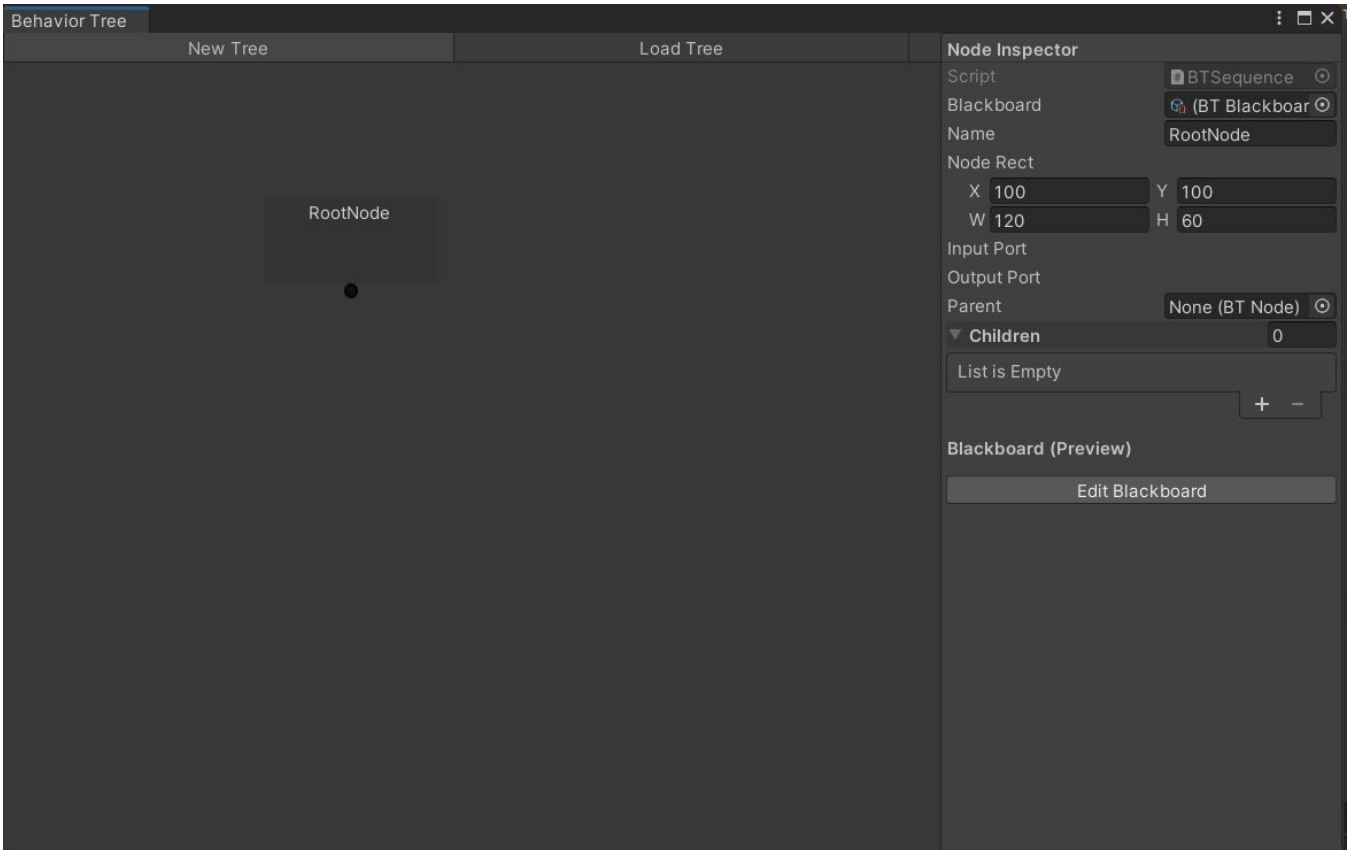
Easy Behavior Tree – Documentation

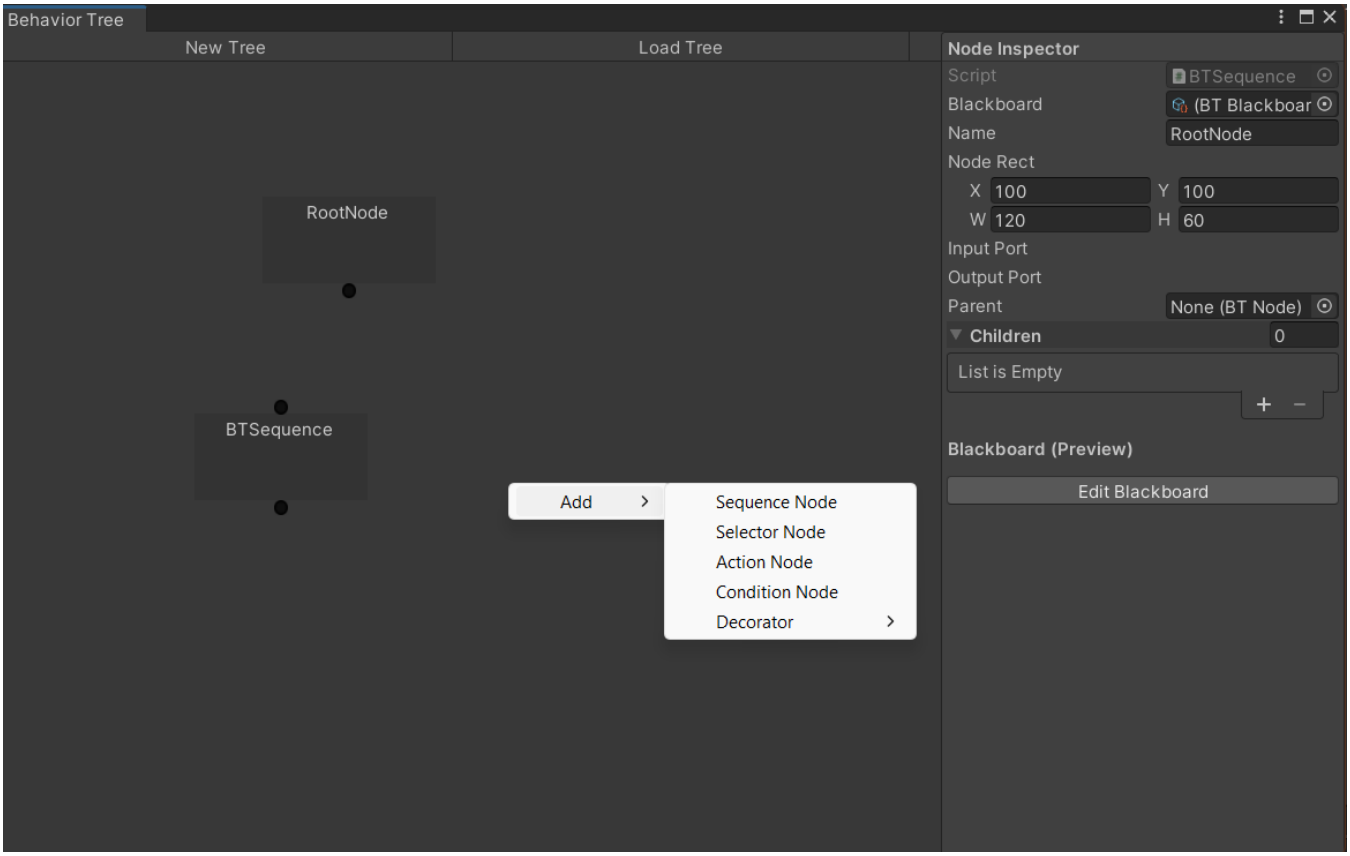
Quick Start:

1. Download the Easy Behavior Tree Package and drag and drop it into the Assets folder in the Unity Editor.
2. Go to Window > Behavior Tree Editor to start creating a new behavior tree or load an existing one.
3. Click the “New” button and navigate to where you want to save your new behavior tree. **Important:** The behavior tree must be saved in the assets folder or a subfolder of the assets folder.
4. The root node will be created for you and it is a “Sequence” node. Clicking on the node will reveal information about that node, and the blackboard shared between all nodes on this tree.

