User Guide

Visual Scripting for Unity

Formatted for LaTeX
by MultiMarkdown

Contents

Part I

Introduction

Chapter 1

What is iCanScript?

> Think twice before you start programming or you will program twice before you start thinking.
>
> - Unknown

More then 50 years after ENIAC[1], most programmers still rely solely on textual editors to express their creative thoughts into programs that can be compiled or interpreted by computers.

As the programs increase in complexity, many corporations encourage the interdisciplinary[2] discussion of the high-level concepts overseeing the software product to be created. In all cases, these high-level concepts are expressed in diagrams for it is the simplest and most direct way of building and iterating on those fundamental aspects of the product. It is this trend to graphically convey software concepts that conducted to the creation of iCanScript.

iCanScript is part of a new breed of visual editors that focuses on expressing programmatic behaviour using nodal diagrams. It integrates with existing toolsets extending the text editor for writing Unity[3] scripts. It offers a visual workflow to build, review, and publish programming logic preserving the ability to write low-level logic using your favourite coding software.

[1] ENIAC (Electronic Numerical Integrator And Computer) was the first electronic general-purpose computer. It was Turing-complete, digital, and capable of being reprogrammed to solve a full range of computing problems.

[2] The term interdisciplinary is used to express a team composed of various expertise or talent not limited to software engineering.

[3] Unity is a trademark of Unity Technologies.

Chapter 2

Feature Overview

- Fully integrated functional & state chart diagrams;
- Unlimited nesting of nodes allows for structured visual scripts;
- Visual scripts can be saved and loaded to/from Prefabs to create your own personal visual library;
- Each node can be displayed as:
  - unfolded (displays nested nodes);
  - folded (hides nested nodes) or;
  - iconized.
- Visual Editor zoom in/out enables bird's-eye view and eases navigation of large diagrams;
- Visual library includes Unity's runtime functionality;
- Library can be expanded with your own scripts and external packages;
- Auto-save when scene is saved;
- Auto-compiled when application is started;
- Port values are displayed in real-time when the application is executing;
- Available for Standard and Professional versions of Unity.

Chapter 3

When to use Visual Scripts?

Our goal in creating iCanScript is to simplify and accelerate scripting within Unity's development environment. iCanScript offers a visual alternative to traditional textual scripts enabling greater accessibility for both programmers and non-programmers.

We believe that depicting the scripts allows for a larger community of talents to interact with the high-level logic that drive your games. It simplifies the understanding of the overall structure to all individuals in your production.

So can visual scripts replace all textual scripts in a production?

Probably not... The fact is that visual and textual scripts have different strengths and weaknesses.

Visual scripts excel at summarizing complex interactions that typically span multiple source files when implemented using textual scripts. These interactions include:

- mission control logic;
- behaviour state charts;
- gameplay logic; and
- any high-level functional diagrams.

Textual scripts are best at expressing tight calculations that can be implemented in a single source file. The textual scripts can thereafter be integrated into iCanScript as building blocks of high-level logic. These algorithms include:

- calculation loops;
- search loops;
- complex mathematical expressions; and
- large selectors (switch-case).

We hope that you find the right balance in using iCanScript combined with traditional scripting methods to increase productivity and, more importantly, *HAVE MORE FUN* in writing your scripts.

Chapter 4

Who will Benefit from Using iCanScript?

The Novice and Intermediate Programmer: The novice programmer will greatly benefit from the immense Unity library available for visual scripting. The programmer will find enjoyable the ability to create ***state charts*** and ***data flow diagrams*** in a visual and coherent environment. The ability to ***aggregate multiple functions and state charts*** into a ***package node*** combined with the ***unlimited nesting*** capabilities of iCanScript provides a sandbox to design at multiple levels of abstraction.

The Expert Programmer: For the expert programmer, iCanScript ***complements textual scripting with visual scripting***. Using .NET reflection and optional meta-attributes, textual scripts can easily be published to the iCanScript library. Therefore, the ***programmer remains in full control over the scripting workflow*** deciding when and where visual scripting benefits the overall project.

Non-Technical Individual: iCanScript visual environment greatly simplifies the ***understanding of script structure*** for all individuals including non-technical individuals. By removing the "needy greedy" details of programming languages, iCanScript allows the ***non-technical individual to create or modify simple scripts***. The creation of large and maintainable scripts require a minimal knowledge of programming structures (even with visual scripting) and may prove to be challenging for non-technical individuals.

Part II

Installation & Upgrade

Chapter 5

Installation

Please note that the installation of iCanScript from the Unity Asset Store is currently not discussed. It will be added in future releases of this document.

 iCanScript is a downloadable plugin for the Unity 3D game engine. It comes in the form of a Unity package that must be installed for each project.

The initial download of iCanScript is available from the project website[1]. User registration is required to obtain the latest version of the iCanScript software via email. An overview of the registration, download and installation procedure is available in the following video:

[2]

Follow these steps to install iCanScript for the first time:

[1] http://www.icanscript.com
[2] http://youtu.be/LJIqMaUNoAU

1. Visit the iCanScript web site at www.icanscript.com;
2. Click the download button from the home page;
3. Fill-in the registration form (the email is important);
4. Shortly after, you will receive an email with the latest version of iCanScript in attachment;
5. Open the email and save the iCanScript package;
6. Launch Unity and import the iCanScript package (see Figure 5.1);
7. You are now ready to start . . .

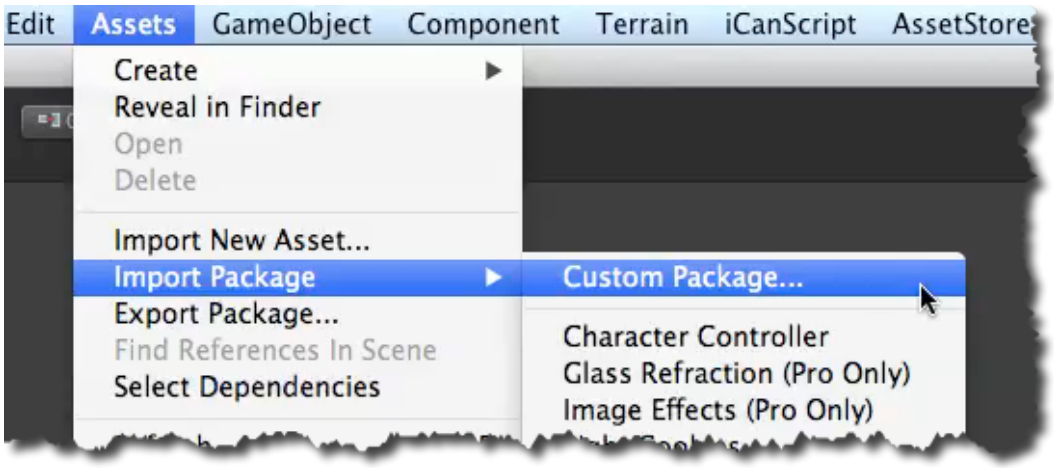


Figure 5.1: Importing iCanScript into Unity.

You now have the knowledge to download and install iCanScript in your own project. I propose that you move on to the [quick start tutorial][] where you will build your first visual script.

iCanScript User Interface Tips

Subsequent software releases can be downloaded using the upgrade (chapter 7) feature of iCanScript.

Chapter 6

Removing iCanScript

The following steps are needed to uninstall iCanScript from your project:

- Remove the ***iCanScript*** folder;
- Open the *Gizmos* folder and remove the ***iCanScriptGizmo.png*** file;
- Remove the optional ***iCanScript_Nodes*** package;

Note: Uninstalling the previous version of iCanScript is required when performing an upgrade.

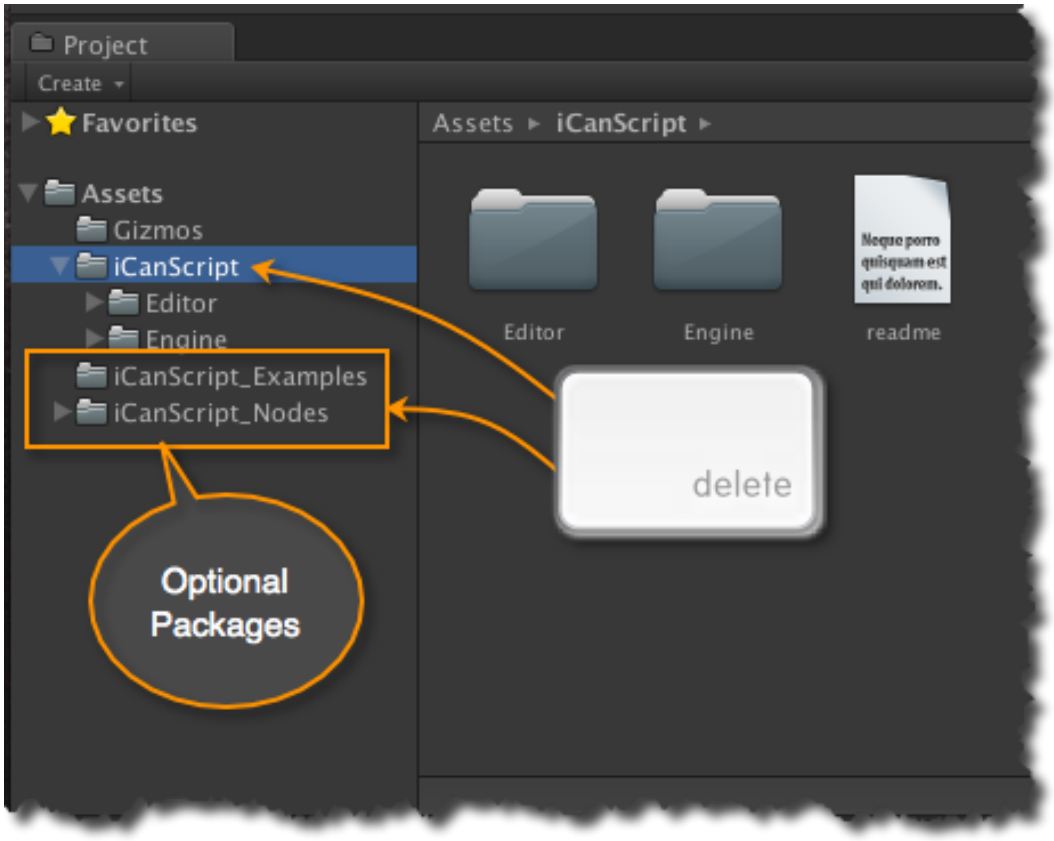


Figure 6.1: Uninstalling the iCanScript packages.

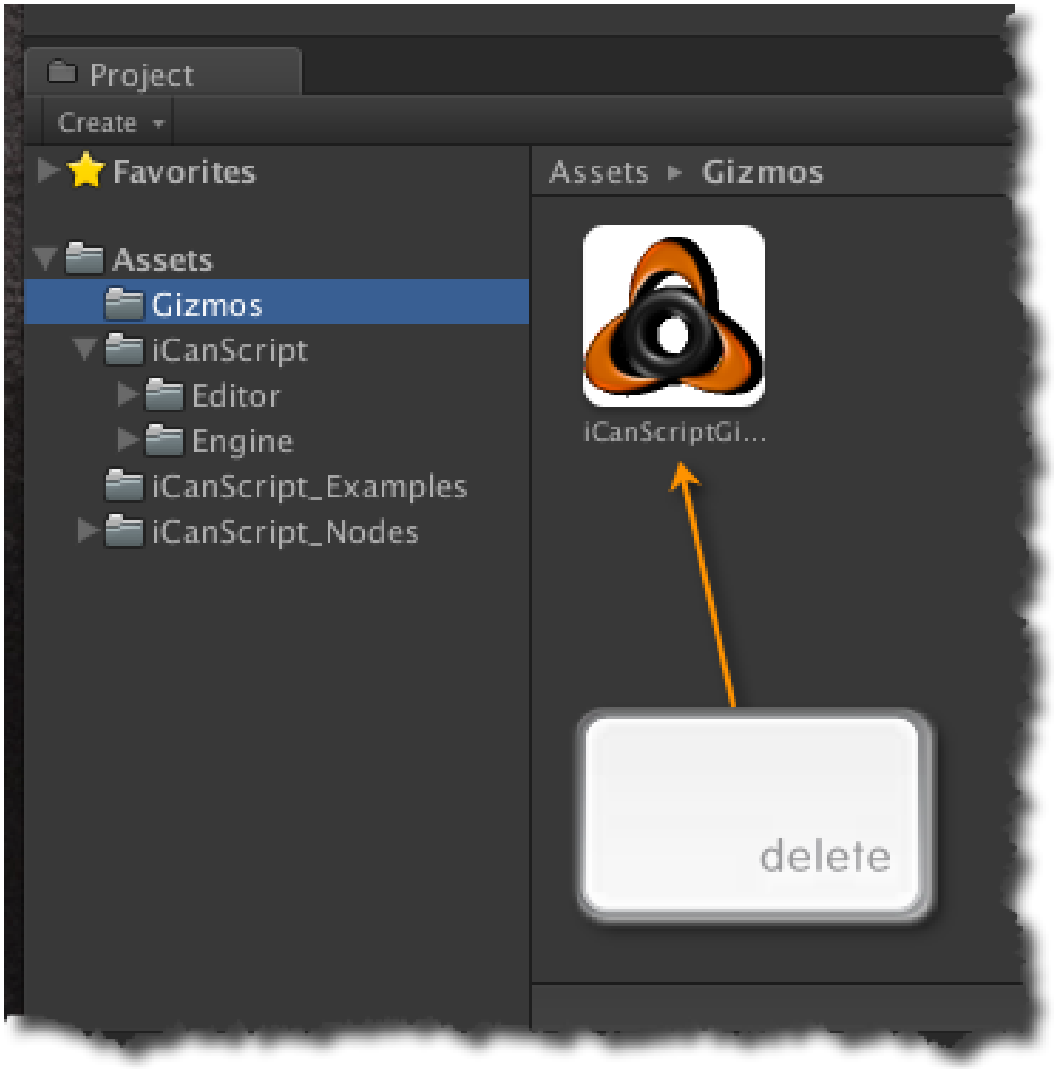


Figure 6.2: Removing the iCanScript Gizmo.

Chapter 7

Upgrading iCanScript

Improvements to iCanScript are regularly made available as downloadable software updates.

The following steps are needed to verify for and update to the latest version of iCanScript:

1. Verify for an update using the menu item: ***Help->iCanScript->Check for Updates...***;
2. If an update is available, a dialog box will guide you to the download page;
3. Download the latest version of iCanScript;
4. Open Unity and uninstall the current version of iCanScript (see Removing iCanScript (chapter 6));
5. Import the latest version of iCanScript (see Installation (chapter 5));
6. An alert box will be shown if data conversion is required after the upgrade;
    - data conversion is performed in memory;
    - each visual script data is converted independently;
    - converted data is persisted when the scene is saved;
    - data rollback is performed if scene is not saved after an upgrade.
7. Save the scene to complete the upgrade.

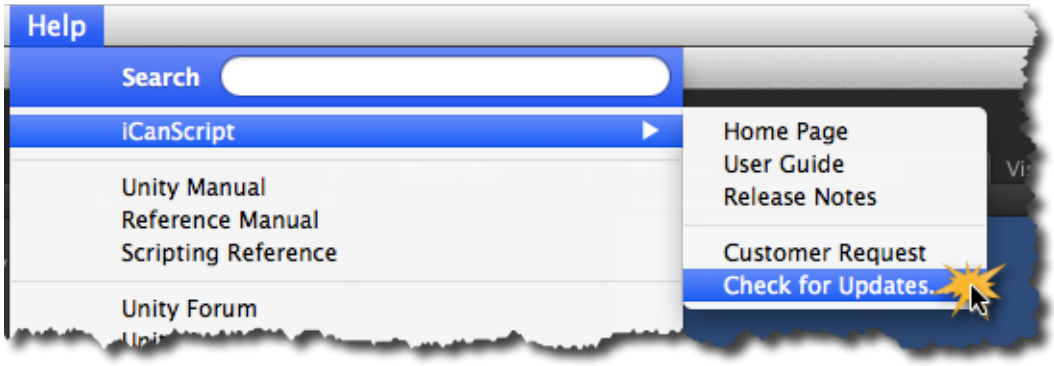


Figure 7.1: Verifying for an update.

Part III

Visual Script Anatomy

Chapter 8

Nodes, Ports, and Bindings

A good understanding of the fundamental building blocks of a visual script is necessary to effectively use iCanScript.

A visual script includes three (3) types of component being:

- **Nodes**;
- **Ports**; and ...
- **Bindings.**

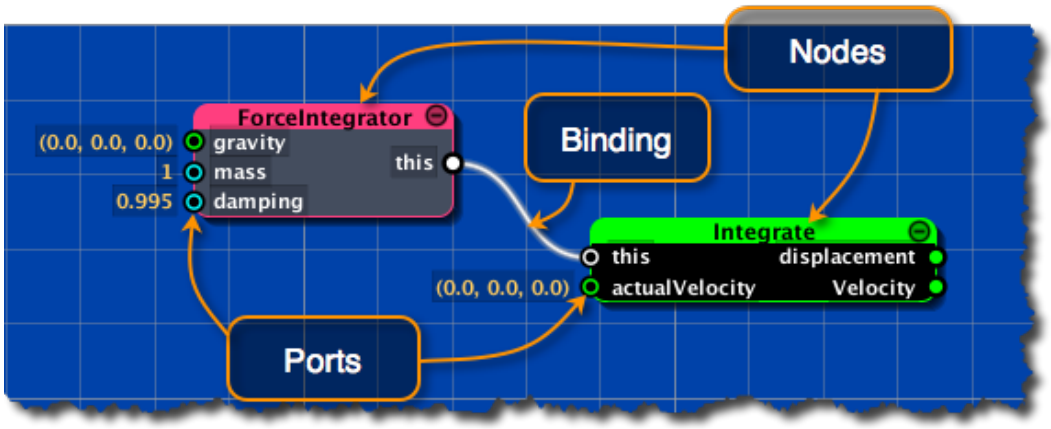


Figure 8.1: Visual Script Building Blocks.

The **node** is the primary ingredient of visual scripts. Its main purpose is to encapsulate various type of behaviours such as:

- variables & functions;
- algorithms;

- states and state charts; and
- nested visual scripts (i.e. packages or submodules).

The **ports** are the public interfaces to nodes. Multiple ports may exist on the same node each representing a distinct interface. A single port is directional and is either an input port or an output port both never both. The ports are further classified as:

- *data flow:* feeding or extracting data to/from the node behaviour;
- *control flow:* controlling the execution state of the node (*boolean* value);
- *state transition:* controlling the state transition triggers (state charts only).

The **bindings** define relationships between ports. Bindings can only be created between compatible ports. Furthermore, bindings are directional and can only exist between an output port and one or more inputs port(s). That is, connecting input ports together or output ports together is not permitted. The following summarizes the permissible bindings:

- from an output data port to one or more input data or control port(s) if the data type is compatible;
- from an output control port to one or more input data or control port(s) if the data type is compatible with a *boolean* value;
- from an output state port to one input state port using a transition trigger module.

Exception: iCanScript includes a special port, named the *Multiplexer Port*, that bridges multiple output ports into a single output port. This port is especially useful when multiple exclusive execution paths of a visual script must combine into a single set of data values. (see Data Flow Diagram (chapter 13) for additional details).

iCanScript Advanced Topic

iCanScript internally implements the *Multiplexer Port* using a specialized *data multiplexer node*. For convenience to the user, this multiplexer node is always iconized and positioned on the edge of the parent node giving the illusion of being a *multiplexer port*.

Chapter 9

Node Anatomy

The *node* is by far the most elaborate component of visual scripts. It plays a key role in the structure, execution, and layout of the visual script.

All nodes in iCanScript share a common set of attributes:

- **Name:** A character string representation of the node;
- **Type:** Identifies the node specialization.
- **Tree-like Hierarchical Structure:**
  - A parent node;
  - Zero or more child components:
    - *Ports*;
    - *Child Nodes* (nested Visual Script).
- **Graphical Representation:** (see figure 9.2
  - Position within the parent node;
  - Display State:
    - *Unfolded*;
    - *Folded*; or
    - *Iconized*.

The following image depicts an unfolded node as seen in the *Visual Editor*:

The following image depicts the same unfolded node as seen in the *Tree View*:

1. **Node Name:** User configurable name for the node. The name is editable in the inspector or the hierarchy tree. The programmatic function /type name is used by default.

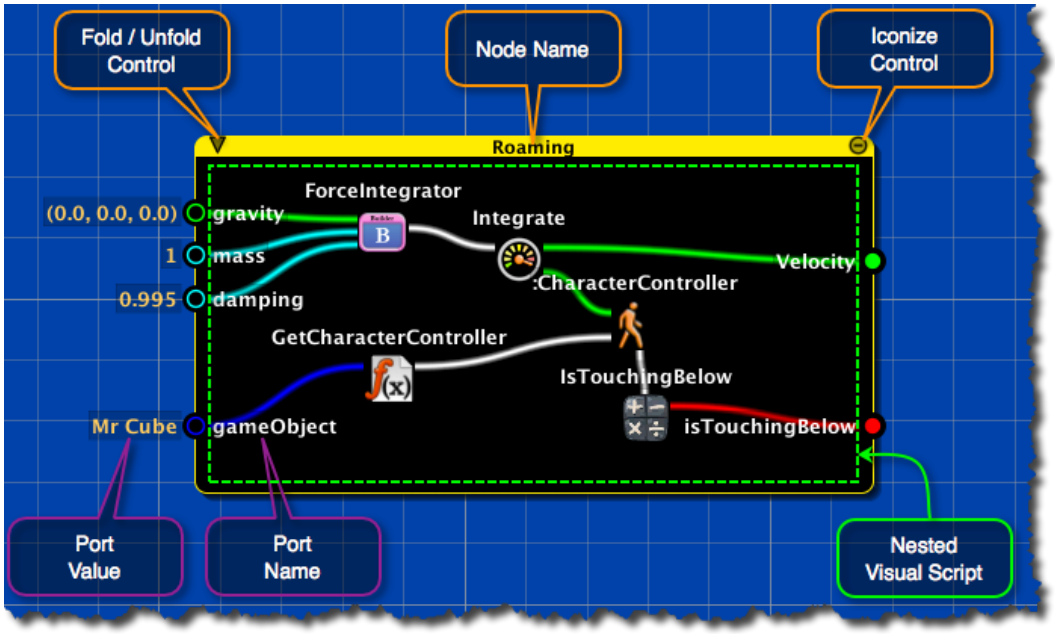


Figure 9.1: Anatomy of an unfolded node as seen in the *Visual Editor*.

2. **Iconize Control:** Clicking this pictogram causes the node to take its iconic representation.
3. **Fold/Unfold Control:** Clicking this pictogram toggles the node between its folded and unfolded representation. The fold/unfold control is available only for those node types that can contain nested visual scripts. (see [Node Types] for details).
4. **Node Ports:** Ports are positioned on any of the four node edges.
   - *Port Name* is displayed inside the node;
   - *Port Value* is display outside the node.
5. **Nested Visual Script:** The central area of the node is used to manage the nested child nodes. The child node container is only visible when the node is unfolded. See section [Node Types] for details on which node types support nested children.

