BeeHIVE:

A behavior tree solution

Nilo Fernandes Varela

DOCUMENTATION

Contents

1 INTRODUCTION

WIP

1.1 Behavior Trees

WIP

1.2 Tool overview

WIP

2 QUICK GUIDE

Here is a tutorial on how to get BeeHIVE up and running on a project. This cover the basic nodes and features, but builds a foundation for tool usage.

2.1 Project Setup

Before we get started, we need the BeeHIVE files, if you downloaded the package from asset store, obviously you already have the files needed, and already imported to your project, so you are good to go. If you downloaded from the public repository, you can open the project that comes with it. After that its done, you can right click the BeeHIVE folder, located under the Assets folder, and click *Export Package* as shown in figure 1.

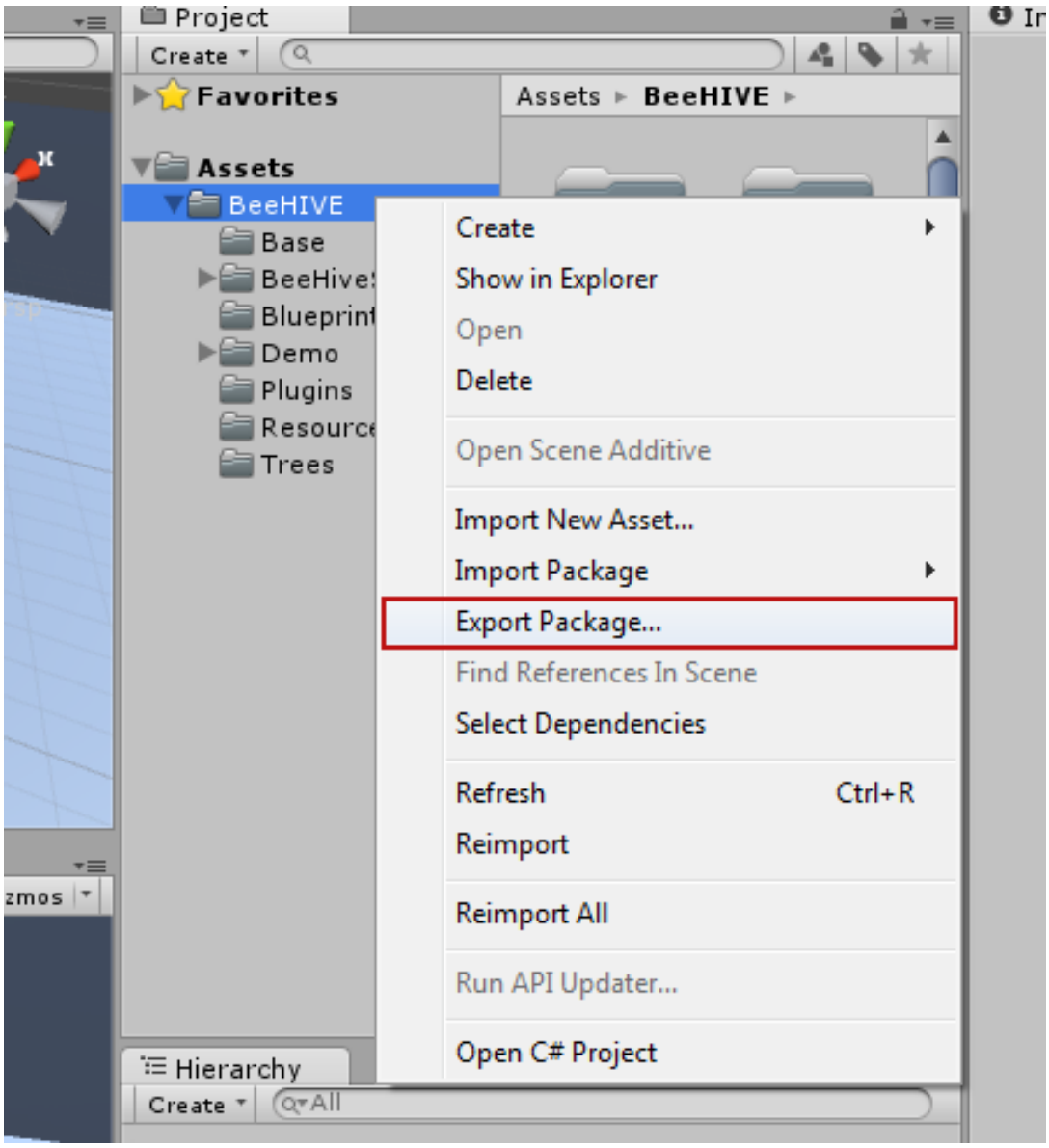

Figure 1: Exporting package

A pop-up window will appear, leave everything checked and click *Export*. Chose a location and a name for the package, and click *Save*. Now you have a package that can be imported in any project by *right clicking the Assets folder / Import Package / Custom Package*, as shown in figure 2.

