Welcome to JigScript!

JigScript is a mini scripting language that is optimized for Unity3d. JigScript lets you code in Unity, during Play mode, in a C# syntax that is optimized to bring out the power of Unity3d. Developers can code games much faster while those new to coding can learn JigScript as an introductory level of C#. JigScript should allow a wider userbase and greater flexibility for Unity3d. JigScript was conceived as a tool to give a scientist access to many aspects of level design and it grew into a powerful scripting tool.

JigScript is ideal for prototyping, user interface development and level event scripting. JigScript is not is a full featured scripting language: for a complete language, use C#, JavaScript and Boo that already exist in Unity. However, in many cases, simplicity and easy extensibility are more important than an extensive list of features, most of which may never be used in your game. JigScript can also be used in your game as its own scripting language giving your players, or a subject matter expert, the ability to customize or mod your game.

This manual will show you how to install and get started using JigScript in your games. To learn more, troubleshoot and to contribute your extensions to the JigScript community, please go to http://Forum.JigScript.com.

Table of Contents

Installation

If you downloaded from the Unity Asset store, JigScript will automatically be imported into your project; skip to the next paragraph.

> If you have been provided with an import package for JigScript, then you will need to use the following process:
> 1. Right click (control left click on mac) on the Assets folder in your unity 3d project.
> 2. From the displayed menu, select **Import Package** then select **Custom Package**.
> 3. Navigate to the location on your local computer where the JigScript package is and select Open.

Once JigScript has been imported into your project, there will be a JigScript folder in your project under the Assets folder. The JigScript folder contains the following folders:

Folder	**Function**
Documentation	Contains this manual and example scenes and scripts.
Editor	Contains JigScript files for the unity IDE. These editor files create the Tools→JigScript menu and post processing functions that are run when your JigScript files change.

Engine	Contains the core engine files that compile and run JigScript code.
Libraries	Contains extension libraries that add callable functions, properties and variables to JigScript.
Materials	Contains simple materials used in the examples.
Prefabs	Contains the JigScript and JigMat prefabs. The JigScript prefab when added to your scene enables it to run JigScript files. The JigMat prefab is used by JigScript to represent a script assignable material.
Resources	Contains JigScript code and other resources that are loaded at run time by JigScript script. *Note: The GUI List control uses the JigSkin, listitem and listitemOn.*
Test	Contains additional test scripts and scenes that has been developed to test JigScript functionality. These scripts may be useful to you in understanding some of the features and use of JigScript.

The Editor, Engine, Libraries, Prefabs and if you use the list control Resources/JigScript/Skins folders are required. The remaining of folders can be safely removed.

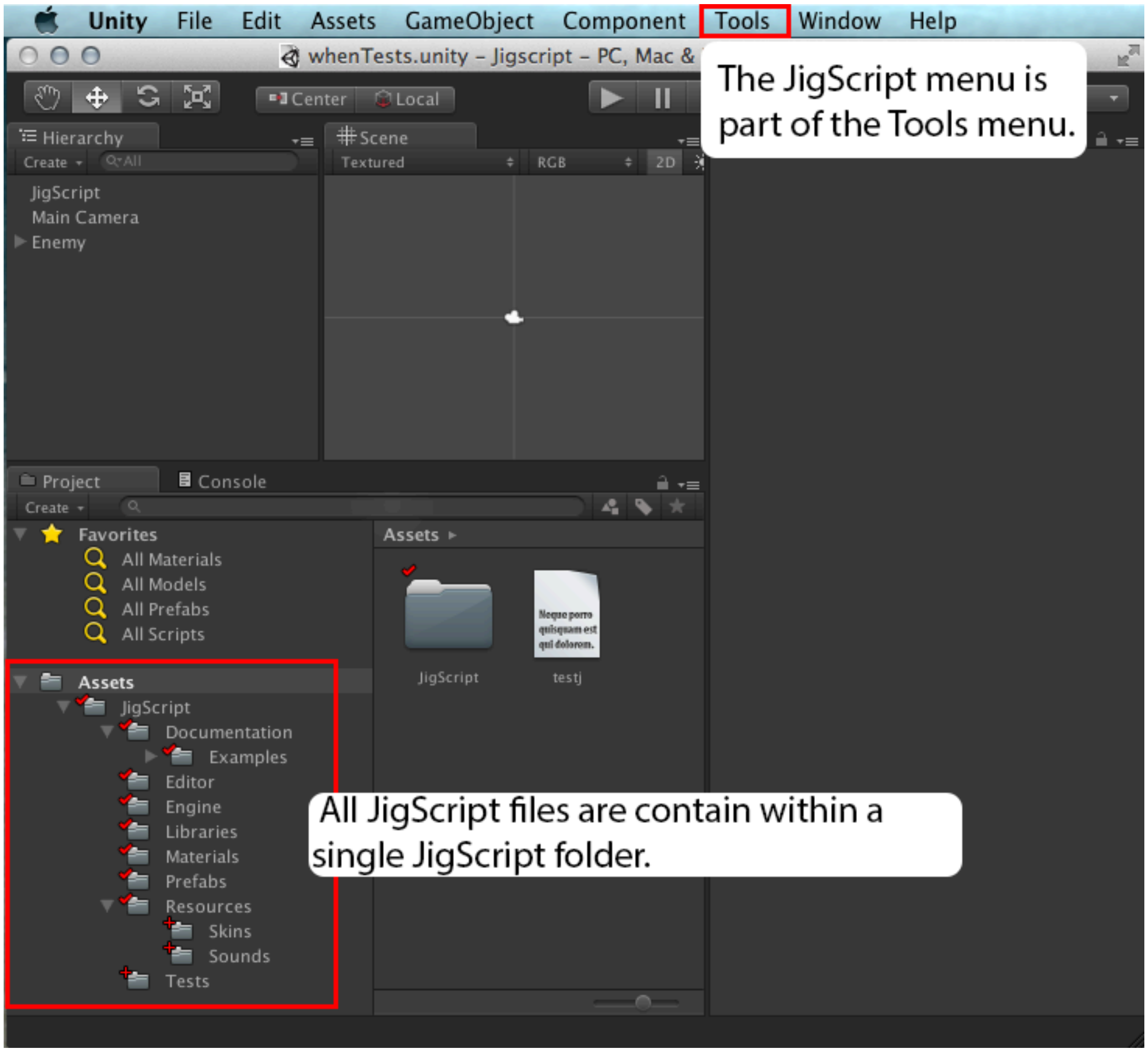


Figure 1, After JigScript is installed; you will see a Tools menu and a JigScript folder.

Setup and Use

To use JigScript in your scene, you need to add the JigScript prefab to the Hierarchy. The simplest way of doing this is to select the **Add Jig Script Prefab** option from the Tools→JigScript menu. This menu option can also be accessed by pressing ⌘P on Mac or ^ P on Windows.

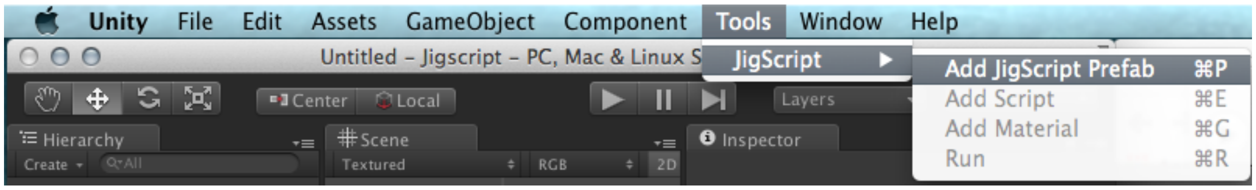


Figure 2, JigScript dropdown menu allows you to quickly access JigScript functions.

Once you have the JigScript prefab is in your scene, click on the JigScript prefab and look at its inspector window. You will see a field called **Script File**. This is a Text Asset that contains the text file that JigScript will execute. To set this asset, simply select the text asset you want to run and drag it into this field.

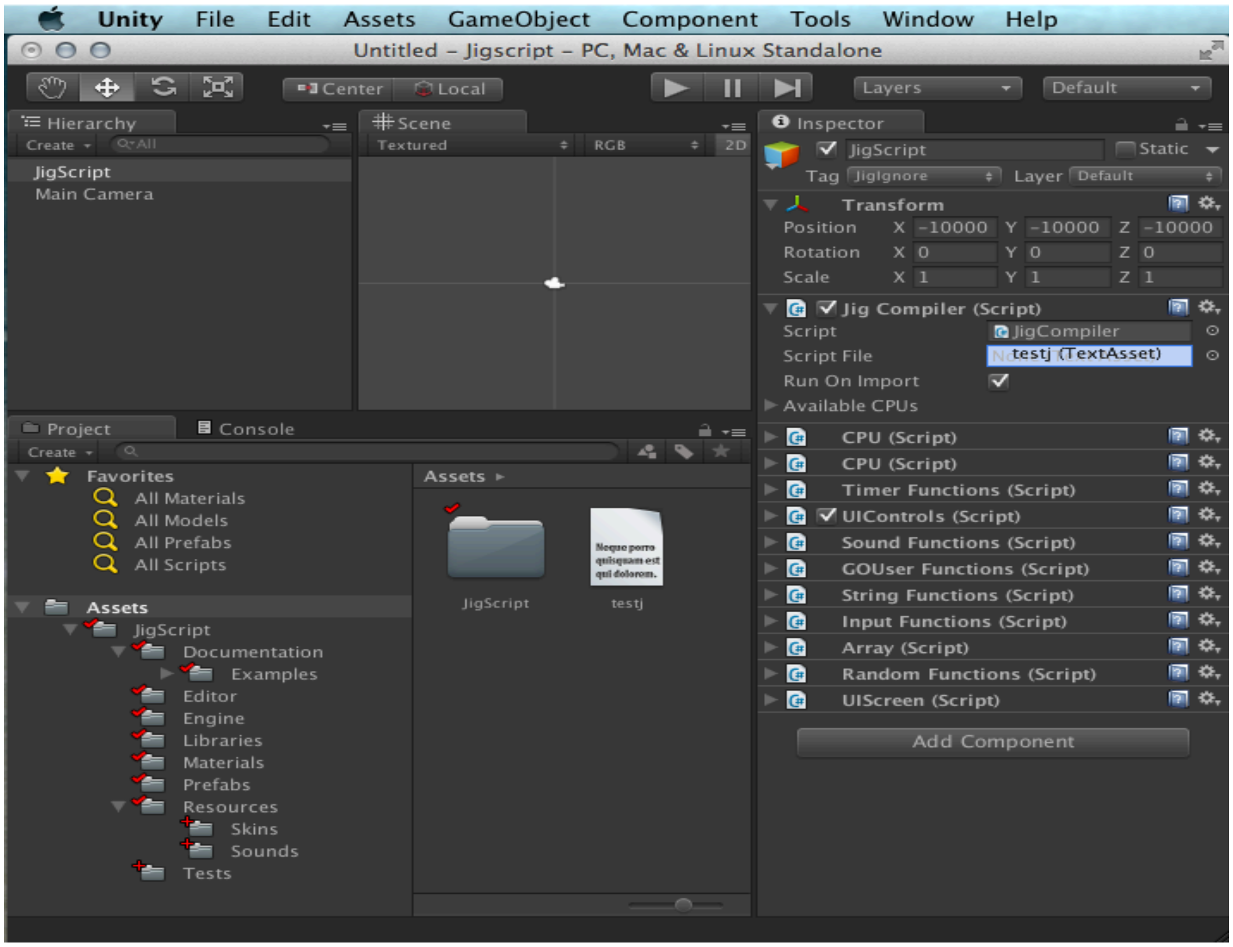
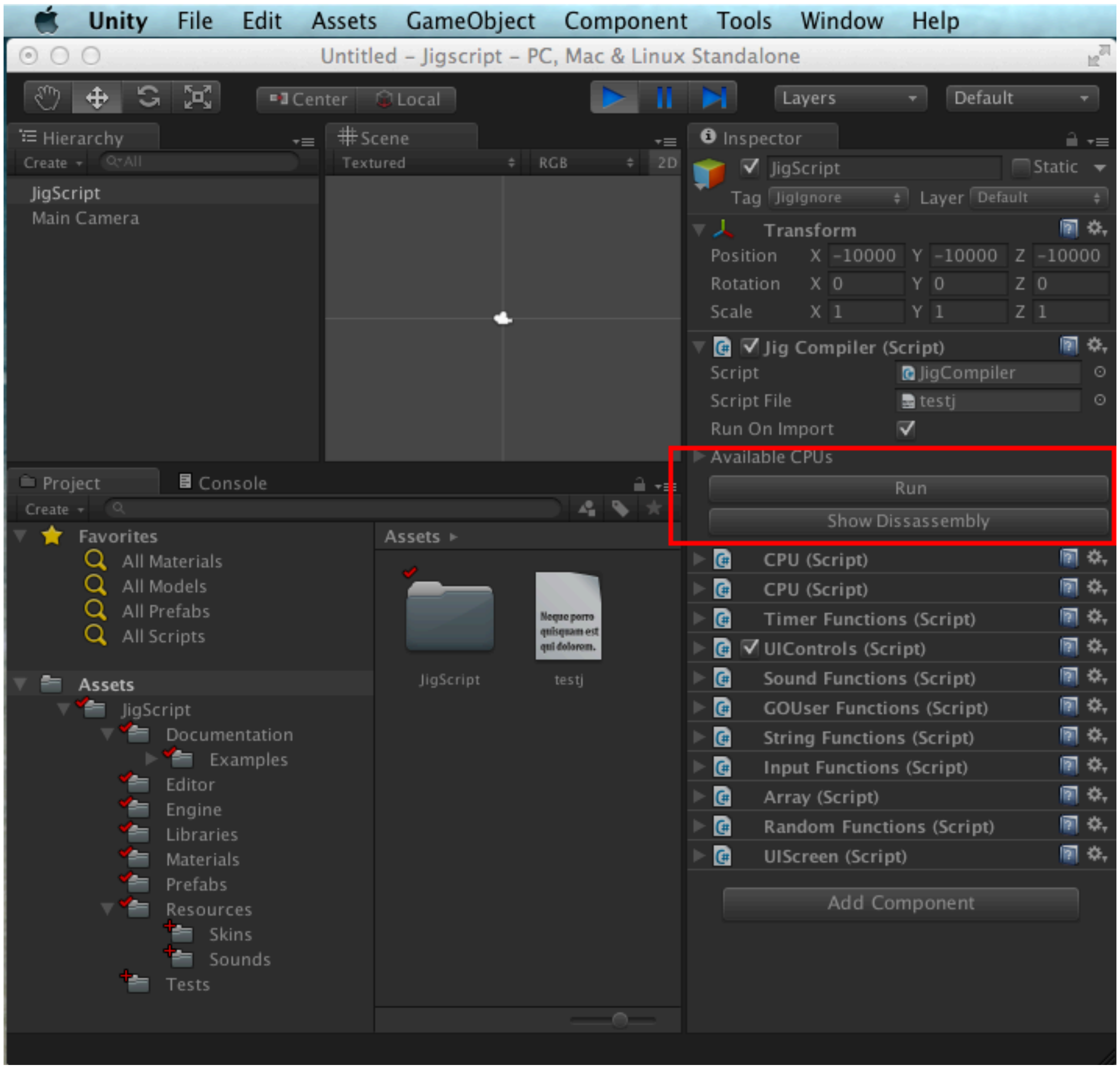


Figure 3, JigScript inspector window in edit mode.

