Behavior Designer Documentation

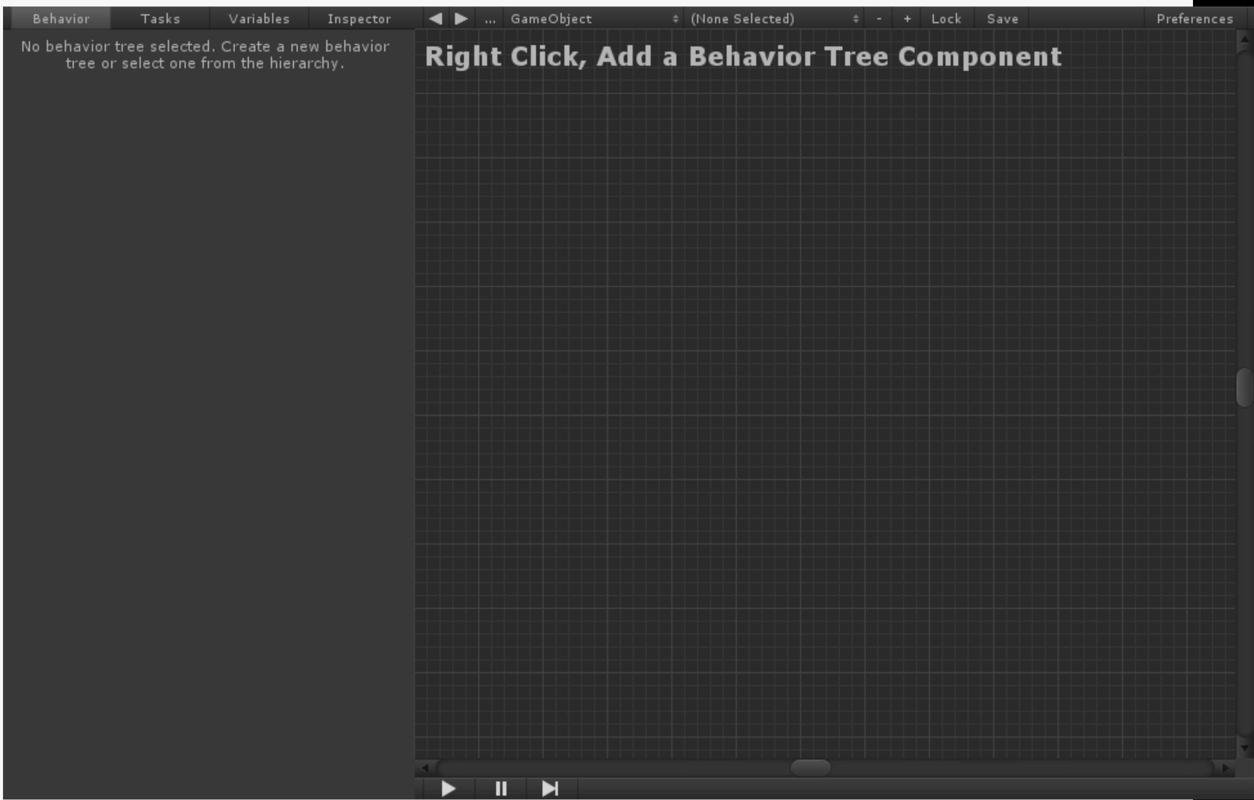
Thank you for your purchase! The most recent documentation (with better formatting) can be found online. If you have any questions feel free to post on the forums or email support@opsive.com.

Overview

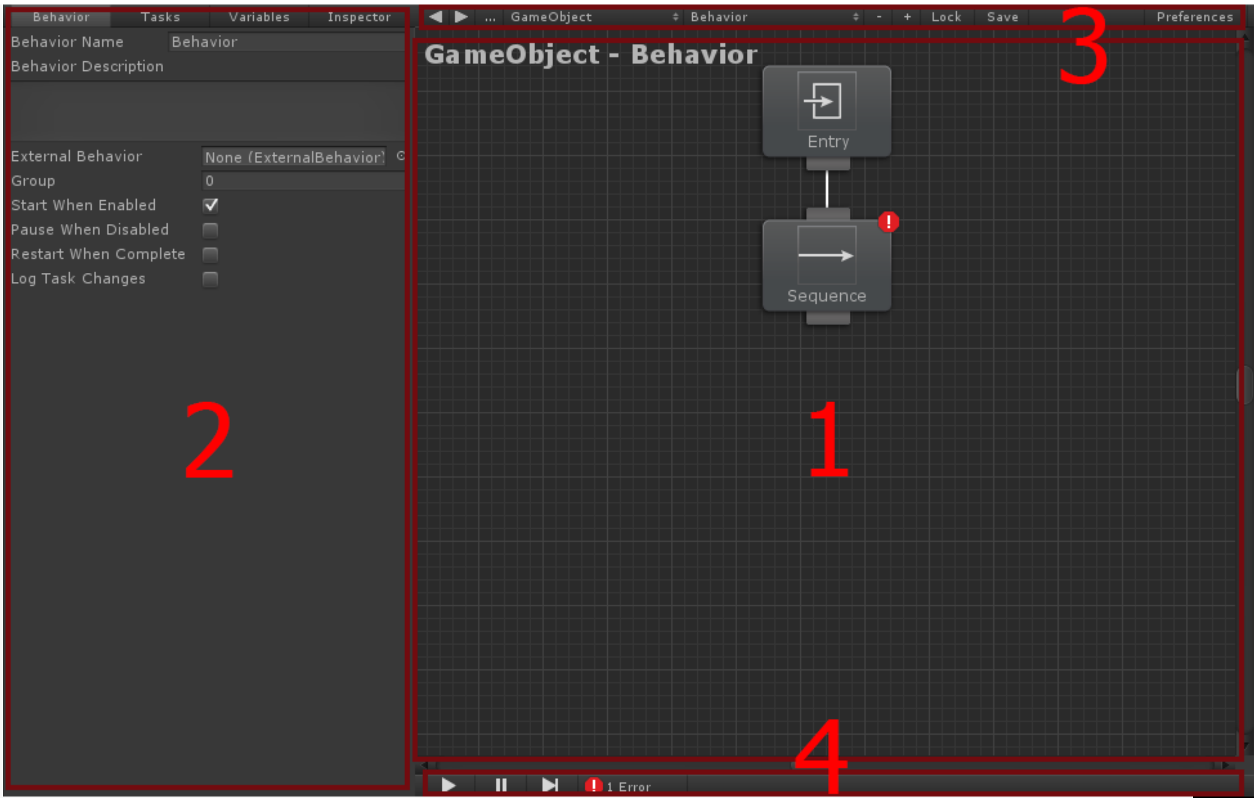
Behavior Designer is a behavior tree implementation designed for everyone - programmers, artists, designers. Behavior Designer offers a powerful API allowing you to easily create new tasks. it offers an intuitive visual editor with PlayMaker and uScript integration which makes it possible to create complex 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 don't know what behavior trees are take a look at our quick overview of behavior trees. With Behavior Designer you don't need to know how behavior trees are implemented but it is a good idea to know some of the key concepts such as the types of tasks (action, composite, conditional and decorator). You can watch the video version of this topic here.

When you first open Behavior Designer you'll be presented with the following window:

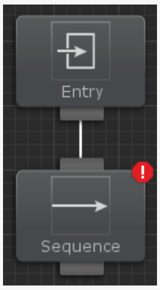


There are four sections within Behavior Designer. From the screenshot below, section 1 is the graph area. It is where you'll be creating the behavior trees. Section 2 is a properties panel. The properties panel is where you'll be editing the specific properties of a behavior tree, adding new tasks, creating 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.

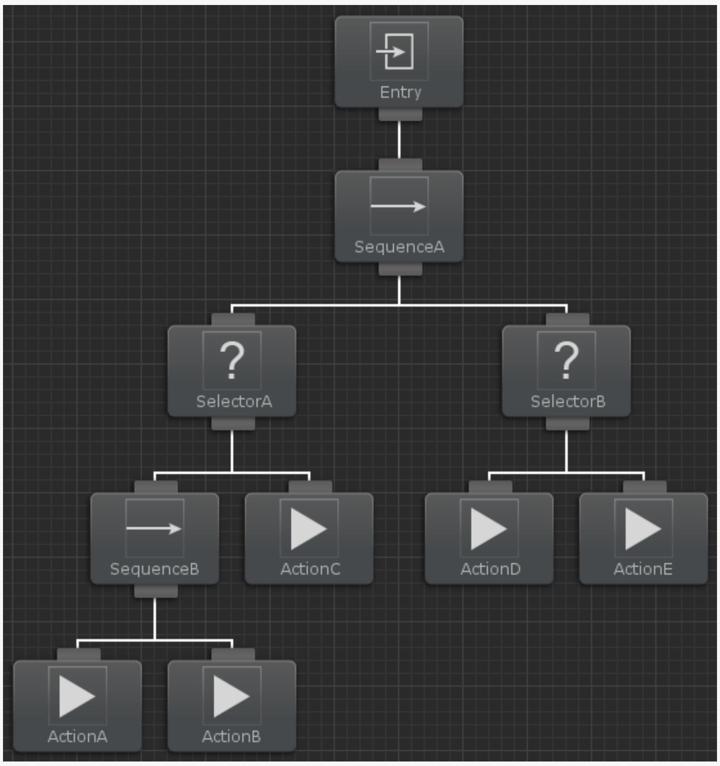


Section 1 is the main part of Behavior Designer that you'll be working in. Within this section you can create new tasks and arrange those tasks into a behavior tree. To start things off, you first need to add a Behavior Tree component. The Behavior Tree component will act as the manager of the behavior 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. Once a task has been added you'll see the following:



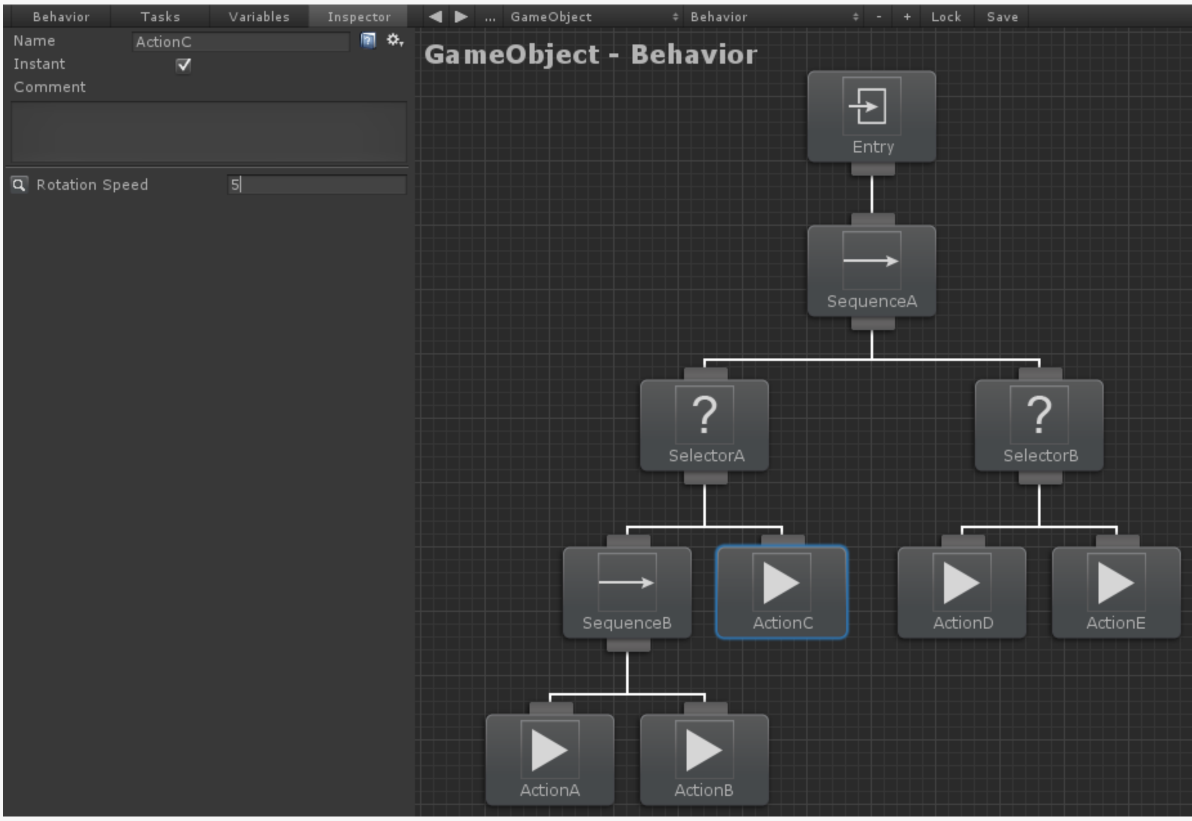
In addition to the task that you added, the entry task also gets added. The entry task acts as the root of the tree. That is the only purpose of the entry task. The sequence task has an error because it has no children. As soon as you add a child the error will go away. Now that we've added our first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

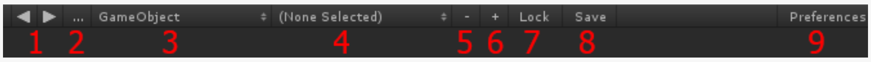
Behavior Designer will execute the tasks in a depth first order. You can change the execution order of the tasks by dragging them to the left/right of their sibling. From the screenshot above, the tasks will be executed in the following order:

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



Now that we have a basic behavior tree created, lets 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 can be found here.



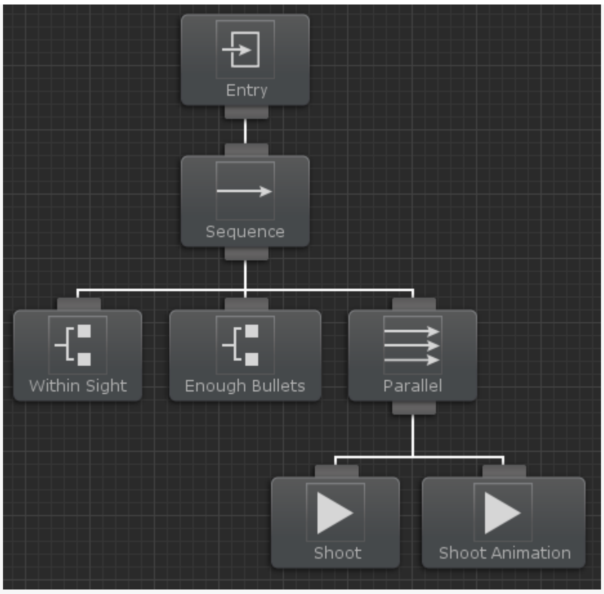
The final section within the Behavior Designer window is the operations toolbar. The operations toolbar is mostly used for selecting behavior trees as well as adding/removing behavior trees. The arrows with the number 1 label will navigate between the behavior trees that you have opened. The drop down box with the number 2 label will list all of the behavior trees that are within the scene or the project. This means that it will include prefabs. The drop down box with the number 3 label will list any game object that has a behavior tree component added to it. This is also within the scene or project. Finally, the drop down box with the number 4 label will list any behavior trees that are attached to the game object that is selected from the number 3 drop down box.

The button with the number 5 label will remove the currently selected behavior tree. The button with the number 6 label will add a new behavior tree. The "Lock" button (number 7) will keep the active behavior tree selected even if you select a different game object within the hierarchy or project window. The "Save" button (number 8) will save the current behavior tree out as an asset. Finally, the "Preferences" button (number 9) will show any 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.

If you would rather see a behavior tree in action rather than read about it take a look at the Behavior Designer trial version and load any of the sample projects. The sample project videos will walk you through how the behavior tree works.



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.

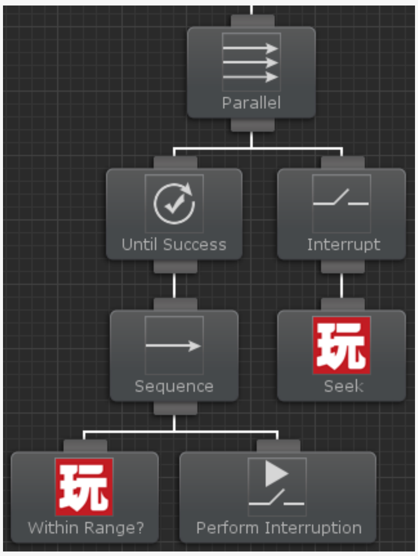
Behavior Designer takes all of these concepts and packages it up in an easy to use interface with an API that is similar to Unity's MonoBehaviour API. Behavior Designer includes many composite and decorator classes within the standard installation. Action and conditional tasks are more game specific 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, or they can be created using PlayMaker or uScript. In addition, many videos have been created to make learning Behavior Designer as easy as possible.

For information on the implementation of a behavior tree, take a look at this AltDevBlog post.

Behavior Trees or Finite State Machines

On the Unity Forums SteveB asked an interesting question: why a behavior tree and why not a finite state machine (PlayMaker)? According to some, the age of finite state machines is over. We aren't going to go that far, but we are going to say that a finite state machine should not be the only AI technique that you use in your game. The true power comes when you combine both behavior trees and finite state machines together.

Before we continue, we want to point out that finite state machines are by no means required for behavior trees to work. Behavior trees work exceptionally well when used all by themselves. The CTF and RTS sample projects were created using only behavior trees. Behavior trees describe the *flow* of the AI whereas finite state machines can be used to describe the *function*.



Behavior trees have a few advantages over finite state machines: they provide lots of flexibility, are very powerful, and they are really easy to make changes to. But they definitely do not replace the functionality of finite state machines. This is why when you combine a behavior tree with a finite state machine, you can do some really cool things.

Lets first look at the first advantage: flexibility. With a finite state machine (such as PlayMaker), how do you run two different states at once? The only way we have figured it out is to create two separate finite state machines. 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. In addition, lets say that you also have another task running in parallel and it detects a condition where it needs to stop the PlayMaker tasks from running. All you need to do for this situation is add an interrupt task and that task will be able to end the PlayMaker tasks immediately.

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.

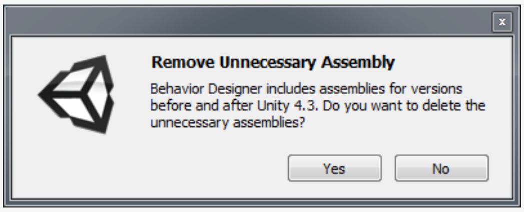
Another advantage of behavior trees are that they are powerful. That isn't to say that finite state machines aren't powerful, it is just that they are powerful in different ways. In our view behavior trees allow your AI to adopt to current game state easier than finite state machines do. It is easier to create a behavior tree that will adopt 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.

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 finite state machine you have to change the transitions between states. With a behavior tree, all you have to do is drag the task. You don't really 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.

Just like behavior trees have advantages over finite state machines, finite state machines have different advantages over behavior trees. This is why the true magic happens when you join a behavior tree with a finite state machine. You can use PlayMaker for all of the condition/action tasks and Behavior joining Behavior Designer with PlayMaker is where the true magic happens. You can use PlayMaker for all of the condition/action tasks and Behavior Designer for the composite/decorator tasks. With this setup you'd be playing off of each others strengths. The flexibility of a BT and the functionality of a finite state machine.