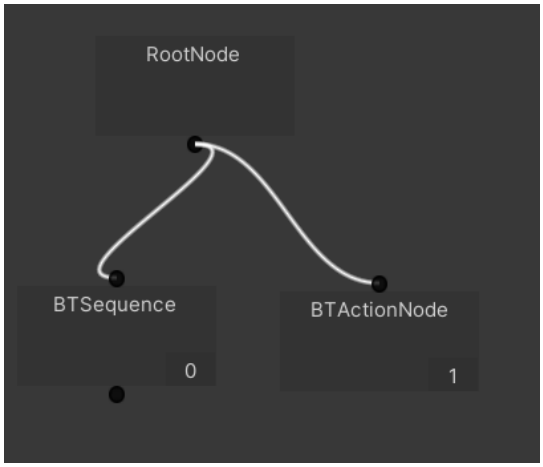


Important: The parent and child list are visible for bug fixing and SHOULD NOT be assigned directly. Doing so will break the behavior tree.

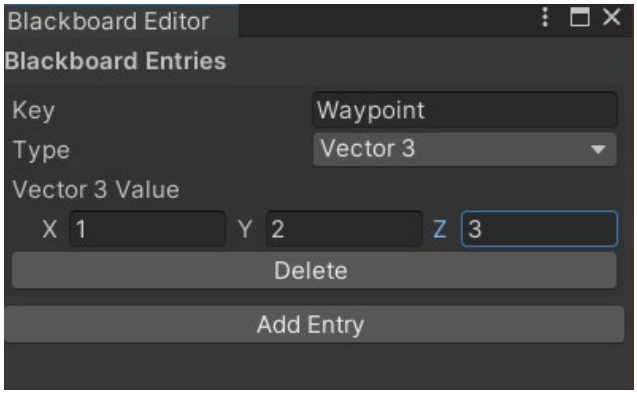
5. You can add new nodes by right-clicking empty space in the window.



6. The black dots above / below each node are the input / output ports respectively. You can click and drag to draw a connection from an input port to an output port, or vice versa. They cannot connect to ports of the same type. Once you have made the connection, when you click on the node, you should see that the parent and child fields have been updated in the inspector. You can also rename the nodes using the name property.
7. The numbers beneath each node indicate the order they will be evaluated and ticked. For example, if the parent node is a sequence, (in this case RootNode), the node with a “0” must be a SUCCESS before the node with a “1” will perform it’s action. If the “0” node never returns SUCCESS, then any nodes with a number greater than 0 will never be run.



8. The blackboard can be changed by clicking the “Edit Blackboard” button on any node, and a new window will appear. Click “Add Entry” to add a new variable that will persist between playthroughs. The “Key” is the name that you can use to reference that blackboard “Value”. The value is automatically of the type that you select from the “Type” dropdown.



Important: The “GameObject type references an existing prefab that you have made, NOT an object that exists at runtime.

9. Action nodes run user defined functions that you have created in a script. Simply put the name of the function in the “Function Name” section of the inspector.
10. Create a new script and reference the behavior tree you made. Use the *Tick(GameObject context)* method to start your tree. You can call this in the Update method if you want to tick every frame. The context is usually *this.gameObject*, but should be whatever game object has the functions you are calling with your action nodes.
11. Simply create a new method in your script and put it’s name as the “Function Name” of the action node, and as long as the context is the game object with a script with that function attached, it will run that function when that action node is ticked.
12. You can change blackboard values by first referencing the blackboard with:

```
BTBlackboard bb = bt.rootnode.GetBlackboard();
```

In this case, “bt” is the name of the BehaviorTree variable. Next, you can set the blackboard values with:

```
bb.Set<type>(“Key name”, value);
```

Editor Controls

Open the editor window by going to Window > Behavior Tree Editor on the toolbar at the top of the screen.