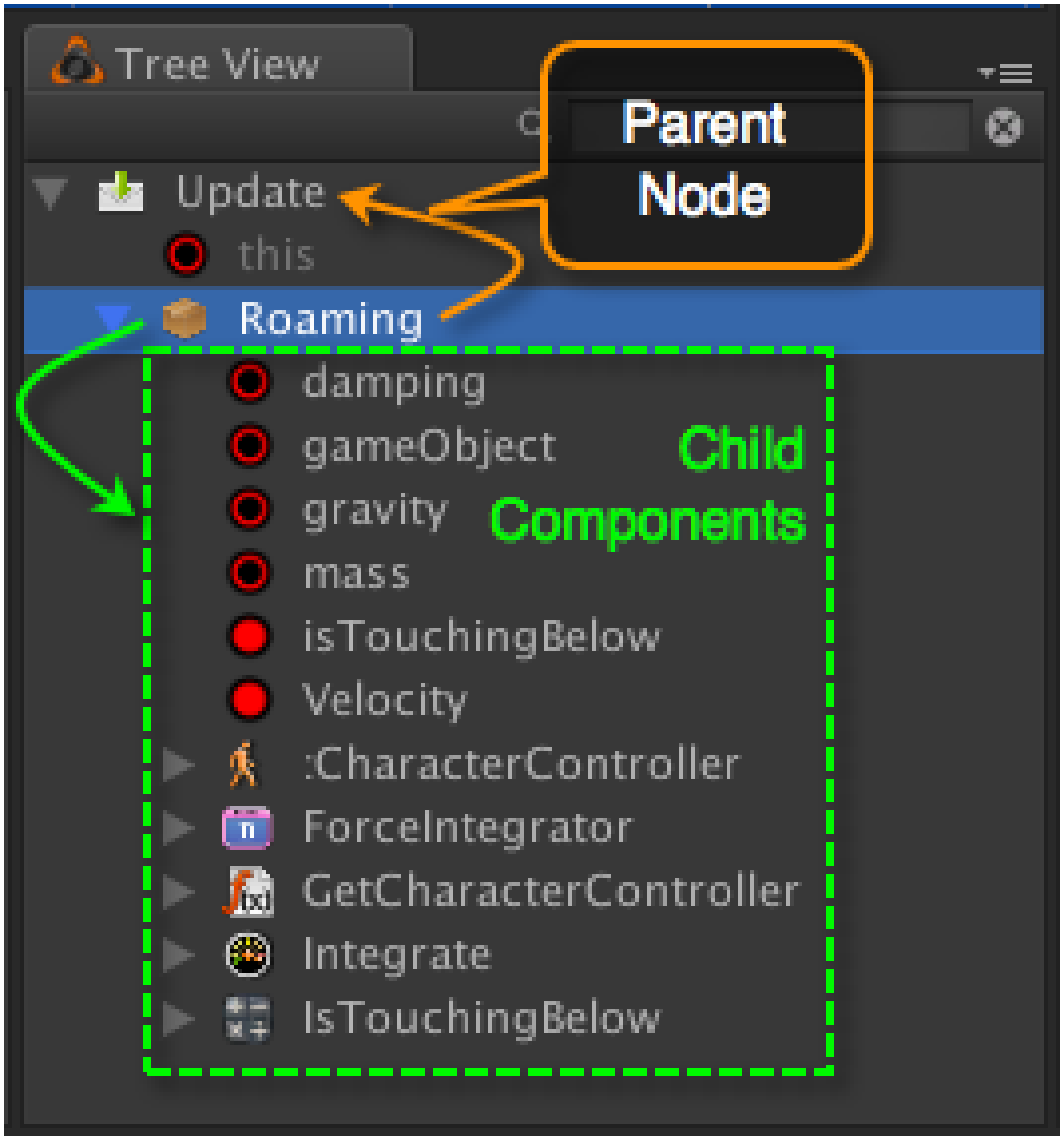


Figure 9.2: Anatomy of an unfolded node as seen in the *Tree View*.

Chapter 10

Specialized Nodes

The nodes come in various types or specialization. For example, their is a specialization that represents *Variables*, another that represents *Functions* and yet another for *Message Handlers*.

Chapter 11

Port Types

Chapter 12

Node Nesting

Chapter 13

Data Flow Diagram

Chapter 14

State Chart Diagram

14.1 States

Entry Function

Update Function

Exit Function

Nested State

14.2 Transitions

Trigger Function

Transition Behaviour

Part IV

Integration with Unity

Chapter 15

Unity Integration

iCanScript is designed to seamlessly integrated into the Unity development environment. It builds on the philosophies and application programming interfaces (APIs) available to Unity script programmers.

iCanScripts integrates with the following Unity concepts:

- Visual scripts are standard Unity *Components*;
- Each visual script is compiled into an Unity *Behaviour* and executed as such;
- All Unity engine programming interfaces are available in the iCanScript library database;
- All *Behaviour Messages* are implemented as *Message Handler Nodes*;
- Visual script libraries are implemented with Unity *Prefabs*;
- Visual scripts are saved and loaded with the game object they are attached to;
- Visual scripts are compiled when starting the Unity engine.

iCanScript extends the Unity Concepts with:

- Functional diagrams;
- State Chart diagrams;
- Automatic data driven execution sequencing;
- Automatic deadlock avoidance.

Note: Proper usage of iCanScript requires minimal knowledge of the Unity platform.

Chapter 16

Adding a Visual Script

To use iCanScript, you first need to install a visual scripts on one of your game objects. To illustrate this process, we will add a visual script to a sphere that will later be used as a trigger zone.

To do so, you must:

1. Select a game object to contain the visual script;
2. Create a visual script using the menu item: ***iCanScript->Create Visual Script***.

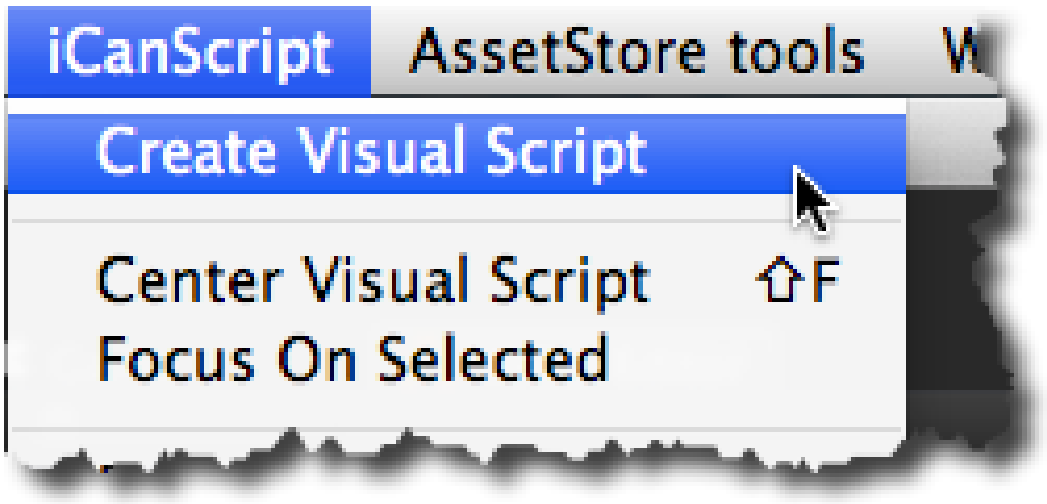


Figure 16.1: Create Visual Script menu item.

Et voilà, you have a visual script !!! It does not do anything yet but it is ready to listen and react to Unity messages. We will look at defining and handling Unity messages in the next section. For now, let's examine the changes brought when creating the visual script.

When iCanScript creates a visual script, it attaches two (2) Unity script components on the game object. These components are:

1. ***iCS_ VisualScript (persistent storage)***
2. ***iCS_ Behaviour (runtime compiler & execution services)***

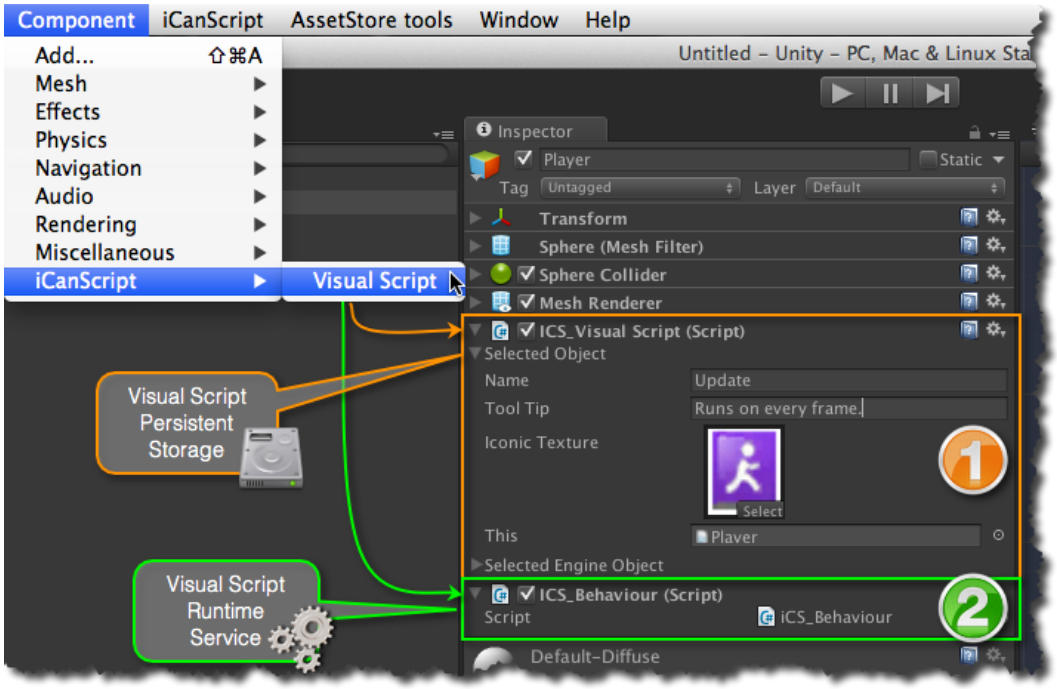


Figure 16.2: iCanScript visual script components.

The *iCS_ VisualScript* contains the visual script persistant data including the definitions of nodes, ports, bindings as well as their layout information. It is populated and modified by the iCanScript editors.

The *iCS_ Behaviour* is dynamically created from the visual script data and includes the source code for the message handlers of the visual script. It uses the execution services of the iCanScript engine library (i.e. iCanScriptEngine.dll) to manage the execution flow and resolve data contention (also known as deadlocks).

> **Note:** You must delete the *iCS_ VisualScript* component to remove the visual script from the game object. The *iCS_ Behaviour* component will be re-created if you delete it while the *iCS_ VisualScript* is installed.

Chapter 17

Libraries & Prefabs

Visual script libraries are created using Unity's Prefab concept.

Part V

How To

Part VI

Example Project

Chapter 18

Objective

In this section, we examine the core constructs of iCanScript with the help of a small example. To that end, we shall first establish the working parameters of our example by creating a storyline and setting up the scene before implementing the visual scripts.

The following is a summary of the steps to build our first example:

1. Create a storyline (chapter 19) to define the working parameters for our example;
2. Identify and define the actors (chapter 20) for our story;
3. Build a scene in Unity populated with the defined actors (chapter 20);
4. Design the visual script for moving Mr Cube (chapter 23);
5. Design the visual script for enabling & disabling Ms Light (chapter 24);
6. Complete the example by adding a Panic State (chapter 29) for Mr Cube.

Chapter 19

Storyline

If you are like me, you prefer learning the workings of a tool in the context of a small example. Like all good game designers, I like to give a soul to my project using a storyline. We can then extract working scenarios to direct the creation of our visual scripts.

Here goes the story:

> In the land of Emptiness lives Mr Cube. Mr Cube has a busy life roaming around on an invisible 2D plane. Since Mr Cube is a close relative of Mr Fly, he is attracted to Ms Light that also lives in the land of Emptiness. Little does Mr Cube know is that Ms Light is shy and she get "turned off" if anyone comes to close to her. When in the dark, Mr Cube panics and runs around in all directions. Luckily for Mr Cube, Ms Light gets "turned on" when Mr Cube is far enough from her.

(Wow! we got romance, terror and deception. What a great plot!)

Chapter 20

Our Actors

Before creating visual scripts, we first need to build a scene in Unity and populate it with actors. Based on the storyline, the actors are (the camera has been purposely omitted):

- **Mr. Cube** (I'll let you guess the shape to use);
- **Ms. Light** (a directional light);
- **Near Trigger Zone** to turn off *Ms. Light* (a sphere will do fine);
- **Far Trigger Zone** to turn on *Ms. Light* (again a sphere will do fine).

The main characteristics of the actors are:

- All actors live on a 2D plane with the Y-axes set to zero (0);
- Ms. Light and both Trigger Zones are centred at (0,0,0);
- Both Trigger Zones are configured to trigger when colliding with *Rigid Bodies*;
- The Near Trigger Zone will be 2 meters in diameter;
- The Far Trigger Zone will be 3 meters in diameter;
- Mr. Cube is one meter in dimensions;
- Mr. Cube includes a *Rigid Body* component to generate collision triggers.

You may ask yourself why two trigger zones? We want to avoid *Ms Light* from rapidly transitioning on and off while *Mr Cube* nears her. If only one trigger zone existed, *Mr Cube* could navigate on the edge of the trigger zone causing *Ms Light* to flicker. By creating a buffer area between the on trigger (near) and the off trigger (far), we remove the possibility of *Ms Light* flickering.

Chapter 21

Creating the Unity Project

We are now ready to build our scene in Unity. First, we need a fresh new Unity project with an installed version of iCanScript. For the purpose of this tutorial, the project will be named "Houpi Youpi" (I don't know how to call it! So please bear with me...).

Open Unity and select the ***File->New Project...*** menu item as shown in the image below:

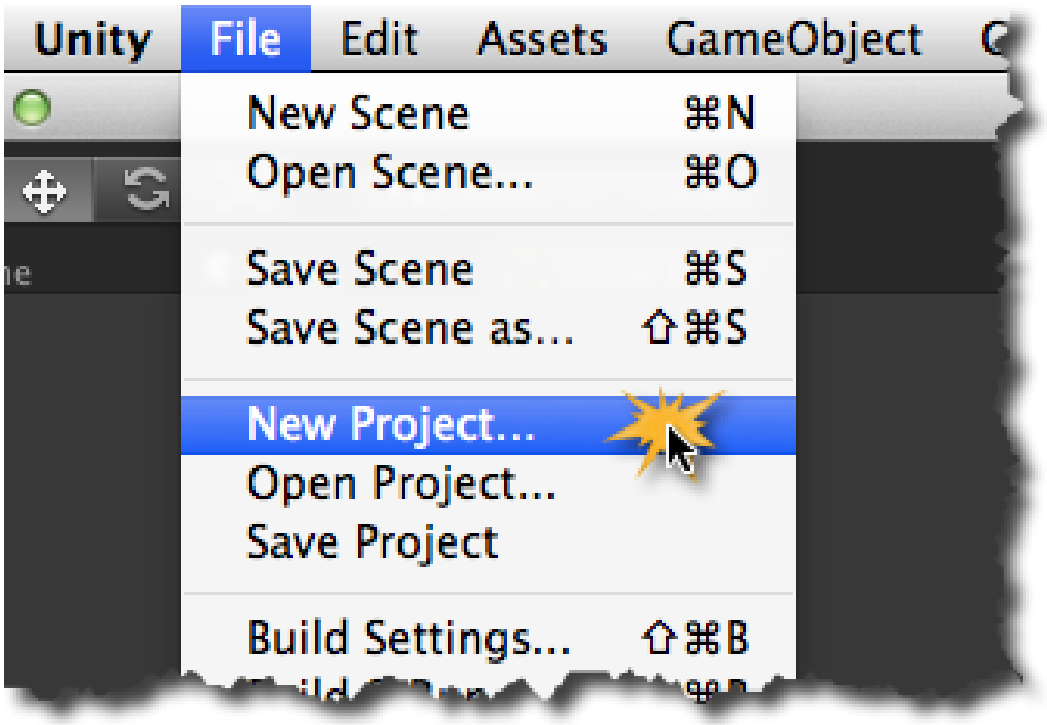


Figure 21.1: Create a new Unity project.

Next you will be asked for a project name and prompted to install standard Unity pack-

ages. Enter *"HoupiYoupi"* as the project name. None of the Unity packages will be required for this tutorial. So don't select any and close the dialog box by clicking the ***Create Project*** button.

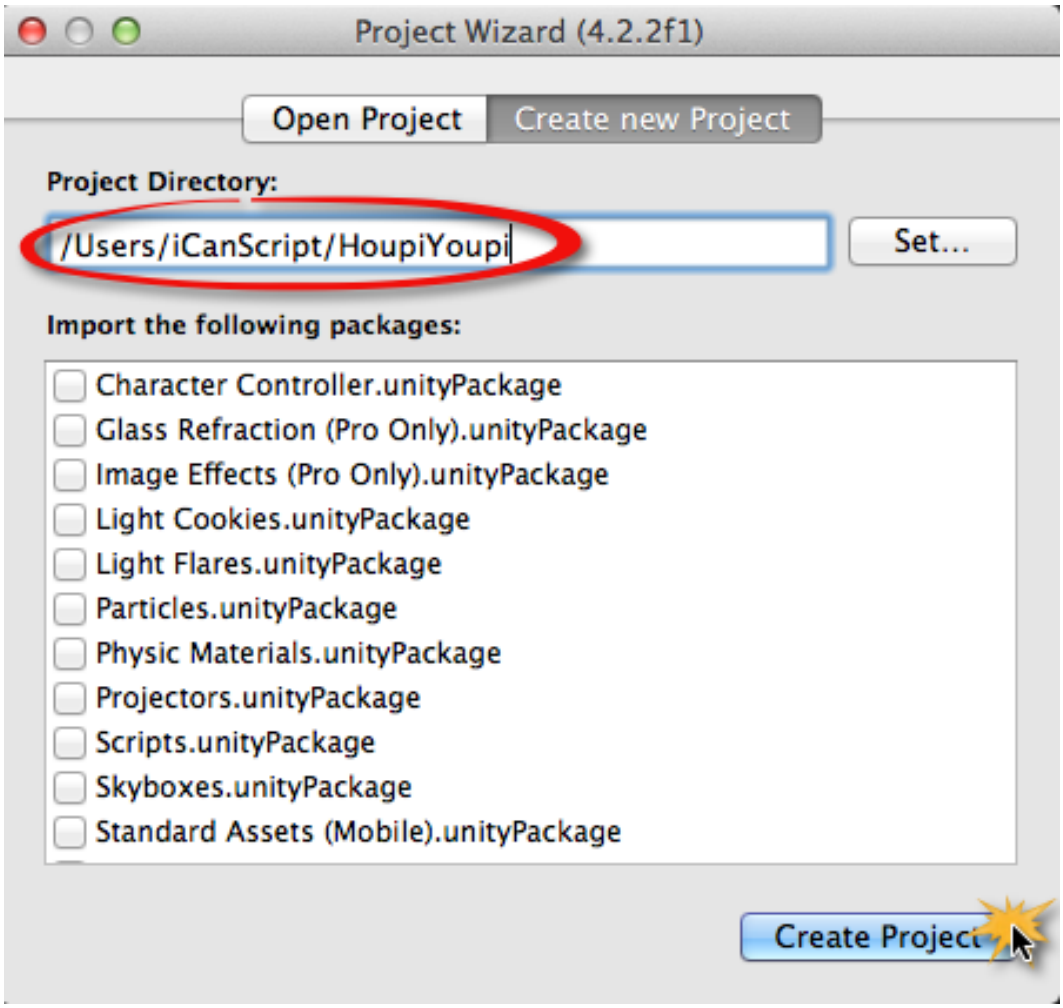


Figure 21.2: Create *Houpi Youpi* Project.

You now have a bare project in which to import iCanScript. Please follow the instructions in the installation (chapter 5) section to download the latest version of iCanScript if you haven't already done so. Use the ***Assets->Import Package->Custom Package...*** menu item to import iCanScript into your project. Once you complete the import, your project panel in Unity should look as follows:

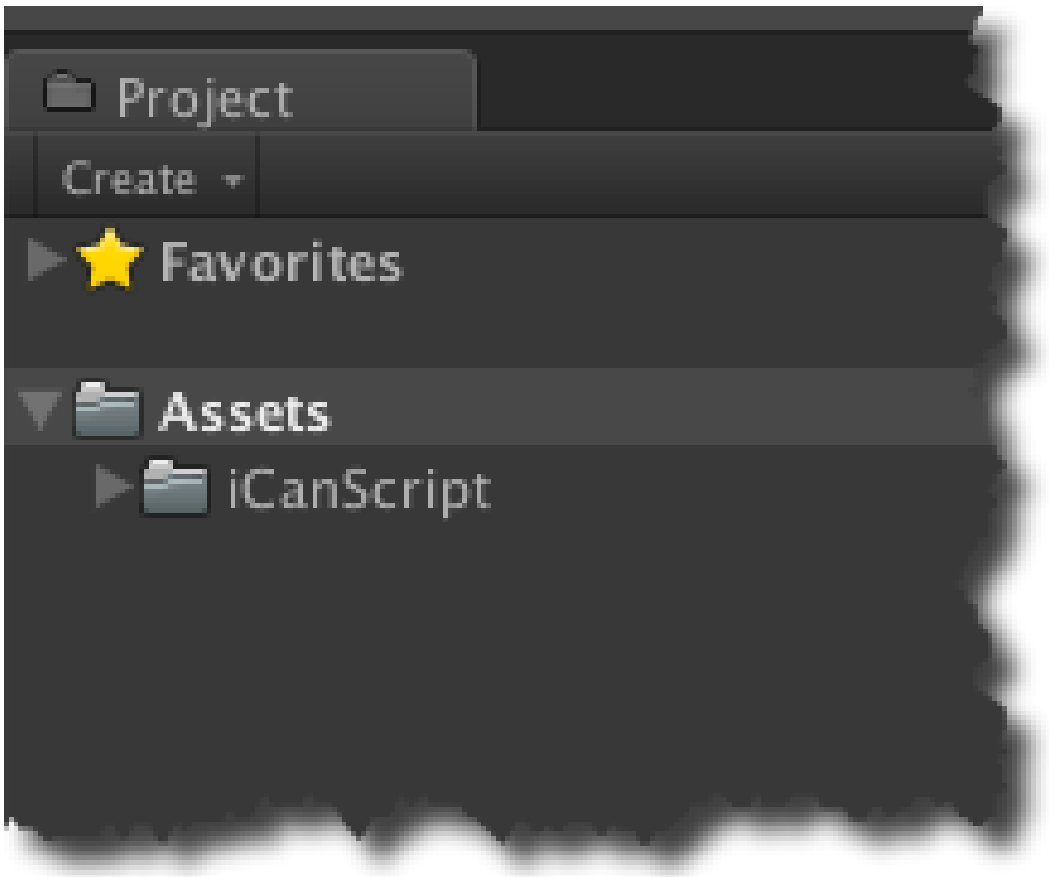


Figure 21.3: Unity project with the iCanScript package.

