

Behavior Designer

Table of Contents

Behavior Designer	2
Overview	3
What is a Behavior Tree?	7
Behavior Trees or Finite State Machines	9
Installation	10
Behavior Tree Component	10
Create a Behavior Tree from Script	12
Behavior Manager	13
Tasks	14
Parent Tasks	15
Writing a New Conditional Task	16
Writing a New Action Task	19
Debugging	22
Variables	24
Dynamic Variables	27
Global Variables	27
Creating Shared Variables	28
Accessing Variables from non-Task Objects	28
Conditional Aborts	29
Events	32
External Behavior Trees	33
Networking	35
Referencing Tasks	36
Object Drawers	38
Variable Synchronizer	40
Syncing Animations	42
Referencing Scene Objects	43
Task Attributes	44
Integrations	46
Dialogue System	47
Opsive Character Controllers	49
Playmaker	60
uScript	65
Videos	68

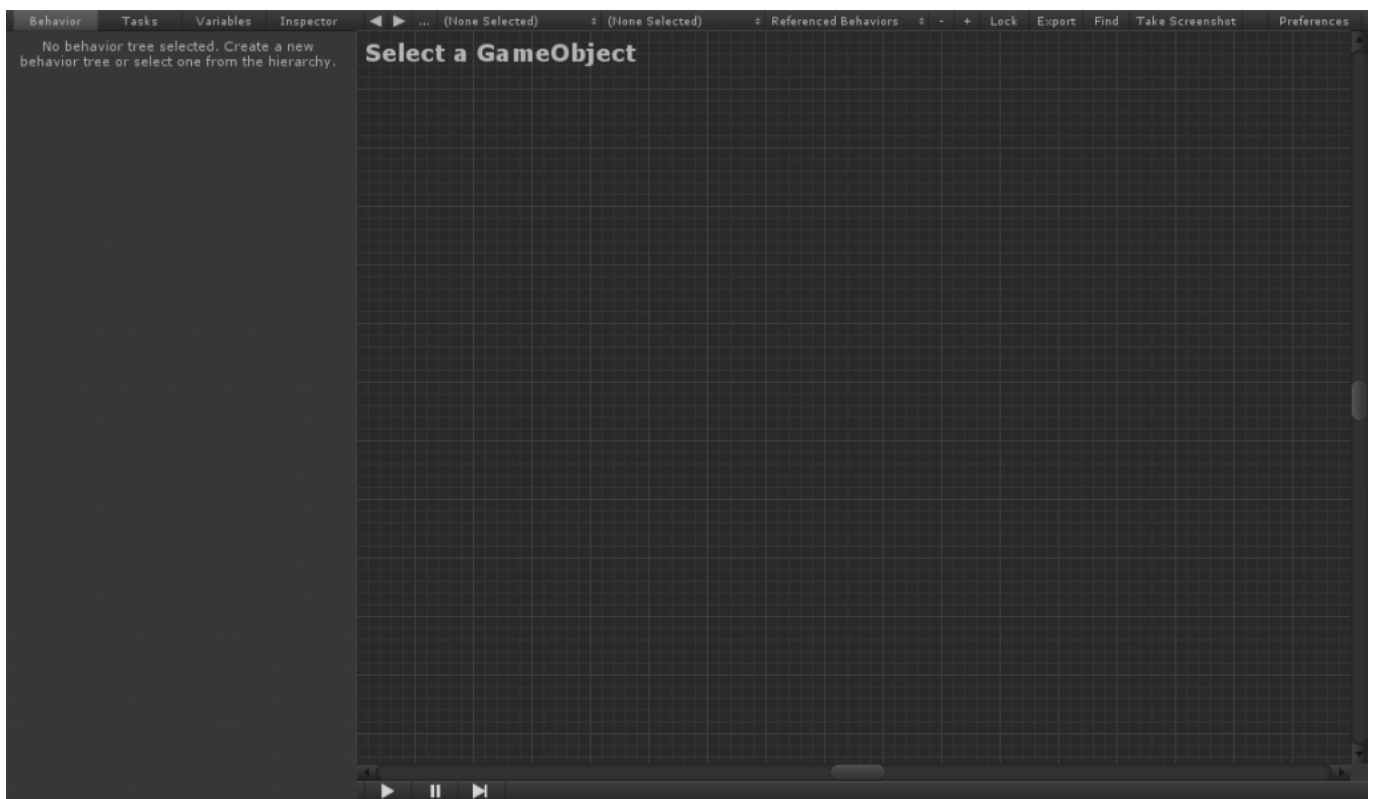
Behavior Designer

Overview

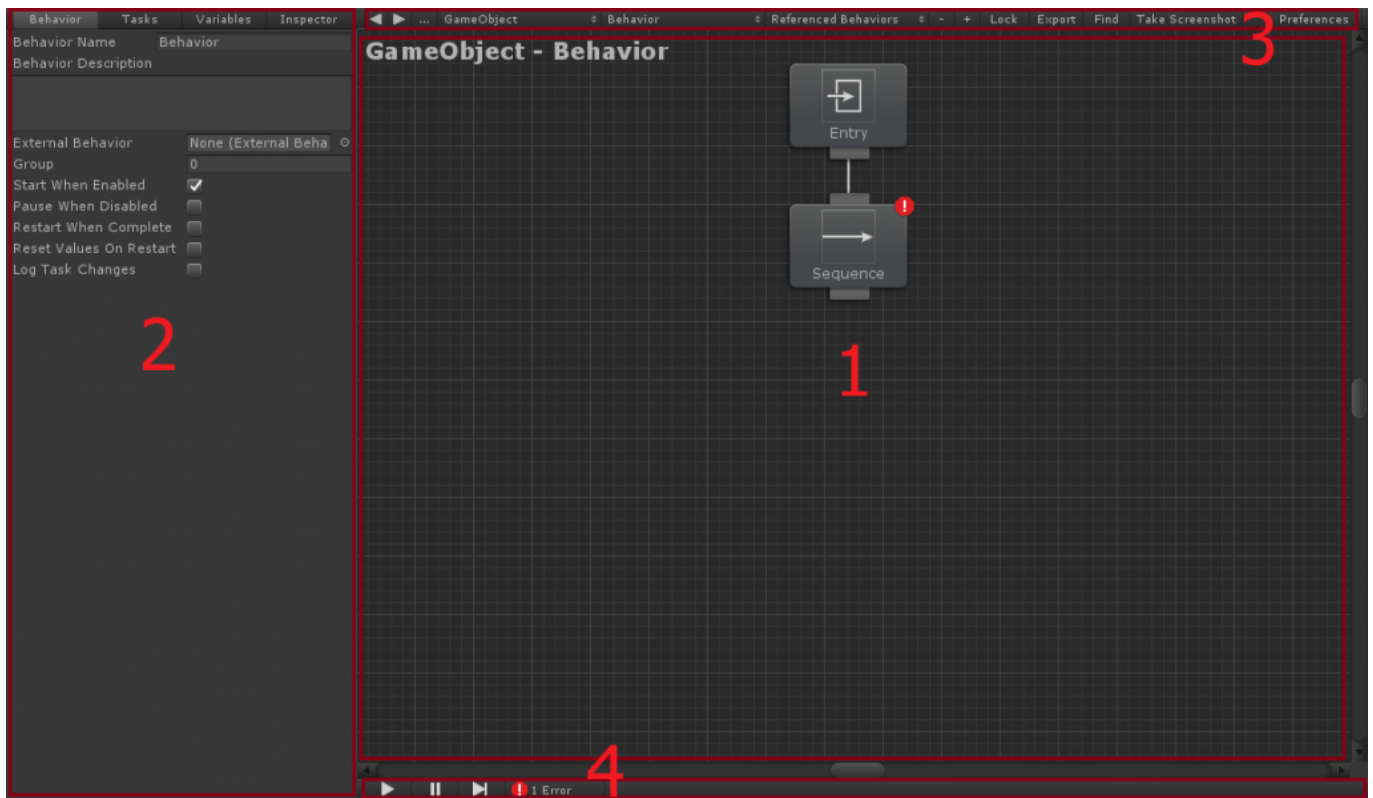
Behavior Designer is a behavior tree implementation designed for everyone - programmers, artists, designers. Behavior Designer offers a powerful API allowing you to easily create new tasks. it offers an intuitive visual editor with extensive third party integration making it possible to create complex AIs without having to write a single line of code.

This guide is going to give a general overview of all aspects of Behavior Designer. If you're just getting started with behavior trees we have a "Behavior Tree Basics" [video series](#). This page also has a quick [overview](#) of behavior trees. With Behavior Designer you don't need to know the underlying behavior tree implementation but is a good idea to know some of the key concepts such as the types of tasks (action, composite, conditional and decorator).

When you first open Behavior Designer you'll see the following window:

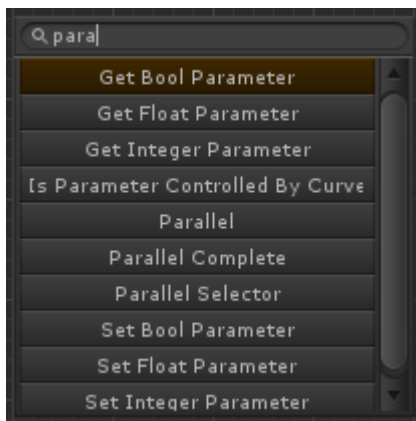


There are four sections within Behavior Designer. From the screenshot below, section 1 is the graph area. It is where you'll be creating the behavior trees. Section 2 is a properties panel. The properties panel is where you'll be editing the specific properties of a behavior tree, adding new tasks, creating new variables, or editing the parameters of a task. Section 3 is the behavior tree operations toolbar. You can use the drop down boxes to select existing behavior trees or add/remove behavior trees. The final section, section 4, is the debug toolbar. You can start/stop, step, and pause Unity within this panel. In addition, you'll see the number of errors that your tree has even before you start executing your tree.

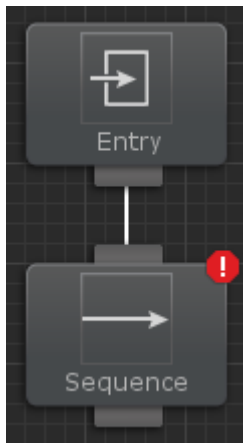


Section 1 is the main part of Behavior Designer that you'll be working in. Within this section you can create new tasks and arrange those tasks into a behavior tree. To start things off, you first need to add a Behavior Tree component. The Behavior Tree component will act as the manager of the behavior tree that you are just starting to create. You can create a new Behavior Tree component by right clicking within the graph area and clicking "Add Behavior Tree" or by clicking on the plus button next to "Lock" within the operations area of section 3.

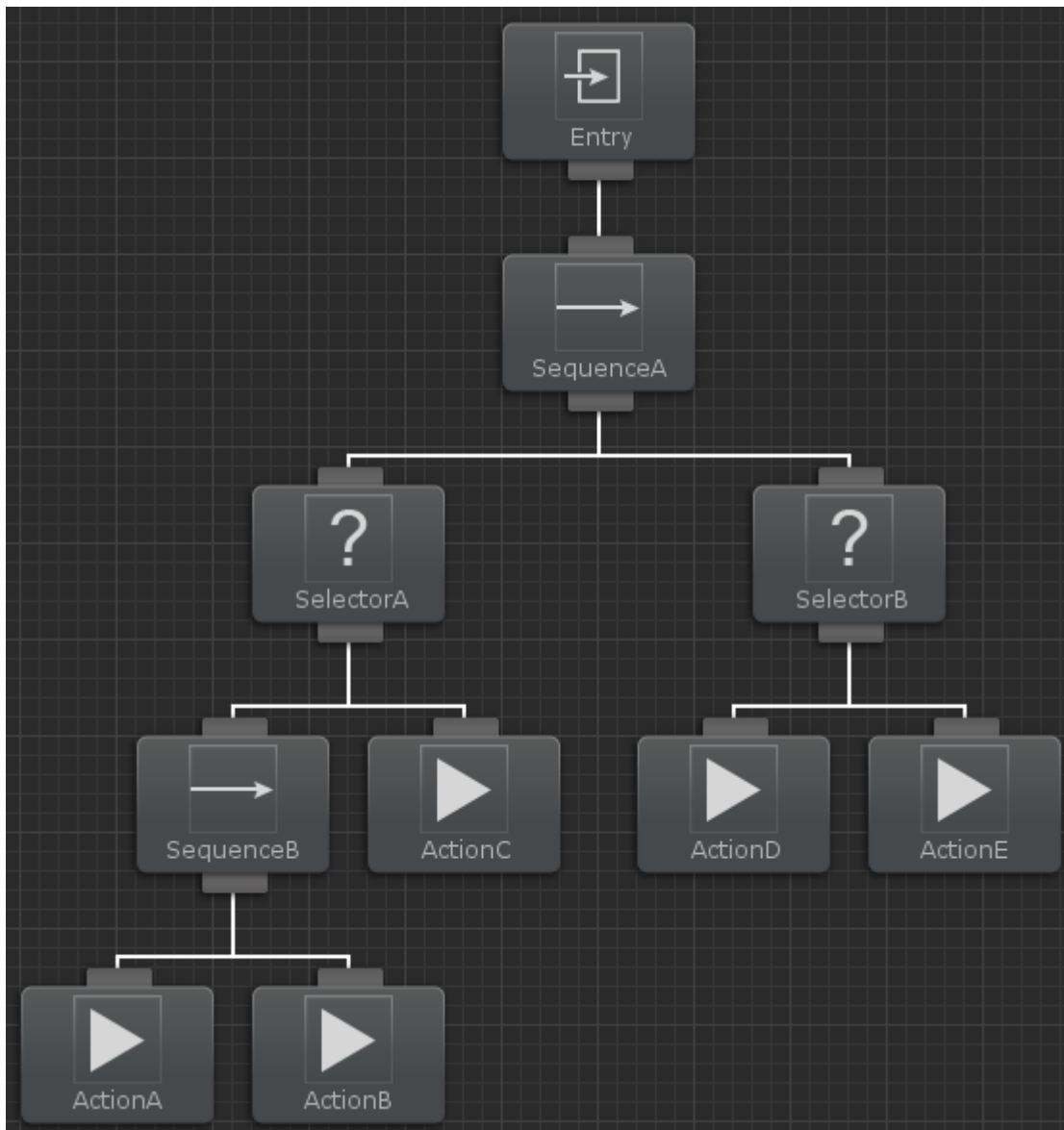
Once a Behavior Tree has been added you can start adding tasks. Add a task by right clicking within the graph area or clicking on the "Tasks" tab within section 2, the properties panel. New tasks can also be added by pressing the space bar and opening the quick task search window:



Once a task has been added you'll see the following:



In addition to the task that you added, the entry task also gets added. The entry task acts as the root of the tree. That is the only purpose of the entry task. The sequence task has an error because it has no children. As soon as you add a child the error will go away. Now that we've added our first task lets add a few more:

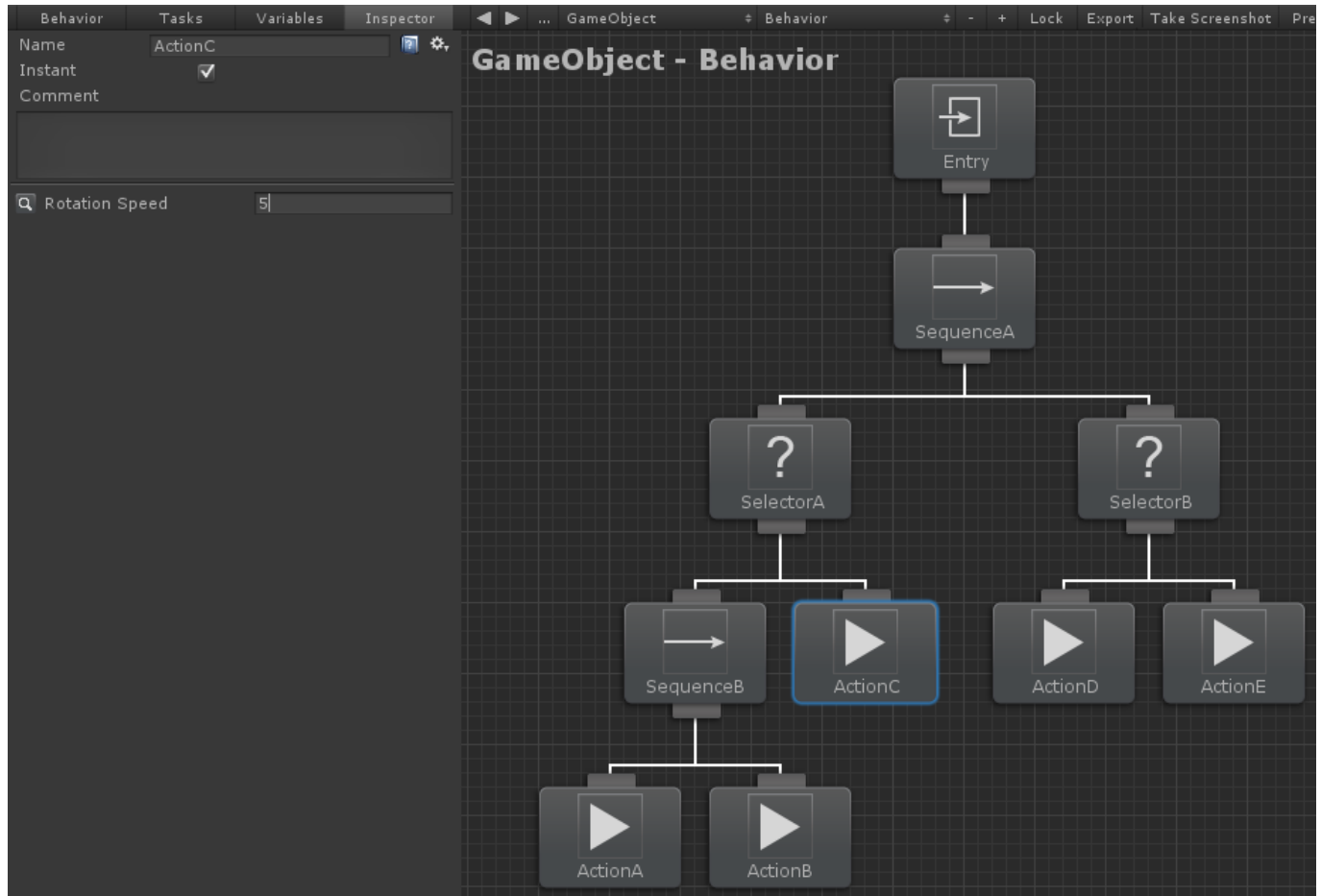


You can connect the sequence and selector task by dragging from the bottom of the sequence task to the top of the selector task. Repeat this process for the rest of the tasks. If you make a mistake you can selection a connection and delete it with the delete key. You

can also rearrange the tasks by clicking on a task and dragging it around.

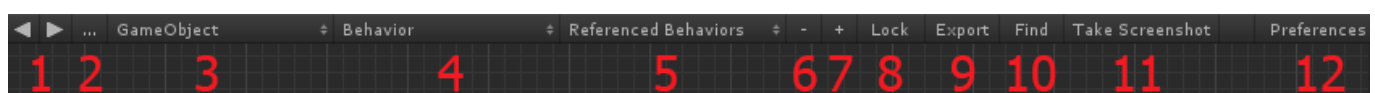
Behavior Designer will execute the tasks in a depth first order. You can change the execution order of the tasks by dragging them to the left/right of their sibling. From the screenshot above, the tasks will be executed in the following order:

SequenceA, SelectorA, SequenceB, ActionA, ActionB, ActionC, SelectorB, ActionD, ActionE



Now that we have a basic behavior tree created, let's modify the parameters on one of the tasks. Select the ActionC node to bring up the Inspector within the properties panel. You can see here that we can rename the task, set the task to be instant, or enter a task comment. In addition, we can modify all public variables the task class contains. This includes assigning variables created within Behavior Designer. In our case the only public variable is the *Rotation Speed*. The value that we set the parameter to will be used within the behavior tree.

There are three other tabs within the properties panel: Variables, Tasks, and Behavior. The variables panel allows you to create variables that are shared between tasks. For more information take a look at the [variables topic](#). The tasks panel lists all of the possible tasks that you can use. This is the same list as what is found when you right click and add a task. This list is created by searching for any class that is derived from the action, composite, conditional, or decorator task type. The last panel, the behavior panel, shows the inspector for the Behavior Tree component that you added when you first created a behavior tree. More details on what each option does is on the [Behavior Component Overview page](#).

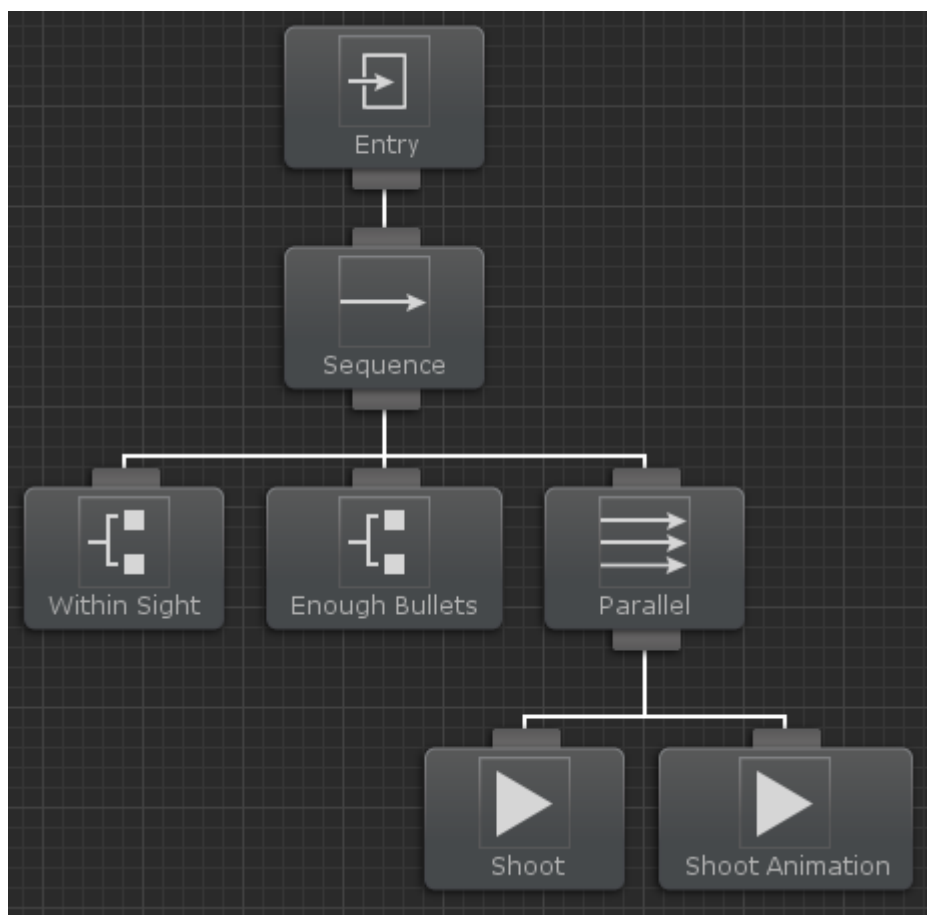


The final section within the Behavior Designer window is the operations toolbar. The operations toolbar is mostly used for selecting behavior trees as well as adding/removing behavior trees. The following operations are labeled:

- Label 1: Navigate back/forward between the behavior trees that you have opened.
- Label 2: Lists all of the behavior trees that are within the scene or project (includes prefabs).
- Label 3: Lists any GameObject within the scene that has the behavior tree component added to it.
- Label 4: Lists any behavior tree that are attached to the GameObject that is selected from label 3.
- Label 5: Lists any external behavior trees that the current behavior tree references
- Label 6: Removes the currently selected behavior tree.
- Label 7: Adds a new behavior tree to the GameObject.
- Label 8: Keeps the current behavior tree active even if you have selected a different GameObject within the hierarchy or project window
- Label 9: Exports the behavior tree to an external behavior tree asset.
- Label 10: Opens the find dialogue which can search your behavior tree.
- Label 11: Takes a screenshot of the current behavior tree.
- Label 12: Shows the Behavior Designer preferences.

What is a Behavior Tree?

Behavior trees are a popular AI technique used in many games. Halo 2 was the first mainstream game to use behavior trees and they started to become more popular after a [detailed description](#) of how they were used in Halo 2 was released. Behavior trees are a combination of many different AI techniques: hierarchical state machines, scheduling, planning, and action execution. One of their main advantages is that they are easy to understand and can be created using a visual editor.



At the simplest level behavior trees are a collection of tasks. There are four different types of tasks: action, conditional, composite, and decorator. Action tasks are probably the easiest to understand in that they alter the state of the game in some way. Conditional tasks test some property of the game. For example, in the tree above the AI agent has two conditional tasks and two action tasks. The first two conditional tasks check to see if there is an enemy within sight of the agent and then ensures the agent has enough bullets to fire his weapon. If both of these conditions are true then the two action tasks will run. One of the action tasks shoots the weapon and the other task plays a shooting animation. The real power of behavior trees comes into play when you form different sub-trees. The two shooting actions could form one sub-tree. If one of the earlier conditional tasks fails then another sub-tree could be made that plays a different set of action tasks such as running away from the enemy. You can group sub-trees on top of each other to form a high level behavior.

Composite tasks are a parent task that hold a list of child tasks. From the above example, the composite tasks are labeled sequence and parallel. A sequence task runs each task once until all tasks have been run. It first runs the conditional task that checks to see if an enemy is within sight. If an enemy is within sight then it will run the conditional task that checks to see if the agent has any bullets left. If the agent has enough bullets then the parallel task will run that shoots the weapon and plays the shooting animation. Where a sequence task executes one child task at a time, a parallel task executes all of its children at the same time.

The final type of task is the decorator task. The decorator task is a parent task that can only have one child. Its function is to modify the behavior of the child task in some way. In the above example we didn't use a decorator task but you may want to use one if you want to stop a task from running prematurely (called the interrupt task). For example, an agent could be performing a task such as collecting resources. It could then have an interrupt task

that will stop the collection of resources if an enemy is nearby. Another example of a decorator task is one that reruns its child task x number of times or a decorator task that keeps running the child task until it completes successfully.

One of the major behavior tree topics that we have left out so far is the return status of a task. You may have a task that takes more than one frame to complete. For example, most animations aren't going to start and finish within just one frame. In addition, conditional tasks need a way to tell their parent task whether or not the condition was true so the parent task can decide if it should keep running its children. Both of these problems can be solved using a task status. A task is in one of three different states: running, success, or failure. In the first example the shoot animation task has a task status of running for as long as the shoot animation is playing. The conditional task of determining if an enemy is within sight will return success or failure within one frame.

Behavior Designer takes all of these concepts and packages it up in an easy to use interface with an API that is similar to Unity's MonoBehaviour API. Behavior Designer includes many composite and decorator classes within the standard installation. Action and conditional tasks are more game specific so not as many of those tasks are included but there are many examples within the [sample projects](#). New tasks can be created by [extending from](#) one of the task types. In addition, many [videos](#) have been created to make learning Behavior Designer as easy as possible.

Behavior Trees or Finite State Machines

In what situations do you use a behavior tree over a finite state machine (such as Playmaker)? At the highest level, behavior trees are used for AI while finite state machines (FSMs) are used for more general visual programming. While you can use behavior trees for general visual programming and finite state machines for AI, this is not what each tool was designed to do. According to some, the age of [finite state machines is over](#). We aren't going to go that far, but behavior trees definitely have their advantages over finite state machines when it comes to AI.

Behavior trees have a few advantages over FSMs: they provide lots of flexibility, are very powerful, and they are really easy to make changes to.

Lets first look at the first advantage: flexibility. With a FSM, how do you run two different states at once? The only way is to create two separate FSMs. With a behavior tree all that you need to do is add the parallel task and you are done - all child tasks will be running in parallel. With Behavior Designer, those child tasks could be a PlayMaker FSM and those FSMs will be running in parallel.