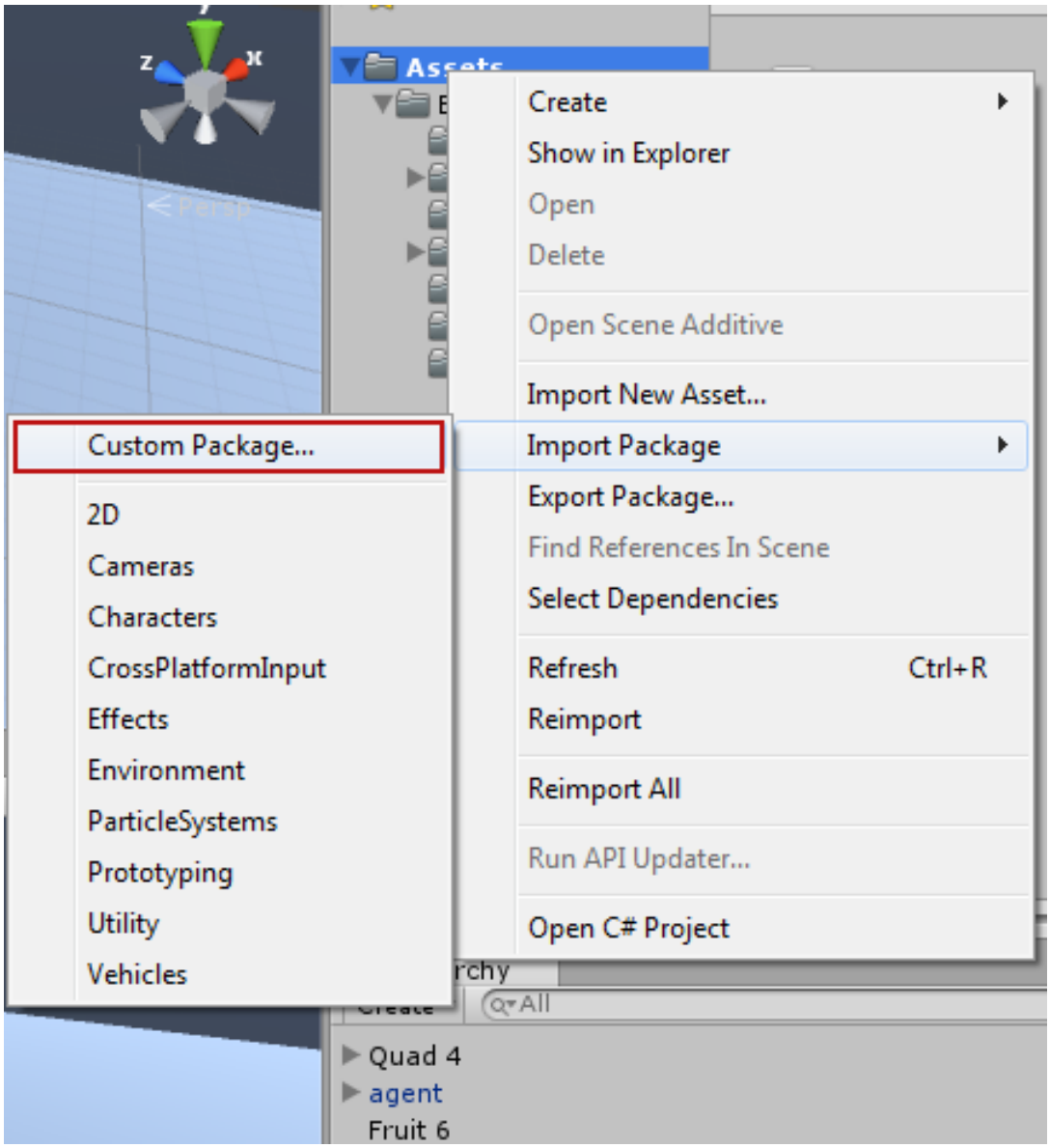

Figure 2: Exporting package

2.2 Tutorial

Now that you know how to import BeeHIVE into a project, we will go trough a simple tutorial to get you started. First of all, open up the Demo folder located inside the BeeHIVE folder, than open the Tutorial Demo folder and double click the Tutorial scene.

What we are going to do its a simple AI that can play pong against a human,

is worth saying that in many cases, such as this, the AI its too simple to even bother using something like behavior trees, this is just a case study to get you used to the core concepts, besides that, the pong example you are about to see is far from a well done and fully developed game, so expect some weird bugs from time to time, and keep in mind that is for learning purposes only. So without further ado, let's do this.

PROJECT OVERVIEW

The tutorial project have almost everything you need for the base game, the most important class being the *PongController*, which holds all the methods related player capabilities, in this case, stuff that has to do with movement. The human controller inherits from this class to control his 'character' with keyboard input, and the AI we are about to create will it directly, but instead of using keyboard input, it will use commands defined in the behavior tree. You can hit play now and control the player on the left with the arrow keys, but the AI wont give any trouble whatsoever.

CREATING A BEEHIVE AGENT

Let's create a new C# script and call it *PongAI*. Add the script to the *AIPlayer* object on the scene and open it up. This script will be the bridge between BeeHIVE and the *PongController*, so the first thing we need to do is to add the BeeHIVE library to it, so the class will look like this:

```
using UnityEngine;
using System.Collections;
using BeeHive;

public class PongAI : MonoBehaviour {

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

        }
}
```

