uLua

A N T

Software

uLua

ANT Software

Version v1.0.0

Sun Jul 3 2022

Table of Contents

uLua

uLua is a Game API framework for Unity. It aims to streamline the development of a Game API in Lua for your Unity Project.

uLua wraps around MoonSharp and provides an object oriented framework for API development in Lua. It works by setting up an application-wide Lua context and exposing game objects to it. Objects exposed to Lua can then be accessed in Lua scripts to implement game behaviour. In addition, user-defined Lua scripts may be executed at runtime to allow modding of your project.

uLua implements the following features:

- Script execution framework which allows Lua scripts to be executed from the Resources folder or an external directory.
- Event handling system which allows you to invoke events in C# and handle them in Lua.
- Base classes which expose your game objects and data structures to Lua in order to develop your Game API.
- Callback function system for your Game API objects.

Dependencies

- MoonSharp for Unity

Documentation

This is the complete documentation for uLua. For any further questions do not hesitate to contact support@antsoftware.co.uk.

I have put together a demo game which uses uLua, and which comes with its own documentation of the API and source code. You may find it here.

This project is the result of many hours of hard work. Please support me by leaving a review on the asset store! Follow my Twitter!

Usage Tutorial

Note: You must install the Unity MoonSharp plugin before you can use uLua. Check the *Dependencies* section above.

uLua consists of the following main scripts:

- **uLua.Lua:** A wrapper class providing an application-wide Lua context.
- **uLua.API:** Class that implements an event handling system and a script execution framework.
- **uLua.ExposedClass:** Class which exposes its instances to Lua. To use as a base for data structures which will be accessible in your Game API.
- **uLua.ExposedMonoBehaviour:** MonoBehaviour script which exposes its instances to Lua. To use as a base for game objects which will be accessible in your Game API.

The following tutorial is a thorough introduction to the different classes and scripts in the uLua toolkit and will walk you through performing basic tasks with uLua.

1. The API class

The uLua.API class sets up various aspects of the uLua framework such as script execution directories and the event handling system.

To start using uLua, add the uLua.API script to an object in your Unity scene.

The following settings can be configured in the inspector UI:

- **SceneScript** : The name of the Lua script to be executed when this scene is loaded, e.g. "MyGameScene"
- **ResourceDirectory** : Directory for Lua scripts within the Resources folder.
- **ExternalDirectory** : Directory for external Lua scripts. `Application.persistentDataPath` is used unless otherwise specified.
- **LuaScriptsExtension** : The file extension for all external Lua scripts. Set to *lua* by default.
- **AllowExternalSceneScript** : If enabled, the uLua.API class attempts to load the `SceneScript` the `ExternalDirectory` .
- **AllowExternalScripts** : If enabled, allows user scripts to be executed which extend the `SceneScript` .
- **UserScriptsPath** : Path for user scripts under `ExternalDirectory` . Scripts in this path are loaded automatically and executed if `AllowExternalScripts` is enabled.
- **EnableConsoleMessages** : If enabled, messages will be output to the console at runtime.

The uLua.API class implements various static methods which will be useful as you develop your Game API. These methods are introduced in the sections below.

For now, you can stick to the default settings and continue to the next section.

2. Executing Lua scripts

The uLua.API class provides a scripting framework which allows you to execute Lua code by wrapping around the MoonSharp Lua interpreter. Lua scripts may be executed from the Resources folder of your project or from an external directory.

The uLua.API class uses a main Lua script, named `SceneScript` , which is executed as soon the scene is loaded. Additional user scripts may be executed from an external directory to extend the Scene Script.

2.1. Using the Scene Script

The simplest way to execute a Lua script is by making use of the `SceneScript` option of the uLua.API class.

As an example, consider the following script which executes a print command.

MyGameScene.lua

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

To execute this script, create a file named ***MyGameScene.lua*** and place it in the Resources folder in your project. Select the game object which contains the uLua.API class and set the `SceneScript` option to "MyGameScene".

When you run your scene in the editor, you should notice the message "Hello World!" output to the console. Congratulations, you have executed your first Lua script!

Note: Although the file is named "MyGameScene.lua", you must omit the file extension in the inspector UI.

2.2. Loading the Scene Script Externally

The uLua.API class allows you to load the scene script from an external directory. To do so, you must enable the option `AllowExternalSceneScript` of the uLua.API class from

the inspector UI. You must then move your scene script ("MyGameScene.lua" in the above example) to the external directory.

The external directory is set to Unity's `Application.persistentDataPath` by default. For more information about this directory path, check the relevant Unity documentation.

Place your script in the appropriate directory and run the scene in the editor. You should notice the message "Hello World!" output to the console.

When the `AllowExternalSceneScript` option is enabled, the uLua.API class will attempt to load the scene script from the external directory **first** . If the scene script is not found in the external directory, it will then be loaded from the Resources folder. The intended use of this feature is to allow users to override the default implementation of scene scripts by redefining them at runtime. This is very useful if you want to allow users to modify your game behaviour.

However, there are may be times when simply extending your scene script is all that is needed. This may be achieved by enabling external user scripts.

2.3. External User Scripts

External user scripts may be used to extend the functionality of your scene script without overriding the script in the Resources folder. To enable user scripts, enable the `AllowExternalScripts` option in the inspector UI.

Because user scripts are loaded automatically, you have to set a specific path for all such scripts using the `UserScriptsPath` option. The default setting is "UserScripts/". This path is relative to the `ExternalDirectory` described above. Any scripts found in the `UserScriptsPath` will be executed by the uLua.API class **after** the `SceneScript` is executed.