One more example of flexibility is the task guard task. In this example you have two different tasks that play a sound effect. The two different tasks are in two different branches of the behavior tree so they do not know about each other and could potentially play the sound effect at the same time. You don't want this to happen because it doesn't sound good. In this situation you can add a semaphore task (called a Task Guard in Behavior Designer) and it will only allow one sound effect to play at a time. When the first sound finishes playing the second one will start playing.

Another advantage of behavior trees are that they are powerful. That isn't to say that FSMs aren't powerful, it is just that they are powerful in different ways. In our view behavior trees allow your AI to react to current game state easier than finite state machines do. It is easier to create a behavior tree that will react to all sorts of situations whereas it would take a lot of states and transitions with a finite state machine in order to have similar AI. In order to achieve the same results with a FSM you would end up with a [spaghetti state machine](#).

One final behavior tree advantage is that they are really easy to make changes to. One of the reasons behavior trees became so popular is because they are easy to create with a visual editor. If you want to change the state execution order with a FSM you have to change the transitions between states. With a behavior tree, all you have to do is drag the task. You don't have to worry about transitions. Also, it is really easy to completely change how the AI reacts to different situations just by changing the tasks around or adding a new parent task to a branch of tasks.

With that said, behavior trees and FSMs don't have to be mutually exclusive. Behavior trees can describe the *flow* of the AI while the FSM describes the *function*. This combination gives you the power of behavior trees while still having the functionality of FSMs.

Installation

After Behavior Designer is imported you can access it from the Tools toolbar. You can access the runtime source code by extracting downloading and extracting the Runtime Source Code package located [here](#). Before you extract this package ensure that you have deleted the runtime and editor assemblies otherwise you'll get a compile error.

In order to compile Behavior Designer for the Universal Windows Platform (UWP) you must use the runtime source code instead of the compiled DLL. No compile settings need to be changed - Behavior Designer can compile with .Net Core enabled.

When you try to run your UWP app you may get an error indicating that the tasks cannot be found. In order to fix this the following line within TaskUtility.GetTypesWithinAssembly (within TaskUtility.cs) should be changed from:

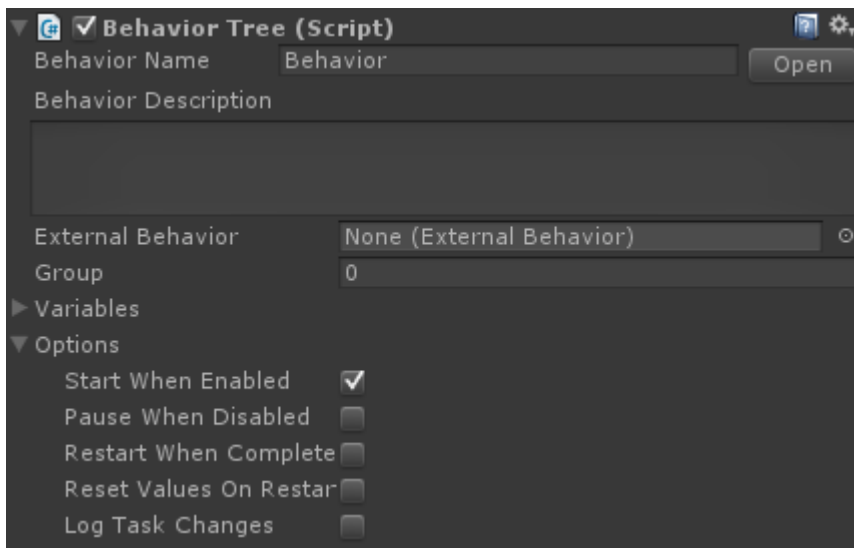
```
loadedAssemblies = GetStorageFileAssemblies(typeName).Result;
```

to:

```
loadedAssemblies = new List();  
loadedAssemblies.Add("Assembly-CSharp");
```

This will allow Unity's C# assembly to be found within the app.

Behavior Tree Component



The behavior tree component stores your behavior tree and acts as the interface between Behavior Designer and the tasks. The following API is exposed for starting and stopping your behavior tree:

```
public void EnableBehavior();
public void DisableBehavior(bool pause = false);
```

You can find tasks using one of the following methods:

```
TaskType FindTask<TaskType>();
List<TaskType> FindTasks<TaskType>();
Task FindTaskWithName(string taskName);
List<Task> FindTasksWithName(string taskName);
```

The current execution status of the tree can be obtained by calling:

```
behaviorTree.ExecutionStatus
```

A status of Running will be returned when the tree is running. When the tree finishes the execution status will be Success or Failure depending on the task results.

The following events can also be subscribed to:

```
OnBehaviorStart
OnBehaviorRestart
OnBehaviorEnd
```

The behavior tree component has the following properties:

Behavior Name

The name of the behavior tree

Behavior Description

Describes what the behavior tree does

External Behavior

A field to specify the external behavior tree that should be run when this behavior tree starts

Group

A numerical grouping of behavior trees. Can be used to easily find behavior trees. The CTF sample project shows an example of this

Start When Enabled

If true, the behavior tree will start running when the component is enabled

Pause When Disabled

If true, the behavior tree will pause when the component is disabled. If false, the behavior tree will end

Restart When Complete

If true, the behavior tree will restart from the beginning when it has completed execution. If false, the behavior tree will end

Reset Values On Restart

If true, the variables and task public variables will be reset to their original values when the tree restarts

Log Task Changes

Used for debugging. If enabled, the behavior tree will output any time a task status changes, such as it starting or stopping

Create a Behavior Tree from Script

In some circumstances you might want to create a behavior tree from script instead of directly relying on a prefab to contain the behavior tree for you. For example, you may have saved out an [external behavior tree](#) and want to load that tree in from a newly created behavior tree. This is possible by setting the externalBehavior variable on the behavior tree component:

```
using UnityEngine;
using BehaviorDesigner.Runtime;

public class CreateTree : MonoBehaviour
{
    public ExternalBehaviorTree behaviorTree;
```

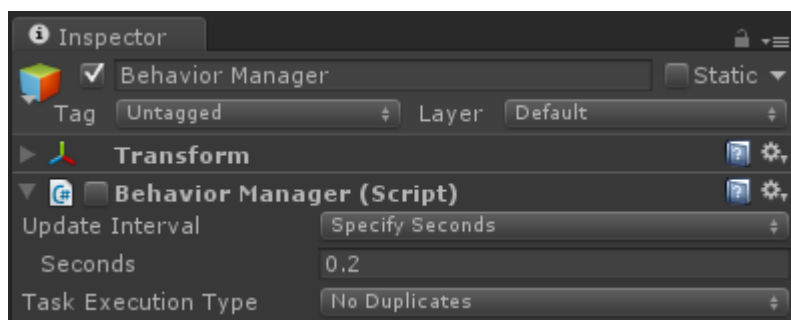
```

private void Start()
{
    var bt = gameObject.AddComponent<BehaviorTree>();
    bt.ExternalBehavior = behaviorTree;
    bt.StartWhenEnabled = false;
}
}

```

In this example the public variable `behaviorTree` contains a reference to your external behavior tree. When the newly created tree loads it will load the external behavior tree for all of its tasks. To prevent the tree from running immediately we set `StartWhenEnabled` to false. The tree can then be started manually with `bt.EnableBehavior()`.

Behavior Manager



When a behavior tree runs it creates a new `GameObject` with a `Behavior Manager` component if it isn't already created. This component manages the execution of all of the behavior trees in your scene.

You can control how often the behavior trees tick by changing the update interval property. "Every Frame" will tick the behavior trees every frame within the Update loop. "Specify Seconds" allows you to tick the behavior trees a given number of seconds. The final option is "Manual" which will give you the control of when to tick the behavior trees. You can tick the behavior trees by calling `tick`:

```
BehaviorManager.instance.Tick();
```

In addition, if you want each behavior tree to have its own tick rate you can tick each behavior tree manually with:

```
BehaviorManager.instance.Tick(BehaviorTree)
```

`Task Execution Type` allows you to specify if the behavior tree should continue executing tasks until it hits an already executed task during that tick or if it should continue to execute the tasks until a maximum number of tasks have been executed during that tick. As an example, consider the following behavior tree:



The Repeater task is set to repeat 5 times. If the Task Execute Type is set to No Duplicates, the Play Sound task will only execute once during a single tick. If the Task Execution Type is set to Count, a maximum task execution count can be specified. If a value of 5 is specified then the Play Sound task will execute all 5 times in a single tick.

Tasks

At the highest level a behavior tree is a collection of tasks. Tasks have a similar API to Unity's MonoBehaviour so it should be really easy to get started [writing your own tasks](#). The task class has the following API:

```
// OnAwake is called once when the behavior tree is enabled. Think of it as a constructor.
```

```
void OnAwake();
```

```
// OnStart is called immediately before execution. It is used to setup any variables that need to be reset from the previous run.
```

```
void OnStart();
```

```
// OnUpdate runs the actual task.
```

```
TaskStatus OnUpdate();
```

```
// OnEnd is called after execution on a success or failure.
```

```
void OnEnd();
```

```
// OnPause is called when the behavior is paused or resumed.
```

```
void OnPause(bool paused);
```

```
// Returns the priority of the task, used by the Priority Selector.
```

```
float GetPriority();
```

```
// Returns the utility of the task, used by the Utility Selector for Utility Theory.
```

```
float GetUtility();
```

```
// OnBehaviorComplete is called after the behavior tree finishes executing.
```

```
void OnBehaviorComplete();
```

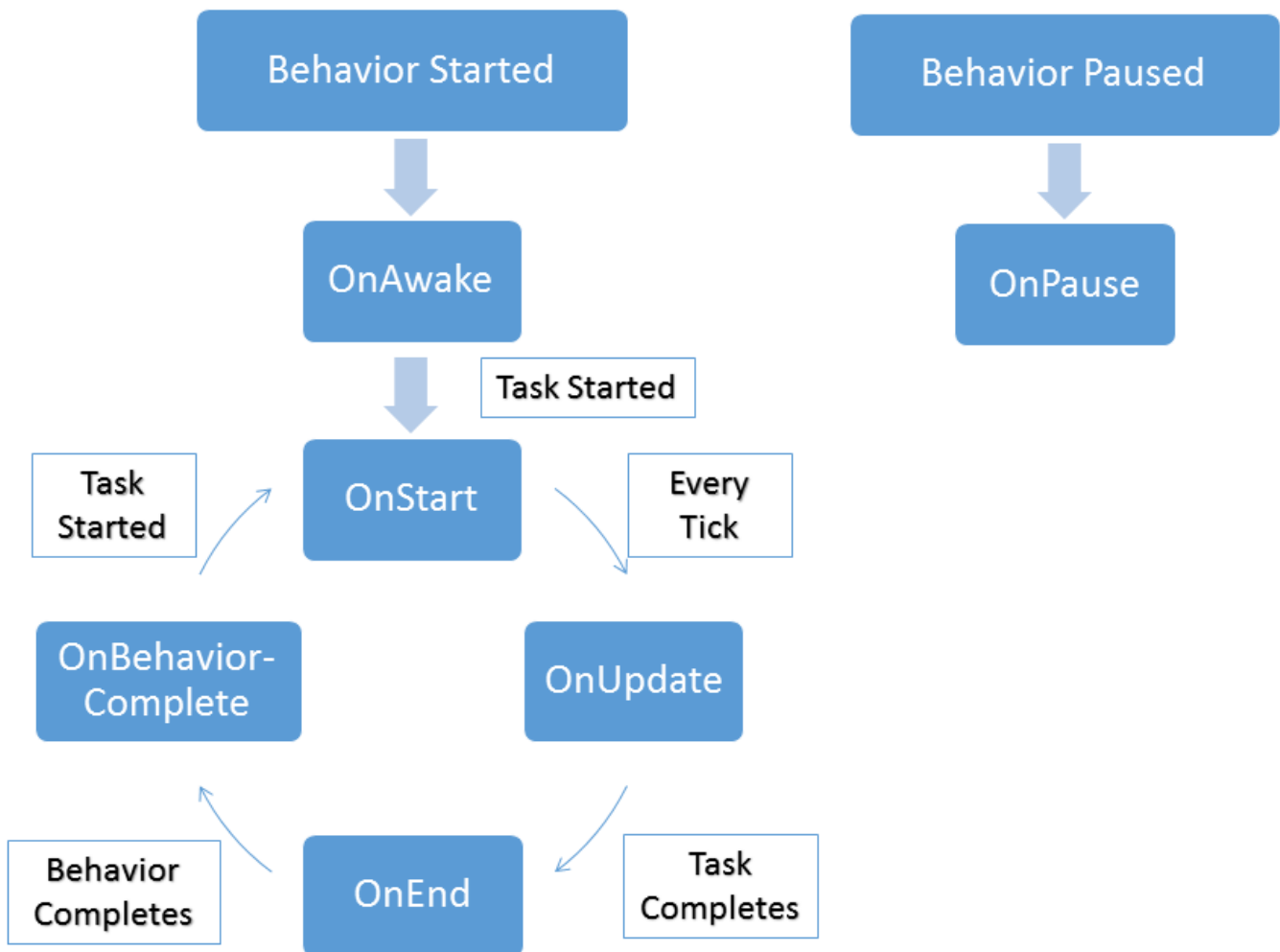
```
// OnReset is called by the inspector to reset the public properties
void OnReset();

// Allow OnDrawGizmos to be called from the tasks.
void OnDrawGizmos();

// Keep a reference to the behavior that owns this task.
Behavior Owner;
```

Tasks have three exposed properties: name, comment, and instant. Instant is the only property that isn't obvious in what it does. When a task returns success or fail it immediately moves onto the next task within the same update tick. If you uncheck the instant task it will now wait a update tick before the next task gets executed. This is an easy way to throttle the behavior tree.

The following flow chart is used when executing the task:



Parent Tasks

Parent Tasks are the composite and decorator tasks within the behavior tree. While the ParentTask API has no equivalent API to Unity's MonoBehaviour class, it is still pretty easy to determine what each method is used for.

```
// The maximum number of children a parent task can have. Will usually
```



```

be 1 or int.MaxValue
public virtual int MaxChildren();

// Boolean value to determine if the current task is a parallel task.
public virtual bool CanRunParallelChildren();

// The index of the currently active child.
public virtual int CurrentChildIndex();

// Boolean value to determine if the current task can execute.
public virtual bool CanExecute();

// Apply a decorator to the executed status.
public virtual TaskStatus Decorate(TaskStatus status);

// Notifies the parent task that the child has been executed and has a
status of childStatus.
public virtual void OnChildExecuted(TaskStatus childStatus);

// Notifies the parent task that the child at index childIndex has
been executed and has a status of childStatus.
public virtual void OnChildExecuted(int childIndex, TaskStatus
childStatus);

// Notifies the task that the child has started to run.
public virtual void OnChildStarted();

// Notifies the parallel task that the child at index childIndex has
started to run.
public virtual void OnChildStarted(int childIndex);

// Some parent tasks need to be able to override the status, such as
parallel tasks.
public virtual TaskStatus OverrideStatus(TaskStatus status);

// The interrupt node will override the status if it has been
interrupted.
public virtual TaskStatus OverrideStatus();

// Notifies the composite task that an conditional abort has been
triggered and the child index should reset.
public virtual void OnConditionalAbort(int childIndex);

```

Writing a New Conditional Task

This topic is divided into two parts. The first part describes writing a new conditional task, and the second part ([available here](#)) describes writing a new action task. The conditional task will determine if any objects are within sight and the action class will towards the

object that is within sight. We will also be using [variables](#) for both of these tasks. We have also recorded a video on this topic and it is available on [YouTube](#).

The first task that we will write is the Within Sight task. Since this task will not be changing game state and is just checking the status of the game this task will be derived from the Conditional task. Make sure you have the BehaviorDesigner.Runtime.Tasks namespace included:

```
using UnityEngine;
using BehaviorDesigner.Runtime.Tasks;

public class WithinSight : Conditional
{
}
```

We now need to create three public variables and one private variable:

```
using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

public class WithinSight : Conditional
{
    public float fieldOfViewAngle;
    public string targetTag;
    public SharedTransform target;

    private Transform[] possibleTargets;
}
```

The *field of view angle* is the field of view that the object can see. *Target tag* is the tag of the targets that the object can move towards. *Target* is a [Shared Variable](#) which will be used by both the Within Sight and the Move Towards tasks. If you are using Shared Variables make sure you include the BehaviorDesigner.Runtime namespace. The final variable, *possible targets*, is a cache of all of the Transforms with the *target tag*. If you take a look at the [task API](#), you can see that we can create that cache within the OnAwake or OnStart method. Since the list of possible transforms are not going to be changing as the Within Sight task is enabled/disabled we are going to do the caching within OnAwake:

```
public override void OnAwake()
{
    var targets = GameObject.FindGameObjectsWithTag(targetTag);
    possibleTargets = new Transform[targets.Length];
    for (int i = 0; i < targets.Length; ++i) {
        possibleTargets[i] = targets[i].transform;
    }
}
```

This OnAwake method will find all of the GameObjects with the *target tag*, then loop through them caching their transform in the *possible targets* array. The *possible targets*

array is then used by the overridden OnUpdate method:

```
public override TaskStatus OnUpdate()
{
    for (int i = 0; i < possibleTargets.Length; ++i) {
        if (WithinSight(possibleTargets[i], fieldOfViewAngle)) {
            target.Value = possibleTargets[i];
            return TaskStatus.Success;
        }
    }
    return TaskStatus.Failure;
}
```

Every time the task is updated it checks to see if any of the *possible targets* are within sight. If one target is within sight it will set the target value and return success. Setting this target value is key as this allows the Move Towards task to know what direction to move in. If there are no targets within sight then the task will return failure. The last part of this task is the WithinSight method:

```
public bool WithinSight(Transform targetTransform, float
fieldOfViewAngle)
{
    Vector3 direction = targetTransform.position -
transform.position;
    return Vector3.Angle(direction, transform.forward) <
fieldOfViewAngle;
}
```

This method first gets a direction vector between the current transform and the *target transform*. It will then compute the angle between the direction vector and the current forward vector to determine the angle. If that angle is less than *field of view angle* then the *target transform* is within sight of the current Transform.

That's it for the Within Sight task. Here's what the full task looks like:

```
using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

public class WithinSight : Conditional
{
    // How wide of an angle the object can see
    public float fieldOfViewAngle;
    // The tag of the targets
    public string targetTag;
    // Set the target variable when a target has been found so the
subsequent tasks know which object is the target
    public SharedTransform target;

    // A cache of all of the possible targets
```

```

private Transform[] possibleTargets;

public override void OnAwake()
{
    // Cache all of the transforms that have a tag of targetTag
    var targets = GameObject.FindGameObjectsWithTag(targetTag);
    possibleTargets = new Transform[targets.Length];
    for (int i = 0; i < targets.Length; ++i) {
        possibleTargets[i] = targets[i].transform;
    }
}

public override TaskStatus OnUpdate()
{
    // Return success if a target is within sight
    for (int i = 0; i < possibleTargets.Length; ++i) {
        if (WithinSight(possibleTargets[i], fieldOfViewAngle)) {
            // Set the target so other tasks will know which transform
            is within sight
            target.Value = possibleTargets[i];
            return TaskStatus.Success;
        }
    }
    return TaskStatus.Failure;
}

// Returns true if targetTransform is within sight of current
transform
public bool WithinSight(Transform targetTransform, float
fieldOfViewAngle)
{
    Vector3 direction = targetTransform.position -
transform.position;
    // An object is within sight if the angle is less than field of
view
    return Vector3.Angle(direction, transform.forward) <
fieldOfViewAngle;
}
}

```

Continue to the second part of this topic, [writing the Move Towards task](#).

Writing a New Action Task

This topic is a continuation of the previous topic. It is recommended that you first take a look at the [writing a new conditional task](#) topic first.

The next task that we are going to write is the Move Towards task. Since this task is going to be changing the game state (moving an object from one position to another), we will derive the task from the Action class:

```
using UnityEngine;
using BehaviorDesigner.Runtime.Tasks;

public class MoveTowards : Action
{
}
```

This class will only need two variables: a way to set the speed and the transform of the object that we are targeting:

```
using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

public class MoveTowards : Action
{
    public float speed = 0;
    public SharedTransform target;
}
```

The *target* variable is a SharedTransform and it will be set from the Within Sight task that will run just before the Move Towards task. To do the actual movement, we will need to override the OnUpdate method:

```
    public override TaskStatus OnUpdate()
    {
        if (Vector3.SqrMagnitude(transform.position -
target.Value.position) < 0.1f) {
            return TaskStatus.Success;
        }
        transform.position = Vector3.MoveTowards(transform.position,
target.Value.position, speed * Time.deltaTime);
        return TaskStatus.Running;
    }
```

When the OnUpdate method is run, it will check to see if the object has reached the *target*. If the object has reached the *target* then the task will success. If the *target* has not been reached yet the object will move towards the *target* at a speed specified by the *speed* variable. Since the object hasn't reached the target yet the task will return running.

That's the entire Move Towards task. The full task looks like:

```
using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;
```

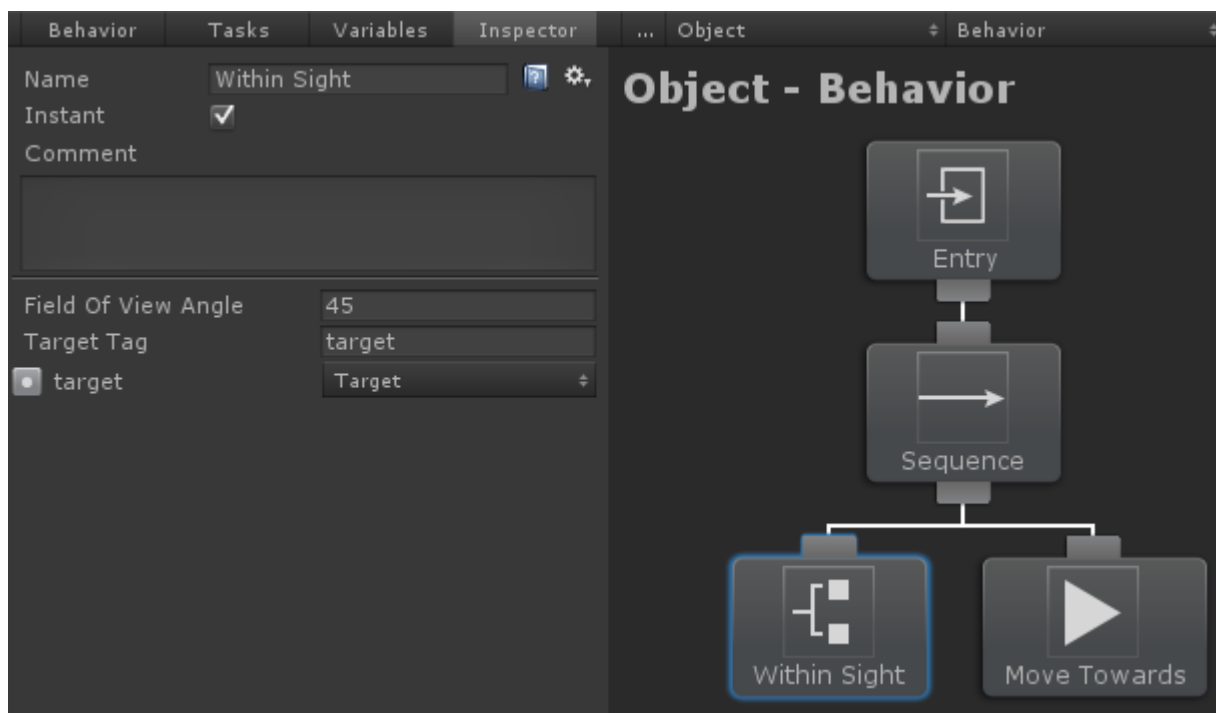
```

public class MoveTowards : Action
{
    // The speed of the object
    public float speed = 0;
    // The transform that the object is moving towards
    public SharedTransform target;

    public override TaskStatus OnUpdate()
    {
        // Return a task status of success once we've reached the target
        if (Vector3.SqrMagnitude(transform.position -
target.Value.position) < 0.1f) {
            return TaskStatus.Success;
        }
        // We haven't reached the target yet so keep moving towards it
        transform.position = Vector3.MoveTowards(transform.position,
target.Value.position, speed * Time.deltaTime);
        return TaskStatus.Running;
    }
}

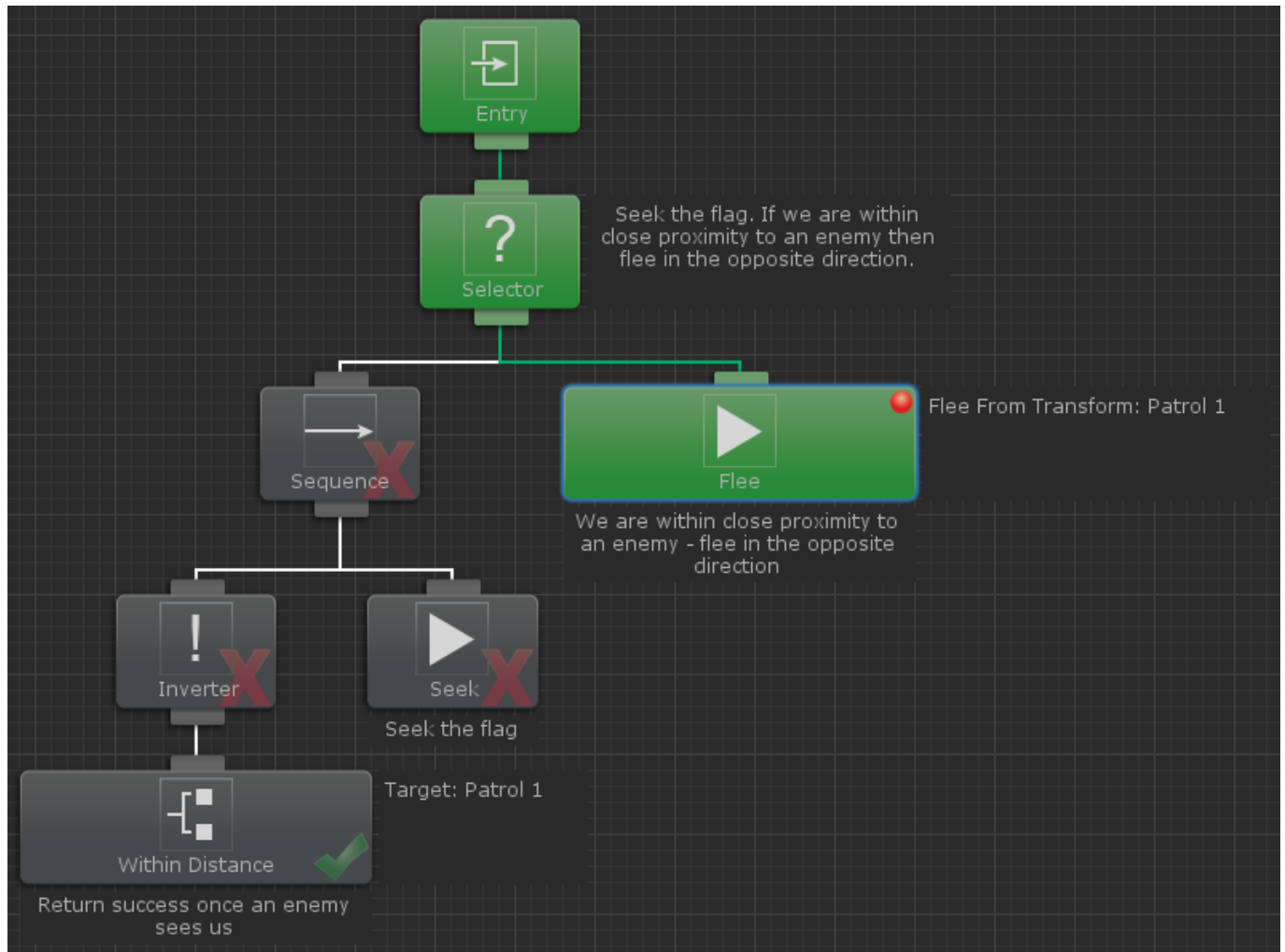
```

Now that these two tasks are written, parent the tasks by a sequence task and set the variables within the task inspector. Make sure you've also created a new variable within Behavior Designer:



That's it! Create a few moving GameObjects within the scene assigned with the same tag as targetTag. When the game starts the object with the behavior tree attached with move towards whatever object first appears within its field of view. This was a pretty basic example and the tasks can get a lot more complicated depending on what you want them to do. All of the tasks within the sample projects are well commented so you should be able to pick it up from there. In addition, we have written some more documentation on the continuing topics such as [variables](#), [referencing tasks](#) and [task attributes](#).

