BehaviourTree

A Behaviour Tree is a behavioural decision model that can be used to set the behaviour of any object that can mount a Behaviour Tree Entity component, such as a monster or a custom NPC.
The tree structure used for the behaviour tree is consistent with the way humans think and allows for more intuitive behavioural design. Modularity also helps developers to make modifications and adjustments.
In this tutorial, you will be taught how behaviour trees work and how to create one. In the example, it will be shown how to create a behaviour tree that can be used by zombies that patrol and chase the player.

Behavioural tree composition and operational logic

The behavioural tree consists of logical and leaf nodes and a special root node.

The Behaviour Tree traverses all the nodes in that Behaviour Tree that can be executed in each Tick (one or more frames, which can be modified in the properties of that Behaviour Tree) in the defined order, and finally stops at a node.

The behaviour tree will start from the root node and will be executed in left-to-right, top-to-bottom order until it stops at the stop node or until it traverses all connected behaviour tree nodes.

> Behaviour trees are always fixed to execute from the root node, so only nodes directly or indirectly connected to the root node will be executed. The order of node execution is marked in the editor with a numerical sequence number; nodes that will not be executed have no sequence number.
>
> For each Tick, when the behaviour tree stops at a node, none of the nodes after that node in the execution order will be executed.

Nodes of the behavioural tree

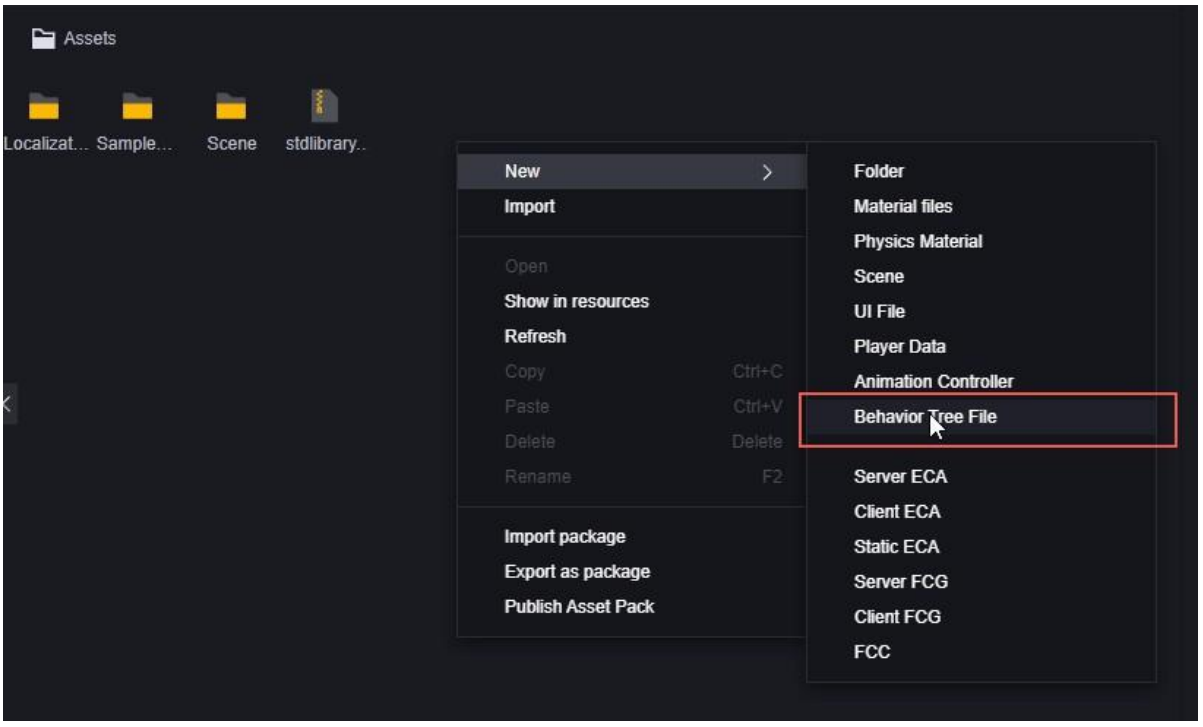
Root node: the initial node that every tree has, all logic starts running from the root node. The root node is created automatically and cannot be deleted.

Logic nodes: nodes classified as Composite, Decorator. Logic nodes determine the operating logic of child nodes. Logic nodes cannot be used as the last node of a branch.

Leaf nodes: nodes classified as Condition, Action. Leaf nodes determine the actual behaviour of the entity. A leaf node must be used as the last node of a branch.

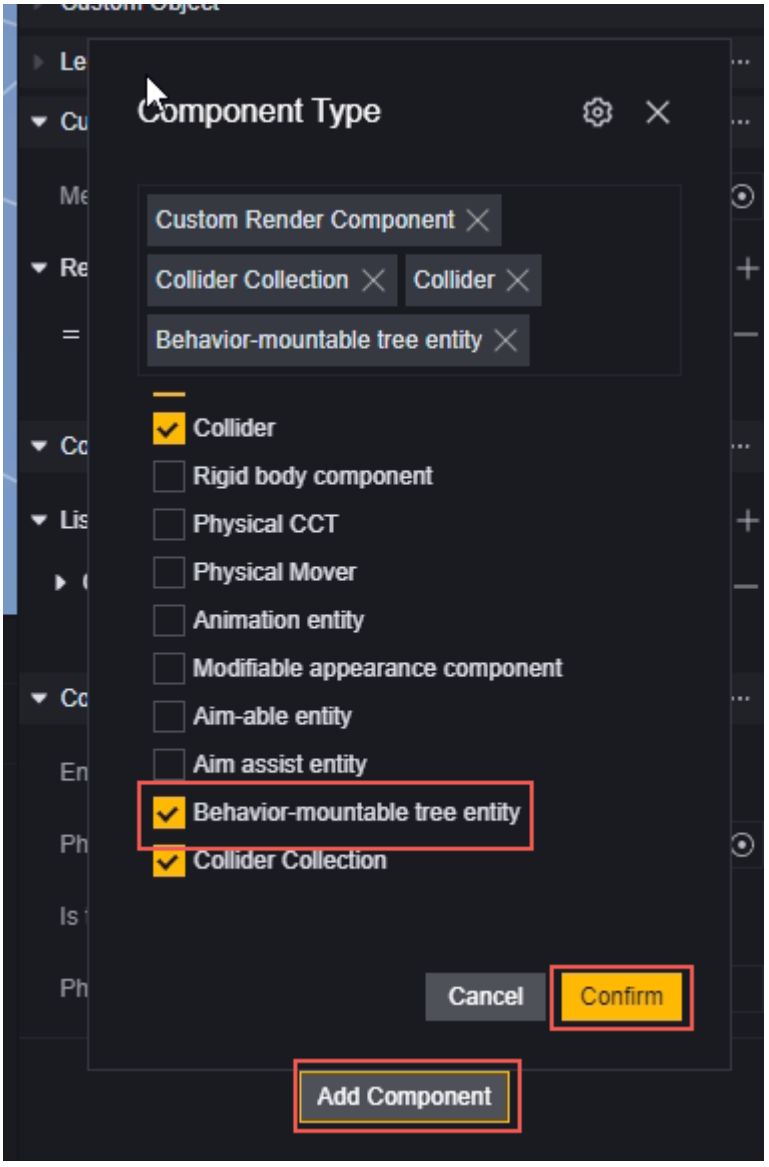
Creating a Behaviour Tree

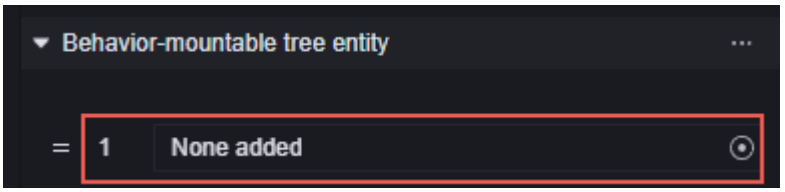
Behaviour tree files can be created by right-clicking in the Assets window. You can also manage behaviour tree files in the Assets window. Behaviour tree files have a .xbt extension.



