle to its parent. Release the mouse button to create a link. The node handle is the small UI element that protrudes from the top of the node.



Note that some nodes cannot be connected together. If this happens, then the dragged line will turn red.

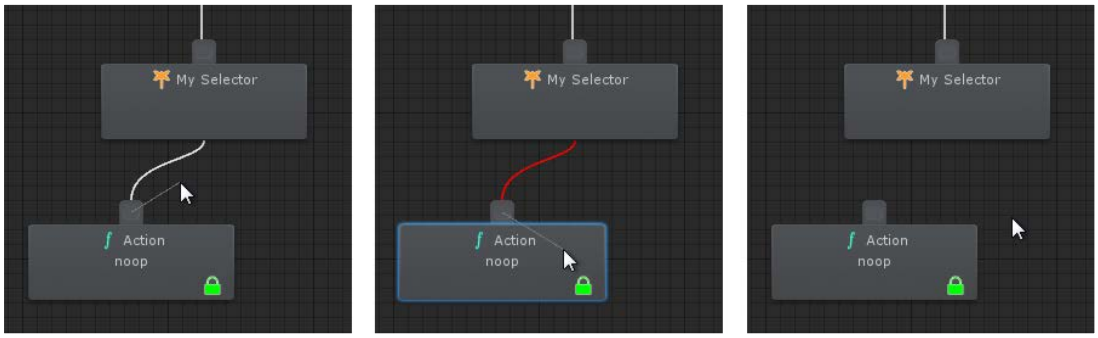
Also, some links will necessarily replace other existing links. If this is the case, any links that would be deleted will turn red.

3.7.1. Unlinking nodes

To delete a link, there are two ways to do it.

Firstly, you can right-click any node and choose “Unlink from parent”. This will remove the link that leads to that node.

Secondly, you can perform a gesture that links a node to itself. This will remove the link to that node handle.



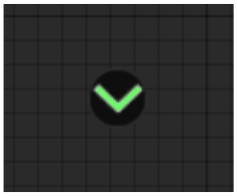
3.8. Moving and Re-ordering Nodes

Nodes can be moved by dragging with the left-mouse button. You can also hold down the control key to drag a node and its children together.

Nodes are executed from left to right. To re-order a set of nodes, (For example in a sequence) then you simply have to move them around.

3.9. The Start Node

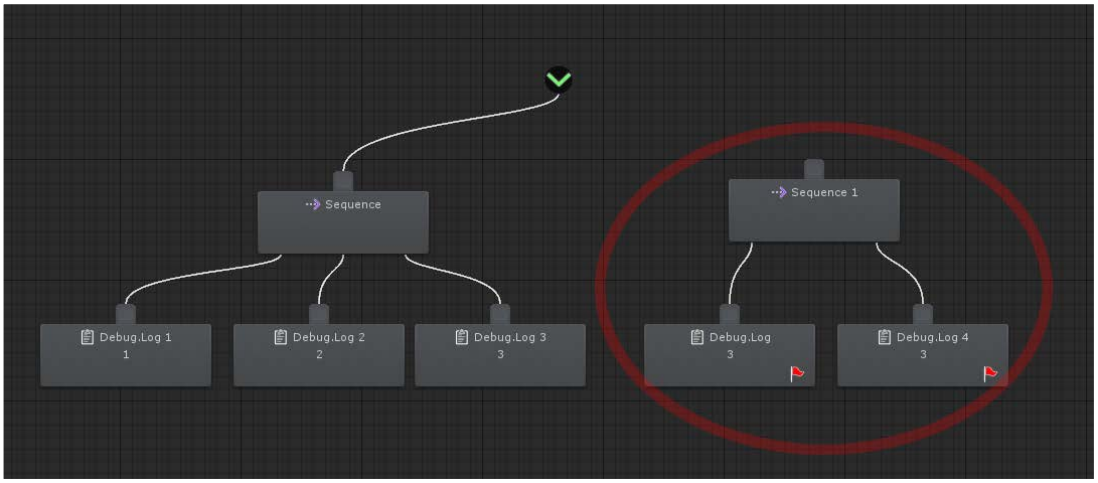
The start node is marked by a circular node with a green downwards arrow.



It has no function on its own other than to point to the initial node. If no node is linked to the start node then no useful behavioural code will be generated.

Any nodes that cannot be traced back to the start node will generate node code whatsoever and will have no impact on your application.

This is useful because you can have unused nodes lying around when you are developing ideas and switch them around by re-linking the start node.



In the above example, the nodes circled in red are not connected to the start node, so they will play no part in the tree and will generate no code. These nodes might represent an alternative idea so you can easily switch to those nodes by linking the other sequence node to the start node, which will also exclude the nodes on the left.

3.10. Duplicating Nodes

You can duplicate existing nodes by selecting the node with the left mouse button, then pressing control-D. The node and all its settings (Apart from its links to other nodes) will be duplicated.

3.11. Deleting Nodes

To delete a node, you must right-click on the node and select Delete Node.

It's good to maintain a tidy workplace, but note that unused nodes will not generate code. See the section on linking to the start node for more information.

4. Playback Mode

You cannot edit your tree in playback mode, but InstinctAI does offer some useful debugging features.

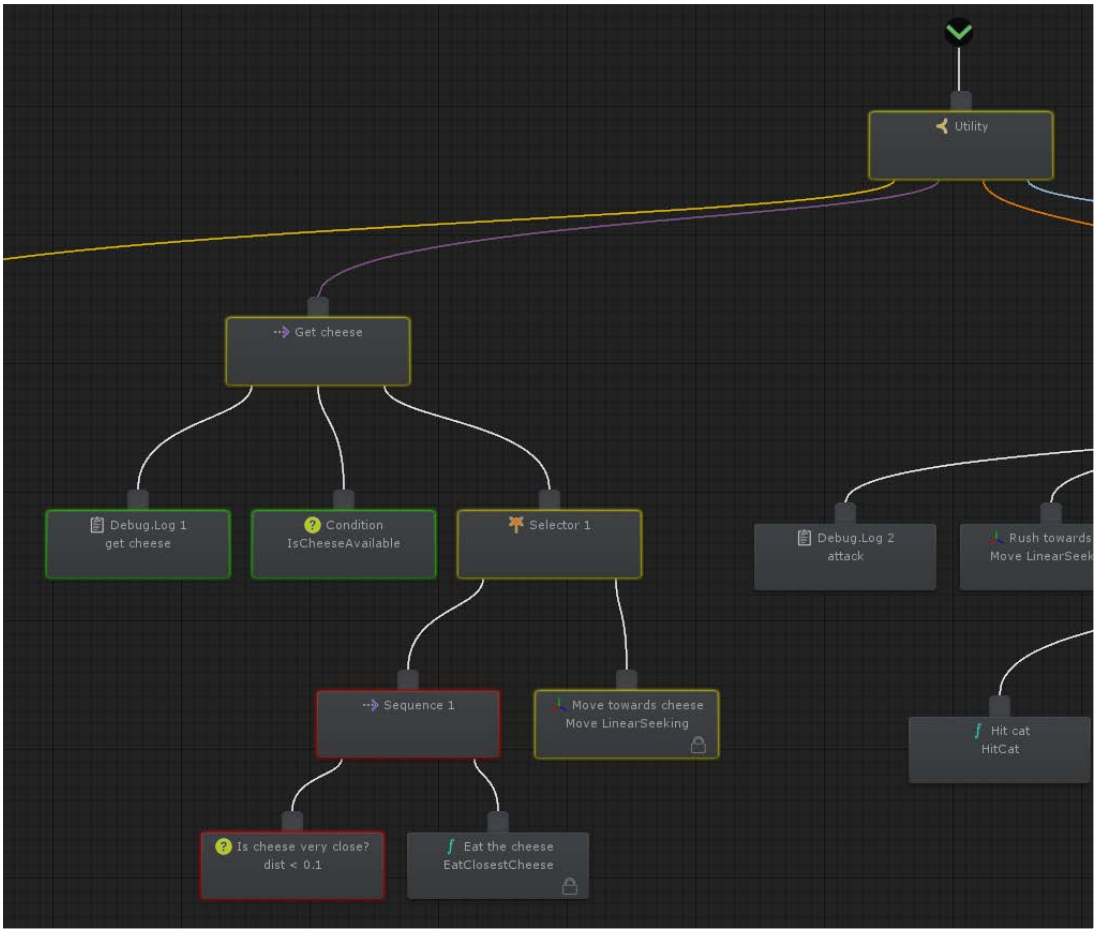
You may be surprised to see an inspector warning in playback mode stating that node values cannot be modified.

The reason for this is that InstinctAI is a code generator. Any modifications while the application is running would necessarily result in a change to the script code of your application and this is not supported.

Any properties on your game objects can of course still be changed in the usual way – just not the properties of the tree itself.

4.1. Live Update

When your application is running, it's often useful to be able to see what the nodes are doing. InstinctAI will display this as live state information in the tree editor.



The image above shows part of a running tree where live nodes are shown illuminated with colours. Running nodes will either be green (For success), red (For failed) or amber (For running). Nodes without a colour glow are not run.

To view live information for a tree, you must select the GameObject to which the tree is attached. Live update information will be displayed only if:

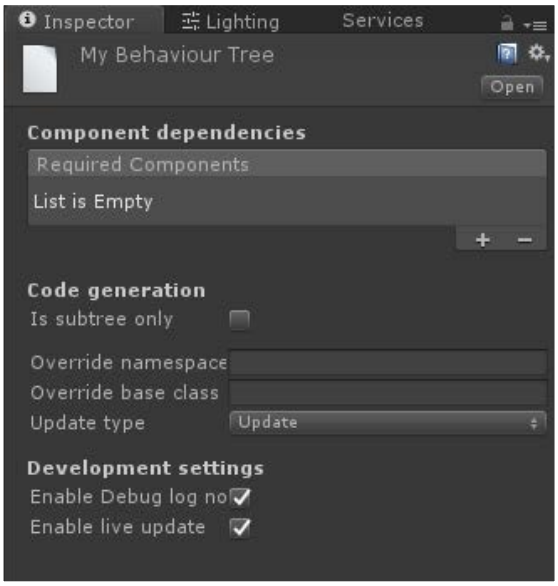
- A `GameObject` with a behaviour tree attached is selected in the hierarchy view.
- The tree editor is open and has the same tree displayed as the one attached to the object.
- The code generation for that tree is up to date.
- The application is running.

Note that if live update is disabled because your code is out of date, then a warning will be overlaid on the editor.

4.1.1. Disabling Live Update

Live updates have a small performance cost when running in the Unity IDE. For release builds, live update is removed from the generated code and the cost disappears. Nevertheless, there are some situations where you would want to disable live update in order to get small performance gains in the development environment.

Live update can be disabled on a per-tree basis from the tree inspector. To view the tree inspector, select the tree file in your project view or click the background of the tree editor.



Uncheck the Enable live update option from the bottom of the inspector options.

You must then regenerate your code.

Note that if you then try to view a tree in playback mode where live update has been disabled, a warning will be overlaid on the editor.

5. About Behaviour Trees

Behaviour trees are a way of modelling different tasks that a game character or some other agent needs to choose from and complete. They are similar in some ways to finite state machines (FSMs), but do not suffer from the spaghetti connections that complex FSM can sometimes get bogged down with.

One way of thinking about a behaviour tree is that it is like a hierarchical state machine where some of the movements between nodes are dictated by rules rather than explicit connections.

The building blocks of behaviour trees are called nodes, and you build the tree by creating connections between them.

5.1. Basic Rules

There are rules that govern the execution of behaviour trees that universally apply to all nodes. Nodes can be in a number of different states. They can be in one of:

- Success
- Fail
- Running

These states are exposed to your code in an `enum` called `NodeVal`.

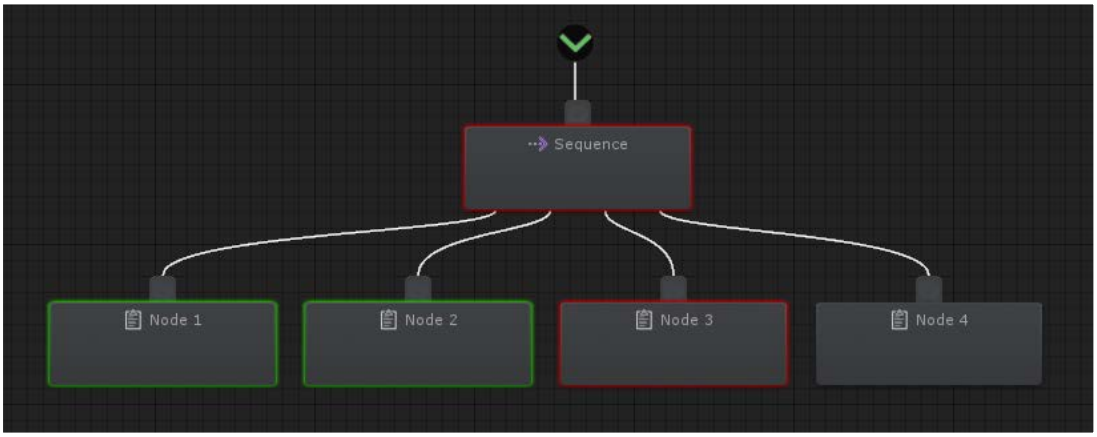
```
namespace com.kupio.instinctai
{
    public enum NodeVal
    {
        Success,
        Fail,
        Running,
        Error,
        Invalid
    }
}
```

You will notice two other states; `Error` and `Invalid`.