As an example, you may create the following Lua script:

MyUserScript.lua

```
print ("This is a user script.");
```

Place this file under the `ExternalDirectory` in the path set by `UserScriptsPath` ("UserScripts" folder by default). Run your scene and you should notice the following two messages output to the console:

```
Hello World!
This is a user script.
```

2.4. Manual Script Execution

So far, we have used the different inspector settings of the uLua.API class to execute Lua scripts. However, the default functionality implemented in uLua.API may not always fit your project workflow. For this reason, the uLua.API class also allows you to execute Lua scripts on demand from anywhere in your project. The following two static methods are available to use in your code:

uLua.API.ExecuteExternalFile()

uLua.API.ExecuteFile()

These methods make use of the script execution framework defined by uLua.API. This means that you do not need to specify the execution directories when calling these functions. As long as a game object with your uLua.API settings is set up in your scene, you can execute external or internal Lua scripts by the path to their filename. Check the full documentation below for a list of available parameters and more information on how to use these methods.

3. Using the Event Handling System

In the previous section we went over the Lua script execution framework of uLua.API. Another main feature of uLua.API is its event handling system. Events can be very useful when implementing game behaviour. uLua allows you to invoke and handle game events in your Game API by implementing event handlers in Lua scripts.

3.1. Invoking Events

Events are invoked by the uLua.API class. The related method is static, which means it can be executed from anywhere in your project. To invoke an event named `PlayerHealthChanged`, you can use the following command anywhere in your scene:

```
uLua.API.Invoke("PlayerHealthChanged");
```

When invoking an event, you may pass any arguments as optional parameters following the event name. For instance:

```
uLua.API.Invoke("MyEventWithParameters", 5, 1, "text")
```

You can use any name you like when invoking your game events, however, the same event names must be used for the event handler registration.

3.2. Registering Event Handlers

Until an event handler has been registered, the uLua.API.Invoke() command will have no effect. Event handlers are registered and removed by the uLua.API class. The related methods are static, which means they can be executed from anywhere in your project.

To register an event handler for the `PlayerHealthChanged` event you can use the following command in C#:

```
uLua.API.RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
```

This command registers the `OnPlayerHealthChanged` global Lua function to be called when the `PlayerHealthChanged` event is invoked.

Note: Keep in mind that event handlers can be registered as members of other objects in Lua. For the sake of simplicity, we use a global event handler in this step of the tutorial.

You can now use your scene script to implement the `OnPlayerHealthChanged` function.

MyGameScene.lua

```
function OnPlayerHealthChanged()
    -- Do stuff
end
```

Any arguments passed to the uLua.API.Invoke() method are automatically handled by Lua and can be included in the event handler definition. For instance:

```
function MyHandlerWithParameters(arg1, arg2, arg3)
    -- Do stuff with arg1, arg2, arg3
end
```

The methods to register and remove event handlers are also available in Lua. As a result the same event handler may instead be registered in your Lua scene script using a similar syntax. Returning to our previous example:

MyGameScene.lua

```
function OnPlayerHealthChanged()
    -- Do stuff
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
```

These two approaches are equivalent in functionality. However, allowing event handlers to be registered by the user in Lua scripts allows for a lot more flexibility in script design.

3.3. Removing Event Handlers

Event handlers may be removed by calling the following command in C#:

```
uLua.API.RemoveEventHandlers();
```

Or the following equivalent command in a Lua script:

```
RemoveEventHandlers();
```

These commands remove all *global* event handlers.

Note: Again, keep in mind that event handlers do not need to be globals. For the sake of simplicity, we use global event handlers in this step of the tutorial.

3.4. The SceneLoaded Event

The uLua.API class invokes only one event, called `SceneLoaded` . The `SceneLoaded` event is invoked after the scene script is executed, and before any user scripts are executed.

You may register an event handler for the `SceneLoaded` event as you would for any other event:

MyGameScene.lua

```
function OnPlayerHealthChanged()
    -- Do stuff
end

function OnSceneLoaded()
    -- Do other stuff
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
```

4. Exposing Objects to Lua

So far we have shown how the scene script can be structured to handle events invoked in your Unity scene. However, the API that we have set up so far has no game objects exposed to it that would allow us to implement game behaviour.

To expose objects to the Game API, uLua provides two approaches: the uLua.ExposedClass and uLua.ExposedMonoBehaviour classes. Any script that inherits these classes will automatically expose its instances to the API.

When exposing game objects to the Game API, the uLua framework uses an object-oriented approach. Each game object is exposed to the Game API with a unique Lua handle. As we will explore in the following sections, this becomes important when developing your API and structuring your API object scripts.

4.1. The ExposedMonoBehaviour script

uLua.ExposedMonoBehaviour is a MonoBehaviour script. It may be used similarly to MonoBehaviour, but has the added functionality of exposing the game object to which it is attached to Lua.

When a class inherits uLua.ExposedMonoBehaviour, all public members of that class are accessible in Lua. This makes uLua.ExposedMonoBehaviour the base for developing your Game API.