The JigScript Prefab needs to be added to any scene in which you want to run JigScript files.

To run JigScript scripts you must be in play mode. In Play mode 2 buttons are added to the JigScript Inspector window, Run and Disassembly.



JigScript Static
Tag JigIgnore Layer Default

Transform
Position X -10000 Y -10000 Z -10000
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Jig Compiler (Script)
Script JigCompiler
Script File testj
Run On Import ✓
Available CPUs

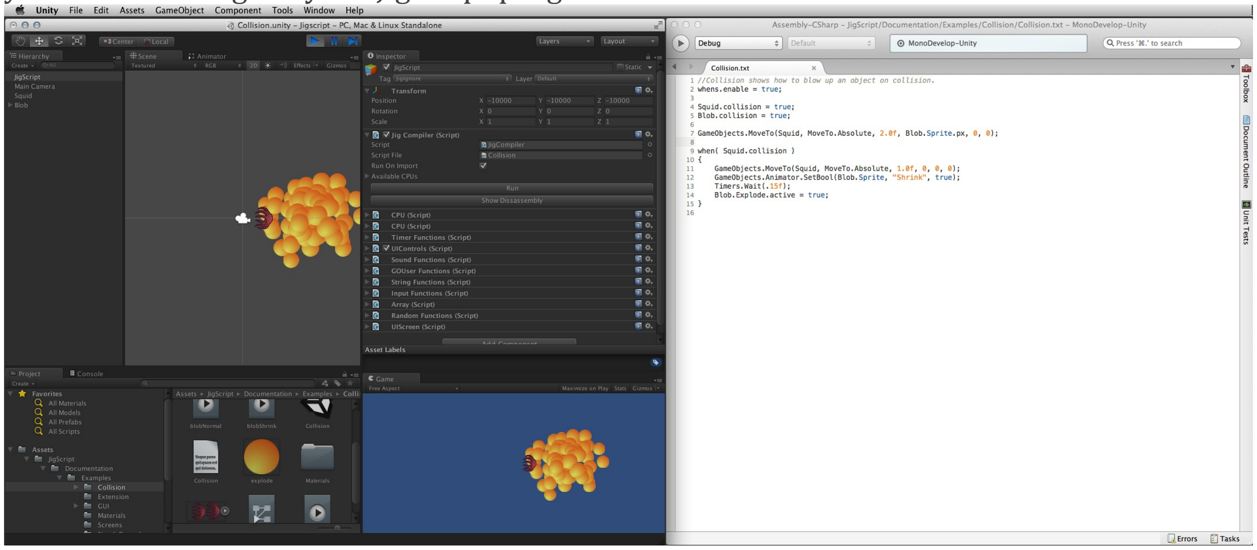
CPU (Script)
CPU (Script)
Timer Functions (Script)
UIControls (Script)
Sound Functions (Script)
GOUser Functions (Script)
String Functions (Script)
Input Functions (Script)
Array (Script)
Random Functions (Script)
UIScreen (Script)

Figure 4, JigScript inspector window in Play Mode.

The Run button performs the same function as Tools→JigScript→Run menu option. It is present in the inspector simply for convenience. The Show Disassembly button is used to show the run time code the compiler generates for your script. Normally you will never need to use this button, it is included here simply in case you are curious as to what the run time code looks like or you want to make your scripts as efficient as possible.

To set the run script you must be in Edit mode. While it can be set in play mode, when you exit play mode all of your settings are restored to their value when you entered play mode. This is standard Unity behavior

The checkbox 'Run on Import,' shown in Figure 4, is special; it is used with the PostProcessJigScript.cs class and runs your JigScript whenever it is updated and your focus changes back to Unity from MonoDevelop. This allows for very quick changing and testing of JigScript applications without the need to click the run button in the inspector every time you make a change to your JigScript program.



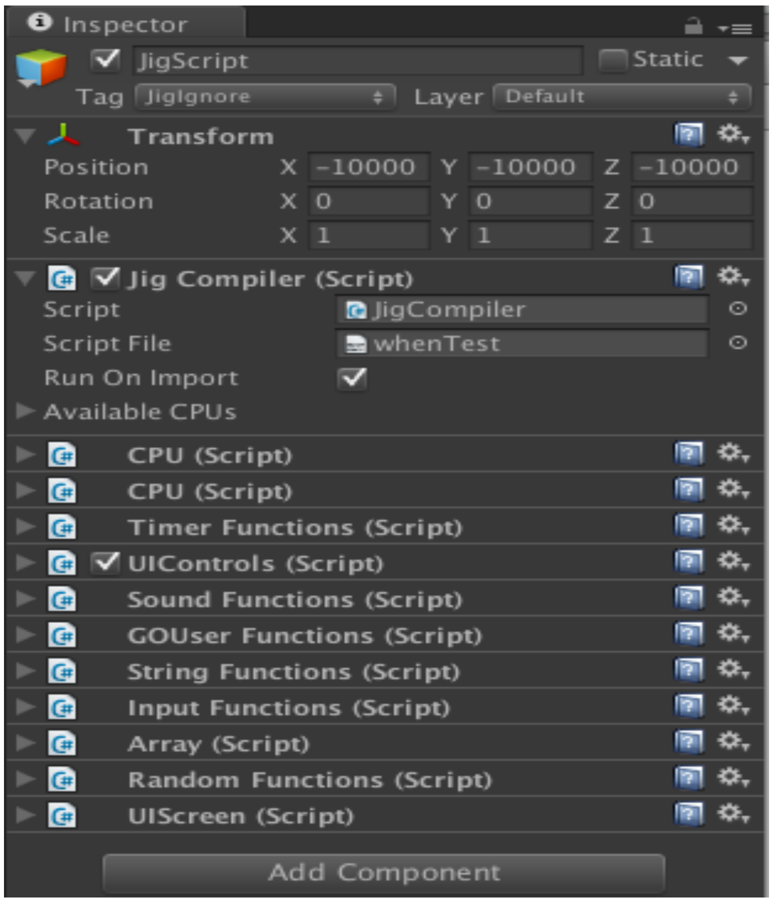
```
//Collision shows how to blow up an object on collision.
whens.enable = true;

Squid.collision = true;
Blob.collision = true;

GameObjects.MoveTo(Squid, MoveTo.Absolute, 2.0f, Blob.Sprite.px, 0, 0);

when( Squid.collision )
{
    GameObjects.MoveTo(Squid, MoveTo.Absolute, 1.0f, 0, 0, 0);
    GameObjects.Animator.SetBool(Blob.Sprite, "Shrink", true);
    Timers.Wait(.15f);
    Blob.Explode.active = true;
}
```

Figure 5. This is JigScript in Unity Play mode showing your Run Script open in Mono develop. You can stay in Play Mode while you add script to your Run Script. You can modify or add script to your game while it is running.



Inspector

✓ JigScript — Static

Tag: JigIgnore — Layer: Default

Transform

	X	Y	Z
Position	-10000	-10000	-10000
Rotation	0	0	0
Scale	1	1	1

✓ Jig Compiler (Script)

Script: JigCompiler

Script File: whenTest

Run On Import: ✓

Available CPUs

CPU (Script)

CPU (Script)

Timer Functions (Script)

✓ UIControls (Script)

Sound Functions (Script)

GOUser Functions (Script)

String Functions (Script)

Input Functions (Script)

Array (Script)

Random Functions (Script)

UIScreen (Script)

Figure 7. The JigScript Inspector Window. The JigCompiler component we have already used to enter our Run Script. The JigCompiler and the compenents below it are run time user function libraries that come with JigScript.

Under the Hood

Figure 7 shows the full Inspector Window for JigScript. The JigCompiler and CPU files contain the compiler and run time software CPU respectively. The remaining scripts are function libraries provided with the JigScript package. The functionality of these libraries are listed here:

UIControls	Simple GUI that works with the Immediate Mode GUI in Unity 3d.
Sounds	Adds functions and properties that play back, check and stop the playing of sounds. Any sounds played with these functions need to be under the **Resources** folder.
GOUser Functions	Adds script-callable functions that allow you to perform operations on game objects.
Timer	Adds script-callable timer wait and time functions.
String	Adds some basic string functionality like Substring and Length.
Input	Adds access to the Unity Input classes Axis and Button functions.

Array	Adds basic array functionality like copy, fill, length and clear.
Random	Addsrandom number generation functions.
UIScreen	Adds screen-processing functionality.

Examples

In the Documentation→Examples folder are several example scenes with associated JigScript files.

Collision	Shows how to detect and explode a game object when another object collides with it.
Extension	Shows how to extend JigScript with your own functions, variables and custom properties.
GUI	Shows how to create and use GUI controls.
Materials	Shows how to change the material of a game object.
Physics	Shows how to use AddForce() and physics in scene where the models already have been setup for physics. GORAF1 shows an example of physics in a scene where the physics is enabled at run time from the JigScript.
Screens	Shows how to get the screen width and height and display the available screen modes.
SimpleConsole	Shows how to use JigScript to create a simple programmable console in your games.
Sound	Shows how to play sounds in your JigScript scripts.

CPUs and Available CPUs

The **CPUs and Available CPUs section** contain CPU classes that run your script's compiled code. You can have any number of CPUs per JigScript object. To add another CPU simply select the CPU.CS class in the JigScript/Engine folder in the JigScript object in your scene. Then, add an entry to the available CPUs by changing the array size. Finally drag the new CPU to the Available CPUs Element. By default JigScript will run your application on CPU 0 however you can program this to occur on a different CPU. The SimpleConsole.cs script that comes with JigScript does this with this statement: *compiler.RunScript(script, 1);* The number 1 says to run on CPU 1 not the default which is CPU 0. This allows for a degree of parallelism between running scripts. Variables either global or local are shared among all CPUs but each CPU contains its own program code and processing logic! *Note: Normally you will never need more than 2 CPUs. The first CPU is used for any scripts that run and the second is used if you want to provide a Console window in your game. JigScript automatically spins up and spins down and lighter weight run time CPU when needed to run your script in the most efficient manner possible.*

The JigScript Language

If you are familiar with C, C++, C# Java or JavaScript then you will be very comfortable with JigScript. The syntax has been kept as close as possible to these languages. This is by design so that when you switch between programming in C# in unity and then switch to creating game scripts in JigScript you will not have to stop and switch mentally between programming languages.

A JigScript program is a series of statements contained in a text file. For example, the following statement displays the text Hello World in the Unity Console window:

```
print("Hello World");
```

The script starts running with the first statement encountered and ends when all of the statements have been run. A JigScript statement starts with a reserved word sometimes known as a command and ends with a semi-colon ; symbol. In the previous example, **print** was the command and the statement ended with semi-colon ;.

There can be any number of statements in a JigScript program. Some statements require that the statements they operate on, to be grouped. Grouped statements are known as a statement block. Statements are grouped with curly braces, like in C , JavaScript and C#. For example, the following code:

```
for(var count=0; count<10; ++count)
{
   print("Hello World");
}
```

prints **Hello World** 10 times to the console by placing the print statement inside a for loop. The reserved word *for*, requires a statement block. *Note: In JigScript statement blocks must* *be explicitly specified. This is the case even for statement blocks that contain only a single statement. This is different than C, C# and C++ which allow single statements to form an implicit statement block which often leads to hard to find errors.*

Like a JigScript program, there can be any number of statements in a statement block and statement blocks can contain other statement blocks if needed.

Comments

Comments are bits of text that are ignored by the compiler. JigScript supports two types of comments; Line and Multi-Line Comments.

Line Comments

Line comments begin with a double forward slash. The compiler ignores everything up to the end of the line when it encounters a line comment.

Example

```
var a; //This is a line comment.
```

Multi-Line Comments

Multi-line comments begin with the characters /* and end with the characters */. If you have programmed in C, C++ or C# you will recognize this style of commenting. When the JigScript compiler encounters /* it ignores everything until it encounters another */. Multi-line comments cannot be nested. This is in keeping with the standards in common use today.

Example

```
var a;
/*
This is a multi-line
comment.
*/
```

Variables

All programming languages provide variables. Variables are named bits of memory in which your scripts store data. Variables in JigScript are, for the most part, type less. They can contain strings, integers, floats, bools, vectors, quaternions, uicontrols even game objects. Conversion between types is handled internally by calling the ConvertTo() and PromoteTo() methods of the Value class. For example:

```
var left = "hello";
var b = 10;

print(left + b);
```

Prints hello10 to the console, because the variable b would first be promoted to a higher class type the string. The resulting string "10" would then be appended to the contents of the variable named left; resulting in "hello10" which would then be printed.

In JigScript, variable names must begin with a letter or an underscore _ character. Case is significant so *Avariable* and *aVariable* are two different variables. Variables must be declared before they can be used. To declare a variable, use the *var*, *local* or *global* keywords. Variable names can contain any number of letters, numbers, periods, or _ characters.

Variables can be initialized in their declaration; otherwise they are initialized to 0 by default.