Debugging



When a behavior tree is running you will see different tasks change colors between gray and green. When the task is green that means it is currently executing. When the task is gray it is not executing. After the task has executed it will have a check or x on the bottom right corner. If the task returned success then a check will be displayed. If it returned failure then an x will be displayed. While tasks are executing you can still change the values within the

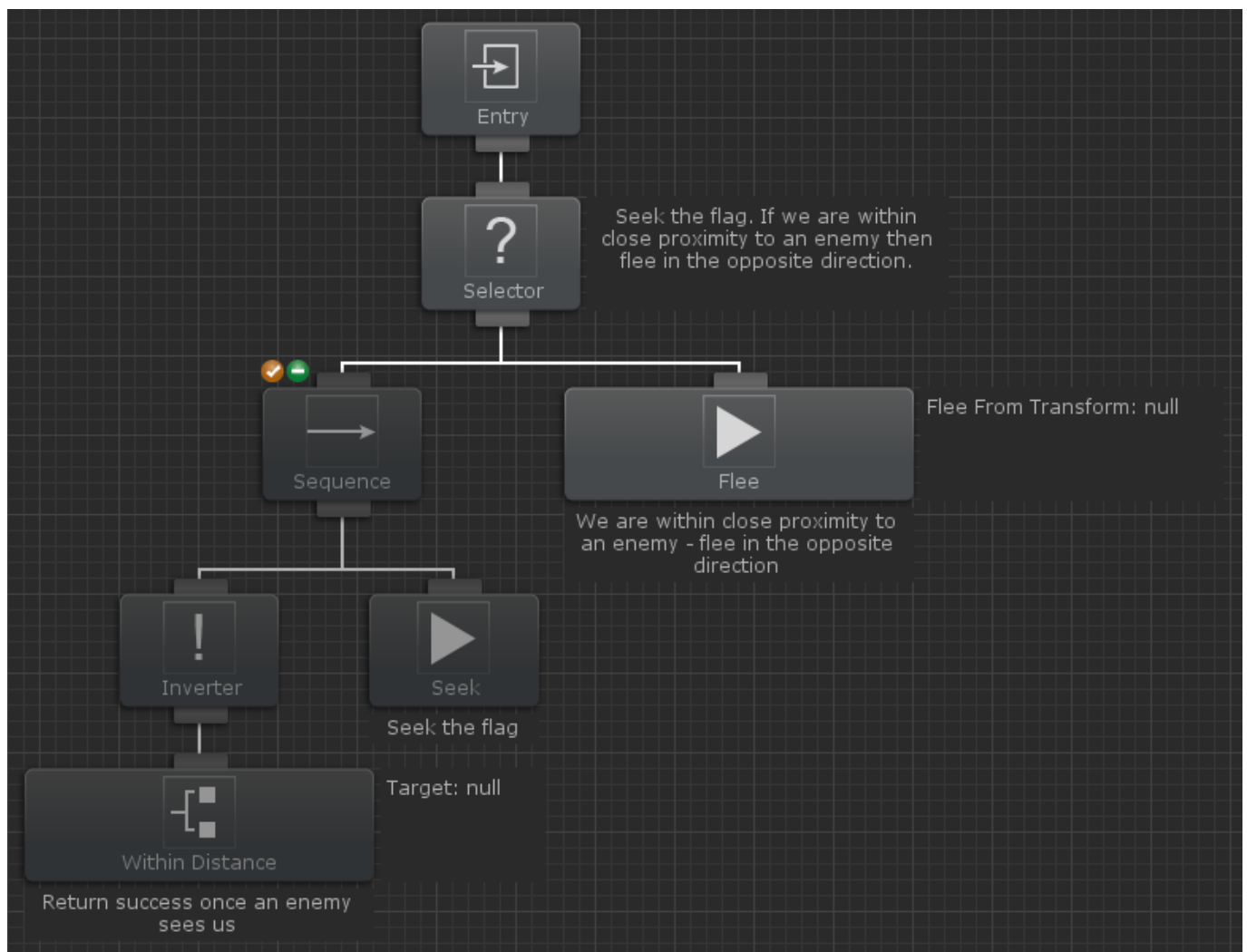
inspector and that change will be reflected in game.



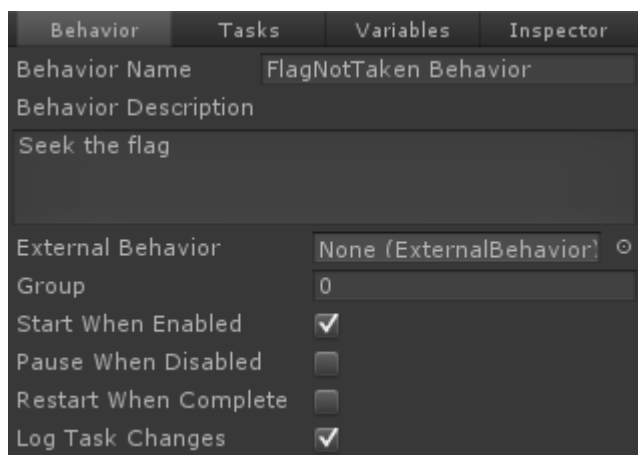
Right clicking on a task will bring up a menu which allows you to set a breakpoint. If a breakpoint is set on a particular task then Behavior Designer will pause Unity whenever that task is activated. This is useful if you want to see when a particular task is executed.



When a task is selected you have the option of watching a variable within the graph by clicking on the magnifying glass to the left of the variable name. Watched variables are a good way to see the value of a particular variable without having to have the task inspector open. In the example above the variables “Fleed Distance” and “Flee From Transform” are being watched and appear to the right of the Flee task.



Sometimes you only want to focus on a certain set of tasks and prevent the rest from running. This is possible by disabling a set of tasks. Tasks can be disabled by hovering over the task and selecting the orange X on the top left of the task. Disabled tasks will not run and return success immediately. Disabled tasks appear in a darker color than the enabled tasks within the graph.



One more debugging option is to output to the console any time a task changes state. If *Log Task Changes* is enabled then you'll see output to the log similar to the following:

```
GameObject - Behavior: Push task Sequence (index 0) at stack index 0
GameObject - Behavior: Push task Wait (index 1) at stack index 0
GameObject - Behavior: Pop task Wait (index 1) at stack index 0 with
status Success
GameObject - Behavior: Push task Wait (index 2) at stack index 0
GameObject - Behavior: Pop task Wait (index 2) at stack index 0 with
status Success
GameObject - Behavior: Pop task Sequence (index 0) at stack index 0
with status Success
Disabling GameObject - Behavior
```

These messages can be broken up into the following pieces:

```
{game object name } - {behavior name}: {task change} {task type}
(index {task index}) at stack index {stack index} {optional status}
```

{game object name} is the name of the game object that the behavior tree is attached to.

{behavior name} is the name of the behavior tree.

{task change} indicates the new status of the task. For example, a task will be pushed onto the stack when it starts executing and it will be popped when it is done executing .

{task type} is the class type of the task.

{task index} is the index of the task in a depth first search.

{stack index} is the index of the stack that the task is being pushed to. If you have a parallel node then you'll be using multiple stacks.

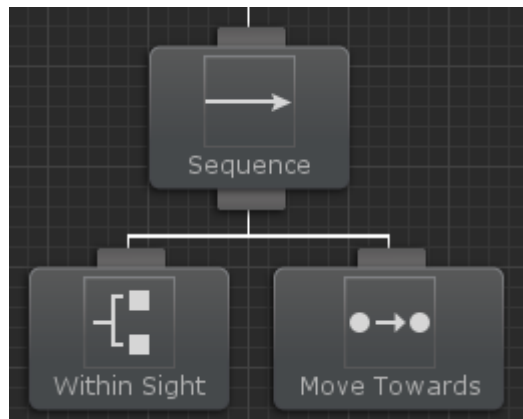
{optional status} is any extra status for that particular change. The pop task will output the task status.

Variables

One of the advantages of behavior trees are that they are very flexible in that all of the tasks are loosely coupled - meaning one task doesn't depend on another task to operate. The drawback of this is that sometimes you need tasks to share information with each other. For example, you may have one task that is determine if a target is Within Sight. If the target is

within sight you might have another task Move Towards the target. In this case the two tasks need to communicate with each other so the Move Towards task actually moves in the direction of the same object that the Within Sight task found. In traditional behavior tree implementations this is solved by coding a blackboard. With Behavior Designer it is a lot easier in that you can use variables.

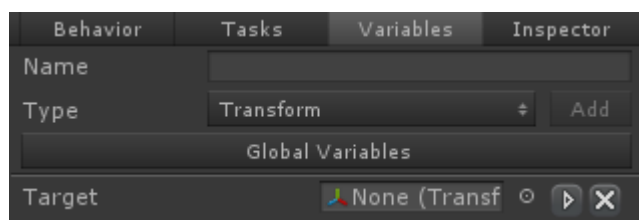
In our previous example we had two tasks: one that determined if the target is within sight and then the other task moves towards the target. This tree looks like:



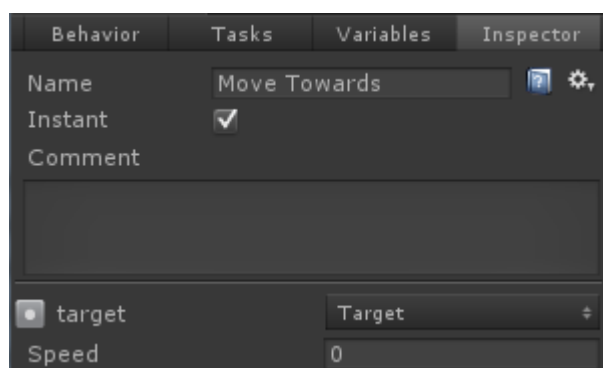
The code for both of these tasks is discussed in the [Writing a New Task](#) topic, but the part that deals with variables is in this variable declaration:

```
public SharedTransform target;
```

With the SharedTransform variable created, we can now create a new variable within Behavior Designer and assign that variable to the two tasks:

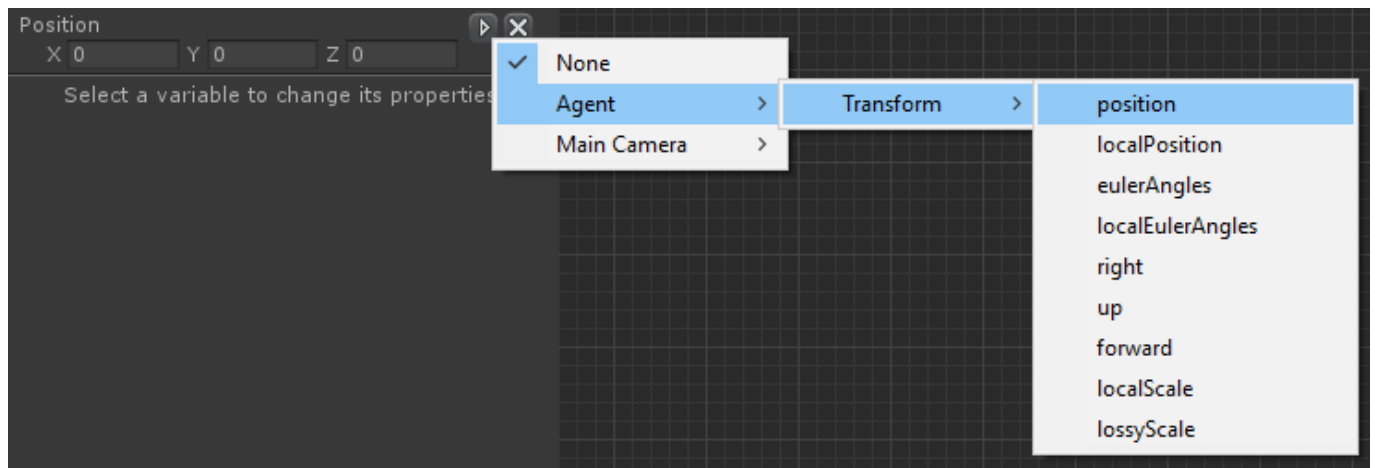


Switch to the task inspector and assign that variable to the two tasks:



And with that the two tasks can start to share information! You can get/set the value of the shared variable by accessing the Value property. For example, `target.Value` will return the transform object. When Within Sight runs it will assign the transform of the object that comes within sight to the Target variable. When Move Towards runs it will use that Target variable to determine what position to move towards.

Looking at the variable within the inspector, if you look to the left of the delete button you'll see a triangle pointing to the right. This is the button for Variable Mappings.



Variable Mappings allow your SharedVariable to map to a property of the same type. This allows you to quickly get or set a value on a MonoBehaviour component. As an example, let's say that we want to get the position of the agent. It is possible to use the Get Position task with a non-mapped variable, but that adds unnecessary tasks to your behavior tree. Instead, if you map the variable to the Transform.position property, whenever the value of the variable is accessed, it will instead use the property that it is mapped to. This allows you to get or set the position of the Transform without any extra tasks.

Behavior Designer supports both local and global variables. [Global Variables](#) are similar to local variables except any tree can reference the same variable. Variables can be referenced by non-Task derived classes by [getting a reference](#) to from the behavior tree.

The following shared variable types are included in the default Behavior Designer installation. If none of these types are suitable for your situation then you can [create your own shared variable](#):

- SharedAnimationCurve
- SharedBool
- SharedColor
- SharedFloat
- SharedGameObject
- SharedGameObjectList
- SharedInt
- SharedMaterial
- SharedObject
- SharedObjectList
- SharedQuaternion
- SharedRect
- SharedString
- SharedTransform
- SharedTransformList
- SharedVector2
- SharedVector3Int
- SharedVector3
- SharedVector3Int

- SharedVector4

Dynamic Variables

Dynamic variables allow you to use temporary variables that are limited in scope. These variables are great if you want to share data between a limited number of tasks and do not need to access the variable outside of those tasks. Dynamic variables can be created by clicking the circle to the right of the variables field and then selecting “(Dynamic)”:



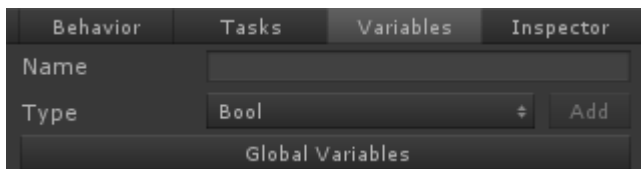
After the dynamic variable has been created you can then type the name of the variable.



The dynamic variable will now be used within your tree. The dynamic variable will have the same value for any fields that reference the same dynamic variable name. The name is case sensitive and should not be the same as a local variable within the tree.

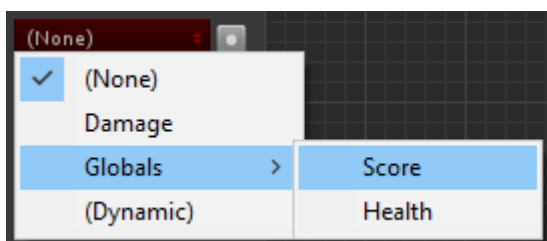
Global Variables

Global variables are similar to local variables except any behavior tree can access an instance of the same variable. To access global variables, navigate to the Window->Behavior Designer->Global Variables menu option or from within the Variables pane:



When a global variable is first added an asset file is created which stores all of the global variables. This file is created at /Behavior Designer/Resources/BehaviorDesignerGlobalVariables.asset. You can move this file as long as it is still located in a Resources folder.

Global variables are assigned in a very similar way as local variables. In the task inspector, when you are assigning a global variable the global variables are located under the “Globals” menu item:



Global variables can also be [accessed from non-Task derived objects](#).

Creating Shared Variables

New Shared Variables can be created if you don't want to use any of the built in types. To create a Shared Variable, subclass the SharedVariable type and implement the following methods. The keyword OBJECT_TYPE should be replaced with the type of Shared Variable that you want to create.

```
[System.Serializable]
public class SharedOBJECT_TYPE : SharedVariable<OBJECT_TYPE>
{
    public static implicit operator SharedOBJECT_TYPE(OBJECT_TYPE
value) { return new SharedOBJECT_TYPE { Value = value }; }
}
```

It is important that the "Value" property exists. The variable inspector will show an error if the new Shared Variable is created incorrectly. Shared Variables can contain any type of object that your task can contain, including primitives, arrays, lists, custom objects, etc.

As an example, the following script will allow a custom class to be shared:

```
[System.Serializable]
public class CustomClass
{
    public int myInt;
    public Object myObject;
}

[System.Serializable]
public class SharedCustomClass : SharedVariable<CustomClass>
{
    public static implicit operator SharedCustomClass(CustomClass
value) { return new SharedCustomClass { Value = value }; }
}
```

Accessing Variables from non-Task Objects

Variables are normally referenced by [assigning](#) the variable name to the task field within the Behavior Designer inspector panel. Local variables can also be accessed by non-Task derived classes (such as MonoBehaviour) by calling the method:

```
behaviorTree.GetVariable("MyVariable");
behaviorTree.SetVariable("MyVariable", value);
behaviorTree.SetVariableValue("MyVariableName", value);
```

When setting a variable, if you want the tasks to automatically reference that variable then

make sure a variable is created with that name ahead of time. The following code snippet shows an example of modifying a variable from a MonoBehaviour class:

```
using UnityEngine;
using BehaviorDesigner.Runtime;

public class AccessVariable : MonoBehaviour
{
    public BehaviorTree behaviorTree;

    public void Start()
    {
        var myIntVariable =
(SharedInt)behaviorTree.GetVariable("MyVariable");
        myIntVariable.Value = 42;
    }
}
```

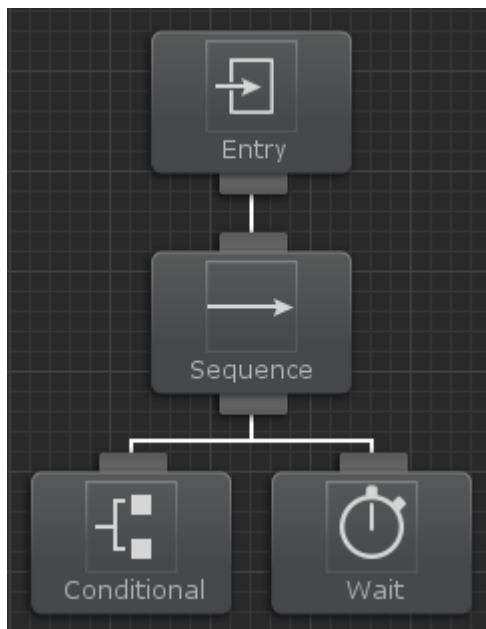
In the above example we are getting a reference to the variable named “MyVariable” within the Behavior Designer Variables pane. Also, as shown in the example, you can get and set the value of the variable with the SharedVariable.Value property.

Similarly, global variables can be accessed by getting a reference to the GlobalVariable instance:

```
GlobalVariables.Instance.GetVariable("MyVariable");
GlobalVariables.Instance.SetVariable("MyVariable", value);
```

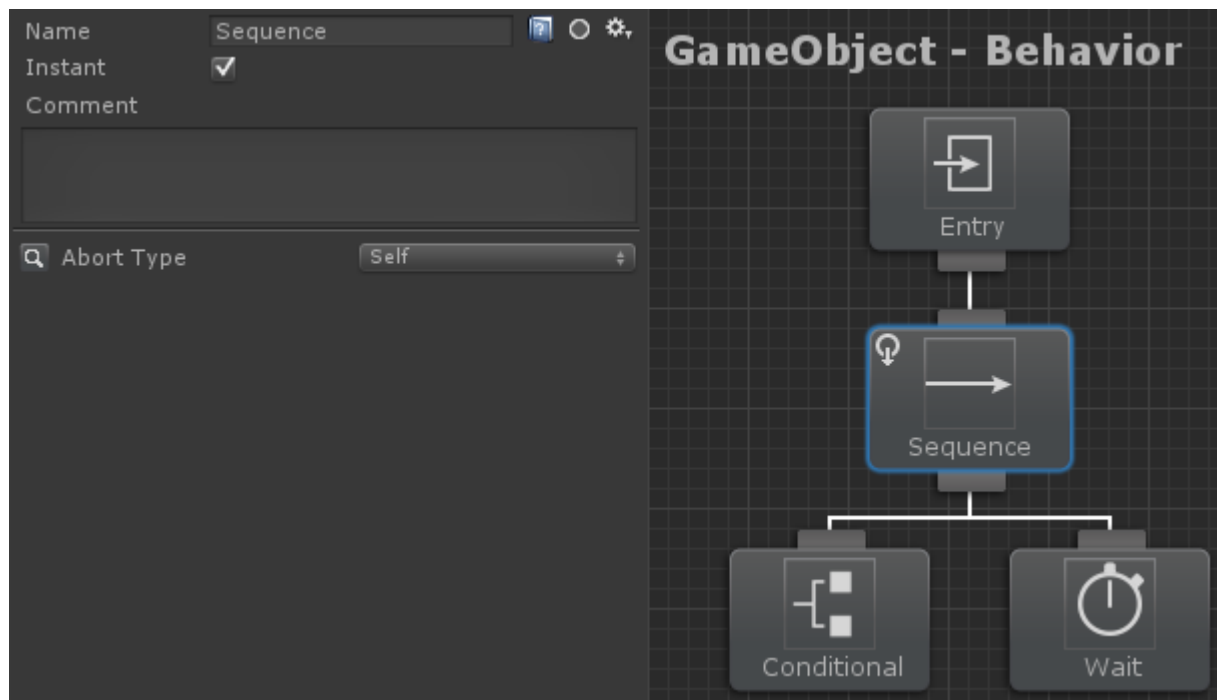
Conditional Aborts

Conditional aborts allow your behavior tree to dynamically respond to changes without having to clutter your behavior tree with many Interrupt/Perform Interrupt tasks. This feature is similar to the Observer Aborts in Unreal Engine 4. Most behavior tree implementations reevaluate the entire tree every tick. Conditional aborts are an optimization to prevent having to rerun the entire tree. As a basic example, consider the

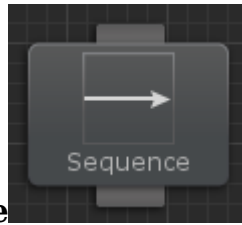


following tree:

When this tree runs the Conditional task will return success and the Sequence task will start running the next child, the Wait task. The Wait task has a wait duration of 10 seconds. While the wait task is running, lets say that the conditional task changes changes state and now returns failure. If Conditional aborts are enabled, the Conditional task will issue an abort and stop the Wait task from running. The Conditional task will be reevaluated and the next task will run according to the standard behavior tree rules. Conditional aborts can be accessed from any Composite task:

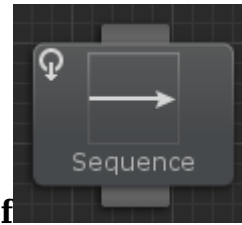


There are four different abort types: None, Self, Lower Priority, and Both.



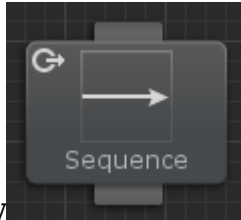
None

This is the default behavior. The Conditional task will not be reevaluated and no aborts will be issued.



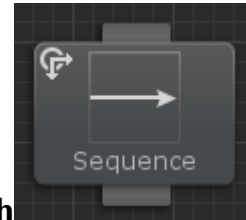
Self

This is a self contained abort type. The Conditional task can only abort an Action task if they both have the same parent Composite task.



Lower Priority

Behavior trees can be organized from more important tasks to least important. If a more important Conditional task changes status then can issue an abort that will stop the lower priority tasks from running.



Both

This abort type combines both self and lower priority.

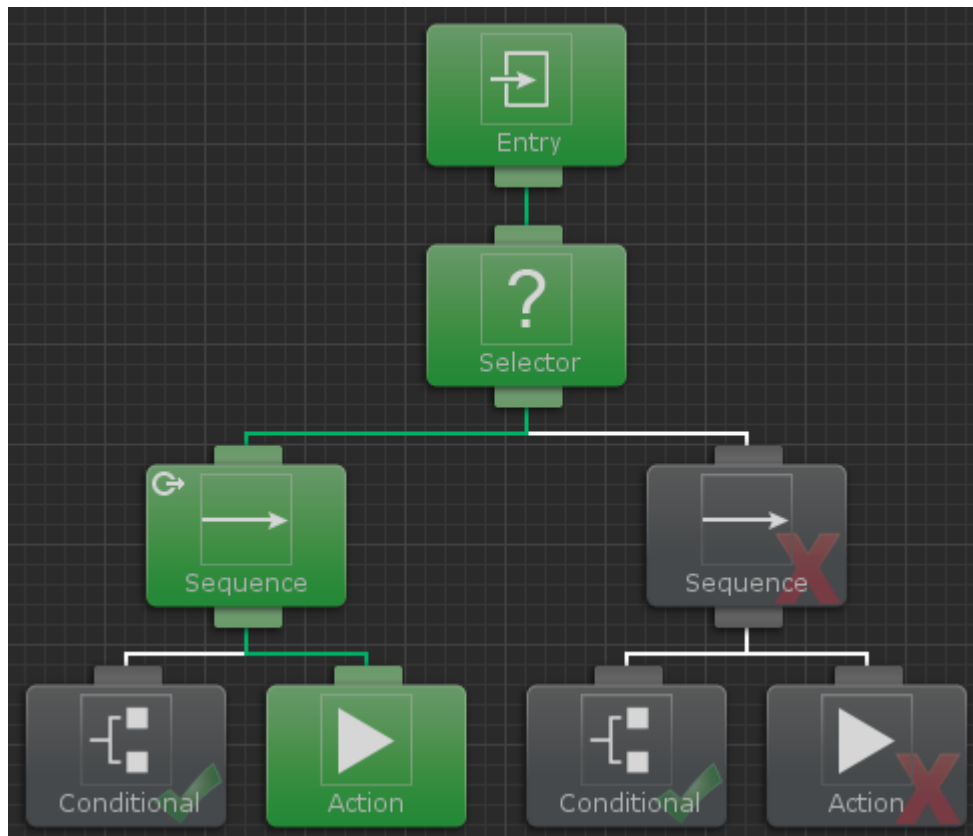
Conditional aborts can also be thought of the following way:

Lower Priority: will reevaluate when any task to the right of the current branch is active.

Self: will reevaluate when any task within the current branch is active.

Both: will reevaluate when any task to the right or within the current branch is active.