21.1 Adding Actors to the Scene

Ok, we are ready to bring our actors into the scene. For each new project, Unity creates a default scene with the main camera object. You will add the actors to the default scene using the ***GameObject->...*** menu.

Ms Light

Let's start by adding *Ms Light*. Select the ***GameObject->Create Other->Directional Light*** menu item and, *swoosh ... Ms Light* appears.

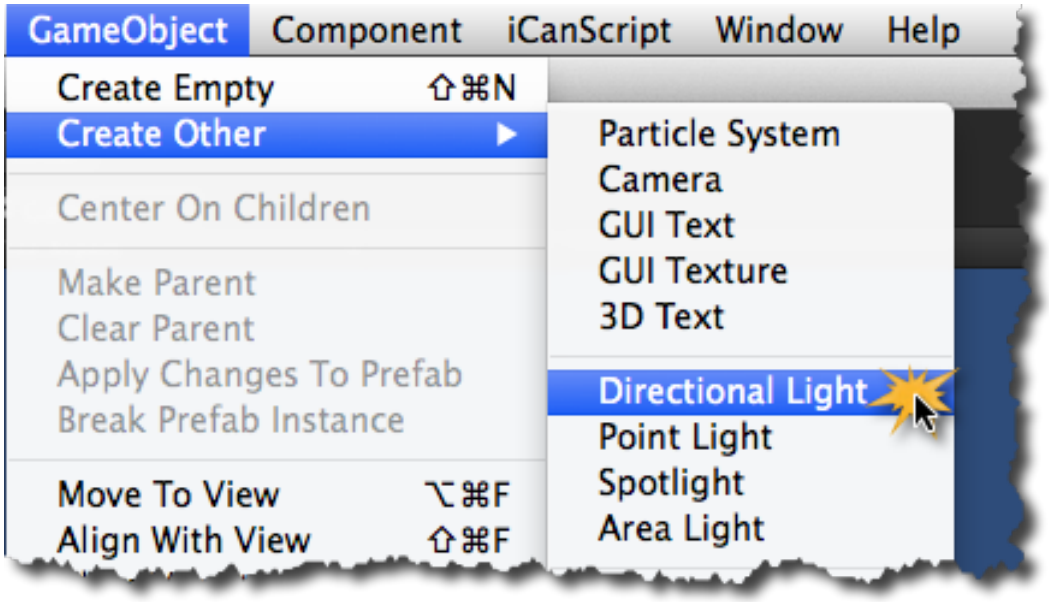


Figure 21.4: Adding Ms Light to the scene.

Double click on the *"Directional light"* label in the hierarchy panel to rename it to *"Ms Light"*. Because *Ms Light* is a directional light, she lights the scene equally regardless of her position. For consistency shake with our story, I suggest you position her at (0,0,0) as show in Figure 21.5.

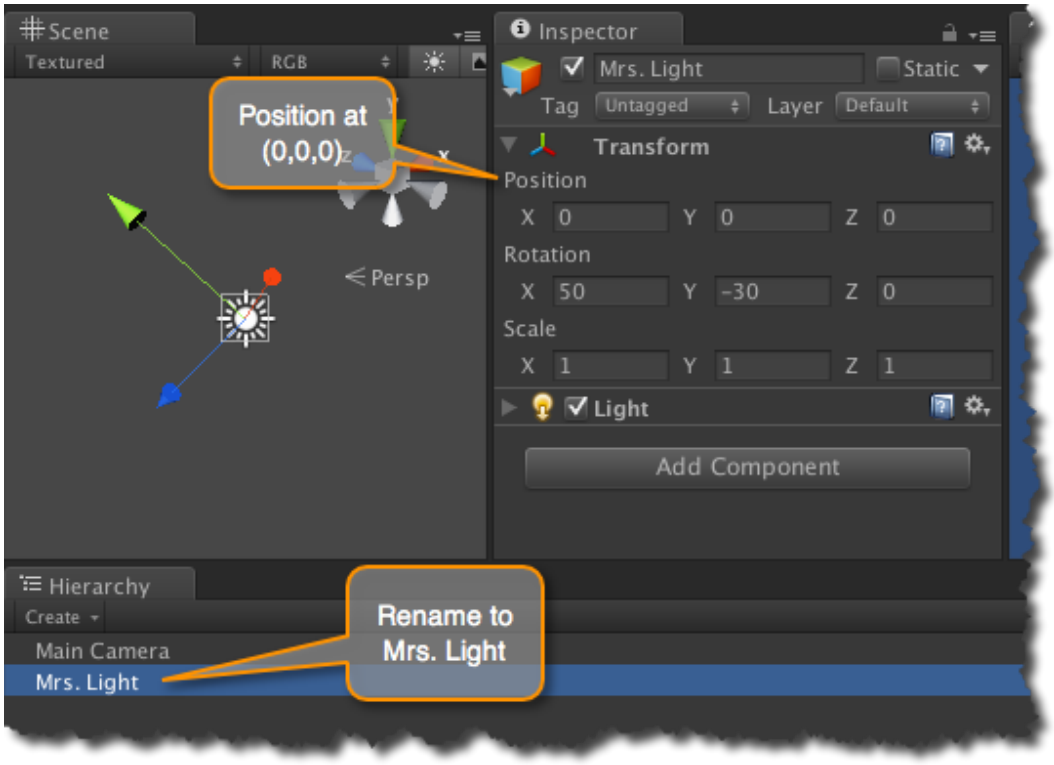


Figure 21.5: Renaming and positioning Ms Light.

Unity Tips

What's a Directional Light?

A **directional light** illuminates the scene with uniform intensity regardless of the distance from the light source. It is often used to simulate the sun. Its light direction and color can be configured to create various time-of-day effects and neat extra-terrestrial atmospheres. The default color and direction of the directional light source is adequate for our example.

Near & Far Trigger Zones

The two trigger zones are spheric in shape and are create with the ***GameObject->Create Other->Sphere*** menu.

After you have created both *trigger zones*, you need to configure them as follows:

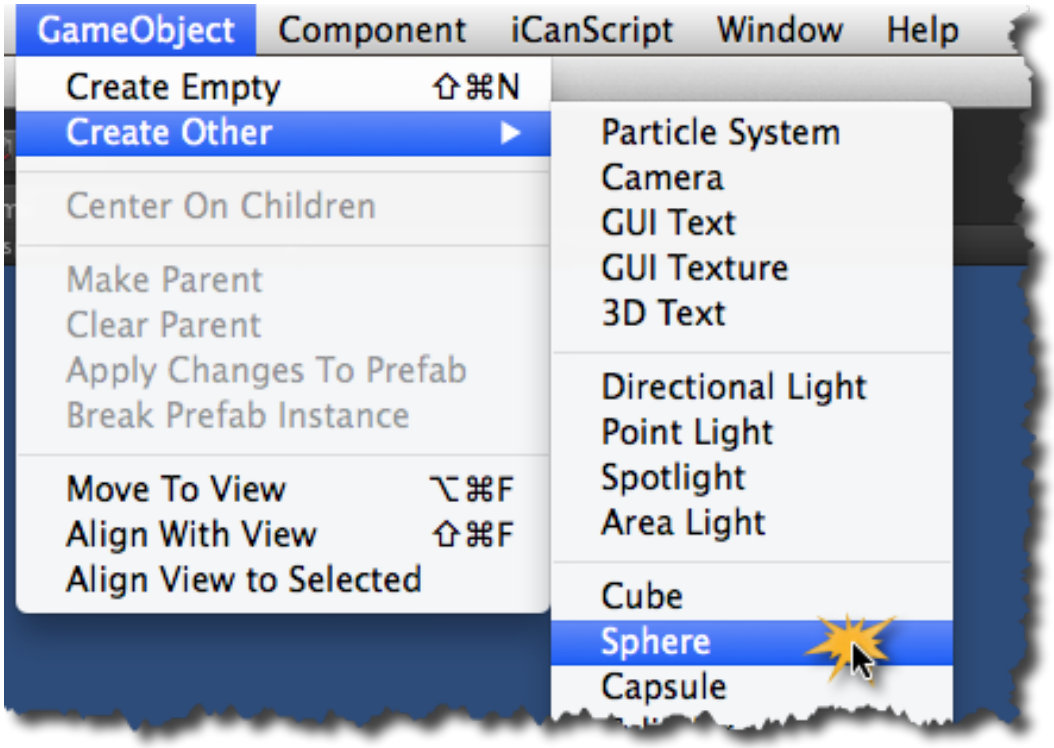


Figure 21.6: Adding Trigger Zones to the Scene.

1. Name one of them *"Near Trigger Zone"* and the other *"Far Trigger Zone"*;
2. As for *Ms Light*, position both trigger zones at (0,0,0);
3. Now resize the *"Near Trigger Zone"* to 2 meters and the *"Far Trigger Zone"* to 3 meters by setting the scale to (2,2,2) and (3,3,3) respectively;
4. The final step is to enable the **trigger** behaviour of both spheres. This is realized by clicking the ***Is Trigger*** checkbox in the *Sphere Collider* component of the trigger zones (see 21.7).

Mr Cube

We are now ready for our final actor: *Mr Cube*. Use the ***GameObject->Create Other->Cube*** menu item to add *Mr Cube* to the scene.

Once more, you need to set the initial parameters of the new actor. *Mr Cube* must be configured as follows:

1. Rename the cube object to *"Mr Cube"* (sexy!);
2. Relocate *Mr Cube* outside both trigger zones at position (5,0,0);

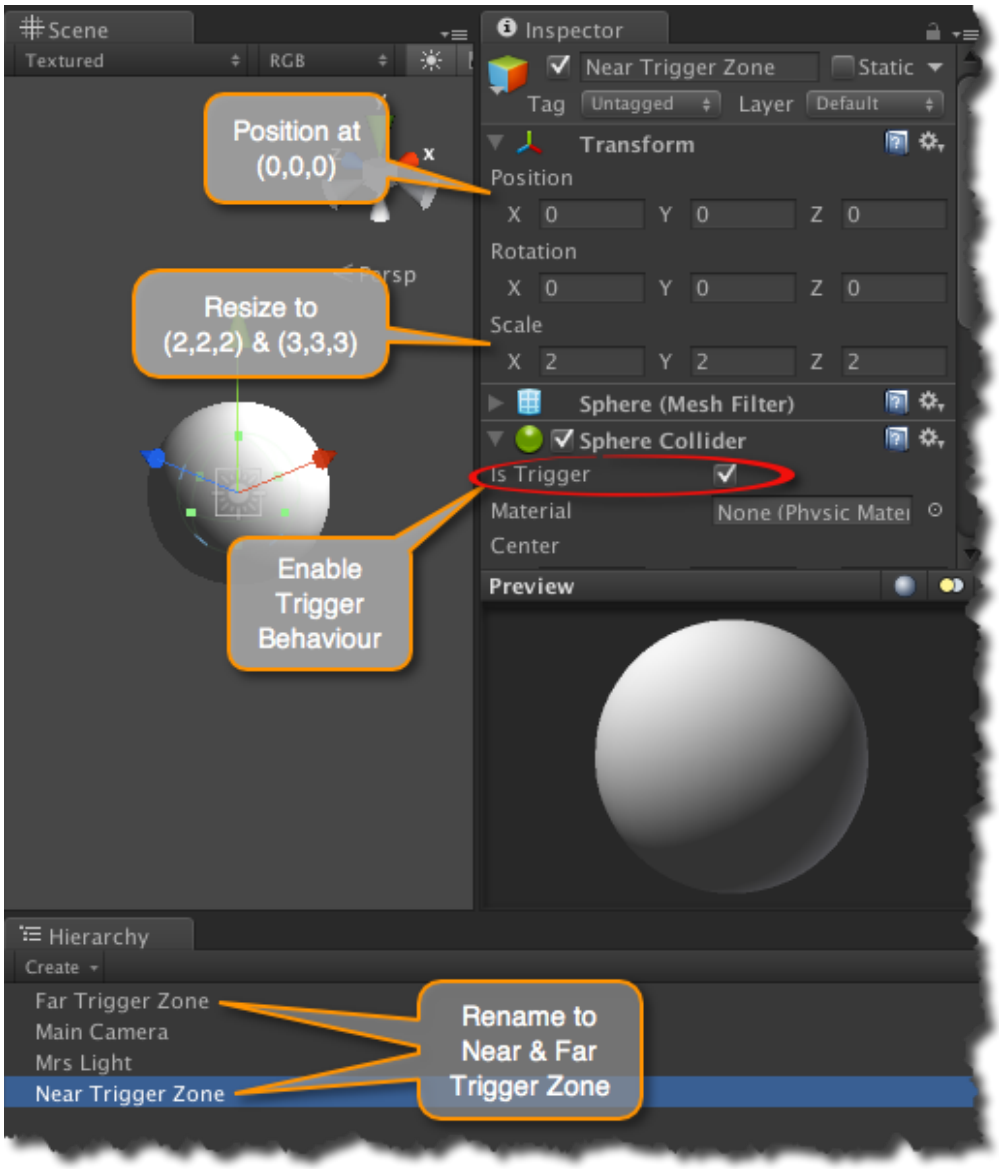


Figure 21.7: Configuring Near & Far Trigger Zones.

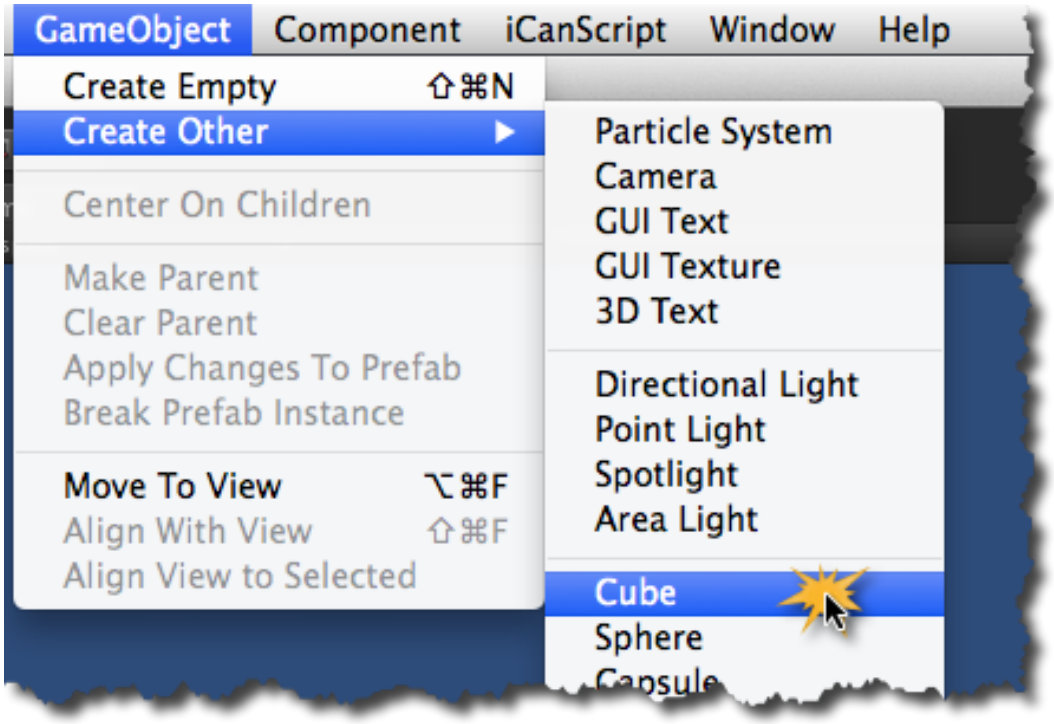


Figure 21.8: Adding Mr Cube to the scene.

3. For *Mr Cube* to collide with the trigger zones, it must have a *Rigid Body*. Add the *Rigid Body* to *Mr Cube* using the ***Component->Physics->RigidBody*** menu item (see figure 21.10);

4. The *Rigid Body* component is configured to use gravity by default. Our land of *Emptiness* is deprived of this *Newtonian* concept and therefore requires that you remove all gravity behaviour. Select the *Rigid Body* component of *Mr Cube* and uncheck the ***Use Gravity*** checkbox as illustrated in figure 21.11.

21.2 Hiding the Trigger Zones

All of our actors are now included in the scene. However, we still need to make one last adjustment: hide the trigger zones.

The trigger zones are the *"magic"* that makes actors change behaviour and everyone knows that *"magic"* must be invisible. So, let's hide those spheres for no one to see. To hide the trigger zones, you need to disable the *Mesh Renderer* as depicted in the below figure.

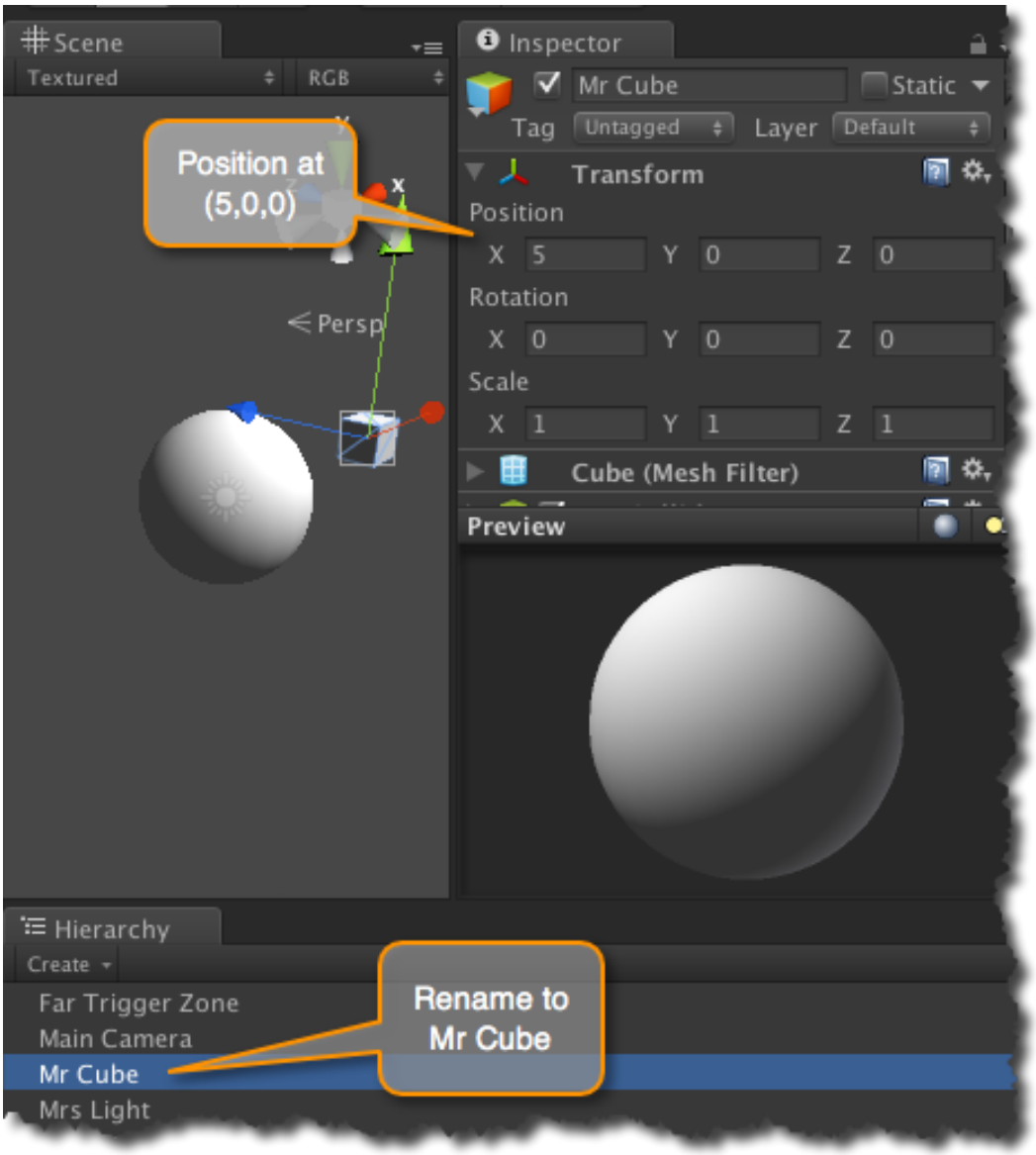


Figure 21.9: Adjusting Name and Position of Mr Cube.

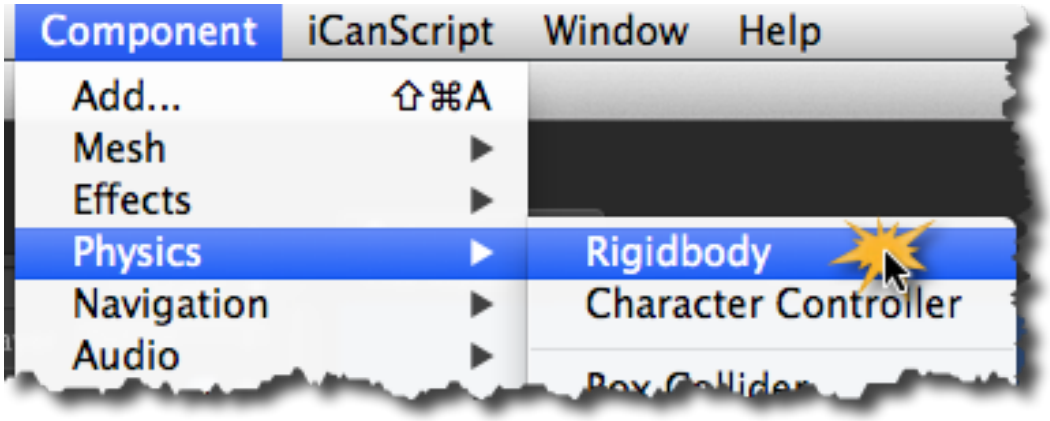


Figure 21.10: Adding a Rigid Body to Mr Cube.

Unity Tips

What's a Renderer?

The **renderer** is the Unity entity responsible for drawing the scene objects. Each object in a Unity scene includes a renderer component to control its drawing properties. By disabling the **Mesh Renderer**, you are telling Unity to forgo the drawing of the trigger zones hence hiding them from the user. Hiding the trigger zones does not change the behaviour of other components such as collision detection that is dear to us.

You are now ready to build your visual scripts!!!

(finally you must be saying to yourself).