Mounting a behavioural tree

Before you can mount a behaviour tree for any entity, you must first add a "Mountable Behaviour Tree Entity" component to it in the Inspector panel. Once the Mountable Behaviour Tree component has been added, the behaviour tree file can be quickly mounted.





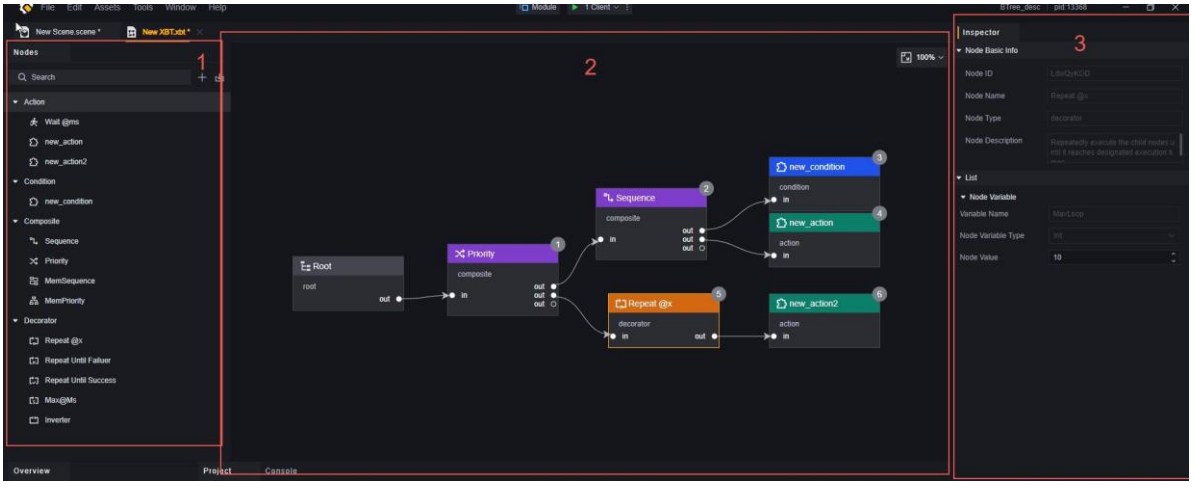
Only one behaviour tree can be mounted per entity.

Editing the behavioural tree

Opening a behaviour tree file opens the Edit Behaviour Tree panel.

The Behaviour Tree editing panel is divided into three sections:

1. Node List
2. canvas (artist's painting surface)
3. Inspector



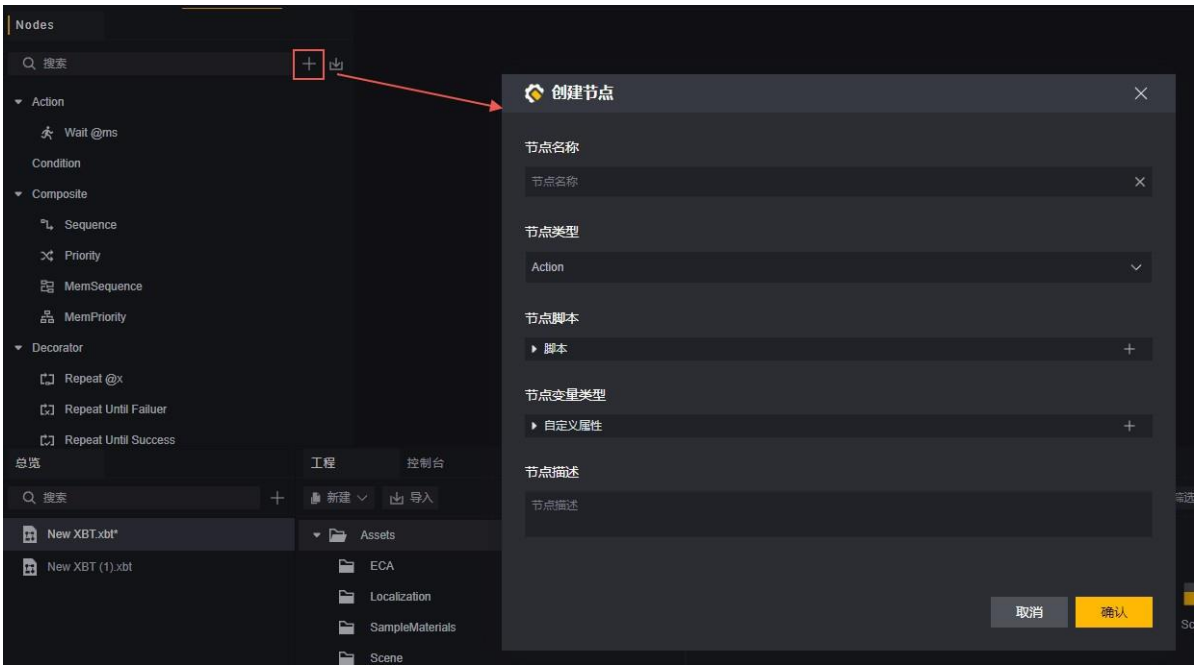
node list

All officially supplied, customised and imported nodes are displayed here. Drag and drop nodes into the canvas to create a brand new corresponding node.

Supports node search.

The buttons in the upper right corner allow you to create custom nodes or import nodes.

Custom nodes



Click "+" in the upper-right corner to open the Create Node panel. In the Create Node panel, you can customise the following properties of the node:

1. Node Name.
2. Node types, currently only nodes of custom Action and Condition types are supported.
3. Node Scripts, currently the Create Node panel only supports adding existing scripts to a custom node and editing them after creation.
4. A node variable type for which custom properties can be created for the node to be called by the script.
5. Node Description.

All of the above support modification after creation, except for the 2. node type.

import node

Clicking the Import button in the upper right corner
opens the Import File Selection screen. You can import
nodes from other projects with the node
suffix .xbttemp.

It should be noted when importing that imported nodes may have scripts that do not exist for this project.

> Important! Changes made to nodes in the node list will not act on nodes that have been dragged into the canvas.