As an example, let's say you want to expose your `Player` object to Lua. For simplicity, let's say the Player has only one member: its health. You may start by defining a `Player` script which inherits ExposedMonoBehaviour:

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour<Player> {
    public int Health = 100;
}
```

Notice the generic type parameter `<Player>` in the class definition. uLua.ExposedMonoBehaviour takes one generic type parameter. At runtime, the uLua.ExposedMonoBehaviour script uses this type parameter to register that type to Lua.

All game objects which contain the `Player` script will be exposed to the Game API when the object's `Start()` method is called. This feature may be customised by changing the `ExposeOn` setting of uLua.ExposedMonoBehaviour to `Awake`, or to `None`, allowing you to expose objects manually by using uLua.API.Expose<T>().

You may now attach the `Player` script to a game object in your scene. As an example let's name that object `MyPlayer`. The uLua.ExposedMonoBehaviour class will use the game object's name to generate a handle in Lua. This means that a game object named `MyPlayer` in your Unity Scene will be accessible by the same name in Lua.

Note: Game objects may not have spaces or special characters in their name if they are to be exposed to Lua. Object names must follow Lua naming conventions for variables.

The `Player` script we defined above has a public field named Health. Building on the *MyGameScene.lua* script from the previous section, the following script shows how the object `MyPlayer` and its member `Health` are accessible in Lua:

MyGameScene.lua

```
function OnPlayerHealthChanged()
end

function OnSceneLoaded()
    print (MyPlayer.Health);
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
```

The above script will output the health of the object `MyPlayer` to the console when the scene is loaded.

It is important to note that the Lua object `MyPlayer` is a reference, not a copy of the Unity object. This means that if the `Health` member is altered in Lua, its value in Unity will also change.

To make use of the `PlayerHealthChanged` event, we can build on the `Player` class design.

In the modification below, we have made the `Health` variable private and renamed it to `_Health`. We then implement a public getter property named `Health`, and a public method named `Damage()`. The `Damage()` method changes the value of `_Health` and invokes the `PlayerHealthChanged` event.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour<Player> {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health {
        get { return _Health; }
    }
}
```

You may then use the `Damage()` method in Lua, and also make use of the invoked event `PlayerHealthChanged` as shown below.

MyGameScene.lua

```
function OnPlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(5);
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
```

The above example should be enough to get you started on developing your Game API. You should experiment until you find a design that works for your game objects. For instance, if your Player object is comprised of several components (as is common in Unity), and you need properties of these components to be accessible in Lua, then you will have to write part of your `Player` script as a wrapper class, implementing getter and setter properties for its components as necessary.

4.2. The ExposedClass script

uLua.ExposedClass is intended for data structures which need to be exposed to the Game API. It is similar to uLua.ExposedMonoBehaviour, however, the key difference is that uLua.ExposedClass is a simple C# class instead of a MonoBehaviour script.

As an example, you may use uLua.ExposedClass to describe a weapon item in your game. A simple class definition is the following:

Weapon.cs

```
using uLua;

public class Weapon : ExposedClass<Weapon> {
    public int Damage = 0;

    // Public constructor
    public Weapon(string Name, LuaMonoBehaviour Context = null, bool ExposeOnInit = true):
base(Name, Context, ExposeOnInit) {
    }
}
```

Classes which inherit uLua.ExposedClass need to implement the public constructor as shown in the example code above. This is because the base constructor is used to expose the object to the Game API and to initialise its `Name` and `Context` properties.

Note: We have not yet covered what the `Context` object is. This will be explained in the next section.

Therefore, to expose an instance of `Weapon` to Lua, all you need to do is instantiate it with the `new` keyword. In the following example, we instantiate a `Weapon` named `Sword` in the `Awake()` method of the `Player` script.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour<Player> {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health {
        get { return _Health; }
    }

    private void Awake() {
        Weapon Sword = new Weapon("Sword");
    }
}
```

As long as this object instantiation is made somewhere in your code, you may access the `Sword` instance of the `Weapon` class as a global in Lua. Again, building on the *MyGameScene.lua* script from the previous section:

MyGameScene.lua

```
Sword.Damage = 5;

function OnPlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(Sword.Damage);
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
```

In this example we set the value of the `Sword.Damage` property to 5, and use that as a parameter when calling the method `Damage()`.

Similarly to uLua.ExposedMonoBehaviour, the class definition given above will expose all `Weapon` objects to Lua when they are instantiated. This feature may be disabled by setting the optional `ExposeOnInit` parameter of the uLua.ExposedClass constructor to `false`, allowing you to expose objects manually by using uLua.API.Expose<T>().

An example of that is shown below.

Weapon.cs

```
using uLua;

public class Weapon : ExposedClass<Weapon> {
```

```
    public int Damage = 0;

    // Public constructor
    public Weapon(string Name, LuaMonoBehaviour Context = null): base(Name, Context,
false) {
    }
}
```

4.3. Object Context

In the previous two sections we went over the basics of using uLua.ExposedClass and uLua.ExposedMonoBehaviour classes to expose objects to your Game API. The `Context` object is an important feature of these classes.

The `Context` object allows you to set a hierarchy for objects exposed to your Game API. If a `Context` object is not specified, a Lua object is defined as a global by default. This is what we have done so far. If a `Context` object is specified, Lua objects are defined as fields of the `Context` object instead. This feature may be used for organisation purposes as your API grows in scope.

As an example, we return to the `Player` and `Weapon` scripts from the previous section:

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour<Player> {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health {
        get { return _Health; }
    }

    private void Awake() {
        Weapon Sword = new Weapon("Sword");
        Weapon PlayerSword = new Weapon("Sword", this);
    }
}
```

In this case we have instantiated a second instance of `Weapon`. Both instances are named `Sword`. However, the `PlayerSword` instance is in the context of the `Player` object. This is indicated by the second parameter (keyword `this`) in the `Weapon()` constructor.