Create a new behavior tree by pressing “New Tree” and load an existing tree with “Load Tree”.

Left-click a node to select it and view it in the inspector. Click anywhere else to deselect it.

Left-click and drag a node to move it around.

Left-click an input or output port and drag the mouse to draw a connection between nodes.

Release left-click while dragging and over a port to form a connection. Releasing while not over a port cancels the connection.

Right-click a node to bring up options for that node, mainly delete.

Right-click a port to bring up options for that port, mainly break connections.

Middle-mouse-click and drag to pan around the behavior tree graph.

Using the Editor:

The editor has several different parts. The new / load toolbar at the top, the inspector on the right, and the behavior tree window in the center – left. When you create a new tree, the root node will be created for you. Clicking on it will show parameters of that node in the inspector. It is recommended that you leave all of them as is, except for the name property and editing the blackboard. The name property can help keep track of what does what on your tree and help keep things organized. The node rect property defines the size and position of the node within the tree and should only be changed if you know what you are doing, since changing the width / height will make the formatting of the tree weird.

Important: The Parent / Children parameters should **NEVER** manually be changed in the inspector. They are there to debug and give you information about what is connected to what. If you change them manually, it will be changed for the behavior tree, but it will not be reflected in the behavior tree editor, and it has no safeguards against things breaking the same way the editor does.

The Edit Blackboard button can be used to keep track of variables that you care about between play sessions or conditions to check with condition nodes. Condition and Action nodes have additional properties, and those can be changed as needed.

Node Details:

BTNode:

This is the base class for all nodes on the behavior tree. All child classes must implement the Execute method to define what they do.

The Initialize method can be overridden to remove input / output ports on the node, or do other initial operations that are unique to that node.

The Tick method is what is called to tick the node, and should only be used in the BehaviorTree script. Call the Tick method from the behavior tree instead of ticking individual nodes.

The Execute method should not be called directly but must be overridden in each child class to define what should happen when it is ticked. It returns a NodeState, which is:

{IDLE, RUNNING, SUCCESS, FAILURE}

The draw method draws the node in the behavior tree window, and changes the color based on the status in game. If the editor is open while the game is running, you can see the status of each node, based on its node state.

Gray = IDLE
Yellow = RUNNING
Green = SUCCESS
Red = FAILED

This method can be overridden to have child nodes that appear different in the editor. It also calls the draw methods for the input / output ports and the Index number.

The ResetStatus method can be used to reset this node and its child nodes back to IDLE state. This is useful for making sure states are not saved between play sessions.

The Move method simply moves the node and the ports when the user moves it in the behavior tree editor. This should only be overridden if you have a custom GUI element that you want to move when this node is moved.

The SetBlackboard method is in case you are building the tree first and injecting the blackboard later. The GetBlackboard method gets the blackboard that was created when the tree was created, or the last blackboard that was set with SetBlackboard.

Root Node:

The root node is a type of sequence node that is created at the same time as the behavior tree. It has only an output port and can have any number of child nodes. It cannot be deleted. When the behavior tree is ticked, this node is ticked first, then the first child node. If that child node returns SUCCESS, the next child node is ticked. It will keep trying to tick the first child node until it returns SUCCESS.

BTSequence:

A sequence node has both an input and an output port. When it is ticked by the parent node that is connected to the input port, it ticks its children in order until one returns FAILURE, then it returns FAILURE to its parent node. If all its children return SUCCESS, it returns SUCCESS to its parent node. This effectively runs a series of actions in order.

