Opsive Home Games Assets About

Behavior Designer Documentation

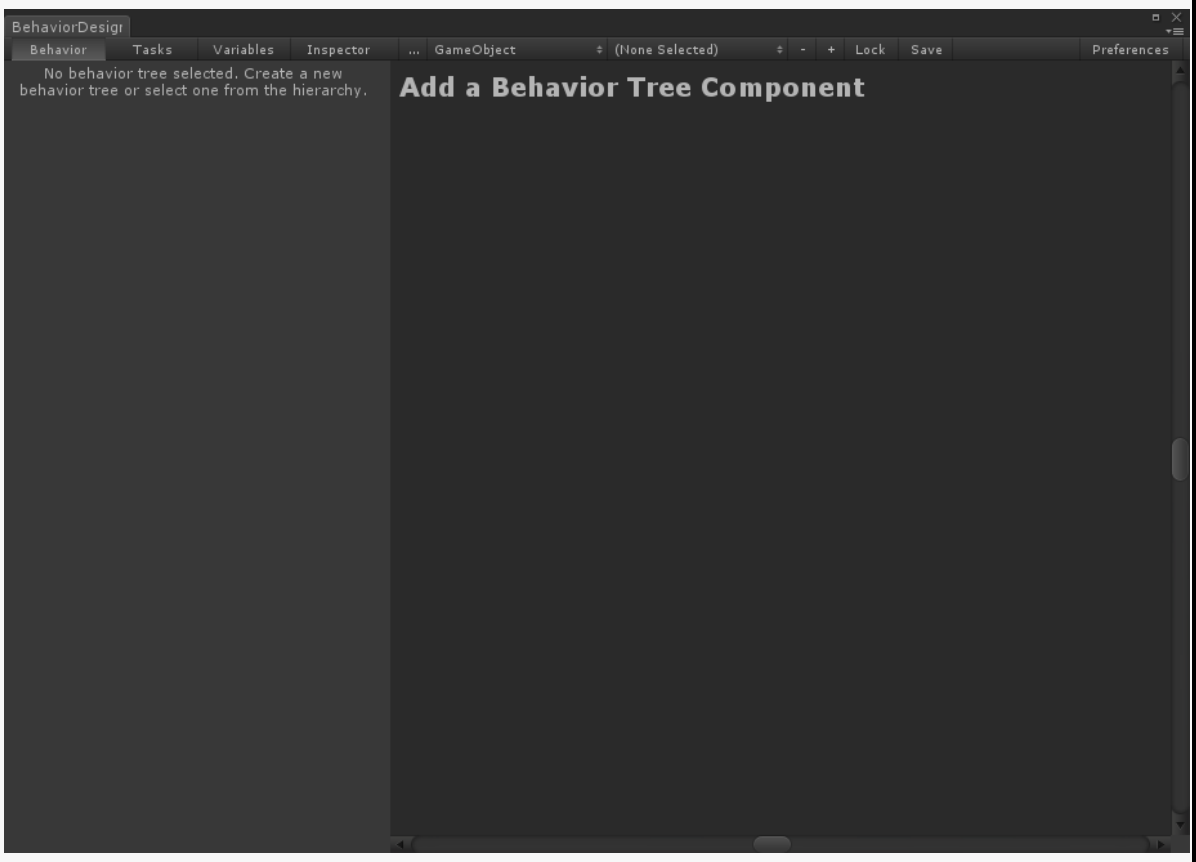
Thank you for your purchase! The most recent documentation (with better formatting) can be found online. If you have any questions feel free to post on the Unity forums or email support@opsive.com.

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).

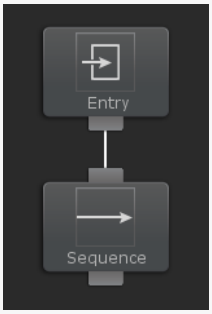
When you first open Behavior Designer you'll be presented with the following window:



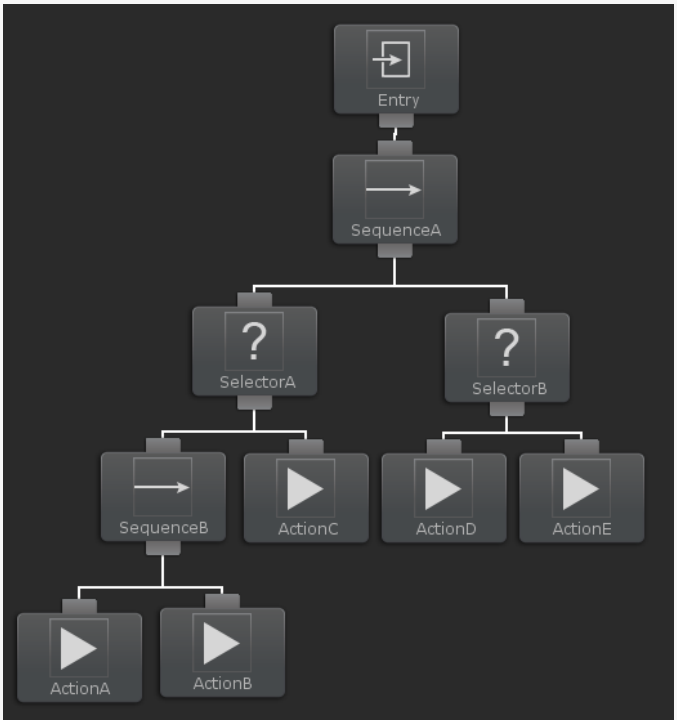
There are three sections within Behavior Designer. From the screenshot below, section 1 is the graph area. It is where you'll be creating the behavior trees. Section 2 is a properties panel. The properties panel is where you'll be editing the specific properties of a behavior tree, adding new tasks,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

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:



