How To Use BT Builder

Adding Behavior Component

You can add a behavior to a GameObject or prefab just like any other component. The BT_Behave component, which manages the behavior execution, is added automatically, if there isn't already one. You can add several behaviors to the same GameObject.

Component menu

You can add a behavior to currently chosen game object from the Component menu (Component-> Behavior AI -> Behavior Tree).

Inspector

You can also use the Add Component -button in the inspector. NOTE: Project must be compiled first. If the Behavior AI option is not visible, compile project (simply entering and exiting play mode does this).

Assigning a Behavior File

Behaviors made with the BT Builder are saved as xml files. To run a behavior, you have to assign a behavior file to the tree component. This can be done in several ways.

Edit Tree File

The tree component has an Edit tree file -button, which opens the behavior editor. You can either open an existing behavior file, or make a new behavior from scratch. When the behavior is saved, it is also assigned to the tree component.

This is also useful for quickly editing values that are set in the editor. NOTE: Values set in editor are treated like constants and saved in the tree file itself. Changing values from one gameobject will change those values for all objects that use the same file. If you want to use the same behavior structure for different gameobjects with different value settings, you have to use different save files, or set values through the Blackboard, which is unique for each BT_Behave component.

The behavior file is only used for creating and initialising the actual behavior instance, so editing the file during play mode has no effect. Edit Tree File -button is disabled during play mode.

Choose file from folder

You can assign a behavior file to the component like any other asset. Clicking the circle to the right of the object field opens the object picker, where you can choose a behavior file.

You can also drag a file from the folder view and drop it to the file field.

Editor Window

Editor View

Creating a Behavior

When you open the editor, there is always a RootNode placed on the left. This is the starting point of the behavior. The RootNode can only have one child node.

Editing Behaviors

Priority Order

The execution order of each node's children is determined by their position in the editor. The highest child is always the first in order. To change the execution order, you can simply drag a node to a different position. To move a whole branch, you only need to move its root (the node where the branch starts).

Snap to Grid

Snap to Grid -button aligns the nodes in the tree to their current order. It only affects the nodes that are part of the tree, i.e. connected to the RootNode. It is especially useful, when you need to move large parts of the tree.

Snapping is not necessary for the running of the behavior, but it helps to keep the editor window ordered.

Node controls

Connecting Nodes

Nodes can be connected by clicking the arrow buttons on each side of the node. A left arrow can only be connected to a right arrow. Clicking the same node buttons again will break the connection.

There are some restrictions to the possible connections:

RootNode and decorator nodes can only have one child. Trying to add another child to these node types will fail.

Leaf nodes (actions and conditions) only have a left arrow, since they cannot have child nodes.

Connections cannot form a loop. Trying to make a connection that would cause a loop will fail.

All internal and decorator nodes must have at least one child.

Editing Variables

If a node class has variable properties of currently supported types, you can edit their values in the behavior editor. Clicking the Edit button on a node will open the editing dialog window.

Editable properties should be treated like constant values that will not be changed during gameplay, like limit values for conditions, timers etc., or initial values that can be referenced later.

All objects that use the same behavior file, will also have the same property values.

Expanding subtrees

If a node has children, it has an Expand button in the lower right hand corner (”E”). This is a toggle that shows or hides the child nodes and subtrees of the node.

When working with larger trees, it can be useful to hide parts of the tree.

Destroying a node with hidden subtrees or child nodes will also destroy all its hidden nodes.

Nodes with Notes

You can add notes to any node in the tree. Clicking the “i” button in the top left corner shows or hides the note field. You can keep the notes visible for any number or nodes. This can be very useful if you want to hide a big subtree, but want to keep in mind what it contains.

Notes are also shown as tooltip, when you hover over the node title.

When you save a behavior with imported subtrees in it (see Working With Subtrees), the name of the imported file will be added to the notes. You can later edit the notes, if you don’t want to keep it.