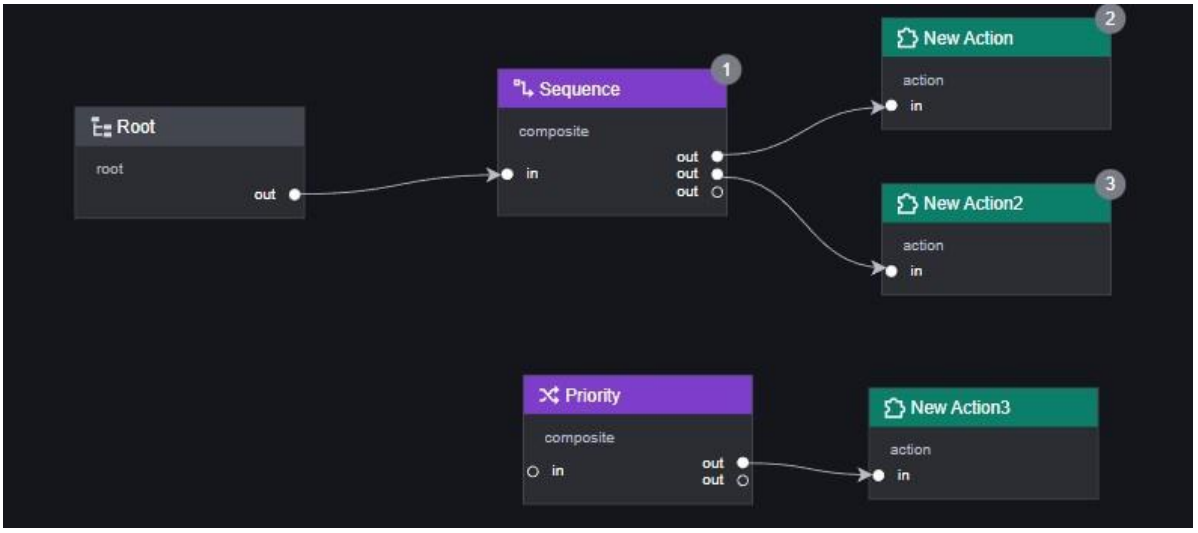
canvas (artist's painting surface)

The canvas is the main area for editing the behaviour tree. After dragging nodes from the node list into the canvas, you

can arbitrarily arrange the behaviour tree nodes and pass the
The "out" to "in" method connects any two nodes with a connecting line, thus editing their execution order. The canvas will always contain a fixed root node.

When connecting from the "out" tab of one node to the "in" tab of another node, we call the node to which "out" is connected the parent (output node), and the node to which "in" is connected the child (input node). The node to which "in" is connected is the child node (input node).
Depending on the type of node, a node can have one or more "out", but it has and can only have one "in". (Except for the root node, which has no corresponding input node.)

The behaviour tree will only execute nodes that are directly or indirectly connected to the root node, called activated nodes. Activated nodes are automatically identified in the canvas and ordered from left to right, top to bottom, with the grey number in the top right corner of the node being the order in which they are called, and no numbers are shown for nodes that are not activated.



> Panels can be dragged by holding down the left or middle mouse button. Zoom the panel by Ctrl+mouse wheel.

Node View Panel

Each node in the canvas contains the following information or some of it: node basic

information, script, and list of node variables. Node basic information: includes node ID, node

name, node type, node description.

Script: the script mounted by this behaviour tree node, a behaviour tree node can only mount one script. Scripts control the behaviour tree to run to that node.

The behaviour that needs to be performed at the point.

List of node variables: a list of some of the variables required for the node to run, partly officially provided and partly added by the creator when creating the node.

> The node ID is automatically assigned in the basic information of the node, and other information can be modified in full in the node list on the left, but the node dragged into the canvas can only modify the node name.
>
> Scripts can be added when creating a node or edited in the canvas, but only one script can be mounted on a node.
>
> The list of node variables can be set at creation time or in the canvas.
> Always note that the nodes in the variable list and the nodes in the canvas are not equal, and the data used for the specific behavioural tree must be based on the nodes on the canvas.

Behavioural tree node types

There are 4 types of Behaviour Tree nodes: Action, Condition, Composite, Decorator.

The nodes classified as Composite and Decorator are logical nodes and cannot be used as the last node on a branch.

Action, Condition are leaf nodes and can only be used as the last node on a branch.

Action

Nodes of type Action represent "behaviours" that need to be performed by the behaviour tree, except for the official node "Wait @ms", which is created and defined by the creator.Actions are often used to make entities of the behaviour tree, or entities belonging to it, perform specific behaviour. Action nodes can be used to create modular custom behaviours, such as walking to a specific place, releasing a specific skill, or activating a specific effect.

There are only two types of execution results for Action nodes: Complete and Running.

- Wait @ms: This node contains a variable "WaitTime", which is used to wait for a specific amount of time (in milliseconds, the specific wait time is the value of the "WaitTime" variable) during the execution of the behaviour tree. The node will always return "Running" if the wait time has not expired, and will return "Finished" after the wait time node.

> Note: Since the behaviour tree is executed once per tick, using the "Wait" tuple in the behaviour tree node may cause the behaviour tree to run stuck by repeating the wait at every tick. It is therefore recommended to use the Wait @ms node instead of the "wait" behaviour in the behaviour tree script.

Condition

Nodes of type Condition represent conditional judgements that need to be made by the behaviour tree. All Conditon nodes are created and defined by the creator. Condition nodes are often used to determine whether a particular condition holds.

The Condition node is executed with one and only two results: true (True) and false (False).

Composite

There are only four nodes of Composite type and they are all official nodes. This type of node is mainly used to control the execution order of child nodes.

- Priority: implements a child node or relationship, traverses the child nodes in the defined order, stops until one child node returns true, and returns true. returns false if all child nodes return false. does not support storing the results of a single execution until the next execution.
- Sequence: implements child nodes and relationships, traverses child nodes in defined order, stops until one node returns false, and returns false. returns true if all child nodes return true. does not support storing the results of a single execution until the next execution.
- MemPriority: same as Priority, but supports storing the result of a single execution until the next execution.
- MemSequence: same as Sequence, but supports storing the result of a single execution until the next execution.

> Note: The function of Composite to control the execution order of child nodes is only effective for all "child nodes" connected to the Composite node.
>
> Customisation of Composite nodes is not supported.

Decorator

Nodes of type Decorator are mainly used to control the execution logic of the behaviour tree. There are currently only five official Decorator nodes.

- Repeat @X: this node contains a variable "MaxLoop", the main function is to let the directly connected child nodes repeat the execution of "MaxLoop" times, and return the result of the last execution.
- Repeat Until Failure: This node allows directly connected children to repeat until the result is "Failure".
- Repeat Until Success: This node allows directly connected child nodes to repeat until the return result is "Success", i.e. "true".
- Max @Ms: This node contains a variable "MaxTime". When executing a directly connected child node, this node determines if it has timed out, and returns "False" if it has.
- Inverter: This node inverts the result of the execution of its directly connected children. For example, inverting "False" to "True". Customisation of the Decorator node is not supported.

Behavioural tree node scripting ground rules

Each editable node in the behaviour tree supports the addition of up to one script.