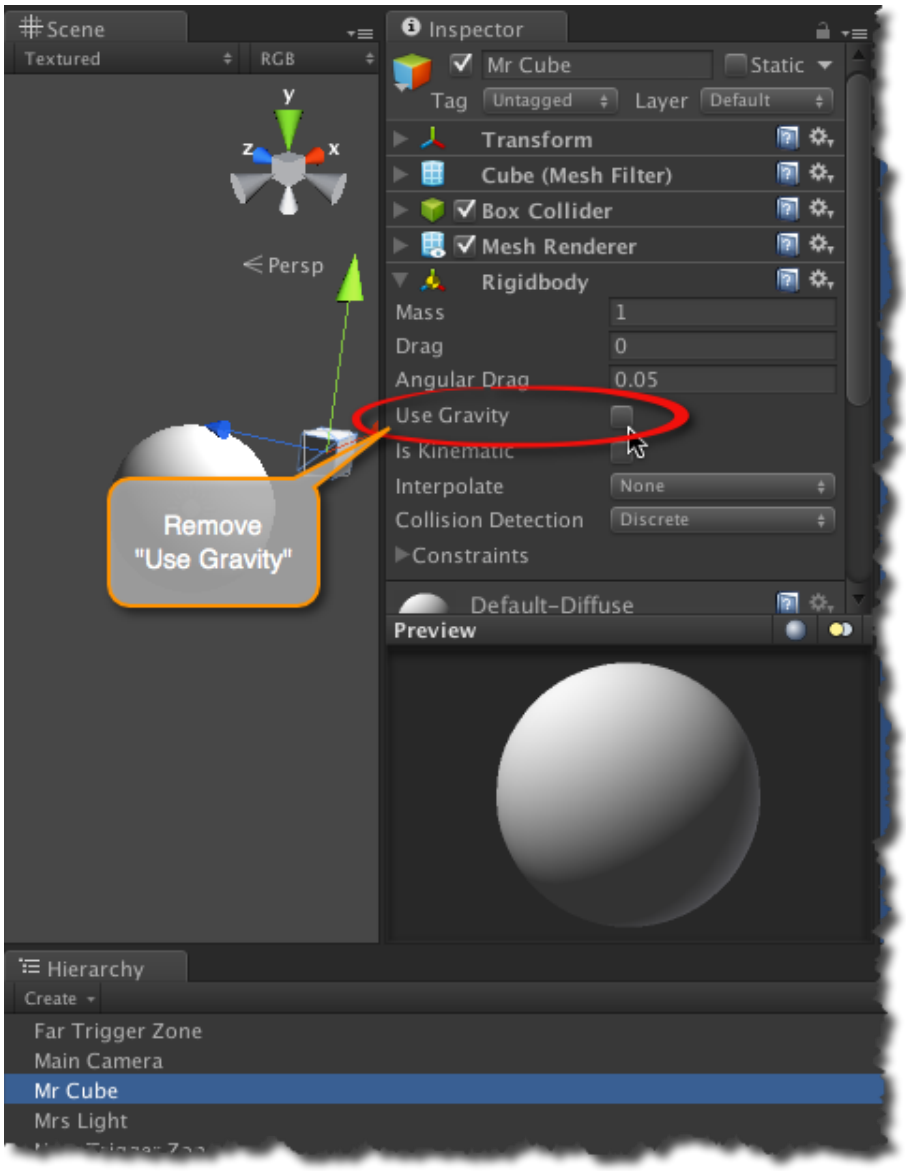


Figure 21.11: Removing gravity from Mr Cube.

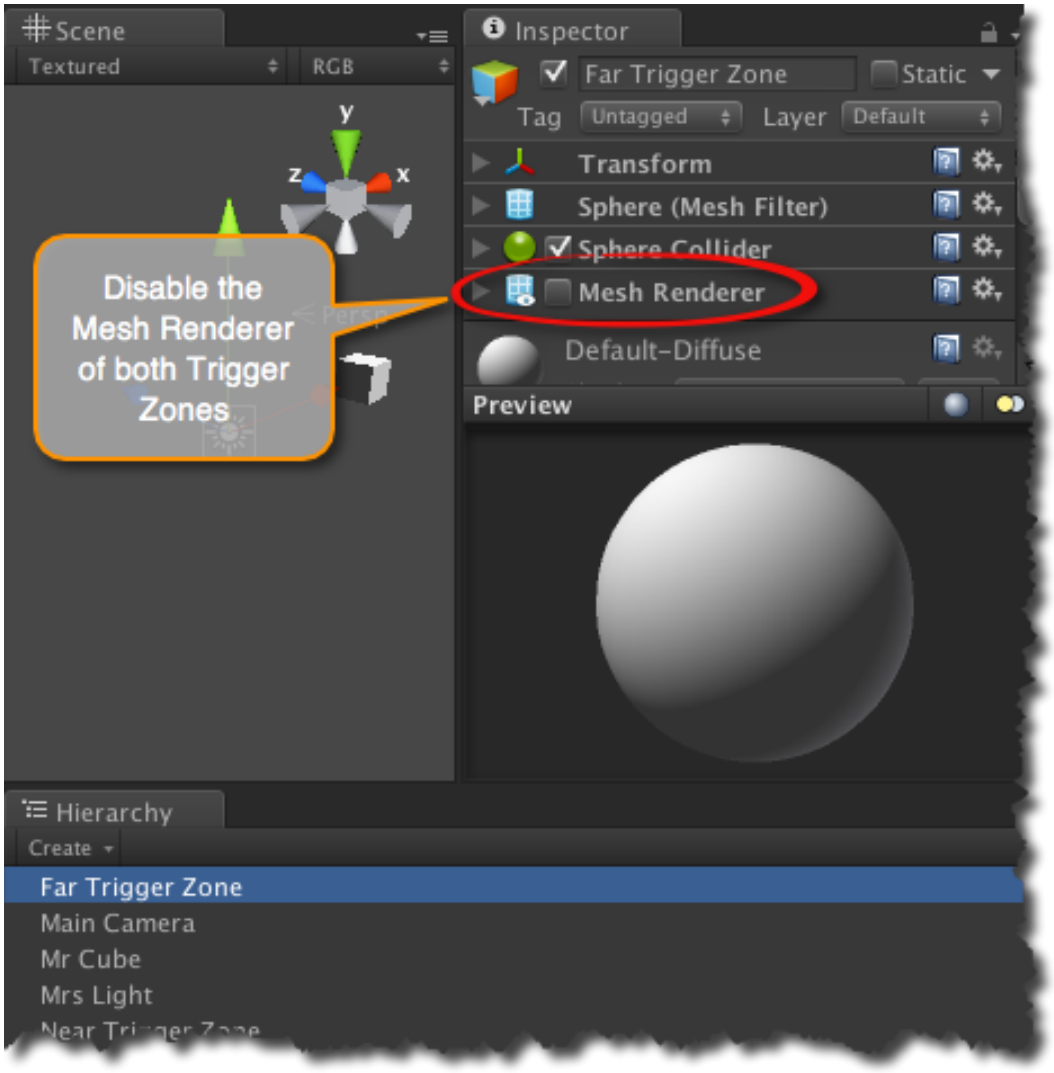


Figure 21.12: Disable the Mesh Renderer of both Trigger Zones.

Chapter 22

Opening the iCanScript Editors

You are almost ready to create your first visual script. Before you do so, you need to open the following four iCanScript editors:

- Visual Editor;
- Library Tree;
- Hierarchy Tree and;
- Instance Wizard.

All of the editors can be opened for the ***Window->iCanScript->...*** menu item. The *Visual Editor* requires a significant screen area so I propose that you combine it with the *Scene* or *Game* window. All of the other editors are narrow and tall and may be combine with the Unity *Project*, *Hierarchy*, or *Inspector* panels.

My preferred window setup for working with iCanScript is:

- the ***Visual Editor*** is combined with *Unity's Scene*;
- the ***Library Tree*** and ***Hierarchy Tree*** share the bottom section with *Unity's Project* and;
- the ***Instance Wizard*** is combined with *Unity's Inspector*.

Of course the layout is a matter of taste and you should organize it to your licking (a two screen layout is the best!).

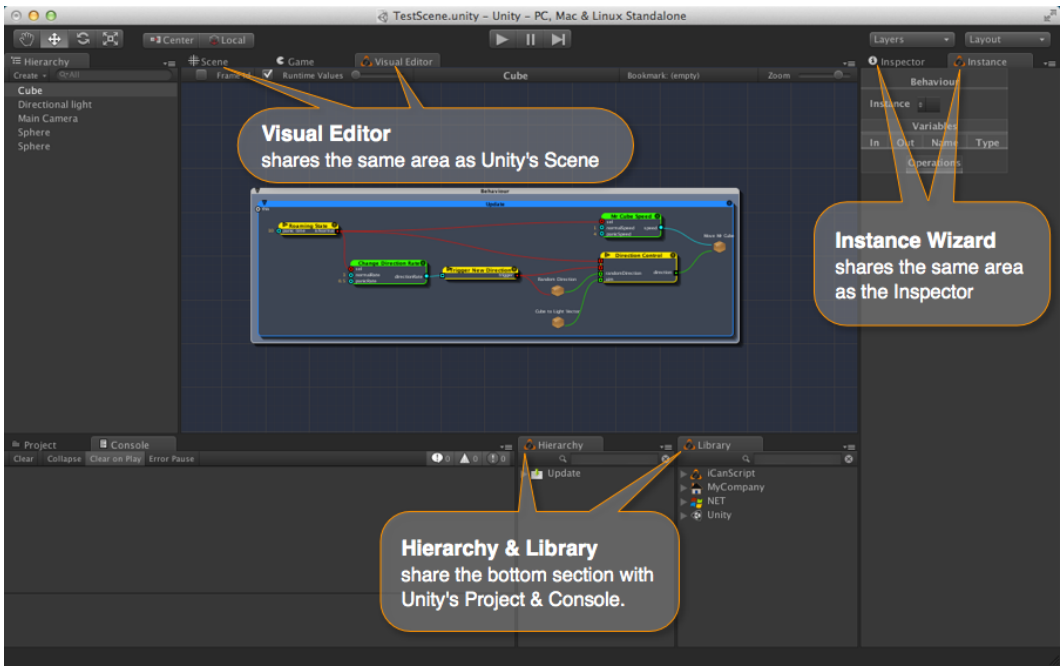


Figure 22.1: Proposed iCanScript Window Layout.

Chapter 23

Moving Mr Cube (step 1)

Finally, you are ready to write your first visual script. *Mr Cube* movement is ratter complex and shall be implement in phases. In this section, you will create a visual script to move *Mr Cube* at a consistent speed regardless of the performance of the computer or game platform running the script.

What you will learn...

In this section you will learn several key aspects of iCanScript including:

- Adding a visual script to a game object;
- Defining message handler nodes to process Unity messages;
- Structuring your visual scripts by encapsulating functionality inside *Package* nodes;
- Designing logic flows and computations using visual scripts;
- Using and binding scene objects into your visual scripts;
- Navigating and extracting entity, variable and function nodes from the iCanScript library.

23.1 Installing a Visual Script on *Mr Cube*

The first step is to install a visual script on *Mr Cube*. This is accomplished by:

1. Selecting *Mr Cube* in the Hierarchy and;
2. Clicking on the menu item ***Component->iCanScript->Visual Script***.

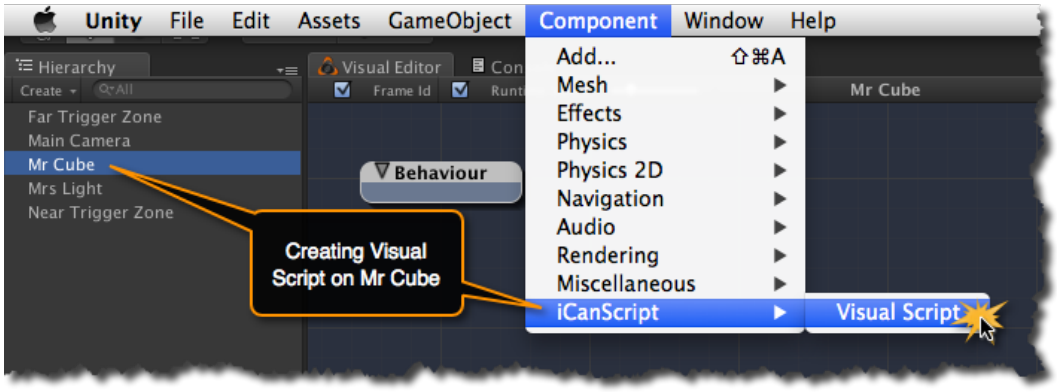


Figure 23.1: Installing a visual script on *Mr Cube*.

A visual script is now installed on *Mr Cube* and ready to receive Unity messages. To edit the visual script, you must select *Mr Cube* (if not already done) and activate the *Visual Editor* by clicking on its tab.

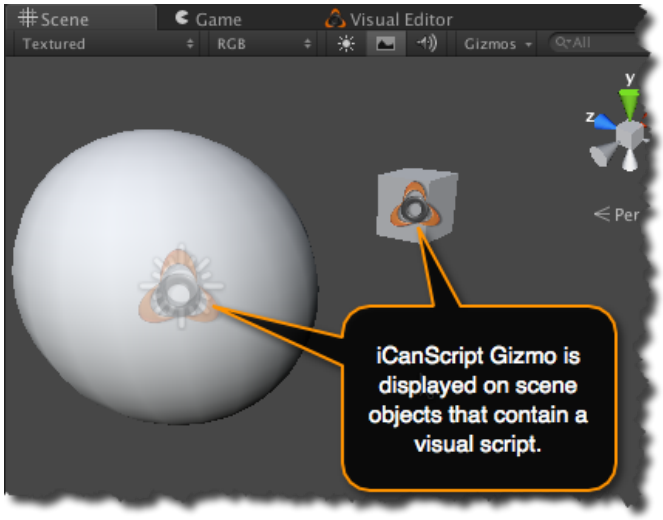
iCanScript User Interface Tips

Visual Editor :: Centring the visual script (*Shift-F*):

The hot key ***Shift-F*** can be used to resize and reposition the visual script in the centre of the viewport. (See the [navigation][] for additional hot keys.)

Scene Editor :: Visualizing which object contains a visual script:

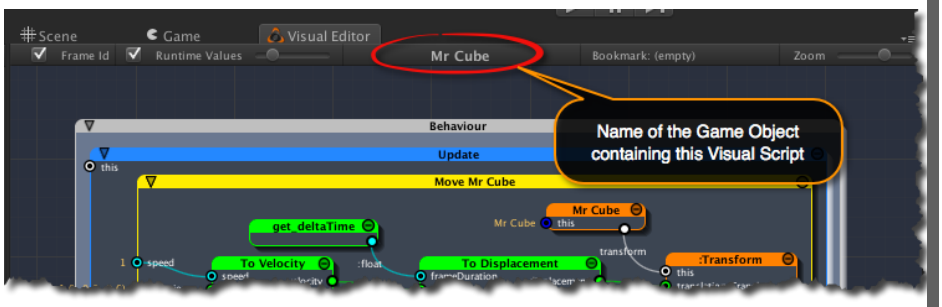
In the Scene editor, iCanScript displays its logo in front of each object that contains a visual script.



Visual Editor :: Selecting the visual script to edit:

The ***Visual Editor*** provides a graphical view of the visual script installed on the selected game object. To edit a visual script, you first need to select the game object that contains the visual script.

The name of the game object containing the visual script is displayed in the toolbar of the *Visual Editor*.



Note: The Visual Editor continues to edit the same visual script (previous selection) if the object selected in the Hierarchy panel does not contain a visual script.

Currently your visual script contains a single grey node named: *Behaviour*. For iCanScript, the *Behaviour* node is special in two ways:

- first, it's the top-level node that orchestrates the execution of all visual scripts;
- secondly, its content is limited to *Message Handler* nodes.

Before furthering our example, let's take a moment to examine some of the core constructs of iCanScript:

iCanScript Core Concepts

iCanScript supports two categories of nodes:
Action nodes:

> ***Action*** nodes execute functionality created outside the realm of iCanScript. Nodes extracted from libraries and handwriting code are good examples of action nodes.
>
> iCanScript does not have the ability to look inside or alter the functionality underlying *Action* nodes. For iCanScript, *Action* nodes are black-boxes that can be inter-connected, conditionally executed, and packaged to create high-level functionality.

Composite nodes:

> ***Composite*** nodes are used to organize, abstract, and control a subset of the visual script. The subset is embedded, using nesting, inside the composite node creating a parent /child relationship.
>
> A *Composite* node activates its child nodes if the conditions for it own activation are met. For example, a message handler node will execute its internal visual script only when it receives the appropriate message.
>
> Composite nodes exist in several flavours each with distinct behaviour and trigger conditions.

iCanScript Core Concepts

Every visual script is composed of one *Behaviour* node and one or more *Message Handler* node(s).

Behaviour Node:

> ***Behaviour*** is a special composite node that coordinates the execution of the overall visual script. It waits for messages sent from Unity's engine to trigger a subset of the visual script.
>
> The ***Behaviour*** node uses ***Message Handler*** nodes to identify which messages the script operates on.
>
> Before creating a visual script, you must first decide which of the Unity messages triggers the execution of your script. A message handler node must be created and embedded in the *Behaviour* node for each message your visual script responds to.

Details pertaining the available Unity messages are documented in the *MonoBehaviour* section of the Unity Script Reference guide[a].

[a] http://docs.unity3d.com/Documentation/ScriptReference/MonoBehaviour.html

Message Handler Nodes:

> A *Message Handler* node is a composite node associated with a specific Unity message. Its purpose is to bridge a Unity message with a subset of your visual script.
>
> When the *Behaviour* node receives a message, it updates the input port values (message parameters) and activates the corresponding *Message Handler* node. The script you create to respond to the message must be embedded in the corresponding *Message Handler* node.

23.2 Installing the *Update* Message Handler

Before displaying a new frame, Unity sends an *Update* message to each game object in the scene asking them to prepare for the upcoming frame. This *Update* message is ideal to trigger the execution of the visual script to move *Mr Cube*. It gives you the chance to recompute *Mr Cube* position immediately before the scene is displayed.

Your next step is to install the *Update* message handler node whom will serve as the parent trigger for your visual script. The creation of the *Update* message handler node is

realized as follows:

1- Right click on the *Behaviour* node to reveal the list of message handlers it supports;
2- Click the + ***Update*** menu item to install the message handler node;

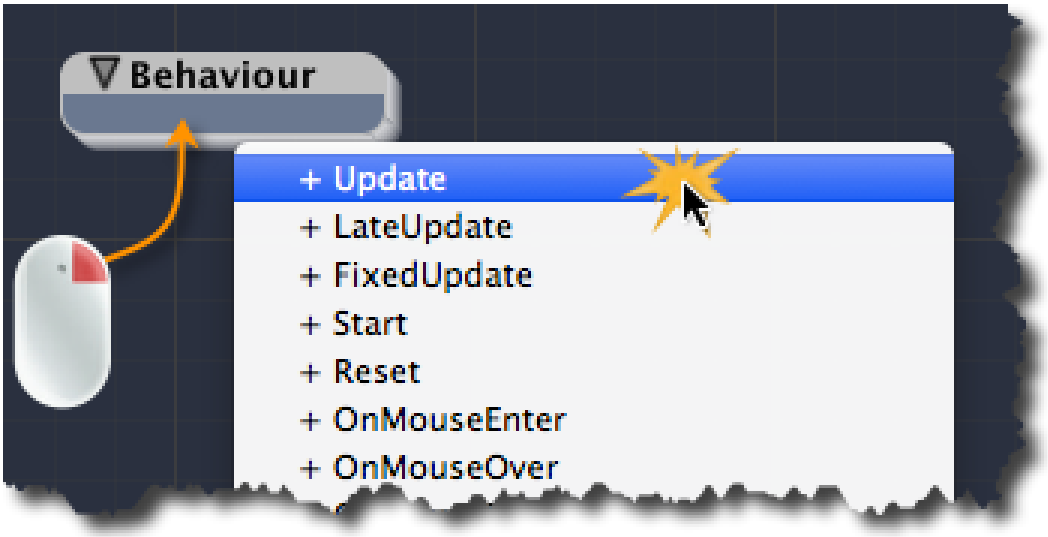


Figure 23.2: Installing the Update message handler.

A new child node called *Update* is create inside the *Behaviour* node. The *Update* node is blue indicating that it is a message handler. You are now ready to create the visual script to move *Mr Cube* inside the *Update* node.

23.3 Visual Script Overview

Mr Cube visual script will evolve to become somewhat involved as you increase its functionality. To avoid complex and overblown graphs, you should consider structuring the visual script from its inception.

The following diagram depicts the high-level design of the visual script you will create for Mr Cube:

It's good practice to segregate and encapsulate functionality into tight bundles and iCanScript has the right *Composite* node to do so: (drum roll...) the '***Package***'.

The *Package* is iCanScript most flexible node. It can contain complex graphs and expose only those ports that are made public by the visual script designer. To modify a *Package*, it will need to be ***unfolded***, showing its internal graph. Once you have completed its functionality, you may want to ***fold*** it so that it displays as a singular node or ***iconize*** it to reduce visual clutter.

The following diagram shows the controls to fold/unfold and iconize a *Package* node.

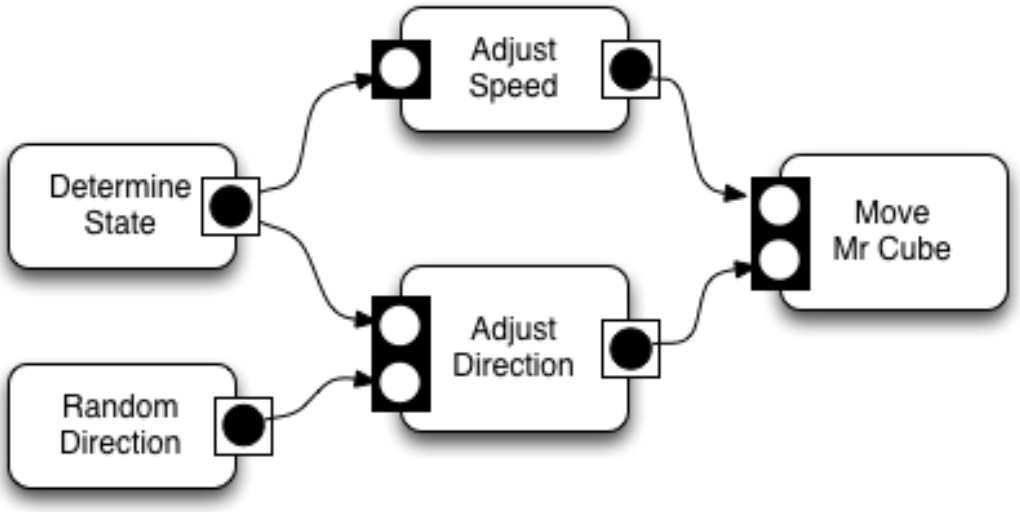


Figure 23.3: *Mr Cube* visual script overview.

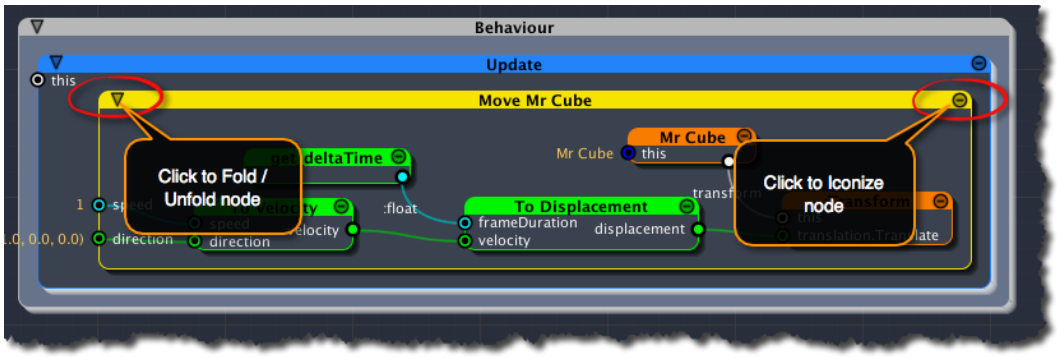


Figure 23.4: Node display state controls.

