Lua Framework

Version 1.1

Contents

Introduction

Lua Framework is set of tools which allow to easily and automatically convert data defined in the Lua script language to .NET objects, and vice-versa. It operates in a similar way to existing XML or JSON readers, but instead of a markup language, you now have access to a powerful programming language to define your game or application's logic. Like many top-selling games did, choosing Lua can streamline the game design process and most importantly, allow easy-to-implement modding capacities.

Lua Framework is built on the power of **MoonSharp**, a modern .NET implementation of the Lua language. As opposed to previous .NET to Lua bridges such as LuaInterface, NLua or UniLua, MoonSharp provides a very intuitive user experience with fast learning curve, fast performance, is regularly updated, and supports Unity's Mono implementation of the .NET language out of the box, on both Unity and Unity Pro platforms.

For now, the **Lua Framework** has two main modules: LuaReader and LuaWriter. LuaReader automatically maps Lua variables and tables to .NET objects. LuaWriter creates a Lua script representation of .NET objects. Currently supported types are:

- Built-in types: **`bool`**, **`int`**, **`float`**, **`double`**, **`string`**, **`byte`** & **`decimal`**;
- Lua functions (**`Closure`**);
- Enums;
- Unity-specific structs: **`Color`**, **`Color32`**, **`Rect`**, **`Vector2`**, **`Vector3`** & **`Vector4`**;
- ANY custom classes with public properties;
- One-dimensional or two-dimensional arrays of any supported type;
- Generic Lists or Dictionaries of any supported type;
- Any possible nesting, for example **`List`**<**`Dictionary`**<**`string`**, **`Vector3`**[]>>.

Installation

1. Copy the LuaFramework folder to your assets folder. If you are updating from a previous version, simply delete the previous version folder before installing the new one.
2. Install MoonSharp from http://www.moonsharp.org/, copying the required DLLs to your assets folder.

LuaReader Basics

Before calling the **LuaReader** class methods, you must import the `LuaFramework` and `MoonSharp.Interpreter` namespaces, with the following directives:

```
using LuaFramework;
using MoonSharp.Interpreter;
```

Representation of Lua values in MoonSharp

All **LuaReader** methods take either **Table** or **DynValue** arguments, two MoonSharp types which represent Lua variables:

- **Table** represents a Lua table;
- **DynValue** is a container for any Lua type, including table, but also function, number, string or nil.

If needed, you can easily convert between the two:

```
Table table = someDynValue.Table;
DynValue dynValue = DynValue.NewTable(someTable);
```

For more info, you can consult the tutorials and reference on moonsharp.org.

Executing Lua scripts with MoonSharp

To get the data into one of those two types, you must execute a string containing Lua code inside a MoonSharp Lua script environment. First, you need to create a Lua environment:

```
Script lua = new Script();
```

Then, you need to execute your string of Lua code:

```
lua.DoString(luaCode);
```

And that's it! But how to actually define objects and retrieve the data once the script is executed? There are two main approaches.

Defining objects with Lua, option 1: using variables

A first option is to create variables in the Lua **`Script`**, and then "get" those variables from the **`Script`** after the execution of the code. This approach is best when you have a small number of specific objects to define. Typical applications include .ini files or save games.

For example, let's say we wish to define an array of vectors. To define it, we could have use following Lua code:

```
myVectors = {
	{0, 0, 45},
	{2, 0.5, 0},
	{5, 0, -2.5},
}
```

After executing this in a **`Script`**, this script will now contain a global variable named myVectors, which contains a table of tables of numbers. You can get it by accessing the Globals table of the Script:

```
string s = @"
myVectors = {
	{0, 0, 45},
	{2, 0.5, 0},
	{5, 0, -2.5},
}";
Script lua = new Script();
lua.DoString(s);
DynValue myVectors = lua.Globals.Get("myVectors");
```

Then, we can simply use the **`LuaReader.`**`Read` method to convert that Lua table to a .NET **`Vector3`** array:

```
Vector3[] myVectorArray = LuaReader.Read<Vector3[]>(myVectors);
```

The Read method automatically maps Lua data to .NET types, as long as the data is properly formatted, for example if you're not trying to convert a table of strings to a float. Doing the following:

```
float f = LuaReader.Read<float>(myVectors);
```

Will result in f being assigned the default value of 0, as LuaReader cannot convert the table to a float value. LuaReader can handle the case without raising an exception however. For more detailed information on the usage and syntax of the Read function, see the reference section below.

