LUASCRIPTOBJECT

MugHeadStudios

OVERVIEW

> *Note that this package **REQUIRES** the **MoonSharp** package to work, you can find it here:*
> *https://assetstore.unity.com/packages/tools/moonsharp-33776*

Why use Lua with Unity?

Lua is a lightweight, fast, and powerful scripting language that is commonly used in game development. It is designed to be easy to learn and use, with a simple syntax and flexible data types.

In a Unity project, Lua can be used to add custom behavior and logic to game objects and scripts. It can also be used to create custom tools and editor extensions. Because Lua is interpreted at runtime, it allows for rapid prototyping and iteration during development. Additionally, Lua scripts can be easily modified and hot-reloaded, allowing developers to quickly test and refine their code.

Overall, the inclusion of lua in a Unity project can provide developers with greater flexibility and control over their game or application.

What is a LuaScriptObject?

The LuaScriptObject is a class that allows us to run our lua scripts in Unity. It is designed to be exceptionally easy to use and integrate into your project. It also helps with a difficult task in lua scripts, in that it automatically deals with coroutines in a way that we can yield and resume scripts easily without worrying about how it is working behind the scenes. By default we have added handy lua functions like **wait**, **wait_until**, and **always** for running time or condition sensitive actions within our scripts.There are many features of this package, it is recommended you at the very least scan through this documentation to get an idea on how to set it up and use it's powerful features.

Table Of Contents

> *Please note if you see any mistakes please contact me using one of the options at the end of this document.*

1. Prerequisites

To use Lua in your game or application, you will need to first import the **MoonSharp** package (https://assetstore.unity.com/packages/tools/moonsharp-33776) to your project.

Also, any C# files that use or reference this package should include the **MugHeadStudios.Lua** namespace:

```
using MugHeadStudios.Lua;
```

2. Setting up with the LuaSetup class

If you don't want to extend the global or script object's functions you don't need to worry about setting anything extra up as by default we register many common types and functions with **MoonSharp** automatically, note that we can automatically register our own types by simply adding the attribute **[MoonSharpUserData]** to our classes. However if you would like to register some extra Unity components or something else that we can't manually add the MoonSharpUserData attribute to then please read below.

By default, **LuaSetup.DefaultSetup()** registers common types with the interpreter, such as **GameObject**, **Transform**, and **Vector3**. This method also calls the **UserData.RegisterAssembly()** method, which finds any class with the **[MoonSharpUserData]** attribute and adds it automatically to the interpreter, this way you don't have to run custom setup for adding your own types to the MoonSharp interpreter if you simply add this attribute to your classes.

If you want to customise the setup, you can add a special **LuaSetup** extension method named **CustomLuaSetup** that will be automatically called at the start of your application by **LuaSetup** itself. For example:

```
using UnityEngine;
using MoonSharp.Interpreter;
using MugHeadStudios.Lua;

[MoonSharpUserData]
public static class LuaExtensions
{
    public static void CustomLuaSetup(this LuaSetup luaSetup)
    {
        // Register custom type
        luaSetup.RegisterType<SomeType>();

        // Not recommended, but we can hack global's as below
        luaSetup.scriptModifier = (luaScriptObject, script) => {
           script.Globals["example_variable"] = "Some string!";
           script.Globals["example_function"] = (Action)(() => { /* Some function */ });
        };
    }

    // We could also add our own custom LuaScriptObect or LuaGlobal extension methods
    // here, more on that topic in section 3d. and 4a.
}
```

3. Using the LuaScriptObject component

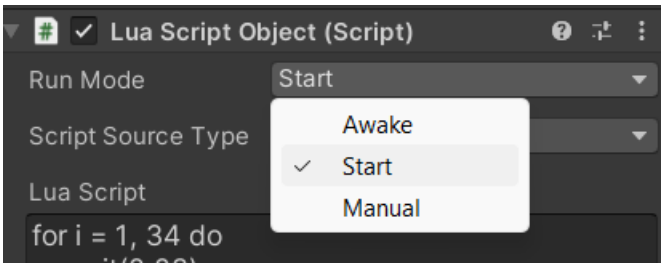
In order to use the **LuaScriptObject**, you will need to add the **LuaScriptObject** component to your game object. You can do this either in the Unity editor or programmatically in your code.

```
// Programmatically (Within a MonoBehaviour game object)
this.AddComponent<LuaScriptObject>();
```