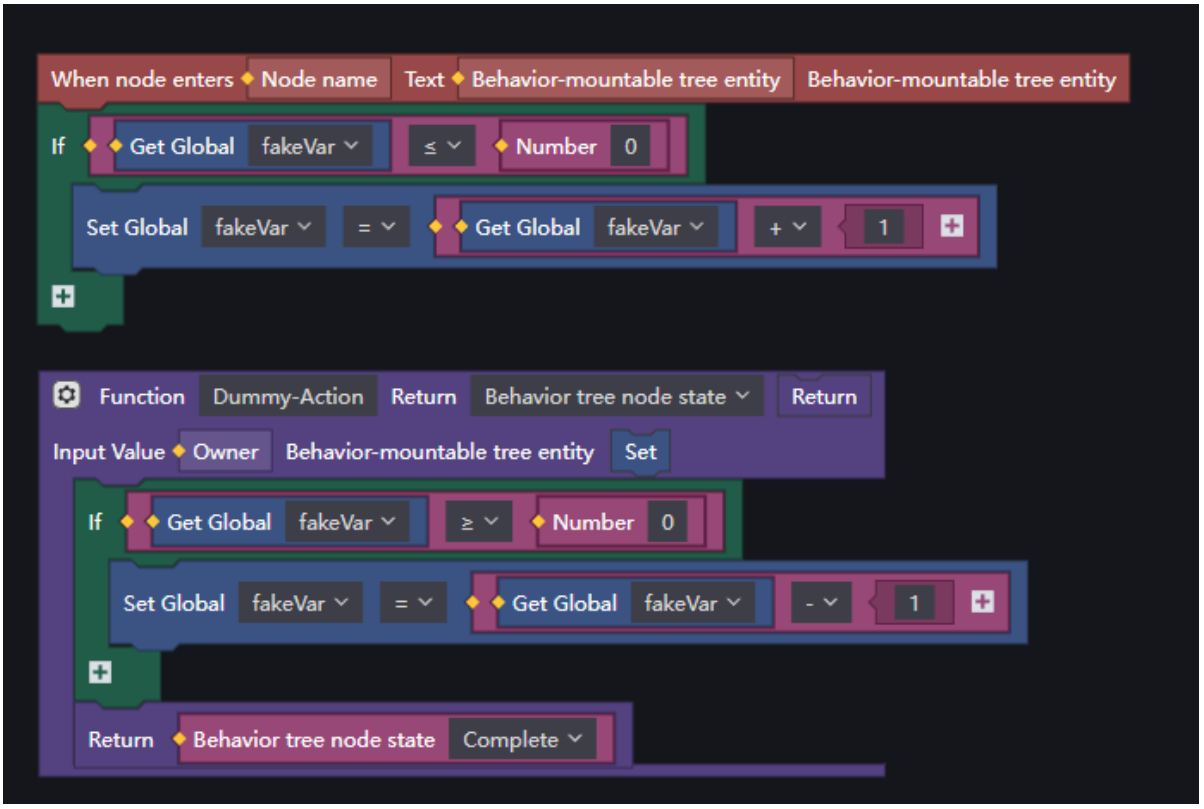
Editable nodes are all of type Action or Condition, and depending on the type, the node script must return the specified return value.

The script for the Condition node must return a Bool value.

Scripts for Action nodes must return a value of type "Action Tree Node Runtime Status", i.e. "Complete" or "In Progress".

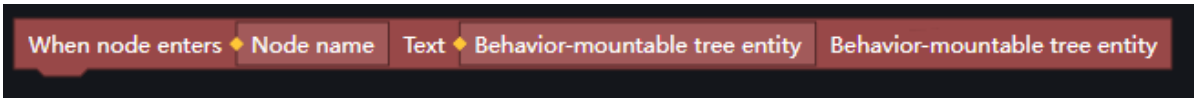
> Assuming that multiple return values are returned to the behaviour tree in the script, the behaviour tree will by default run with the first returned return value for that node.
>
> The officially provided base nodes are already preconfigured with return values, see Node Types for details.
>
> Logical nodes make logical judgements based on leaf node return values, see also node types for details.

The node needs to carry a special function with the type of input and output parameters that must be specified for the behaviour tree to be called. The requirements for special functions are explained below.



It is recommended to use the "when node enters" event for initialisation and to write the node logic into special functions.

This allows each tick to execute the special function directly when the node returns an "in progress" result, without having to repeat the initialisation.

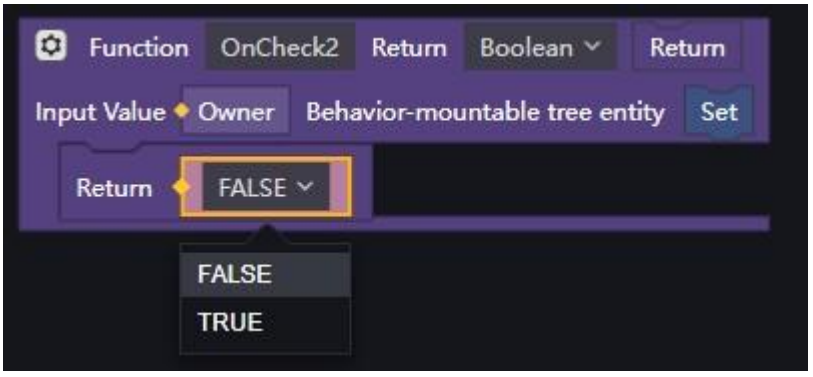


How to return the results of a run in a node

The behaviour tree makes return value determinations by reading the value of a special function of the type that there is, and can only be, one in a script at the same node.

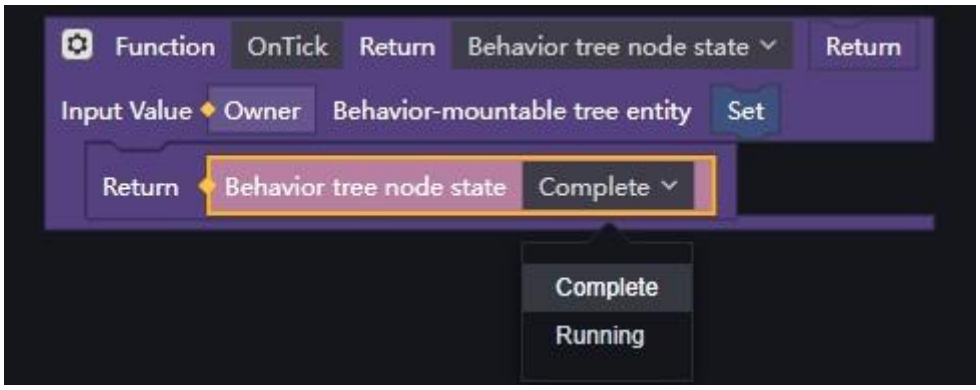
This function is written differently in Condition scripts and Action scripts.

The write-up under the Condition node is shown in Fig:



The input parameter is: Mountable Behaviour Tree Entity (Owner, i.e. the entity whose corresponding behaviour tree file we have mounted).The return value is: Bool (the result of running this node).