Defining objects with Lua, option 2: using functions

A second approach is to use functions in the Lua **`Script`** which call functions in C# which handle the definition task, usually adding the newly defined object to some sort of collection. In this case the definition happens as the Lua script is executed. This approach is cleaner (no global variables left in the Lua state) and best when you have a large number of objects to define, without knowing the exact quantity beforehand. Typical applications include defining items, enemies, NPCs dialogue, even maps or levels.

For example, let's say we have an **`Enemy`** class, which has a name, health and attackPower properties:

```
public class Enemy
{
	public string name { get; set; }
	public int health { get; set; }
	public float attackPower { get; set; }
}
```

To define our enemies, we would ideally use the following Lua script:

```
defineEnemy{
	name = "Bandit",
	health = 50,
	attackPower = 3.5,
}

defineEnemy{
	name = "Bandit Leader",
	health = 125,
	attackPower = 5,
}
```

The {} Lua syntax allows for function calls with a variable number of arguments, which are sent packed together in a single table. To process the above script, we first need to define a collection to hold our enemies definitions, and then declare a C# method to process the table sent by the Lua function and add the definitions to the collection:

```
public List<Enemy> enemyDefinitions = new List<Enemy>();

public void DefineEnemy(DynValue luaTable)
{
	Enemy enemy = LuaReader.Read<Enemy>(luaTable);
	enemyDefinitions.Add(enemy);
}
```

As you see, the Read method will automatically map the Lua table to our custom Enemy class without us needing to do anything else.

The last thing we need to do is to register our .NET DefineEnemy method as a global in the Lua environment, so that the script is able to call it. This is done most easily by passing the method as a generic delegate, casting it using the **Action<T >** type, where T is the type of the argument the method receives:

```
Script lua = new Script();
lua.Globals[“defineEnemy”] = (Action<DynValue>) DefineEnemy;
```

To use this, you do not need to understand how delegates work under the hood, you can simply copy the above syntax which will be valid for all Lua function calls that use the {} variable syntax.

Once the above is done, we can execute our Lua script, and while executing, the Lua code will call twice the DefineEnemy method, resulting in two Enemy objects being added to the enemyDefinitions collection:

```
lua.DoString(s); // Where s contains the above script.
```

The advantage of using a scripting language versus a markup language like XML, is that it allows dynamic definitions like the following script:

```
baseBanditHealth = 50
baseBanditAttackPower = 3.5

defineEnemy{
	name = “Bandit”,
	health = baseBanditHealth,
	attackPower = baseBanditAttackPower,
}

defineEnemy{
	name = “Bandit Leader”,
	health = baseBanditHealth * 2 + 25,
	attackPower = baseBanditAttackPower + 1.5,
}

defineEnemy{
	name = “Bandit Lord”,
	health = baseBanditHealth * 3 + 75,
	attackPower = baseBanditAttackPower + 3,
}
```

Using such a structure allows to make global adjustments much easier, rather than manually adjusting all individual enemies if you decide that Bandits as a whole should be weaker or stronger. It also allows procedural definitions.

LuaWriter Basics

Before calling the **LuaWriter** class methods, you must import the LuaFramework namespace, but you do not need to import MoonSharp as **LuaWriter** is creating strings of Lua code, not interpreting them.

LuaWriter derives from the standard C# **StringWriter** class. It works like similar **HtmlTextWriter** or **XmlWriter** classes, but providing specific methods for converting objects to an Lua syntax. It is a stream writer, which you should use with the using clause, to ensure that the writer is properly disposed of, eventually transferring its contents to a string declared outside the using block:

```
string luaCode;
using (LuaWriter luaWriter = new LuaWriter())
{
	// Write instructions
	luaCode = luaWriter.ToString();
}
```

Indentation

You can of course use any methods that **StringWriter** supports. The only difference is that the Write and WriteLine methods take into account the indentation capacities **LuaWriter** supports through the AddTab and RemoveTab methods:

- AddTab adds a tab character to the beginning of all following lines, until you call RemoveTab. AddTab can be called multiple times and is cumulative.
- RemoveTab removes a tab added with AddTab. If you call RemoveTab when at level 0 of indentation, it simply does nothing.

So, if you use WriteLine while a tab is in effect, it will write "\n\t" to the stream, instead of just "\n" like a normal **StringWriter** would do.