To access the non-global `Sword` object in Lua, you may use the syntax `MyPlayer.Sword`, as shown below:

MyGameScene.lua

```
Sword.Damage = 5;
MyPlayer.Sword.Damage = 2;

function OnPlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(Sword.Damage);
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
```

Here we have added a command to set the `Sword.Damage` property to 2 for the `Sword` object which is in the context of our `MyPlayer` object.

The `Context` is of type uLua.LuaMonoBehaviour, which is the base of uLua.ExposedMonoBehaviour. As a result, the `Context` must be a game object, and cannot be a uLua.ExposedClass object.

To set the context of a uLua.ExposedMonoBehaviour game object, you can use its public member `Context` or the inspector UI.

4.4. Using Object Callback Functions

In this section we will go over implementing callback functions for your Game API objects. This is a feature of both uLua.ExposedClass and uLua.LuaMonoBehaviour which allows callback methods to be invoked in Unity and handled in Lua.

The callback system is similar to the event handling system. The key difference is that callbacks are object-specific. In contrast, event handlers may be defined as globals or as members of any object.

Invoking a callback function is as simple as invoking an API event. It is achieved using the methods uLua.ExposedClass.Invoke() and uLua.ExposedMonoBehaviour.Invoke().

In the following piece of code I have invoked the `OnDamageTaken` callback in the `Player` script.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour<Player> {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
        Invoke("OnDamageTaken");
    }

    public int Health {
        get { return _Health; }
    }

    public void Awake() {
        Weapon Sword = new Weapon("Sword");
        Weapon PlayerSword = new Weapon("Sword", this);
    }
}
```

You may implement this callback function anywhere in Lua. The callback function must be a member of a `Player` object, because it is invoked by the `Player` script. In this case, our `Player` object is named `MyPlayer`, so we implement the `OnDamageTaken` callback function as member of `MyPlayer`.

MyGameScene.lua

```
Sword.Damage = 5;
MyPlayer.Sword.Damage = 2;

function OnPlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(Sword.Damage);
```

```
end

function MyPlayer:OnDamageTaken()
    print (self.Health);
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
```

In Lua, the ":" syntax in `MyPlayer:OnDamageTaken()` is used as syntactic sugar to mimic object oriented design. Defining the callback function as `MyPlayer:OnDamageTaken()` allows use of the `self` keyword to access members of this object.

In our example, the event `PlayerHealthChanged` and the callback function `OnDamageTaken` achieve the same effect. There are operational differences between the two so you must choose whether to use a callback function or an event on a case by case basis.

The main differences between the two options are the following:

- Event handlers may be defined as globals or as a member of any object. Callback functions are object-specific.
- To implement a callback function, you must know the Lua handle of an object (`Context.Name`) or have a reference to the object, e.g. returned from a function.

4.5. Hiding Object Members in Lua

Sometimes it is unsafe to expose certain class members to your Game API. To prevent a public member from being exposed to Lua, you may add the relevant MoonSharp attribute to a class:

```
[MoonSharpHideMember("MemberName")]
```

For instance, if you wanted to hide the `Awake()` method of the `Player` script described above, you may use the following syntax:

Player.cs

```
using MoonSharp.Interpreter;
using uLua;

[MoonSharpHideMember("Awake")]
public class Player : ExposedMonoBehaviour<Player> {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
        Invoke("OnDamageTaken");
    }