Installation

Behavior designer ships with four assemblies which contain versions that run on Unity 3.5.7 - 4.2.2 ("pre4_3") and Unity 4.3+ ("post4_3"). Immediately after Behavior Designer is imported a dialog will pop up asking if you want Behavior Designer to remove the unnecessary assembly:



If you select yes the script will automatically remove the assembly that does not correspond with your Unity version. If you select no the script will not run and you will keep getting this message every time you import until you either manually remove the assembly or manually remove the script. If you choose to remove the files manually they are located at:

```
/Assets/Behavior Designer/Editor/BehaviorDesignerEditor.dll.post4_3
/Assets/Behavior Designer/Editor/BehaviorDesignerEditor.pre4_3.dll
/Assets/Behavior Designer/Editor/DLLSelector.cs
/Assets/Behavior Designer/Runtime/BehaviorDesignerRuntime.dll.post4_3
/Assets/Behavior Designer/Runtime/BehaviorDesignerRuntime.pre4_3.dll
```

You'll need to reimport Behavior Designer if you if you import the pre4_3 assemblies and later update to Unity 4.3+. After Behavior Designer is imported you can access it from the Window toolbar. If you will be writing your tasks in UnityScript you will need to make a minor directory change to enable the UnityScript class to see the C# classes.

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.

Accessing UnityScript/Boo Tasks

Even though all of the Behavior Designer tasks are written in C#, tasks can also be written in UnityScript or Boo. Due to the order that Unity compiles scripts, you'll first need to rearrange the Behavior Designer directory. By default, Behavior Designer installs in the following locations:

```
/Behavior Designer/Editor/...
/Behavior Designer/Runtime/...
/Behavior Designer/Third Party/...
/Gizmos
```

The only change that you need to make is to move the Runtime and Third Party directories to a folder that gets compiled first, such as Plugins. You will then have the following directory structure:

```
/Behavior Designer/Editor/...
/Gizmos
/Plugins/Behavior Designer/Runtime/...
/Plugins/Behavior Designer/Third Party/...
```

You will then be able to inherit your UnityScript/Boo object from a Task subclass, just as you would in C#. For example, the following UnityScript task is inherited from Action:

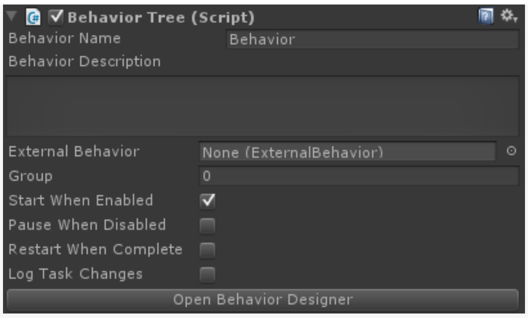
```
#pragma strict

class UnityScriptAction extends BehaviorDesigner.Runtime.Tasks.Action
{
 ...
}
```

If you have extracted the runtime source code you will need to make a similar change.

Compiling for the Windows Store/Phone

In order to compile Behavior Designer for the Windows Store and Windows Phone you must use the runtime source code instead of the compiled DLL. For instructions on how to extract the runtime source code take a look at the bottom of the installation topic. No compile settings need to be changed - Behavior Designer can compile with .Net Core enabled.

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 behavior tree component has the following properties:

Name	Description
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.
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.

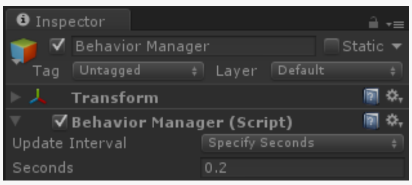
Creating a Behavior Tree from Script

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;

void Start () {
var bt = gameObject.AddComponent();
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 BehaviorManager 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);
```

Name	Description
Update Interval	An enum that specifies how often the behavior trees should update.

Tasks

At the highest level a behavior tree is a collection of tasks. Tasks have a really 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:

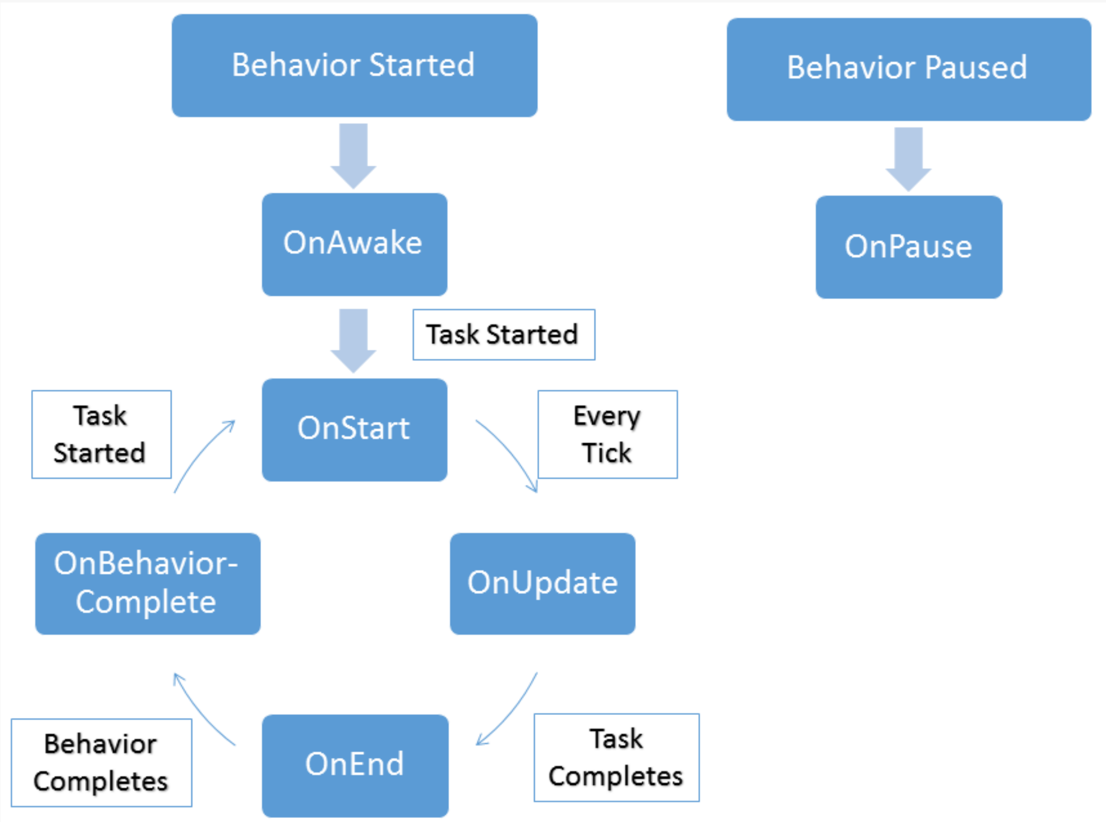
```
// OnAwake is called once when the behavior tree is enabled. Think of it as a constructor
```

```
public virtual void OnAwake();

// OnStart is called immediately before execution. It is used to setup any variables that need to be reset from the previous run
public virtual void OnStart();

// OnUpdate runs the actual task
public virtual TaskStatus OnUpdate();

// OnEnd is called after execution on a success or failure.
public virtual void OnEnd();

// OnPause is called when the behavior is paused and resumed
public virtual void OnPause(bool paused);

// The priority select will need to know this tasks priority of running
public virtual float GetPriority();

// OnBehaviorComplete is called after the behavior tree finishes executing
public virtual void OnBehaviorComplete();

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

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

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

Tasks are derived from the Unity class ScriptableObject. ScriptableObject is not derived from MonoBehaviour so normally to get access to the game object that this behavior tree is attached to you would have to do Owner.gameObject or Owner.GetComponent. To prevent you from having to do that with every single task we have added properties that already do that for you. You can directly call gameObject and it will return the gameObject that the task is attached to. Unlike MonoBehaviour, this property is cached so you don't even have to cache it yourself.

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 OnChildRunning();

// Notifies the parallel task that the child at index childIndex has started to run
public virtual void OnChildRunning(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 here.

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 fieldOfViewAngle is the field of view that the object can see. targetTag 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, possibleTargets, is a cache of all of the Transforms with the targetTag. If you take a look at the task API, you can see that we can create that cache within the 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 targetTag, then loop through them caching their transform in the possibleTargets array. The possibleTargets 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 possibleTargets 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 to 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 then fieldOfViewAngle then the target transform is within sight of the current transform. One thing to note is that unlike MonoBehaviour objects, all tasks already have all of the MonoBehaviour components cached so we do not need to precache the transform component.

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 targetting:

```
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:

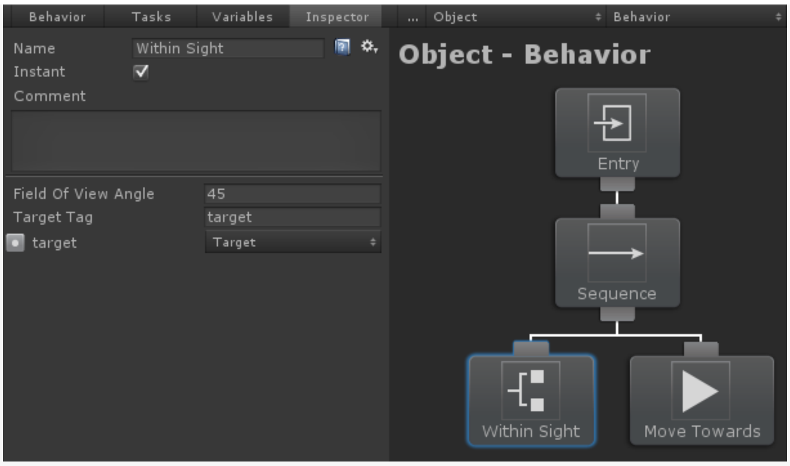
```
public override TaskStatus OnUpdate()
{
if (Vector3.SqrMagnitude(transform.position - target.Value.position) < 0.1f) {
return TaskStatus.Success;
}
transform.position = Vector3.MoveTowards(transform.position, target.Value.position, speed * Time.deltaTime);
return TaskStatus.Running;
}
```

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:

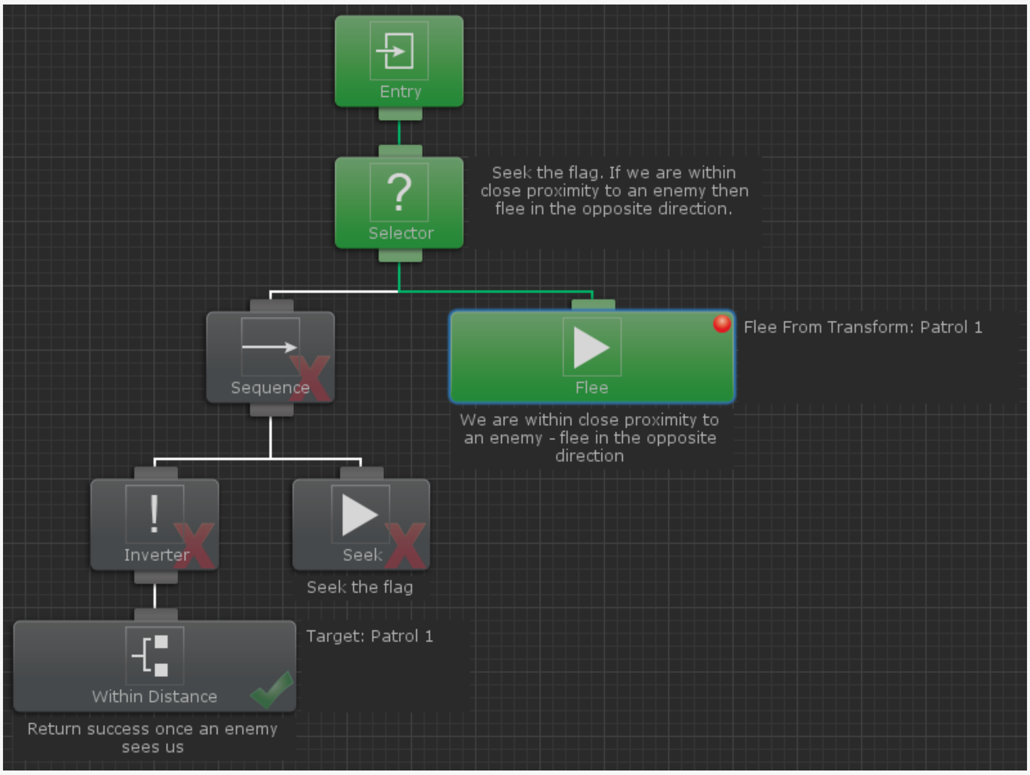
```
using UnityEngine;
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;

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:

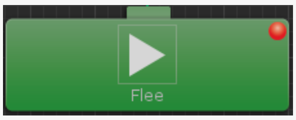


That's it! Create a few moving GameObjects within the scene assigned with the same tag as targetTag. When the game starts the object with the behavior tree attached with move towards whatever object first appears within its field of view. This was a pretty basic example and the tasks can get a lot more 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

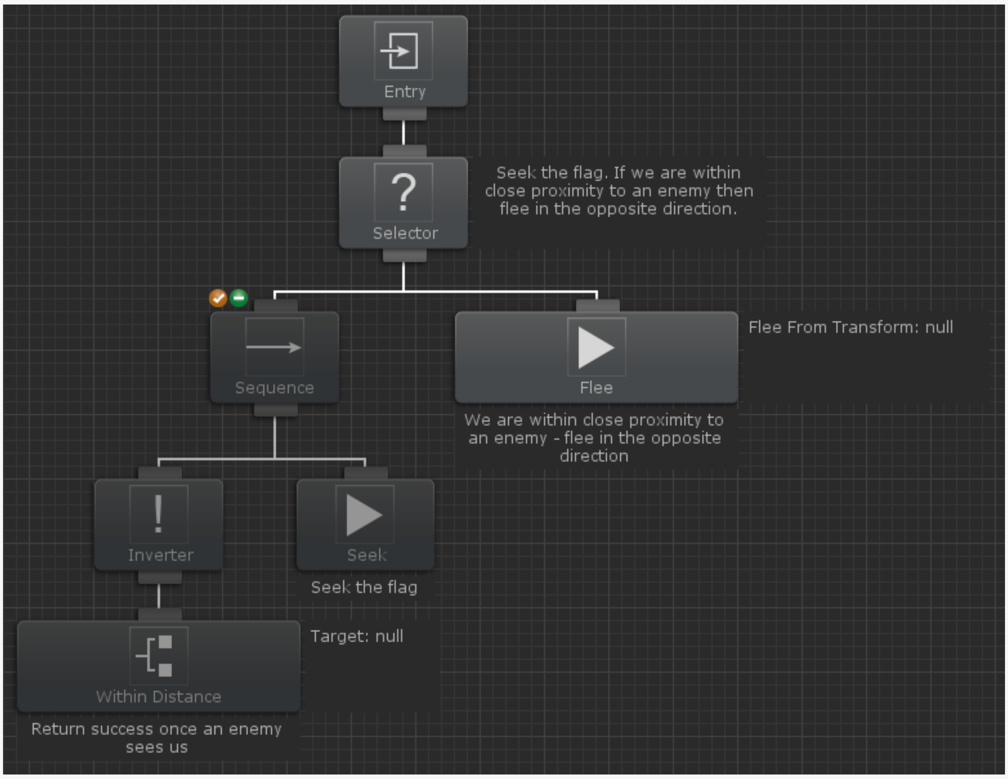


When a behavior tree is running you will see different tasks change colors between gray and green. When the task is green that means it is currently executing. When the task is gray it is not executing. After the task has executed it will have a check or x on the bottom right corner. If the task returned 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.

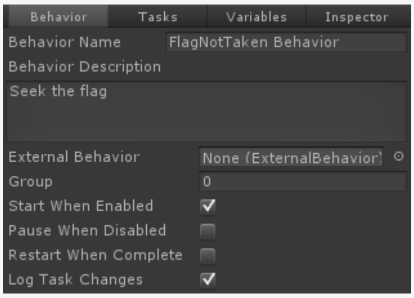


Right clicking on a task will bring up a menu which allows you to set a breakpoint. If a breakpoint is set on a particular task then Behavior Designer will pause Unity whenever that task is activated. This is useful if you want to see when a particular task is executed.

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:

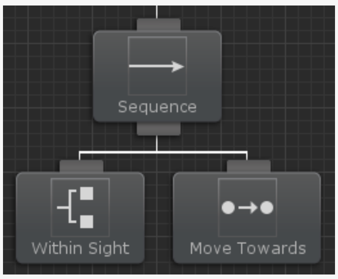
```
{game object name } - {behavior name}: {task change} {task type} (index {task index}) at stack index {stack index} {optional status}
```

{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.

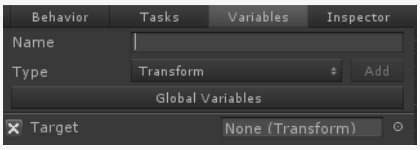
In our previous example we had two tasks: one that determined if the target is within sight and then the other task moves towards the target. This tree looks like:



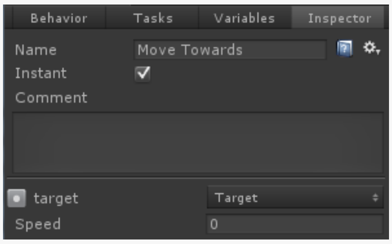
The code for both of these tasks is discussed in the Writing a New Task topic, but the part that deals with variables is in this variable declaration:

```
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.

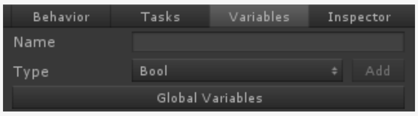
Behavior Designer supports both local and global variables. Global Variables are similar to local variables except any tree can reference the same variable. Variables can be referenced by non-Task derived classes by getting a reference to from the behavior tree.

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:

- SharedBool
- SharedColor
- SharedFloat
- SharedGameObject
- SharedGameObjectList
- SharedInt
- SharedObject
- SharedObjectList
- SharedQuaternion
- SharedRect
- SharedString
- SharedTransform
- SharedTransformList
- SharedVector2
- SharedVector3
- SharedVector4

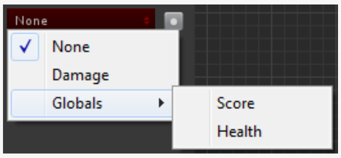
Global Variables

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 OBJECTTYPE should be replaced with the type of Shared Variable that you want to create.