Writing C# objects to the stream

The main method LuaWriter provides is WriteObject, which takes any of the supported types and converts it to a Lua representation. Booleans become true or false, strings are converted to a litteral representation with quotes, collections, arrays and classes become tables. Here's an example:

```
Dictionary<string, Color32> colors = new Dictionary<string, Color32>();
colors[“Red”] = new Color32(255, 0, 0, 255);
colors[“Green”] = new Color32(0, 255, 0, 255);
colors[“TransparentBlue”] = new Color32(0, 0, 255, 128);

string luaCode;
using (LuaWriter luaWriter = new LuaWriter())
{
	luaWriter.Write(“colors = ”);
	luaWriter.WriteObject(colors);
	luaCode = luaWriter.ToString();
}
```

If you were to output the contents of luaCode, you would have the following:

```
colors = {
	Red = {255, 0, 0, 255},
	Green = {0, 255, 0, 255},
	TransparentBlue = {0, 0, 255, 128},
}
```

LuaReader Reference

LuaReader.Read

```
T LuaReader.Read<T>(DynValue luaValue)
T LuaReader.Read<T>(Table luaTable)
```

This is the main **LuaReader** method. Reads a lua object contained in a **DynValue** or **Table** type, and converts its contents to a newly created .NET object of type T. If the lua value provided cannot be converted to the specified type, the default is returned. Example usage:

```
int[] intArray = LuaReader.Read<int[]>(luaTable);
```

Supported types:

- Usage for all built-in types is straightforward:

```
bool = true/false
int = 1
float = 0.5
double = 0.5
string = "Hello World!"
byte = 128
decimal = 0.00005
```

- For enums, the Lua script must provide a string representation. For example, if you have an enum called **Direction** with values `Left`, `Front` & `Right`, you would define it in Lua like this:

```
direction = "Front"
```

- All supported Unity structs must be defined in Lua as a table, following the standard order of arguments the struct constructor uses. If an argument is missing, LuaReader will assign it the default value of 0 for everything except alpha for **Color** (defaults to 1) and **Color32** (defaults to 255), allowing you to define colors with only 3 arguments when you don't require transparency. Supported types:

```
color = {r, g, b, a}
color32 = {r, g, b, a}
rect = {x, y, width, height}
vector2 = {x, y}
vector3 = {x, y, z}
vector4 = {x, y, z, w}
```

- Arrays and Lists must be defined in Lua as a table. One and two dimensional arrays are supported. For example:

```
intArray = {0, 1, 2, 3, 4, 5}
stringList = {"one", "two", "three"}
```

- Dictionaries must be defined in Lua as an indexed table, most commonly with string keys. For example:

```
dictionary = {
        red = “A warm color”,
        blue = “A cold color”,
}
```

- Custom classes must be defined in Lua as an indexed table, with string keys representing public properties. For example:

C#:

```
public class Person
{
        public string name { get; set; }
        public int age { get; set; }
}
```

Lua:

```
person = {
        name = “John”,
        age = 26,
}
```

LuaReader.ReadClassData

```
void LuaReader.ReadClassData<T>(T clrObject, DynValue luaValue)
void LuaReader.ReadClassData<T>(T clrObject, Table luaTable)
```

Reads a lua table contained in a **DynValue** or **Table** type, and maps its contents to an existing .NET object of type T. The option to use a **DynValue** argument is provided as convenience, but it must contain a valid Lua table. The difference with `Read` is that `Read` creates the object from scratch, while `ReadClassData` loads properties into an existing instance of the class. This can be useful if you need to create an instance with a custom constructor, or if you wish to reuse objects (pooling). Internally, `Read` calls `ReadClassData` after creating an instance. Considering the **Person** class above, the two examples below are strictly equivalent, but the second allows you to do something with the object before loading the properties from Lua:

Example 1:

```
Person person = LuaReader.Read<Person>(luaTable);
```

Example 2:

```
Person person = new Person();
LuaReader.ReadClassData(person, luaTable);
```

LuaReader.ReadSingleProperty

```
void LuaReader.ReadSingleProperty<T>(T clrObject, string propertyName DynValue luaValue)
void LuaReader.ReadSingleProperty<T>(T clrObject, string propertyName Table luaTable)
```

Takes a lua table contained in a **DynValue** or **Table** type, and maps a single property out of it to an existing .NET object of type T. This is similar in function to ReadClassData, but only loads one property. This can be useful for example if you wish to manually load properties one by one because you need them set in a particular order. These two examples are thus equivalent:

Example 1:

```
Person person = new Person();
LuaReader.ReadClassData(person, luaTable);
```

Example 2:

```
Person person = new Person();
LuaReader.ReadSingleProperty(person, "name", luaTable);
LuaReader.ReadSingleProperty(person, "age", luaTable);
```

LuaReader.AddCustomReader

```
void LuaReader.AddCustomReader(Type type, Func<DynValue, object> reader)
```

Adds a custom reader or overrides the default reader behavior for a specific .NET type. Useful if you wish to format your Lua data in a custom way or implement reader for structs with a constructor-like table syntax. Example 1 shows the standard way to define a **Person** using the class example provided above:

Example 1:

```
person = {
        name = "John",
        age = 26,
}
```

A custom reader as shown in the example 2 could allow us to condense the definition if you have a large number of objects to define following a common pattern. Note the two arguments given to `AddCustomReader()`, first the **`typeof()`** to identify the type, and then a lambda expression that accepts a generic **DynValue** parameter representing the incoming Lua value, and which must output a .NET object of the desired type. Notice also that you can use the **LuaReader** class inside the custom converter to convert any desired values.

Example 2:

C#:

```
LuaReader.AddCustomReader(typeof(Person), dynValue =>
        {
                var luaTable = dynValue.Table;
                return new Person{
                        name = LuaReader.Read<string>(luaTable.Get(1)),
                        age = LuaReader.Read<int>(luaTable.Get(2))
                };
        });
```

Lua:

```
person = {“John”, 26}
person2 = {“Maria”, 31}
person3 = {“Paul”, 17}
```

LuaReader.RemoveCustomReader

```
void LuaReader.RemoveCustomReader(Type type)
```

Removes a previously added custom reader.

LuaWriter Reference

LuaWriter.Write

```
void LuaWriter.Write(string str);
```

Writes a string to the current stream, calling the parent method **`StringWriter.Write`**, but taking into account indentation added with `AddTab`.

LuaWriter.WriteLine

```
void LuaWriter.WriteLine(string str);
```

Writes a string to the current stream and adds a new line character, calling the parent method **`StringWriter.WriteLine`**, but taking into account indentation added with `AddTab`.

LuaWriter.AddTab

```
void LuaWriter.AddTab();
```

Adds a tab indentation level to the beginning of the current and all future lines, until `RemoveTab` is called. `AddTab` can be called multiple times and is cumulative.

LuaWriter.RemoveTab

```
void LuaWriter.RemoveTab();
```

Removes a tab indentation level added with `AddTab`. If you call `RemoveTab` when at level 0 of indentation, it simply does nothing.

LuaWriter.WriteObject

```
void LuaWriter.WriteObject<T>(T obj, [bool multiline = true], [bool
trailingComma = false]);
```

This is the main **`LuaWriter`** method. Writes to the stream a Lua representation of the provided .NET object. There are two optional arguments:

- `multiline` defaults to true, and determines if created tables should span multiple lines with one line per element, or be all on the same line:

```
string[] array = {“Spring”, “Summer”, “Autumn”, “Winter”};
using (LuaWriter luaWriter = new LuaWriter())
{
	luaWriter.WriteObject(array);
	luaWriter.WriteObject(array, false);
}
```

The first will write:

```
{
	“Spring”,
	“Summer”,
	“Autumn”,
	“Winter”
}
```

The second will write:

```
{“Spring”, “Summer”, “Autumn”, “Winter”}
```

- `trailingComma` defaults to false, and determines if a comma should be added after the element is created, useful the object is written in a table.

Supported types:

- Lua representation for all built-in types is straightforward. Note that strings are written as a literal representation with bounding quotes:

```
bool -> true/false
int -> 1
float -> 0.5
double -> 0.5
string -> “Hello World!”
byte -> 128
decimal -> 0.00005
```

- For enums, LuaWriter creates a string representation. For example, if you have an enum called **`Direction`** with values `Left`, `Front` & `Right`, you would get the following in Lua:

```
Direction.Front -> “Front”
```

- All supported Unity structs are represented in Lua as a table, following the standard order of arguments the struct constructor uses. Supported types:

```
color -> {r, g, b, a}
color32 -> {r, g, b, a}
rect -> {x, y, width, height}
vector2 -> {x, y}
vector3 -> {x, y, z}
vector4 -> {x, y, z, w}
```