The write-up under the Action node is shown in the figure:



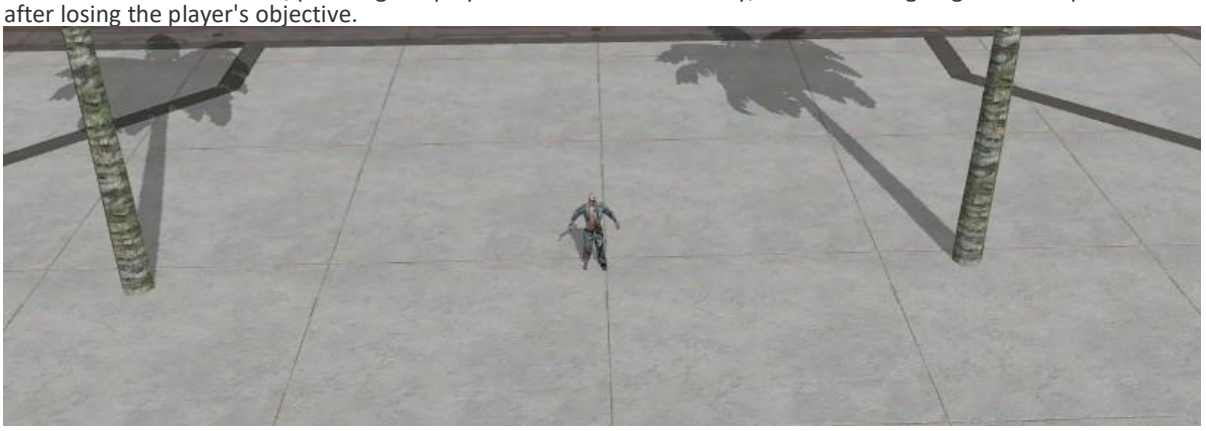
The input parameter is: the mountable behaviour tree entity (Owner, i.e. the entity whose corresponding behaviour tree file we have mounted). The return value is: the state of the behaviour tree node (the result of the node's operation).

> The behaviour tree automatically reads the node script for functions of that type (you need to make sure that there is and is only one function of the same format), and in every
>
> Tick calls this function to get a return value. There is no need to call that particular type of function separately. Each time a behaviour tree node is run, each event is triggered in the following order: when the node enters, when a special function is called, and when the node exits. Only one such special function can exist per script; do not design a function with the same type of return value as its input value. You can program other meta-logic you wish to execute within the function, just make sure that the function eventually returns a return value of the corresponding type.
> Non-special format function can have unlimited number of scripts, will not affect the normal operation of the behaviour tree and logical judgement. In this special function, temporarily does not support the call asynchronous function, must be when the tick immediately return results.

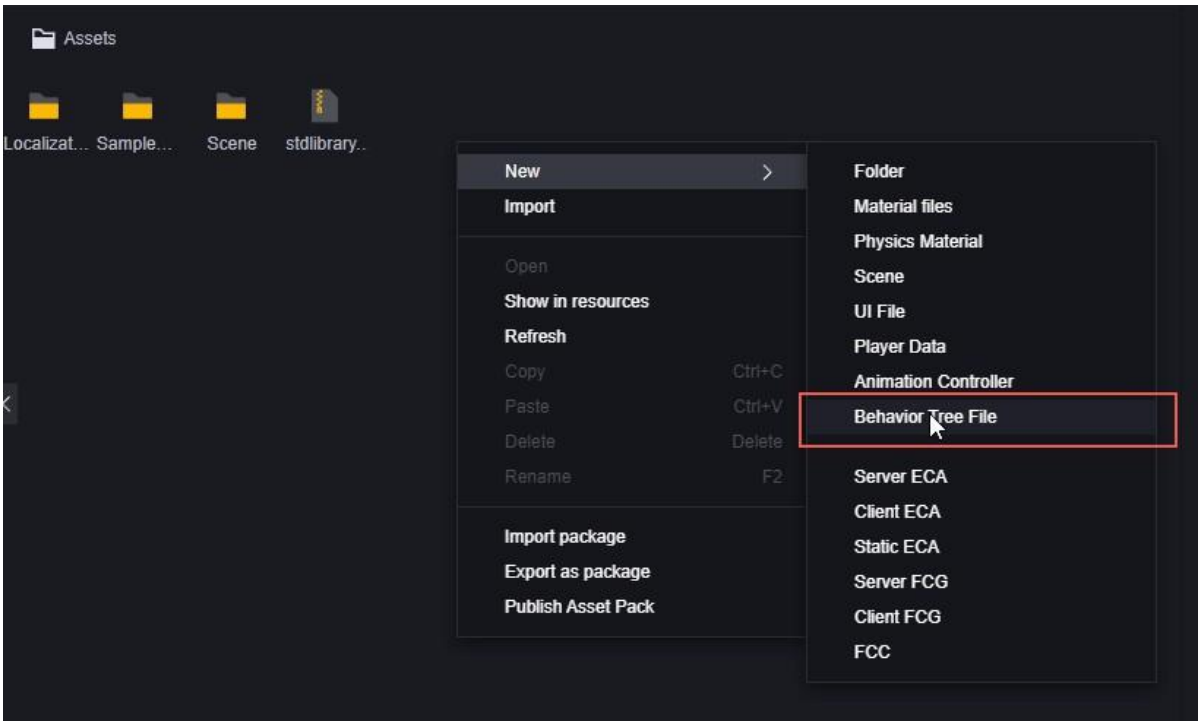
typical example

As an example of a simple requirement, a behaviour tree is created and applied. The actual writing of the script will not appear in this example.

Assume a demand:
On the scene: one zombie, two coconut trees. Use the Behaviour Tree to have the zombie constantly patrolling between the coconut trees, pursuing the player when one exists nearby, and continuing to go back on patrol after losing the player's objective.



First, create a behavioural tree file and edit it:



Analysing the needs, the possible actions of the zombies are:

1. patrol (police, army or navy)
2. Pursuit of players
3. Return to patrol paths

The conditions for zombies to change their actions are:

1. Found the player.
2. Lost the player.
3. Found himself out of the patrol path.
4. Found myself in the patrol path

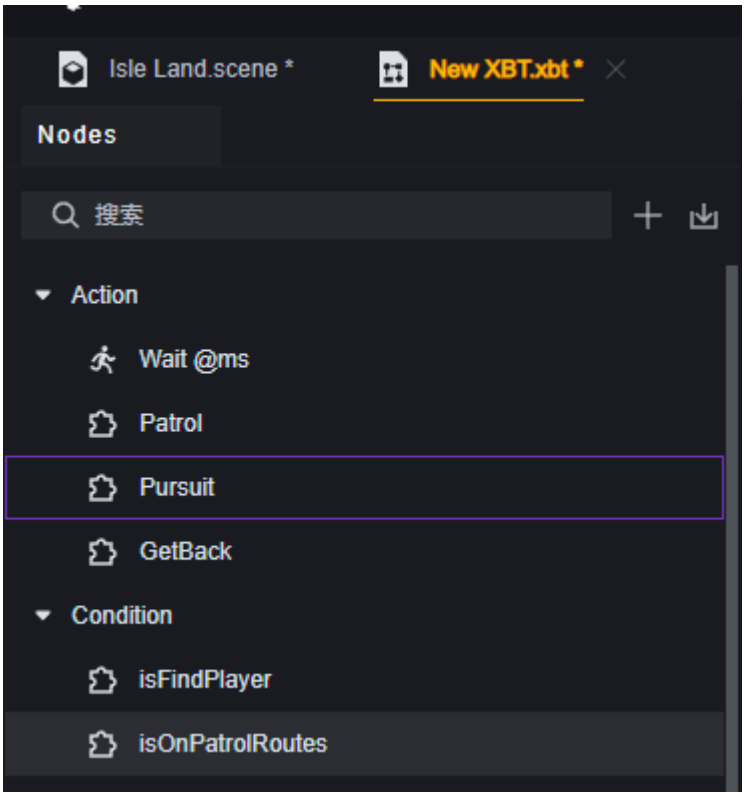
Where 1 and 2 and 3 and 4 are both positive and negative of a condition. Therefore only two Condition nodes need to be created. Create all custom nodes:

Action node:

1. patrol (police, army or navy)
2. Pursuit of players
3. Return to patrol paths

Condition node:

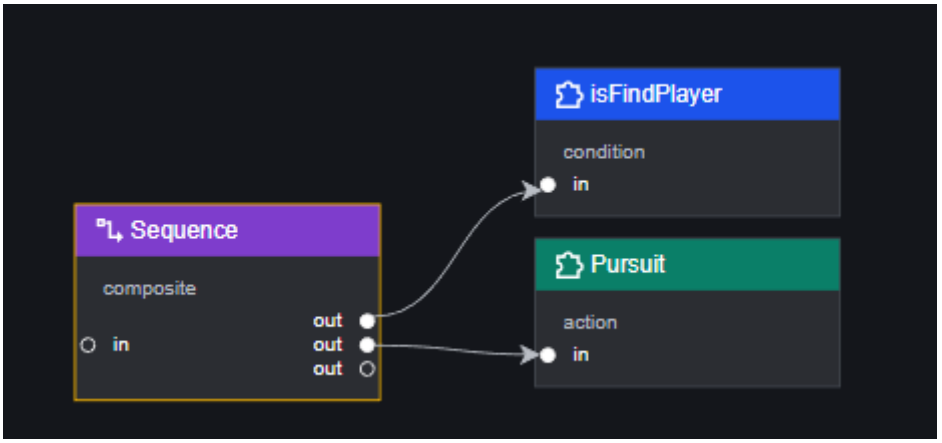
1. Whether or not the player is found
2. Whether or not in the path of a patrol



Analyse the conditions of the Action node:

Action node	**Implementation condition 1**	**Implementation of condition 2**
patrol (police, army or navy)	The zombie didn't spot the player.	On patrol paths
Pursuit of players	Zombie Spotting Players	
Return to patrol paths	The zombie didn't spot the player.	Out of patrol path

Take out the behaviour that requires the fewest conditions: chasing the player, and group it with its condition nodes using Sequence nodes:

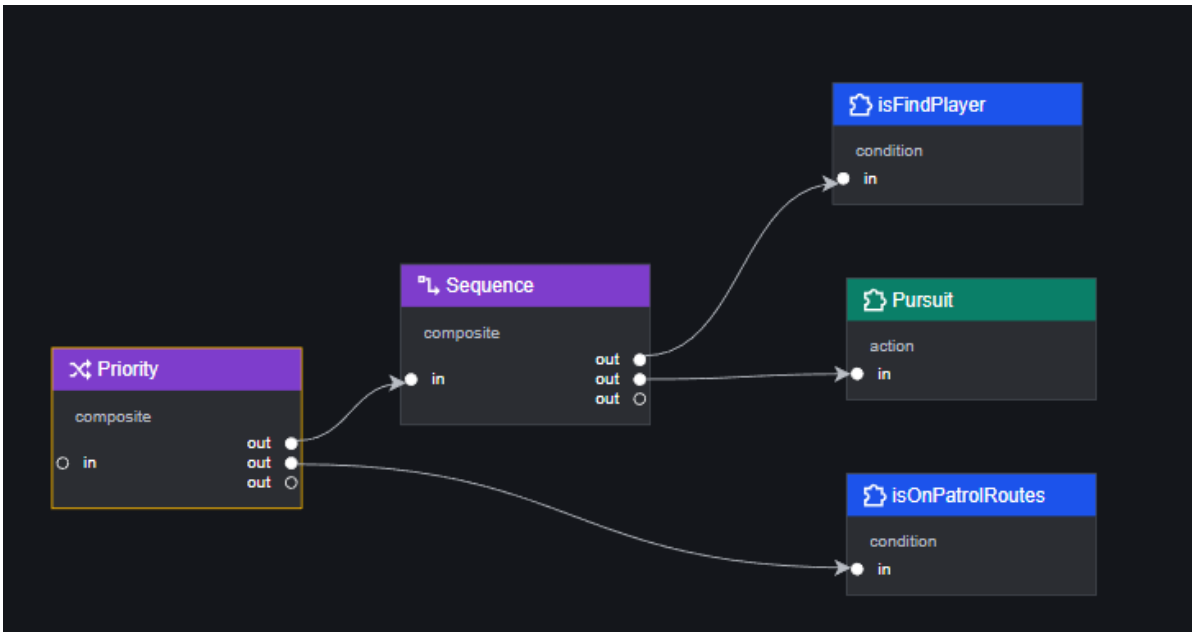


> Sequence node: the child nodes are executed in order until the child nodes return false, then stop and return false; if all child nodes return true, then return true.

In this group, the pursuit behaviour must be executed after the whether-player-node-is-found returns true.

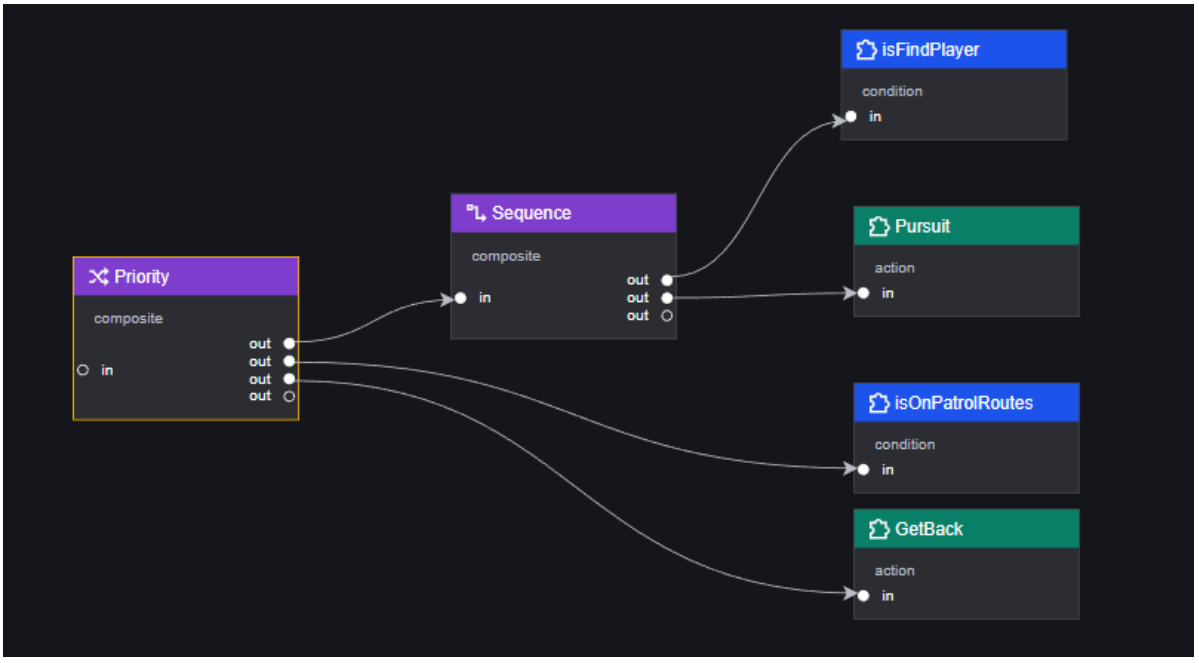
If the zombie doesn't find the player, the actual behaviour performed by the zombie is to be decided based on the result of another condition: whether it is on the patrol path or not, so we need a logical node to connect the previous group with the new condition.
Again, since this is dealing with the logic that the zombie didn't find the player at this point, the Sequence node in the previous group must have a return value of false, so our new logic node needs to be one that will continue to run even if it receives a false.