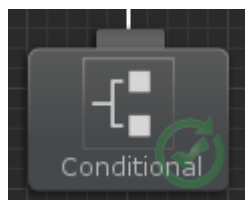
The following example will use the lower priority abort type:



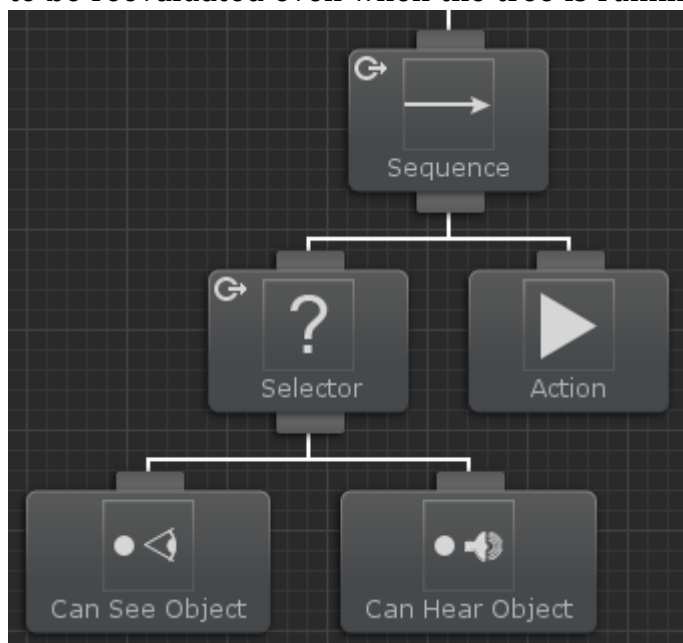
In this example the parent Sequence task of the left branch has an abort type of lower | 32

priority. Lets say that the left branch fails and moves the tree onto the right branch due to the Selector parent task. While the right branch is running, the very first Conditional task changes status to success. Because the task status changed and the abort type was lower priority the Action task that is currently running gets aborted and the original Conditional task is rerun.

The conditional task's execution status will have a repeater icon around the success or failure status to indicate that it is being reevaluated by a conditional abort:



Conditional aborts can be nested beneath one another as well. For example, you may want to run a branch when one of two conditions succeed, but they both don't have to. In this example we will be using the Can See Object and Can Hear Object tasks. You want to run the action task when the object is either seen or heard. To do this, these two conditional tasks should be parented by a Selector with the lower priority abort type. The action task is then a sibling of the Selector task. A Sequence task is then parented to these two tasks because the action task should only run when either of the conditional tasks succeed. The Sequence task is set to a Lower Priority abort type so the two conditional tasks will continue to be reevaluated even when the tree is running a completely different branch.



The important thing to note with this tree is that the Selector task must have an abort type set to Lower Priority (or Both). If it does not have an abort type set then the two conditional tasks would not be reevaluated.

Events

The event system within Behavior Designer allows your behavior trees to easily react to changes. This event system can trigger an event via code or through behavior tree tasks.

Events can be signaled through the behavior tree with the Send Event and the Has Received Event tasks. When an event should be signaled, the Send Event task should be used. The Has Received Event task is a conditional task and will return success as soon as the event has been received. An event name can be specified for both of these tasks.

In addition to being able to send events via the behavior tree, events can be sent through code. The BehaviorTree.SendEvent method will allow you to send an event to the specified behavior tree. For example:

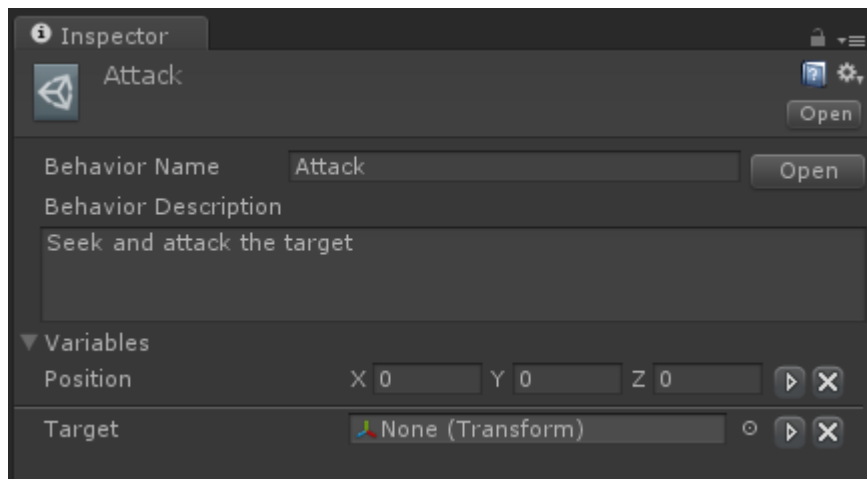
```
var behaviorTree = GetComponent<BehaviorTree>();  
behaviorTree.SendEvent<object>("MyEvent", Vector3.zero);
```

In this example the “MyEvent” event will be sent to the behavior tree component with a parameter value of Vector3.zero. If the behavior tree contains the Has Received Event task then it will react accordingly. If the Has Received Event task is receiving event then the template type must be a valid of object.

You are also able to receive events from outside the behavior tree. To continue with the “MyEvent” example, you can receive this event by using the BehaviorTree.RegisterEvent method. BehaviorTree.UnregisterEvent will stop listening for that event.

```
public void OnEnable()  
{  
    var behaviorTree = GetComponent<BehaviorTree>();  
    behaviorTree.RegisterEvent<object>("MyEvent", ReceivedEvent);  
}  
  
public void ReceivedEvent(object arg1)  
{  
  
}  
  
public void OnDisable()  
{  
    var behaviorTree = GetComponent<BehaviorTree>();  
    behaviorTree.UnregisterEvent<object>("MyEvent", ReceivedEvent);  
}
```

External Behavior Trees



In some cases you may have a behavior tree that you want to run from multiple objects. For example, you could have a behavior tree that patrols a room. Instead of creating a separate behavior tree for each unit you can instead use an external behavior tree. An external behavior tree is referenced using the Behavior Tree Reference task. When the original behavior tree starts running it will load all of the tasks within the external behavior tree and act like they are its own. Any SharedVariable within the external tree of the same name and type of its parent tree will automatically be overridden. For example, if the parent tree has a SharedInt named "MyInt" with a value of 7, and the External Tree has a SharedInt named "MyInt" with a value of 0, when the tree runs MyInt will have a value of 7 within the External Tree.

Behavior Tree Reference

The Behavior Tree Reference task allows you to run another behavior tree within the current behavior tree. You can create this behavior tree by saving the tree as an external behavior tree. One use for this is that if you have an unit that plays a series of tasks to attack. You may want the unit to attack at different points within the behavior tree, and you want that attack to always be the same. Instead of copying and pasting the same tasks over and over you can just use an external behavior and then the tasks are always guaranteed to be the same. This example is demonstrated in the RTS sample project located on the [samples page](#).

The GetExternalBehaviors method allows you to override it so you can provide an external behavior tree array that is determined at runtime.

The Behavior Tree Reference task allows you to specify variables per-reference task. For most cases this is not needed because variables will automatically be transferring from the parent tree to the External Tree. However, if you have multiple Behavior Tree Reference tasks in the same tree all pointing to the same External Tree, you may want to be able to specify different variable values per tree. In this situation you can specify the variable value on the Behavior Tree Reference task and that variable value will be transferred to the External Tree.

Pooling

External behavior trees can be pooled for better performance when switching between many external trees. When the external behavior tree is instantiated the Init method should be called to deserialize the external behavior tree. Pooled external behavior trees can then

be assigned to the behavior tree component just like non pooled external behavior trees. For example:

```
using UnityEngine;
using BehaviorDesigner.Runtime;

public class ExternalPoolExample : MonoBehaviour {
    public BehaviorTree behaviorTree;
    public ExternalBehavior externalBehaviorTree;

    private ExternalBehavior[] externalPool;
    private int index;

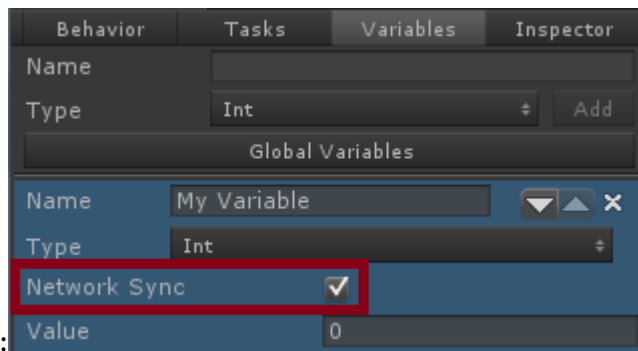
    public void Awake()
    {
        // Instantiate five External Behavior Tree which will be
        reused. The Init method will deserialize the External Behavior Tree.
        externalPool = new ExternalBehavior[5];
        for (int i = 0; i < externalPool.Length; ++i) {
            externalPool[i] =
Object.Instantiate(externalBehaviorTree);
            externalPool[i].Init();
        }
    }

    public void OnGUI()
    {
        // Assign the next External Behavior Tree within the
        simplified pool.
        if (GUILayout.Button("Assign")) {
            behaviorTree.DisableBehavior();
            behaviorTree.ExternalBehavior = externalPool[index];
            behaviorTree.EnableBehavior();
            index = (index + 1) % externalPool.Length;
        }
    }
}
```

Networking

Behavior Designer supports Unity's networking system introduced with Unity 5.1. Networking is a complex topic so it is highly recommended that you first go through Unity's [networking documentation](#) before continuing.

Shared Variables can automatically be synchronized over the network from the server to the client. This can be enabled by opening the details of the variable and selecting "Network



Sync”:

The `ENABLE_MULTIPPLAYER` [compiler definition](#) must be added for the variables to sync correctly.

Due to a current networking limitation, ClientRPC calls cannot be overloaded so the type has to be known ahead of time. This means that only the following types of variables can be synchronized:

- bool
- Color
- float
- GameObject
- int
- Quaternion
- Rect
- string
- Transform
- Vector2
- Vector3
- Vector4

The Behavior component from the [runtime source code](#) must be used in order to allow the variables to be synchronized. This is a result of a current Unity networking bug. ClientRPC calls cannot be called on the parent class from a subclass. A bug report has been submitted and are waiting for a fix. This bug manifests itself by displaying the following warning in the console:

Failed to invoke RPC [RpcPath]([ObjectID]) on netID [NetID]

Where [RpcPath] is the method path, [ObjectID] is the ID of the object, and [NetID] is the network ID.

Referencing Tasks

When writing a new task, in some cases it is necessary to access another task within that task. For example, TaskA may want to get the value of TaskB.SomeFloat. To accomplish this, TaskB needs to be referenced from TaskA. In this example TaskA looks like:

```
using UnityEngine;
using BehaviorDesigner.Runtime.Tasks;
```

```
public class TaskA : Action
{
    public TaskB referencedTask;

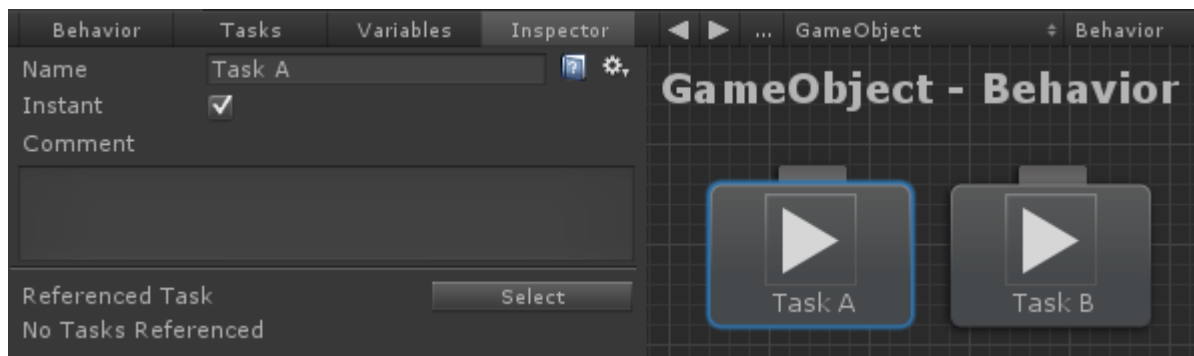
    public void OnAwake()
    {
        Debug.Log(referencedTask.SomeFloat);
    }
}
```

TaskB then looks like:

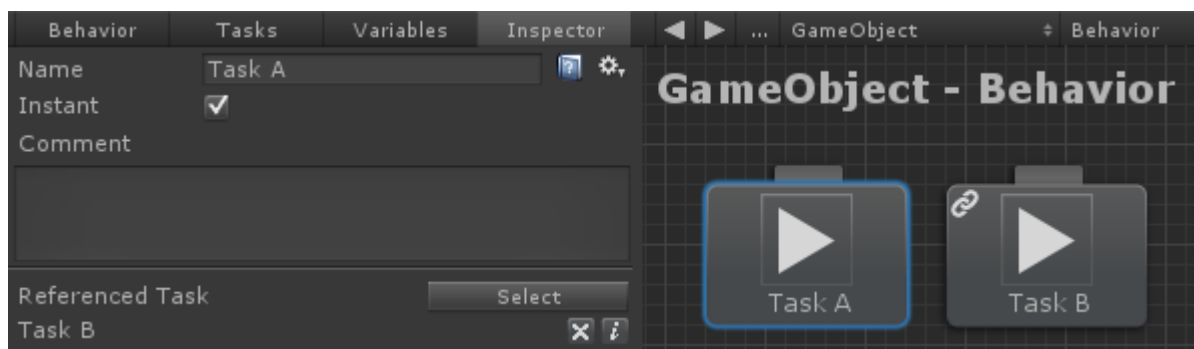
```
using UnityEngine;
using BehaviorDesigner.Runtime.Tasks;

public class TaskB : Action
{
    public float SomeFloat;
}
```

Add both of these tasks to your behavior tree within Behavior Tree and select TaskA.



Click the select button. You'll enter a link mode where you can select other tasks within the behavior tree. After you select Task B you'll see that Task B is linked as a referenced task:



That is it. Now when you run the behavior tree TaskA will be able to output the value of TaskB's SomeFloat value. You can clear the reference by clicking on the "x" to the right of the referenced task name. If you click on the "i" then the linked task will highlight in



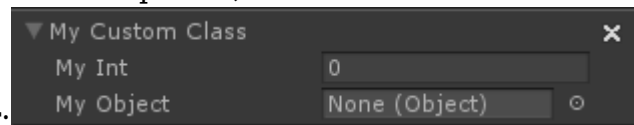
orange:

Tasks can also be referenced using an array:

```
public class TaskA : Action
{
    public TaskB[] referencedTasks;
}
```

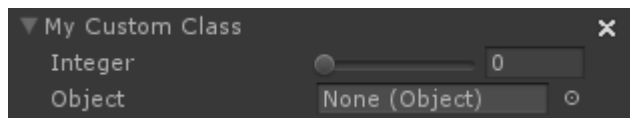
Object Drawers

Object Drawers are very similar to the Unity feature [Property Drawers](#). Object drawers allow you to customize the look of different objects within the inspector. As an example, we will modify the Shared Custom Object example found in the [Creating Your Own Shared Variable](#) topic. With the default inspector, the SharedCustomClass variable looks like the



following in the inspector:

For this example, we will limit the range of the integer between 0 and 10 using object drawers:



The following object drawer was used to accomplish this (this script goes in an Editor folder):

```
using UnityEngine;
using UnityEditor;
using BehaviorDesigner.Editor;

[CustomObjectDrawer(typeof(CustomClass))]
public class CustomClassDrawer : ObjectDrawer
{
    public override void OnGUI(GUIContent label)
    {
        var customClass = value as CustomClass;
        EditorGUILayout.BeginVertical();
        if (FieldInspector.DrawFoldout(customClass.GetHashCode(),
label)) {
            EditorGUI.indentLevel++;
            customClass.myInt = EditorGUILayout.IntSlider("Integer",
customClass.myInt, 0, 10);
            customClass.myObject = EditorGUILayout.ObjectField("Object",39
```

```

customClass.myObject, typeof(UnityEngine.Object), true);
    EditorGUI.indentLevel--;
}
EditorGUILayout.EndVertical();
}
}

```

The only method that you need to override for object drawers to work is the `OnGUI(GUIContent label)` method. The label field is the name of the field that is being drawn. Just like property drawers, you can specify a object drawer by the class type or by attributes. The example above is using the class type method.

As another example, we will convert the Ranged Attribute used in Unity's example to a Object Drawer. First we need to create the attribute:

```

using UnityEngine;
using BehaviorDesigner.Runtime.Tasks;
using BehaviorDesigner.Runtime.ObjectDrawers;

public class RangeAttribute : ObjectDrawerAttribute
{
    public float min;
    public float max;

    public RangeAttribute(float min, float max)
    {
        this.min = min;
        this.max = max;
    }
}

```

Now that the attribute is created, we need to create the actual object drawer (this script goes in an Editor folder):

```

using UnityEngine;
using UnityEditor;
using BehaviorDesigner.Editor;

[CustomObjectDrawer(typeof(RangeAttribute))]
public class RangeDrawer : ObjectDrawer
{
    public override void OnGUI(GUIContent label)
    {
        var rangeAttribute = (RangeAttribute)attribute;
        value = EditorGUILayout.Slider(label, (float)value,
rangeAttribute.min, rangeAttribute.max);
    }
}

```

Once both of these have been created, we can use it within a task:


```

using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;
using BehaviorDesigner.Runtime.ObjectDrawers;

public class NewAction : Action
{
    [Range(5, 10)]
    public float rangedFloat;
    public override TaskStatus OnUpdate()
    {
        Debug.Log(rangedFloat);
        return TaskStatus.Success;
    }
}

```

This will show up in the task inspector as:



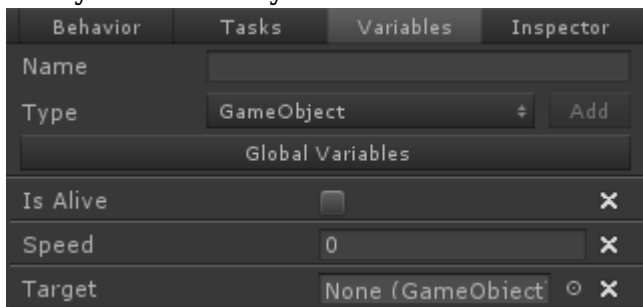
Variable Synchronizer

[Shared Variables](#) are great for sharing data across tasks and behavior trees. However, in some cases you want to share to same variables with non-behavior tree components. As an example, you may have a GUI Controller component which manages the GUI. This GUI Controller displays a GUI element indicating whether or not the agent being controlled by the behavior tree is alive. It does this by having a boolean which says whether or not the agent is alive:

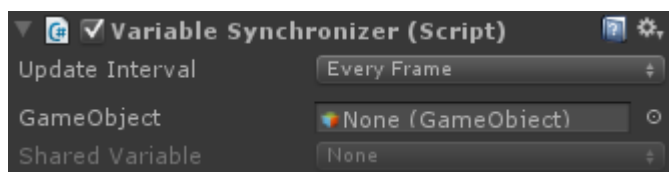
```
public bool isAlive { get; set; }
```

With the Variable Synchronizer component, you can automatically keep this boolean and the corresponding Shared Variable synchronized with each other.

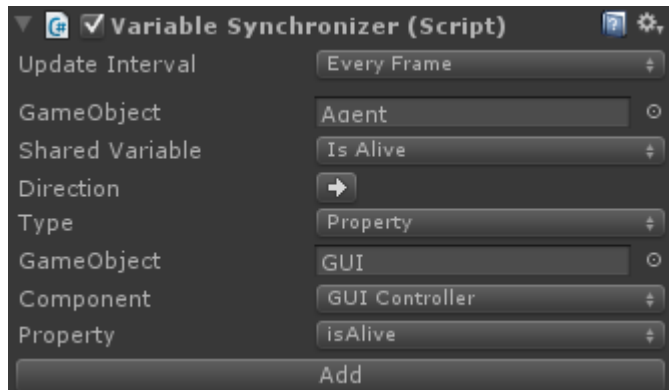
To setup the Variable Synchronizer, first make sure you have created the Shared Variables that you want to synchronize. For this example we created three Shared Variables:



Following that, add the Behavior Designer/Variable Synchronizer component to a GameObject.

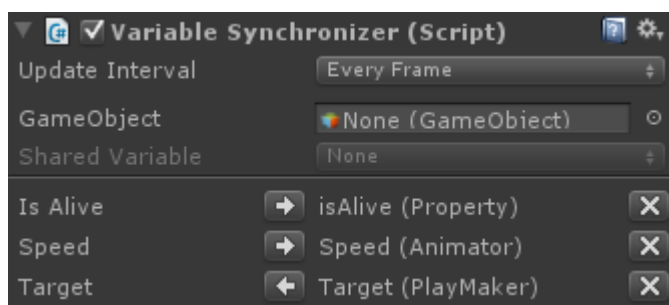


Next, start adding the Shared Variable that you want to keep synchronized. For this example we are going to add the Is Alive variable that was previously mentioned.



1. Specify the GameObject which contains the behavior tree that has the Shared Variable that you want to synchronize.
2. Select from the popup box which Shared Variable you want to use.
3. Specify a direction. If the arrow is pointing to the left then you are setting the Shared Variable value. If the arrow is pointing to the right then you are getting the Shared Variable value.
4. Specify the type of synchronization. Currently the following types are supported: Behavior Designer, Property, Animator, and PlayMaker.
5. The remaining steps will depend on the type of synchronization selected. In this example Property was selected so you'll need to select the component which contains the property that you want to synchronize with the Shared Variable.
6. Click Add.

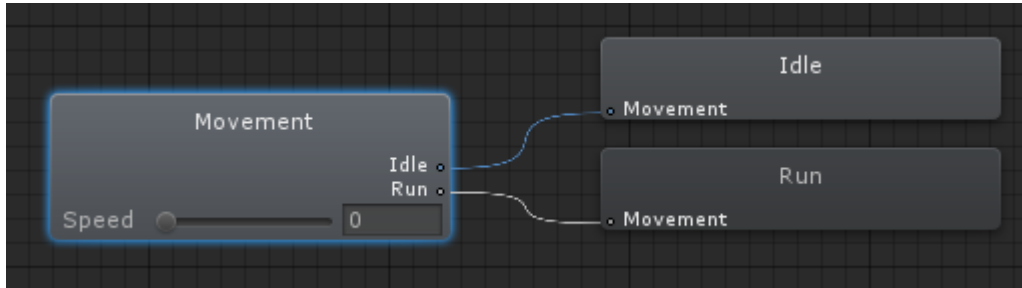
Once added the Is Alive Shared Variable will set the isAlive property at an interval specified by Update interval. The following screenshot contains a few more synchronized variables:



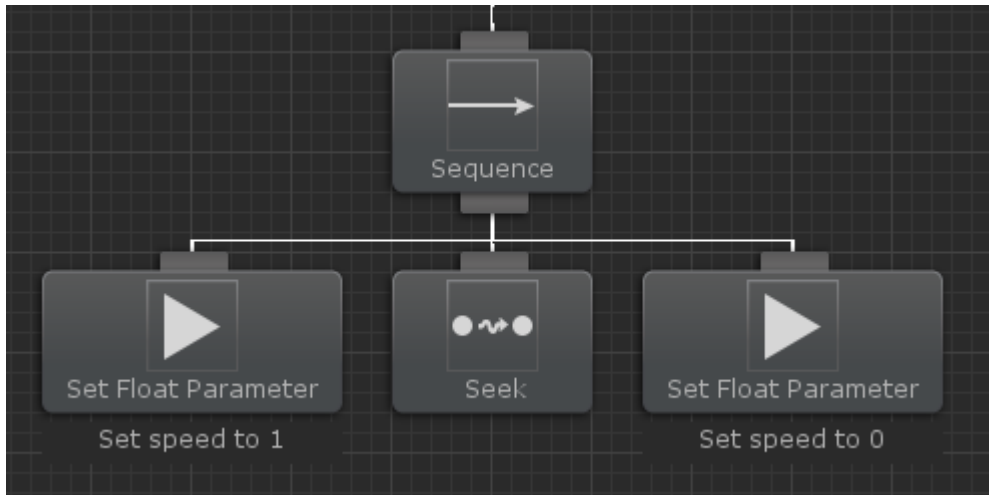
- The Is Alive Shared Variable is setting the isAlive property.
- The Speed Share Variable is setting the Speed Animator parameter.
- The Target Shared Variable is being set by the Target PlayMaker variable.

Syncing Animations

Behavior Designer includes a set of Animator tasks which allow you to play animations within the tree. Consider the tree below:



This is an extremely simple Animator Controller that uses a blend tree to blend between the Idle and Run state based on the Speed parameter. If you'd like to then sync this blend tree with the Seek task your behavior tree would then look similar to:



Before the Seek task starts the Speed parameter is set to 1 so the blend tree can play the run animation. After the Seek task completes the Speed parameter is then set to 0 to play the Idle animation again.

With a small behavior tree and Animator Controller this process works well. However, as the amount of animations grow for your agent this method starts to become extremely cumbersome. In addition, your behavior tree also starts to become extremely verbose with all of the Animator tasks and if you change a transition within your Animator Controller you'd have to rework the behavior tree to work with that Animator Controller change.

There is a better way.

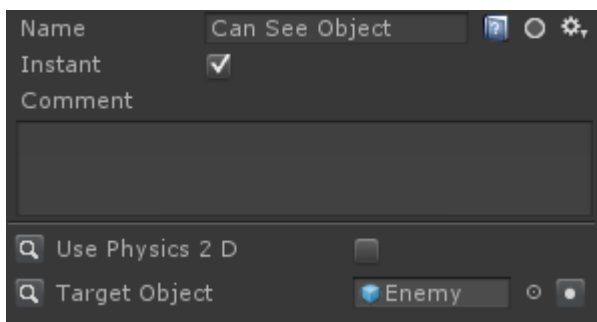
The recommended approach to syncing animations within your behavior tree is to not sync the animations at all within the behavior tree. Instead your agent's character controller should do it for you. Using Unity's NavMeshAgent as an example, when the behavior tree sets the NavMeshAgent destination (such as with the Seek task) the NavMeshAgent's velocity will change to move towards the destination. The character controller should then use the velocity and translate the movements into parameters that the Animator Controller can understand. MecWarriors had an excellent tutorial on how to accomplish this. The MecWarriors site is no longer operational but a cached version of the tutorial can be found

[here](#). A similar implementation can be done with Apex Path and A* Pathfinding Project.

The advantage of this approach is that your behavior tree is then not aware of the animations at all and it can even work with [root motion](#). You also don't over-complicate your tree with Animator tasks and makes your tree a lot cleaner. The Ultimate Character Controller uses this approach and for that asset we created a pretty large behavior tree. You can see the behavior tree on [this page](#) and you'll notice that there are no Animator tasks within the tree. A bridge component is linking the NavMeshAgent's velocity to the character controller's inputs which then allow the Third Person Controller to take care of the animations.