    public int Health {
        get { return _Health; }
    }

    public void Awake() {
        Weapon Sword = new Weapon("Sword");
        Weapon PlayerSword = new Weapon("Sword", this);
    }
}
```

This would prevent the `Awake()` method from being called within the Game API. This could also be achieved by defining the `Awake()` method as `protected` or `private`. However, that is not an option in cases where a specific member must be public so that other scripts in your project can have access to it.

5. Further Support

This covers a large portion of the features in uLua! I hope that this tutorial together with the documentation are enough to get you started on your API development. However, for any further questions do not hesitate to contact support@antsoftware.co.uk.

I have put together a demo paddle game project which utilises uLua, and which comes with its own documentation of the API and source code. You may find it here.

This project is the result of many hours of hard work. Please support me by leaving a review on the asset store! Also, follow my Twitter!

Namespace Index

Namespace List

Here is a list of all documented namespaces with brief descriptions:

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Namespace Documentation

uLua Namespace Reference

Namespace containing the uLua project.

Classes

- class API
  This script sets up a Game API framework to your Unity scene.
- class ExposedClass
  Class structure exposed to Lua. You should use this class as a base for your API data structures.
- class ExposedMonoBehaviour
  MonoBehaviour script exposed to Lua. You should use this class as a base for your API game objects.
- interface IHasLuaIndexer
  Interface used to implement a fully indexed Lua object.
- class IndexedUserDataDescriptor
  Custom user data descriptor to more accurately implement Lua syntax in the Moonsharp interpreter. Makes use of the IHasLuaIndexer interface.
- class Lua
  Wrapper class which streamlines use of the MoonSharp Lua context.
- class LuaClass
  Class structure which establishes a Lua object framework. For internal use.
- class LuaMonoBehaviour
  MonoBehaviour script which establishes a Lua object framework. For internal use.

Enumerations

- enum ExposeOn
  Indicates when to expose an object to Lua.

Detailed Description

Namespace containing the uLua project.

Class Documentation

uLua.API Class Reference

This script sets up a Game API framework to your Unity scene.

Public Member Functions

- void AddUserScript (string Name)
  Adds a script to the user scripts by name.

- void ClearUserScripts ()
  Clears the user scripts list.

- void RemoveUserScript (string Name)
  Removes the specified script from the user scripts.

Static Public Member Functions

- static bool ExecuteExternalFile (string File, string Index="", bool Force=false)
  Executes a script from the external directory.

- static bool ExecuteFile (string File, string Index="", bool Force=false)
  Executes a script from the resource directory.

- static void Expose< T > (T Object, LuaMonoBehaviour Context=null)
  Exposes an object to Lua.

- static void Invoke (string Event, params object[] args)
  Invokes a game event.

- static void RegisterEventHandler (string Event, string HandlerName, LuaMonoBehaviour Context=null)
  Registers an event handler to be called when an event is invoked.

- static void RemoveEventHandlers (LuaMonoBehaviour Context=null)
  Removes all global event handlers or event handlers registered in a specific context.

- static void RegisterIndexedType< T > ()
  Registers a type that implements the IHasLuaIndexer interface to Lua.

- static void RegisterType< T > ()
  Registers a type to Lua.

Static Public Attributes

- static string **ExternalDirectory** = ""

Directory for external Lua scripts. The path is determined at run-time.

- static string **ResourceDirectory** = ""
  Directory for Lua scripts within the Resources folder.

- static string **LuaScriptsExtension** = ""
  File extension for all external Lua scripts.

Detailed Description

This script sets up a Game API framework to your Unity scene.

Includes methods to execute Lua scripts, expose game objects to the Game API, and invoke events. Inherits from MonoBehaviour. Intended use is to add to an object in the scene and set up for use.

This script invokes the `SceneLoaded` event when the scene is loaded.

Member Function Documentation

void uLua.API.AddUserScript (string *Name*) `[inline]`

Adds a script to the user scripts by name.

User scripts are executed from the specified path in the `ExternalDirectory` after the scene is loaded.

void uLua.API.ClearUserScripts () `[inline]`

Clears the user scripts list.

This method has no effect on the Game API if the user scripts have already been executed.

static bool uLua.API.ExecuteExternalFile (string *File*, string *Index* = "", bool *Force* = false) `[inline]`, `[static]`

Executes a script from the external directory.

Parameters

File	The Lua file to execute. The file extension must be omitted.
Index	(Optional) A unique identifier for this script. Not recommended to leave blank, especially if the same script is likely to be executed more than once.
Force	(Optional) If true, the method will overwrite any previous script with the same Index. Otherwise, any previous script with the same Index will be called from a hash table.

static bool uLua.API.ExecuteFile (string *File*, string *Index* = "", bool *Force* = false) `[inline]`, `[static]`

Executes a script from the resource directory.

Parameters

File	The Lua file to execute. The file extension must be omitted.
Index	(Optional) A unique identifier for this script. Not recommended to leave blank, especially if the same script is likely to be executed more than once.
Force	(Optional) If true, the method will overwrite any previous script with the same Index. Otherwise, any previous script with the same Index will be called from a hash table.

static void uLua.API.Expose< T > (T *Object*, LuaMonoBehaviour *Context* = null)[inline], [static]

Exposes an object to Lua.

Called internally by uLua.ExposedMonoBehaviour.Start(). Can be called manually at an earlier point or for objects which have `ExposeOnStart` disabled.

Parameters

Object	The object to be exposed to Lua. The generic type `T` must implement the IHasLuaIndexer interface.
Context	The Lua context of the object. If null, the object will be exposed as a global.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

static void uLua.API.Invoke (string *Event*, params object[] *args*)[inline], [static]

