A Scriptable 2D Game Engine

Cyrus Hanlon

December 13, 2017

Contents

1 Introduction

There are many game engines that already exist but nothing combines ease of use with raw speed.

From Unreal Engine [**UE**] with its complete design versus much lighter weight engines such as LÖVE [**LOVE**], there is a very wide range of technologies and techniques at work. There is a definitive gap between large heavy engines and the small and lightweight that I intend to fill. By having many features written into the core engine but keeping a lightweight and easy to use API at the front, it will be possible to quickly learn how to prototype games, as well as, achieve excellent real world performance.

A very important part of any game engine is the interface between developer and the inner workings. There are many scripting languages that can be chosen from and there are many important factors that come into play when selecting a language to use, the most important factors include: popularity, speed and learning curve. Lua is at the top of the pile when it comes to games. Other languages such as JavaScript and Python are extremely popular in other areas but are not anywhere near as often used in games as they are more difficult to bind.

Embedded scripting languages need to have access to core elements of the game engine to allow for developers to create fully fledged games without having to touch the lower level. Having the core engine open source also allows game developers to extend the Lua interface as required and push the changes to the repository.

2 Lua

2.1 About Lua

Lua was created for extending applications, there was an increase in demand for customisation and no was language available that combined procedural features with powerful data description facilities. [**Ierusalimschy02software**] Lua came from the languages SOL and DEL [**Ierusalimschy01theevolution**] both of which were made by the company Tecgraf.

SOL is an acronym for Simple Object Language was a specialized data description language used for customising software. The SOL syntax was strongly influenced by the syntax of BiBTeX [**Lamport:1989:LDP:63364**]. DEL is an acronym for Data Entry Language, an entity in DEL is essentially an object or record in other languages [**Ierusalimschy01theevolution**]. Lua was created as a unique combination of the two more primitive languages, it was designed to avoid unique syntax so that anyone could use it effectively with limited training and for the language to have built in type checking.

Lua is written in C and so, is very cross platform [**LuaSource**], it is used in projects ranging from Cheat Engine [**CheatEngine**], an open source memory scanner and hex editor for Windows, to Android and iOS games using something such as Cocos2d [**COCOS2D**], an open source software framework used in the creation of apps or games. Therefore Lua is widely used thanks to its simple learning curve, ease of implementation using the provided minimalistic C API and its fast execution when compared to similar languages. It is also used in many non game settings such as in embedded systems using eLua [**eLua**] thanks to its low memory footprint.

Lua allows for loading modules at run time that were not required at build time, this allows for extremely flexible program extensions. A module is made available through the use of an ordinary table with functions or variables being accessed through the table. Loading, compiling then executing Lua files from within Lua is easily possible and allows for things such as hot reloading code vastly improving the development rate as developers could just save a file and the Lua program would simply start using it.

2.1.1 Running Lua

Lua is not interpreted directly but is instead compiled into bytecode, this is then run on the Lua virtual machine. Because of this Lua is a fast language, considering how high level it is. Lua used a stack based virtual machine

initially but with the release of Lua 5.0, Lua began to use a register based machine. It also uses a stack for allocating activation records where the registers live [**IerusalimschyImplementation**].

First class functions with lexical scoping is difficult to implement for languages that use a stack based vm, Lua uses a stack for local variables and a heap for when they go out of scope while being referenced to by nested functions [**IerusalimschyImplementation**]. Using this concept Lua successfully implements features such as closures that rely on register based VMs.

Lua can also be used from the command line which allows for productivity scripts to be written and used without the need for it to be embedded into another application. Lua can also be used as a standalone program using the tool srLua which was created by one of the Lua authors [**tecgrafLibAndTools**], this allows the language to be used almost anywhere on almost anything.

2.1.2 Syntax

The syntax of Lua was designed so that it combined simple procedural syntax with powerful data description constructs, the general syntax is very easy to learn with its roots in SOL and DEL, both were designed for ease of use. It borrows much of its syntax from other languages, its comment syntax, for example, is similar to that of Haskell and SQL. It's control flow is similar to that of pascal using keywords such as `then` and `end` for `if` statements. It is, however, renowned for being very easy to write with its simple syntax and extremely powerful table constructs. A quirk when working with Lua is the fact that tables start at an index of 1 instead of 0 as is very common throughout programming languages.