Now we need to turn *PongAI* into a BeeHIVE agent, simply by inheriting from a conveniently named class called *BeeHIVEAgent*. Now your code should look something like this:

```
using UnityEngine;
using System.Collections;
using BeeHive;

public class PongAI : BeeHIVEAgent {

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

        }
}
```

BeeHIVEAgent is a very simple class, that aims to ensure that you have the proper setup without much work. If that inheritance ever gets in your way for some reason you can always copy that code directly into yours.

The next step is initialize BeeHIVE and whatever behavior associated with it. We have two ways of doing that, the first its overriding the *Start* method and calling the base *Start*, and the second way its to call a method called *InitBeeHive* on your script, at the *Start*. For sake of simplicity we are going with the second one. Your *Start* method should look like this:

```
        void Start () {
                InitBeeHive();
        }
```

From now on you have a valid agent, all you need its to implement the actual actions that can be used on the system. If you save your code and get back to the editor, you will see some variables at the inspector, but don't worry about those now, we will get there. Last thing before we get to actions, is to get a reference to the *PongController* script, so we can use it to move the player around. Doing that your *Start* should look like this:

```
    PongController pongController;

    void Start () {
        pongController = GetComponent<PongController>();
```

```
        InitBeeHive();
    }
```

IMPLEMENTING VALID ACTIONS

For a method to be recognizable as a valid action by BeeHIVE, the return type has to be a enum called *BH_Status*, which can assume values of *BH_Status.Success*, *BH_Status.Failure* and *BH_Status.Running*. The *Success* is returned when a action successfully completed, and *Failure* is the other way around. *Running* is returned when the actions need more iterations to complete, or maybe do not complete at all. We will see examples of all of those.