Invokes a game event.

Parameters

Event	The name of the event to be invoked.
args	(Optional) Parameters for the event handler.

static void uLua.API.RegisterEventHandler (string *Event*, string *HandlerName*, LuaMonoBehaviour *Context* = null)[inline], [static]

Registers an event handler to be called when an event is invoked.

This method is exposed to Lua as a global and may be called from a Lua script.

Parameters

Event	The name of the event.
HandlerName	The name of the event handler function.
Context	(Optional) The object that contains the event handler implementation. If not specified, the event handler will be global.

static void uLua.API.RegisterIndexedType< T > ()[inline], [static]

Registers a type that implements the IHasLuaIndexer interface to Lua.

Intended for use with LuaMonoBehaviour and LuaClass.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

static void uLua.API.RegisterType< T > () `[inline]`, `[static]`

Registers a type to Lua.

Any C# or Unity type may be registered to use within the API. Use with caution.

static void uLua.API.RemoveEventHandlers (LuaMonoBehaviour *Context* = null) `[inline]`, `[static]`

Removes all global event handlers or event handlers registered in a specific context.

This method is exposed to Lua as a global and may be called from a Lua script or within C#.

Parameters

Context	(Optional) The object that contains the event handler implementation. If not specified, all global event handlers will be removed.

void uLua.API.RemoveUserScript (string *Name*) `[inline]`

Removes the specified script from the user scripts.

User scripts are executed from the specified path in the `ExternalDirectory` after the scene is loaded.

uLua.ExposedClass< T > Class Template Reference

Class structure exposed to Lua. You should use this class as a base for your API data structures.

Public Member Functions

- ExposedClass (string Name, LuaMonoBehaviour Context=null, bool ExposeOnInit=true)
  Public constructor. Exposes this object to Lua.

Additional Inherited Members

Detailed Description

Class structure exposed to Lua. You should use this class as a base for your API data structures.

Instances of this class are automatically exposed to Lua. The generic type parameter `T` is used to register that type to Lua. All public members of derived classes will be exposed to Lua. Inherits LuaClass.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

Constructor & Destructor Documentation

uLua.ExposedClass< T >.ExposedClass (string *Name*, LuaMonoBehaviour *Context* = `null`, bool *ExposeOnInit* = `true`) `[inline]`

Public constructor. Exposes this object to Lua.

Parameters

Name	Sets the name of the object exposed to Lua.
Context	Sets the context of the object exposed to Lua.
ExposeOnInit	(Optional) Enables/disables the automatic exposure of this object to Lua.

uLua.ExposedMonoBehaviour< T > Class Template Reference

MonoBehaviour script exposed to Lua. You should use this class as a base for your API game objects.

Public Attributes

- ExposeOn **ExposeOn** = ExposeOn.Start
  Indicates when this object will be exposed to Lua.

Protected Member Functions

- virtual void Awake ()
  Exposes the game object to Lua if ExposeOn is set to 'Awake'. The Context object must be set prior to this method being called.

- virtual void Start ()
  Exposes the game object to Lua if ExposeOn is set to 'Start'. The Context object must be set prior to this method being called.

Additional Inherited Members

Detailed Description

MonoBehaviour script exposed to Lua. You should use this class as a base for your API game objects.

Instances of this class are automatically exposed to Lua. The generic type parameter `T` is used to register that type to Lua. All public members of derived classes will be exposed to Lua. Inherits LuaMonoBehaviour.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

Member Function Documentation

virtual void uLua.ExposedMonoBehaviour< T >.Awake () `[inline]`, `[protected]`, `[virtual]`

Exposes the game object to Lua if ExposeOn is set to 'Awake'. The Context object must be set prior to this method being called.

This method is called by Unity Engine during game object initialisation. Objects will not be exposed to Lua if `ExposeOn` is set to None.

virtual void uLua.ExposedMonoBehaviour< T >.Start () `[inline]`, `[protected]`, `[virtual]`

Exposes the game object to Lua if ExposeOn is set to 'Start'. The Context object must be set prior to this method being called.

This method is called by Unity Engine during game object initialisation. Objects will not be exposed to Lua if `ExposeOn` is set to None.

uLua.IHasLuaIndexer Interface Reference

Interface used to implement a fully indexed Lua object.

Properties

- DynValue this[DynValue Key] `[get, set]`
  Returns a Lua value indexed by the DynValue Key.

- DynValue this[string Key] `[get, set]`
  Returns a Lua value indexed by the string key.

- LuaMonoBehaviour Context `[get, set]`
  Used to access/set the context of an object.

- string Handle `[get]`
  Used to access the unique Handle of a Lua object.

- bool IsExposed `[get]`
  Used to track if an object has been exposed to Lua.

- string Name `[get]`
  Used to access the name of an object.

Detailed Description

Interface used to implement a fully indexed Lua object.

Defines various properties (Context, Name, Handle, etc.) which are implemented in LuaClass and LuaMonoBehaviour.

Property Documentation

LuaMonoBehaviour uLua.IHasLuaIndexer.Context `[get]`, `[set]`

Used to access/set the context of an object.

The `Context` object represents the parent of an object in Lua. If the `Context` is null, the object will be global in Lua. Must be of type uLua.LuaMonoBehaviour.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

string uLua.IHasLuaIndexer.Handle `[get]`

Used to access the unique Handle of a Lua object.

The unique `Handle` of a Lua object is defined as: `Context.Name` . If an object's context is null, its unique handle is simply its name.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

bool uLua.IHasLuaIndexer.IsExposed `[get]`

Used to track if an object has been exposed to Lua.

Objects may be marked as exposed by calling the `Expose()` method.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

string uLua.IHasLuaIndexer.Name `[get]`

Used to access the name of an object.

The name of an object cannot be changed after it has been exposed to Lua.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