Arrays

JigScript provides powerful array handling functionality that simplifies many common programming tasks. JigScript arrays are handled slightly differently than you may be used to. In JigScript array elements start at element 1. The array index 0 is special and means, "Apply this operation to the entire array". Index values greater than 0 mean, "Apply this operation to only this element of the array". Variables without the array square braces are interpreted to mean array element [1]. This keeps things simple so that when you need to apply an operation to an entire array you use [0] and when you don't you use variable [element]. *Note: The numeric expression within the square braces does not need to be a value; if it is an expression JigScript will calculate the value from the expression. For example, A[10+20/2+x], JigScript would first calculate the value of 10+20/2+x before calculating which index should be used for the array A.*

This greatly simplifies operations like copying arrays or adding a value to all elements of an array, or even moving all game objects by 10 units to the left. All of these operations are now supported with a single like of code! Even better, all of this code is handled natively in C# so that it runs as fast as possible. In most programming languages if you wanted to copy arrays of the same length, say array1 to array2, you would need to write code similar to the following:

```
for(int ii=0; ii<10; ii++)
{
   array2[ii] = array1[ii];
}
```

In JigScript this operation can be accomplished with a single line of code:

```
array2[0] = array1[0];
```

If array1 is larger than array2, JigScript will automatically increase the size of array2 to mat ch. These same operations work with math operators. The following code would add the n umber 10 to all elements of array2:

```
array2[0] = array2[0] + 10;
```

You can even add one array to another as shown here:

```
array2[0] = array2[0] + array1[0];
```

In JigScript there is no difference between an array and a variable. Unlike C, C++ and C# though, remember, arrays in JigScript start at element 1 like in the LUA scripting language. Arrays are automatically extended when required so you do not need to worry about sizing or dimensioning them. To use a variable as an array add the [] operators after the variable name and provide the index inside the []. For example:

```
var myarray;

myarray[10] = "hello";
```

Creates a variable myarray and then extends it to 11 elements so that it can contain the string hello in element 10.

Pro Tip: You can declare an array to have a specific number of elements and then set all those elements to a value with 2 lines of code as shown here:

```
var myarray[10] = 0;
myarray[0] = "some text";
```

This is very fast as the CPU handles all of the fill code.

Reserved Words

var or local

The *var* and *local* reserved words declare local variables. A local variable is one that is scoped to the statement block and file in which it occurs. A local variable is available to statement blocks that come after its definition and before the statement block ends.

For example:

```
var character.strength = 10;
var character.dexterity = 5;
var character.endurance = 0;

if ( character.strength == 10 )
{
   var endurance = 3;
   if ( character.dexterity == 5 )
   {
      endurance = 10; //line 7
   }
   character.endurance = endurance;
}
print(character.endurance)
```

The local variable **endurance** declared in line 7 is available in the statement block that follows the if (character.dexterity == 5) statement. However, the endurance variable is not available outside of the statement block belonging to the if (character.strength == 10) block.

global

The global key word defines variables that are available everywhere. The variable declaration statement must be encountered before the variable is used. This is true even for global variables and is the same behavior as other programming languages.

Scoping

Variable scope indicates the level of code from which they are accessible. Scope itself refers to the code grouping that begins with an { opening bracket and ends with a } closing bracket.

Scoping in JigScript works much the same way it does in C#. Consider the following example script:

```
var a;

fun foo(x, y)
{
  var b;

  if (true)
  {
      var c;
  }
}
```

In this example:

- var a is available everywhere in this file after a.
- var b is available everywhere inside the function foo, including the if statement.
- var c is only available inside the if statement.

Access operators

In JigScript all variables are public, however you can simulate the effect of public and private by placing your local variable definitions in separate files. Local variables are only available within the file in which they are defined.

Promotions and Type Conversions

JigScript is a somewhat type less language. If two components of an operation are involved, JigScript uses a set of rules similar to other programming languages to make the conversion. These rules in the order of evaluation are shown here:

1. If an operation involves two operands and one of them is of type string, the other one is converted to string.
2. If an operation involves two operands and one of them is of type float, the other one is converted to float.
3. If an operation involves two operands and one of them is of type int, the other one is converted to int.
4. Empty variables, "variables set to the reserved word empty" are converted to the type of the other component if the other component is string, float or int. If both components are empty an error is generated.
5. If the type is a complex type, GameObject, Material, Vector, Extension Function, Script Function or UIControl the conversion fails and an error is generated.

Note: Unlike C# bools are promoted to int, float or strings.

For example:

```
var a;
a = “Hello” + 1;
print(a);
```

Would convert the number 1 to a string before appending it to the string “Hello”. This results in the string “Hello1” being stored in variable a.

Forcing a conversion type

In some cases you may find it necessary to force a specific conversion of one type to another. In script you can force this conversion by applying the appropriate conversion property. There are 3 conversion properties that are supported:

toInt	**Converts the variable to an integer type. If the variables value cannot be converted, it is set to 0. For example:** **var a = “10”;** **print(a.toint);**
toFloat	Converts the variable to a float type. If the variable cannot be converted it is set to 0.0. For example: var a = “10.4”; print(a.toFloat);

toString	Converts the variable to a string type. This conversion always succeeds. For example:
	
	var a = 11.34;
	print(a.toString);

Ignored GameObject names On Import

When you press the play button in Unity or when your application starts running in the Unity player, JigScript scans your current scene for game objects. Any game object that has the tag JigIgnore and any child objects of an object set to JigIgnore are ignored. Some game objects and prefabs are Unity system objects. JigScript also ignores objects with these names:

Pro Tip: Placing game objects as a child of the JigScript prefab, or Setting the gameobject tag to JigIgnore will cause them to be ignored by the compiler. This allows you to add game objects to your scene that are not imported to JigScript variables.

HandlesGO	Cylinder	cylinder
PreviewMaterials	Cone	cone
dial_plane	Sphere	sphere
dial_arrow	Cube	cube
JigScript	Torus	torus
Preview Camera	DefaultGeneric	dial_flat
arrow	root	

Functions

fun [functionName] (parameters, ...) {}

Script functions are declared with the keyword fun. A function allows you to group pieces of your script together and call it from other parts of your script. Functions must be declared before they are used. All functions are global in scope. JigScript functions are declared in the following way:

```
fun functionName([parameters], ...)
{

}
```

To call a function, simply write its name and give it the parameters specified in its declaration like so:

```
functionName(parameter1, parameter2);
```

Function parameters are always local variables and they are scoped to the function in which they occur. One difference between JigScript and other programming languages is that parameter values are always passed by value not reference. This means that you need to use the return keyword if you want to return a value to the caller of the function. For example, the following code demonstrates how to update a passed in parameter value and return the updated value:

```
fun Add10(myParameter)
{
   myParameter = myParameter + 10;

   return myParameter;
}
```

The function's caller could then update their parameter as shown here:

```
var myParam = 10;
myParam = Add10(myParam);
```

Conditional Statements

if (condition) {}

If statements make decisions based on the evaluation of the Boolean condition that was passed to it. If the condition passed into the if statement evaluates to true, then the statements within the { open and } close brackets after the if statement are executed. If the condition evaluates to false, the statements within the brackets are skipped.

Example

```
var a = 1;
var b = 2;

if(a < b)
{
    a = b;
}
```

else {}

Else is an optional keyword which pairs with if statements. If an else statement follows an if statement, the statements inside the brackets of the else will be executed when the condition evaluates to false.

Example

```
var a = 4;
var b = 3;

if(a < b)
{
    print("a is less than b");
}
else
{
    print("a is not less than b");
}
```

else if (condition) {}

You can add else if conditions after the initial if statement. This allows you to check for a series of conditions.

Example

```
var a = 10;

if ( a == 1 )
{
   print("Hello 1");
}
else if (a == 10 )
{
   print("Hello 10");
}
else
{
  print("Hello");
}
```

when (condition) {}

When statements make decisions based on the evaluation of a Boolean condition. As soon as the condition passed into the when statement evaluates to true, the statements within the { open and } close braces in the when statement are executed. While the condition evaluates to false, the statements within the brackets are skipped. When statements are re-evaluated each time an element of the condition has been modified. When statements combine the functionality of a rule based language with the syntax of a procedural language. *Note: If a when statement contains a call to an external user function, then the condition in the when statement is evaluated periodically instead. The reason for this is that the JigScript CPU has no way of knowing ahead of time if the external function changed something. Normally this is not a problem as when statements are very efficient, even when there are lots of whens. However, in some cases you may want more control over when a when statement is triggered. In this case you can use an update() statement to create a timed polling loop as shown in Example3.*

When statements are not on by default. To activate them you need to set the **whens.enable** variable to true. You can turn when processing on or off by setting this variable to true or false as required by your JigScript program. This allows you to setup your programs state before activating when statement processing. The **whens.enable** variable is set to false each time your script is run.

Example 1

```
whens.enable= true;

for(var i=0; i<10; ++i)
{
   Timers.Wait(.1f);
}

//prints the numbers 0 to 9 to the console one every .1 second

when(i < 10)
{
   print(i);
}
```

Example 2

```
var button1 = “Click Me”;

UIControls.Button(button1, 10, 10, 100, 100);

whens.enable = true;

when( UIControls.IsClicked(button1) )
{
   print(“Clicked!”);
}
```

Example 3

```
var button1 = “Click Me”;
var buttonClicked;

UIControls.Button(button1, 10, 10, 100, 100);

Update(.1f)
{
   buttonClicked = UIControls.IsClicked(button1);
}

whens.enable = true;

//Called a maximum of 1 time every .1 second.
when( buttonClicked )
{
   print(“Clicked!”);
}
```

Pro Tip: When statements are processed in Coroutine software CPUs. This makes them very efficient because they run in parallel. Using variables in the when condition is even more efficient as the Software CPU knows exactly when a variables value changes. In this example, we show you how to parallelize filling an array by using 2 when statements.

Example 4

```
var a;
a[100] = 0;
a[0] = 100;

var first = false;
var second = false;
var done = false;

whens.enable = true;
```

```
first = true;
second = true;

var start = Timers.Time();

when(first)
{
   first = false;
   for(var n=0; n<50; ++n)
   {
      a[n] = "line " + n;
   }
}

when(second)
{
   second = false;
   for(var n=50; n<100; n++)
   {
      a[n] = "line " + n;
   }
   done = true;
}

when(done)
{
   done = false;
   print(Timers.Time() - start);
   print(a[0]);
}
```

Iteration Statements

for (initializer; condition; iterator) { }

The for statement, creates a loop that runs a statement group repeatedly until the specified condition evaluates to false. A for loop contains 3 parts separated by semicolons; an initializer, a condition, and an iterator.

The initializer is an expression that is executed before the condition is checked for the first time. Typically this is used to initialize a variable used for iteration. Variables can be declared and initialized inside the initializer expression.

The condition is a Boolean expression that is evaluated to determine if the loop should exit or continue. The loop will continue iterating until this condition evaluates to false.

The iterator section determines what happens after the each iteration of the loop. Typically this is used to increment the variable declared in the initialization section.

Example

```
var example = 0;

//prints a multiple of 5, ten times
for(var i = 0; i < 10; ++i)
{
    example = example + 5;
    print(example);
}
```

while (condition) { }

The while loop runs a statement or a group of statements repeatedly until a specified condition evaluates to false.

The condition is a Boolean expression that is evaluated to determine if the loop should exit or continue. The while loop will continue iterating until the condition evaluates to false.

Example

```
var example = 50;

var iterator = 0;

//prints a multiple of 5, ten times
while(iterator < 10)
{
    print(example);
    example = example - 5;
    ++iterator;
}
```

repeat (expression) {}

Deprecated: Placing repeat() inside a when or update statement allows repeats to be run in parallel. Avoid using, repeat() will be removed in a future version of JigScript. It remains here purely for backwards compatibility.

Repeat takes an expression that is the number of times to repeat something and then does it as efficiently as possible. Repeat is much faster than creating for or while loop as script variables are not used for counters. Instead all of that happens behind the scenes in the CPU. The caveat is that repeat () cannot be nested. To get the current iteration count you can access the global **repeat.count** variable, which is updated on each pass, though the statement block that is attached to the repeat loop.

Example

```
//prints 1-10 to the console
repeat(9)
{
   print(1 + repeat.count);
}
```

update (time) {}

Update sets up a function that is called every time seconds. The update statement cannot be nested. The time value must be a float number that is greater than .005 seconds. Update statements are run in parallel like when statements.

Example

```
//prints to the console once per second
update(1.0f)
{
   print("Print this line once per second.");
}
```

Flow of Control Statements

break

The break statement causes an immediate exit of the current statement block in which it occurs. Normally you would use break to exit a loop early.

Example

```
//prints 1-5 to the console before exiting the loop early.
for(var i = 0; i < 10; ++i)
{
    if(i > 4)
    {
        break;
    }

    print(i + 1);
}
```

continue

The continue statement skips to the condition evaluation section of the closest enclosing loop statement.

Example

```
//prints 1, 2, three, 4, 5 to the console.
for(var i = 0; i < 5; ++i)
{
    if(i == 2)
    {
        print("three");
        continue;
    }

    print(i + 1);
}
```

return [expression];

The return statement is used inside of a script function to return to the caller of a script function. The return statement also allows for an optional expression to be evaluated and its result returned to the caller.

Example 1

```
var example = "returnEarly";

//prints to the console and returns before executing further statements in the function
fun foo()
{
    if(example == "returnEarly")
    {
        print("This will be printed.");
        return;
    }

    print("This will not be printed.");
}

foo();
```

Example 2

```
fun foo(a, b)
{
    return a + b;
}
```

```
//prints 4 to the console
print(foo(2, 2));
```

yield

The yield statement causes your script to yield execution back to unity. The JigScript engine handles this automatically for most cases, but large looping scripts with a high number of instructions per yield may find situations in which using yield may be useful for performance.

Example

```
//continuous loop that gives control back to unity so that the IDE does not stall.
while(true)
{
   yield;
}
```

Import Statement

import "scriptName"

The import statement is used to add other script files to the current script. This is similar to how libraries are incorporated and used in other programming languages. The script file to be added must be in the Assets\Resources folder or a sub folder of Assets\Resources.