Actions to move up and down are continuous, so there is never a point where it succeed or fail in this particular case, every time we call it, is always on running state. For that reason we can implement the move actions like this:

```
    public BH_Status MoveUp(){
        pongController.MoveUp();
        return BH_Status.Running;
    }

    public BH_Status MoveDown(){
        pongController.MoveDown();
        return BH_Status.Running;
    }
```

The stop action is a bit different, and there is no condition that can prevent its execution in this game, so every time we call it, it can return a success status, like this:

```
    public BH_Status StopMoving(){
                pongController.Stop();
                return BH_Status.Success;
        }
```

The last action we going to implement is to check if the ball is above or below the agent, so it can make a decision to what direction to move. In this example we will implement only a action to check if ball is above.

```
public BH_Status BallIsAbove(){
    float myHeight = pongController.myTransform.position.y;
    float ballHeight = pongController.ball.position.y;
    if(myHeight < ballHeight)
        return BH_Status.Success;
```

```
        else
            return BH_Status.Failure;
}
```

Remember to save your code, and get back to the editor.

BUILDING A TREE

Now for the fun part, we are going to create a tree to model the behavior of our opponent. Start by clicking *Window / BeeHIVE Editor*, like shown on figure 3.

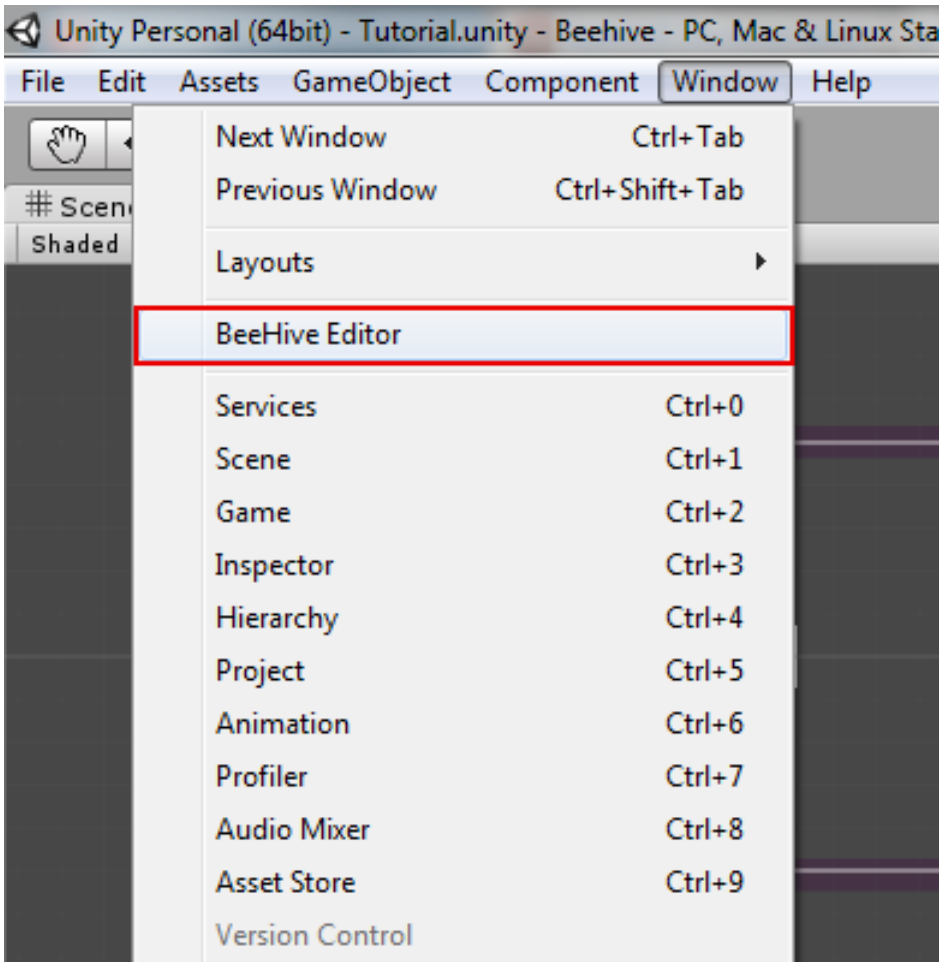

Figure 3: BeeHIVE Editor access

The node editor will pop up, and if you get confused at any point, you can check the figure 4, where it explains all the elements present in it.