3a. Run mode

You can specify how you want the lua script to be run. There are three options available:

- **Awake**: The script will be run when the **Awake** method is called on the game object.
- **Start**: The script will be run when the **Start** method is called on the game object.
- **Manual**: The script will not be run automatically, and must be run manually using the **RunLuaScript** method.



You can either set this in the inspector (as above), or in code:

```
// Eg. Set run mode to Manual
luaScriptObject.runMode = LuaScriptObject.RunMode.Manual;
```

Remember that if you extend the **LuaScriptObject** and override it's Awake, Start or Update method you will need to call the base related method otherwise the associated lua script will not be run automatically or the sub routines will not run (more on that later).

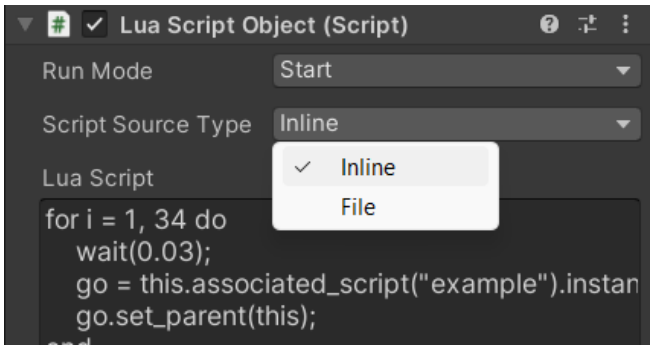
If you have set the **runMode** to **Manual**, you will need to manually run the lua script using the **RunLuaScript** method:

```
// Run lua script manually
luaScriptObject.RunLuaScript();
```

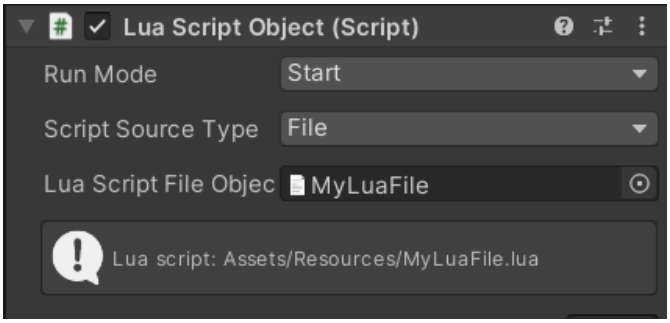
3b. Script source type

You can set the lua script for the **LuaScriptComponent** by either:

- By setting to "**Inline**" mode (default), and writing your lua in the inspector.



- By setting to "**File**" mode, and dragging in a .lua file (must be a descendant of the **Resources/** directory).



- Or finally by using **“Inline”** mode, but setting the luaScript field in script.

```
// Manually set lua script
luaScriptObject.luaScript = @"
    debug('Hello, World!');
";
```

3c. Fields and functions

Note that within an objects lua script you can access it’s methods with the “**this”** global keyword. This keyword is registered within the **RunLuaScript** method on this object. In other words, you can run methods (or custom extension methods) within this objects lua script like so:

```
-- object's own lua script (“this” example)
debug(this.name);
this.destroy();
```

List of LuaScriptObject’s public fields and functions (for use in script) below:

-- Fields --

this.gameObject – Returns this **GameObject**, which exposes MonoBehaviour’s fields and methods into lua for you

this.creationTime – Returns the time that this **LuaScriptObject** was created

this.position – Returns a **LuaTransformVector** (See below)

this.rotation– Returns a **LuaTransformVector** (See below)

this.scale– Returns a **LuaTransformVector** (See below)