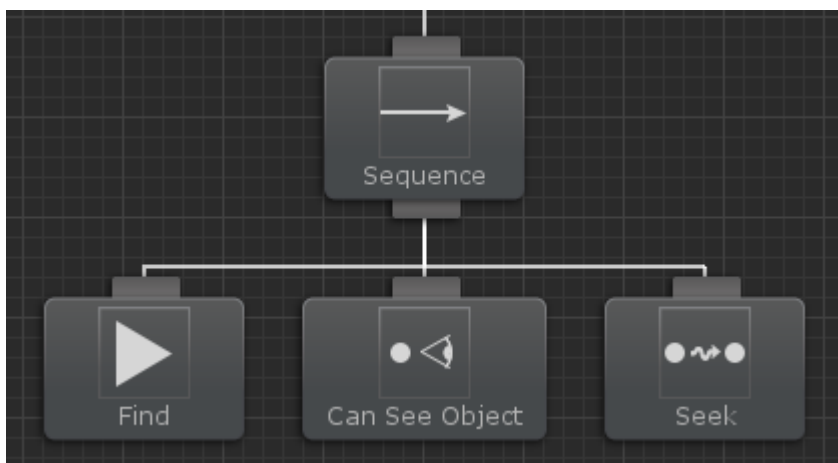
Referencing Scene Objects

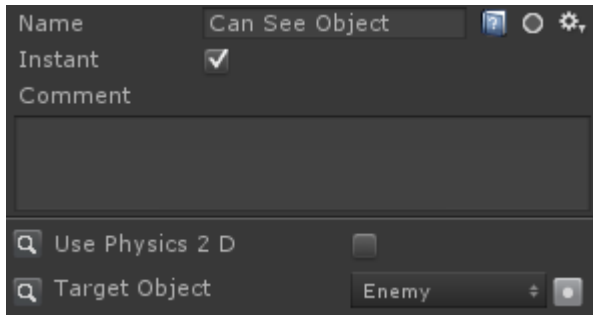
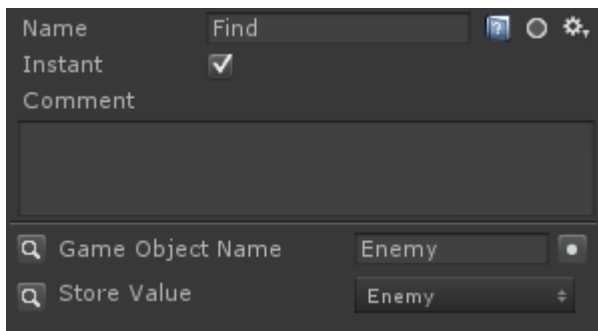
As you are creating your tree it is common practice to reference objects within the scene from a Shared Variable or task. For example, in this tree Can See Object is determining if the Enemy object can be seen:



As you are testing you find that everything is working well and now you'd like to make a prefab out of it so you can have multiple agents in your scene. As soon as you make the agent a prefab and remove it from the scene you'll find that the Target Object reference has gone missing. The reason this occurs is because as a project level asset (the prefab, or global variables file) cannot reference objects in the scene. This is a Unity restriction and the way to overcome it is to populate the variable at runtime after you have spawned the prefab.

The scene reference object can be populated at runtime in many different ways. One way is to use the Find task and search for the object by name:





When the Find task runs it'll search for a GameObject named "Enemy" and then place it in a SharedVariable. When Can See Object runs it'll then use that SharedVariable to search for the target object.

Another way to populate the value at runtime is to have a component already in the scene which has a reference to the object. After the prefab has been spawned this component can then set the value of a SharedVariable that the Can See Object uses.

```
public class Spawner : MonoBehaviour
{
    public GameObject m_Enemy;

    public void PrefabSpawned(BehaviorTree behaviorTree)
    {
        behaviorTree.SetVariableValue("Enemy", m_Enemy);
    }
}
```

Task Attributes

Behavior Designer exposes the following task attributes: HelpURL, TaskIcon, TaskCategory, TaskDescription and LinkedTask.

Help URL

If you open the task inspector panel you will see on the doc icon on the top right. This doc icon allows you to associate a help webpage with a task. You make this association with the HelpURL attribute. The HelpURL attribute takes one parameter which is the link to the webpage.

```
[HelpURL("http://www.example.com")]
public class MyTask : Action
```

```
{
```

Task Icon

Task icons are shown within the behavior tree using the `TaskIcon` attribute and are used to help visualize what a task does. Paths are relative to the root project folder. The keyword `{SkinColor}` will be replaced by the current Unity skin color, "Light" or "Dark".

```
[TaskIcon("Assets/Path/To/{SkinColor}Icon.png")]  
public class MyTask : Action  
{
```

Task Category

Organization starts to become an issue as you create more and more tasks. For that you can use `TaskCategory` attribute:

```
[TaskCategory("Common")]  
public class Seek : Action  
{
```

This task will now be categorized under the common category:



Categories can be nested by separating the category name with a slash:

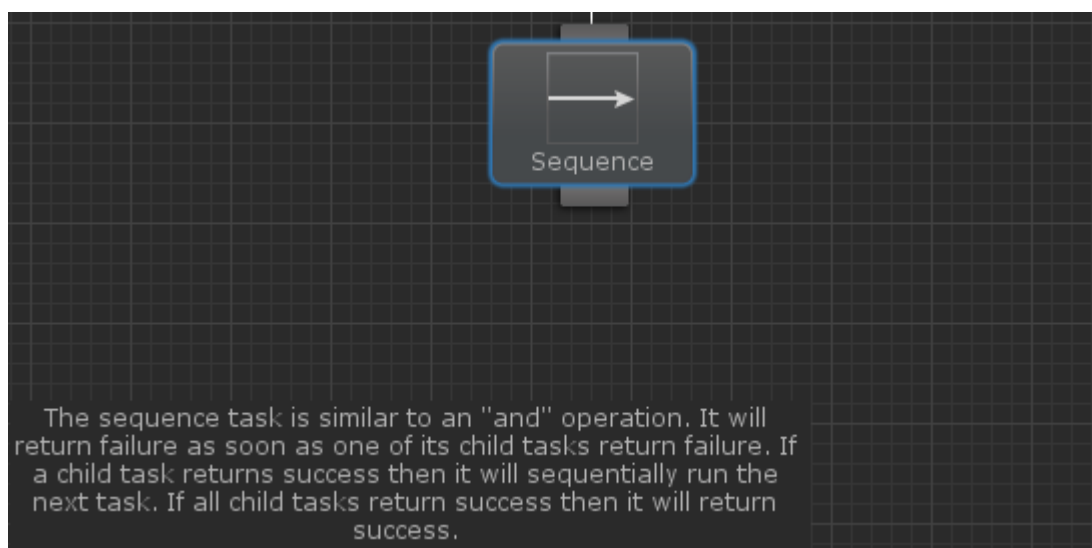
```
[TaskCategory("RTS/Harvester")]
public class HarvestGold : Action
{
```

Task Description

The TaskDescription attribute allows you to show your class-level comment within the graph view. For example, the sequence description starts out with:

```
[TaskDescription("The sequence task is similar to an \"and\" operation. ...")]
public class Sequence : Composite
{
```

This description will then be shown in the bottom left area of the graph:



Linked Task

[Variables](#) are great when you want to share information between tasks. However, you'll notice that there is no such thing as a "SharedTask". When you want a group of tasks to share the same tasks use the LinkedTask attribute. As an example, take a look at the task guard task. When you reference one task with the task guard, that same task will reference the original task guard task back. Linking tasks is not necessary, it is more of a convince attribute to make sure the fields have values that are synchronized. Add the following attribute to your field to enable task linking:

```
[LinkedTask]
public TaskGuard[] linkedTaskGuards = null;
```

To perform a link within the editor perform the same steps as [referencing another task](#).

Integrations

Behavior Designer includes many tasks which integrate with third party assets. For most of those integrations, no extra steps are required and they can be added to a behavior tree and then have their values assigned. However, the integrations below have a more detailed explanation for how they work.

[Dialogue System for Unity](#)

[Playmaker](#)

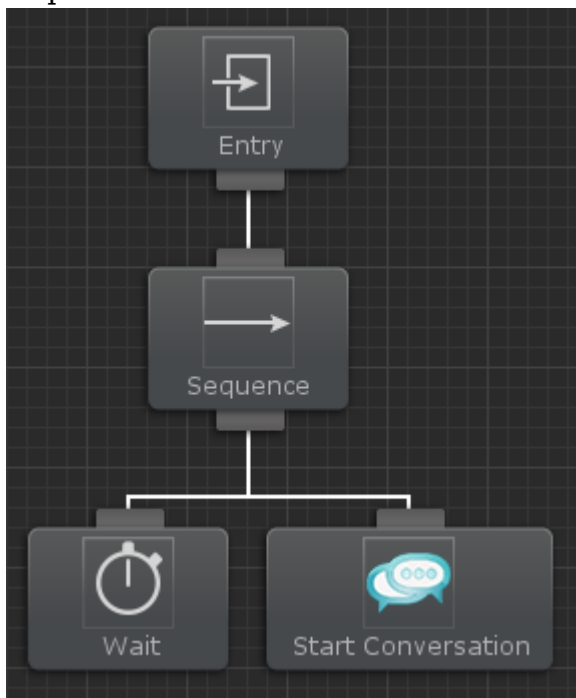
[Ultimate Character Controller](#)

[uScript](#)

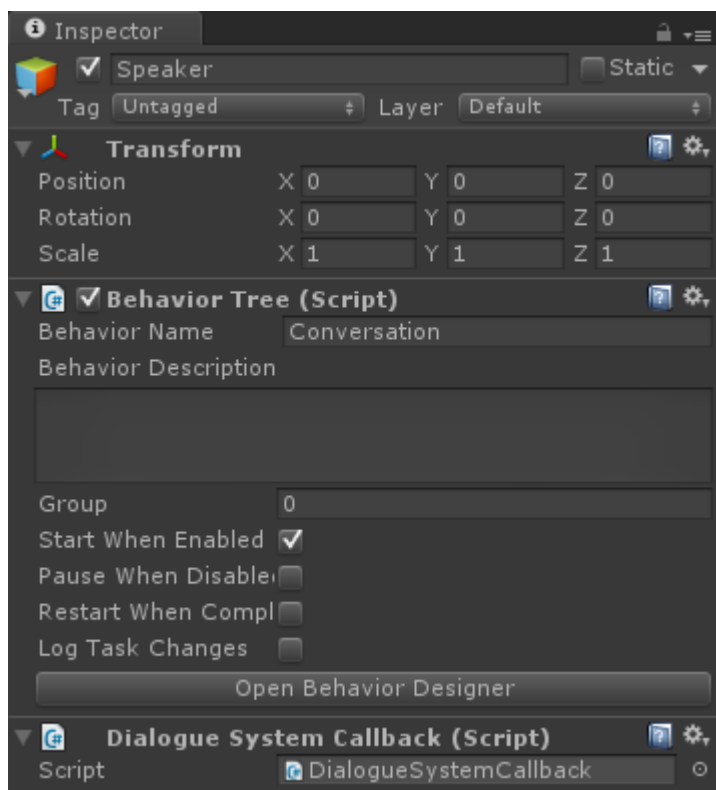
Dialogue System

The [Dialogue System](#) is a complete dialogue system for Unity. Behavior Designer is integrated with the Dialogue System by allowing you to manage conversations, barks, sequences, and quests within your behavior tree. Also, Dialogue System is integrated with Behavior Designer so it can synchronize variables with Lua and start/stop behavior trees with sequence commands. More information on this side of the integration can be found [here](#). All of the Dialogue System integration files are located on the [integrations page](#).

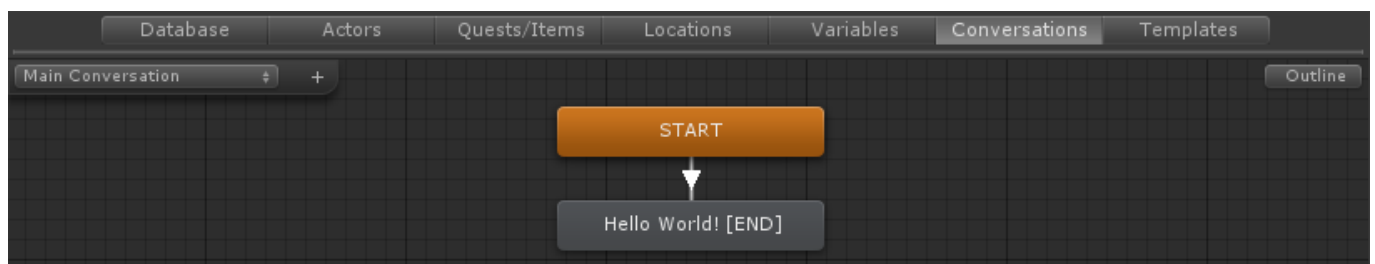
To get started, first make sure you have Dialogue System for Unity installed and have imported the integration package. Once those files are imported you are ready to start creating behavior trees with the Dialogue System. Let's create a very basic tree with a sequence task which has a Wait task and a Start Conversation task:



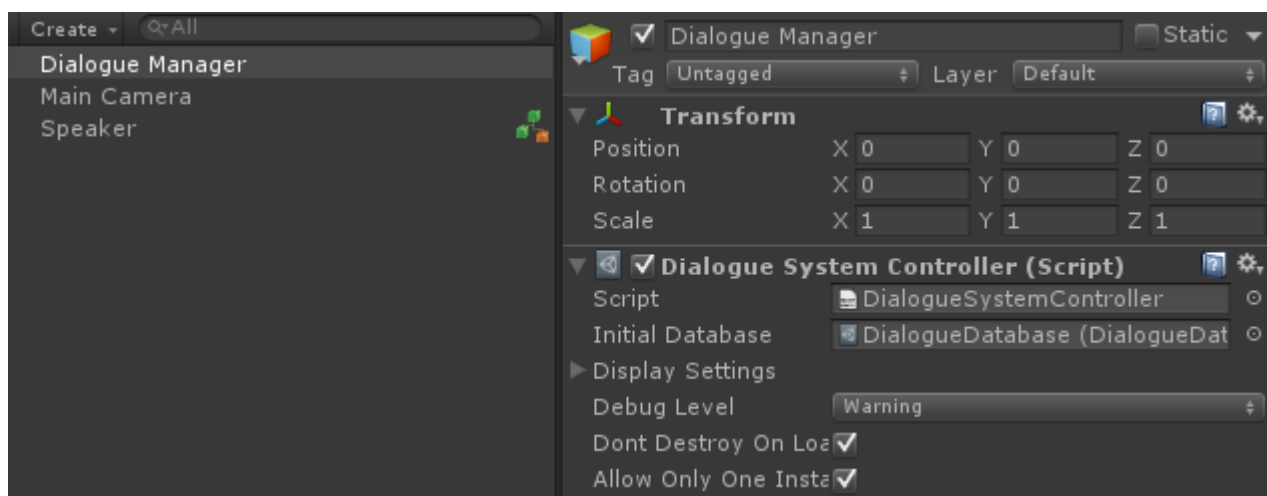
When the Dialogue System finishes with a conversation or sequence it will callback to Behavior Designer to let Behavior Designer know that it is done. In order for this to occur the Dialogue System Callback component must be added to the same GameObject that your behavior tree is on:



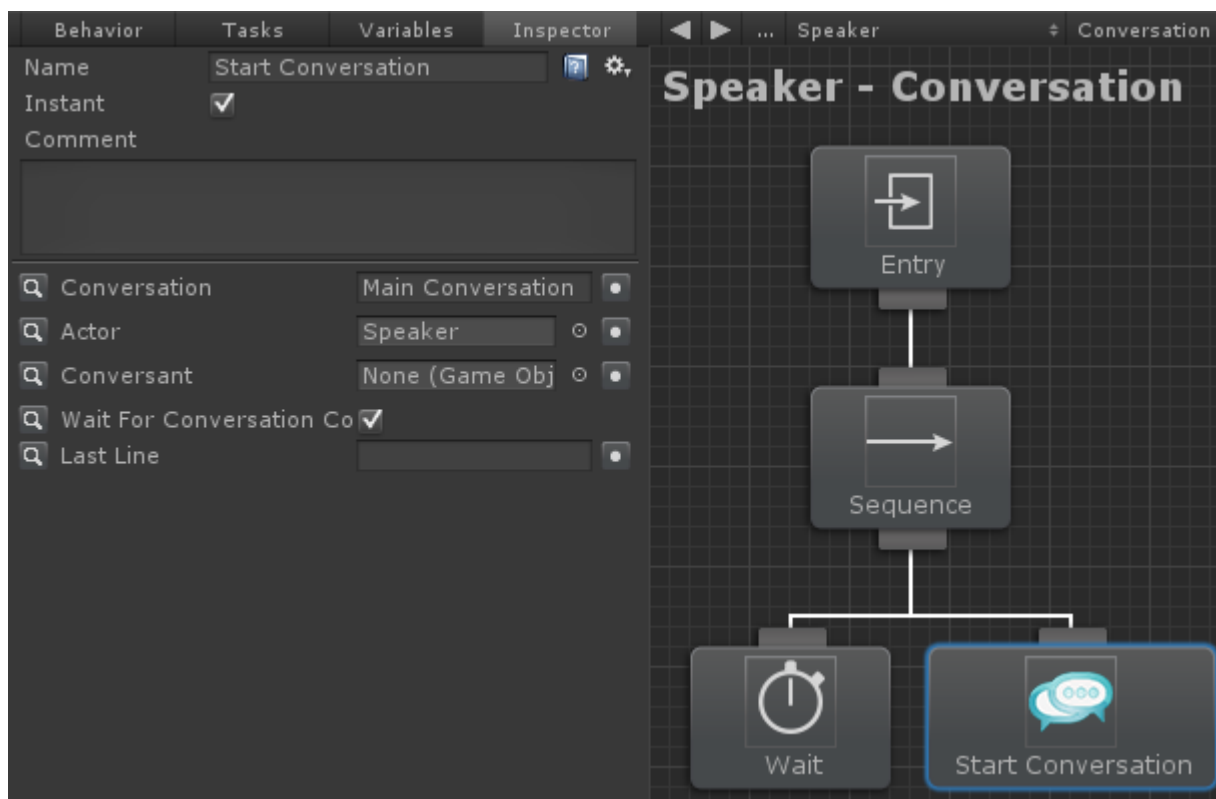
Now we are ready to start creating the actual conversation. Create a new Dialogue System Database and create a basic conversation:



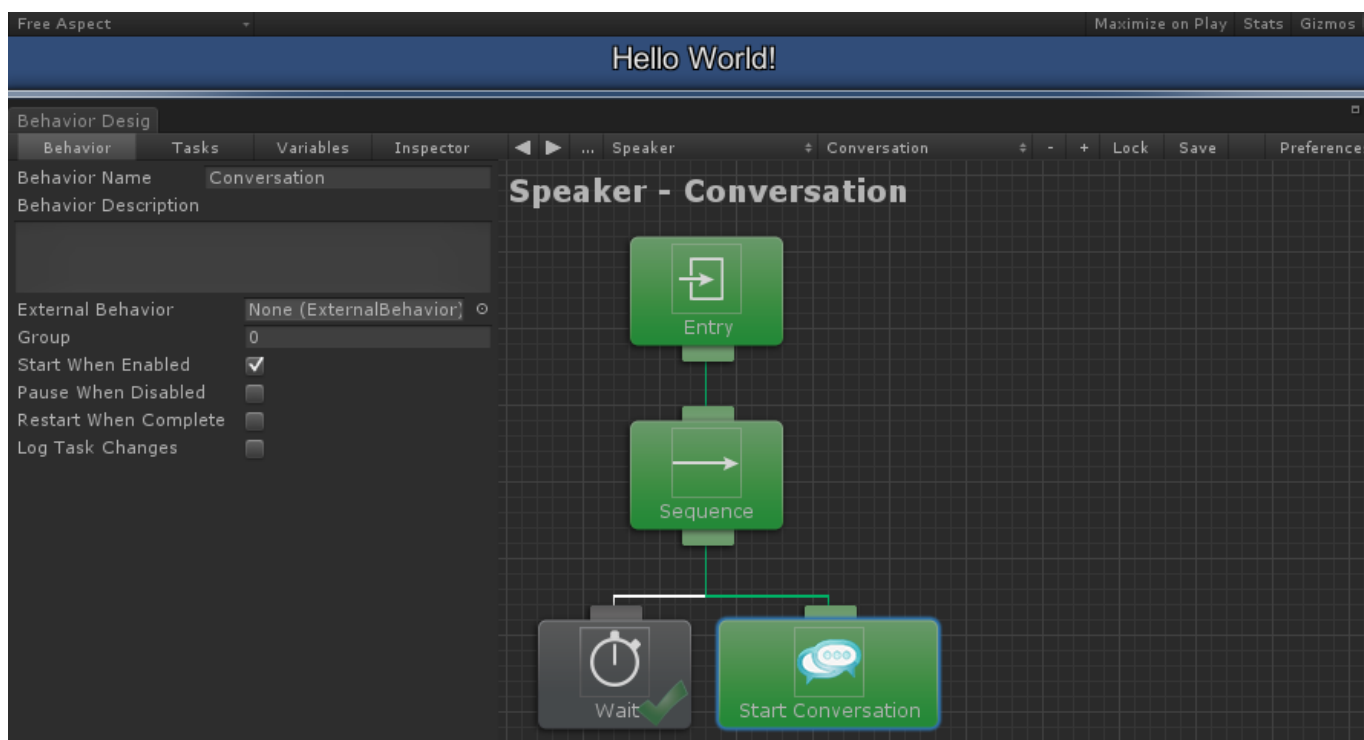
Make note of the conversation name because that will be needed later. Assign that database to the Dialogue System Controller:



The last step is to simply assign the values within the Start Conversation task. The only two values that are required are the conversation name and the actor GameObject:



Once those values have been assigned, hit play and you'll see the text "Hello World" appear at the top of the game screen:



This topic hardly scratches the surface for what is possible with Behavior Designer / Dialogue System integration. For a more complex example, take a look at the Dialogue System sample project.

Opsive Character Controllers

This integration package can be downloaded on the [integrations page](#).

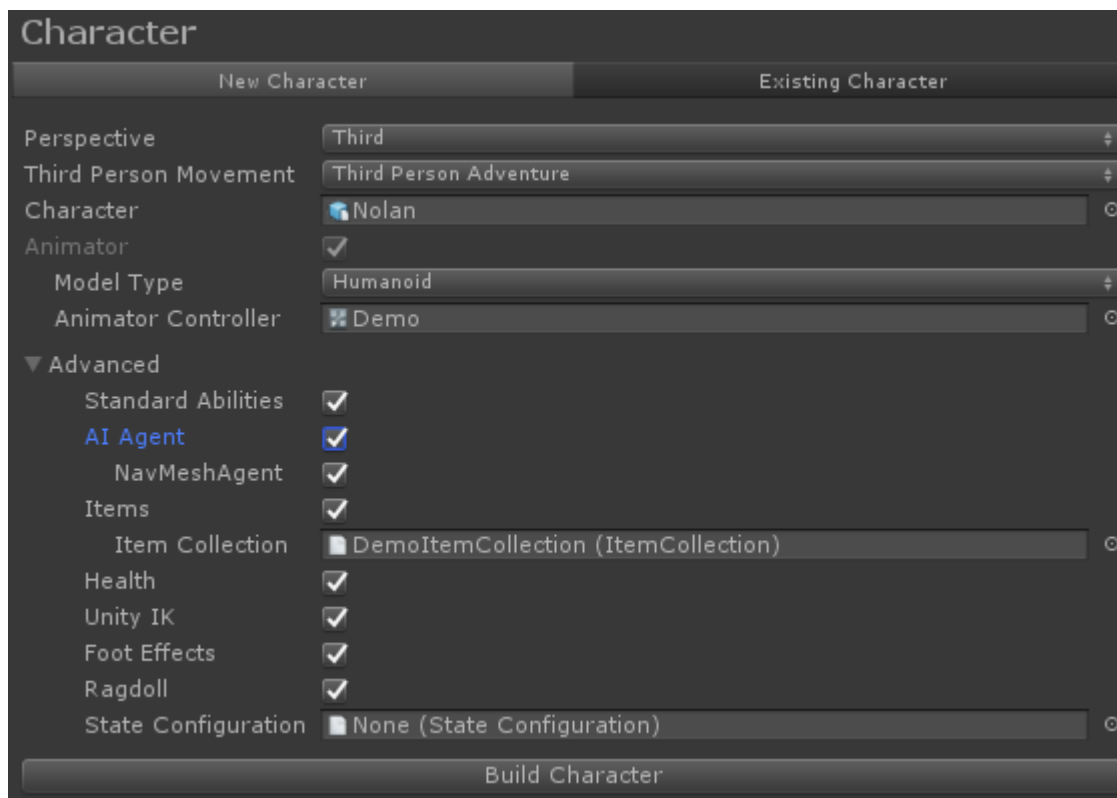
The Opsive Character Controllers are integrated with Behavior Designer allowing your AI to perform any function that a player-controlled character can perform. The Ultimate Character Controller integration contains a demo scene that is designed to work with both shootable and melee weapons.

When the integration is imported there will be four sample scenes included:

- First Person Melee Scene
- First Person Shooter Scene
- Third Person Melee Scene
- Third Person Shooter Scene

The behavior tree for these four scenes are exactly the same and the only difference is the player perspective and which weapon the characters have equipped.

When you are creating a new character that should be used with Behavior Designer ensure you have enabled AI Agent within the Character Manager. If you are using Unity's Navigation Mesh you should also enabled the NavMeshAgent toggle.



After the character has been created the following components should also be added:

- Behavior Tree
- Behavior Tree Agent

The Behavior Tree Agent component can be found within the Opsive Character Controller [integration package](#). In the demo scene you'll notice that there is also a Demo Agent component added which adds the character's health and ammo as property's for Behavior Designer's [Property Mapping](#) feature.

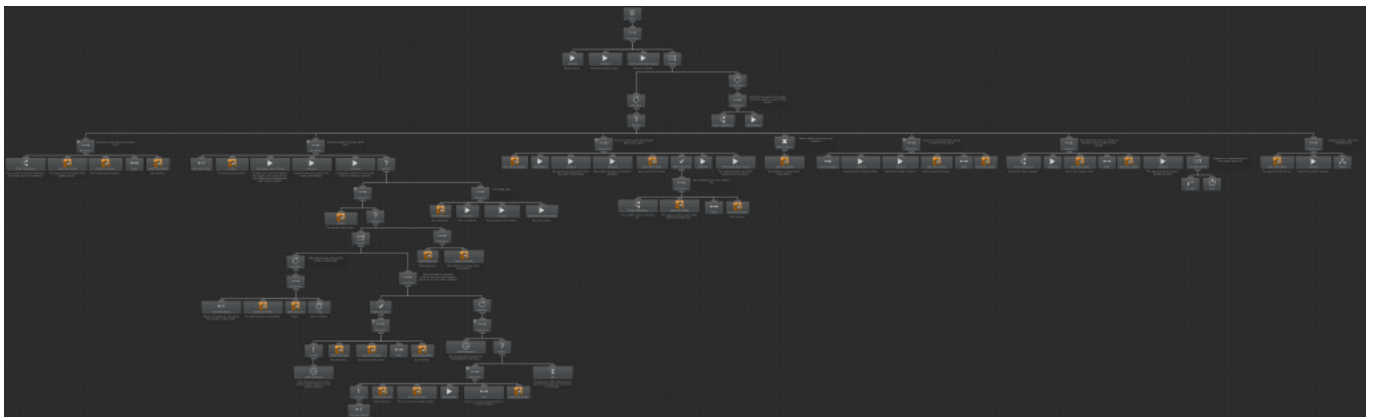
After the character has been setup it's time to create your own behavior tree. The included

behavior tree provides a great overview of the behavior tree flow so the rest of this document will describe how that behavior tree works. In order to get the most out of Behavior Designer it is highly recommended that you create your own behavior tree.

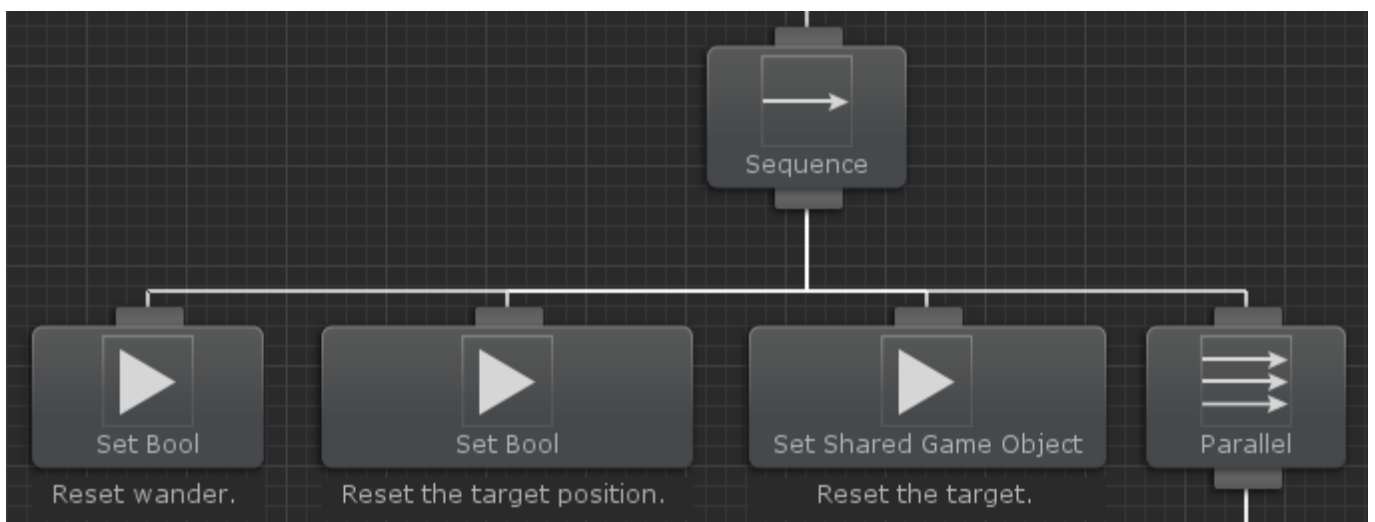
The included behavior tree will perform the following functions:

- Attack if the player is within sight.
- Attack if the player attacks the agent.
- Move towards the audio source location if a sound from the player is heard.
- Search for the player if the player is lost.
- Get health if needed.
- Get ammo if needed.
- Patrol if no other actions are necessary.