It is the Priority node:

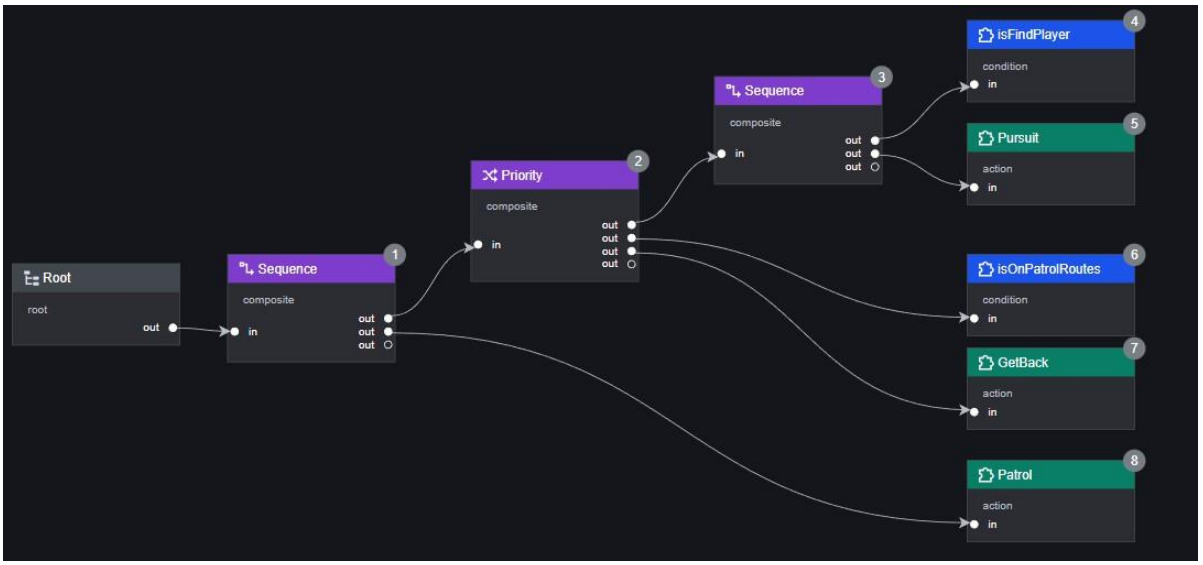


> Priority node: the child nodes are executed in order until the child nodes return true and stop, and return true; if all child nodes return false, then return false.

This group makes a judgement on whether the zombie is on the patrol path or not, provided that the zombie has not found the player. Based on the characteristics of the Priority node, we can directly connect the behaviour of the zombie when the "Is on patrol path" node returns false: return to patrol path



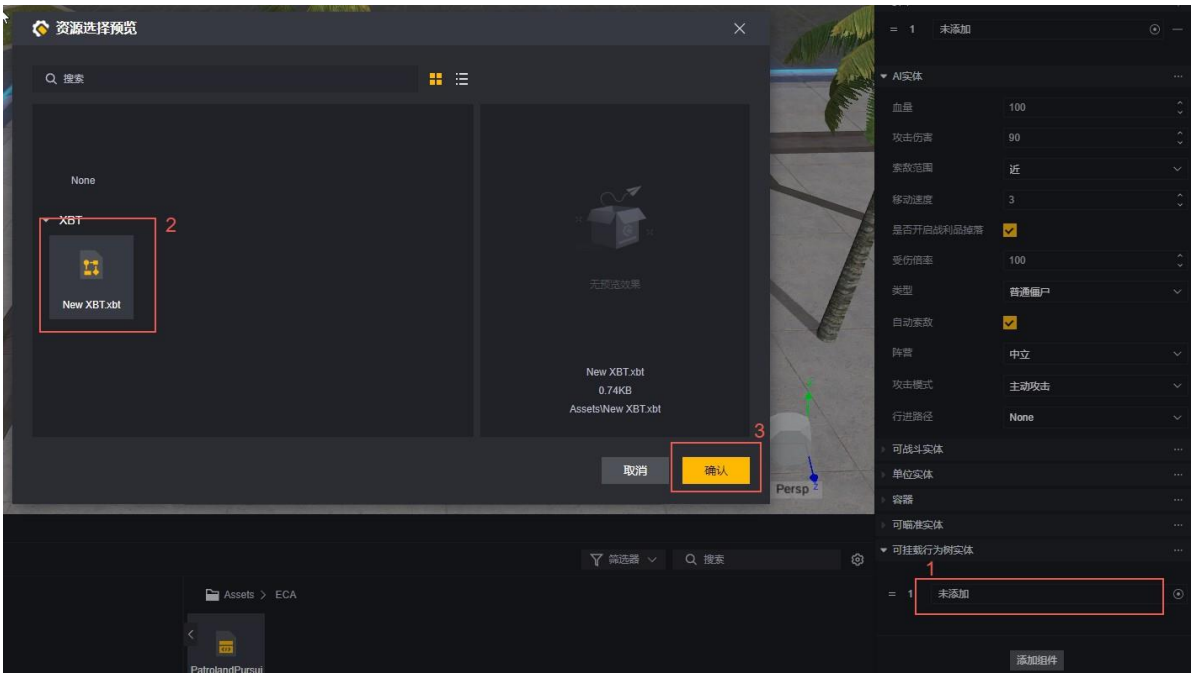
Then when the "Is on patrol path" node returns true, the Priority node terminates because it receives a true and returns a true value. We need a logical node that will continue to perform the last behaviour of the zombie when its child nodes return true: patrolling.



In this way, a behavioural tree is created.

Next, we need to mount this behaviour tree on the zombie entity: select the zombie and add the "Mountable Behaviour Tree Entity" component to it

Add to it the behavioural tree you just edited:



be highly successful

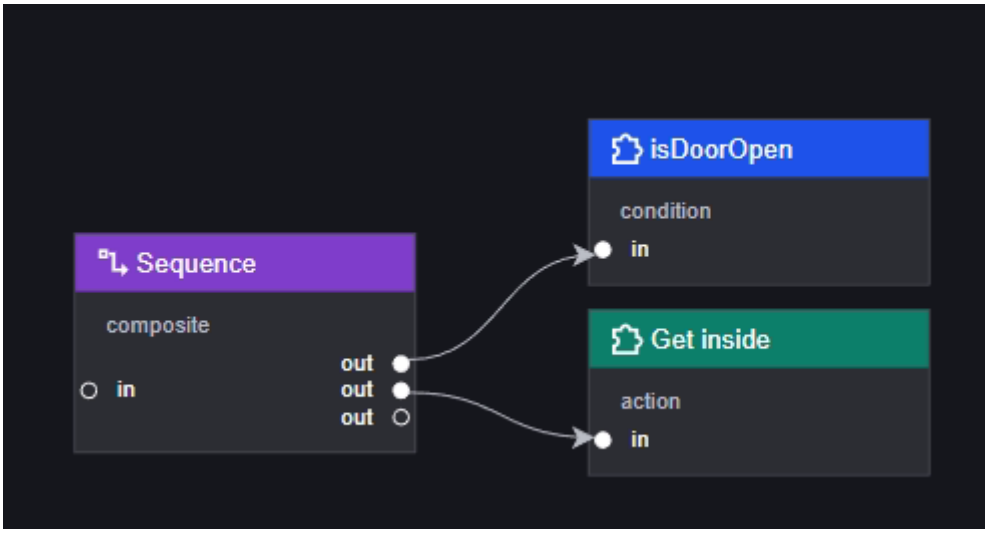
> The examples are intended as a demonstration of one method of production and are for reference only.
> The same logic can be thought of differently to draw the canvas of the behavioural tree, and you can follow your own preferences in how you approach it.
> It is also possible to continue editing the behaviour tree after adding it to an entity, and it is not necessary to edit it in the same process as in the example.

replenish

pre-conditions

When behaviour X must be executed after satisfying condition A, at which point we call condition A a precondition for behaviour X. Logical units can be constructed using Sequence nodes.