Example 1

```
//imports the BaseLibrary.txt file that is stored in the Assets\Resources folder
import "JigScript/Examples/Invaders/defines";
```

Operators

Like C, C++ and C#, JigScript is known as operator rich. This means that it defines a large number of basic operations that can be performed. Operations with a higher precedence are performed before operations with lower precedence. The exception to this rule is grouping operations. For example:

```
var a = 100 * 3 - 2;
print(a);
```

Would print the value 298 to the console because the precedence of multiply (30) is greater than subtraction (26). So the expression is evaluated as:

100 * 3 = 300
300 – 2 = 298

Parentheses are used to change the order of operations. For example:

```
var a = 100 * (3 - 2);
print(a);
```

Would print 100 to the console because the expression inside the parentheses is evaluated before the multiplication.

The pre and post increment and decrement operations deserve special mention. These are unary operations and only work on variables. Pre increment or decrement operations update the value of the variable to which they are attached. There cannot be any spaces between the operation and the variable. For example, ++A would add one to variable A before variable A is used. This is known as a pre-increment operation. A++ would add one to variable A after the value in variable A is used. Note: The pre and post operations also work on arrays, so that ++A[0] would add 1 to all of the elements of array A.

JigScript defines the following operators:

Symbol	Operation	Precedence	Description
++	Pre-increment	100	Increment the variable to the right by one before evaluating the expression.
--	Pre-decrement	100	Decrement the variable to the right by one before evaluating the expression.
*	Multiply	30	Multiply the left and right values together.
/	Divide	29	Divide the value on the left by the value on the right.
%	Modulo	28	Determine the remainder of an integer division of the value on the right with the value on the left. This will force the values to integer type.
+	Addition	27	Add the number on the right from the number on the left.

-	Subtraction	26	Subtract the number on the right from the number on the left.
<<	Shift Left	23	Shift number on right by number on left.
>>	Shift Right	23	Shift number on right by number on left.
<	Less Than	21	Compare the value on the right to the value on the left.
>	Greater Than	21	Compare the value on the right to the value on the left.
>=	Greater Than or Equal To	21	Compare the value on the right to the value on the left.
<=	Less Than or Equal To	21	Compare the value on the right to the value on the left.
==	Equal To	21	Compare the value on the right to the value on the left.
!=	Not Equal To	21	Compare the value on the right to the value on the left.
&	Bitwise And	15	And the left and right value together.
|	Bitwise Or	14	Or the left and right value together.
^	Bitwise Exclusive or	13	Exclusive or the left and right value together.
&&	Logical And	12	Combine the conditional expression on the left with the conditional expression on the right.
||	Logical Or	11	Combine the conditional expression on the left with the conditional expression on the right.
,	Parameter Separator	10	Separate parameters sent to a user-defined function.
=	Assignment	9	Assigns the value of an expression to a variable.
+=	Assignment	9	Add the value of the right side expression from the variable and assigns the result to the variable on the left. Example, v += .1f; same as v = v + .1f;
-=	Assignment	9	Subtracts the value of the right side expression from the variable and assigns the result to the variable on the left. Example, v -= .1f; same as v = v - .1f;
*=	Assignment	9	Multiplies the value of the right side expression from the variable and assigns the result to the variable on the left. Example, v *= .1f; same as v = v * .1f;
/=	Assignment	9	Divides the value of the right side expression from the variable and assigns the result to the variable on the left. Example, v -= .1f; same as v = v / .1f;
%=	Assignment	9	Mods the value of the right side expression from the variable and assigns the result to the variable on the left. Example, v %= 1; same as v

			= v % 1;
&=	Assignment	9	Bitwise ands the value of the right side expression from the variable and assigns the result to the variable on the left. Example, v &= 1; same as v = v & 1;
|=	Assignment	9	Bitwise ors the value of the right size expression from the variable and assigns the result to the variable on the left. Example, v |= 1; same as v = v | 1;
^=	Assignment	9	Bitwise exclusive ors the value of the right side expression from the variable and assigns the result to the variable on the left. Example, v ^= 1; same as v = v ^ 1;
(	Begin Arithmetic Grouping	8	Change the grouping precedence of a set of expressions.
)	End Arithmetic Grouping	7	Change the grouping precedence of a set of expressions.
[	Open Bracket	6	Begin the subscript for an array.
]	Close Bracket	5	End the subscript for an array.
;	Statement Terminator	3	Ends the statement. For example, a = a + 1;
{	Begin Statement Group	2	Begins a statement group, sometimes a group is referred to a block or block of statements.
}	End Statement Group	1	Ends a statement group, sometimes a group is referred to a block or block of statements.
++	Post Increment	0	Add 1 to the preceding variable after the expression is evaluated.
--	Post Decrement	0	Subtracts 1 from the preceding variable after the expression is evaluated.

Library User Functions

JigScript comes with a rich assortment of library functions that you can use in your scripts. You can also add your own specially written function libraries. If you are interested in this feature see the section on Extending JigScript.

GameObject Properties & User Functions

The GameObjects that are provided with JigScript are defined in the GOUserFunctions.cs file. This file is located in Assets\JigScript\Libaries folder. To use any of these functions drag and drop the GOUserFunctions.cs file onto the JigScript game object. *Note: If you use the JigScript prefab these functions are added for you automatically.*

Properties

The following properties are present on all GameObject variables:

.px
Set or Get the location of the game object on the X axis.

Example
```
//Move a cube named fcube across the screen.
for(var ii=1.0f; ii<3.0f; ii = ii + .1f)
{
   fcube.px = ii;
}
```

Pro Tip: You can use the GameObjects.MoveTo, GameObjects.ScaleTo and GameObjects.RotateTo to move game objects as well. These functions give you more precise control than you can get by using a simple loop.

.py
Set or Get the location of the game object on the Y axis.

.pz
Set or Get the location of the game object on the Z axis.

.sx
Set or Get the scale of the game object on the X axis.

.sy
Set or Get the scale of the game object on the Y axis.

.sz
Set or Get the scale of the game object on the Z axis.

.rx
Set or Get the rotation of the game object on the X axis.

.ry
Set or Get the rotation of the game object on the Y axis.

.rz
Set or Get the rotation of the game object on the Z axis.

.vx
Set or Get the game object's vector on the X axis.

.vy
Set or Get the game object's vector on the Y axis.

.vz
Set or Get the game object's vector on the Z axis.

.dragBegin

Gets the dragBegin state of the game object. If the gameobject is not being dragged and it starts to be dragged then this value will return true. In order for this property to return the dragBegin state the clickable property must be set to true.

Example

```
Sprites.A40.clickable = true;
whens.enabled = true;
when( Sprites.A40.dragBegin)
{
   print("game object is being dragged");
}
```

.collision

Sets the collision checking state of a game object or returns the collision state of the game object. The collision property must be set on objects before any collisions will be detected.

Example:

```
//Collision shows how to blow up an object on collision.
whens.enable = true;

Squid.collision = true;
Blob.collision = true;

GameObjects.MoveTo(Squid, MoveTo.Absolute, 2.0f, Blob.Sprite.px, 0, 0);

when( Squid.collision )
{
   GameObjects.MoveTo(Squid, MoveTo.Absolute, 1.0f, 0, 0, 0);
   GameObjects.Animator.SetBool(Blob.Sprite, "Shrink", true);
   Timers.Wait(.15f);
   Blob.Explode.active = true;
}
```

.selected

Returns true when the mouse is pressed on a game object. In order for this property to work you first need to set the.clickable property to true.

Example

```
whens.enable = true;

clickie.clickable = true;
when( clickie.selected )
{
   print("selected");
}
```

.material

Sets the game object's material. This property can only be set, it is write only.

Example

```
ACube.material = green;
Timers.Wait(1.0f);
ACube.material = blue;
```

.other

Gets the game object that this game object has collided with. *Note: The game object must have the collision property set to true to enable collision checking before collisions will work for the game object.*

Example 1

```
whens.enable = true;

print("checking collision");
cubef.collision = true;
cubey.collision = true;

GameObjects.MoveTo(cubef, MoveTo.Absolute, 1.0f, 6, 0, 0);

when( cubef.collision )
{
   print(cubef.other);
}
```

Pro Tip: if you pass in an array[0] then any collision that happens for any gameobject in the array will return true.

Example 2

```
Rewards[0].clickable = true;

when(Rewards[0].clickable)
{
   print(Rewards[0].other);
}
```

.dragging
Gets the dragging state of the game object contained in the variable. This property is read only.

Example
```
whens.enable = true;

cubef.clickable = true;
when( cubef.dragging )
{
   print("Cube F is being dragged.");
}
```

.dragStart
Gets the dragStart state of the game object. This property is read only.

Example
```
whens.enable = true;

cubef.clickable = true;
when( cubef.dragStart)
{
   print("dragging has started.");
}
```

.dragEnd
Gets the dragEnd state of the game object. This property is read only.

Example
```
whens.enable = true;

cubef.clickable = true;
when( cubef.dragEnd)
{
   print("dragging has ended.");
}
```

.hover
Get the hover or mouse over state for the game object. This property is read only. The specified game object must first be enabled to receive hover events by setting the clickable property to true.

Example
```
cubef.clickable = true;
```

```
when( cubef.hover )
{
   print("The mouse is over cube F.");
}
```

.clickable

Sets the clickable state of the game object. This field must be set to true in order to enable the dragBegin, selected, dragging, dragEnd and hover properties.

.active

Sets the show state for the game object. If true the game object is shown. If false the game object is hidden.

Example 1

```
Sprites.A40.active = true;
```

Pro Tip: You can show or hide a collection of game objects by using an array and passing in the [0] element.

Example 2

```
var g1;
g1[1] = GameObjects.Create("MyGameObject");
g1[2] = GameObjects.Create("MyGameObject");
g1[3] = GameObjects.Create("MyGameObject");
g1[4] = GameObjects.Create("MyGameObject");

//Hide all game objects in the array.
g1[0].active = false;
Timers.Wait(1.0f);

//Show all game objects after waiting.
g1[0].active = true;
```

.physics

Prepares a game object variable so that it acts like a physical object. This basically means that the rigidbody and collider are added to the game object if they are not already present. There is no need to use this property on game object variables if they have already been setup in the scene so that they have rigidbody and collider.

.MoveComplete

This property is set to true when the movement on a game object is completed. If this property is applied to the [0] element of an array of game objects then, all game objects moves need to be completed before this property will return true.

.ScaleComplete

This property is set to true when the scale on a game object is completed. If this property is applied to the [0] element of an array of game objects then, all game objects scales need to be completed before this property will return true.

.RotateComplete

This property is set to true when the rotation on a game object is completed. If this property is applied to the [0] element of an array of game objects then, all game objects rotates need to be completed before this property will return true.

.toInt

Converts the variable to an integer type. If the variable value cannot be converted, it is set to 0.

Example

```
var a = "10";
print(a.toint);
```

.toFloat

Converts the variable to a float type. If the variable cannot be converted it is set to 0.0.

Example

```
var a = "10.4";
print(a.toFloat);
```

.toString

Converts the variable to a string type. This conversion always succeeds.

Example

```
var a = 11.34;
print(a.toString);
```

GameObjects.Create

The GameObjects.Create function creates a new instance or clone of an existing game object in the scene hierarchy.

Definition

GameObjects.Create(Variable)

Parameter	Description
Variable	GameObject to be created.

Returns

GameObject type variable

Example

```
//This example creates a new instance of a game object named Example in the scene heirarchy.

var newGameObject = GameObjects.Create(Example);
```

GameObjects.Destroy

The GameObjects.Destroy function deletes the specified game object from the scene hierarchy.

Definition

GameObjects.Destroy(Variable)

Parameter	Description
Variable	GameObject to be destroyed.

Example

```
//This example creates a new instance of a game object named Example in the scene heirarchy, and then destroys it.

var newGameObject = GameObjects.Create(Example);

GameObjects.Destroy(newGameObject);
```

GameObjects.MoveTo

GameObjects.MoveTo interpolates the location of the referenced game object over a specified time to the specified coordinates.

Definition

GameObjects.MoveTo(Variable, Locality, Time, X, Y, Z)

Parameter	Description
Variable	GameObject to be moved.
Locality	If set to MoveTo.Relative, the vector supplied in X, Y and Z is added to the current location of the GameObject. If set to MoveTo.Absolute then the GameObject is moved to the location in world coordinates specified in X, Y and Z.
Time	Time in seconds it takes to arrive at the specified location.
X	Specified location to move to on the X axis.
Y	Specified location to move to on the Y axis.
Z	Specified location to move to on the Z axis.

Example 1

```
//This example translates the created game object newGameObject from its current location
to the coordinates (1, 2, 3) over 2 seconds.

var newGameObject = GameObjects.Create(Example);

var moveTime = 2.0f;
var xCoord = 1;
var yCoord = 2;
var zCoord = 3;

GameObjects.MoveTo(newGameObject, MoveTo.Relative, moveTime, xCoord, yCoord,
                   zCoord);
```

Example 2

```
//This example translates the game object Example from the scene heirarchy from its current
location to the coordinates (0, 1, 2) immediately.

var moveTime = 0.0f;
var xCoord = 0;
var yCoord = 1;
var zCoord = 2;

GameObjects.MoveTo(Example, MoveTo.Absolute, moveTime, xCoord, yCoord, zCoord);
```

GameObjects.ScaleTo

GameObjects.ScaleTo interpolates the scale of the referenced game object over a specified time to the specified scale.

Definition

GameObjects.ScaleTo(Variable, Locality, Time, X, Y, Z)

Parameter	Description
Variable	GameObject to be scaled.
Locality	If set to ScaleTo.Relative, the scale supplied in X, Y and Z is added to the current scale of the GameObject. If set to ScaleTo.Absolute then the GameObject is scaled to the values specified in X, Y and Z.
Time	Time in seconds it takes to scale the specified GameObject.
X	Specified scale factor on the X axis.
Y	Specified scale factor on the Y axis.
Z	Specified scale factor on the Z axis.

Example

```
//This example interpolates the scale of the created game object newGameObject from its current size to (2, 2, 2) over 5 seconds.

var newGameObject = GameObjects.Create(Example);

var scaleTime = 5.0f;
var xScale = 2;
var yScale = 2;
var zScale = 2;

GameObjects.ScaleTo(newGameObject, ScaleTo.Absolute, scaleTime, xScale, yScale, zScale);
```

GameObjects.RotateTo

GameObjects.RotateTo interpolates the rotation of the referenced game object over a specified time to the specified rotation.