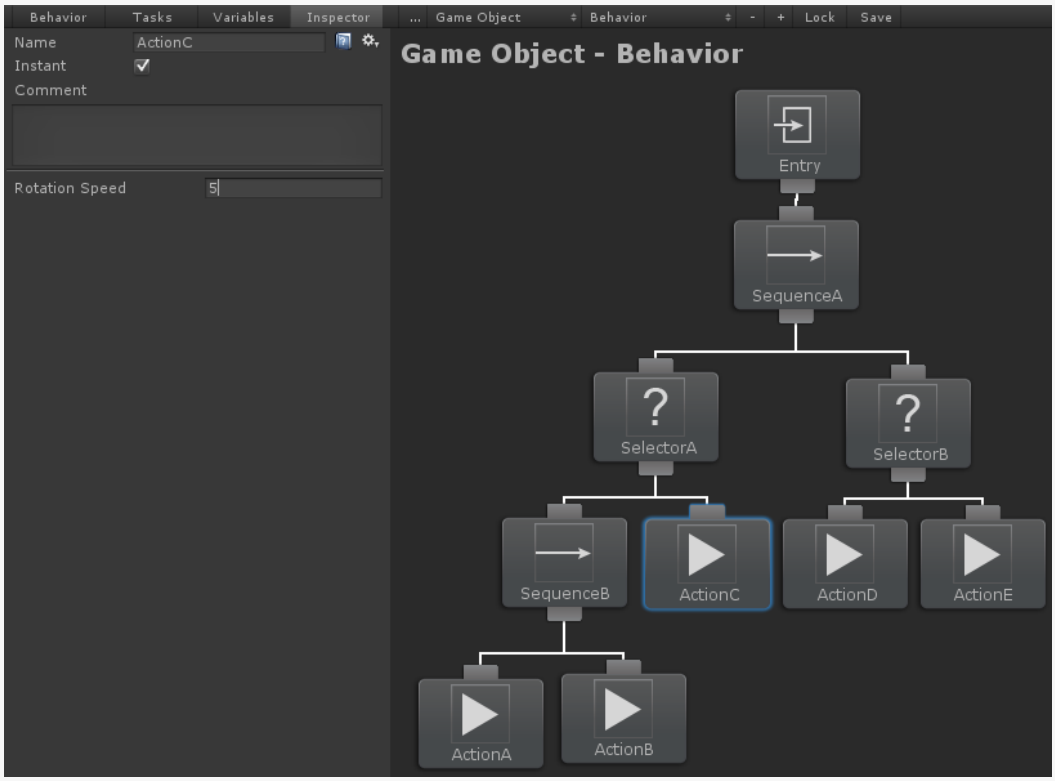
In addition to the task that you added, the entry task also gets added. The entry task acts as the root of the tree. That is the only purpose of the entry task. Now that we've added our first task lets add a few more:



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.

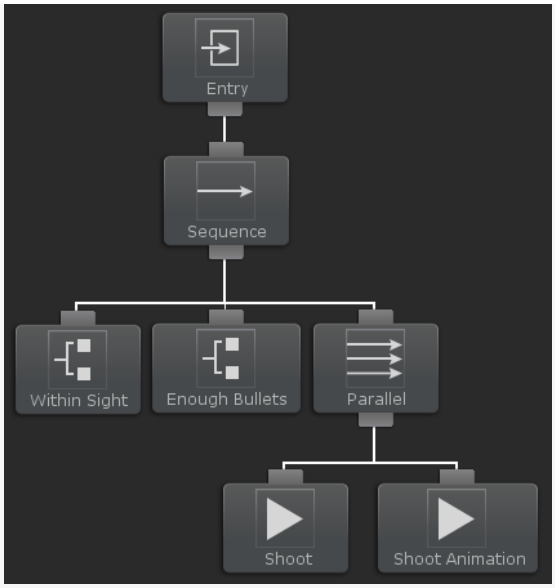
There are three other tabs within the properties panel: Variables, Tasks, and Behavior. The variables panel allows you to create variables that are shared between tasks. For more information take a look at the variables topic. The tasks panel lists all of the possible tasks that you can use. This is 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 drop down box with the number 1 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 2 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 3 label will list any behavior trees that are attached to the game object that is selected from the number 2 drop down box.

The button with the number 4 label will remove the currently selected behavior tree. The button with the number 5 label will add a new behavior tree. The "Lock" button (number 6) 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 7) will save the current behavior tree out as an asset. Finally, the "Preferences" button (number 8) 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 want to see behavior trees in action, take a look at one of the sample project videos.



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.

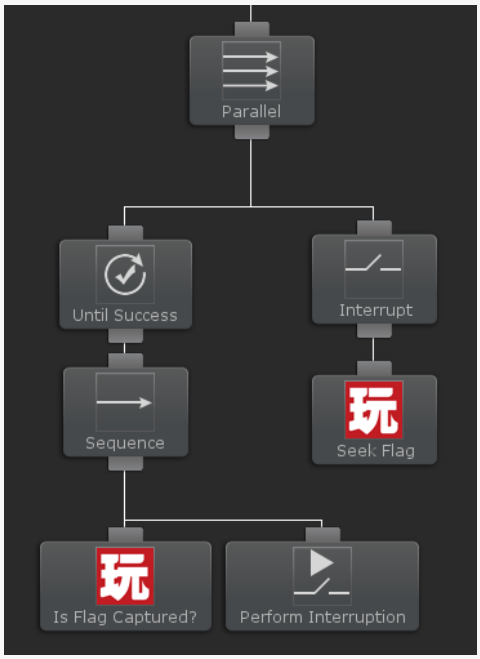
Behavior Designer takes all of these concepts and packages it up in an easy to use interface with an API that is similar to Unity's MonoBehaviour API. Behavior Designer includes many composite and decorator classes within the standard installation. Action and conditional tasks are more game specific 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)? In our view that is the wrong question to ask. Instead the question should be why use a behavior tree and a finite state machine **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.

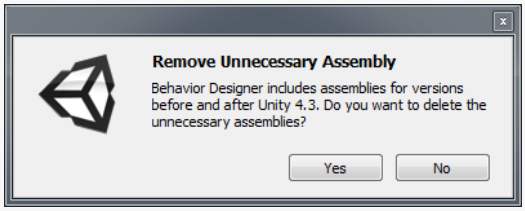
One final behavior tree advantage is that they are really easy to make changes to. One of the reasons behavior trees became so popular is because they are easy to create with a visual editor. If you want to change the state execution order with a finite state machine you have to change the transitions 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

The next release of Behavior Designer will include the runtime source code so this process will change

Behavior designer ships with four assemblies: two for the runtime and two for the editor, each having a version that runs 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 assemblies:

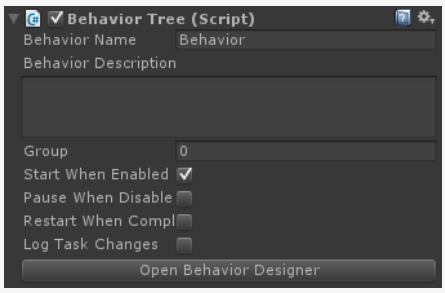


If you select yes the script will automatically remove the assembly that does not correspond with your Unity version. If you select no the script will not run and you will keep getting this message every time you import until you either manually remove the assemblies or manually remove the script. If 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.