23.4 Creating the *"Move Mr Cube Package"*

Let's create a package to encapsulate the basic movement functionality of *Mr Cube*. This is achieved by:

- Right clicking on the *Update* node to display the context sensitive menu and;
- Selecting the *+ **Package*** menu item.

iCanScript adds a package node under the *Update* message handler. By default, the package is yellow and named *":Package"*. A name that better describes the purpose of the package would be better suited. Do let's rename your new package to: *"Move Mr Cube"*.

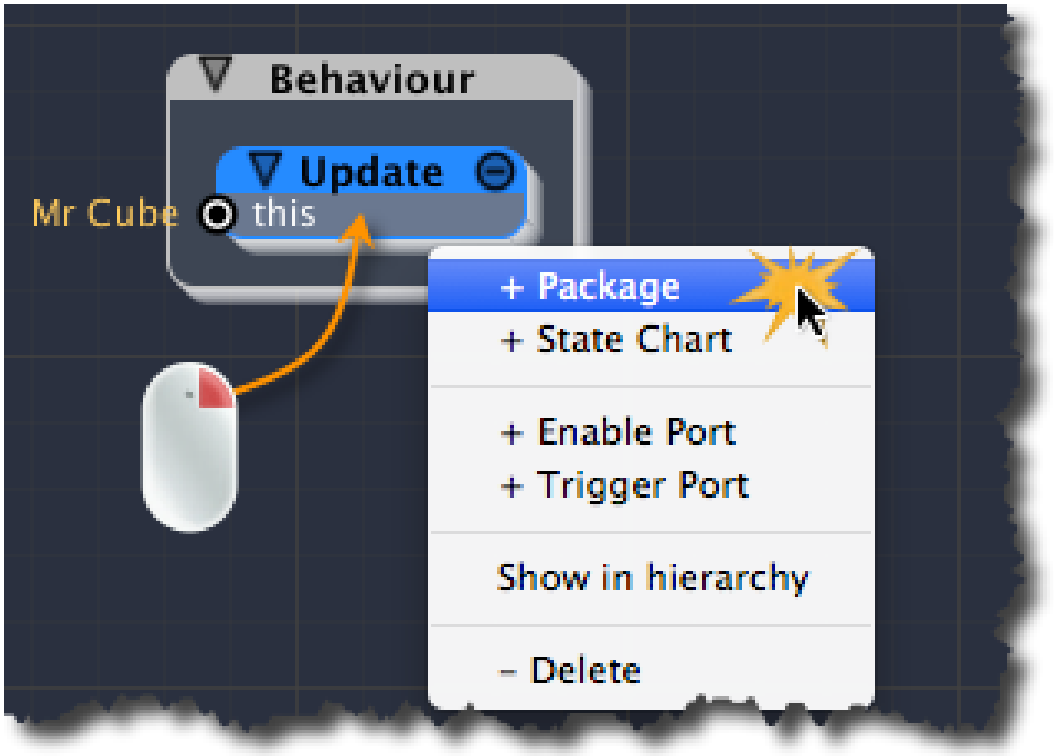


Figure 23.5: Creating the *"Move Mr Cube"* Package.

iCanScript User Interface Tips

Modifying name of visual elements:

Modifying the node or port name is realized either from the *Inspector* or the iCanScript *Hierarchy Tree*.

The *Inspector* grants detail access to the node or port that is selected in the visual script. It is however limited the selected element only and will require re-selecting if more then one element needs to be modified.

The iCanScript *Hierarchy Tree* offers a tree-like navigation of the entire visual script. It also allows for changing visual element names. In this tutorial, we will always use the *Hierarchy Tree* when modifying node and port names.

Let's use the *Hierarchy Tree* to change the name of the package.

From the *Visual Editor*, right click on the *:Package* and select the ***Show in hierarchy*** menu item (figure 23.6). This action displays the selected node in the *Hierarchy Tree*.

Figure 23.6: Show Package node in iCanScript Hierarchy tree.

Double click on the package name in the *Hierarchy Tree* to modify it to *"Move Mr Cube"* (figure 23.7).

Note: The *Visual Editor* and *Hierarchy Tree* are two views into the visual script. This means that modifications done in the *Hierarchy Tree* are reflected in the *Visual Editor* and vice-versa.

23.5 Adding *Mr Cube* to the Visual Script

It's now time to put beef into your hamburger! The first ingredient you need is *Mr Cube*. He can be brought into the visual script by dragging him from Unity's Hierarchy panel into the *Move Mr Cube* package (see figure 23.8).

The new *Mr Cube* node is coloured orange indicating that it is an instance of an entity. Factually, *Mr Cube* is an instance of a *Game Object*: a complex entity that contains several components defining its capabilities.

Figure 23.7: Renaming the package to *Move Mr Cube*.

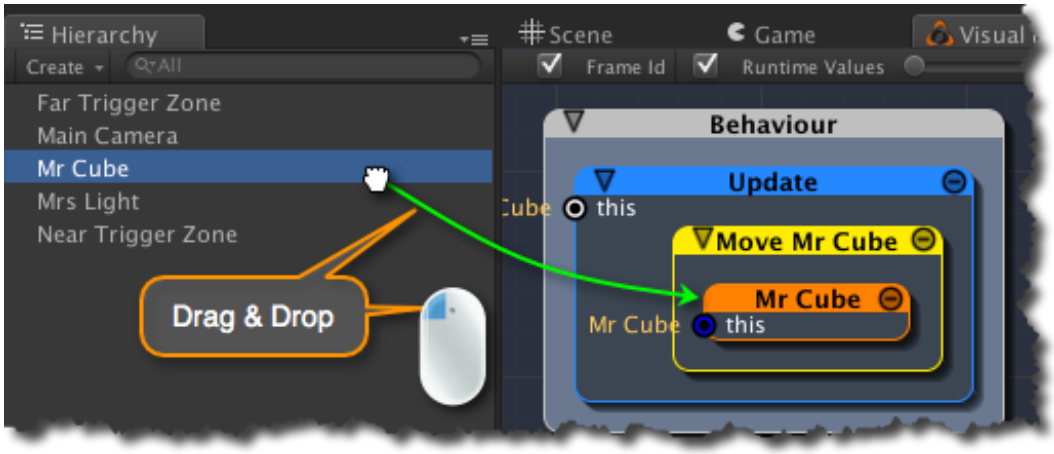


Figure 23.8: Adding Mr Cube to the *Move Mr Cube* package.

iCanScript User Interface Tips

Instance Nodes and the *Instance Wizard* iCanScript includes a dedicated editor called the *Instance Wizard* to manage the variables and operations associated with nodes representing object instances. The *Instance Wizard* is automatically brought forward when such a node is selected.
Using a simple point-and-click interface, the *Instance Wizard* can expose or hide the internal variables and operations of the instance. It is divided in three (3) sections:
Instance (top section):

> The top section consists of a drop down menu that lists the available functions to create an instance of the proper node type and bind it to the *'this'* port. Since we have dragged in *Mr Cube*, iCanScript has assumed that the instance should be set to the *Mr Cube* game object.

Variables (middle section):

> The middle section displays all of the variables (fields & properties) of the instance. You may expose those variables as input and/or output ports by modifying the checkbox on the left of the variable name.

Operations (bottom section):

> The bottom section gives access to the operations available for the instance as a list of buttons. Clicking on the operation name exposes that operation on the instance node. Clicking again removes the operation.

23.6 Exposing the *Transform* of *Mr Cube*

In Unity, a game object and its components form a group of inter-related objects. The game object maintains a list of all of its components and each component keeps a reference back to the game object as depicted in figure 23.9.

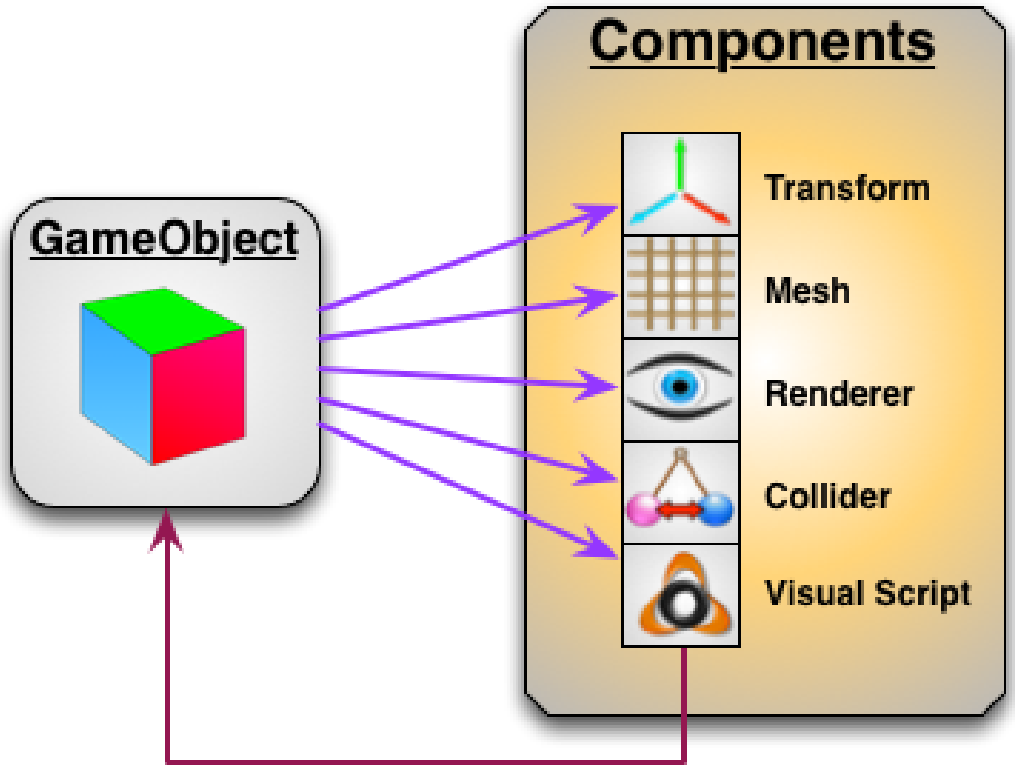


Figure 23.9: Unity *Game Object* and associated *Components*.

To gain access to a component of the game object, you first need to extract it as a separate instance node. You can then interact with the component variables and operations using the *Instance Wizard*.

Unity Tips

The number and type of components that can be attached to a game object is not limited to the listed displayed in figure 1. The Unity library includes a set of standard components and allows for extending the game object functionality with user defined components.

When you create a visual script on a game object, you are in fact extending that game object by attaching an *iCS_ VisualScript* component to it. iCanScript then uses the iCS_VisualScript component to edit, compile and run you visual script.

The component of interest for moving *Mr Cube* is called the *Transform*. It defines the position, rotation, and scale of the game object. Moving *Mr Cube* requires that you change its position hence change its *Transform* component.

Exposing the *Transform* of *Mr Cube* requires that: 1- you reveal the port associated with the *Transform* component and then; 2- extract the transform node to gain access to its variables and operations.

STEP #1: Revealing the *transform* port:

Revealing the *transform* port of *Mr Cube* is a simple matter of: - selecting the *Mr Cube* node to reveal the *Instance Wizard* (figure 23.10); - clicking the checkbox on the left side of the *"transform"* variable in the *Instance Wizard* (you may need to scroll to the bottom of the *Variables* section).

STEP #2: Extracting the *transform* instance node:

The next step is to extract the transform node from *Mr Cube*. The good news is that iCanScript includes a feature that automatically creates an instance node when you drag a port into an empty area.

Since the *Move Mr Cube* package tightly surrounds the *Mr Cube* node, dragging the port into an empty area inside the *Move Mr Cube* node may be challenging (Undo can be used if mistakes are made). In the current situation, the easiest way is to drag the *transform* port up towards the title bar of the *Move Mr Cube* node and release it their (see figure 23.11).

You can later reposition the new transform node by dragging it from its title bar. Likewise, you can relocate the ports by sliding them on the boundaries of the node.

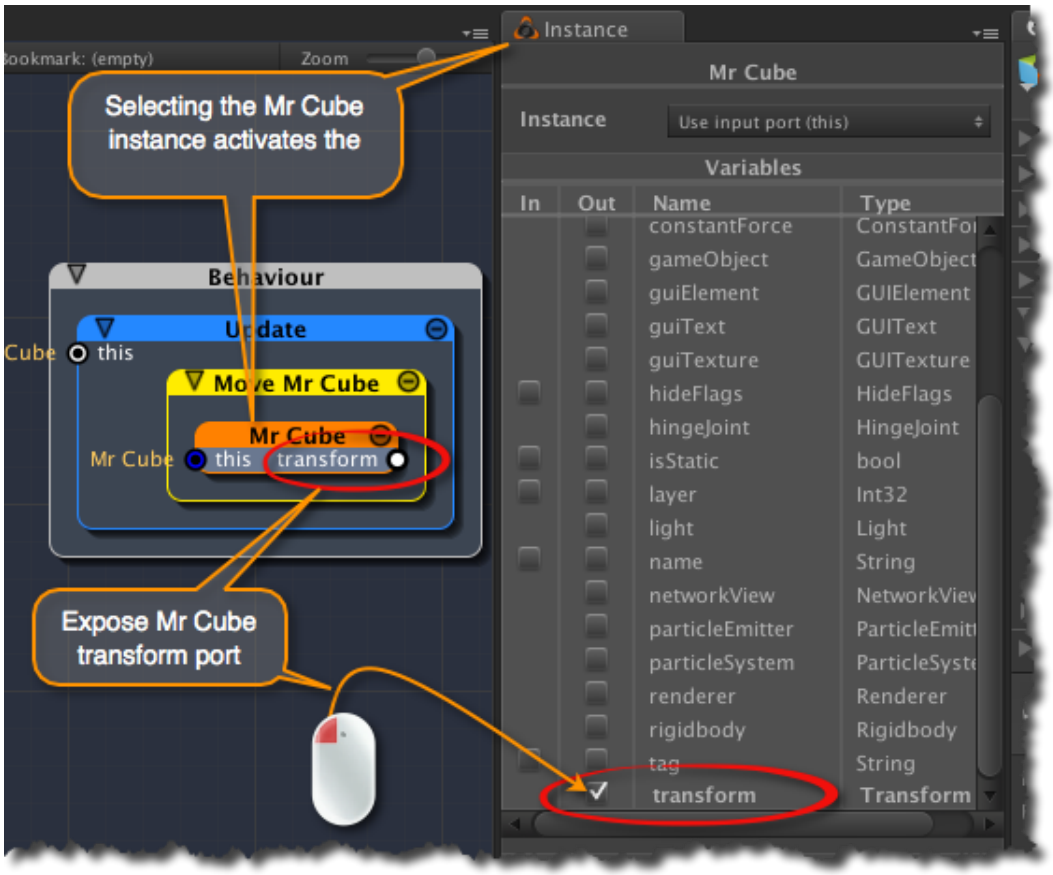


Figure 23.10: Reveal Mr Cube transform port.

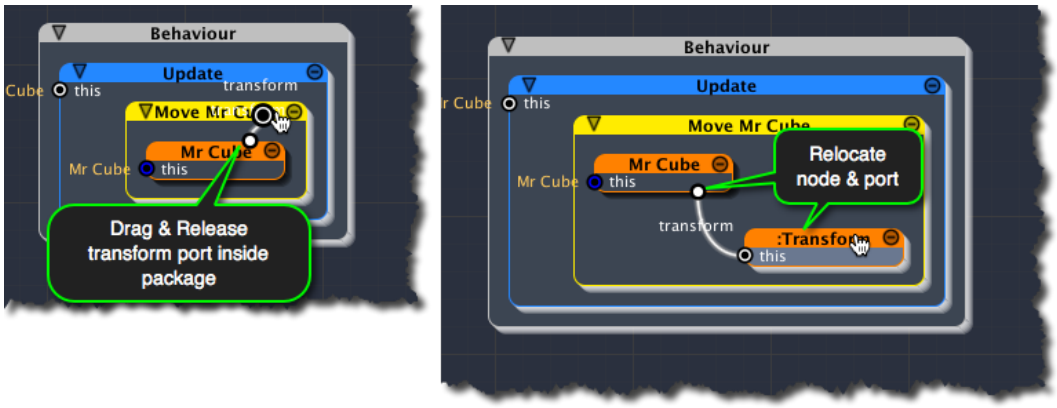


Figure 23.11: Extract Mr Cube transform.

Unity Tips

Game Object Transform:

Every *Game Object* in Unity includes a *Transform* component. The *Transform* component defines the position, rotation, and scale of the object with respect to its parent. If no parent exists, then the *Transform* is considered global – relative to the world coordinates – as for all actors in our example.

23.7 Adding the *Translation* operation to the *Transform* node

Now that you have access to the *Transform* instance node, you can select it and browse its variables and operations using the *Instance Wizard* (figure 23.12).

If you browse the *Operations* section, you will find that the *Transform* node includes several *Translate(. . .)* operations that can be used to move *Mr Cube*. We are interested in the *Translate* operation that utilizes a *Vector3* type for the displacement.

To add the *Translate* operation to your visual script, you will need to depress the ***Translate(translation:Vector3)*** button of the *Instance Wizard*. Once the operation is added, its button is shown as depressed and the text as bolded. Clicking the button a second time will remove the operation from the visual script. Make certain that the *Translate* operation is added before moving on.

With the *Translate* operation added, you'll notice a new port named *"translation.Translate"* appearing on the *Transform* node. As you may have guessed, this new input port configures the translation to be applied by the *Translate* operation.

The default value for the translation is (0,0,0). You will learn to create a visual script to dynamically change the translation value in the sections to follow.

23.8 Moving with Consistent Velocity

Your next task is to feed a displacement value to the *Translate* operation of *Mr Cube*. While this seams easy, it does present a challenge. . .

The Challenge: The problem is that the displacement is applied for each frame and that the frame rate (# of frames /second) is dependent on the performance of the computer or gaming platform. This means that if the same displacement value is applied, *Mr Cube* will move lightning fast on high-end gaming computers and turtle slow on entry level portable devices.

The Solution: To maintain constant velocity on all platform regardless of their performance, you need to adjust the magnitude of the displacement according to the frame rate. So how do you do that?

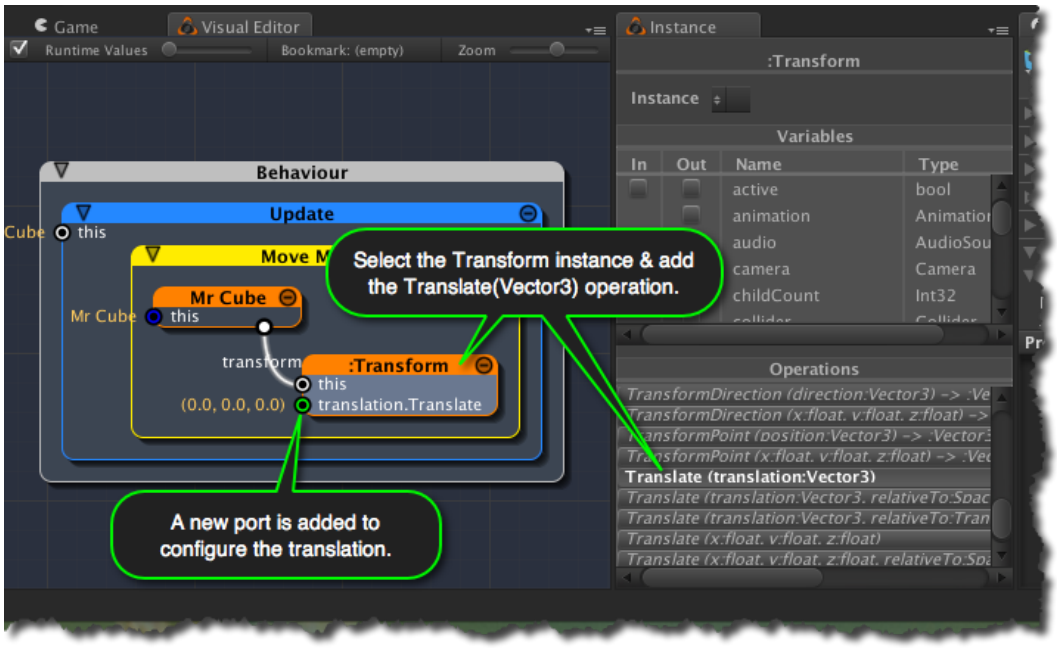


Figure 23.12: Add Mr Cube translation operation.

As starters, you need to control the velocity of *Mr Cube* and derive the displacement from it. The displacement is computed by equation #1:

Eq. #1. Calculation of the displacement for a frame

```
displacement= velocity * frame_duration;
```

The result of this equation is that higher the frame rate, smaller are the displacement values added to the movement for each frame. Therefore, the velocity is maintained irrespectively of the computing performance.

From equation 1, you need to bring in play an operator that multiplies the velocity (a *Vector3* value) and the frame duration (a *scalar* value).

Using the search field in the iCanScript library, type the characters *"mul"* to reveal the available multiply operators. Drag the multiply operation under the *Vector3* type into the *Move Mr Cube* package as depicted in figure 23.13.

The *"op_ Multiply"* node is coloured green indicating that it is a function. Functions are created outside iCanScript and imported into the iCanScript library. In this particular case, the multiply operator is a member function of the *Vector3* class that is part of the Unity Engine library (the hierarchy in the *Library Tree* is reminiscence of this structure).

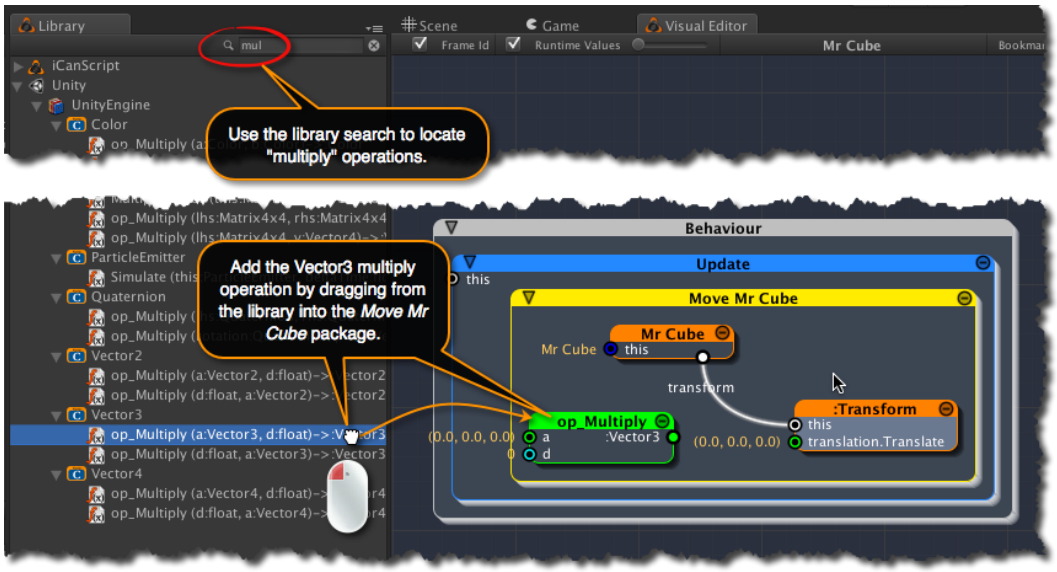


Figure 23.13: Adding a function to convert velocity into displacement.