Definition

GameObjects.RotateTo(Variable, Locality, Time, X, Y, Z)

Parameter	Description
Variable	GameObject to be rotated.
Locality	If set to RotateTo.Relative, the rotation supplied in X, Y and Z is added to the current rotation of the GameObject. If set to RotateTo.Absolute then the GameObject is rotated to world orientation as specified in X, Y and Z.
Time	Time in seconds it takes to rotate the specified GameObject.
X	Specified rotation as Euler angles in degrees on the X axis.
Y	Specified rotation as Euler angles in degrees on the Y axis.
Z	Specified rotation as Euler angles in degrees on the Z axis.

Example

```
//This example interpolates the rotation of the created game object newGameObject 360
degrees over 2 seconds.

var newGameObject = GameObjects.Create(Example);

var rotationTime = 2.0f;
var xRotation = 0;
var yRotation = 0;
var zRotation = 360;

GameObjects.RotateTo(newGameObject, RotateTo.Absolute, rotationTime, xRotation,
                     yRotation, zRotation);
```

GameObjects.IsMoveComplete

Deprecated:

GameObjects.IsMoveComplete returns a bool that determines whether the specified game object is in the middle of a movement or is not moving.

Definition

GameObjects.IsMoveComplete(Variable)

Parameter	Description
Variable	GameObject to be evaluated.

Returns

Returns true if the move have been completed. False if the move is still in progress.

Example

```
//This example rotates the created game object newGameObject after it finishes a movement.

var newGameObject = GameObjects.Create(Example);

var moveTime = 1.0f;
var xPos = 1;
var yPos = 0;
var zPos = 0;

GameObjects.MoveTo(newGameObject, MoveTo.Relative, moveTime, xPos, yPos, zPos);

when( GameObjects.IsMoveComplete(newGameObject))
{
    var xRot = 0;
    var yRot = 0;
    var zRot = 360;

    GameObjects.RotateTo(newGameObject, RotateTo.Relative, moveTime, xRot, yRot, zRot)
;
}
```

GameObjects.IsScaleComplete

GameObjects.IsScaleComplete returns a bool that determines whether the specified game object is in the middle of a scaling operation or is not scaling.

Definition

GameObjects.IsScaleComplete(Variable)

Parameter	Description
Variable	GameObject to be evaluated.

Returns

Returns true if the scale have been completed. False if the scale is still in progress.

Example

```
//This example moves the created game object newGameObject after it finishes a scaling oper
ation.

var newGameObject = GameObjects.Create(Example);

var moveTime = 2.0f;
var xScale = 3;
var yScale = 3;
var zScale = 3;

GameObjects.ScaleTo(newGameObject, ScaleTo.Relative, moveTime, xScale, yScale, zScale);

when( GameObjects.IsScaleComplete(newGameObject))
{
   var xPos = 1;
   var yPos = 0;
   var zPos = 0;

   GameObjects.MoveTo(newGameObject, MoveTo.Relative, moveTime, xPos, yPos, zPos);
}
```

GameObjects.IsRotateComplete

GameObjects.IsRotateComplete returns a bool that determines whether the specified game object is in the middle of rotating or is not rotating.

Definition

GameObjects.IsRotateComplete(Variable)

Parameter	Description
Variable	GameObject to be evaluated.

Returns

Returns true if the rotate have been completed. False if the rotate is still in progress.

Example

```
//This example moves the created game object newGameObject after it finishes rotating.

var newGameObject = GameObjects.Create(Example);

var moveTime = 1.5f;
var xRot = 0;
var yRot = 0;
var zRot = 360;

GameObjects.RotateTo(newGameObject, RotateTo.Relative, moveTime, xRot, yRot, zRot);

when( GameObjects.IsRotateComplete(newGameObject))
{
    var xPos = 1;
    var yPos = 0;
    var zPos = 0;

    GameObjects.MoveTo(newGameObject, MoveTo.Relative, moveTime, xPos, yPos, zPos);
}
```

GameObjects.SetMinDragDistance

GameObjects.SetMinDragDistance sets the minimum allowed distance to count as dragging for a specified game object.

Definition

GameObjects.SetMinDragDistance (Variable, Distance)

Parameter	Description
Variable	GameObject to check whether a click has been registered.
Distance	Minimum distance required to count as dragging.

Example

```
var newGameObject = GameObjects.Create(Example);

GameObjects.SetClickable(newGameObject, true);

var minDistance = 2.0f;

GameObjects.SetMinDragDistance(newGameObject, minDistance);
```

GameObjects.MousePosition

GameObjects.MousePosition returns the position of the mouse cursor in world coordinates on the specified game object.

Definition

GameObjects.MousePosition (Variable)

Parameter	Description
Variable	GameObject to check mouse position on.

Returns

Vector3 variable, use the vx and vy properties to get the specific mouse x or mouse y component.

Example

```
//This example prints the position of the mouse when newGameObject is clicked

var newGameObject = GameObjects.Create(Example);

GameObjects.SetClickable(newGameObject, true);

when(GameObjects.IsClicked(newGameObject))
{
    print(GameObjects.MousePosition(newGameObject));
}
```

GameObjects.Clear

GameObjects.Clear deletes all cloned instances of game objects from the scene hierarchy.

Definition

GameObjects.Clear()

Returns

null

Example

```
var newGameObject = GameObjects.Create(Example);

GameObjects.Clear();
```

GameObjects.Rigidbody.Set

GameObjects.Rigidbody.Set changes the value of any variable within the Rigidbody class.

Definition

GameObjects.Rigidbody.Set(Variable, Parameter, ...)

Parameter	Description
Variable	GameObject to be evaluated.
Parameter	Name of the parameter within the Rigidbody class to set.
...	Value(s) of variables within the specified parameter to set.

Example

```
//This example creates the game object newGameObject, enables it to register collision events and then modifies the isKinematic, constraints, and velocity variables associated with the Rigidbody on newGameObjects.

var newGameObject = GameObjects.Create(Example);

GameObjects.MakeTrigger(newGameObject);

GameObjects.Rigidbody.Set(newGameObject, "isKinematic", true);

GameObjects.Rigidbody.Set(newGameObject, "constraints",
RigidbodyConstraints.FreezeRotationX | RigidbodyConstraints.FreezeRotationY);

var xVelocity = 0;
var yVelocity = 10;
var zVelocity = 0;

GameObjects.Rigidbody.Set(newGameObject, "velocity", xVelocity, yVelocity, zVelocity);
```

GameObjects.Rigidbody.AddForce

Adds a force to the rigidbody attached to the specified game object. As a result the rigidbody will start moving.

Definition

GameObjects.Rigidbody.AddForce(Variable, X, Y, Z)

Parameter	Description
Variable	GameObject to add force to.
X	Amount of force to add on the X axis.
Y	Amount of force to add on the Y axis.
Z	Amount of force to add on the Z axis.

Example

```
//This example creates the game object newGameObject,  and then applies a force in the posit
ive x direction.

var newGameObject = GameObjects.Create(Example);

var xForce = 10;
var yForce = 0;
var zForce = 0;

GameObjects.Rigidbody.AddForce(newGameObject, xForce, yForce, zForce);
```

GameObjects.ComparePositions

Compares 2d or 3d bounding boxes of the specified game objects and returns true or false.

Definition

GameObjects.ComparePositions(Variable1, Variable2)

Parameter	Description
Variable1	First GameObject to compare positions with.
Variable2	Second GameObject to compare positions with.

Returns

Returns true if the positions overlap or false if they do not overlap.

Example

```
//This example, creates two game objects and deletes them when their positions overlap.

var newGameObjectA = GameObjects.Create(Example);

var moveTimeA = 0.0f;
var xPosA = 0;
var yPosA = 0;
var zPosA = 0;

GameObjects.MoveTo(newGameObjectA, MoveTo.Absolute,
                   moveTimeA, xPosA, yPosA, zPosA);

var newGameObjectB = GameObjects.Create(Example);

var moveTimeB = 0.0f;
var xPosB = 4;
var yPosB = 0;
var zPosB = 0;

GameObjects.MoveTo(newGameObjectB, MoveTo.Absolute,
                   moveTimeB, xPosB, yPosB, zPosB);

var moveTimeC = 2.0f;

GameObjects.MoveTo(newGameObjectA, MoveTo.Absolute,
                   moveTimeC, xPosB, yPosA, zPosA);

when(GameObjects.ComparePositions(newGameObjectA, newGameObjectB))
{
  GameObjects.Destroy(newGameObjectA);
  GameObjects.Destroy(newGameObjectB);
}
```

GameObjects.GetDistance

Returns the absolute distance between the specified game objects.

Definition

GameObjects.GetDistance (Variable1, Variable2)

Parameter	Description
Variable1	GameObject to calculate distance from.
Variable2	GameObject to calculate distance to.

Returns

The distance between the game objects.

Example

```
//This example, creates two game objects and prints their distance from each other to the
console.

var newGameObjectA = GameObjects.Create(Example);

var moveTimeA = 0.0f;
var xPosA = 0;
var yPosA = 0;
var zPosA = 0;

GameObjects.MoveTo(newGameObjectA, MoveTo.Absolute,
                                moveTimeA, xPosA, yPosA, zPosA);

var newGameObjectB = GameObjects.Create(Example);

var moveTimeB = 0.0f;
var xPosB = 4;
var yPosB = 0;
var zPosB = 0;

GameObjects.MoveTo(newGameObjectB, MoveTo.Absolute,
                                moveTimeB, xPosB, yPosB, zPosB);

print("Distance between A and B: " + GameObjects.GetDistance(newGameObjectA,
        newGameObjectB);
```

GameObjects.Animator.SetInteger

Sets an Integer property associated with a state of an animation on a game object.

Definition

GameObjects.Animator.SetInteger(Variable, State, Value)

Parameter	Description
Variable	GameObject to set the specified integer on.
State	Name of the animation state to set the specified integer on.
Value	Value to set the specified integer to.

Example

```
var newGameObject = GameObjects.Create(Example);

var exampleValue = 0;

GameObjects.Animator.SetInteger(newGameObject, "exampleState", exampleValue);
```

GameObjects.Animator.SetFloat

Sets a Float property associated with a state of an animation on a game object.

Definition

GameObjects.Animator.SetFloat(Variable, State, Value)

Parameter	Description
Variable	GameObject to set the specified float on.
State	Name of the animation state to set the specified float on.
Value	Value to set the specified float to.

Example

```
var newGameObject = GameObjects.Create(Example);

var exampleValue = 1.0f;

GameObjects.Animator.SetFloat(newGameObject, "exampleState", exampleValue);
```

GameObjects.Animator.SetBool
Sets a Bool property associated with a state of an animation on a game object.

Definition
GameObjects.Animator.SetBool(Variable, State, Value)

Parameter	Description
Variable	GameObject to set the specified bool on.
State	Name of the animation state to set the specified bool on.
Value	Value to set the specified bool to.

Example
```
var newGameObject = GameObjects.Create(Example);

var exampleValue = true;

GameObjects.Animator.SetBool(newGameObject, "exampleState", exampleValue);
```

GameObjects.Animator.GetInteger
Gets an Integer property associated with a state of an animation on a game object.

Definition
GameObjects.Animator.GetInteger(Variable, State)

Parameter	Description
Variable	GameObject to get the integer from.
State	Name of the animation state to get the integer from.

Returns
Returns the current value as contained in the Animation component.

Example
```
var newGameObject = GameObjects.Create(Example);

var exampleInt = GameObjects.Animator.GetInteger(newGameObject, "exampleState");
```

GameObjects.Animator.GetFloat

Gets a Float property associated with a state of an animation on a game object.

Definition

GameObjects.Animator.GetFloat(Variable, State)

Parameter	Description
Variable	GameObject to get the float from.
State	Name of the animation state to get the float from.

Returns

Returns the current value as contained in the Animation component.

Example

```
var newGameObject = GameObjects.Create(Example);

var exampleFloat = GameObjects.Animator.GetFloat(newGameObject, "exampleState");
```

GameObjects.Animator.GetBool

Gets a Bool property associated with a state of an animation on a game object.

Definition

GameObjects.Animator.GetBool (Variable, State)

Parameter	Description
Variable	GameObject to get the bool from.
State	Name of the animation state to get the bool from.

Returns

Returns the current value as contained in the Animation component.

Example

```
var newGameObject = GameObjects.Create(Example);

var exampleBool = GameObjects.Animator.GetBool(newGameObject, "exampleState");
```

JigCompiler, built in Functions

print

Prints contents to the Unity Console window.

Definition

print(Variable, ...)

Parameter	Description
Variable	Variable containing a value to be printed to the console.
...	Additional variable to be printed on the second line of the console.

Example

```
var exampleValue = 5;

var exampleString = "This is the number five: ";
var anotherString = "This is a new line.";

print(exampleString + exampleValue, anotherString);
```

save

Saves the variable to the user preferences file under the key name. *Note: Currently only int, float, bool and string values can be saved.*

Definition

save(Key, Variable)

Parameter	Description
Key	Key name used to save the contents of the variable.
Variable	Variable to be saved.

Example

```
var exampleValue = 2;

save("exampleKey", exampleValue);
```

load

Retrieves the contents of the specified key in the user preferences file. The key must have been previously saved with the save function.

Definition

load(Key)

Parameter	Description
Key	Key name that the variable was saved to with the save() function.

Returns

The saved value of the variable stored at Key.

Example

```
var exampleValue = load("exampleKey");
```

delete

Deletes the value and the key saved with the save() function.

Definition

delete(Key)

Parameter	Description
Key	Key name of saved variable to delete.

Example

```
delete("exampleKey");
```

Random User Function

Random
Returns a random value between the specified range of numbers.

Definition
Random(Min, Max)

Parameter	Description
Min	Minimum random value to be generated.
Max	Maximum random value to be generated.

Returns
int type variable if Min and Max are integers
float type variable if Min and Max are floats

Example
```
var minimum = 2;
var maximum = 9;

var exampleInt = Random(minimum, maximum);

print("This is an integer value between 2 and 9: " + exampleInt);

var minFloat = 0.0f;
var maxFloat = 100.0f;

var exampleFloat = Random(minFloat, maxFloat);

print("This is a float value between 0 and 100: " + exampleFloat);
```

Timers User Functions

Timers.Wait

Suspends script progress for a specified number of seconds.