These other states are used internally and are safe to ignore in most normal circumstances.

The tree will execute from the top down, and left to right. The tree traversal will be controlled by the node values that are resolved on each node.

For example, a linear sequence node will run each of its child nodes in turn and stop if and when one of its nodes fails.



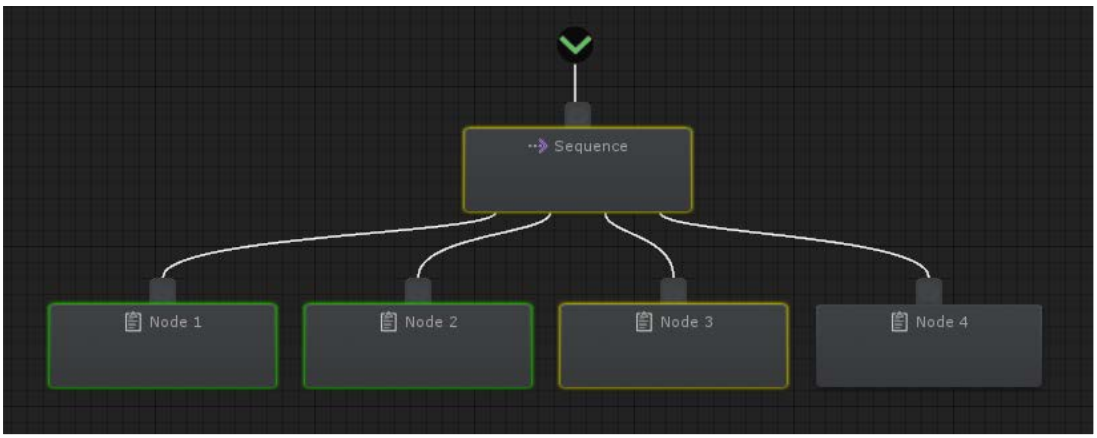
In this example, the sequence node will run through its children from node 1 to 4. The first two nodes succeed, but the third fails. This means that node 4 never gets executed.

It also means that the value of the sequence node itself resolves to a failure. If all of the nodes had succeeded, only then would the sequence node have also succeeded.

5.2. Nodes that run over several frames

Note that a node's action may not resolve to success or failure immediately. In this case, the node will be in a running state.

An example of this might be a node which moves a character from one position to another over time. That node will be in the running state while the character is moving, and will succeed only when the character reaches the target position.



You can see here that a running node in a sequence will also halt progression of the sequence until the node succeeds. In other words, node 4 will not be reached until node 3 has completed and succeeded.

The point to note about this behaviour is that under formal execution of the tree, every frame will execute the tree over again from the root.

In other words, on every frame node 1 and 2 will be executing over again.

This has an advantage since it allows nodes to be easily interruptible. If Node 3 is moving your character around a patrol looking for enemies, then you don't want to wait until the character reaches the target position before it reacts.

For the most part though, you will be concerned with the performance implications of this behaviour, and InstinctAI has a number of ways in which nodes can wait in a particular state, and ways to limit the portion of a tree which gets re-run.

These will be described in more detail in later sections of this guide.

6. Differences from other systems

If you have used behaviour trees before in other systems, or with other Unity assets, then you may have some expectations on how trees in InstinctAI will work. Most behaviour tree systems have their own idiosyncrasies, but this section will enumerate some of the more obvious differences.

6.1. Decorators

Many systems use decorator nodes to modulate the values in nodes or to otherwise modify behaviours. InstinctAI avoids decorator nodes in an effort to preserve space in the editor by not having too many nodes cluttering the screen.

The same functionality is available, but is normally hidden in the node options in the inspector.

For example, an inverter node might be available in one system which inverts the value of a chid node (Success becomes failure and vise versa). This functionality is instead available on the node which you want to invert. Simply select the node and choose 'Invert' as the node value option.

More information about this and other decorator-like functionality will be described throughout the rest of this guide.

6.2. Where is the Blackboard?

Many other behaviour tree systems make use of a concept known as a 'blackboard'. The blackboard is a group of values that the tree has access to that it can modify or refer to.

There is no concept of a blackboard in InstinctAI because it is not required. Instead, your values can be stored in the normal properties of your `MonoBehaviour` class.

If you want to add a property to your class that can be referenced from the tree, simply open the class in your class editor and add it.

In many cases, InstinctAI can also refer to values in other components that are attached to your game object. More information on how to set this up can be found in the component references section.

7. Value use in Nodes

Most nodes require some sort of value input. E.g. a node that checks the distance from a position will require some way of specifying that position, or for specifying a distance as a `float` value.

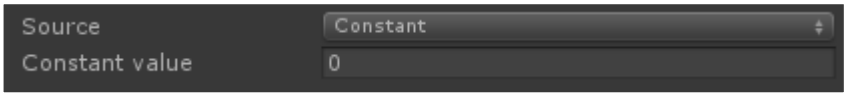
> *Note*
> Most values in nodes are mandatory. If any are unset then you will see a code generation error in the Unity console.

7.1. Value Sources

Nodes generally can take values from different types of places. Depending on the node and the type of value, these places might be properties, constants, random values or the results of method calls. These can either be on the `MonoBehaviour` itself, or it can be on some other component that is attached to the `GameObject`. This section provides an overview of how all this works for nodes in a general sense.

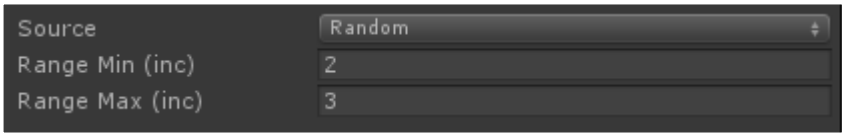
7.2. Constant values

Constant values are the simplest. Simply select constant (Or in some cases 'Vector') and enter the desired value.



7.3. Random values

Random values are usually `float` values, and are specified as a minimum and a maximum range. Values in the range are inclusive.



7.4. Properties

Properties are any properties available to your `MonoBehaviour` that match the type required by the node.

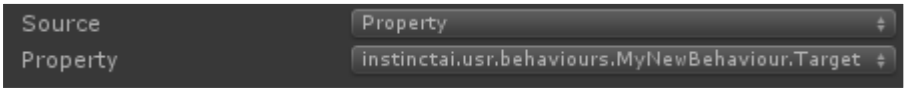
To add a new property, simply add it to your class in the usual way.

```
namespace instinctai.usr.behaviours
{
    using UnityEngine;

    public partial class MyNewBehaviour : MonoBehaviour
    {
        public Vector3 Target;
    }
}
```

In the example above, the `Target` property has been added as a `Vector3`.

Once added, it will appear in the node inspector. Choose it from the list to bind it to the node.



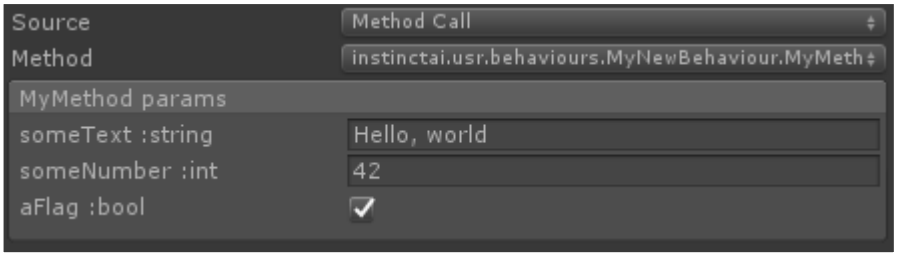
Note that you also have access to private, protected or any other properties inherited from a custom base class.

7.5. Method Calls

Method calls can be used for values in exactly the same way as properties. You can select any available method with a suitable return type, and even methods with primitive parameters.

You can use methods from other components too. See the preceding section on other components for more information on setting this up.

When you select a method call, if there are any parameters required, then an inspector control will appear to allow you to enter those parameters.



See also the advanced use section on more information on more complex things you can do with method calls.

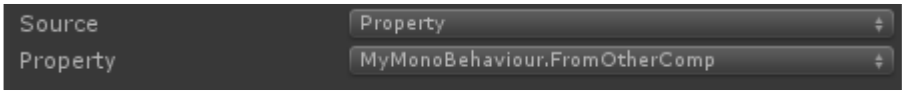
7.6. Other Components

InstinctAI also provides a mechanism to expose properties and methods on other components that are attached to the `GameObject`. To enable this, you need to make your tree aware of the other component types.

Go to the tree inspector by selecting the tree in the project view, or by clicking the background of the editor. At the top of the inspector, there is a list of required components.

Add the class that you need to access properties of by clicking the + button.

This will do two things. Firstly, the selected class will become a required component on your `MonoBehaviour`. Secondly, the properties of the other component will automatically appear in the property and method lists in your node inspectors.



8. Node Options

There are a few options that are common to most types of node.

8.1. Overriding Values

Nodes evaluate to success, failure or running states. These values can be overridden to give finer control over the traversal of the tree. This is controlled with the return value option of each node.

The return value option can be one of the following values:

Normal

This is the default and the node's value is not modified.

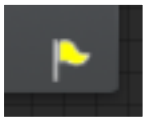
Always Succeed

This forces the node to always succeed. Useful to proceed a sequence in all circumstances.

Always Fail

This forces the node to always fail. Useful to proceed a selector in all circumstances.

Always Running



This forces the node to always have a running state. This is not normally useful except in development to try out ideas.

Invert

This makes failing nodes succeed and succeeding nodes fail. Running nodes are unaffected.

9. Node Reference

In this section, we will look at each of the different types of node that are available and explain their behaviours.

9.1. Decision Making Nodes

Behaviour trees are not decision trees. They are intended to express a hierarchy of behaviours. That said, there are nodes which must control the flow throughout the tree in order to select correct behaviours for agents in your game. You won't find logic nodes or script-like logic statements. Instead, flow is controlled mainly by two different types of nodes. Selectors and Sequencers.

Note that the Utility node is a specialised type of selector node, but it is a far more complex type of node and so has its own section later within this manual.

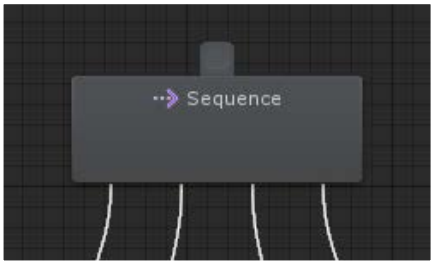
9.1.1. Linear Selectors

A linear selector is a node which tries each of its children in turn until it finds one that succeeds. It will try each child in turn from left to right, and to change the order in the editor, see the section on re-ordering nodes.

A selector is commonly used to priorities other sub-trees of behaviour. Place the most important behaviour to the left (E.g. take cover from fire) and the least important to the right (E.g. patrol an area).

Note that a selector must have at least one child node, otherwise a code generation error will appear on the Unity console.

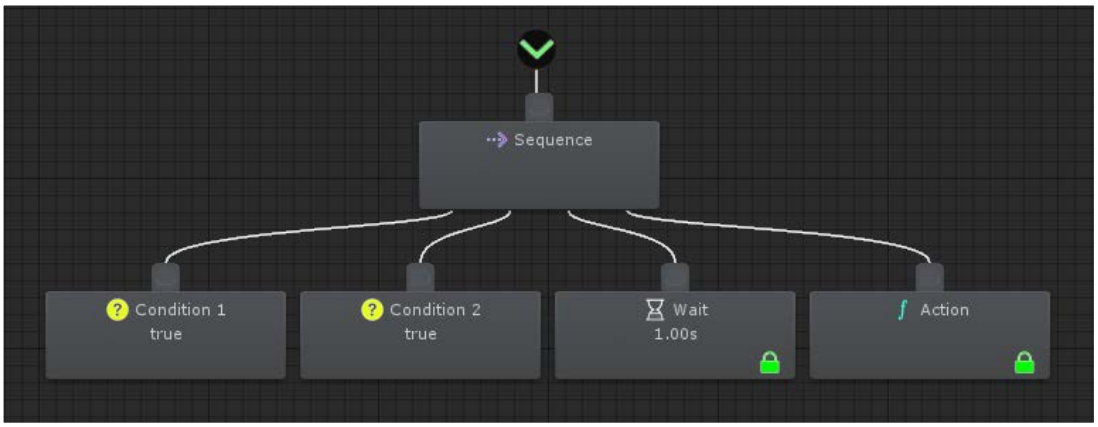
9.1.2. Linear Sequencers



A linear sequence node is similar to a selector except that it tries each of its children until it finds one that fails. It will succeed itself only when all of its children succeed.