The new node is created with generic names. To improve on the clarity of the visual script, you should rename the node and its ports to reflect the intended functionality.

Follow these steps to rename the *op_Multiply* node:

1. Right click on the *"op_Multiply"* node to bring up the context menu;
2. Select the *Show in Hierarchy* menu item;
3. From the hierarchy tree, unfold the node and rename it and its ports as depicted in figure 23.14.

iCanScript User Interface Tips

You need to double click on the name in the hierarchy tree to edit it.

The next step is to bind the *displacement* output port of the *To Displacement* node to the *translation* input port on the *Transform* node. This is super simple to accomplish: just drag the *"displacement"* port onto the *"translation.Translate"* port and ... Voilà! it's done. Now the computed displacement changes the position of *Mr Cube* on every frame.

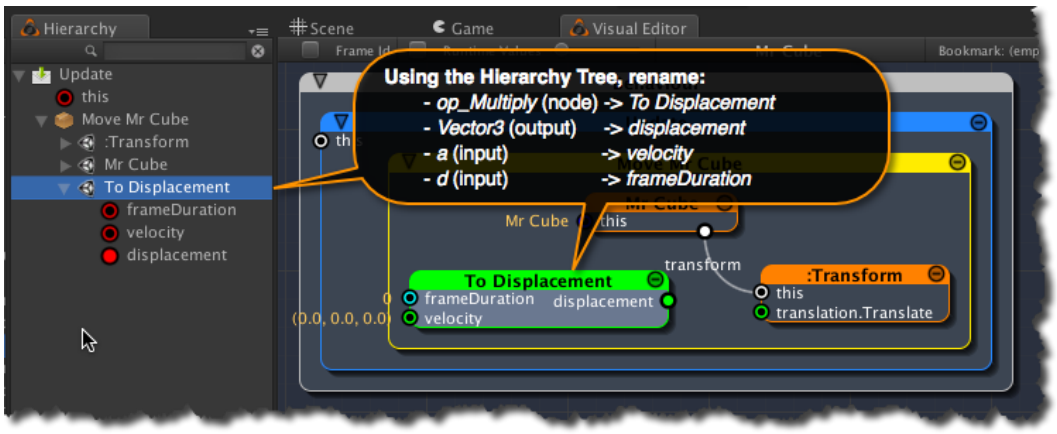


Figure 23.14: Renaming the node that converts the velocity to a displacement.

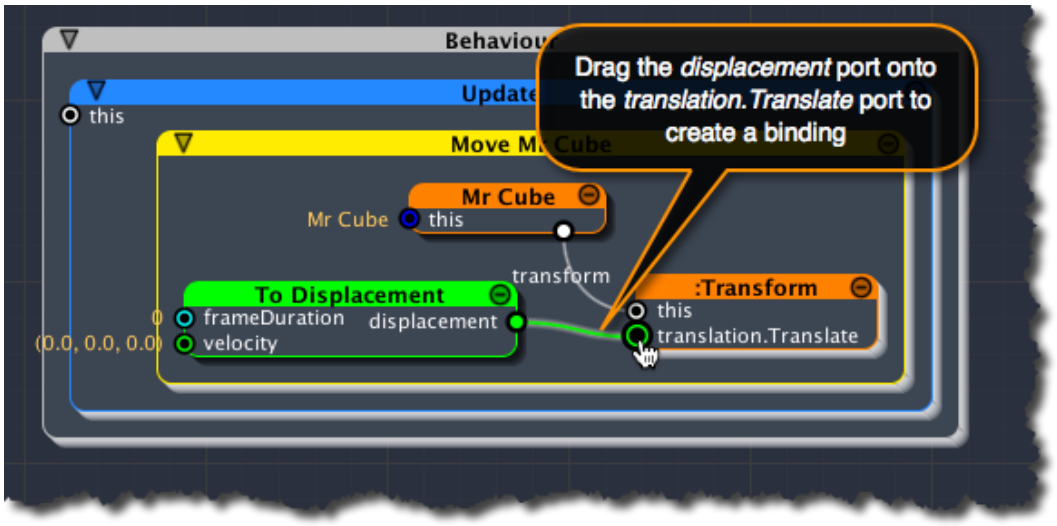


Figure 23.15: Bind the computed displacement to Mr Cube transform.

You are still missing the frame duration value. Unity has us covered with that. The *Time* type supports several time related functions. The duration of the last executed frame is held in a variable named *deltaTime*.

Start typing *"delta"* in the search field of the *Library* panel to expose the *deltaTime* variable. You should see it under the *Time* type of the UnityEngine section.

Drag it from the library into the *Move Mr Cube* package and bind its output (named *:float*) to the *frameDuration* input of the *To Displacement* node. The final result should look like figure 23.16.

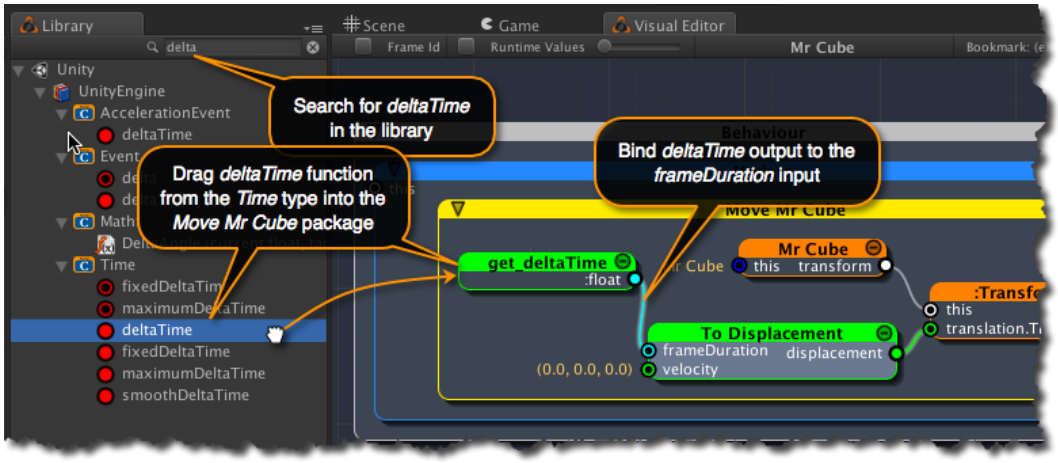


Figure 23.16: Adding frame duration for displacement computation.

Unity Tips

The *CharacterController* component:

Advised users of Unity will have noticed that we have reproduced the functionality to move a game object using relative speed; a functionality that is available in the *CharacterController* component.

We have deliberately avoided the *CharacterController* to focus the learning experience on building visual scripts. The *CharacterController* offers many features that are beyond the scope of this tutorial. We strongly advise that you take some time to learn about the *CharacterController* before building large on complex projects with Unity.

23.9 Controlling Speed and Direction Separately

Referencing back to the overview of the visual script (section 23.3) depicted in figure 23.3, you will notice that the *Move Mr Cube* package receives its input values from the *Adjust Direction* & *Adjust Speed* packages (you will build these packages later on). This means that separate controls for direction and speed are required.

Your next task is to extend your visual script to:

- accept two (2) inputs: a *direction* and a *speed* value;
- compute the velocity value from the *direction* and *speed* inputs;
- bind the "computed velocity" to the *velocity* input of the *To Displacement* node created in the previous section.

In our example, the *speed* and *direction* are defined as:

- the distance travel in one (1) second and;
- a unit vector pointing in the direction of movement;

respectively.

From these definitions, you can calculate the velocity by scaling the direction vector (a unit vector) by the speed (a scale value) as described in equation #2:

Eq. #2. Calculation of the velocity.

```
velocity = speed * direction
```

Note: A *unit vector* is a vector who's length is equal to one (1).

Equation #2 has exactly the same structure as the displacement calculation performed in the previous section: you need to multiply a vector value and a scaler value. This is your chance at trying another cool feature of iCanScript: **node cloning** (without social or legal issues!!!).

To clone the multiply operator, you need to press the ***Shift*** key and drag the *To Displacement* node in an empty area inside the *Move Mr Cube* package (see figure 23.17).

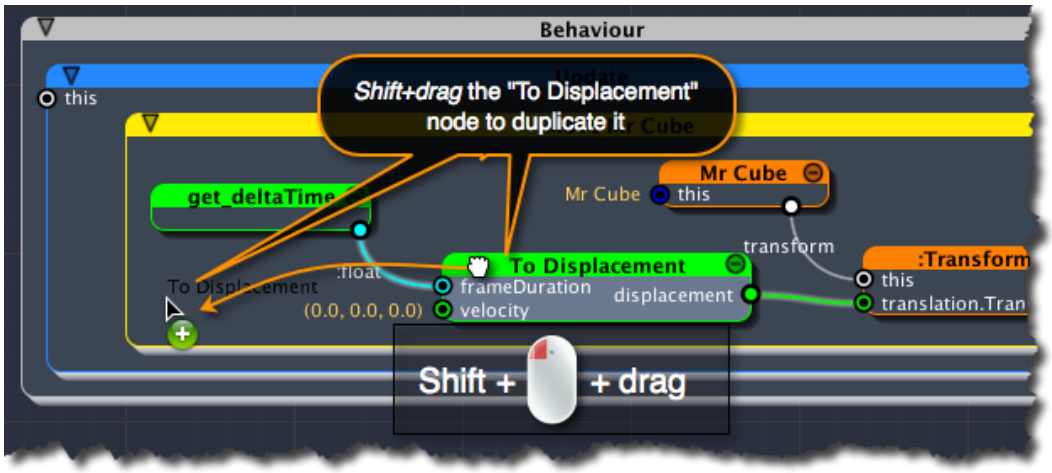


Figure 23.17: Duplicating the vector scale node.

Once more, you should rename the node and ports to better describe the intent. Use the *Hierarchy Tree* as you have done for the *To Displacement* node to rename the new node and its ports as shown in figure 23.18.

You also need to bind the *velocity* output port of the new *To Velocity* node to the *velocity* input port of the *To Displacement* node.

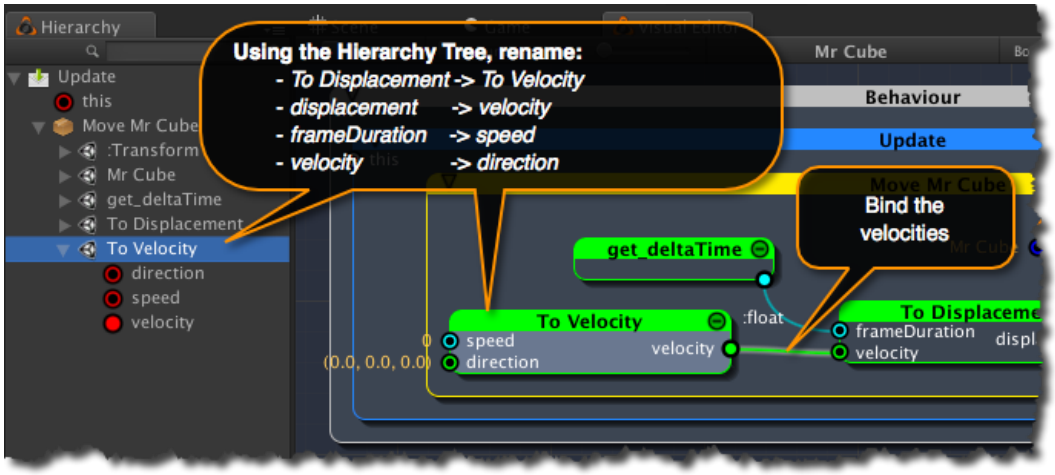


Figure 23.18: Properly rename the “To Velocity” node.

23.10 Publishing the Interface

Now you have completed the functionality of the *Move Mr Cube* package. The last step is to publish the *speed* and *direction* ports so that they can be accessed from outside the *Move*

Mr Cube package. This is easily done by dragging them on the left edge of the *Move Mr Cube* package. The final build of the *Move Mr Cube* package can be seen in figure 23.19.

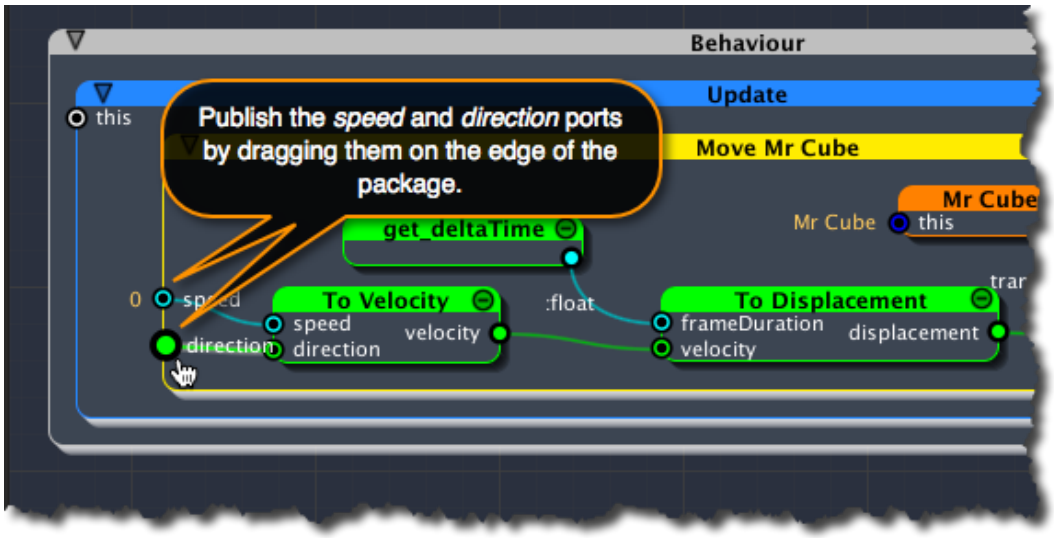


Figure 23.19: Publish speed and direction ports.

23.11 Running the *Move Mr Cube* Visual Script

I bet your fingers are twitching to give your new script a go.

Before you do so, you need to configure initial values for the *speed* and *direction* ports. The initial values are configured using the *Inspector*.

Make certain that the *Mr Cube* game object is selected and open the *Inspector*. You will see that a *visual script component* is installed on *Mr Cube*. Unfold the *visual script component* and you will see that it contains two sections:

- a ***Selected Object*** section and;
- an ***Engine Selected Object*** section.

You will be using the *Selected Object* section to configure the *"speed"* and *"direction"* ports. Unfold the *Selected Object* section if it is folded.

With the *Inspector* open, select the *speed* port in the visual editor. You will see the details of the port presented in the visual script component. Change the speed value from zero (0) to one (1).

Now select the *direction* port in the visual editor. The *Inspector* information will change to match your new selection. Change the *direction* value from (0,0,0) to (–1,0,0).

You are all done; the initial values are now configured.

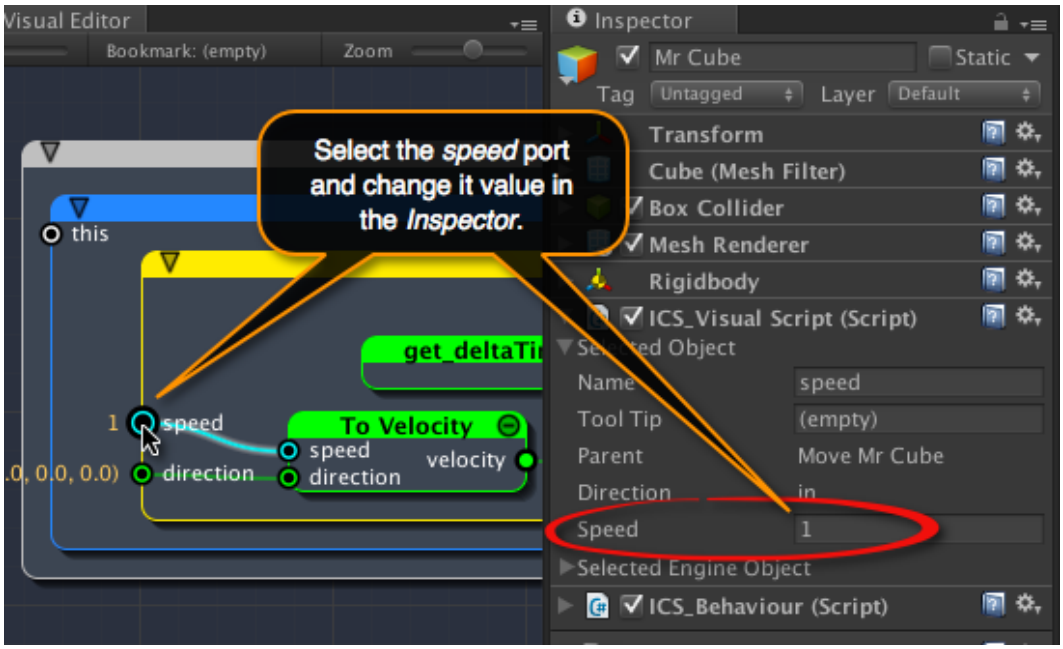


Figure 23.20: Change speed to 1.

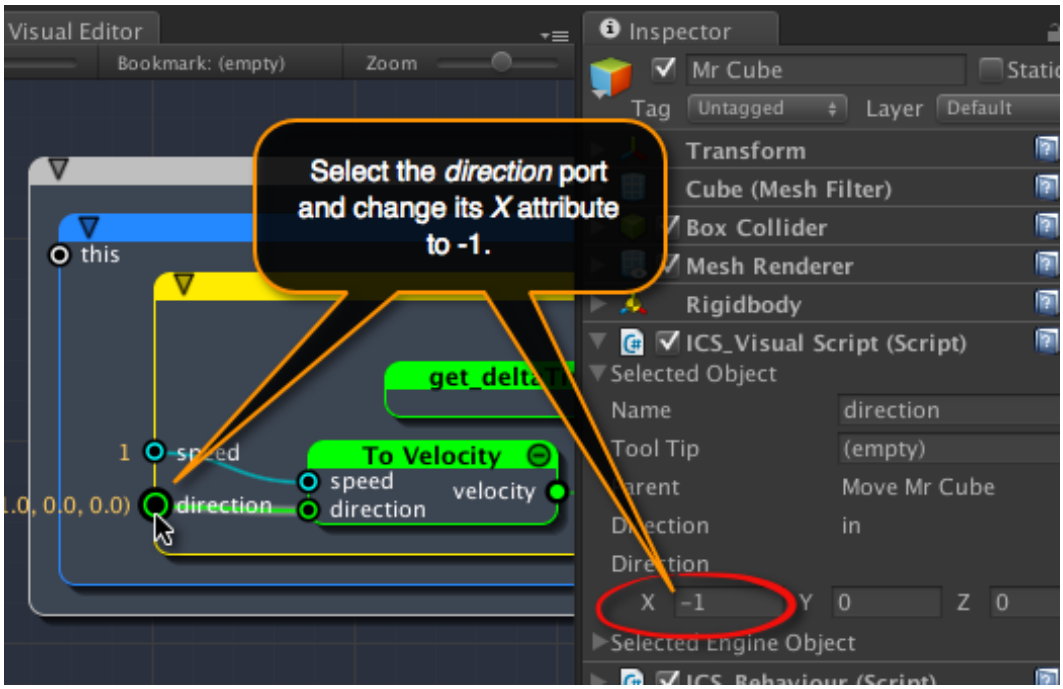


Figure 23.21: Change direction to (–1,0,0).

Hit the run button at the top of the Unity editor to give it a whirl (yeh!!!).

iCanScript compiles and runs your script. *Mr Cube* will start moving towards the centre of the scene (0,0,0) and continues forever in the same direction.

You can configure different values of direction and speed while your script is running. The new values are immediately impacting the behaviour of your script. However, the values configured while the engine is running are temporary and they will revert back to their initial configured values once the Unity game engine is stopped.

 iCanScript User Interface Tips

The details of the selected node or port are accessible from the visual script component in the *Inspector*. The visual script component inspector is divided in two sections:
Selected Object:

This section allows to view and configure the attributes of the selected node or port.

Selected Engine Object:

This section displays in-depth information to help debug the visual script. This information will not be used in this tutorial.

23.12 Accessing Runtime Information

Let's take this opportunity to discuss some of the basic debugging features of iCanScript.

The visual editor remains active while the script is running. Selecting the visual editor tab will bring it forward. You may decide to relocate it so that both the visual editor and the game window be visible.

The visual editor displays the following runtime information in the centre of it's toolbar:

- the current frame #;
- the average frame rate (in frames /sec).

In addition, the Visual Editor toolbar includes several options to control the display of runtime information. These options are:

Enable/Disable display of the frame # : Enable this control to display the last executed frame # in the title bar of each node. The display of the last executed frame # is especially useful when conditional execution is used. It helps differentiate the nodes that are executing from the nodes that are stalled.

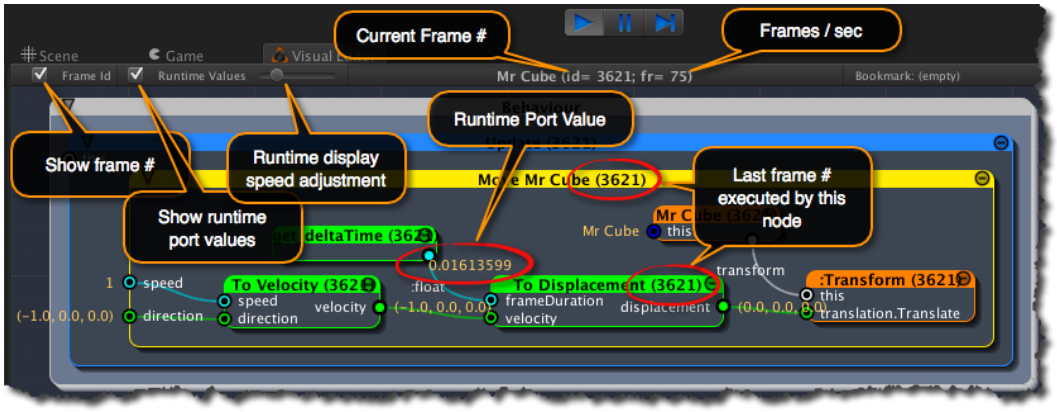


Figure 23.22: iCanScript Visual Editor with runtime debug information.

Enable/Disable display of port values : Enable this control to display the runtime values of the ports. The runtime values are displayed in beige and are periodically refreshed.

Slider to control the refresh period : This slider is used to control the frequency at which the runtime information is displayed in the visual editor. Depending on the performance of your computer, the display of runtime information could impair on the frame rate of your game. Reducing the refresh rate of the runtime information will reduce the performance impact on your game.

23.13 Recap on Your First Visual Script

So what have you done so far? You have:

- Create the Unity scene to host your example project;
- Installed a visual script on the *Mr Cube* game object;
- Created the *Update* message handler to contain and execute your visual script on each frame;
- Created a package to encapsulate the script to control the movement of *Mr Cube* using speed and direction;
- Dragged the *Mr Cube* game object inside the script and extracted its *Transform* component;
- Build a small equation to translate the speed and direction inputs into a displacement to be applied on *Mr Cube* transform;
- Publish the *speed* & *direction* ports onto the *Move Mr Cube* package for easy access by other packages.
- Tested your new script with predefined speed & direction values.