Definition

Timers.Wait(Time)

Parameter	Description
Time	Amount of time in seconds to wait.
Process whens	If true then when statements will be processed if enabled. This is the default and is usually what is needed. However there are situations in which you may need to stop all processing and simply wait. Setting this parameter to false will cause a true wait which is the same as performing yield return new WaitForSeconds(Time); in Unity 3D.

Returns

null

Example

```
//This example suspends script progress for 1.5 seconds.

var time = 1.5f;

Timers.Wait(time);
```

Timers.Time

Returns the delta time between the start of the script and the current time.

Definition

Timers.Time()

Returns

float type variable

Example

```
//This example prints the delta time to the console five times over five seconds.

for(var i = 0; i < 5; ++i)
{
   print(Timers.Time());

   Timers.Wait(1);
}
```

String User Functions

Strings.Substring

Returns a section of the string type variable specified in the first argument starting at the position specified in the second argument at the length specified in the optional third argument.

Definition

Strings.Substring(Variable, Start, Length)

Parameter	Description
Variable	The string type variable to be modified.
Start	Position in the specified string to begin the substring.
Length	Optional parameter. Specifies the position in the specified string to end the substring.

Returns

string type variable

Example

```
var exampleString = "abcdefg";

var newString = Strings.Substring(exampleString, 3);

//the following function call prints "defg" to the console
print(newString);

//the following function call prints "cde" to the console
print(Strings.Substring(exampleString, 2, 3);
```

Strings.Trim

Removes all leading and trailing white-space characters from the specified string.

Definition

Strings.Trim(Variable)

Parameter	Description
Variable	The string type variable to be trimmed.

Returns

string type variable

Example

```
var exampleString = "   example   ";

//the following function call prints "example" to the console
print(Strings.Trim(exampleString));
```

Strings.TrimEnd

Removes all trailing white-space characters from the specified string.

Definition

Strings.TrimEnd(Variable)

Parameter	Description
Variable	The string type variable to be trimmed.

Returns

string type variable

Example

```
var exampleString = "   example   ";

//the following function call prints "   example" to the console
print(Strings.TrimEnd(exampleString));
```

Strings.TrimStart

Removes all leading white-space characters from the specified string.

Definition

Strings.TrimStart(Variable)

Parameter	Description
Variable	The string type variable to be trimmed.

Returns

string type variable

Example

```
var exampleString = "   example   ";

//the following function call prints "example   " to the console
print(Strings.TrimStart(exampleString));
```

Strings.Length

Returns the total number of characters in the specified string.

Definition

Strings.Length(Variable)

Parameter	Description
Variable	The string type variable to be counted.

Returns

int type variable

Example

```
var exampleString = "example";

//the following function call prints "7" to the console
print(Strings.Length(exampleString));
```

Sounds Properties and User Functions

Properties

The following properties are present on all Sound variables:

.play

Gets or Sets the play state of the sound. Setting this property to true causes the sound to play. Setting this property to false causes the sound to stop playing.

Example 1

```
var longSound = Sounds.Create("JigScript/Sounds/long");

longSound.volume = .5f;
longSound.pitch = 1.0f;
longSound.loop = true;
longSound.play = true;
```

Example 2

```
var longSound = Sounds.Create("Sounds/long");
var shortSound = Sounds.Create("Sounds/short");

longSound.volume = .5f;
longSound.pitch = 1.0f;
longSound.loop = true;
longSound.play = true;
Timers.Wait(1.0f);

whens.enable = true;

for(var ii=0; ii<10; ++ii)
{
   shortSound.play = true;
   Timers.Wait(1.0f);
}

longSound.play = false;

when( shortSound.play == true )
{
   print("Bang!");
}
```

.volume

Gets or Sets the volume that the sound will play at. The range is 0 to 1.0f.

Example

```
var longSound = Sounds.Create("JigScript/Sounds/long");

longSound.volume = .25f;
longSound.pitch = 1.0f;
longSound.loop = false;
longSound.play = true;
```

.loop

Gets or Sets the loop state for the sound. If true then the sound will play continuously until play is set to false. If false then the sound will play once.

Example 1

```
var longSound = Sounds.Create("JigScript/Sounds/long");

longSound.volume = .5f;
longSound.pitch = 1.0f;
longSound.loop = false;
longSound.play = true;
```

Example

.pitch

Gets or Sets the pitch of the sound. Values greater than 1 cause the sound to be played faster, while sounds lower than 1 cause the sound to be played slower.

Example

```
var longSound = Sounds.Create("JigScript/Sounds/long");

longSound.volume = .5f;
longSound.pitch = 2.0f;
longSound.loop = false;
longSound.play = true;
```

Sounds.Create

Loads the specified audio clip and prepares it for playing.

Definition

var sound = Sounds.Create(SoundLocation)

Parameter	Description
SoundLocation	Location of the audio file inside the Resources folder to be loaded.

Returns

Identifier that JigScript uses to identify the audio clip to be played or stopped.

Example

```
var soundLocation = "JigScript/Sounds/example";

var loadedSound = Sounds.Load(soundLocation);
```

Sounds.Destroy

Removes a loaded audio clip from memory.

Definition

Sounds.Destroy(SoundLocation)

Parameter	Description
sound	Previously created sound. This value is returned by the Sounds.Create function.

Example

```
var soundLocation = "JigScript/Sounds/example";

var loadedSound = Sounds.Create(soundLocation);
Sounds.Destroy(loadedSound);
```

Sounds.Stop

Stops the specified audio file from playing.

Definition

Sounds.Stop(Sound)

Parameter	Description
Sound	Location of the audio file inside the Resources folder or AudioClip type variable that has previously been loaded.

Returns

null

Example

```
var musicLocation = "JigScript/Sounds/exampleMusic";

var loadedMusic = Sounds.Load(musicLocation);

var halfVolume = 50;
var looping = true;

Sounds.Play(loadedMusic, halfVolume, looping);

Sounds.Stop(loadedMusic);
```

Input User Functions

Inputs.GetAxis

Returns the value of the virtual axis identified by axisName. The value will be in the range -1...1 for keyboard and joystick input.

Definition

Inputs.GetAxis(Axis)

Parameter	Description
Axis	Name of the virtual axis to retrieve input from.

Returns

float type variable

Example

```
var horizontalMovement = Inputs.GetAxis("Horizontal");
```

Inputs.GetButton

Returns true while the virtual button identified by buttonName is held down.

Definition

Inputs.GetButton(Button)

Parameter	Description
Button	Name of the virtual button to retrieve input from.

Returns

bool type variable

Example

```
var mouseDown = Inputs.GetButton("Fire 1");
```

Inputs.Any

Returns true while any key is pressed.

Definition

Inputs.Any()

Returns

bool type variable

Example

```
whens.enable = true;

when(Inputs.Any())
{
    print("A key was pressed!");
}
```

Inputs.Key

Returns true while the specified key is pressed.

Definition

Inputs.Key(KeyName)

Parameter	Description
KeyName	Name of the key to evaluate

Returns

bool type variable

Example

```
whens.enable = true;

when(Inputs.Key("m"))
{
    print("M key was pressed!");
}
```

UIControls Library Variables, Properties, and User Functions

The UIControls extension library allows you to easily prototype and creates 2D UI controls. These controls can contain sprites or text. If the sprite is animated using frames then the control can even display an animation. *Note: In order to display an animation in a control the game object sprite needs to be active in the scene.* If you are interested in this feature check out the GUIButton example scene in the Documentation/Examples folder.

Location Variables

These variables define the placement of UI Elements in relation to the screen.

Variable	Description
UIControls.XY	Use the X and Y coordinates defined by passed in variables to place the UI Element.
UIControls.Left	Align the UI Element with the left edge of the screen.
UIControls.Right	Align the UI Element with the right edge of the screen.
UIControls.Top	Align the UI Element with the top edge of the screen.
UIControls.Bottom	Align the UI Element with the bottom edge of the screen.
UIControls.CenterHorizontal	Align the UI Element centered horizontally on the screen.
UIControls.CenterVertical	Align the UI Element centered vertically on the screen.
UIControls.CenterBoth	Align the UI Element to the center of the screen.
UIControls.Visible	Allow the UI Element to be visible.
UIControls.NotVisible	Hide the UI Element.
UIControls.TopCenter	Set the UI Element to the top center of the screen.
UIControls.BottomCenter	Set the UI element to the bottom center of the screen.
UIControls.LeftCenter	Set the UI Element to the left vertically centered.
UIControls.RightCenter	Set the UI Element to the right vertically centered.

UIScreen Variables

These variables contain the current screen width and height.

Variable	Description
UIScreen.width	The current width of the screen.
UIScreen.height	The current height of the screen.

Properties

The following properties are present on all UIControl variables:

Property	Description
.index	Gets or Sets the index of the selected element in a list control.
.content	Gets or sets the content being displayed in the control.
.changed	Gets the changed state of a control. This property is updated each time the control is

	changed so that it can be used in a when() statement. For example, when(button.changed) { }
.clicked	Gets the clicked state of a control. This property is updated each time the control is clicked so that it can be used in a when() statement. For example, when(button.clicked) { }

.show	Gets or Sets the visible state of a control. If true then the control is visible if false then the control is hidden. Groups of controls can all be hidden or shown with a single statement by placing the controls in an array and then use the [0] element of the array as shown here:
	
	`var array;`
	
	`array[1] = UIControls.Box(UIControls.CenterVertical, false, "This is a box.", 0, 0, 200, 50);`
	`array[2] = UIControls.Box(UIControls.CenterVertical, false, "This is another box.", 300, 0, 200, 150);`
	
	`//Show all box controls.`
	`array[0].show = true;`
	
	`Timers.Wait(1.0f);`
	
	`//Change the content of all box controls to the picture.`
	`array[0].content = picture;`

UIControls.Button

Make a single press button. The user clicks them and something happens immediately.

Definition

UIControls.Button(Location, [true | false], Content, X, Y, Width, Height, [style])

Parameter	Description
Location	Location variable used to define the placement of the UI Element
Show | Hide	If true then the control is created visible, if false then the control is created hidden. The control may be shown by setting the returned variables show property to true.
Content	GameObject or String type variable, which contains the content to be placed on the button.
X	Location of the upper left corner for the button on the X axis.
Y	Location of the upper left corner for the button on the Y axis.
Width	Width of the button.
Height	Height of the button.
[Style]	Optional style parameter. If used this is the name of the custom style in the Skin to be used to style this Box.

Returns

Button type variable

Example

```
//This example creates a default button with the upper left corner at (2, 3).;

var exampleContent = "example";
var xButton = 2;
var yButton = 3;
var widthButton = 150;
var heightButton = 50;

var newButton = UIControls.Button(UIControls.XY, true, “I am a button”,
xButton, yButton, widthButton, heightButton);
```

UIControls.Edit

Make a single-line text field where the user can edit a string.

Definition

UIControls.Edit(Location, [true | false], Content, X, Y, Width, Height, [style])

Parameter	Description
Location	Location variable used to define the placement of the UI Element.
Show | Hide	If true then the control is created visible, if false then the control is created hidden. The control may be shown by setting the returned variables show property to true.
Content	String type variable that contains the content to be placed on the edit field.
X	Location of the upper left corner for the text entry field on the X-axis.
Y	Location of the upper left corner for the text entry field on the Y-axis.
Width	Width of the text entry field.
Height	Height of the text entry field.
[Style]	Optional style parameter. If used this is the name of the custom style in the Skin to be used to style this Box.

Returns

Edit type variable

Example

```
//This example creates a default edit field with the upper left corner at (4, 6).;

var exampleContent = "exampleEdit";
var xEditField = 4;
var yEditField = 6;
var widthEditField = 100;
var heightEditField = 50;

var newEditField  = UIControls.Edit(UIControls.XY,  true,
exampleContent, xEditField, yEditField, widthEditField, heightEditField);
```

UIControls.Label

Make a text or texture label on screen.

Definition

UIControls.Label(Location, [true | false], Content, X, Y, Width, Height, [style])

Parameter	Description
Location	Location variable used to define the placement of the UI Element.
Show | Hide	If true then the control is created visible, if false then the control is created hidden. The control may be shown by setting the returned variables show property to true.
Content	GameObject or String type variable that contains the content to be placed on the label.
X	Location of the upper left corner for the label on the X-axis.
Y	Location of the upper left corner for the label on the Y-axis.
Width	Width of the label.
Height	Height of the label.
[Style]	Optional style parameter. If used this is the name of the custom style in the Skin to be used to style this Box.

Returns

Label type variable

Example

```
//This example creates a default label with the upper left corner at (0, 1).;

var exampleContent = "example";
var xLabel = 0;
var yLabel = 1;
var widthLabel = 100;
var heightLabel = 100;

var newLabel = UIControls.Label(UIControls.XY,  true,
exampleContent, xLabel, yLabel, widthLabel, heightLabel);
```

UIControls.Toggle

Make a toggle radio button. The user clicks them and it toggles immediately.

Definition

UIControls.Toggle(Location, [true | false], Content, X, Y, Width, Height, [style])

Parameter	Description
Location	Location variable used to define the placement of the UI Element
Show | Hide	If true then the control is created visible, if false then the control is created hidden. The control may be shown by setting the returned variables show property to true.
X	Location of the upper left corner for the button on the X axis.
Y	Location of the upper left corner for the button on the Y axis.
Width	Width of the button.
Height	Height of the button.
[Style]	Optional style parameter. If used this is the name of the custom style in the Skin to be used to style this Box.

Returns

Toggle type variable

Example

```
//This example creates a default toggle radio button with the upper left corner at (2, 3).

var xToggle = 2;
var yToggle = 3;
var widthToggle = 50;
var heightToggle = 50;

var newToggle = UIControls.Toggle(UIControls.XY, true, “Example Content”,
xToggle, yToggle, widthToggle, heightToggle);
```

UIControls.List

Make a scrollable list of items on screen.

Definition

UIControls.List(Location, [true | false], variable, X, Y, Width, Height, [style])

Parameter	Description
Location	Location variable used to define the placement of the UI Element.
Show | Hide	If true then the control is created visible, if false then the control is created hidden. The control may be shown by setting the returned variables show property to true.
Variable	Variable that contains the initial elements for the list control. This array may contain both text and game objects.
X	Location of the upper left corner for the list on the X axis.
Y	Location of the upper left corner for the list on the Y axis.
Width	Width of the list.
Height	Height of the list.
[Style]	Optional style parameter. If used this is the name of the custom style in the Skin to be used to style this Box.