Behavior Tree Component



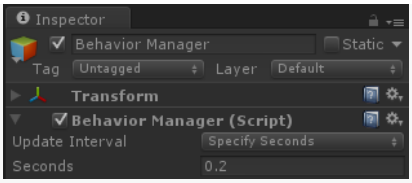
The behavior tree component stores your behavior tree and acts as the interface between Behavior Designer and the tasks. The following API is exposed for starting and stopping your behavior tree:

```
public void enableBehavior();
public void disableBehavior(bool pause = false);
```

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.
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.

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();
```

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

Tasks

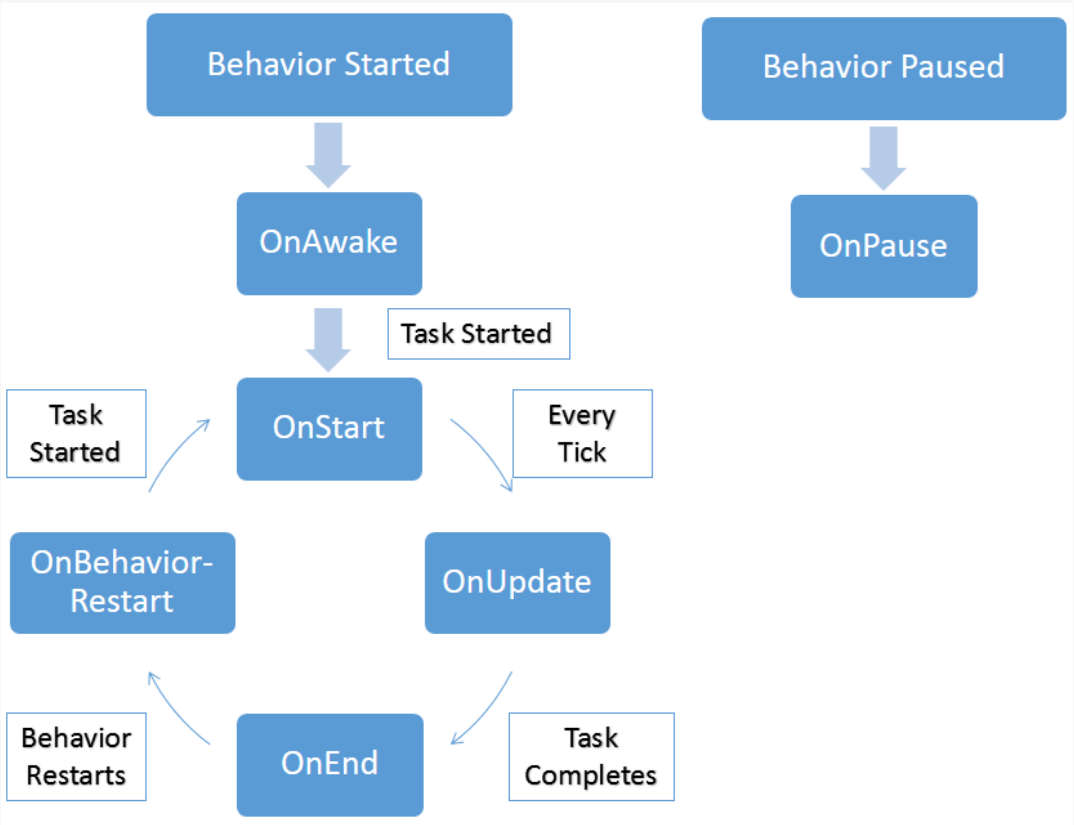
At the highest level a behavior tree is a collection of tasks. Tasks have a really similar API to Unity's MonoBehaviour so it should be really easy to get started writing your own tasks. The task class has the following API:

```
// OnAwake is called once before the task is executed. 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();

// OnBehaviorRestart is called after a complete behavior execution and the behavior is going to restart
public virtual void OnBehaviorRestart();

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

// Same as Editor.OnSceneGUI except this is executed on a runtime class
public virtual void OnSceneGUI();

// 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();
```

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.

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 hte 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;
}
```

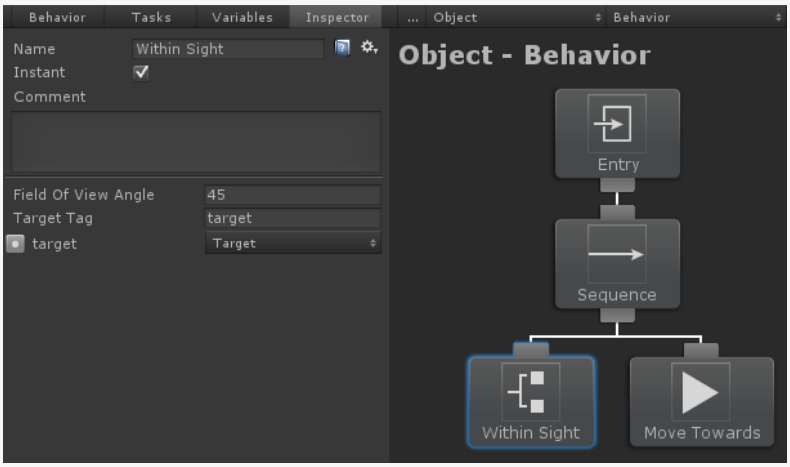
When the OnUpdate method is run, it will check to see if the object has reached the target. If the object has reached the target then the task will success. If the target has not been reached yet the object will move towards the target at a speed specified by the speed variable. Since the object hasn't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

Referencing Tasks

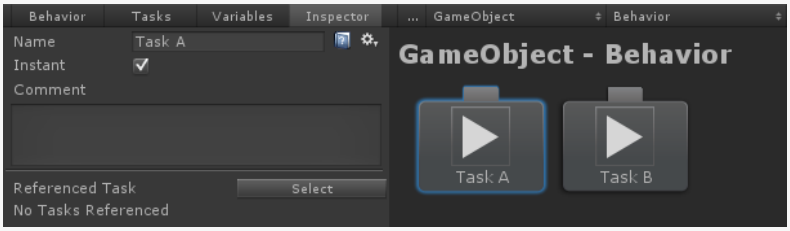
When writing a new task, in some cases it is necessary to access another task within that task. For example, TaskA may want to get the value of TaskB.SomeFloat. To accomplish this, TaskB needs to be referenced from TaskA. In this example TaskA looks like:

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

public class TaskA : Action
{
    public TaskB referencedTask;

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

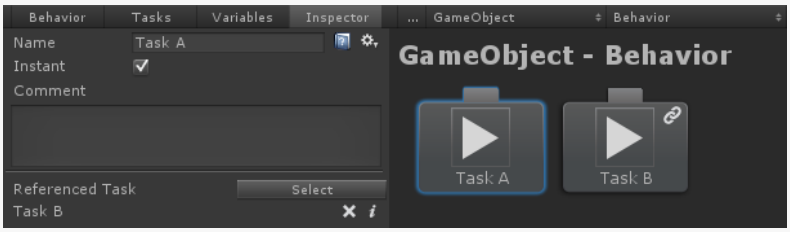
TaskB then looks like:

```
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;
}
```

When a task is referenced it can be linked which will share the value of synchronized fields.