Chapter 24

Enabling & Disabling Ms Light

To Be Continued ...

Chapter 25

Homing on Ms Light

To Be Continued ...

Chapter 26

Creating a Timer Utility

To Be Continued ...

Chapter 27

Changing Direction

To Be Continued ...

Chapter 28

Mr Cube Roaming State

To Be Continued ...

Chapter 29

Adding a Panic State

To Be Continued ...

Chapter 30

Beautifying your Scripts

To Be Continued ...

Part VII

User Interface

Chapter 31

Menus

iCanScript installs itself in Unity's Edit, Component, Windows and Help menus. This section describes the iCanScript functionality associated with each menu item.

31.1 Edit Menu

iCanScript extends Unity's ***Edit*** menu with navigation features and the ability to create a Visual script as depicted in the below figure.

Edit->iCanScript->Create Visual Script

Summary:

```
Attaches a visual script component to the selected game object.
```

Description:

The ***Create Visual Script*** menu item is used to attach an iCanScript Visual Script component to the selected game object.

The Visual Script component is at the root of your visual script. It includes the iCanScript engine code that compiles and runs the visual script when the Unity engine runs. It also saves & reloads your visual script when the scene in which it is contained is saved or reloaded.

A game object containing an iCanScript Visual Script can be edited using the visual editor.

Edit->iCanScript->Center Visual Script

Summary:

```
Centres the visual script in the viewport.
```

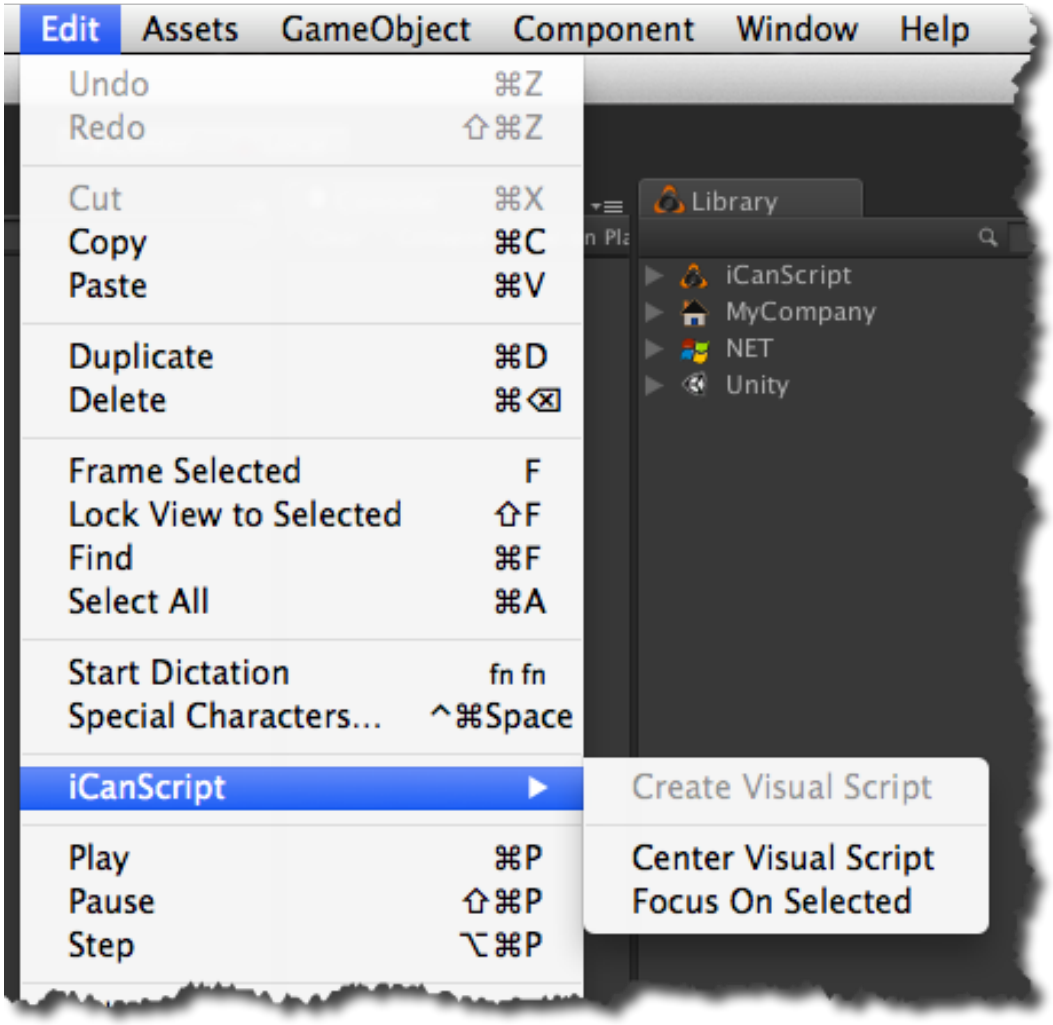


Figure 31.1: iCanScript Edit menu extension.

Table 31.1: Create Visual Script Enable State

Menu State	Condition
Enabled	- Selected game object without a Visual Script component.
Disabled	- No selected object;- Selected object already has a Visual Script component.

Description:

The ***Center Visual Script*** menu item is used to reposition the visual script in the centre of the visual editor viewport.

The zoom factor of the viewport may be adjusted to improve visual script visibility.

Edit->iCanScript->Center Selected

Summary:

```
Centres the selected node / port in the viewport.
```

Description:

The ***Center Selected*** menu item is used to reposition the selected node in the centre of the visual editor viewport.

The zoom factor of the viewport may be adjusted to improve selected node visibility.

31.2 Component Menus

Creating a visual script with iCanScript requires that you install the iCanScript Visual Script component on your game object. The iCanScript Visual Script component can be attached using Unity's top-level *Component* menu as well as the *Add Component* button located in the Inspector.

Attaching a Visual Script component

The ***Create Visual Script*** menu item is used to attach an iCanScript Visual Script component to the selected game object.

The Visual Script component is at the root of your visual script. It includes the iCanScript engine code that compiles and runs the visual script when the Unity engine runs. It also saves & reloads your visual script when the scene in which it is contained is saved or reloaded.

A game object containing an iCanScript Visual Script can be edited using the visual editor.

Table 31.2: Create Visual Script Enable State

Menu State	Condition
Enabled	- Selected game object without a Visual Script component.
Disabled	- No selected object;- Selected object already has a Visual Script component.

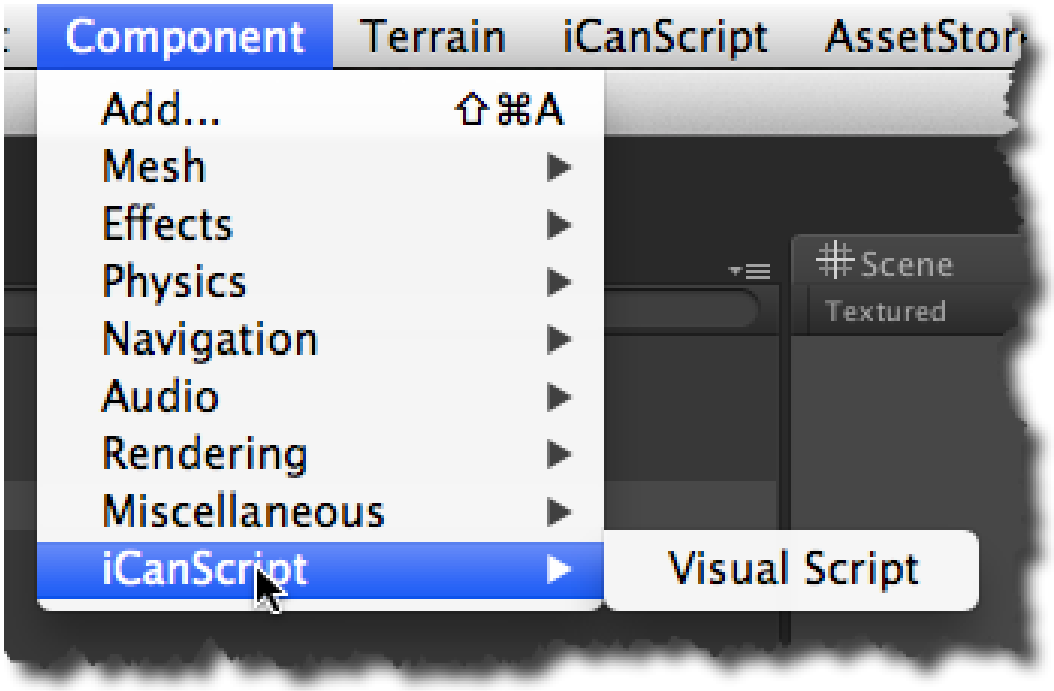


Figure 31.2: iCanScript Component menu extension.

31.3 Window Menu

iCanScript installs menu items to open its five (5) editors & panels in Unity's top-level *Window* menu.

Window->iCanScript->Preferences

Summary:

```
Opens the Preferences Panel
```

Description:

iCanScript ***Preference Panel*** is used to configure the appearance and global behaviour of iCanScript editors & panels.

The iCanScript preference configuration is save for the current user session and is independent of visual script saved data.

The Preference Panel also includes iCanScript version information.

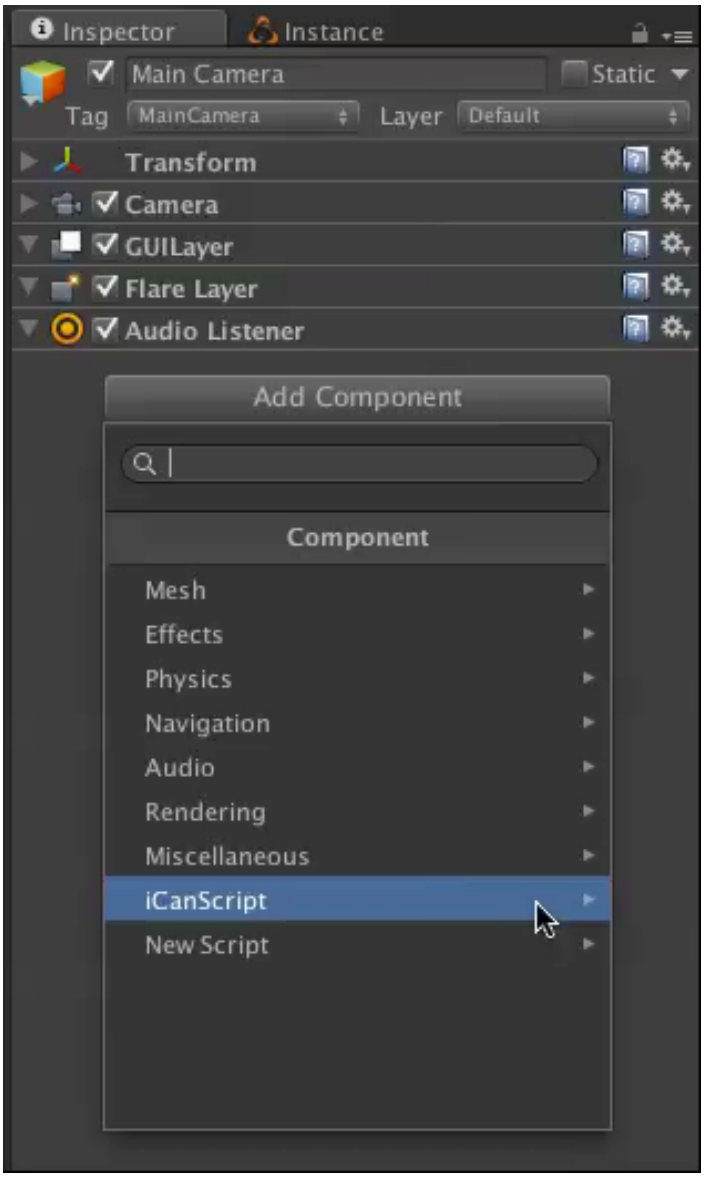


Figure 31.3: iCanScript Inspector->Component menu extension.

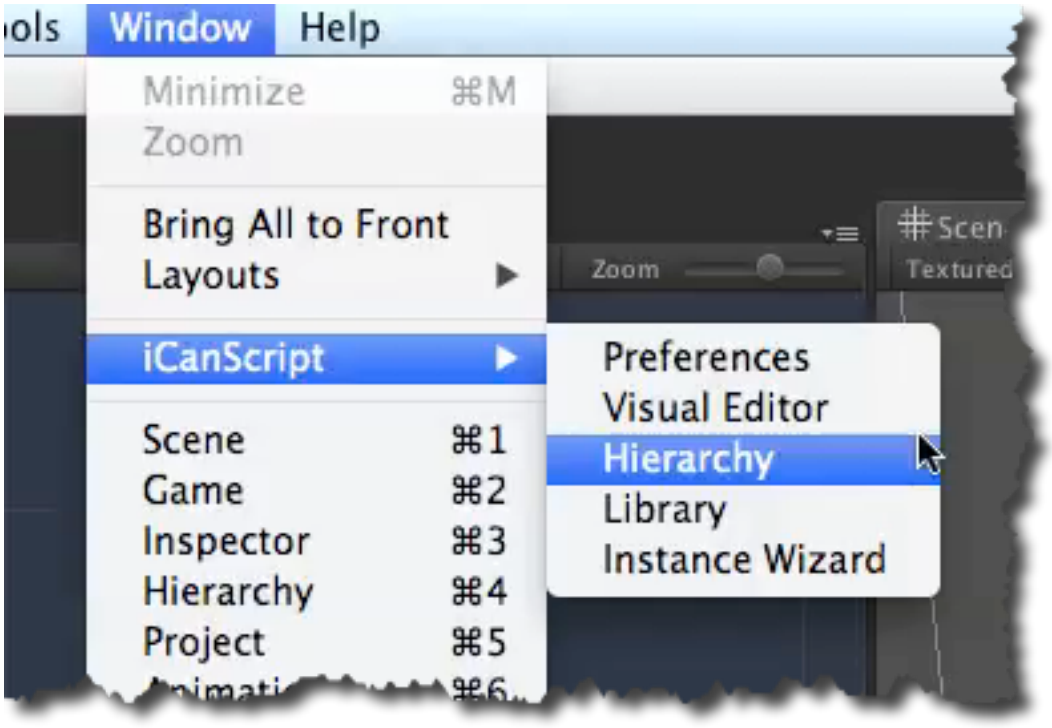


Figure 31.4: iCanscript Window menu extension.

Window->iCanScript->Visual Editor

Summary:

```
Opens the Visual Editor.
```

Description:

The Visual Editor is the core editor of iCanScript. It is where you will create and modify your visual scripts.

The Visual Editor displays the visual script associated with the currently selected game object. The visual editor will be empty if no game object is selected or the selected game object does not include an iCanScript Behaviour component.

Window->iCanScript->Hierarchy

Summary:

```
Opens the visual script hierarchy tree.
```

Description:

The Hierarchy Tree Editor is used to navigate and edit your visual script as a tree view. It complements the graphical visual editor.

You can edit the node & port names and quickly navigate your visual script from the hierarchy tree editor.

Window->iCanScript->Library

Summary:

```
Opens the node library tree.
```

Description:

The ***Library Panel*** is a repository of all available nodes that can be dragged into your visual script. It includes Unity's core library, iCanScript base node library as well as some of the .NET functionality.

The Library Panel can easily be extended with your own nodes. Refer to section Extending iCanScript to learn how to add your own nodes in the library.

Window->iCanScript->Instance Wizard

Summary:

```
Opens the node instance wizard.
```

Description:

The ***Instance Wizard*** is used to intelligently present and configure the fields, properties, and functions associated to the type of a node in your visual script.

The instance wizard will automatically activate when selecting a node associate with a specific type (i.e. .NET class).

31.4 Help Menu

iCanScript extends Unity's ***Help*** menu to provide quick access to the product web site, documentation and support pages.

Help->iCanScript->Home Page

Summary:

```
Opens iCanScript home page in the default Web browser.
```

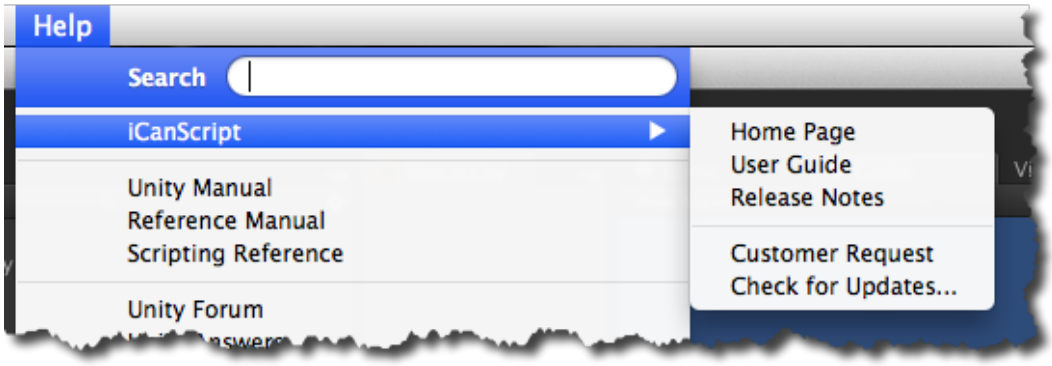


Figure 31.5: iCanScript Help menu extension.

Description:

The ***Home Page*** menu item is used to open the home page of the iCanScript Web site in a new internet browser window.

An internet connection is required to access the iCanScript home page.

Help->iCanScript->User's Guide

Summary:

```
Opens iCanScript user manual in the default Web browser.
```

Description:

The ***User's Guide*** menu item is used to open the iCanScript online user guide in a new internet browser window.

An internet connection is required to access the iCanScript online user guide.

Help->iCanScript->Release Notes

Summary:

```
Opens latest release notes in the default Web browser.
```

Description:

The ***Release Notes*** menu item is used to open the iCanScript latest release information in a new internet browser window.

The release notes are sorted from the latest release to the oldest release for your convenience. A copy of the release notes in PDF format can also be downloaded from the web site.

An internet connection is required to access the iCanScript release note pages.

Help->iCanScript->Customer Request

Summary:

```
Opens the feature request and bug report form.
```

Description:

The ***Customer Request*** menu item is used to submit feature requests and bug reports for the iCanScript product. You can also browse the status of existing customer requests.

The customer request form is Web based and requires an active internet connection.

Help->iCanScript->Check for Updates...

Summary:

```
Validates the current version against latest available release.
```

Description:

The ***Check for Update*** menu item is used to determine if you have the latest official release of iCanScript. You will be prompted to download the latest version if your version is out-of-date.

Validation of the latest iCanScript version requires an active internet connection.

Part VIII

Extending iCanScript

Chapter 32

Extending iCanScript

From its inception, iCanScript was designed to be extendable with the addition of user defined nodes. Once installed, the user defined nodes are accessible from the [Library Tree].

Extending the iCanScript library is realized by:

1. Tagging your Source Code (chapter 33);
2. Importing Public Members (chapter 34);
3. Adding Message Handlers (chapter 35).

The following picture illustrates how the iCanScript library gets populated and used. You are given three access points (purple) to include your own packages and nodes inside the iCanScript library. The iCanScript library gets repopulated for every recompile of the Unity scripts.

Tagged Source Code: You can add to the iCanScript library by tagging your source code with specialized .NET attributes. This provides for fine control of which elements in your source code are published to the iCanScript user.

Importing Public Members: You can add to the iCanScript library by batch importing all *public* members of specified programatic types. This method is specially useful for importing libraries for which you do not have the source code.

Message Installation: The .NET reflection technology allows for dynamic messaging; a technic used to dynamically invoke a functionality only if it is defined for a software object. A script responds to dynamic messages if it includes the appropriate message handlers. You can populate the iCanScript library with the signature of message handlers for

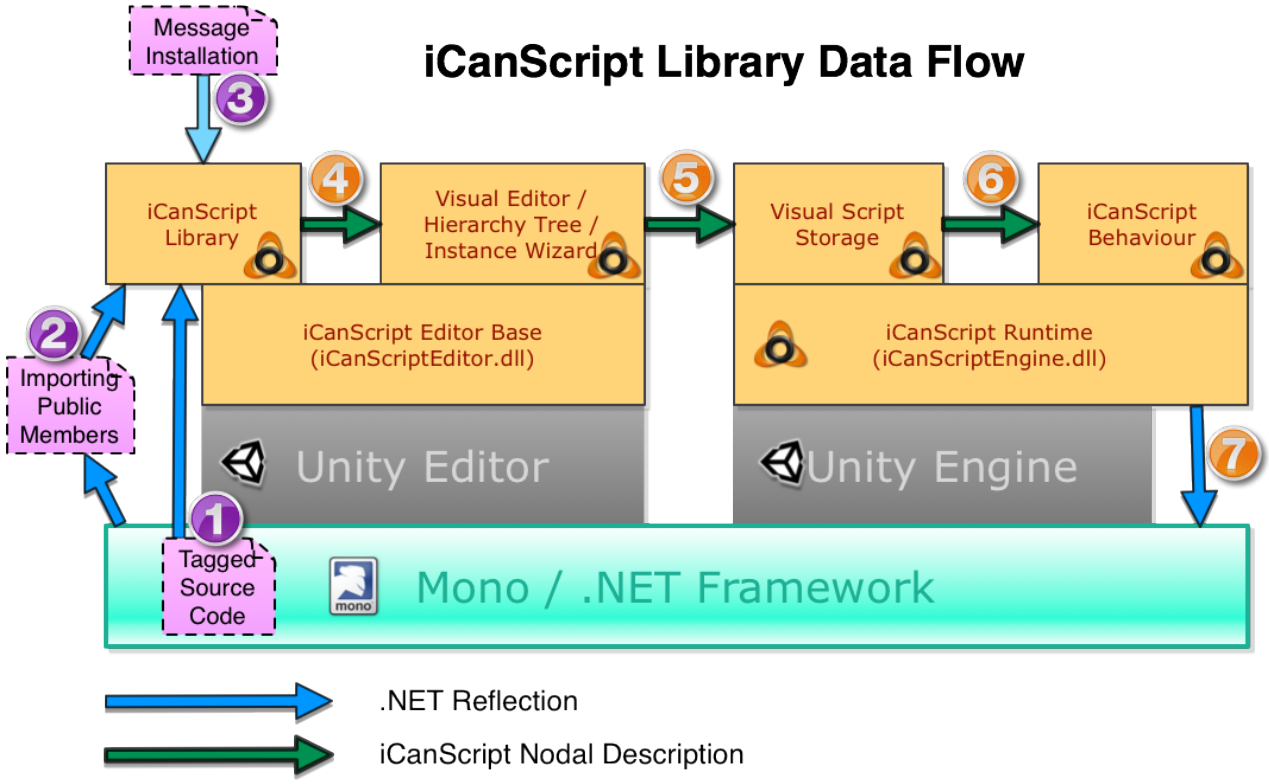


Figure 32.1: iCanScript library information flow.

given programatic types. Afterward, the signature can be used by the iCanScript user to build the message handler nodes.


