Copyright Notice

Important!

You are reading an offline, compiled version of the uScript Development Guide. There may be a newer version of the guide with updated information available online. To see the online version of this guide at any time, or download the latest compiled version for offline reading, please go to **http://docs.uscript.net/development**

This is version **1.0.0** of the uScript Development Guide.

Overview

Welcome to the uScript Development Guide! This guide contains a collection of information useful to modifying and extending uScript functionality through creating new nodes. If you are looking for information on using uScript itself, please see the uScript User Guide instead. Also, you may want to be sure to look at the "*Advanced uScript Topics*" section of the *uScript User Guide* as well for important, related information to working with uScript.

uScript supports two types of custom nodes:

> **Action Nodes** are triggered by uScript, perform custom logic, push values back out to uScript, and specify which subsequent branches can be executed. Examples of existing Action Nodes are the Delay Node and Add Float Node.
>
> **Event Nodes** are triggered by external events in Unity or your game. Once triggered they behave as Action Nodes by pushing values to uScript and specifying which subsequent branches can be executed. Examples of existing Event Nodes are Trigger Events and Input Events.

File Locations

uScript's Runtime Folder

The *uScriptRuntime* folder, located in the root *uScript* folder, contains all the Nodes (.cs files) which ship with uScript. It also contains the *uScriptRuntime.dll* which is a series of attributes and base classes for the uScriptEditor and uScript nodes to reference.

Files currently located here include:

> **uScriptRuntime.dll** - The uScript Editor has to know some basic functionality about the nodes it displays. This includes attributes, stubbed methods, etc. Since the editor is a dll it requires this basic functionality to also be in a dll (a compiled dll can't reference loose code files). This is our only reason for using uScriptRuntime.dll instead of loose code files. uScriptRuntime.dll contains no functional code outside of defining node attributes and stubbed methods.
>
> **uScriptUnityVersion.cs** - This is a simple helper file we use to keep track of which version of Unity uScript is running in for the purpose of handling various features, functionality and potential code deprecations between versions.
>
> **Official uScript Nodes** - Inside the Nodes folder you can find all of the nodes that ship with uScript. These nodes are standard C-Sharp script files that can be used as reference for creating your own nodes. Please note however that if you modify these nodes in any way, your changes may be overridden and your work lost if ever updating or reinstalling uScript. It is usually better to create your own nodes for use when needed and save them in the *uScriptProjectFiles* folder.

Node Files And The Generated Graph Code

uScript generated code works much like the uScript graph you create. The generated code is simply a series of links which call the nodes they are connected to. The Nodes folder contains these nodes as .cs files. If we embedded the nodes within your generated code then your generated code would become bloated and wouldn't be able to leverage future optimization to these nodes.

uScript's Project Folder

uScript saves all generated files for your project to a specific project folder called *uScriptProjectFiles*. This folder is located in the root of your Unity project's Asset folder. It is recommended that any custom nodes you make be added to the Nodes sub-folder here. This will prevent your custom work from potentially being lost of overridden when updating or re-installing uScript into your project.

Attributes

Both Action Nodes and Event Nodes share the same core set of attributes which can be used to provide the end user with helpful information.

AssetPathField

AssetPathField(*AssetType.assetType*) can be used to specify that the Asset Browser Window should be used to select the parameter's asset.

```
void Input([AssetPathField(AssetType.AudioClip)]
AudioClip clip)
```

Currently supported asset types are:

- AnimationClip
- AudioClip
- CubeMap
- Flare
- Font
- GUISkin
- Material
- Mesh
- MovieTexture
- PhysicMaterial
- Prefab
- RenderTexture
- Shader
- TextAsset
- Texture2D

DefaultValue

DefaultValue(*object value*) can be used on any property or parameter to specify default values.

```
void Input([DefaultValue(1.5f)] float volume)
```

Driven

Driven() can be used only for Action Nodes and only on methods. After the node is initially signaled [Driven] methods will automatically execute each tick. Returning false from a [Driven] method will turn off the automatic execution until the node is signaled again. Please see the Driven Tutorial for more information.

```
[Driven]
void UpdateEachTick(out float newValue)
```

FriendlyName

FriendlyName(*string name*) or FriendlyName(*string name, string description*) can be used on any property, parameter, method, return value or class to specify the corresponding visual name (and description) for the node. If FriendlyName is not specified the reflected C# value will be used.

```
[FriendlyName("Play Sound", "Plays a sound in Unity")]
public class uScriptAct_PlaySound : uScriptLogic

    [FriendlyName("Start Sound", "Starts playing
    the sound")]
    void Input(float volume)
```

NodeAuthor

NodeAuthor(*string name, string url*) is used to specify the node author's name and URL path. This information is displayed on various uScript node reference windows.

```
[NodeAuthor("uScript Team", "http://www.uscript.net")]
public class uScript_Input : uScriptEvent
```

NodeAutoAssignMasterInstance

NodeAutoAssignMasterInstance(*bool autoAssign*) is used on Event Nodes which should automatically be assigned to the uScript Master GameObject.

```
[NodeAutoAssignMasterInstance(true)]
public class uScript_Input : uScriptEvent
```

NodeDeprecated

NodeDeprecated(*Type upgradeToType*) can be used on any node you wish to deprecate and optionally replace with a new node type. If a new type is specified the user will have the option to upgrade the existing node and all compatible parameters will be transferred. Nodes with the [Deprecated] attribute will still function; however the user will not be able to place new instances.

Deprecating a node:

```
[NodeDeprecated()]
public class uScriptAct_PlaySound : uScriptLogic
```

Deprecating a node and specifying a replacement node:

```
[NodeDeprecated(typeof(uScriptAct_PlaySound2))]
public class uScriptAct_PlaySound : uScriptLogic
```

NodeDescription

NodeDescription(*string value*) is used to specify a description of your node's functionality. This information is displayed on various uScript node reference windows.

```
[NodeDescription("This is how my node works")]
public class uScript_Input : uScriptEvent
```

NodeHelp

NodeHelp(*string value*) is used to specify a help URL for your node. This information is displayed on various uScript node reference windows.

```
[NodeHelp("http://www.uscript.net/nodes/uScript_
Input.html")]
public class uScript_Input : uScriptEvent
```

NodeNeedsGuiLayout

NodeNeedsGuiLayout(*bool value*) is used to tell the code generation that GUILayout is required in the node. useGUILayout will be set to true if any node implements this

attribute with a true value. See the Unity documentation for more information on the Unity GUI system.

```
[NodeNeedsGuiLayout(true)]
public class uScript_Input : uScriptEvent
```

NodePath

NodePath(*string value*) is used to specify where the node appears in the Node Palette and uScript Context Menu. Paths can be delimited by forward slashes ('/').

```
[NodePath("Events/Input")]
public class uScript_Input : uScriptEvent
```

NodePropertiesPath

NodePropertiesPath(*string value*) is used to specify where the node's properties appears in the Node Palette and uScript Context Menu. Paths can be delimited by forward slashes ('/').

```
[NodePropertiesPath("Events/Input/Properties")]
public class uScript_Input : uScriptEvent
```

NodeToolTip

NodeToolTip(*string value*) is used to display any relevant tool tip information pertaining to your node.

```
[NodeToolTip("My ToolTip Information")]
public class uScript_Input : uScriptEvent
```

RequiresLink

RequiresLink() can be used to specify a link is required for this parameter and this parameter is not to be editable in the property grid.

```
void Input([RequiresLink] float volume)
```

SocketState

SocketState(*bool visible, bool locked*) can be used to specify the initial visual socket state and behavior for any property or parameter. Sockets can be visible or hidden and locked or unlocked. The default is SocketState(*true, false*).

```
void Input([SocketState(true, false)] float volume)
```

Creating Action Nodes

Writing a custom node is relatively straight forward. At its core you need to do only one thing: **Derive your class from uScriptLogic**

Now to get more specific: uScript looks at all the uScriptLogic classes and interprets them as visual nodes. Any public methods you have will be placed as sockets on the left side of the node. There is a limitation to be aware of: Only 1 parameter signature is allowed. This means all public methods of the same node must have the same argument signature.

Delay Node Example

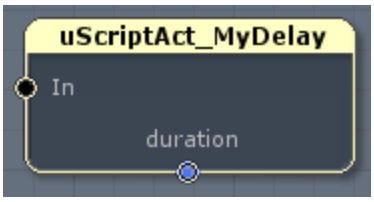
For our example we will create a Delay node. This node will accept a number of seconds (for a countdown), fire off an immediate output and then fire off a signal when the count down reaches 0.

To start, create a new c# script file in your project's *uScriptProjectFiles/Nodes* folder and call it "*uScriptAct_MyDelay.cs*". Then open the file for edit and delete any pre-placed template code with the following code:

```
using UnityEngine;

public class uScriptAct_MyDelay : uScriptLogic
{

    public void In( float duration )
    {
    }

}
```

> **Note!** - *Please remember that Unity currently needs to have the class name match the actual file name for the class!*

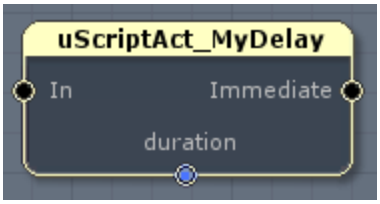
This code will be represented as a node with a single input, "In".



The first thing we want to do is to allow the graph to continue executing after it calls our In. uScript allows continual execution through public 'get' properties. Each public property you have will show up as an output socket on the right side of the node. At run time uScript will query that property to see if it can continue execution on that branch. For our Delay node we want execution to continue immediately so we add a public property which always returns true.

```
public class uScriptAct_MyDelay : uScriptLogic
{

    public bool Immediate { get { return true; } }

    public void In( float duration )
    {
    }

}
```

Now our node will have an In socket on the left and an Immediate socket on the right:



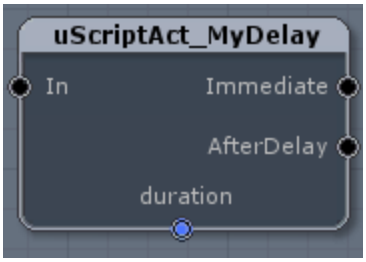
Next we want to fire an event when our countdown has reached 0. To do this we add an event handler and a public event.

```
public class uScriptAct_MyDelay : uScriptLogic
{

    public delegate void uScriptEventHandler
    (object sender, System.EventArgs args);
```

```
    public event uScriptEventHandler AfterDelay;

    public bool Immediate { get { return true; } }

    public void In( float duration )
    {
    }

}
```

Note! - *the current version of the uScriptLogic class does not allow a custom event argument, you must use System.EventArgs. If you require custom event arguments please consult the "Creating Event Nodes" topic.*

Now our node will have an In socket on the left, an Immediate socket on the right and an AfterDelay socket on the right:



Let's add the code which fires this AfterDelay event when the countdown reaches 0.

```
public class uScriptAct_MyDelay : uScriptLogic
{

    public delegate void uScriptEventHandler
    (object sender, System.EventArgs args);
    public event uScriptEventHandler AfterDelay;

    public bool Immediate { get { return true; } }
    private float m_TimeToTrigger;

    public void In( float duration )
    {
```

```
            m_TimeToTrigger = duration;
        }

        public void Update( )
        {
            if ( m_TimeToTrigger > 0 )
            {
                m_TimeToTrigger -=
                UnityEngine.Time.deltaTime;

                if ( m_TimeToTrigger <= 0 )
                {
                    if ( AfterDelay !=
                    null ) AfterDelay(
                    this, new
                    System.EventArgs());
                }
            }
        }
    }
```

Notice we added an Update method. This method is called every tick by uScript. The current list of automatically called functions for a uScriptLogic node are as follows: Update, LateUpdate, FixedUpdate, OnGUI. These are called by Unity which fall through to your custom node.

Congratulations! You've written your first uScript node. You can pretty it up and make it more user friendly with Attributes. Below is an example of the Delay node using attributes, please see the "Attributes" topic for a full list of node attributes and documentation.

The full node code and resulting node:

```
// uScript Action Node
// (C) 2010 Detox Studios LLC
```

```
// Desc: Delays the AfterDelay output signal by x
seconds.

using UnityEngine;
using System.Collections;

[NodePath("Actions/Time")]
[NodeCopyright("Copyright 2011 by Detox Studios LLC")]
[NodeToolTip( "Delays execution of a script.")]
[NodeDescription("Delays execution of a script but can
also fire off an immediate response.\n \nDuration:
Amount of time (in seconds) to delay.")]
[NodeAuthor("Detox Studios LLC",
"http://www.detoxstudios.com")]
[NodeHelp("http://uscript.net/manual/node_
delay.html")]

[FriendlyName("My Delay")]
public class uScriptAct_MyDelay : uScriptLogic
{

    public delegate void uScriptEventHandler
    (object sender, System.EventArgs args);
    private float m_TimeToTrigger;

    [FriendlyName("Immediate Out")]
    public bool Immediate { get { return true; } }

    [FriendlyName("Delayed Out")]
    public event uScriptEventHandler AfterDelay;

    [FriendlyName("In")]
    public void In( [FriendlyName("Duration")]
    float Duration )
    {

        m_TimeToTrigger = Duration;

    }

    public void Update( )
    {
```

```
            if ( m_TimeToTrigger > 0 )
            {

                m_TimeToTrigger -=
                UnityEngine.Time.deltaTime;

                if ( m_TimeToTrigger <= 0 )
                {

                    if ( AfterDelay !=
                    null ) AfterDelay(
                    this, new
                    System.EventArgs());

                }

            }

        }

    }
```

Self-Executing Action Nodes

Occasionally there is a need for nodes to drive themselves and push values out to the graph without being continually signaled by an external input. An example of this is an Interpolation node which should push out new interpolated values to the graph each tick.

You can use the [Driven] attribute to do just this. [Driven] tells uScript to call a method each tick (after an initial signal) and continue executing down the node chain as if it was signaled from an external node.

Float Interpolator Example

This example will show you how to create an self executing action node. Let's take our Float Interpolator as a simple example:

```
public class uScriptAct_InterpolateFloatLinear :
uScriptLogic
{

    ...

    public bool Started { get { return m_
    Lerper.AllowStartedOutput; } }
    public bool Stopped { get { return m_
    Lerper.AllowStoppedOutput; } }
    public bool Interpolating { get { return m_
    Lerper.AllowInterpolatingOutput; } }
    public bool Finished { get { return m_
    Lerper.AllowFinishedOutput; } }

    public void Begin(

        float startValue,
        float endValue,
        float time,
        uScript_Lerper.LoopType loopType,
        float loopDelay,
        int loopCount,
        out float currentValue

    )
    {

        ...

    }

    [Driven]
    public bool DoInterpolation(out float
    currentValue)
    {

        float t;
```

```
            bool isRunning = m_Lerper.Run( out t
            );

            if ( isRunning )
            {

                m_LastValue = Mathf.Lerp( m_
                Start, m_End, t );

            }

            currentValue = m_LastValue;

            return isRunning;

        }

    ...
    }
```

Notice the [Driven] attribute over the *DoInterpolation* method, uScript will call this each tick (after the initial signal to *Begin*). If true is returned it will push the *currentValue* parameter to the connected uScript local variable and then uScript will continue executing down the node chain following the output rules (*Started*, *Stopped*, *Interpolating*, *Finished*). If false is returned uScript will ignore *currentValue* and will stop automatically executing this method until a signal to *Begin* is sent.

[Driven] method signatures are dependent on the other method signatures. They must match the output parameters of other methods and not contain any input or ref parameters. In the above example; *DoInterpolation* has a single 'out float' which matches the *Begin*'s 'out float'. This is because input values are not pushed to the input parameters for self executing methods. If you require input parameters, you can save the values from the standard (non [Driven]) methods and use them within the [Driven] method.

Creating Event Nodes

Custom event nodes are useful if you want to signal execution of a uScript graph based on some logic in your game. For this tutorial we'll assume we want to execute a uScript graph when Unity tells us a key has been pressed.

In order for your events to be properly reflected you must derive your class from *uScriptEvent*. uScript looks at all *uScriptEvent* classes and interprets them as event nodes (i.e. nodes which can start the execution of a graph).

> **Note!** - *Event nodes cannot have any input sockets, their purpose is to start execution and not be inline of existing execution. If you require input sockets you may want to consider our action node tutorials found in the "Creating Action Nodes" topic.*

Input Event Node Example

In this example, we will create the Input event node. This node will execute whenever key input is detected by Unity.

Let's create our stub class (the file is called *uScript_MyInputEvents.cs*):

```
using UnityEngine;

public class uScript_MyInputEvents : uScriptEvent
{
}
```

Next you'll want to add an Event handler for every event you wish to dispatch. The events will appear as output sockets on the right side of the node. Please note: If you have multiple events, they must share the same System.EventArgs definition.

For our input node we will want to fire off a single event if a key is pressed or released:

```
using UnityEngine;

public class uScript_MyInputEvents : uScriptEvent
{
```

```
    public delegate void uScriptEventHandler
    (object sender, System.EventArgs args);

    public event uScriptEventHandler KeyEvent;

}
```

This code will be represented by a node with a single output on the right side and an Instance parameter on the bottom. The Instance parameter allows the user to specify which GameObject this node should be attached to (more on this at the end of the tutorial).

Finally, we'll want to add the logic which fires off the event. Event nodes are capable of listening to anything supported by MonoBehaviour, in this case we will use Update:

```
using UnityEngine;

public class uScript_MyInputEvents : uScriptEvent
{
    public delegate void uScriptEventHandler
    (object sender, System.EventArgs args);

    public event uScriptEventHandler KeyEvent;

    private bool m_AnyKeyWasDown = false;

    void Update()
    {
        if (Input.anyKey)
        {
            m_AnyKeyWasDown = true;
```

```
                if (KeyEvent != null) KeyEvent
                (this, new System.EventArgs
                ());

            }
            else if ( true == m_AnyKeyWasDown )
            {

                //no key is down now but it
                was the last frame
                //so send a key up
                m_AnyKeyWasDown = false;

                if (KeyEvent != null) KeyEvent
                (this, new System.EventArgs
                ());

            }

        }

    }
```

Congratulations! Your event dispatching is complete. However, to make it fully usable there are two special attributes we need to be concerned with (see details for all these attributes in the "Attributes"topic). The first is *NodeAutoAssignMasterInstance* and the second is *NodeComponentType*.

NodeAutoAssignMasterInstance is useful if you would like uScript to automatically assign this node to the *_uScript* Master GameObject. (*uScriptEvent* nodes derive from *MonoBehaviour* which means they must be attached to a GameObject in order to function, this is what the Instance socket is for). This is helpful for any global events (e.g. Input) because users will not be required to hook up the Instance socket.

```
[NodeAutoAssignMasterInstance(true)]
```

NodeComponentType tells uScript whichever GameObject this node is assigned to must also have a component of type added in order to function correctly. In the case of our Input node, there is no special component required so we can use the *Transform* type (all GameObjects in Unity have a *Transform* component).

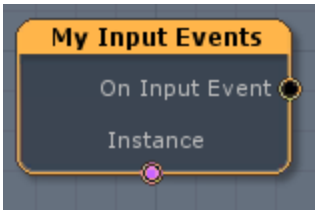
```
[NodeComponentType(typeof(Transform))]
```

*If we were doing a trigger, which requires a Collider component, we would instead use: [NodeComponentType(typeof(**Collider**))].*

Below is the full code with all of the attributes and a picture of the final node:

```
using UnityEngine;

[AddComponentMenu("uScript/Event Components/Input")]
[NodeAutoAssignMasterInstance(true)]
[NodeComponentType(typeof(Transform))]

[NodePath("Events/My Input Events")]
[NodeCopyright("Copyright 2011 by Detox Studios LLC")]
[NodeToolTip("Input Events fires out any time input is
detected from the keyboard, mouse, or joystick.")]
[NodeDescription("Input Events fires out any time
input is detected from the keyboard, mouse, or
joystick.")]
[NodeAuthor("Detox Studios LLC",
"http://www.detoxstudios.com")]
[NodeHelp("http://uscript.net/manual/node_
nodoc.html")]

[FriendlyName("My Input Events")]
public class uScript_MyInputEvents : uScriptEvent
{

    public delegate void uScriptEventHandler
    (object sender, System.EventArgs args);

    [FriendlyName("On Input Event")]
    public event uScriptEventHandler KeyEvent;

    private bool m_AnyKeyWasDown = false;

    void Update()
    {

        if (Input.anyKey)
        {
```

```
                m_AnyKeyWasDown = true;

                if (KeyEvent != null) KeyEvent
                (this, new System.EventArgs
                ());

            }
            else if ( true == m_AnyKeyWasDown )
            {

                //no key is down now but it
                was the last frame
                //so send a key up
                m_AnyKeyWasDown = false;

                if (KeyEvent != null) KeyEvent
                (this, new System.EventArgs
                ());

            }

        }

    }
```



Sending Custom Event Arguments Example

This tutorial will cover sending out custom event arguments with your custom event node. Make sure you're already familiar with making your own basic event node as covered in the *Input Event Node Example* above.

This event node will focus on listening for triggers and sending out an event with the 'Instigator' (the GameObect which interacted with the trigger) as an event argument.

As with the first tutorial we will derive from *uScriptEvent* (which is a *MonoBehavior* object) and then we'll implement the methods *OnTriggerEnter*, *OnTriggerExit*, and *OnTriggerStay* which will be called by Unity.

Let's stub out the node class, called *uScript_MyTriggers.cs*, to start:

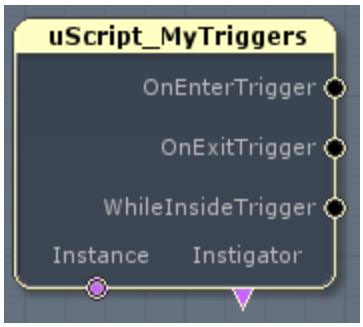
```
using UnityEngine;

public class uScript_MyTriggers : uScriptEvent
{
    public delegate void uScriptEventHandler
    (object sender, System.EventArgs args);

    void OnTriggerEnter(Collider other)
    {
    }

    void OnTriggerExit(Collider other)
    {
    }

    void OnTriggerStay(Collider other)
    {
    }

}
```

Next we will change the EventArgs to be a custom event argument class which will pass our Instigator on to the listeners. Also, we'll add the event handlers so the output events will show up on the right side of our node.

```
using UnityEngine;

public class uScript_MyTriggers : uScriptEvent
{
    public delegate void uScriptEventHandler
    (object sender, TriggerEventArgs args);

    public class TriggerEventArgs :
    System.EventArgs
    {
```

```
        private GameObject m_Instigator;
        public GameObject Instigator { get {
        return m_Instigator; } }

        public TriggerEventArgs(GameObject
        instigator)
        {
            m_Instigator = instigator;
        }
    }

    //event handlers
    public event uScriptEventHandler
    OnEnterTrigger;
    public event uScriptEventHandler
    OnExitTrigger;
    public event uScriptEventHandler
    WhileInsideTrigger;

    void OnTriggerEnter(Collider other)
    {
    }

    void OnTriggerExit(Collider other)
    {
    }

    void OnTriggerStay(Collider other)
    {
    }
}
```

Notice any trigger event arguments are represented visually as bottom output sockets on the event node:



Moving right along, lets fill our trigger arguments and dispatch the events:

```
using UnityEngine;

public class uScript_MyTriggers : uScriptEvent
{
    public delegate void uScriptEventHandler
    (object sender, TriggerEventArgs args);

    public class TriggerEventArgs :
    System.EventArgs
    {
        private GameObject m_Instigator;
        public GameObject Instigator { get {
        return m_Instigator; } }

        public TriggerEventArgs(GameObject
        instigator)
        {
            m_Instigator = instigator;
        }
    }

    //event handlers
    public event uScriptEventHandler
    OnEnterTrigger;
    public event uScriptEventHandler
    OnExitTrigger;
```

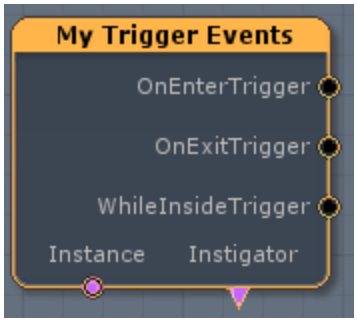
```
    public event uScriptEventHandler
    WhileInsideTrigger;

    void OnTriggerEnter(Collider other)
    {
        if ( OnEnterTrigger != null )
        OnEnterTrigger( this, new
        TriggerEventArgs(other.gameObject) );
    }

    void OnTriggerExit(Collider other)
    {
        if ( OnExitTrigger != null )
        OnExitTrigger( this, new
        TriggerEventArgs(other.gameObject) );
    }

    void OnTriggerStay(Collider other)
    {
        if ( WhileInsideTrigger != null )
        WhileInsideTrigger( this, new
        TriggerEventArgs(other.gameObject) );
    }
}
```

That's it! Now you know how to create custom event nodes which dispatch custom event arguments. See below for the full code with attributes:

```
using UnityEngine;

[AddComponentMenu("uScript/Event Components/Trigger")]
[NodeComponentType(typeof(Collider))]

[NodePath("Events")]
[NodeCopyright("Copyright 2011 by Detox Studios LLC")]
[NodeToolTip("Fires an event signal when a GameObject
enters, exits, or stays in a trigger.\n \nInstigator:
```

```
The GameObject that interacted with the trigger
(Instance).")]
[NodeDescription("Fires an event signal when a
GameObject enters, exits, or stays in a trigger.")]
[NodeAuthor("Detox Studios LLC",
"http://www.detoxstudios.com")]
[NodeHelp("http://uscript.net/manual/node_
nodoc.html")]

[NodePropertiesPath("Properties/Triggers")]
[FriendlyName("My Trigger Events")]
public class uScript_MyTriggers : uScriptEvent
{

    public delegate void uScriptEventHandler
    (object sender, TriggerEventArgs args);

    public class TriggerEventArgs :
    System.EventArgs
    {

        private GameObject m_Instigator;

        [FriendlyName("Instigator")]
        [SocketState(false, false)]
        public GameObject Instigator { get {
        return m_Instigator; } }

        public TriggerEventArgs(GameObject
        instigator)
        {

            m_Instigator = instigator;

        }

    }

    //event handlers
    public event uScriptEventHandler
    OnEnterTrigger;
    public event uScriptEventHandler
    OnExitTrigger;
```

```
    public event uScriptEventHandler
    WhileInsideTrigger;

    void OnTriggerEnter(Collider other)
    {

        if ( OnEnterTrigger != null )
        OnEnterTrigger( this, new
        TriggerEventArgs(other.gameObject) );

    }

    void OnTriggerExit(Collider other)
    {

        if ( OnExitTrigger != null )
        OnExitTrigger( this, new
        TriggerEventArgs(other.gameObject) );

    }

    void OnTriggerStay(Collider other)
    {

        if ( WhileInsideTrigger != null )
        WhileInsideTrigger( this, new
        TriggerEventArgs(other.gameObject) );

    }

}
```



Adding Drag Drop Support To Nodes

If you would like to allow the user to drag/drop items on to your custom Action Node, you can override the *EditorDragDrop* method. EditorDragDrop will be called when an object is requested to be dropped. If you wish to accept this object (or anything this object might reference) as one of your input parameters simply return a Hashtable with the key being your input name and the value being the input parameter.

Here is an example for the Play Sound node which can accept an AudioClip and a GameObject:

```
#if UNITY_EDITOR

    public override Hashtable EditorDragDrop(
    object o )

    {

        if ( typeof
        (AudioClip).IsAssignableFrom(o.GetType
        ()) )
        {

            AudioClip ac = (AudioClip) o;

            Hashtable hashtable = new
            Hashtable( );
            hashtable[ "Audio Clip" ] =
            ac;

            return hashtable;

        }
        if ( typeof
        (UnityEngine.GameObject).IsAssignableF-
        rom(o.GetType()) )
        {

            GameObject go = (GameObject)
            o;

            Hashtable hashtable = new
            Hashtable( );
```

```
            hashtable[ "Target" ] =
            go.name;

            return hashtable;

        }
        return null;

    }

#endif
```

Modifying Existing Nodes

If you wish to add or remove functionality to an existing custom node without replacing (or deprecating) it then you can simply modify the node's code. Depending on the changes, it could cause compiler errors with previously generated uScript code.

To fix any compiler errors:

1. Open up the uScript Editor.
2. Select '*Rebuild All uScripts*' from the *File Menu*. This will regenerate all of your uScript code.

> **Note!** - *If you removed any Inputs, Outputs or Parameters; any links to these sockets will be properly removed when the uScripts are loaded.*

Deprecating Nodes

If you wish to mark a custom node as deprecated (i.e. should no longer be used) add the *NodeDeprecated* attribute. Deprecated nodes still function but can no longer be placed by a user and will render in uScript as hot pink.

If you wish to have a new node replace the deprecated node, add the new node type in the deprecated arguments (e.g. `[NodeDeprecated(typeof(uScriptAct_MyNewNode))]`). This will give the user the option of upgrading the deprecated node to the new node type.

See "Attributes" on page 6

References

uScript User Guide - Provides more general information on using uScript.

uScript Forum - Join our forum to have the uScript developers and community help you with specific problems.

Unity Script Reference - See this documentation for general information on programming in the Unity environment.