DynValue uLua.IHasLuaIndexer.this[DynValue Key] `[get], [set]`

Returns a Lua value indexed by the DynValue Key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

DynValue uLua.IHasLuaIndexer.this[string Key] `[get], [set]`

Returns a Lua value indexed by the string key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

uLua.IndexedUserDataDescriptor< T > Class Template Reference

Custom user data descriptor to more accurately implement Lua syntax in the Moonsharp interpreter. Makes use of the IHasLuaIndexer interface.

Detailed Description

Custom user data descriptor to more accurately implement Lua syntax in the Moonsharp interpreter. Makes use of the IHasLuaIndexer interface.

Allows objects in Lua to be indexed with both the '.' and '[]' operators. Credit to user tw39124 from the MoonSharp forum for the original implementation.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

uLua.Lua Class Reference

Wrapper class which streamlines use of the MoonSharp Lua context.

Static Public Member Functions

- static void Call (DynValue Function, params object[] args)
  Calls a Lua function with optional parameters.

- static void ExecuteScript (string Code, string Index="", bool Force=false)
  Executes the specified code within the Lua context.

- static T Get< T > (string Name)
  Finds and returns the Lua global with the specified name.

- static void ObjectCall< T > (T Object, string FunctionName, params object[] args)
  Calls a Lua object's function.

- static DynValue ValueToLuaValue (object Value)
  Converts an object to a DynValue.

- static void Remove (string Name)
  Removes a global from the Lua context.

- static void Set (string Index, object Value)
  Exposes an object as a global with the specified index in Lua.

Properties

- static DynValue NewFunction `[get]`
  Returns a new Lua function.

- static Table NewTable `[get]`
  Returns a new Lua table.

Detailed Description

Wrapper class which streamlines use of the MoonSharp Lua context.

Implemented as a static class to ensure the Lua context is available application-wide.

Member Function Documentation

static void uLua.Lua.Call (DynValue *Function*, params object[] *args*) `[inline], [static]`

Calls a Lua function with optional parameters.

Parameters

Function	The `DynValue` of a Lua function.
args	(Optional) Parameters for the Lua function.

static void uLua.Lua.ExecuteScript (string *Code*, string *Index* = `""`, bool *Force* = `false`) `[inline]`, `[static]`

Executes the specified code within the Lua context.

Parameters

Code	The Lua code to execute.
Index	(Optional) A unique identifier for this script. Not recommended to leave blank, especially if the same script is likely to be executed more than once.
Force	(Optional) If true, the method will overwrite any previous script with the same index. Otherwise, any previous script with the same index will be called from a hash table.

static T uLua.Lua.Get< T > (string *Name*) `[inline]`, `[static]`

Finds and returns the Lua global with the specified name.

The returned object is converted to the type specified by the generic type parameter `T` .

static void uLua.Lua.ObjectCall< T > (T *Object*, string *FunctionName*, params object[] *args*) `[inline]`, `[static]`

Calls a Lua object's function.

Parameters

Object	The object whose function will be called. The generic type `T` must implement the `IHasLuaIndexer` interface.
FunctionName	The name of the callback function to be called.
args	(Optional) Parameters for the callback function.

Type Constraints

T : *IHasLuaIndexer*

static void uLua.Lua.Remove (string *Name*) `[inline]`, `[static]`

Removes a global from the Lua context.

Any Lua objects defined as members of the global are invalidated.

static void uLua.Lua.Set (string *Index*, object *Value*) `[inline]`, `[static]`

Exposes an object as a global with the specified index in Lua.

For object types that implement the IHasLuaIndexer interface, use `uLua.API.Expose()` instead.

Parameters

Index	The index, or name, of the value within Lua.
Object	The value to be defined as a global within Lua.

static DynValue uLua.Lua.ValueToLuaValue (object *Value*) `[inline], [static]`

Converts an object to a DynValue.

May be used with any C# object type.

Property Documentation

DynValue uLua.Lua.NewFunction `[static], [get]`

Returns a new Lua function.

Used as a placeholder for empty callback functions.

Table uLua.Lua.NewTable `[static], [get]`

Returns a new Lua table.

Creates a new Table type Lua value.

uLua.LuaClass Class Reference

Class structure which establishes a Lua object framework. For internal use.

Public Member Functions

- LuaClass (string Name, LuaMonoBehaviour Context=null)
  Public constructor.

- void Expose ()
  Used to raise a flag when an object is exposed to Lua.

- void InvokeLua (string FunctionName, params object[] args)
  Invokes a Lua callback function.

- void Register (string FunctionName, string Code="", string args="")
  Registers a Lua callback function. This method is available in Lua.

Properties

- DynValue this[DynValue Key] `[get, set]`
  Returns a Lua value indexed by the DynValue Key.

- DynValue this[string Key] `[get, set]`
  Returns a Lua value indexed by the string key.

- LuaMonoBehaviour Context `[get, set]`
  Used to access/set the context of an object.

- string Handle `[get]`
  Used to access the unique Handle of a Lua object.

- bool IsExposed `[get]`
  Used to track if an object has been exposed to Lua.

- string Name `[get, protected set]`
  Used to access the name of an object.

Detailed Description

Class structure which establishes a Lua object framework. For internal use.

Instances of this class can be exposed to Lua by calling `uLua.API.Expose()` . All public members of derived classes will be exposed to Lua. Prior to exposing an object of type uLua.LuaClass, the relevant type `T` must be registered to Lua by calling `uLua.API.RegisterIndexedType()` .

Constructor & Destructor Documentation

uLua.LuaClass.LuaClass (string *Name*, LuaMonoBehaviour *Context* = null)[inline]

Public constructor.

Parameters

Name	Sets the name of the object exposed to Lua.
Context	Sets the context of the object exposed to Lua.

Member Function Documentation

void uLua.LuaClass.Expose ()[inline]

Used to raise a flag when an object is exposed to Lua.

The property `IsExposed` can be used to check if this method has been called. The `IsExposed` flag cannot be reset.

Implements uLua.IHasLuaIndexer.

void uLua.LuaClass.InvokeLua (string *FunctionName*, params object[] *args*)[inline]