Preconditions are used to satisfy if-then like logic requirements, such as the door must be open to enter the house:

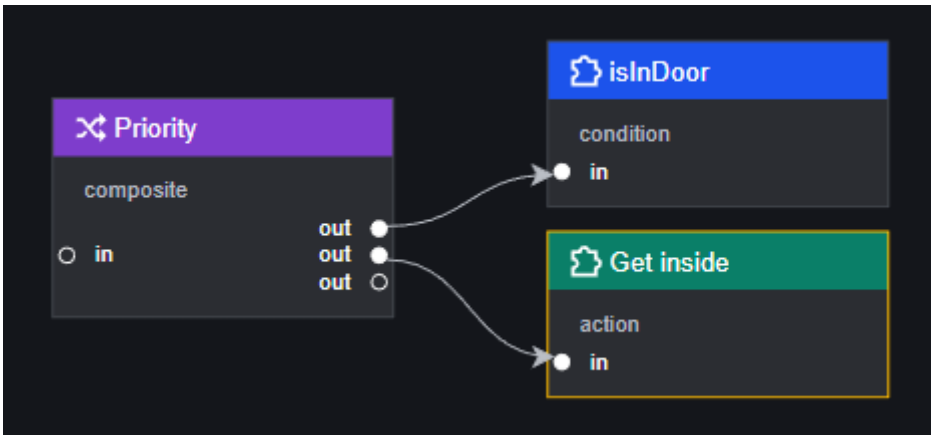


postcondition

When the behaviour X has been executed, condition B must hold, at which point condition B is said to be a postcondition of behaviour X. Logical units can be constructed using Priority nodes.

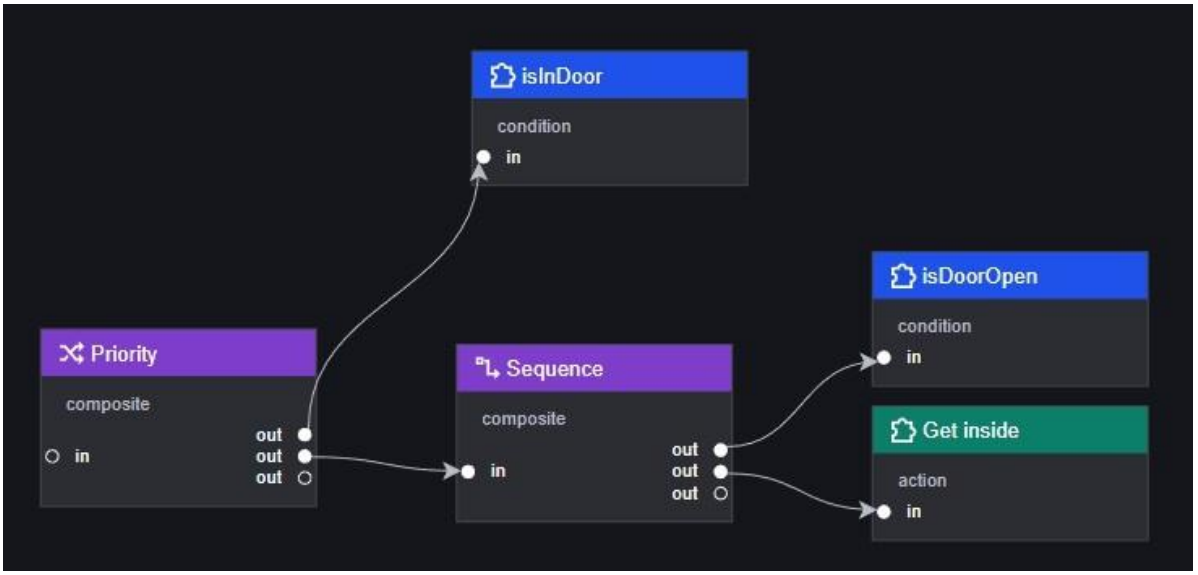
Postconditions are used to ensure that the condition must be satisfied if the logic block as a whole executes successfully. For example if the enter house action is executed successfully, the unit must

In the house:

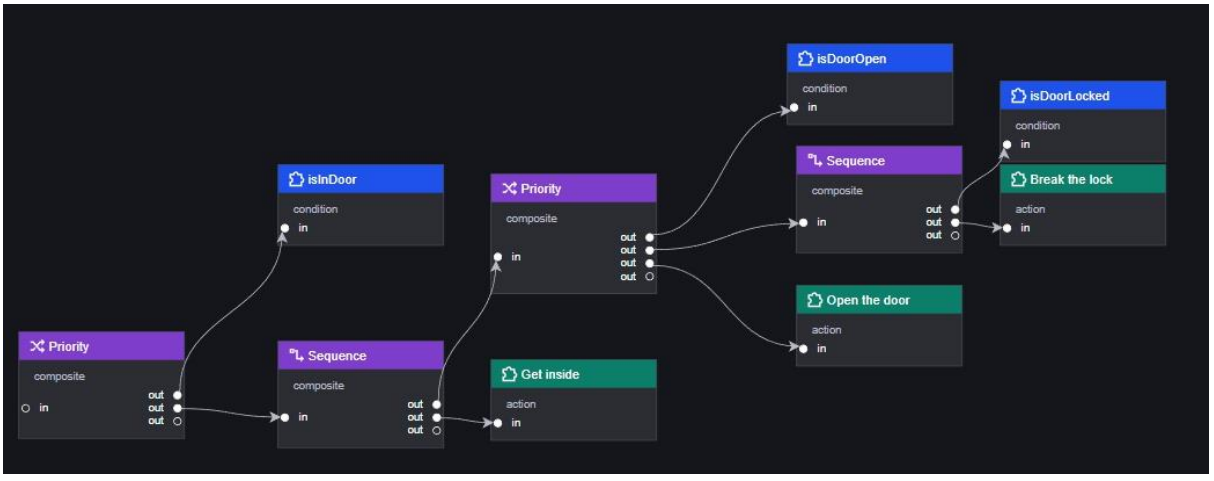


PPA model (Postcondition-Precondition-Action)

Combining the two conditions above allows us to use a widely used design approach, the PPA model.



The advantage of this pattern is that the whole behavioural tree can be considered as logic in the context of postconditions, which better supports hierarchical design, e.g. a door may have a locked state:



The new logic will mainly affect whether the door opens or not, and it is sufficient to edit the extended logic by making the door open as a post-condition.