A sequencer is commonly used to sequence an ordered set of actions.

It is also commonly used to gate an action predicated on one or more conditions.



In the above example, both conditions 1 and 2 must be successful (true) before the sequence will continue to the next nodes. It will then wait for 1s and then perform some action.

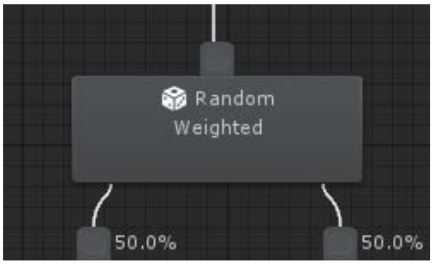
Note that a sequencer must have at least one child node, otherwise a code generation error will appear on the Unity console.

The dynamic checkbox in the sequencer's inspector gives special treatment to the running state of child nodes.

Normally, if there are no other latch behaviours in the nodes in question, the tree will start from the root on every frame. In the case of a dynamic sequencer, if one of its children is in the running state, then it will return to the first child on the next frame.

In the example above, if the sequence was marked as dynamic, then the conditions would be constantly re-tested if the wait node was running. Nodes elsewhere on the tree however would not run.

9.1.3. Random Selector



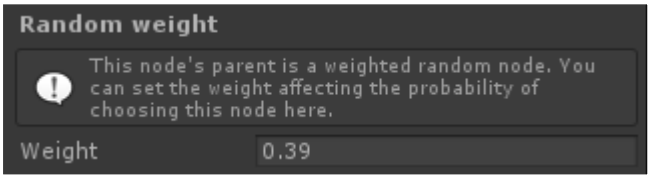
A random selector will simply choose one of its children at random to execute.

The selection can be made in one of two ways. It can be either uniform, or weighted.

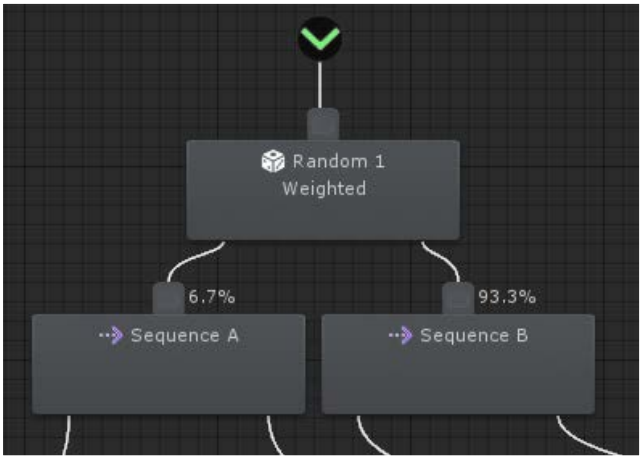
In a uniform selection, each child node will have an equal chance of being chosen.

In a weighted selection, different nodes can be given different weights with higher weights being more likely to be chosen. A weight of 6 has double the chance of being chosen as a weight of 3.

To set the weight of a node, the weighting must be set on the child node in question. When you select a node, and it is the child of a weighted random node, then a new option will appear.



The percentage chance of selection will appear next to the node in the editor.

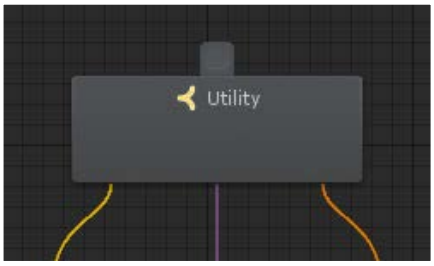


> *Note*
> An easy way to set a chance as a percentage is just to use percentages as your weight numbers. Just make sure they add up to 100.

The random node also has an anti-repeat option.

This is only available when in uniform mode. When checked, the random choice will not choose the same option twice in a row.

9.1.4. Utility Selector



The Utility selector is a specialised selector which selects which of its children to run based on the node with the highest utility, (or usefulness) at any given moment.

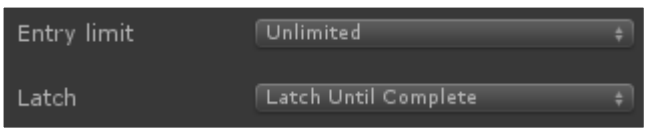
This node is a complex one and has its own editor, apart from the normal inspector controls.

For an in-depth explanation of this node and its editor, see the utility editor section later in this guide.

9.2. Action Nodes

Put simply, action nodes are nodes that do something.

Action nodes are always leaf nodes; that is to say that they do not have child nodes of their own. In this section, we will look at each of the action nodes that you can use to build more complex behaviours. You can also create your own custom action nodes. These will be described in the section on Custom Nodes.



9.2.1. Entry Limits

All action nodes have an entry limit option. This will limit the number of times a node can be run before the tree is reset. The tree is reset when it begins again from the root. The entry limit icon will show on nodes with this function enabled.

By default, the option is set to unlimited.

Other options are to limit by rate (By limiting the number of entries per second) or by count (By limiting the number of entries to a specific number).

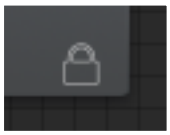
9.2.2. Latch Options

The latch option on action nodes describes how long the tree should linger on the node when it's in a running state. The default value varies between nodes, but many will default to the Latch until complete option.

To describe why latches are useful, let's first consider the 'No latch' option.

> *Note*
> Latch options are not available to nodes that are part of a parallel node. In this case, you should use the latch option of the parallel node itself. See the section on parallel nodes for more information.

No latch



If an action is unlatched then the tree will begin again from the root on the next frame. For small/medium numbers of agents this might be completely acceptable from a performance perspective, and in many situations it may even be desirable.

Unlatched actions are inherently interruptible. As an example, consider a movement node which is moving a character across the screen to a specific location. If the character needs to be interrupted at some point (Perhaps it comes under fire) then you want the behaviour to immediately change to something else.

In this example, the no-latch option on the movement node would be a simple way to achieve this. The behaviour tree of the character would be fully re-examined on each frame.

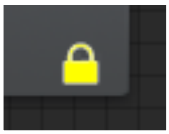
Latch until complete

This option is the perfect opposite of the no-latch option.

With this option, the node will remain active on every frame until it is completed. This is the best option from a performance perspective because no other nodes are considered while the latched node is running.

An example of a node where this might be useful is a recoil movement when a player has been hit. You might decide that the recoil must be played out in full and it should be uninterruptible.

Reset tree after timeout

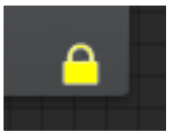


This option will latch the node in a similar way to Latch until complete. The difference is that you can set a timeout to prevent a node running too long.

If a node completes normally then it proceeds as usual.

If the timeout triggers then the tree will reset to the root node on the next frame and all nodes will be reconsidered.

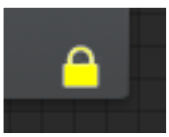
Complete after timeout



This option is similar to Reset tree after timeout but it instead of resetting to the root node it will behave as though the node had completed.

For example, if the node is part of a sequence, then after a timeout the tree will simply proceed to the next node in the sequence.

Reset to node after timeout



This option is also similar to reset tree after timeout.

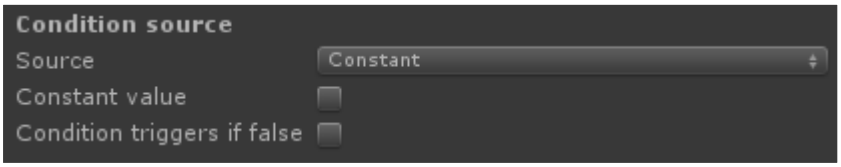
For this option though, you can reset to a specific node in the tree rather than evaluating the entire tree again.

Any available nodes to jump to will appear in the Latch jump target list. Not all nodes will be available. Most nodes that the tree must pass through to get the action node are normally available jump targets.

This is one reason that it's important to name your nodes clearly; so that they may be listed clearly in choices of jump targets in the inspector.

Reset tree after condition

Similar to Reset tree after timeout, but instead of a timeout triggering a bail-out of a running node, you can use your own custom condition.



The source of the condition is a boolean value which can be a method call, or a property. It can also be a constant, but this is only really of use in development if you want to force a particular value to test out an idea. See the section on custom condition nodes for more information on how to create condition values.

The Condition triggers if false toggle will essentially invert the value of the condition. This means that if it becomes false, the node will bail out and the tree will reset.

Reset to node after condition

Similar to Reset to node after timeout, this will allow you to choose at what point the tree will reset to.

Also, similar to Reset tree after condition, this will allow you to specify a custom boolean bail-out condition. See the section on custom condition nodes for more information on how to create condition values.

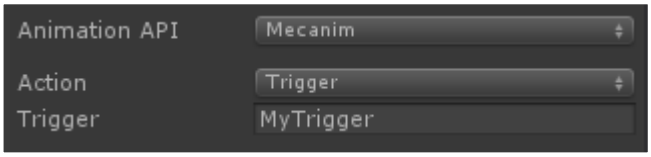
Complete after condition

Similar to Complete after timeout, this will auto-succeed the node.

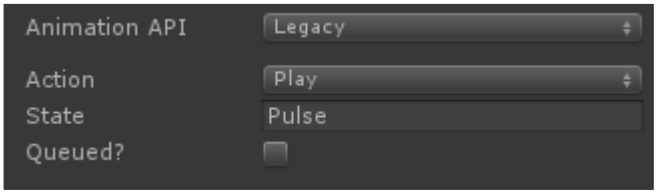
Also, similar to Reset tree after condition, this will allow you to specify a custom boolean succeed condition. See the section on custom condition nodes for more information on how to create condition values.

9.2.3. Animation

An animation node can trigger an animation in an animation component attached to the `GameObject`. It works with both mecanim and the older legacy animation API. When you add this node to your tree, the generated MonoBehaviour will be marked as requiring either the `Animator` or `Animation` Unity engine component.



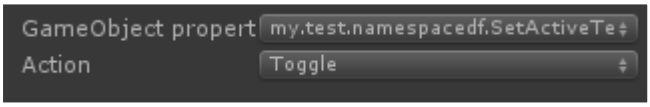
In the mecanim mode, you can trigger animations by entering the name of the trigger. Other options are play, cross-fade and setting values in the mecanim animator variables.



In legacy mode you can play, blend, cross-fade, rewind or stop animations in a way which mimics the functionality of the legacy API.

9.2.4. Set Active

The set active node will set a `GameObject` in the scene active or inactive. The target object must be available as a property of type `GameObject`.



The node has the option to set, unset or toggle the target object's active state.

This node always succeeds immediately.

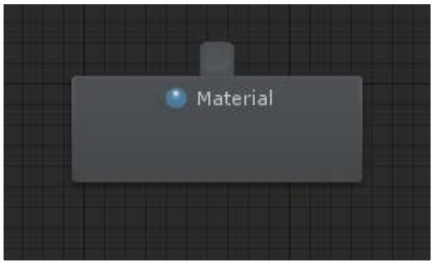
9.2.5. Invoke Event

Invoke event will invoke a `UnityEvent` which is exposed as some property on your `MonoBehaviour`, or in an attached component.

Simply select the event from the drop-down list and it will be invoked when the node is entered.

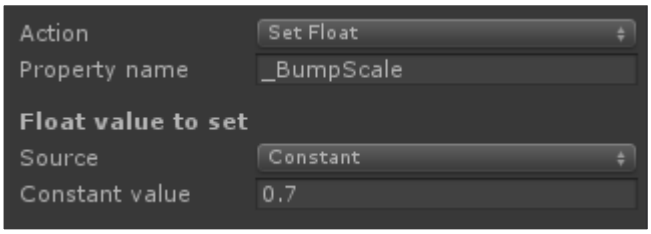
This node always succeeds immediately.

9.2.6. Material



This node allows you to modify materials on the current object's renderer by modifying shader values. When you add this node to your tree, the generated `MonoBehaviour` code will be marked as requiring the `Renderer` Unity engine component.

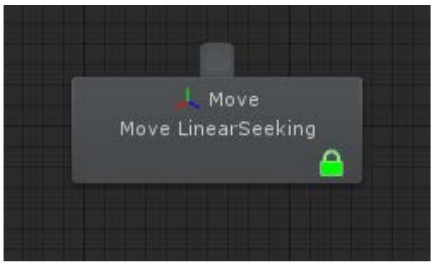
You can choose to set a numerical or colour value in the material shader. Choose the action to perform from the inspector drop-down box.



The value to be set can be a constant, or it can be taken from the result of a property or method call. It can also be set to a random value. See the section on value use in nodes for more information on how this works.