An example of Lua's control statements are shown in listing **??**. The use of `elseif` as a single keyword is useful and avoids the need to use an `end` per `if`. The conditions in each of the statements do not need to be wrapped in parenthesis and allow for logical operators.

```
1  local x = 30
2  if x > 25 then
3      print("x is less than 25")
4  elseif x < 35 then
5      print("x is greater than 35")
6  end -- prints out x is less than 25
7
8  while true do
9      print("loop!")
10 end -- this loops forever printing loop!
```

```
11
12 repeat
13     print("loop!")
14 until false -- this loops forever printing loop!
15
16 for i=1, 10, 3 do
17     print(i)
18 end -- prints 1 4 7 10
```

Listing 1: Lua Syntax Sample 1

There are two types of `for` loop in Lua, one is a numeric `for` and is shown in listing **??**, the other is the generic `for` that uses an iterator and allows for the traversal of all values. The numeric `for` uses a very simple syntax and so is very easy to write, the first argument is the starting value of the iteration variable, the second is the end value and the third is the value to add with each loop.

The logical operators in Lua are `and`, `or` and `not`. Logical operators and control statement expressions take `nil` and `false` as `false` and anything else as `true` [**LuaMan**] This allows for some unique syntax that more advanced Lua programmers will use to reduce the amount of code required.

Each of the three logical operators behave differently when used in this way. The `and` operator returns its first argument if it is `false` and otherwise returns its second argument. The `or` operator returns its first argument if this value isn't `false` and returns its second argument otherwise. The `not` operator always returns `true` or `false` based on the argument. The Lua 5.3 manual [**LuaMan**] has an excellent set of examples as shown in listing **??**.

```
1 10 or 20            --> 10
2 10 or error()       --> 10
3 nil or "a"          --> "a"
4 nil and 10          --> nil
5 false and error()   --> false
6 false and nil       --> false
7 false or nil        --> nil
8 10 and 20           --> 20
```

Listing 2: Lua Syntax Sample 2 (taken from [**LuaMan**])

Using this behaviour of logical operators allows for quickly and easily assigning variables using, for example, a configuration file.

Lua allows for assigning multiple variables on the same line as shown in listing **??**. When used in conjunction with returning multiple values from a function this can reduce the amount of code required and therefore improve

development time.

```lua
a, b = 100, 200
print(a + b) -- prints out 300
```

Listing 3: Lua Syntax Sample 3

With the introduction of Lua 5.2, the `bit32` library added support for bitwise operations and with Lua 5.3 came native support for bitwise operators. Behaviour is exactly the same as in any other language with support for them as is shown in listing **??**.

```lua
print(1 & 2)  -- bitwise and prints out 0
print(3 | 4)  -- bitwise or prints out 7
print(5 ~ 6)  -- bitwise xor prints out 3
print(7 >> 1) -- bitwise left shift prints out 3
print(8 << 1) -- bitwise left shift prints out 16
print(~9)     -- bitwise not prints out -10
```

Listing 4: Lua Bitwise Operations

Lua has some syntactic sugar which helps to simplify what the Lua developer will have to interact with.

```lua
-- Inline declaration of a table
local tab = {
    foo= 45,
    tab = {
        x = 23
    },
    place = 10
}
```

Listing 5: Syntactic Sugar Sample 1

In listing **??** we can see how to inline declare a table with a sub table that is also inline declared.

```lua
function tab.func(self, y)
    print(self.tab.x + y)
end

tab1.func(tab, 25)
```

Listing 6: Syntactic Sugar Sample 2

We can see how to declare a member function using some syntactic sugar in listing **??**. Which is equivalent to listing **??**.

```lua
-- Declaration of an example member-like function
function tab:func(y)
    print(self.tab.x + y)
end

tab:func(25)
```

Listing 7: Syntactic Sugar Sample 3

There are also several ways to access entries within a table all of which can be mixed and matched based on the programmers preference.

```lua
print(tab.place)          -- prints out 10
print(tab["place"])       -- prints out 10
print(tab.tab["x"])       -- prints out 23
```

Listing 8: Syntactic Sugar Sample 4

Lua's support for coercion allows for the automatic conversion of strings and numbers, this helps developers with rapid development as they would not need to have to think about converting types in many situations when Lua itself handles it for them. Bitwise operators always convert numbers to integers, mathematic operations always convert operands to floats [**LuaMan**].