Notes are saved with the behavior.

Saving / Opening Behaviors

Working with Subtrees

You can also create and save behaviors in smaller parts. Any behavior that you have saved can be added to another behavior tree. To add a saved behavior to another behavior, check the option “import as subtree” and choose the file you want to import. This will keep the currently edited behavior open. The opened behavior will appear to the window as a single node, with the name of the file at the bottom. The subtree node can then be attached to the edited behavior like any other node, and expanded to show the full branch.

When the full tree is saved, the imported subtree will be treated like any other part of the tree. The subtree name is added to the node’s notes. The original subtree file is not changed.

Warnings

Tree Depth

Tree depth is the deepest level of nodes in the currently open behavior. Since the executing code is recursive, very deep trees can cause a stack overflow.

There is no hard limit for maximum tree depth,

Not Connected to Root

The editor saves the behavior recursively from the root node. Only nodes that are connected to the root will be saved in the tree file; any other nodes or subtrees in the window will simply be ignored.

Not Ending in Leaf

All branches of the behavior tree have to end in a leaf node, since only leaf nodes will define a return value. Calling an internal node or decorator that has no children will result in error.

Invalid Nodes

It is possible to open a tree file withs nodes that no longer exist in the project. Any nodes not found in the current node script folder are marked red. It is still possible to save the behavior for future editing, but trying to run it will result in error.

Blackboard

Accessing the Blackboard

You can access the blackboard from anywhere in the game by getting a reference to the BT_Behave component.

The BaseNode class has a reference to its owner's BT_Behave, which you can use from inside a node:

agent.blackBoard.SetInt(5, "someInt");

Or if you need to use custom type data, you can use the object type dictionary:

agent.blackBoard.Set<customtype>(value, "somevariable");

If the GameObject has several trees, they all use the same blackboard.

Typed Data

There are also typed versions of the containers. In the current version (1.1.2) these can be accessed from a BT_Blackboard instance in the BT_Behave component. They have get and set methods named by the data type (e.g. GetInt and SetInt for integers. Currently supported types are:

int

float

string

double

bool

Vector2

Vector3

Quaternion

Color

The benefit of typed containers is that they don’t need boxing. Especially on mobile platforms using a lot of variables boxed to object type can cause lag.

Tips

Best practise for using the blackboard is to process the data and update it from outside the behavior itself. For example if behavior needs to know the distance to the closest enemy, this can be calculated elsewhere and only the distance saved to the blackboard. Then the behavior can use it directly.

If you need the same kind of nodes to use different values for a variable, you can define the individual variable name by storing it in a string property, which can be modified in the editor.

Visual Debugger

Using Visual Debugger

You can follow the behavior execution during play mode by opening the visual debugger window from the tree component. The debugger shows the tree graph, with symbols of the return value of each node after traversal. Nodes that were not visited wil show a grey rectangle.

Each tree can have its own debugger window open at the same time, even on different game objects. The name of the tree file and the name of the owner object are displayed at the top of the debugger window. Windows can also be moved and resized.

Show Last Run

If an error occurs in a tree, you can open the debugger from the “Show last run” -button. Since Unity3D does not save changes to game data during play mode, the last traversal info can only be accessed while the play mode is still on. Error info will also be shown in the log.

Nodes

Creating Custom Nodes

To add a custom node into the behavior editor, you need to create a class that represents the node.