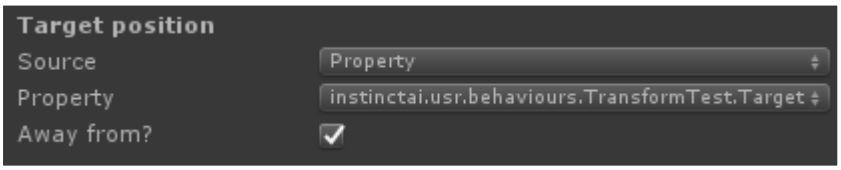
This node will always succeed immediately.

9.2.7. Move



The move node will move the attached `GameObject` towards some target position over time. The node will return the running state while the object is moving, and will succeed when it reaches the target.

> *Note*
> The move node simply changes the transform of the object. If you want to move a physics object, you should instead use the Rigid body node or the Navigation node.

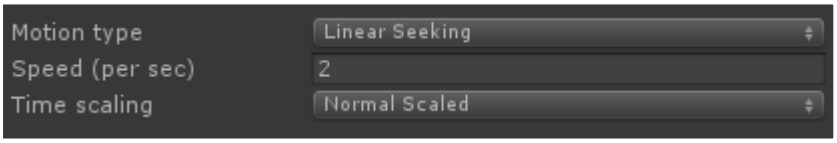


The target to move towards is specified as a `Vector3` property or as `Vector3` returning method call. It can also be set to a constant value if the source is set to Vector. See the section on value use in nodes for more information on how this works.

The Away From toggle will make the object move away from the given point rather than towards it.

> *Note*
> Moving away from a point means it will never reach the point and the node will be in the running state forever. See the completion latch conditions for ways to set custom completion criteria for the node.

There are also some options that can affect how the object will move.



The motion type may be one of the following

Immediate

Immediate moves will instantly move the object to the target position and will succeed immediately.

Linear Seeking

Linear seeking moves will move at a constant speed towards the target until the target is reached. The seeking part of the name means that the target may change while the node is running and it will always move towards the most recent position.

In linear seeking mode, the speed is specified in units per second, where the time aspect can be in scaled time if required.

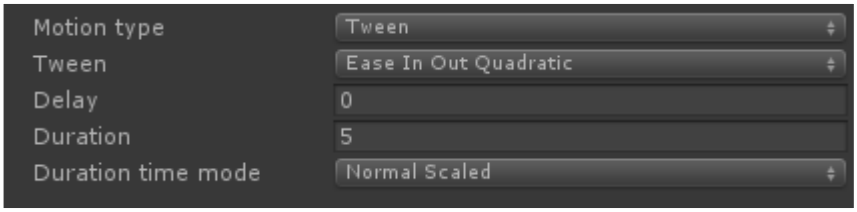
Forward

Forward motion moves in the object's `Vector3.forward` direction at the specified speed. It can also be moved backward if the away from toggle is checked.

In this mode, the state will always be running, so a latch should not be used, or a custom completion condition should be set. See the completion latch conditions for more information.

Tween

In the tween mode, movement is softened by accelerating and/or decelerating at the beginning and end of the motion.



Ease in will accelerate from 0, and ease out will decelerate to 0 at the target. There are a number of options available for how easing is applied and the strength of that easing.

Using tweens however implies some limits on the way movement works, when compared to the other movement modes.

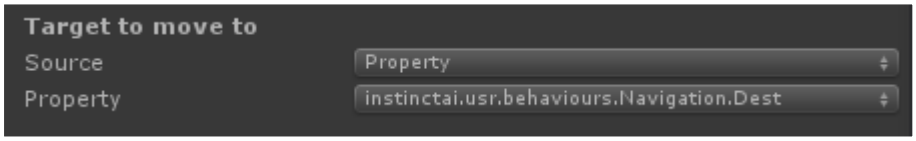
Firstly, the movement is no longer a seeking behaviour, so the target position will remain constant until it is reached; you can't move the target around.

Secondly, the movement is now specified by total time rather than by speed. In other words, set the duration to 5 and the motion will take exactly 5 seconds to complete from current starting position to the target position.

9.2.8. Navigation

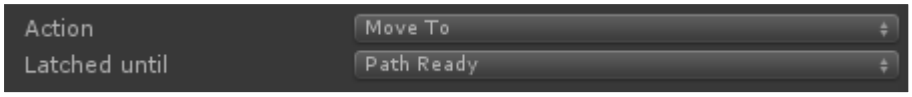
The navigation node will move your character around a `NavMesh`. When you add this node to your tree, `NavMeshAgent` becomes a required component on your `MonoBehaviour`.

The position you want to move to is set with the target property on the node.



The position is a `Vector3` which is set from a property, method call or a constant value. See the section on value use in nodes for more information on how this works.

The navigation node has a number of options for which action it is to perform.



Move to

Move to will initiate a movement towards the target position. If a latch is set on the node, then the Latched until option will control what the latch is waiting for. It can either wait for Unity to calculate the path (Which may take more than one frame) or it can wait until the character has reached the target.

If the destination cannot be reached, then the node will fail.

Warp to

The warp to option will move the character immediately to the start position. The node will succeed immediately too, unless the target is unreachable, in which case it will fail.

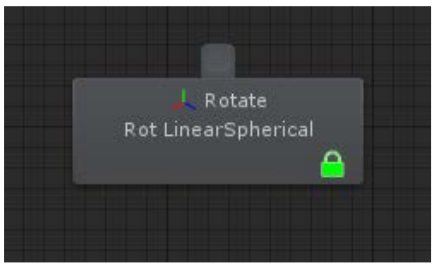
Stop

If the character is moving towards a target, then this node option will stop it in its tracks.

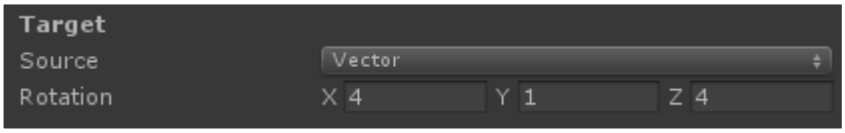
Resume

If the character is part-way along a path and has been stopped, then this option will start it moving again. This option also has the same latch options as the Move to mode, and can wait for the character to reach the target.

9.2.9. Rotate

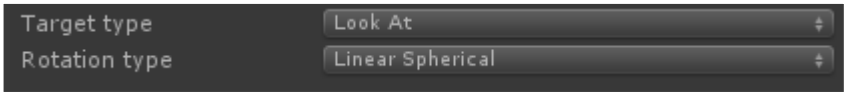


The rotation node operates in a similar way to the move node, but instead of changing an object's world position, it will change an object's orientation over time.



The target can be set to a constant value as a Vector. Alternatively, it can be a property or method call with a value of `Vector3`, `Transform`, `GameObject` or `Quaternion`. See the section on value use in nodes for more information on how this works.

How the target value is interpreted is specified in the rotation type options.



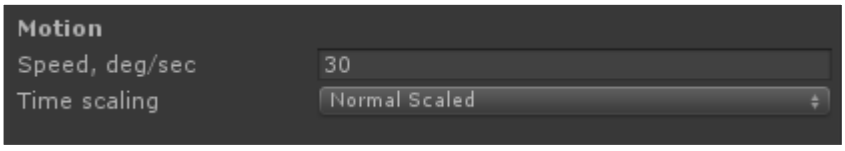
The target type may be set to angles, representing Euler angles, or it may be set to Look at, representing some position in space that you want to rotate towards. Finally, it may be Look away, specifying that the object should turn away from some point in space.

The rotation type is similar to the movement type in the move node. It may be one of the following

Linear Spherical

This will rotate at a constant speed towards the target. Spherical refers to the type of rotation – the path it takes will draw a straight line across the surface of an imaginary sphere. In other words, this is the most natural looking path to take when interpolating a rotation between two values.

The movement speed is specified in degrees per second.

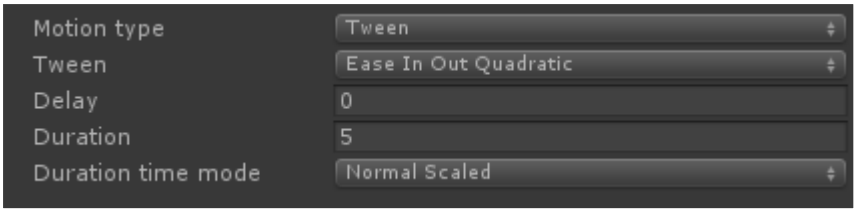


Immediate

Immediate mode will change the rotation to the target value immediately. This mode will not enter the running state.

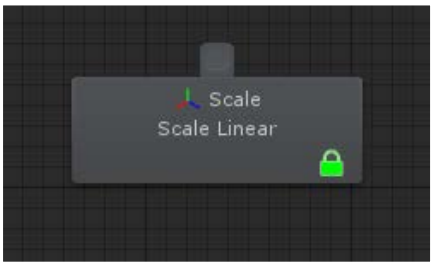
Tween

Tweening will smoothly accelerate and decelerate the rotation.

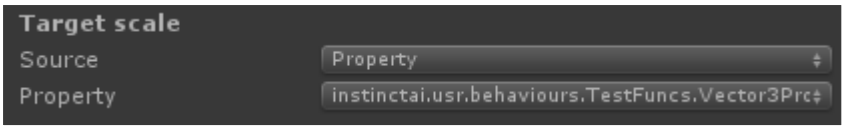


Tweening in rotations is subjected to the same caveats and restrictions that affect the movement node. See that section for more information on how tweens limit behaviour.

9.2.10. Scale

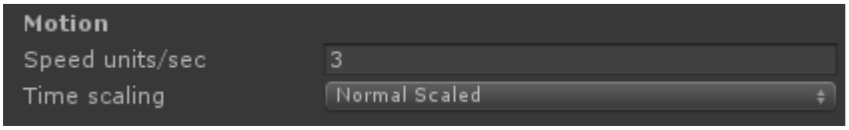


Scaling, like movement and rotation, will alter an object's transform. This time it alters the object's scale towards some target value.



The target scale can be taken from a constant value (Vector option) or it can be taken from a Vector3 property or method call. See the section on value use in nodes for more information on how this works.

The speed is set in units of scale per second.



In other words, if you want to scale from scale 1 to scale 3 in 1 second, the speed should be 2 units per second.

The scale motion type can be one of the following.

Linear

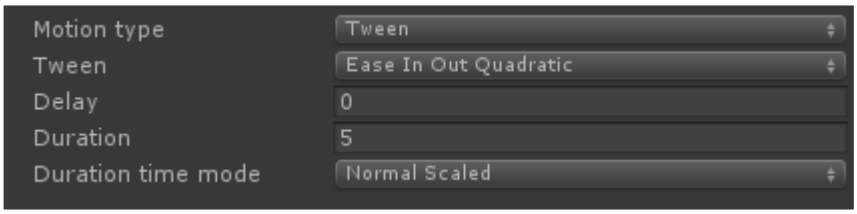
Linear will change the scale smoothly towards the target scale at a constant rate.

Immediate

Immediate will change the scale to the target value immediately. This mode will not enter the running state.

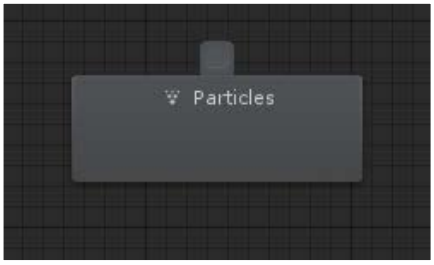
Tween

Tweening will smoothly accelerate and decelerate the change in scale.

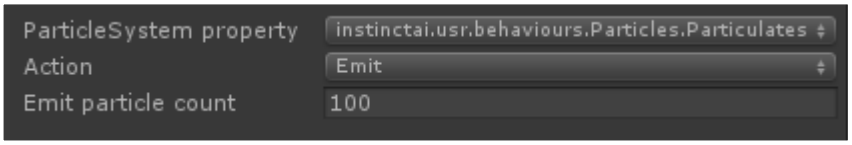


Tweening in scale nodes is subjected to the same caveats and restrictions that affect the movement node. See that section for more information on how tweens limit behaviour.

9.2.11. Particles



The particles node can interact with a particle emitter using Unity's shuriken particle system.



The particles to be interacted with must be exposed as a property of the type `ParticleSystem` from the Unity engine.

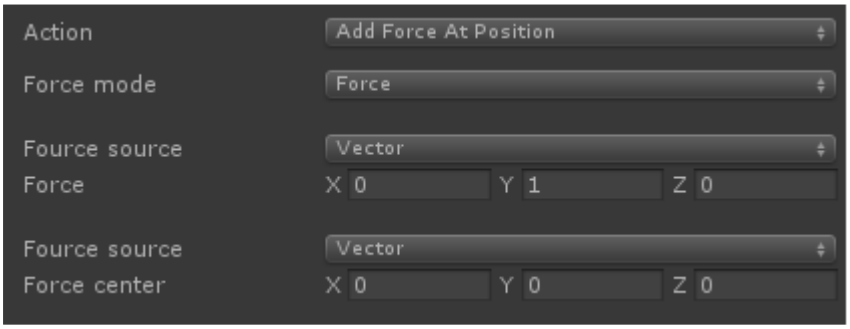
You can then select a method to perform on the particles from the options Play, Pause, Stop or Emit.