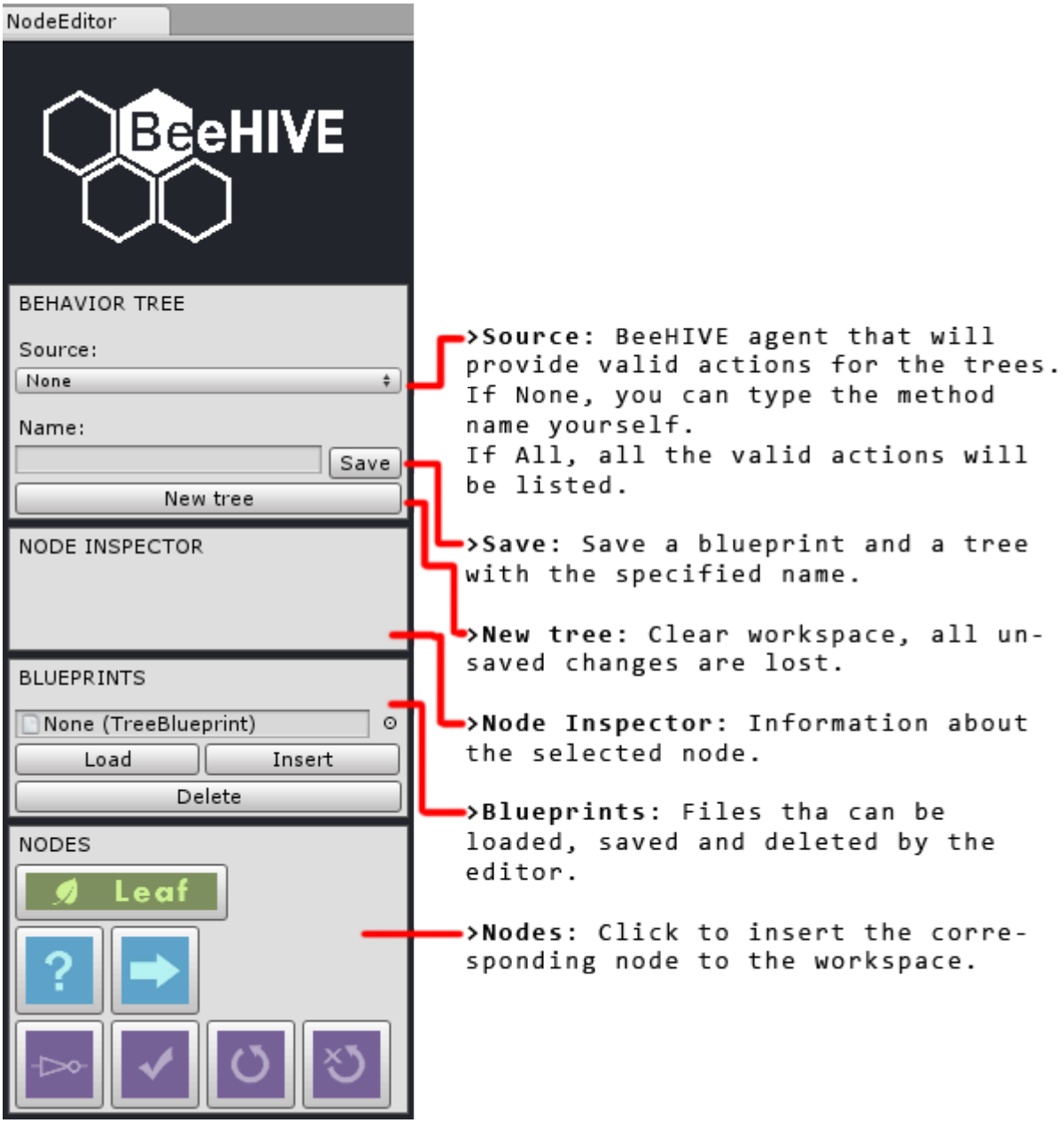

Figure 4: BeeHIVE Editor menu

If you don't already know, is recommended that you read a bit about tree structures, so you can have a full understanding of what i am talking about. Anyway, in BeeHIVE, the trees are searched on a *depth-first* fashion, and from left to right, a example of search order can be seen in figure 5. We have to keep in mind that the order in which we arrange the nodes, will carry meaning to the behavior.

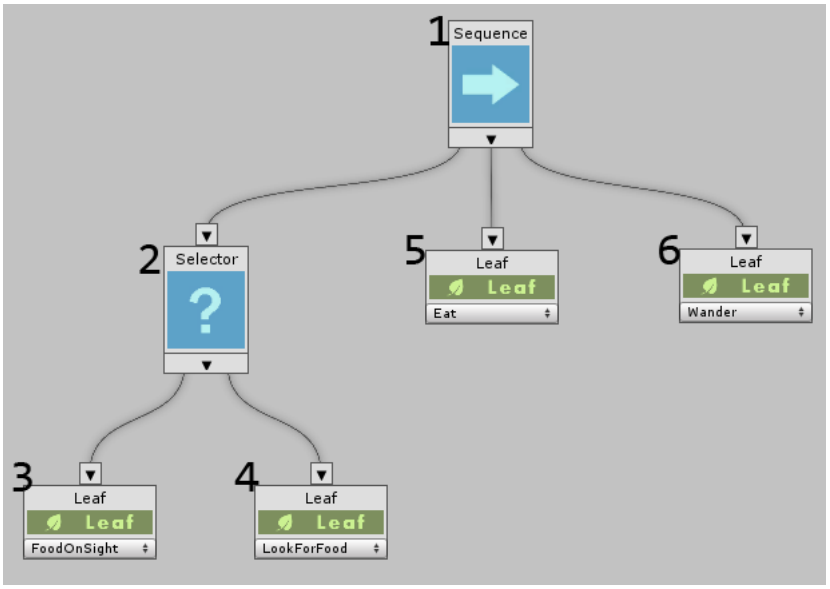

Figure 5: Depth-first search order

Lets start building our tree from the bottom. Click the leaf icon on the left, or right click anywhere on the workspace and create two leaves. Those nodes will always be at the bottom of the tree, so they can only have connections coming in. Leaves are a key part of the system, because they will hold references to one of those actions we implemented earlier. At this point, leaves might have a text field for you to write the name of the method you want to reference, you are free to do it that way, but lets try something easier. At the very top, the first option is called source, and has a drop-down menu for you pick a valid BeeHIVE agent, and hopefully, we already made one, so click the arrow and chose the class you created, like shown in figure 6.

