uSCRIPT

Visual Scripting Tool

Closed Beta Quick Start Guide

v1.3

Detox Studios

Table of Contents

Disclaimer

Please be aware that this is beta software! **You should not rely on this software for actual production work at this time.** As we update the software through the beta process, you should expect to potentially loose work you have done with uScript.

We also do not recommend adding uScript to an existing project you may currently be working on without first backing that project up BEFORE installing uScript.

uScript has been designed for Unity 3 and will not work in earlier versions.

Installation

The uScript Visual Scripting Tool is an editor tool for version 3 of the Unity engine. Please follow these steps to install uScript

1. Download and unzip the **uScript_ClosedBeta.zip** file to a preferred location on your computer.

2. Run Unity 3 and select Import Package/Custom Package from the "Assets" menu and browse to the **uScript.unitypackage** file to install uScript:
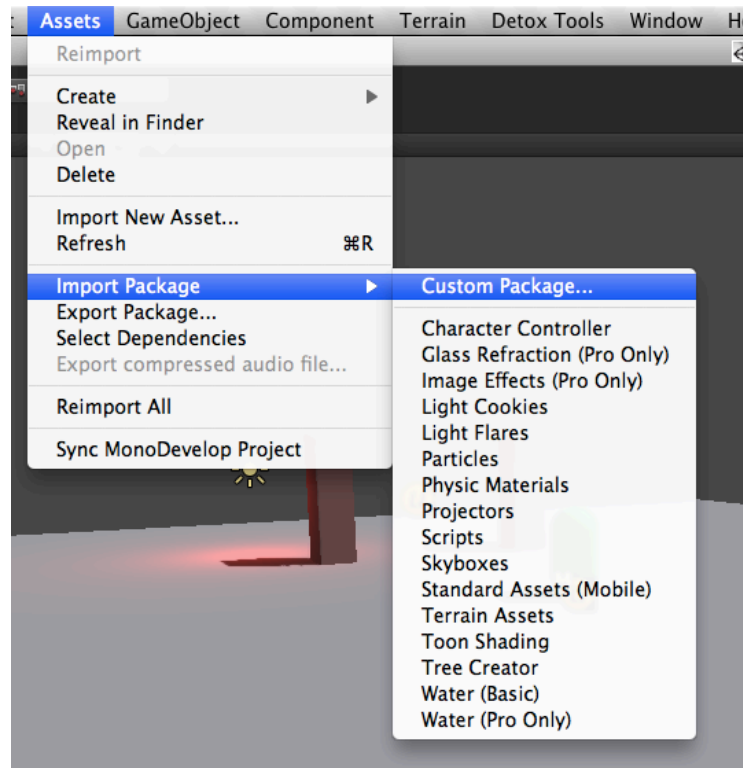
3. When the package window comes up, make sure everything is selected and choose "Import".

4. Once uScript has installed, you should now have a uScript folder in the root of your Unity projects Assets folder. You can run the tool by going up to the "Detox Tools" menu in Unity or by pressing Control+U (or Command+U on Mac).

a. *Note: there is a bug in Unity that may prevent this menu from showing up initially. Just click on any other menu to force Unity to refresh its menus.*

5. The first time you run uScript, it will create an "uScript Master GameObject" in your scene. This is needed for uScript to run. You will see this GameObject's uScript Gizmo icon appear in your scene:

uScript Key Concepts

The following are key concepts regarding uScript that will help you understand how uScript works and to make the most out of it. Many of these things are requirements to work with the way Unity does things.

uScript Master Object

uScript needs to make a GameObject for itself that contains some key uScript components. The default name for this GameObject is '_uScript'. For your uScript to work in your scene, it must be assigned to the object! If it was not assigned automatically by uScript when you first save your file, you can drag the C# script (NOT the one with SubSeq_ in the name!) onto this object.

Output Files

uScript currently generated three files for each uScript "graph" you create. Let's assume you create a new uScript called "MyGreatGame"—these are the files uScript would generate:

- **MyGreatGame.uscript** – This is the "master" binary file that uScript uses to create your uScript. You don't want to delete this file (unless you want to actually delete the uScript permanently). This file can generate/re-generate the other two files below. Sometimes you will want to do this when uScript (or a node) has been updated and you receive warnings or errors.

5

- **MyGreatGame.cs** – This is a "wrapper" output C# script file. This is the file that would be assigned to the uScript master GameObject (see above) in order to have your uScript run in the scene. This file is automatically built/rebuilt by uScript from the .uscript binary file above.

- **SubSeq_MyGreatGame.cs** – This file is where all the "magic" is. This is the file that contains all your uScript graph as pure C# script. This file can also be used in uScript as a Subsequence like in Unreal's Kismet. These files are actually more powerful/flexible than a Subsequence and a better term (and one we will be switching to) would be "nested uScripts". This file is automatically built/rebuilt by uScript from the .uscript binary file above.

uScript Event Components

Some event types are triggered by a specific object and so a specific uScript component needs to be added to that object. As an example, if you plan to use a GameObject as a trigger and you want to fire off uScript logic based on that trigger, you need to make sure that GameObject has a *uScript_Triggers.cs* component assigned to it. In fact, if you try to assign a GameObject as an instance (see below) of a Trigger event in uScript, it will not let you assign it if the uScript_Triggers.cs script component has not been assigned to that GameObject.

More global-based event components are just assigned to the uScript master GameObject—like for OnGameStart (*uScript_Global.cs*) and OnKeyPress (*uScript_Input.cs*) event nodes.

Instances

Many of uScripts events require you to sign a specific GameObject to it so uScript knows what object you want to fire that event. Using the Trigger example again, if you have place a Trigger event node, you need to tell uScript what trigger specifically you want to make that event fire (you could have hundreds in your scene!). This is done by assigning an instance (specific GameObject) to the property of the event. Again, as mentioned above, that GameObject will also need to have the proper component assigned to it so uScript will be able to listen for when the event occurred (or think of it as the GameObject telling uScript when something relevant has happened to it).
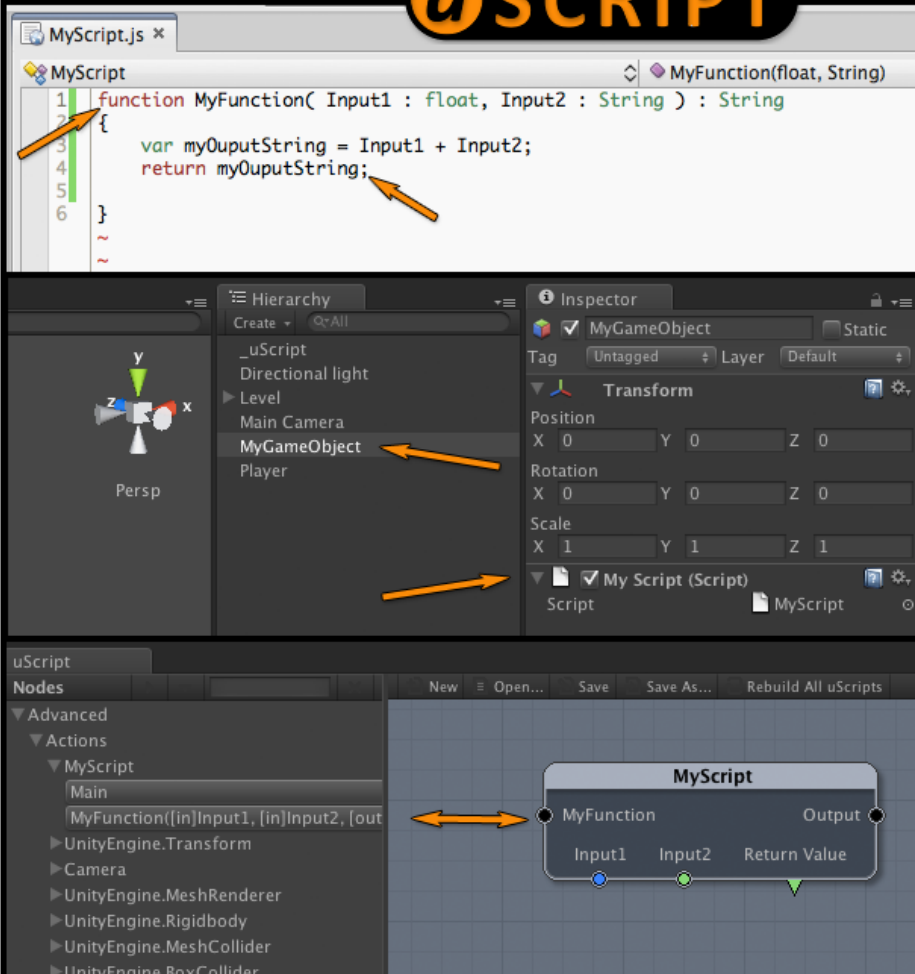
Many things in reflection (see below) also require you to assign an instance in order to tell uScript exactly what GameObject you wish to use.

Reflection

A feature for the more technical, uScript will use dotnet reflection in order to visualize many aspects of your existing scripts in Unity. It is important to note though that uScript can only see scripts that are assigned to GameObjects/active in the scene. uScript will reflect any public Actions (methods/functions), Properties (Properties/Public Variables exposed to the Unity inspector – both Get and Set), and Variables (all variable types). You must assign an instance (see above) to them so uScript knows what GameObject you want to use.

Reflection example in uScript

www.detoxstudios.com

Known Issues

Undo/Redo Hot Keys Not Working

uScript is not currently using the standard Undo/Redo hot keys. Please use the Right-Click context menu to access Undo/Redo for now.

Uniquely Named GameObjects

uScript uses the name of GameObject s in order to find them. We cannot rely on Unity's unique ID number system, as Unity itself does not keep this information once Unity is closed—it will assign new unique IDs the next time you run the editor. This means that uScript cannot store unique IDs for GameObjects in it's own save files—they would be useless the next time you run Unity.

This means you should name GameObjects something unique if you plan to use them/reference them with uScript to ensure uScript is using the correct GameObject. Unfortunately we will not be able to fix this issue until the Unity editor itself supports maintaining unique IDs between Unity sessions.

Missing Event Nodes Section

When you first run uScript, the "Events" section is not available. If this happens, just close uScript and re-open it.

Missing Detox Tools Menu

When first installing uScript, the "Detox Tools" menu in Unity may not appear. Just click on any other menu item to force Unity to update its menus and you should then see the Detox Tools menu appear.

uScript Window Small On First Run

You may notice that the uScript window is small when you first run it. You should just be able to resize the window and order to agree to the Beta EULA and start using uScript. uScript should remember your window size/position information after the first run.

uScript Gizmos

Sometimes Gizmos will stay on a GameObject that uScript is no longer using/referencing. This will be fixed in a future release.

Box Connection Lines

You cannot currently just select connection lines by using the box select method.

Hot Keys

Note: On Mac, replace Control (CTRL) with the Command (CMD) key.

General

uScript supports some standard hot key combinations. More are coming online all the time. Hot keys for things such as Load/Save and Undo/Redo are not yet implemented. The following are known to be working currently.

Copy – CTRL+C

Paste – CTRL+V

Cut – CTRL+X

Undo – Not yet supported. Use Right-Click context menu for access to undo/redo.

Redo – Not yet supported. Use Right-Click context menu for access to undo/redo.

Canvas

uScript currently uses a **non-destructive method** for node manipulation on the canvas. What this means is that the control layout is designed to prevent you from accidently moving nodes around while panning the canvas.

Canvas Panning – Left-Click anywhere on the canvas to pan/scroll the canvas view (including on a node).
Node Selection – Left-Click on a node to select it (it will turn a light yellow).

Multi-Select – CTRL+Left-Click to multi-select nodes and connection lines.

Remove from Multi-Selection – CTRL+ALT+Left-Click on a selected node to remove it from the current selection.

Box Select – CTRL+ALT+Left-Click Drag to create a selection box.

Context Menu – Right-Click to bring up a context menu with options regarding what you Right-Clicked on.

Move Nodes – Shift+Left-Click to move/drag nodes on the canvas.

Delete – Press DEL to delete selected items on the canvas. On Macs without a delete key you can use the *fn+delete* key combination to activate delete.

Setting Up Your First uScript

Coming Soon! For now we recommend you watch the videos we have posted showing the use of uScript.

YouTube: http://www.youtube.com/user/uScriptTool

Screencast: http://www.screencast.com/users/uScript/folders/uScript/

Creating Your First Action Node

Writing a custom node is relatively straightforward.

At its core you need to do only one thing:
1. 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 right 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.

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.
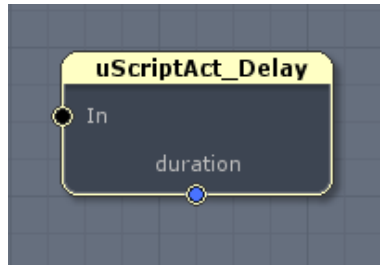
Let's create our stub:

```
using UnityEngine;

public class uScriptAct_Delay : uScriptLogic
{
   public void In( float duration )
   {
   }
}
```

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 they've called our In. uScript allows continual execution through public 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_Delay : 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.
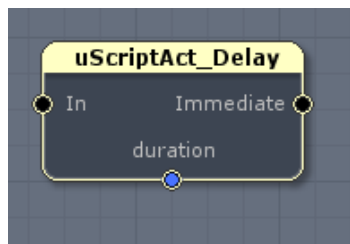
Next we want to fire an event when our countdown has reached 0. To do this we add an event handler and a public event.

```
public class uScriptAct_Delay : 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 )
    {
```
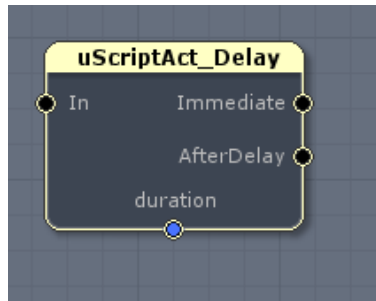
```
        }
    }
```

Please note, the current version of the uScriptLogic class does not allow a custom event argument, you must use System.EventArgs.

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_Delay : 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 override 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 falls through to your custom node.

Congratulations! You've written your first uScript node. Now let's pretty it up with some attributes to help your end user take full advantage of your node.

There are multiple class attributes available to you. Below I've added them to our Delay node, and I will follow with a brief explanation:

```
[NodePath("Action/Misc")]
[NodeLicense("http://www.detoxstudios.com/legal/eula.html")]
[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.")]
[NodeAuthor("Detox Studios LLC", "http://www.detoxstudios.com")]
[NodeHelp("http://uscript.net/manual/node_delay.html")]

public class uScriptAct_Delay : uScriptLogic
{
...
```

- **NodePath** is where this node will show up in the menu options.
- **NodeLicense** and **NodeCopyright** aren't currently used but will be leveraged in the future to display important licensing information pertaining to your node.
- **NodeToolTip** isn't currently used, but will be for tool tips pertaining to your node.
- **NodeDescription** shows a brief description in the uScript Reference Panel.
- **NodeAuthor** isn't currently used but will be shown in our uScript Reference Panel.
- **NodeHelp** is used by the Online Reference button in the uScript Reference Panel to open a web link.

Finally this brings us to the **FriendlyName** attribute. This attribute can be used with any of your parameters, properties and class names to give 'user friendly' text to your node. Anywhere you use a FriendlyName attribute, uScript will replace the parameter name with your FriendlyName text.

We can take full advantage of this with our Delay node.

```
[FriendlyName("Delay")]
public class uScriptAct_Delay : 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
    )
...
```


Congratulations! You have just made your first uScript node more user friendly.

Please feel free to look at our nodes which ship with uScript for further ideas on how to extend uScript's functionality.

Glossary of Terms

uScript uses the following terminology:

Node – Sometimes referred to a logic blocks, they are visual block on the canvas that has input and output sockets and, optionally, bottom variable sockets. There are two major types of nodes.

14

Event Node – An orange node that is used to fire off based on a specific event happening (like on game start, a key is pressed, or when a trigger is entered). These nodes are always the start of a chain of node logic and do not have an input socket like other nodes.

Action Node – Also known as a logic node, these nodes perform actions/logic when fired. They usually have bottom variable sockets for inputting or outputting variables. They may also have multiple input and output sockets depending on the nodes purpose.

Socket – Sockets are the small connection points on nodes that connection lines draw to. Input sockets (always found on the left/right of nodes) are round with black centers, while variable sockets (always found on the bottom edge of nodes) are either round if they input a variable value or a triangle if they output a variable value. Variable socked centers are color coded to match the variable type they take (string, int, float, bool, etc.)

Connection Lines – Connection lines are the lines that are drawn between the nodes.

Canvas – the section of the uScript Tool where the visual programming graph is located.

Graph – sometime used to describe an entire uScript visual script or a uScript script (.cs) output file.