- Arrays and Lists are written as Lua tables. For example:

```
intArray -> {0, 1, 2, 3, 4, 5}
stringList -> {“one”, “two”, “three”}
```

- Dictionaries are written in Lua as an indexed table. Only **`string`** or **`int`** keys are supported. Strings that are not valid keys in Lua (because of spaces for example) are detected and written accordingly. For example:

```
dictionary ->
{
        red = “A warm color”,
        blue = “A cold color”,
        [“light blue”] = “A lighter blue”,
}
```

- Custom classes are written in Lua as an indexed table, with string keys representing public properties. For example:

C#:

```
public class Person
{
        public string name { get; set; }
        public int age { get; set; }
}
```

Lua:

```
person -> {
        name = “John”,
        age = 26,
}
```

LuaWriter.WriteProperty

```
void LuaWriter.WriteProperty<T>(T obj, string propertyName, [bool multiline =
   true], [bool trailingComma = false]);
void LuaWriter.WriteProperty<T>(T obj, PropertyInfo propertyInfo, [bool
   multiline = true], [bool trailingComma = false]);
```

This is similar to `WriteObject`, but is used to write only one public property of the class. Internally, `WriteObject` calls `WriteProperty` for each public property when given a class object. It is useful when you do not wish to convert an entire object to Lua, or wish to do it in a particular way – `WriteProperty` is especially useful in custom serialization implemented through the **`ICustomLuaSerializer`** interface (see below). There are two versions of this method, as you can provide either the name of the public property as a string, or the **`PropertyInfo`** describing the property, if you are yourself somehow iterating through the object with reflection. So, using the Person class above, you could write the following:

```
Person person = new Person{name = “John”, age = 35};
using (LuaWriter luaWriter = new LuaWriter())
{
        // Example 1
        luaWriter.WriteObject(person);
        // Example 2
        luaWriter.WriteProperty(person, “name”);
        luaWriter.WriteProperty(person, “age”);
}
```

Example 1 outputs:

```
{
	name = “John”,
	age = 35,
}
```

Example 2 outputs:

```
name = “John”,
age = 35,
```

LuaWriter.SetRequiredAttribute

```
void LuaWriter.SetRequiredAttribute(Type attributeType)
```

This method allows to optionally set a required attribute that public properties need to have to be serialized. The Lua Framework provides the **`LuaSerializable`** attribute already defined for you, but you can use your own if you prefer. If it is set, only public properties tagged with the corresponding attribute will be serialized by `WriteObject`. Use **`null`** to remove the requirement. Example:

```
public class Person
{
	[LuaSerializable]
	public string name { get; set; }
	public int age { get; set; }
}

//

Person person = new Person{name = “John”, age = 35};
using (LuaWriter luaWriter = new LuaWriter())
{
	luaWriter.SetRequiredAttribute(typeof(LuaSerializableAttribute));
	luaWriter.WriteObject(person);
}
```

Outputs, only name being processed:

```
{
	name = “John”,
}
```

ICustomLuaSerializer Interface

```
public interface ICustomLuaSerializer
{
	void Serialize(LuaWriter luaWriter);
}
```

The Lua Framework provides this interface to allow custom serialization of objects, instead of the automatic one. If a class that implements this interface is processed by `WriteObject`,

instead of writing it to Lua by scanning all public properties, `WriteObject` will call the Serialize method of the interface, providing it with the current **`LuaWriter`** instance. Note that `WriteObject` will still open and close Lua table brackets for the object. Example:

```
public class Person : ICustomLuaSerializer
{
	public string name { get; set; }
	public int age { get; set; }
	public bool ageIsConfidential { get; set;}

	public void Serialize(LuaWriter luaWriter)
	{
		luaWriter.WriteProperty(this, “name”, trailingComma: true);
		if (!ageIsConfidential)
		{
			luaWriter.WriteProperty(this, “age”, trailingComma: true);
		}
		else
		{
			luaWriter.WriteLine(“-- Age is confidential!”);
		}
	}
}

//

Person person = new Person{
	name = “John”,
	age = 35,
	ageIsConfidential = true};
using (LuaWriter luaWriter = new LuaWriter())
{
	luaWriter.WriteObject(person);
}
```

Outputs:

```
{
	name = “John”,
	-- Age is confidential!
}
```