The entire behavior tree looks like:



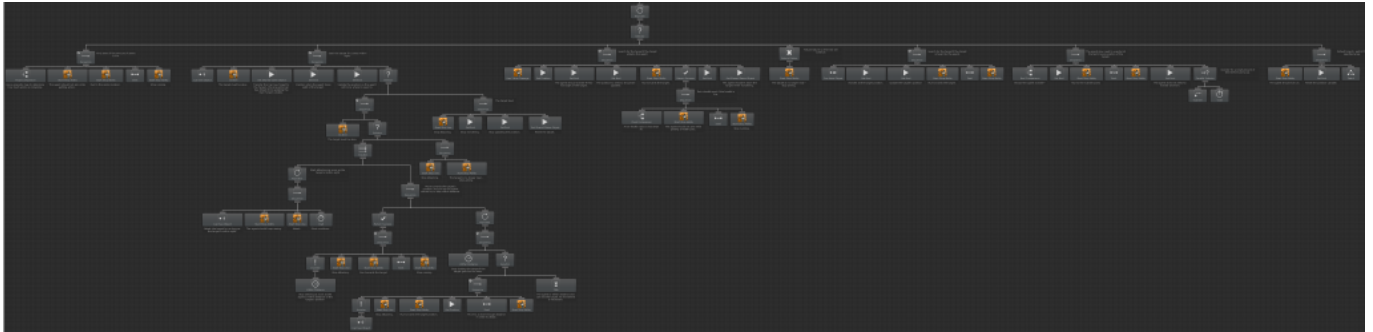
Behavior trees evaluate from top to bottom, left to right. This means that the very first task that runs is the Sequence task directly beneath the Entry task:



The Sequence task is a Composite task that will evaluate all of its children until a child returns a status of failure. The Sequence task can be thought of as an AND between the child tasks. The first three tasks that this Sequence task will execute relate to resetting the behavior tree to get it ready for the next run. The variables that these tasks are resetting will be described later. A Parallel task is then used to execute multiple children at the same time. There are two branches that this Parallel task will execute:

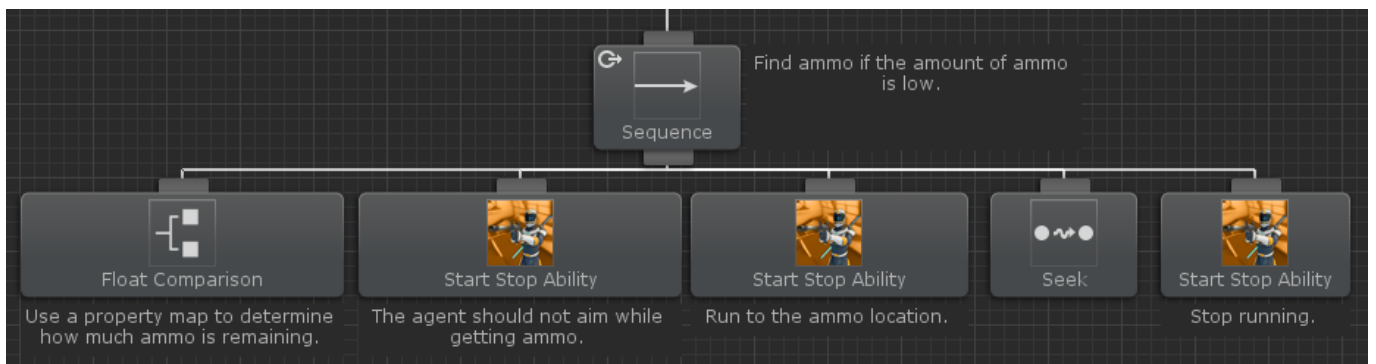
- Left branch which will perform the actual actions such as the movement or attack.
- Right branch which will keep the player's location up to date when the Update Position variable is true.

The main functionality of the tree is contained within the left branch:



This branch is parented with a Repeater task which is set to Repeat Forever. The Repeater task is a Decorator which will keep executing the child branch for as long as requested. Since Repeat Forever is enabled the Repeater will keep the branch active until the behavior tree ends. A Selector is parented to the Repeater. The Selector task will execute its children until a child returns success. The Selector can be thought of as an OR between the child tasks.

Because behavior trees evaluate from left to right the branches on the left have a higher priority than those on the right. This means that the very first branch that will execute is the branch that checks to determine if the agent is out of ammo:

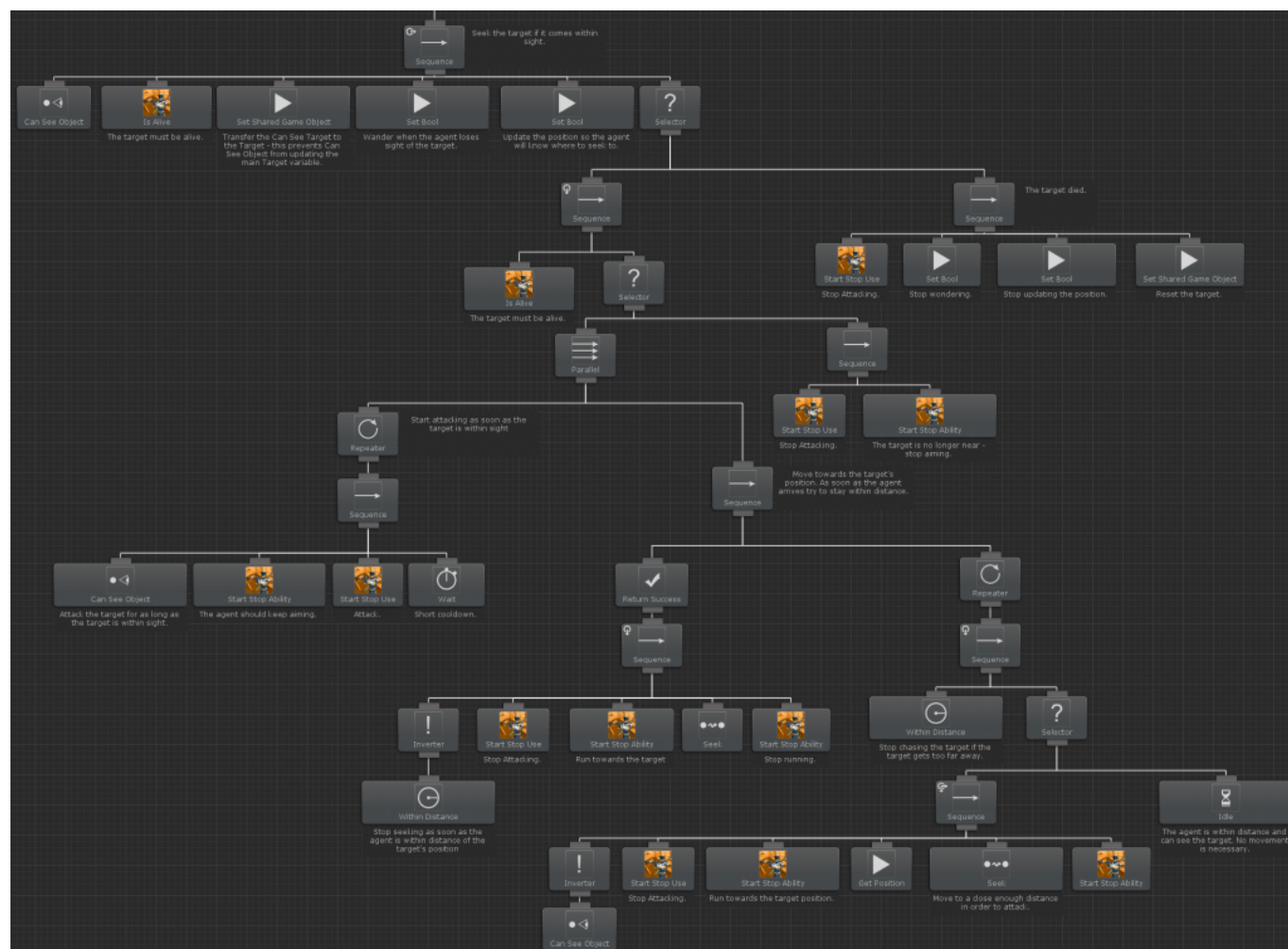


The Sequence task of the ammo branch has a Lower Priority [Conditional Abort](#) set. Conditional Aborts allow the behavior tree to reevaluate child Conditional tasks when other branches are active. If the character is out of ammo then the character can't do anything so this branch has the highest priority. If a lower priority branch (any branch to the right of the ammo branch) is active and the agent runs low on ammo then the conditional abort will abort the lower priority branch and start executing the ammo branch.

The ammo branch is a relatively simple branch that first checks to determine how much ammo the character has with the Float Comparison Conditional task. This Float Comparison task will compare the agent's ammo to a constant number. The amount of ammo is retrieved through the Ammo property mapping setup through the Demo Agent component. If this amount is less than the constant then the Float Comparison task will return success and the branch will execute.

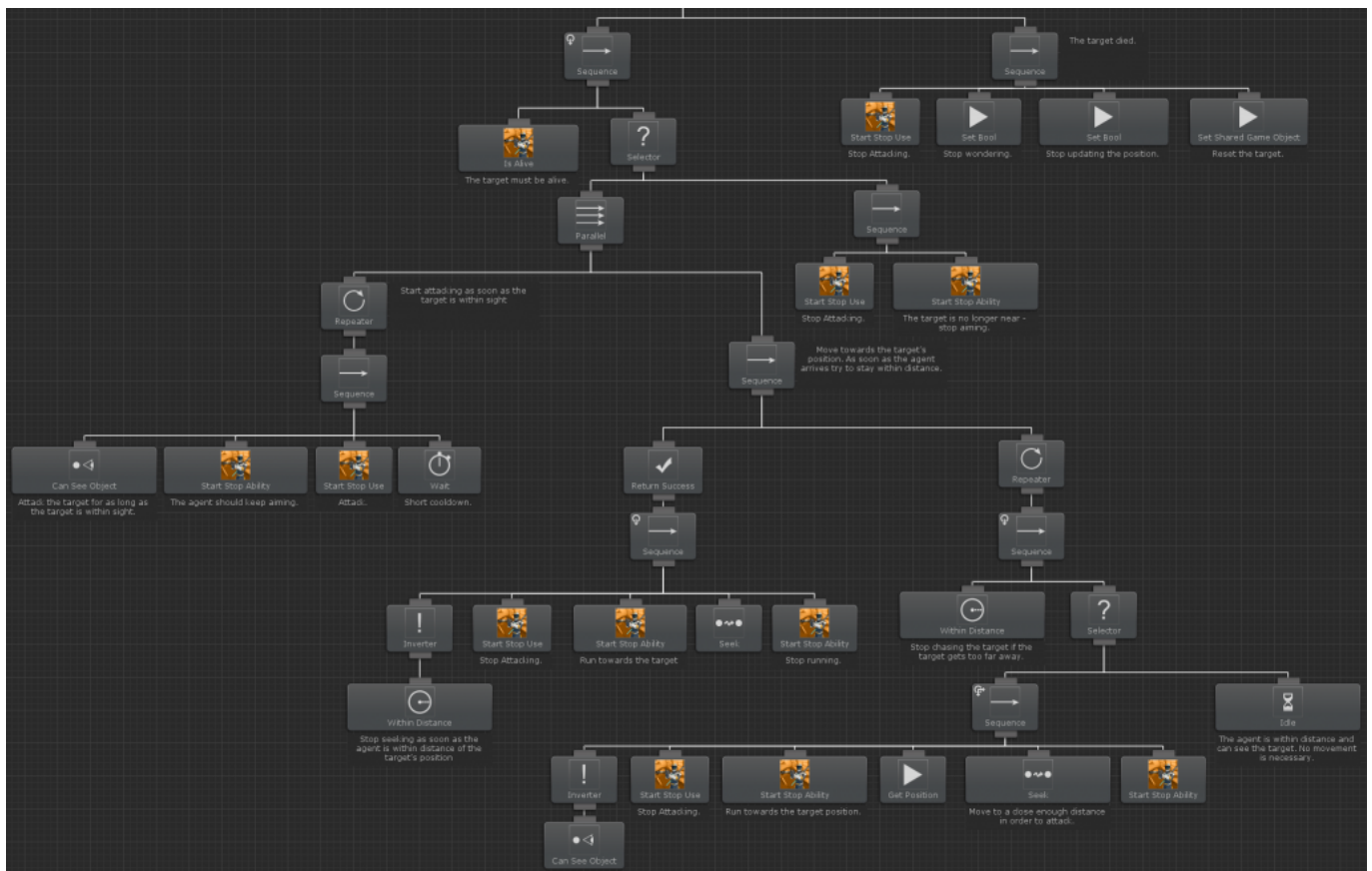
If the agent is low on ammo then it will stop aiming through the Start Stop Ability task. This is a task created for the Ultimate Character Controller and it will start or stop an ability | 53

based on the values specified. In this case the task will stop the Aim ability because the character shouldn't aim when retrieving ammo. Another Start Stop Ability task is run and this task will start the Speed Change task so the character will start running towards the ammo location. The Seek task (from the [Movement Pack](#)) will use Unity's navigation mesh to move the character to the specified position. After the character has arrived at the position the Start Stop Ability task is executed again only this time it will stop the Speed Change ability so the character will stop running.

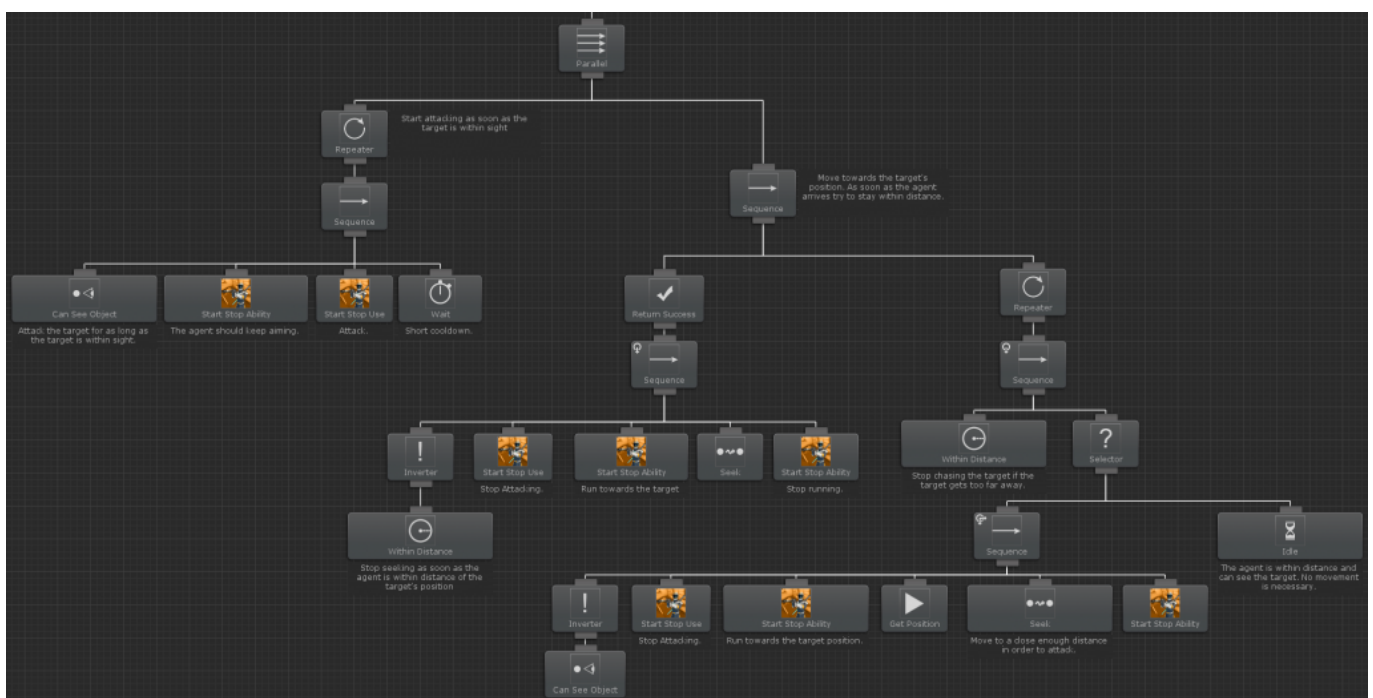


If the agent doesn't need ammo then the next highest priority branch is the Can See branch. This branch contains the main action in that it will actually attack the player when the player is within sight. Similar to the Ammo branch, the can see branch uses a Lower Priority Conditional Abort so it will abort any lower priority branch. The Can See branch will be able to abort any branches to the right of it, which means that it will not be able to abort the Ammo branch because the Ammo branch is to the left of the Can See branch.

The first task within the Can See branch is the Can See Object task from the Movement Pack. Can See Object will use a layer mask to search for any objects within sight. The player is on the Character layer so the task will only search for objects that are on that layer. Once the task finds the player the Is Alive task executes which will determine if the player is alive. There's no need to attack a dead character. The Set Shared GameObject task is then a helper task that will transfer the found Can See Object value to the Target variable. This is done so when other tasks operate on the found player variable the original object isn't lost. Two Set Bool tasks are then executed which indicates that the agent should wander if the player is lost and that the target's position should be updated. The right branch under the original Parallel task will then set the target's position to the Last Position variable.

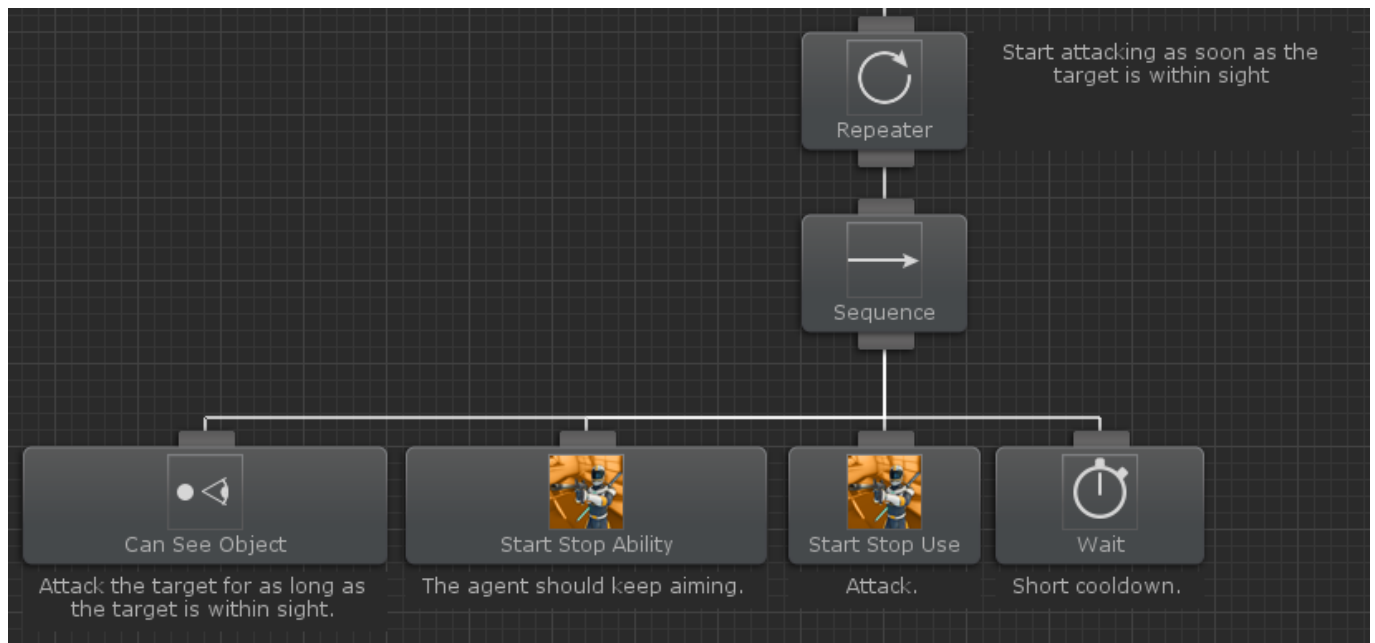


A Selector task is then run which will first evaluate the left branch. The goal of the left branch is to attack the target for as long as the target is in sight. To accomplish this a Sequence task is used with the Self Conditional Abort. The Self Conditional Abort is a new abort type and this abort type will reevaluate the Conditional task for as long as the current branch is active. In this case the current branch contains the Is Alive task along with a Selector so it will reevaluate Is Alive for as long as the agent can see the target. The Is Alive task will return failure when the target is no longer alive.

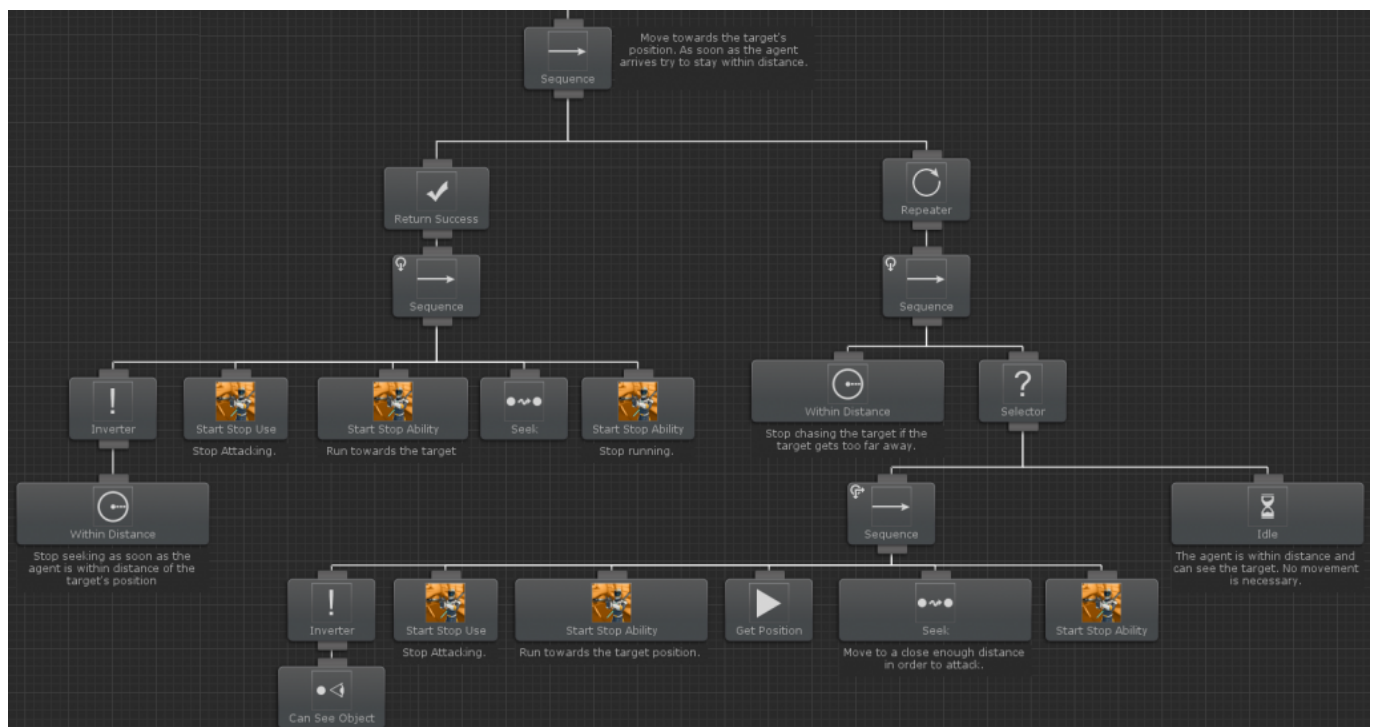


The first branch that the Selector task will evaluate has a Parallel task as the parent. This Parallel task runs two branches:

- The left branch attacks if the target is within sight
- The right branch will keep the agent near the target



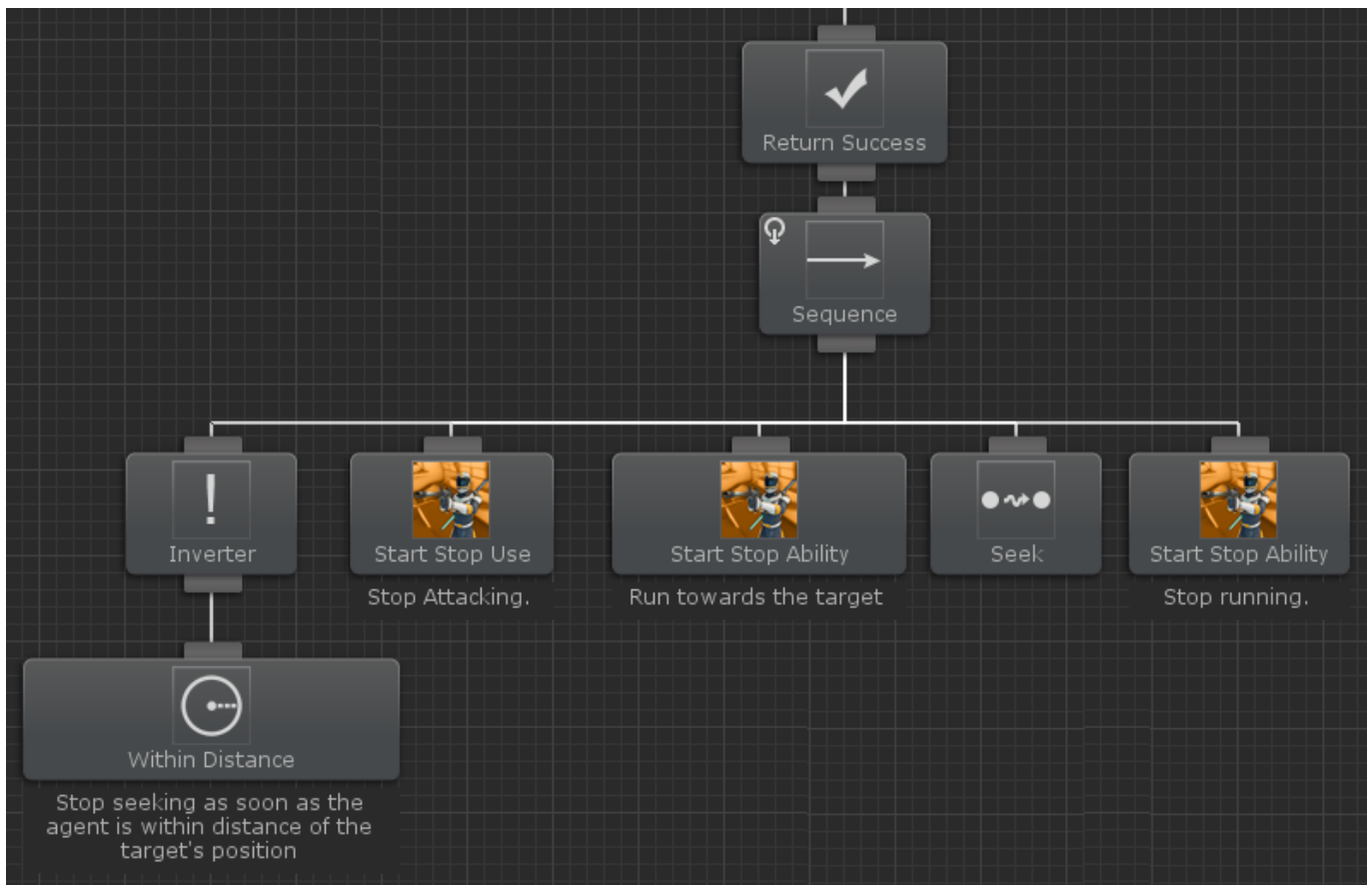
The left Attack branch uses another Can See Object task to determine if the agent is near the target. This Can See Object task is different from the first Can See Object task as it has a smaller distance magnitude so the agent will only attack when near the target. If the target is near the Start Stop Ability task will execute to keep the character aiming. After the agent is aiming the Start Stop Use task will execute which will actually use the item, which will either fire the assault rifle or swing the sword. A Wait task is then used to prevent the agent from trying to attack every frame.