-- Functions --

this.yield(params DynValue[] args) -- Yields the coroutine and returns a DynValue object. The optional args parameter can be used to pass values to the calling coroutine.

this.resume() -- Resumes the coroutine.

this.run() -- Manually runs the script.

this.wait(float seconds) -- Makes the script wait for the given number of seconds before continuing. Note you can also call this with just wait() as a global function.

this.wait_until(Func<bool> condition) -- Makes the script wait until the given condition is met before continuing. Note you can also call this with just wait_until() as a global function.

this.always(Action action) -- Executes the given action forever (or until this object is destroyed or disabled). Note you can also call this with just always() as a global function.

this.set_parent(object parent) – Sets transform parent. Parent can be **LuaScriptObject**, **GameObject**, or **Transform**.

this.created() -- Returns the creation time of the game object.

this.active() -- Returns a boolean indicating whether the game object is active.

this.active(bool active) -- Sets whether the game object is active or not.

this.component(string name) -- returns a component by name

this.set_data(string key, object value) -- Sets the value for the given key in the data dictionary.

this.get_data(string key) -- Returns the value for the given key in the data dictionary.

this.key_down(string keyCode) -- Returns whether key is down (See UnityEngine.KeyCode enum reference for details).

this.sin(float frequency = 1, float amplitude = 1) -- Returns sin of time since creation with frequency and amplitude.

this.cos(float frequency = 1, float amplitude = 1) -- Returns cos of time since creation with frequency and amplitude.

this.tan(float frequency = 1, float amplitude = 1) -- Returns tan of time since creation with frequency and amplitude.

this.instantiate() -- instantiates self and returns the instantiated game object's LuaScriptObject component

this.associated_object(string name) -- returns the associated game object by name

this.associated_script(string name) -- returns the associated game object's LuaScriptObject component by name

this.destroy() -- destroys the game object

-- Note on LuaTransformVector's –

The **position**, **rotation**, and **scale** fields are **LuaTransformVector** objects which allow us to manipulate a game objects transform properties with ease. Here are some examples of how to use them:

Setter examples:

```
this.position.x = 5 -- Set the x position of this game object to 5
this.position.translate(0, 0.1) -- Move this object by 0.1 on the Y axis
this.scale.xy = this.sin() -- Pulse the x & y scale of this object with sine
this.rotation.set(vector3) -- Set the rotation to an existing vector3
this.rotation.translate(0, 0, delta_time()) -- Rotate object's z axis by delta time
```

Getter examples:

```
this.position.get() -- Gets the xyz position as a vector2
this.position.x / y / z -- Gets the x position as a float, same for y or z
this.position.xy_vector -- Gets the xy position as a vector2
this.position.xz_vector -- Gets the xz position as a vector2
this.position.yz_vector -- Gets the yz position as a vector2
```

See the **LuaScriptObject/Runtime/LuaTransformVector.cs** file for all getters / setters and methods.

3d. Extending LuaScriptObject or it's functions

To extend the functionality of **LuaScriptObject** and add your own functions, just create a static class with the **[MoonSharpUserData]** attribute and add custom functions with the extension style syntax for LuaScriptObject, note that you can include any number of arguments or return any data type that has been registered eg:

```
[MoonSharpUserData]
public static class LuaExtensions
{
    // Add your own custom object functions here, eg:
    public string example(this LuaScriptObject luaScriptObject, string message)
    {
        Debug.Log(luaScriptObject.name + ": " + message);
    }
}
```

Now your custom object functions can be used in Lua scripts like so:

```
-- Homemade object function showcase
this.example("Example message");
```

4. Global lua functions (LuaGlobal.cs)

The **LuaGlobal** class is a class that holds a collection of functions that can be accessed by Lua scripts. These functions are made available to the Lua interpreter by being registered in the **LuaSetup.cs** file and placed in the global scope of the interpreter when **LuaScriptObject** calls **RunLuaScript**. This means that the functions can be called directly from any Lua script without the need to reference an object. Eg:

```
-- Calling a function from LuaGlobal in Lua script
someObject = find_object("GameObject");
destroy(someObject);
```

Important note: Unlike the **LuaScriptObject** functions, you cannot overload functions with the same name, this is because when adding the Global Lua functions to the script interpreter they are stored as individual methods within a dictionary, rather than serializing the entire object itself.