If you choose emit, you can also enter the number of particles to emit, which will be emitted each time the node is entered.

9.2.12. Rigid Body

The `RigidBody` node can interact with the physics properties of the attached `GameObject`.

It can operate in either 2D or 3D mode, and depending on which you choose, it will add a required component of type `RigidBody` or `RigidBody2D` to the `MonoBehaviour` class.



The action to perform is specified in the options Action and Force mode. The actions available are one of the following.

Add force

Adds a physical force to the `RigidBody` where the force value is the direction of the force.

Add relative force

Similar to add force, but works relative to the local coordinate system of the attached `GameObject`.

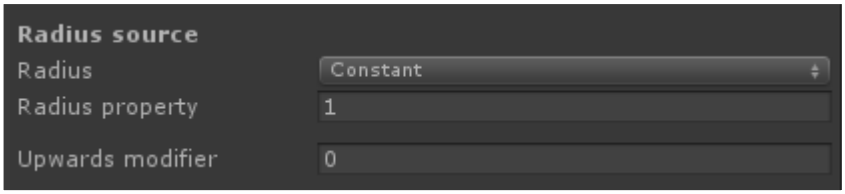
Add force at position

Similar to add force, but applies the force at the given force centre value. This may result in a rotation of the object.

Add explosion force

This action is unavailable in 2D mode.

Adds an explosive force calculated from a force centre and explosion radius.

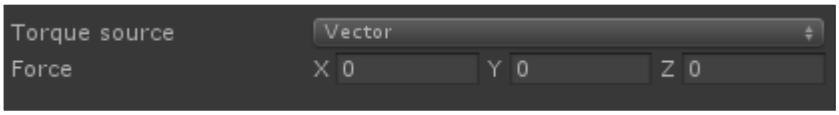


The upwards modifier value will also make the explosion seem to lift objects upwards.

Add torque

This action is unavailable in 2D mode.

Adds a rotational torque to the object.



Add relative torque

This action is unavailable in 2D mode.

Similar to add torque, but works in the object's local coordinate system.

Move position

This will move the object to the target position. The settings on the `RigidBody` component will determine how the object should move.

Move rotation

This will rotate the object to a target orientation. The settings on the `RigidBody` component will determine how the object should rotate.

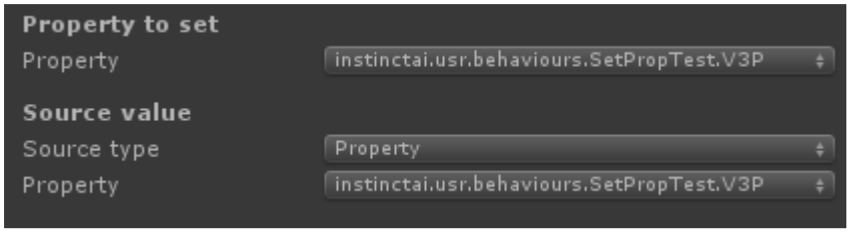
9.2.13. Set Property

The set property node will set the value of a primitive property on your `MonoBehaviour`. It can do this either simply by assignment, or it can calculate the new value over time.

The assignment mode can be one of Assign, Tween or Rove.

Assign

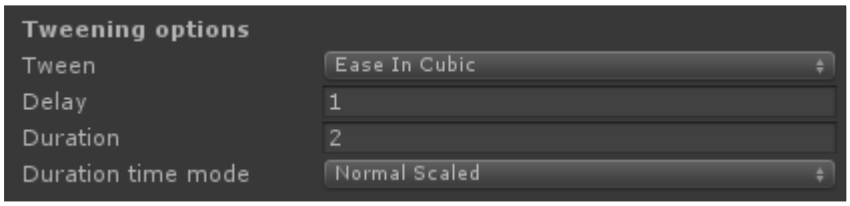
Assign is the simplest since it simply assigns a value to a property and succeeds immediately.



The value to set the property to can be a constant, another property or it can be the result of a method call. When selecting a source value, only sources that match the type of the target property will be listed.

Tween

Tween mode will move the property value from its current value to a target value over time.

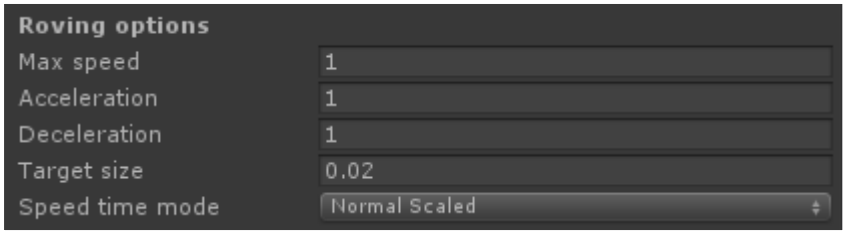


Acceleration and/or deceleration will be applied to the value. If you do not want this effect, set the tween mode to Linear.

Note also that in tween mode, the latch must be set to Latch until complete. If not, there will be a code generation error reported on the Unity console.

Rove

The rove option (Roving value) is similar to tween except it works with changing target values. An example of this might be where you want a gun turret to seek out a target by rotating towards it, but you want it to accelerate and decelerate as though it was tweening.

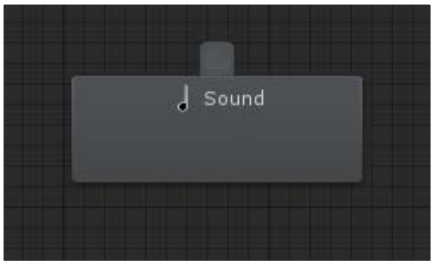


To enable roving, you must supply a maximum speed value, as well as acceleration and deceleration values expressed as units per second per second.

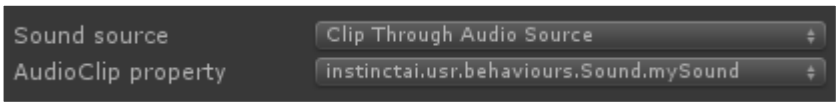
You should also supply a target size, which is a threshold below which the value is considered to have reached its target. When this occurs, the true value of the target value will be immediately assigned to the target property.

Note also that in tween mode, the latch must be set to Latch until complete. If not, there will be a code generation error reported on the Unity console.

9.2.14. Sound



The sound node can trigger audio or sound effects. If you use this node in your tree, then the `AudioSource` component will be marked as a required component on the `MonoBehaviour` (Unless you use the clip at position source mode).



The component has three source modes, each of which always succeed immediately.

Clip through Audio source

This will play the audio that is set up on the Audio clip property setting through the attached `AudioSource` component.

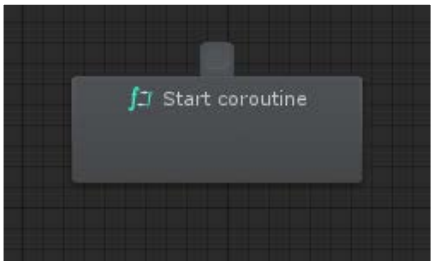
Clip at position

This will play the specified clip property as though it was positioned at the current objects position in world space.

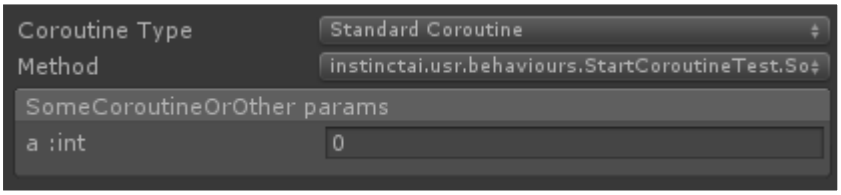
Audio source

This will simply play the attached `AudioSource` component as it has been configured in the inspector.

9.2.15. Start Coroutine



This node will start a coroutine if you have specified one in your code.



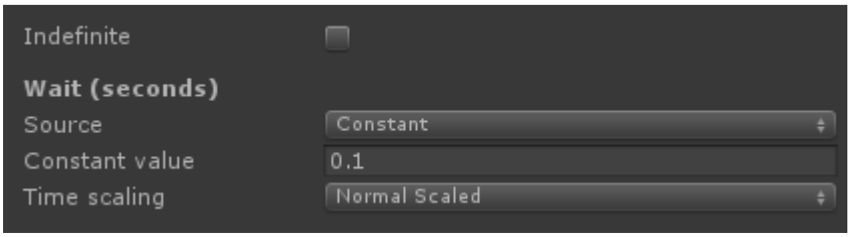
Add a function with the return type `IEnumerator` to your `MonoBehaviour` and it will become available in the method list of this node's inspector.

Any parameters that are required by your coroutine will be specifiable here too.

This node always succeeds immediately.

9.2.16. Wait

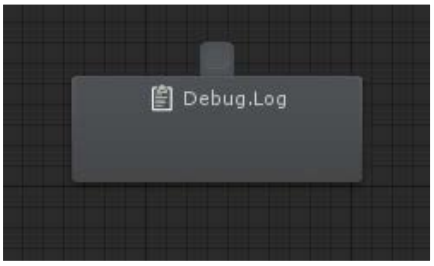
The wait node will simply wait for a set period of time.



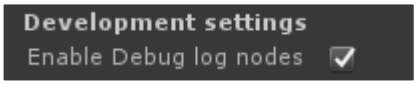
The length of time to wait (In seconds) can be set from a property, a method call or a constant.

The wait can also be made indefinite by checking the indefinite toggle. This is useful when combined with a custom conditional latch when you want to wait on some condition.

9.2.17. Debug Log



The debug log node will simply output a message to the Unity console. It's useful for development if you want to trace the behaviour of the tree.

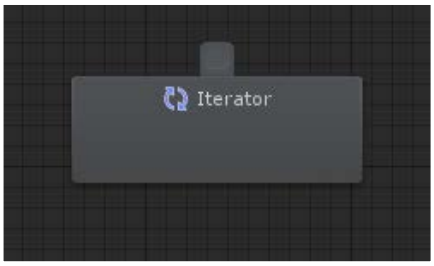


You can also disable all debug log nodes on a tree-by-tree basis. Simply go to the inspector of the tree itself by clicking on the tree object in the project view. You will see a toggle for debug log nodes in the inspector. Uncheck this and regenerate the code; the debug log nodes will no longer be active in the code.

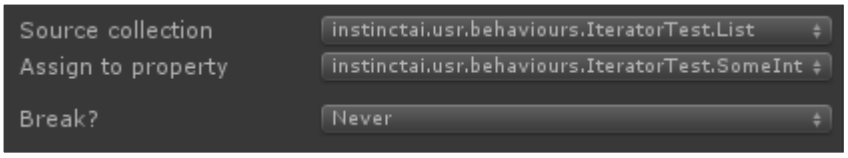
9.3. Loop Nodes

Loop nodes are a class of node that will repeat their child node a number of times. Loop nodes of all kinds can have only one child node, so this makes them act as a decorator node to the child node.

9.3.1. Iterator



An iterator node will repeat a node a number of times; once for each member of some collection.



The source collection must be a C# collection type.

On each loop, the next value in the collection is assigned to the specified property. The properties listed are properties that match the collection type.

By default, the loop will succeed once the collection is exhausted. There is however an optional break condition to control early exit from the loop.

Never

This is the default option. The loop will not break early.

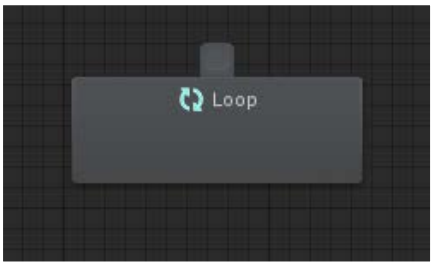
On Failure

This option will terminate the loop if the child loop evaluates to failure.

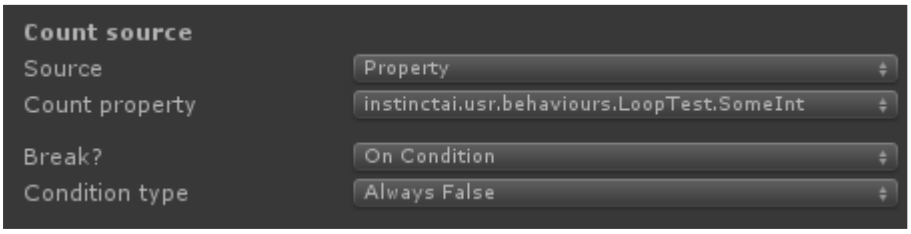
On Condition

This option will terminate the loop if the specified custom boolean condition is triggered.

9.3.2. Loop



The simple counted loop node will simply run its child node a set number of times.