- Create a script in the appropriate folder (Actions / Conditions / Decorators / Internals). The script must have the same name as the class. Make sure the spelling is correct!
- Set the class to inherit from BaseNode.
- Override the Action() method.
- Best practice is to keep it as simple as possible. Ideally one node should do only one thing, so the behavior is easy to modify by moving the nodes rather than rewriting them.
- Action() must return the result state of the node.
- BT_State.SUCCESS if the action is completed successfully
- BT_State.RUNNING if the action is not yet complete and should continue in the next traversal
- BT_State.FAILURE if the action fails in some expected way (e.g. a condition is not met)
- BT_State.ERROR if the action fails in an unexpected way (e.g. required data is missing). Error will stop the BTree component executing.
- You can also override other methods, if you need them:
- _init() is for any initialization logic and is called when the behavior is first created. You can set variables to the Blackboard here. Trying to access a variable that is not found in Blackboard will create an error, but will not stop behavior execution.
- StartAction() is for any logic needed to begin the action. It is mainly useful for actions that take more than one frame to execute. It is called once before the Action() method, but not again while it is running.
- EndAction() is for any logic to end the running action. It is called when

the Action() is finished successfully, and also if another branch interrupts the action.

- All variables that need to be updated or accessed from outside the node should be set to the Blackboard. However, you can also use local variables inside the node. If you need a value that can be set in the editor, you must make it a property.

Actions

These hold the actual functionality of the behavior. One behavior tree can execute several actions during one traversal, depending on its structure. Actions can hold any sort of functionality.

Ideally, actions should not access other gameobjects or components directly, but save the resulting data to the blackboard.

Action nodes are mainly game specific, so they must be defined by the user (see Creating Custom nodes).

DebugWriter

DebugWriter node writes a log message every time it is accessed. The message can be edited in the Editor Window. There can be several DebugWriters in the same tree, and each can have a different message.

Conditions

It is best to put any condition checks into separate nodes, so that they can be easily moved or reused. Like action nodes, conditions are mainly user-defined (see Creating Custom nodes).

RandomCond

Returns BT_State.SUCCESS or BT_State.FAILURE depending on a random value. Success percentage can be edited in the Editor window. The range is 0-100, so that a value of 50 means a 50% chance of success.

Decorators

Decorators are nodes that modify tree execution order, or its child's return value.

A decorator can have only one child node. There are a few decorator nodes included in the behavior tool, but you can also create your own decorators.

AlwaysSuccess

Calls its child and returns BT_State.SUCCESS, unless the child returns BT_State.RUNNING or BT_State.ERROR. This can be useful for example when a failed action should not stop a sequence.

AlwaysFailure

Calls its child and returns BT_State.FAILURE, unless the child returns BT_State.RUNNING or BT_State.ERROR.

Inverse

Inverts the return value (BT_State.SUCCESS / BT_State.FAILURE). Error and running values are always returned as-is.

BT_WaitForSeconds

Only calls its child if the specified time has passed. The time limit can be set in the editor. Returns BT_State.RUNNING while the counter is running.

RepeatTimes

Calls its child the specified amount of times, unless the child returns BT_State.ERROR. The return value of the repeater is the last one it gets back from the child.

Internals

Internal nodes control the tree traversal. Internals call their children and use the returned state values to decide how to continue.

Parallel

Parallel node runs all its child nodes on every traversal. The return value of the parallel node depends on the collected values of its children: if number of failed children is greater than allowed limit, the parallel also fails. Otherwise if one or more child nodes return running, the parallel is also running. Error from any child node is returned immediately.

You can set the amount of accepted failures in the editor.

Selector

Selector runs its children in priority order, until it gets a return value success or running.

Plain Selector starts from the beginning of its children on every traversal, so a higher priority branch can interrupt a less important action.

SelectorMemory

SelectorMemory starts checking its children from the one that was successful on the last traversal.

RandomSelector

Picks one child at random and executes it.

Sequence

The Sequence node runs all its children in priority order, unless it receives a failure or error from a child node. There are two kinds of sequence nodes implemented in the editor.

Plain Sequence starts at the beginning of its children on every traversal. This is useful for conditioned branches. A plain sequence can have a child or subtree that takes more than one traversal to complete; if the child is reached again on the next traversal, it will continue its execution.

SequenceMemory

SequenceMemory is used for chaining actions that take longer to complete. It starts from the child that was left running on the last traversal. The previous children are not visited again, until the sequence has finished and can start again.