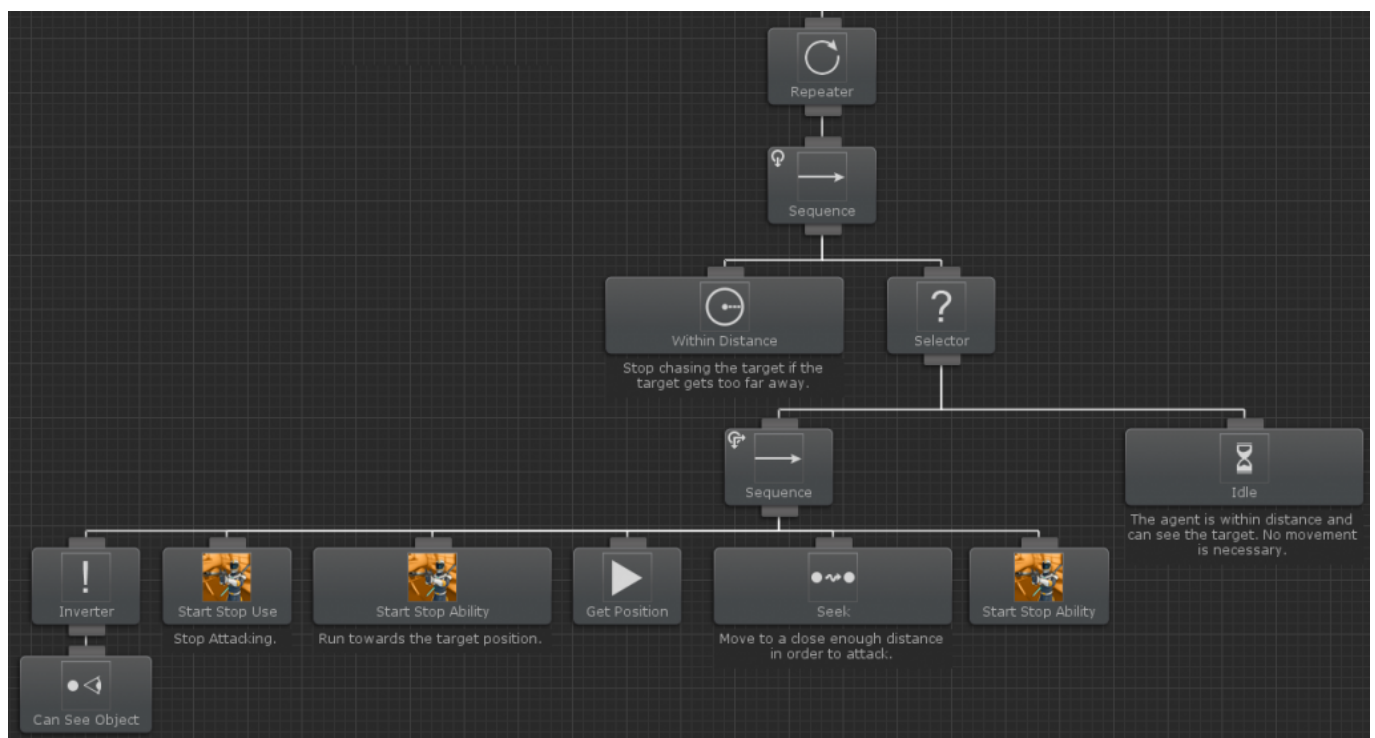
While the Attack branch is running the Move Towards branch will also run. This branch contains two parts:

- Move into position after first acquiring the target.

- Stay in position after arriving in position.

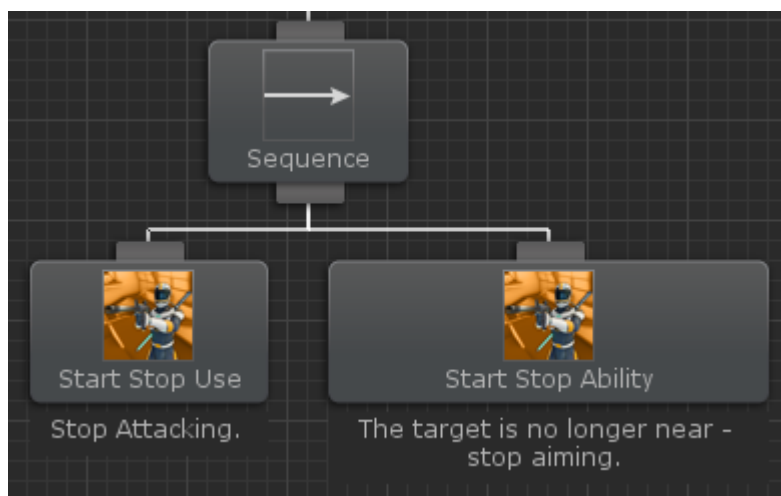


To move into position the Within Distance task is used to determine if the agent is within distance of the target. Since the agent will not be within distance after first acquiring the target the Within Distance task will return failure. Because the agent should keep moving towards the target for as long as they are not within distance an Inverter is used to keep the branch active for as long as the agent is not within distance of the target. A Self Conditional Abort is used so the Within Distance task will reevaluate for as long as any tasks within the current branch are active, in this case it will be the Seek task. As soon as Seek moves the agent close to the target the Within Distance task will return success and the Inverter will change that success to a failure so the branch will then stop executing. While the Seek task is active the Speed Change ability will be activated so the agent will run towards the target.

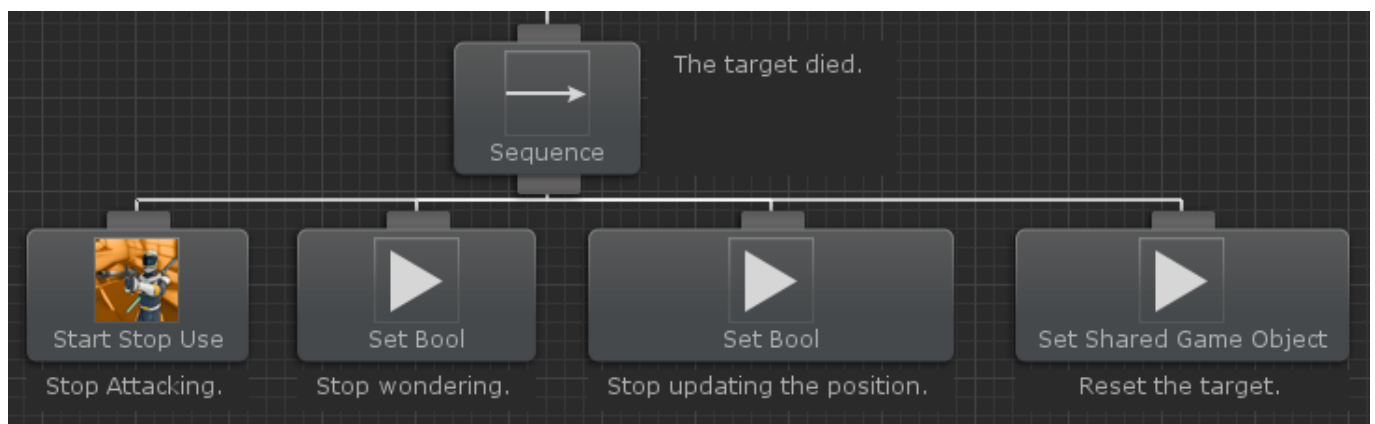


After the agent has move to be within distance of the target the right branch will keep the agent near the target. There are three main differences between this branch and the previous Move Towards branch:

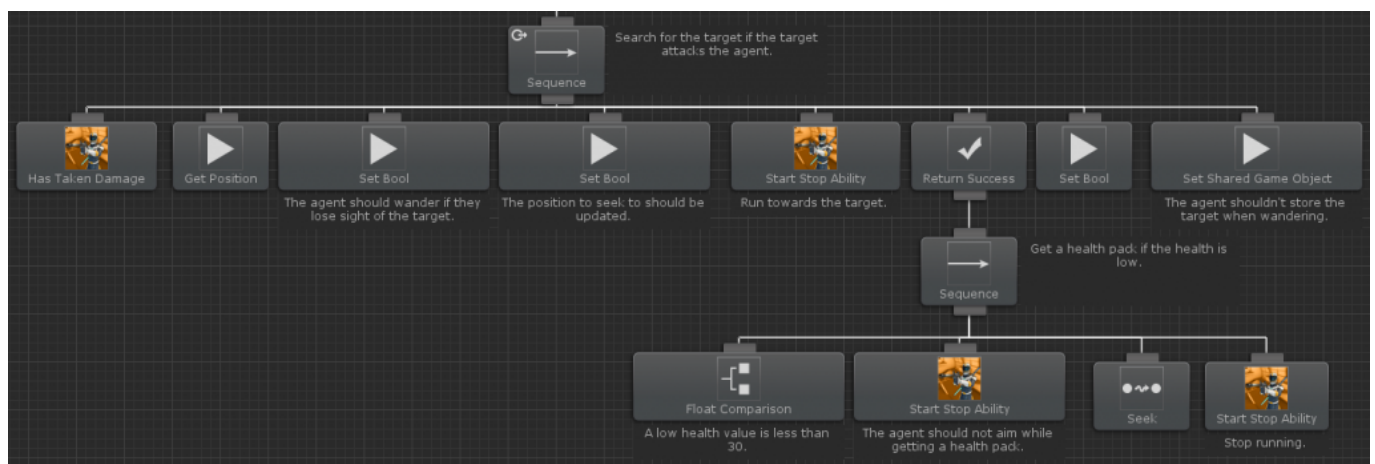
- The current branch will have the character walk instead of run while seeking the target.
- A Can See Object task is used in addition to Within Distance so the agent will always be in sight of the target while attacking. As an example the target may move inside a structure which would cause prevent the agent from seeing the target even though the target may still be near the agent.
- An Idle task is used if the agent doesn't need to seek into any position. If the target is standing still and is within sight then the agent can keep attacking without having to change positions.



The agent will now attack and move towards the target while within distance. If the target gets away from the agent and moves too far away then the right Move Towards branch will return failure and execute the Reset branch. This Reset branch runs two Actions which will ensure the agent is no longer attacking and is no longer aiming.

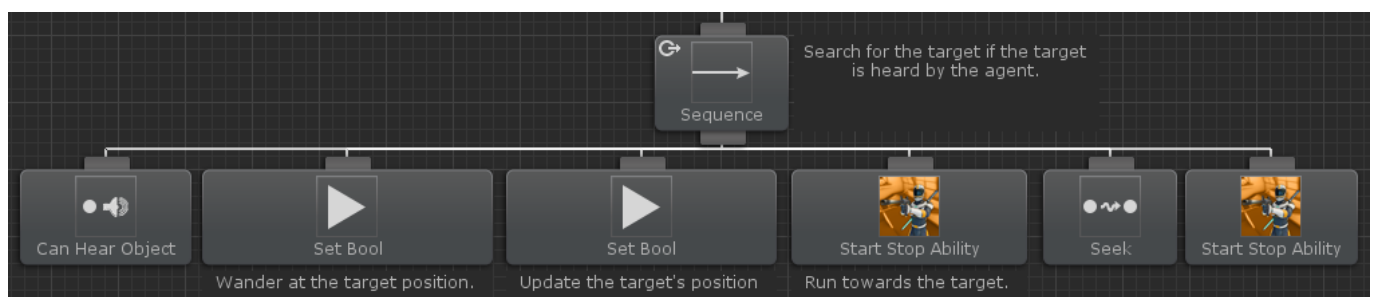


When the target was first acquired the Wander and Update Position variables were set so the agent will search for the target if the target is lost. If the target is killed by the agent then there is no need for the agent to wander so the Target Death branch will reset all of the attacking related variables back to the default.



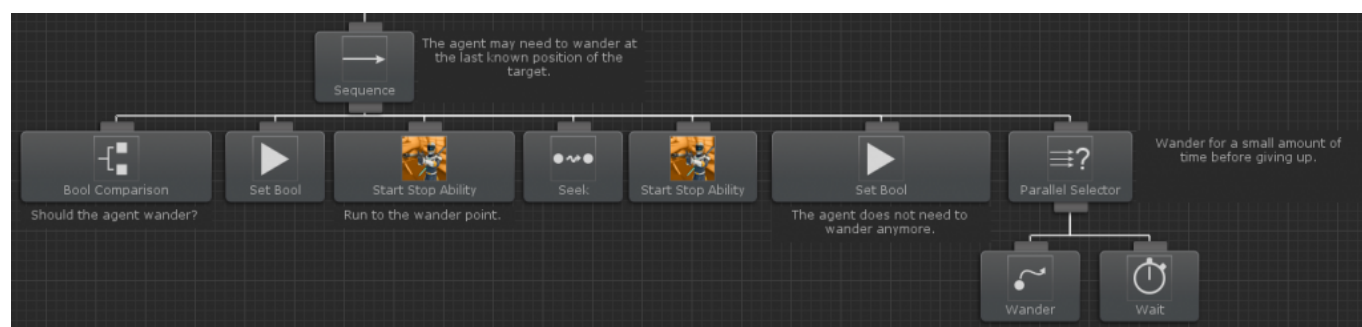
If the agent doesn't need to get ammo and the player is not within sight then the next highest priority branch will run. This branch uses the Ultimate Character Controller's event system to determine if the agent has taken damage. If the agent has taken damage then the Has Taken Damage task will return success and the branch will execute. When the agent has taken damage the branch will first get the position of the object that it took damage from and set Wander to true. Before the agent moves towards the target it will first determine if their health is critically low. If the health is critically low then the agent will seek towards a health pack.

After the agent has either retrieved a health pack or determines that more health is not needed the branch will end. Remember that the Wander variable was set to true so the Wander branch will start executing if there are no more higher priority branches.



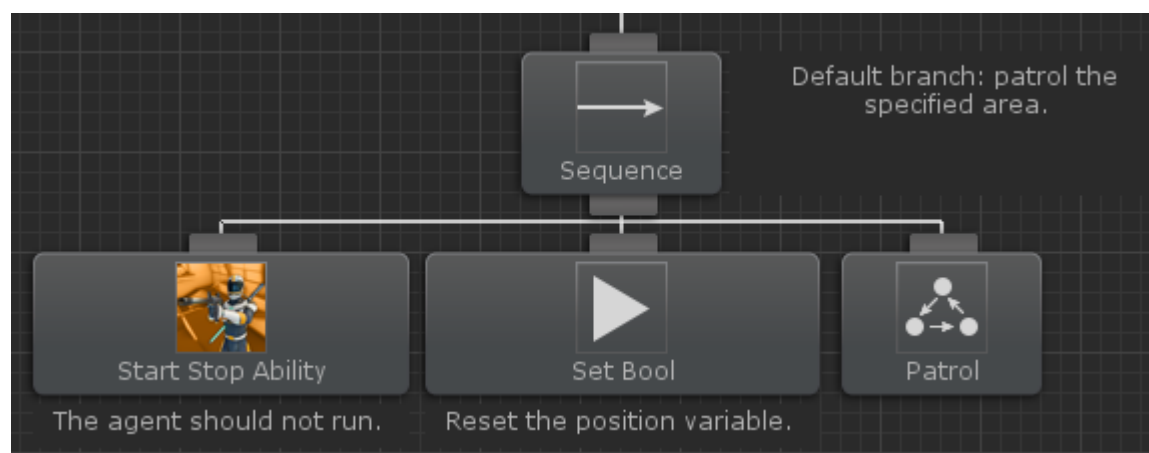
The next highest priority branch is the Can Hear branch. This branch uses the Can Hear Object task from the Movement Pack to determine if any audio sources are emitted a sound.

If any audio is heard then the agent will run towards the audio source position by starting the Speed Change task and using the Seek task to move into position. The Wander variable is also set to true so the agent will wander if the target can't be seen by the time the agent arrives at the Seek location.

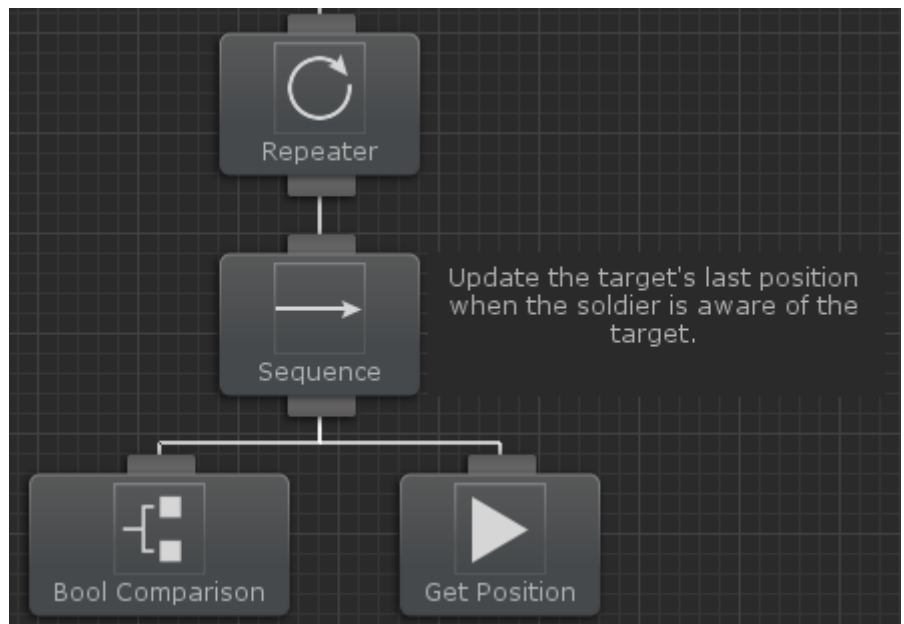


Throughout the execution of the tree there have been multiple cases where the Wander task was set to true. The Bool Comparison task will compare this Wander task variable to true to determine if the agent should wander. Before the agent can wander they first have to Seek to the Last Position. This variable will be set whenever the Update Position variable is true and it will indicate the last position that the target was located. When the agent arrives at the Last Position the Wander task will then execute which allows the agent to search for the target.

Remember that during this time the higher priority branches are being reevaluated so if the player comes within sight of the agent at any time then the Can See branch will abort the current branch so the character can start to attack. The Wander task is parented to the Parallel Selector task which will execute all children until a child returns success. To the right of the Wander task is the Wait task which will return success after a predetermined duration. This will prevent the agent from wandering forever and put a cap to the amount of time that the agent wanders.



If no other branches need to execute then the tree will fall back to the Patrol branch. This branch will move the character between patrol points and keep the character moving until a



higher priority branch aborts it.

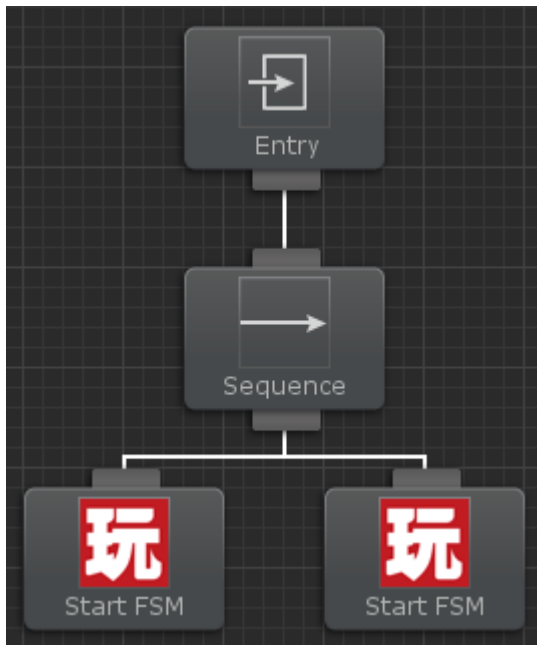
As the Ammo, Can See, Taken Damage, Can Hear, Wander, and Patrol branches are executed near the top of the tree the Update Position branch will also execute. This branch will update the position of the target when the Update Position is true. This branch is separated out from the rest of the tree because not just a single branch can set Update Position to true so to prevent having to add the same tasks multiple times this branch has been added near the top under a Parallel task.

Now that you've completed this overview you should have enough knowledge to get started to create you own tree with the Ultimate Character Controller. Make sure you go through the demo tree a few times to get familiar with Behavior Designer's visual editor.

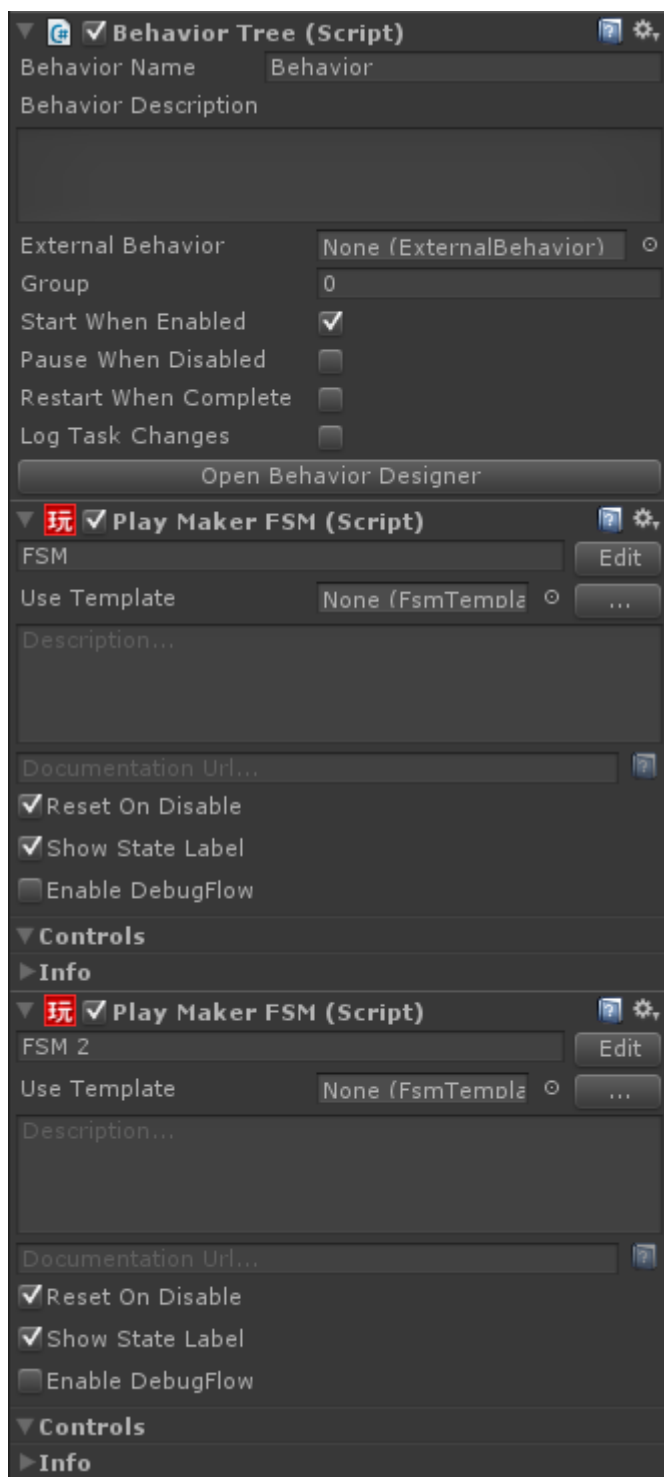
Playmaker

[Playmaker](#) is a popular visual scripting tool which allows you to easily create finite state machines. Behavior Designer integrates directly with PlayMaker by allowing PlayMaker to carry out the action or conditional tasks and then resume the behavior tree from where it left off. PlayMaker integration files are located on the [integrations page](#) because Playmaker is not required for Behavior Designer to work.

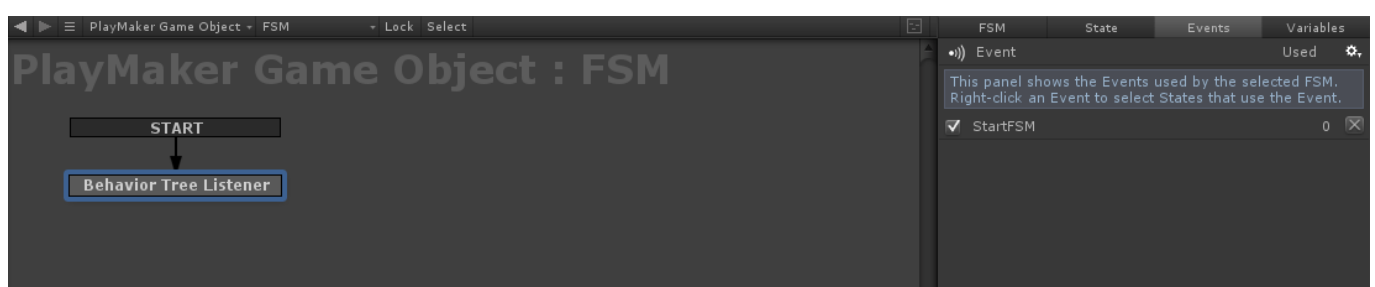
To get started, first make sure you have Playmaker installed and have imported the integration package. Once those files are imported you are ready to start creating behavior trees with Playmaker. To get started, create a very basic tree with a sequence task who has two Start FSM child tasks:



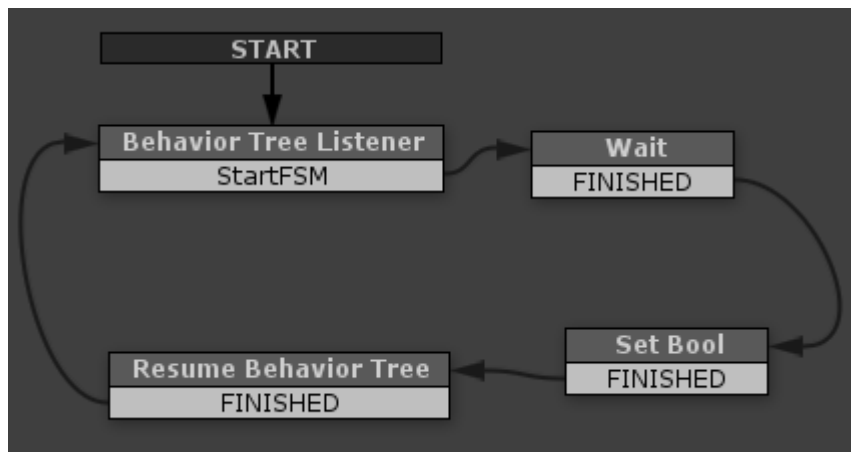
Next add two Playmaker FSM components to the same game object that you added the behavior tree to.