Returns

List type variable

Example

```
//This example creates a default list with the upper left corner at (0, 1).

var array;

repeat(100)
{
    array[repeat.count] = "Line " + repeat.count;
}

var xList = 0;
var yList = 1;
var widthList = 100;
var heightList = 100;

var newList = UIControls.List (UIControls.XY, true,
array[0], xList, yList, widthList, heightList);
```

UIControls.Box

Make a box for frame on screen.

Definition

UIControls.Box(Location, [true | false], Content, X, Y, Width, Height, [style])

Parameter	Description
Location	Location variable used to define the placement of the UI Element.
Show | Hide	If true then the control is created visible, if false then the control is created hidden. The control may be shown by setting the returned variables show property to true.
Content	GameObject or String type variable, which contains the content to be placed on the label.
X	Location of the upper left corner for the box on the X axis.
Y	Location of the upper left corner for the box on the Y axis.
Width	Width of the box.
Height	Height of the box.
[Style]	Optional style parameter. If used this is the name of the custom style in the Skin to be used to style this Box.

Returns

Variable containing the identifier or handle that the UIControls class uses to identify this box control.

Example

```
//This example creates a box control;
UIControls.Box(UIControls.CenterVertical, true, "This is a box.", 0, 0, 200, 50);
```

UIControls.SetSkin

Apply the specified UI skin to all GUI items. All UI skins must be under the Resources folder.

Definition

UIControls.SetSkin(Location)

Parameter	Description
Location	Location of the UI Skin under the Resources folder.

Example

```
var exampleContent = "example";
var xButton = 2;
var yButton = 3;
var widthButton = 50;
var heightButton = 50;

var newButton = UIControls.Button(UIControls.XY, exampleContent,
xButton, yButton, widthButton, heightButton);

UIControls.SetSkin("Resources/exampleSkin");
```

UIControls.Clear

UIControls.Clear deletes all created controls. Normally you will never need to call this as this is called by the compiler when your script it compiled.

Definition

UIControls.Clear()

Example

```
var exampleContent = "example";
var xButton = 2;
var yButton = 3;
var widthButton = 50;
var heightButton = 50;

var newButton = UIControls.Button(UIControls.XY, exampleContent,
xButton, yButton, widthButton, heightButton);

UIControls.Clear();
```

Arrays Property & User Functions

Property

The following property is present on all arrays in JigScript:

Property	Description
.length	Access the length of the array.

Arrays.Length

Returns the length of the specified array. Arrays will be the size of the last element accessed; they are dynamically sized based on need.

Definition

Arrays.Length(Variable)

Parameter	Description
Variable	Name of the array to count.

Returns

int type variable

Example

```
var exampleArray;
exampleArray[10] = 0;

//the following function call prints "11" to the console
print(Arrays.Length(exampleArray));

exampleArray[20] = 0;

//note the same function call now returns "21" to the console
print(Arrays.Length(exampleArray));
```

Arrays.Copy

Copies the content of the source array into the destination array.

Definition

Arrays.Copy(Destination, Source)

Parameter	Description
Destination	Name of the array to copy to.
Source	Name of the array to copy from.

Returns

null

Example

```
var exampleArrayA;
exampleArrayA[11];
//contents of exampleArrayA are 000000000000

var exampleArrayB;
exampleArrayB[5];

var startElement = 0;
var fillValue = 1;

Arrays.Fill(exampleArrayB, startElement, fillValue);
//contents of exampleArrayB are now 111111

Arrays.Copy(exampleArrayA, exampleArrayB);
//contents of exampleArrayA are now 111111000000
```

Arrays.Clear

Reduces an array down to the first element.

Definition

Arrays.Length(Variable)

Parameter	Description
Variable	Name of the array to clear.

Returns

null

Example

```
//declare array
var exampleArray;
//array contents is now 0

var arraySize = 4;
var startElement = 0;
var fillValue = 2;

exampleArray [arraySize];
//array contents is now 0, 0, 0, 0, 0

Arrays.Fill(exampleArray, startElement, fillValue);
//array contents is now 2, 2, 2, 2, 2

Arrays.Clear(exampleArray);
//array contents is now 2
```

Arrays.Fill

Writes the specified value into each element of an array after the specified starting element.

Definition

Arrays.Fill(Variable, StartElement, Value)

Parameter	Description
Variable	Name of the array to fill.
StartElement	Location of the element in the array to begin filling from.
Value	Value to put in each element.

Returns

null

Example

```
//declare array
var exampleArray;
//array contents is now 0

var arraySize = 5;
var startElementA = 0;
var startElementB = 3;
var fillValueA = 1;
var fillValueB = 9;

exampleArray[arraySize];
//array contents is now 0, 0, 0, 0, 0, 0

Arrays.Fill(exampleArray, startElementA, fillValueA);
//array contents is now 1, 1, 1, 1, 1, 1

Arrays.Fill(exampleArray, startElementB, fillValueB);
//array contents is now 1, 1, 1, 9, 9, 9
```

Arrays.Delete

Deletes the specifed element or range of elements from a specified array.

Definition

Arrays.Delete(Variable, StartElement, ElementsToDelete)

Parameter	Description
Variable	Name of the array to delete from.
StartElement	Location of the element in the array to begin deleting from.
ElementsToDelete	Number of elements including the StartElement to delete.

Returns

null

Example

```
//declare array
var exampleArray;
//array contents is now 0

var arraySize = 5;
var startElement = 1;
var fillValue = 1;

exampleArray[arraySize];
//array contents is now 0, 0, 0, 0, 0

Arrays.Fill(exampleArray, startElement, fillValue);
//array contents is now 1, 1, 1, 1, 1

var deleteElement = 1;
var elementsToDelete = 1;

Arrays.Delete(exampleArray, deleteElement, elementsToDelete);
//array contents is now 1, 1, 1, 1
```

Adding your own JigScript Functions, Variables and Custom Properties.

JigScript can be extended with classes and methods that you write in C#. This allows you to add any specific functionality that your project requires. Once the function is added you can call it from your JigScripts like any other function.

Adding callable script functions to JigScript requires 3 steps:

1. Add a class that derives from JigExtension.
2. Inside this class write the methods for the functions you want to add.
3. Over ride the Initialize method and add a call to the compiler's AddFunction method to link your functions code to JigScript.

In the initialize method you can also add any global variables you want to have access to in your JigScripts by calling the Variables.Create method.

Custom properties can also be added by calling the Variables.CreateCustomProperty and providing a Get and Set method.

These steps are described in more detail below:

1. Create a new class file in your project.
2. Open your new class and replace MonoBehaviour with JigExtension. This will allow the Jig Script compiler to property initialize the extensions.
3. Add a using NightPen.JigScript; statement to the namespaces in your new class. This will allow you to use the JigScript functions.
4. If any of your extensions will need access to the JigCompiler create a variable in your class using the following pattern:

   ```
   JigCompiler    jigCompiler;
   ```

   Create your extension functions using the following pattern.

   ```
   IEnumerator MyFunctionName( List<Value> values )
   {
           if ( values.Count != NumberOfValuesRequired )
           {
                Debug.LogError("return value = MyFunction(value1, value2, ...);");
           }
           else
           {
               //Add your code here.
           }
           yield return 0;
   }
   ```
5. After you have added all of the functions you need, add the code for any getter and setter functions for any custom properties your extension will support.

```
public override void Initialize(JigCompiler compiler)
{
}
```

If your extension functions need access to the compiler set the jigCompiler variable to the passed in compiler variable as shown here:

```
public override void Initialize(JigCompiler compiler)
{
      this.jigCompiler = compiler;
}
```

6. Next, add any script variables by calling the Variables class Create function. For example, here is how the Rigidbody CollisionDetectionMode.Continuous variable is created:

   ```
   Variables.Create("CollisionDetectionMode.Continuous",
   new Value((int)CollisionDetectionMode.Continuous,
             "CollisionDetectionMode.Continuous"));
   ```

   The first parameter is the name of the variable as it shows up in the script. The section parameter is the variables initial value. The Value also contains a name of the variable. This is to allow tracing of variables and their values to be simpler.

7. Next, for each function you created that you want to have available in JigScript add a function call to the compilers AddFunction method. For example, to add the MyFunctionsName function:

   ```
   compiler.AddFunction("MyFunction.Name”, MyFunctionsName);
   ```

   The first parameter is the name of the function as it will appear in the JigScript. The second parameter is the function name you created when you added your functions to the class.

8. Add a call to link the getter and setter functions for any custom properties you created.

   ```
   Variables.CreateCustomProperty("propertyName", getterfunction, setterfunction);
   ```

Note: Custom Properties are available for every variable type once they are added. See the section on Custom Properties for more information concerning adding custom properties.

9. Add your new class to a game object in your scene. This can be the JigScript game object if you like. The standard library functions provided with JigScript are added in this manner.

Return Value

You can also return a value from your user function. To do so, call the Add the function of the List<Value> values parameter supplied to your user function. For example to return the string “Tracing…” from the TraceFunction you would:

```
public class MyFunctions : MonoBehaviour
{
   private JigCompiler compiler;

   IEnumerator Trace(List<Value> values)
   {
      if ( values.Count > 0 )
      {
         for(int ii=0; ii<values.Count; ++ii)
         {
            values[0].ConvertTo(Value.ValueType.String);
            Debug.Log(values[0].S);
         }
      }
      values.Add(new Value(“Tracing….”, “TraceFunction”);
      yield return 0;
   }

   void Start()
   {
      this.compiler = GetComponent<JigCompiler>();

      this.compiler.AddFunction("MyFunctions.Trace", Trace, true);
   }
}
```

Parameter Lists

When your function is called it is provided a list of Value types. Value is a class that contains the values provided by the script. For example,

Trace(“line 1”, “line 2”, “line 3”);

Would send the Trace function a list of 3 values. Each value would contain a string. To find out how many values are provided you check the list’s Count property. Each value may in fact me an array; contain custom properties and so on. The simplest way to access these values in your extension function is to use the Variables classes Expand method. This method will expand the variable and pass back a List<Values> to your extension function. The print() function that comes standard with JigScript shows a simple way of getting all of a variables associated values. This code is shown here as well:

```
private IEnumerator PrintFunction( List<Value> values )
{
   string s = string.Empty;
   if ( values.Count<1 )
   {
      Debug.LogError("print(variable, ...);");
   }
   else
   {
      foreach( Value p in values )
      {
         List<Value> expanded = Variables.Expand(p, true);

         for(int ii=1; ii<expanded.Count; ++ii)
         {
            expanded[ii].ConvertTo(Value.ValueType.String);
            s += expanded[ii].S + "\r\n";
         }
      }

      printOutputHandler(s);
   }

   yield return 0;
}
```

Type Conversion

If your function requires a particular type of value, you can call the Value.ConvertTo() method and provide the convert type the you require. Conversion of types follows the same rules as the compiler. If the value is already of the required type no conversion takes place. For example, to ensure the first passed in value is a float type you could call:

```
values[0].ConvertTo(Value.ValueType.Float);
```

Value class

The value class represents a value in the JigScript engine. It has several constructors, operator overloads and values as well as ConvertTo method. The Value properties that will be most usefull to you as an extension writer are:

```
this.index
this.arrayIndex
this.T
this.I
this.F
this.S
this.B
this.G = null;
this.field = 0;
this.isVariable
```

The isVariable, index and arrayIndex fields are used in combination to identify, read and write variables. This allows your extension to read a variables current value by calling the Variables.Read() method and write a new value to a variable by calling the Variables classes Store() method. You should always use Store() to store a value as this method is responsible for determining if a value has changed which lets the when statements know that their conditions need to be checked.

The T field tells you the type of data currently stored in this value. The actual data value is stored in the I integer, F float, S string, B Boolean, G GameObject or Material properties. More than one data value can be used at a time, however you will need to account for this in your extension or custom property code.

Return Values

Your function can also return a value back to the script. To return a value simply create a new Value() class instance containing the value you wish to return and add it to the parameter list of values passed to your user function.. For example:

```
values.Add(new Value(1.0f, "Trace.Return"));
```

Considerations for When Statements

When your extension function is called, processing for whens stops until your function returns. In most cases this is what you want to have happen because the CPU has effectively transferred control to your extension function. However, there are also times when you want when processing to continue even while your extension function is executing. This is the purpose of the compiler's ProcessWhens() function. The Wait function that is part of the Timers library, Timers.Wait, calls the compilers ProcessWhens() in a loop so that when statements can continue to be processed while waiting:

```
private IEnumerator WaitFunction( List<Value> values )
{
   if ( values.Count != 1 )
   {
      Debug.LogError("Timer.Delay(time in seconds);");
   }
   else
   {
      values[0].ConvertTo(Value.ValueType.Float);

      float currentTime = Time.time;
      float endTime = values[0].F + currentTime;

      while( Time.time < endTime )
      {
         yield return new WaitForFixedUpdate();
         compiler.ProcessWhens();
      }
   }

   yield return 0;
}
```

Custom Properties

You can create custom properties for variables. Custom properties are similar to fields or properties in C# and other programming languages. The custom properties like content or collision are in fact created in this way. To create a custom property you call the Variables class function CreateCustomProperty. For example, here is how the length property is added:

```
Variables.CreateCustomProperty("length", GetLength, SetLength);
```

The GetLength and SetLength are methods or functions that you provide that are called when your property is encountered. GetLength is a getter function while SetLength is a setter function.