BTSelector:

A selector node has both an input and output port. When it is ticked by the parent node that is connected to the input port, it ticks its children in order until one returns SUCCESS, then it returns SUCCESS to its parent node. If all its children return FAILURE, it returns FAILURE to its parent node. This effectively chooses the first available action that it can perform.

BTInverter:

An inverter node has both an input and output port and can only have one child since it is a “Decorator”. It takes the result of the child node and inverts it. If the child node returns SUCCESS, the inverter will return FAILURE to its parent node, and vice versa. This class inherits from BTDecorator instead of BTNode so that it is limited to one child.

BTActionNode:

The action node is the core of any behavior tree. It allows the tree to interface with other code and does something when it is ticked. It has only an input port, since it is a “leaf” node. It will always return SUCCESS if the function name that is defined in the editor is valid. The function name is only valid if the behavior tree is ticked from a script that contains that function name.

BTCondition Node:

The condition node is a “leaf” node so it only has an input. It checks to see if a bool blackboard value is the same as the expected value. If it is, it returns SUCCESS. The bool blackboard value can be set in a user defined script.

Creating a Custom Node:

You can create your own custom node easily in two steps by creating a new script that inherits from BTNode, implements the Execute method, and is serializable.

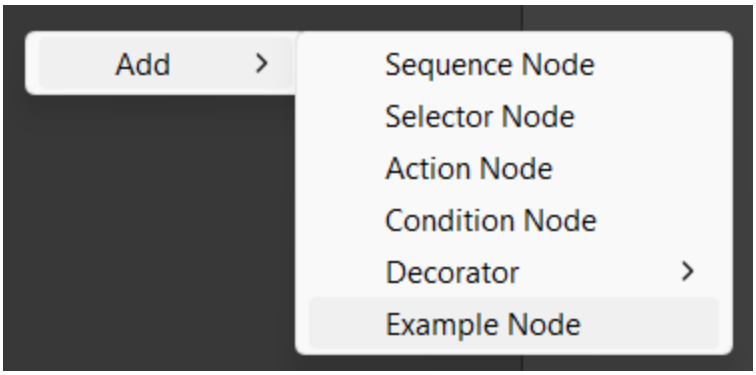
```
[Serializable]
Unity Script | 0 references
public class ExampleNode : BTNode
{
    6 references
    public override void Initialize(Vector2 position, BTBlackboard bb)
    {
        base.Initialize(position, bb);

        //Define any unique initialization code...

        //ex. OutputPort = null (to make a leaf node)

    }
    2 references
    protected override NodeState Execute(GameObject context)
    {
        //Whatever you want to do when the node executes...

        return NodeState.SUCCESS;
    }
}
```

Once you have the new node, you can make it usable in the editor by going to the BehaviorTreeEditor.cs file. Find the ShowContextMenu(Vector2 position) method.

```
1 reference
private void ShowContextMenu(Vector2 position)
{
    GenericMenu menu = new GenericMenu();
    menu.AddItem(new GUIContent("Add/Sequence Node"), false, () => CreateNode<BTSequence>(position));
    menu.AddItem(new GUIContent("Add/Selector Node"), false, () => CreateNode<BTSelector>(position));
    menu.AddItem(new GUIContent("Add/Action Node"), false, () => CreateNode<BTActionNode>(position));
    menu.AddItem(new GUIContent("Add/Condition Node"), false, () => CreateNode<BTConditionNode>(position));
    menu.AddItem(new GUIContent("Add/Decorator/Inverter Node"), false, () => CreateNode<BTInverter>(position));
    menu.ShowAsContext();
}
1 reference
```

Copy one of the menu items and replace the content with your new node, and the class name between the < > of CreateNode.

```
menu.AddItem(new GUIContent("Add/Example Node"), false, () => CreateNode<ExampleNode>(position));
```



You can now use your node like any other node. Any public or serialize field variables defined in your node will show up in the inspector. Those variables should probably remain constant.