List of LuaGlobal functions below:

debug(string message, bool withTime = false) -- Outputs message to the console with or without timestamp.

time() -- returns the current time as a float.

delta_time() -- Returns the current delta time as a float

wait(float seconds) -- Makes the script wait for the given number of seconds before continuing. Returns a DynValue object. Note that the script calling this function will automatically pass itself to the LuaGlobal sub routine. If the object is destroyed or dissabled the sub routine will delete itself.

wait_until(Func<bool> condition) -- Makes the script wait until the given condition is met before continuing. Returns a DynValue object. Note that the script calling this function will automatically pass itself to the LuaGlobal sub routine. If the object is destroyed or dissabled the sub routine will delete itself.

always(Action action) -- Executes the given action repeatedly. Note that the script calling this function will automatically pass itself to the LuaGlobal sub routine. If the object is destroyed or dissabled the sub routine will delete itself.

vector2(float x, float y) -- Returns a Vector2 with the given x and y values

vector3(float x, float y, float z) -- Returns a Vector3 with the given x, y, and z values

distance_between_v2(Vector2 a, Vecto2 b) – Returns the linear distance between to Vector 2's as a float.

distance_between_v3(Vector3 a, Vecto3 b) – Returns the linear distance between to Vector 3's as a float.

distance_between(float a, float b) – Returns the distance (difference) between two float values as a float.

load_scene(string name) – Loads the scene based on the given name

scene_name() – Gets the current active scene name

set_data(string key, object value) -- Set global data by key and value. If the key already exists, it updates the value. If the key does not exist, it adds the key and value to the dictionary.

get_data(string key) – Gets global data by key. The value returned will be casted to it's actual type. Returns false if the key does not exist.

get_type(object obj) – Returns the type of the given object as a string (Note the object type must be registered)

instantiate(GameObject gameObject) -- Instantiates and returns a copy of the given gameObject

destroy(GameObject gameObject) -- Destroys the give gameObject

find_object(string name) -- Returns the GameObject with the given name

find_script(string name) -- Returns the LuaScriptObject component of the GameObject with the given name

random_int(int min, int max) -- Returns a random integer between min and max (inclusive)

random_float(float min, float max) -- Returns a random float between min and max (inclusive)

sin(float frequency = 1, float amplitude = 1) -- Returns sin of current time with frequency and amplitude.

cos(float frequency = 1, float amplitude = 1) -- Returns cos of current time with frequency and amplitude.

tan(float frequency = 1, float amplitude = 1) -- Returns tan of current time with frequency and amplitude.

4a. Extending LuaGlobal's functions

Just like the **LuaScriptObject** class, you can extend **LuaGlobal** and add your own global functions, just use/create a static class with the **[MoonSharpUserData]** attribute and add custom functions with the extension style syntax for **LuaGlobal**, make sure these methods are static, as they are accesible globaly eg:

```
[MoonSharpUserData]
public static class LuaExtensions
{
    // Add your own custom object functions here, eg:
    public static int add(this LuaGlobal luaGlobal, int a, int b)
    {
        return a + b;
    }
}
```

Now your custom global functions can be used in lua scripts like so:

```
-- Homemade global function showcase
add(3, 4); -- returns 7
```

8. Conclusion

Thank you for using MugHeadStudios' LuaScriptObject, if you have any suggestions, notice any mistakes within the documentation, or bugs in the package, please let me know, it is very much appreciated.

9. Contact

Craig Dennis (Zephni)
MugHeadStudios

Contact me by email: zephni@hotmail.co.uk

Discord: https://discord.gg/S7Y3RyUpGh