The iCanScript editors extracts from the library node and port information when building the visual script. The visual script manifest and layout information is saved along with the game object that includes the *iCS_ VisualScript* component.


iCanScript also generates the *iCS_ Behaviour* code that is needed by Unity to execute the the Visual Script. The behaviour code includes:

1. the message handlers;
2. the control logic that determine the execution order of nodes, and;
3. the nodes converted into .NET code.


When the Unity engine is ran, the iCS_Behaviour code instantiates the variables, invokes the .NET code of the nodes, and responds to the message handlers. The iCanScript core executive (iCanScriptEngine.dll) assures the execution order and resolves data contention.

Chapter 33

Tagging your Source Code

iCanScript includes several .NET attributes to control which part of your source code is to be included in the iCanScript library. These attributes enables fine control over which fields, properties and functions become visible to the iCanScript user.

iCanScript scans the .NET assemblies to populate its library with public classes tagged with the *iCS_Class* attribute. The assembly scan is performed after every recompile of the Unity scripts.

The following table depicts the mapping between C# source code and iCanScript objects:

Table 33.1: Mapping between C# source code and iCanScript objects.

Source Code	iCanScript Objects
class	 Class /Type Node
field	 Port
property	 Port
constructor	 Variable Builder Node
instance function	 Function Node with a *"this"* input port
class function	 Function Node without a *"this"* input port

Note: All source code elements tagged with an iCanScript attribute *MUST*

HAVE a public programmatic scope.

33.1 iCanScript .NET Attribute Reference

The following two tables enumerates the supported iCanScript attributes and their associated parameters:

Table 33.2: iCanScript .NET Attributes

Attribute Name	Targets	Parameters
iCS_Class	class or structure	**CompanyCompanyIconLibraryIconTooltipBaseVisibility**
iCS_Function	constructor,function,property get,property set	**NameReturnIconTooltip**
iCS_InPort	field	
iCS_OutPort	field	
iCS_InOutPort	field	

Table 33.3: iCanScript .NET Attribute Parameters.

Parameter Name	Type	Description
BaseVisibility	bool	if true: adds the base classes public variables and functions.
Company	string	The name shown at the first level of the library tree.
CompanyIcon	string	Path to the iconic representation of the company.
Icon	string	Path to the iconic representation of the node.
Name	string	Function name substitute.
Library	string	The name shown at the second level of the library tree.
Return	string	Port name for the function *return value*.
Tooltip	string	Brief description of the component.

33.2 iCS_Class Attribute

iCanScript adds a type node to its library for each class and structure marked with the *iCS_Class* attribute.

Note that iCanScript will only add *public* classes and structures.

Example:

33.3 iCS_Function Attribute

You can publish to the iCanScript library your C# properties, constructors, instance function and class function using the *iCS_Function* attribute.

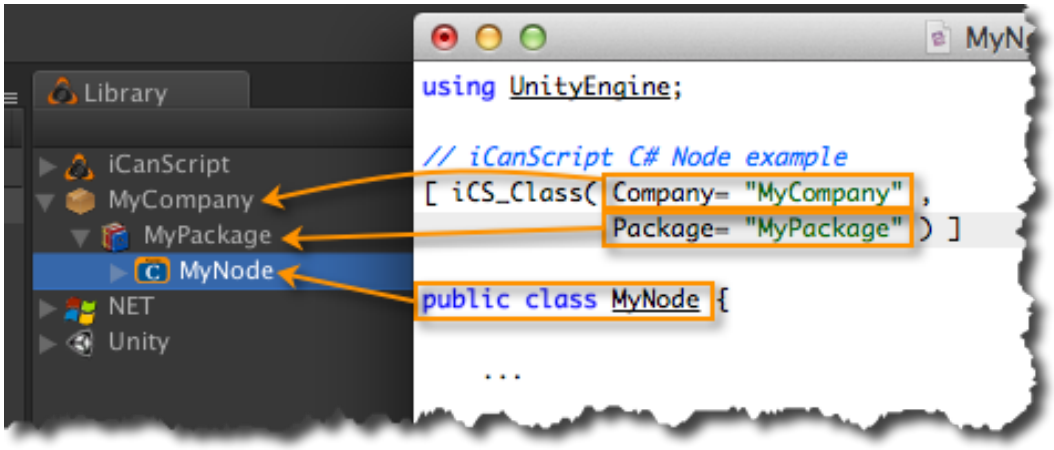


Figure 33.1: iCS_Class attribute example.

See Tagging your Source Code (chapter 33) for details on the C# source code to iCanScript object mapping.

> **Note:** The *iCS_Function* attribute can only be applied to *public* members of the C# class.

Example:

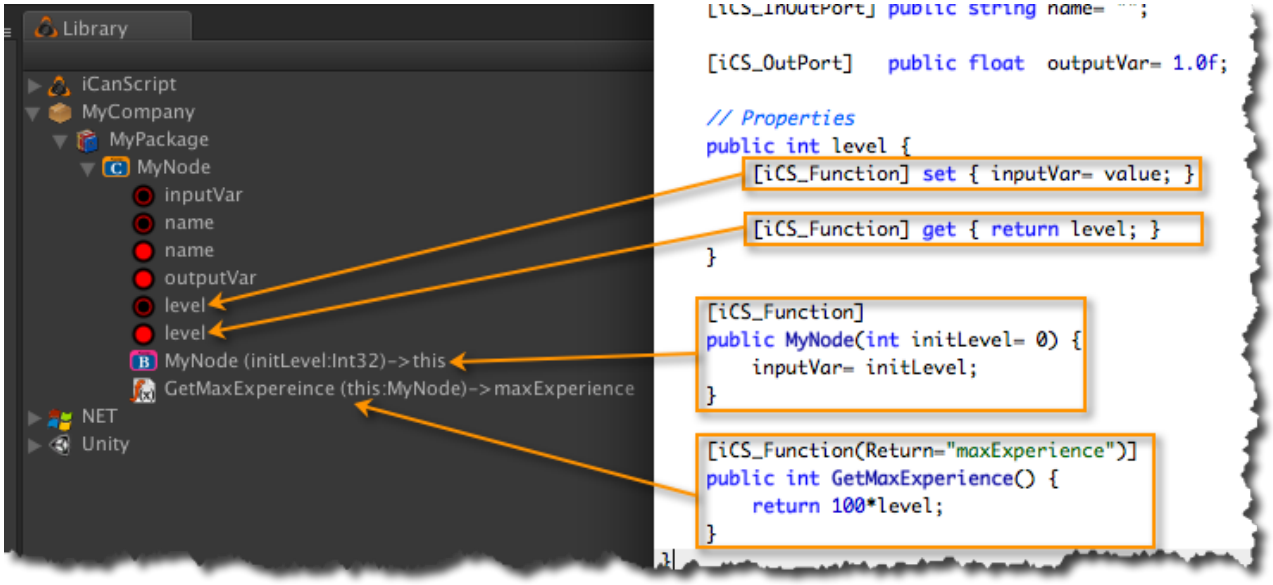


Figure 33.2: iCS_Function attribute example.

33.4 iCS_InPort, iCS_OutPort, and iCS_InOutPort Attributes

You can publish to the iCanScript library your C# fields using the *iCS_InPort*, *iCS_OutPort*, and *iCS_InOutPort* attributes.

> **Note:** The *iCS_xxPort* attributes can only be applied to *public* fields of C# classes.

Example:

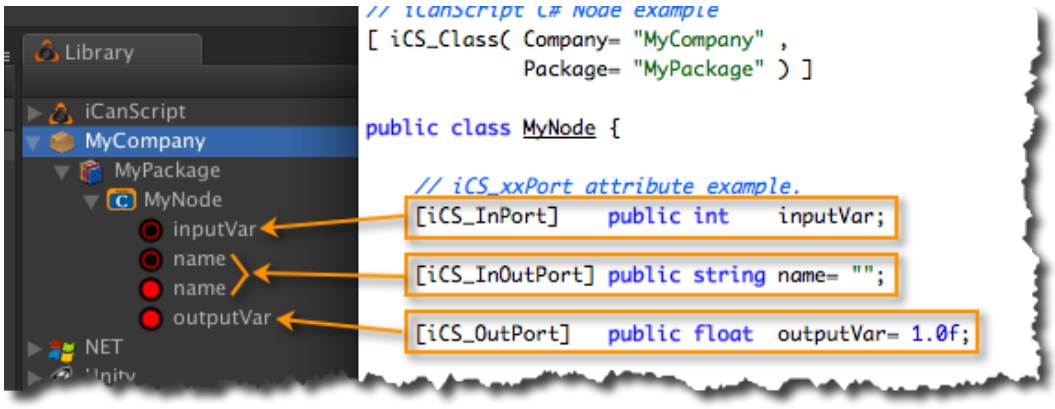


Figure 33.3: Figure 2. iCS_xxPort attribute example.

Chapter 34

Importing Public Members

iCanScript supports invoking a custom installer to add all public fields, properties and functions of an object definition into the iCanScript library.

This method of populating the iCanScript library is ideal if:

- you are not in control of the source code (i.e. using a vendor library) or;
- you want to add all public fields, properties and functions of your classes without tagging your source code with iCanScript custom attributes.

34.1 Understanding the Custom Installer

On every script recompile, iCanScript seeks for and invokes the *void PopulateDataBase()* static function of the *iCS_ CustomInstaller* static class to populate it library.

> **Note:** iCanScript uses runtime binding to avoid generating compilation errors if the custom installer is not present.

The signature of the iCanScript library custom installer is:

```
// iCanScript library custom installer class
public static class iCS_CustomInstaller {
    // Function invoked to populate the iCanScript library.
    public static void PopulateDataBase() {
        // ==> INSTALL YOUR NODES HERE <==
        ...
    }
}
```

The iCanScript distribution comes with a custom installer template to be use as a base for your custom installer. It is located in the editor section of the iCanScript package as show in the following diagram.

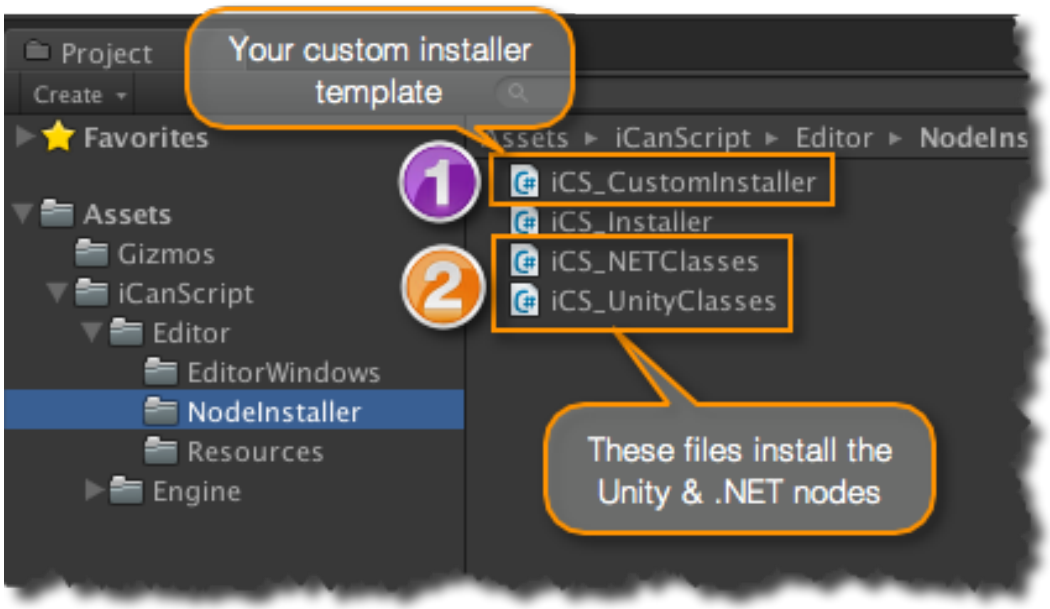


Figure 34.1: Extending Library with Custom Installer.

The *iCS_CustomInstaller* file should be used as a starting point to create your own installer. You will need to move this file into your package and modify it to specify the types to be included in the iCanScript library.

The files *iCS_NETClasses* and *iCS_UnityClasses* install all of the .NET and Unity types that come with the iCanScript distribution. Browsing those files will give you a better understanding on what needs to be done when creating your custom installer.

34.2 Modifying the Custom Installer Template

> **Important:** It is important that you move the custom installer template file outside of the iCanScript package before you modify it. This will avoid losing your changes on subsequent iCanScript package upgrades.

Their is five sections of interest in the custom installer template, four of which you will need to modify. The following provide the details:

The template file that comes in the distribution is disabled. This avoids conflicts with your own custom installer on subsequent product upgrades. To activate your custom installer, you need to uncomment the definition of iCS_USE_CUSTOM_INSTALLER.

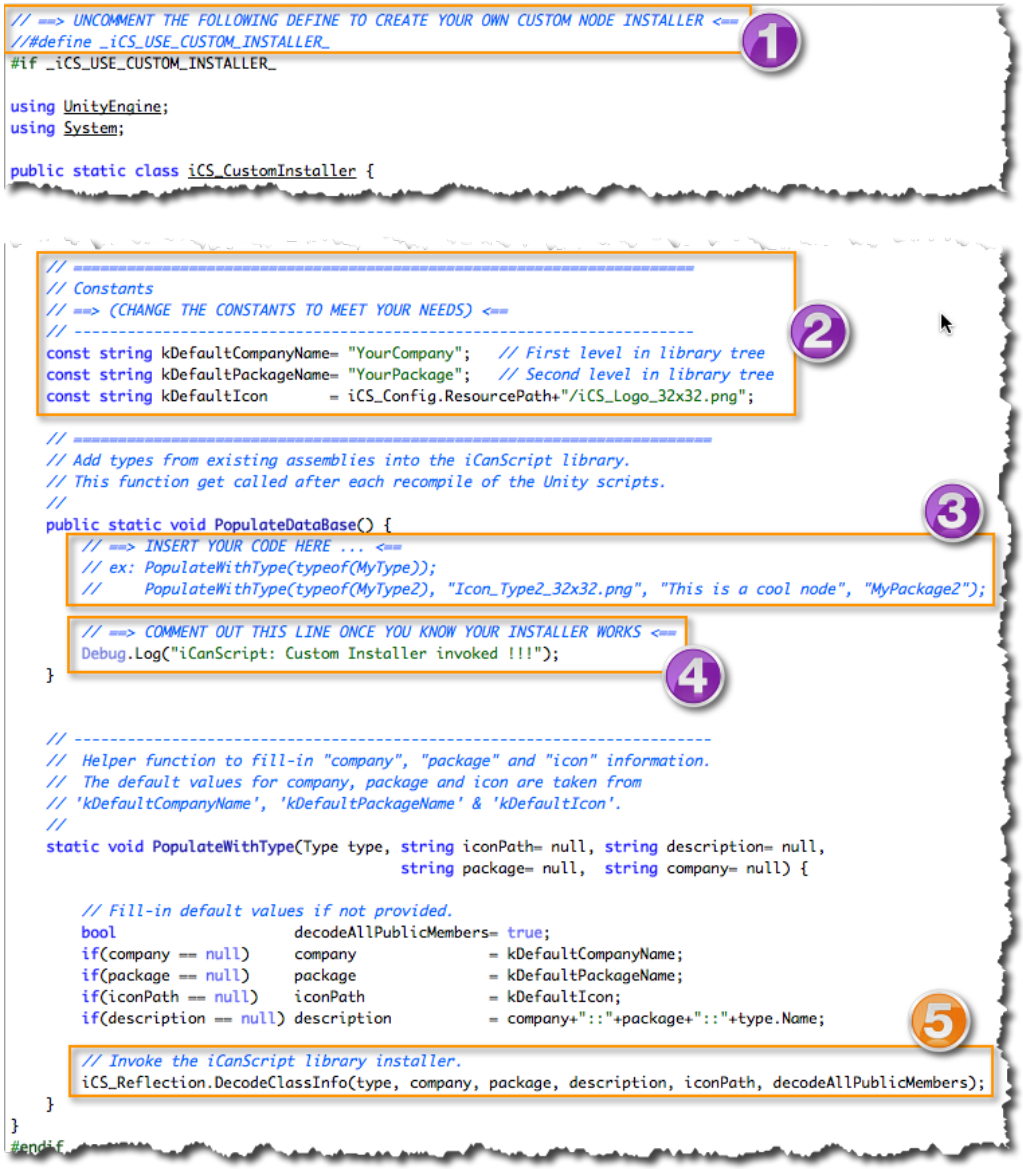


Figure 34.2: Custom Installer Template File.


The template includes the helper function *PopulateWithType(...)* that fills-in the default company name, package name, and icon if they are not specified. The default values are taken from *kDefaultCompanyNam e*, *kDefaultPackageName*, and *kDefaultIcon* constants. Change the value of those defaults to reflect your situation.


This is were the bulk of your installer will be coded. You need to invoke the helper function *PopulateWithType(...)* for every type you include in the iCanScript library. Only the type information is mandatory but it is suggested that you also fill-in the description and install an icon that depicts the type. Package name and company name can also be provided if the default values do not suffice.


By default, the iCanScript custom installation template displays a message when it is invoked. This is useful to determine if the installer is properly activated. You can remove or comment out this log message once you know your installer is invoked.


The *iCS_ Reflection.DecodeClassInfo(...)* is the main programatic interface for populating the iCanScript library. For your convenience, it is wrapped by the function *PopulateWithType(...)*.

Chapter 35

Adding Message Handlers

The Tagging your Source Code (chapter 33) and Importing Public Members (chapter 34) installing mechanism extract existing functionality from the object definition to publish into the iCanScript library. However, they do not provide for defining message handlers for the Unity framework. This is because the Unity messages are dynamically created at runtime and are therefore not visible in the object definitions.

The message handlers nodes have the following characteristics:

- they are package nodes;
- they includes child node(s) that implement the behaviour for the message;
- their input ports contain the parameter values of the message.

The following image depicts a game object behaviour with four (4) message handlers:

You rarely need to add message handlers to the iCanScript library but if do, you will need to invoke the following function:

```
void iCS_LibraryDatabase.AddMessage(    System.Type classType,
                                            string messageName,
                                            iCS_StorageClass storageClass,
                                            iCS_Parameter[] parameters,
                                            iCS_FunctionReturn functionReturn,
                                            string description,
                                            string iconPath)
```

Note: The iCanScript distribution installs the Unity message handlers in file: *"iCanScript/Editor/NodeInstaller/iCS_ UnityClasses.cs"*. Please use the Unity installer as example to create your own message handlers.

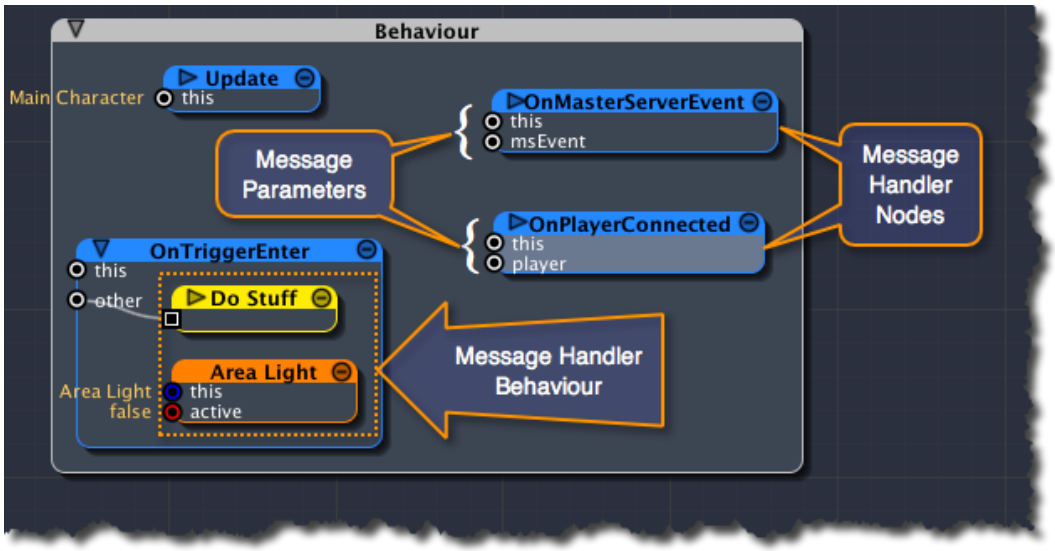


Figure 35.1: Behaviour Message Handlers Example.

Table 35.1: AddMessage parameter descriptions.

Parameter Name	Description
classType	The programatic type that will accept the message
messageName	The name of the message to handle
storageClass	Either *iCS_Storage.Class* or *iCS_Storage.Instance*
parameters	Message parameters. See *iCS_UnityClasses* file for example.
functionReturn	Message return type. See *iCS_UnityClasses* file for example.
description	A short description of the message.
iconPath	The icon to use when the message handler is iconized.

Part IX

Appendices

Chapter 36

Keyboard Shortcuts

iCanScript includes several keyboard shortcuts (hot-keys) to accelerate visual script edition. The following tables details the iCanScript hot-keys regrouped by functional area.

36.1 Visual Script Navigation

The visual script can be navigating by changing the node that serves as the display root. Use the following keyboard and mouse commands to navigate your Visual Scripts.

Table 36.1: Visual Script Navigation Shortcuts.

Navigation Hot-Keys	Action Performed
Ctrl-Double Click	*Child Node:* Set new display root node.*Display Root Node:* Revert to previous display root.
[	Moves backward in navigation history.
]	Moves forward in the navigation history.

36.2 Selection Navigation

Table 36.2: Selection Shortcuts.

Selection Hot-Keys	Action Performed
Up Arrow	Moves the selection to the parent node.
Down Arrow	Moves the selection to the first child node.
Right Arrow	Moves the selection to the next sibling node.
Left Arrow	Moves the selection to the previous sibling node.
F	Repositions the selected node or port in the centre of the visual editor viewport
Shift-F	Centres the visual script in the middle of the visual editor viewport

Table 36.3: Bookmark Shortcuts.

Bookmarks Hot-Keys	Action Performed
B	Bookmarks the active selection.
G	Moves the selection to the active bookmark
S	Swaps bookmark. Bookmarks the active selection and moves the selection to the previous bookmark.
C	Connects the bookmarked port and the selected port (requires compatible data types)

36.3 Bookmarks

36.4 Expand /Fold

Table 36.4: Expand /Fold Shortcuts.

Expand /Fold Keys	Action Performed
Enter	Expands the selected node:- **Action Nodes:** Iconized -> Folded- **Composite Nodes:** Iconized -> Folded -> Unfolded
Alt-Enter	Maximizes the selected node:- **Action Nodes:** Iconized -> Folded- **Composite Nodes:** Iconized or Folded -> Unfolded
Shift-Enter	Collapses the selected node:- **Action Nodes:** Folded -> Iconized- **Composite Nodes:** Unfolded -> Folded -> Iconized
Alt-Shift-Enter	Iconizes the selected node.

36.5 Edition

Table 36.5: Quick Deletion Shortcuts.

Edition Keys	Action Performed
L	Performs an auto-layout of the selected binding.
Del	Deletes the currently selected object with user confirmation. The selection is moved to the parent node on deletion.
Shift-Del	Deletes the currently selected node (no user confirmation). The selection is moved to the parent node.