Invokes a Lua callback function.

The callback function must be implemented in a Lua script as a member of this object. Alternatively, you can register a callback function by explicitly calling uLua.LuaMonoBehaviour.Register() with the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be called.
args	(Optional) Parameters for the callback function.

void uLua.LuaClass.Register (string *FunctionName*, string *Code* = "", string *args* = "")[inline]

Registers a Lua callback function. This method is available in Lua.

Used to register a callback function by providing the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be registered.
Code	(Optional) The body of the function to be registered in Lua. If omitted, an empty function with the specified name is defined.
args	(Optional) Parameters for the callback function. Must be separated by comma as they would be in a Lua function definition, e.g. 'arg1,arg2'.

Property Documentation

LuaMonoBehaviour uLua.LuaClass.Context `[get], [set]`

Used to access/set the context of an object.

The `Context` object represents the parent of an object in Lua. If the `Context` is null, the object will be global in Lua. Must be of type uLua.LuaMonoBehaviour.

Implements uLua.IHasLuaIndexer.

string uLua.LuaClass.Handle `[get]`

Used to access the unique Handle of a Lua object.

The unique `Handle` of a Lua object is defined as: `Context.Name` . If an object's context is null, its unique handle is simply its name.

Implements uLua.IHasLuaIndexer.

bool uLua.LuaClass.IsExposed `[get]`

Used to track if an object has been exposed to Lua.

Objects may be marked as exposed by calling the `Expose()` method.

Implements uLua.IHasLuaIndexer.

string uLua.LuaClass.Name `[get], [protected set]`

Used to access the name of an object.

The name of an object cannot be changed after it has been exposed to Lua.

Implements uLua.IHasLuaIndexer.

DynValue uLua.LuaClass.this[DynValue Key] `[get], [set]`

Returns a Lua value indexed by the DynValue Key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.IHasLuaIndexer.

DynValue uLua.LuaClass.this[string Key] `[get], [set]`

Returns a Lua value indexed by the string key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.IHasLuaIndexer.

uLua.LuaMonoBehaviour Class Reference

MonoBehaviour script which establishes a Lua object framework. For internal use.

Public Member Functions

- void Expose ()
  Used to raise a flag when an object is exposed to Lua.

- void InvokeLua (string FunctionName, params object[] args)
  Invokes a Lua callback function.

- void Register (string FunctionName, string Code="", string args="")
  Registers a Lua callback function. This method is available in Lua.

Protected Member Functions

- virtual void OnDestroy ()
  Removes the object from Lua.

Properties

- DynValue this[DynValue Key] `[get, set]`
  Returns a Lua value indexed by the DynValue Key.

- DynValue this[string Key] `[get, set]`
  Returns a Lua value indexed by the string key.

- LuaMonoBehaviour Context `[get, set]`
  Used to access/set the context of an object.

- string Handle `[get]`
  Used to access the unique Handle of a Lua object.

- bool IsExposed `[get]`
  Used to track if an object has been exposed to Lua.

- string Name `[get]`
  Used to access the name of an object.

Detailed Description

MonoBehaviour script which establishes a Lua object framework. For internal use.

Instances of this class can be exposed to Lua by calling `uLua.API.Expose()` . All public members of derived classes will be exposed to Lua. Prior to exposing an object of type

uLua.LuaMonoBehaviour, the relevant type `T` must be registered to Lua by calling `API.RegisterIndexedType()`.

Member Function Documentation

void uLua.LuaMonoBehaviour.Expose () `[inline]`

Used to raise a flag when an object is exposed to Lua.

The property `IsExposed` can be used to check if this method has been called. The `IsExposed` flag cannot be reset.

Implements uLua.IHasLuaIndexer.

void uLua.LuaMonoBehaviour.InvokeLua (string *FunctionName*, params object[] *args*) `[inline]`

Invokes a Lua callback function.

The callback function must be implemented in a Lua script as a member of this object. Alternatively, you can register a callback function by explicitly calling uLua.LuaMonoBehaviour.Register() with the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be called.
args	(Optional) Parameters for the callback function.

virtual void uLua.LuaMonoBehaviour.OnDestroy () `[inline]`, `[protected]`, `[virtual]`

Removes the object from Lua.

This method is called by Unity Engine when a game object is destroyed.

void uLua.LuaMonoBehaviour.Register (string *FunctionName*, string *Code* = `""`, string *args* = `""`) `[inline]`

Registers a Lua callback function. This method is available in Lua.

Used to register a callback function by providing the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be registered.
Code	(Optional) The body of the function to be registered in Lua. If omitted, an empty function with the specified name is defined.
args	(Optional) Parameters for the callback function. Must be separated by comma as they would be in a Lua function definition, e.g. 'arg1,arg2'.

Property Documentation

LuaMonoBehaviour uLua.LuaMonoBehaviour.Context `[get]`, `[set]`

Used to access/set the context of an object.

The `Context` object represents the parent of an object in Lua. If the `Context` is null, the object will be global in Lua. Must be of type uLua.LuaMonoBehaviour.

Implements uLua.IHasLuaIndexer.

string uLua.LuaMonoBehaviour.Handle `[get]`

Used to access the unique Handle of a Lua object.

The unique `Handle` of a Lua object is defined as: `Context.Name` . If an object's context is null, its unique handle is simply its name.

Implements uLua.IHasLuaIndexer.

bool uLua.LuaMonoBehaviour.IsExposed `[get]`

Used to track if an object has been exposed to Lua.

Objects may be marked as exposed by calling the `Expose()` method.

Implements uLua.IHasLuaIndexer.

string uLua.LuaMonoBehaviour.Name `[get]`

Used to access the name of an object.

The name of an object cannot be changed after it has been exposed to Lua.

Implements uLua.IHasLuaIndexer.

DynValue uLua.LuaMonoBehaviour.this[DynValue Key] `[get]` , `[set]`

Returns a Lua value indexed by the DynValue Key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.IHasLuaIndexer.

DynValue uLua.LuaMonoBehaviour.this[string Key] `[get]` , `[set]`

Returns a Lua value indexed by the string key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.IHasLuaIndexer.

Index