Task Attributes

Behavior Designer exposes the following task attributes: HelpURL, TaskIcon, TaskCategory, LinkedTask, and InheritedField. SharedField and SynchronizedField have been deprecated.

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("{SkinColor}ParallelIcon.png")]
public class Parallel : Composite
{
```

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
{
```

Variables are great when you want to share information between tasks. However, you'll notice that there is no such thing as a "SharedTask". For this we have the LinkedTask and it is discussed in detail here.

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.

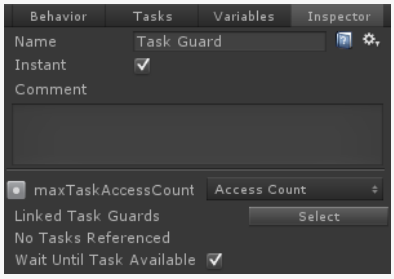
```
[InheritedField]
public float moveSpeed;
```

Linking Tasks

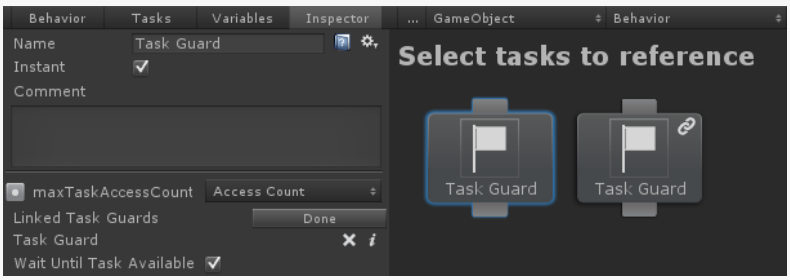
Variables are great when you want to share information between tasks. However, you'll notice that there is no such thing as a "SharedTask". When you want a group of tasks to share the same tasks use the LinkedTask attribute. As an example, take a look at the task guard task. When you reference one task 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;
```

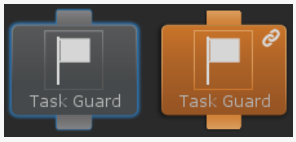
To perform a link within the editor first select the original task that has a field that you want to link another task to. Now look at the inspector, to the right of the field you'll see a "Select" button.



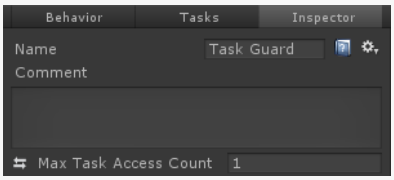
If you click that button you'll be placed in a mode where you can select other tasks. Once you select another task you'll then see a link icon appear on the top right of that newly linked task.



You can clear the link by clicking on the "x" to the right of the linked task name. If you click on the "i" then the linked task will highlight in orange:



Synchronized Field



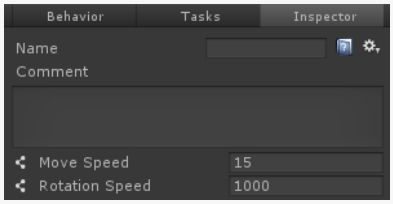
Synchronized fields have been deprecated in version 1.1. Use variables instead.

When a task is linked against another task, all of the fields who are marked with the SynchronizedField attribute will share the same value.

```
[SynchronizedField]
public int maxTaskAccessCount = 1;
```

When you change the value of maxTaskAccessCount on one task, that same field will be changed to the same value for any other linked tasks. If the synchronized field is an array, that new value will be propagated to every task. Tasks which are shared have the shared icon shown above, to the left of "Max Task Account Count". Synchronized fields are only used by the editor and are the values are not synchronized during runtime.

Shared Field



Shared fields have been deprecated in version 1.1. Use variables instead.

Shared fields are very similar to SynchronizedField attribute except that they don't only affect linked tasks. If a task has a property with the SharedField attribute it will share its value with the other tasks within the behavior tree:

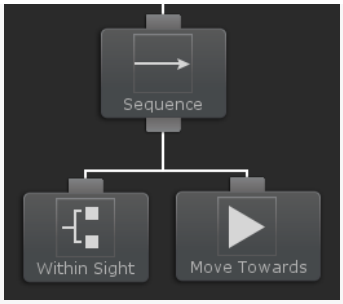
```
[SharedField]
public float moveSpeed = 1;
```

For example, if the variable moveSpeed has the SharedField attribute than any other task that also has a shared moveSpeed variable will always have the same value. If you change the moveSpeed task within one task than any other task with the moveSpeed field will also change to the same value. Shared fields are marked with the icon shown below, to the left of "Move Speed" and "Rotation Speed". Shared fields are only used by the editor and are the values are not shared during runtime.

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.

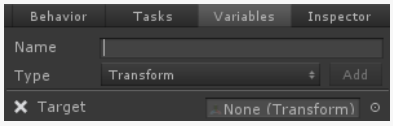
In our previous example we had two tasks: one that determined if the target is within sight and then the other task moves towards the target. This tree looks like:



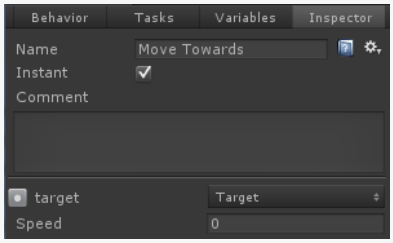
The code for both of these tasks is discussed in the Writing a New Task topic, but the part that deals with variables is in this variable declaration:

```
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:

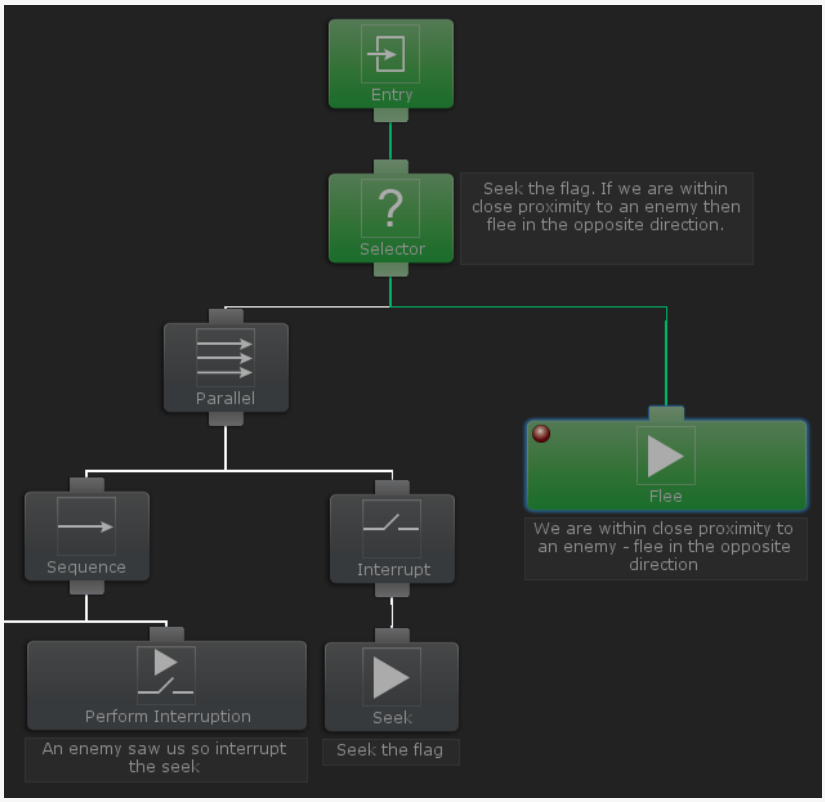


And with that the two tasks can start to share information! You can get/set the value of the shared variable by accessing the Value property. For example, target.Value will return the transform object. When Within Sight runs it will assign the transform of the object that comes within sight to the Target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

The following shared variable classes are supported:

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

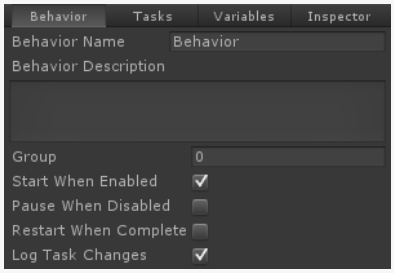
Debugging



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. 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.



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
```

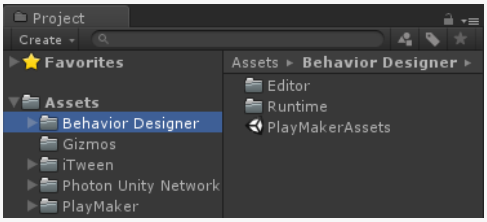
These messages can be broken up into the following pieces:

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

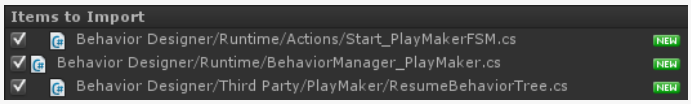
{game object name} is the name of the game object that the behavior tree is attached to. {behavior name} is the name of the behavior tree. {task change} indicates the new status of the task. For example, a task will be pushed onto the stack when it starts executing and it will be popped when it is done 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.

PlayMaker Integration

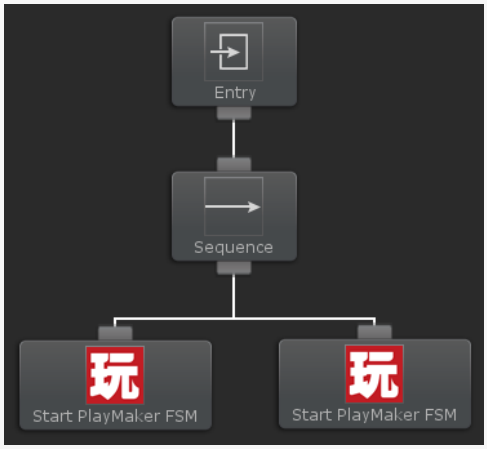
PlayMaker is a popular visual scripting tool which allows you to easily create finite state machines. Behavior Designer integrates directly with PlayMaker by allowing PlayMaker to carry out the action or conditional tasks and then resume the behavior tree from where it left off. We had to place all of the PlayMaker files in a separate Unity package since PlayMaker is not required for Behavior Designer to work:



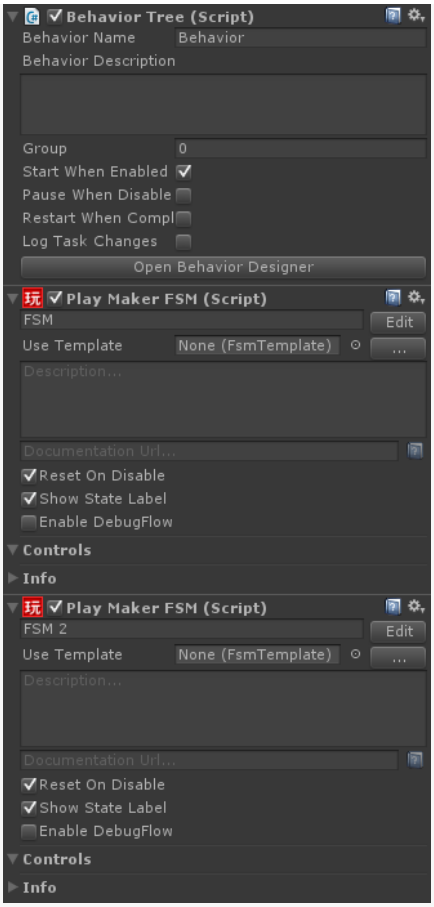
To get started, first make sure you have PlayMaker installed. Next, import PlayMakerAssets.unitypackage:



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 PlayMaker child tasks:

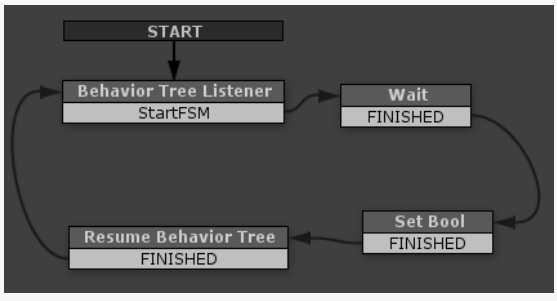


Next add two PlayMaker FSM components to the same game object that you added the behavior tree to.

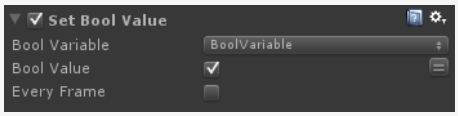


Open PlayMaker and start creating a new FSM. This FSM is going to be a simple FSM to show how Behavior Designer interacts with PlayMaker. For a more complicated FSM take a look at the FPS sample project. Behavior Designer starts the PlayMaker FSM by sending it an event. Create this event by adding a 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.

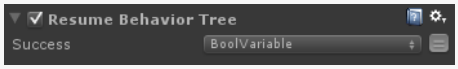
Add a transition from that event along with a wait state, a set bool state, and a resume behavior tree state. Make sure you transition from the Resume Behavior Tree state to the Behavior Tree Listener state so the FSM can be started again from Behavior Designer.



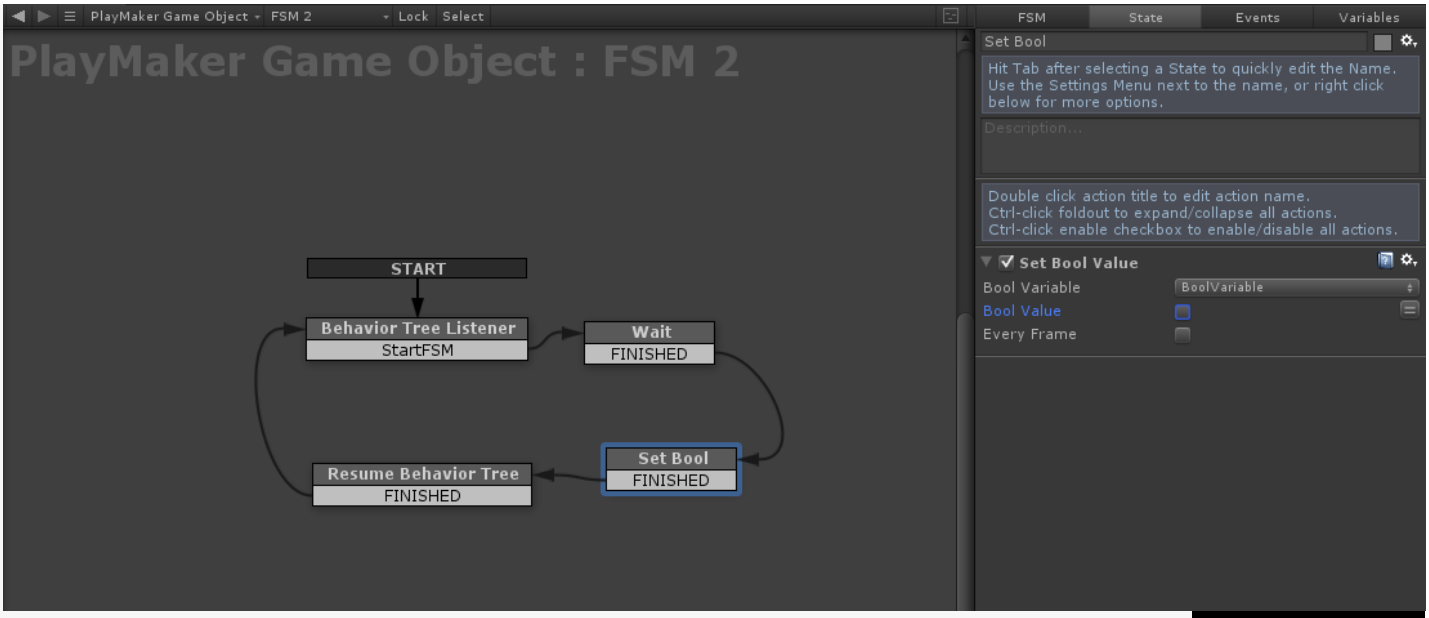
Create a new variable within the Set Bool state and set that value to true.



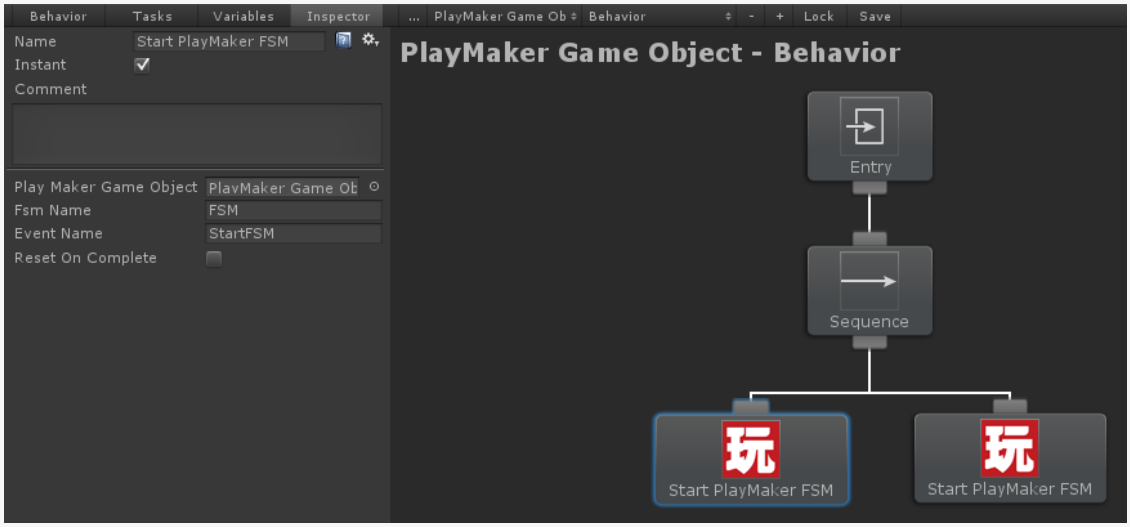
Then within the Resume Behavior Tree state we want to return success based off of that bool value:



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.

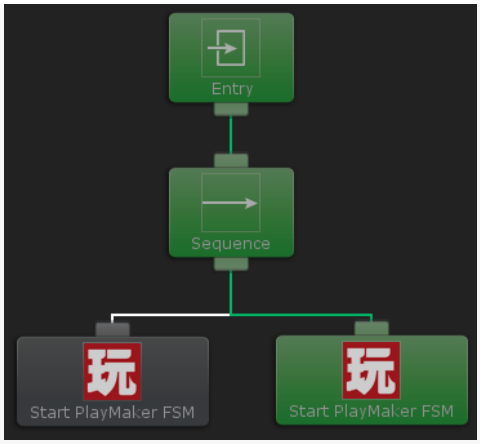


We are now done working in PlayMaker. Open your behavior tree back up within Behavior Designer. Select the left PlayMaker task and start assigning the values to the variables. PlayMaker Game Object is assigned to the game object that we added the PlayMaker FSM components to. FSM Name is the name of the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

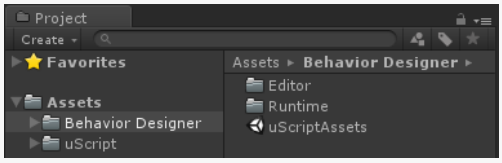


If you were to swap the tasks so the second PlayMaker task runs before the first PlayMaker FSM, the behavior tree will never get to the first PlayMaker FSM because the second PlayMaker FSM returned failure and the sequence task stopped executing its children.

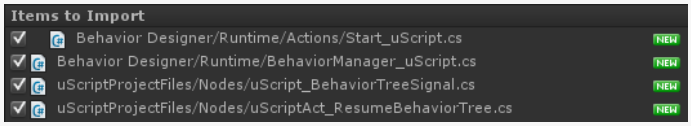
---

uScript Integration

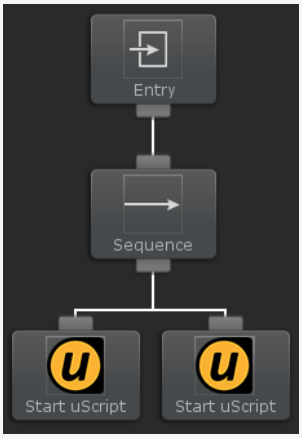
uScript is a popular visual scripting tool which allows you to create complicated setups without needing to write a single line of code. Behavior Designer integrates directly with uScript by allowing uScript to carry out the action or conditional tasks and then resume the behavior tree from where it left off. We had to place all of the uScript files in a separate Unity package since uScript is not required for Behavior Designer to work:



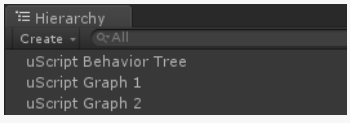
To get started, first make sure you have uScript installed. Next, import uScriptAssets.unitypackage:



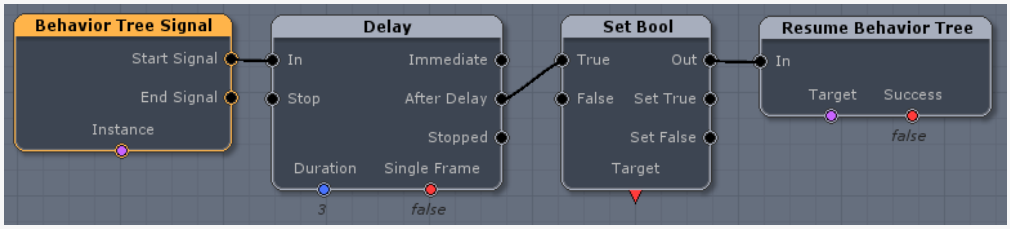
Once those files are imported you are ready to start creating Behavior Trees with uScript! To get started, create a very basic tree with a sequence task who has two uScript child tasks:



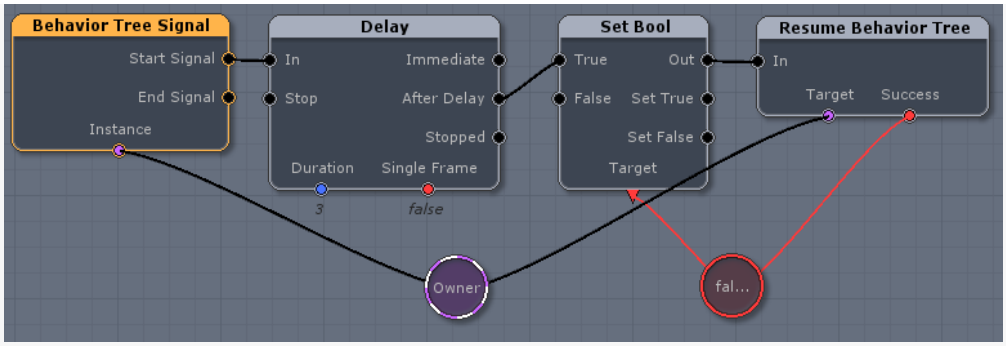
Now we need to create two GameObjects which will hold the compiled uScript graph:



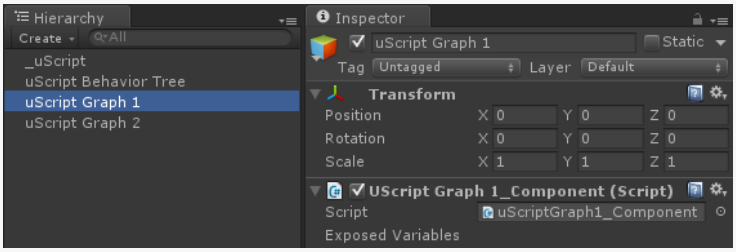
Open uScript and start creating a new graph. Add the Behavior Tree Signal node, located under Events/Signals. When Behavior Designer wants to start executing a uScript graph it will start from this node. This node contains two events – Start Signal and End Signal. Start Signal is used when the behavior tree task starts running. End Signal gets called if the behavior tree task has to end prematurely – for example, if it was interrupted. For our graph we are only going to create a few nodes, the soccer 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/Utilities:



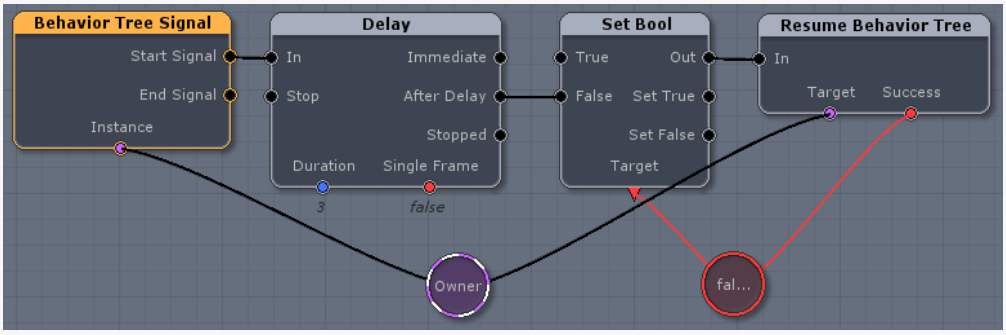
Now we need to create a Owner GameObject and bool variable.



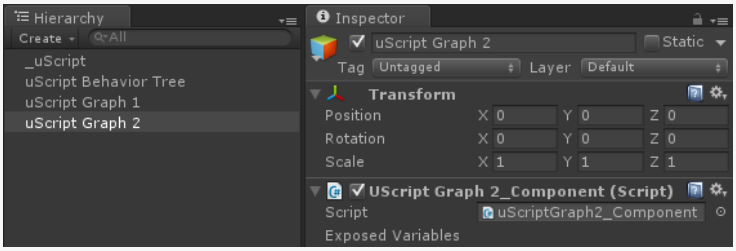
Save the uScript graph and assign the component to your first uScript graph GameObject. Answer no if uScript asks if you want to assign the component to the master GameObject.



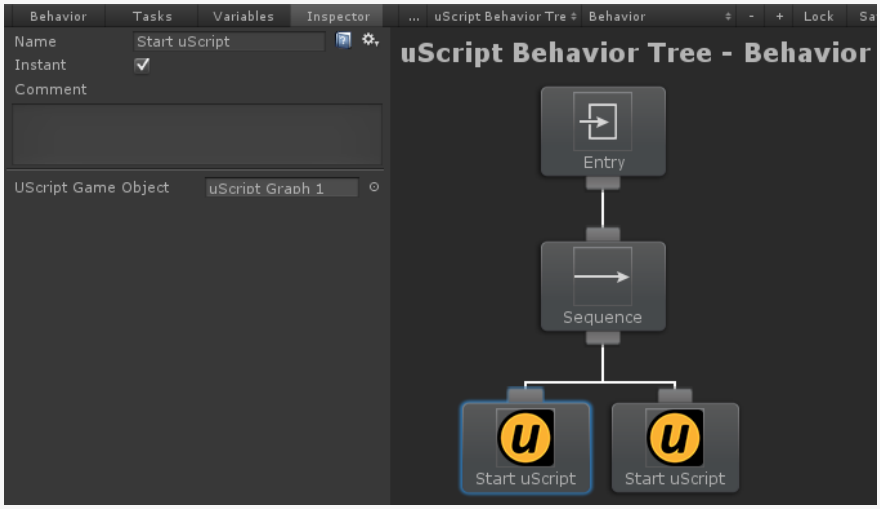
Create one more uScript graph. Make it the same as the last graph except set the bool to false:



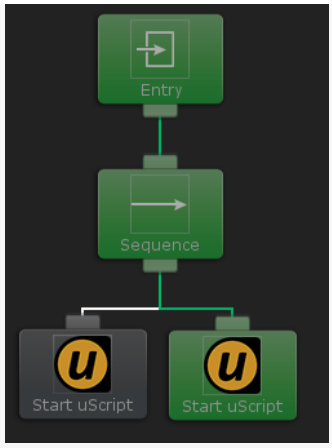
Finally save that graph and assign the component to the second uScript GameObject:



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.

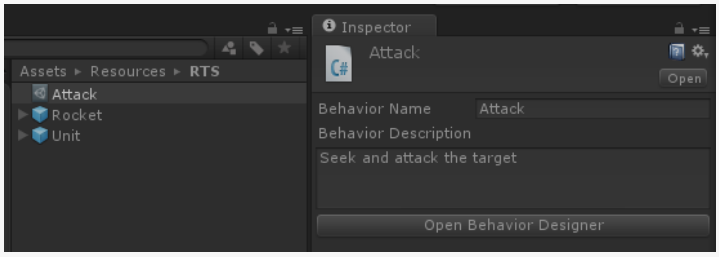


Do the same for the right uScript task, only assign the uScript GameObject to your second uScript graph GameObject. That's it! When you hit play you'll see the first uScript task run for three seconds, followed by the second uScript task.



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.

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.

Task Types

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

- Entry Task
- Actions
  - Behavior Tree Reference
  - Log
  - Perform Interruption
  - Start PlayMaker FSM
  - Start uScript
  - Start Behavior
  - Stop Behavior
  - Wait
  - External Behavior Tree
- Composites
  - Sequence
  - Selector
  - Parallel
  - Parallel Selector
  - Priority Selector
  - Random Selector
  - Random Sequence
- Conditionals
  - Random Probability
- Decorators
  - Until Success
  - Interrupt
  - Inverter
  - Repeater
  - Return Failure
  - Return Success
  - Task Guard
  - Until Failure

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
- Log
- Perform Interruption
- Start PlayMaker FSM
- Start uScript
- Start Behavior
- Stop Behavior
- Wait
- External Behavior Tree

External Behavior Tree

The external behavior tree task has been deprecated in version 1.1. Use the behavior tree reference task instead.

The external behavior tree task allows you to run another behavior tree within the current behavior tree. One use for this is that if you have a 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.

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 getExternalBehavior method allows you to override it so you can provide an external behavior tree 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?

Start PlayMaker 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. More details can be found in the PlayMaker Integration section.

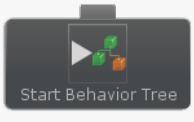
Name	Description
playMakerGameObject	The GameObject that the PlayMaker FSM component is attached to.
FsmName	The name of the FSM component. This allows you to have multiple FSM components on a single GameObject.
eventName	The name of the event to fire to start executing the FSM within PlayMaker.
resetOnComplete	When the PlayMaker FSM is complete should we restart the FSM back to its original state? This variable is used by the Behavior Manager.

Start uScript

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. More details can be found in the uScript Integration section.

Name	Description
uScriptGameObject	The GameObject that the uScript component is attached to.
status	The return status of uScript after it has finished executing.

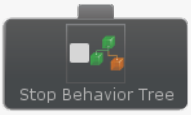
Start Behavior



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.

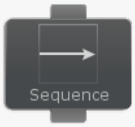
Composites

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.

- 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

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?

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.

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

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.
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.

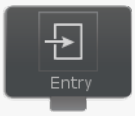
Until Failure

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.

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 Unity forums.