```
[System.Serializable]
public class SharedOBJECTTYPE : SharedVariable
{
public OBJECTTYPE Value { get { return mValue; } set { mValue = value; } }
[SerializeField]
private OBJECTTYPE mValue;

public override object GetValue() { return mValue; }
public override void SetValue(object value) { mValue = (OBJECTTYPE)value; }

public override OBJECTTYPE ToString() { return mValue == null ? "null" : mValue.ToString(); }
}
```

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
{
public CustomClass Value { get { return mValue; } set { mValue = value; } }
[SerializeField]
private CustomClass mValue;

public override object GetValue() { return mValue; }
public override void SetValue(object value) { mValue = (CustomClass)value; }

public override string ToString() { return (mValue == null ? "null" : mValue.ToString()); }
}
```

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 methods

```
behaviorTree.GetVariableName("MyVariableName");
behaviorTree.SetVariableName("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;
}
}
```

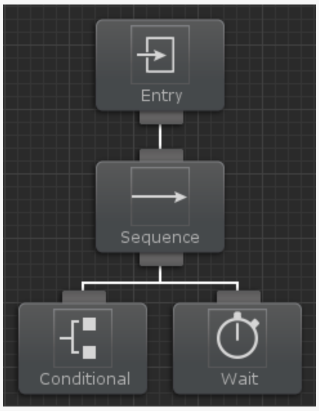
In the above example we are getting a reference to the variable named "MyVariable" within the Behavior Designer Variables pane. Also, as shown in the example, you can get and set the value of the variable with the SharedVariable.Value property.

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

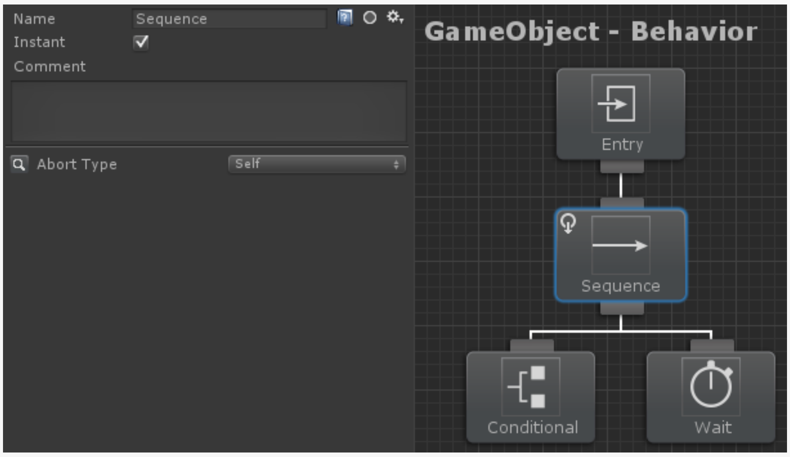
```
GlobalVariables.Instance.GetVariable("MyVariable");
GlobalVariables.Instance.SetVariable("Name", value);
```

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 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:

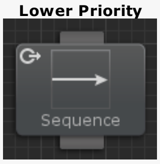


There are four different abort types: None, Self, Lower Priority, and Both.

None

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

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.

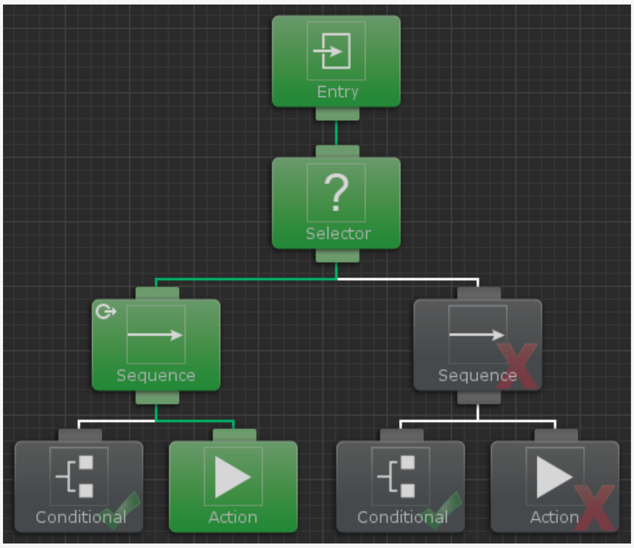
Self

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.

Both

This abort type combines both self and lower priority

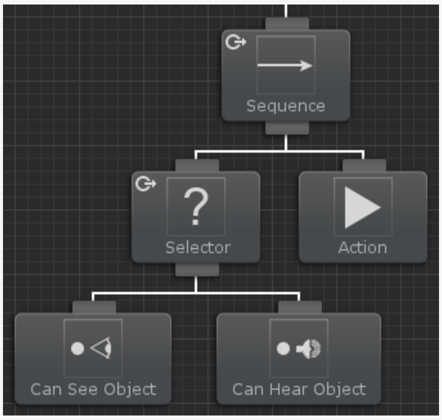
The following example will use the lower priority abort type:



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

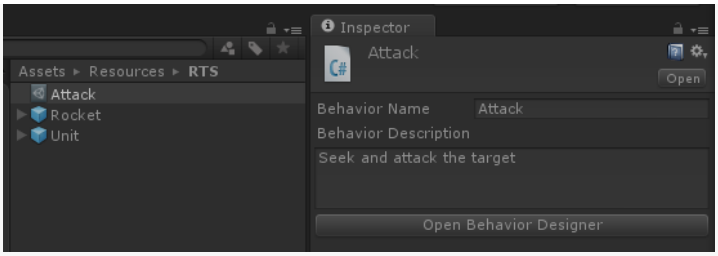
The conditional task's execution status will have a repeater icon around the success or failure status to indicate that it is being reevaluated by a conditional abort:

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 Self (or Both). If it does not have an abort type set then the two conditional tasks would not be reevaluated.

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. Furthermore, external behavior trees can [inherit](#) to make using external behavior trees even easier to use.

Referencing Tasks

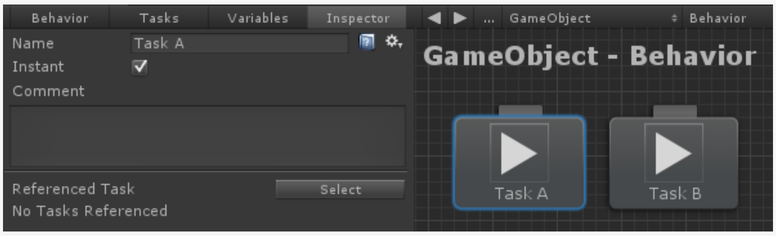
When writing a new task, in some cases it is necessary to access another task within that task. For example, TaskA may want to get the value of TaskB.SomeFloat. To accomplish this, TaskB needs to be referenced from TaskA. In this example TaskA looks like:

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

public class TaskA : Action
{
public TaskB referencedTask;

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

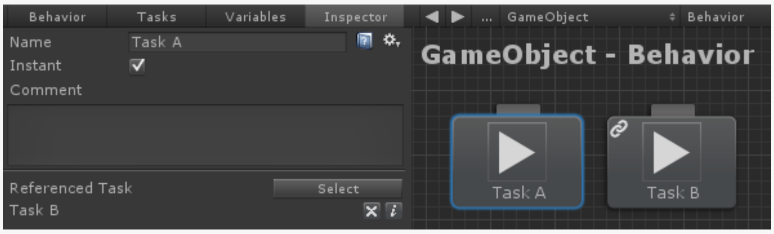
TaskB then looks like:

```
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:



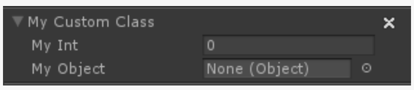
That is it. Now when you run the behavior tree TaskA will be able to output the value of TaskB's SomeFloat value. You can clear the reference by clicking on the "x" to the right of the referenced task name. If you click on the "i" then the linked task will highlight in orange:

Tasks can also be referenced using an array:

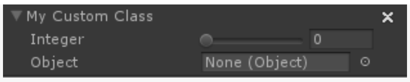
```
public class TaskA : Action
{
public TaskB[] referencedTasks;
}
```

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", 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;

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;

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:

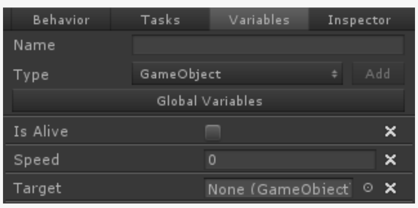
Variable Synchronizer

[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:

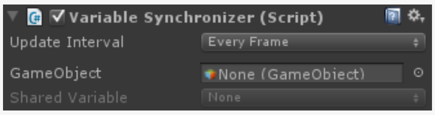
```
public bool isAlive { get; set; }
```

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

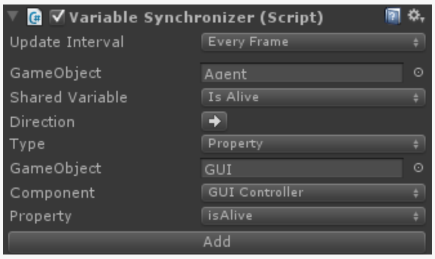
To setup the Variable Synchronizer, first make sure you have created the Shared Variables that you want to synchronize. For this example we created three Shared Variables:



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

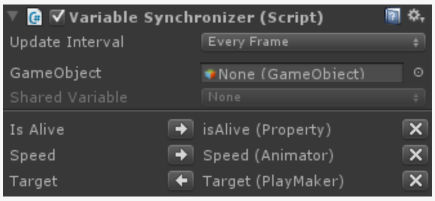


Next, start adding the Shared Variable that you want to keep synchronized. For this example we are going to add the Is Alive variable that was previously mentioned.



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.

Task Attributes

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

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:

```
[HelpURL("http://www.opsive.com/assets/BehaviorDesigner/documentation.php?id=27")]
public class Parallel : Composite
{
```

The HelpURL attribute takes one parameter which is the link to the webpage.

In addition to the HelpURL, a task can have the TaskIcon attribute:

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

Task icons are shown within the behavior tree 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".

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:

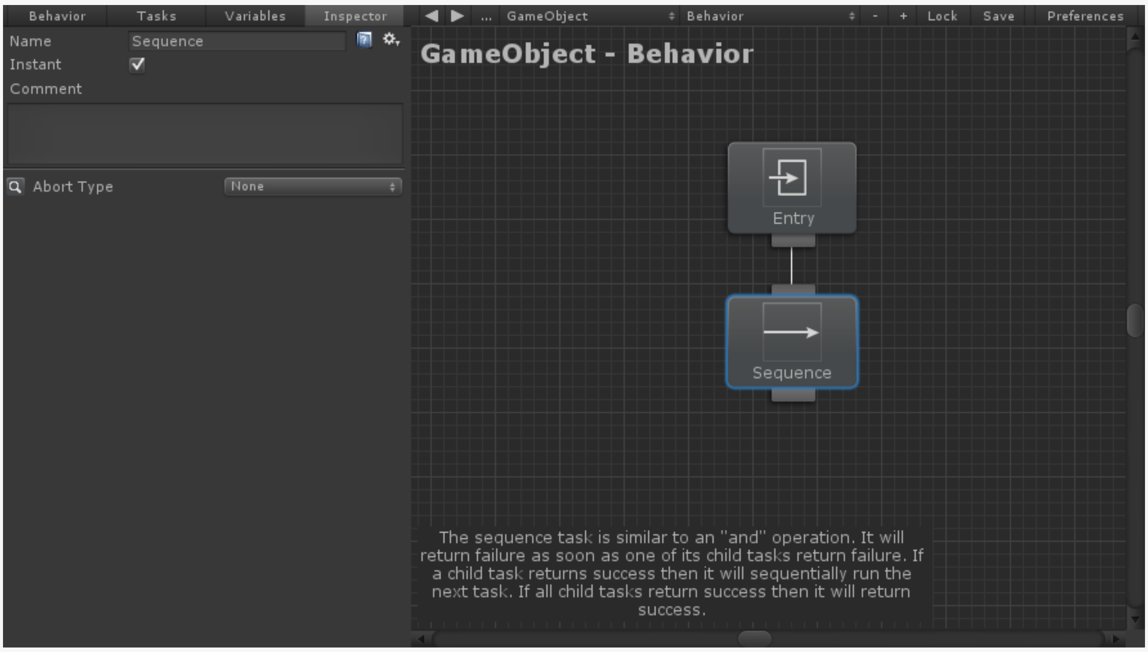
Categories may be nested by separating the category name with a slash:

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

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:



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.

The InheritedField attribute is the last attribute exposed by Behavior Designer. Imagine a situation where you have a lot of external trees and the only thing that changes between them is one variable, such as the speed that the unit moves. In previous Behavior Designer versions you would have to create multiple behavior trees each with a different speed set or use a blackboard class. You can now add the InheritedField attribute to a variable and the value will be passed down from the external behavior tree task. In our move speed example, this will allow you to only have one external tree and change the move speed by changing the value on the external behavior tree task. The RTS sample project has an example of using the inherited field attribute.

```
[InheritedField]
public float moveSpeed;
```

Third Party Integrations

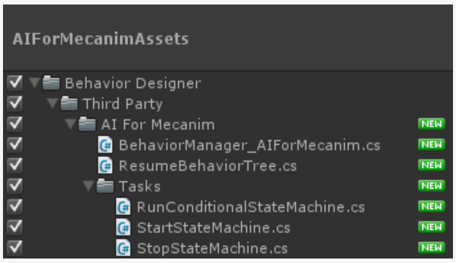
Behavior Designer includes many tasks which integrate with third party assets. For most of those integrations, no extra steps are required and they can be added to a behavior tree and then have their values assigned. However, the following integrations take a small amount of more work in order to fully work:

- AI For Mecanim
- Dialogue System
- Motion Controller
- PlayMaker
- UFPS
- uScript

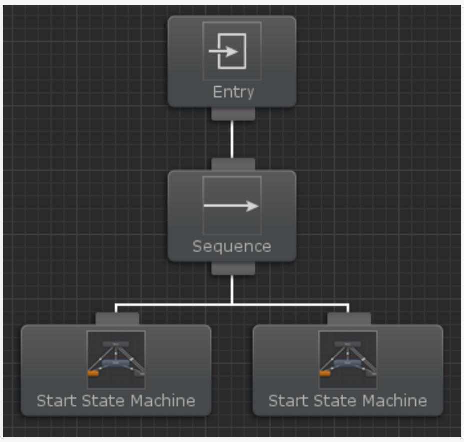
AI For Mecanim

AI For Mecanim allows you to create state machines with an interface similar to the mecanim animator interface. Behavior Designer is integrated with AI For Mecanim by allowing you to start and stop these state machines from within a behavior tree, as well as run a state machine as a conditional task. AI For Mecanim also includes a set of actions that allow you to start and stop a behavior tree from within the state machine. All of the AI For Mecanim integration files located on the integrations page.

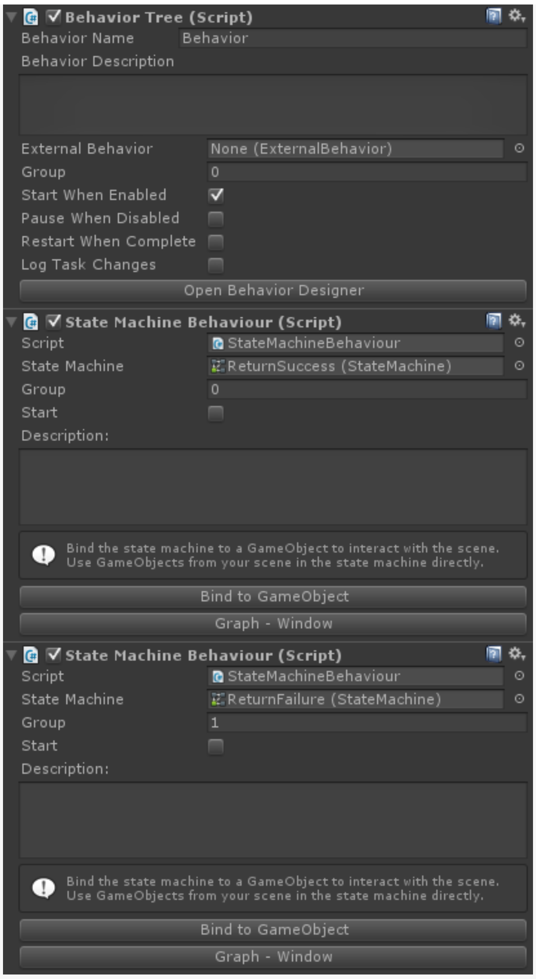
To get started, first make sure you have AI For Mecanim installed. Next, import AIForMecanimAssets.unitypackage:



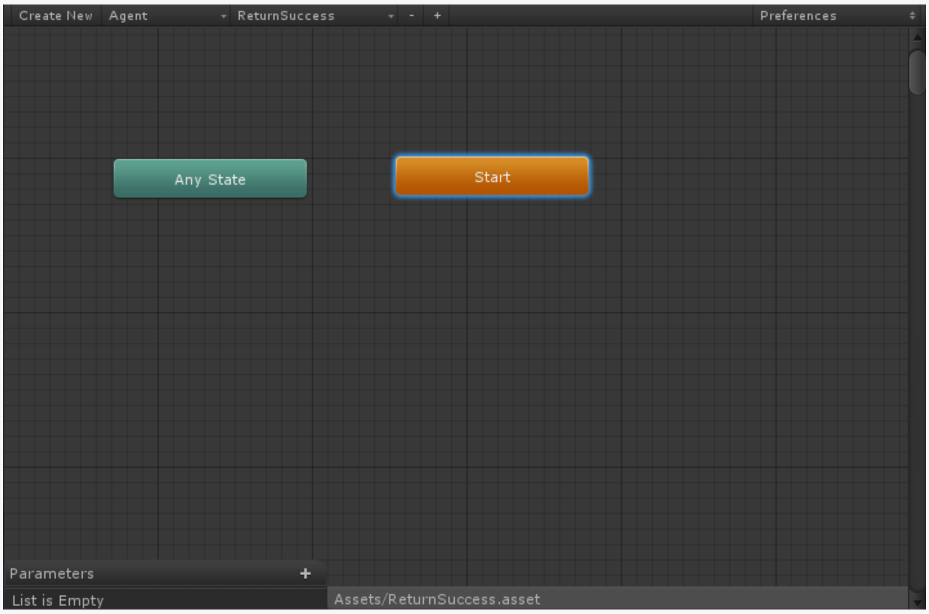
Once those files are imported you are ready to start creating behavior trees with AI For Mecanim! To get started, create a very basic tree with a sequence task who has two Start State Machine child tasks:



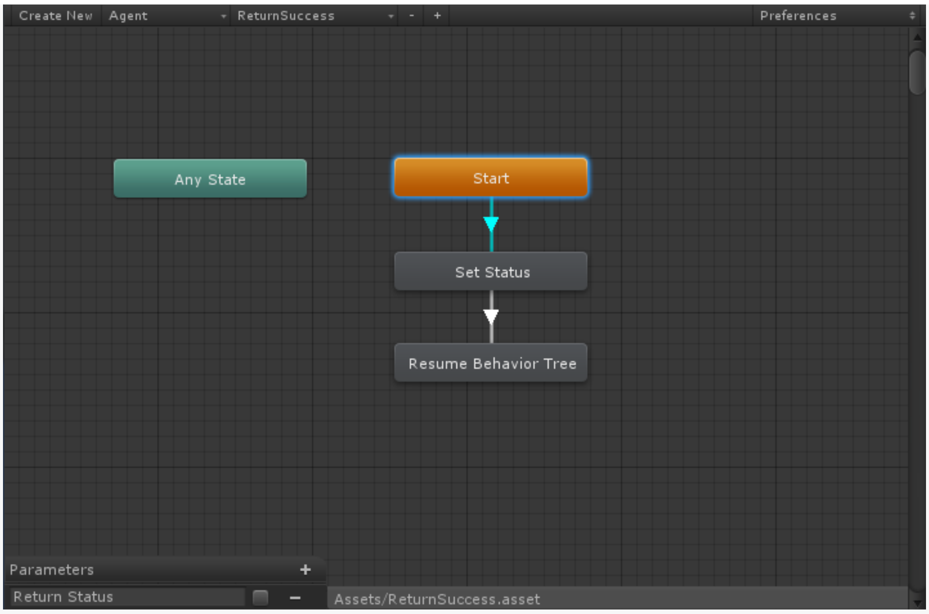
Next add two State Machine Behaviour components to the same GameObject that you added the behavior tree to. Since there are multiple State Machine Behaviours on the same GameObject ensure you have set the group number. In addition, assign the State Machine field to a new StateMachine and prevent the State Machine from starting when enabled.



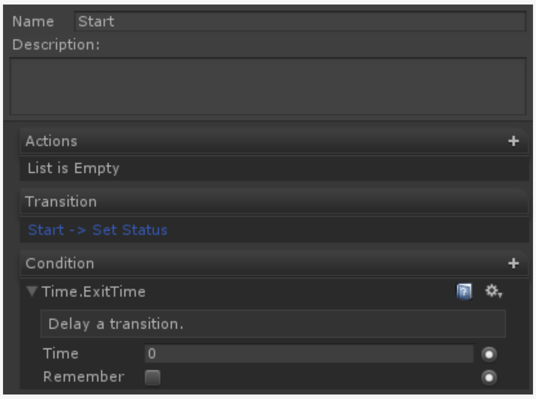
Open the AI For Mecanim editor and create a new state machine using one of the StateMachine objects that was just assigned to the State Machine Behaviour. Behavior Designer starts the state machine from the "Default" state so create a new state and ensure it is orange indicating that it is the default state.



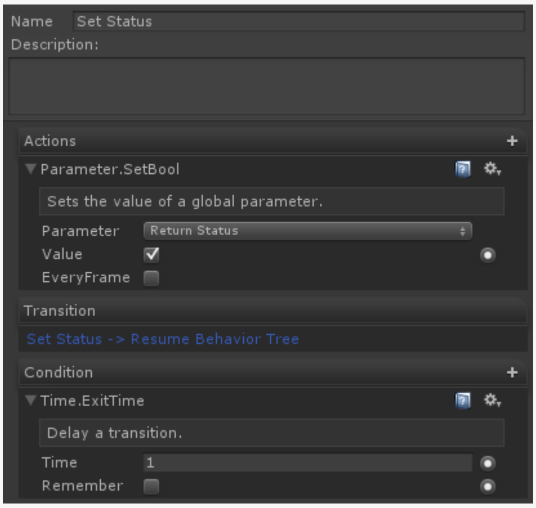
Create a new bool parameter (named Return Status) and two more states (named Set Status and Resume Behavior Tree). Add transitions from Start to Set Status and Set Status to Resume Behavior Tree. This state machine will simply set a bool to indicate the return status, wait a second, and finally resume the behavior tree.



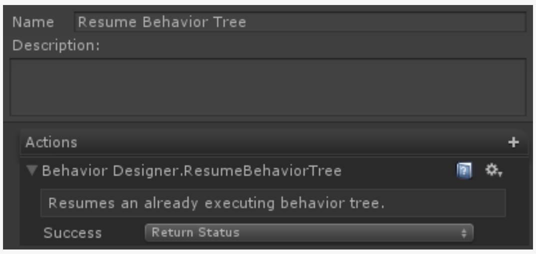
Select the Start state and view the State Inspector. For this state we only need to add a conditional which exits the state immediately.



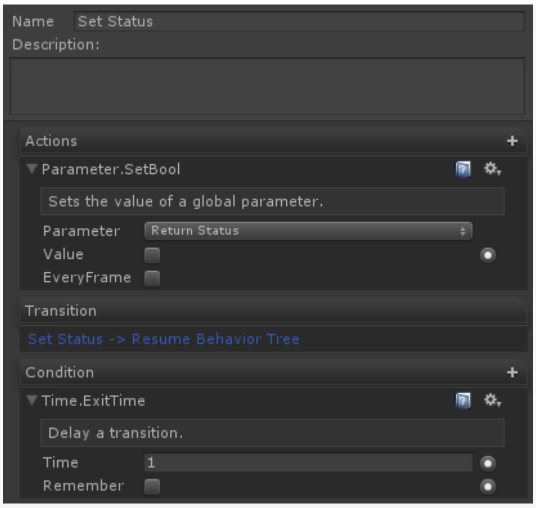
Select the Set Status state and view the State Inspector. Add the Parameter -> Set Bool action. This action will set the Return Status parameter to true. In addition, add a condition that exits the state after 1 second.



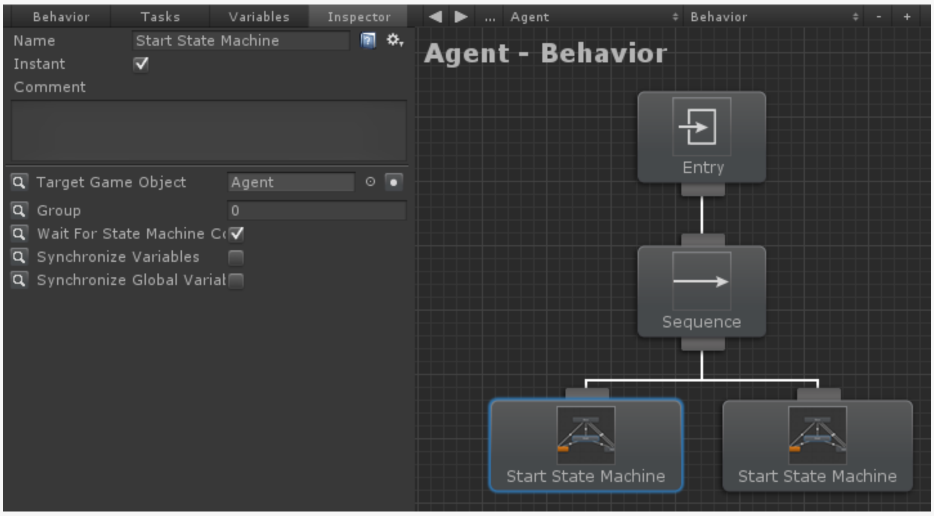
Select the Resume Behavior Tree state and view the State Inspector. Add the Behavior Designer -> Resume Behavior Tree action. Ensure you have set the Success variable to the Return Status parameter.



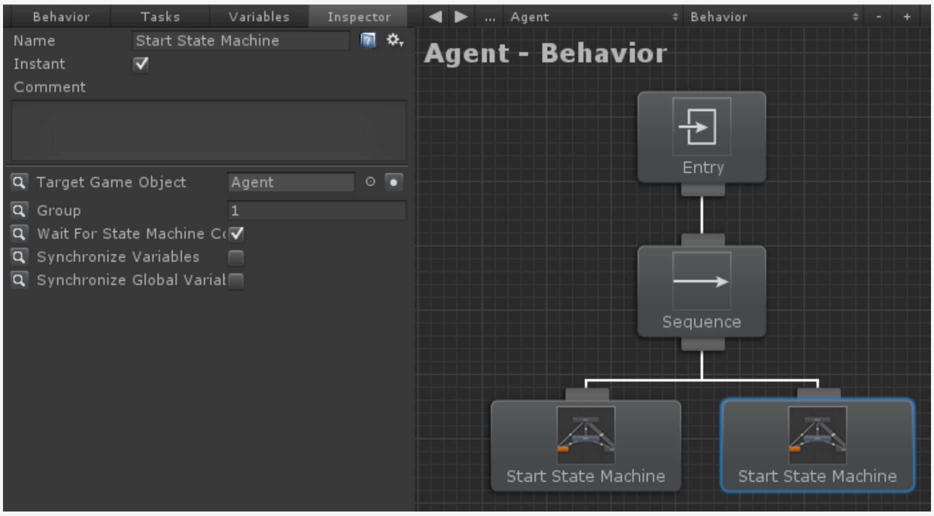
We are done setting up this state machine. Perform the same steps for the second state machine that we created earlier, only this time set the Return Status parameter to false within the Set Bool action.



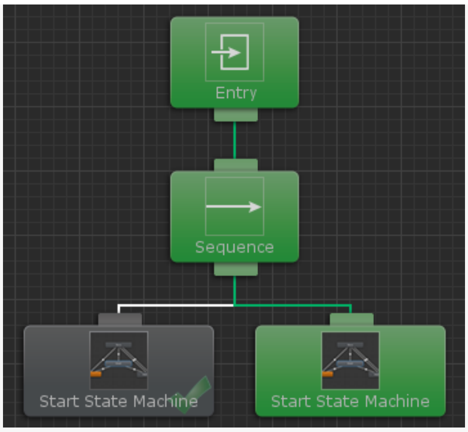
Open your behavior tree in Behavior Designer again and assign the Target Game Object and Group to point to the first state machine.



Set the fields of the second state machine task as well, making sure the group is set to 1.



We are now ready to run the behavior tree with AI For Mecanim integration! The first state machine's task will return success after 1 second because we set the Return Status parameter to true within the Set Status action. Similarly, the second state machine task will run for 1 second only this time it was return failure because the Return Status was set to false.

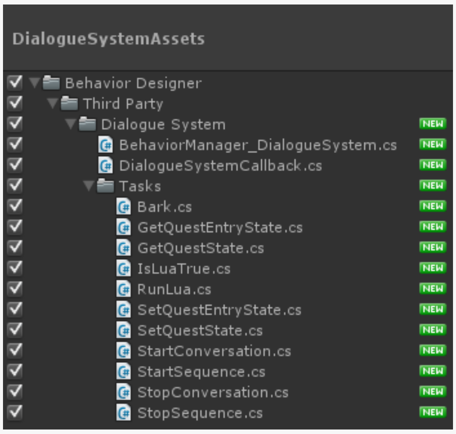


The behavior tree will never get to the second state machine if you were to swap the state machine tasks because the first state machine task returns failure.

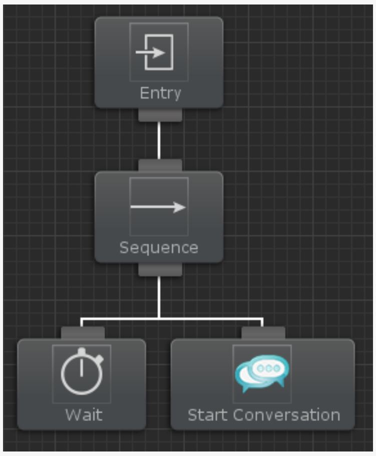
Dialogue System

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](#).

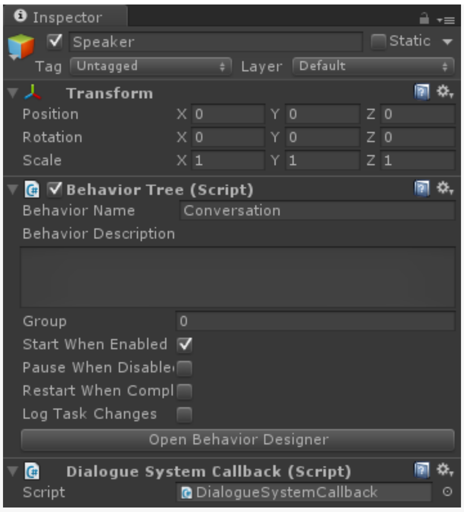
To get started, first make sure you have Dialogue System for Unity installed. Next, import DialogueSystemAssets.unitypackage:



Once those files are imported you are ready to start creating behavior trees with the Dialogue System! To get started, create a very basic tree with a sequence task which has a Wait task and a Start Conversation task:

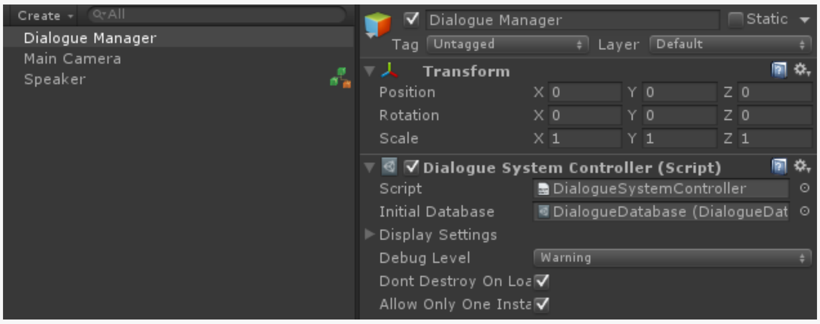


When the Dialogue System finishes with a conversation or sequence it will callback to Behavior Designer to let Behavior Designer know that it is done. In order for this to occur the Dialogue System Callback component must be added to the same GameObject that your behavior tree is on:

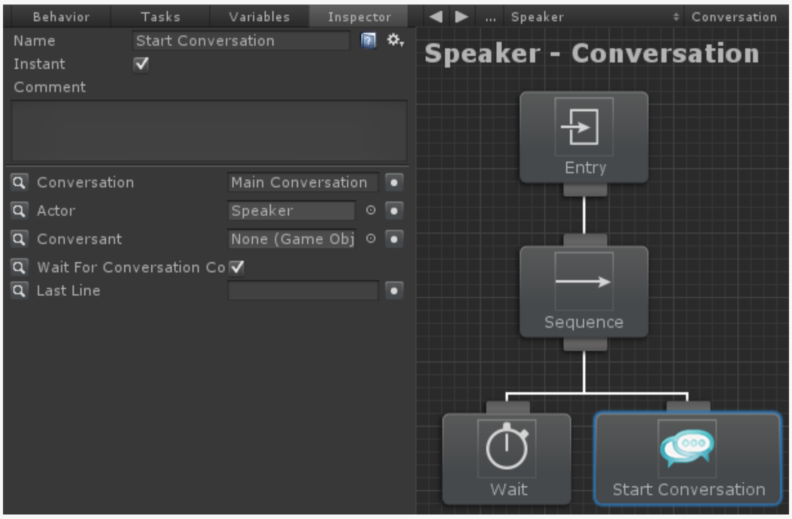


Now we are ready to start creating the actual conversion. Create a new Dialogue System Database and create a basic conversation:

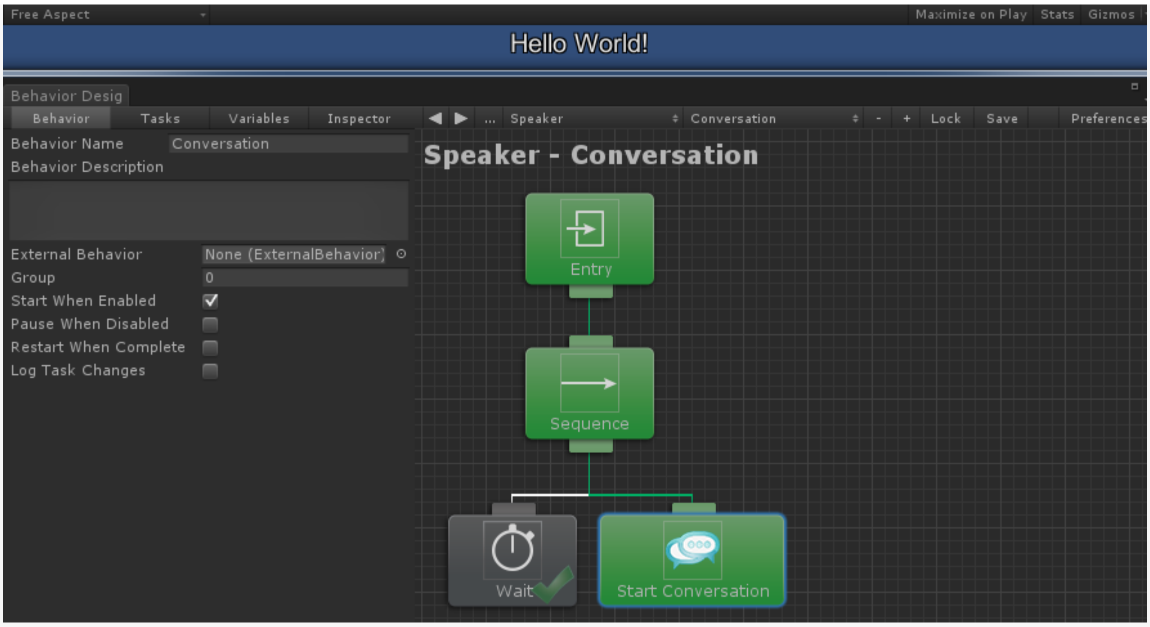
Make note of the conversation name because that will be needed later. Assign that database to the Dialogue System Controller:



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:



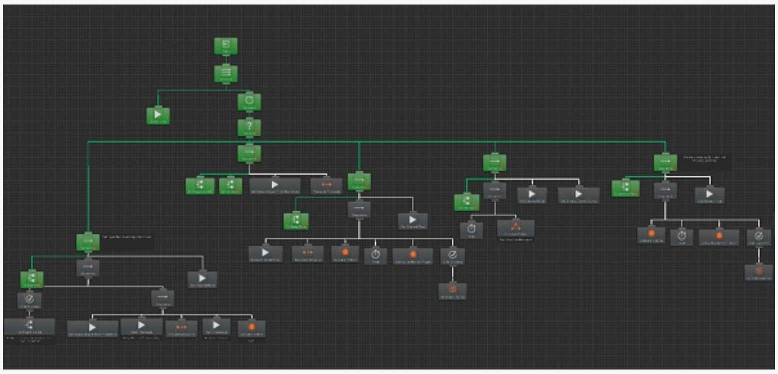
This topic hardly scratches the surface for what is possible with Behavior Designer / Dialogue System integration. For a more complex example, take a look at the Dialogue System [sample project](#).

Motion Controller

[Motion Controller](#) allows you to add any type of motion to your character. With Behavior Designer integration, your agent will come alive by walking, running, jumping, and climbing as if they were controlled by another player.

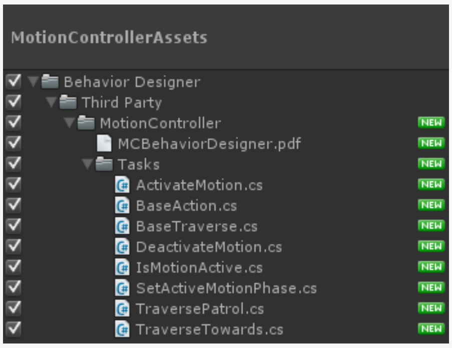
We were contacted by Tim from ootii on integrating Motion Controller with Behavior Designer. Unlike other integrations, Tim wanted more than just task/script integration: he wanted to create a complete tree that brings a character to life to really showcase the integration between the two assets. After a couple of months of work, we have that tree completed.

In the [Goblin Life](#) sample scene, you control a goblin. This goblin can move with the included Motion Controller actions such as walking, jumping, and climbing. This part isn't new. What is new is that your goblin character has many goblins surrounding him. All of these goblins are controlled by a behavior tree with tasks that are integrated with Motion Controller. This is a zoomed out view of that behavior tree:



Because some of the tasks use layers, we had to place the project in a zip file instead of the standard Unity package. Once you have downloaded this zip file *do not open* the GoblinLife scene yet. First import Motion Controller and Behavior Designer. Once those two packages are imported download the Motion Controller tasks on the [integrations page](#).

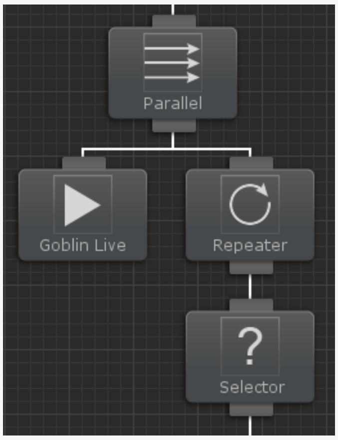
Import the Motion Controller Unity package. This package contains the Motion Controller tasks as well as a overview PDF which describes the integration.



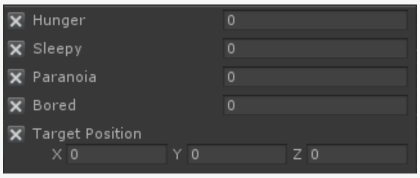
Once you have imported these three assets you can open the Goblin Life scene. If you accidentally opened the Goblin Life scene ahead of time it's no problem, just make sure you reload the scene before you hit play in Unity.

Once you play the scene you'll see that there are several activities that the goblins take part in. They can eat, sleep, patrol, and gamble when they become bored. This behavior tree makes use of conditional aborts, external behavior trees, and global variables.

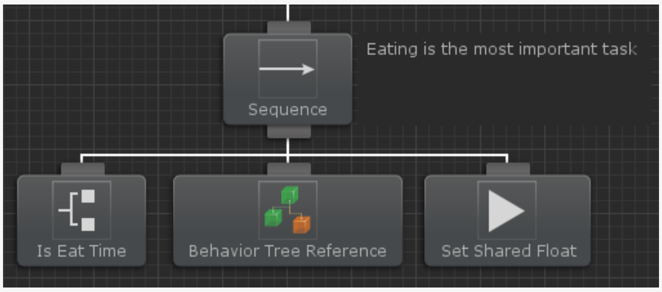
The root of the tree contains a parallel task which has two children: the Goblin Live task and a selector task which contains the various actions for the goblins.



The Goblin Live task updates Shared Variables in order to determine the current state of the goblin.

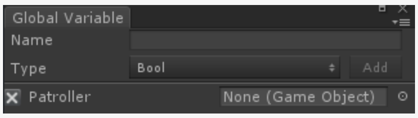


As an example, every tick the hunger variable will be updated by the Hunger Increase Rate. This causes the hunger variable to grow as time goes on, and the Is Eat Time task further down the tree will check to see if that hunger value is greater than a specified value.

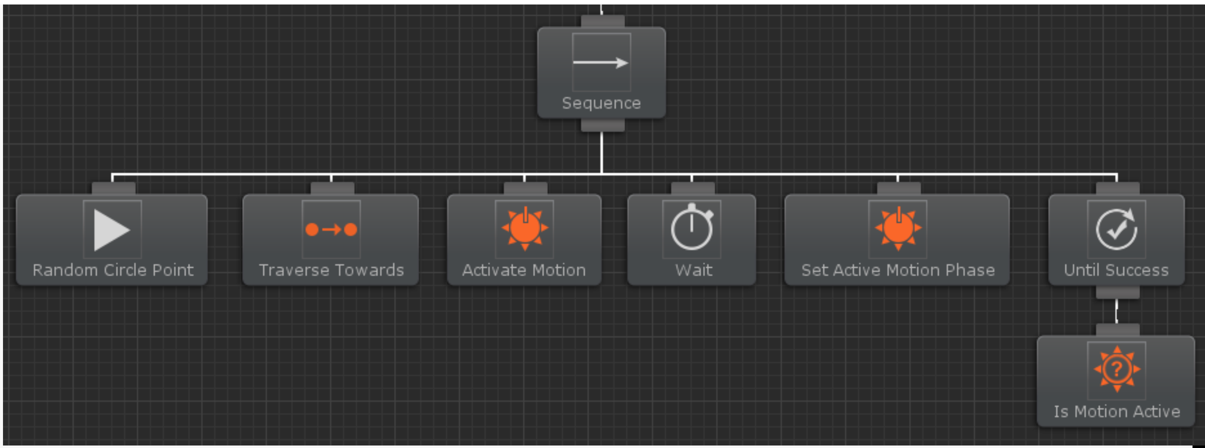


The sequence task has a lower priority conditional abort setup so when the hunger value is greater than the specified value it will abort whatever task is currently running and start the external tree within the behavior tree reference task. Once that external tree finishes executing the Set Shared Float task will reset the hunger value back to 0.

The Goblin Life tree contains four other branches similar to this one. They are arranged from highest priority to lowest priority: eat, sleep, patrol, move close to another goblin, and gamble. Each parent composite task of these branches have a conditional abort set the Lower Priority so that branch will always take priority over a lower branch. The only branch that is unique is the Patrol branch. The Patrol branch uses a global variable in order to determine if the goblin should go on patrol. If another goblin is on patrol then the current goblin should not start patrolling, and the global variable helps with this decision.



Within each branch is a set of Motion Controller actions that do the actual movement. For example, here is the Sleep branch:



These tasks are processed in the following order:

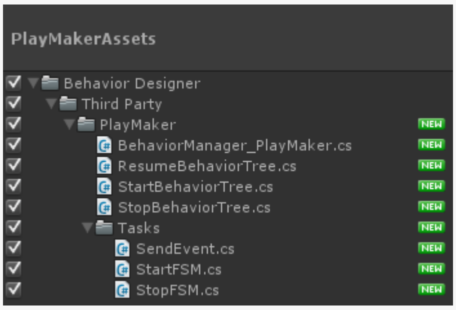
1. Random Circle Point finds a random location based on a center point and a radius.
2. Traverse Towards is a Motion Controller action that has the goblin walk towards the random location found earlier. If there's an obstacle in his way, he'll climb over it.
3. Activate Motion is a Motion Controller action that plays a specific motion. In this case, 'Lay Down and sleep'.
4. Wait for the goblin to finish sleeping.
5. Set Active Motion Phase is a Motion Controller action that progresses a motion forward. In this case, it's time to tell the goblin to wake up.
6. Waiting until the wake up portion of the motion finishes.
7. Is Motion Active is a condition we're check to see if we've finished waking up. Once we're done, #6 finishes successfully and the whole sequence is complete.

The rest of the branches are setup similarly. For a full listing of all of the Motion Controller tasks take a look at this topic.

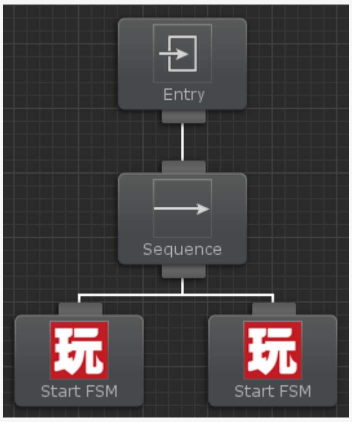
PlayMaker

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.

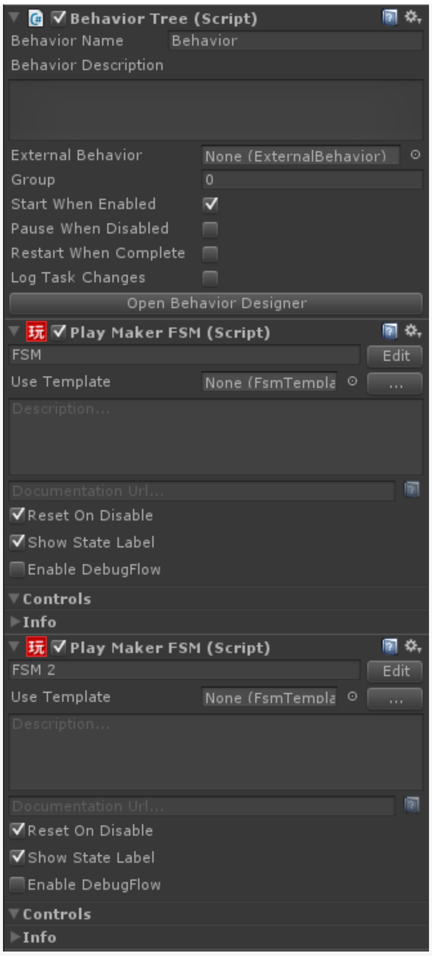
To get started, first make sure you have PlayMaker installed. Next, import PlayMakerAssets.unitypackage:



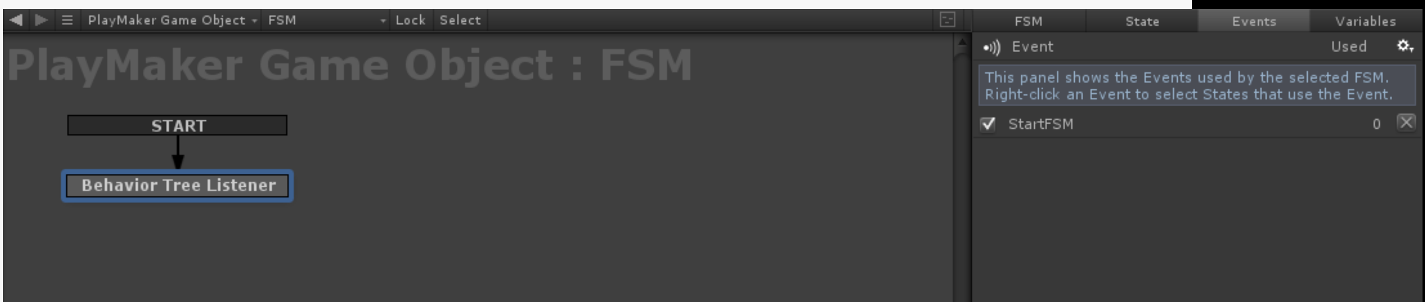
Once those files are imported you are ready to start creating behavior trees with PlayMaker! To get started, create a very basic tree with a sequence task who has two Start FSM child tasks:



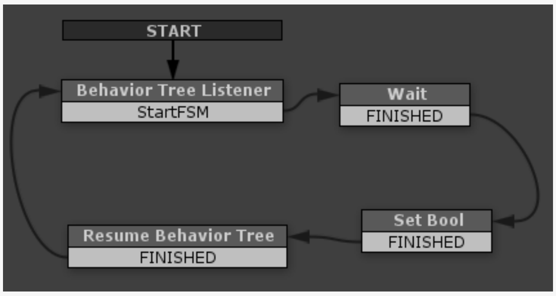
Next add two PlayMaker FSM components to the same game object that you added the behavior tree to.



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 [FPS 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.

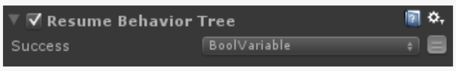


Add a transition from that event along with a wait state, a set bool state, and a resume behavior tree state. Make sure you transition from the Resume Behavior Tree state to the Behavior Tree Listener state so the FSM can be started again from Behavior Designer.

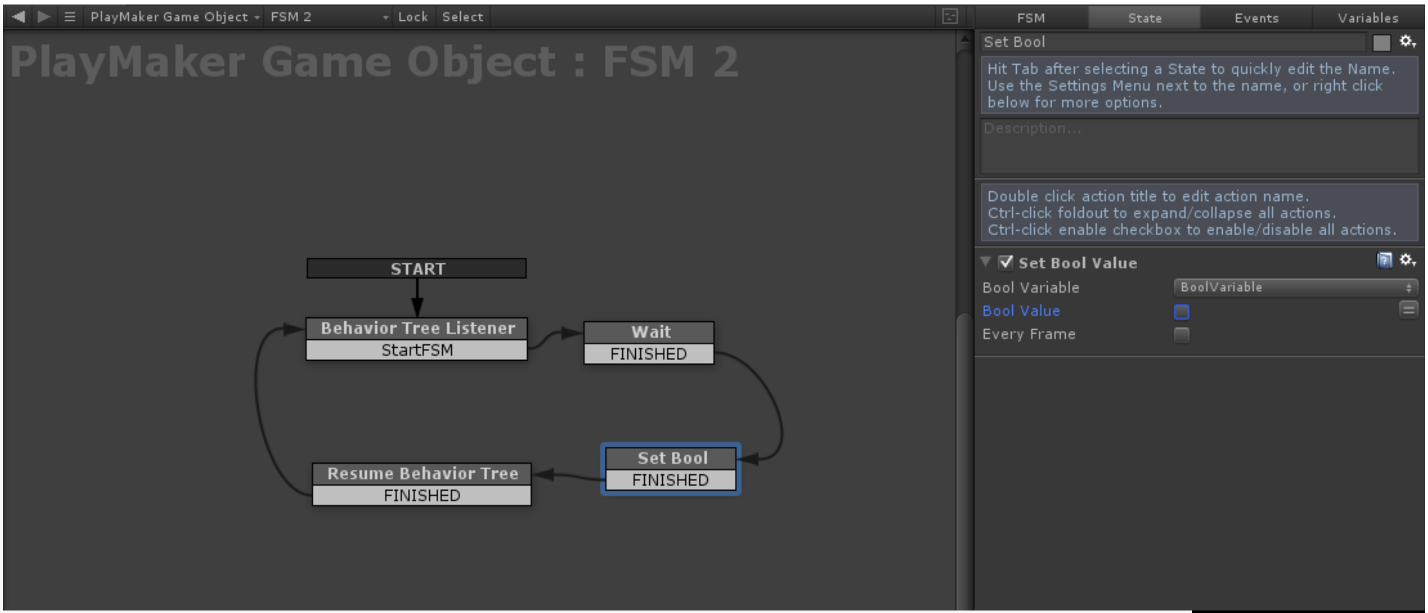


Create a new variable within the Set Bool state and set that value to true.

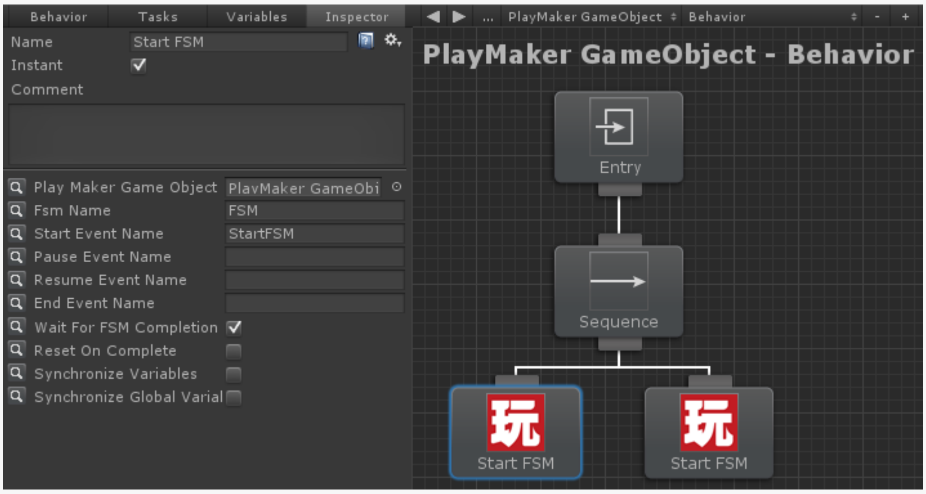
Then within the Resume Behavior Tree state we want to return success based off of that bool value:



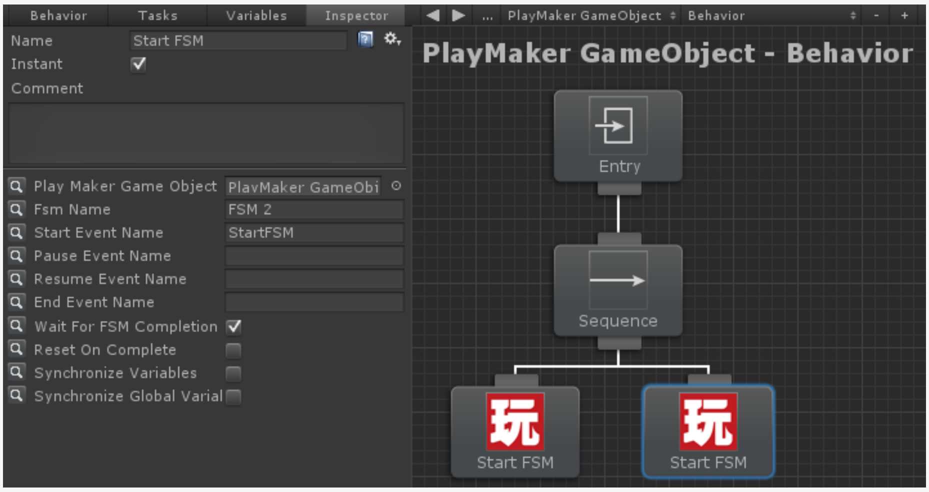
That's it for this FSM. Create the same states and variables for the second FSM that we created earlier. Do not set the bool variable to true for this FSM.



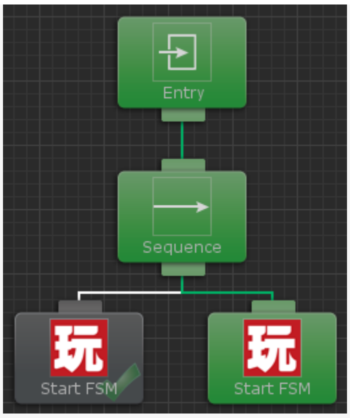
We are now done working in PlayMaker. Open your behavior tree back up within Behavior Designer. Select the left PlayMaker task and start assigning the values to the variables. PlayMaker Game Object is assigned to the game object that we added the PlayMaker FSM components to. FSM Name is the name of the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

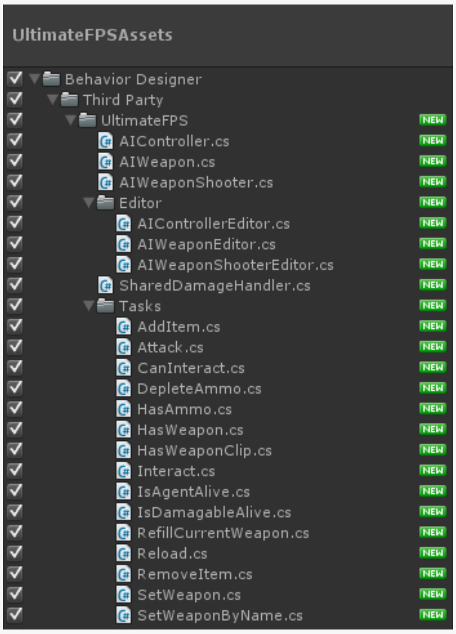


If you were to swap the tasks so the second PlayMaker task runs before the first PlayMaker FSM, the behavior tree will never get to the first PlayMaker FSM because the second PlayMaker FSM returned failure and the sequence task stopped executing its children.

UFPS

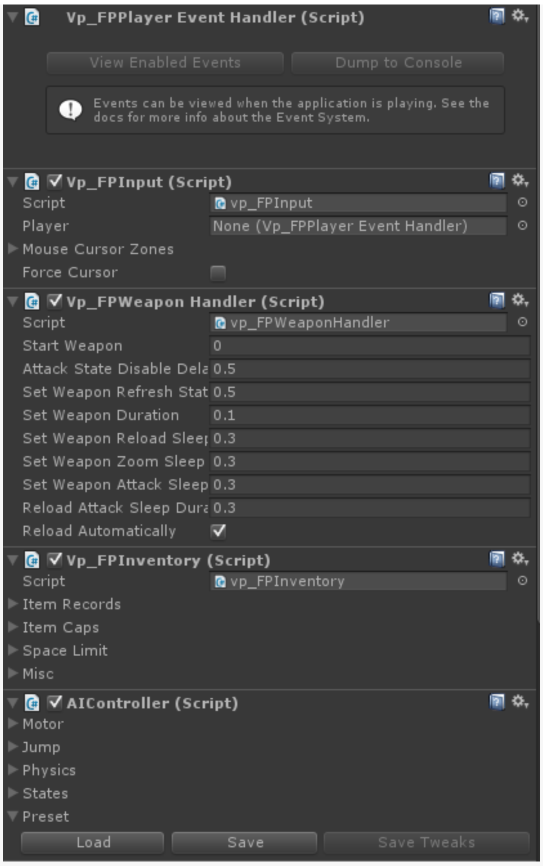
[Ultimate FPS](#) is a FPS asset which allows you to get a first person shooter up and running quickly. It has many features which manages the camera, weapons, inventory, and a lot more. Behavior Designer includes tasks which allow you to add the UFPS controls on an AI agent. Because UFPS is not specifically designed to be placed on an AI agent there is some extra setup required. UFPS integration files are located on the [integrations page](#) because Behavior Designer doesn't require UFPS to work.

To get started, first make sure you have UFPS installed. Next, import UltimateFPSAssets.unitypackage:



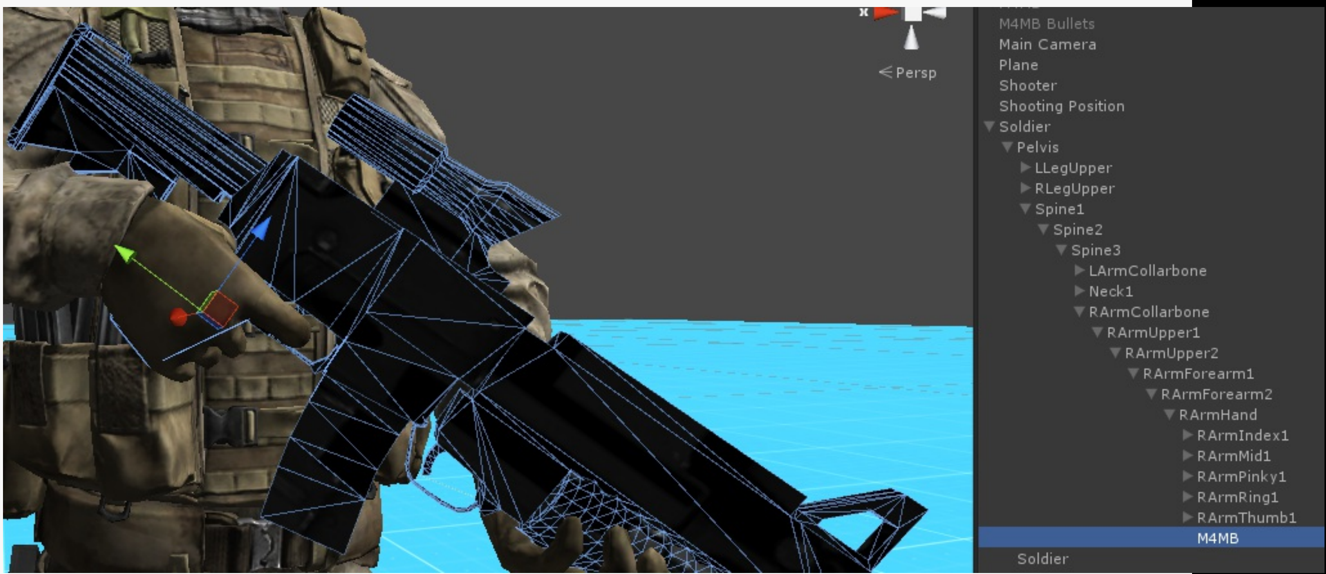
In this example our AI agent will be the soldier found in the Unity Bootcamp sample project. Add the following components to your agent:

vp_FPPlayerEventHandler
vp_FPInput
vp_FPWeaponHandler
vp_FPInventory
AIController

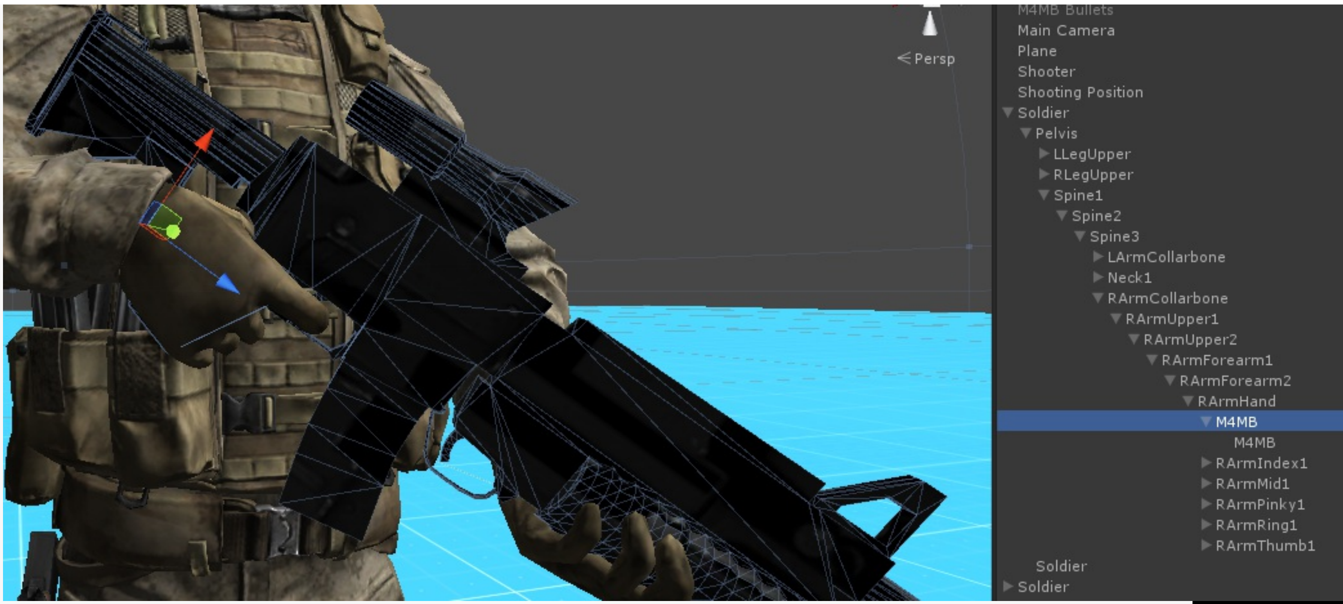


The first four components (those starting with "vp_") are standard UFPS components and should be familiar to you. The last component, AIController, is a small class derived from vp_FPController which disables the vp_FPInput component. vp_FPInput is used for first person player input. Since the agent needs to be controlled by the behavior tree we do not need this component. Ideally we wouldn't even add this component at all but it a required component by vp_FPController. We chose this design because it allows us to automatically get the UFPS updates without having to change anything.

It is now time to add a weapon. Add the weapon to the solder's hand GameObject within the hierarchy window. In our case the M4 assault rifle has already been added to the solder. Take a look at the position handle within the scene.

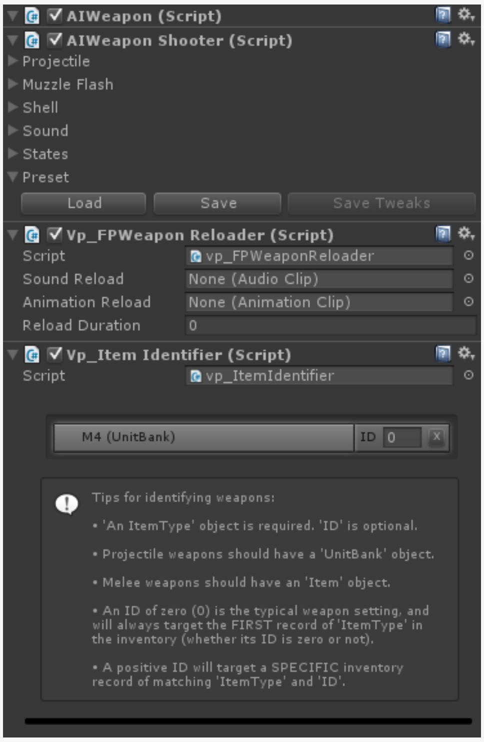


If the blue arrow (the forward vector) is facing in the same direction as the weapon then you do not have to perform the next step. When the UFPS tasks goes to aim the weapon they need to know which direction is forward. If the weapon's forward direction is not it's 'actual' forward direction then we need to add a parent GameObject which corrects this:



A parent GameObject (also called M4MB) has been added and now you can see that the blue arrow is facing in the same direction as the weapon. Once this is complete we can start adding the weapon components. The following components need to be added to the weapon's parent GameObject (or the original GameObject if no parent is needed):

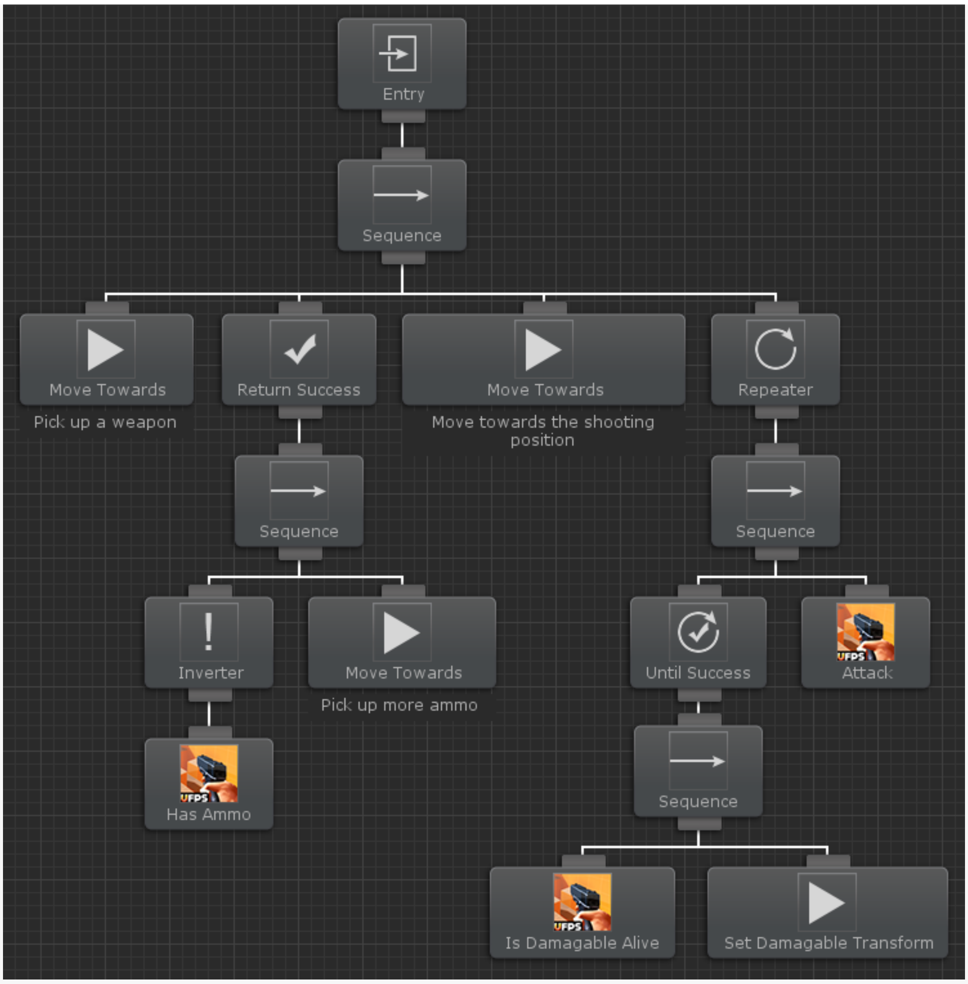
AIWeapon
AIWeaponShooter
vp_FPWeaponReloader
vp_ItemIdentifier



The last two components are standard UFPS and should already be familiar to you. The AIWeapon component is derived from vp_FPWeapon and it is basically there to prevent the vp_FPWeapon component from updating the position/rotation of the weapon. Since the AI is not in first person view we do not want UFPS managing the position of the weapon. This should be done with animations instead. AIWeaponShooter is derived from vp_Shooter and it is the script that actually shoots the weapon.

This is the only extra setup required. The rest of the steps (such as setting up the inventory) are similar to a standard UFPS setup which you can refer to from the [UFPS manual](#).

In the [UFPS sample](#) scene we created the following behavior tree:



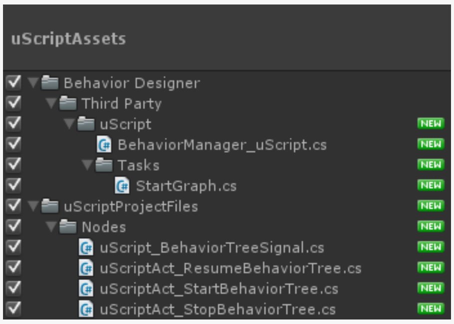
This behavior tree will have the soldier shoot at a target, reload, and pickup more bullets when necessary.

---

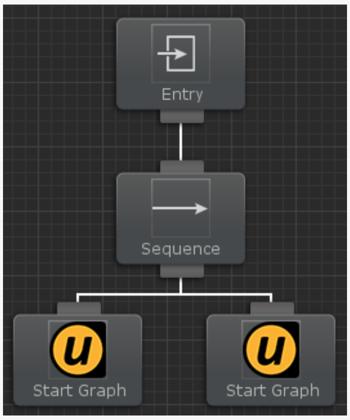
uScript

[uScript](#) is a popular 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.

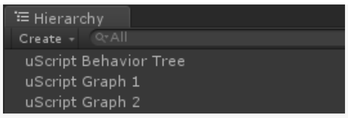
To get started, first make sure you have uScript installed. Next, import uScriptAssets.unitypackage:



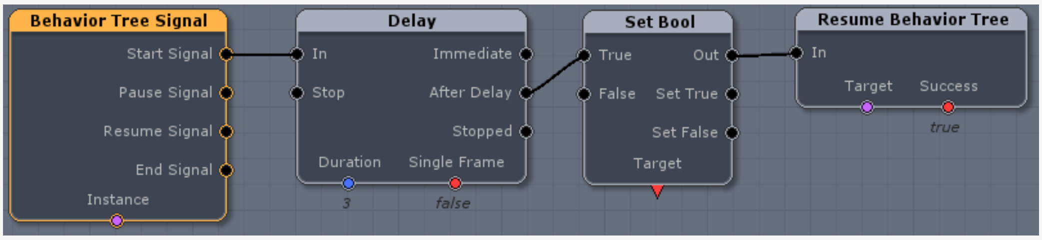
Once those files are imported you are ready to start creating behavior trees with uScript! To get started, create a very basic tree with a sequence task who has two Start Graph child tasks:



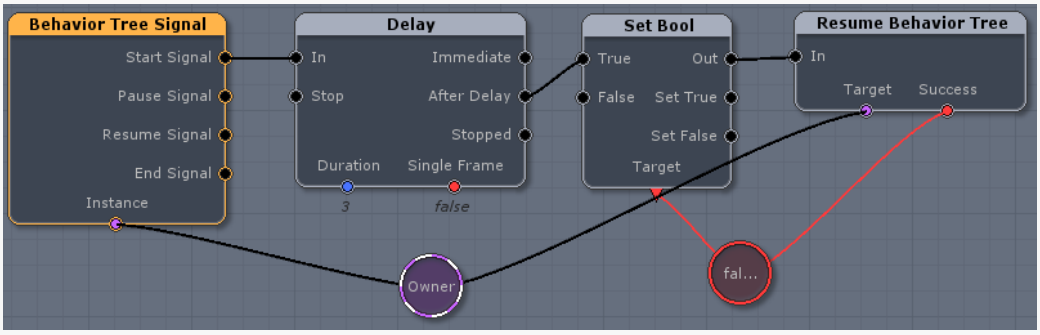
Now we need to create two GameObjects which will hold the compiled uScript graph:



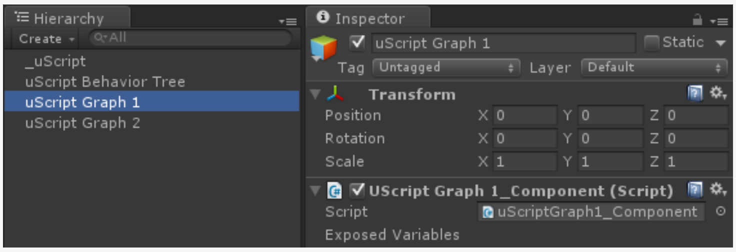
Open uScript and start creating a new graph. Add the Behavior Tree Signal node, located under Events/Signals. When Behavior Designer wants to start executing a uScript graph it will start from this node. This node contains four events – Start Signal, Pause Signal, Resume Signal, and End Signal. Start 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:



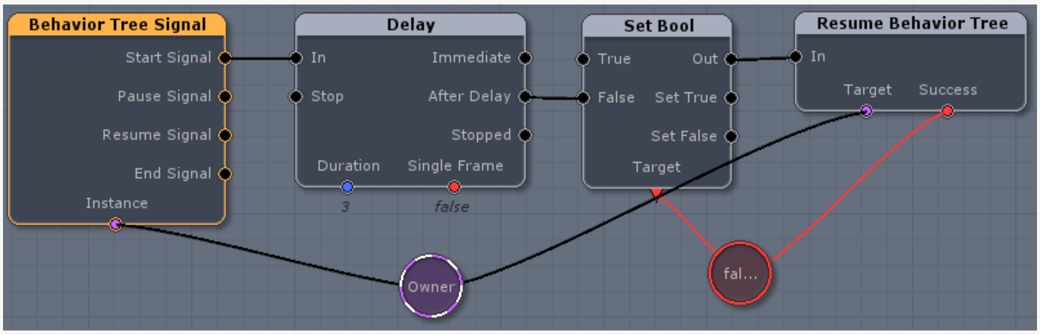
Now we need to create a Owner GameObject and bool variable.



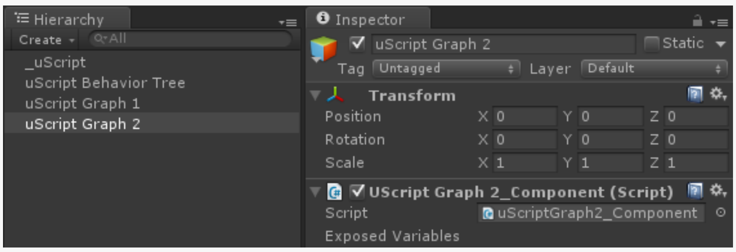
Save the uScript graph and assign the component to your first uScript graph GameObject. Answer no if uScript asks if you want to assign the component to the master GameObject.



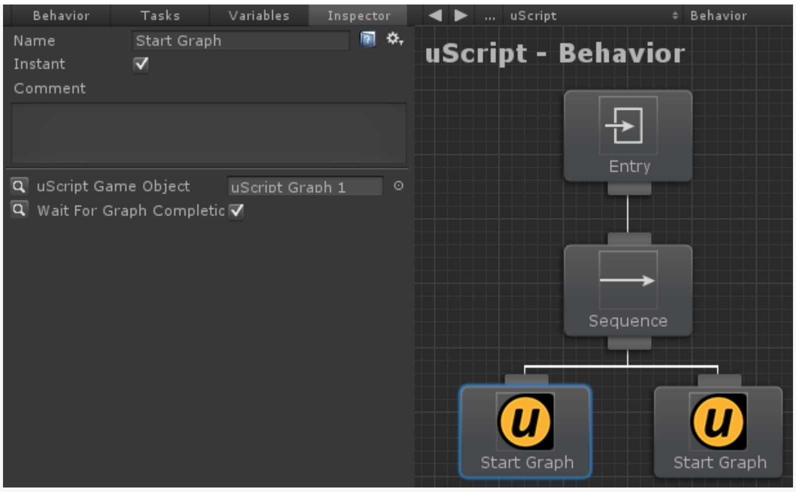
Create one more uScript graph. Make it the same as the last graph except set the bool to false:



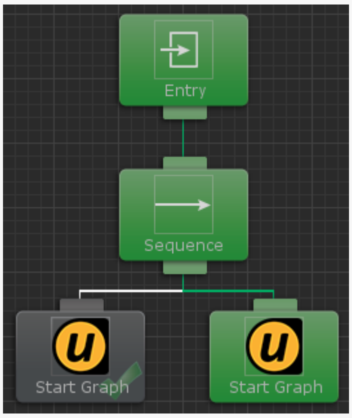
Finally save that graph and assign the component to the second uScript GameObject:



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.

Task List

A collection of tasks form a behavior tree. Behavior Designer includes the tasks listed below with its default installation. For more tasks take a look at the sample projects or the Movement Pack.

- Actions
  - Behavior Tree Reference
  - Log
  - Perform Interruption
  - Restart Behavior Tree
  - Start Behavior
  - Stop Behavior
  - Wait
  - Invoke Method
  - Get Field Value
  - Get Property Value
  - Set Field Value
  - Set Property Value
- Composites
  - Sequence
  - Selector
  - Parallel
  - Parallel Selector
  - Priority Selector
  - Random Selector
  - Random Sequence
- Conditionals
  - Random Probability
  - Compare Field Value
  - Compare Property Value
- Decorators
  - Conditional Evaluator
  - Interrupt
  - Inverter
  - Repeater
  - Return Failure
  - Return Success
  - Task Guard
  - Until Failure
  - Until Success
- Basic Tasks
  - Animation
  - Animator
  - AudioSource
  - Behaviour
  - BoxCollider
  - BoxCollider2D
  - CapsuleCollider
  - CharacterController
  - CircleCollider2D

Actions

Action tasks alter the state of the game. For example, an action task might consist of playing an animation or shooting a weapon.

Behavior Designer includes the following actions with its default installation. For more action examples take a look at sample projects.

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.

Log

Log is a simple task which will output the specified text and return success. It can be used for debugging.

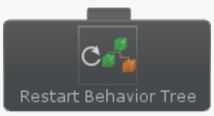
Name	Description
text	Text to output to the log.
logError	Is this text an error?

Perform Interruption

Perform the actual interruption. This will immediately stop the specified tasks from running and will return success or failure depending on the value of interrupt success.

Name	Description
interruptTasks	The list of tasks to interrupt. Can be any number of tasks.
interruptSuccess	When we interrupt the task should we return a task status of success?

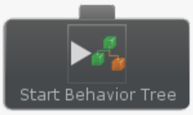
Restart Behavior Tree



Restarts a behavior tree, returns success after it has been restarted.

Name	Description
behavior	The behavior tree that we want to start. If null use the current behavior

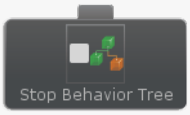
Start Behavior



Start a new behavior tree and return success after it has been started.

Name	Description
behavior	The behavior that we want to start. If null use the current behavior.

Stop Behavior



Pause or disable a behavior tree and return success after it has been stopped.

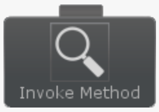
Name	Description
behavior	The behavior that we want to stop. If null use the current behavior.
pauseBehavior	Should the behavior be paused or completely disabled.

Wait

Wait a specified amount of time. The task will return running until the task is done waiting. It will return success after the wait time has elapsed.

Name	Description
waitTime	The amount of time to wait.

Invoke Method



Invokes the specified method with the specified parameters. Can optionally store the return value. Returns success if the method was invoked.

Name	Description
targetGameObject	The GameObject to invoke the method on
componentName	The component to invoke the method on
methodName	The name of the method
parameter1	The first parameter of the method
parameter2	The second parameter of the method
parameter3	The third parameter of the method
parameter4	The fourth parameter of the method
storeResult	Store the result of the invoke call

Get Field Value

Gets the value from the field specified. Returns success if the field was retrieved.

Name	Description
targetGameObject	The GameObject to get the field on
componentName	The component to get the field on
fieldName	The name of the field
fieldValue	The value of the field

Get Property Value

Gets the value from the property specified. Returns success if the property was retrieved.

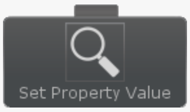
Name	Description
targetGameObject	The GameObject to get the property of
componentName	The component to get the property of
propertyName	The name of the property
propertyValue	The value of the property

Set Field Value

Sets the field to the value specified. Returns success if the field was set.

Name	Description
targetGameObject	The GameObject to setthe field on
componentName	The component to set the field on
fieldName	The name of the field
fieldValue	The value to set

Set Property Value



Sets the property to the value specified. Returns success if the property was set.

Name	Description
targetGameObject	The GameObject to setthe property of
componentName	The component to set the property of
propertyName	The name of the property
propertyValue	The value to set

Composites

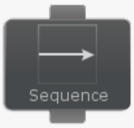
Composite tasks are parent tasks that hold a list of child tasks. For example, one composite task may loop through the child tasks sequentially while another task may run all of its child tasks at once. The return status of the composite tasks depends on its children.

Behavior Designer includes the following composites with its default installation. For more composite examples take a look at sample projects.

Every composite task holds the property which specifies if conditional aborts should be used.

- Sequence
- Selector
- Parallel
- Parallel Selector
- Priority Selector
- Random Selector
- Random Sequence

Sequence

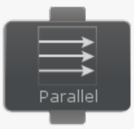


The sequence task is similar to an "and" operation. It will return failure as soon as one of its child tasks return failure. If a child task returns success then it will sequentially run the next task. If all child tasks return success then it will return success.

Selector

The selector task is similar to an "or" operation. It will return success as soon as one of its child tasks return success. If a child task returns failure then it will sequentially run the next task. If no child task returns success then it will return failure.

Parallel



Similar to the sequence task, the parallel task will run each child task until a child task returns failure. The difference is that the parallel task will run all of its children tasks simultaneously versus running each task one at a time. Like the sequence class, the parallel task will return success once all of its children tasks have return success. If one tasks returns failure the parallel task will end all of the child tasks and return failure.

Parallel Selector

Similar to the selector task, the parallel selector task will return success as soon as a child task returns success. The difference is that the parallel task will run all of its children tasks simultaneously versus running each task one at a time. If one tasks returns success the parallel selector task will end all of the child tasks and return success. If every child task returns failure then the parallel selector task will return failure.

Priority Selector

Similar to the selector task, the priority selector task will return success as soon as a child task returns success. Instead of running the tasks sequentially from left to right within the tree, the priority selector will ask the task what its priority is to determine the order. The higher priority tasks have a higher chance at being run first.

Random Selector

Similar to the selector task, the random selector task will return success as soon as a child task returns success. The difference is that the random selector class will run its children in a random order. The selector task is deterministic in that it will always run the tasks from left to right within the tree. The random selector task shuffles the child tasks up and then begins execution in a random order. Other than that the random selector class is the same as the selector class. It will continue running tasks until a task completes successfully. If no child tasks return success then it will return failure.

Name	Description
seed	Seed the random number generator to make things easier to debug.
useSeed	Do we want to use the seed?

Random Sequence

Similar to the sequence task, the random sequence task will return success as soon as every child task returns success. The difference is that the random sequence class will run its children in a random order. The sequence task is deterministic in that it will always run the tasks from left to right within the tree. The random sequence task shuffles the child tasks up and then begins execution in a random order. Other than that the random sequence class is the same as the sequence class. It will stop running tasks as soon as a single task ends in failure. On a task failure it will stop executing all of the child tasks and return failure. If no child returns failure then it will return success.

Name	Description
seed	Seed the random number generator to make things easier to debug.
useSeed	Do we want to use the seed?

Conditionals

Conditional tasks test some property of the game. For example, a condition might be to check if an object is within sight or determine if the player is still alive.

Behavior Designer includes the following conditionals with its default installation. For more conditional examples take a look at sample projects.

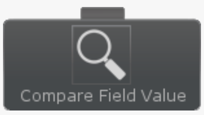
- Random Probability
- Compare Field Value
- Compare Property Value

Random Probability

The random probability task will return success when the random probability is above the succeed probability. It will otherwise return failure.

Name	Description
successProbability	The chance that the task will return success.
seed	Seed the random number generator to make things easier to debug.
useSeed	Do we want to use the seed?

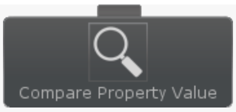
Compare Field Value



Compares the field value to the value specified. Returns success if the values are the same.

Name	Description
targetGameObject	The GameObject to setthe field on
componentName	The component to set the field on
fieldName	The name of the field
compareValue	The value to compare to

Compare Property Value



Compares the property value to the value specified. Returns success if the values are the same.

Name	Description
targetGameObject	The GameObject to setthe property of
componentName	The component to set the property of
propertyName	The name of the property
compareValue	The value to compare to

Decorators

The decorator task is a wrapper task that can only have one child. The decorator task will modify the behavior of the child task in some way. For example, the decorator task may keep running the child task until it returns with a status of success or it may invert the return status of the child.

Behavior Designer includes the following decorators with its default installation. For more decorator examples take a look at sample projects.

- Conditional Evaluator
- Interrupt
- Inverter
- Repeater
- Return Failure
- Return Success
- Task Guard
- Until Failure
- Until Success

Conditional Evaluator

Evaluates the specified conditional task. If the conditional task returns success then the child task is run and the child status is returned. If the conditional task does not return success then the child task is not run and a failure status is immediately returned. The conditional task is only evaluated once at the start.

Name	Description
reevaluate	Should the conditional task be reevaluated every tick?
conditionalTask	The conditional task to evaluate

Interrupt

The interrupt task will stop all child tasks from running if it is interrupted. The interruption can be triggered by the perform interruption task. The interrupt task will keep running its child until this interruption is called. If no interruption happens and the child task completed its execution the interrupt task will return the value assigned by the child task.

Inverter

The inverter task will invert the return value of the child task after it has finished executing. If the child returns success, the inverter task will return failure. If the child returns failure, the inverter task will return success.

Repeater

The repeater task will repeat execution of its child task until the child task has been run a specified number of times. It has the option of continuing to execute the child task even if the child task returns a failure.

Name	Description
count	The number of times to repeat the execution of its child task.
repeatForever	Allows the repeater to repeat forever.
endOnFailure	Should the task return if the child task returns a failure.

Return Failure

The return failure task will always return failure except when the child task is running.

Return Success

The return success task will always return success except when the child task is running.

Task Guard

The task guard task is similar to a semaphore in multithreaded programming. The task guard task is there to ensure a limited resource is not being overused. For example, you may place a task guard above a task that plays an animation. Elsewhere within your behavior tree you may also have another task that plays a different animation but uses the same bones for that animation. Because of this you don't want that animation to play twice at the same time. Placing a task guard will let you specify how many times a particular task can be accessed at the same time. In the previous animation task example you would specify an access count of 1. With this setup the animation task can be only controlled by one task at a time. If the first task is playing the animation and a second task wants to control the animation as well, it will either have to wait or skip over the task completely.

Name	Description
maxTaskAccessCount	The number of times the child tasks can be accessed by parallel tasks at once. Marked as SynchronizeField to synchronize the value between any linked tasks.
linkedTaskGuards	The linked tasks that also guard a task. If the task guard is not linked against any other tasks it doesn't have much purpose. Marked as LinkedTask to ensure all tasks linked are linked to the same set of tasks.
waitUntilTaskAvailable	If true the task will wait until the child task is available. If false then any unavailable child tasks will be skipped over.

Until Failure

The until failure task will keep executing its child task until the child task returns failure.

Until Success

The until success task will keep executing its child task until the child task returns success.

Basic Tasks

Behavior Designer includes a large number of tasks to accomplish basic operations, such as getting the velocity of a Rigidbody or playing a Mecanim state. The following categories of tasks are included:

- Animation
- Animator
- AudioSource
- Behaviour
- BoxCollider
- BoxCollider2D
- CapsuleCollider
- CharacterController
- CircleCollider2D
- GameObject
- Math
- Renderer
- Rigidbody
- Rigidbody2D
- SharedVariable
- SphereCollider
- Transform

Animation

The following tasks are included in the Animation category:

Blend
CrossFade
CrossFadeQueued
GetAnimatePhysics
IsPlaying
Play
PlayQueued
Rewind
Sample
SetAnimatePhysics
SetWrapMode
Stop

Animator

The following tasks are included in the Animator category:

CrossFade
GetApplyRootMotion
GetBoolParameter
GetDeltaPosition
GetDeltaRotation

GetFloatParameter
GetGravityWeight
GetIntegerParameter
GetLayerWeight
GetSpeed
InterruptMatchTarget
IsInTransition
IsParameterControlledByCurve
MatchTarget
Play
SetApplyRootMotion
SetBoolParameter
SetFloatParameter
SetIntegerParameter
SetLayerWeight
SetLookAtPosition
SetLookAtWeight
SetSpeed
SetTrigger
StartPlayback
StartRecording
StopPlayback
StopRecording

AudioSource

The following tasks are included in the AudioSource category:

GetIgnoreListenerPause
GetIgnoreListenerVolume
GetLoop
GetMaxDistance
GetMinDistance
GetMute
GetPan
GetPanLevel
GetPitch
GetSpeed
GetPriority
GetSpread
GetTime
GetTimeSamples
GetVolume
IsPlaying
Pause
Play
PlayDelayed
PlayOneShot
PlayScheduled
SetIgnoreListenerPause
SetIgnoreListenerVolume
SetLoop
SetMaxDistance
SetMinDistance
SetMute
SetPan
SetPanLevel
SetPitch
SetPriority
SetRolloffMode
SetScheduledEndTime
SetScheduledStartTime
SetSpread
SetTime
SetVelocityUpdateMode
SetVolume
Stop

Behaviour

The following tasks are included in the Behaviour category:

GetIsEnabled
IsEnabled
SetIsEnabled

BoxCollider

The following tasks are included in the BoxCollider category:

GetCenter
GetSize
SetCenter
SetSize

BoxCollider2D

The following tasks are included in the BoxCollider2D category. These tasks first need to be extracted from the BasicTasks2D Unity Package.

GetCenter
GetSize
SetCenter
SetSize

CapsuleCollider

The following tasks are included in the CapsuleCollider category:

GetCenter
GetDirection
GetHeight
GetRadius
SetCenter
SetDirection
SetHeight
SetRadius

CharacterController

The following tasks are included in the CharacterController category:

GetCenter

GetHeight
GetRadius
GetSlopeLimit
GetStepOffset
GetVelocity
IsGrounded
Move
SetCenter
SetHeight
SetRadius
SetSlopeLimit
SetStepOffset
SimpleMove

CircleCollider2D

The following tasks are included in the CircleCollider2D category. These tasks first need to be extracted from the BasicTasks2D Unity Package.

GetCenter
GetRadius
SetCenter
SetRadius

GameObject

The following tasks are included in the GameObject category:

ActiveInHierarchy
ActiveSelf
CompareTag
Destroy
DestroyImmediate
Find
FindWithTag
GetComponent
GetTag
SendMessage
SetActive
SetTag

Math

The following tasks are included in the Math category:

BoolComparison
BoolOperator
FloatComparison
FloatOperator
IntComparison
IntOperator
RandomBool
RandomFloat
RandomInt
SetBool
SetFloat
SetInt

Renderer

The following task is included in the Renderer category:

IsVisible

Rigidbody

The following task is included in the Rigidbodycategory:

AddExplosionForce
AddForce
AddForceAtPosition
AddRelativeForce
AddRelativeTorque
AddTorque
GetAngularDrag
GetAngularVelocity
GetCenterOfMass
GetDrag
GetFreezeRotation
GetIsKinematic
GetMass
GetPosition
GetRotation
GetUseGravity
GetVelocity
IsKinematic
IsSleeping
MovePosition
MoveRotation
SetAngularDrag
SetAngularVelocity
SetCenterOfMass
SetConstraints
SetDrag
SetFreezeRotation
SetIsKinematic
SetMass
SetPosition
SetRotation
SetUseGravity
SetVelocity
Sleep
UseGravity
WakeUp

Rigidbody2D

The following tasks are included in the Rigidbody2D category. These tasks first need to be extracted from the BasicTasks2D Unity Package.

AddForce
AddForceAtPosition
AddTorque
GetAngularDrag

GetAngularVelocity
GetDrag
GetFixedAngle
GetGravityScale
GetIsKinematic
GetMass
GetVelocity
IsKinematic
IsSleeping
SetAngularDrag
SetAngularVelocity
SetDrag
SetFixedAngle
SetGravityScale
SetIsKinematic
SetMass
SetVelocity
Sleep
WakeUp

SharedVariable

The following tasks are included in the SharedVariable category:

CompareSharedBool
CompareSharedColor
CompareSharedFloat
CompareSharedGameObject
CompareSharedGameObjectList
CompareSharedInt
CompareSharedObject
CompareSharedObjectList
CompareSharedQuaternion
CompareSharedRect
CompareSharedString
CompareSharedTransform
CompareSharedTransformList
CompareSharedVector2
CompareSharedVector3
CompareSharedVector4
SetSharedBool
SetSharedColor
SetSharedFloat
SetSharedGameObject
SetSharedGameObjectList
SetSharedInt
SetSharedObject
SetSharedObjectList
SetSharedQuaternion
SetSharedRect
SetSharedString
SetSharedTransform
SetSharedTransformList
SetSharedVector2
SetSharedVector3
SetSharedVector4
SharedGameObjectToTransform
SharedTransformToGameObject

SphereCollider

The following tasks are included in the SphereCollider category:

GetCenter
GetRadius
SetCenter
SetRadius

Transform

The following tasks are included in the Transform category:

Find
FindChild
GetChild
GetChildCount
GetEulerAngles
GetLocalEulerAngles
GetLocalPosition
GetLocalRotation
GetLocalScale
GetParent
GetPosition
GetRotation
IsChildOf
LookAt
Rotate
RotateAround
SetEulerAngles
SetLocalEulerAngles
SetLocalRotation
SetLocalScale
SetParent
SetPosition
SetRotation
Translate

Third Party

Behavior Designer contains tasks for the following third party assets:

- 2D Toolkit
- AI For Mecanim
- Camera Path Animator
- Core GameKit
- Dialogue System
- Final IK
- Master Audio
- Motion Controller
- NGUI
- PlayMaker
- Pool Manager
- Simple Waypoint System

- [Ultimate FPS]
- [uScript]
- [uSequencer]

2D Toolkit

The following tasks are included in the 2D Toolkit integration:

Name	Description
Commit TextMesh	Commit a TextMesh. This is so you can change multiple parameters without reconstructing the mesh repeatedly, simply use that after you have set all the different properties.
Get Sprite Color	Get the color of a sprite.
Get Sprite ID	Get the sprite id of a sprite.
Get TextMesh Anchor	Get the scale of a sprite.
Get TextMesh Colors	Get the anchor of a TextMesh.
Get TextMesh Font	Get the colors of a TextMesh.
Get TextMesh Inline Styling	Get the inline styling flag of a TextMesh.
Get TextMesh Max Chars	Get the maximum characters number of a TextMesh.
Get TextMesh Num Drawn Characters	Get the number of drawn characters of a TextMesh.
Get TextMesh Properties	Get the textMesh properties in one go just for convenience.
Get TextMesh Scale	Get the scale of a TextMesh.
Get TextMesh Text	Get the text of a TextMesh.
Get TextMesh Texture Gradient	Set the texture gradient of the font currently applied to a TextMesh.
Is Playing	Check if a sprite animation is playing.
Is TextMesh Inline Styling Available	Check that inline styling can indeed be used (the font needs to have texture gradients for inline styling to work).
Make Sprite Pixel Perfect	Make a sprite pixel perfect.
Make TextMesh Pixel Perfect	Make a TextMesh pixel perfect.
Pause Animation	Pause a sprite animation.
Play Animation	Plays a sprite animation.
Resume Animation	Resume a sprite animation.
Set Animation Frame Rate	Set the current clip frames per seconds on a animated sprite.
Set Sprite Color	Set the color of a sprite.
Set Sprite ID	Set the sprite id of a sprite. Can use id or name.
Set Sprite Scale	Set the scale of a sprite.
Set TextMesh Anchor	Set the anchor of a TextMesh.
Set TextMesh Colors	Set the colors of a TextMesh.
Set TextMesh Font	Set the font of a TextMesh.
Set TextMesh Inline Styling	Set the inlineStyling flag of a TextMesh.
Set TextMesh Max Chars	Set the maximum characters number of a TextMesh.
Set TextMesh Properties	Set the TextMesh properties.
Set TextMesh Scale	Set the scale of a TextMesh.
Set TextMesh Text	Set the text of a TextMesh.
Set TextMesh Texture Gradient	Set the texture gradient of the font currently applied to a TextMesh.
Stop Animation	Stops a sprite animation.

AI For Mecanim

The following tasks are included in the AI For Mecanim integration:

Name	Description
Start State Machine	Start executing a AI For Mecanim State Machine. The task will stay in a running state until the State Machine has returned success or failure. The State Machine must finish with a Resume From AI For Mecanim action.
Stop State Machine	Stops executing a State Machine Behaviour. The task will return success immediately.
Run State Machine	Run a AI For Mecanim State machine that completes within the same frame. If the State Machine does not complete in time then the task will return failure. The State Machine must finish with a Resume From AI For Mecanim action.

Camera Path Animator

The following tasks are included in the Camera Path Animator integration:

Name	Description
Get Path Speed	Gets the speed of a Camera Path Animator path.
Pause	Pauses a Camera Path Animator path. Returns success if the path was paused.
Play	Plays a Camera Path Animator path. Returns success if the path was started.
Seek	Seeks to a specified point within the Camera Path Animator path. Returns success if the path animator was found.
Set Animation Mode	Sets the animation mode on a Camera Path Animator. Returns success if the mode was set.
Set Orientation Mode	Sets the orientation mode on a Camera Path Animator. Returns success if the mode was set.
Set Path Speed	Sets the speed of a Camera Path Animator path.
Stop	Stops a Camera Path Animator path. Returns success if the path was stopped.

Core GameKit

The following tasks are included in the Core GameKit integration:

Name	Description
Add Float	Add the value of a float World Variable.
Add Int	Add the value of a int World Variable
Attack Or Hit Points Add	Add attack points or hit points to a Killable in Core GameKit
Attack Or Hit Points Mod	Change attack points or hit points of a Killable in Core GameKit
Despawn	Despawn any one item
Despawn All Prefabs	Despawn all prefabs in Core GameKit
Despawn Killable	Despawn a Killable object
Despawn Prefabs of Type	Despawn all prefabs of one type in Core GameKit
Destroy	Destroy a Killable object
End Triggered Wave	End a wave of a certain event type in a Triggered Spawner
End Wave	End the current wave
Get Current Hit Points	Get the current Hit Points of a Killable in Core GameKit
Get Float	Get the value of a float World Variable
Get Int	Get the value of a int World Variable
Goto Wave	End the current wave in Core GameKit and go to a new Level or Wave of your choice!
Is Tiggered Wave Spawning	Check if a wave of a certain event type is spawning in a Triggered Spawner
Kill All Prefabs	Kill all prefabs in Core GameKit. Only prefabs with a Killable component will be affected
Kill Prefabs of Type	Kill all prefabs of one type in Core GameKit. Only prefabs with a Killable component will be affected
Multiply Float	Multiply the value of a float World Variable
Multiply Int	Multiply the value of a int World Variable
Pause Wave	Pause the current wave
Prefab Despawned Count	Get total number of a pooled prefab set up in Pool Boss in Core GameKit, despawned copies
Prefab Is In Pool	Return a boolean indicating if a Transform is pooled in Pool Boss in Core GameKit
Prefab Spawned Count	Get total number of a pooled prefab set up in Pool Boss in Core GameKit, spawned copies
Prefab Total Count	Get total number of a pooled item set up in Pool Boss in Core GameKit, spawned and despawned copies
Prefab Type Count In Pool	Get total number of different prefabs set up in Pool Boss in Core GameKit
Restart Wave	Restart the current wave
Resume Wave	Resume the current wave from a pause
Set Float	Set the value of a float World Variable
Set Int	Set the value of a int World Variable
Spawn From Pool	Spawn one prefab from Pool Boss in Core GameKit
Spawn One	Spawn one item from a Syncro Spawner
Take Damage	Inflict points of damage on a Killable
Temporary Invincibility	Make a Killable object invincible for X seconds

Dialogue System

The following tasks are included with Dialogue System integration:

Name	Description
Bark	Makes an NPC bark.
Get Quest Entry State	Gets the state of a quest entry in a quest.
Get Quest State	Gets the state of a quest.
Is Lua True	Returns if the Lua code is true.
Run Lua	Runs Lua code.
Set Quest Entry State	Sets the state of a quest entry in a quest.
Set Quest State	Sets the state of a quest.
Start Sequence	Starts a cutscene sequence.
Stop Conversation	Stops the current conversation.
Stop Sequence	Stops a sequence.

Final IK

The following tasks are included in the Final IK integration:

Name	Description
Aim IK	Manages the AimIK component
Biped IK	Manages the BipedIK component
CCD IK	Manages the CCDIK component
FABR IK	Manages the FABRIK component
FABR IK Root	Manages the FABRIKRoot component
FABR IK Root Chain	Manages a chain of a FABRIKRoot component
FBB IK Body	Manages a FullBodyBipedIK effector
FBB IK Limb	Manages a FullBodyBipedIK limb. You can alternately use FBBIKEffector and FBBIKChain and FBBIKMapping
FBB IK Settings	Manages the general settings of a FullBodyBipedIK component
IK Execution Order	Controls the updating order of IK components
Limb IK	Manages the LimbIK component
Look At IK	Manages the LookAtIK component
Pause Interaction	Pauses an interaction with the InteractionSystem
Resume Interaction	Resumes a paused interaction with the InteractionSystem
Start Interaction	Starts an interaction with the InteractionSystem
Stop Interaction	Stops an interaction with the InteractionSystem
Trigonometric IK	Manages the TrigonometricIK component

Master Audio

The following tasks are included in the Master Audio integration:

Name	Description
Add Ducking Group	Add a Sound Group to the list of sounds that cause music ducking.
Change Variation Pitch	Change the pich of a variation (or all variations) in a Sound Group.
Fade Bus	Fade a Bus to a specific volume over X seconds.
Fade Group	Fade a Sound Group to a specific volume over X seconds.
Fade Out All Of Sound Group	Fade all of a Sound Group to zero volume over X seconds.
Fade Playlist	Fade the Playlist volume to a specific volume over X seconds.
Fire Custom Event	Fire a Custom Event.
Get Current Playlist Clip Name	Get the name of the currently playing Audio Clip in a Playlist.
Mute Bus	Mute all Audio in a Bus.
Mute Everything	Mute all sound effects and Playlists.
Mute Group	Mute a Sound Group.
Mute Playlist	Mute a Playlist.
Next Playlist Clip	Play the next clip in a Playlist.
Pause Bus	Pause all Audio in a Bus.
Pause Everything	Pause all sound effects and Playlists.
Pause Group	Pause all Audio in a Sound Group.
Pause Mixer	Pause all sound effects. Does not include Playlists.
Pause Playlist	Pause a Playlist.
Play Playlist By Clip Name	Play a clip in the current Playlist by name.
Play Random Playlist Clip	Play a random clip in a Playlist.
Play Sound	Play a Sound.
Remove Ducking Group	Remove a Sound Group from the list of sounds that cause music ducking.
Set Bus Volume	Set a single bus volume level.
Set Group Volume	Set a single Sound Group volume level.
Set Master Volume	Set master volume level.
Set Playlist Volume	Set the Playlist Master volume to a specific volume.
Solo Bus	Solo all Audio in a Bus.
Solo Group	Solo a Sound Group.
Start Playlist By Name	Start a Playlist by name.
Stop All Of Sound	Stop all of a Sound Group.
Stop Bus	Stop all Audio in a Bus.
Stop Everything	Stop all sound effects and Playlists.
Stop Mixer	Stop all sound effects. Does not include Playlists.
Stop Playlist	Stop current Playlist.
Stop Transform Sound	Stop sounds made by a Transform.
Toggle Ducking	Turn music ducking on or off.
Toggle Group Mute	Toggle the mute button of a Sound Group.
Toggle Group Solo	Toggle the solo button of a Sound Group.
Toggle Playlist Mute	Toggle mute on a Playlist.
Unmute Bus	Unmute all Audio in a Bus.
Unmute Everything	Unmute all sound effects and Playlists.
Unmute Group	Unmute a Sound Group.
Unmute Playlist	Unmute a Playlist.
Unpause Bus	Unpause all Audio in a Bus.
Unpause Everything	Unpause all sound effects and Playlists.
Unpause Group	Unpause all Audio in a Sound Group.
Unpause Mixer	Unpause all sound effects. Does not include Playlists.
Unpause Playlist	Unpause a Playlist.
Unsolo Bus	Unsolo all Audio in a Bus.
Unsolo Group	Unsolo a Sound Group.

Motion Controller

The following tasks are included with the Motion Controller integration:

Name	Description
Activate Motion	Activates a motion on the avatar's Motion Controller.
Deactivate Motion	Deactivates a running motion on the avatar's Motion Controller.
Set Active Motion Phase	Set's the animator's motion phase for the active motion.
Is Motion Active	Returns success when the specified motions is active.
Traverse Towards	Moves the Motion Controller based avatar to a specific point, but gives the avatar the ability to jump and climb past obstacles.
Traverse Patrol	Moves the Motion Controller based avatar along specific points, but gives the avatar the ability to jump and climb past obstacles.

NGUI

The following tasks are included in the NGUI integration:

Name	Description

Get Label Text	Stores the UILabel text.
Get Scroll Bar Value	Stores the UIScrollBar value.
Get Slider Value	Stores the UISlider value.
Get Widget Alpha	Stores the UIWidget alpha value.
Get Widget Color	Stores the UIWidget color value.
Set Label Text	Sets the UILabel text.
Set Scroll Bar Value	Sets the UIScrollBar value.
Set Slider Value	Sets the UISlider value.
Set Sprite	Replaces the current UISprite with the given sprite.
Set Widget Alpha	Sets the UIWidget alpha value.
Set Widget Color	Sets the UIWidget color value.
Set Widget Enabled	Enables or disables the UIWidget.
Simulate Click	Simulates a click on the UIButton or UIPlayTween component.
Widget Enabled	Returns success if the UIWidget is enabled, otherwise failure.

PlayMaker

PlayMaker integration details can be found on the PlayMaker Integration topic. The following tasks are included with the PlayMaker integration:

Name	Description
Start FSM	Start executing a PlayMaker FSM. The task will stay in a running state until PlayMaker FSM has returned success or failure. The PlayMaker FSM must contain a Behavior Listener state with the specified event name to start executing and finish with a Resume From PlayMaker action.
Stop FSM	Sends an event to a PlayMaker FSM. The task will return success immediately.
Run Conditional FSM	Run a PlayMaker FSM that completes within the same frame. If the FSM does not complete in time then the task will return failure. The PlayMaker FSM must contain a Behavior Listener state with the specified event name to start executing and finish with a Resume From PlayMaker action.
Send Event	Stops executing a PlayMaker FSM. The task will return success immediately.
Broadcast Event	Broadcasts an event to a PlayMaker FSM. The task will return success immediately.

Pool Manager

The following tasks are included in the PoolManager integration:

Name	Description
Check If Prefab Pool Exists	Returns success if the prefab already exists within the pool.
Create Pool	Creates a new GameObject with a SpawnPool Component which registers itself with the PoolManager.Pools dictionary.
Create Prefab Pool	Creates a new PrefabPool in this pool and instances the specified number of instances.
Despawn	If the passed GameObject is managed by the SpawnPool, it will be deactivated and made available to be spawned again.
Destroy All Pools	Destroys all SpawnPools in PoolManager.Pools including all instances and references as well as the GameObjects.
Destroy Pool	Destroys a SpawnPool in PoolManager.Pools including all instances and references as well as the GameObject.
Get Pool Group	Gets the specified SpawnPool group.
Get Pool Instances Count	Get the number of spawned instances in the pool.
Get Pools Count	Get the number of SpawnPools in the pool.
Spawn	Spawn an object from the pool if one is available.

Simple Waypoint System

The following tasks are included in the Simple Waypoint System integration:

Name	Description
Chase Speed	Changes speed of a walker object.
Get Waypoint of Path	Gets the desired waypoint of a bezier path.
Pause Movement	Pauses movement of a walker object.
Resume Movement	Resumes movement of a previously paused walker object.
Set Delay at Waypoint	Sets delay at a waypoint of a walker object.
Set Path	Sets the path of a walker object and starts movement. Can use the path name or the path object.
Set Waypoint of Path	Sets the desired waypoint of a path.
Start Movement	Starts movement of a walker object.
Stop Movement	Stops movement of a walker object. Optionally resets it to start.
Update Bezier Path	Recalculates waypoints of a bezier path in case it moved. Note that this is a performance-heavy action.

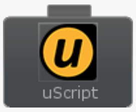
Ultimate FPS

The following tasks are included in the UFPS integration:

Name	Description
Add Item	Adds a item to the inventory specified by its name and count.
Attack	Aims and attacks with the current weapon. Returns success after the weapon has been used to attack.

Can Interact	Returns success if the agent can itneract otherwise failure.
Deplete Ammo	Depletes the ammo by one unit.
Has Ammo	Returns success if the agent has ammo, otherwise failure.
Has Weapon	Returns success if the agent currently has a weapon, otherwise failure.
Has Weapon Clip	Returns success if the agent currently has a weapon clip, otherwise failure.
Interact	Interacts with the current object.
Is Agent Alive	Returns success if the agent has a health greater than 0, otherwise failure.
Is Damagable Alive	Returns success if the damage handler has a health greater than 0, otherwise failure.
Refill Current Weapon	Refills the current weapon.
Reload	Reloads the current weapon.
Remove Item	Removes the specified item from the inventory.
Set Weapon	Sets the current weapon to the weapon specified by its index.
Set Weapon By Name	Sets the current weapon to the weapon specified by its name.

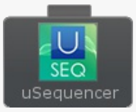
uScript



uScript integration details can be found in the uScript Integration topic. The following tasks are included with uScript integration:

Name	Description
Start Graph	Start executing a uScript graph. The task will stay in a running state until the uScript graph has returned success or failure. The uScript graph must contain a Behavior Signal to start executing and finish with a uScript Resume Behavior action.
Run Conditional Graph	Run a uScript graph that completes within the same frame. If the graph does not complete in time then the task will return failure. The uScript graph must contain a Behavior Signal to start executing and finish with a uScript Resume Behavior action.

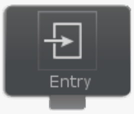
uSequencer



The following tasks are included in the uSequencer integration:

Name	Description
Is Sequence Playing	Returns success if a uSequencer sequence is playing.
Pause Sequence	Pauses a uSequencer sequence.
Play Sequence From Time	Plays a uSequencer sequence from a specified time. Returns success if the sequence can be started.
Play Sequence	Plays a uSequencer sequence. Returns success if the sequence can be started.
Set Sequence Time	Sets the current time of a uSequencer sequence.
Stop Sequence	Stops a uSequencer sequence.

Entry Task



The entry task is a task that is used for display purposes within Behavior Designer to indicate the root of the tree. It is not a real task and cannot be used within the behavior tree.

Support

We are here to help! If you have any questions/problems/suggestions please don't hesitate to ask. You can email us at support@opsive.com or post on the forum.