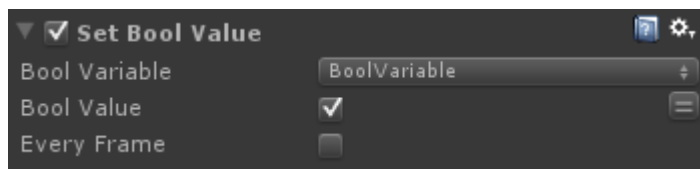
Open Playmaker and start creating a new FSM. This FSM is going to be a simple FSM to show how Behavior Designer interacts with Playmaker. For a more complicated FSM take a look at the Playmaker sample project. Behavior Designer starts the Playmaker FSM by sending it an event. Create this event by adding a new state called “Behavior Tree Listener” and adding a new global event called “StartFSM”. The event must be global otherwise Behavior Designer will never be able to start the FSM.



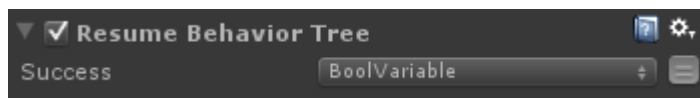
Add a transition from that event along with a wait state, a set bool state, and a resume behavior tree state. Make sure you transition from the Resume Behavior Tree state to the Behavior Tree Listener state so the FSM can be started again from Behavior Designer.



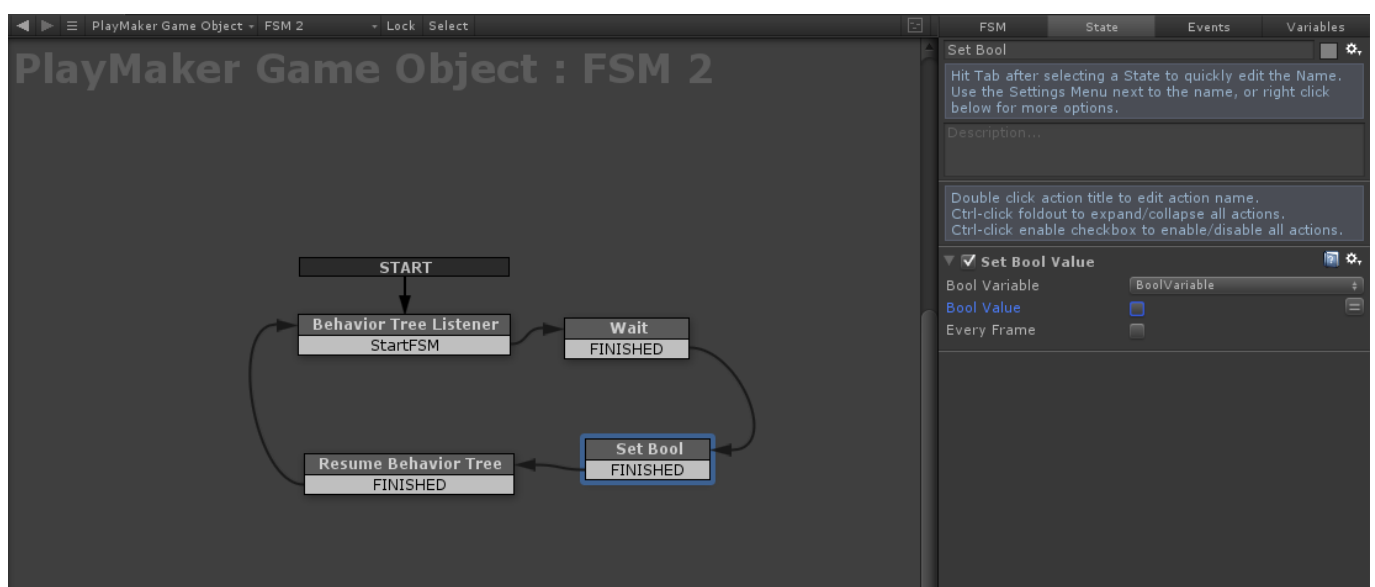
Create a new variable within the Set Bool state and set that value to true.



Then within the Resume Behavior Tree state we want to return success based off of that bool value:

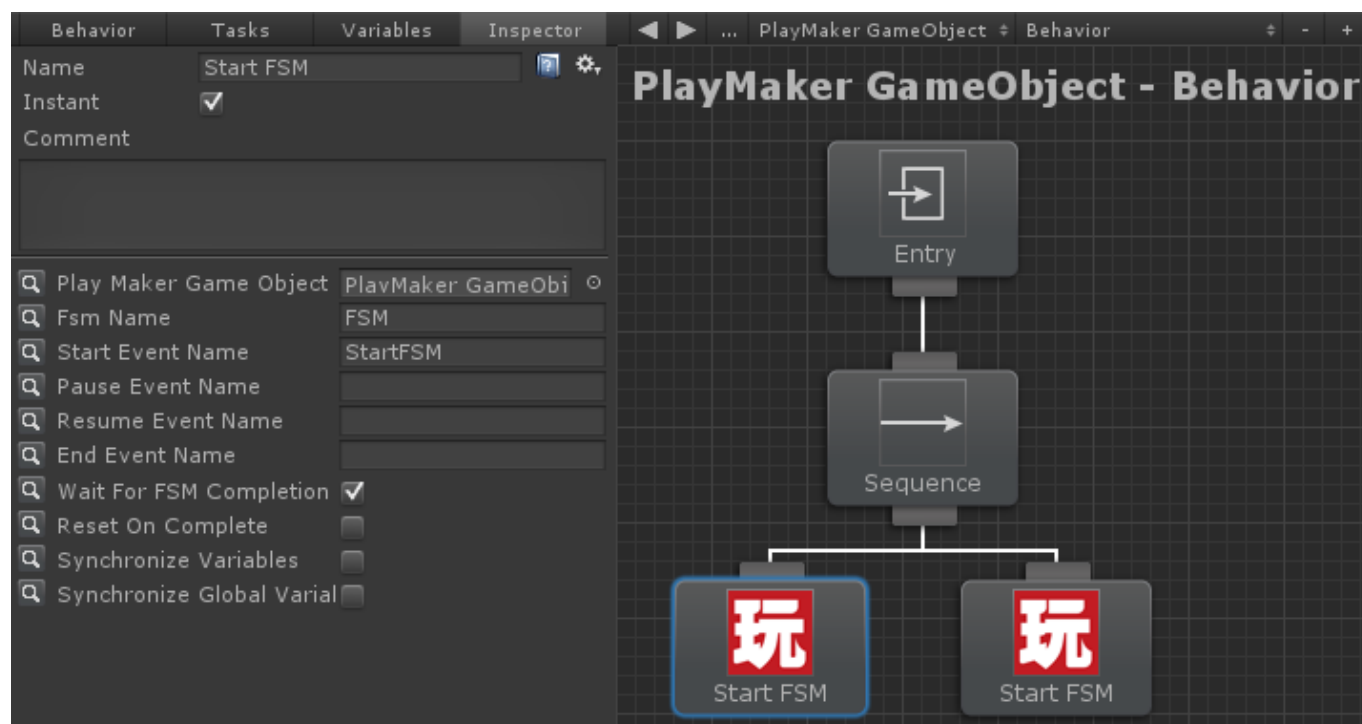


That's it for this FSM. Create the same states and variables for the second FSM that we created earlier. Do not set the bool variable to true for this FSM.

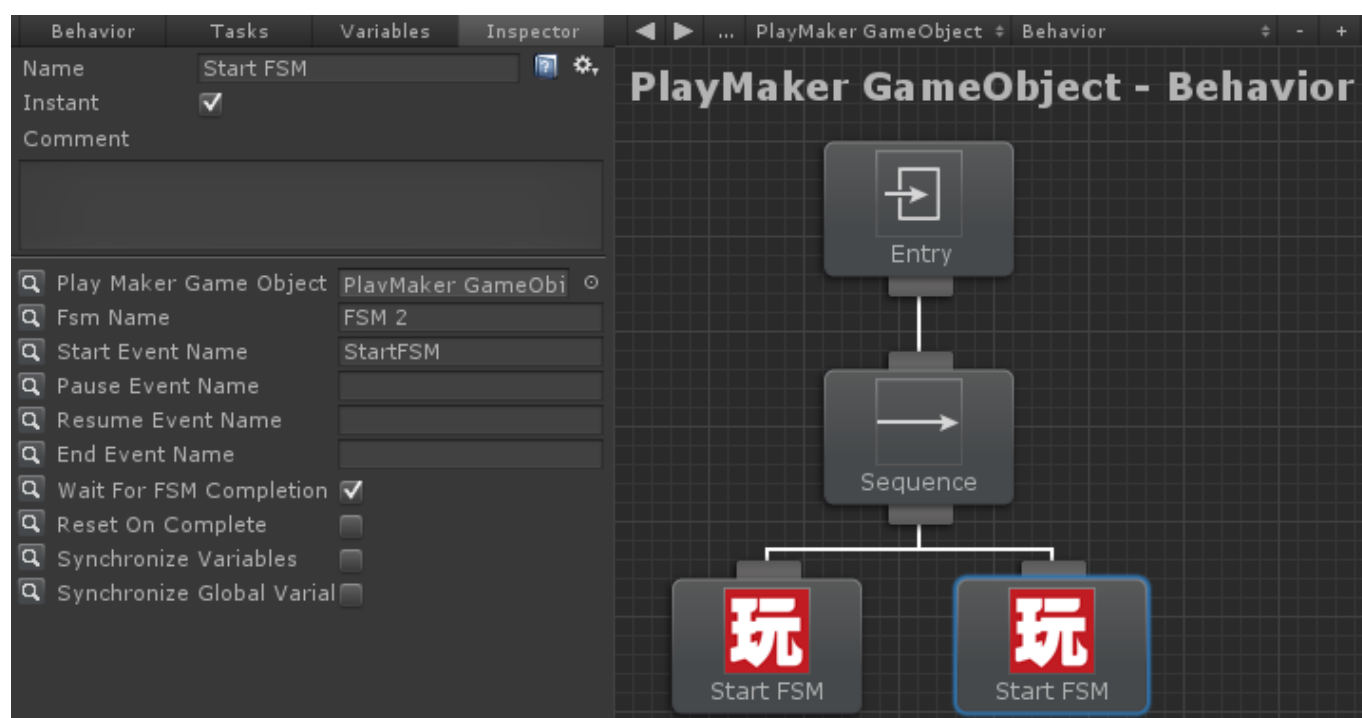


We are now done working in Playmaker. Open your behavior tree back up within Behavior Designer. Select the left Playmaker task and start assigning the values to the variables. Playmaker Game Object is assigned to the game object that we added the Playmaker FSM components to. FSM Name is the name of the Playmaker FSM. Event name is the name of

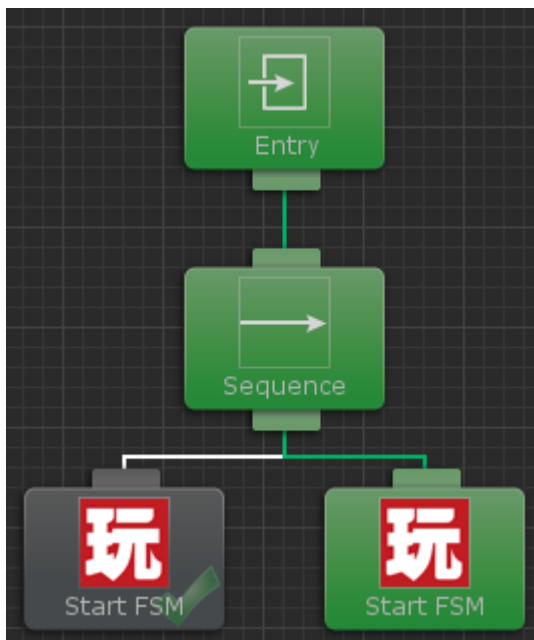
the global event that we created within Playmaker.



Now we need to assign the values for the right Playmaker task. The values should be the same as the left Playmaker task except a different FSM Name.



That's it! When you hit play you'll see the first Playmaker task run for a second and then the second Playmaker task will start running.

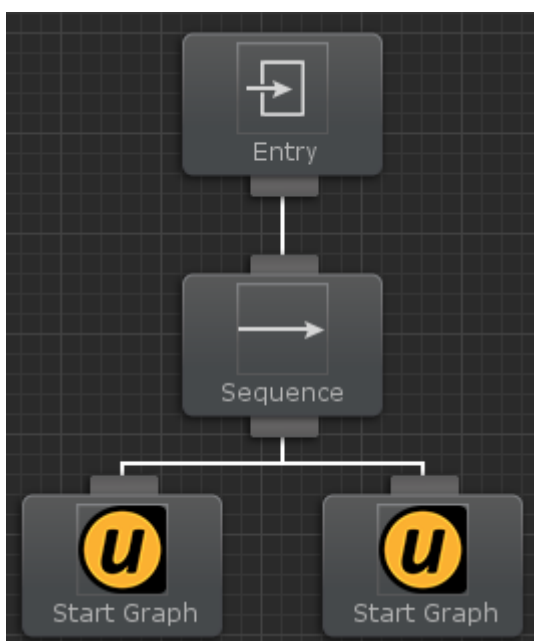


If you were to swap the tasks so the second Playmaker task runs before the first Playmaker FSM, the behavior tree will never get to the first Playmaker FSM because the second Playmaker FSM returned failure and the sequence task stopped executing its children.

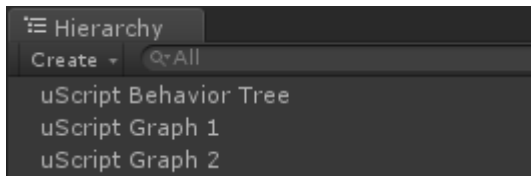
uScript

[uScript](#) is a visual scripting tool which allows you to create complicated setups without needing to write a single line of code. Behavior Designer integrates directly with uScript by allowing uScript to carry out the action or conditional tasks and then resume the behavior tree from where it left off. uScript integration files are located on the [integrations page](#) because uScript is not required for Behavior Designer to work.

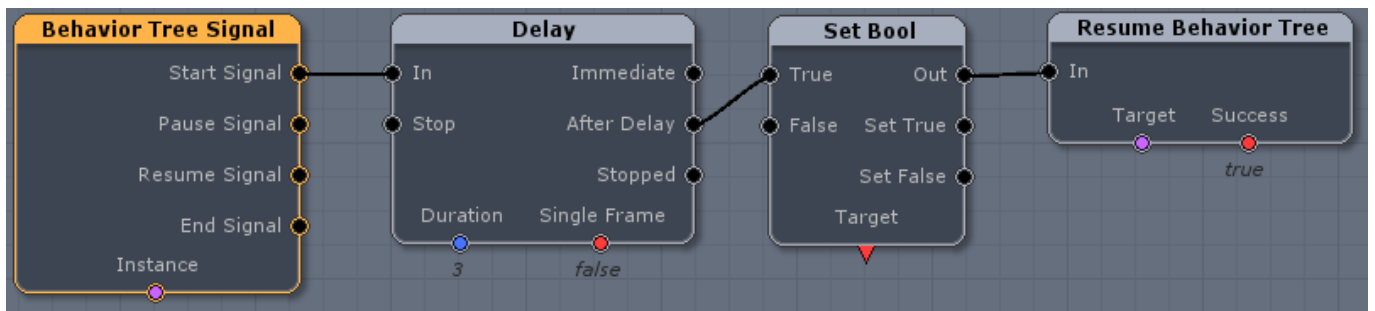
To get started, first make sure you have uScript installed and have imported the integration package. Once those files are imported you are ready to start creating behavior trees with uScript. To get started, create a very basic tree with a sequence task who has two Start Graph child tasks:



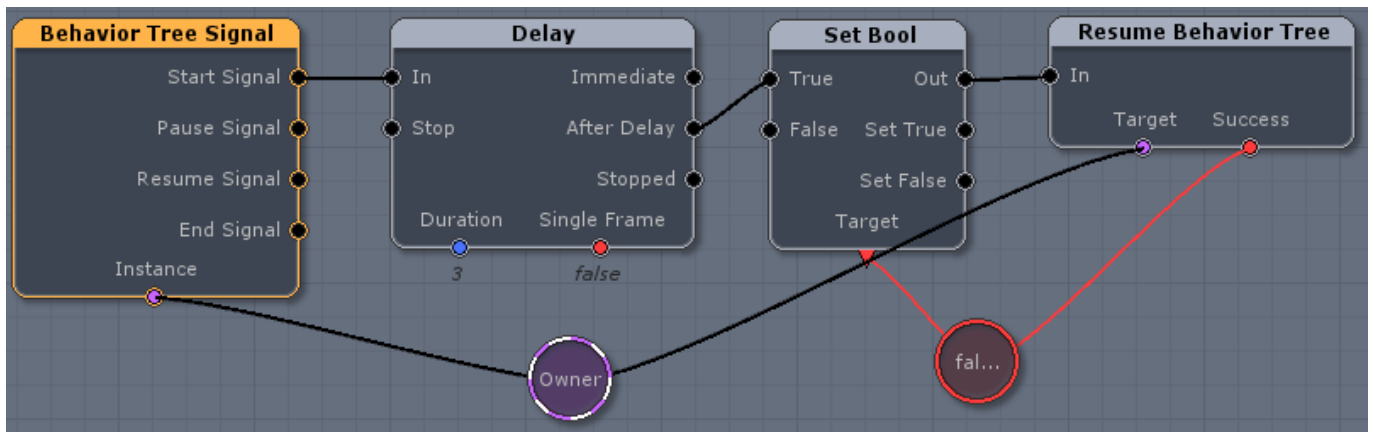
Now we need to create two GameObjects which will hold the compiled uScript graph:



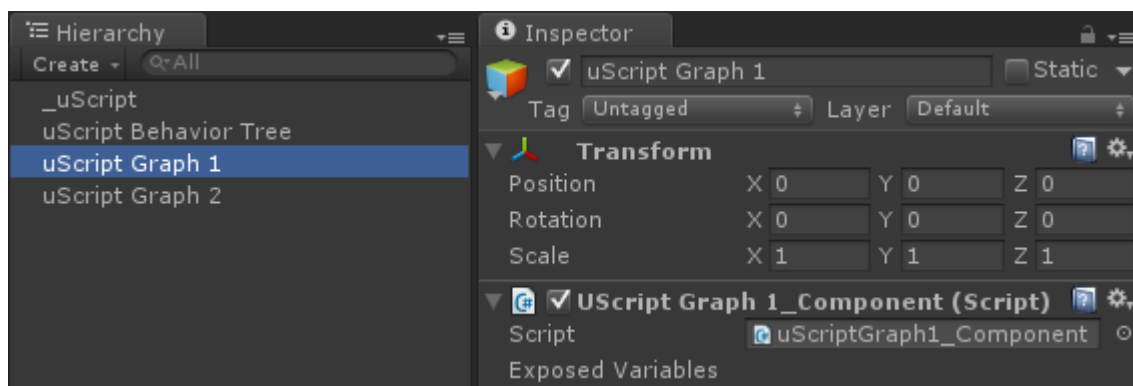
Open uScript and start creating a new graph. Add the Behavior Tree Signal node, located under Events/Signals. When Behavior Designer wants to start executing a uScript graph it will start from this node. This node contains four events: Start Signal, Pause Signal, Resume Signal, and End Signal. Start Signal is used when the behavior tree task starts running. Pause Signal gets called when the behavior tree is paused, and the Resume Signal gets called when the behavior tree resumes from being paused. Finally, End Signal gets called when the uScript task ends. For our graph we are only going to create a few nodes, the uScript sample project shows a more complicated uScript graph. Create a node which has a delay of 3 seconds, sets a bool, then resumes the behavior tree. The Resume Behavior Tree node is located under Actions/Behavior Designer:



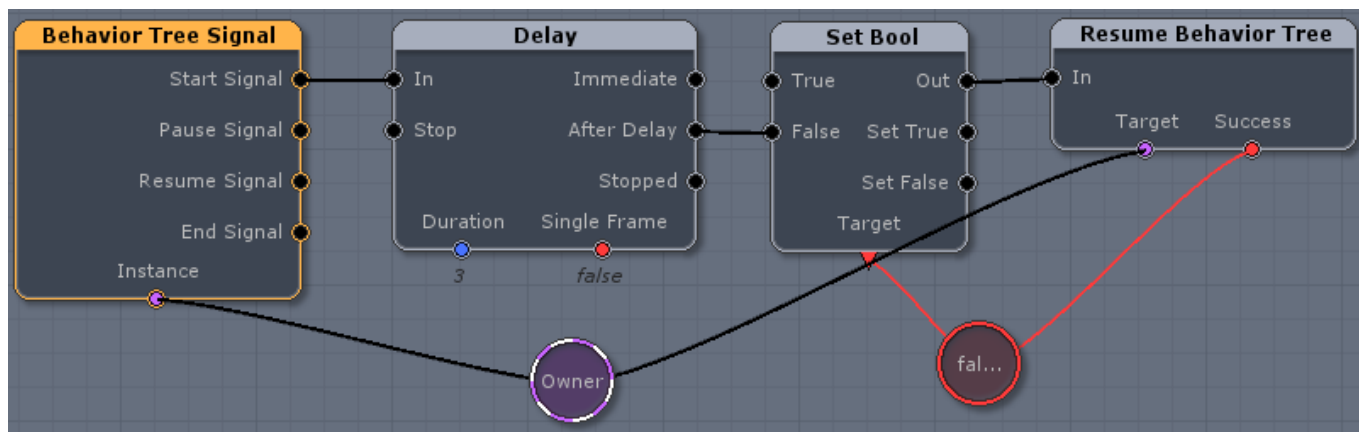
Now we need to create a Owner GameObject and bool variable.



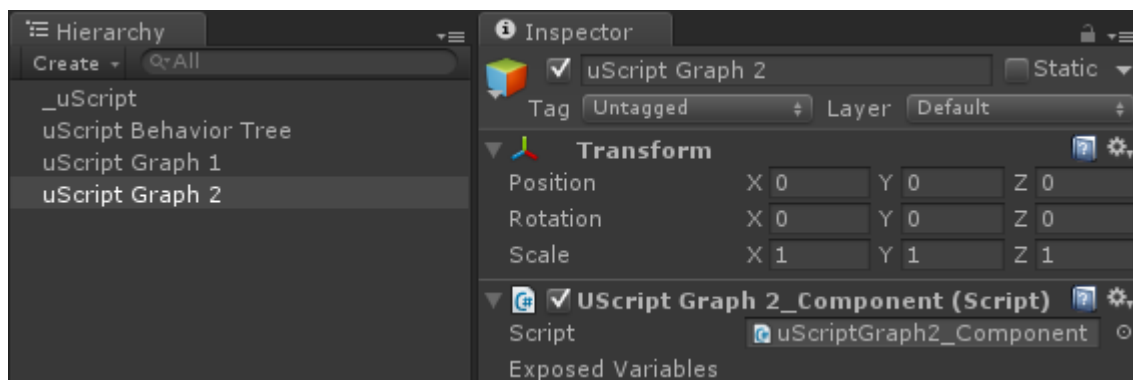
Save the uScript graph and assign the component to your first uScript graph GameObject. Answer no if uScript asks if you want to assign the component to the master GameObject.



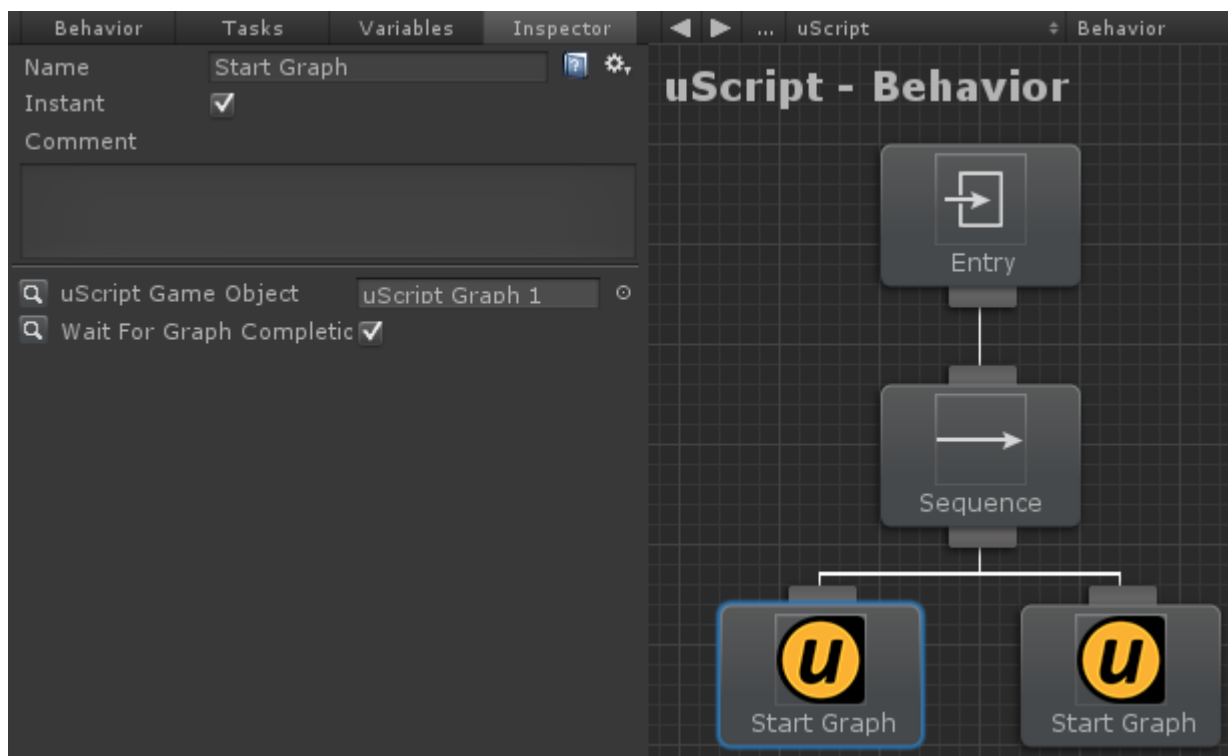
Create one more uScript graph. Make it the same as the last graph except set the bool to false:



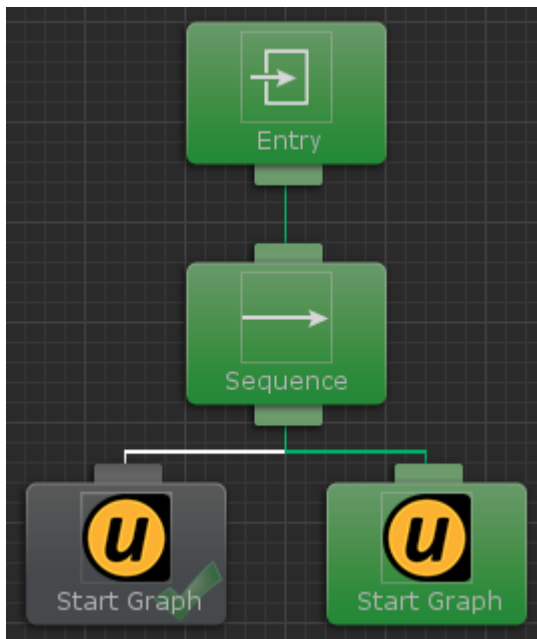
Finally save that graph and assign the component to the second uScript GameObject:



We're almost done. The only thing left to do is to assign the correct uScript GameObject to the tasks within Behavior Designer. Open your behavior tree within Behavior Designer again. Click on the left uScript task and assign the uScript GameObject to your first uScript graph GameObject.



Do the same for the right uScript task, only assign the uScript GameObject to your second uScript graph GameObject. That's it! When you hit play you'll see the first uScript task run for three seconds, followed by the second uScript task.



If you were to swap the tasks so the second uScript graph runs before the first uScript graph, the behavior tree will never get to the first uScript graph because the second uScript graph returned failure and the sequence task stopped executing its children.

Videos

The following videos will describe how to use Behavior Designer: