Documentation version: ***1.0 RC2***

Introduction

Welcome to the uScript User Guide! This is the official manual for the uScript Visual Scripting Tool for Unity.

This section of the user guide covers material on how to install uScript, familiarize yourself with general visual scripting concepts, and provide some general advice you should be aware of before using uScript for the first time.

> **Mobile Users** - you can view a mobile formatted version of this documentation here.

uScript Versions

Please note that this documentation covers all versions of uScript. This includes the *Personal Learning Edition* (PLE), *Basic Edition*, and *Professional Edition*. While all three versions are very similar in their functionality and feature set, there is one exception. The reflection features documented here are for use in the Professional and PLE editions only. If you are interested in upgrading to uScript Professional from the Basic edition, please contact Detox Studios directly at http://www.detoxstudios.com.

Welcome To uScript

Thank you for your interest in the uScript Visual Scripting Tool for Unity 3! uScript allows you to visually create program logic that allows you to bring your ideas to life.

Created by Detox Studios, a company founded by veteran video game developers, uScript was created based on many years of development experience of previous game engines and visual scripting systems used on award-winning video games. It was designed to empower non-technical developers such as game designers, artists, architects and others to directly create runtime programming code visually and to iterate on ideas directly. It is our belief that to be competitive and innovative, it is vital to put the power of creation and iteration directly into the hands of the creative.

How Does It Work?

uScript allows you to listen for events happening in your game and to visually create logic to run when those events happen. The conditions which trigger those events and the complexity of the logic that is run depends purely on what you wish to do. Examples of this might include turning off a light when the player enters a trigger, creating a weapons and ammunition mechanic, moving the player, setting up a working door prefab, or even creating and managing an entire custom inventory system.

Goals

uScript was developed to be production-ready with four major goals in mind:

- **Usability** – It was our number one goal to take something as complex as programming game logic, and visualize it as best as possible in order for non-technical people to be able to do what they want without having to worry about technical aspects getting in their way. This is a very challenging problem as the more technical aspects you hide the more flexibility is lost. We spent a massive amount of time making uScript the easiest to use solution available while keeping the flexibility to do what you want as you become more advanced with the tool. We continue to put a lot of our development efforts into constantly improving this aspect of the product, including product features, documentation, and video tutorials.
- **Iteration** – Any developer worth their salt knows that the best ideas (and products) are ones that can be quickly iterated on in development. The flexibility to quickly try new ideas without fear is critical to inspiring innovation and being competitive. uScript allows you to visualize your ideas quickly and try different things in a very rapid, non-destructive development environment. Also, because uScript allows gameplay logic to be created by other team members besides programmers, it

greatly increases productivity through parallel iteration and experimentation. This frees up a team's engineers to focus on technical innovation and competitive advantages instead of implementing (and re-implementing) product design concepts.

- **Expandability** – Because every product has unique needs and development pipelines, it would be impossible for a visual scripting system to handle 100% of all needs "out of the box". Because of this, uScript was designed to be extremely customizable and extensible for projects big and small. This extensibility is accomplished through several keys features including the ability to create custom visual nodes (logic blocks) using C# code or even through visual scripting with uScript itself (via nested uScripts/recipes), customizable editor UI and full ".Net Reflection" of Unity's objects, components, properties, and variables.
- **Community** – Because of uScript's ability to be customized and expanded in many ways, it was important to us that its users had a place to go to share their nodes, creations, knowledge, and ideas. We also wanted to have a way to share uScript news and updates, and have open two-way dialog with our users regarding uScript. In support of that goal, we have created a community portal (http://uscript.net) for uScript users and future users.

We hope you enjoy uScript as much as we do!

Setting Expectations

A powerful visual scripting system consists of nodes (the building blocks of making a visual script) that do something very small and specific such as add numbers together, move the location of an object, or play a sound. You then use these simple nodes in combination in creative ways to create very complex and powerful logic specific to your project's needs. Your creativity and knowledge are the only limiting factors.

Some users may expect a single logic node that encapsulates a bunch of functionality into it to make their game for them (like a “Make FPS” or “Make Monster Wander” node). The problem with this approach is that it is not feasible to create such “high level” nodes as every game is unique- sometimes in big ways and sometimes in subtle ways. Whatever the case, if we made complex logic nodes, it would greatly limit the flexibility of uScript to allow users to experiment and be creative in their game development as they would be stuck with the functionality built into these "mega-nodes". The purpose of uScript is not to make your game for you-- it is to empower you to make the game you wish to make!

We have put a massive effort into designing uScript to be as easy as possible for non-technical users, but using any visual scripting tool still requires some basic understanding and knowledge by the user of software/game development, math & physics, and game design sensibilities in order to create some of the more advanced visual scripts. If you are not familiar with game development and the Unity technology, you will still have a more difficult time creating your product than those with a greater set of experience and knowledge in these matters. The key is to be sure you take the time to fully understand the tools you are using in order to make them work well for you.

Prerequisites

To use uScript and get the most out of this powerful tool, there are both required and recommended prerequisites:

Required

- **Unity** – uScript works with both the free and pro versions of Unity 4 (and also Unity 3.5.7). You need to have the latest version of Unity installed in order to install and use uScript.
- **uScript** – you of course need to install uScript in order to use the tool. The uScript editor is accessed from within Unity.

Recommended

- **Unity Basics** – You should be familiar with the basic use of Unity. If you have not used Unity before, we recommend you read their documentation and become familiar with the Unity pipeline, development philosophy, and terminology. Specifically, understanding what the following things at the very least will greatly help you in using uScript:
    - GameObject - the foundation object in which most things in Unity derive from.
    - Component - this is how you turn simple, empty GameObjects into amazing objects that can do specific things in your game.
    - Prefab - a great way to to manage and modify complex GameObjects in your game.
    - Scene - this is how you break you game apart into pieces that can be loaded into memory when you need them..
    - Resource Folder - this is how Unity streams assets at runtime.
- **Scripting Basics** – While uScript allows you to visual create scripts for your product without needing to know the syntax of a programming language, it is recommended that you have a basic understanding of how programming works. We have created the Scripting Basics section to give you a primer in some of the basic concepts of scripting.
- **Math & Physics** – Many of the things you may want to do will involve various math and physics to accomplish. While uScript contains the low level nodes needed to perform these complex math operations, an understanding of how to utilize these nodes to get the results you are looking for will be helpful.

Installing & Updating

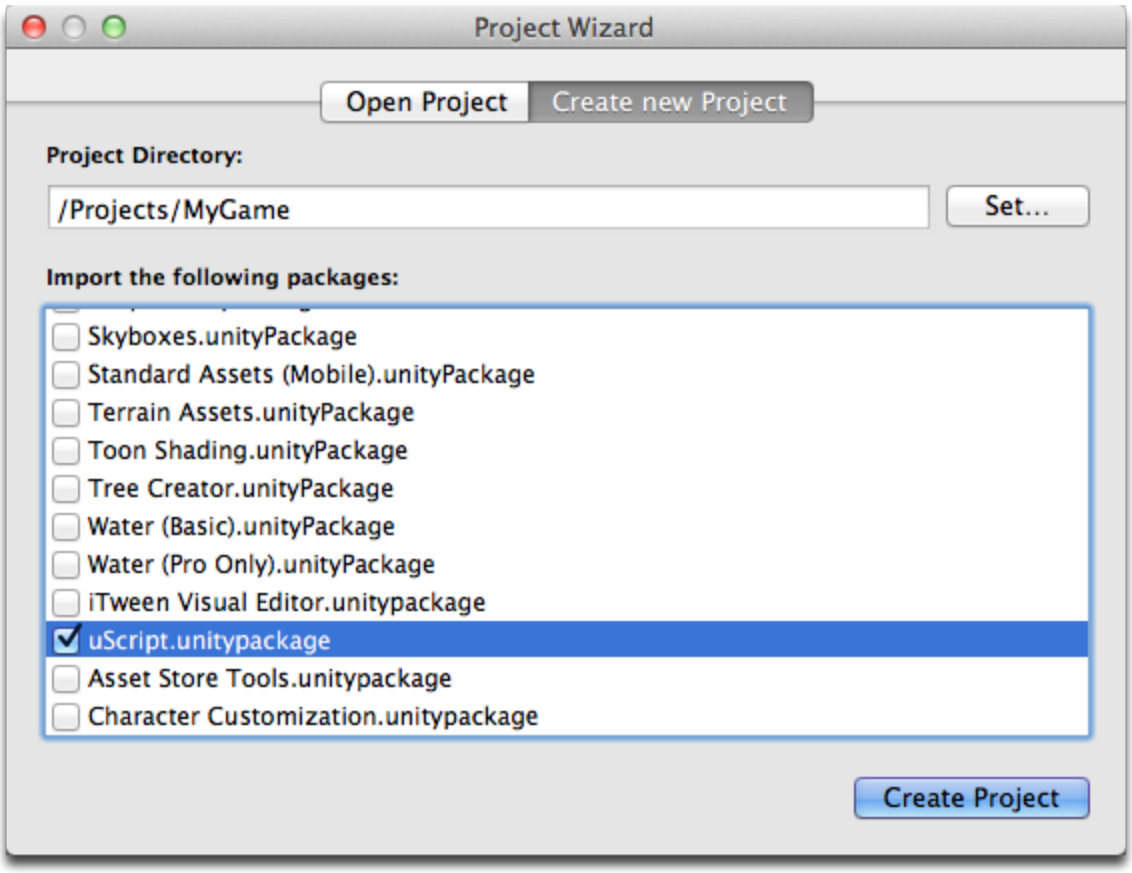
The following section explains how to install or update uScript into a Unity project.

> **IMPORTANT!** - Currently, uScript is officially supported on Unity 3.5.7 and Unity 4.x
>
> ***Note!*** *- If installing or updating uScript into an existing Unity project, we always* ***strongly*** *recommend that you backup your project files first using your preferred backup method!*

Installation

Like other Unity editor tools, uScript needs to be installed in each Unity project you wish to use it in. There are two common ways to install uScript depending on how you purchased it. Both ways are described below.

> ***Unity Asset Store***
> If you have purchased uScript from the Unity Asset Store, you can install uScript into any new project in one of two ways:
>
> **Method 1 - Import the uScript package on project creation**
>
> If you have already purchased uScript through the Unity Asset Store, you can choose to have uScript installed when creating a new Unity project. Just select it from the list in the project creation wizard:



> **Note!** - *If using this method to install uScript into a new project, you should still check to see if you have the latest version of uScript installed. The package that is installed using this method will have been the last version you got from the Asset Store, and may not be the latest version available.*

Method 2 - Import through the Unity Asset Store

You can install uScript at any time into a new or existing Unity project by using the Unity Asset Store window in the Unity editor. You can bring up the Asset Store window from inside the Unity editor. It is located under the Window menu in Unity (*CTRL+9* or *CMD+9* on Mac).

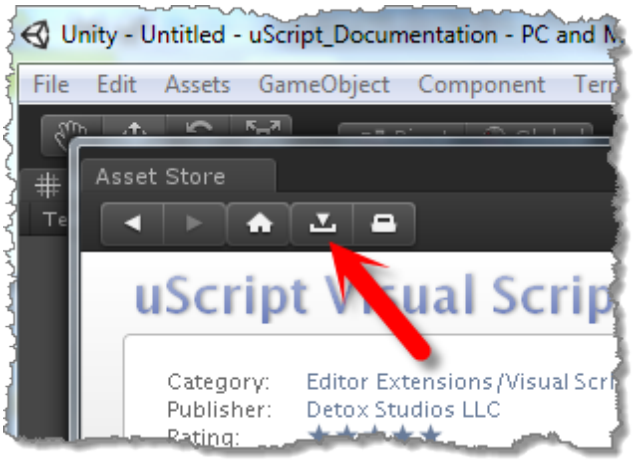
> **Note!** - *Either of the methods shown below can also be used to update an existing install of uScript for a project. If you are able to update an existing uScript install, the "Import" buttons in the*

examples below will instead show as "Update" when a new version of uScript is available for updating.

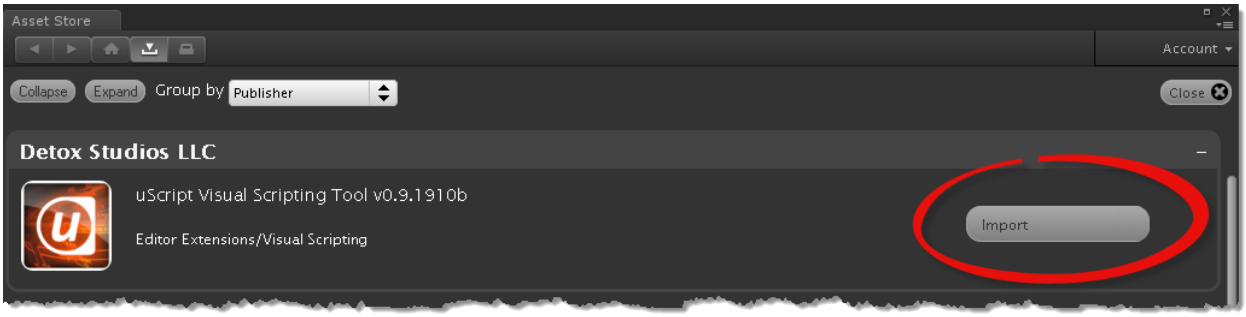
There are two methods to install uScript through the asset store:

Method 1 - Download Manager

Once the Asset Store window appears, you can bring up the Download Manager by selecting this button:



Then, once in the Download Manager, select the "Import" button for uScript:



Method 2 - Asset Store uScript Page

Optionally you can just browse to the uScript store page and select the "Import" button there as well:

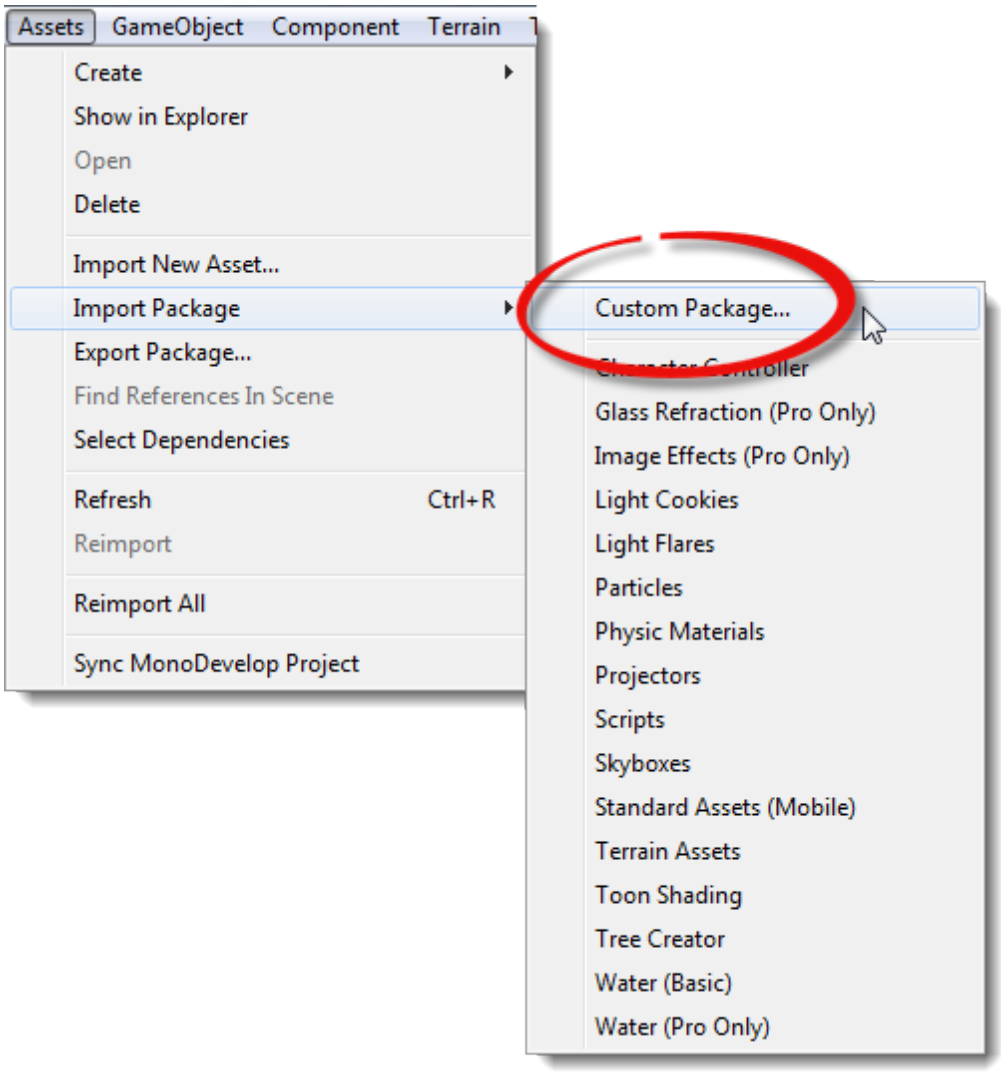
uScript Package Installation

If you have purchased (or downloaded the PLE version) directly from Detox Studios, you will be provided with a Unity installation file for installing uScript.

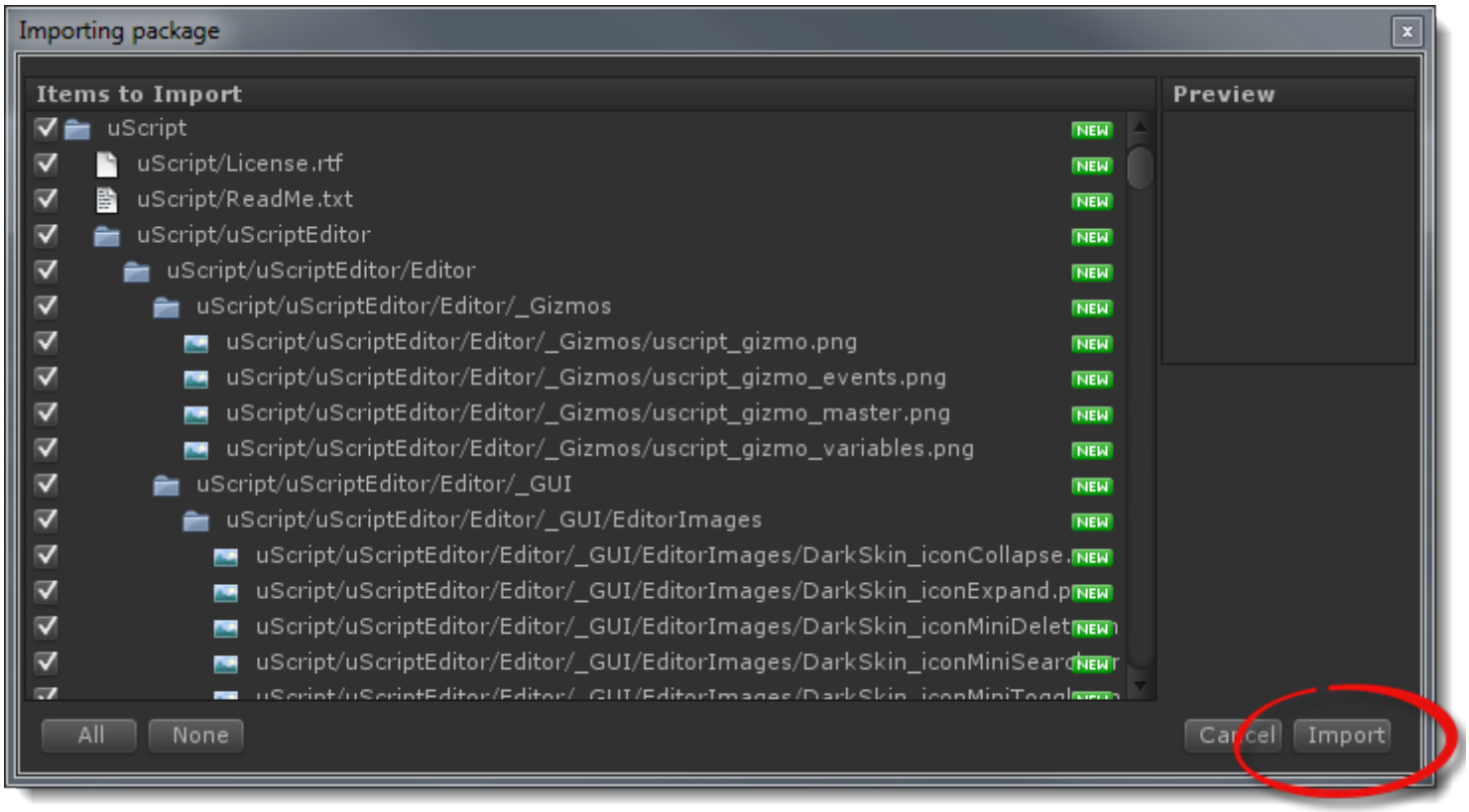
uScript uses the standard Unity package format (.uscriptpackage) and can be installed like any other Unity package file. Please follow the these instructions to install uScript directly from the package file (*uScript.unitypackage* or *uScript_PLE.unitypackage*):

- Make sure you have downloaded and unzip the latest version of uScript from ***http://www.detoxstudios.com/products/uscript/download***:

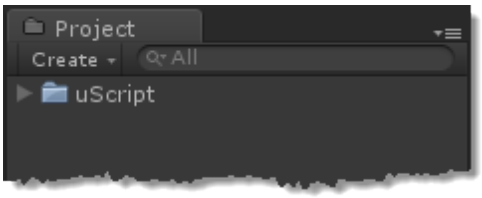
- Run Unity and select *Import Package/Custom Package...* from the “Assets” menu and browse to the uScript.unitypackage file to install uScript:



- When the Importing package window appears in Unity, make sure everything is selected and click the "Import" button:



- Once uScript has installed, you should now have "uScript" folder in the root of your Unity project's "Assets" folder:



Updating uScript

If you are updating or re-installing uScript into a Unity project where uScript was already being used, it is recommended that you perform a clean re-install of uScript as detailed below. Unity's package file system does not currently support the ability to remove obsolete files from a previous install-- it can only update or add files. This means if you do not remove the old version of uScript before installing the new version, you may be left with legacy files that could potentially cause issues in using the new version of uScript.

Note! - *Please be sure you have fully backed up your Unity project before attempting any updates.!*

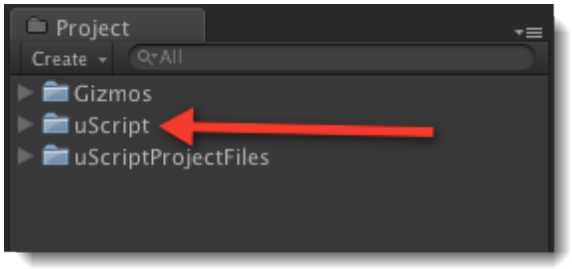
Performing a Clean Re-Install of uScript

A clean re-install is when you delete the root uScript folder before importing the latest *uScript.unitypackage* file into your Unity project.

This is the recommended way to update uScript because we may have changed or deleted old files and Unity's package system will not remove files when installing a new package over an old one. This can lead to old or conflicting files that can cause issues.

Please follow these steps to do a clean re-install:

- Load your Unity project(s) using uScript and delete the root "uScript" folder in your "Project" tab by right-clicking it and selecting "Delete" from the context pop-up menu. This is best done with a blank/new Unity scene open instead of an existing scene you may have uScript graphs in. Just select "New Scene" from the Unity's "File" menu-- there is no need to save this blank scene after you re-install uScript:



IMPORTANT! - *Do not delete your **uScriptsProjectFiles** folder! This is where your project's uScript graphs and any custom nodes are located. They will be lost if you delete this folder.*

- You can now re-install uScript. See "Installing & Updating" in this section for details.

Fixing Compile Errors After Updating

If you get compile errors from Unity after upgrading uScript, you most likely need to delete your generated uScript code files and re-generate them. This can happen when

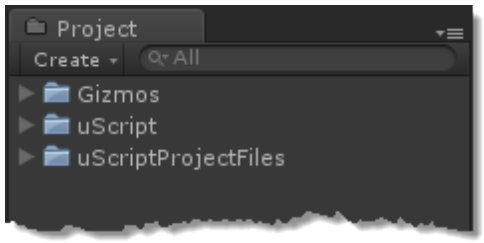
uScript's nodes have been updated to fix bugs or support new features. See "Script Generation" for more information on deleting and regenerating the code files.

Running uScript For The First Time

Intro Text Here

- You can run the uScript Editor by going up to the “Tools/Detox Studios” menu in Unity and selecting “uScript Editor” or by pressing *CTRL+U* (or *CMD+U* on Mac):

- After running the uScript editor for the first time, uScript will add the following items to your project:
  - some uScript gizmos to your projects *Gizmos* folder. If the Gizmos folder does not exist, uScript will create it.
  - a *uScriptProjectFiles* folder in the root of your project's *Asset* folder. This folder is where your uScript project-specific files will go such as; your uScript graphs, generated script files and any custom nodes you wish to add to uScript for this specific Unity project.
  - a GameObject to your open Unity scene called *_uScript*. This is also referred to throughout the User Guide as the uScript Master GameObject.

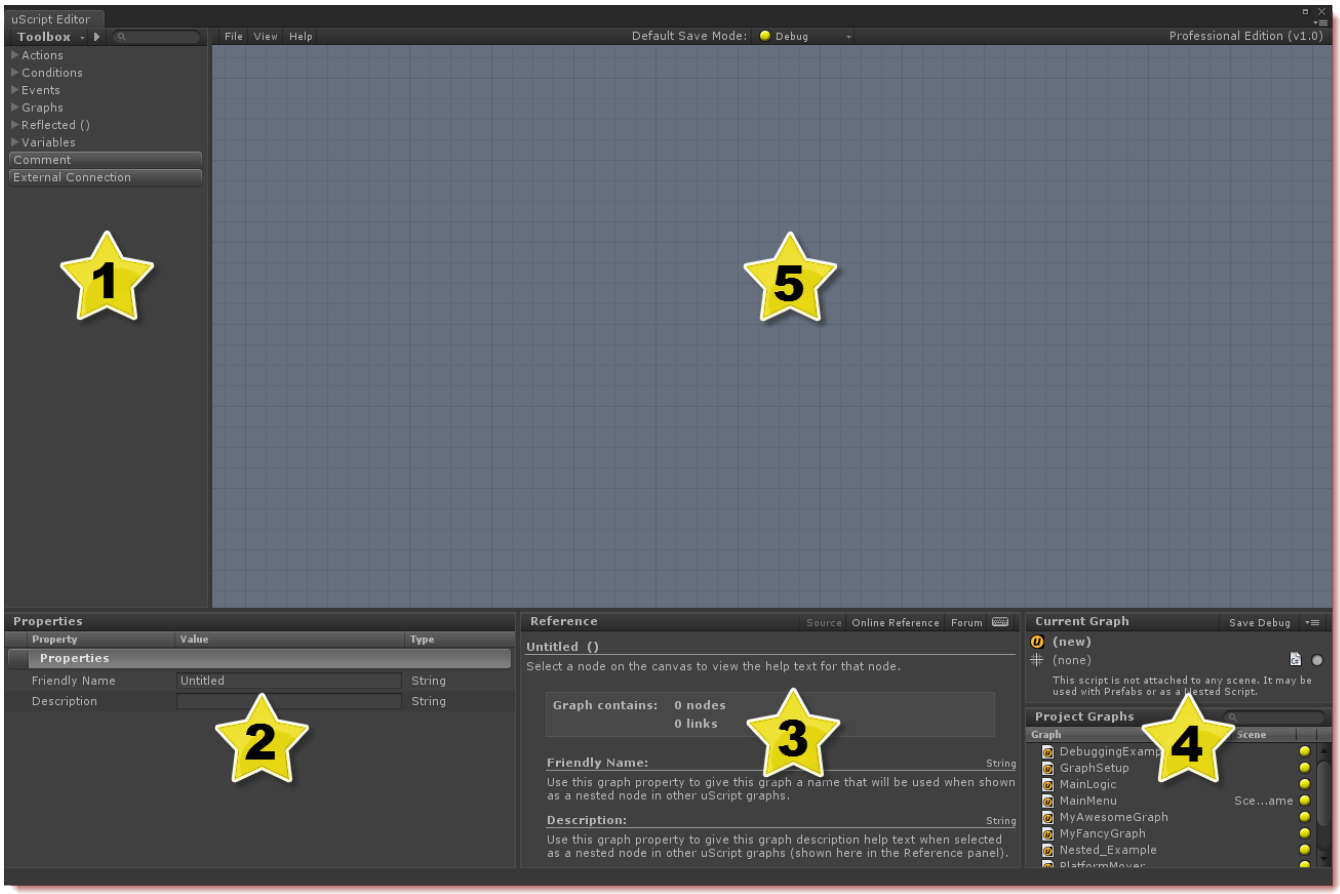


Discovering uScript

This section covers the uScript Editor interface and basic functionality.

Editor Interface

The uScript Editor contains five major sections.



Editor Sections

Action Panel

*See **1** in the above image for this panel's location.*

This panel has more than one view. The default view is the *Toolbox*, which contains a list of all the nodes you can place on the Canvas. The other panel is the *Contents* panel, which shows all the nodes you currently have placed on your graph. Both panels are described in detail below.

Toolbox

The Toolbox panel contains all the possible nodes that can be placed in your graph. To place a node from the Toolbox, just click the node button and a new instance of that node will appear in the center of the Canvas. If you wish to know what the node does before placing it, just hold the mouse cursor over the node button and it's help text will appear in the *Reference Panel*.

By default, the Toolbox contains the following node categories:

> ***Actions -*** This sections contains all the basic action nodes. Action nodes are the basic building block for creating all complex logic in uScript. For more information on Action nodes, see "Nodes" in the *Working With uScript* section.
>
> **Conditions -** This section contains most of the Action nodes related to comparing variables and other key environment data. While they work the same as other Action nodes, they were broken out into their own section for ease in finding them since they pay a key role in building complex logic.
>
> **Events -** This section contains all the Event nodes. All logic in uScript must initially be triggered by an event, making these nodes a critical part of any uScript graph you create. For more information on Event nodes, see "Nodes" in the *Working With uScript* section.

Graphs - Optional. This section is only displayed if your Unity project contains special uScript graphs called Nested Graphs. Nested Graphs will create nodes you can place in other graphs. They are very powerful because they let you visually create new nodes for uScript. For more information on Nested Graphs and Nested Nodes, please the "Graphs" topic in *Working With uScript*.

Reflected ( ) - This section contains all the nodes (including Actions, Properties, and Variables) that have been reflected by uScript from your Unity project. This section will contain the name of your currently loaded scene if you have one loaded such as "Reflected (*YourSceneName*)". The idea is that uScript can only reflect things that are available in your currently loaded Unity scene. For more information on Reflection, see the "Reflected Nodes" topic in the Working With uScript section.

Variables - This section contains all the basic variable nodes that are more commonly used by uScript when creating graphs. If you can't find a variable type you are looking for here, try looking in the Reflected () section for a complete list of all variables. For more information on variables, see the "Variables" topic in the Working With uScript section.

The following nodes are also shown in the root of the Toolbox:

Comment - This node button allows you to place a special node on your graph that you can use to write comment text in.

External Connection - This node button allows you to place an External Connection in your graph. External Connections are used when making Nested Nodes. To learn more about Nested Nodes, see the Nested Graphs section of the "Graphs" topic found in the *Working With uScript* section of this guide.

Tip - *use the small arrow button at the top of the panel next to the search filter field to expand or collapse the entire node tree.*

Tip - *use the filter field at the top of the panel to help you quickly find what nodes you are looking for. Simply start typing key words for the Toolbox to filter the nodes it will show. Be mindful though*

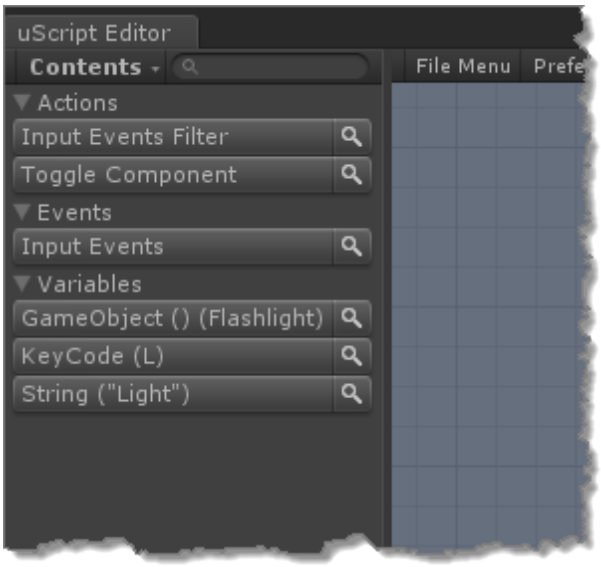
to make sure you clear the selection later to show all the nodes again.

Favorites Section

This section of the Toolbox is hidden when empty, but will appear as soon as you have assigned at least one node as a "Favorite". To assign one or more nodes as favorites, use the Favorites Button in the Properties Panel when the node is selected. The Favorites section is just a way to easily access your most commonly used nodes without having to search through the tool box to find it. You can place a favorite node on the Canvas by either clicking on them here in the panel or by holding their assigned hotkey number while left-clicking on the Canvas.

Contents

The Contents panel allows you to see a list of all the nodes you currently have on your graph. Pressing the node button will select the node in the graph, updating the Properties panel, and clicking on the little magnifying glass icon on the right of each button will focus the canvas on that node.



This panel can be very useful for helping you navigate around a large graph. For more information on using this panel, See "Canvas Navigation".

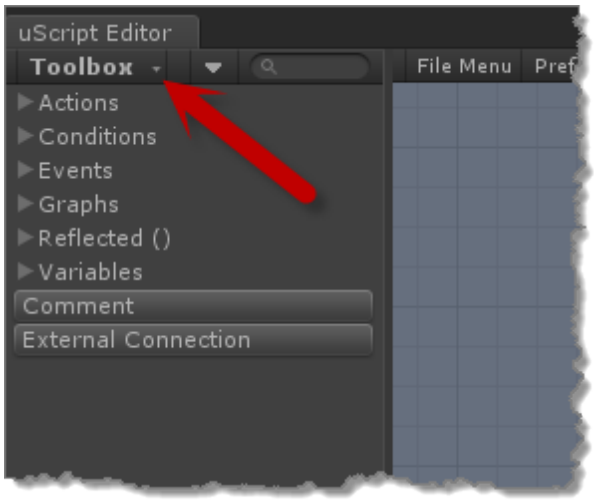
> **Tip** - *the nodes in your Contents list for nodes that have been deprecated will turn a pink/purple color, allowing you to quickly find them.*

> **Tip** - *use the filter field at the top of the panel to help you find what you are looking for on graphs with many nodes. Be mindful though to make sure you clear the selection later to show all the nodes again.*

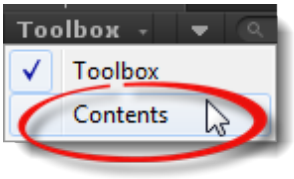
Changing The Displayed Panel

The panel displayed can be changed by clicking on the panel title pulldown (which is set to *Toolbox* by default). The following example shows you how to change the panel from showing the *Toolbox* to showing the *Contents* panel.

To begin, click here:



And select *Contents* from the pulldown:



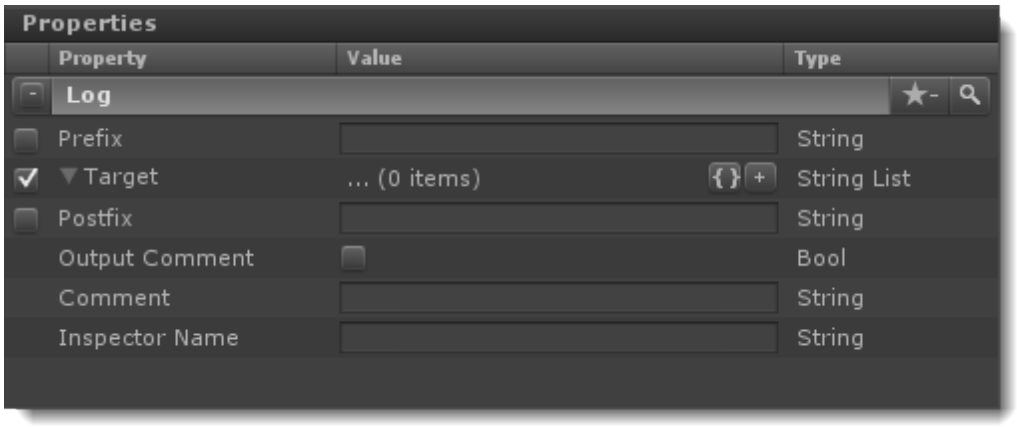
The Contents panel should now be displayed instead of the Toolbox.

Properties Panel

See **2** *in the above image for this panel's location.*

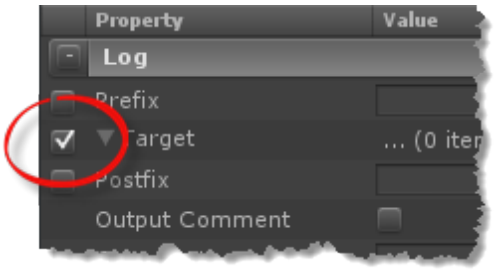
This panel displays all the properties for the selected node(s). Here you can set important values and hide or show sockets on the node. By default only one node's properties at a time can be displayed in this panel. If you wish to increase this number, you can do so in uScript's *Preferences Window*. See the "Editor Preferences" topic in the *Discovering uScript* section of the User Guide.

Each node will have a unique set of properties that can be set. Each node also has a default value for its various properties that will be used if not specified. Here is the properties for a Log node:



The Properties Panel is made up of for key columns:

Socket Shown Checkbox - the checkboxes in the far left column specify if a given socket should be displayed on the node in the canvas. If it is checked, it will have a corresponding visible socket on that node. If a checkbox is not displayed next to a property, that most likely means that it is either a core property common to most nodes or an optional special property and is used by uScript for informational or debugging purposes only.



Property - this shows the name for the property. To learn more about what that specific property does, you can refer to the *Reference Panel* and read the help text for that property.

Value - this is where you can define the data for each property directly on the node. Optionally, you can hook up Variable Nodes to the node's exposed property sockets to define the node's property data. When a variable node is hooked up to a socket, the value field will become disabled. This is because uScript will always use a variable node's value over what might be defined in the *Property Panel*.

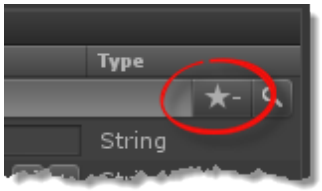
Value Buttons - Some properties may have common Unity buttons depending on their type-- such as the asset browser button or the array buttons used to add elements to an array, as

shown here:

Type - the type column tells you what kind of data or variable is required for each property. See the "Variables"topic in the *Working With uScript* section for more information on variable types.

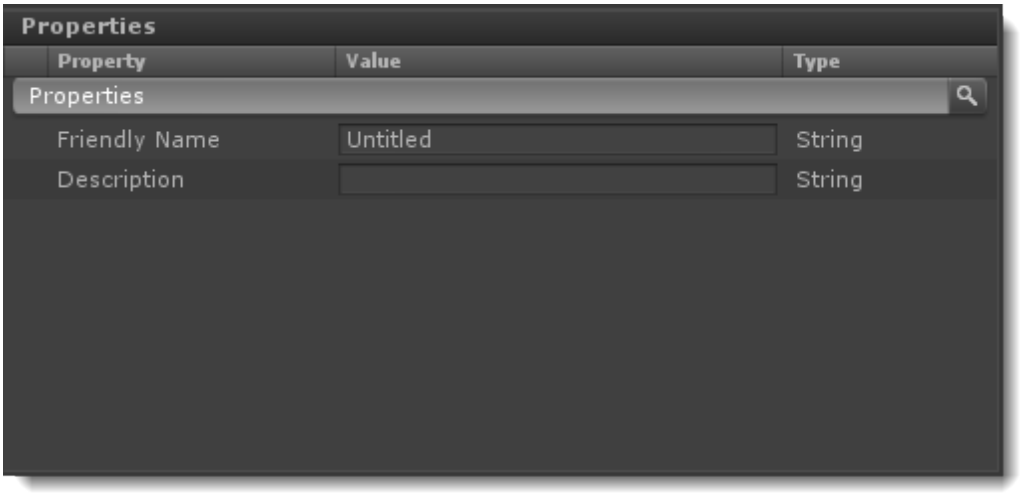
Node Favorites Button

When a node is selected, a small star button appears at the top right of the Properties Panel. This is the Favorites button and allows you to add (or remove) the selected node to the Toolbox's Favorites section (also see the Toolbox section above).



Graph Properties

When no nodes are selected, the properties panel will display the properties for the currently loaded graph:

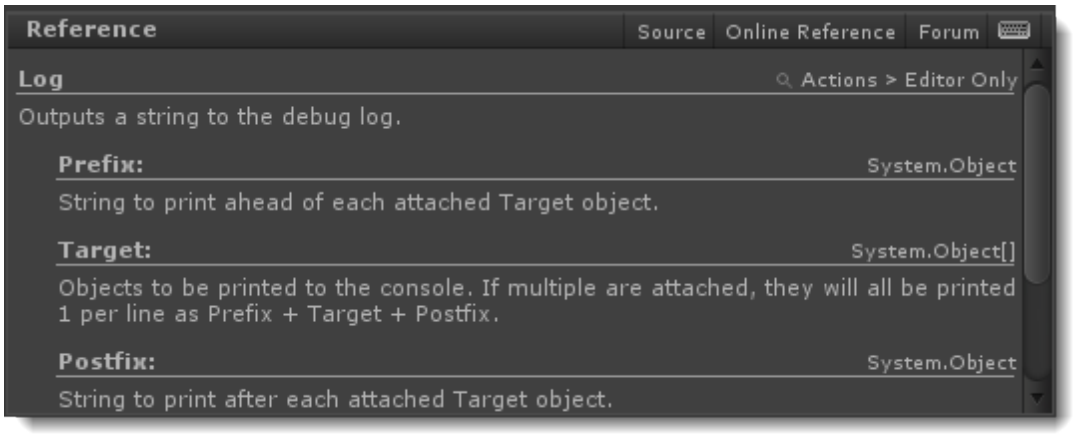


See the *Graph Properties* section of the "Creating Graphs" topic for more information on graph properties and how they are used.

Reference Panel

See **3** *in the above image for this panel's location.*

This panel displays help text of a selected node, or the help text for a node that has focus from the mouse cursor in the Toolbox. The help text for each node is actually generated from the node itself. Also, when making Nested Nodes, you can specify help text in the graph's and External Connection node's properties that will be displayed here (see the *Nested Graph* section of the "Creating Graphs" topic for more details).



Panel Title Bar Buttons

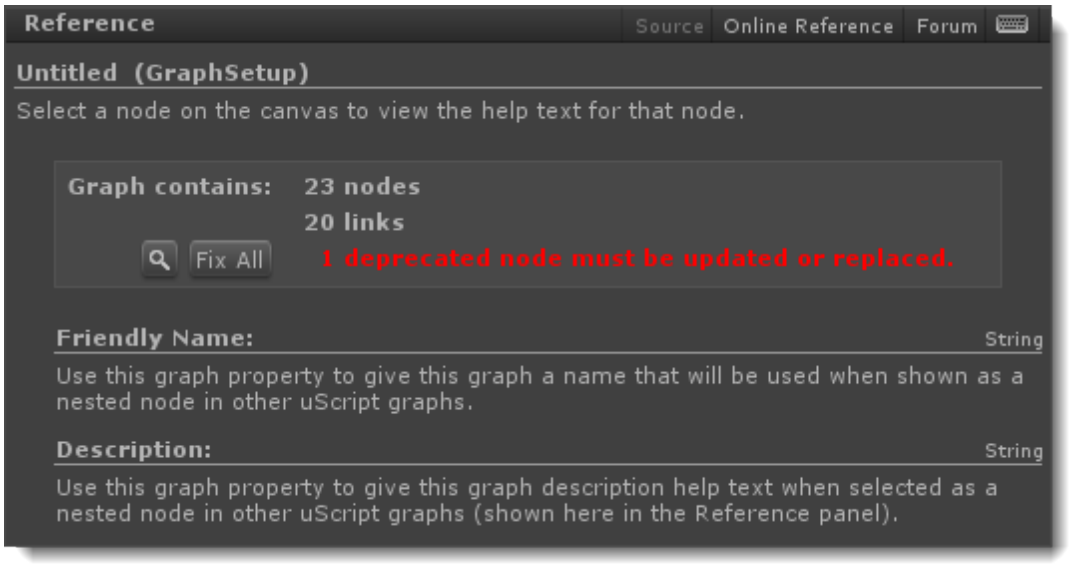
This panel has four buttons in its title bar area:

- **Source -** the source button will use Unity's "ping" feature and show you the source file of that node in Unity's project tab. This button will become disables if the selected item on the Canvas does not have a corresponding source code file.
- **Online Reference -** this button will launch your default web browser and load the latest online version of the uScript documentation.
- **Forum -** this button will launch your default web browser and load the uScript Community's forum. Our forum is a great place to search for past questions that have been answered, hang out with other uScript users, or ask new questions that may not be answered in the documentation.
- **Hot-Key Reference -** this button looks like a little keyboard. When you press it it will bring up a separate Unity window showing you most of uScript's "Hot-Keys"

Graph Information

When no nodes are selected, the Reference Panel will instead show basic information about the currently loaded graph.

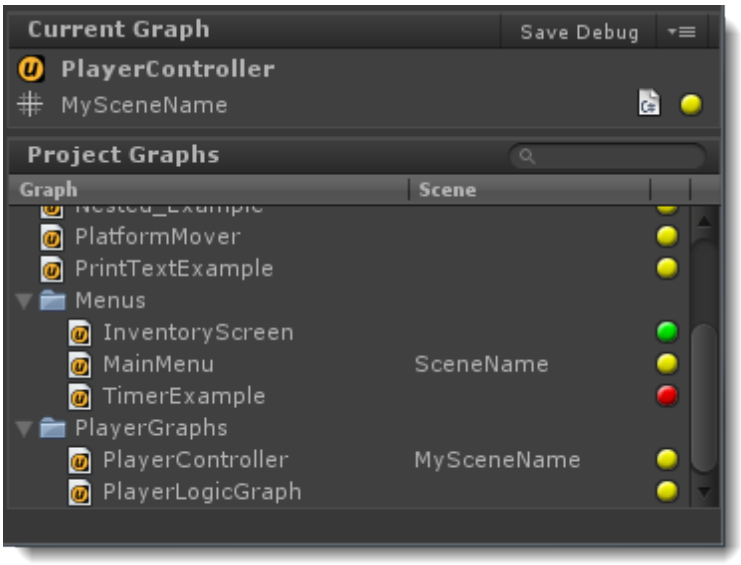
Here you can see that this graph (with a file name called *GraphSetup*, but no assigned "Friendly Name") has 23 nodes and 20 links (connections between nodes). It also has one deprecated node in the graph that you can either focus on or fix from this panel.



Graphs Panel

See **4** *in the above image for this panel's location.*

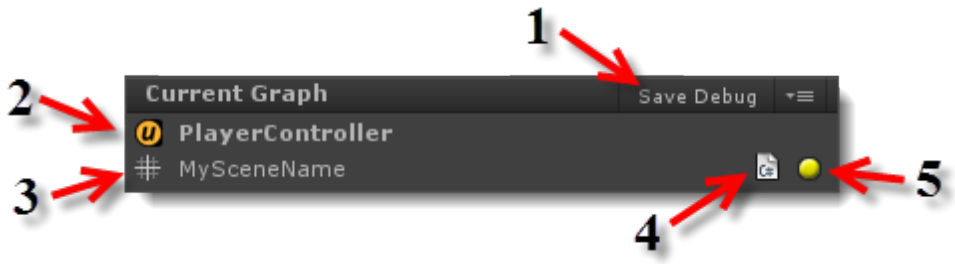
This panel shows you all the graphs you have created in your Unity project. It also gives you some information about your project's graphs.



This Panel is made up of two main sections:

Current Graph Section

This is the upper section of the panel and provides information about the currently loaded graph:



1. Graph Save Button - Press this button to save the currently loaded graph. The button name will reflect the current save method selected in the menu section above uScript's Canvas. This button functions just like the Save file menu item and Save Hot-Key.

2. Graph Name - This is the name of the currently loaded graph. Left-clicking on the graph name will make Unity ping/focus on the *.uscript* file for this graph in Unity's project panel.

3. Unity Scene Name - If this graph was assigned to a specific Unity Scene when it was created, that will be shown here.Left-clicking on the scene name

will make Unity ping/focus on the scene file for this graph in Unity's project panel.

This is set by selecting the option to assign a newly created graph to the *_uScript* Master GameObject in the currently loaded scene on creation when uScript asks if you would like to do that. Graphs that have been assigned to a specific Unity scene should only be edited/saved when that same Unity scene is open in the Unity editor. This is because the graph may be referring to scene-specific GameObjects and other reflected scripts and variables. If a graph that is assigned to a Unity scene is open without the corresponding Unity scene also being loaded into the editor, uScript will warn you and this name will be shown in red along with a warning that you should load the correct Unity scene before editing/saving the graph. Ignoring this warning can result in data loss in your graph.

4. Generated Code Icon - This icon is just there to let you know that the colored Status Icon is for information related to the generated script code for this graph.

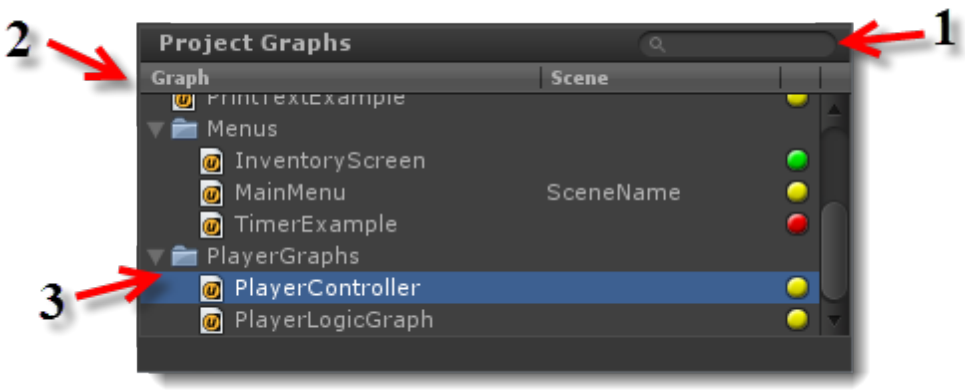
5. Generated Code Status Icon - This simple icon reflects the current state of this graph's generated source code files. Left-clicking on this icon will make Unity ping/focus on the *.cs* component file for this graph in Unity's project panel.

- ***Green -*** The script code is up to date and saved in Release mode.
- ***Yellow -*** The script code is up to date and saved in Debug mode.
- ***Red -*** The script code is not up to date because the graph was last saved in Quick mode.
- ***Gray -*** The script code is missing. Save your graph in either Release or Debug mode to have uScript generate it for you.

Note! *- Right-clicking anywhere in this section will pop up a context menu allowing you to perform some file operations on the graph as well as provide other helpful functionality.*

Project Graphs Section

This is the lower section of the panel and provides a list and basic information about all the graphs in your project.



1. Graph File Search Field - This field will allow you to search and filter what graphs are displayed in the list. This can be very helpful for large projects with lots of graphs a sub-folders. Clear the field of text (or press the little "x" that appears when typing to do it for you) to see all your graphs again.

2. Graph List Bar - The list bar contains three columns (from left to right):

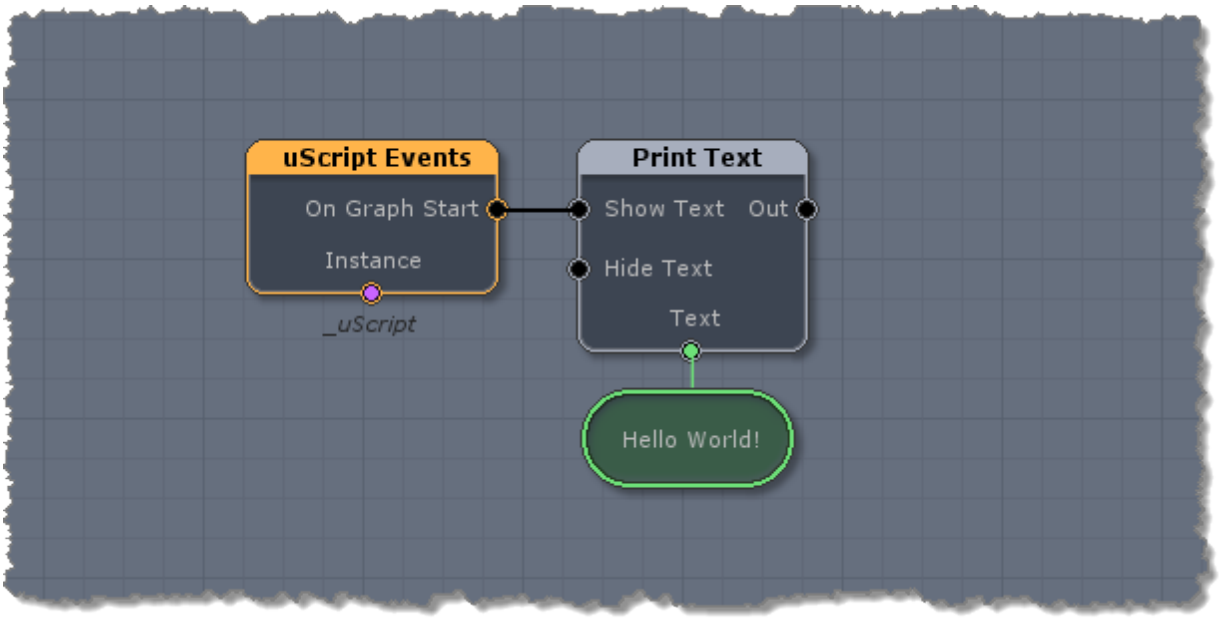
- ***Graph -*** This column displaces the graph file and directory names for the project.
- ***Scene -*** This will display the Unity scene name for any graphs that have been specifically assigned to a Unity scene.
- ***Generated Code Status -*** The right-most column displays a color-coded icon showing you the current status of the graph's generated script code. This color code matches those described in the Current Graph Section previously.

3. Graph File and Directory Names - The main section of the Graph List displays all your project graphs and sub-directories. Double left-click on a file to load it into uScript. Right-clicking anywhere in this area to pop up a context menu allowing you to perform some file operations on the highlighted graph as well as provide other helpful functionality for the graph list.

Canvas

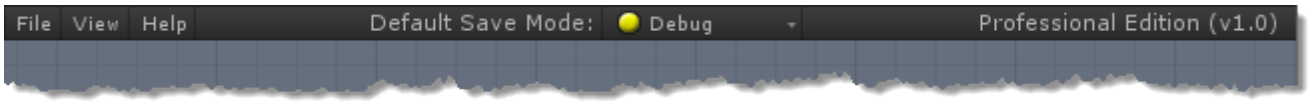
See **5** *in the above image for this panel's location.*

The Canvas is where your create your logic graphs by placing and connecting various nodes. At the top of the Canvas is the Menu Bar which is used for file management and editor preferences. See the Canvas Navigation section below for information on making the most out of the Canvas area.



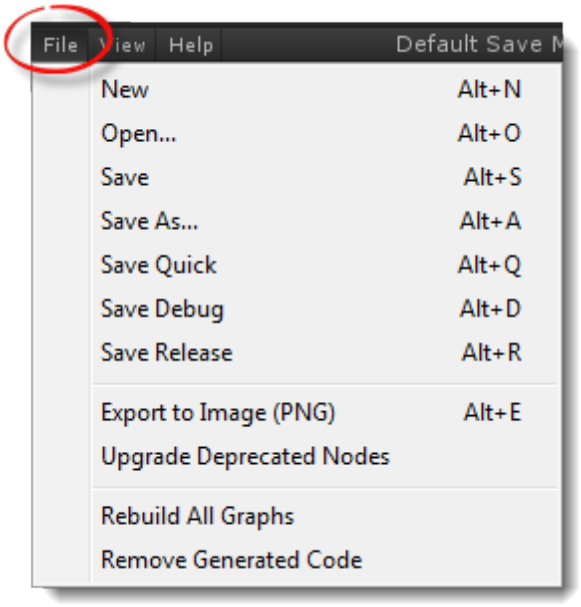
Menu Bar

Across the top of the Canvas is uScript's menu bar. There are four key items found here that are explained below.



File Menu

The file menu is where you can perform file-based operations such as saving and loading uScript graphs or deleting and regenerating uScript graph code files. Click the *File Menu* button to expand it:



Menu Items

The following items can be found in the File Menu:

New - use this to create a blank/new uScript graph for your project.

Open... - open an existing uScript graph in your project. This can also be done via the Graphs Panel.

Save - save the currently open graph using the current set save mode (*Debug, Release, or Quick*). To set the save mode, use the *Save Method pulldown* also found on the Canvas Menu Bar. If you have not already saved the open graph, this will function the same as the "Save As..." option below. This can also be done via the Graphs Panel.

Save As... - saves the currently open graph as a new graph file. You will be asked to specify a name for the graph when using this option. This can also be done via the Graphs Panel when hitting the save button on a newly created graph that has not yet been saved.

Save Quick - save the currently open graph using the Quick save method. This save method is useful when you wish to save some changed to your graph without having to wait for Unity to recompile the generated script files. It will **NOT** generate the code files like using the other "Save" menu items. If you use this quick save option, make sure you perform a full Save (release or debug) before running the game (*pressing Play in Unity*).

Save Debug - save the currently open graph using the Debug save method. Saving graphs as debug will allow you to step debug your graphs by using breakpoints by generating script files with extra debug information used by uScript. See the Visual Debugging section of the "Debugging Your Graphs" section for more information on debugging.

Save Release - save the currently open graph using the Release save method. This is the method that should be used when saving out your graphs for releasing your game. Saving graphs this way creates smaller, more optimized generated script files. You will not be able to use visual step debugging when saving your graphs this way.

Export to Image (PNG) - this will allow you to export your graph as a .png image file. This can be a great way to print out your graphs on a plotter or to post an image of your graph in a forum or as part of a tutorial.

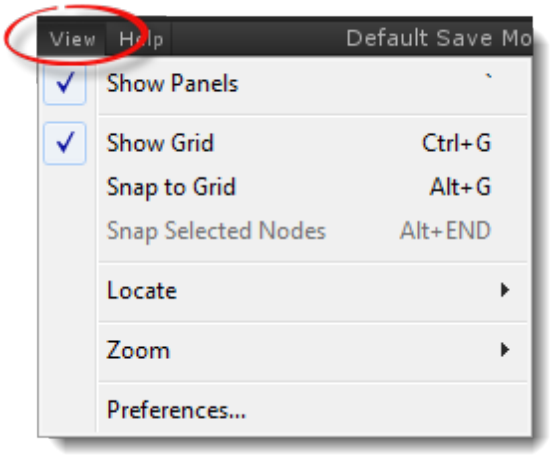
Upgrade Deprecated Nodes - useful for auto-updating all deprecated nodes on your graph at once. You should only use this option if you are sure you wish to have all your nodes updated automatically. You may wish to review each node that needs updating to see if any data might be potentially lost that you care about. See "Pink Nodes" in the *Working With uScript* section for more information on deprecated nodes.

Rebuild All uScripts - this will tell uScript to regenerate the script code for all of the uScript graphs in your Unity project. This can be useful if you have updated custom nodes and they cause Unity to generate compile errors on your existing graphs generated script code. This action is best done with a blank Unity scene loaded to help prevent data loss.

Remove Generated Code - uScript relies on Unity to be able to fully compile it to function properly. If for some reason Unity is having trouble compiling the generated script files, you can quickly delete all of your generated script files with this option. Don't worry though- no work would be lost. As long as your *.uscript* files remain in the project, uScript can always rebuild the generated script code. Note however that you should **NOT** save any Unity scenes while these files are missing as it could cause your scene's connection to the uScript graphs to be lost (and will require manual hookup). This action is best done with a blank Unity scene loaded to help prevent data loss.

View Menu

The view menu contains helpful functionality focused on how you view your graph and nodes in uScript.



Menu Items

The following items can be found in the View Menu:

Show Panels - toggles the uScript panels on and off.

Show Grid - toggles the Canvas grid on and off.

Snap to Grid - toggles grid snapping on the Canvas on and off.

Snap Selected Nodes - snaps the currently selected nodes to their closest grid lines. Only available when nodes are selected.

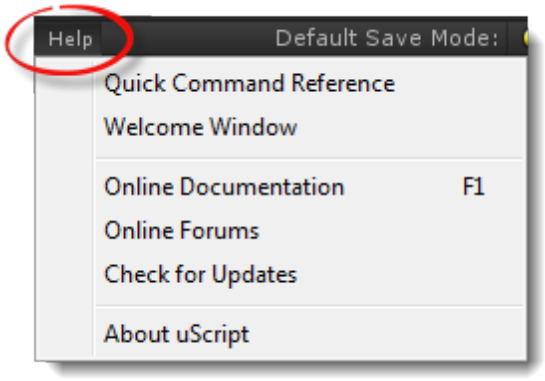
Locate - provides access to some of uScript's most common features for finding nodes on the canvas.

Zoom - provides access to uScript's Canvas zooming features.

Preferences... - brings up uScript's Preferences Window. This window allows you to customize some aspects of uScript. See "Editor Preferences" for more information.

Help Menu

The help menu contains access to uScript's various help resources and other product information and links.



Menu Items

The following items can be found in the Help Menu:

Quick Command Reference- shows uScript's hot-key reference window displaying uScript's most common hot-keys.

Welcome Window - shows uScript's Welcome Window with helpful links to important uScript information and tutorials.

Online Documentation- a direct link to uScript's latest online user documentation (opens your default web browser).

Online Forums- a direct link to our official uScript community and support forum (opens your default web browser).

Check for Updates - forces uScript to check to see if there is a newer version of uScript than what is currently installed.

About uScript - shows uScript's About Window with basic product information.

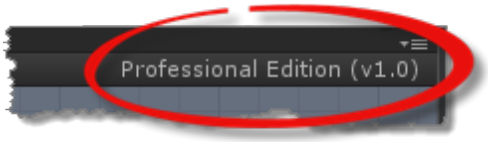
Save Method Pulldown

This pulldown allows you quick access to uScript's preference setting of how uScript should save scripts when saving. See the *"Save Method"* section in the "Editor

Preferences" topic for more information.

uScript Version Information Text

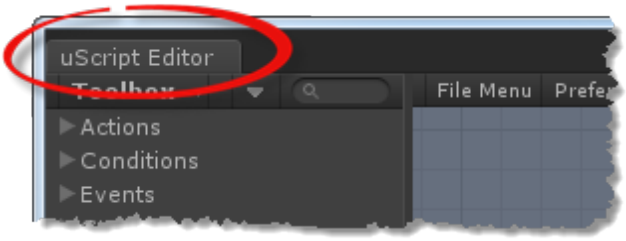
The right-most section of the menu bar shows you what version, and version number, of uScript you are currently running.



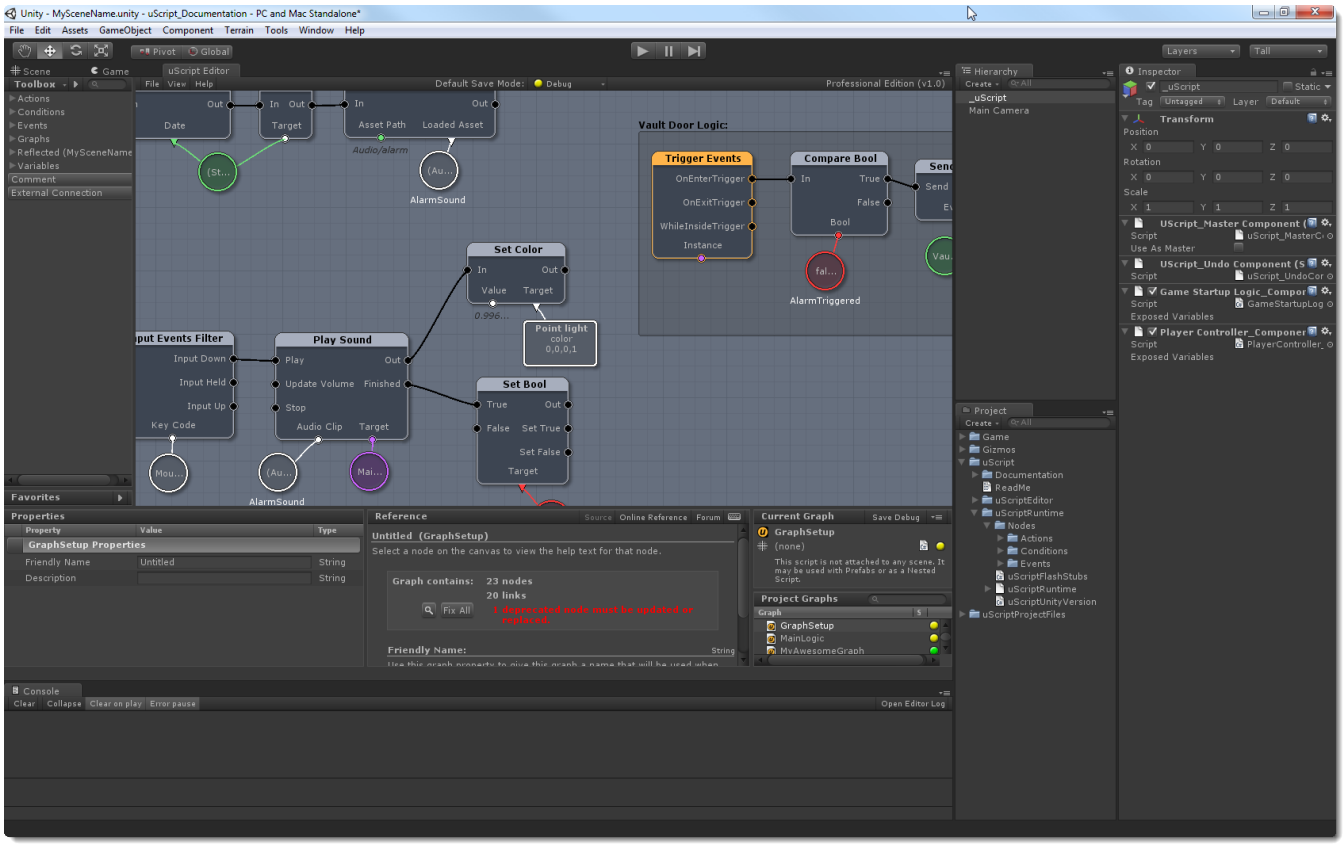
Docking in Unity

It is also possible to dock uScript within the main Unity UI just like any other Unity window. To do this you just need to left-click and drag the "Tab" section of the uScript editor found in the top-left corner over the sport in the main Unity UI you wish to dock uScript into. Docking uScript can be helpful when you are working with a single monitor and wish to easily drag and drop items from the main Unity UI onto the uScript Canvas.

Left-click drag here:



Example of uScript docked into the main Unity Editor UI:


Unity - MySceneName.unity - uScript_Documentation - PC and Mac Standalone*
File Edit Assets GameObject Component Terrain Tools Window Help
Scene
Game
uScript Editor
Toolbox
Actions
Conditions
Events
Graphs
Reflected (MySceneName
Variables
Comment
External Connection
Favorites
Default Save Mode: Debug
Professional Edition (v1.0)
Vault Door Logic:
Trigger Events
OnEnterTrigger
OnExitTrigger
WhileInsideTrigger
Instance
Compare Bool
In
True
False
Bool
AlarmTriggered
Set Color
In
Out
Value
Target
Point light
color
0,0,0,1
Play Sound
Input Down
Input Held
Input Up
Key Code
Play
Update Volume
Stop
Audio Clip
Finished
Target
AlarmSound
Set Bool
True
False
Set True
Set False
Target
Asset Path
Loaded Asset
Audio/alarm
AlarmSound
Hierarchy
Create
_uScript
Main Camera
Inspector
Transform
Position
Rotation
Scale
Project
Game
Gizmos
uScript
Documentation
ReadMe
uScriptEditor
uScriptRuntime
Nodes
Actions
Conditions
Events
uScriptFlashStubs
uScriptRuntime
uScriptUnityVersion
uScriptProjectFiles
Properties
GraphSetup Properties
Friendly Name
Untitled
String
Description
Reference
Untitled (GraphSetup)
Select a node on the canvas to view the help text for that node.
Graph contains: 23 nodes
20 links
Fix All
1 deprecated node must be updated or replaced.
Friendly Name:
Current Graph
GraphSetup
(none)
This script is not attached to any scene. It may be used with Prefabs or as a Nested Script.
Project Graphs
GraphSetup
MainLogic
MyAwesomeGraph
Console
Clear
Collapse
Clear on play
Error pause
Open Editor Log

Canvas Navigation

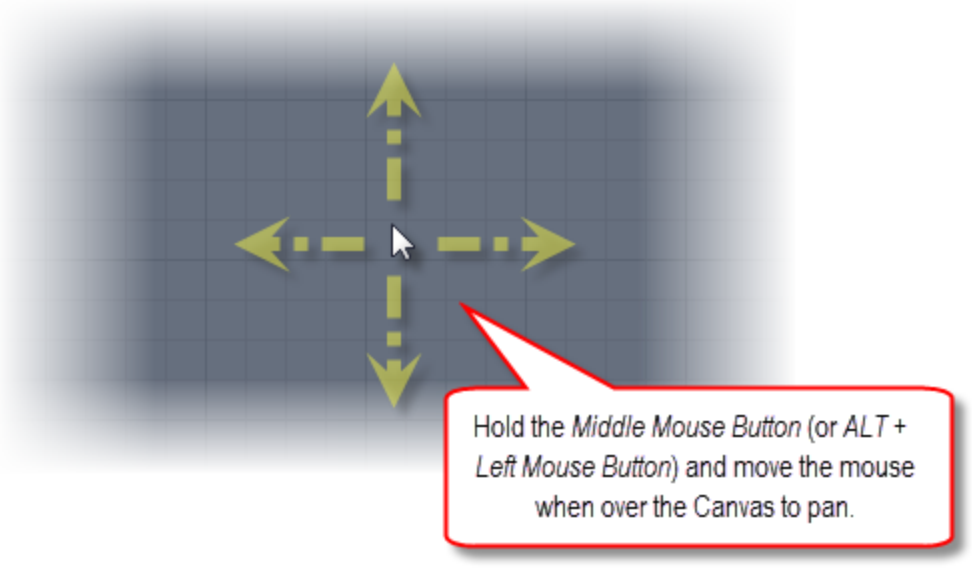
The Canvas is the section of uScript where you create and edit your visual graphs. The uScript Canvas is infinitely large, allowing you to move out in any direction from the initial "home" position in order to create as much visual logic as you like. This of course though leads to the problem of making sure you can still find everything you have created and to quickly be able to move to where you wish to be on the Canvas.

> ***Note!*** *- learn the hot-keys you can use with the Canvas to really become a power user! See "Hot-Keys"*

Moving Around The Canvas

Panning

One of the first thing you will want to do is to pan the Canvas around. This is easily accomplished by holding down the *Middle Mouse Button* while over the Canvas and moving the mouse. If you don't have access to a mouse with a middle mouse button, you can optionally use *ALT + Left Mouse Button* instead.



Other Ways To Get Around

uScript also offers several other useful ways to get around the Canvas and find what you are looking for.

Contents Panel

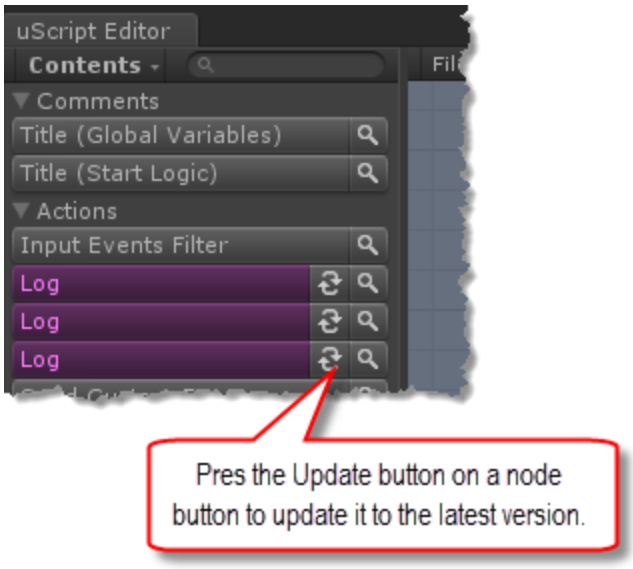
The Contents Panel can be a great way to see all the nodes currently on the Canvas. It also allows you to quickly select the nodes or jump the Canvas focus to a node. For information on accessing the Contents Panel, see "Editor Interface".

Here is an example of the contents Panel for a graph with several nodes on it:

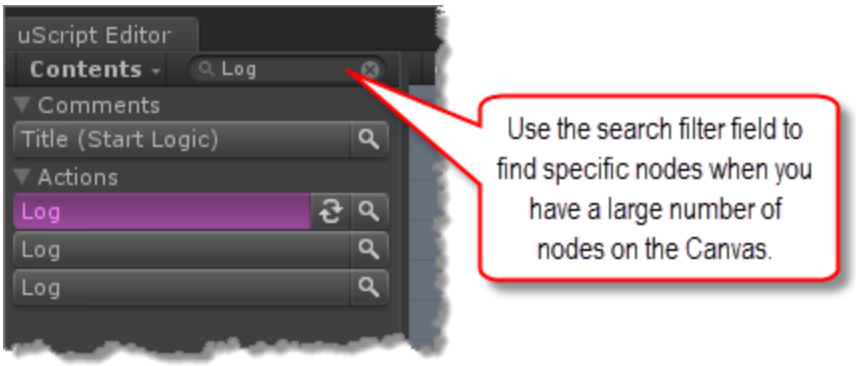
Each "node button" in the list allows you to either select the node (by clicking on the node button itself) or move the Canvas view to center over the selected node (by pressing the search button on the right side of each node button):

Nodes can also be multi-selected by holding down the SHIFT key while clicking on the node buttons:

When nodes on the Canvas are out of date, either because they have been updated in a way that might cause data changes or loss or because the Deprecated attribute was added to a node, the node button for that node will show up as a purple/pink color and add a new button option for the node, allowing you to update the node:



You can also use the Search Filter Field to help you narrow down the nodes visible in the list to help you find things quickly:



Home Key

You can use the HOME key to quickly reset the Canvas position to its center.

Event Node Focusing

Since all graph logic starts with an Event Node, it can be very helpful to quickly jump Canvas focus between the events nodes to find the logic you are looking for. In uScript you can do this by pressing the bracket keys on the keyboard ("**[**" and "**]**"). These two hot-keys will move the canvas focus to the next/previous Event Node.

Canvas Zooming

To zoom the Canvas in and out, you can the *Scroll Wheel* on your mouse. If you do not have access to a *Scroll Wheel* on your mouse (or equivalent on laptop touchpads), we also have some hot-keys to allow for zooming.

Fixed Zooming

You can also perform fixed zooming using keyboard hot-keys. Fixed zooming means that uScript will auto zoom to specific percentages when the key is pressed. This allows for much faster zooming with hot-keys.

The hot-keys to zoom are the:

- **MINUS** key (**-**) - zooms out at 10% increments
- **EQUALS** key (**=**) - zooms in at 10% increments
- **ZERO** key (**0**)- resets the zoom level to 100%

> **Note!** - *these hot-keys are on the main keyboard. The same keys on the numeric pad of keyboards that have them will **not** work (as they are different keys).*

Node Manipulation

Once you know how to move around the Canvas, the next most important step is to be able to manipulate the nodes on the Canvas.

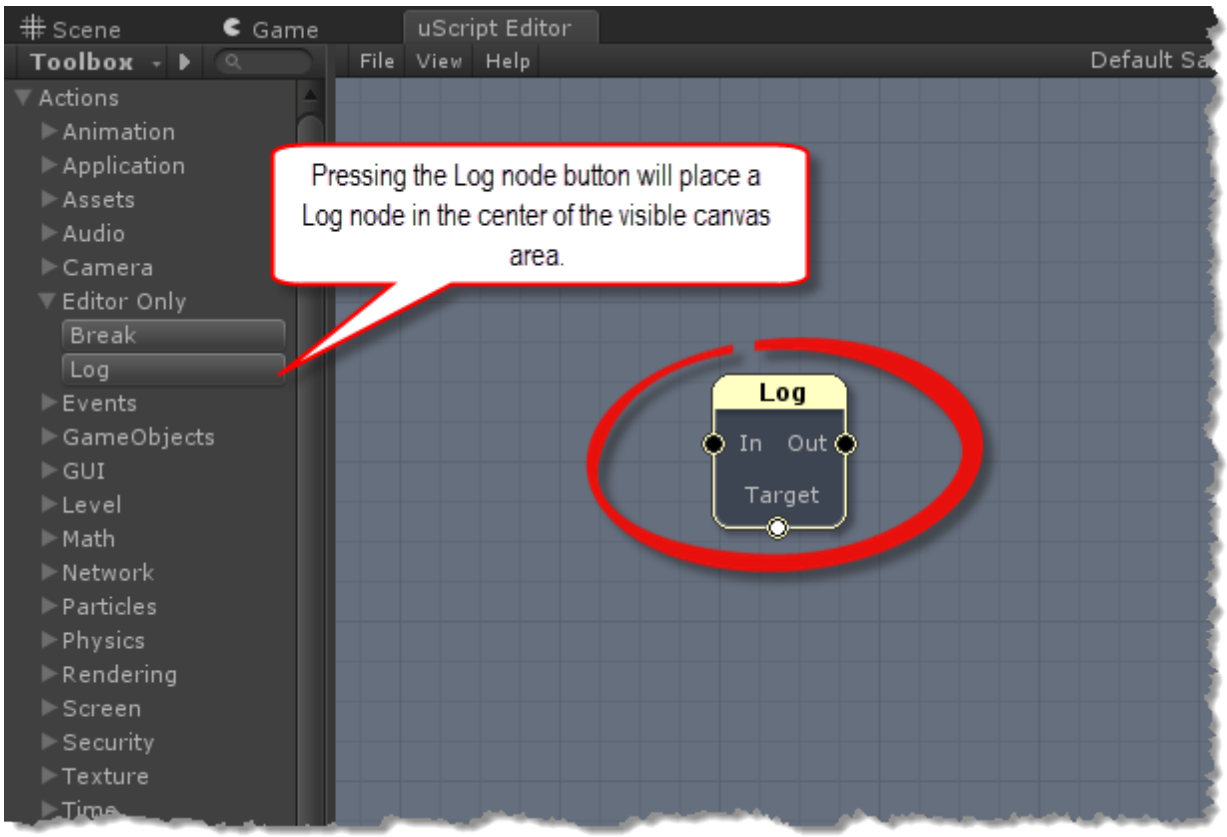
> ***Note!** - all node manipulation can be undone using the Undo/Redo system in Unity. Simply use the standard Undo/Redo hot-keys; CONTROL+Z for Undo and CONTROL+Y for Redo (Use COMMAND instead of CONTROL on the Mac)*

Placing Nodes

There are three main ways to place a new node on the Canvas.

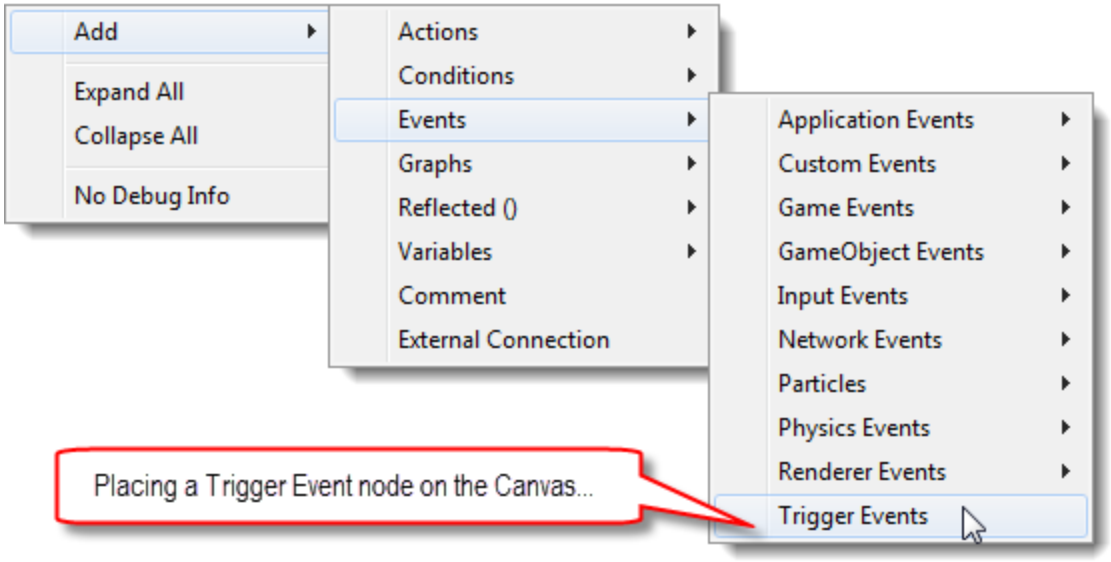
Using The Toolbox

The first is to select the node's node button on the Toolbox. Clicking on the node button will place an instance of that node in the center of the visible Canvas and automatically selects the new node:



Canvas Context Menu

Another way to place new nodes is to Right-Click on the Canvas and use the Canvas Context Menu. Once the Context Menu appears, go to the "Add" section and navigate to the node you wish to place. Once you have selected the node you want, it will be placed at the position on the Canvas where you right-clicked:



Copy Paste An Existing Node

Lastly, you can simply copy an existing node on the Canvas and paste it as a new node. When you do this, the new node will have all the same settings as the old node you copied it from. However, no connection hooked into the original node will be made, allowing you to quickly setup a new node based on an existing one.

Selecting Nodes

To select nodes on the Canvas just Left-Click on the node you wish to select. It will highlight with a yellow border. To deselect a node (or multiple nodes), just Left-Click on a blank area of the Canvas grid.

If you wish to select more than a single node at a time, you can also use the following hot-keys:

- **CTRL+A** - this will select all nodes on the Canvas.
- **SHIFT+Left-Click** - allows you to multi-select node and toggle their selection state on or off.

Selection Box

Lastly, you can draw a selection box. This will select all nodes that it touches. To create a selection box, just Left-Click on the Canvas and drag the mouse cursor. The selection

box will be drawn out as you move the mouse cursor away from its starting location.

You can also use the following hot-keys with box selection:

- **SHIFT** - holding the SHIFT key while doing a box selection will add new nodes to the current selection.
- **CONTROL** - holding the CONTROL (CMD on Mac) key, will remove nodes from the current selection.

Contents Panel

You can also select nodes by using the Contents Panel. Information on using the Contents Panel to select nodes can be found above in this section.

Moving Nodes

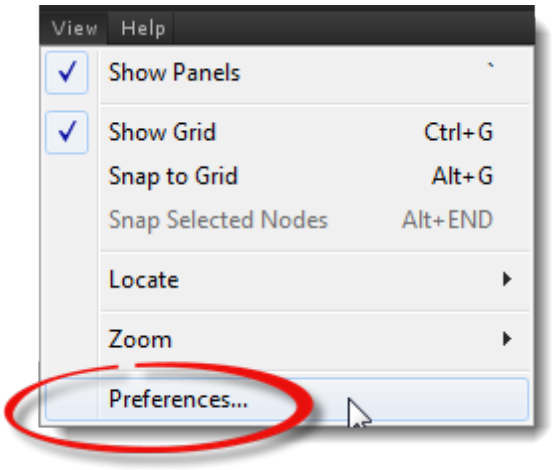
Moving nodes is as simple as Left-Clicking on a node and dragging it with the mouse. If more than one node is currently selected, just Left-Click on any one of them to drag them all together.

Deleting Nodes

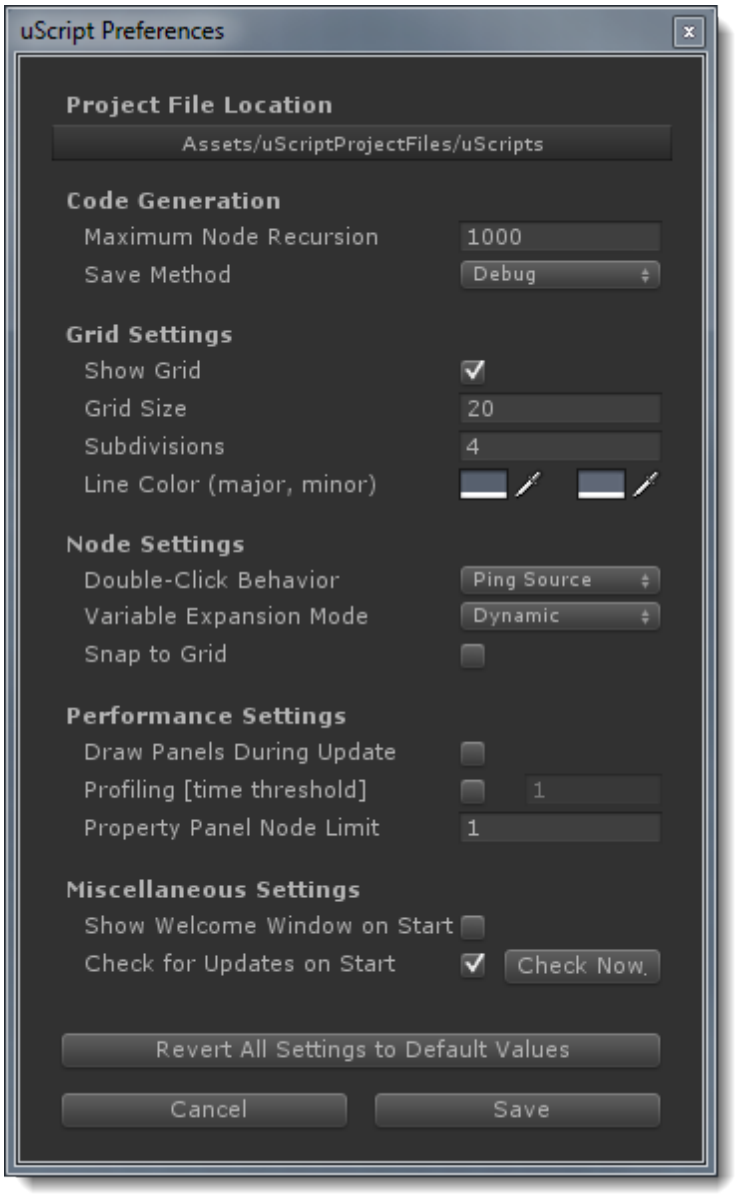
To delete nodes, first you must select them. Press the DELETE or BACKSPACE key to delete selected nodes on the Canvas.

Editor Preferences

The uScript Editor has a few preferences that you can set to your liking. uScript's preferences can be accessed by pressing the Preferences menu item in the View Menu (located above the Canvas in the top left of the uScript Menu Bar).



Pressing the menu item will cause the uScript Preferences window to appear:



Preference Settings

The following describes what each setting does. Press the "*Revert All Settings to Default Values*" button to reset uScript settings to their default values.

> **Note!** - *The settings file used by uScirpt is created when uScript first runs and can be located at* ***../Assets/uScriptProjectFiles/uScriptSettings.settings*** *by default.*

Project File Location

Path Button

Default Value: *Assets/uScriptProjectFiles/uScripts*
Legal Values: *Any legal path within your project's Assets folder.*
This button allows you to modify where uScript will save its graph and generated code files in your project.

Code Generation

Maximum Node Recursion

Default Value: *1000*
Legal Values: *100 - 1000*
Defines a maximum number of times that code generate by uScript will be allowed to call itself (loop/recurs) when generating scripts in debug mode. If you exceed this number, uScript will warn you. This helps to prevent creating infinite loop cases that would cause Unity to become unresponsive.

Save Method

Default Value: *Debug*
Legal Values: *Quick, Debug, Release*
Lets you specify the default save type for uScript.

- ***Quick*** - uScript will only save the .uscript graph file and will not try to generate a C# script from the graph. This is useful when you want to save quick changes without having to wait for Unity to compile the generated script code. Please note, that if you run your project after saving this way, you will not see any of your changes in the running game because no new code was generated by uScript.
- ***Debug*** - uScript will generate scripts for your graphs on save. These graphs will contain extra debug information used by uScript allowing you to perform step debugging of your graphs.
- ***Release*** - uScript will generate scripts for your graphs on save. The generated graphs will be stripped of any debug informaiton that uScript uses. You cannot perform step debugging of your graph if saved this way.

Grid Settings

Show Grid

Default Value: *Checked*
Legal Values: *Checked, Unchecked*
Determines if the Canvas background grid will be shown.

Grid Size

Default Value: *20*
Legal Values: *8 - 100*
Specifies the space, in pixels, between the grid lines of the Canvas background graph.

Grid Major Line Spacing

Default Value: *4*
Legal Values: *1 - 10*
Specifies the spacing of the major grid lines of the Canvas background graph.

Grid Color Major

Default Value: *R:87 B:96 G:110 A:255*
Legal Values: *R:0-25 B:0-25 G:0-255 A:0-255*
Specifies the color of the major grid lines of the Canvas background graph.

Grid Color Minor

Default Value: *R:95 B:103 G:118 A:255*
Legal Values: *R:0-25 B:0-25 G:0-255 A:0-255*
Specifies the color of the minor grid lines of the Canvas background graph.

Node Settings

Double-Click Behavior

Default Value: *Ping Source*
Legal Values: *Ping Source, Open Source, Load Graph Ping Source, Load Graph Open Source*

Defines how uScript should handle double-clicking on nodes with the left mouse button.

- ***Ping Source*** - uScript will use Unity's "ping" feature to highlight the node's source file in the Unity project tab. Nested graph nodes will focus on their generated .cs script file.
- ***Open Source*** - uScript will tell Unity to open the node's source file in the source code editor defined in Unity's settings. Nested graph nodes will open their generated .cs script file.
- ***Load Graph Ping Source*** - Performs the same as *Ping Source* with the exception that if you double-clicked on a nested graph node, it will load that nested graph into the editor instead of pinging it.
- ***Load Graph Open Source*** - Performs the same as *Open Source* with the exception that if you double-clicked on a nested graph node, it will load that nested graph into the editor instead of opening its source .cs script in the default source code editor.

Variable Expansion Mode

Default Value: *Dynamic*
Legal Values: *Always Expanded, Always Collapsed, Dynamic*
Defines how uScript should handle displaying variable nodes when they are selected.

- ***Always Expanded*** - uScript will always show variables nodes fully expanded to fit the value it contains.
- ***Always Collapsed*** - uScript will truncate variable node values and display the node as a circle regardless if it is selected or not.
- ***Dynamic*** - uScript will truncate variable node values and display the node as a circle when not selected and expand them to show the full value when selected.

Snap to Grid

Default Value: *Unchecked*
Legal Values: *Checked, Unchecked*
Specifies if uScript should snap node to the grid. uScript supports simple snapping that will snap the top left corner of nodes to the grid.

Performance Settings

Draw Panels During Update

Default Value: *Unchecked*
Legal Values: *Checked, Unchecked*
Specifies is uScript should draw the contents of it's various UI panels when panning in the Canvas. Because of the massive amount of UI elements that need to be drawn in the uScript Editor you can dedicate all of your CPU's power to more smoothly render the Canvas while it is moving if uScript does not need to redraw the UI elements of the panels at the same time.

Profiling [time threshold]

Default Value: *Unchecked (time threshold: 1)*
Legal Values: *Checked, Unchecked (time threshold: 0.01 - 10)*
This setting is used for uScript debugging purposes only. It allows uScript to print out useful information to Unity's console window related to the time being spent in various parts of uScript's code. The time threshold determines what minimum time needs to elapse in a section of code before printing to the Unity console. This setting should remain off unless you are specifically working with Detox Studios LLC to help debug a performance issue.

Property Panel Node Limit

Default Value: *1*
Legal Values: *1 - 20*
Defines how many selected nodes can display their properties in uScript's Properties Panel at the same time. Setting this to greater than 1 can be helpful if you wish to compare the properties of multiple nodes at once. Setting this number to high may cause performance slowdowns of the uScirpt editor when multiple nodes are selected at the same time on slower computers.

Miscellaneous Settings

Show Welcome Window On Start

Default Value: *Specified by the user when first running uScript.*
Legal Values: *Checked, Unchecked*
Determines if uScript shows it's Welcome Window containing helpful links when the uScript Editor is run.

Check For Updates On Start

Default Value: *Specified by the user when first running uScript.*
Legal Values: *Checked, Unchecked*
Determines if uScript checks to see if an updated version of uScript is available for download when the uScript Editor is run.

Working With uScript

This section of the user guide provides information on working with uScript to create your own visual logic. We do this by describing the various features of uScript as well as providing simple examples of using the various nodes in combination.

Key Concepts

Before you start working with uScript, it is important that you have a good understanding of key concepts for both Unity and uScript.

While uScript's purpose is to unleash the power of programming to those without programming experience, you still need to have an understanding of Unity terminology and its various systems and features.

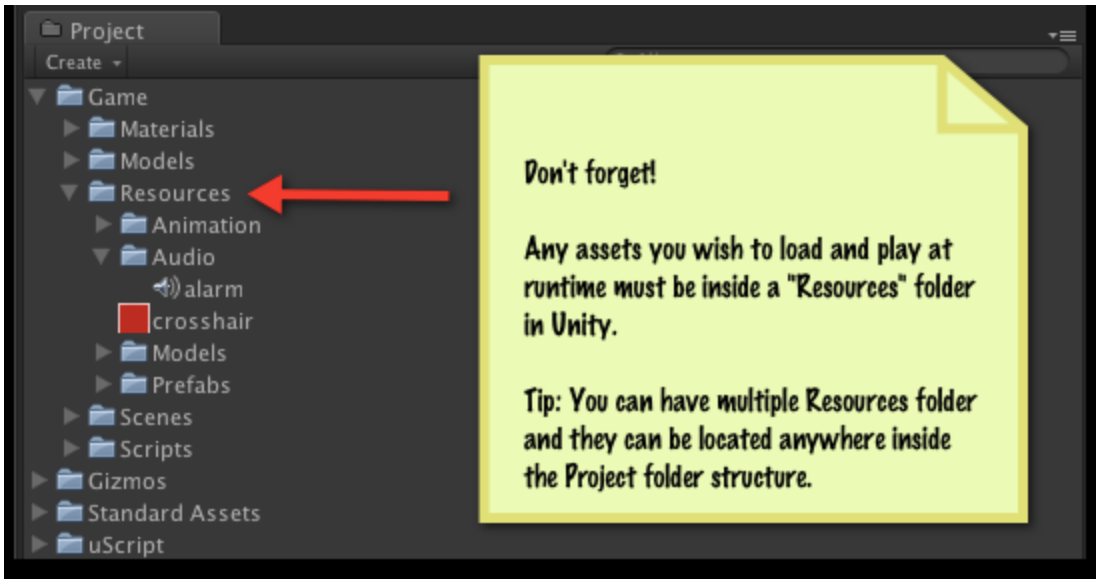
Also, please see the Unity prerequisites area of the "Setting Expectations" topic in the Introduction section for a list of specific Unity concepts we highly recommend you know how to use.

Playing Assets At Runtime

One of the first things many people usually wish to try with uScript is to do something with an existing asset in their game such as spawn a prefab, play a sound or animation, or use some other kind of asset such as a Texture2D, GUISkin, Font, etc. It is important to remember that uScript must follow Unity's rules when it comes to doing this!

Using a Resources Folder

Putting all your assets into a Resources folder is the first/easiest way to access assets to stream in at runtime when you build the game. Unity requires that any asset that is not present in the game/scene in the editor, must be put into a "Resources" folder in order to be loaded while the game is running:

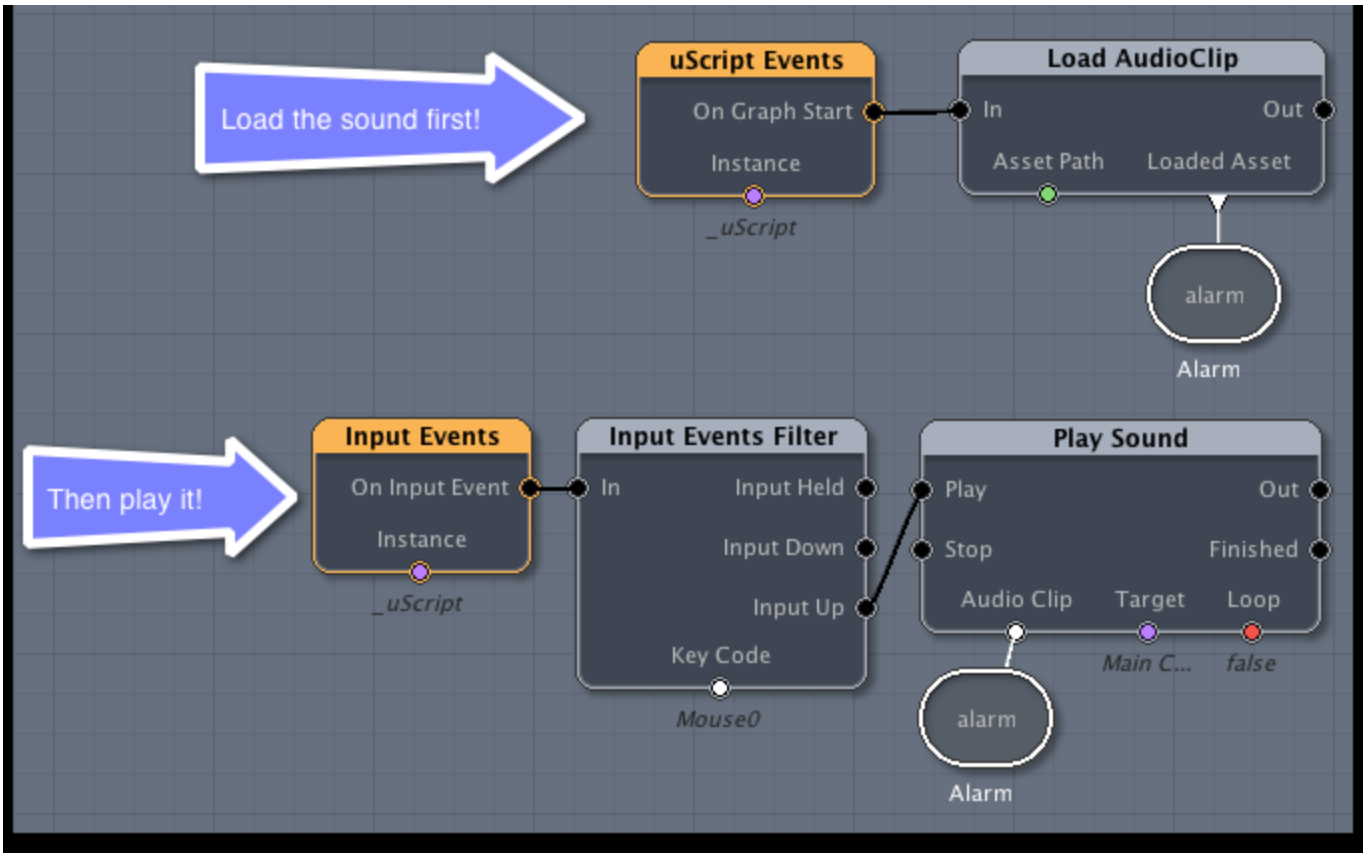


We recommend you familiarize yourself with this aspect of Unity so you understand how it works and what the tradeoffs are. You can read more about the Resources folder here in Unity's documentation:

> **Unity Documentation:** Loading Resources At Runtime - *http://unity3d.com/support/documentation/Manual/Loading%20Resources%20at%20Runtime.html*

If using the Resources folder to stream your assets into the game at runtime, the most important thing to remember is that you will need to load the asset in uScript by using a Load *<Asset Type>* node before you can use access that asset with other nodes that wish to use it.

Here is an example of a sound loaded into uScript when the scene first runs and then other logic that plays the sound whenever the left mouse button is pressed:



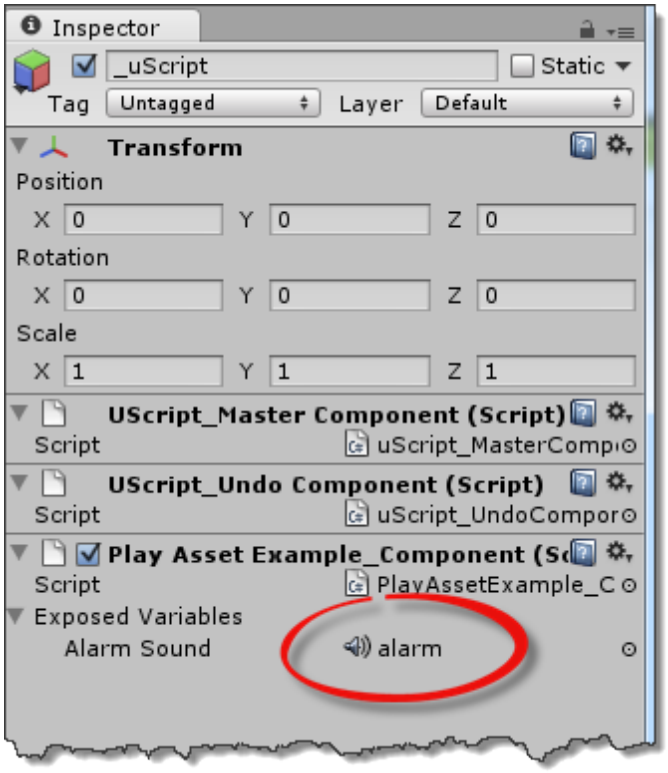
Note! - *You do **NOT** need to load animations or prefabs through a Load <Asset Type> node like other assets. This is because all animations need to be assigned to GameObjects in the editor so it is assumed if the GameObject or prefab is in the Unity scene, the animations have been loaded by Unity as well.*

Using Exposed Named Variables

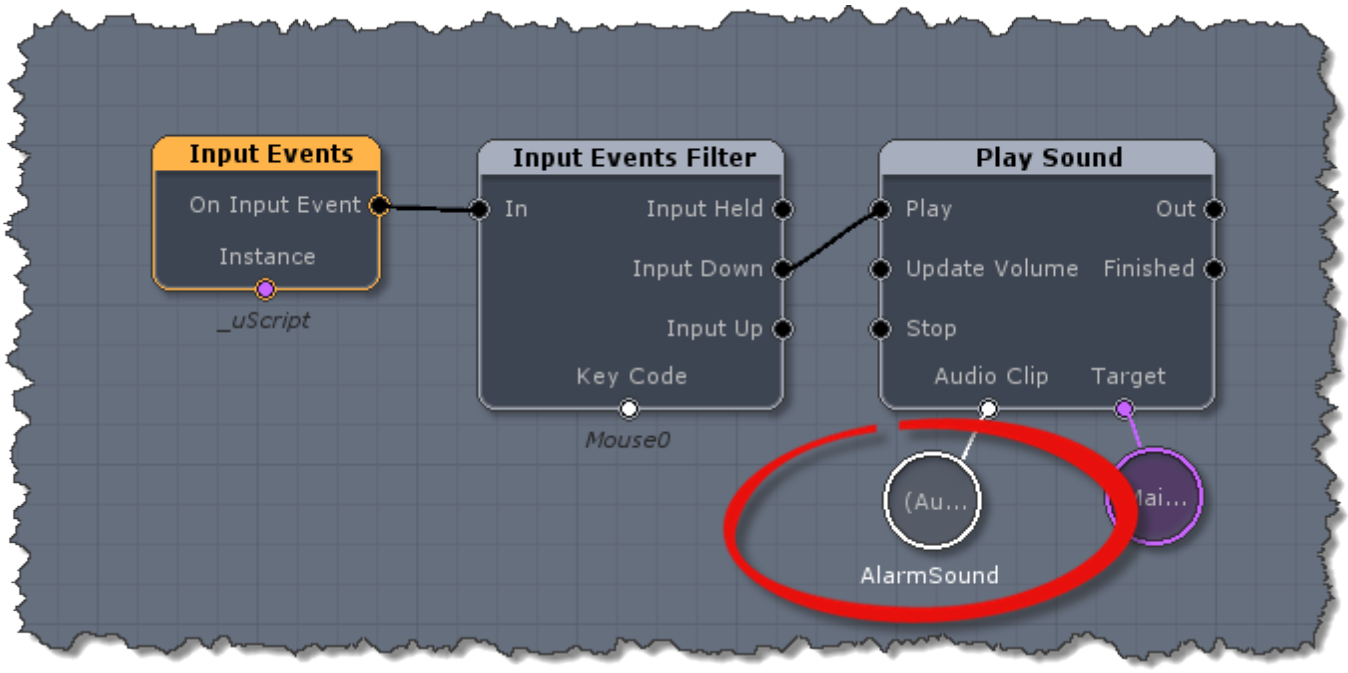
Another way you can access assets at runtime is to assign the asset to a named variable you have exposed in uScript (see Named Variables section within the "Variables" topic in the Working With uScript section of this guide for details regarding exposing Named Variables).

When an asset has been assigned to an exposed variable, Unity knows that it should package up that asset when building the game, so there is no need to manually load that asset from a Resources folder beforehand. This also means that you do not need to have your asset in a Resources folder at all when using this method.

Here is an example of assigning an Audio asset to an exposed Named Variable on a graph called *PlayAssetExample* that is assigned to the _ *uScript* GameObject (an AudioClip variable called *AlarmSound*):



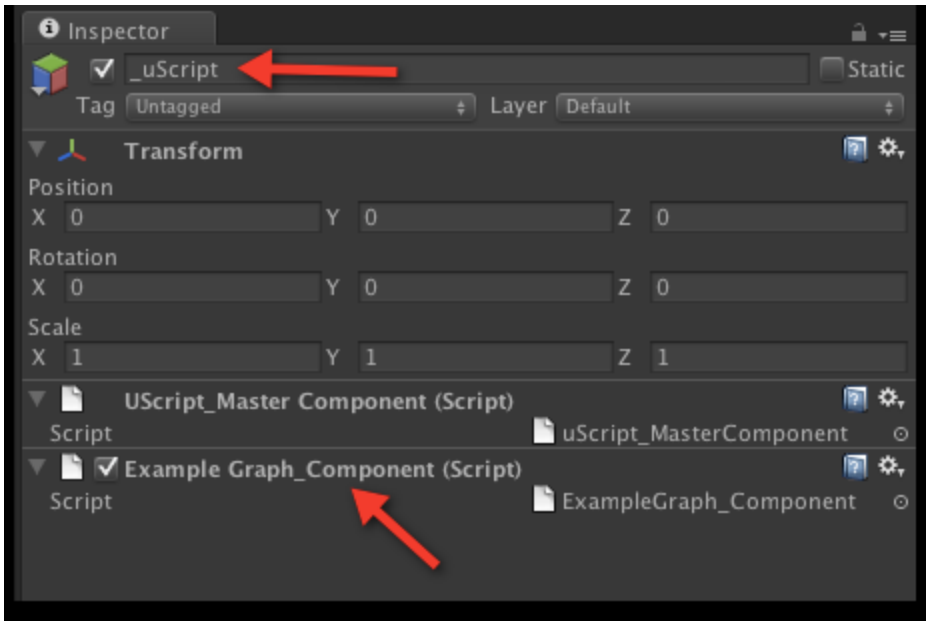
Now that the AudioClip is assigned, we can just tell Unity to play it whenever we like without the need of having to load the asset beforehand:



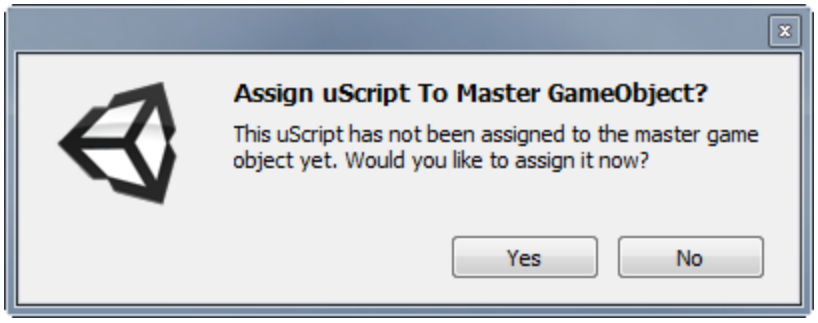
_uScript Master GameObject

uScript needs to make a GameObject for itself that contains key uScript components. The default name for this GameObject is "_uScript". For your graphs to work in your scene, they must be assigned to a GameObject. In most cases they can just be assigned to the master uScript GameObject unless you are doing something specific like Nested or Prefab uScript graphs (see the "Graphs" topic in the Working With uScript section for more information about graph types).

In this case the graph we saved was called "*ExampleGraph*". uScript will generate a component script file and append "_Component" to the end of the name. This graph component is what is assigned to the GameObject for the uScript graph to be active in the Unity scene:



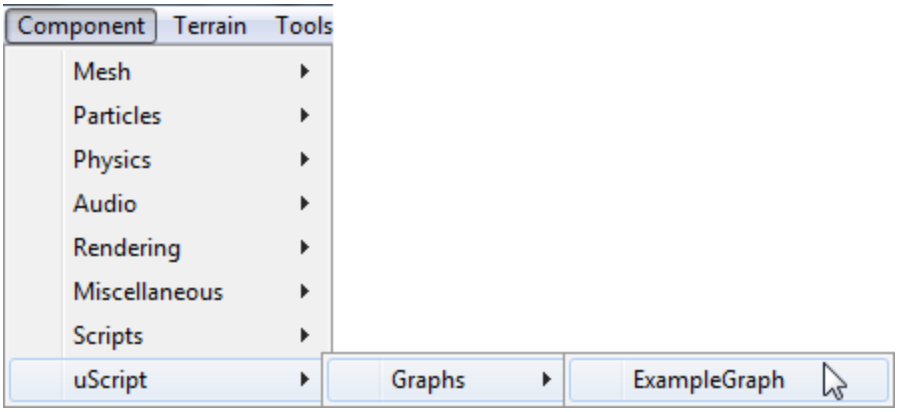
When you first save a new graph, uScript will ask is you would like to attach it to the master GameObject. Just press Yes to have the graph's generated code component to be assigned automatically. If you choose No, then uScript will not assign your graph to any GameObject and you will need to manually assign it to your GameObject of choice.



Note! - *Selecting No is useful for graphs that you wish to assign to Prefabs or not have assigned in the scene at all (like for Nested Graphs).*

If for some reason you wish to manually assign your graphs to the _uScript master GameObject (or any other GameObject in your Unity scene), just locate the generated code component file (it will be called *YourGraphName_* Component.cs) and drag it onto the GameObject's inspector panel. You can

also assign the graph component from Unity's Component menu. Just go to the uScript/Graphs section and choose the graph component you want to assign.



uScript Files

uScript currently generates three files for each uScript graph you create. These files will be located in your project's root assets folder in *uScriptProjectFiles/uScripts/* by default. Let's assume you create a new uScript graph called "MyGreatGame"—these are the files uScript would generate:

- **MyGreatGame.uscript** – This is the "master" binary file that uScript uses to save your uScript graph. 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 code files below at any time. Sometimes you will need to regenerate the code files from this file when uScript has been updated and you receive compile warnings or errors from Unity.

uScript uses the above master graph file to automatically generate the following code files whenever the graph is saved (with the exception of Quick Save) or "Rebuild All Scripts" is used from the file menu in uScript. These files are what Unity actually uses when your game is run:

- **MyGreatGame_Component.cs** – This is a "wrapper" output C# script file. This is the component code that would be assigned to a GameObject in your Unity scene, such as the uScript master GameObject (see above), in order to have your uScript graph active in the scene. This file is automatically built/rebuilt by uScript from the .uscript binary file above.

- **MyGreatGame.cs** – This file is where all the “magic” is. This is the file that contains your entire uScript graph as pure C# code. This file is automatically built/rebuilt by uScript from the .uscript binary file above.

  Note! - *you may notice a slight pause after saving a uScript graph because, when the C# script files are generated, Unity will recompile the script files to update them with your changes. This is why "Quick Save" in uScript can be handy-- it allows you to save changes you have made to the master .uscript file without causing uScript to regenerate the C# script files. Just make sure you do a full save before trying to run your game so that the latest code files are generated!*

Uniquely Named GameObjects

Currently uScript needs to be able to tell which specific GameObject you want it to use by its name. This means that you should have a unique name for any GameObject you wish to use or reference in uScript, as that is the only way uScript can identify a specific GameObject between Unity sessions (a session means quitting and then running Unity again). While Unity does support unique IDs internally for all GameObjects in your scenes, unfortunately these unique IDs are not persistent. This means that we cannot rely on the GameObjects to have the same ID numbers in between closing and reopening Unity, leaving the only choice for the user to make sure they uniquely name their GameObjects they wish to use in uScript.

As an example, if you place three point lights in your scene, Unity will name them all "Point light" by default. If you wish to reference one of those lights in uScript, you should give it a unique name so that uScript can find the correct light you wish to use.

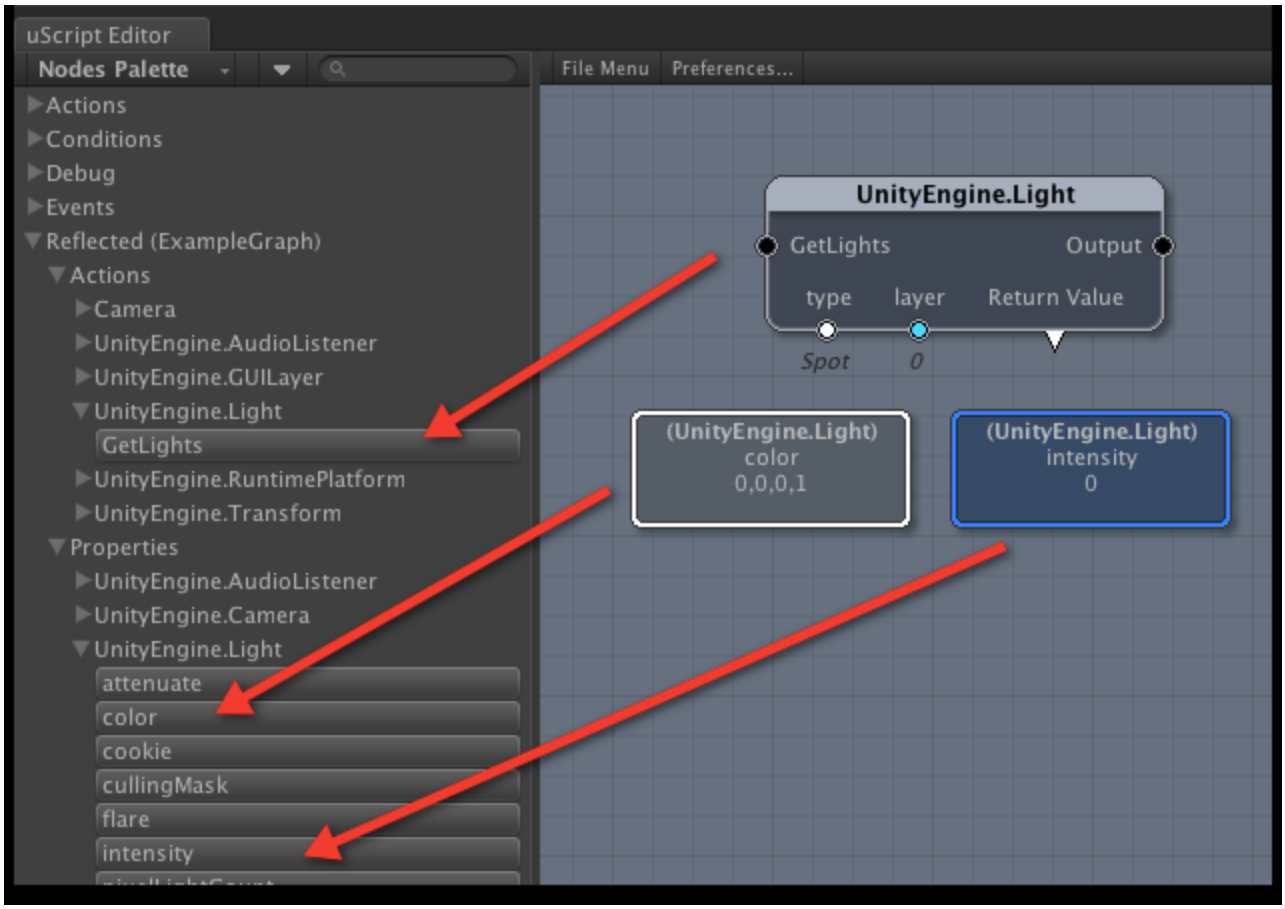
Reflection

uScript will use .Net Reflection in order to visualize many aspects of your existing GameObjects, Components and Scripts (C#, javascript, and Boo) in Unity. In the case of scripts created in either javascript or Boo, you may need to place the scripts inside a "Standard Assets" folder in your Unity project because of the order in which Unity compiles scripts. C# scripts do not have this restriction.

It is important to note that uScript can only reflect things that are part of active GameObjects in your Unity 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 GameObject (see above) to them so uScript knows what specific GameObject you want to use.

All reflected nodes can be found in the Nodes Palette under the "Reflected()" section. If you have a Unity scene with a name, the scene name will be shown between the parentheses - Reflected(*YourSceneName*).

This image shows an example of auto-created nodes made through reflection for some aspects of a light GameObject in the scene. To reiterate-- if you do not have a light in your Unity scene, these light nodes generated through reflection would not show up for you. You must have a light in your scene for them to appear:



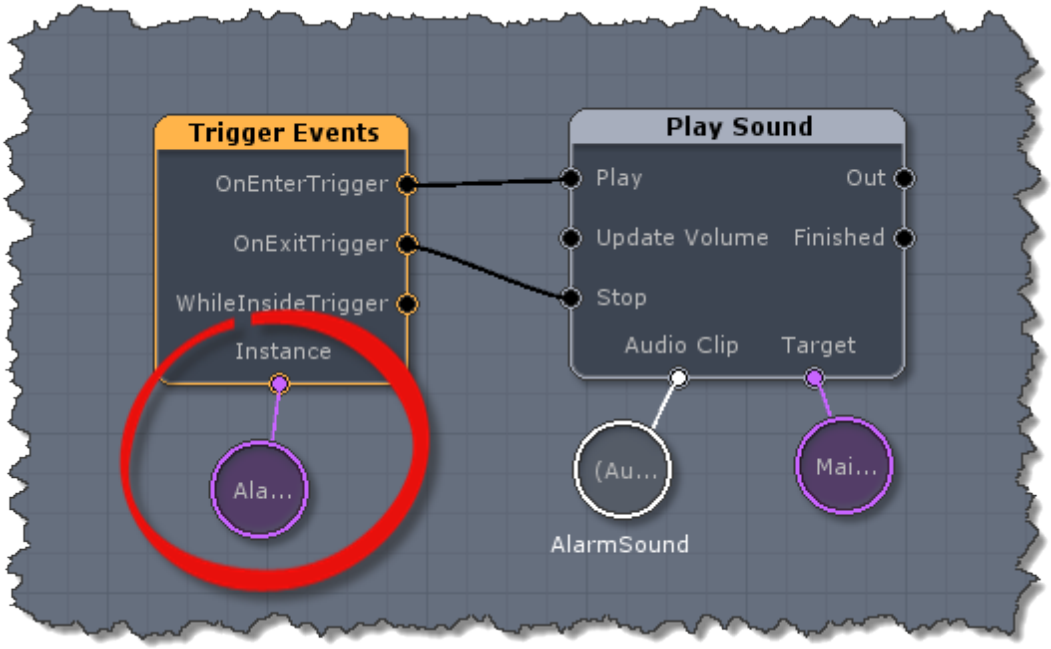
Note! *- the bottom two "square shaped" colored nodes are properties of the GameObject and can be used just like variables. These two would let you change the light's intensity and color. Again, you need to assign the Instance of the light GameObject you wish to effect in the node's properties or uScript won't know which light in your scene you wish to affect!*

Instances

uScript's Event Nodes and many Reflected Nodes require you to assign a specific GameObject to it so that uScript knows what object you want to fire that event or read/write a reflected property. In many cases, for the more global event nodes, using the master *_uScript* GameObject if fine (and uScript will auto-assign the -uScript master GameObject by default when placing many Event Nodes). For other nodes however, you want to tell uScript exactly what GameObject is firing an event.

It is also important to note that for many event types, Unity requires that you have a certain component on your GameObject to work correctly. As an example, if you want to have a trigger, you need to make sure that a Collider component assigned to that GameObject in Unity and that it has the "Is Trigger" checkbox checked.

Using the Trigger Events node as an example, you need to tell uScript what trigger GameObject in your scene you want to make fire that event (you could have hundreds in your scene!). This is done by assigning an instance (a specific GameObject) to the property of the event. In the below image, we are assigning a GameObject trigger called "AlarmTrigger" to the Trigger Events node. This means that when the player enters the specified trigger (AlarmTrigger), the event will fire and cause text to print to Unity's console. No other trigger you may have in your the game will cause this event to fire.



Note! *- in many cases, uScript will not allow you to assign a GameObject as an instance for an event node if it does not have the correct component assigned to it. Keep an eye on the Unity console output for uScript warnings if you try to assign the wrong thing.*

Graphs

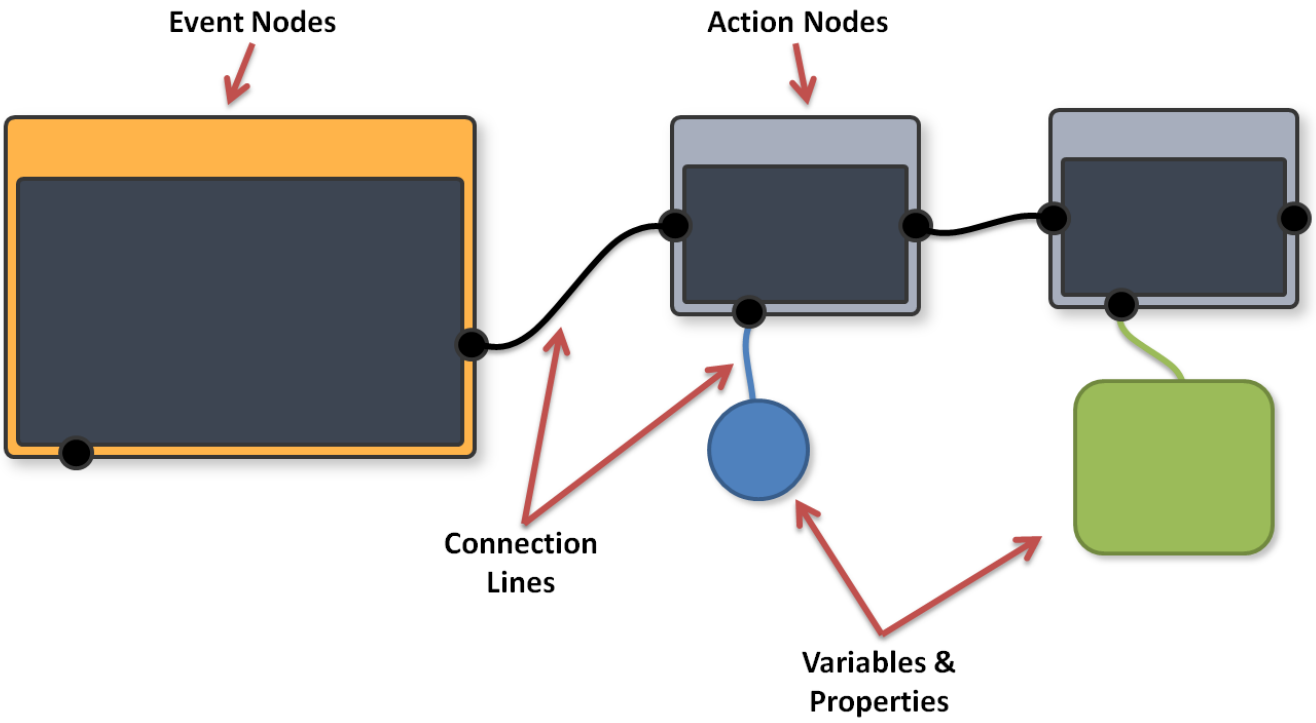
A graph is the term used to describe the collective visual logic you create in uScript on the Canvas. A single graph can have many separate sections of visual logic that are used to do a variety of things while the game is running. A uScript graph is all the logic that is contained within a single uScript file (**.uscript*) you have created with the uScript Editor. Another words, each uScript file you make is considered a uScript graph.

Common usage examples include:

> *"Load up the **graph** and let's take a look at the logic."*
> *"Save the **graph** and run the game to see what happens."*
> *"You need to re-save your **graphs** using debug mode so we can use breakpoints."*

Anatomy Of A Graph

A uScript graph is made up of several different "parts" that are used in combination to create a graph:



Nodes - Almost everything you can put on a graph in uScript is considered a "node". While there are many different types of nodes (see "Nodes" for more information on the

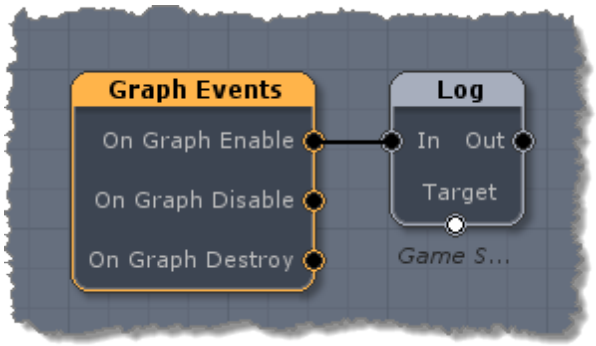
different types of nodes), there are a couple of critical types required to make working logic in a graph:

> **Event / Action Nodes-** In order to have a working graph, you must have at least one Event node (orange) and one Action node (gray) connected together. All graph's **must** contain at least one Event node to execute graph logic.
>
> **Variable / Property Nodes-** These are nodes that hold information/data. While you can make very simple graphs without them, you will most likely want to have Variables of some type so you can use external data from your Unity project. See "Variables" for more information on these types of nodes.

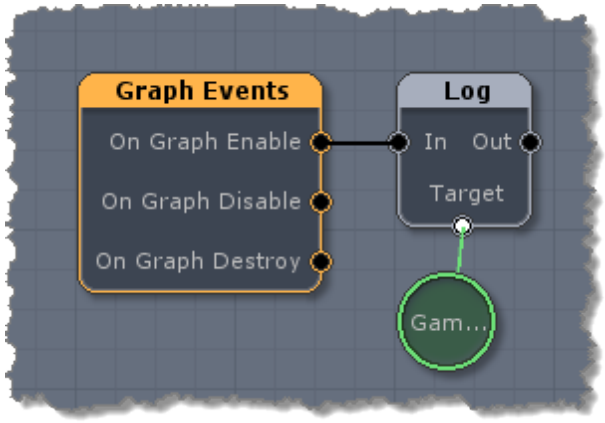
Connection Lines - All nodes must be connected together by Connection Lines in order to function/be accessed. There are two types of Connection Lines:

> ***Signal Connection Lines -*** These connection lines go between the Input and Output sockets on the left/right sides of Event and Action nodes. Think of these as electric wires that are used for activating the nodes in the chain by sending a signal along the wire that action nodes are listening for.
>
> ***Data Connection Lines -*** These are lines that connect your variables and properties to the variable sockets on Event and Action nodes. They tend to be color coded the same as the variable type they use. These Connection Lines will read or write data to connected Variables and Properties depending on if they are hooked up to a Variable Input (read) or Variable Output (write) socket on the node.

Here is an example of the smallest functioning graph you can make in uScript. It prints "*Game Started!*" once to the Unity console when run:



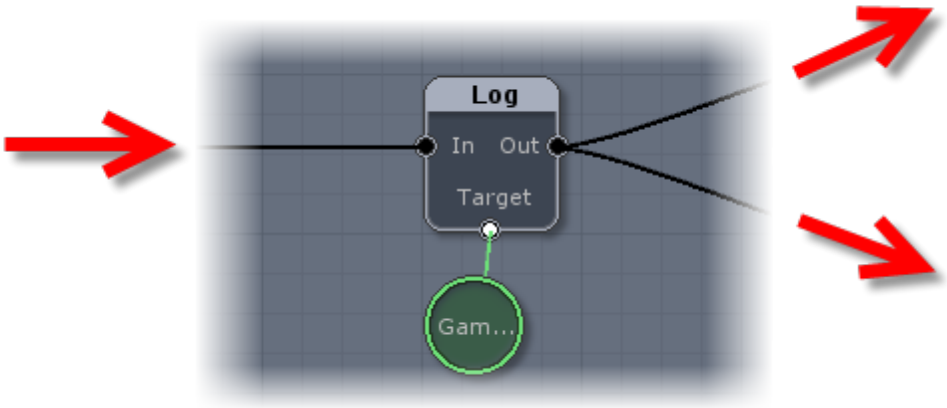
And here is the same logic, just using an external string variable node instead of specifying the "*Game Started!*" text directly in the Log node:



Understanding "The Signal"

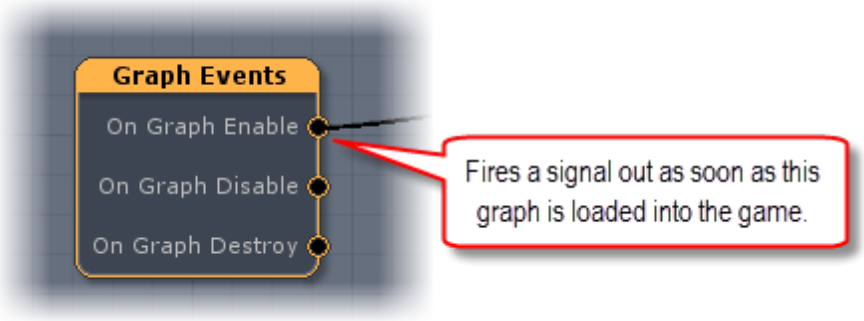
Creating visual scripts means hooking up nodes that perform very specific functions and logic in complex ways to create anything you could imagine. This is done by connecting the nodes together using Connection Lines. This allows nodes to both receive and send signals when you wish them to. The easiest way to think of these Connection Lines are as wires that will send electricity along them, powering the action nodes to do their thing.

uScript graph logic, and the signals is sends to other nodes, is done from left to right. This means that all nodes will listen for signals from Connection Lines hooked up to their Input sockets on the left side of the node and send out signals along Connection Lines hooked up to their Output sockets on the right side of the node.



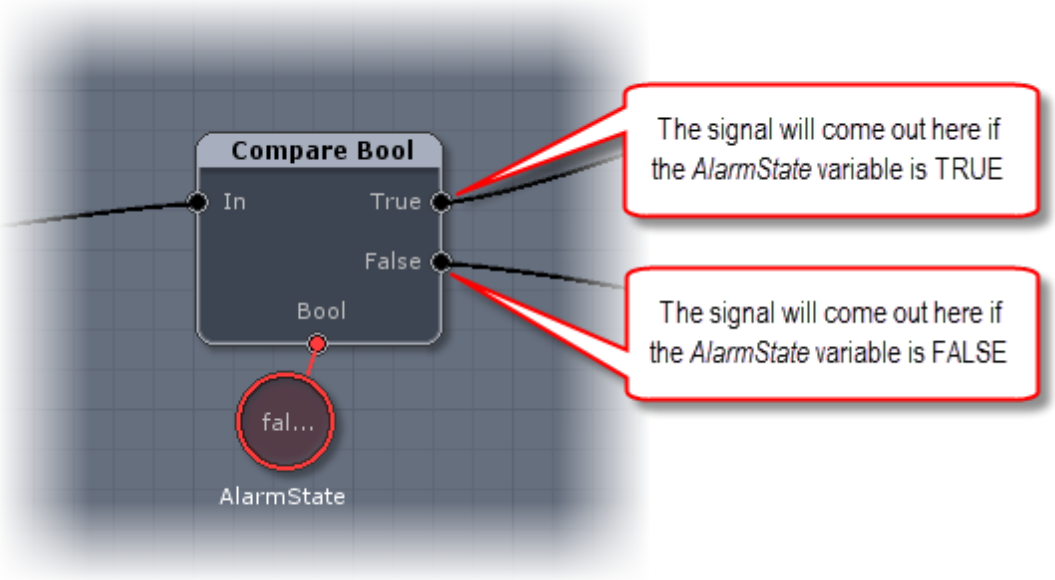
You can control when a signal is sent in several ways:

Event Nodes - All uScript logic that executes must start at some point from an Event Node. Event nodes listen for things to happen in your game and will fire off a signal whenever its specific criteria is met (like entering a specific trigger, loading a scene, pressing a button, etc.). Notice that Even nodes do not have In sockets on them. This is because they are the ones that start signals that are then sent to Action nodes. You can also think of the as the energy source of the electricity.

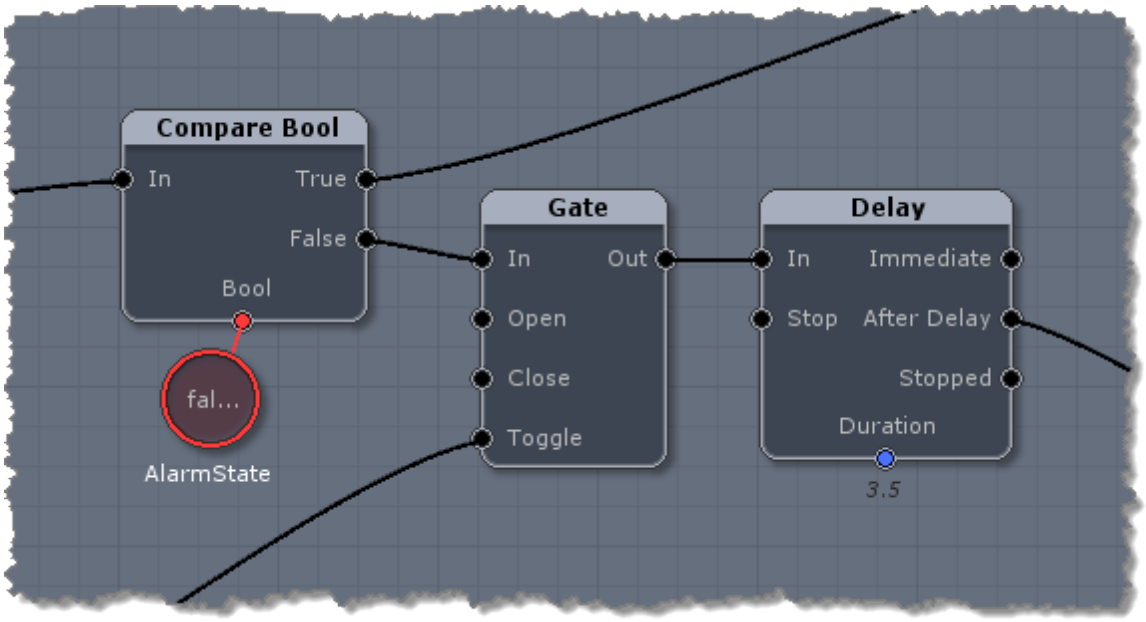


Action Nodes - Action nodes contain Out sockets that will pass along a signal. If and when this signal is passed along, depends on the conditions of the socket. For example, nodes with an "Out" socket will always pass along any signal it receives to that socket as soon as it gets the signal, where something like the "Finished" socket on the Play Sound node will only fire a signal out once the audio is done playing. See the description in the Reference Panel for each node to learn more about when any given node's different Out sockets will fire.

Condition Nodes - Condition nodes are just Action nodes that are used specifically to control signal flow from Connection Lines in useful ways commonly used when creating logic flow. They can also be used in combination to create very complex and powerful criteria to determine where and when a signal should be allowed to continue.



Using Condition nodes in combination for more complex signal control if *AlarmState* is FALSE by also checking to see if a gate is open (toggled between open and closed by other logic in the graph), and if so, then also waiting for 3.5 seconds before allowing the signal to continue on:



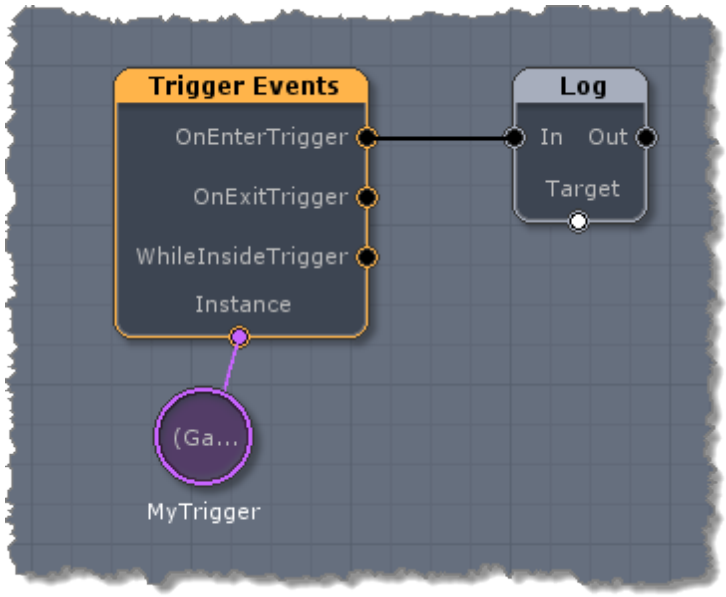
Frequency Of The Signal

How frequently a signal is sent out from an Event or Action nodes depends on the kind of node it is and the type of logic that node is executing. It is important to understand exactly how often a signal is sent and also how action nodes will react to getting multiple signals to avoid unexpected results form your logic.

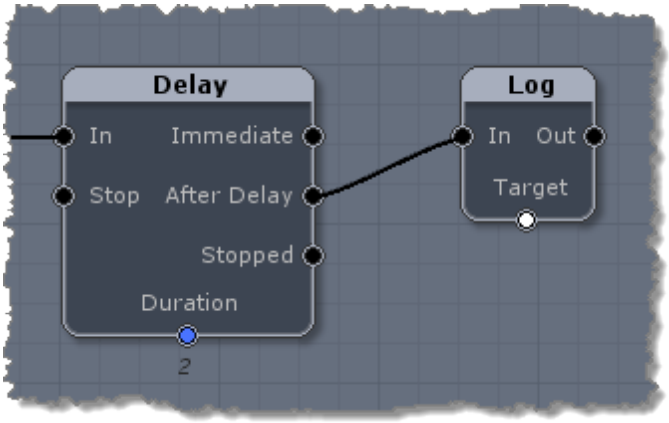
Typically, when a signal will be sent by a specific Output socket on a node fall into one of three categories:

Once - In many cases, only one signal will be sent at a time-- either right when a signal/event is received by the node, or after a delay based on other functionality/criteria of the node. We try to follow this paradigm as much as possible, though there are still many nodes that need to send a signal more frequently than this.

Here is an example of an event node that will only send out a **single** signal to the Log node each time something enters the *MyTrigger* trigger:

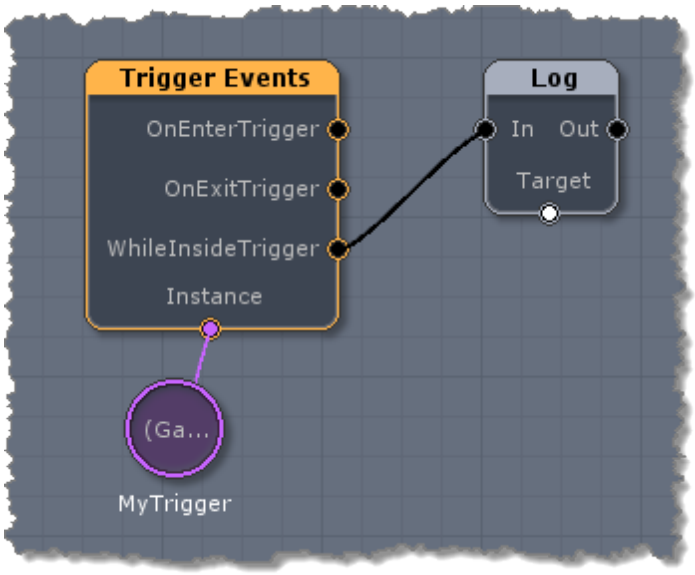


Here is an example of an Action node that will send a **single** signal to the Log node after a 2 second delay for each time it receives a signal on its "In" Input Socket:

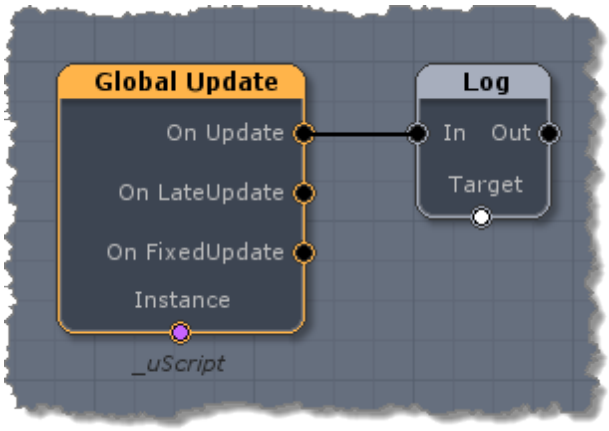


Once Per Tick/Time - The second most common case is that a signal is sent out repeatedly by a node based on every tick of the game (OnUpdate), or based on some other factor of time (seconds, frames, etc).

Here is an example of that same *MyTrigger* trigger. It will fire out a signal **every tick** of the game to the Log node as long as something is inside the trigger:

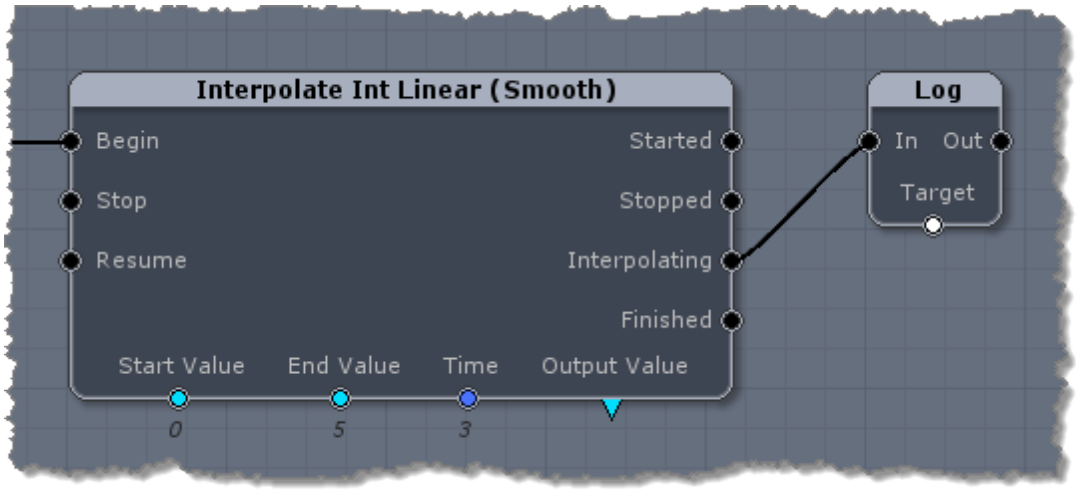


Here is an example of another Event node that will send a signal **every tick** of the game to the Log node:

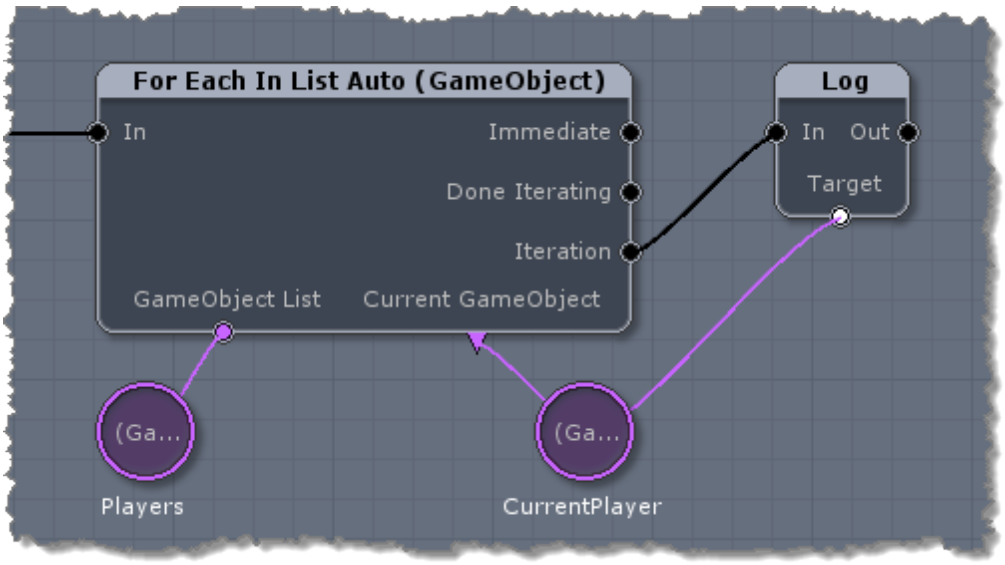


Once Per Iteration/Cycle - Some action nodes have Output sockets that will send out a signal each time something specific happens within the node while executing logic or iterating on multiple things when it receives just a single signal on its Input socket.

Here is an example of an Action node that will send out a signal to the Log node **every time it changes** the Output Value while interpolating-- even though it only received a single signal on its "Begin" Input socket to start the node executing:



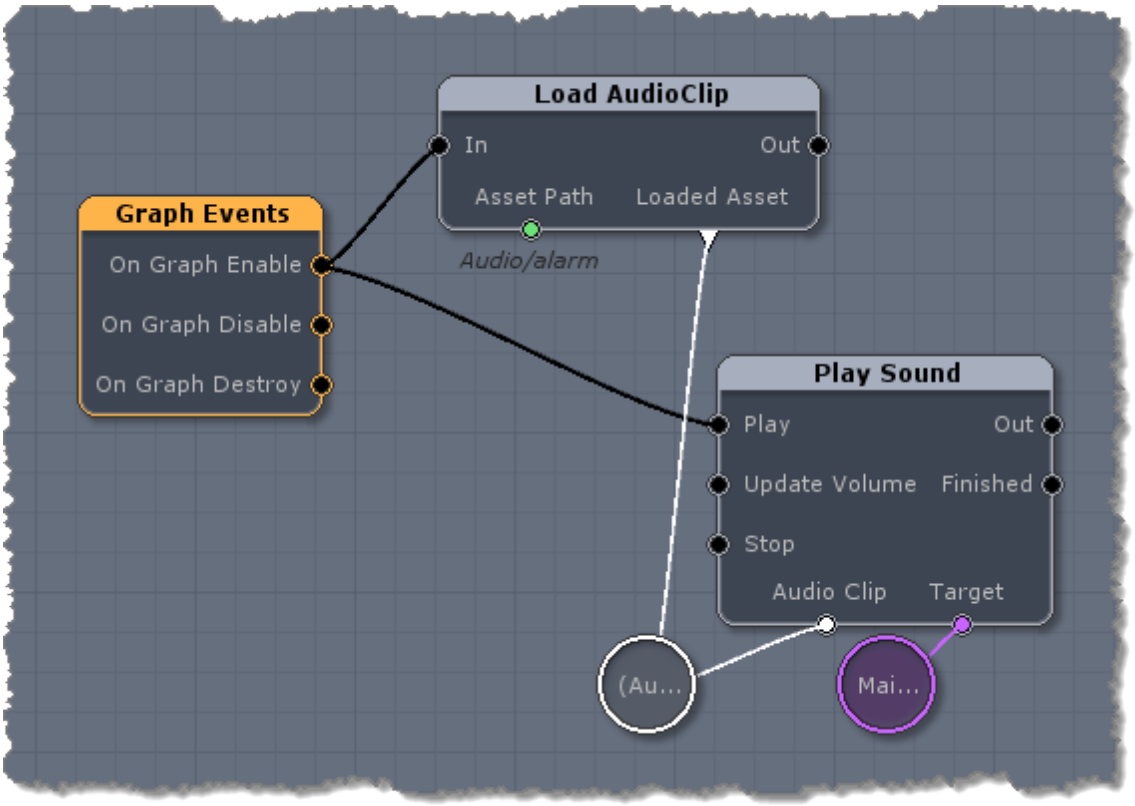
Here is an example of a For Each node iterating through all the players in a GameObject List variable. A signal is sent to the Log node once **for each GameObject** in the *Players* variable (where it prints that specific GameObject to the Unity console via the *CurrentPlayer* variable):



Execution Order

Sometimes it can be important to know exactly **WHEN** signals will reach a node and how to ensure that nodes are executed in the proper order.

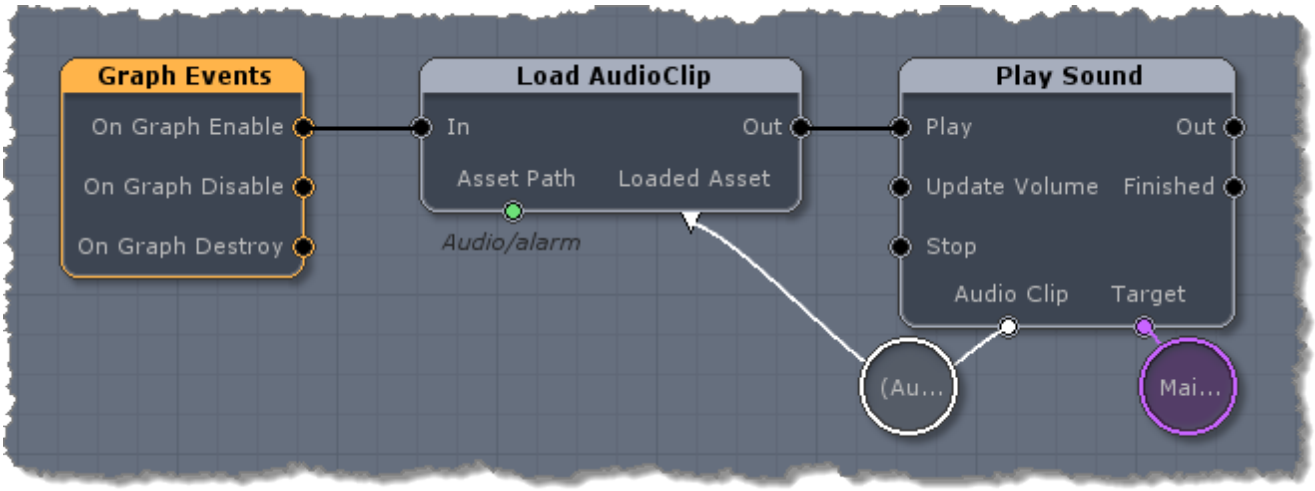
Let's take a look at this example logic:



Notice how there are two separate Connection Lines coming off of the Event node? This means that a signal will be sent down each Connection Line to its respective nodes when the game starts. Unfortunately uScript has no way of knowing in this case which node will be executed first by Unity since we just told Unity to "do both these things when you get a chance".

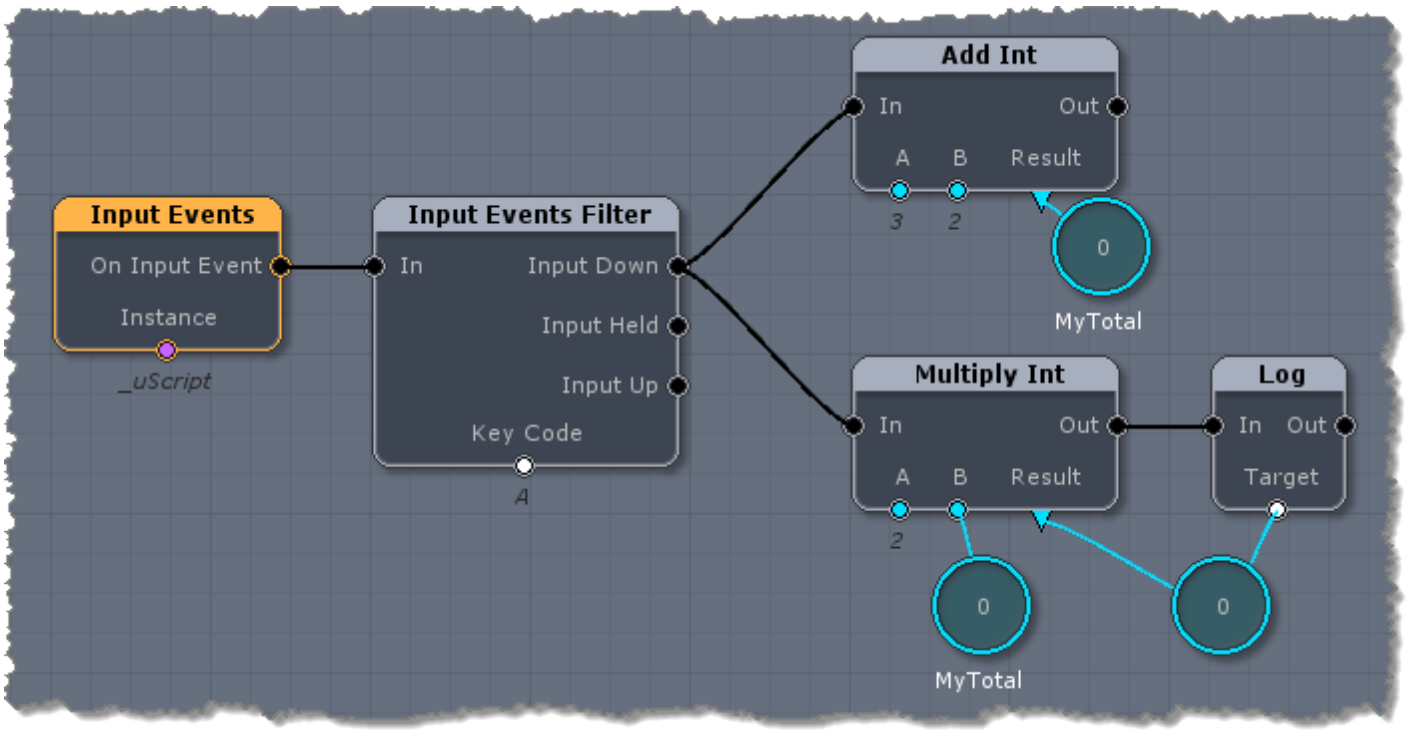
This means we could end up with a case where Unity is trying to play the sound using the AudioClip variable **before** we have had a chance to assign it with the "alarm" sound! We might get lucky and they will execute in the correct order-- but it would not be safe to leave it like this and could cause unwanted bugs later.

It is best to ensure the order of execution by linking these nodes in the order we want them to execute in. By rearranging how this logic is hooked up, we can tell Unity to "please load the sound first and **then** play it":



While the above example is a pretty simple one, you may very well run into more complex setups in large graphs of logic. Make sure you keep an eye out for cases where you might be trying to execute some logic before some other critical logic was executed.

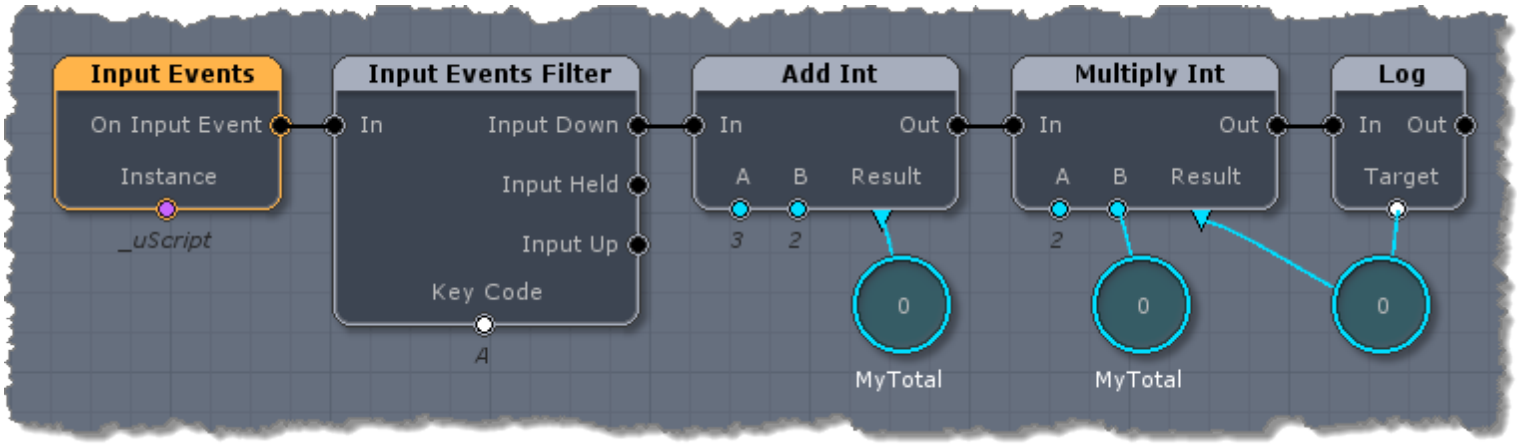
Here is another example where execution order would matter:



In the above graph you can't be sure if the *MyTotal* named variable value will be 0 or 5 when the Multiply node executes. This means you might get either

"0" or "10" printed out to the Unity console.

To ensure the answer is always "10", you should fix the logic to force the order of execution:



Graph Types

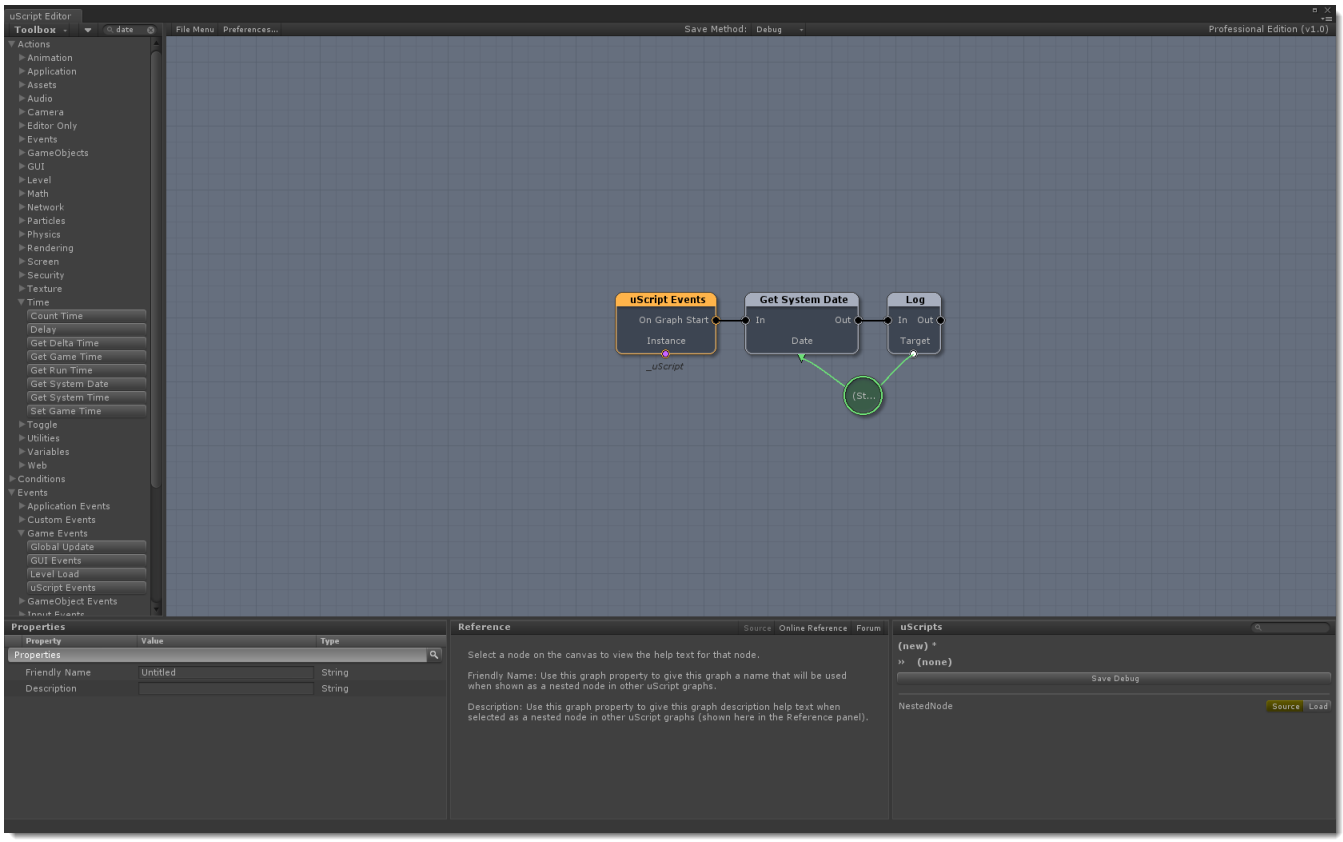
While there are no hard rules in how you use uScript, there are three primary types of graphs that you create with uScript to execute visual scripting logic-- *Scene Graphs*, *Prefab Graphs*, and *Nested Graphs*.

Scene Graphs

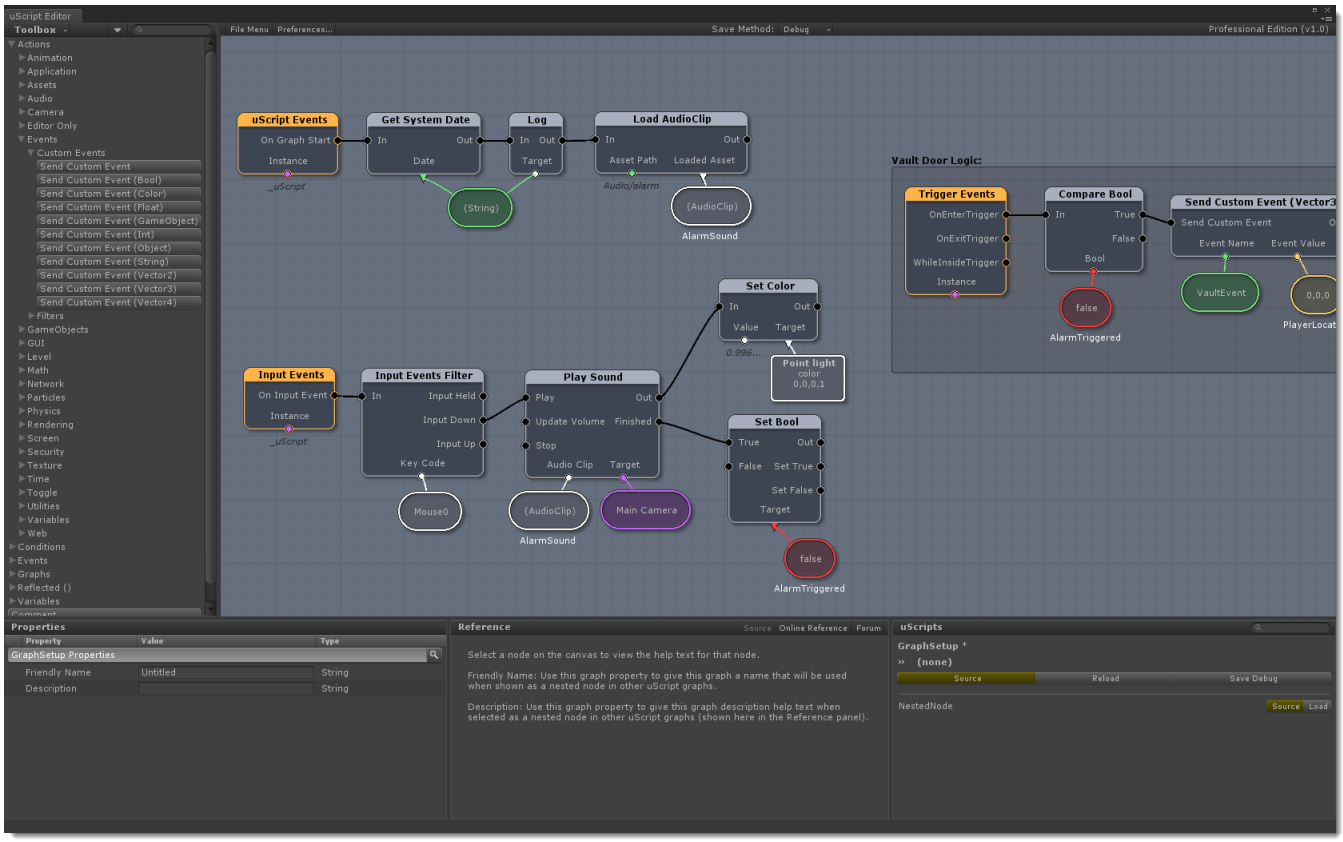
Creating a single uScript graph for your scene to perform gameplay logic is the basic way in which you use uScript. In many cases, it is fine to just have one uScript graph for each of your Unity scenes (levels) as you can manage most of your gameplay logic in this way.

That graph is usually assigned to the uScript master GameObject that it creates for your scene. Because uScript handles things within its graph, you don't end up with a lot of components and scripts cluttering up your scene's individual GameObjects.

Here is an example of a very simple scene graph:



Here is the same uScript graph with more complex logic added to it:



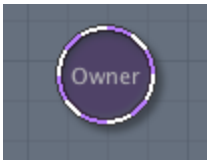
Prefab Graphs

Just like the name implies, these are uScript graphs that you can place on a prefab GameObject which will be spawned into your game at runtime. These graphs are very powerful and allow you to handle GameObject-specific logic on the GameObject/prefab itself. An example of this could be shooting a missile into the world or simple logic that opens and closes a door prefab that is placed throughout your game levels.

Your main gameplay uScript graph would be responsible for spawning the missile when the player hits the fire button for example, but once it is spawned you may have missile-specific logic you want to have on the missile itself so that when it collides with something it will play particle effects, sounds, add explosion forces all by itself-- and then destroy itself from the scene when it is done.

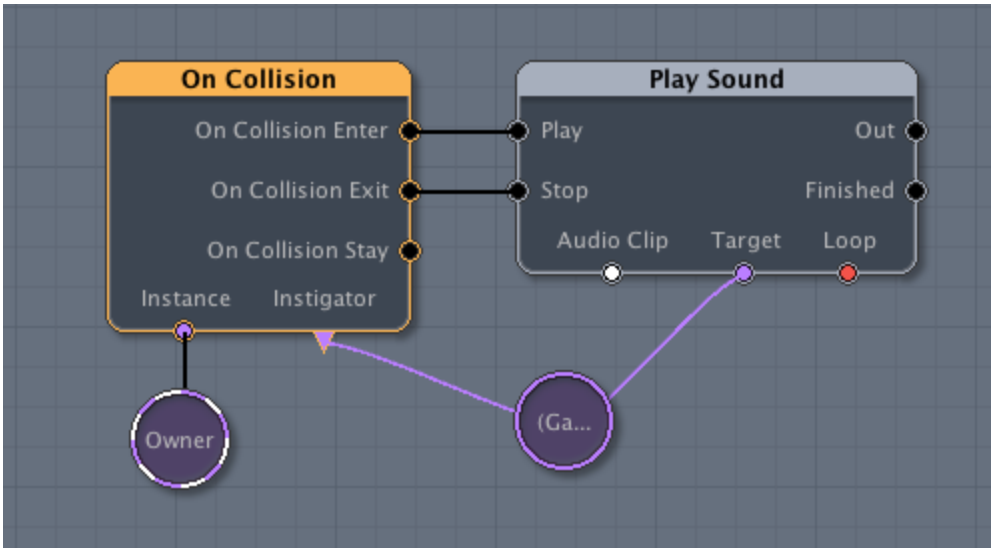
This means each instance of the prefab you spawn in the world will automatically run its own uScript graph logic and you don't have to worry about keeping track of them individually in the main uScript gameplay graph.

In order to create uScript graphs for prefabs, you will want to use the special "Owner GameObject" variable whenever you need to assign a node's Instance property to the prefab. This special variable tells uScript to use whatever GameObject the uScript graph is assigned to.



> ***Note!*** *- for those of you more familiar with scripting, the Owner GameObject variable is the equivalent of "**this**" in many programming languages. It simply tells uScript "Use whatever GameObject this graph's script component is on.".*

Here is an example Prefab Graph using the Owner variable:



Nested Graphs

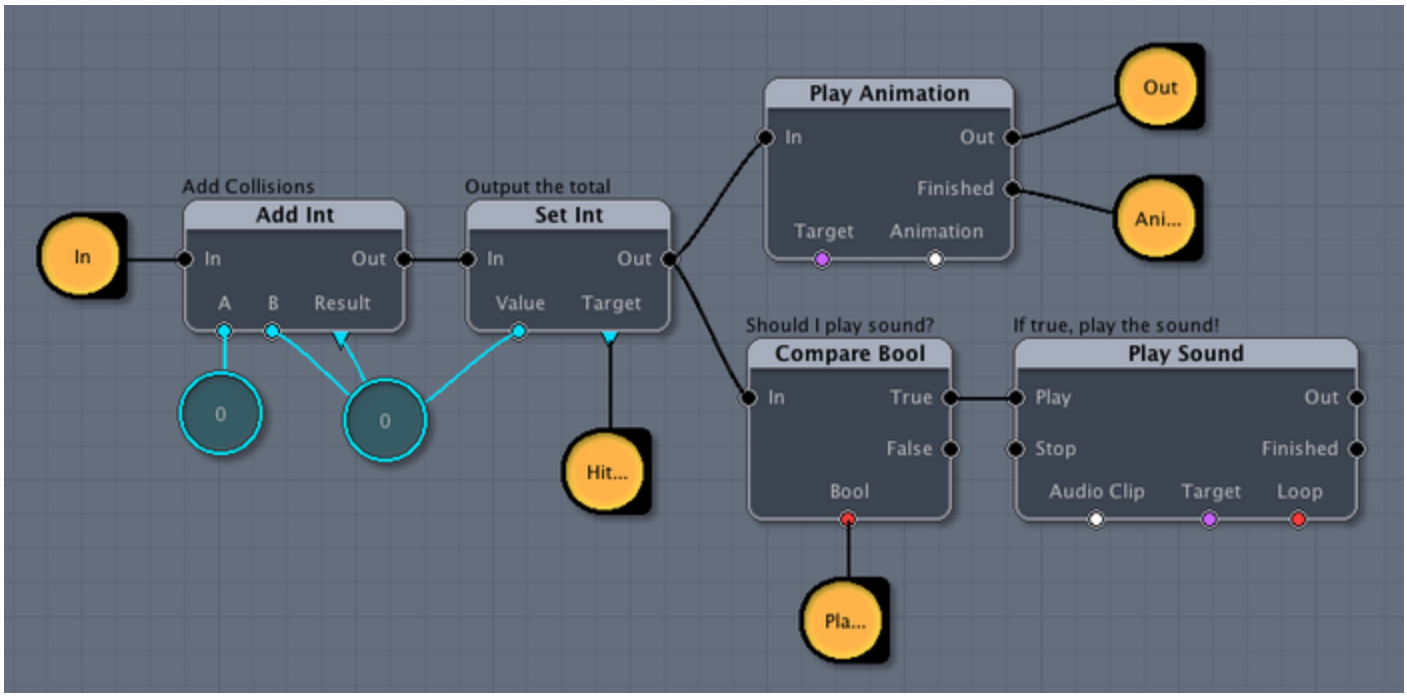
uScript can also be used in another way to help organize and extend the functionality of uScript itself by letting you create your own custom nodes

right from within uScript. Special uScript graphs can be created that will allow non-technical users to take complex visual logic and create their own custom action node from it. We call these *Nested Graphs* because all the complexity of the graph is hidden away behind a single node that is then used in other uScript graphs.

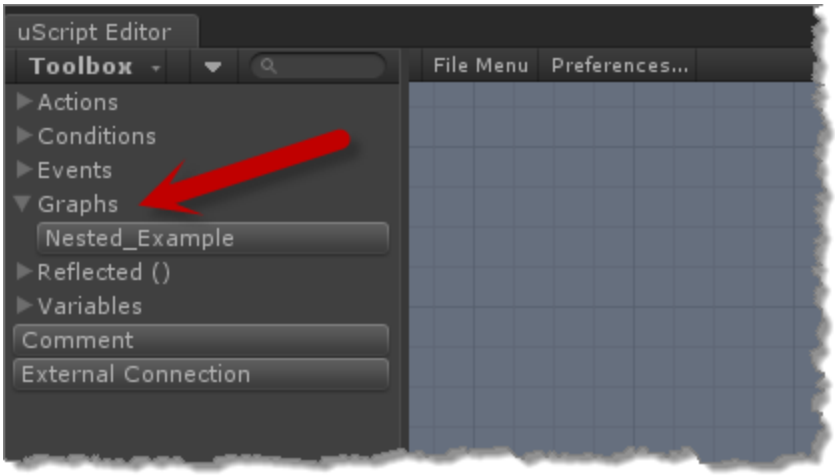
Another way to think of them are as recipes that perform a task that you may want to reuse in different parts of your project or even in completely different projects. You can very easily build up a library of these recipes for yourself and to share with other team members or on the http://www.uscript.net.

In order to make input/output/variable sockets appear on the nested uScript node on your graph, you need to use External Variables within the nested uScript hooked up to the sockets you wish exposed. By using these and naming them, they will create sockets on the nested uScript node.

Here is an example of creating a Nested uScript saved as "Nested_ Example.uscript":



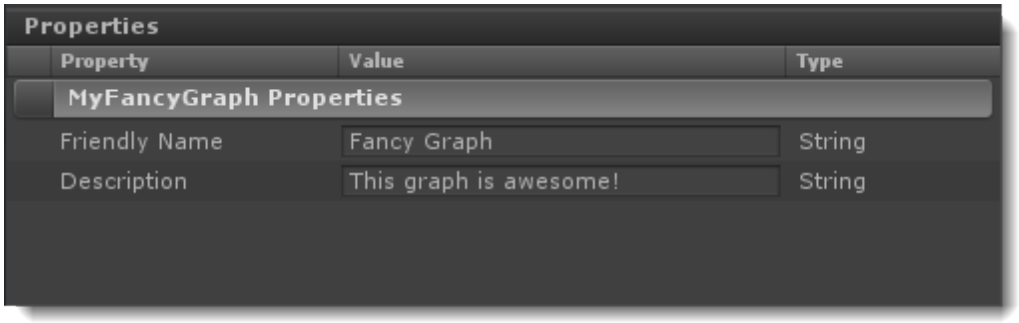
Graphs that contain External Connection nodes will appear in the "Graphs" section of the Nodes Palette and can be placed as nodes in other graphs. The "Graphs" section will not appear if no uScript graphs in your project contain an External Connection:



And here is what the node created from the nested graph looks like when placing it inside another uScript graph:

Graph Properties

Graphs have some properties that you can set on them. These properties are not really used by uScript unless you are making a Nested Graph (see *Nested Graphs* in this section).

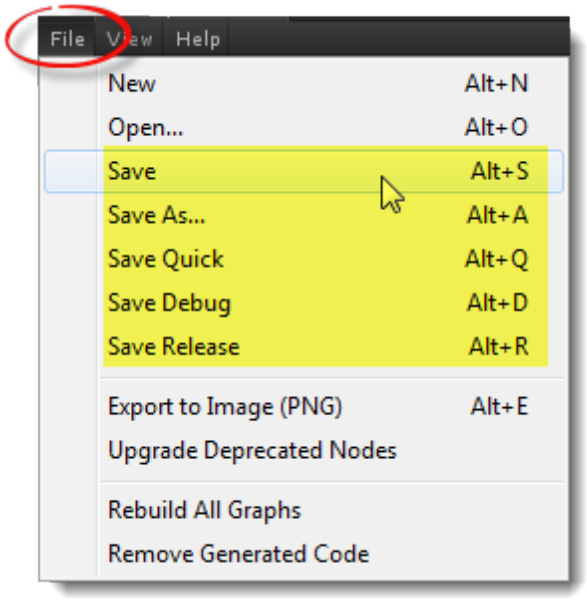


Friendly Name - This lets you set a friendly name for the graph that will be used as both the node title and at the top of the Reference Panel when the nested graph appears as a node on other graphs.

Description - This lets you set a description that represents the nested graphs functionality. This will be displayed in the Reference Panel when the nested graph appears as a node on other graphs and is selected.

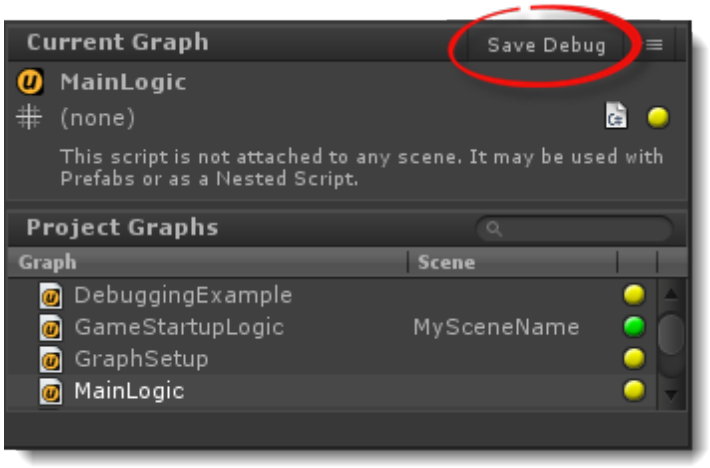
Saving Graphs

Saving your graphs is fairly straightforward. To save the currently open uScript graph, just go to uScript's *File Menu* above the Canvas and choose the save option you wish to use (also note the Save hot-keys assigned to each save option for fast access to the save features).



Note! *- Using Unity's File menu Save Scene option (or the Ctrl+S hot-key for this option), will* ***NOT*** *save your uScript graph-- just the open Unity scene. You must use one of the save options or hot-keys inside of the uScript editor as mentioned here to save your graph.*

Optionally you can press the Save button in the ***uScripts Panel***, shown here:



When saving your uScript graphs, you have three options in how you wish to save:

Debug - This is the default save method and allows you to use uScript's

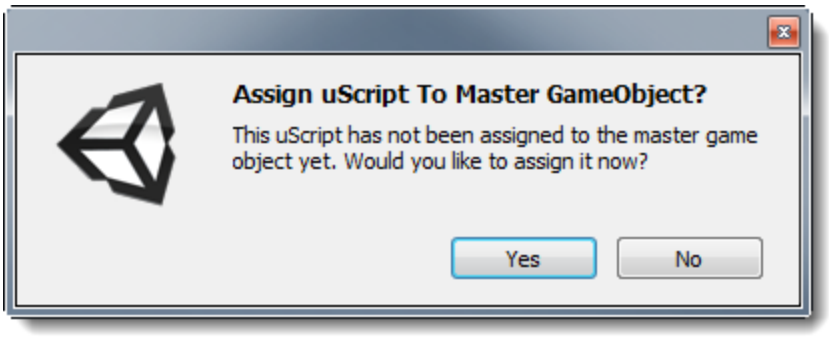
advanced debugging options such as setting breakpoints in your graph logic. Files saved as debug tend to generate slightly larger script files because of the extra debug code added when the scripts are generated.

Release - This creates the most optimized and smallest file size. You should use this save method when you are ready to ship your software. Please note however that you can not use the breakpoint debugging features for files saved this way as all the debug support code has been removed from the generated script files.

Quick - This allows you to very quickly save just your main *.uscript* graph file without regenerating the script code (and therefor avoiding Unity's automatic recompiling of all scripts). This can be handy when you just want to make sure you are saving your work while in the middle of graph editing. Please note however that since uScript is **not** regenerating the actual script code from changes you are making, you will **not** see these changes when you run the game until you have done a full debug or release save of the graph.

Saving to the uScript Master GameObject

When saving a new uScript graph for the first time and generating its script, it will ask you if you wish to assign that graph's generated script component to the Master GameObject.



Whether you wish to assign this new graph's generated script to the Master GameObject really depends on the type of graph you wish to create (see above).

Yes - Select "*Yes*" if you are creating a typical *Scene Graph* where you don't care what GameObject the graph is assigned to. and you wish to have easy access and management of your scene graphs by having them assigned to the same GameObject. Any graph that you choose "Yes" for will also remember the name of the Unity Scene it was saved to. This is used to help warn you if you later try to use this graph in another

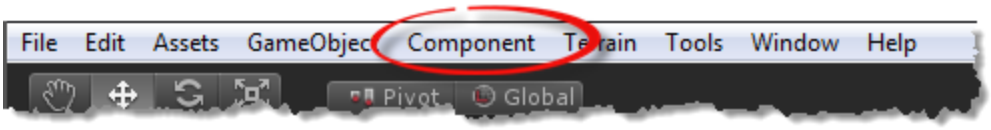
Unity scene. This is done because many scene graphs rely on specific data and GameObjects to exist, and another scene may not have the proper data and GameObjects to allow the graph to function properly. If you are new to uScript and are just making a graph with logic you want to work in your scene at runtime, chances are you want to assign it to the Master GameObject.

No - Select "*No*" if you are making either a *Prefab Graph* or a *Nested Graph*-- or any other time you wish to manually assign the graph to a specific GameObject in your Unity scene.

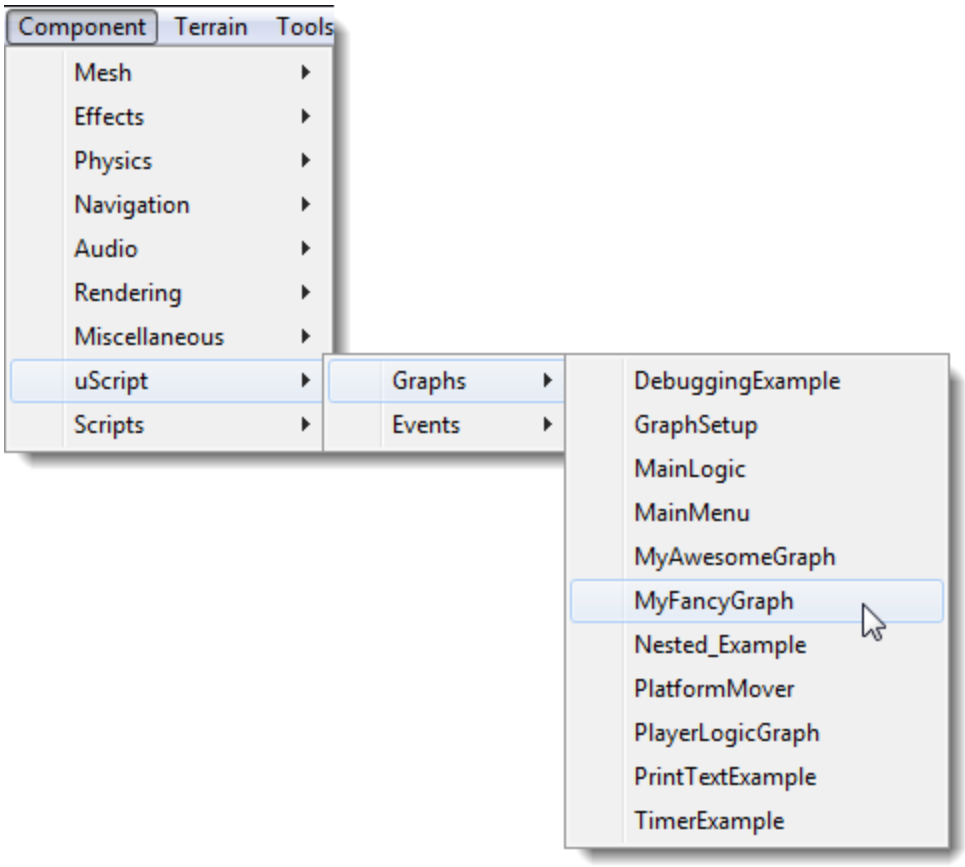
> ***IMPORTANT!*** *- All uScript graphs **must** be assigned to a enabled GameObject in your Unity scene in order to run. graphs that are **not** assigned to a GameObject in your Unity scene will be excluded from Unity when it builds your game.*

Manually Assigning Generated Script Components

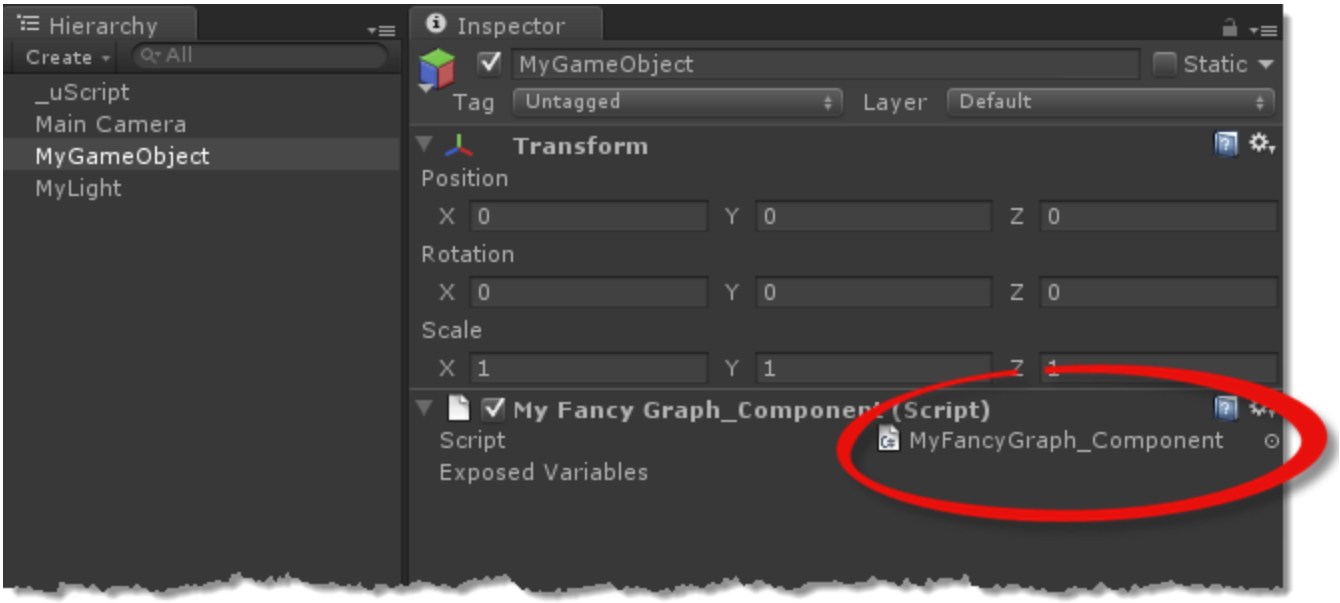
You can manually assign a uScript graph component to any GameObject in your scene by using Unity's "*Component*" menu:



Just open up the "uScript" section of this menu and go to the "Graphs" section:



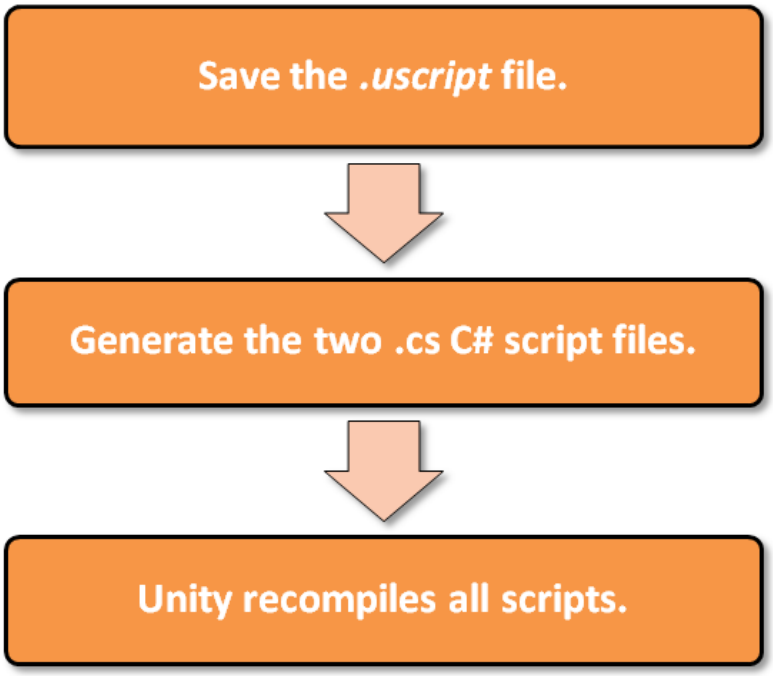
Once the graph's component has been assigned, you can see it in Unity's Inspector panel for that GameObject:



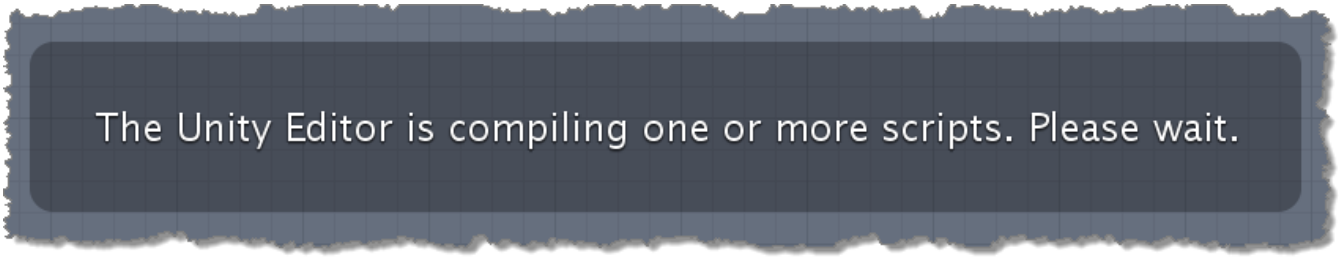
Generating Script Code

When saving your uScript graph files, uScript will automatically generate pure C# code files for use by the game at runtime (unless you choose the "Quick" save method). While there is nothing you need to do or worry about regarding these generated code files, you will see that Unity will recompile all scripts once the new or updated script file has been generated by uScript. This is normal and Unity's default behavior whenever a change, edit or deletion of a script file in the project has been made.

The full order of operations when saving a uScript graph are as follows:



Whenever Unity is compiling scripts, you should see the following message in the uScript Editor window. Just wait for Unity to finish and this message to go away before continuing to use the uScript Editor:



Also see "Script Generation".

Using the Graphs Panel

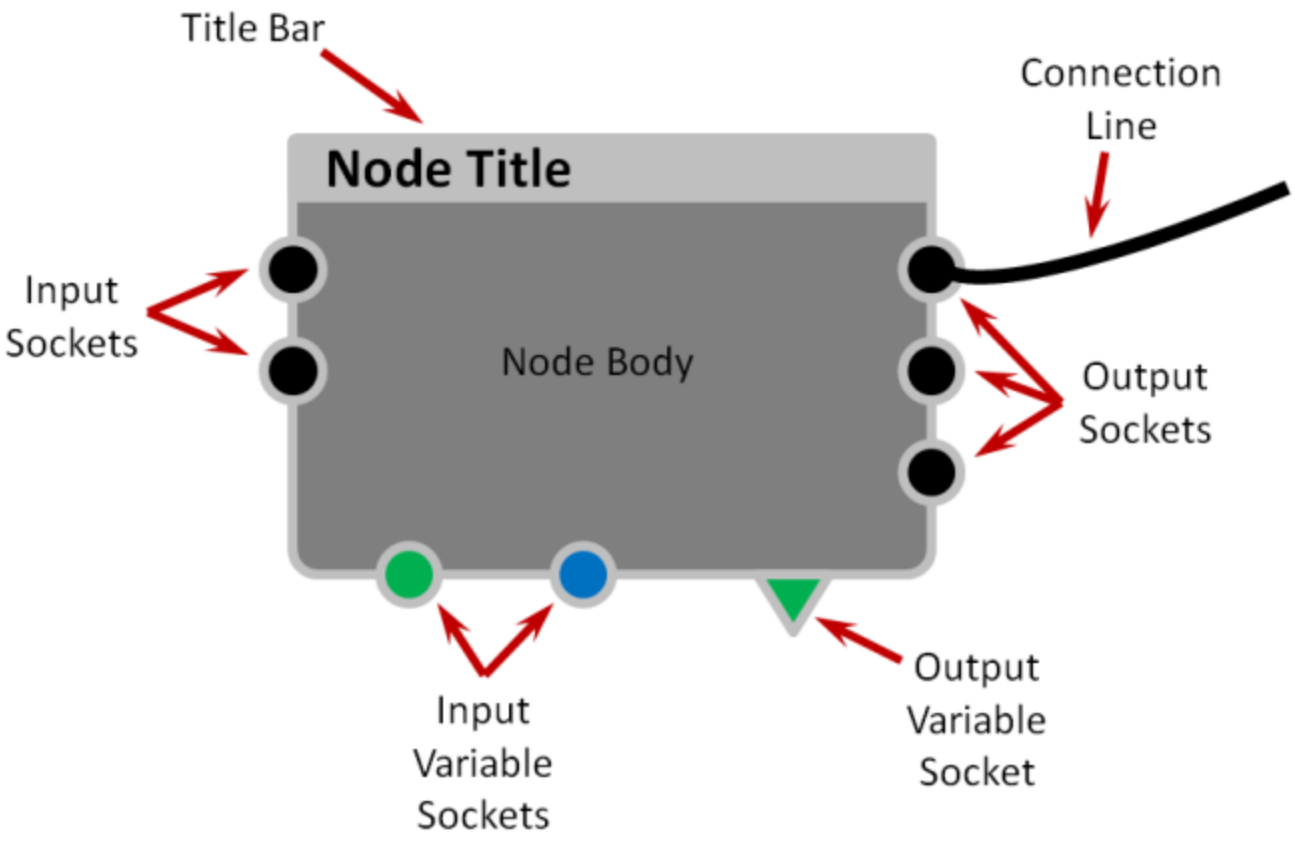
You can quickly view your projects existing uScript graphs by browsing them in the uScript Panel form within the uScript editor. This panel allows you to view, load and save your project's uScript graphs.

Please see the *Graphs Panel* section of "Editor Interface" for details on this panel.

Nodes

Anatomy Of A Node

With the exception of Variable nodes (see "Variables"), all nodes in uScript have the following components:



- **Node Color** - The color of the node border will determine the type of node it is. **orange** for Event Nodes and **gray** for Action Nodes.
- **Title Bar** - The name of the node is displayed here.
- **Node Body** - The main body of the node where the text for each socket is displayed.
- **Connection Line** - While not technically part fo the node itself, connection lines are what connect a node to other nodes and variables to create complex logic by sending a "signal" through the connection that nodes are waiting for to perform their action.
- **Sockets** - Connection lines are drawn between nodes and connect to their sockets:

- ***Input Sockets*** - The sockets on the left side of a node that receive a signal telling the node when to execute its logic. Please note that Event nodes do not have Input Sockets as they are the source of a signal.
- ***Output Sockets*** - The sockets on the right side of a node that send out a signal to any other nodes connected to it with connection lines.
- **Variable Sockets** - There are two types of variable sockets that can optionally be along the bottom of a node:
  - Input Variable Sockets - These sockets read data **in** from a connected variable.
  - Output Variable Sockets - These sockets send data **out** into a connected variable.

Node Types

uScript has three major types of nodes that you use in combination to make a functional uScript graph. For more detailed information on how these node types all work together to great a visual script, please also see "Creating Graphs".

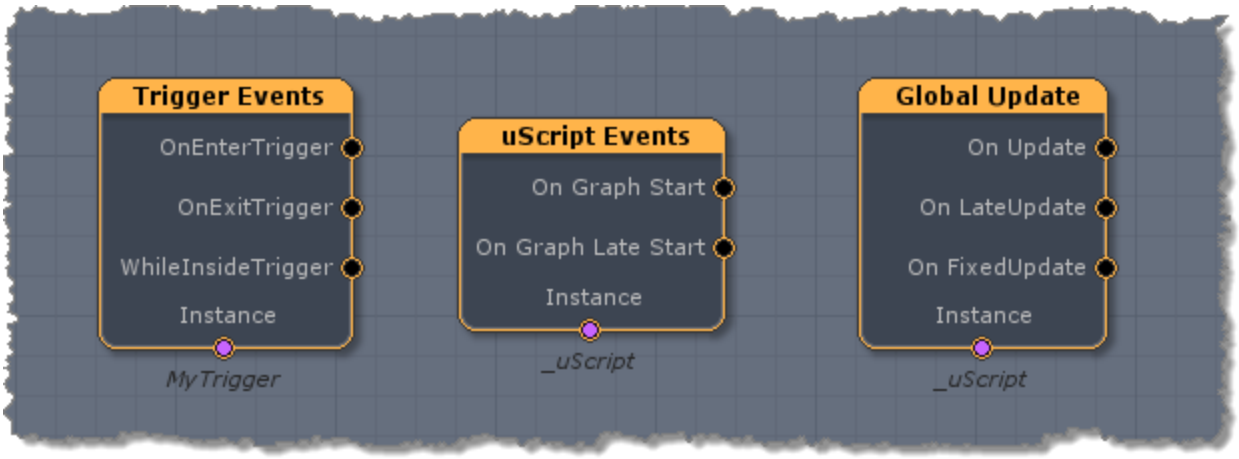
Event Nodes

Event nodes are the driving force behind all uScript logic. uScript is an event driven system that allows you to execute logic when events happen in the game. Event nodes can be found in the "Events" section of the uScript Toolbox.

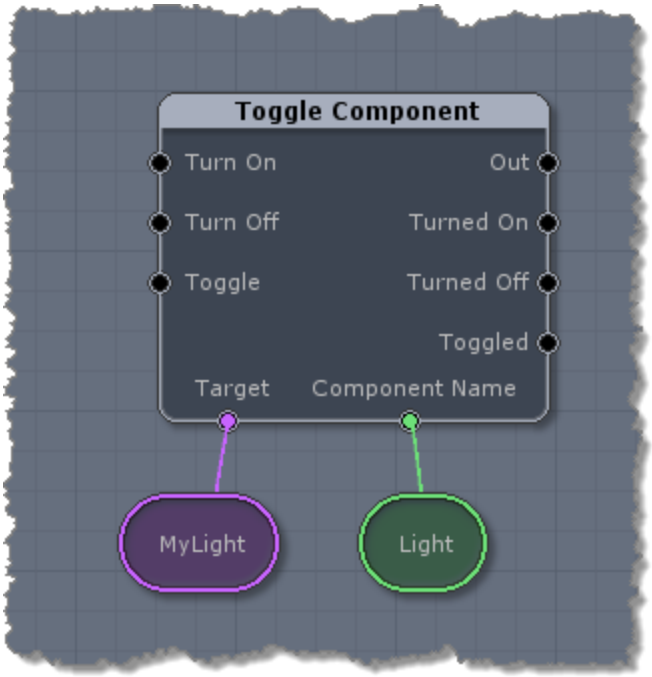
Rules of Event Nodes:

- All graph logic **must** at some point originate from an Event Node if it is to ever execute.
- Event Nodes are **orange** in color.
- Event Nodes have no Input sockets on the left of the node-- they only have Output and Variable sockets.
- Event nodes **must** have their Instance property assigned to a GameObject in order to function. Sometimes uScript will automatically assign the Master GameObject to the node's Instance property, but most of the time you must assign a meaningful GameObject. An example of this would be to assign a specific trigger GameObject when using the Trigger Events node.

Example Event Nodes:



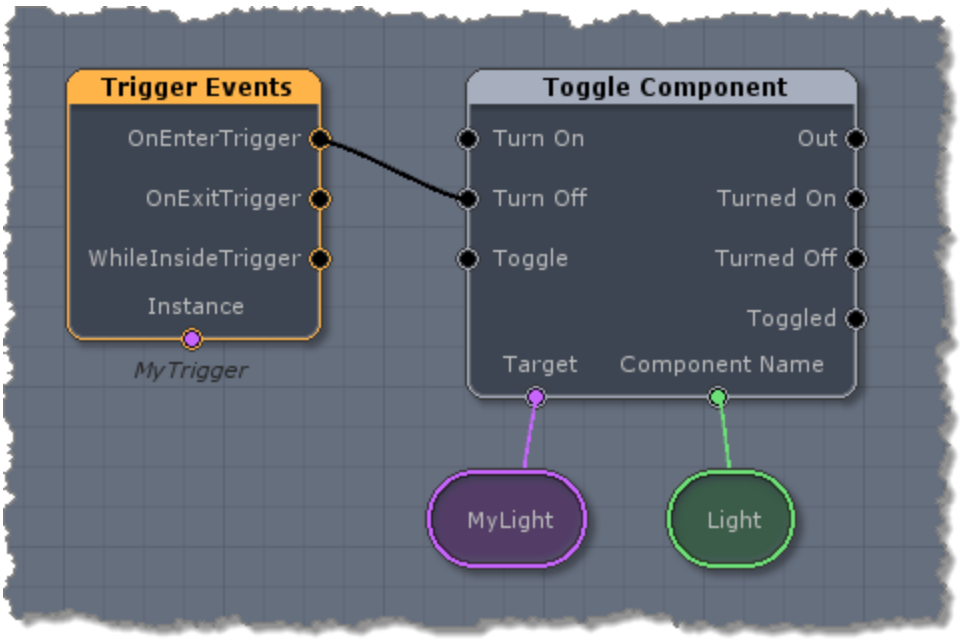
How to use them:
The event node(s) you choose determine when and how any logic hooked up to them will execute. As an example, let's take a look at some very simple logic that turns off the Light component on a GameObject called "MyLight":



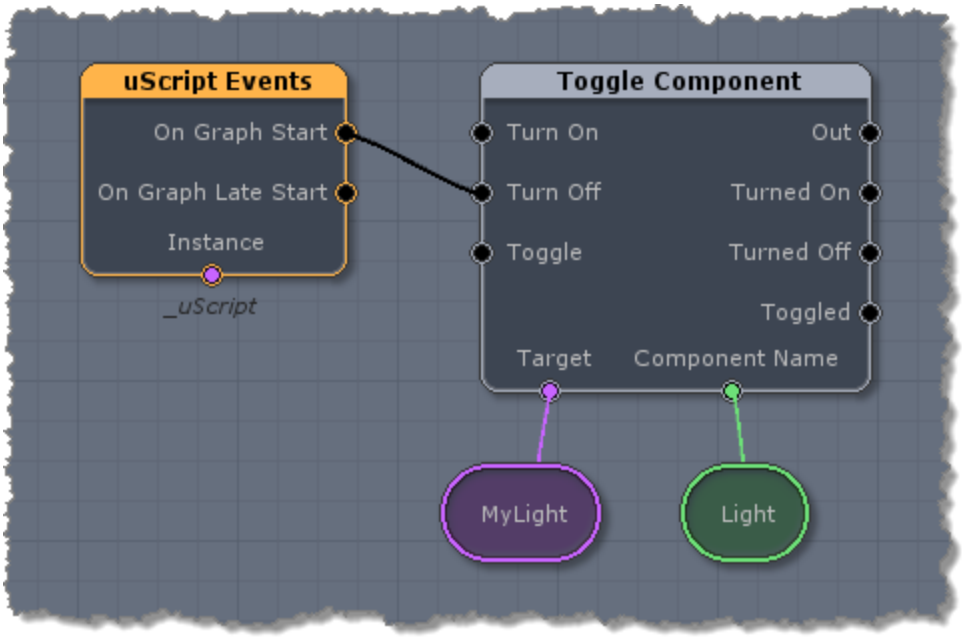
The above logic will never run because there is no event telling it **when** to turn the light off. In order to do that we would need to hook up an Event Node to its "Turn Off" socket (because we want to turn the light off in this example).

We have many choices of events that can help us determine exactly when we want the above logic to execute. Here are three examples using different Event Nodes.

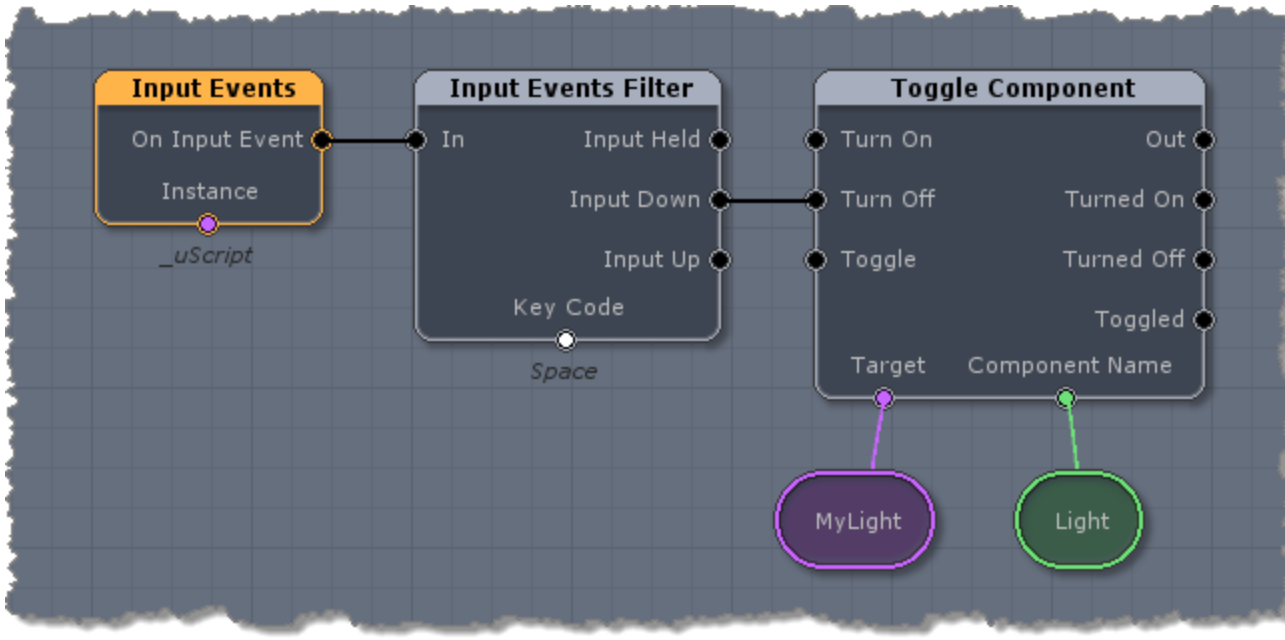
When entering a trigger:



When the graph first runs:



When the player hits the *SPACEBAR* key:



IMPORTANT!- You do not need to hook up an Event Node to every action node directly. You are free to chain action nodes together, but in order for any of the action nodes to receive a signal, there must be an Event Node somewhere at the beginning of the chain.

The following is perfectly fine to do. All these actions nodes will fire in order once the Event has happened (when the graph first loads in this case) and sends out a signal to the first node in the chain:

Action Nodes

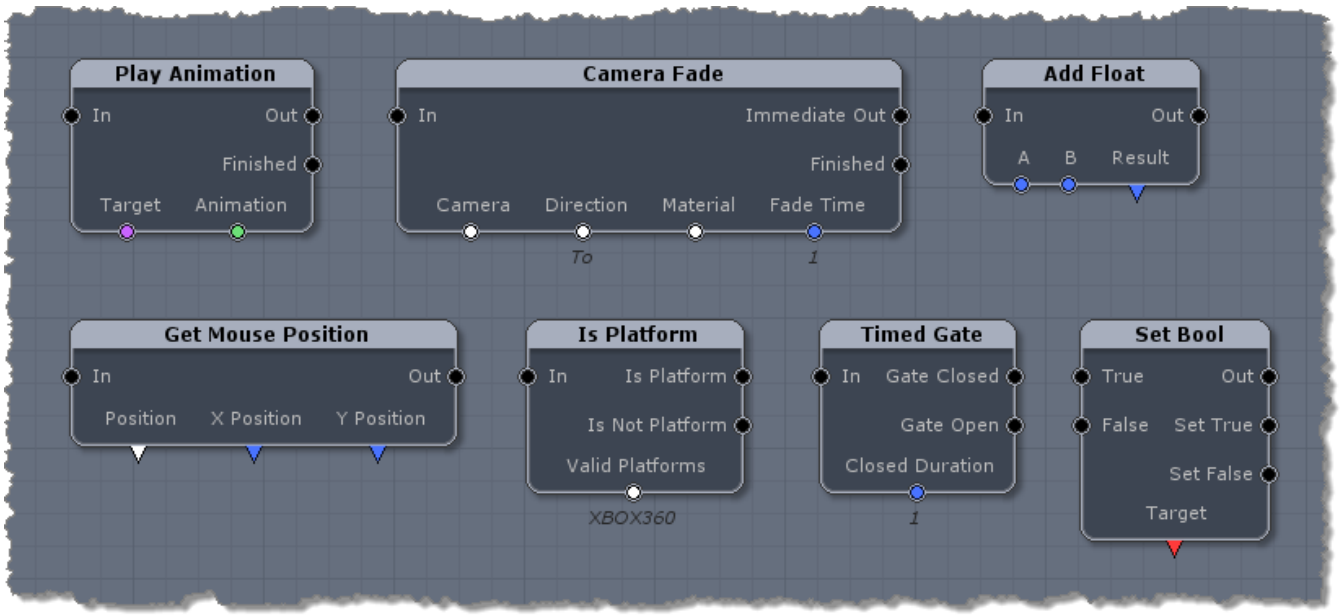
Action Nodes are the "meat" of a visual scripting graph. Each Action node performs a small bit of specific logic. You combine these nodes and their simple actions in different combinations to create complex logic for use in your game. Think of Action Nodes as toy building blocks of simple shapes that you can use in any combination you can imaging to build a complex structure.

Action nodes can be found in the following sections of the uScript Toolbox-- *Actions*, *Conditions*, *Graphs*, and *Reflected* (*reflection is not supported in uScript Basic*).

Rules of Action Nodes:

- Action Nodes must receive a signal on an Input Socket in order to execute
- You can have as many Action Nodes connected together as you like.
- Most Action Nodes have a default "Out" Output Socket that will pass the signal it received along regardless of the node's own execution.
- Most Action Nodes have Variable sockets along the bottom that allows you to expose that node's properties onto the graph for reading/writing of the data at runtime. Usually these sockets can be hidden or shown by checking on the checkbox to the left of the property in the Properties Panel.

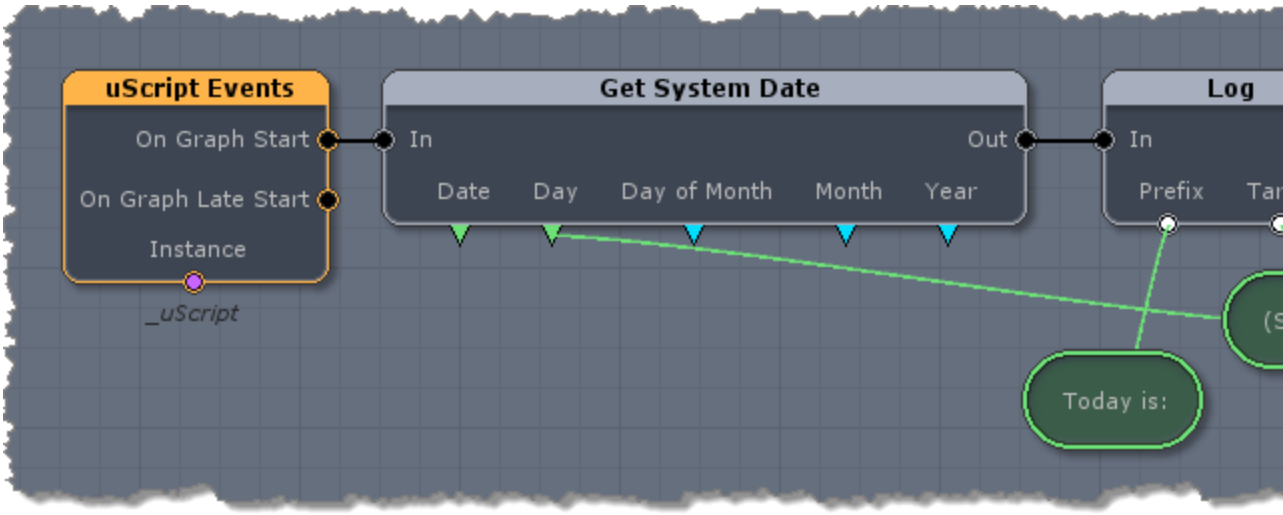
Example Action Nodes:



How to use them:

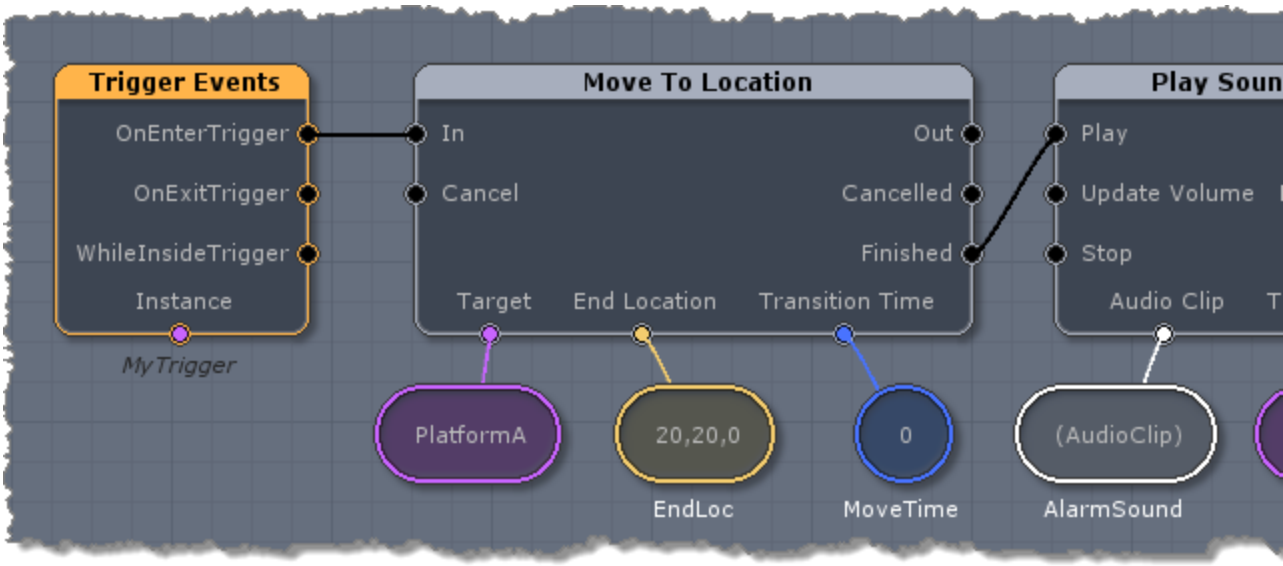
Action Nodes are used together to form complex actions from their simple behaviors. Here are a couple examples of how to use some Action nodes to create specific game logic you might find in a game. Notice that there is an orange Event Node at the start of each of the examples.

Printing the current day to the Unity console when the game starts:



(*the resulting text on a Monday would be "Today is: Monday"*)

Moving a platform when the player hits a trigger and play a sound when it is done moving:



Variable Nodes

Variable nodes are considers support nodes (usually color coded) that are used by both Event and Action nodes to store and retrieve data. Variable Nodes come in a few different types and shapes, but for the most part perform are used in a similar fashion in a uScript graph-- to get or set information used by Action and Event nodes. Please see the "Variables" section for more information on these types of nodes.

Node Documentation

uScript supports in-editor documentation for all nodes. To see the documentation for a specific node, you can either hover the mouse over the node in the Toolbox or select a node on the Canvas. The documentation for the node will show up in the References panel found below the Canvas area.

Please see the *Reference Panel* section of "Editor Interface" for more information.

Variables

Variable nodes, also just called variables most of the time, are a key part of uScript graphs. One way to think of variables are as containers that you can use to store a piece of information. You can either get a piece of information out from one of these containers, or put a piece of information into one of these containers. However, with the exception of List Variables (which we will discuss a bit later in this topic), you can only ever store a single piece of information at a time inside a variable.

All programming languages use variables in some way to store and retrieve data. Because uScript is a visual scripting system, it must also use variables to store the information you wish to use in your uScript graphs. uScript follows some key rules regarding variables, shown here:

Key Rules

- uScript has three types of variables:
  - ***Type Variables*** - a simple variable that can hold a single piece of information of a certain type (texture, sound, text, color, etc.)
  - ***List Variables*** - a list variable is a collection of type variables of the same type. Think of them as a group of containers-- each of which can hold a piece of data (but always of the same type).
  - ***Properties*** - properties are variables that are exposed to uScript through reflection. Though they can be used in the same ways other variables can, they require that you also specify the "owner" of the information by assigning an Instance to the Property's node properties.
- Variables are of a specific "type". The type determines what kind of information can be read or written to it.
- Variables can be assigned a Name, making them a clone of any other variables in a specific uScript graph that share it's name.
  - Named variables can be exposed to Unity's Inspector panel for external access to the variable's information.
- Most uScript nodes have variable sockets along the bottom of them, allowing you to hook up variables as required by the node.
  - You can also directly assign data to the node's variable sockets through the node's properties panel.

Color Codes

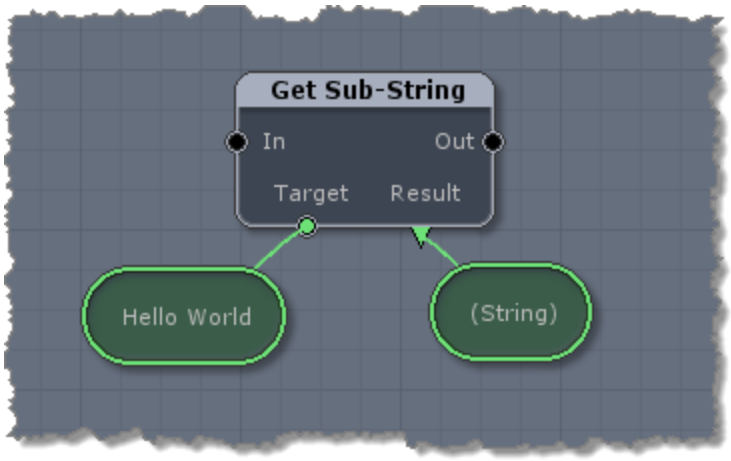
uScript uses color coding on the border of it's variable nodes for the most commonly used variable types in uScript. Any variable type not currently assigned a specific color

will have a white border.

Standard variable types that currently support color coding are:

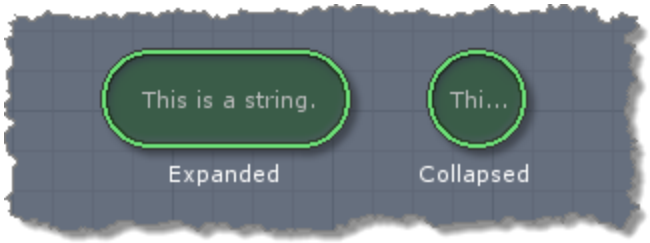
- **Float** - Blue
- **Int** - Cyan (light blue/green)
- **String** - Green
- **Boolean** - Red
- **Vector3** - Gold
- **GameObject** - Magenta (purple)

uScript's Event and Action nodes also use this same color coding for it's bottom variable sockets. Here is an example of a couple of nodes using a String variable. You can see how the green color for the String variable is also matched on the a node's socket that require a String variable for both input and output. Even the connection lines between the variable and the node's socket is the same color (green in this case):

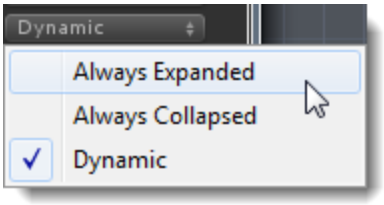


Variable Node Display Options

Variables in uScript are by default displayed as circles that on selection may be expanded to fit the full value of the variable if it does not fit in it's default, collapsed state.



In the uScript Editor's Preferences window you can choose how you wish to have uScript display variable nodes (see "Editor Preferences" for details).



Variable Node Types

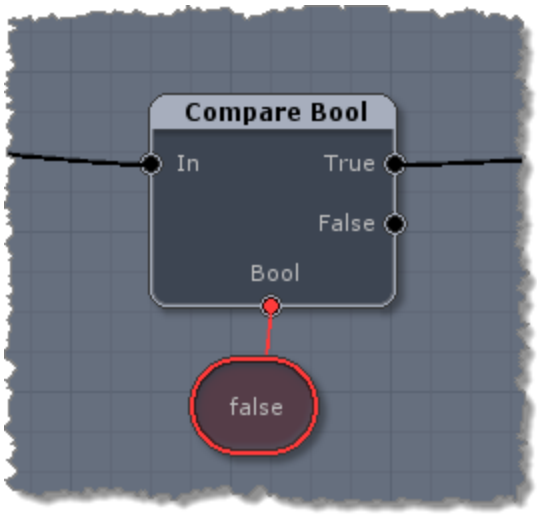
Common Variables

While uScript can use any type of variable that Unity uses through reflection, there is a common set of variable types you will find yourself using repeatedly in uScript. Those variables are described here.

Bool

Bool stands for boolean and it can either store the value ***true*** of ***false***. They work great as a way to decide if you wish to do something or not and many other comparison type situations.

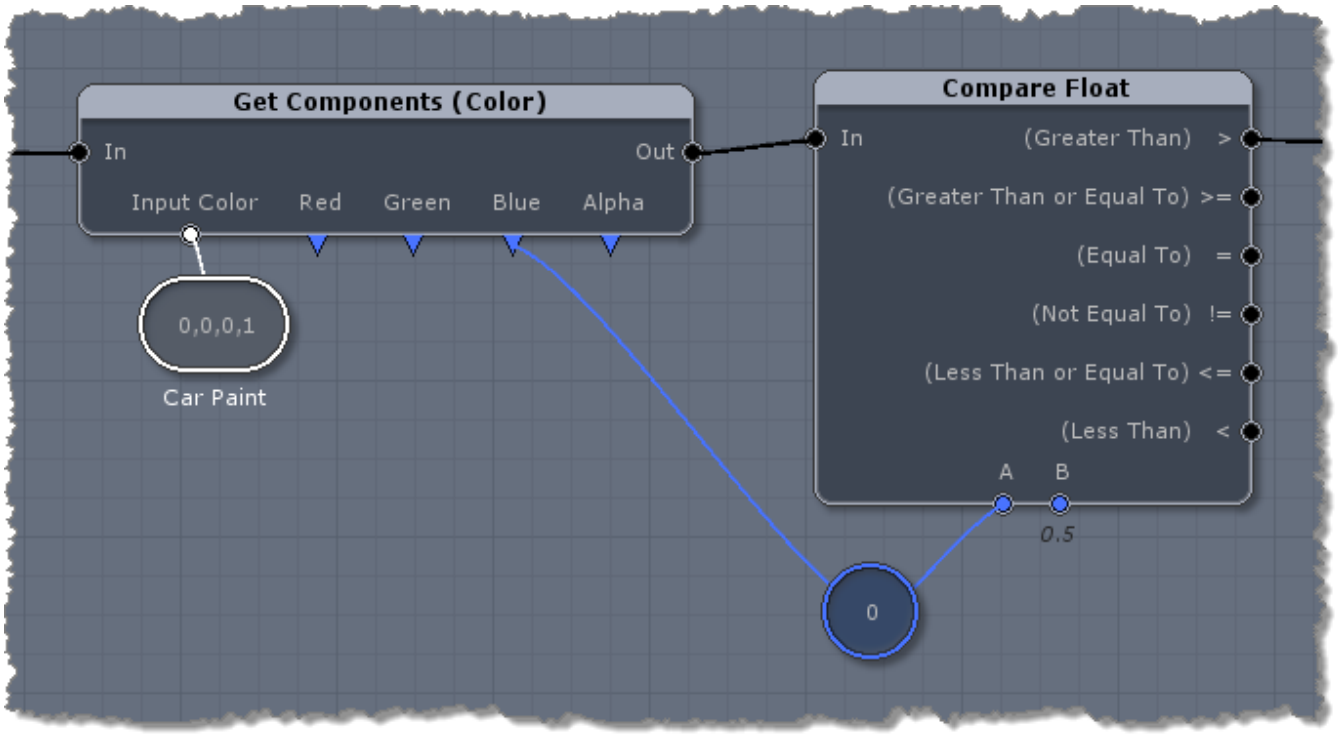
Example - here we check to see if a Bool variable is set to true or false. If it is true, the logic carries on to do something else:



Color

The Color variable stores a color as Red, Green, Blue, and Alpha float components. Legal ranges for the color components are between 0.0 and 1.0.

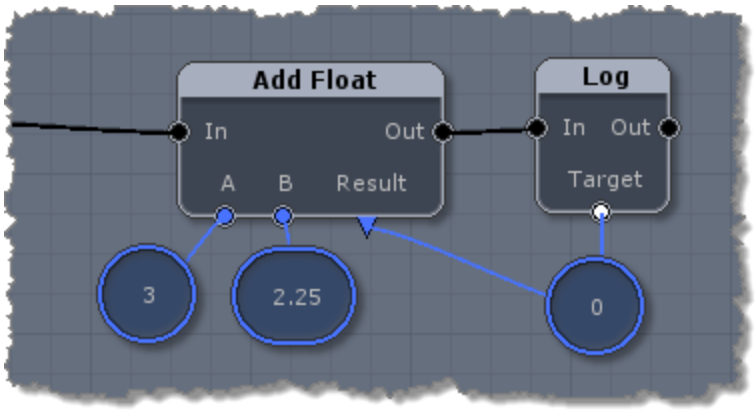
> *Example* - this logic get's the Blue float value from the named "Car Paint" Color variable and then checks to see if it is greater than "0.5":



Float

A float is a number that contains decimal places. Even a "whole" number such as 4 would be considered as 4.0 when it is a float. Most times Unity will want a number as a float as the need ofr decimal places is very important in math. Unity will also commonly use the range of 0.0 - 1.0 to represent percentages and other ranges internally. For example, if you wished to set the Intensity (brightness) on a light to 42%, you would represent that the float value 0.42.

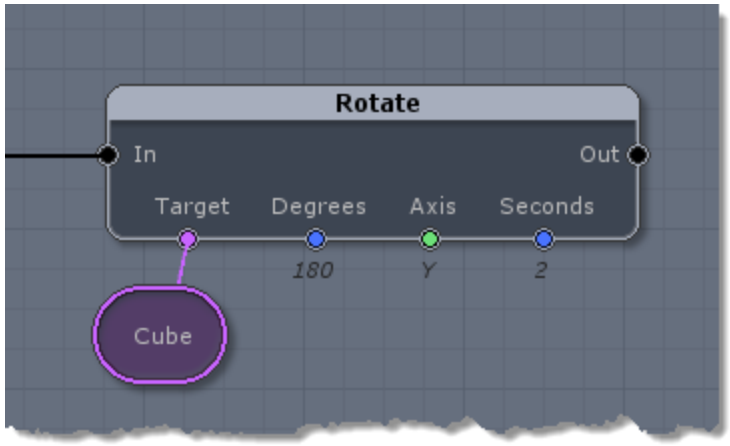
> *Example* - this logic will add two floats together and then print the result to Unity's console window. You might notice that the Float variable hooked up to the "result" socket shows "0". This is because 0 is the default value for a float variable. Once this logic is run in the game, the real value will be inserted into it (5.25 in this case) before the Log node prints out to the console window.



GameObject

Everything you place in your scene in Unity is a GameObject. This variable type allows you to store a specific object from your Unity scene so you can access it inside uScript in order to work with it.

> *Example* - in this example we are rotating a GameObject from the Unity scene called "Cube".

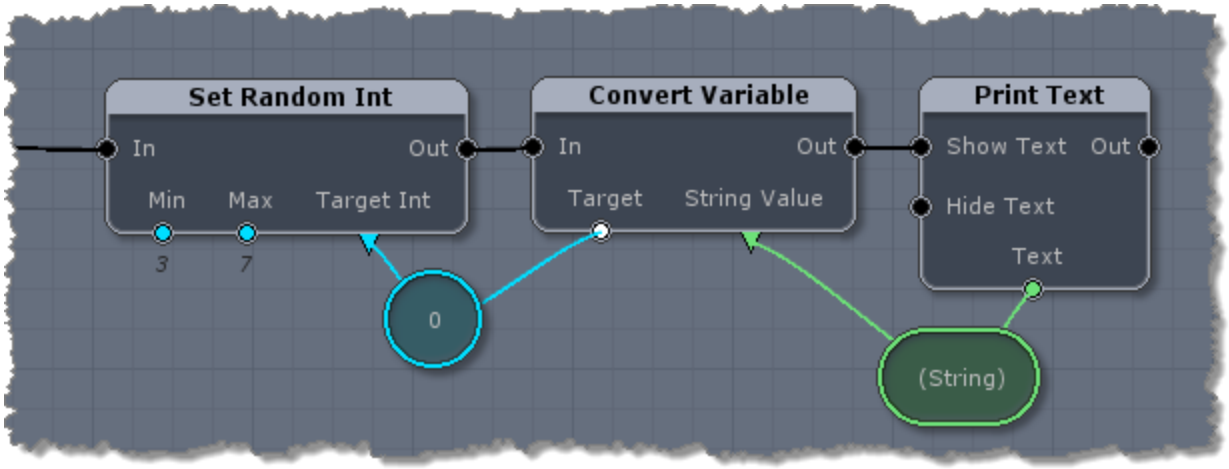


> **Note!** - *uScript relies on Unity to provide the correct GameObject by its name. It is important that you uniquely name any GameObjects in your Unity scene that you wish to directly use with uScript by name. If you do not wish to uniquely name your GameObject (or can't such as with prefabs spawning at runtime), please see the* ***Owner GameObject*** *variable type instead.*

Int

An Int is a whole number and can ***not*** contain any decimal places. Ints are not used as frequently in game logic as float variables, but are very handy for things such as counting totals or other times where you only need to use whole numbers.

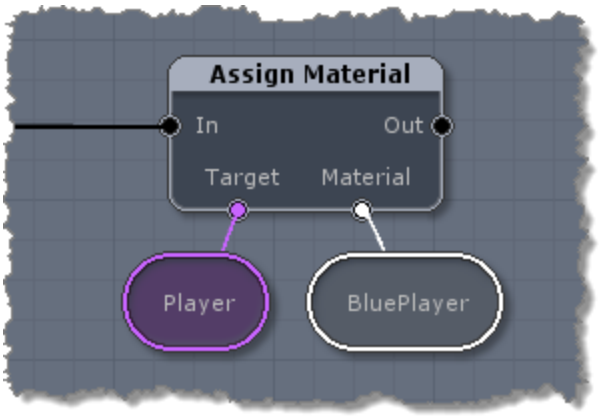
> *Example* - this example sets an Int variable to a random number between 3 and 7, then converts the Int variable value into a string variable (see String below) before finally printing the resulting number to the screen.



Material

The Material variable type holds a Unity material which is made up of texture, color and other information to describe how a GameObject should "look" in the game.

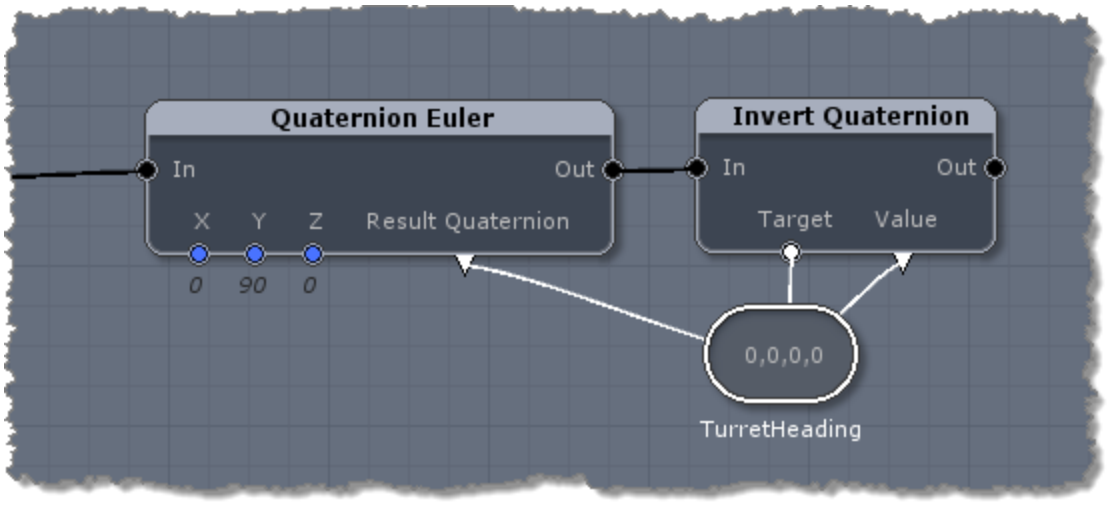
> *Example* - this example assigns a material called "BluePlayer" to a GameObject called "Player".



Quaternion

A Quaternion variable is made up of four float components and describes the rotation of a GameObject in mathematical terms. It is very rare that you would want to actual modify the components of this variable type directly and will instead mostly use by just passing on a Quaternion value you got from Unity into another uScript node or GameObject.

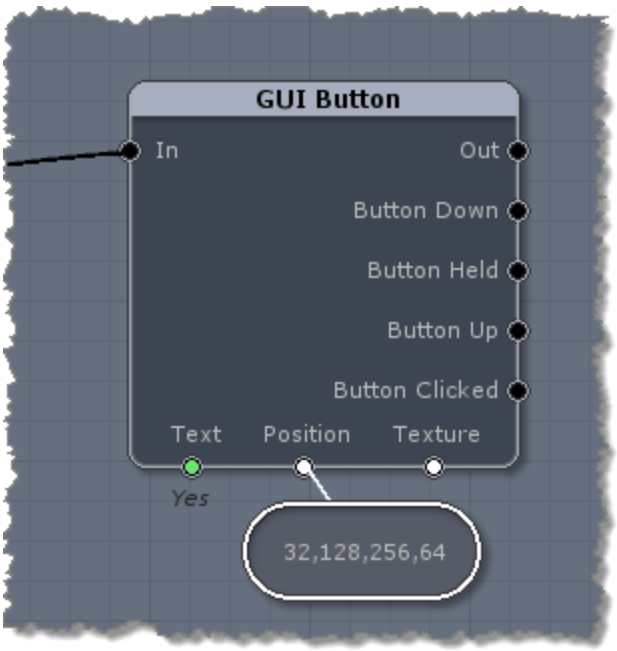
> *Example* - this logic creates a Quaternion value from Euler degree input (90 degrees on the Y axis in this case) and passes that value to a named Quaternion variable called "TurretHeading". It then Inverts the Quaternion variable's value by updating itself in the "Invert Quaternion" uScript node.



Rect

The Rect variable type is used by Unity for things such as the GUI system. It contains four float components (X Position, Y Position, Width, and Height) that define the size and position of a rectangle.

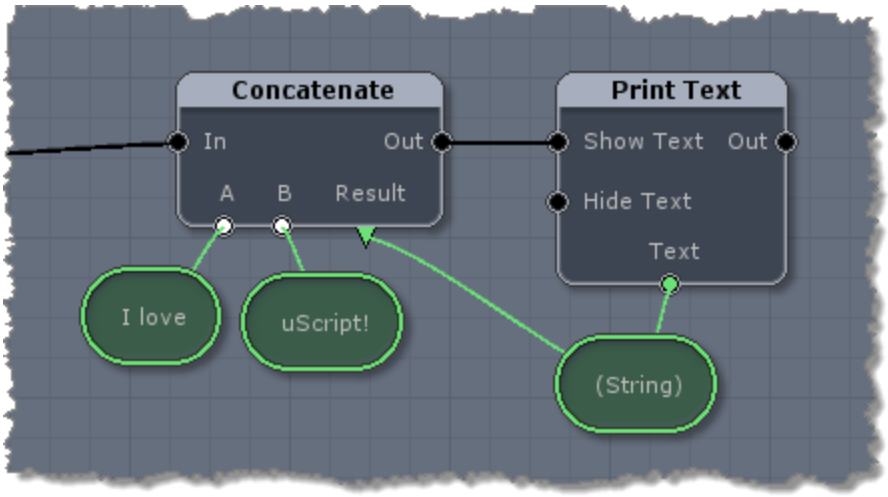
> *Example* - this logic uses a Rect variable to define the position and height of a Unity GUI Button in the scene (with a label of "Yes"). Note that it is telling Unity to draw the top left of the button at 32 on X and 128 on Y for the screen postion, and of a size of 256 wide by 64 high.



String

The String variable type is used to hold text. Even if you assign it a number, Unity will just treat it as text and not a number.

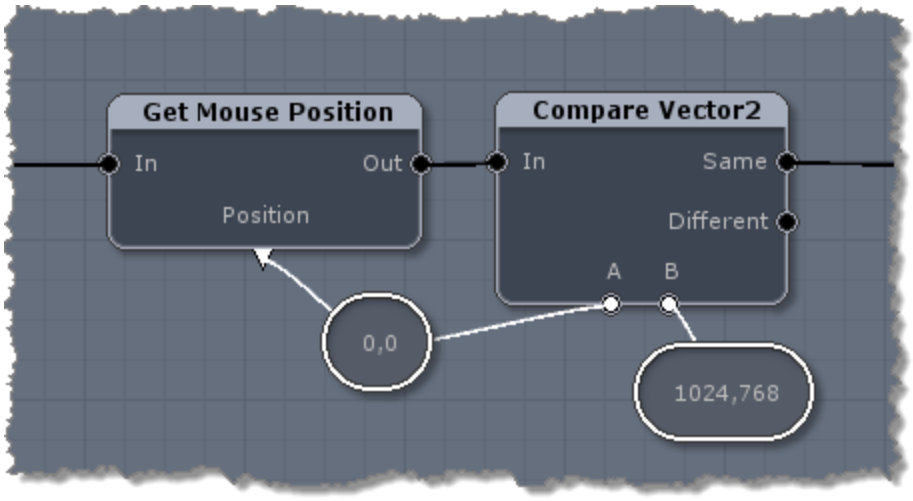
> *Example* - in this logic, we combine (concatenate) the text in two different String variables into one new String variable. The results are printed on the screen ("I love uScript!").



Vector2

The Vector2 variable type holds two float components (X and Y). It is a useful variable type to store a pair of floats that you would normal want to have together, such as a screen resolution or other X and Y location.

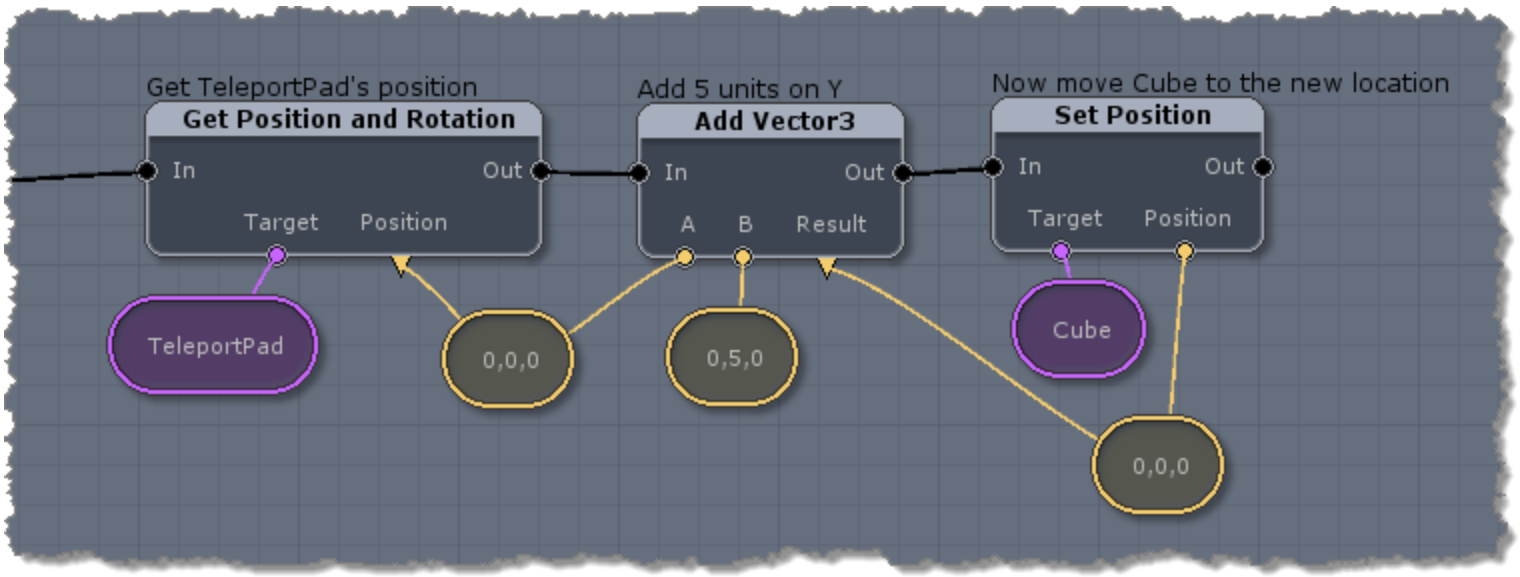
> *Example* - this logic gets the current X and Y position of the mouse cursor on the screen as a Vector2 and then compares it to see if it is at 1024 on X and 768 on Y as specified from the second Vector2 variable. If so, the logic will continue to other nodes.



Vector3

The Vector3 variable type is a very common type to use in Unity. It contains three float components (X, Y, and Z) and is usually used to store a position in 3D space. They are also used to store the force/magnitude of motion in a 3D direction for physics.

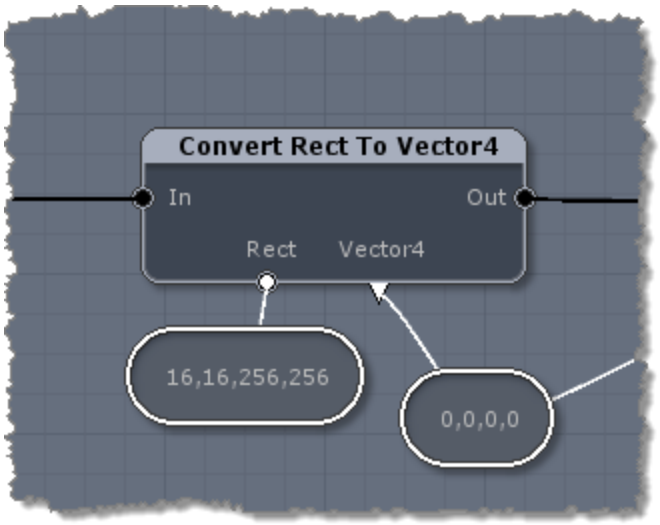
> *Example* - this logic tells the GameObject called "Cube" to teleport 5 units above the position of another GameObject called "TeleportPad" by first getting TeleportPad's 3D position in the game as a Vector3 variable.



Vector4

The Vector4 variable type is basically the same as the Rect variable type mentioned above as it contains four float variable components (X, Y, Z, W). However, this variable type is usually used in physics calculations and other general cases where four float values are useful in defining something.

> *Example* - this logic simply allows you to convert a Rect value into a Vector4 value.



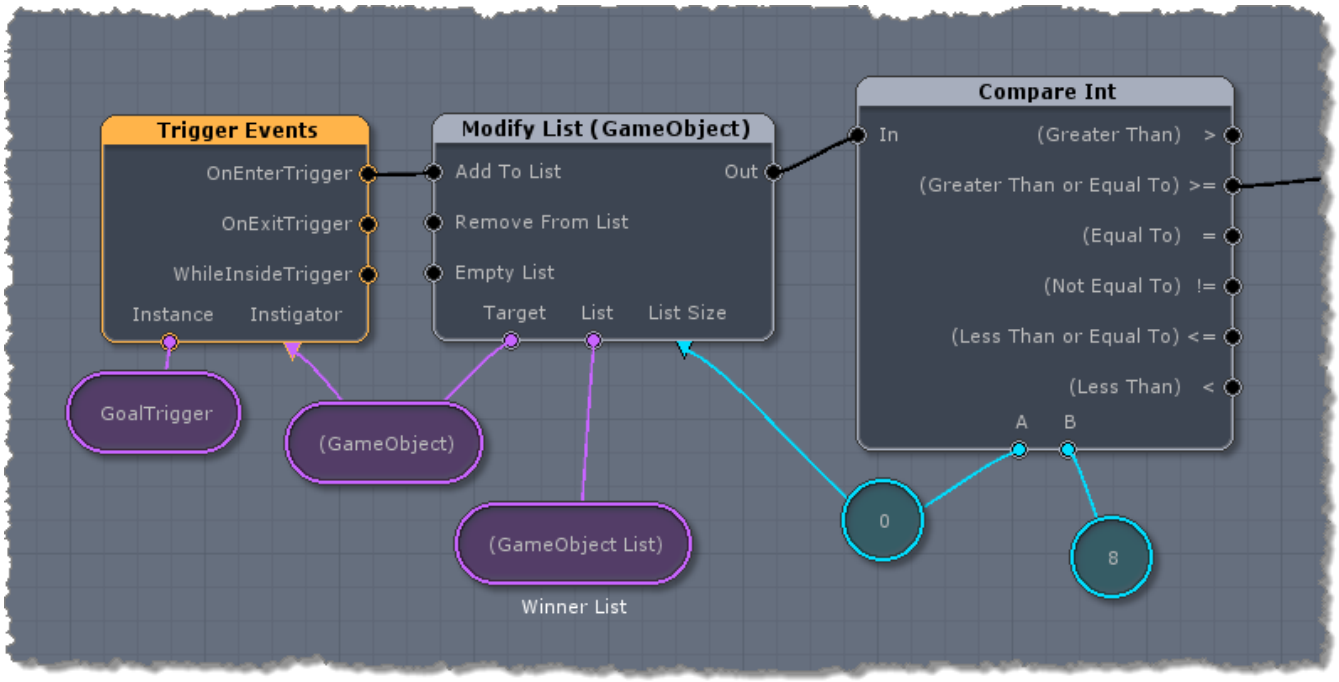
Other Variable Types

These and other more common variables are found in the Variables section of the Toolbox. Also, if you are using the Professional or Personal Learning Edition (*PLE*) of uScript, you can also access every variable type available in your Unity scene through reflection under *Reflection()/Variables*.

List Variables

Sometimes you will want to store a bunch of the same variable type together. uScript does this through List Variables. List Variables are just that-- they store a bunch of variables of the same type together. This allows you to use special "List nodes" we have written to quickly read/write/iterate through the list for various purposes. For those with a programming or scripting background, you can think of a List as a variable array.

A simple example of using a List variable might be if you wanted to end your game once a specific number of GameObjects have touched a trigger (let's say 8 GameObjects and the trigger GameObject is called "GoalTrigger" for example). To do this, you could add each GameObject that touches the trigger to a GameObject List variable and then check the size of the list to see if you have reached 8 GameObjects:



Properties

uScript can reflect public variables being used by other things in Unity ("public", or exposed, means that the developer who created the variable set it to be accessible by other things outside of itself).

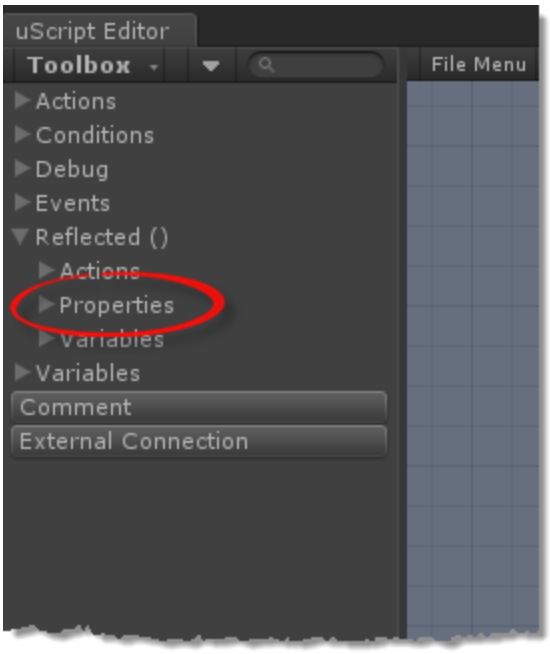
All reflected variables from other objects in Unity are called "Properties" and can be found under the Reflected()/Properties section of uScript. Reflected properties can be a very powerful thing and is also a great way to allow you to pass, or assign, your own uScript graph values outside of your graph to either other uScript graphs or even other Unity scripts or the Unity Inspector panel (see *Named Variables* below).

Please also see "Reflected Nodes" and "Advanced Reflection" for more information about uScript's reflection feature.

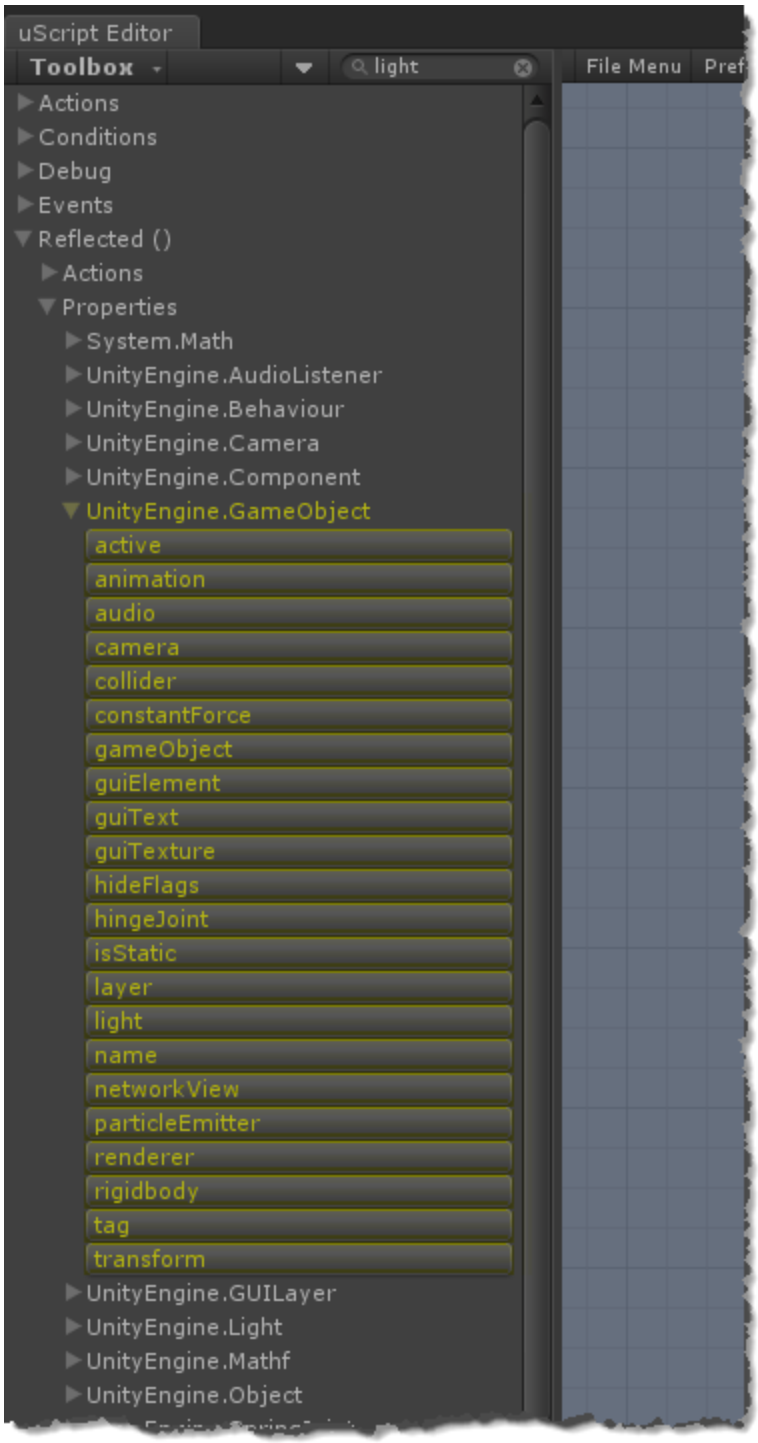
> ***Note!*** *- because properties are part of reflection, this feature is only available using the Professional or Personal Learning Edition (PLE) of uScript.*

Using Properties

To se what properties are available to use, go to the ***Reflected ( )*** section of the Toolbox and open the Properties section:



Once expanded, you can access any properties that are available in the currently loaded Unity scene. For example, here are the default properties available for all GameObjects:

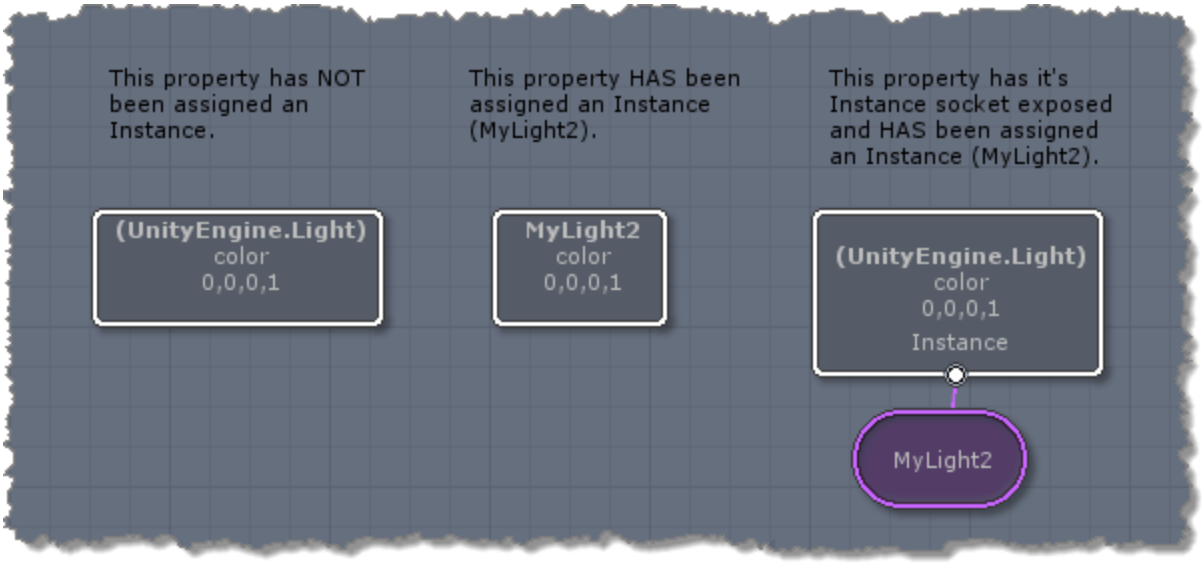


Setting the Property's Instance

Once a property has been placed on the uScript Canvas, it is important that you specify exactly what GameObject you wish to read or write the property to. This is important because you may have many GameObjects in your Unity scene that all have the same

stuff on them-- like a light component. uScript needs to be able to tell Unity which specific **instance** of a GameObject with the light component you wish to effect. To do this you just need to assign the property's Instance property with the proper GameObject.

Here are three Light component color properties. The first is how it would look when first placed on the graph before an Instance was assigned to it. The second shows the Instance property set (to a GameObject called MyLight2), and the third is also set, but through a GameObject variable hooked up to the Instance socket- which was set to visible from the property node's Properties Panel setting:



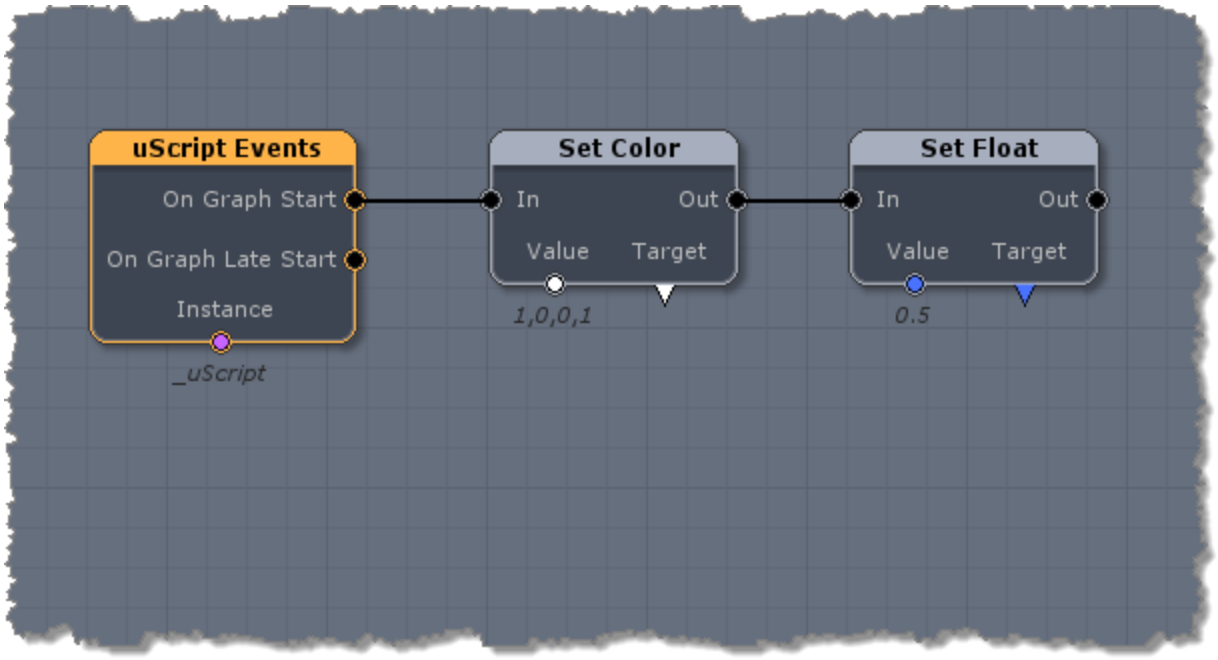
> ***Tip*** *- exposing the Instance socket can be very useful when you want to set the instance GameObject at runtime or use the Owner GameObject variable as the Instance.*

Example - Change Light Color and Intensity

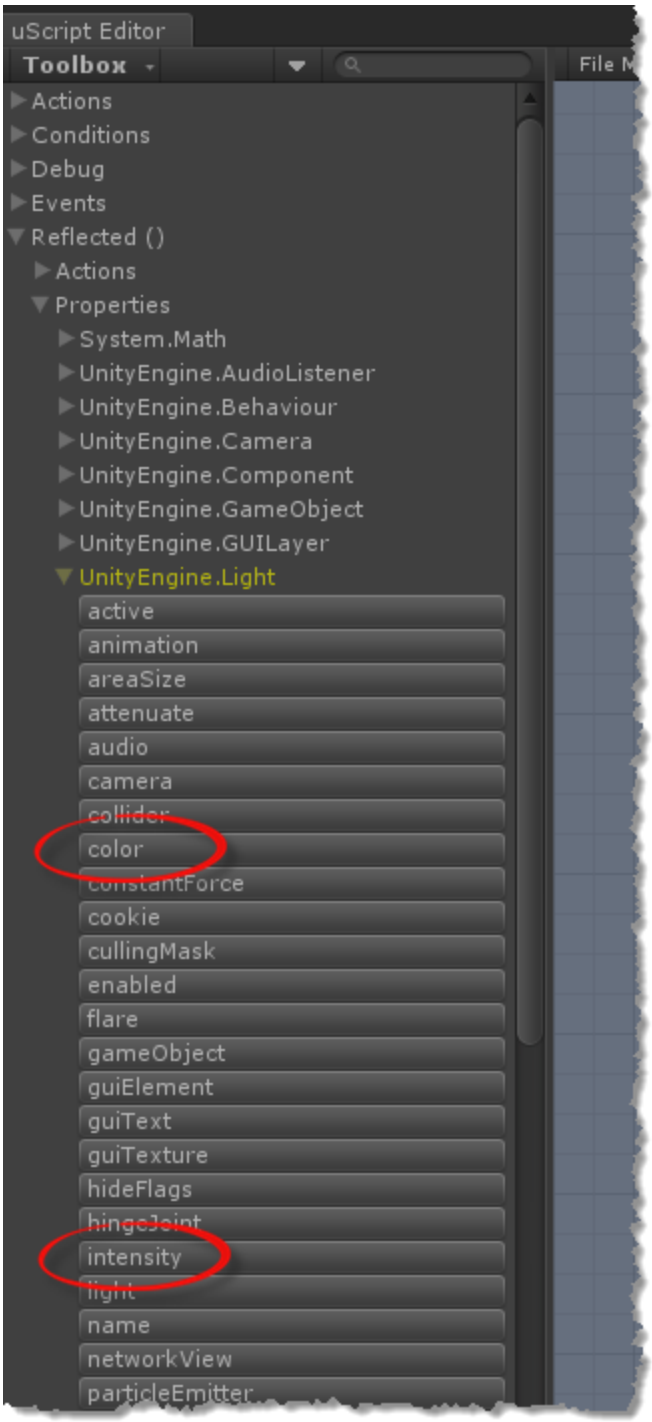
As mentioned above, properties are really just variables. In this example we will set the color and intensity of a specific light in a Unity scene by modifying two of its properties.

Let's say we had three point light GameObjects in the scene called MyLight1, MyLight2, MyLight3 and we wanted to modify the MyLight2 light properties to change the light's color to red and the intensity to 50% (0.5) when the game starts:

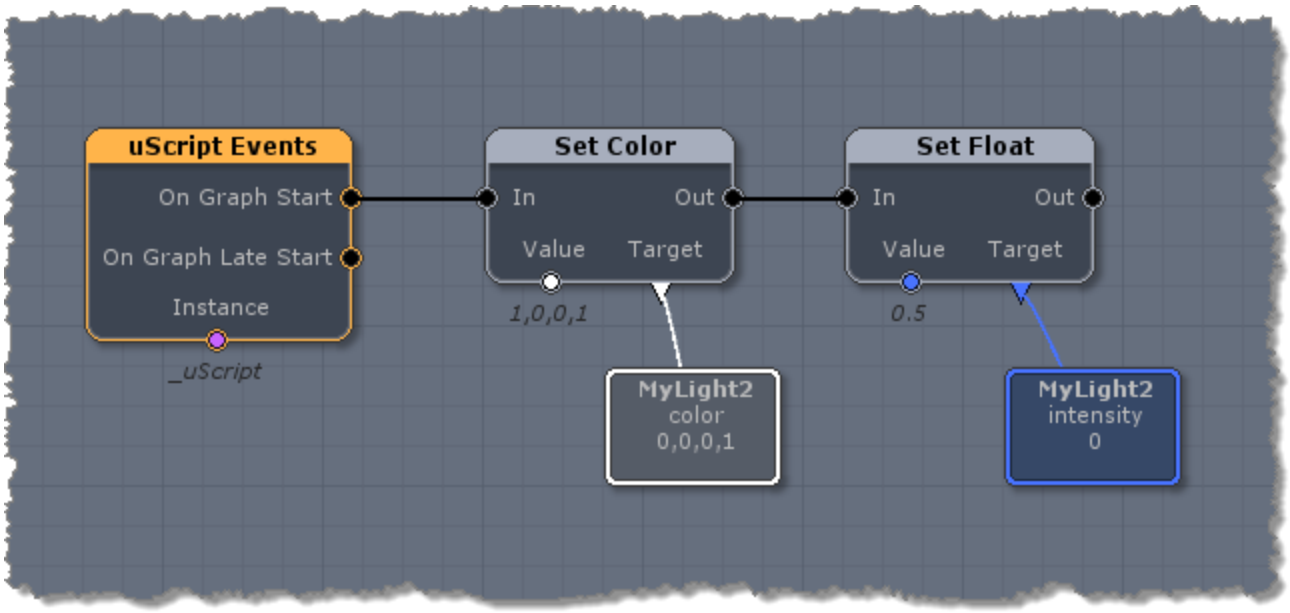
Step 1 - Set up the graph logic:



Step 2 - Find and place the needed property nodes for a light component:



Step 3 - Hook up the placed property nodes and make sure the set their *Instance* property to the specific light we wish to effect (*MyLight2*):



Advanced Variable Use

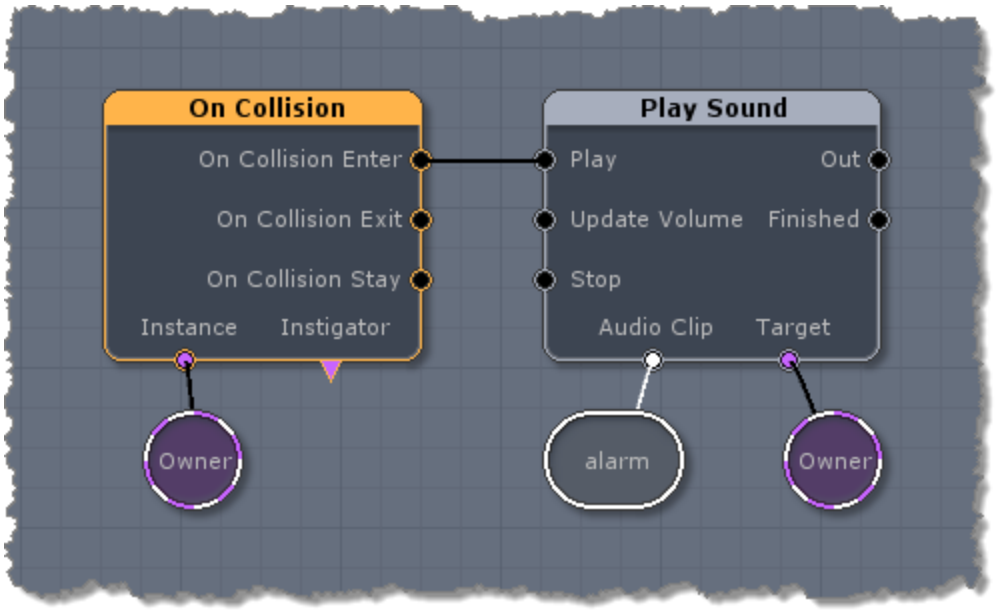
Special Variable Nodes

uScript has a few "Special" variables that are designed to perform specific tasks.

Owner GameObject

The Owner GameObject it a special variable that tells uScript to "use whatever GameObject this uScript graph is assigned to". This allows you to build reusable graph logic for Unity Prefabs or other such uses. For those familiar with programming, the Owner GameObject variable is the equivalent of "*this*" in code.

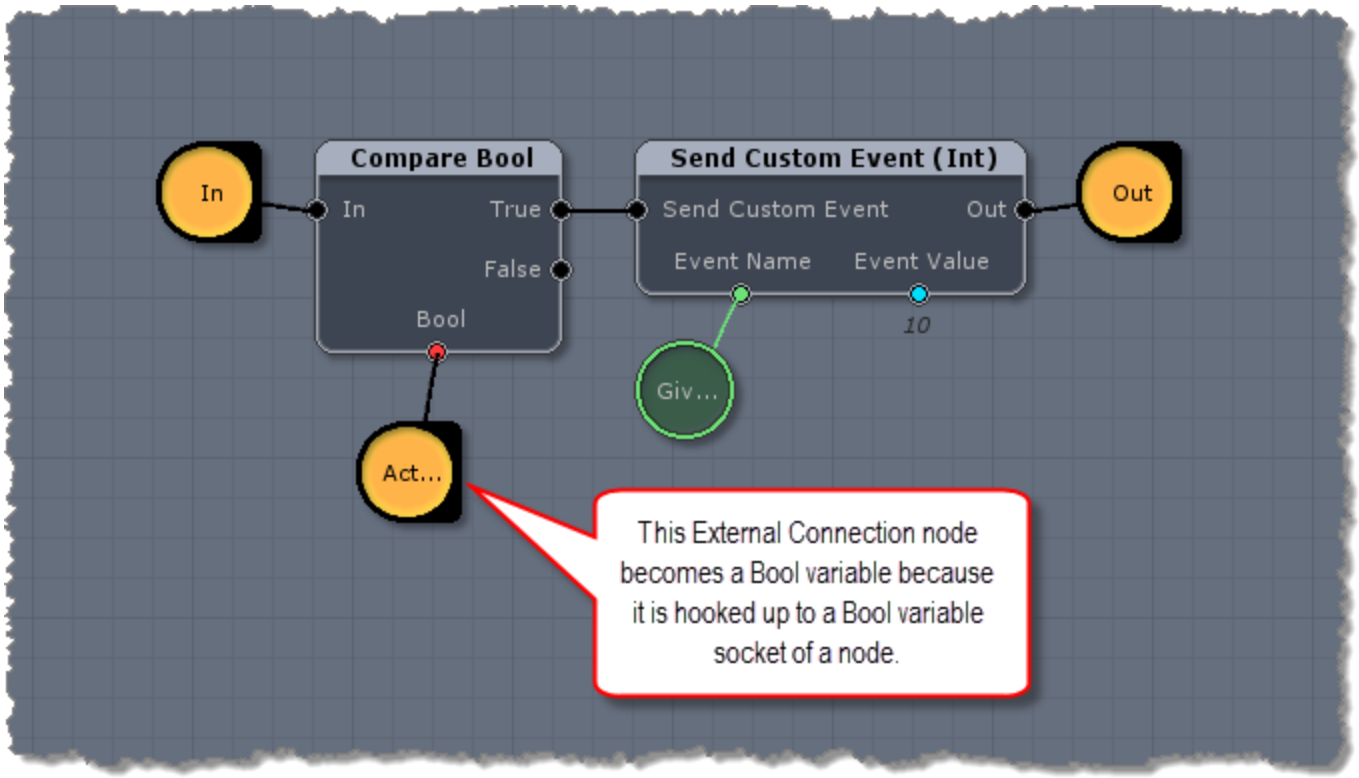
Here is example logic that will play a sound when something collides with a GameObject. Because it uses the Owner GameObject for the Collision Event's instance socket as well as the Play Sound node's target, this graph can be assigned to any GameObject and it will automatically play the sound for that GameObject collision only (and the sound will come from that same GameObject):



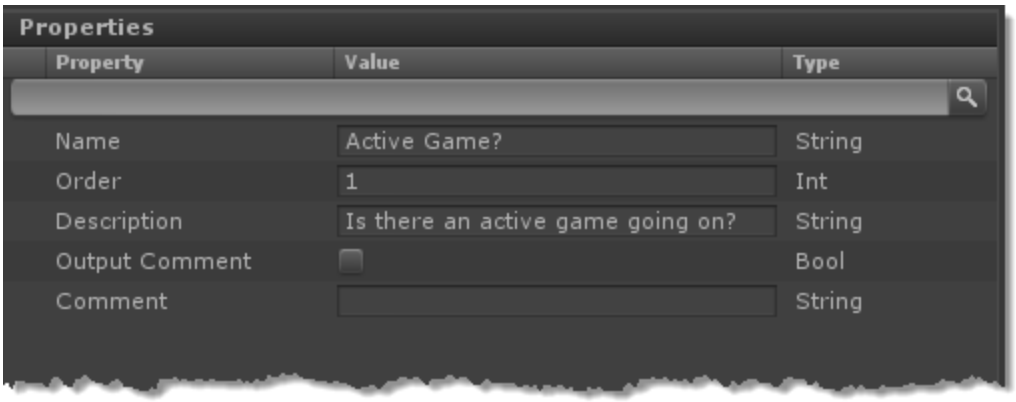
External Connection Node Variables

The External Connection is used when creating nested graph node's in uScript. They can be hooked up to any node socket-- including variable sockets. When an External Connection node is hooked up to a variable socket, it will take on the variable type of the socket is was hooked up to.

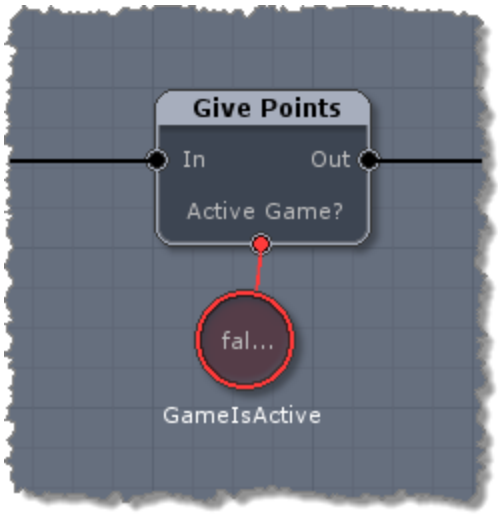
Here is an example of External Connection nodes hooked up to various sockets to create a Nested Graph called "Give Points". Notice that the there is one hooked up to a Bool socket of a node:



It is important to setup the External Connection's properties-- especially a friendly name:



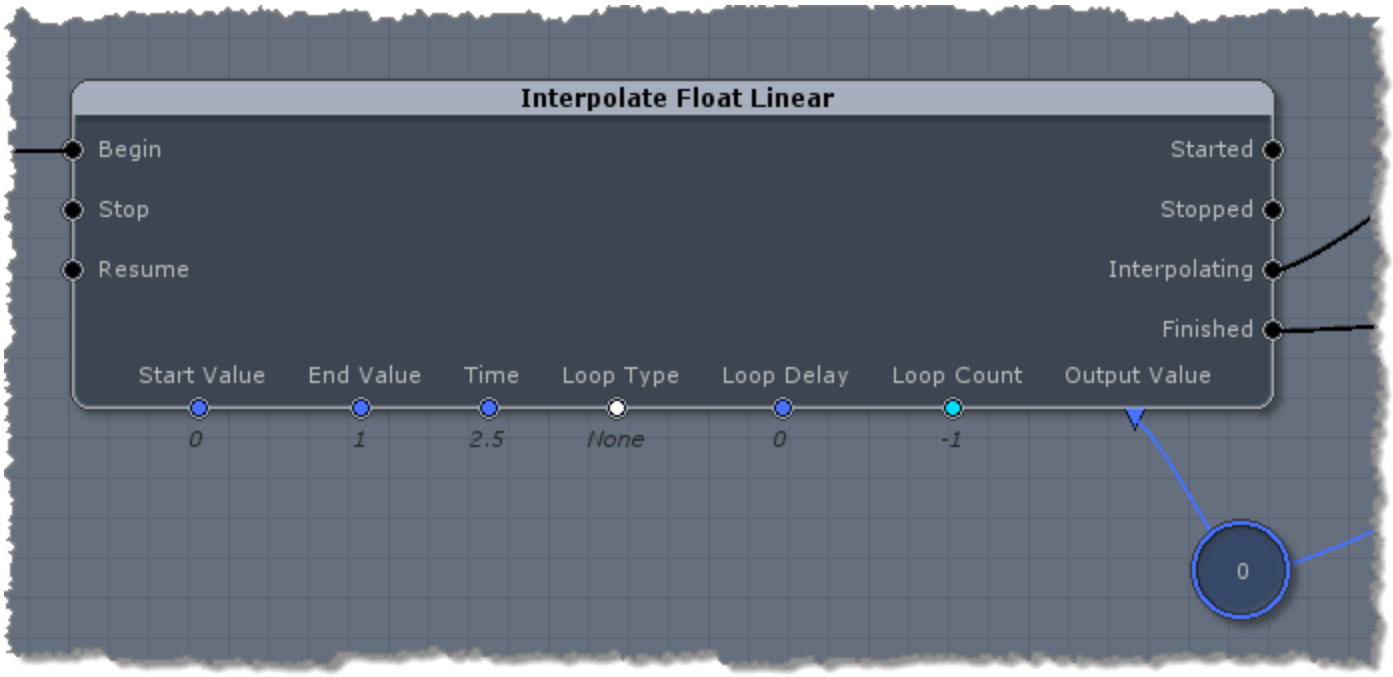
Here is what the nested node looks like (called "Give Points") when used in another uScript graph and hooked up to a Bool variable:



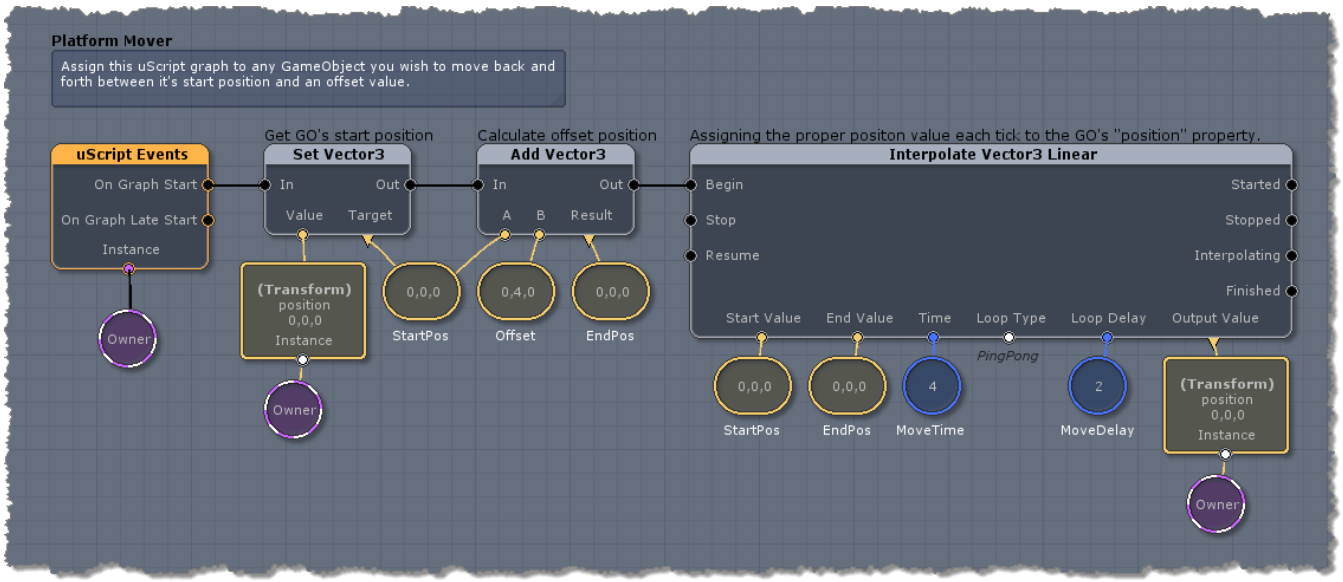
Note! - *please see the Nested Graph section of the topic for more information on creating Nested Graphs and using External Connection nodes.*

Changing Values Over Time

There are many times when you might wish to change the value of a Variable or Property Node over a set time. This can be accomplished using our Linear Interpolation Nodes. These nodes allow you to define a start and end value that will change between the two over a specified time.



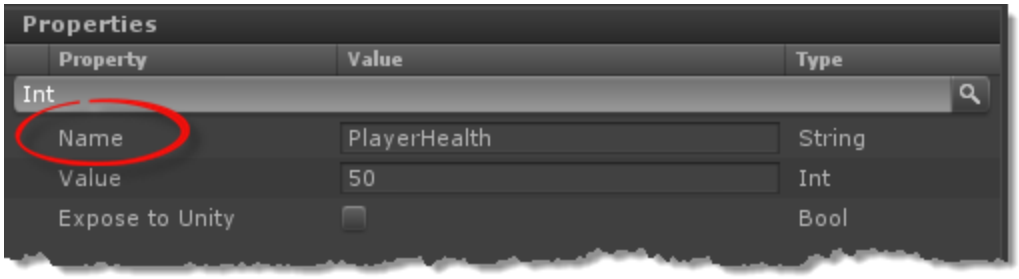
Here is an example where we use an Interpolate node (a Vector3 one in this case) to change the value of a Vector3 over time and feed that value directly into a GameObject's "position" Property Node each tick of the game to move it. Also note we are using the Owner GameObject variable so that this graph can be assigned to any GameObject to move it-- making it very reusable.



Named Variables

Named variables allow you to "share" a variable throughout a uScript graph without needing to draw many lines to the same variable node to access it. uScript will treat any variable of the same type with the same name as the same variable-- no matter how many instances you might have of that named variable on your graph. If you change the value of one of the named variables, all other instances of that named variable will have it's value automatically updated throughout the entire graph.

Naming a variable is as simple as setting it's Name property in the Properties Panel:



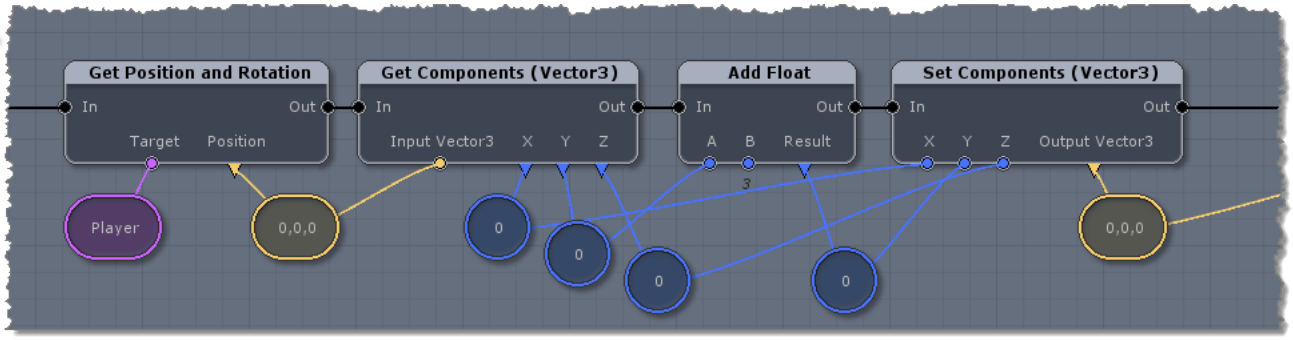
Once you have named a variable, it's name will show up below the variable node on the graph:

Named Variable Rules:

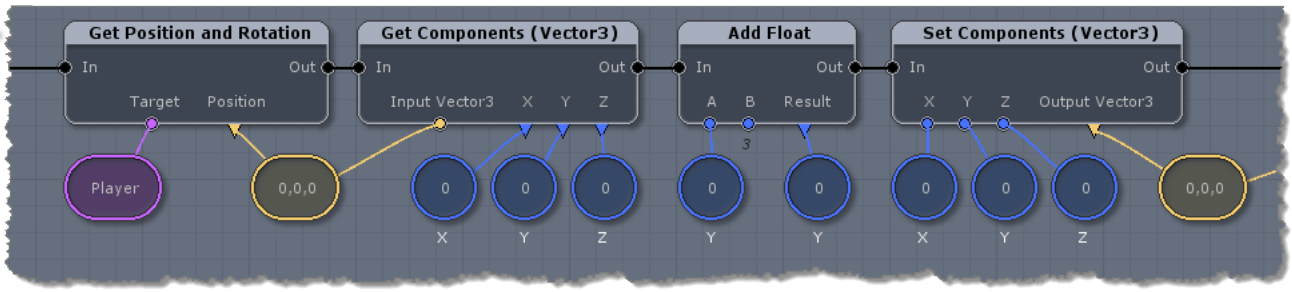
- Named Variables are just regular variables that have been assigned a "friendly name" through the variable node's Properties Panel..
- All instances of a Named Variable must be of the same variable type (string, int, float, vector3, etc.)
- All instances of a Named Variable will always contain the same value, so if you change the value in one instance, all the others will automatically be updated as well.
- Named Variables are unique to each uScript graphs-- they are not shared between graphs in any way.To share values between graphs, please see the Exposed Variables section.
- You can have spaces in your names.

Here is an example of the same logic shown twice. In the first example (1), we are linking the variables directly to all the sockets that need those values. In the second setup (2), we are using named variables to pass along the values to the sockets that need them. These two graphs are functionally the same:

1.



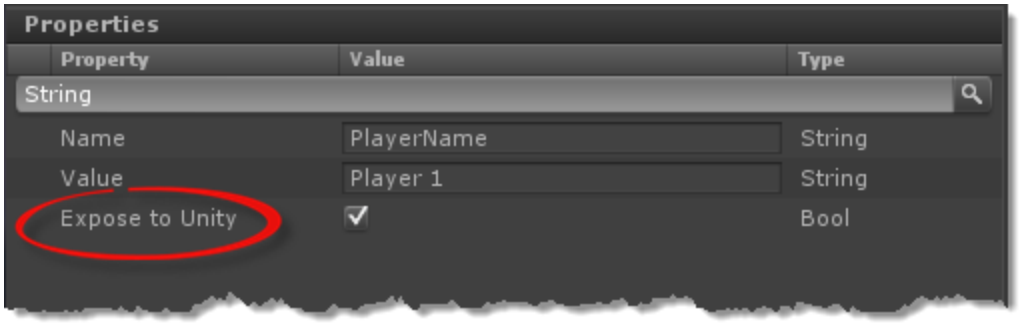
2.



Tip - *It can be very useful to create a section of a large graphs that contain an instance of all Named Variable in the graph. This can make editing and iterating on the graph faster. Here is an example:*

Exposed Variables

Exposed Variables are simply standard variables that have been named (see *Named Variables*) that have also been set to be exposed to Unity's inspector panel. This is done by checking the "Expose to Unity" option in the Variable Node's Properties Panel. Please note however that this option will only be available for Named Variables as Unity needs a name for the variable in order to expose it.



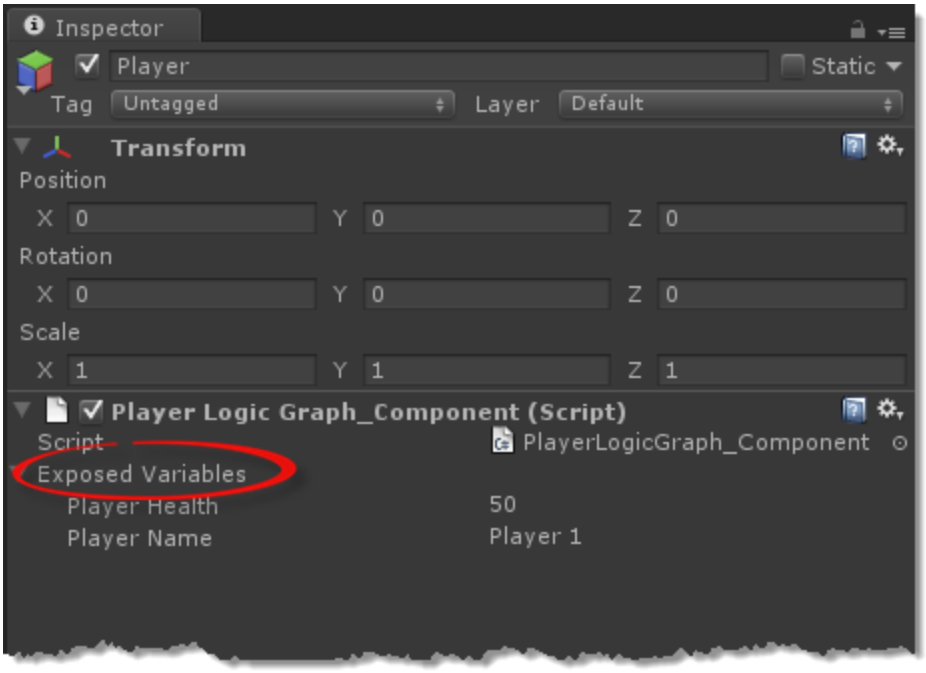
Note! *- for you programmers/scripters out there, exposing a variable is equivalent to making a variable public in a Unity script.*

Exposing variables to Unity has two major benefits:

Inspector Editing

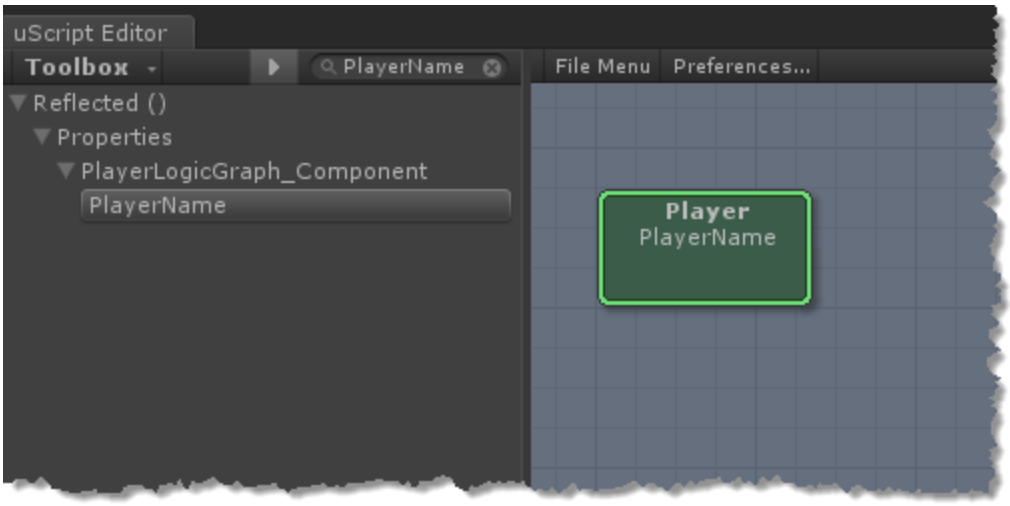
Once you have exposed a variable to the Unity Inspector Panel, you will be able to edit that value from Unity directly without needing to open the uScript editor to view or change the value. This also means you can take advantage of Unity's ability to view and edit the values of exposed variables while the game is running for quick iteration and debugging.

To access exposed variables, open up the uScript graph component on the object that contains the variables you are looking for and expand the "Exposed Variables" section:



Sharing Values Across Graphs as a Property

You can access any exposed variable form another uScript graph as a reflected property (see *Properties* in this section for details). This is a great way to pass a value from one uScript graph to another.



Note! *- Make sure you set the Property Node's Instance to the GameObject that has the graph with the exposed variables assigned to it!*

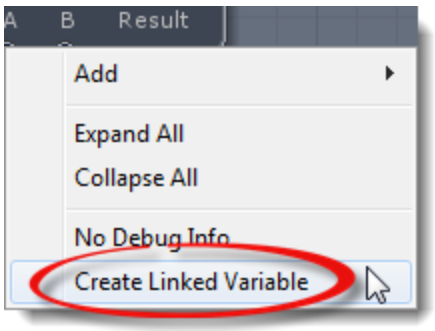
Variable Tips & Tricks

Hot-Keys

uScript has several hot-keys that can be used to quickly place nodes on the graph. For example, holding the "S" key while left-clicking on the canvas will place a String Variable, "B" will place a Bool Variable, etc. For a full list of hot-keys, see the Canvas section of the "Hot-Keys" topic in the References section.

Context Menu

Another way to quickly place a Variable Node and have it automatically hooked up to a socket, is to use the right-click context menu. Simply right-click on a node's variable socket and choose the "Create Linked Variable" option of the menu.



Reflected Nodes

> **IMPORTANT!** - The reflection feature is only available in the *Professional* and *Personal Learning Edition (PLE)* versions of uScript. If you wish to upgrade your license of uScript to the Professional version, please contact Detox Studios.

One of uScript's most powerful features, node reflection, allows uScript to visually represent many parts of your game visually without the need to write custom nodes to access them. More advanced uScript graphs will almost always use at least some reflected nodes.

By default uScript will reflect, as visual nodes, any components or scripts that have been assigned to GameObjects in your Unity scene that contain public methods, properties, or variables.

> **Also see** "Advanced Reflection" in the *Advanced uScript Topics* section for more information about reflection.

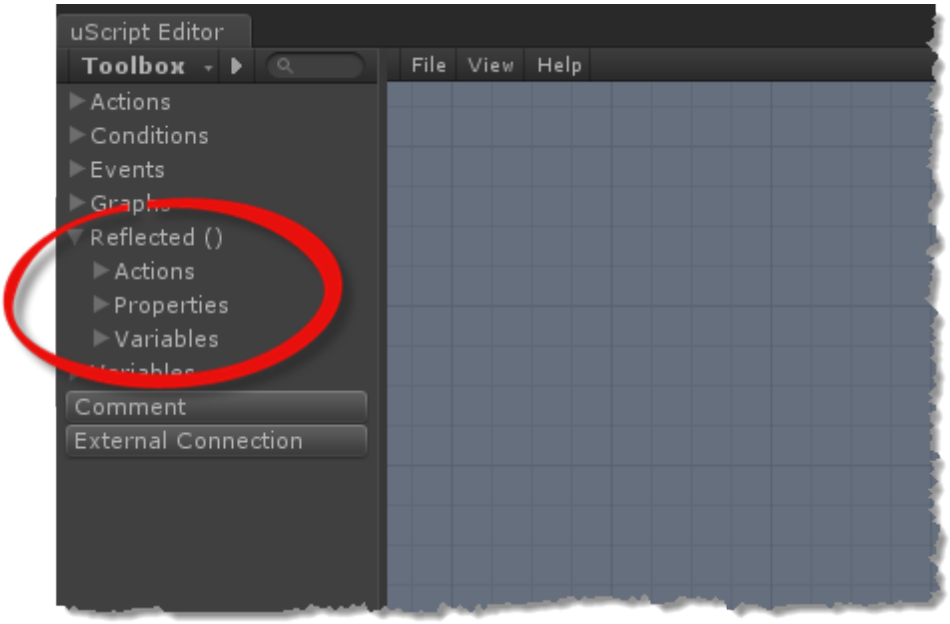
Rules of Reflection

It is important to remember the rules uScript uses to determine what it reflects as visual nodes inside the editor:

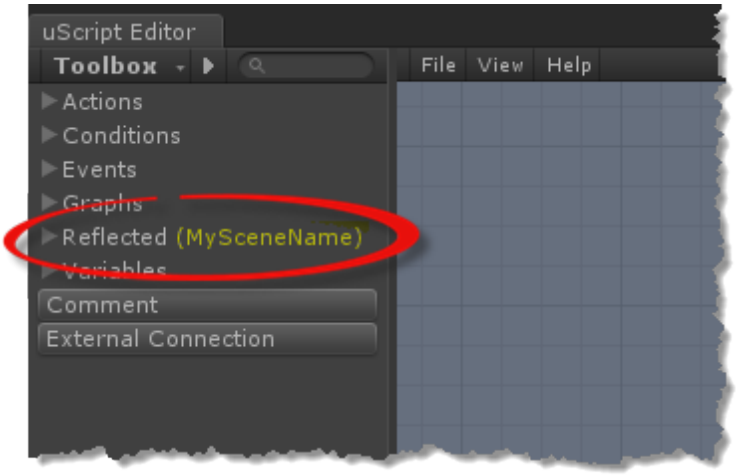
- uScript will reflect Unity runtime components or scripts (referred to collectively as "items" below).uScript can only reflect scripts that have been compiled before the uScript editor. See Unity's documentation regarding compile order if you have problems accessing Java scripts (you should not see this issue when using C# scripts).
- uScript will only reflect items if they meet one of the following criteria:
  - The item exists on a GameObject in the currently loaded Unity scene.
  - Any items specified by reflection in the uScriptUserTypes.cs file.
- All reflected nodes must have their Instance property specified with a specific GameObject that contains the reflected item (see the *Instance Property* section below for more information).
- Reflected Properties are used like any other variable node in uScript-- even though their node shape is different.

Locating Reflected Nodes

All nodes that are being reflected by uScript can be found in the *Reflected ( )* section of the uScript Editor's Toolbox panel.



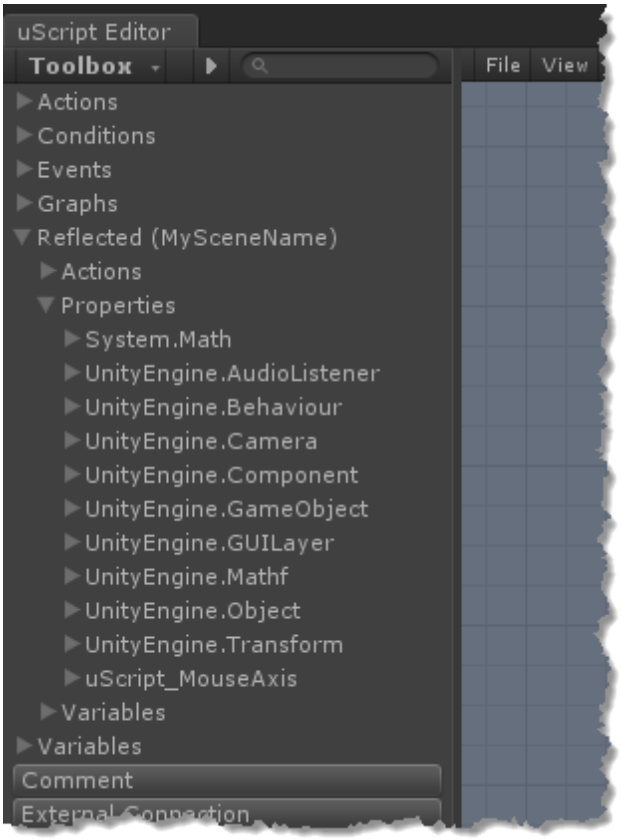
The empty braces will contain the name of the currently open Unity scene if you have one open. In this example, we have a Unity scene open in the editor called "*MySceneName*".



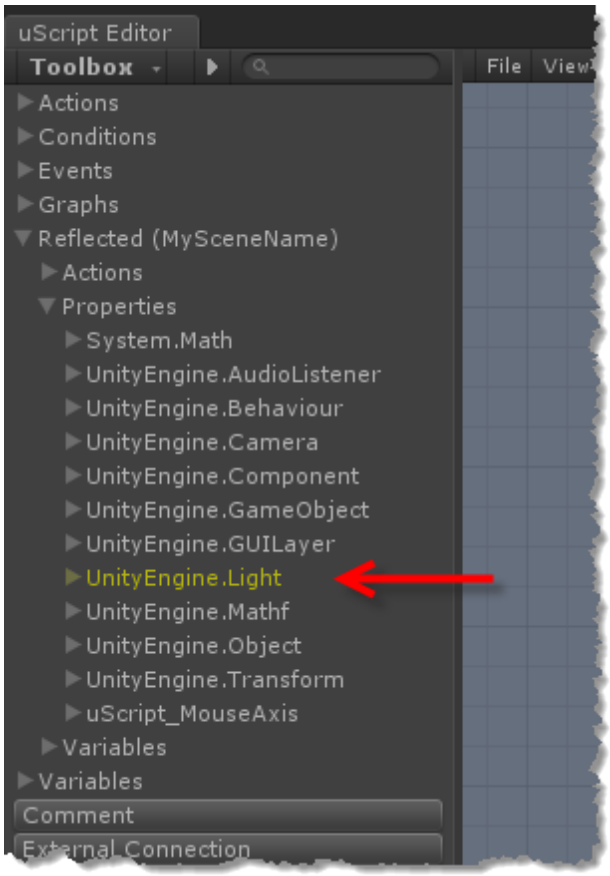
The scene is important because as noted above, uScript can only reflect what is available in a specific Unity scene. As an example, if you have a Unity scene open that has a light GameObject in your scene, you will have access to reflected light properties--

but if you do not have any light GameObjects in your open scene, you won't see any light properties being reflected.

A Unity scene with no lights in it:



The same Unity Scene once a Point Light GameObject has been added:



> ***Note!*** *- You will need to close and re-open the uScript editor for it to detect and refresh what is being reflected by the Unity scene.*

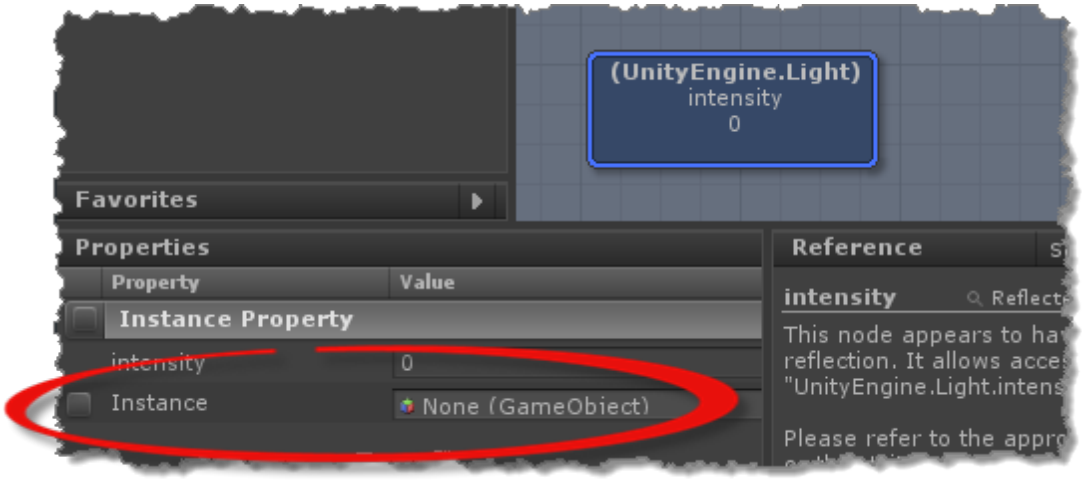
You can force uScript to reflect Unity functionality that is not present in the open Unity scene. This can be handy to access GameObject component properties for things that are only spawned at runtime. To learn more about this, please see the *Reflecting Things That Aren't There* section of "Advanced Reflection".

Instance Property

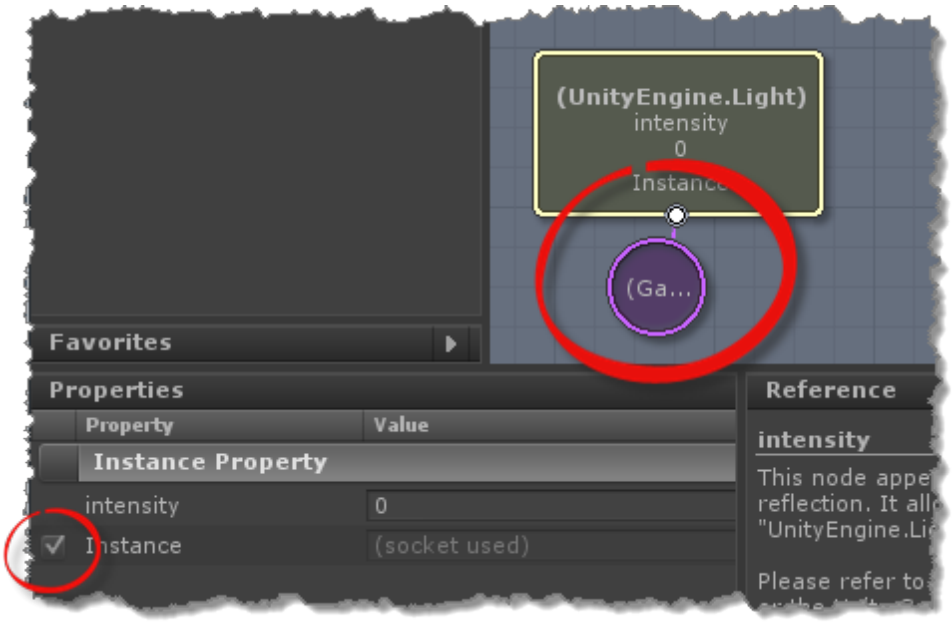
Almost all reflected nodes will have an Instance property that **must be assigned to a GameObject** in order for the reflected node to work. This is because you can easily have more than one instance of something in a Unity scene and uScript needs to know which specific instance you wish to target.

For example, you may have hundreds of Point Lights in your scene, so if you want to tell Unity to turn off a specific light by setting its Intensity property to zero, you will need to let

uScript know exactly which one. This is done by assigning the Instance property of the reflected node:



Note! *- If you wish to assign an instance at runtime in your graph, you can expose the Instance socket. Here is an example of doing this on a reflected Property node:*



Note! *- When you assign an Instance to a reflected Property node with a GameObject from your Unity scene, the generic name within the braces will*

change to the name of that GameObject (this will not happen when exposing the socket and assigning the GameObject instance externally as show above):

Pink Nodes

If you have out of date nodes in your graphs that need updating, uScript will turn them a bright pink color.

Why Nodes Turn Pink

Before getting into how to fix out of date nodes, it is first important to understand **why** nodes might be out of date.

uScript has many nodes and each of these nodes is actually just a C-Sharp script (.cs) that performs a specific function. Other nodes are just part of uScript's reflection system. In both cases uScript just visualizes all this programming script code as visual nodes in the uScript Editor. Because uScript is not in full control of all these visualized scripts, it is important that it has a system in place to handle the case where code it is visualizing as nodes can detect changes that might have been made by someone working on your project.

The primary goal to detect these potential modifications to is to try and handle the changes that were made and preserve as much data in your graphs as possible. This is really important when,for example, you might have a variable in uScript hooked up to a socket on a node and; that node no longer has that socket, the socket has been named differently in the programming code, the socket now takes a different type of variable than before, etc. By not just auto-updating existing nodes on your graph (which could wipe out data you had set on the in the node's Properties Panel) we are able to preserve important node data and settings you might want to have access to before performing the node upgrade.

However, before uScript can rebuild your graphs generated source code and to take advantage of the changes made to either a node or some other reflected code, you must perform the upgrade on the node(s) and make sure everything is set up properly to handle the new code and properties that might have been introduced.

Note! *- Some simple changes to a node's source code will not cause the node to turn pink because uScript will be smart enough to know how to handle the updated source code and automatically fix everything for you.*

Nodes may need to be upgraded for several reasons:

Unity Compile Errors

Because of how Unity's editor tools architecture works, if Unity can not fully compile all the source code of the project due to any compile error, it will not be able to fully compile the uScript Editor itself. When this happens, there is a chance that all nodes on your canvas will show up as pink because uScript is not able to fully function and validate all the projects code to ensure it was the same as before (becauseUnity couldn't compile it all).

When this happens it is important to discover what is causing the error and fix it before using the uScript Editor. If the cause is uScript generated code (which can easily happen if a node you are using in existing graphs has been modified in major ways), you can safely delete that generated code, then run uScript, upgrade the pink node(s) in the offending graph(s) and then regenerate the code.

Updated Node Source Code

As mentioned above, if a node has been modified in such a way as to change the sockets, or properties required by the node, uScript will need to mark it for updating in an effort to preserve your old data so you can access it before updating the node on existing graphs. This also goes for any node created by uScript of existing project code or other middleware through reflection.

Node Marked Deprecated

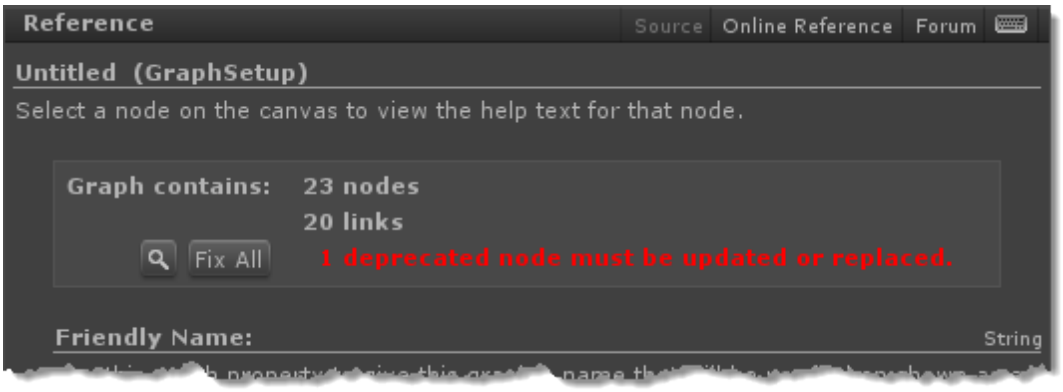
If a node is deprecated through the Deprecate attribute in a node's source code it will be marked pink and in need of replacement or updating (see the *uScript Development Guide's* section on Attributes for more information).

When a node is marked for deprecation without specifying a new node to replace it with, the nodes will need to be removed from existing graphs and

no new instances of that node may be placed. If a new node was specified in the Deprecate attribute, the node must be upgraded to the new node.

Identifying Nodes Needing Updating

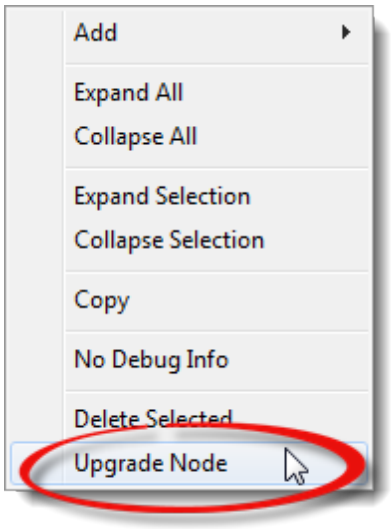
The easiest way to know if your graph has any nodes that need updating is to look at the References Panel (see the *Reference Panel* section of "Editor Interface" for more information) with your graph open in uScript with no nodes selected. This will show you information about the open graph. Here you can see this graph has 1 node in need of updating (note the red text in the image). You can use the buttons there to focus on the node in the Canvas as well as fix it.



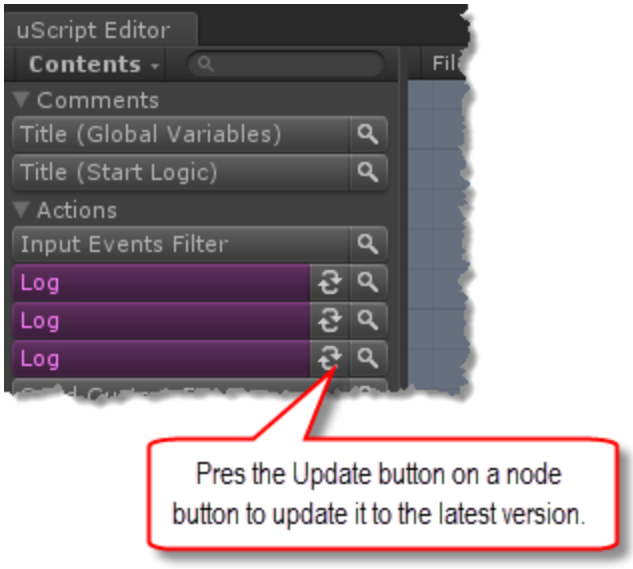
There are other ways to see and fix these nodes as well as shown in the next section.

Upgrading Nodes

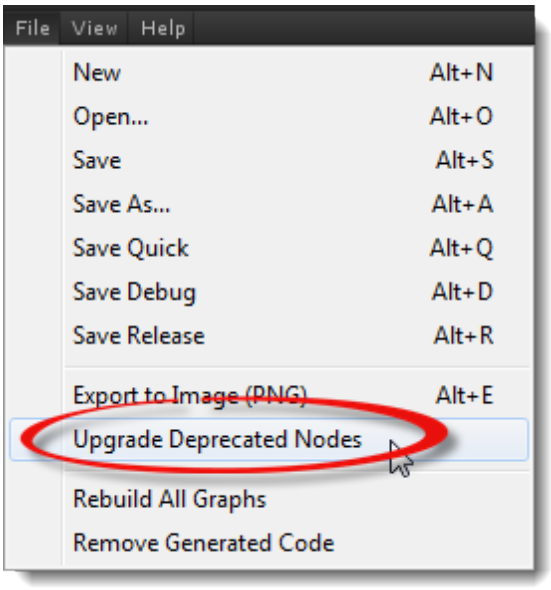
There are three ways to upgrade pink nodes in your graph. The first is to Right-Click them and select "Upgrade Node" option.



You can also use the Upgrade button on pink nodes in the Contents Panel (See "Canvas Navigation" for more information about the Contents Panel):



Lastly, you can have uScript auto-upgrade all pink nodes on your graph by using the "Upgrade Deprecated Nodes" item in uScript's File Menu:


File
View
Help
New Alt+N
Open... Alt+O
Save Alt+S
Save As... Alt+A
Save Quick Alt+Q
Save Debug Alt+D
Save Release Alt+R
Export to Image (PNG) Alt+E
Upgrade Deprecated Nodes
Rebuild All Graphs
Remove Generated Code

Debugging Your Graphs

Sometimes the graphs you make will not work as expected and it will be necessary to track down the problem to fix it. There are several tools at your disposal when needing to debug your uScript graphs.

Preparing to Debug with uScript

Before you start debugging, there are a couple things you should know beforehand.

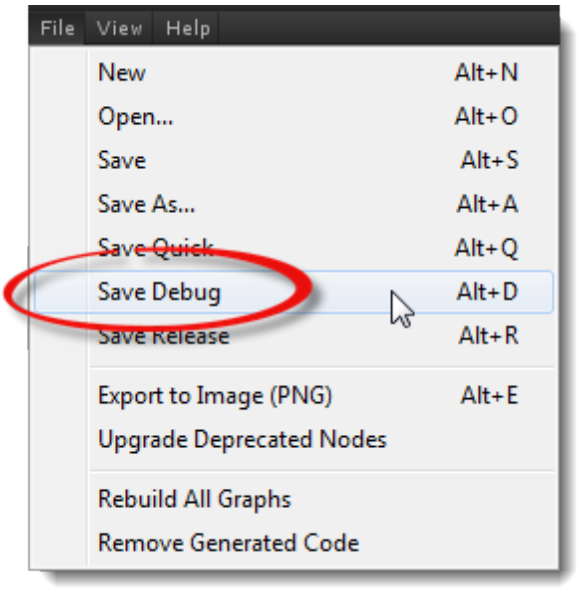
Debug Save Method

uScript allows you to save your graphs a few different ways. It is important that you use the "Save Debug" method to save graphs you wish to also debug. This is because some of the debug methods mentioned below will only work with the graph has been saved in this way.

uScript will add in some extra logic to the exported C# script files it makes to support some debugging features. These debugging features are stripped out for optimization when saved as "Release".

To save a graph with debugging enabled, you can:

1. Select the Save Debug option from uScript's File menu (hot-key *Alt+D*):

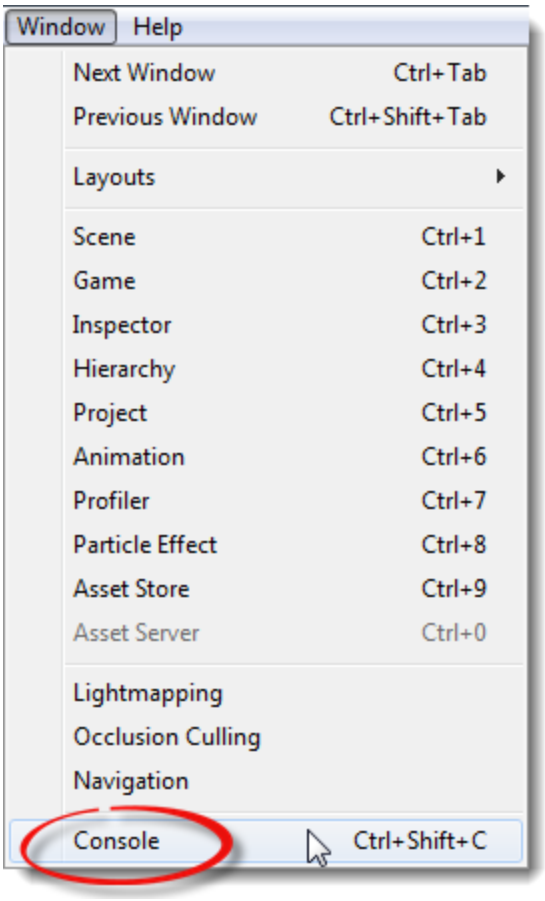


2. Set the default *Save Method* in the Canvas menu to "Debug". This will ensure that whenever you just use "Save" through the File menu, or pressing the save button in the Graphs Panel, it will use the Debug option:

Using Unity's Console Window

Some of uScript's debug methods mentioned here (as well as other important information uScript may be telling you) use Unity's console window for text output. You should be familiar with its use and how to access it in order to take full advantage of the debug methods found below.

Unity's debug console can be opened by either left-clicking on bottom status bar of Unity's editor window, by using the *Ctrl+Shift+C* hot-key combo (use *Cmd+Shift+C* on Mac), or by choosing "Console" from Unity's Window menu.



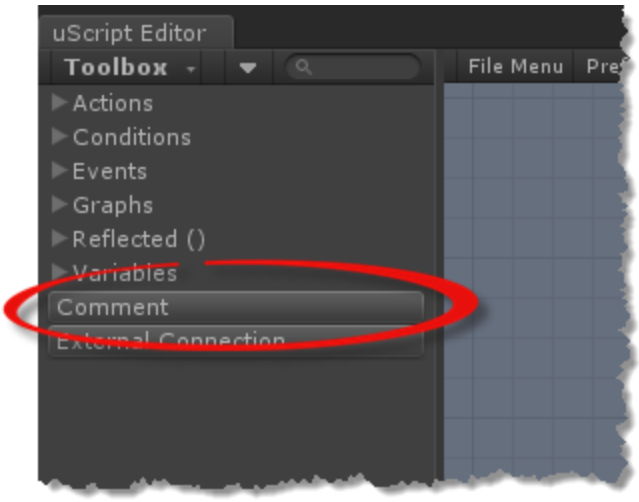
Using Nodes To Debug

There are a few different ways to comment things in uScript. The function of these methods are to provide information to uScript users about important information regarding the graph logic and how it is used either for design time or runtime.

Comment Node

Clearly understanding how a graph has been setup and works is a great first line of defense from needing to debug a graph in the first place. One of the best ways to do this is by using Comment Nodes on your graphs.This is a special node that allows you to provide text information right on the uScript Canvas for your graphs. You can place as many of these as you wish and also customize its background and text colors. These nodes also are positioned "behind" all other nodes so you can also use them to "box in" nodes to help provide more visual context and sectioning of you graphs. This can help greatly for graph organization and readability-- especially on larger graphs.

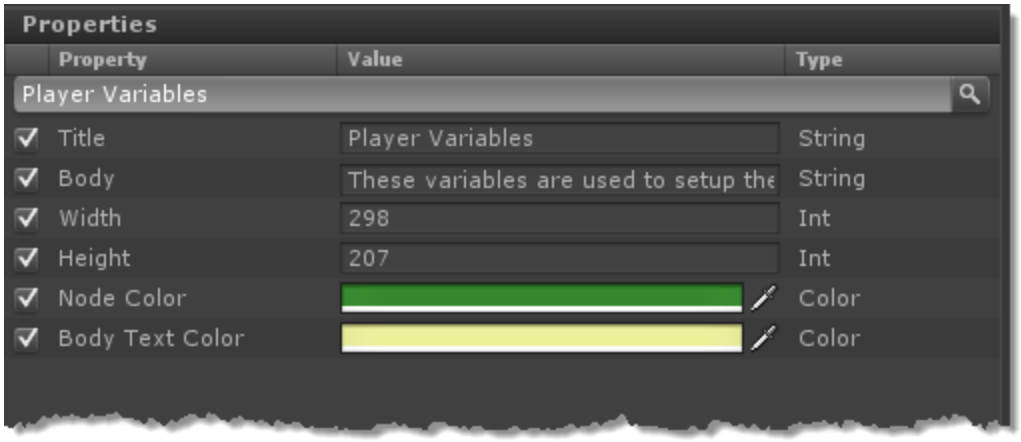
To place a Comment Node on a graph, you can select it from the top level of the Toolbox, or use the hot-key combo *C+LeftMouseButton* to place one at a specific location on the Canvas.



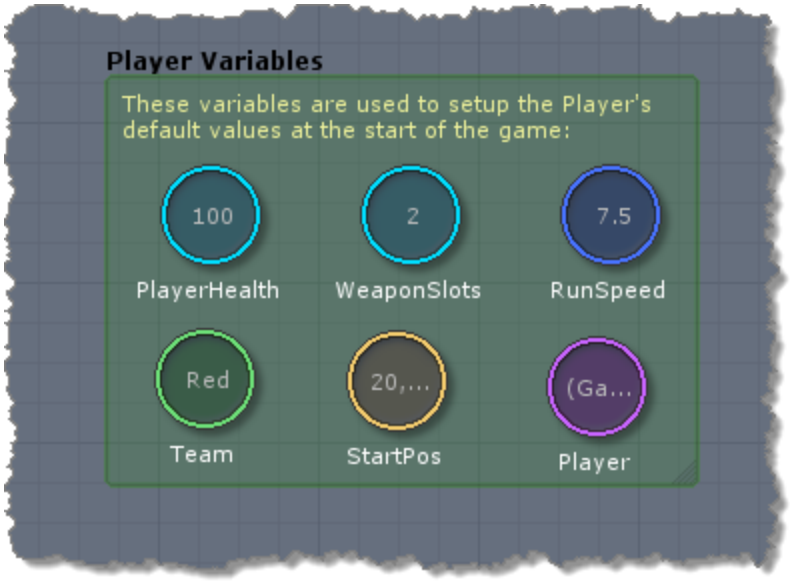
Once placed, you will see a default Comment Node appear on the canvas:

You can move it around like any other node by selecting it. The comment Node can also be resized. To do this, left-click on the resize handle at the lower right of the Comment Node and drag it out to the desired size:

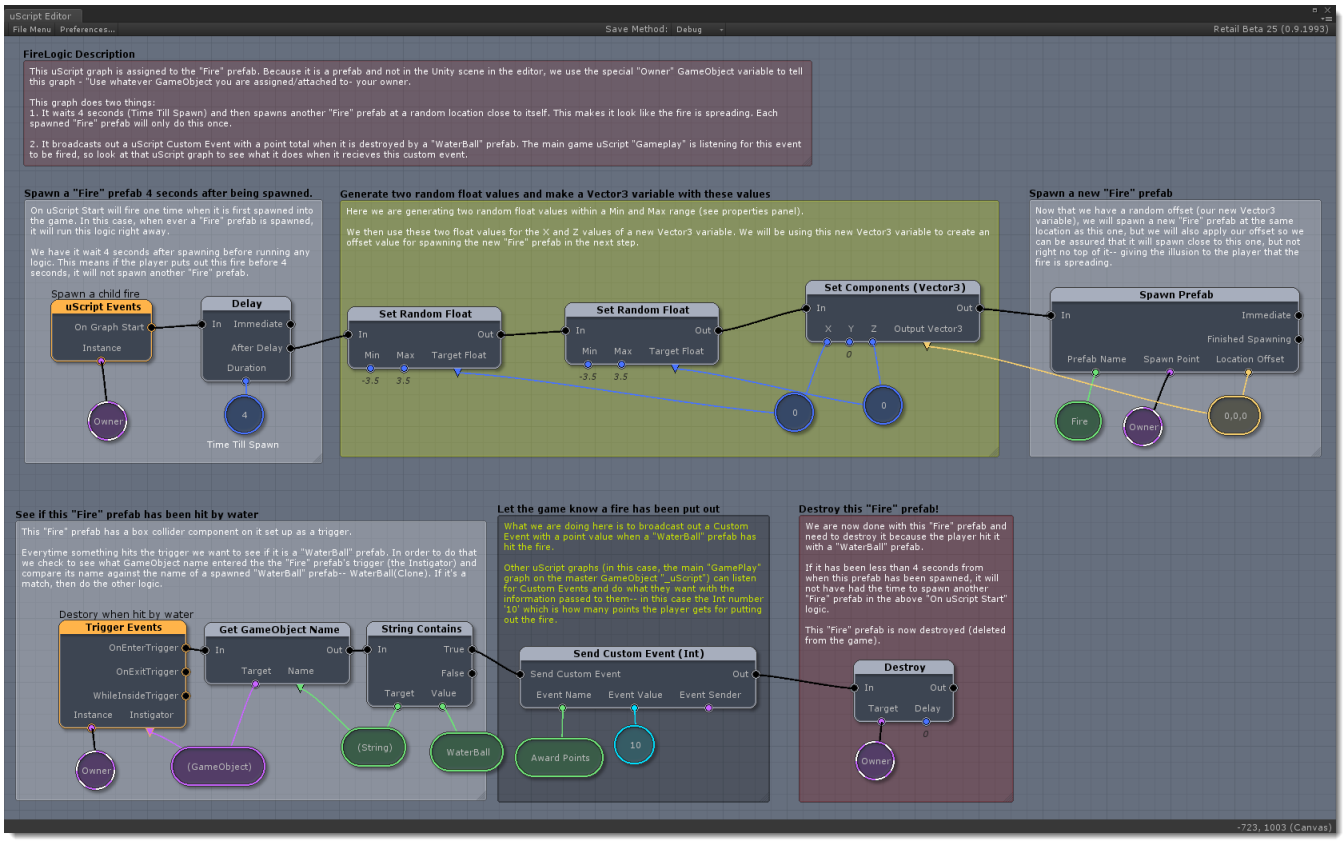
You can edit the properties of the Comment Node by selecting it and editing the settings in uScript's Properties Panel:



Here is an example of a Comment Node setup to hold a bunch of global variables for the player. Notice that "normal" nodes such as these variable nodes can be placed in front of the comment node:



Here is a more complex example of a graph using several Comment Nodes to help explain the details of how the graph works:

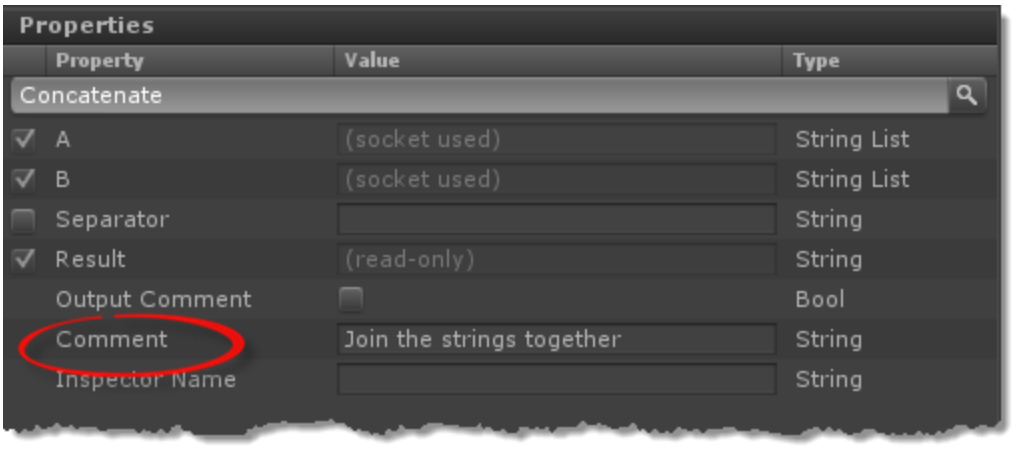


Information Output

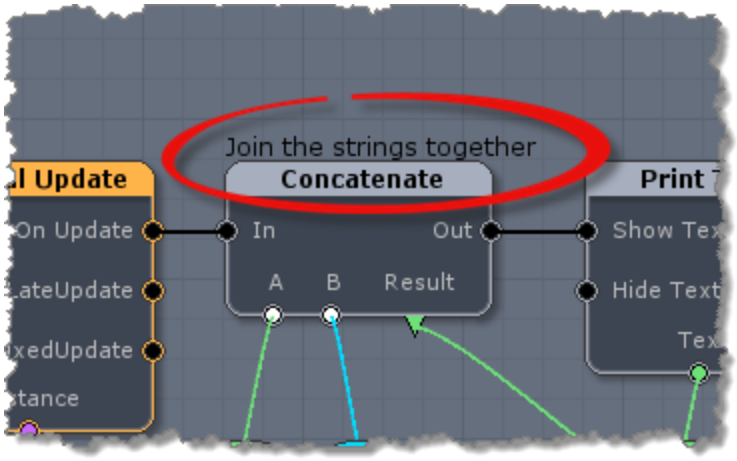
There are three fairly simple ways to output text that will allow you to see the values of variables or if specific logic is even executing at all.

Comment Property On Nodes

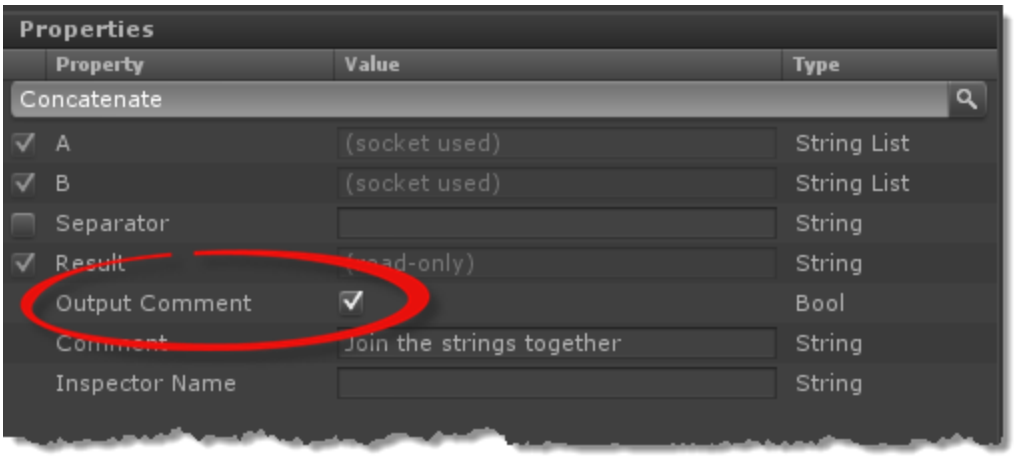
The comment property found on most nodes allows you to place a line of text directly above a node on the uScript Canvas. To enable this, you just need to provide text in the node's Comment property field in the Properties Panel:



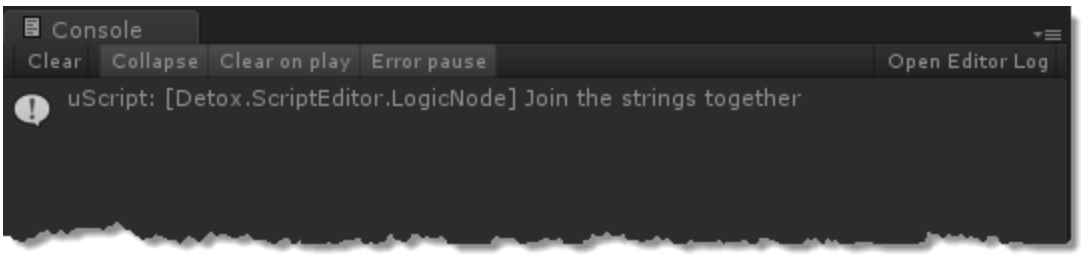
Once this is done, the comment text will appear above the node on the graph:



You can also have this comment print out to Unity's console window each time the node executes (receives a signal). To do this, simply check the node's "Output Comment" property in the Properties Panel and then make sure you are saving the graph using the "Save Debug" option (graphs saved as "Release" will not output debug comments to the console):



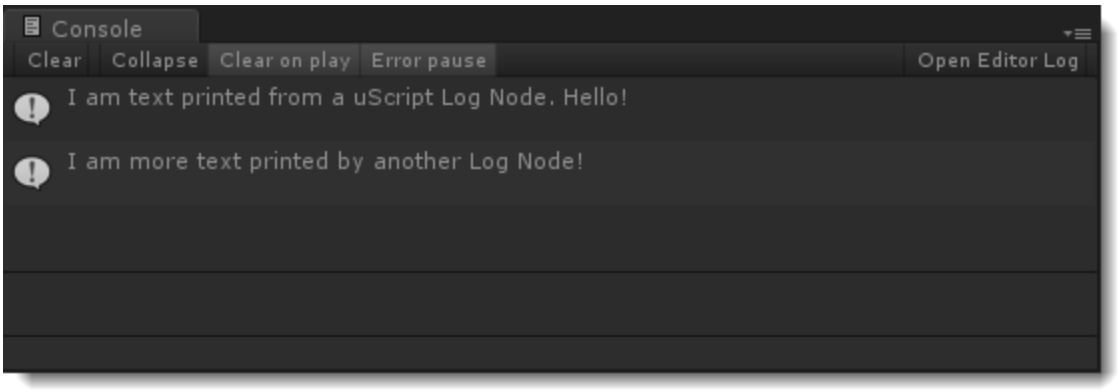
Once this is done, the comment will appear in the console window, also pricing information as to what graph and node type the comment came from:



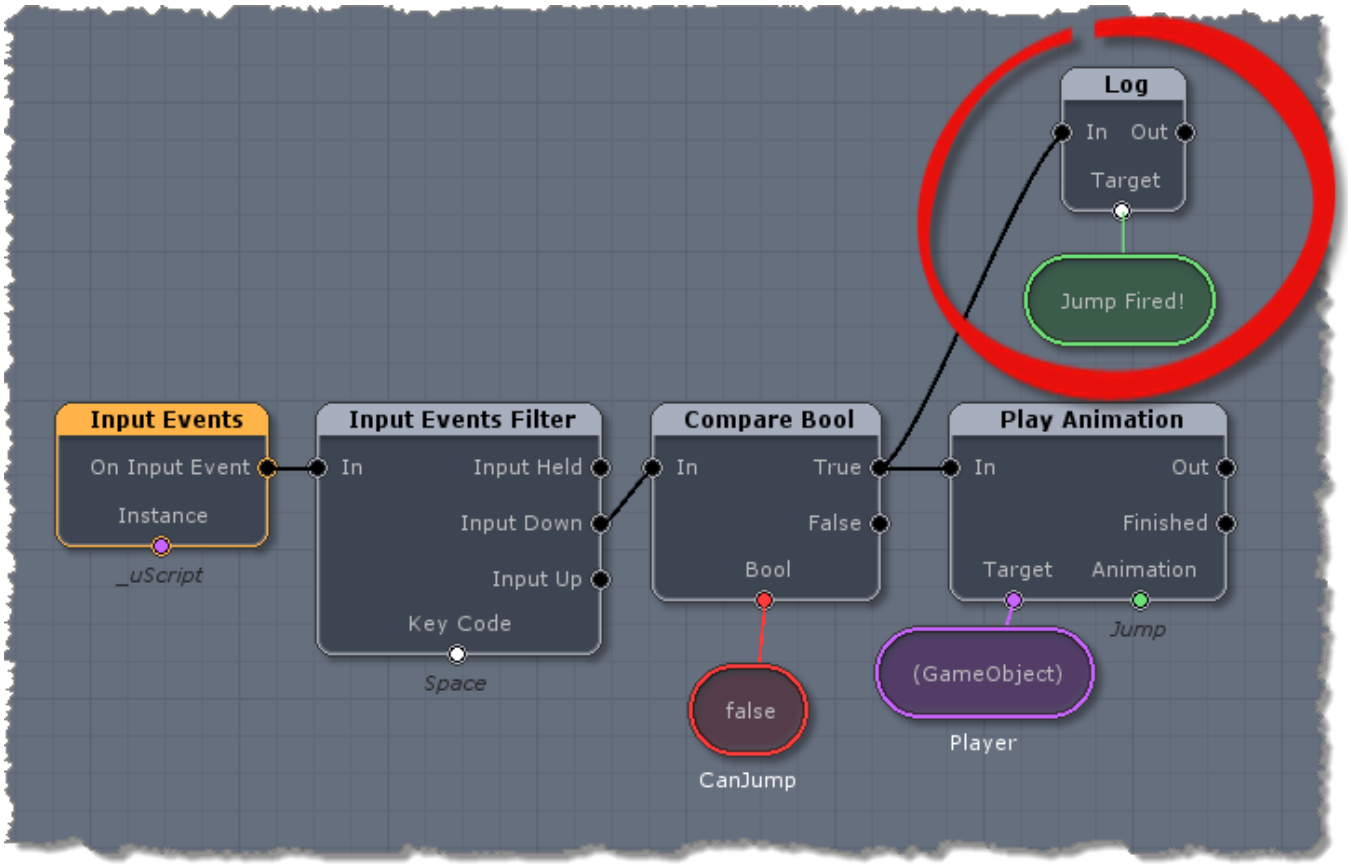
Log Node

The first is to use the Log Node. This node allows you to output values to the console and uses Unity's `Debug.Log()` script command.

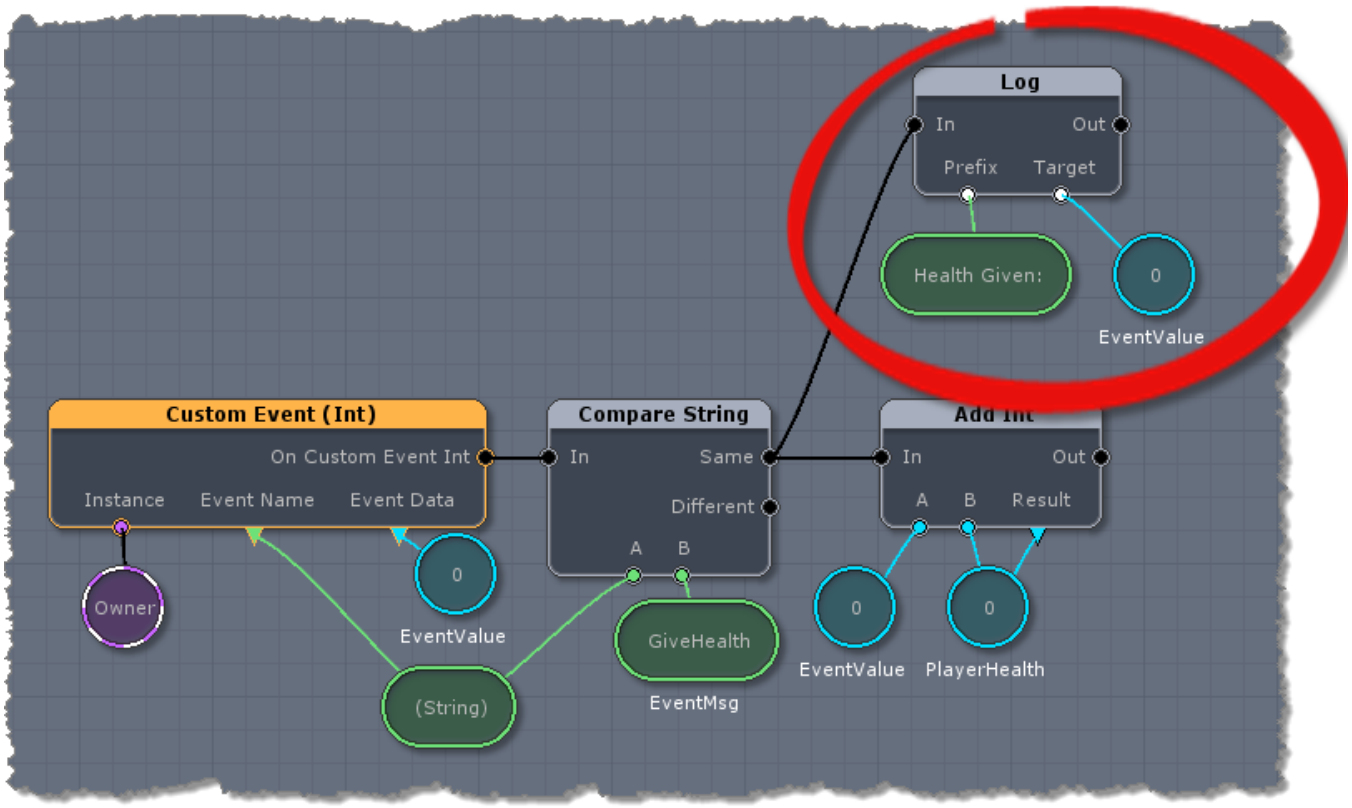
This node allows you to easily print out any variable type value and also to optionally include text before and after that value to Unity's console window.



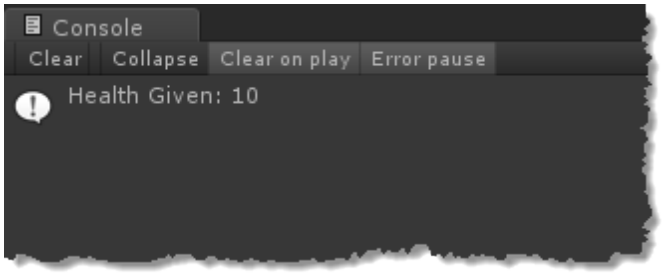
Here is a simple example of using a Log Node to print to the console whenever the player jumps. Notice how the debug logic is not inline with the other logic. This makes it easy to disconnect or remove once no longer needed.



You can also use the Prefix or Postfix sockets to have text or other values appear before or after the Target value. This example shows adding some text before the value to better help the user find the console output text and also know what that value represents.

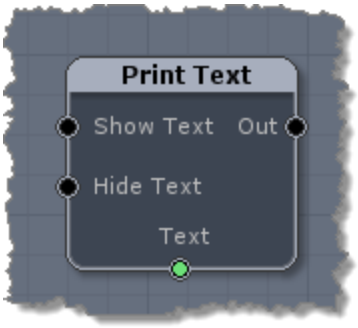


The resulting console output would look like this:

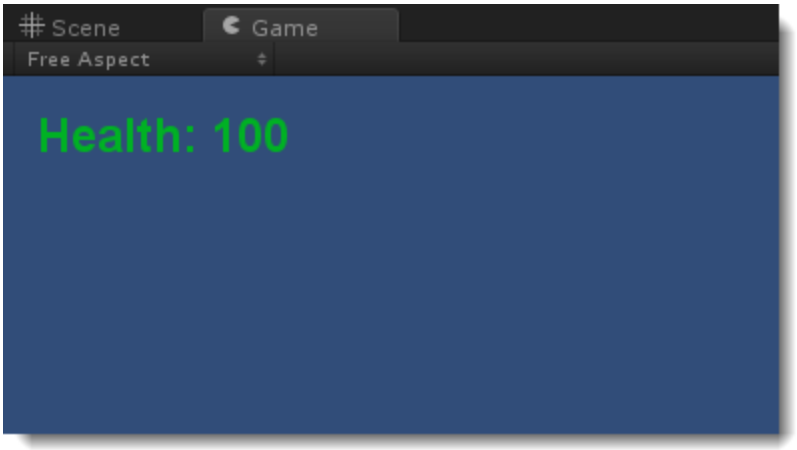


Print Text Node

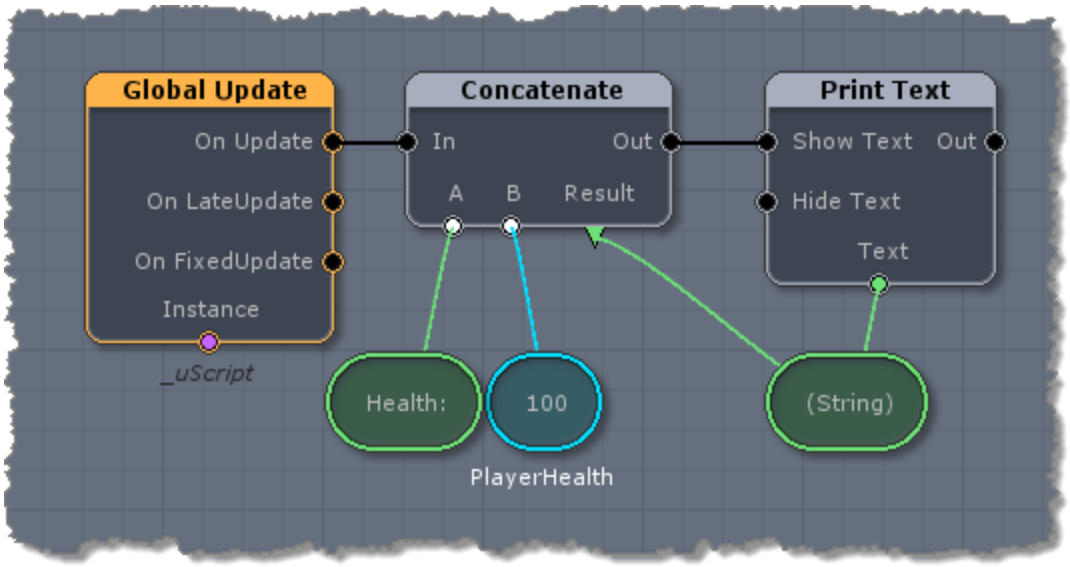
If you would rather print text to the screen instead of Unity's console window, you can use the Print Text Node. This node is very useful when you are running your game on a target device where there is no Unity console to read output text (such as a mobile phone).



Once set up, this node will print text directly onto the game screen:



The above output was generated using the following graph logic. Please note that the Print Text Node's font style or properties will not be applied when running on mobile devices under Unity 3, as Unity 3 does not support that ability:



Note! *- the Concatenate node is very useful to use with the Print Text Node because it allows you to input any variable type and to also provide extra text to the debug output (see the above example).*

Other Nodes
There is also nothing stopping you from using other nodes to help with debugging. For example, you could play a certain sound with a Play Sound node based on logic you specify. Be creative and I'm sure you will think of others that could help you in special situations!

Visual Debugging

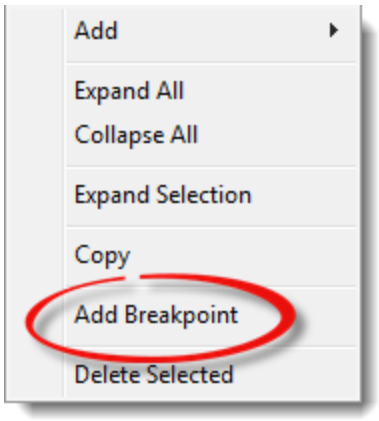
uScript has the ability to pause graph execution in the editor by setting breakpoints on nodes. When you do this, the game will pause execution when that node is about to fire, allowing you to review the state of variables on your graph and to also step through execution using Unity's pause/play system.

Note! *- this system requires that you use the Debug Save option for the graph you wish to visually step debug.*

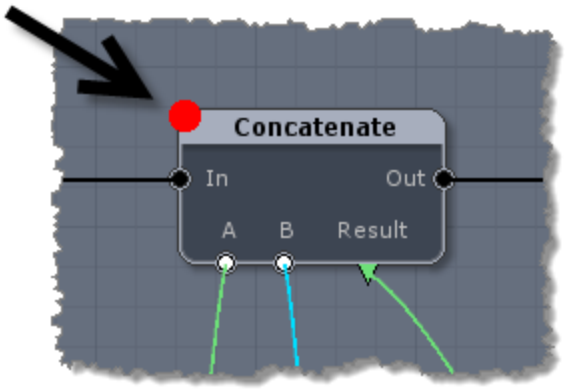
Using Breakpoints

uScript uses breakpoints that you specify to tell uScript where to pause execution of graph logic while the game is running. You will need to re-save your graph (in Save Debug mode) when setting or removing breakpoints form nodes.

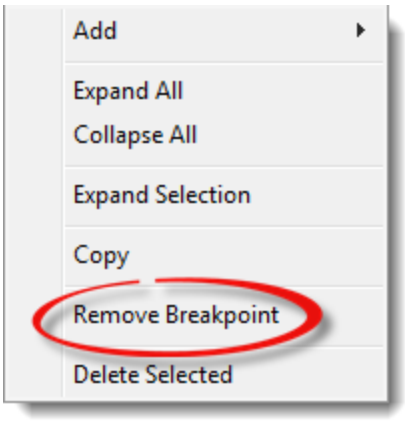
To set up a breakpoint on a node, simply right-click on the node and choose "Add Breakpoint" from the context menu:



Once you do that you will see a red dot at the top left corner of the node. This let's you know there is an active breakpoint set on the node:



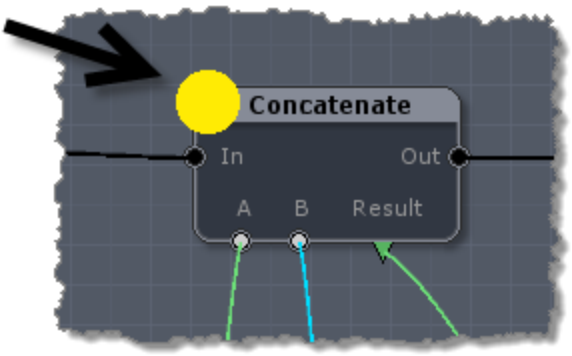
To remove an existing breakpoint from a node, just right-click the node again and choose "Remove Breakpoint" from the context menu:



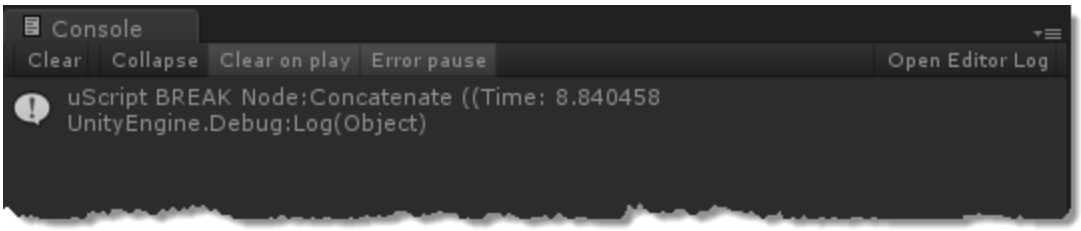
Hitting A Breakpoint

When the game is running and a node with a breakpoint set on it is about to execute, the following things will happen:

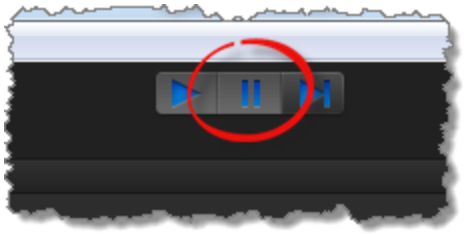
1. uScript will pause the game's execution and change that node's breakpoint from a red dot to a larger yellow dot to show you what node the game is currently executing:



2. uScript will output to Unity's console window with information about which node caused the break:



3. Unity's Pause button in the editor will activate:



Once the game's execution has been paused, you can then either restart execution by pressing the Pause button again, or press the Step button to the right of the Pause button to step through the game manually. If you un-pause game execution, the game will pause again whenever another breakpoint is reached.

For more information on using Unity's Play Mode system, please see the Unity documentation.

Unity MonoDevelop Debugging

Lastly, because uScript generates C# source code, you could also use Unity's MonoDevelop debugging methods to directly debug the exported script code from uScript. Please see the Unity documentation for information on how to do this. Currently it can be found here: http://docs.unity3d.com/Documentation/Manual/Debugger.html

Script Generation

uScript works by converting your visual graph logic into actual C# programming code that Unity will use for your game.

Rebuilding Generated Code Files

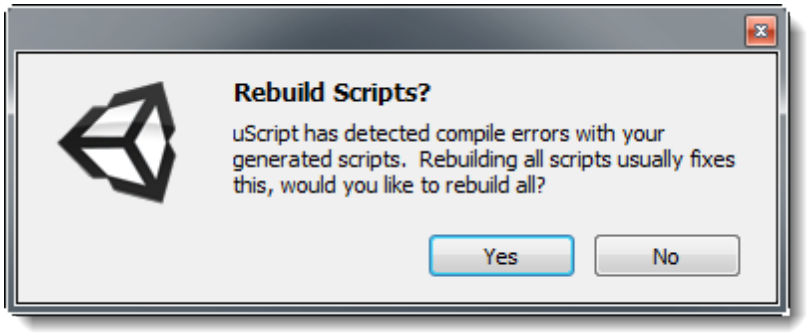
Sometimes this code gets out of sync and will generate compile errors in Unity (usually from a uScript build upgrade that contains new features and improved nodes). When this happens, you need to delete/rebuild your generated scripts so they are in sync with the new uScript build and can take advantage of the new features and enhancements.

Delete Generated Files

There are two methods to delete your generated uScript code files-- through the uScript editor and manually.

Method 1 - Rebuild Scripts Dialog

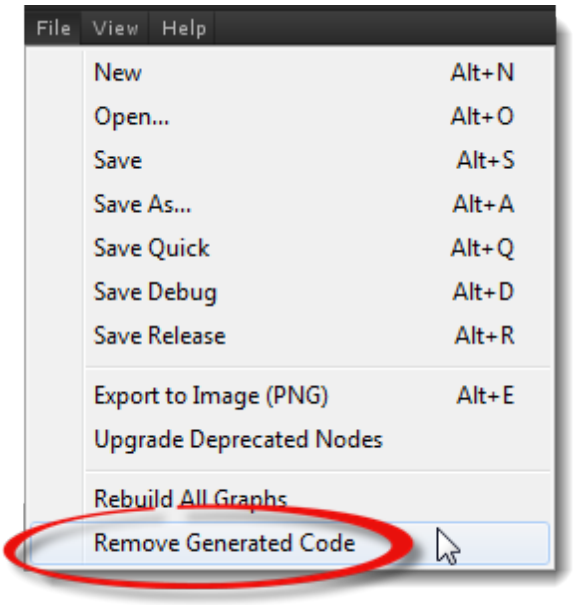
In most cases, uScript itself will detect that it needs to rebuild your uScript's generated code files and ask if you would like to do so in a pop up dialog box. Just press the "Yes" button to have uScript delete and rebuild your generated code files.



> **Note!** - *This method will not work if nodes have been modified (updated or deleted)! In that case, please select "No" from the dialog window shown above and use the "Remove Generated Code" button (shown below).*

Remove Generated Code Button

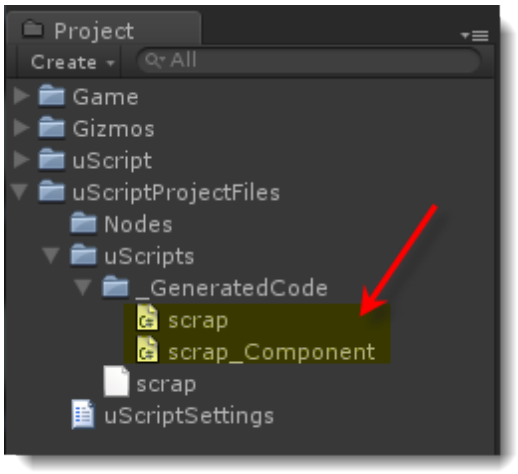
Also, from within the uScript Editor, you can choose to delete your generated code files at any time by pressing the "Remove Generated Code" button in the File Menu above the Canvas.



If the above methods do not work, you can always manually delete your generated code files by following the process below.

Method 2 - Rebuilding Scripts Manually

If all else fails, you can manually delete these files from right within Unity's Project tab. Just open up the "uScriptProjectFiles" folder and then the "uScripts" folder (note, this is different than the root "uScript" folder!). Here you will see a "_GeneratedCode" folder. Open that folder up and delete the C# files (.cs) inside. See this image for an example of where to go to delete the generated files for a uScript graph called "scrap":



Each uScript graph creates two generated C# files, one is the same name as your graph (scrap.cs in the above example) and the other is a component file (scrap_Component.cs in the above example).

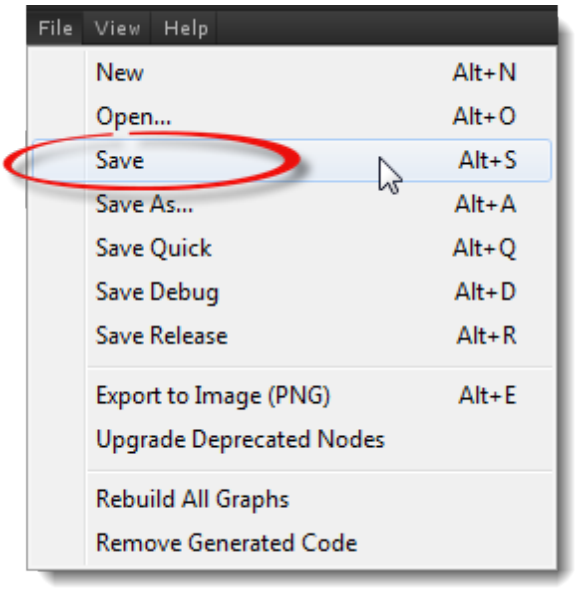
In short, all files inside the "_GeneratedCode" folder are ok to delete. uScript can rebuild them for you from the .uscript files located in "uScriptProjectFiles/uScript". **Do not delete your .uscript files as those are your actual graph files and you will loose your graphs! Just delete the generated .cs files!**

> ***Note!*** *- If you have the Unity console window up, you can usually click on script compile errors there and it will highlight the file is having a problem with. This makes it easy to locate and delete these generated files.*

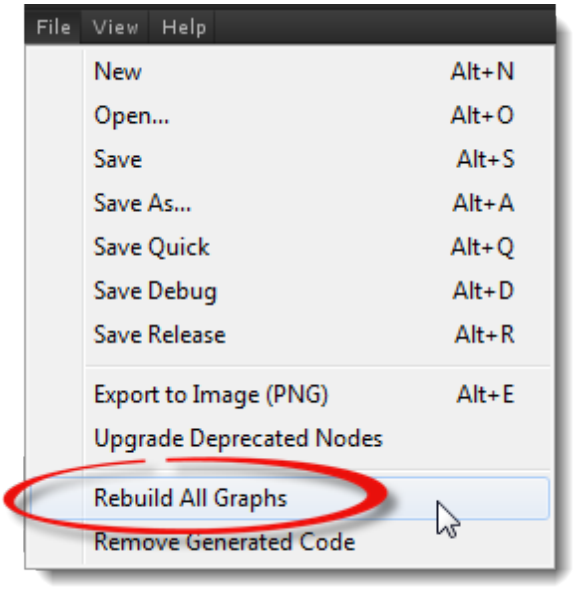
Rebuilding Generated Files

There are two ways to rebuild the generated code files for your graphs.

Save the graph. In the uScript Editor, load the graph you wish to regenerate the code for and hit the "Save" button in the File Menu above the Canvas. uScript will always regenerate your graph's code when you hit the save button.



Rebuild them all at once. In the uScript Editor, press the "Rebuild All uScripts" button in the File Menu above the Canvas. This will rebuild all of your graphs code files all at once.



Creating Graphs

This section of the user guide shows you how to make uScript graphs. In order to best understand this section, you should have fully read the " Working With uScript" topic as it will be assumed you understand these basic concepts and terms of uScript for this section.

Is is also assumed that you understand how to use Unity itself (see "Setting Expectations") and know how to create a new Unity project and install uScript into it (see "Installing & Updating").

Your First Graph

In this tutorial, we will create a very simple graph that will rotate a cube GameObject when you use the arrow keys on the keyboard.

Please note that at first glance this may look like a very complicated tutorial, but it is in fact a very fast and easy graph to make once you know what you are doing (*you could do this in 1-2 minutes total once you know what you are doing!*). This tutorial is **very** detailed with many small steps, explanations, and tips along the way to try and make sure it covers almost any level of user. Take the time to read each step carefully and you will learn lot's of helpful things along the way!

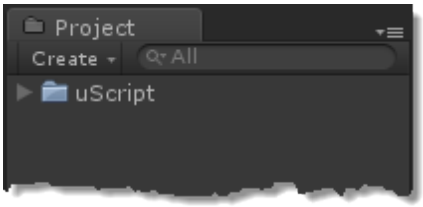
Project Setup

This section of the tutorial focused on creating a new Unity project with uScript installed and setting up the cube GameObject that we will control.

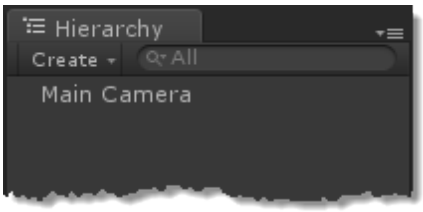
Step 1 - Open Unity and create a new, empty Unity project called "*MyFirstGraph*".

Step 2 - Install the latest version of uScript into this project (see "Installing & Updating" if you need help installing uScript).

You should now have an empty project with a uScript folder showing in the Project tab in the Unity editor:



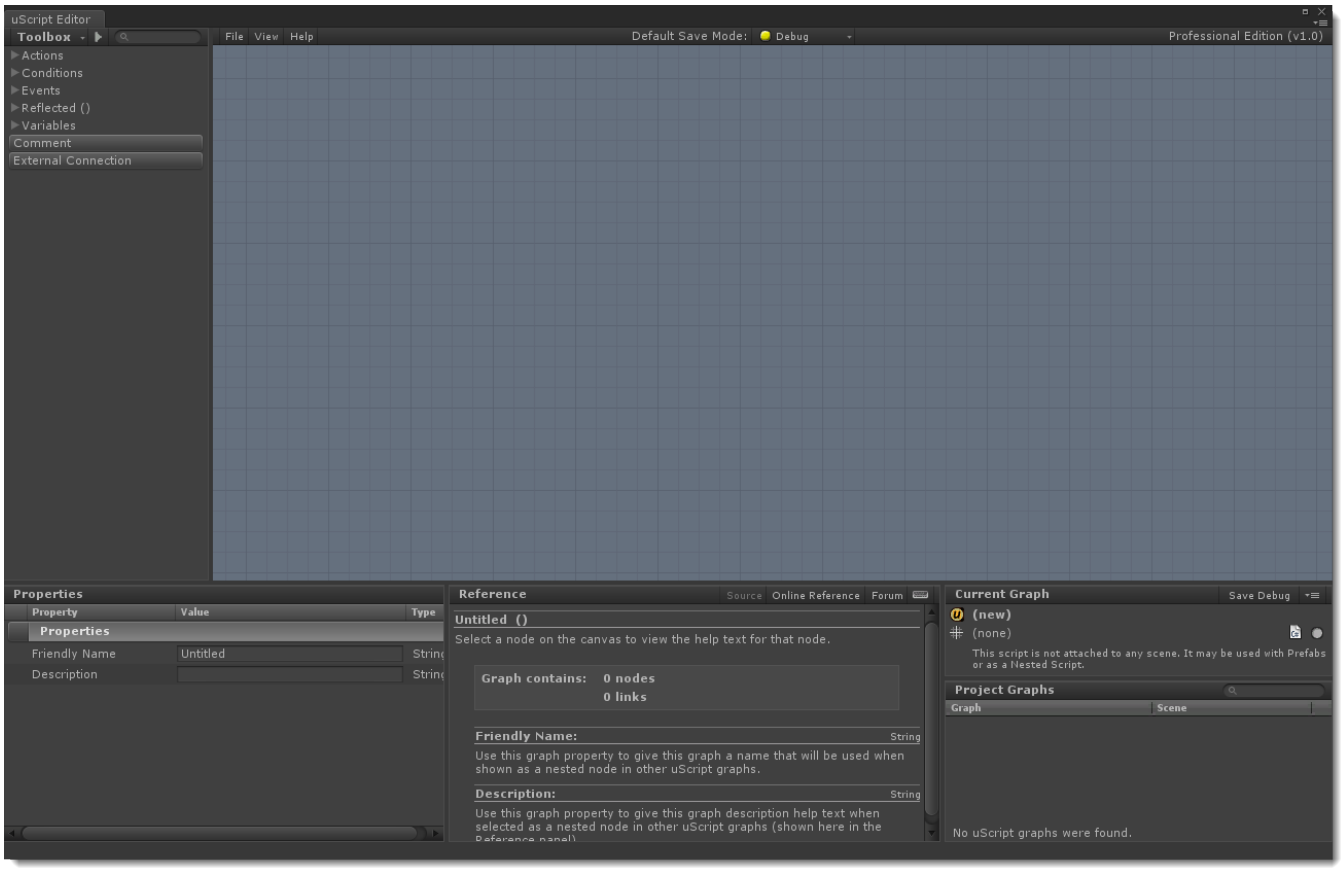
You should also see a single camera GameObject called "Main Camera" in the Hierarchy tab in the Unity editor:



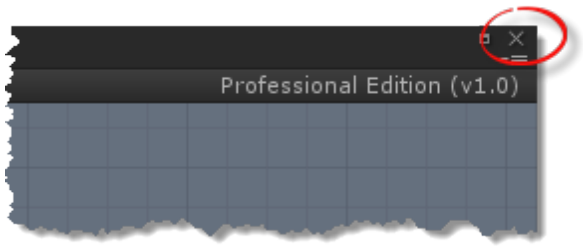
Step 3 - Open the uScript editor for the first time by going to *Tools/Detox Studios* in the Unity menu bar and selecting *uScript Editor*. Note that you can also use the hot-key *Ctrl+U* (*Cmd+U* on Mac) to run uScript if you like:

When running uScript for the first time, you will get several windows that pop up that might include asking you to enable/disable auto-update checking, a license agreement window, and a welcome window with helpful links.

You should now see the uScript window appear:



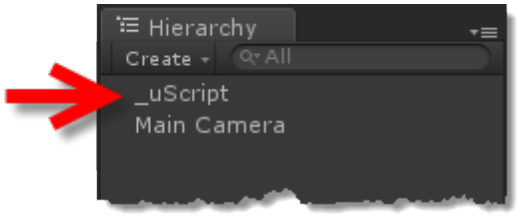
Step 4 - Go ahead and close the uScript editor window by either clicking on the window's close button at the top of the window (a red dot on the left for Mac, or a X button on the top right for Windows):



Notice that uScript adds several things to your Unity project when it is run for the first time:

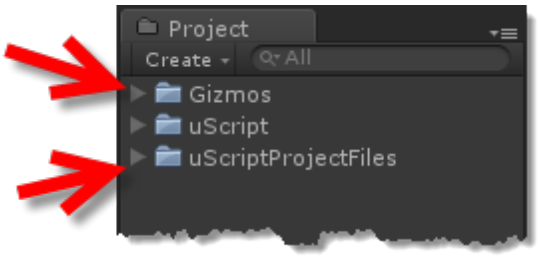
A new GameObject:

uScript has created a new GameObject in your open scene named "*_uScript*". This is the Master GameObject that uScript will use for many default things it does. For now you can just ignore it. You can see this new GameObject in the Hierarchy tab in the Unity Editor:



Two new root directories:

The Project tab in the Unity editor now has two new directories in the root:



***uScriptProjectFiles* -** uScript creates this folder to store all the uScript files that are specific to this project. It is important for uScript to store all your project uScript content here and not in the uScript folder so that you can easily update uScript without accidentally overwriting or deleting your project-specific content.

> **Note! -** *If you use third-party nodes or create your own custom nodes, you should put them in the Nodes folder found within this location. Putting your nodes (or anything else) inside the uScript folder*

structure puts them in danger of being overwritten or deleted if you ever update your uScript installation!

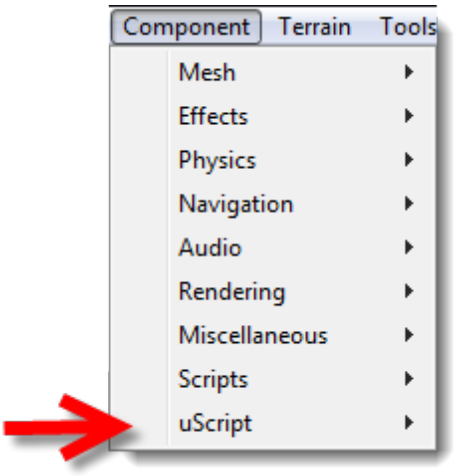
Gizmos - Unity uses this folder to store icons used in the Scene tab of the Unity editor for visualization purposes. We use some uScript icons to help you see what GameObjects are being used by uScript (see the Unity documentation to learn more about gizmos).

A gizmo in the Scene tab:

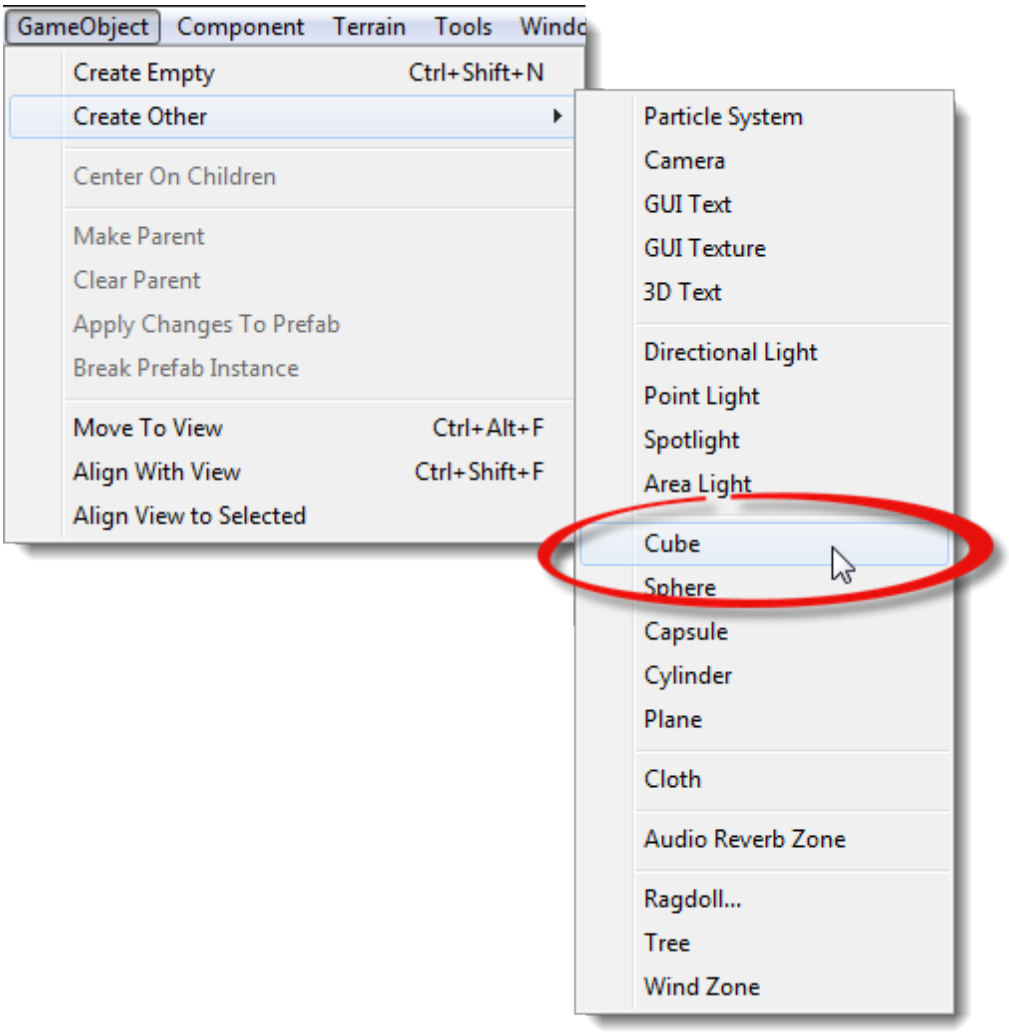
As mentioned above, uScript uses some of its own gizmos in Unity to help you find things in your Unity scenes that are uScript-related. In this case you can see a gizmo on the *_uScript* GameObject that uScript created for the scene (*the little "M" on the Gizmo stands for "Master GameObject"*):

A new section in the Components menu:

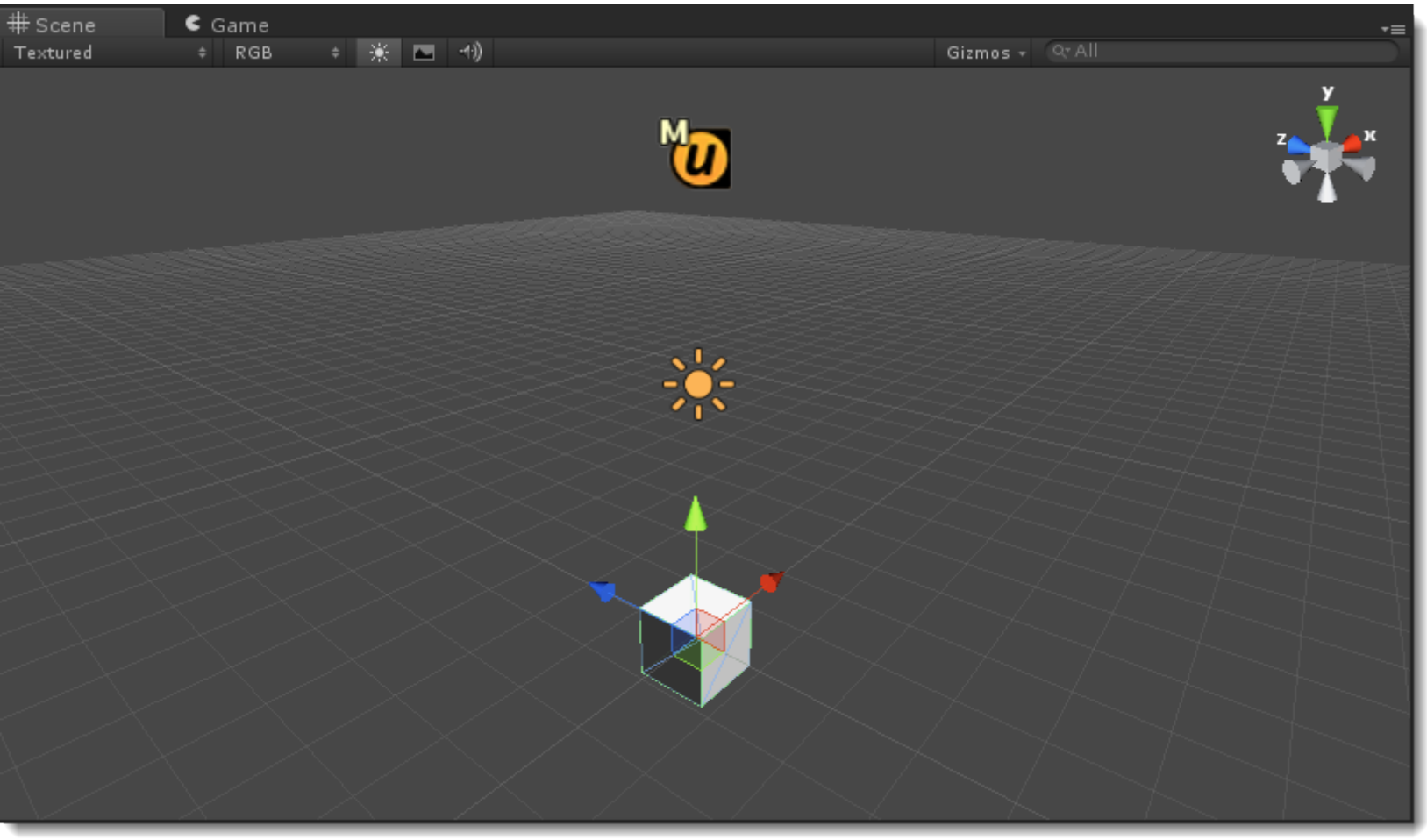
We have also created a new section in the *Components* menu in Unity's menu bar. This will be used in the future to quickly assign Prefab Graphs and other helper script components to GameObjects in your scenes. For now you can ignore it.



Step 5 - Now we need to create the cube GameObject we will be spinning in this tutorial. To do this, just go up to the *GameObject* menu in Unity's menu bar and choose *GameObject/Create Other/Cube* to create a new cube GameObject in your scene called "*Cube*":



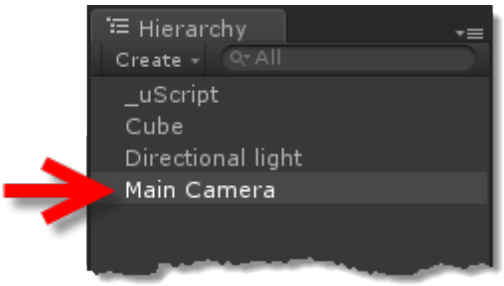
Step 6 - We should also place a light in our scene so we can better see the Cube. To do this, go back to the *GameObject/Create Other* menu in Unity and select "Directional Light" to put a new directional light in to your scene. Feel free to move it away from your cube if you like *(you can do the same with the _uScript GameObject as well if you like-- it won't hurt anything if you move it somewhere else in your scene*). You should now have a Scene view in the editor that looks something like this (if you did indeed move your light an d_uScript objects away form the center of the world):



For my setup above, I changed the **Y** position on the *_uScript* GameObject to "*6*" and the **Y** position of the *Directional Light* GameObject to "*4*" through the Unity Inspector tab.

Step 7 - Lastly, lets get the camera into a better position so we have a more useful view of the Cube when we rotate it. To do this:

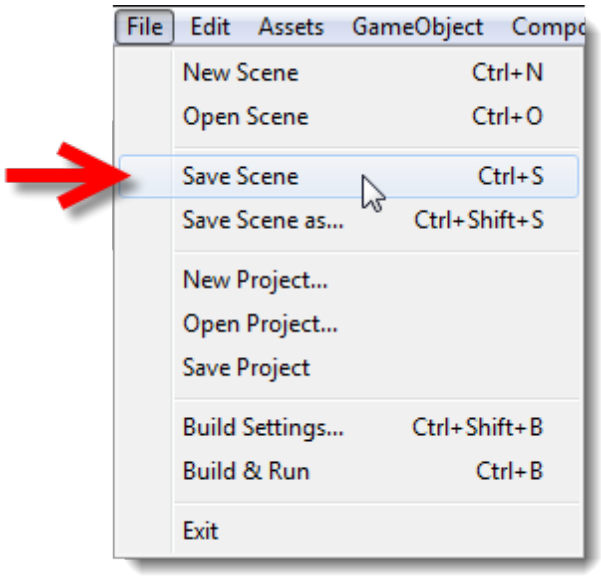
1. Select the Main Camera GameObject in the Hierarchy tab in the Unity Editor:



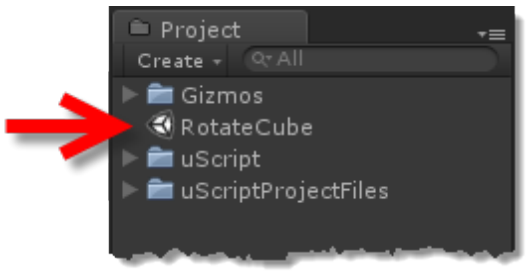
2. In the Transformation section of the Unity Inspector tab, change the following values:

Position **Y**: 3.5
Position **Z**: -5
Rotation **X**: 28

Step 8 - Now we can save our Unity scene before continuing. To do this, just go up to the File menu in Unity's menu bar and select "Save Scene":



You should get a popup asking for a name and location to save the Unity scene. Just call the scene "*RotateCube*" and save it in the current default location. You should now see this file appear in Unity's Project tab:



Our scene is now set up and we are ready to start using uScript!

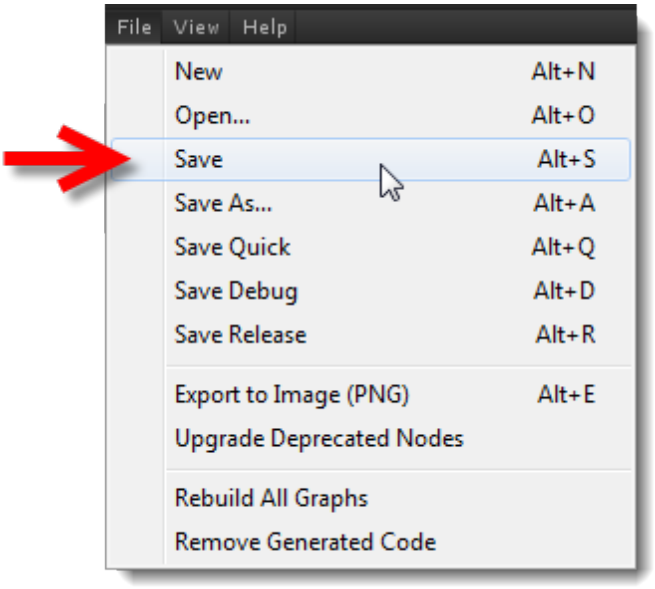
Making the Cube rotate with uScript

This section shows you how to use uScript to make the Cube rotate left and right using a uScript graph.

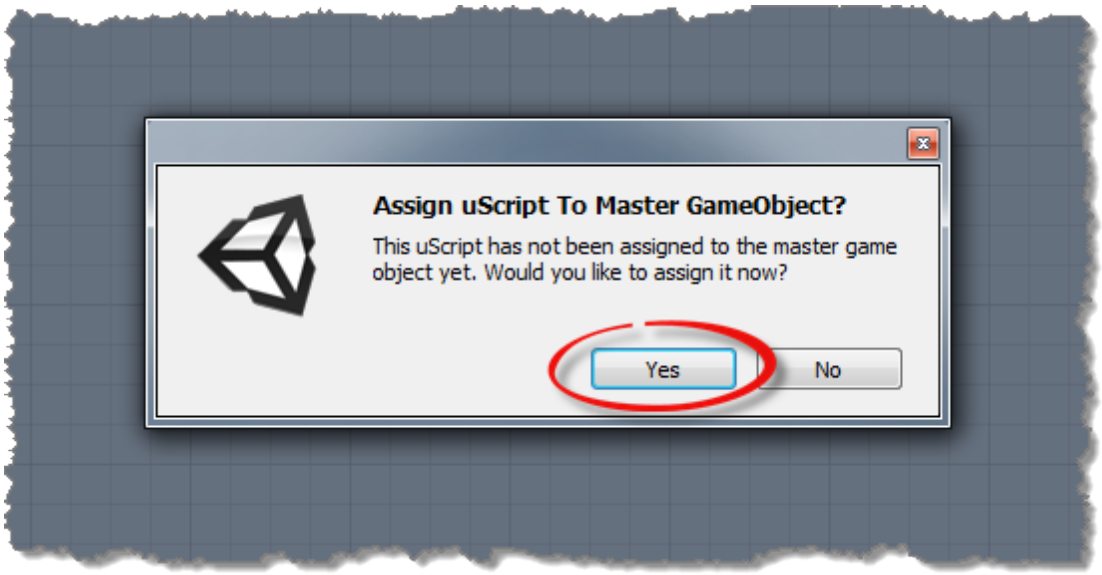
Step 9 - Now let's create a graph to control the Cube's rotation.

1. Open the uScript editor again using either the hot-key (*Ctrl+U* on Windows or *Cmd+U* on Mac) or by going to the Unity menu bar and selecting *Tools/Detox Studios/uScript Editor*.

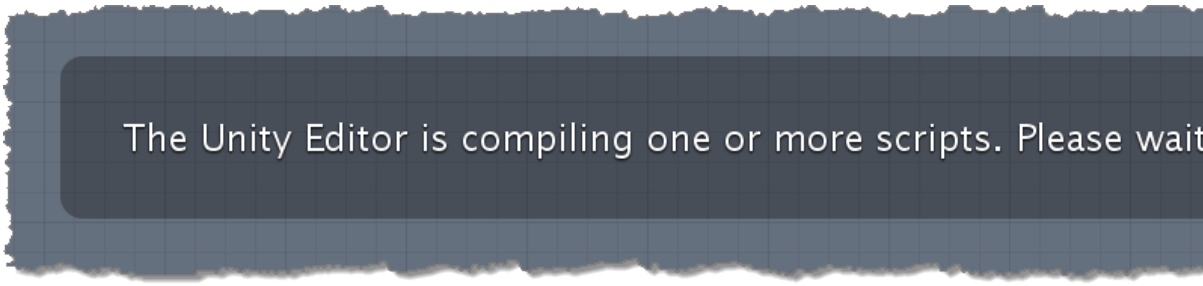
2. Once the uScript editor is open, go to the File menu above the uScript Canvas and choose *File/Save*:



3. Save the file with the name "*CubeController*" in the default location. Note that when you press the Save button, uScript will pop up a window asking you if you wish to assign this graph to the Master GameObject. Because this is a simple scene graph, choose the "***Yes***" button to continue (*if we were making a Prefab or Nested Graph we would have chosen "No"-- more on those in later tutorials*).



Note! - *when you save the graph, uScript will automatically create the C# script files Unity will use in the game (see the "Generating Script Code" section of the "Graphs" topic for more information on this). When Unity detects any new or updated script files in the project it will automatically recompile your scripts. uScript will pop up a notice when this is happening. Just wait it out before continuing on. The speed of your script recompile will vary with your computer and project size/complexity of scripts:*



Once your graph is saved and Unity has recompiled scripts, proceeded to Step 10.

Step 10 - A great way to figure out what you want to do when visual scripting is to create a simple sentence for what you are trying to do and then to break that sentence down into the parts of the graph you will need to make.

In this case the sentence might look like this:

"When the player presses the left or right arrow keys, the Cube will rotate left or right."

Now let's break that sentence down into the things we will need to do in our visual script to make that happen:

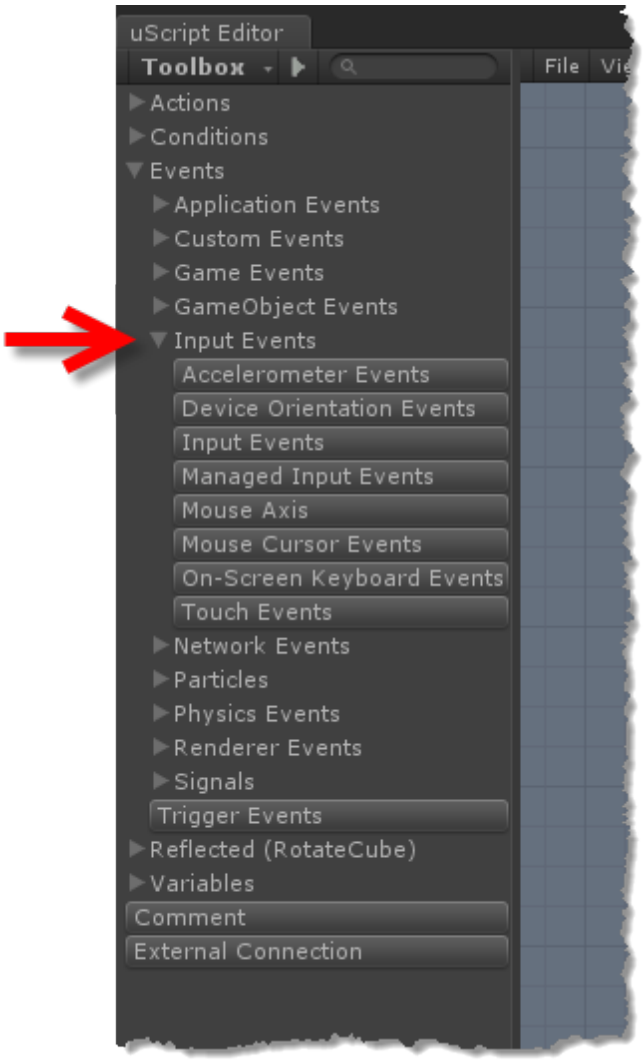
***"When the player presses the left or right arrow keys**, the **Cube will rotate left or right**."*

Ok, let's start with the first bit (*When the player presses the left or right arrow keys*)...

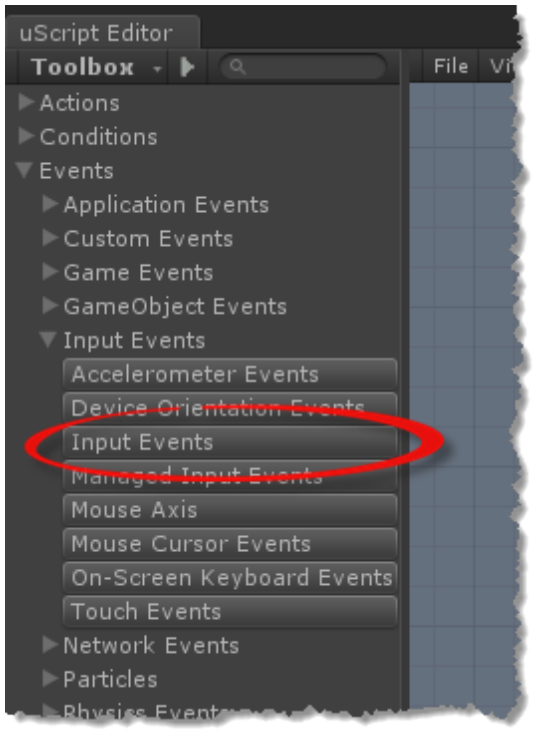
Step 11 - First we need to place and ***Event Node*** into our empty graph. Every uScript graph needs at least one Event Node in order to work. This is because Event nodes are the nodes that listen for the game to do things and then send out signals into your uScript graphs to allow you to trigger actions (Action Nodes). To learn more about nodes, see "Nodes".

In this case we will want to place an *Input Events* event node. The *Input Events* node will fire any time the player presses a key-- just what we need for the "*When the player presses*" part! To place this node in your graph, we must find it in the Toolbox panel of the uScript editor (see "Editor Interface" for information about the uScript editor interface).

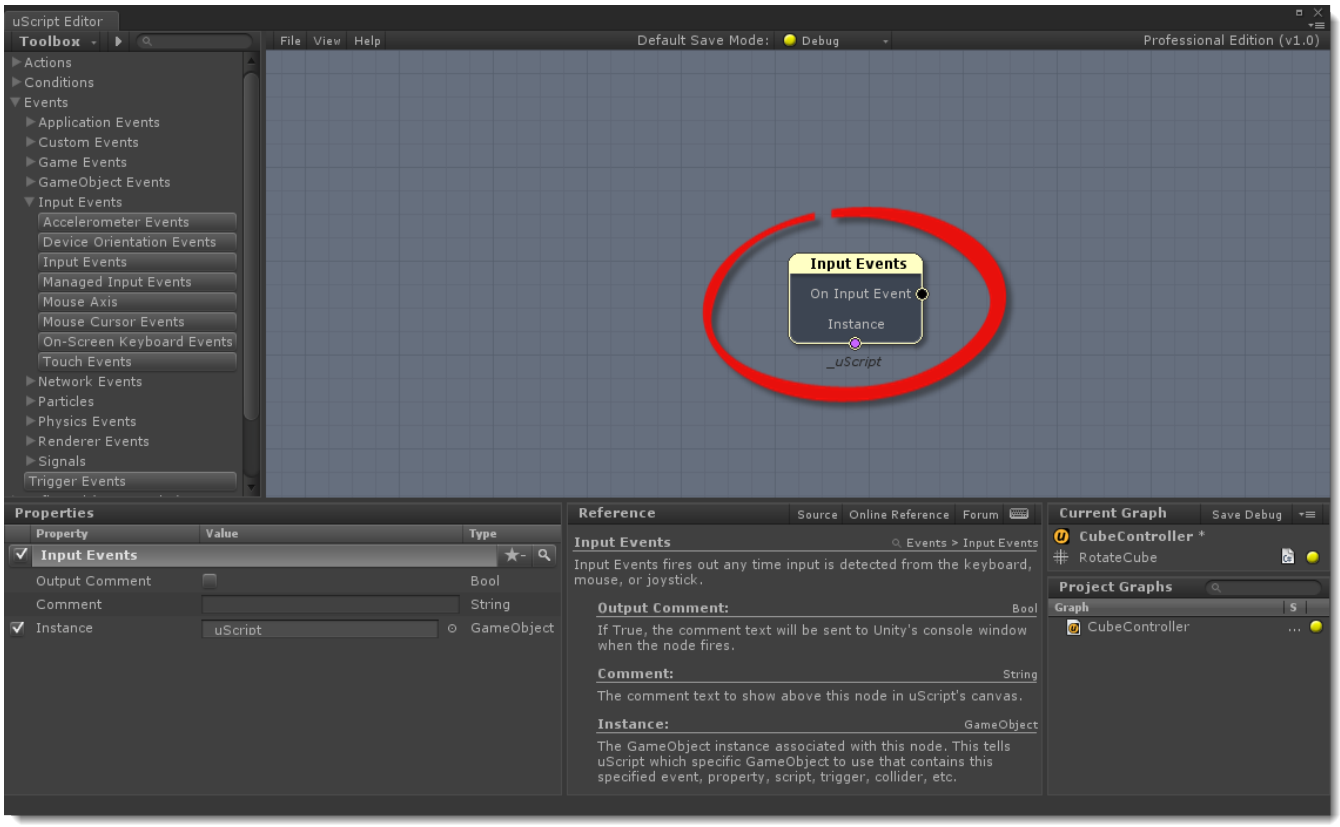
1. To find this node, open the "*Events*" node section of the node tree, then the "*Input Events*" section under that:



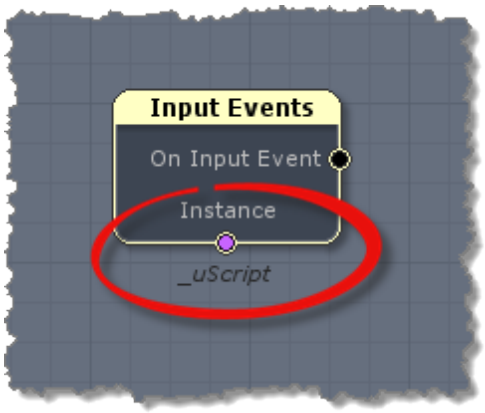
2. Once there, ***left-click*** on the "*Input Events*" node button to place one on the uScript Canvas.



You should now see a single Input Events selected (selected nodes highlight in yellow) in the center of the Canvas:



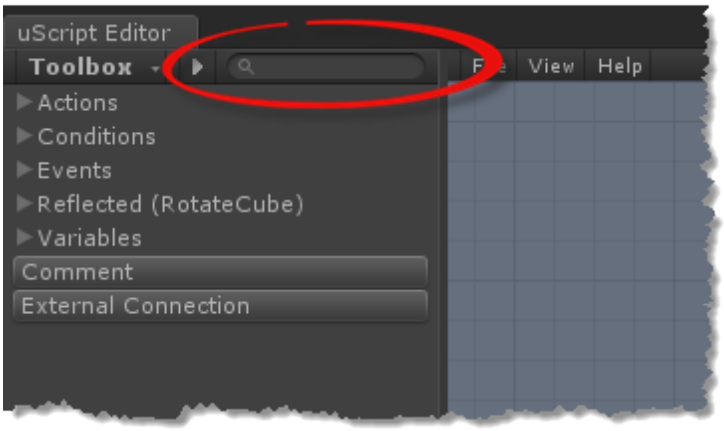
One thing to notice is that this node has an *Instance* socket at the bottom of it (most Event Nodes do). It was automatically assigned to the Master GameObject (*_uScript*):



When a node has an Instance socket, it just means that the GameObject assigned to the Instance is the one responsible for listening to the events from Unity. In many cases, uScript will just assign this to the Master GameObject in your scene automatically so you won't need to worry about it-- but for some nodes you will need to manually assign the GameObject to use for it to work properly (or at all). An example of this would be the *Trigger Events* event node. You would want to tell Unity exactly which trigger GameObject it should use (because you could have many triggers in your scene and it wouldn't know which one to use otherwise!). See the "Instances" section of the "Key Concepts" topic for more information.

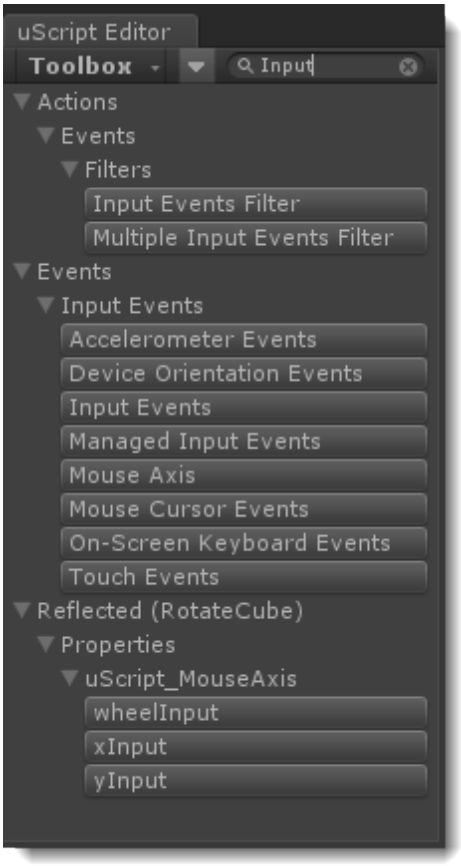
Step 12 - We now have an event node that will fire any time a key is pressed. However, we only care about when the left and right arrow keys are pressed (remember the "*the left or right arrow keys*" part of our sentence?).In order to filter out other key pressed we need to add some *Input Events Filter* action nodes to our graph.

1. As always to place new nodes, go to the Toolbox Panel in uScript. Instead of hunting through the full list of nodes to find the one we want, let's use the filter field to find it quickly:

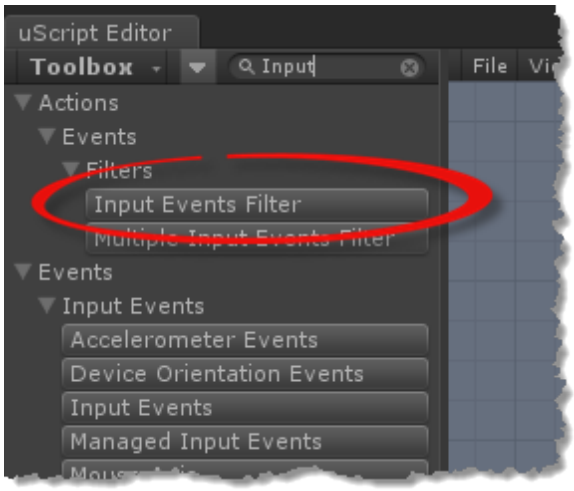


Inside the Toolbox's search filter, type "*Input*" (without the quotes). Notice how the contents of the Toolbox changes to just show anything with the word "Input" in the name. It also auto-expanded

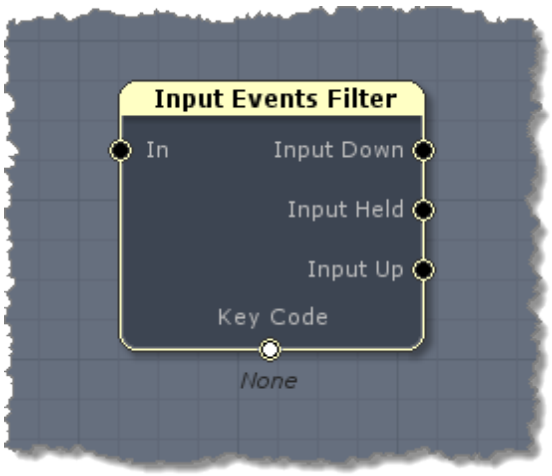
the results for you so you can easily see everything inside the node tree:



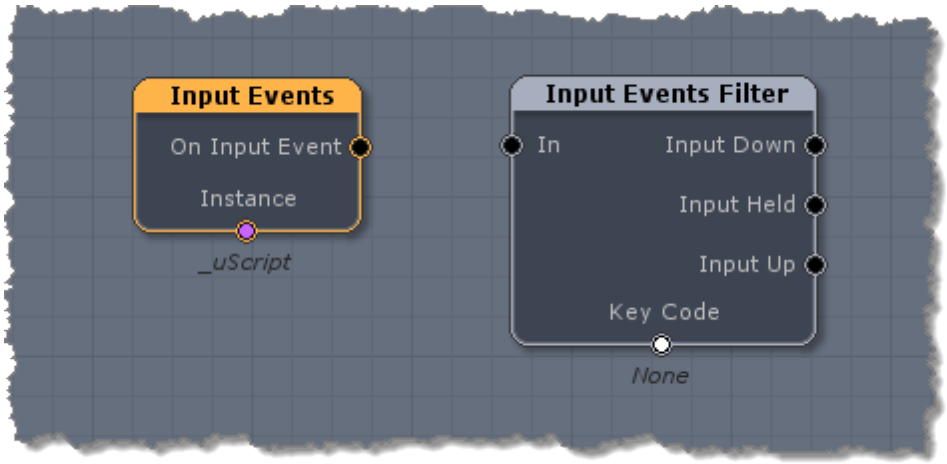
2. Click on the *Input Events Filter* button once to place an instance of the node on the Canvas:



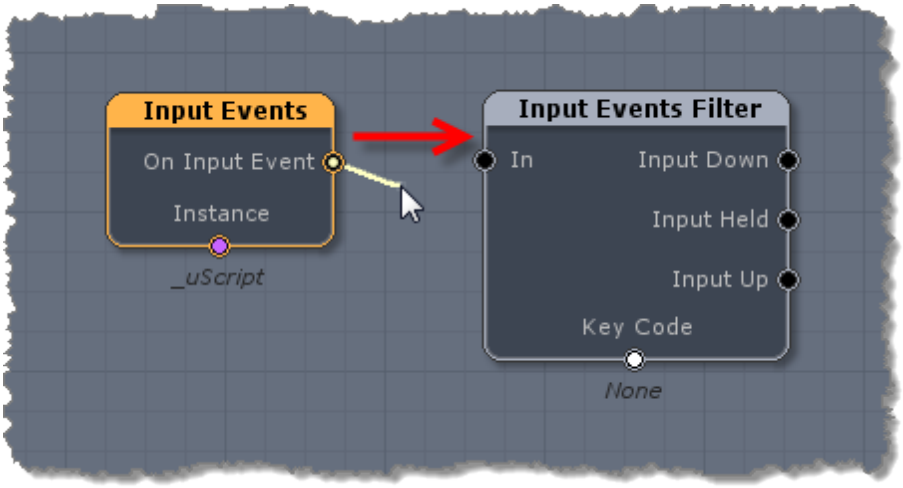
You should now see a selected *Input Events Filter* node on the Canvas-- most likely on top of, and covering, the *Inputs Event* action node if you have not yet moved that node or panned the canvas:



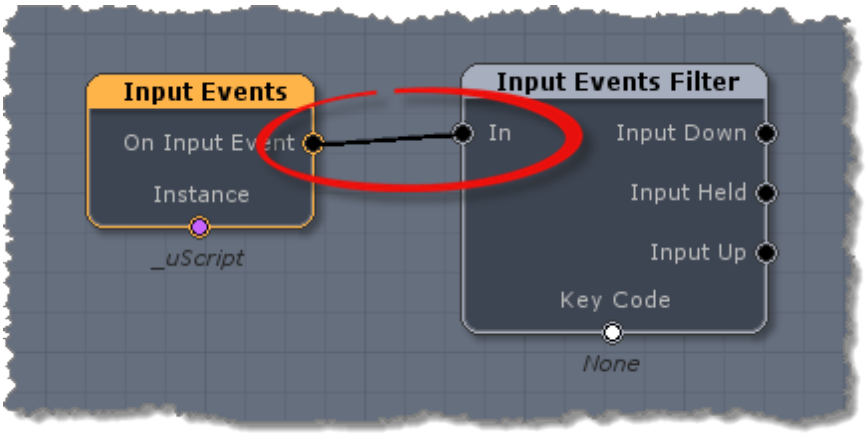
3. Let's move this node to the right of Input Events action node by **left-clicking** on the node and continuing to hold down the left mouse button while dragging the mouse to the right. Once you have done this, you graph should look similar to this:



4. Now let's connect the two nodes together with a Connection Line. To do that, just **left-click** inside the little black "*On Input Events*" socket on the right side of the *Input Events* action node and drag your mouse over to the "*In*" socket on the left side of the *Input Events Filter* node. While doing this you should see a yellow line coming off the node's socket to your mouse pointer. Make sure your mouse pointer is in the center of the "*In*" socket's black area when you let go in order for the connection to be made (it should light up yellow):

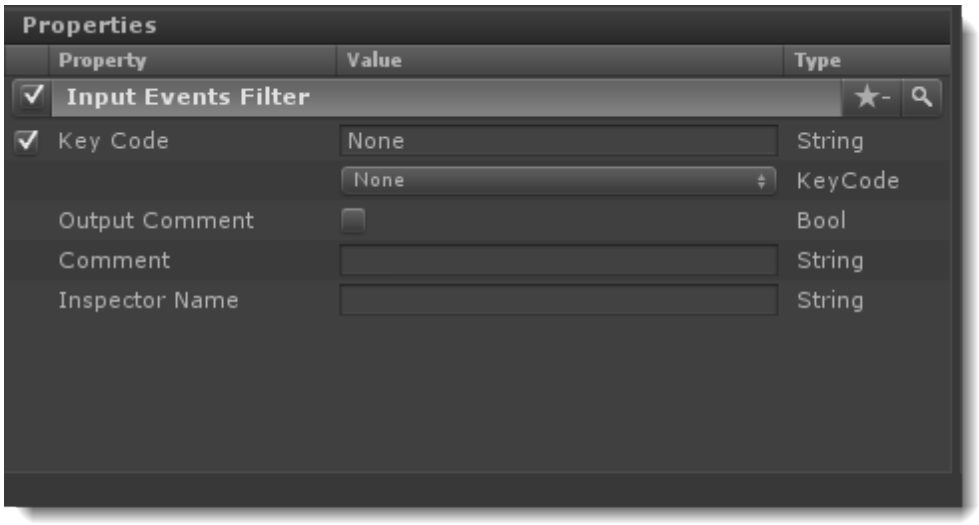


Once the connection is successfully made, you should see the two sockets connected with a black Connection Line:

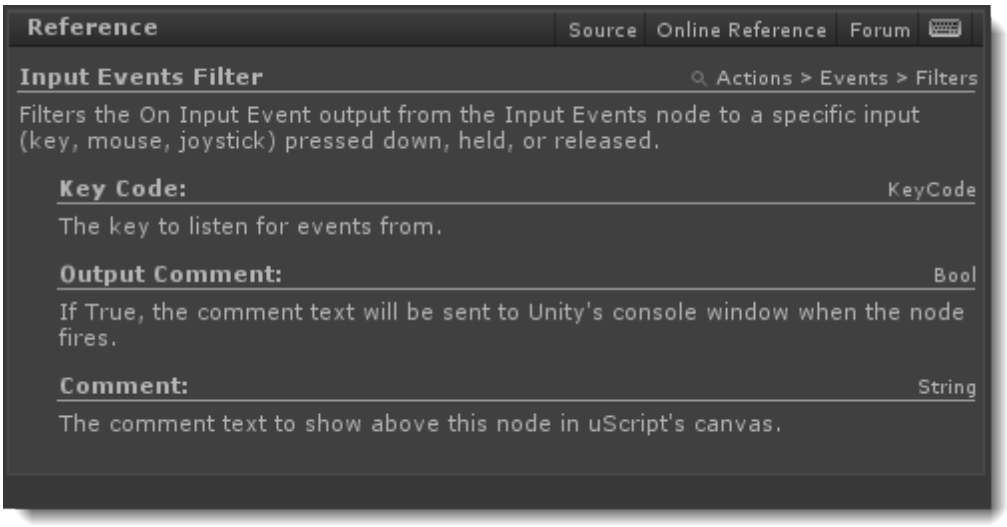


Connection Lines are how you tell uScript what nodes should fire and in what order. In this case we are telling uScript to run the *Input Events Filter* logic once it receives a signal from the *Input Events* action node (which it will every time the player pressed a key). To learn more about Connection Lines, see "Graphs".

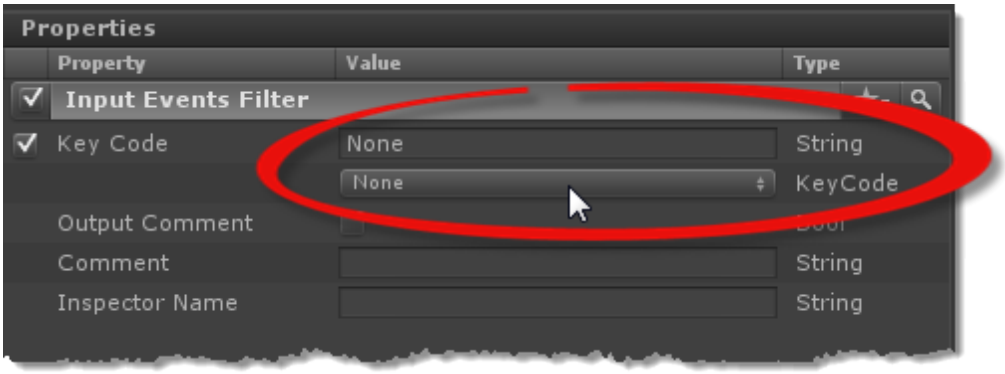
Step 13 - Now we need to set up the properties of the *Input Events Filter* node to tell it what key we want it to listen for. To do this, select the node (it will highlight yellow). When you do this you will see the contents of the Properties Panel in the lower left of the uScript editor change to show you the properties of the selected node (the Input Events Filter node in this case):



Also note that uScript's Reference Panel also updates to display the node help text to describe exactly what that node does and a description of it's properties. This is very useful for learning about what each node does:

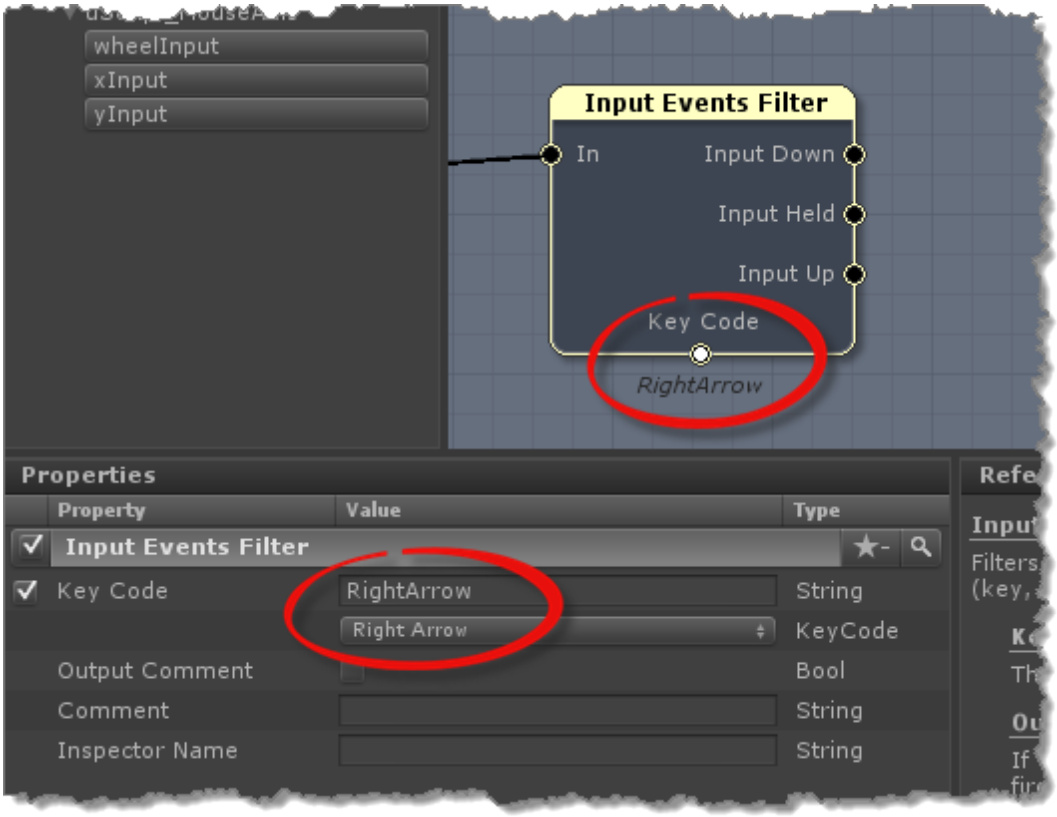


1. As you can see from the node's help text in the Reference Panel, we will want to change its "Key Code" property to tell is what key we are listening for:



Select the pulldown list for this property in the Properties Panel and scroll down/choose "Right Arrow":

Once you have done that, you will see "*RightArrow*" appear in the String field as well as under the node's "Key Code" socket in the graph:



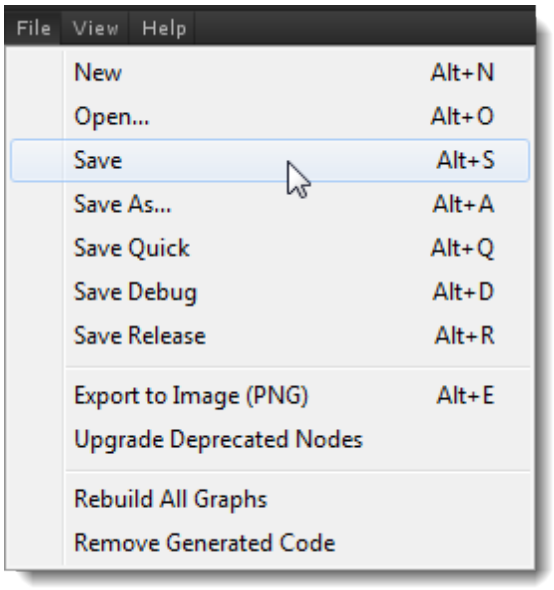
Note! - *you could optionally have typed "RightArrow" into the property's String field and it would find it in the pulldown for you. Once you know Unity's names for the keys this can be hand as their Key Code pulldown*

list can be pretty unwieldy (especially in Windows).

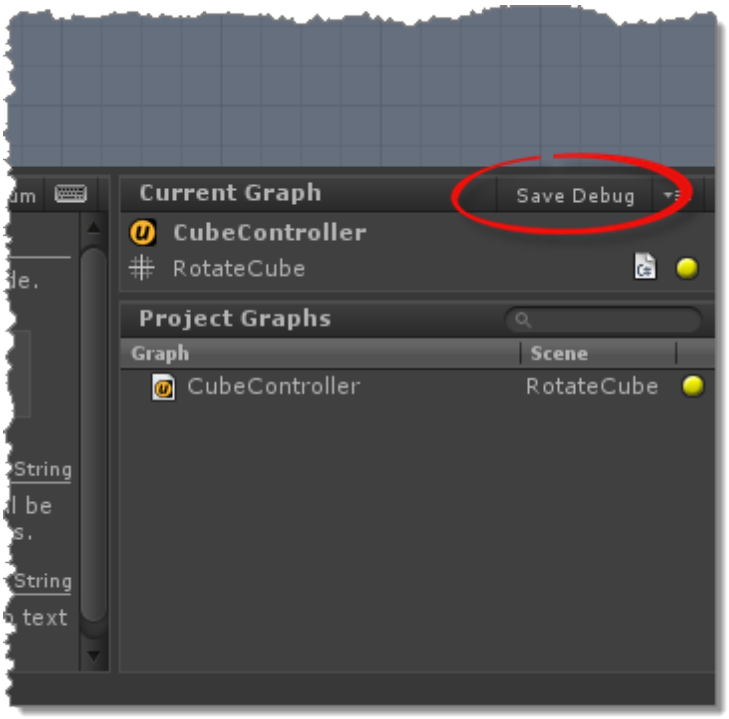
Step 14 - Now let's save the graph so we don't loose our work in case a meteor hits or something! You can do this in one of three ways:

A. Use the hot-key *Alt+S* to save the open graph (*note that the standard Ctrl+S won't work to save your graph because Unity has that reserved-- it will save your Unity scene instead*).

B. Select "Save" from uScript's File menu (*don't worry about all those other save choices for now!*):



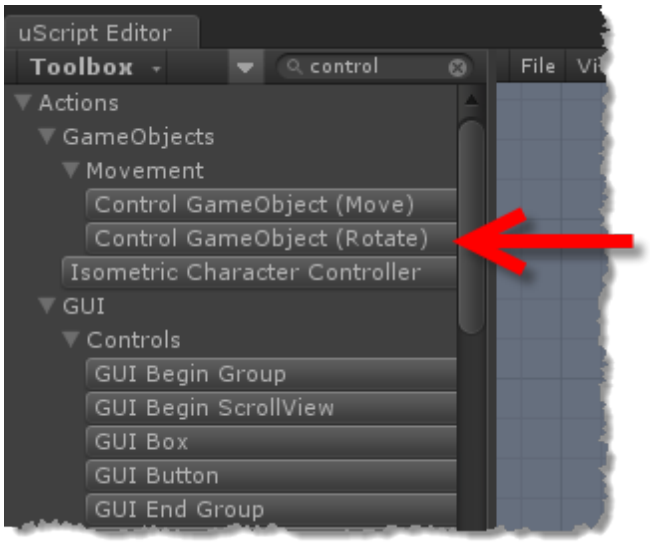
C: Use the Save button in the Graphs Panel in the bottom right of the uScript editor:



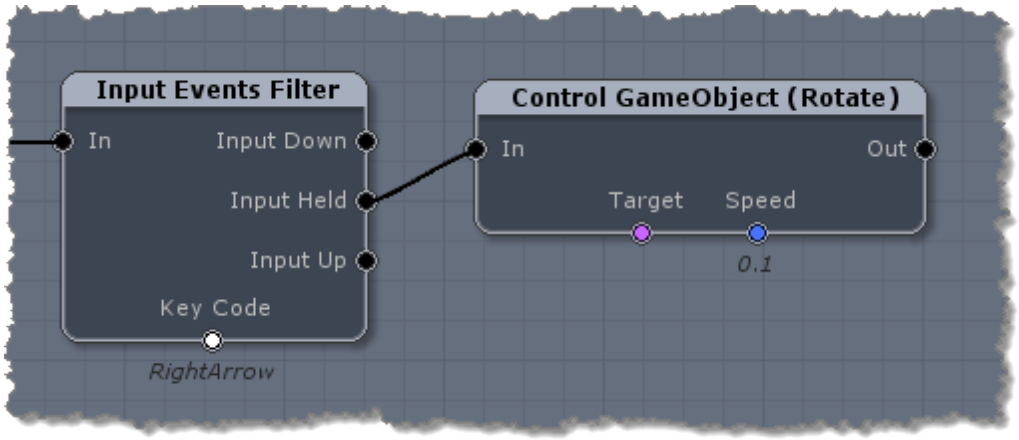
Remember that whenever you save, uScript will rebuild the graph's C# script file, triggering Unity to rebuild the scripts.
Once your graph has been saved, we are ready for the next part. Remember the last part of our logic sentence ("**Cube will rotate left or right**"? Let's work on that now...

Step 15 - Now we are ready to rotate the Cube to the right when your are holding down the Right Arrow key! uScript actually has several action nodes that allow you to rotate GameObjects in different way, but the one we want to use in this case it the *Control GameObject (Rotate)* node. This is because this node was designed to work when it is receiving a constant signal every tick of the game-- from like when you are holding down a key on the keyboard. Other rotate and movement nodes in uScript are setup to work when just receiving a single signal to the node and then handling the rotation/movement over time internally.

1. Lets place a *Control GameObject (Rotate)* node onto the Canvas. Find it in the Toolbox by either navigating to *Actions/GameObjects/Movement* section, or by first typing "*control*" into the Toolbox's search filter to have it appear more quickly, and then click the node button:



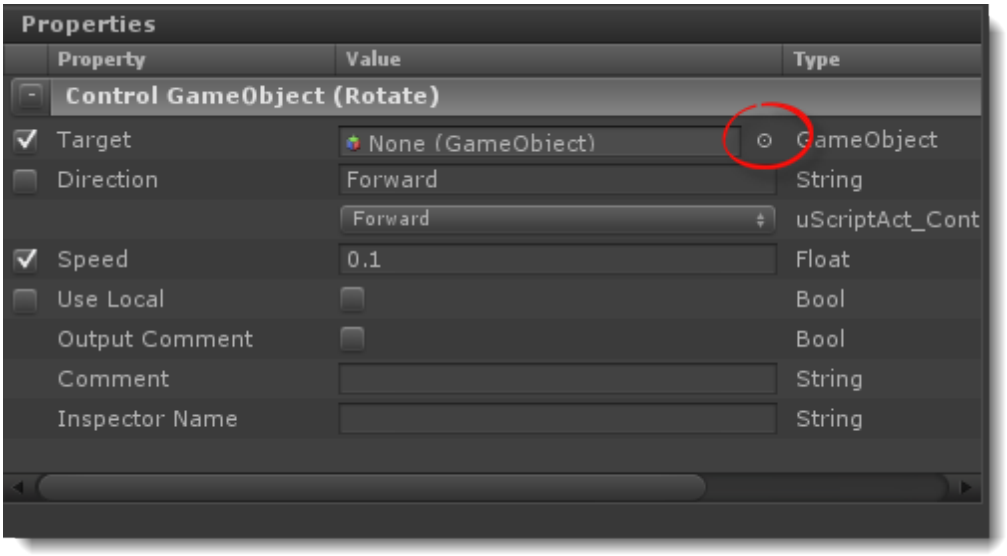
2. Once the node is on the graph, move it into place to the right of the *Input Events Filter* node. Then create a Connection Line between the *Input Events Filter* node's "*Input Held*" socket and the *Control GameObject (Rotate)* node's "*In*" socket:



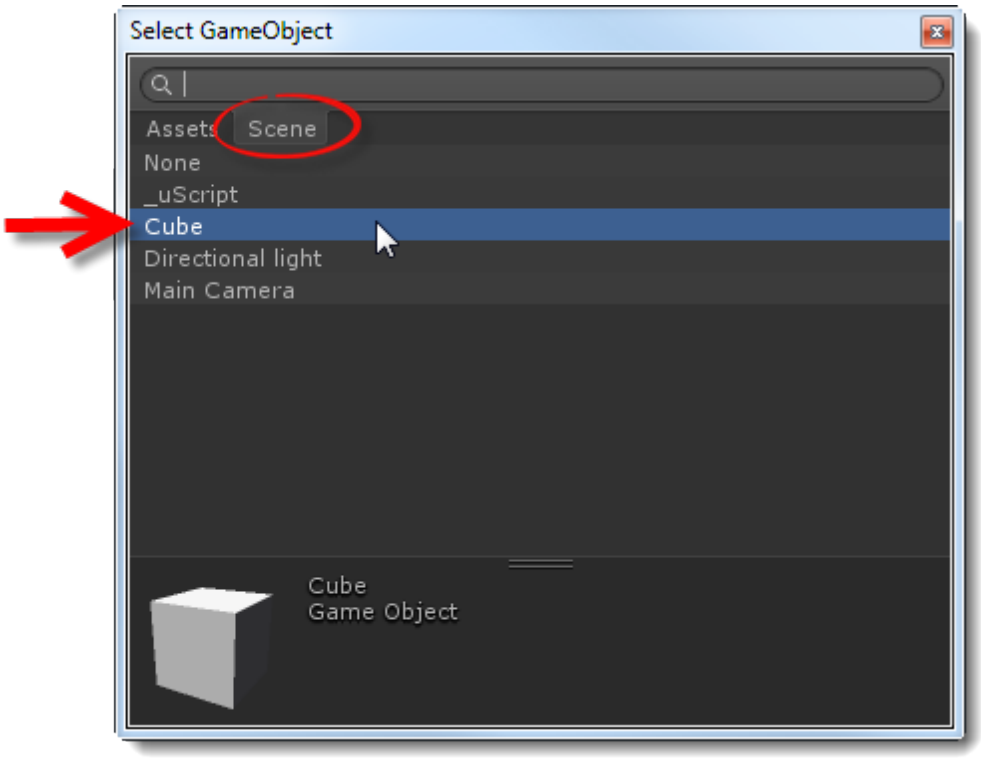
By hooking this new node up to the "Input Held" socket of the *Input Events Filter* node, we are saying we want to send a signal to the *Control GameObject (Rotate)* node as long as the *RightArrow* key is being pressed by the player.

Step 16 - Now we need to setup the properties of the *Control GameObject (Rotate)* node so that it knows how we want it to rotate the Cube. Select the node and change the following properties:

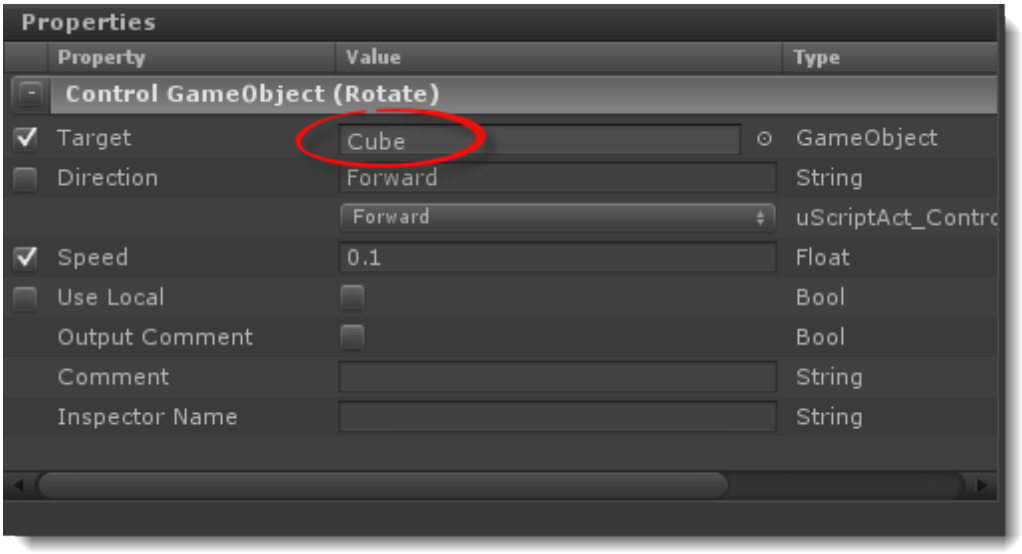
1. First we need to set the Target property to point to the GameObject in the scene we want the node to rotate-- in this case the Cube. Select the little browse button to the right of the Target field to bring up the asset browser:



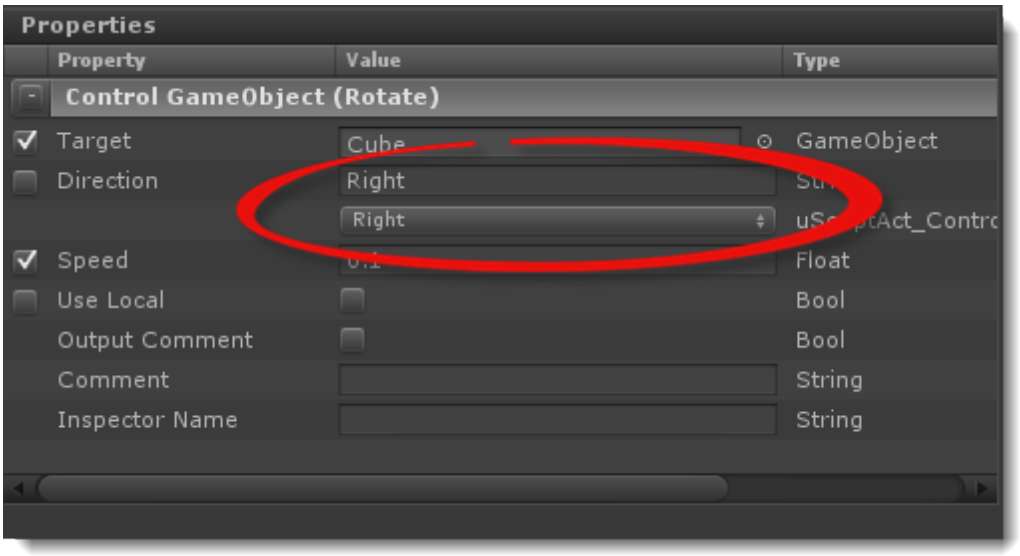
Then select the "*Scene*" tab in the browser window and double-click on the Cube GameObject in the list:



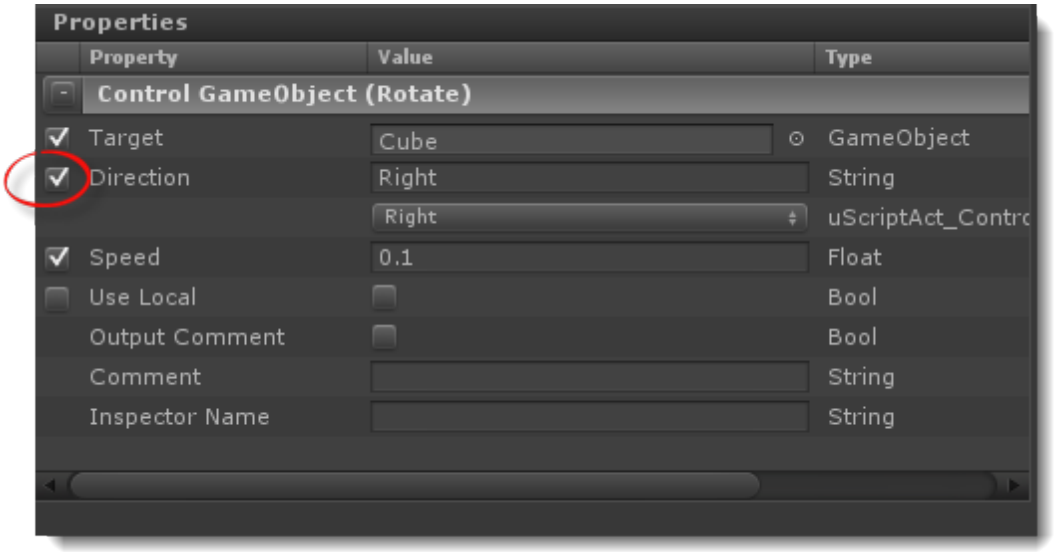
You should now see the Cube GameObject in the Target property's field:



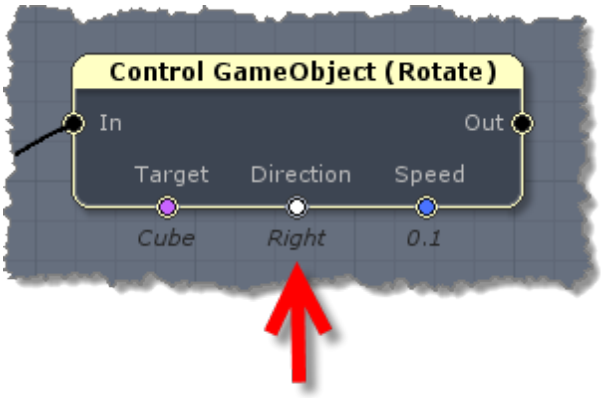
2. Now we need to tell the node which direction it should rotate the Cube. Since this node get's a signal when the *RightArrow* key is pressed, lets assign the Direction property to "*Right*". Do this by selecting "*Right*" from the pulldown list next to the property:



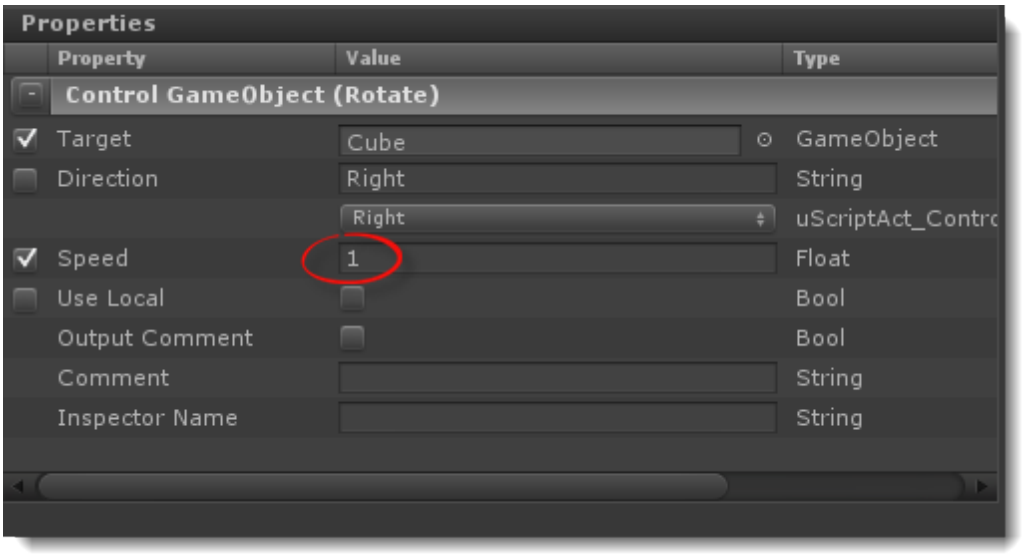
Also, let's check the box to the left of the Direction property. This will expose this socket on the node, allowing us to see what direction it is set to without needed to select the node and look in the Properties Panel (*to learn more about what can be done with the Properties Panel, see the "Properties Panel" section of the "Editor Interface" topic*):



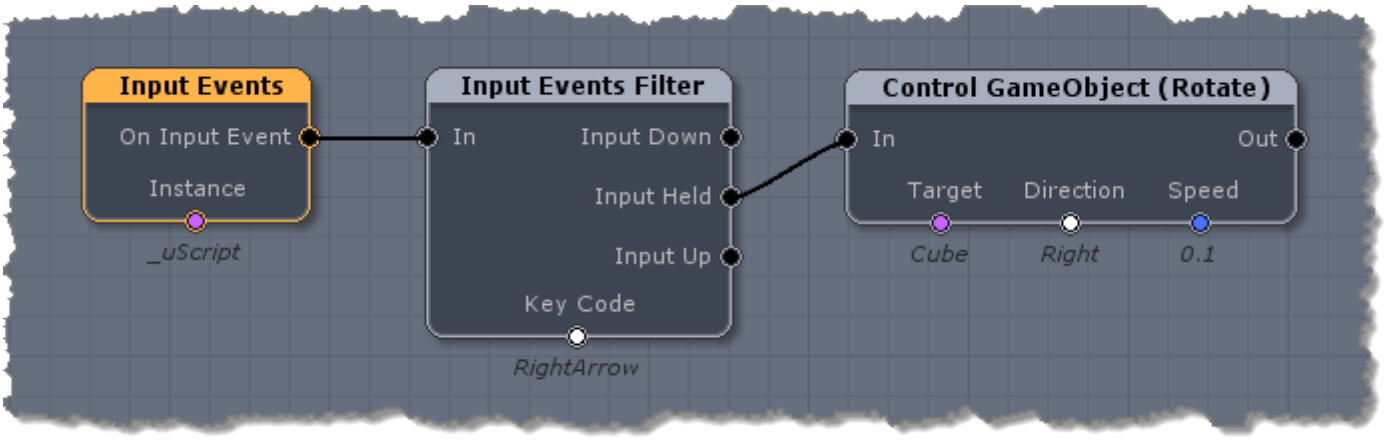
Once you do this, you will now see that socket appear on the node (*this is also known as "exposing" the socket to the graph*):



3. Lastly, we should set the Speed property a little higher so that the Cube will turn quickly. Let's set it from "*0.1*" to "*1*" (*10 times as fast*):



Step 17 - You should now have a graph that looks like this:



1. Let's **save our graph again** (using the *Alt+S* hot-key or one of the other save methods). Note that it is *very important* to save your graph after making any changes or additions to it because this is when uScript will also build the C# script file Unity will use when you run the game. If you don't save your graph after making changes, those changes will not be used when the game is run!

2. Now that our graph is saved and the scripts updated, let's close the uScript editor and run the game by pressing the play button in Unity:

3. Once the game has started, try holding down the Right Arrow key on your keyboard. You should see the Cube turning right. Stop pressing the key and the Cube will stop rotating. Press the key again and it will begin turning to the right more. Success!

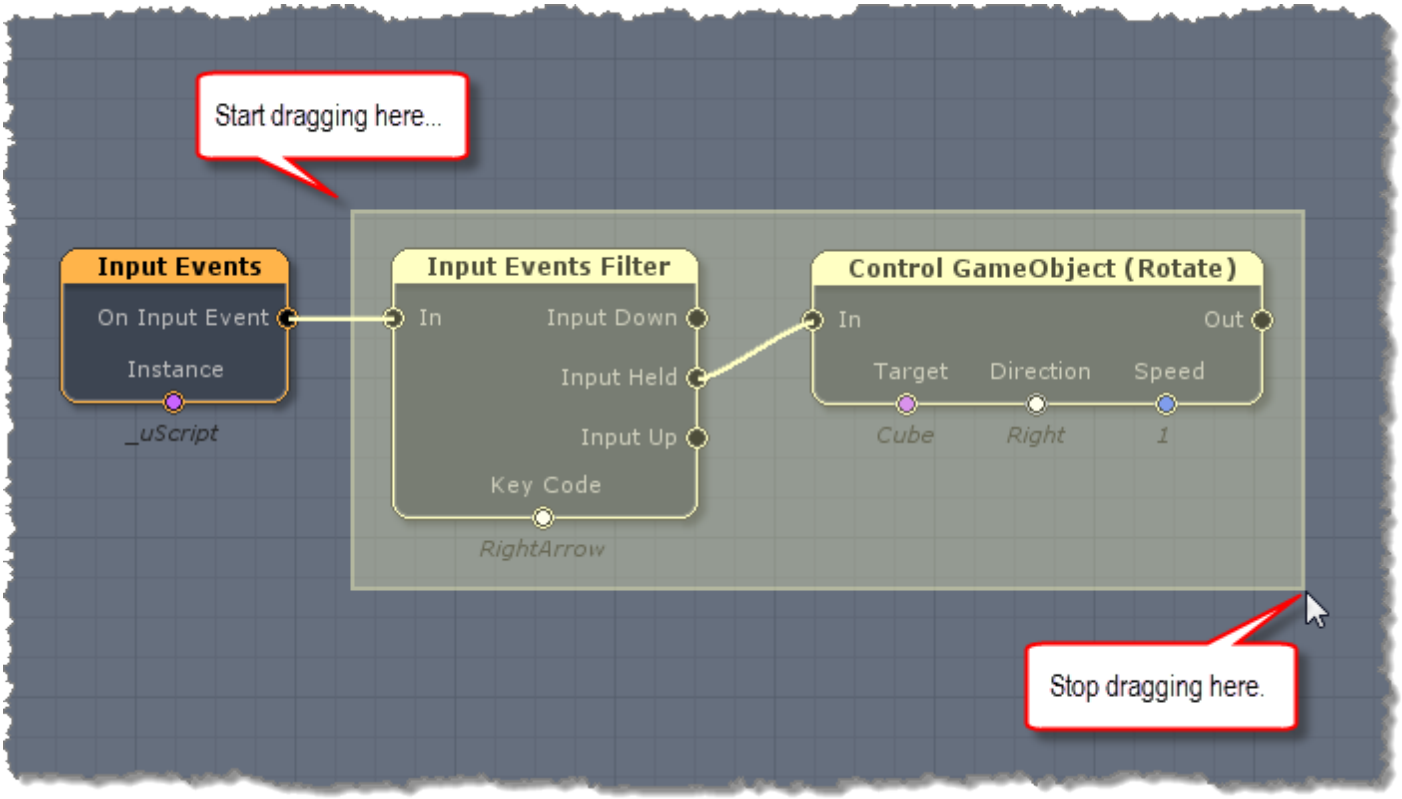
Now try pressing the Left Arrow key. Nothing happens as expected-- because we haven't set that part up yet! Let's do that now!

4. Press the Play button in Unity again to stop the game from playing. Then re-open the uScript editor window. Your last open graph (*CubeController*) should be already loaded and ready to add to!

Step 18 - Now we need to setup the logic for when you hit the Left Arrow key. Luckily the nodes and logic to that is the same as the Right Arrow key setup with a few minor property changes, we can take a shortcut!

1. Let start by "Box Selecting" the nodes we want to copy. To do this all you need to do is press the left mouse button on the Canvas to the top left of where you want to start your selection box and continue to hold down the left mouse button while dragging your mouse pointer to where you want your box selection to end.

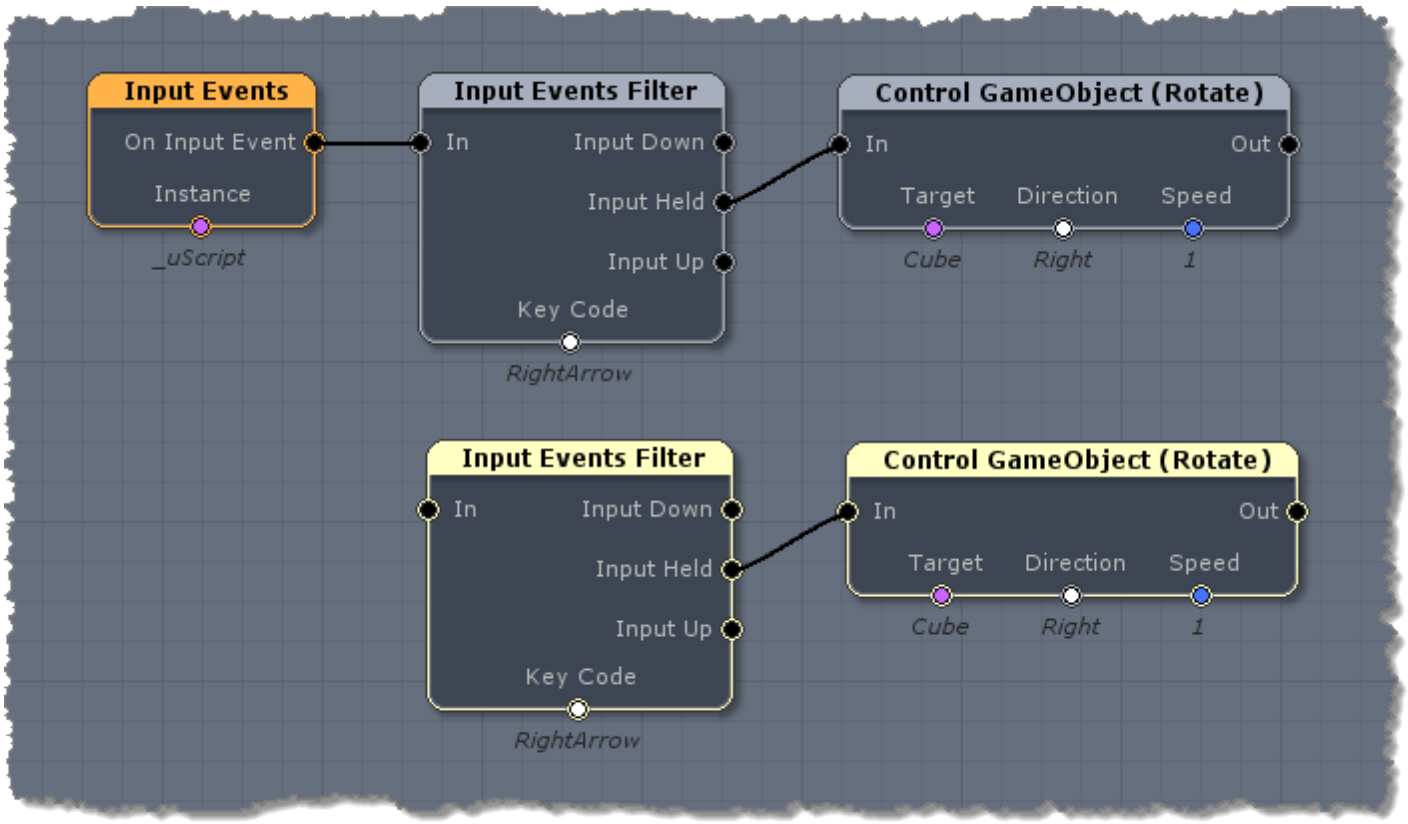
While doing this, touching the selection box will become selected (highlighted in yellow):



2. Now you can copy and paste the selected nodes. You can do this by either using the standard *Ctrl+C* (for Copy) and *Ctrl+V* (for Paste) hot-keys (*Cmd+C* and *Cmd+V* on Mac), or by right-clicking on the selected node and using "Copy" from the context menu that appears. If you use the context menu, you would then right-

click on a blank spot on the Canvas where you want your new node pasted and again right-click and choose "Paste".

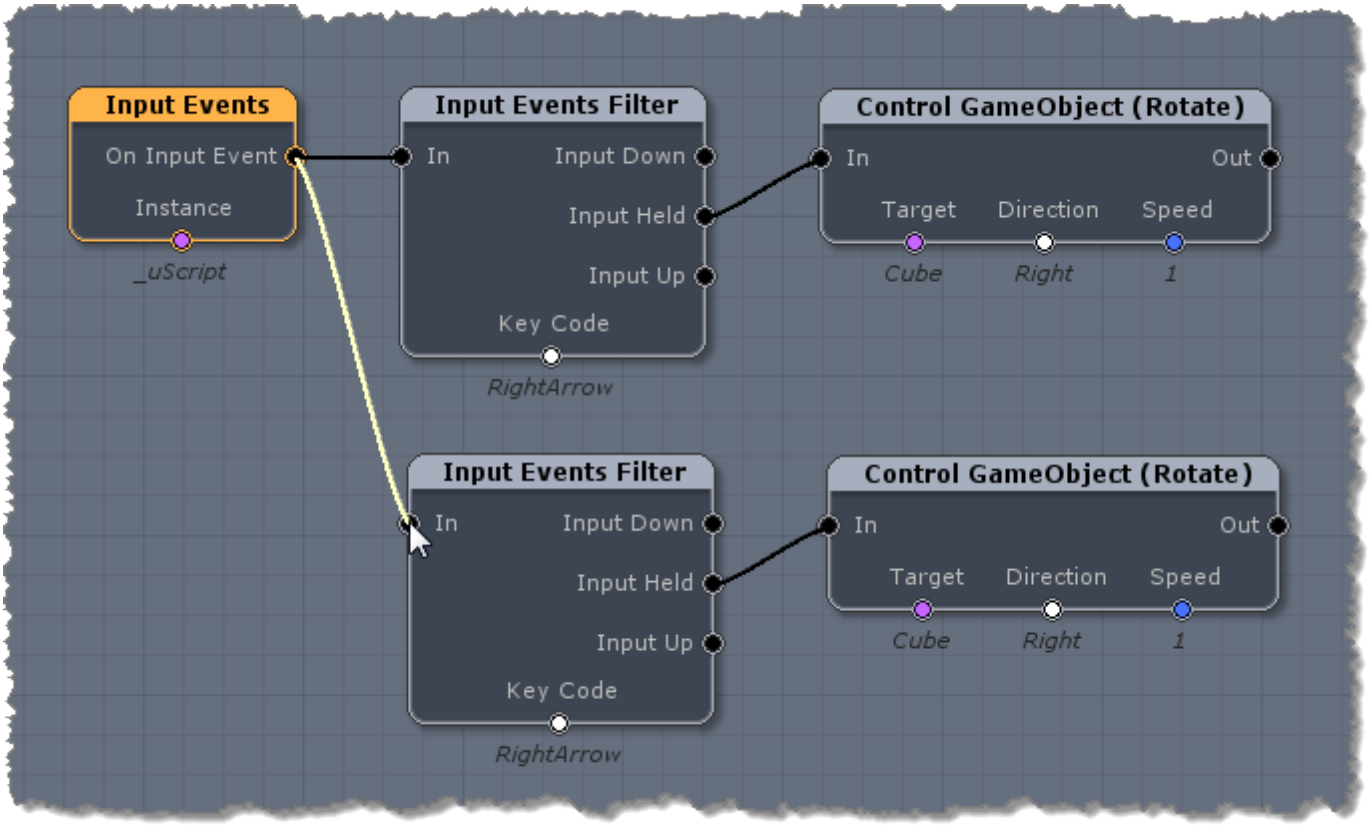
3. Move the newly pasted node underneath the existing one so it looks something like this:



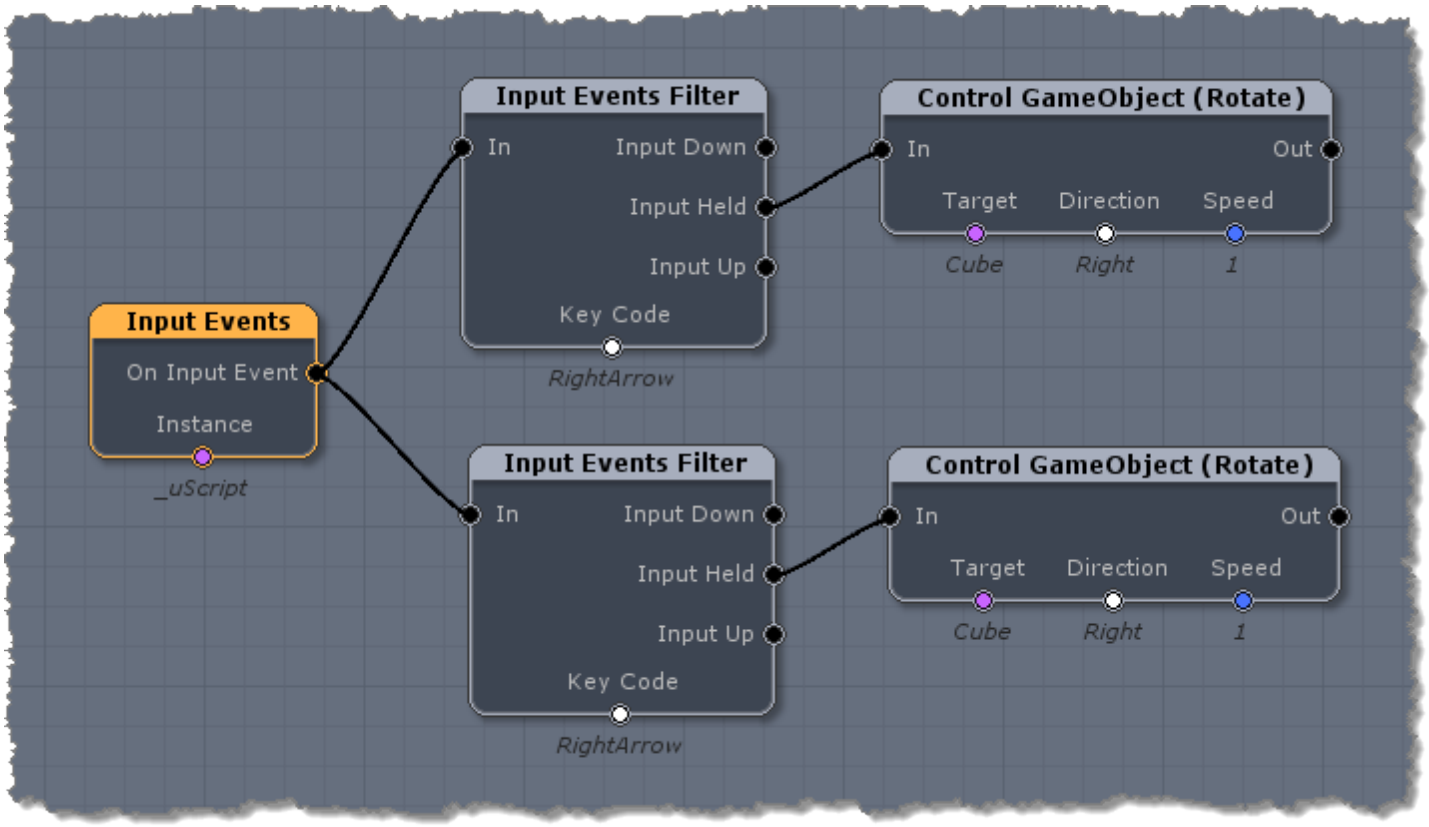
Note! - *depending on the size of your uScript editor window, you may soon need to start scrolling around the Canvas as your graph gets bigger than can fit on screen at once. To scroll the Canvas, you can either hold down the middle mouse button while moving the mouse or use the Alt+Left Mouse Button option while moving the mouse. To learn more about navigating around the Canvas, see "Canvas Navigation".*

4. The first thing we need to do now is to connect the *Input Events* action node to the new *Input Events Filter* node. To do this, let's

drag out another connection line from the *Input Events* event node's "*On Input Event*" socket to the "*In*" socket on the newly copied *Input Events Filter* node:



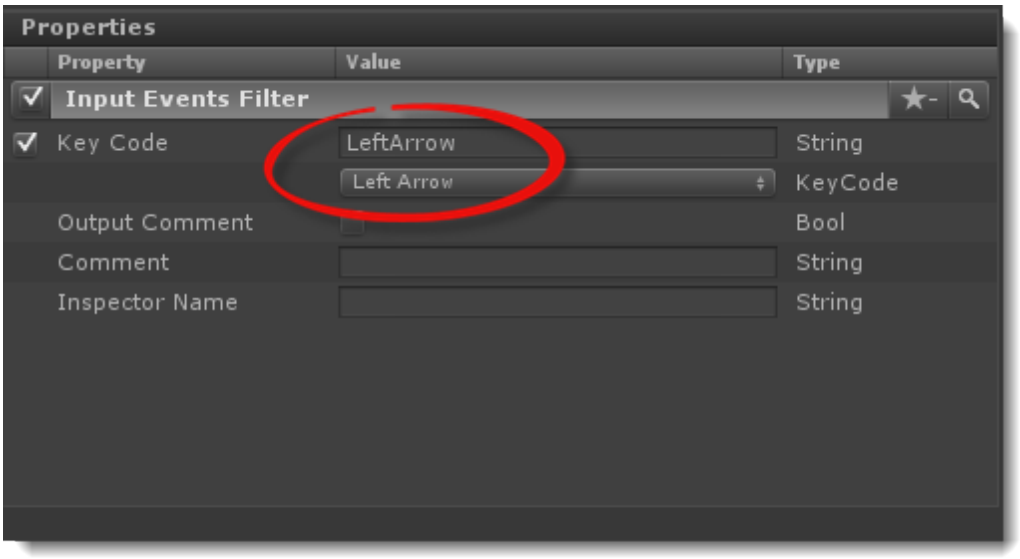
5. Now let's select the *Input Events* action node and move it down a bit on the Canvas so things look a bit more balanced in our graph (*note that the node positions don't at all effect how they work-- I just like a clean graph!*). Your graph should now look something like this:



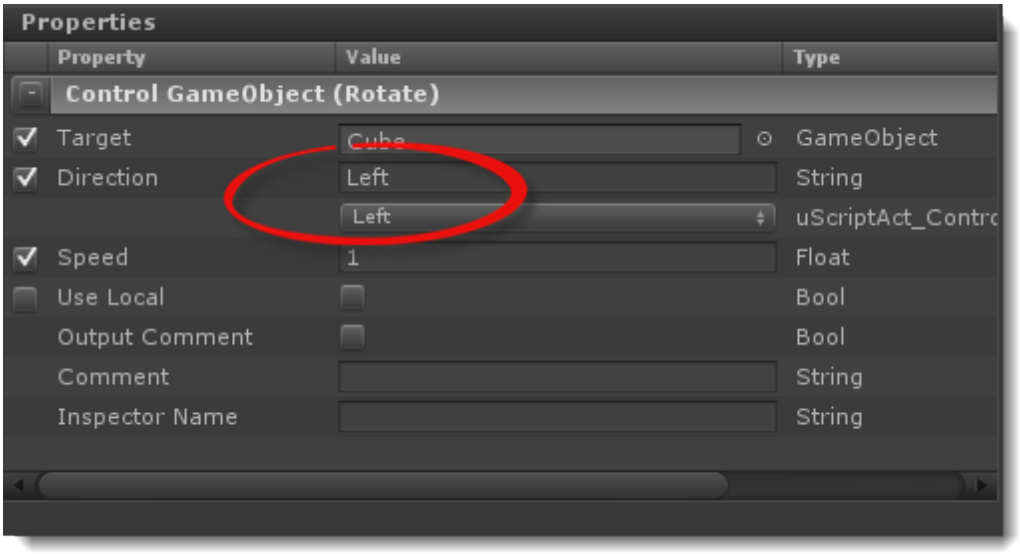
Step 19 - You now have two sets of nodes hooked up with the exact same properties. Let's change the necessary node properties to get the Left Arrow key working properly:

1. First we need to set the lower *Input Events Filter* node to instead look for the Left Arrow.

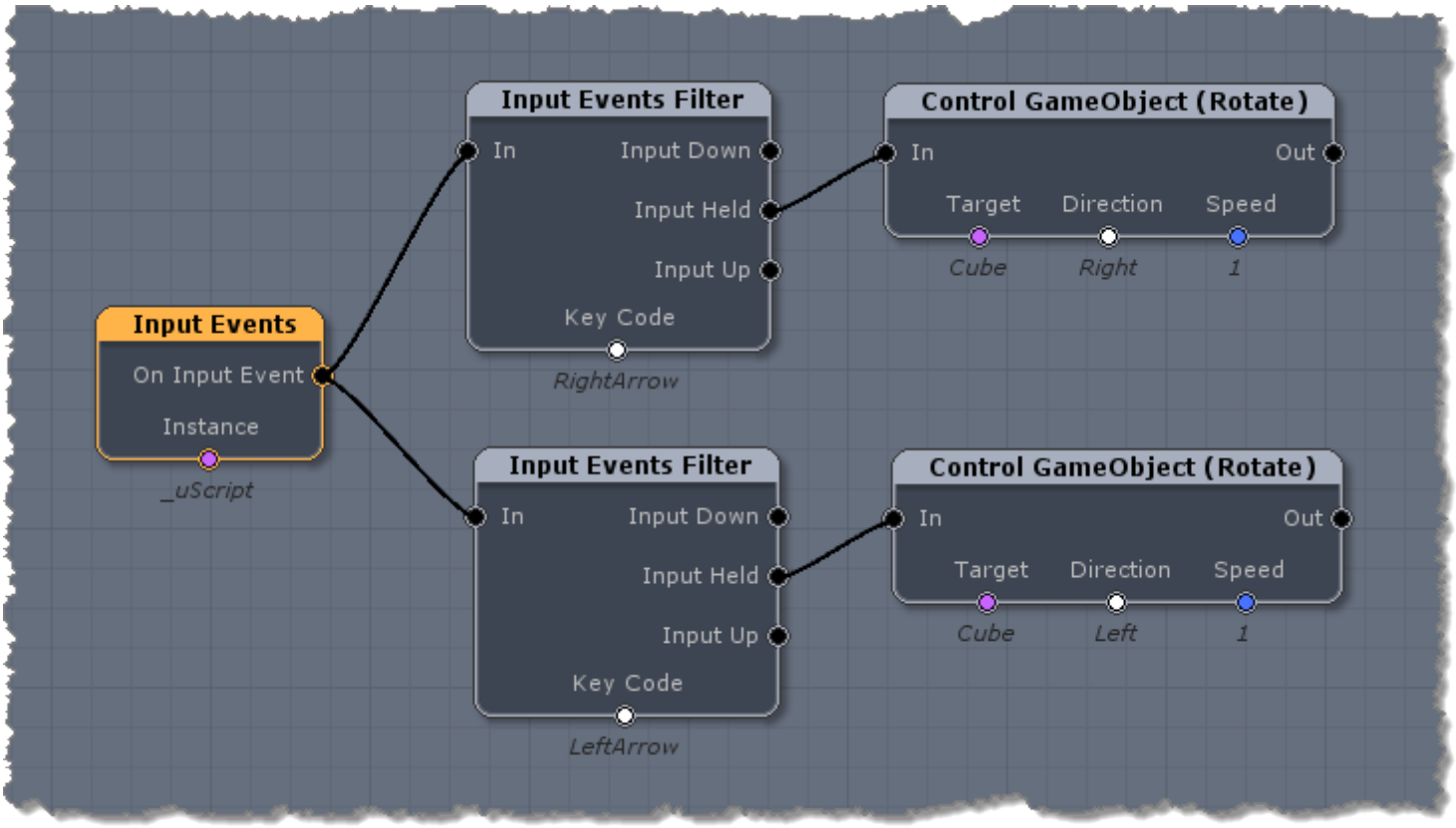
Select the new *Input Events Filter* node and change its *Key Code* property to the Left Arrow just like you did the original node for the Right Arrow in Step 13 (*remember that you could just type in "LeftArrow" (without a space or the quotes) into the String field to have it find the correct selection for you quickly*):



2. Now we need to change the lower Control GameObject (Rotate) node's Direction property to "Left". Select the node and change that property now:



Your final graph should now look like this:



Notice how this graph reads a lot like our original sentence - *"When the player presses the left or right arrow keys, the Cube will rotate left or right."*

3. Now save your graph and close the uScript editor because we are done!

Step 20 - We are now ready to test our final graph!

1. Once again press the Play button in Unity:

2. Go ahead and press the Left and Right Arrow keys to watch the Cube rotate left and right.

Tutorial completed!

Making A Prefab Graph

Unlike a regular "scene graph", which in many cases only works in the Unity scene that it was made for, a "prefab graph" is a graph that is designed to work with any GameObject it is assigned to-- regardless of the scene or how many GameObjects have that graph assigned. It is called a prefab graph because most of the time, that is what you will use these graphs on (Unity prefabs), but they can work on any GameObject really.

> ***Note! -*** *This tutorial assumes you have completed the initial tutorial (see "Your First Graph"), or are at least now familiar with both Unity and uScript editors and their basic interface and functionality (like how to create GameObjects and child GameObjects in Unity, and saving graphs and how to find and place nodes in uScript).*

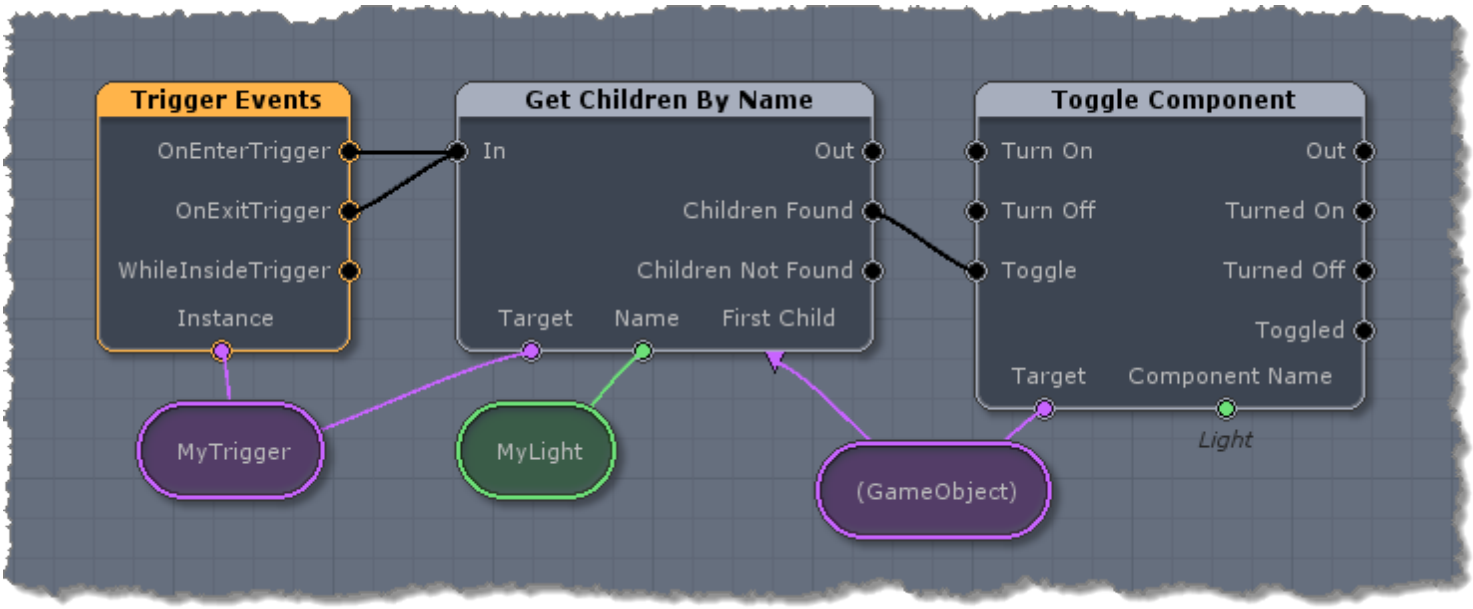
When To Use Them

A prefab graph isn't actually made any differently than a normal scene graph actually-- it is just about following a couple of rules and using a special variable uScript provides to allow you to make graphs that do not rely on anything in a Unity scene specifically.

You want to make a graph a "prefab graph" anytime you want to:

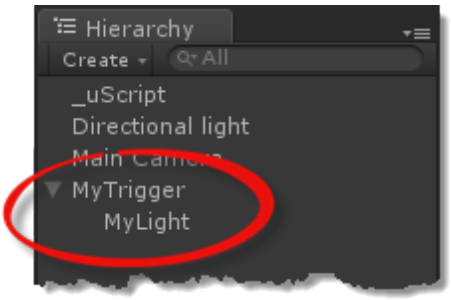
1. Make a graph that is going to be part of a Unity prefab.
2. Make a graph is easily reusable and assignable to any GameObject and in any and/or multiple scenes/levels/maps.

Take this scene graph example:

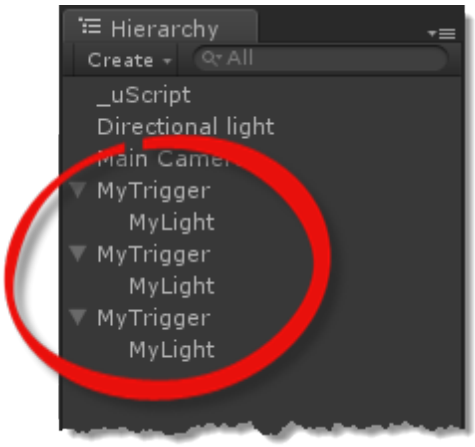


It is assigned to the Master GameObject in the scene (*_uScript*) and it is looking for a specific trigger GameObject called "*MyTrigger*". When *MyTrigger* is entered by the player (well, by anything in this setup) it will look for a child GameObject of the Trigger called "*MyLight*" and will toggle the light's "Light" component on and off (turning the light on and off).

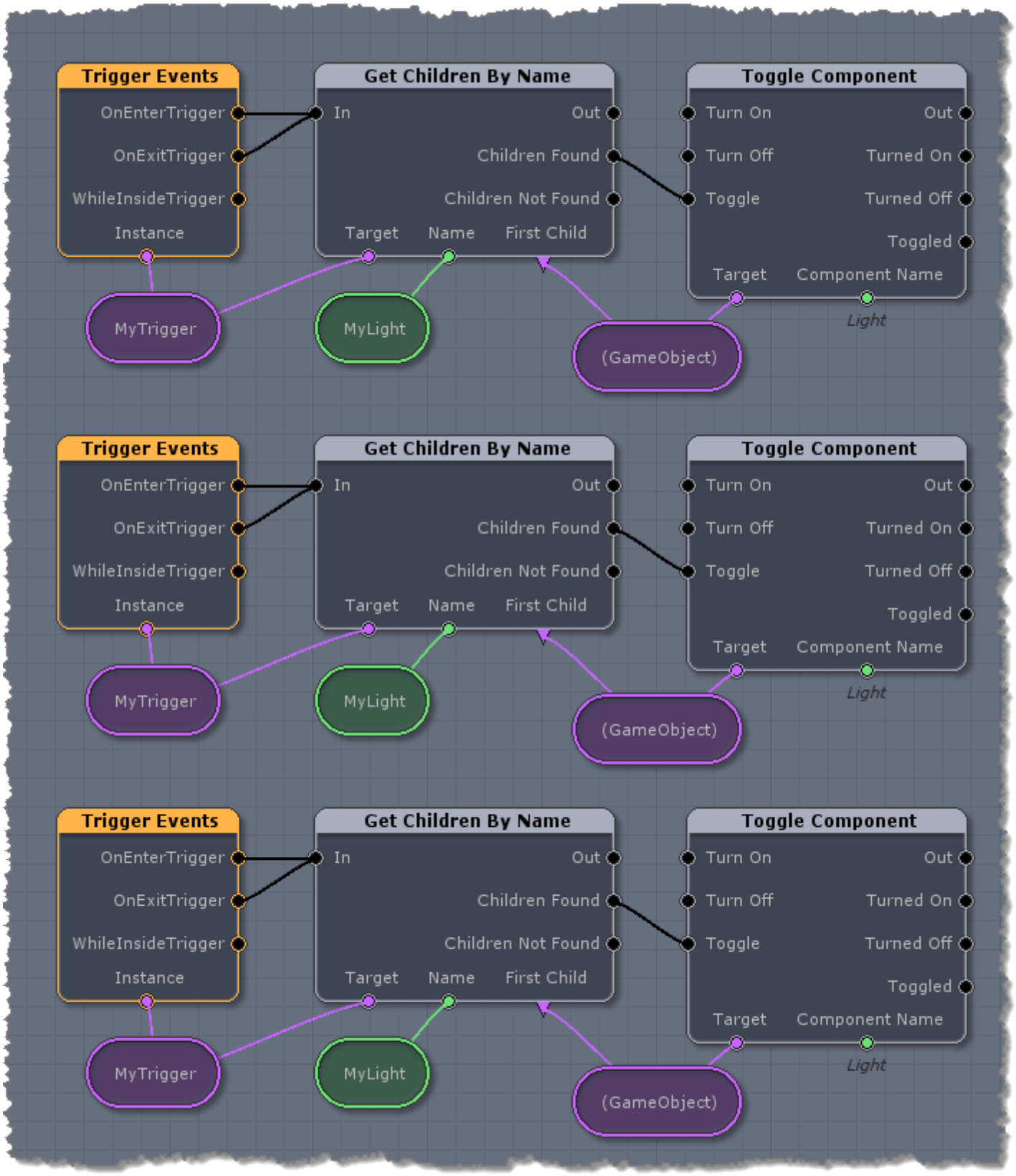
Here is what the GameObject setup looks like in Unity's Hierarchy tab for reference:



Simple enough and this will work great. What happens though if you want to have these *MyTrigger* GameObjects setup all over the place-- or use them in all your game's levels/maps? The first problem that can arise is just wanting more than one of these in your level: Let's say we want three now:

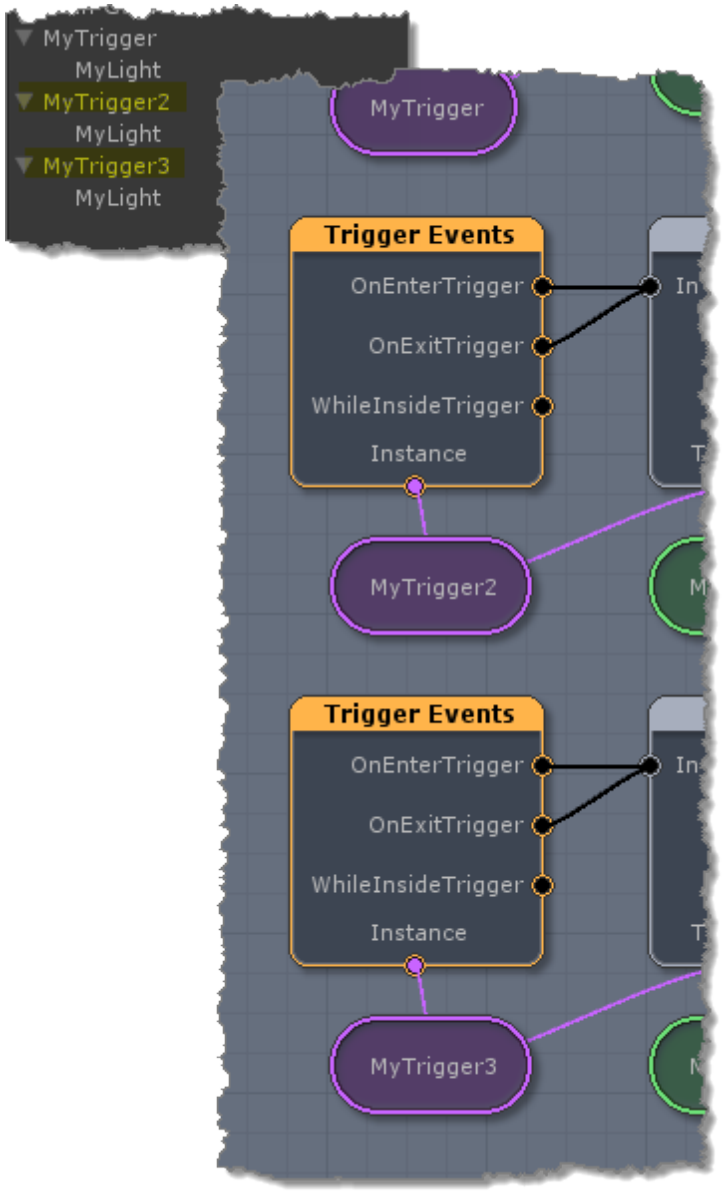


Well now you might have a graph that looks like this:


Trigger Events
OnEnterTrigger
OnExitTrigger
WhileInsideTrigger
Instance
Get Children By Name
In
Out
Children Found
Children Not Found
Target
Name
First Child
Toggle Component
Turn On
Turn Off
Toggle
Out
Turned On
Turned Off
Toggled
Target
Component Name
Light
MyTrigger
MyLight
(GameObject)
Trigger Events
OnEnterTrigger
OnExitTrigger
WhileInsideTrigger
Instance
Get Children By Name
In
Out
Children Found
Children Not Found
Target
Name
First Child
Toggle Component
Turn On
Turn Off
Toggle
Out
Turned On
Turned Off
Toggled
Target
Component Name
Light
MyTrigger
MyLight
(GameObject)
Trigger Events
OnEnterTrigger
OnExitTrigger
WhileInsideTrigger
Instance
Get Children By Name
In
Out
Children Found
Children Not Found
Target
Name
First Child
Toggle Component
Turn On
Turn Off
Toggle
Out
Turned On
Turned Off
Toggled
Target
Component Name
Light
MyTrigger
MyLight
(GameObject)

The bad news is that this won't even work because you know have three GameObjects called *MyTrigger*! Unity and uScript now won't know which *MyTrigger* GameObject goes as the Instance for which *Trigger Events* event node because they are all named the same (the same holds true when trying to find the child *MyLight* GameObject on the *Get Children By Name* node)!

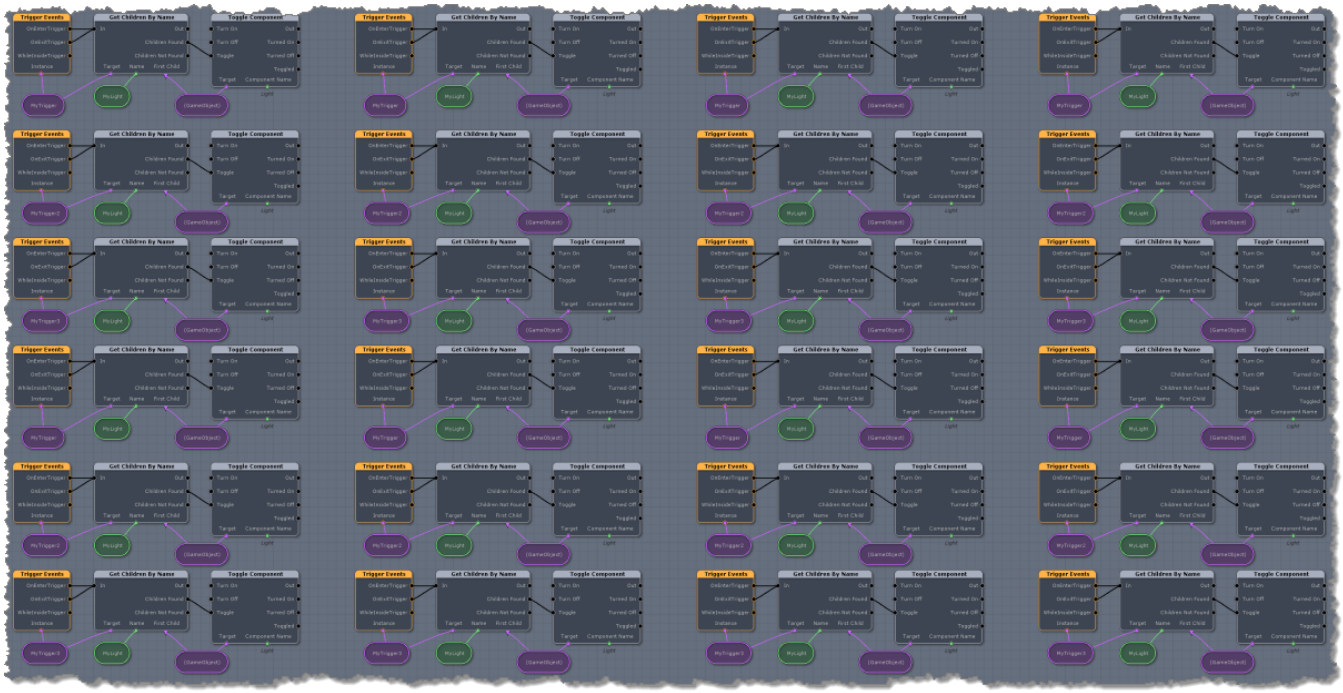
In order to fix this we would need to manually rename two of them and then update the graph to point to them:



Not only is that a pain to do-- but what if you want 50, 100, or 5000 instances of that trigger and light setup in your level?!? What if after you do all that work there is a bug-- or you want to change or add more logic to that setup (like also play a sound)-- you would

have to manually add the logic to every one manually (and we have games to make here!!!)? You can see that this would all get out of hand very quickly!

You **really** don't want this:



This is where prefab graphs come in. Let's take this example and show you a much better way to do something like this...

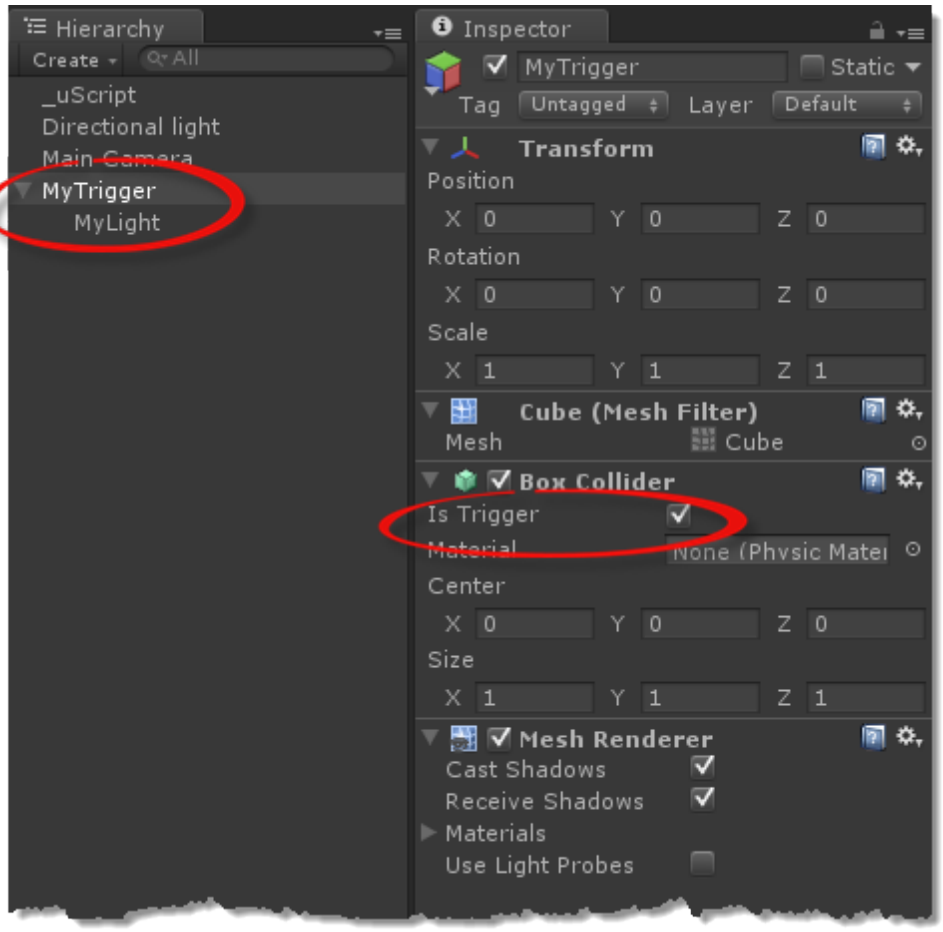
The Prefab Graph Way

To instead create a prefab graph that can easily control as many *MyTrigger* setups as you might have in your game, we just need to do a couple of things differently.

Scene Setup

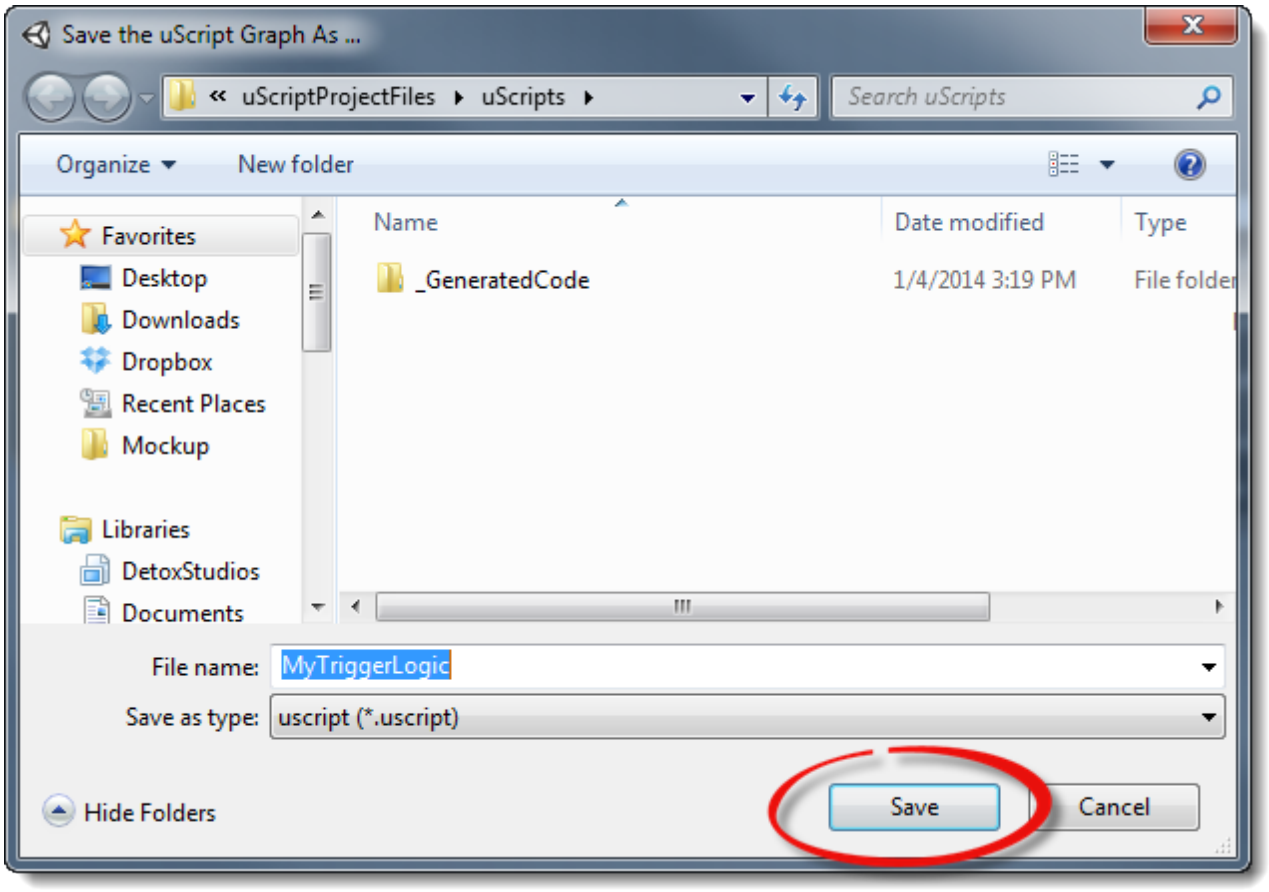
Before we get started, please create a new Unity project with uScript installed. You might also want to installed Unity's Character Controller if you plan to easily run around and test your work. If you do that I also suggest putting in a Plane for a floor to run on!

Then create a new GameObject called "*MyTrigger*" with a *Box Collider* component (I just placed a *Cube* GameObject), as well as a child *Point Light* GameObject called "*MyLight*". Be sure to set the "Is Trigger" flag on *MyTrigger* so it will work with the Trigger Events action node:

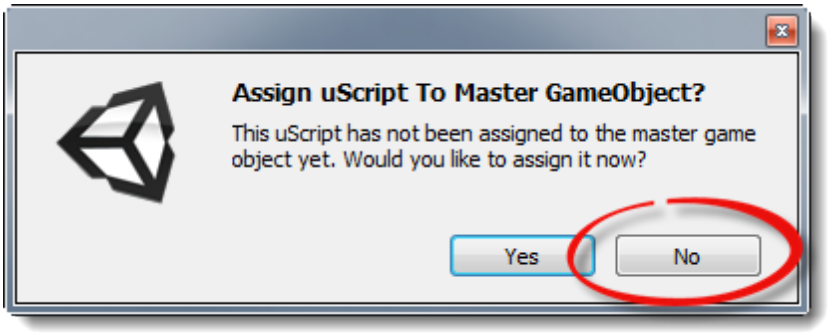


Step 1 - Let's by making a new graph in uScript and saving it:

1. When the Save dialog comes up, call this graph "*MyTriggerLogic*" and press the Save button:



2. When asked if you want to assign this graph to the scene's Master GameObject (*_uScript*), select the "No" button:



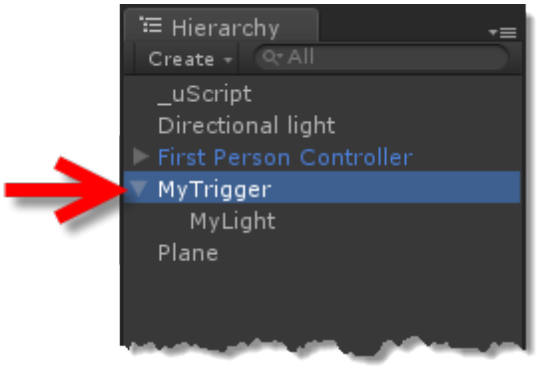
Selecting no here will tell uScript to not assign this graph to any GameObject. This means that before the game will use it we will need to manually assign it to

the GameObject in the scene we want to assign it to.

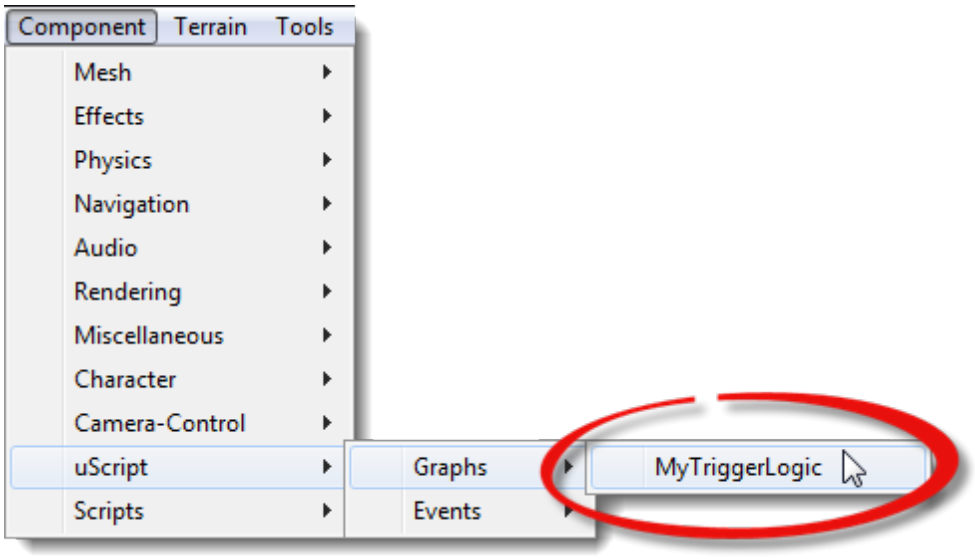
Step 2 - Now that we have created a new blank graph and saved it, we should go assign it to the GameObject we want it on manually.

> ***Note! -*** *assigning your graph to specific GameObjects and Prefabs manually instead of using the Master GameObject of the scene is the key to creating a Prefab Graph!*

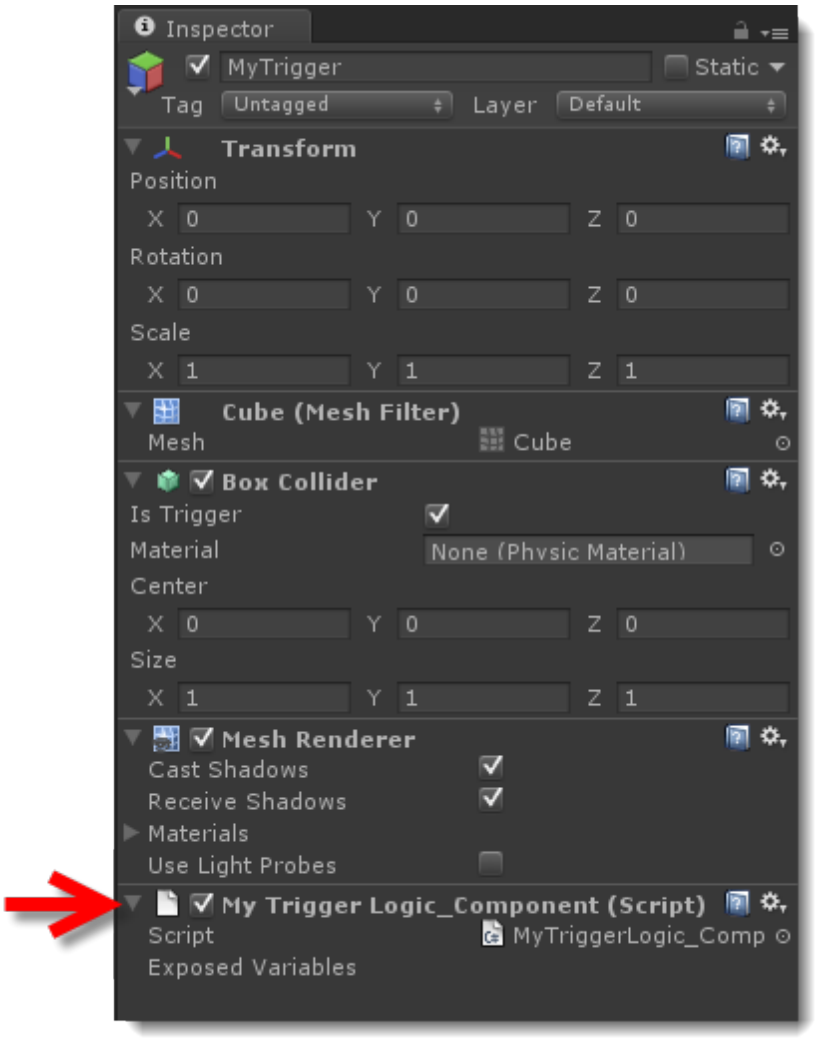
1. Select the GameObject you wish to assign the *MyTriggerLogic* graph to in Unity's Hierarchy tab (*MyTrigger*):



2. Then, with the *MyTrigger* selected, go to Unity's *Component* menu and choose the *MyTriggerLogic* graph from *Component/uScript/Graphs/*

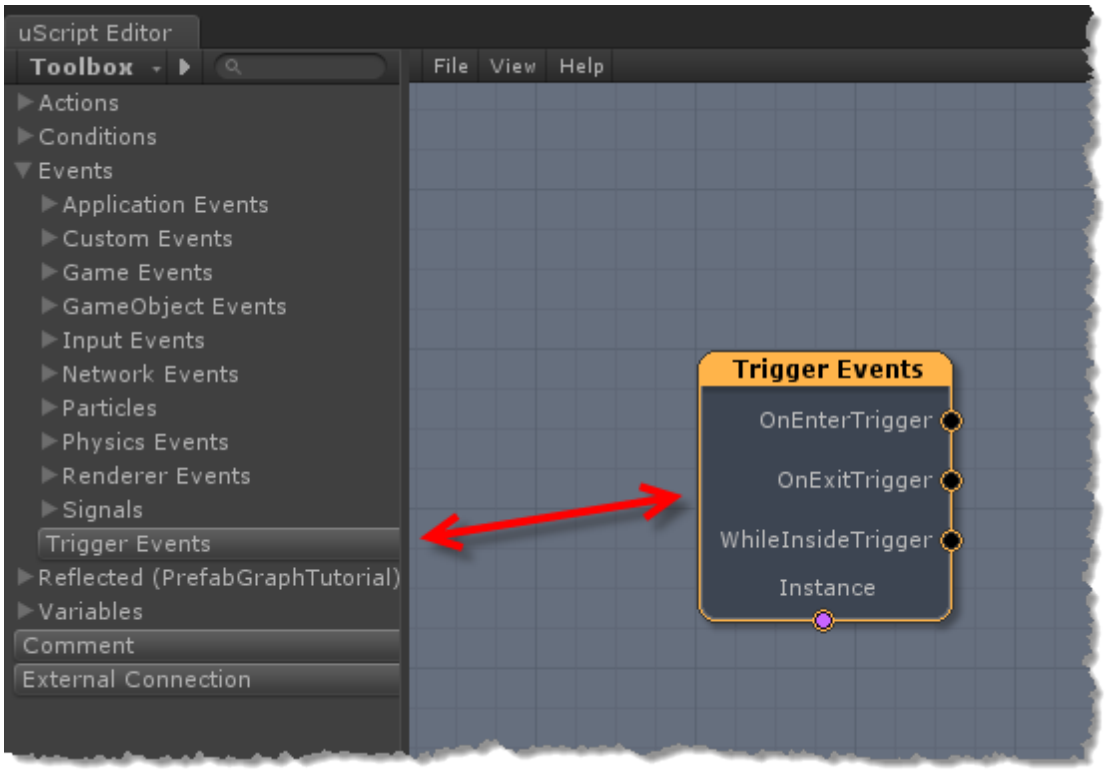


This will assign your graph to the selected *MyTrigger* GameObject in the scene:



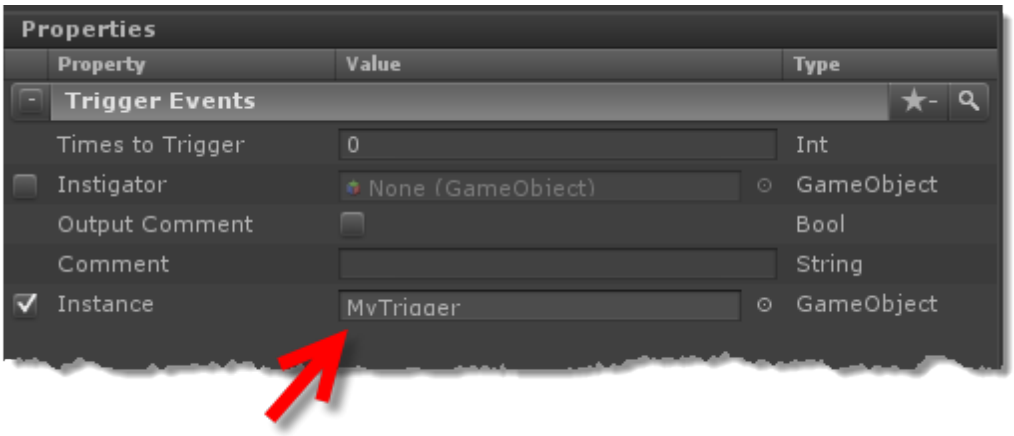
Step 3 - Now that our graph is assigned to the correct GameObject, we can go back into uScript and create the graph logic we will need.

1. First let's place the Trigger Events event node. Find it in the Toolbox and left-click it to place one on the Canvas (*remember that you can also use the Toolbox's search filter to help you find things quicker!*).This node will allow us to know when the trigger is hit:

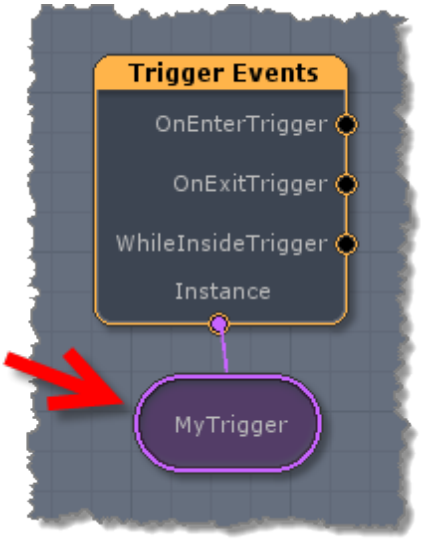


2. Now we need to setup the Trigger Events action node's *Instance* property to tell Unity which trigger we want to use for this node.

> Normally we would just go into the Properties Panel for this node and assign the GameObject we want to use there (***don't do this***)...



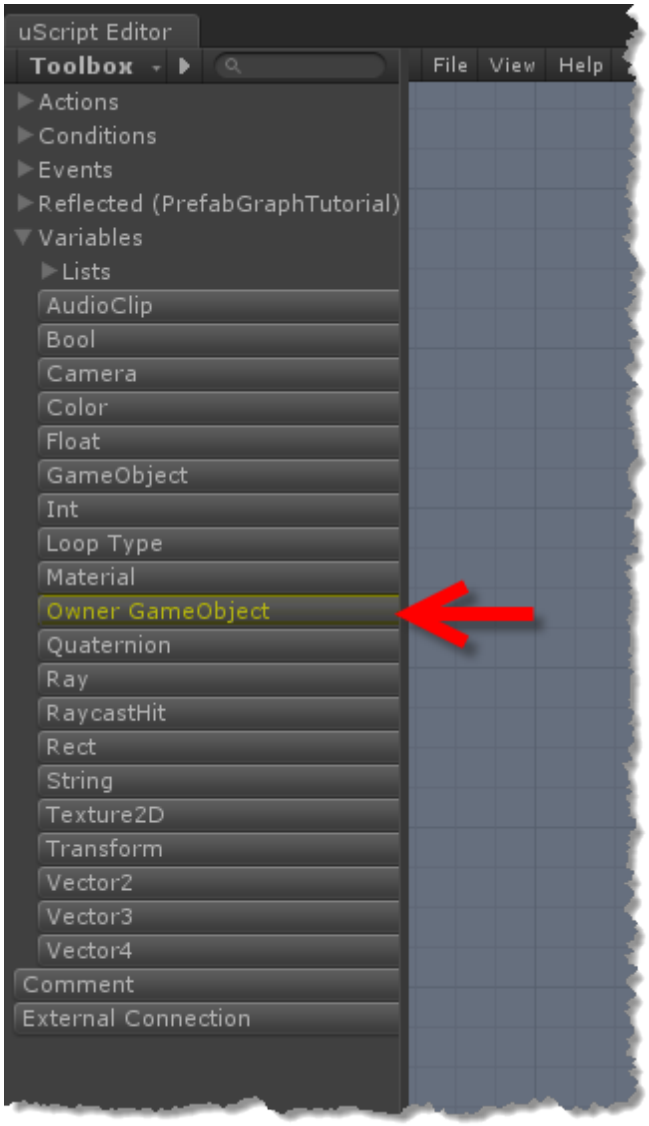
... or you might place a GameObject variable node and hook it up externally (***don't do this either***)...



As explained earlier in this tutorial however, assigning a specific GameObject is a bad way go if we ever want to use this logic more than once because then we have to manually rename copies of the *MyTrigger* GameObject manually.

Instead, what we want to do is to use a special variable node called the ***Owner GameObject***

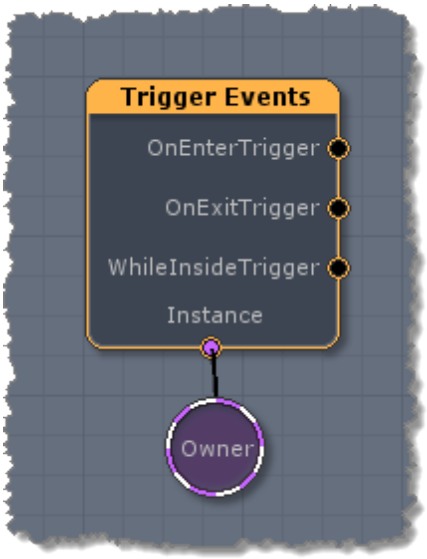
variable. This variable can be found in the Toolbox under the *Variables* section of the node tree:



Clicking that will place an *Owner GameObject* variable node onto the Canvas:

This variable is of the type "*GameObject*" and can be hooked up to any node socket that works for regular GameObject variables. The difference however is that you don't assign a GameObject to it. Instead, the variable will always use the GameObject that the graph is assigned to.

(***Do this!***) Go ahead an hook it up to the "*Instance*" socket of the *Trigger Events* event node:



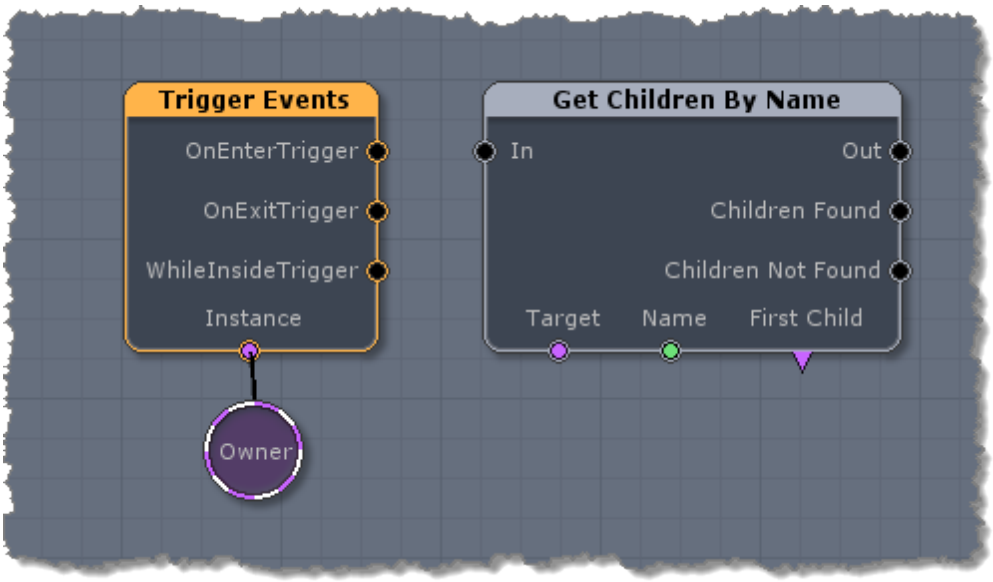
What you have done is told Unity -"*use whatever GameObject this graph is assigned to as the trigger for this node*".

This one variable type opens the door to making graphs that are not "hard coded" to work with a

specific GameObject from a scene and instead can be assigned to any GameObject.

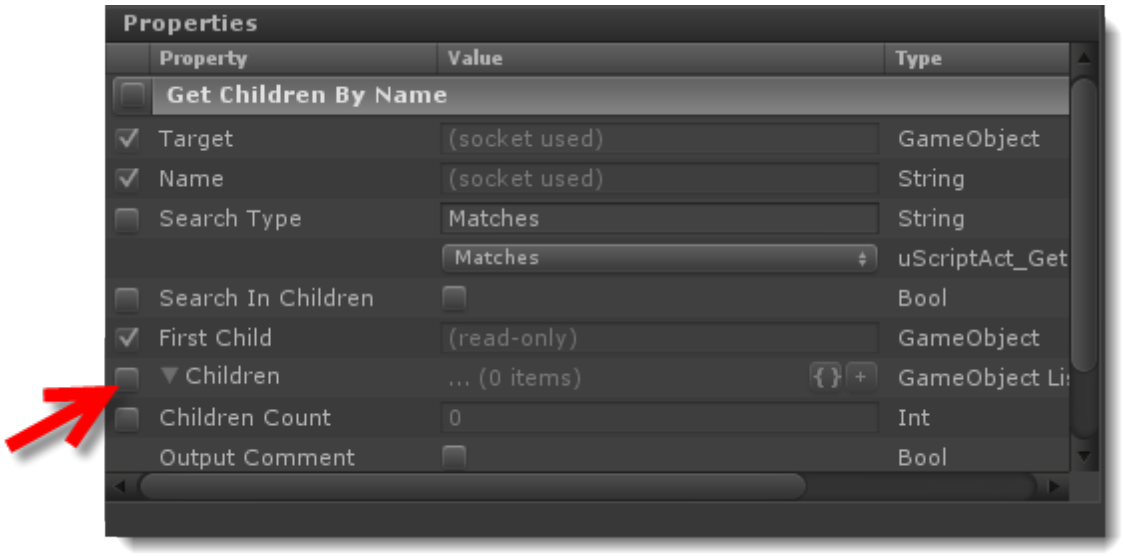
Step 4 - We are no ready to build the rest of this graph. Next we should hunt for the child GameObject named "MyLight" so we can toggle it on and off.

1. Place a *Get Children By Name* node to the right of the *Trigger Events* event node on the Canvas (you can find it under *Actions/GameObjects/* in the Toolbox):

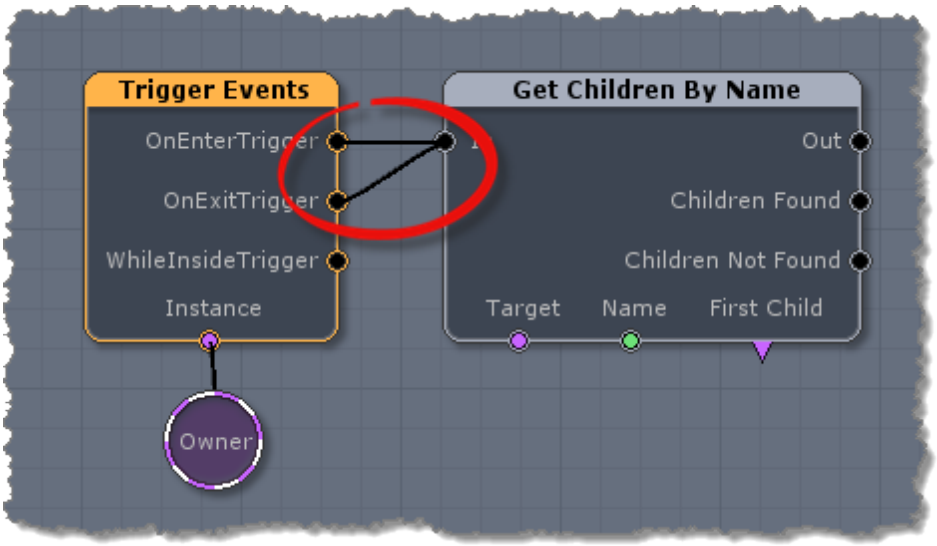


This node will allow us to easily find the *MyLight* child GameObject so we can access it to turn it's light on and off.

> ***Note! -*** *You may notice that my node seems to be missing the "Children" variable output socket on the bottom of the node. That is because we don't need it for our logic and I hid it by unchecking the socket in the Properties Panel for the node:*



2. Connect both the "*OnEnterTrigger*" and "*OnExitTrigger*" sockets of the Trigger Events action node to the "*In*" socket of the Get Children By Name node:

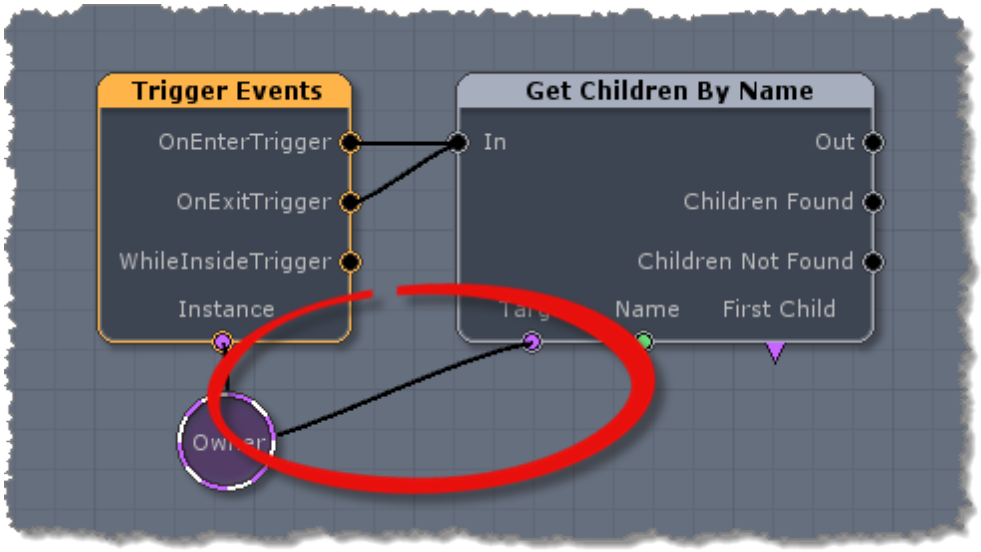


By connecting both sockets to it we will send a signal to the node (and the nodes hooked up after it) every

time something either enters of leaves our trigger volume.

Step 5 - Now we need to setup the *Get Children By Name* node's properties.

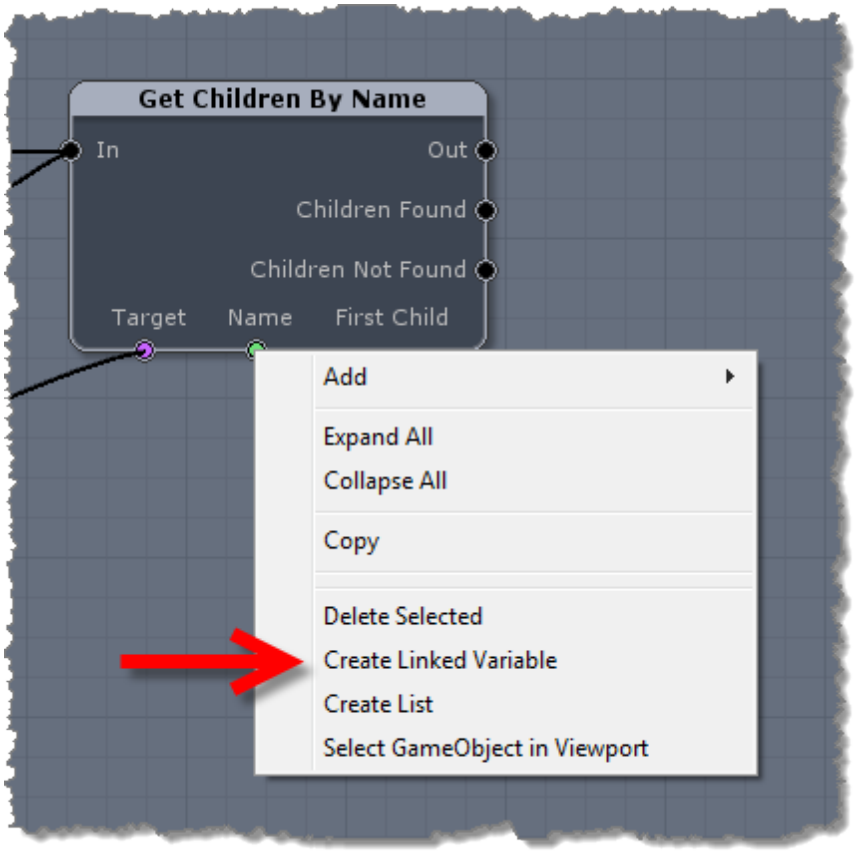
1. First we need to choose the "*Target*" for this node. The Target property tells the node which GameObject we want to look for children on. We know this should be the same GameObject that the trigger is on (the *MyTrigger* GameObject in this case), so we just need to assign it to the Owner GameObject as well (again, this tells Unity - "*use whatever GameObject this graph is on as the Trigger*"). Drag a connection line from the "*Target*" socket to the existing Owner GameObject variable that is already connected to the "*Instance*" socket of the Trigger Events event node:



2. Now we need to choose the name of the child GameObject to search for. In our case this will be a GameObject called "*MyLight*". You could go ahead and type this into the "*Name*" property in the Properties Panel, but let's go ahead and use a trick to

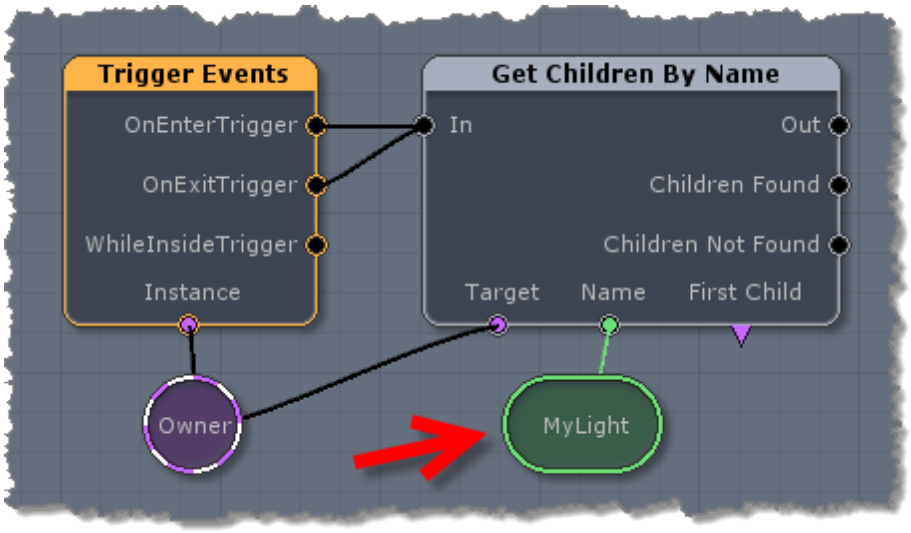
quickly make an external String variable that is hooked up to the socket.

Right-click on the "*Name*" socket of the *Get Children By Name* node. Select "*Create Linked Variable*" from the context menu that pops up:



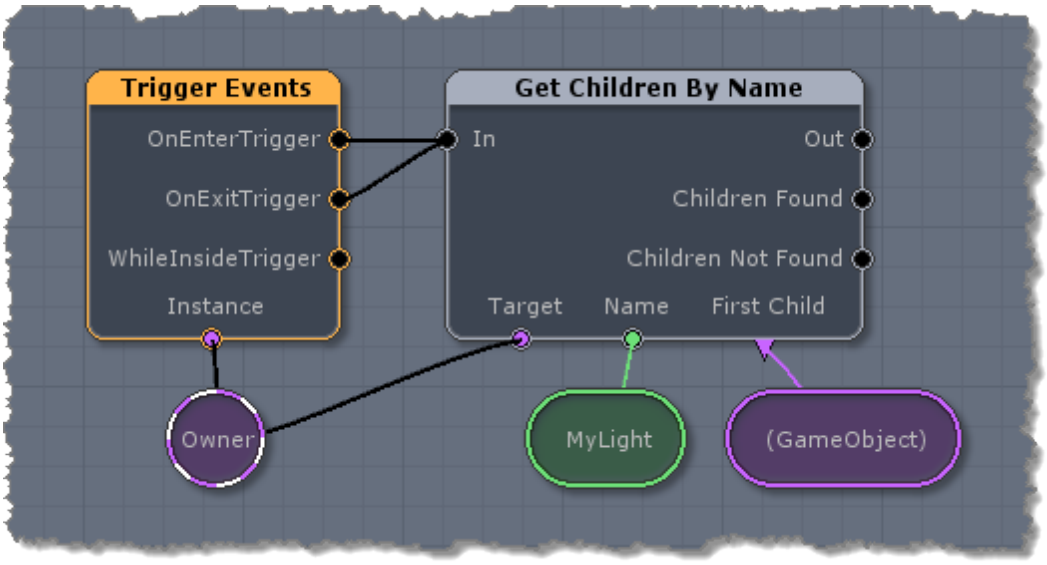
This will automatically place a variable of the correct type on the canvas for you and have it hooked up to that socket.

Go ahead and move the variable a bit lower and type "*MyLight*" (without the quotes) in as the value for the variable via the Properties Panel:



3. Lastly we need to add a GameObject variable to the "*First Child*" variable output socket on the *Get Children By Name* node. This is where the node will return the GameObject we need to use for the next bit of logic.

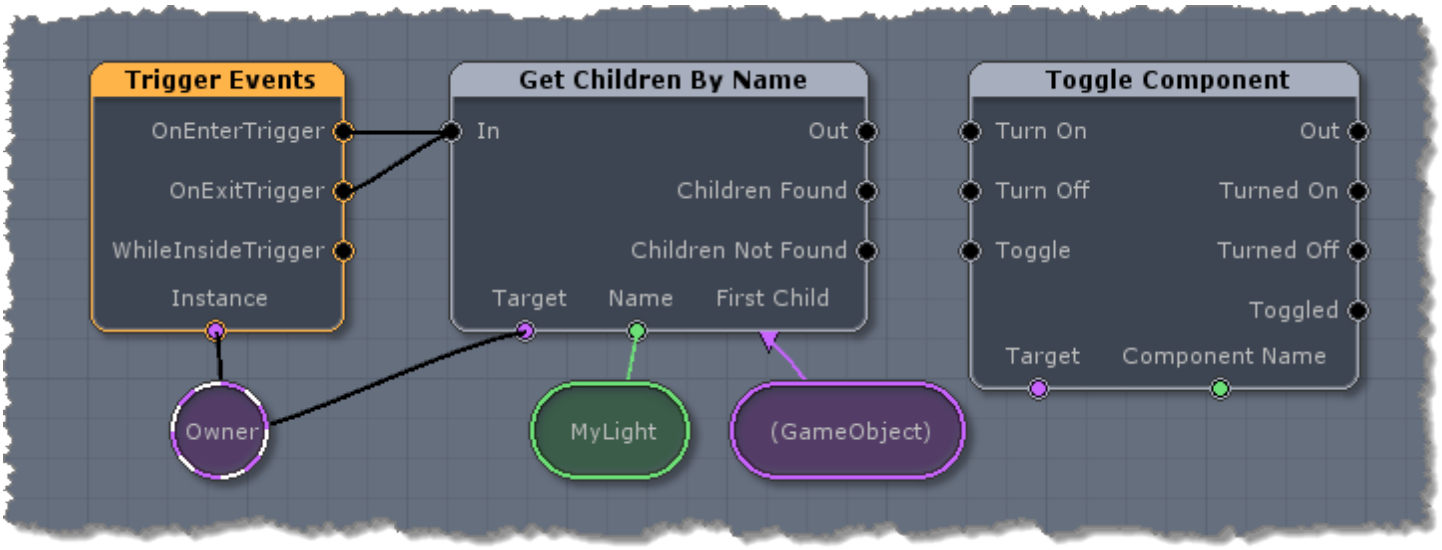
Let's use another shortcut to create the GameObject variable we need. Try holding down the "***G***" key on your keyboard and left-click on the Canvas below the node. You should now have a new GameObject variable there. Go ahead an hook it up to the node's "*First Child*" socket so that your graph looks like this:



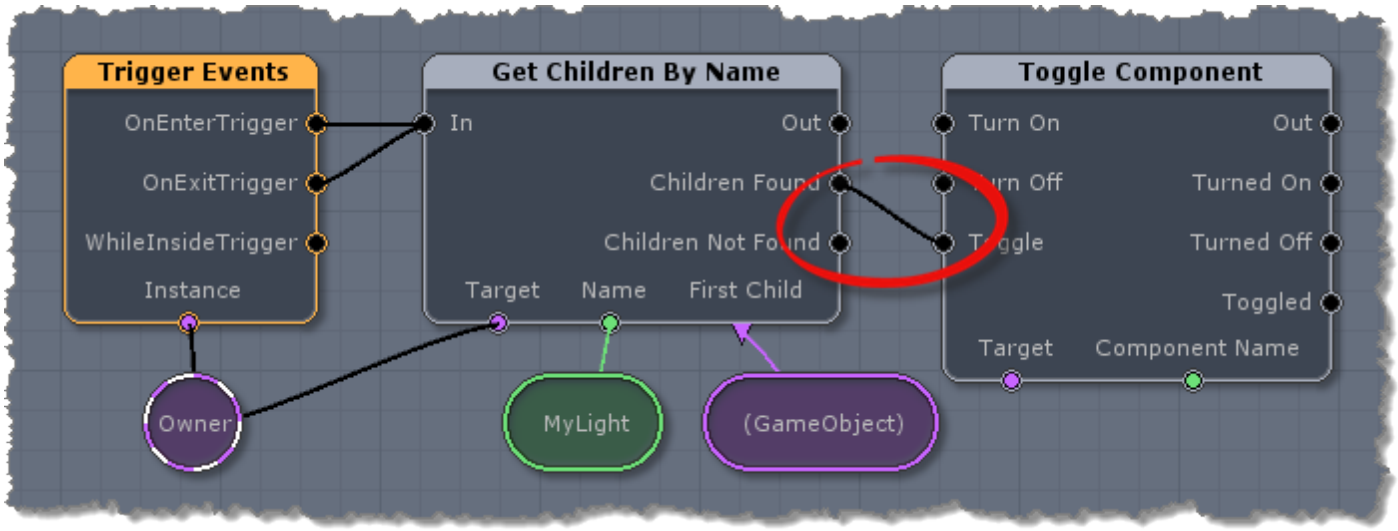
It is fine to leave this new GameObject variable node blank. It will get populated with a value once the node is run because it is a variable output node (notice the triangle shape of the socket). To learn more about variables, see "Variables" and to learn more about sockets, see "Nodes").

Step 6 - Now that we have the *MyLight* child GameObject going into our new GameObject variable, we can place our last node to turn the light on and off. One of the easiest ways to turn a light on and off in Unity to to enable or disable the *Light* component of the GameObject. A node that is perfect for that is the *Toggle Component* node.

1. Find the *Toggle Component* node in the Toolbox and place one on the Canvas to the right of the *Get Children By Name* node:



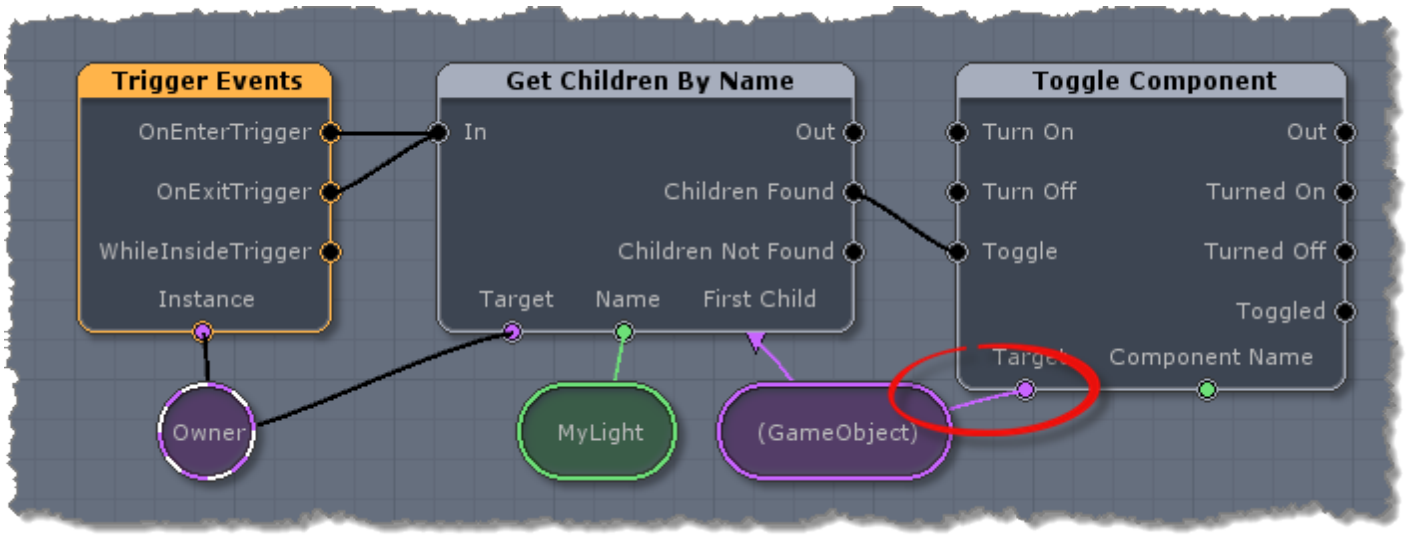
2. Connect the "*Children Found*" socket on the *Get Children By Name* node to the "*Toggle*" socket on the *Toggle Component* node.



We are using the "*Children Found*" socket so that we will only try to toggle the *Light* component if the *MyLight* child GameObject was found.

Step 7 - We are now ready to set the properties of the Toggle Component node.

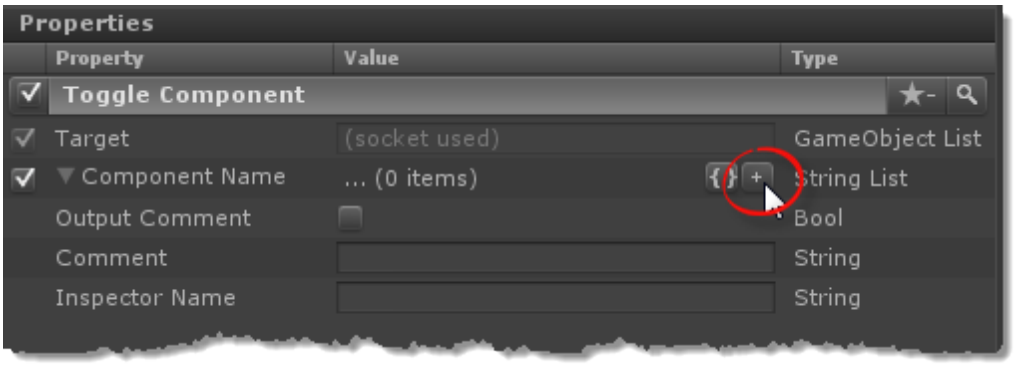
1. First we need to choose the *Target* (the GameObject whose component we want to toggle). In this case we want to use the GameObject variable that contains the *MyLight* GameObject that was found by the *Get Children By Name* node. Go ahead an connect the "*Target*" variable socket to that GameObject variable node by left-clicking in the center of the "Target" socket and dragging the line over GameObject variable and letting go:



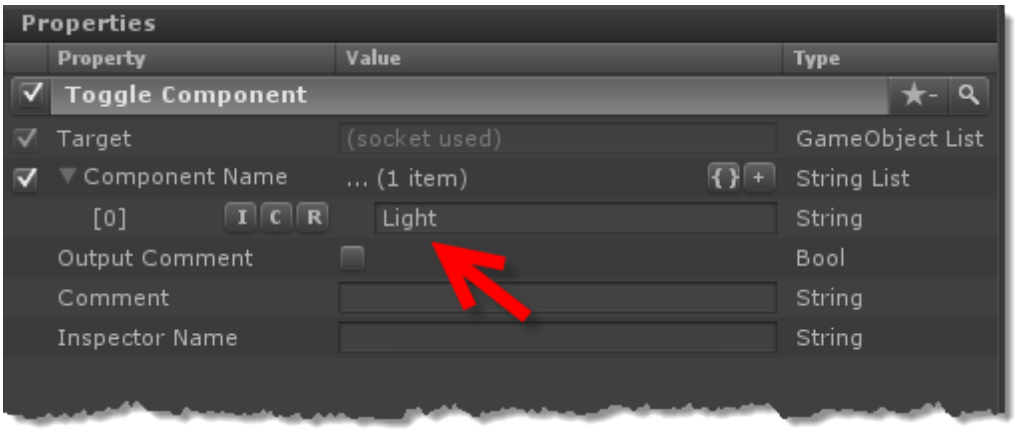
2. No we need to tell the *Toggle Component* node which component of the *Target* GameObject we want to toggle on and off. In our case it will be the *Light* component.

We need to select the node and input "*Light*" (without the quotes) into the "*Component Name*" property within the Properties Panel. However, when we go to do that we see that there isn't anywhere to input the value!

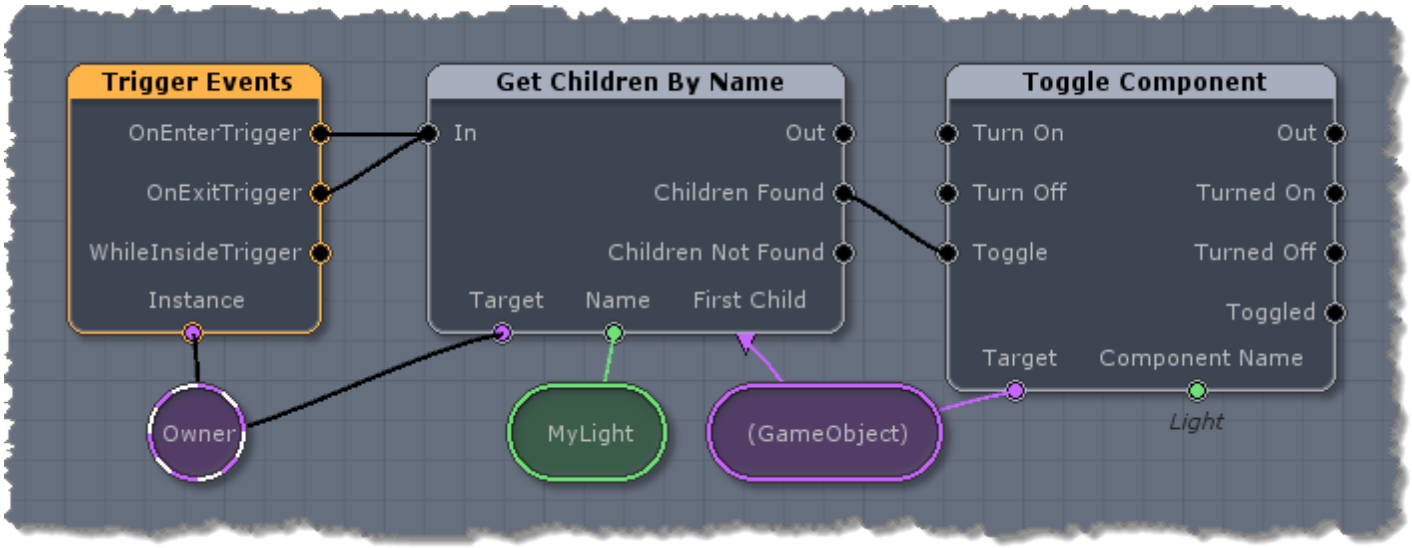
This is because this node let's you input more than one component name at a time to toggle and therefor uses Unity's default array list UI. Just click on the little "+" button to create a field to input the value:



Once you have done this and entered the value "*Light*" in the property, it should look like this:



You should now have a completed graph that looks like this:



Our graph is now completed! If you had setup a scene for testing, you should be able to run into the trigger and watch the light turn on and off as you enter/exit it.

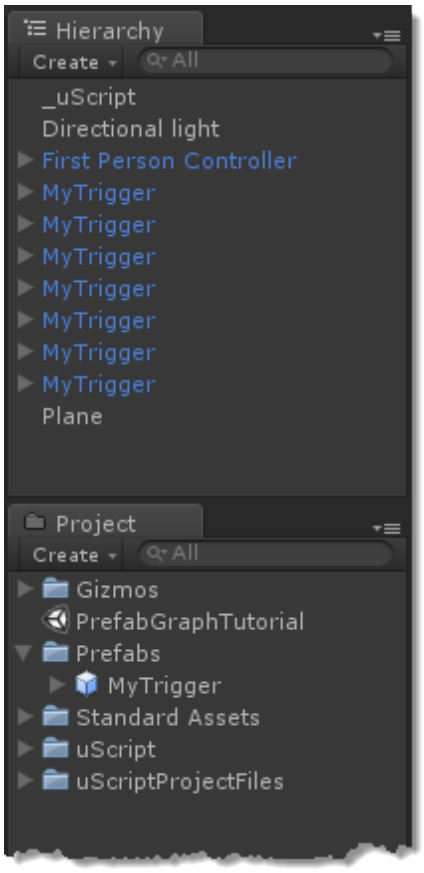
You have made a prefab graph!

Extra Credit

If you wanted to really see the power, you could turn the completed *MyTrigger* setup into an actual Prefab in your project (see Unity's documentation about doing that!) and place many instances of the *MyTrigger* prefab in your test project. You will see that they all work independently from each other.

You could then also add more logic to the single prefab graph (such as playing a sound of something even cooler) and all your prefabs will automatically inherit the changes because they are all using the same graph.

Many *MyTrigger* Prefabs in the scene:


Hierarchy
Create
All
_uScript
Directional light
First Person Controller
MyTrigger
MyTrigger
MyTrigger
MyTrigger
MyTrigger
MyTrigger
MyTrigger
Plane
Project
Create
All
Gizmos
PrefabGraphTutorial
Prefabs
MyTrigger
Standard Assets
uScript
uScriptProjectFiles

Making A Nested Graph

Delete this text and replace it with your own content.

Making Multiple Graphs

Sending Messages Between Graphs

Sending Data Between Graphs

Graph Making Tips

Delete this text and replace it with your own content.

Advanced uScript Topics

This section covers advanced and specialized topics regarding using and developing with uScript.

Advanced Reflection

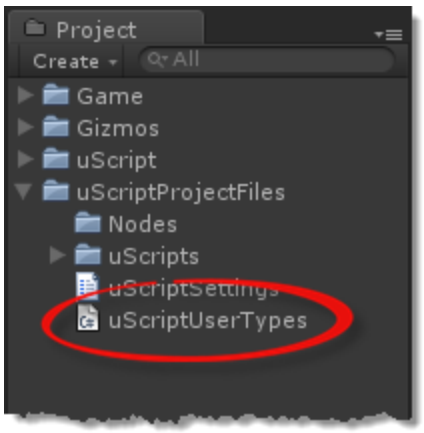
IMPORTANT! - The reflection feature is only available in the *Professional* and *Personal Learning Edition (PLE)* versions of uScript. If you wish to upgrade your license of uScript to the Professional version, please contact Detox Studios.

Reflecting Things That Aren't There

By default uScript will reflect anything that is in the currently loaded scene in the Unity Editor. There are times however that you may want to use reflected nodes and properties for things that are not in the scene at design-time but do appear once the game is running (like maybe assigning a component to a GameObject at run-time). In order to do this in uScript, you can assign the things you wish to force to be reflected by editing the ***uScriptUserTypes.cs*** file found in the root of the *uScriptProjectFiles* folder of your Unity project.

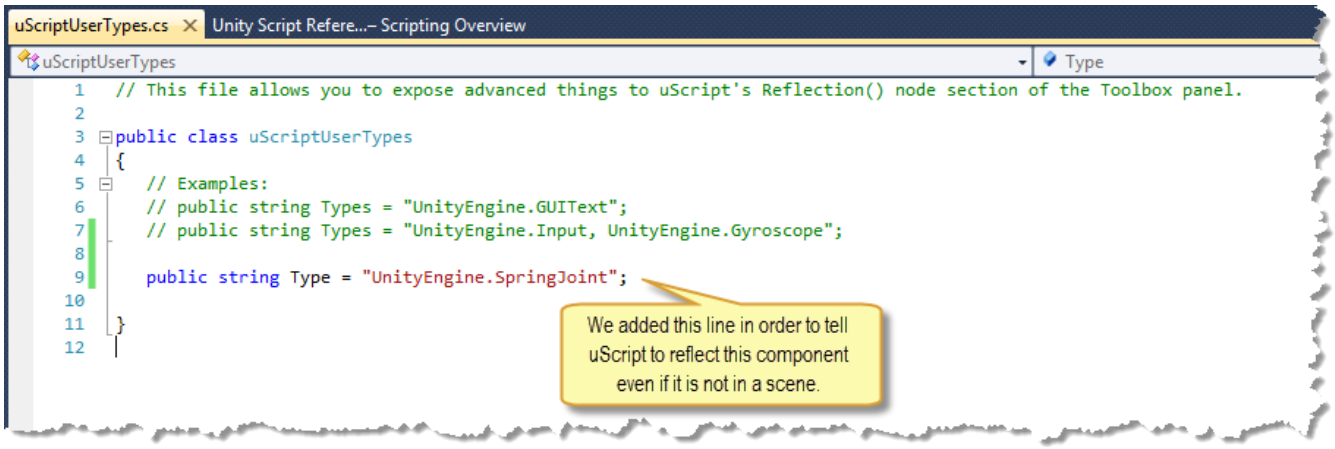
To modify the *uScriptsUserTypes.cs* file, do the following:

1. Open the file in your source code editor of choice (Unity used *MonoDevelop* by default). Just double-click the file in Unity to bring up Unity's default code editor. This file is located in the root of your project's *uScriptProjectFiles* folder:

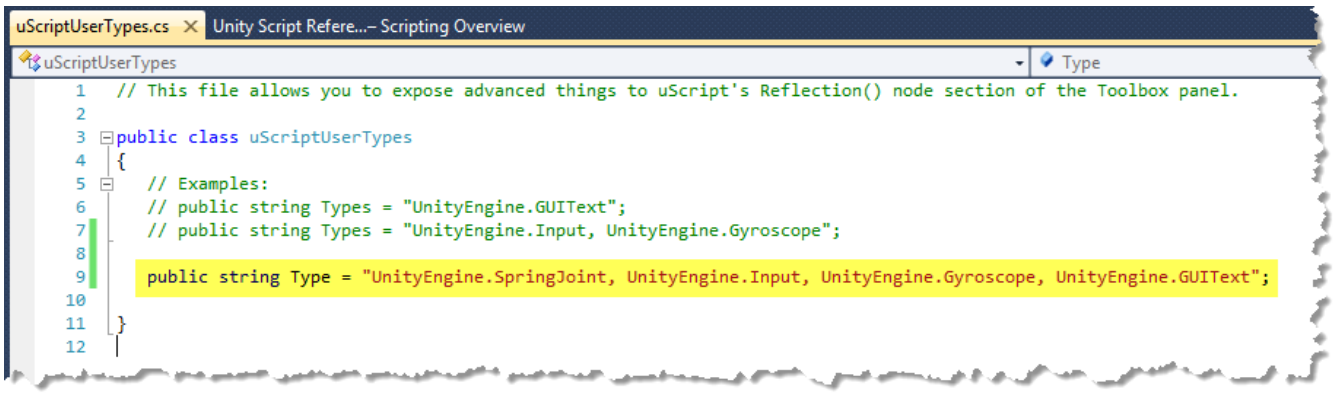


2. Once open, you can add Unity objects to be reflected. In this example we will be adding the SpringJoint component, though you can add any valid

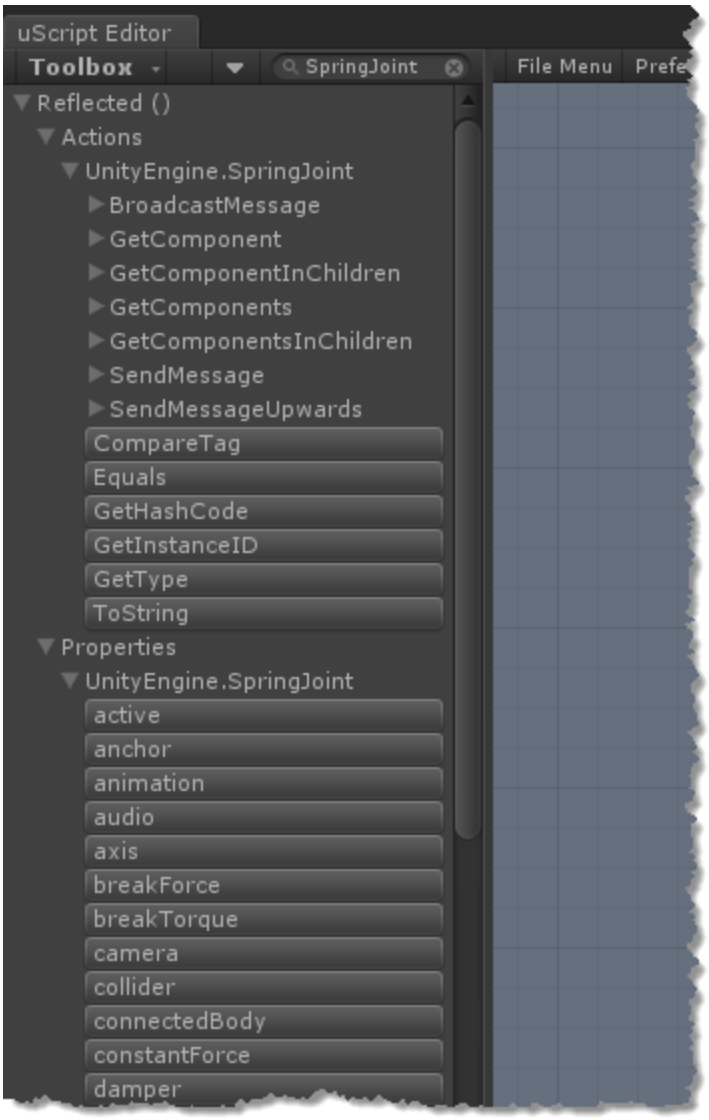
Unity object type:



Note! - *if you wish to add more than one Type to be reflected, please assign them to the same Type string variable and use commas to separate them like this:*



3. Save the file and return to Unity. If the file was edited properly, Unity will compile it. Once it is compiled close and re-open uScript *(if uScript is already open, you will need to close it first*) to see the additional items you have specified to be reflected. Like all reflected nodes, the new nodes will appear under the *Reflected ( )* section of the Toolbox:



Note! - *You will only be able to use these newly reflected nodes on GameObjects you are certain contain the reflected item when you plan to use it at runtime. As always, make sure you have assigned the Instance property for any reflected node to the GameObject you wish to refer to when accessing the specific instance of a reflected item.*

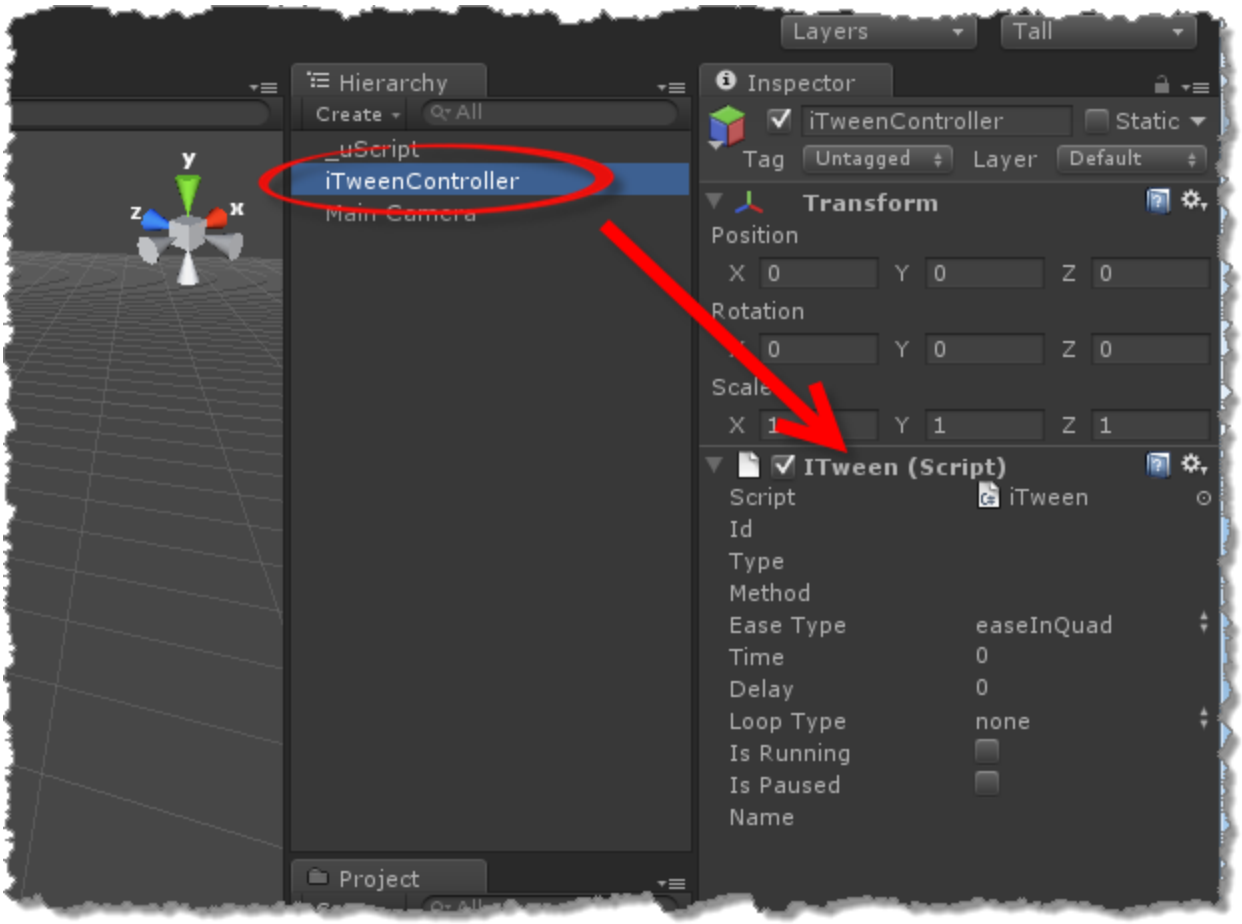
Reflecting Other Middleware

In order to reflect most other Unity middleware in uScript, you will need to either define the item to be reflected (as shown above in "*Reflecting Things That Aren't There*"), or by

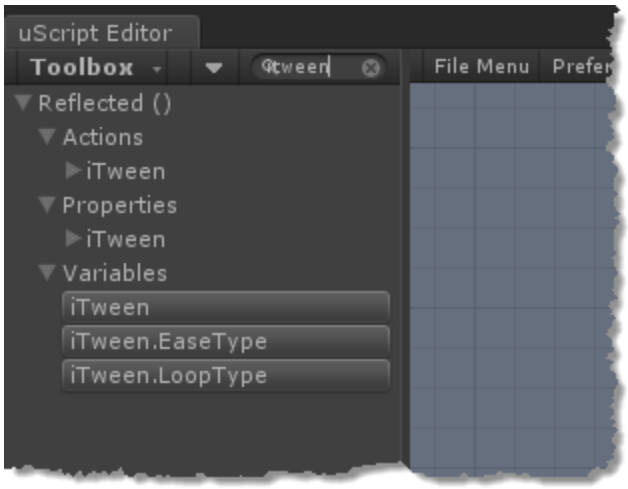
assigning the middleware script to a GameObject. In this example we will show you how to reflect iTween as an example.

1. Install the middleware into your Unity project.

2. Create a new, empty GameObject (ours is called *iTweenController* in this example) and assign the middleware's script file to it:



3. Now open the uScript Editor and look for the middleware to be reflected in the *Reflected ( )* section of the Toolbox:



Note! - *Some middleware may require custom nodes to be created to get the most out of using it in uScript. Some middleware providers (such as NGUI for example) have created nodes to better integrate there middleware with uScript. We recommend that you contact the middleware author as well as stop by the uScript forum to see if there has been custom nodes created for the specific middleware you are using.*

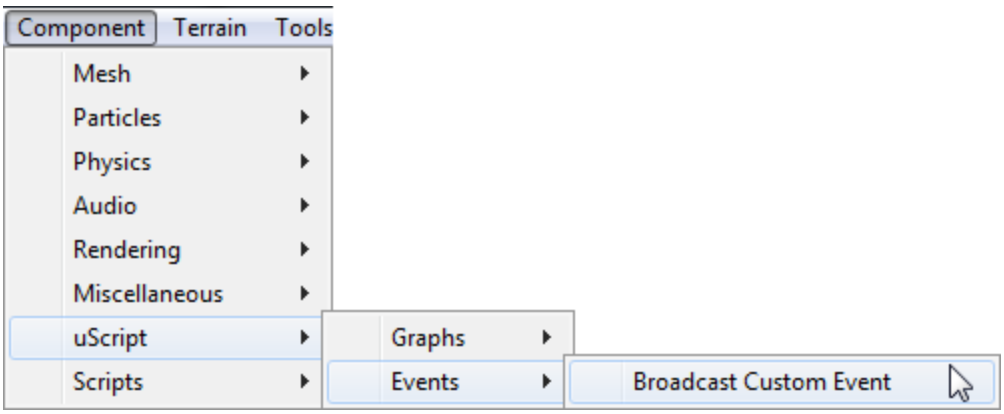
Animation Event Triggering

uScript ships with a script that you can use to remotely trigger logic in uScript called BroadcastCustomEvent.cs. The most popular use for this script is to trigger uScript logic from Unity's Animation Event system.

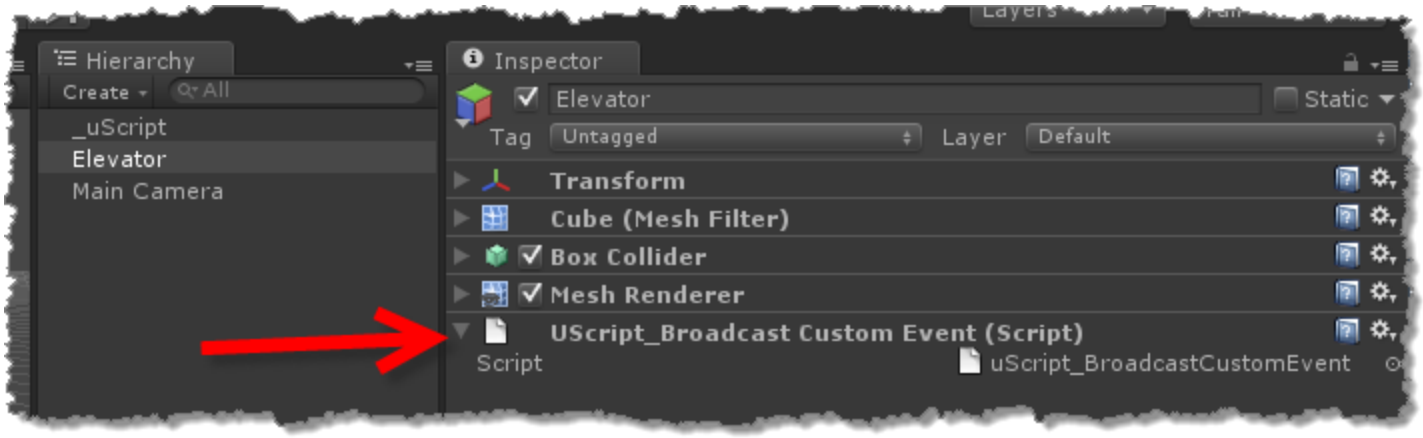
This can be very useful for uScript logic you wish to trigger based on a specific animation (or frame of an animation).

To use this script in your own project, follow these steps:

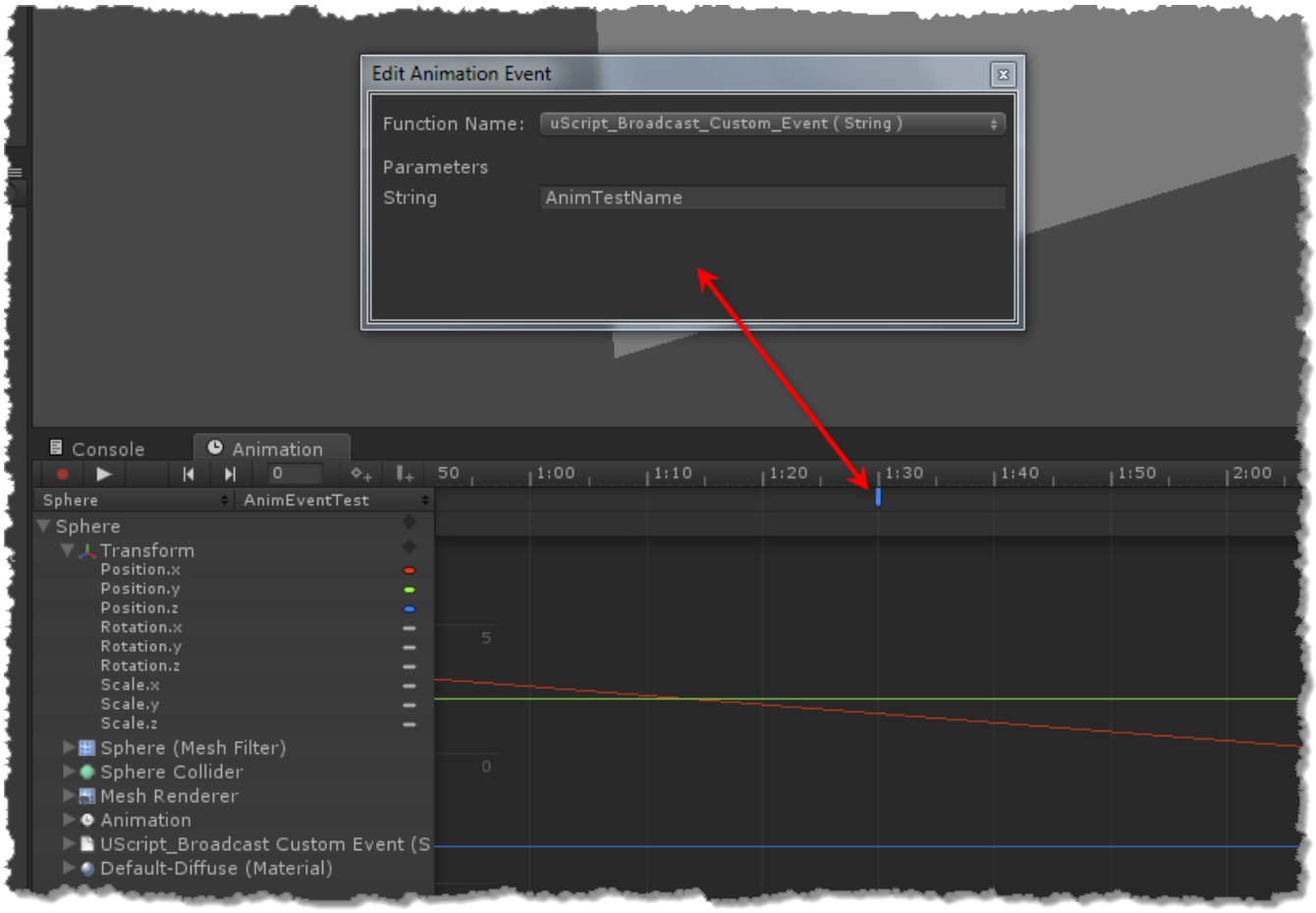
1. Assign the component to the GameObject with the animation you wish to add an Animation Event on:



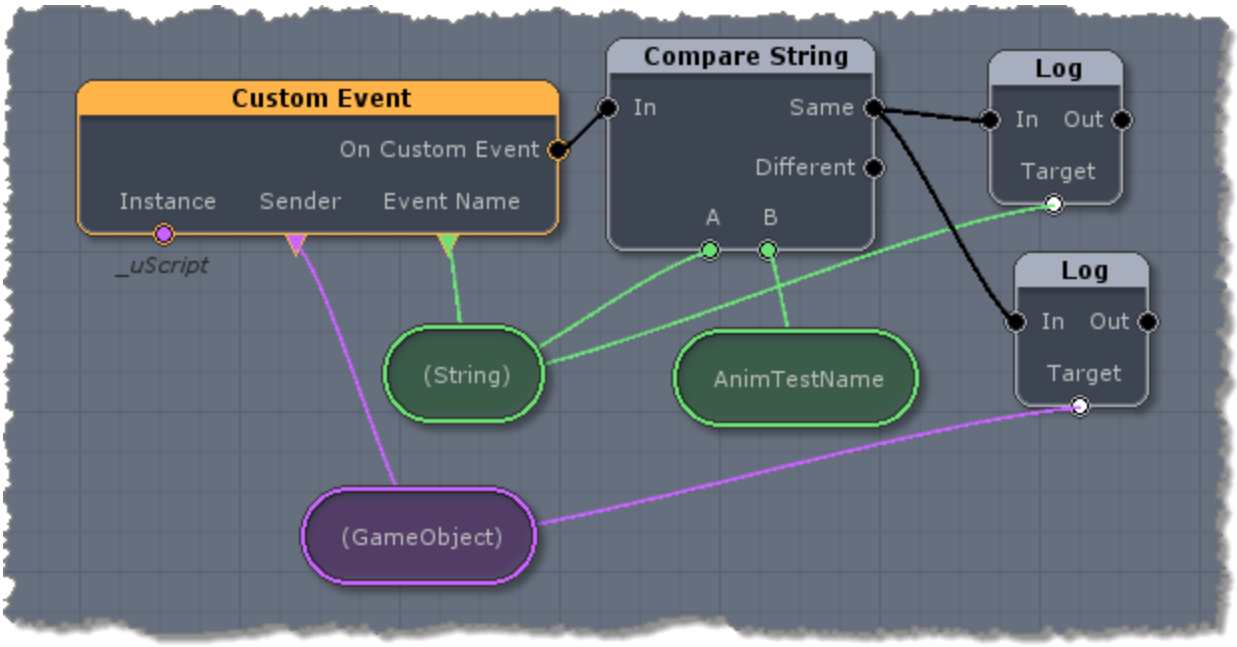
You will now have this script component on the GameObject:



2. Then create a new Animation Event in an existing animation for your GameObject and select the uScript function (*uScript_Broadcast_Custom_Event*), as well as provide the Event Name you wish to broadcast (*AnimTestName* in this case):



3. Now you can go into uScript and create a Custom Event event node and any following logic you wish to run when AnimTestName is broadcast:



Note! - *those also familiar with scripting can also use this script in other script files to also trigger logic from outside of uScript.*

Distributing Content With uScript Graphs

This section covers ways you can safely (legally) distribute projects and content that use uScript for redistribution through either a Unity package file or to the Unity Asset Store.

If you have used uScript to create scripts and prefabs that you wish to redistribute, you must be careful to not distribute the uScript editor tool itself. This would be a violation of the End User License Agreement (*EULA*) you agreed to when purchasing and using uScript either directly from Detox Studios or through the Unity Asset Store. The PLE version of uScript must also follow these guidelines to ensure uScript itself is not distributed.

> **Note!** - *If you wish to redistribute uScript graphs that you want others to be able to view or edit in the uScript editor, they must have their own uScript license. The purpose of this documentation is to show you how to distribute content such as prefabs and Unity packages for direct use by others without uScript installed. One suggestion if you just want them to see how the graphs are made would be to use uScript's screenshot export feature and include the PNG files along with your content.*

What Should Never Be Redistributed

We have tried to make it as easy as possible to separate the uScript Editor from the support files that may freely be distributed. For clarity, here is a list of files and directories that should never be included in any content you wish to distribute to others:

- The ***uScriptEditor*** folder - This folder contains the actual uScript Visual Scripting Tool editor files (most importantly the uScript.dll file) and should **never** be redistributed. No files in this folder are needed to run any content and graphs generated by uScript. There only purpose is to allow you to edit existing graphs and create new ones. If you wish to allow others to view, edit, or create uScript graphs they will need their own license of uScript-- either Retail or PLE depending on how the original content was authored and the usage conditions (free or commercial use).
- The ***License.rtf*** (or *License_PLE.rtf*) file - This file can be found in the root *uScript* folder and is for licensed users of uScript who purchased uScript directly through the Detox Studios website (this file does not exist in the Unity Asset Store version as they have their own license agreement). It should not be redistributed with your content.

- The ***ReadMe.txt*** file - This file can be found in the root uScript folder and is for licensed users of uScript only. It should not be redistributed with your content.

What You Can (And Should) Redistribute

The following parts of uScript were designed to be allowed to for redistribution as required by your content. As a legally licensed user of uScript, you may freely redistribute these files in whole or in part as you see fit:

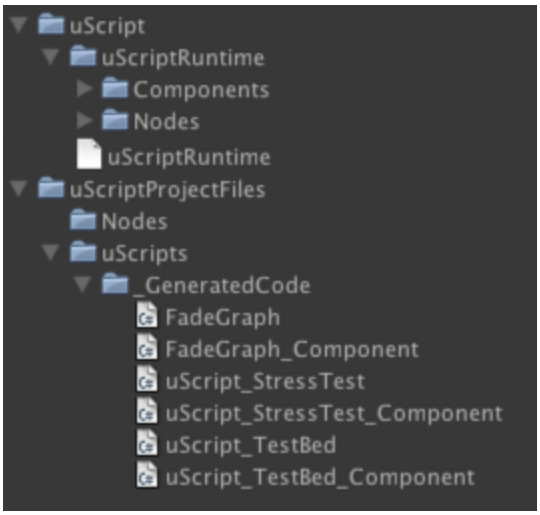
- The ***uScriptRuntime*** folder (required) - This folder and its contents (including the *uScriptRuntime.dll* and all nodes) should be included. It is fine to distribute the root uScript folder that this folder resides in as long as you do not include the uScriptEditor folder or any of its contents (see above section for details) which also resides inside the root uScript folder.
    - Distributing needed nodes only (optional) - if you know exactly which nodes you used in your uScript graphs you may be able to delete unused nodes files (they are C# .cs files) from inside the Nodes folder in order to reduce your overall file size and clutter to those you are distributing your work to.. You will know if you deleted a node you were using because Unity will generate compile errors. Again, this is optional though and should only be attempted if you feel comfortable doing so.
    - The ***uScriptUnityVersion.cs*** file (optional) - There is no need to redistribute this file as it is only used by the uScript Visual Scripting Tool editor. It won't hurt anything if it is included however.
- The ***uScriptProjectFiles*** folder (required) - You may redistribute this folder and any files and sub-directories within it. The only thing you may need to be mindful of is if you have put any third party nodes in the *uScriptProjectFiles/Nodes* folder that you may have licensed from others. In that case you will need to check with the node author(s) to be sure you can redistribute those custom nodes. If you found these nodes on the uScript forum (*http://uscript.net/forum*) you can freely use and distribute the nodes found there as everyone must agree that anything posted there is free to use by anyone for any purpose (please see the forum user agreement for more details).
    - Files you may not wish to include from *uScriptProjectFiles* (but are free to do so if you wish):
        - The ***uScriptSettings.settings*** file (optional)- there is not much use in redistributing the *uScriptSettings.settings* file as only the uScript editor uses that file.
        - Your ***.uscript*** graph files (optional) - There is no need to redistribute your *.uscript* graph files as they are only used by the uScript Visual Scripting Tool editor (which should not be redistributed). While redistributing these files should hurt anything you should be aware that they will take up extra space in your package and also give people

access to your graphs if they also have uScript (which you may not want). The only files that are required are the C# script files that uScript generated (that your content actually uses) in the *_GeneratedCode* sub-folder.

- The ***uScript Gizmos*** in the root Gizmos folder (optional) - If you wish, you may also redistribute the uScript Gizmos found in your Unity project's root *Assets/Gizmos* folder-- though they are not required for your content to run and should only be considered if you wish to have the uScript Gizmos show up for others on content you have generated with uScript.

Example

Here is an example of what your uScript files should look like after deleting all the forbidden uScript editor files from a Unity project (*see above*), leaving only the files and directories that should be redistributed with your work:



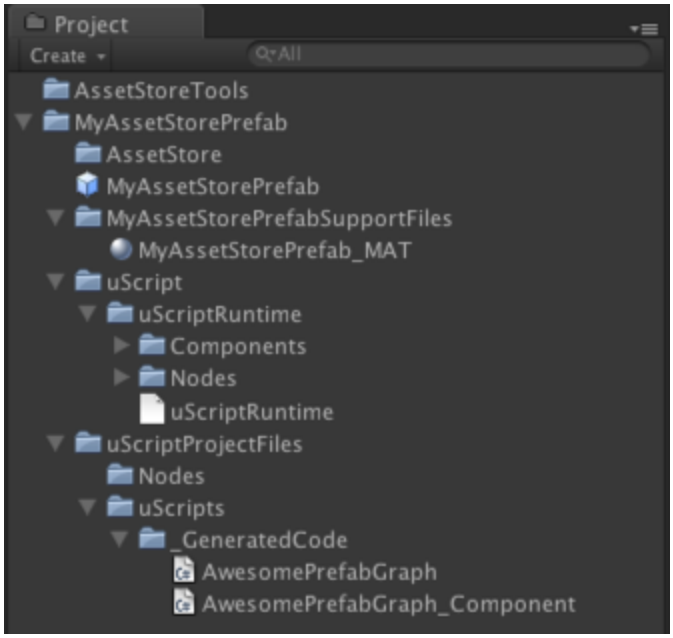
Moving uScript Files For Redistribution

It is recommended that if you will be redistributing something using uScript that you may want to "hide" all the uScript files and folders. This is a fairly straightforward process through the Unity Editor, though there are a few things you will want to keep in mind:

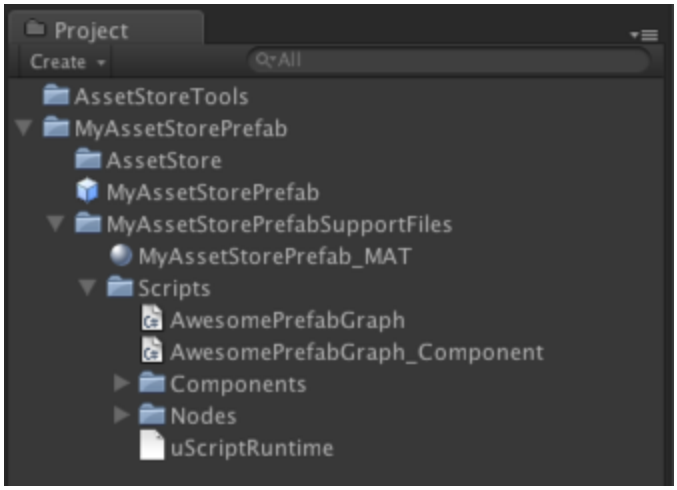
- If you plan to redistribute multiple content that all use uScript, you may not want to move the uScript files from their default locations because they users could end up installing multiple packages that might put the same named script files into multiple locations in their project. This would cause Unity to generate compile errors since the C# (.cs) files would be duplicated in multiple places within their Unity project.

- Moving the uScript files may make it harder for you to update your content later (or at least more time consuming).

That said, here is an example of a prefab ready for submission to the Unity Asset Store using the default uScript folder structure...



... and here is the same example where the uScript files have been moved deeper into the package to hide them away (everything is under a Scripts folder)...



Using uScript With Large Teams

When using uScript on a large team where not everyone has a uScript license (if all your developers do have a license of uScript, there is nothing to worry about), you will need to be sure that you do not install, or add to source control, the *uScript/uScriptEditor* folder and files. The files contained in this folder are only used for the uScript Editor itself. If you have developers that need to load uScript graphs into the editor for any reason, they will need their own licensed copy of uScript.

For more information on hat files should and should not be distributed with your project (including through source control), see the topic "Distributing Content With uScript Graphs" in this section.

> ***Note!*** *- Some large teams using source control software such as Perforce have reported needing to setup their source control system to make sure it handles uScript's .uscript files as binary (which they are) in order to avoid issues or file corruption from attempting to merge revisions and such.*

Modifying uScript

uScript has been designed to be expanded and modified in several ways, For more information on ways you can extend uScript for your project's needs, such as creating new nodes, please see the uScript Development Guide.

References

This section contains useful information regarding resources beyond this user guide that you may find helpful in learning and using uScript.

Hot-Keys

Note! - *if you are using O SX (Mac), replace Control (CTRL) with the Command (CMD) key.*

General

Action	Key	Alt-Key	Mouse	Alt-Mouse	G-UI
Cut	CTRL + X				
Copy	CTRL + C				
Paste	CTRL + V				
Undo	CTRL + Z				
Redo	CTRL + Y				
Close uScript	CTRL + W				

Canvas

Action	Key	Alt-Key	Mouse	Alt-Mouse	GUI
Zoom In/Out			Mouse Wheel		
Zoom Out (by 10%)	-				
Zoom In (by 10%)	=				
Zoom to 100%	0 (*zero*)				
Hide/Show Panels	` (*tilde key*)				
Delete Selected	DELETE	BACKSPACE			
Center on next event node	]				
Center on previous event node	[				
Reset Canvas to center	HOME	CTRL + H			
Deselect All	ESC		LMB (*over Canvas*)		Canvas
Toggle Grid	CTRL + G				
Place String variable where clicked			S + LMB		Canvas
Place Vector3 variable where clicked			V + LMB		Canvas
Place Int variable where clicked			I + LMB		Canvas
Place Float variable where			F + LMB		Canv-

Canvas (continued)

Action	Key	Alt-Key	Mouse	Alt-Mouse	GUI
clicked					as
Place Bool variable where clicked			B + LMB		Canv-as
Place GameObject variable where clicked			G + LMB		Canv-as
Place Object variable where clicked			O + LMB		Canv-as
Place Comment node where clicked			C + LMB		Canv-as
Place External Connection where clicked			E + LMB		Canv-as
Place Log node where clicked			L + LMB		Canv-as
New node selection			LMB (*over node)*		Canv-as
Toggle node selection			SHIFT + LMB		Canv-as
Marquee - New box selection			LMB + Drag		Canv-as
Marquee - Add to selection			SHIFT + LMB + Drag		Canv-as
Marquee - Remove from selection			CTRL + LMB + Drag		Canv-as
Move node			LMB + Drag		Canv-as
Pan Canvas			ALT + LMB + Drag	MMB + Drag	Canv-as / Nod-e
Context Menu			RMB		Canv-as / Nod-e

Unity

Action	Key	Alt-Key	Mouse	Alt-Mouse	G-UI
Launch the uScript Editor	CTRL + U				

Tutorials

Detox Studios provides many video tutorials. For a list of current video tutorials, please go here: http://www.uscript.net/docs/index.php?title=Tutorials

Example Projects

Detox Studios provides many examples through a Unity project called *ExampleProjects* that contains many scenes showing different ways to use uScript with Unity. You can download this Unity project file here:
http://www.uscript.net/docs/index.php?title=Example_Projects

> **Note! -** *make sure you download the correct version of this project for the version of Unity you are using and to follow the installation instructions found on the web page.*

Web Links

uScript Community Website - uscript.net

uScript Community Forum - uscript.net/forum

uScript Facebook Page - www.facebook.com/uScript

uScript Twitter - twitter.com/uScript

uScript YouTube - www.youtube.com/user/uscripttool

uScript Screencast - www.screencast.com/users/uScript

Detox Studios Website - www.detoxstudios.com

Unity Technologies Website - unity3d.com

Unity's Online Documentation - docs.unity3d.com