String concatenation using the `..` operator converts numbers to strings wherever is needed but using the `string.format` function will allow for fine tuned control of how the number will be converted.

Examples of Lua's coercion in action are shown in listing **??**.

```lua
print(1 .. 2.2 .. "three")          -- prints out 12.2three
print("1" & "2")                    -- prints out 0
print(1.1 & 2)                      -- errors out
print("1.1" / 2.2)                  -- prints out 0.5
print(string.format("%f", "5.5"))   -- prints out 5.500000
```

Listing 9: Coercion Sample 1

Lua's tables have a length operator in the form of `#`, it returns the length of a table as would be found by the `ipairs` operator. It counts through integers starting at 1 and stopping when the first nil value is found, this means that the operator is only useful for finding the length of array-like tables.

2.1.3 Functions

Lua allows for a function to return multiple values, this is an interesting concept and can have some downsides, it can make it quite difficult to correctly name a function to reflect the return values. It is a very powerful feature and if handled correctly can improve productivity especially when combined with Lua's variable number of arguments. An example is shown in listing **??**.

```
1 function sum(...)
2     local args={...}
3     local ans = 0
4     for k, v in pairs(args) do
5         ans = ans + v
6     end
7     -- returns the sum of all values and the number of values
8     return ans, #args
9 end
```

Listing 10: Functions Sample 1

Functions, in Lua, are first class values with proper lexical scoping [**Ierusalimschy:2013:PLT:25026**
A first class value in Lua means that a function is essentially the same as any other regular variable, this allows a Lua developer to pass functions into other functions as arguments, return functions, use functions of keys and values of tables and can be assigned to a variable. This is a very valuable trait for a high level scripting language to have as it allows for very interesting techniques such as functional programming, closures are fully supported in Lua as a result of this.

Lua has proper tail calls which allows for recursion. If the last thing that a function does is call another function, the calling function is no longer required and is no longer required on the stack, this allows for an infinite recursion without overflowing the stack [**Ierusalimschy:2013:PLT:2502646**]. An example of a Lua tail call is shown in listing **??**.

```
1 function func(x) return 50 end
2
3 function tailCall(x)
4     return func(x)
5 end
```

Listing 11: Functions Sample 2

Without proper tail calls, recursion could overflow the stack but with proper

tail calls this is impossible and fits in perfectly with a high level scripting language such as Lua.

2.1.4 Tables

Tables are the only data structure in Lua but are extremely flexible and can be used to provide the functionality of almost any data structure as required. They are very similar in use to a hash map and allow for any variable of any type to be assigned to a key or value within the same table with the exception of `nil`, it is also possible to treat a table similar to an array as shown in listing **??**.

```
1 -- declaration of a table using numerical indices
2 local array = {
3     "this is a string",
4     105,
5     function(x) print("this is a function"..x) end
6 }
7
8 print(array[1])              -- prints out this is a string
9 print(array[2])              -- prints out 105
10 array[3]("!")               -- prints out this is a function!
```

Listing 12: Tables in Lua Sample 1

Lua has a variety of built in functions for the use of manipulating and getting data from tables all available through the table library [**LuaMan**].

To loop over all key-value pairs in a table lua provides the `ipairs` and `pairs` iterators. Used with a `for` loop a Lua developer is provided with a powerful method of quickly and easily accessing anything within a table as shown in listing **??**.

```
1 local tab = {
2     foo = "this is a string",
3     105,
4     "this is another string",
5     [333] = function(x) print("this is a function"..x) end
6 }
7
8 for k,v in pairs(tab) do
9     print(k, v)
10 end
11 --[[ prints out the following:
12 1    105
13 2    this is another string
```

```
14 333 function: 0x1663b90
15 foo this is a string
16 ]]
17
18 for k,v in ipairs(tab) do
19     print(k, v)
20 end
21 --[[ prints out the following:
22 1   105
23 2   this is another string
24 ]]
```

Listing 13: Tables in Lua Sample 2

The `pairs` function iterates over all elements in a table while the `ipairs` function will return all key-value pairs where the keys are numbers, starting at the first index and stopping when it hits the first nil value. [**Ierusalimschy:2013:PLT:2502646**]

Getting the length of a Lua array is found by calling `table.getn(tab)` however, to find the size of a table with mixed key types or non-consecutive indices, a generic `for` loop with a count variable would have to be used. In cases where mixed key types are used it shouldn't be required to know the size as the pairs cannot be accessed by a normal for loop in any case.