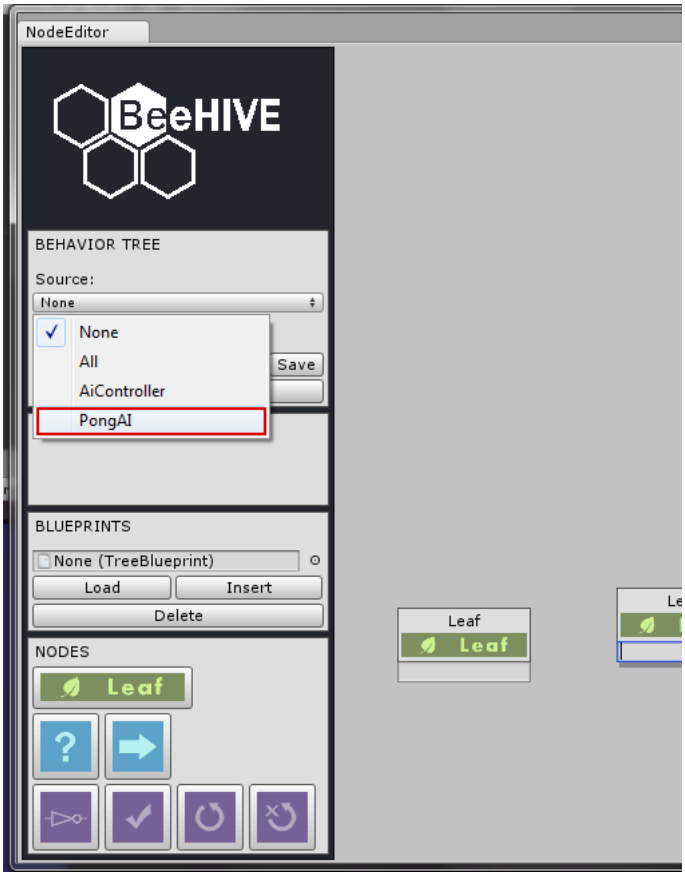

Figure 6: Selecting the source agent.

Note that after you selected a source the text-field turns into a drop-down list, in which you can easily select one of the actions.

Now we want to check if the ball is above the agent, and if it is, we want the agent to move up. So we are going to select *BallIsAbove* on the left node, and *MoveUp* on the right node.

To connect both leaves we are going to create a sequence node (the blue one with a arrow), this node execute his children from left to right, and if any of them returns a *Failure* status, the sequence node will stop the execution and will also return *Failure* to its parent. If all the children returns a *Success* status, the sequence node will also return *Success* to its parent. Basically, the sequence node act like a AND gate.

Add a sequence node and click the arrow at bottom of the node to initiate a connection. You will see a line being draw, and than you can click in one of the nodes to make a connection. Repeat the process for the other one, make sure it look like figure 7.

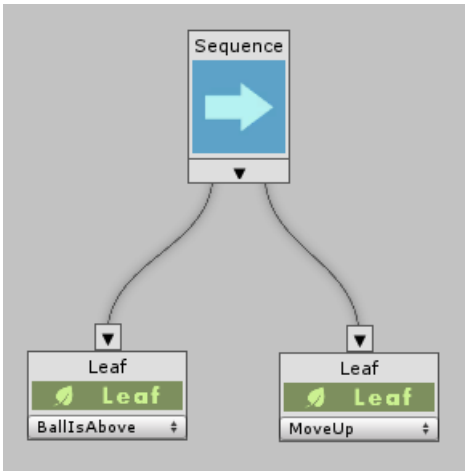

Figure 7

If you make any mistakes you can break a connection by clicking the arrow that sits on top of the node.

At this point you can enter a name and click *Save* to create a usable file for the agent. Open a folder called Trees, that is where all trees are saved, so you might be able to see the one you just created. Now select the AIPlayer object, and at the inspector of the *PongAI* class you will see a property called *Behavior Tree Obj*. Drag and drop your tree in that property to set it to this agent.

If you hit play now you can already see some reaction from AI, but it can only go up, lets fix that.

If you closed the BeeHIVE editor, open it up again, and load your blueprint using the property on the third section. Now we want to opposite action to happen when the ball is not above the agent, so we are going to create two more leaves, one will be *MoveDown*, and the other will be *BallIsAbove*, but this time we have to invert the result to make node return *Success* when the ball is not above. To do that we will also need to add a inverter node (the first purple node on the tool bar). And to connect everything up we need another sequence node. Our tree should be something like figure 8.