The count can be a simple constant, or it can be taken from an integer property or method call.

Note that the count value is checked on each frame, so the count can change mid-loop.

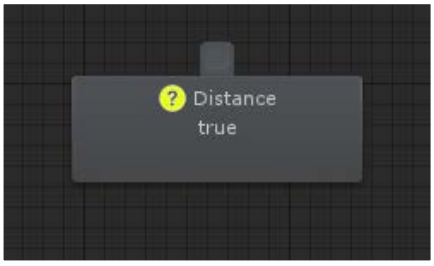
The loop can also exit early with a break condition. The break options are the same as those in the iterator node.

9.4. Condition Nodes

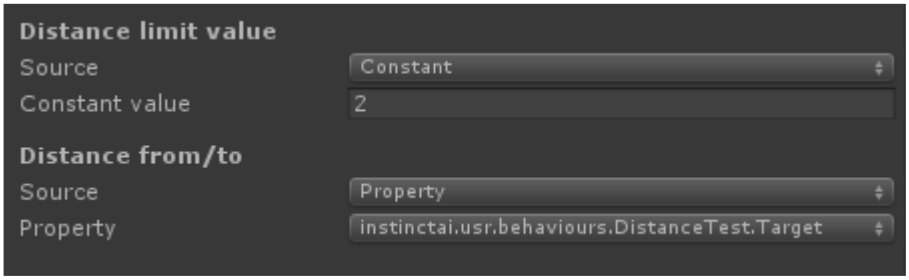
Condition nodes are, (like action nodes) always leaf nodes. They will either succeed or fail based on some boolean decision.

Much of the time, for a condition in your tree you will want a custom condition node. This is described later in the Custom Nodes section of this manual.

9.4.1. Distance Check



The distance check node will evaluate to success if the distance to a point is less than some limit, otherwise it evaluates to failure.



The distance check node has two main properties – the distance limit and the point to measure to.

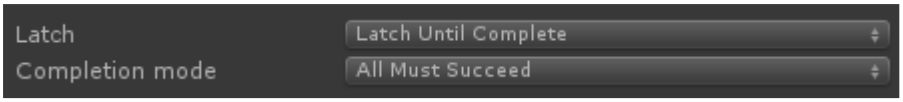
The limit value can be any `float` constant, property or method call. For more information on values, see the section on value use in nodes.

The point to measure to can be any `Vector3` value representing a world position. The distance is measured from the world position of the attached `GameObject`.

9.5. Aggregation Nodes

9.5.1. Parallel

A parallel node will run each of its children simultaneously, allowing more than one to be in the running state at a time.



Any children of a parallel node will have their latch options disabled. The latch on the parallel node becomes the virtual latch state for all the child node operations.

The parallel node may only contain action nodes, so for example sequences cannot be parented to a parallel node.

Note too that the parallel node has a Completion mode option. It can be one of the following.

All must succeed

The parallel node will wait for all children to no longer be in the running state before returning its own value. It will succeed only if all its children succeed.

Any must succeed

The parallel node will complete once any of its children succeeds. Any nodes still in the running state will be aborted.

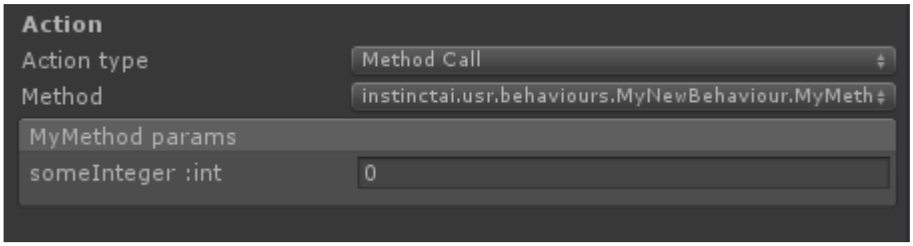
9.6. Custom Nodes

Custom nodes are nodes that require you to write supporting code. These can be conditional nodes, or custom actions that can do whatever you like.

9.6.1. Custom Actions

Custom actions are simply methods in the `MonoBehaviour` that perform some task. You can write a custom action method by writing a method that has the `NodeVal` return type.

In the node itself has only two modes.



The first is the noop action type, which simply does nothing.

The second is the more useful Method Call action type which calls a custom method.

Here is an example of the simplest custom action that you can write.

```
using com.kupio.instinctai;
using UnityEngine;

public partial class MyNewBehaviour : MonoBehaviour
{
    public NodeVal MyMethod(int someInteger)
    {
        return NodeVal.Success;
    }
}
```

When you add this to your class it will appear in the list of methods in the custom action node inspector.

Parameters are optional, but if you add them and they have primitive types then you can supply parameter values in the inspector. Be careful - if you add parameters of types that are unsupported then the method will not be listed in the inspector.

In this example the method does nothing but succeed immediately. Other values you might return are `NodeVal.Fail` or `NodeVal.Running`. If you return `Running`, then the function will be called again on subsequent passes or subsequent frames until it succeeds or fails.

For more information on custom action methods, see the section on special variable names.

9.6.2. Custom Conditions

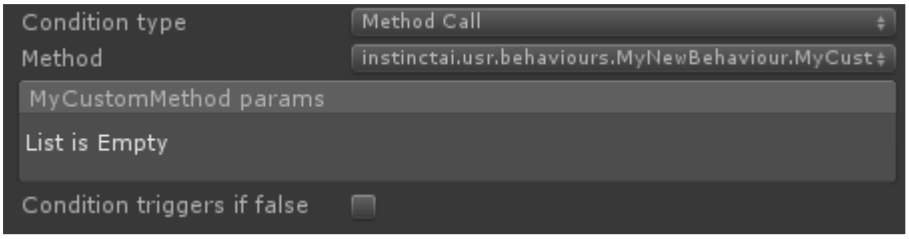
Custom conditions, like custom actions, require you to write a code to support boolean decisions.

A custom condition can be a boolean property or a method call. These two pieces of code are both examples of valid custom conditions.

```
public bool MyCustomProp;
```

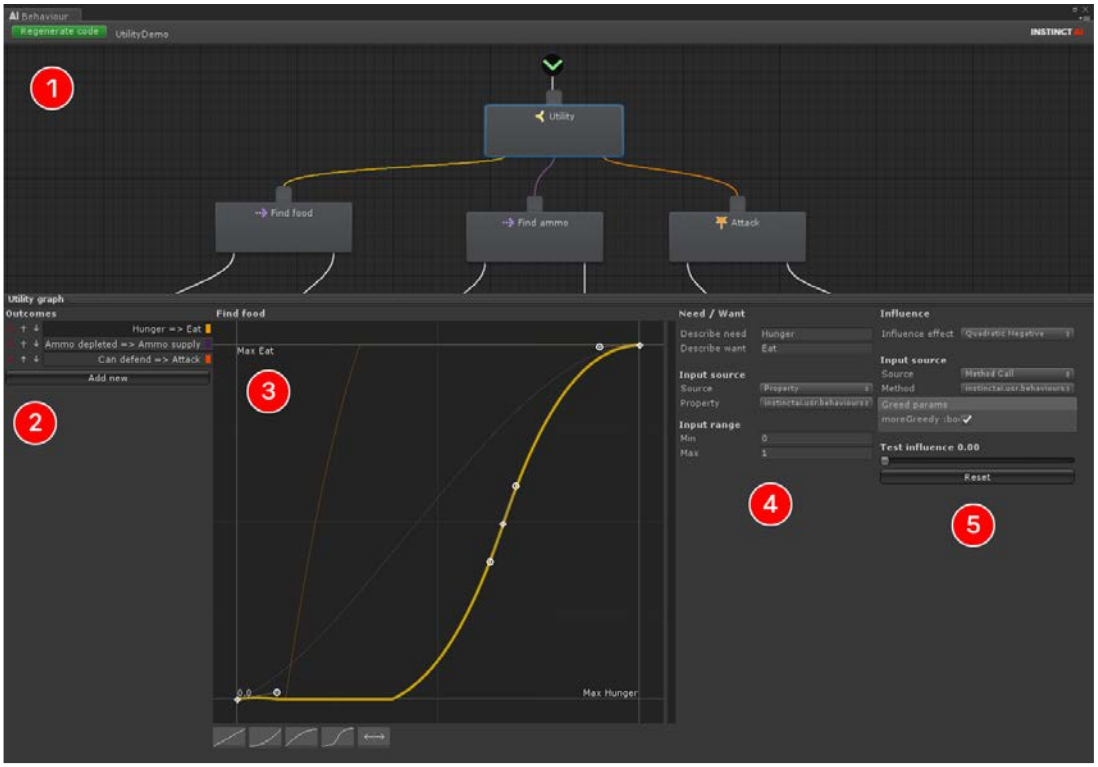
```
public bool MyCustomMethod()
{
    return true;
}
```

If you added these to your MonoBehaviour, then both would appear in the custom condition inspector.



There is also a Condition triggers if false toggle. This inverts the condition, so if this is checked, then the condition succeeds only if the method or property is false.

10. Utility Editor



The Utility node is a selector node that selects which of its children to run based on which has the most utility, (or usefulness) at any given moment.

Above is an overview image of a utility node showing the different sections of the utility editor.

Panel ❶ is the normal tree editor. Panels ❶, ❷, ❸, ❹ and ❺ appear when a utility node or an immediate child of the utility node is selected.

10.1. The Outcome Panel

The outcome panel is the panel to the left of the utility graph, and marked as ❷.

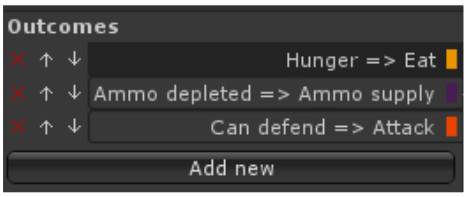
The Utility node must select from one of its children according to which is the best choice. To do this, each decision must be expressed as an outcome, and there will be one outcome definition for each child node.

In the overview image there are three child nodes, and three possible matching outcomes. To make the association clear, the outcome list is decorated with colours and the colours are reflected in the lines between the nodes.

When you first create a utility node, there will be no outcomes created, even if there are nodes connected to it. You must explicitly create the correct number of outcomes. To do this, press the Add new button in the outcomes panel.

If you have a mismatch between the number of child nodes and the number of outcomes, then there will be a code regeneration error in the Unity console.

You can re-order the outcomes by pressing the up/down arrows next to them, and you can delete them with the small red cross.



The order of outcomes from top to bottom matches the order of the nodes from left to right.

> *Note*
> The maximum number of child nodes supported by a utility selector is limited to 32.

10.2. Needs and Wants

Each outcome is expressed in terms of a need and a want. For example, a hunger level in your character represents a need for food. This should map onto a want to find food.

For each outcome, you should label the need and want at the top of the panel marked as ❹.

Take care with the terms, as it is important to make the terms clear and unambiguous.

10.3. Utility Curves

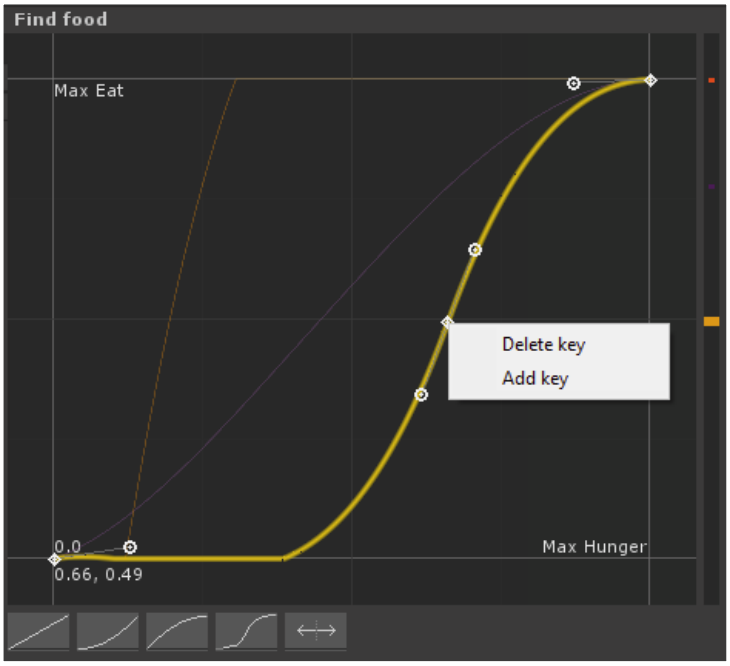
The central part of the editor is the utility curve editor. This is shown in the overview image as ❸.

The curve editor works in a similar way to the Unity animation curve editor.

First select your outcome on the left, and the curve will be highlighted in the outcome's colour. Along the horizontal axis is labelled with the outcome's want (For example hunger) from minimum to maximum. Along the vertical axis is labelled with the outcome's need.

The idea is to create a curve that represents how one maps to the other. For example, with hunger there may be no want to find food at low values, but the need to eat may rise sharply at some point.