The getter function calls you with a Value. You need to write the code that translates the raw Value passed to you into the form that you want the property to return. For example, in the length property the script programmer is using the length property to get the length of an array. So the GetLength() function is passed the name of the array. If you want to be sure the user passed in a variable you can check the Value's isVariable property. In the case of the code shown here, Variables.length() will do this for us so we did not need to check.

```
private Value GetLength(Value v)
{
    v.I = Variables.Length(v);
    v.T = Value.ValueType.Integer;

    return v;
}
```

The setter function is similar in concept but slight more complex to implement. When a setter is called you are provided with the **dest** variable which is where the value is to be placed, the array index and the value that is to be used to update the destination variable. If you want setter to persist the values to the Variable database then you need to make a change to the dest variable. For example, the length method when assigned a value sets the length of the array to that number if the number is < the current length of the array. This is accomplished by calling the Variables.Delete() function.

```
private void SetLength(Value dest, int arrayIndex, Value source)
    {
        if ( dest.isVariable )
        {
            source.ConvertTo(Value.ValueType.Integer);

            int len = Variables.Length(dest);
            int index = source.I;
            if ( index <= 0 )
            {
                index = 0;
            }
            Variables.Delete(dest, index, len - index);
        }
    }
```

Another simpler example of a setter function is the px property. In this case we only want to update the gameobject attached to the passed in destination variable. So we don't need to call any of the Variables store, clear or delete functions.

```
  private void SetPX( Value dest, int arrayIndex, Value source )
  {
    List<Value> va = Variables.Expand(dest, false);

    source.ConvertTo(Value.ValueType.Float);

    for(int ii=1; ii<va.Count; ++ii)
    {
      if ( va[ii].T == Value.ValueType.GObject )
      {
        va[ii].G.transform.position = new Vector3(source.F, va[ii].G.transform.position.y, v
a[ii].G.transform.position.z);
      }
    }
  }
```

Take note of the special way in which the variables are updated. When your setter is called it may be called with an actual array index like array[2] or array[1] or it may be called with array[0] in this last case your setter should update the entire array. The simplest way to do this is to call the Variables.Expand function which will return the entire array of values to your setter function. Passing in false will prevent any getters from being called when the array is created. This is normally what you want but in some cases you want the actual getter values. Look at the PrintFunction in the JigCompiler.cs file if you are interested in seeing an implementation that calls Expand without passing in false.

Once you have the array of values from expand you can loop over them and make any changes you see fit.

Caution: You cannot call the Variables.Store() function from inside your Set method because the Variables.Store() method is calling your set method. To do so will set up a recursive loop that can cause a stack over run.

Script Variables

Your extension functions can also read and write script variables. The static Variables class provides methods that your extension methods can call to work with script variables. All of the methods in Variables are static so that you can call them from any extension that you write.

`public static bool Exists(string name)`
Returns true if a variable with the name "name" exists.

`public static int Index(string name)`
Returns the actual index of a variable with the name "name" if it exists. If not returns -1.

public static string Name(int index)
Reverse of index, returns the actual internal name of the variable at the location identified by index.

public static int Length(Value v)
Returns the length of the variable identified by Value.

public static void ClearLocals()
Clears all local variables. This is called each time your script is compiled.

public static List<Value> Expand(Value v)
Returns a List of all values that are associated with the passed in variable. If the Value does not contain a variable, *isVariable* member is false, then only the passed in value is returned as the element 1 if the list. The first element or [0] of the list is always set to the passed in variable.

public static void FillArray(Value vt, int start, Value vf)
Fills the passed in array beginning at the start index with the value contained in the vf value. The array is expanded if necessary.

public static void Clear(Value vt)
Clears the array associated with the variable identified by vt.

public static void Delete(Value vt, int firstRow, int rowsToDelete)
Deletes elements of the array associated with the value vt.

public static Value Read(string name, int arrayIndex)
Returns the value of the variable identified by name at array index. The first value is located at element [1].

public static Value Read(int index, int arrayIndex)
Returns the value of the variable identified by index at array index. The first value is located at element [1].

public static void Store(Value dest, int arrayIndex, Value source)
Stores the source value into the dest value at arrayIndex.

public static void Copy(Value to, Value from)
Copies one array to another.

public static void CreateUnique(ref Value v)
Creates a uniquely named variable. The passed in value contains the initial value. The name, index and other internal parameters are set in the passed in value.

public static int Create(string name, Value v)
Creates a variable with the name, name and stores the value v at this variables location. The

index of the created variable is returned.

Global versus Locals

The compiler variable functions abstract the complexity of working with global and local variables away from the developer. Locals are always checked before Global variables. Locals are scoped to the file, function and curly brace level in which they occur. All variables in JigScript are processed within locks. This means that the programmer does not have to worry about serialization issues when using JigScript.

Internals

The implementation of JigScript as provided by the Unity 3d package is arranged in several folders. All of the files necessary though are included in three folders: Editor, Engine and Libraries. The Editor folder contains the source files that implement the JigScript menu and importer that runs your JigScripts when they have changed and been reimported into Unity from Monodevelop. The Engine folder contains the compiler, CPU, Variables and other sources that make JigScript work. The final folder Libraries contains the add in library functions that are supplied as part of the standard JigScript package.

How the compiler works

This is a very brief over view of how the JigScript compiler works. The JigCompiler.cs class is responsible for converting your JigScript source into a format that can be consumed by the CPU class. The CPU class is a software CPU. This means that it emulates what a CPU would do in order to run a program. The operations that this CPU performs are customized though to efficiently run JigScript compiled code.

To Compile a script you call the JigCompiler() method and pass it a string containing the script you want to compile. After some initialization work, The Compile method calls the JigCompiler ProcessExpression() method. This method is the top-level analyzer. It is responsible for calling the correct compile function based on the type of token or identifier that is encountered. This method is completely reentrant which is necessary because expressions can contain other expressions. The ProcessExpression() method calls the lexer which in JigCompilers case is ProcessExpression(). ProcessExpression is the method that actually reads your script. ProcessExpression() reads or consumes characters from your script one at a time and determines what action it needs to take. This is similar to applying a regular expression evaluation though much less error prone and faster. C# regular expressions tend to large and slow to process unless precompiled which increases their size. Once a determination has been made as to what type of token was encountered, the ProcessExpression() function calls the appropriate processing or compilation function. In bison, flex or yacc based compilers, or in .net CLR these functions would be referred to as creating a syntax expression tree. If JigScript were a complete language with classes, multi-dimensional arrays, operator overloading and so on creating a discrete expression tree would be needed. However, JigScripts goal is to strip away those features of a programming language you don't need or that can be better handled in C# or JavaScript and let you focus instead on quickly getting your application to market! So the construction of a complete language tree would be extreme overkill.

Once a Token has been encountered one of three things happen:
If the token is a number or variable it is pushed onto a value stack. This results in some code being written to the CPU class to do the getting and pushing of the value. The CPU instruction that does this is called PUSHV. If the value is not a variable then the instruction used is PUSHI. PUSHV stands for push variable and PUSHI stands for push immediate value.

If the token is an operator then several things can happen. If it is not a unary operator like pre or post increment and it is not a grouping operator, then if its operational precedence that is higher than the operator on top of the operator stack; or the operator stack is empty it is simply pushed onto the operator stack. If the operator's precedence is less than the operator currently on the top of the operator stack, the operator on top of the operator stack is popped off and two values are popped off of the value stack. These are then sent to the handler function that adds instructions to the CPU program to perform the operators operation.

If the token is a reserved word, then that reserved words handler is called. That handler may call the ProcessExpression() method itself which is why ProcessExpression() needs to be reentrant.

This process continues until the operator stack is empty which means that the expression has been completely parsed. Note, however that while the expression may have been parsed completely the program may not have been. So Compile keeps calling ProcessExpression() until either an error occurs or no more expressions are left to be parsed.

Once the program has been compiled, its data will be stored in the CPUs program, functions and the Variable classes variables. Running the compiled program consists of calling the CPU classes Run function.

If you are interested in learning more about how compilers work there are many sources and excellent books on the Internet. Several Universities also offer courses in compiler creation.

JigScript Grammar

The EBNF language grammar for JigScript is included for completeness and reference.

```
The following is the EBNF grammar for JigScript.

<program>::=                  statement_list>
<statement_list>::=           statement> <statement_list>
                            | statement>
<statement_block>::=          {" "}"
                            | {" <statement_list> "}"
<statement>::=                <declaration_statement>
                            | <loop_statement>
                            | <decision_statement>
                            | <when_statement>
                            | <import_statement>
                            | <control_statement>
                            | <yield_statement>
<declaration_statement>::=    <variable_declaration>
                            | <function_declaration>
<loop_statement>::=           <while_statement>
                            | <for_statement>
                            | <repeat_statement>
<decision_statement>::=       <if_statement>
<variable_declaration>::=     var <id> ";"
                            | local <id>  ";"
                            | global <id>  ";"
                            | var id "=" <expression>  ";"
                            | global id "=" <expression> ";"
<function_declaration>::=     fun "(" <parameter_list> ")" <statement_block>
<while_statement>::=          while "(" <expression> ")" <statement_block>
<for_statement>::=            for "(" <expression> ";" <expression> ";" <expression> ")" <stateme
nt_block>
<repeat_statement>::=         repeat "(" <expression> ")" <statement_block>
<if_statement>::=             if "(" <expression> ")" <statement_block>
                            | if "(" <expression> ")" <statement_block> else <statement_block>
<import_statement>::=         import string_constant ";"
<control_statement>::=        break ";"
                            | continue ";"
                            | return ";"
                            | return <expression> ";"
<yield_statement>::=          yield ";"
<when_statement>::=           when "(" <expression> ")" <statement_block>
<expression>::=               assignment_expression>
                            | <conditional_expression>
                            | <arithmetic_expression>
                            | <logical_expression>
                            | <equality_expression>
                            | <relational_expression>
                            | <bitwise_expression>
                            | <prefix_expression>
                            | <postfix_expression>
<assignment_expression>::=    <id> "=" <expression>
                            | <id> "+=" <arithmetic_expression> <assignment_expression>
                            | <id> "-=" <arithmetic_expression> <assignment_expression>
                            | <id> "*=" <arithmetic_expression> <assignment_expression>
                            | <id> "/=" <arithmetic_expression><assignment_expression>
                            | <id> "%=" <bitwise_expression> <assignment_expression>
                            | <id> "&=" <bitwise_expression> <assignment_expression>
                            | <id> "|=" <bitwise_expression> <assignment_expression>
                            | <id> "^=" <bitwise_expression> <assignment_expression>
<conditional_expression>::=   <logical_expression>
                            | <relational_expression>
<arithmetic_expression>::=    <value> "+" <value>
                            | <value> "+" <value> <arithmetic_expression>
                            | <value> "-" <value>
                            | <value> "-" <value> <arithmetic_expression>
                            | <value> "*" <value>
                            | <value> "*" <value> <arithmetic_expression>
                            | <value> "/" <value>
```

```
                              | <value> "/" <value> <arithmetic_expression>
                              | <value> "%" <value>
                              | <value> "%" <value> <arithmetic_expression>
                              | <value> << <value>
                              | <value> << <value> <arithmetic_expression>
                              | <value> >> <value>
                              | <value> >> <value> <arithmetic_expression>
<logical_expression>::=         <value> || <value>
                              | <value> || <value> <logical_expression>
                              | <value> && <value>
                              | <value> && <value> <logical_expression>
<equality_expression>::=        <value> == <value>
                              | <value> == <value> <equality_expression>
                              | <value> != <value>
                              | <value> != <value> <equality_expression>
<relational_expression>::=      <value> "<" <value>
                              | <value> "<" <value> <relational_expression>
                              | <value> "<=" <value>
                              | <value> "<=" <value> <relational_expression>
                              | <value> ">" <value>
                              | <value> ">" <value> <relational_expression>
                              | <value> ">=" <value>
                              | <value> ">=" <value> <relational_expression>
<bitwise_expression>::=         <integer_value> "&" <integer_value>
                                <integer_value> "&" <integer_value> <bitwise_expression>
                                <integer_value> "|" <integer_value>
                                <integer_value> "|" <integer_value> <bitwise_expression>
                                <integer_value> "^" <integer_value>
                                <integer_value> "^" <integer_value> <bitwise_expression>
<prefix_expression>::=          ++ <id>
                              | -- <id>
<postfix_expression>::=         <variable> ++
                              | <variable> --
<value>::=                      <id>
                              | <const>
<const>::=                      integer
                              | float
                              | string
<variable>::=                   <id>
<repeat.count>::=               <expression>
<id>::=                         <id_char_start> <id_char_body>
                              | <id_char_start>
<id_char_body>::=               <id_char>
                              | <id_char> <id_char_body>
<id_char>::=                    <id_char_start>
                              | number_char
<id_char_start>::=              letter
                              | period
                              | underscore
<parameter_list>::=             id
                              | id "," <parameter_list>
integer_constant::=             "+" number
                              | "-" number
float_constant::=               "+" float
                              | "-" float
string_constant::=              """ <chars> """
float::=                        "." number
                              | number "."
                              | number "." number

number::=                       number_char
letter::=                       letter_char
period::=                       "."
underscore::=                   "_"
<chars>::=                      char chars
                              | char
char::=                         space_char
                              | symbol_char
                              | number_char
                              | letter_char
                              | escape_char
escape::=                       "\n"
                              | "\t"
                              | "\r"
space_char::=                   " "
```

```
letter_char::=                "A"
                            | "B"
                            | "C"
                            | "D"
                            | "E"
                            | "F"
                            | "G"
                            | "H"
                            | "I"
                            | "J"
                            | "K"
                            | "L"
                            | "M"
                            | "N"
                            | "O"
                            | "P"
                            | "Q"
                            | "R"
                            | "S"
                            | "T"
                            | "U"
                            | "V"
                            | "W"
                            | "X"
                            | "Y"
                            | "Z"
                            | "a"
                            | "b"
                            | "c"
                            | "d"
                            | "e"
                            | "f"
                            | "g"
                            | "h"
                            | "i"
                            | "j"
                            | "k"
                            | "l"
                            | "m"
                            | "n"
                            | "o"
                            | "p"
                            | "q"
                            | "r"
                            | "s"
                            | "t"
                            | "u"
                            | "v"
                            | "w"
                            | "x"
                            | "y"
                            | "z"
number_char::=                "0"
                            | "1"
                            | "2"
                            | "3"
                            | "4"
                            | "5"
                            | "6"
                            | "7"
                            | "8"
                            | "9"
symbol_char::=                "!"
                            | "\""
                            | "#"
                            | "$"
                            | "%"
                            | "&"
                            | "\\"
                            | "'"
                            | "("
                            | ")"
                            | "*"
                            | "+"
| ","
| "-"
| "."
| "/"
| ":"
| ";"
| "<"
| "="
| ">"
| "?"
| "@"
| "["
| "]"
| "^"
| "_"
| "`"
| "{"
| "|"
| "}"
| "~"
| "]"
```