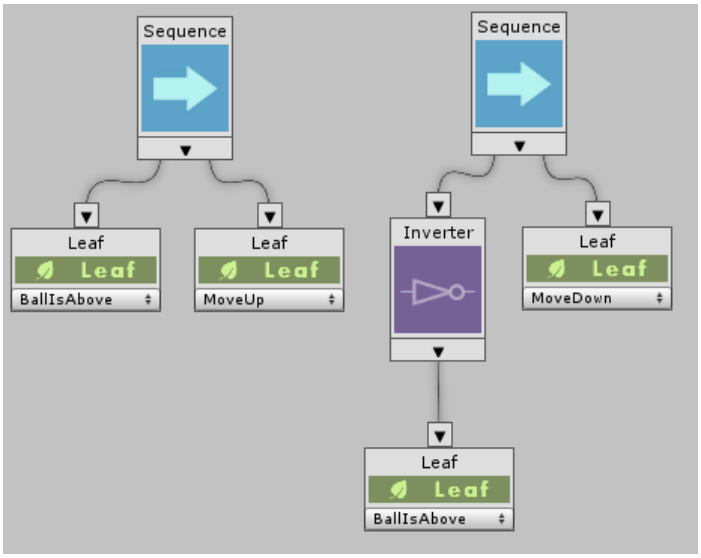

Figure 8

Note that at this point we have a tree with two roots, or even better, we have actually two separate trees, in this case the system would have no idea of what root to use, so this should be avoided. So lets connect these two sequence nodes together with a selection node (blue node with a question mark). The selection node works like a OR gate, which means that if any of its children return a success status, it will not process any further, and will return a success status to its parent. Putting in a simple way, the ai will check if the ball is above, and if it is, it will not check if it is bellow. Now we have a tree like figure 9.

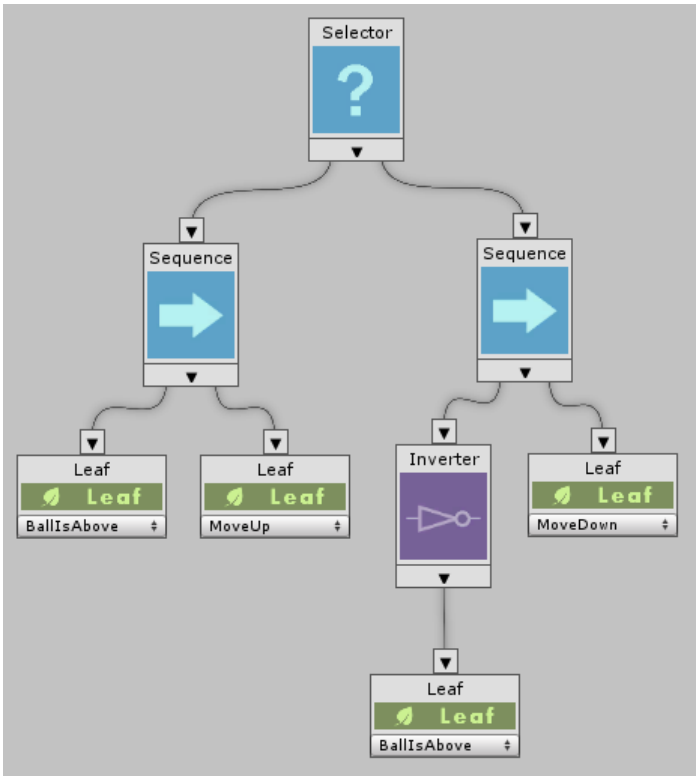

Figure 9

Right now you can save and hit play, and you will have a really basic ai that move with a very straight forward logic, but does the trick for a learning project such as this.

That last thing we can play with is the tick rate. When you select AIPlayer object and look at the inspector, you will see a variable called, guess what, Tick Rate. That variable define the time interval in which the tree will be searched, and in practical way, can be increase to reduce performance costs if you have to many agents behaving at once, or to quickly adjust the ai reaction time. Try to play with that value and see how the ai changes its behavior, will be almost like adjusting the difficulty.

I really encourage you to extend this example a bit, implementing new actions, maybe add a laser gun that you can shoot the opponent with, and the ai would have to make decisions on when to shoot, anyway anything that sounds fun.

I hope this tutorial was useful, give me any feedback you might have to help me improve this doc, including typos or poorly constructed phrases (struggling a bit with English :)).

Enjoy the tool and HAVE FUN.

3 TOOL REFERENCE

WIP

REFERENCES