To edit the curve, you can add points by right-clicking and choosing Add key from the context menu. Points can be deleted the same way.

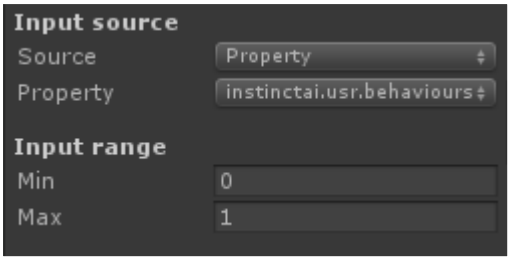


The fewer points you can get away with the better it is from a performance perspective.

There are also preset curve shapes in the buttons below the graph to use as a starting point.

10.4. Input values

To drive this system, you need to provide input values. This is done in the input mapping section of panel ❹.



Input values, like in many other node values can be either float properties or method calls. They can also be constants or random values, but these types don't generally make sense in utility nodes; they are provided simply for the sake of completeness.

For each input value, you would also set a minimum and maximum value range.

For example, you might have a property that contains the hunger level of a character, and you might decide that it ranges from 0 to 100. You would choose the property name in the input source, and enter 0 and 100 into the min/max boxes.

Once you have done this for each input, and once you have tuned the curve for each to reflect the curve mapping of each input and output, then you are basically done. You can regenerate the code and the selector will evaluate the curves based on the range of input values. It will then select the best outcome and run that child node.

10.5. Fall-back behaviour

There is also an option for fall-back behaviour in the main inspector – apart from the utility editor panels.

This defines what should happen when the best selected outcome fails. By default, the option is set to fail. In this mode, the utility node will also fail and will therefore perform no action whatsoever.

The second option is Next best. In next best mode, the utility node will select the next best option. It will continue to select the next best option until it either finds one that succeeds, or it runs out of options.

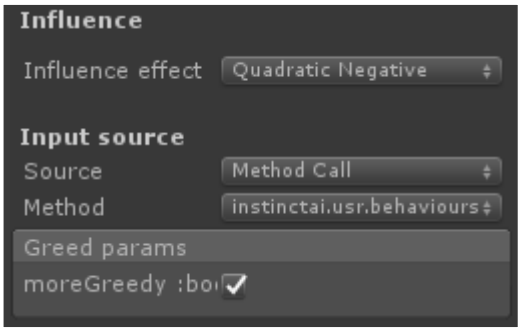
10.6. Influence values

There is also an optional step called influence values. This is shown in the overview image as panel ❺.

Influence values distort the curve either negatively or positively. The idea is that these can represent personality differences between agents. For example, you might want to set up a gluttony value that determines how tempted by food a character is. By increasing the gluttony influence you can make a character more likely to seek food at lower levels of hunger.

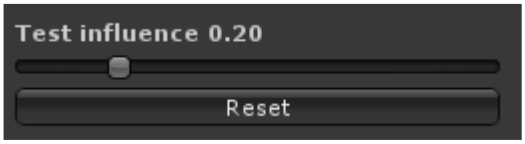
To set one up, you must create a `float` property or method in your class. The value of this property must be in the range 0 to 1. Normally this would be a property so that you can set the character's personality values in the inspector as you would any other value.

Once you have set up the property, simply select it in the influence value panel and set the type of influence that it has.



The influence effect value has a range of different options for positive and negative effects to varying degrees of strength.

The best way to work out what the right effect to choose is, is to try them out. For this purpose, there is a test slider on the panel.



Simply move the test slider and the effect on the curve will be shown visually in the graph editor.

> *Note*
> You can still edit the curve with the test value set to some non-zero value.

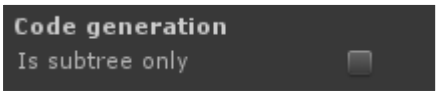
11. Code Generation Options

Some code generation options are available in the inspector for the tree itself.

11.1. Subtrees

Subtrees are currently unsupported, but some incomplete functionality is still exposed in the interface.

In the tree options is a checkbox marked Is subtree only.



This turns the start node of the tree orange to denote that it is a subtree. Unfortunately, the subtree node is set for release in a future update.

It does however have a useful side effect, which is to disable code generation for the tree, so the functionality has been exposed for that reason only.

11.2. Base class

In the tree options you will see an option for changing the base class.

By default, it is empty and defaults to `MonoBehaviour`.

If you override this, you should choose another class which is a subclass of `MonoBehaviour`. If you choose a non-`MonoBehaviour` class then the code generation will fail.

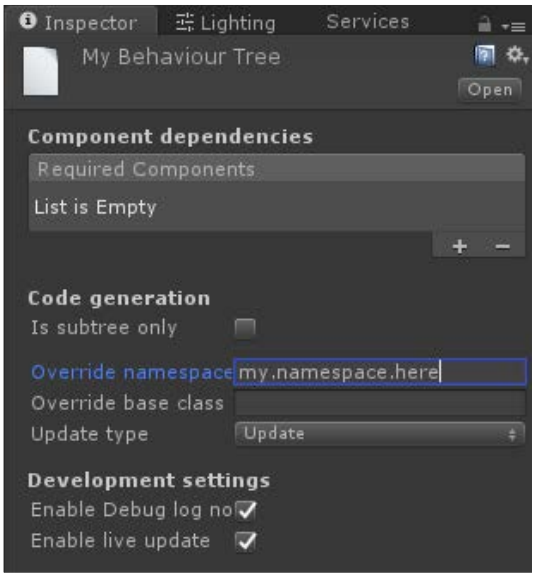
This feature is useful if you want to share custom actions between different trees. Place your custom action code into a common superclass and you can make use of it in multiple places.

> *Note*
> This is one of the few times when InstinctAI will modify the code in the non-generated portion of the class. For this reason, you should make sure you have backed up your files before changing the base class.

11.3. Namespace

All behaviour tree code must exist in a namespace. Having no namespace is not an option.

To change the namespace, you can select the tree object in the project view and change the namespace in the inspector. The necessary file changes will be made automatically.



If you leave this blank it will default to the namespace `instinctai.usr.behaviours`.

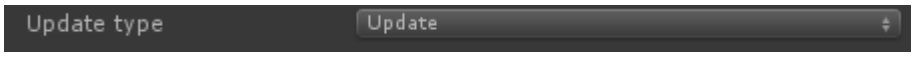
Normally, InstinctAI will only ever re-write the generated portion of the class and will leave your user edited part alone. When changing namespaces, this is one of the few times InstinctAI will modify your own part of the class.

For this reason, it is recommended that you make a backup first if you are changing namespaces in your behaviour trees.

11.4. Update Loop Options

By default, InstinctAI generates an Update method in the generated portion of the class. This is not always desirable since you often want to be able to add your own code to the update method.

Options to help with this can be found in the tree options.



This can be one of three options.

Update

This is the default as described above.

Fixed Update

This is similar to the default, but will generate a `FixedUpdate` method instead of an `Update` method.

Manual

Manual gives you the most control. The tree is generated with an `UpdateAI` method. This should be called manually from your own code – most likely from an `Update` method that you create yourself.

This allows you to be more flexible and do things such as not call the tree update on every frame.

12. Advanced Topics

12.1. Special Variable Names

When using method calls to get values in nodes, some values can be automatically populated.

If one of your parameters is named `nodeFrameCount`, for example, the parameter will not appear in the node inspector parameter list. Instead, it will be populated with the number of calls to this node since the tree was reset.

You can use this to detect first calls to a node by checking for `nodeFrameCount == 0`.

12.2. Published State Changes

Sometimes you want to know what state the tree is in from your own code. To help with this, InstinctAI provides a feature called published state changes.

Each node has a checkbox which, when enabled, allows you to label a state change that you can then read from your code.

Let's say we have two nodes that we want to know the state of from our code. In each node, we label the state with a unique name.

When you regenerate the code, InstinctAI will generate an enum that matches your labels.

```
public enum PubState
{
    None,
    MyFirstState,
    MySecondState
}
```

You can then simply check the state from your own code.

```
if (this.State == PubState.MyFirstState)
{
    Debug.Log("In first state");
}
```

It's important to note that the state only changes when the node with a published state change is entered. It does not reset when the tree resets, so it will have an old value until a node changes it.

If this functionality matters, then simply publish a state change on the root node.

> *Note*
> Nodes that are children of parallel nodes cannot publish state changes.

12.3. Attaching multiple trees to objects

Note that there is no reason at all why you can't have multiple behaviour trees attached to one `GameObject`. Simply attach them in the same way you would attach a range of other components to the same object.

You might want to have separate trees governing different aspects of a character, for example one tree controlling where the character walks, and another controlling the character's gaze and facial expressions.

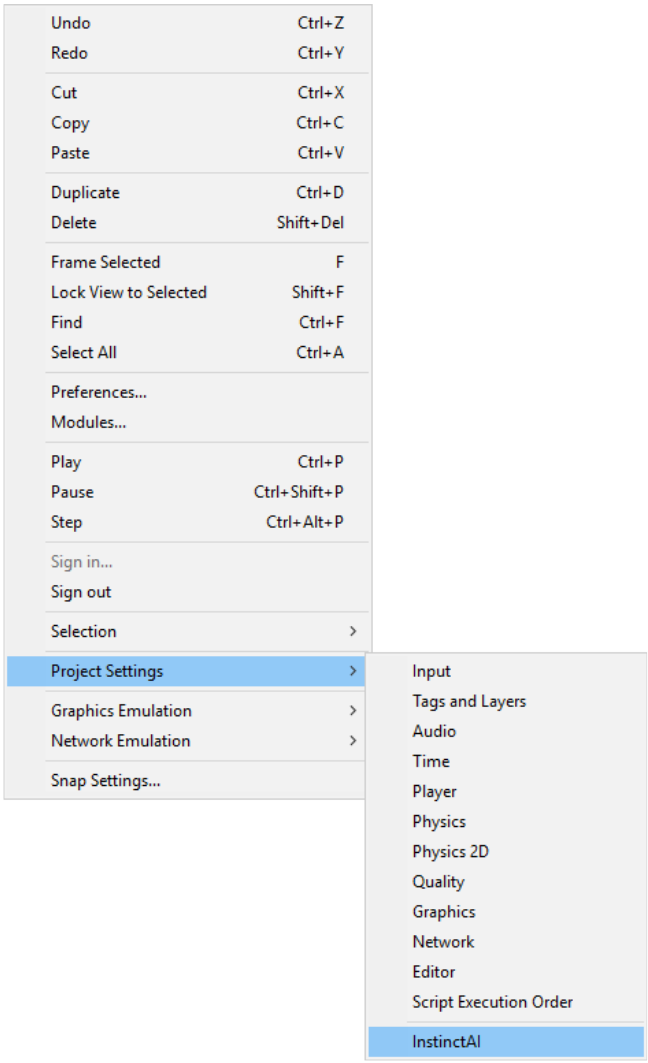
So long as the functionality does not overlap and you're clear about the role of each behaviour tree, then there should be no functional problems.

The only down-side is that Live update does not operate correctly in this case since it makes an assumption that you only have one tree per object.

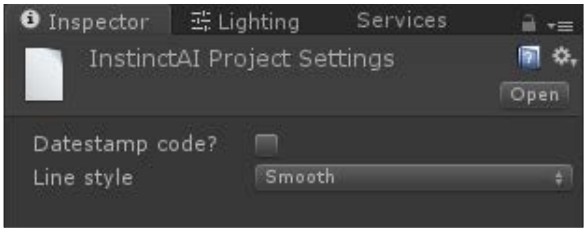
This issue is being addressed in a future update of the asset.

12.4. Project Settings

InstinctAI adds some settings to your project. To edit them, choose InstinctAI from Unity's project settings sub-menu.



This will open up the InstinctAI project preferences in the inspector.

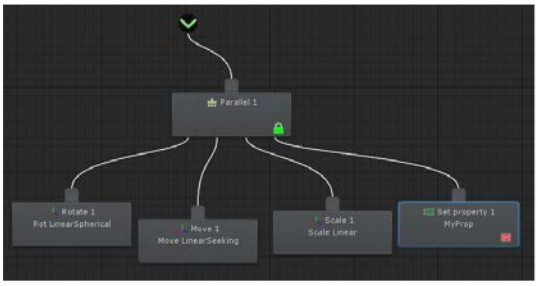


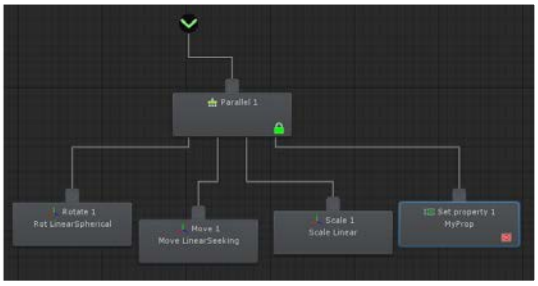
12.4.1. Date-stamp code

If checked, then when code is generated it will have a date-stamp in the file prologue at the top of the file. This can be useful to verify when the content of a file was generated, but is normally switched off since it can generate useless diffs when using source control applications such as git or subversion.

12.4.2. Line style

By default, the behaviour tree editor will draw lines between nodes using smooth curves. If you prefer to see stepped lines then you can change the display mode here.



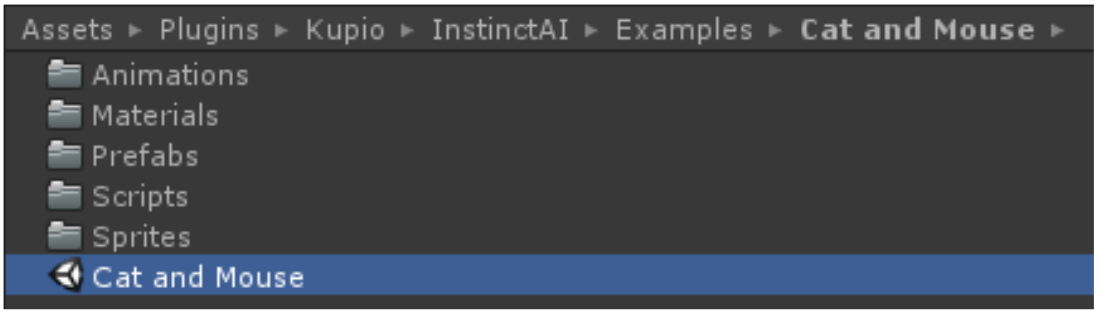


13. Examples

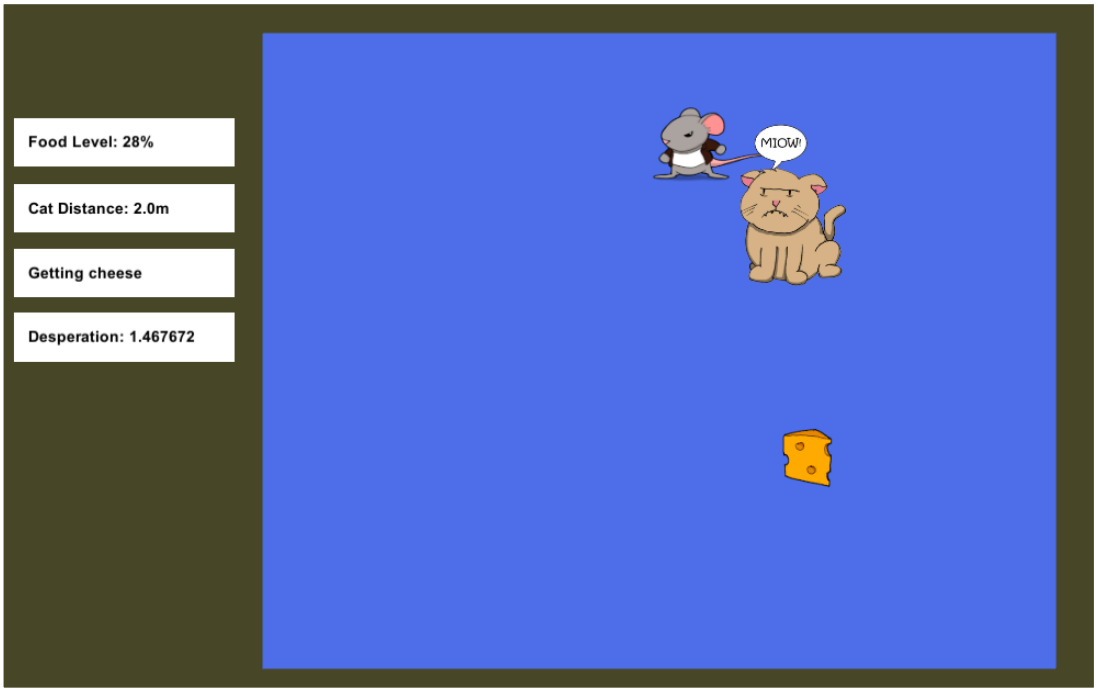
Examples are provided and can be found by loading the scenes in this location:

13.1. Cat and Mouse

The cat and mouse demo shows several behaviour trees working together to create a complex behaviour. You can load the example scene from the following location:



When you load the scene and press play, you will see a cat and a mouse interacting on the screen.



There are four boxes down the left showing some internal details. In order, from top to bottom, these are:

- The mouse's food level which reduces over time making him hungry.
- The mouse's distance from the cat
- The mouse's chosen behaviour
- A number representing the mouse's desperation level

There are also four behaviour trees working in the scene.

Miaow tree

There is a simple timed sequence attached to the cat which simply makes the cat say miow at random intervals by playing an animation.

Cat tree

There is a second tree attached to the cat which makes it wander around at random. It also reacts to being hit by the mouse.

Arena tree

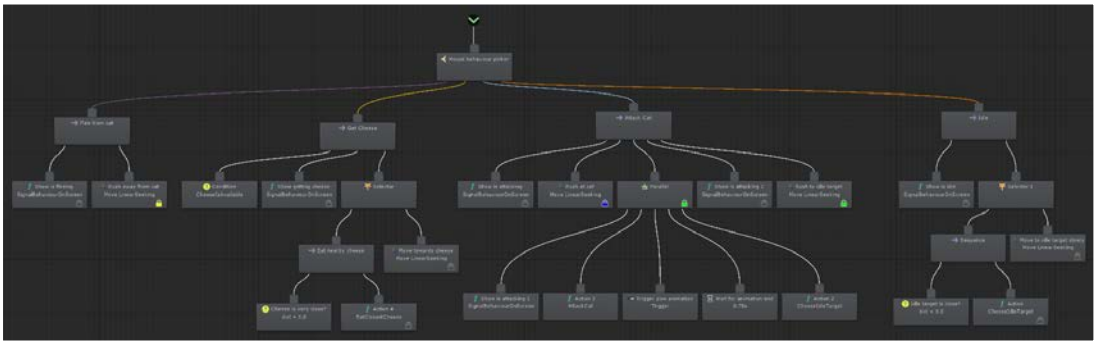
There is a tree attached to the blue background which randomly spawns pieces of cheese into the scene.

Mouse tree

This is the most complex behaviour tree. The mouse uses a utility node to determine the best course of action at any given moment.

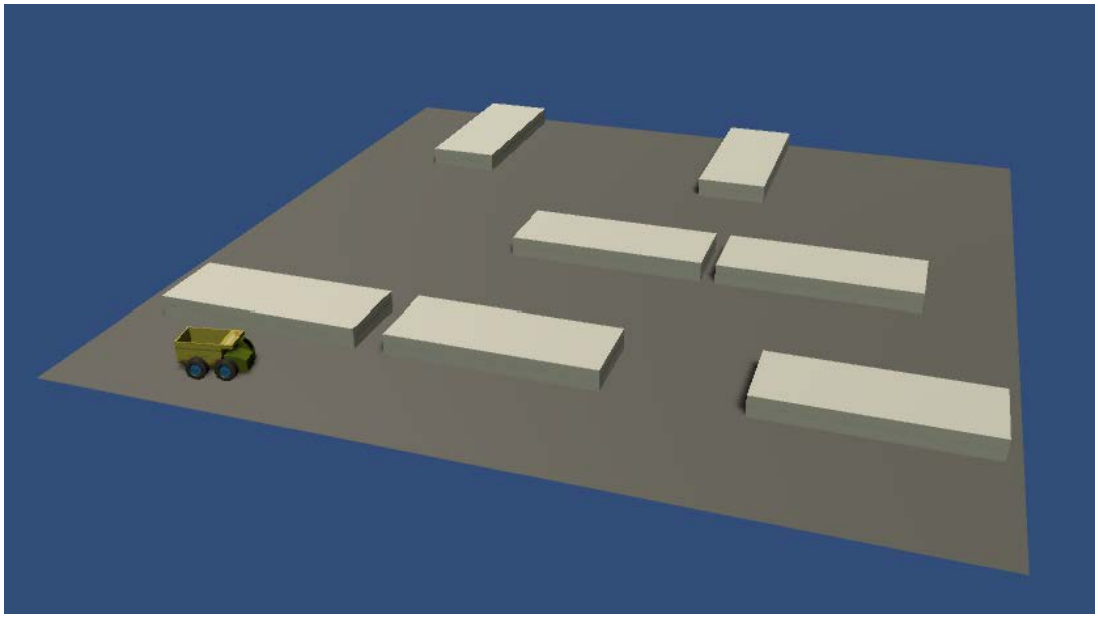
It will either:

- Flee from the cat if the cat gets too close
- Seek out the closest piece of cheese if it gets too hungry
- Attack the cat if it feels desperate (Hungry and the cat is too close)
- Just wander idly if it is otherwise perfectly satisfied.



13.2. Dumper Truck

The dumper truck demo is a simple demo showcasing a truck driving between predetermined points in an area at random, navigating using a NavMesh.



The truck picks a random point using the random and set property nodes. It then moves to the location, plays an animation and the sequence begins again.

14. Performance

This section contains information on the performance of the library, and tips on how to keep performance high.

14.1. Latches and dynamic nodes

Creating a behaviour tree is a balance between being able to have your character react immediately in all situations (No latching) and narrowing the scope for re-evaluation of the tree in order to keep performance high.

Look up the topics of latches and dynamic sequence nodes for more information on how to keep performance high.

14.2. Garbage Collection

InstinctAI has been designed to allocate no memory after initialisation of a behaviour tree.

If you are seeing garbage being created in the Unity profiler, please first refer to the Unity documentation on garbage collection. This outlines a number of (some quite surprising) places where memory is allocated.

If you are sure that none of these solutions would help, then feel free to contact us and we'll help to try to get to the bottom of the issue.

15. Troubleshooting

15.1. Common Problems

Here are some common problems, helpful information and possible remedies.

15.1.1. Generated code does not compile

Sometimes you may find that Unity complains about code errors in generated code. This usually happens when another part of the project changes and renders the generated code incorrect.

An example of this might be a change in a property name that is referenced from a behaviour tree node.

In this case there is a catch-22 situation because the node can't update its list of available properties until the code error is fixed, and you can't fix the code error until you edit the node.

In these situations, it's recommended to delete the generated portion of the class file. This will allow the editor to catch up with code changes and you should be able to simply alter the node and regenerate.

> *Note*
> An update to the asset which will make this situation easier to work with is planned.

15.1.2. Live update does not work

Live update will be disabled if the code is out of date, so the most common problem is forgetting to hit the regenerate code button.

Another common issue is that live update assumes you have one tree per object. Multiple trees are perfectly acceptable, but it does not play well with live update at the present time.

> *Note*
> An update to the asset that addresses this issue is planned for a future update.

15.1.3. There is no full transform node

If you want to apply full transform changes to an object, then at the moment the only solution is to place move, rotate and scale nodes under a parallel node. There are no plans to change this, but if it is an issue please contact us and we can prioritise it as an issue.

15.1.4. Random values don't seem random

Remember that your node that generates a random value may be executed on every frame. If you are seeing this as a problem, then you may need to reconsider how your tree is structured.

15.1.5. Can't see the Utility editor

If you've selected a utility node but can't see the editor, it may be that resizing the editor panels may have resulted in the utility panel being minimized to the bottom of the window.

Look for the title bar of the utility editor at the bottom of the tree editor and drag it upwards.

This is a known issue.

15.1.6. My Method or Property is not being listed

There are a number of possibilities for this. These are the most common.

Firstly, check that the return type of your method or property matches that expected by the node.

Also check that the parameter types are supported. Try deleting the method parameters to see if any are preventing the method from being listed.

Finally, any compiler errors will prevent the method lists from being updated. Resolve these first before setting properties in your nodes. If the compiler error is in a generated tree file, it is save to delete it, set the node properties and then regenerate the code again.

15.2. Other Known Issues

There are a number of known issues with the asset at the time of writing. A current list of known issues is available on request. If you are affected by any problems, please check that you have an up-to-date version as your problem may already have been fixed. Alternatively, send an email to us; user feedback helps us to prioritise fixes and updates. See the feedback section for details.

16. Feedback

All feedback is welcome, including feature requests. Please allow 24 hours for email response. We try to respond very quickly, but bear in mind that we may not be in your time zone.

If you have problems with the asset, then please attempt to contact us first before leaving any feedback on the Unity asset store. We are happy to resolve user issues, and your issue may be very quick to resolve.

To contact us about support or anything at all, simply email us at:

support@kupio.com

Thank you ☺

17. Appendix I – Hints and Tips

Use duplicate to create nodes with the right settings

If you don't like the defaults for things like latches, then a good tip is to create a node with the right settings and place it on the tree canvas. Duplicate this node when you need to create one of the same type instead of creating one from the menu, and you'll always default to the right latch.

18. Appendix II - Changelog

1.0

Initial release

19. Index