Weak tables are tables that consist of elements that are weak references, this can be the keys or the values, if the only reference left to an object is a weak reference the garbage collector collects it. They are often used when a word needs to be reserved without actually creating an object in memory.

2.1.5 Metatables and Object Oriented Programming

All tables in Lua are permitted to have a metatable[**Ierusalimschy:2013:PLT:2502646**]. They allow for changing the behaviour of various aspects of a lua table, such as the add operator, by setting the `__add` value to a new function this code will be called instead of the default add functionality as is shown in listing **??**.

```
1 local metatable = {
2     __add = function (a, b)
3         return 5 -- the result will always be 5
4     end
5 }
6
7 t1 = {}
8 -- set the meta table of t1 to metatable
9 setmetatable(t1, metatable)
10 local t2 = t1 + t1
```

```
11 print(t2) -- prints out 5
```

Listing 14: Metatables Sample 1

These functions are called metamethods and allow for any interaction with a table to be altered. While classes do not exist in Lua, using a meta table and first class value functions, behaviour similar to object oriented programming can be achieved especially with the use of factory methods. An example of a class is shown in listing **??**. On line 13 a table with the metatable `Dog` is created and returned to `myDog`.

```
1  Dog = {}
2  function Dog:new()
3      local o = {}
4      setmetatable(o, self)
5      self.__index = self
6      return o
7  end
8
9  function Dog:bark()
10     print("Bark!")
11 end
12
13 local myDog = Dog:new() -- creates instance of Dog
14 myDog:bark() -- prints out Bark!
```

Listing 15: Metatables Sample 2

While this concept is very useful, if the `new` function is ever changed by mistake, a difficult to locate bug could be created.

2.1.6 Couroutines

A coroutine is similar to a thread in that it has its own line of execution, its own stack, its own local variables, and its own instruction pointer but shares global variables and almost anything else with other coroutines. The main difference to a thread is that a program with threads runs things concurrently while a coroutine in Lua is collaborative and at any given time only 1 is running [**Ierusalimschy:2013:PLT:2502646**]. While there may not be a performance gain from using coroutines, it allows a developer to easily orchestrate multitasking that would otherwise be difficult to implement.

To create a coroutine it is as simple as calling the `coroutine.create` and passing a function as shown in listing **??**.

```lua
co = coroutine.create(function (a,b)
    coroutine.yield(a + b, a - b)
end)
print(coroutine.resume(co, 20, 10))  -- prints out true  30  10
```

Listing 16: Couroutine Sample [**Ierusalimschy:2013:PLT:2502646**]

2.1.7 Garbage Collection

Lua automatically manages memory, Lua destroys objects when all references to them are lost. There is no function to delete them manually, a program can only create objects which means that the developer will not need to bother with the majority of memory management [**Ierusalimschy:2013:PLT:2502646**]. Lua has no issues dealing with cyclic references unlike simple garbage collectors. Lua allows for manually calling the garbage collector using the `collectgarbage` function as even the best garbage collector isn't perfect, the Lua garbage collector sometimes needs some help. It is even possible to manually pause, stop and restart the garbage collector to allow for fine tuned adjustments to the use of memory in a Lua program, it is also possible to get the total amount of memory being used.

In very high performance scenarios a Lua developer may need to assume control of the garbage collector, a simple example is shown in listing **??**.

```lua
collectgarbage("stop") --stops the garbage collector
print(collectgarbage("collect")) --prints out 0
mytable={}
for i=1,10000,1 do mytable[i]=i end --fill the table with 10000 items

print(collectgarbage("count")) --prints out 285
mytable = nil -- we remove the reference to the table
print(collectgarbage("count")) --prints out 285

collectgarbage("collect") --runs the garbage collector
print(collectgarbage("count")) --prints out 28
```

Listing 17: Manual Garbage Collector Sample

Although the reference to the table is removed the table still exists in memory until the next pass of the garbage collector.

2.1.8 The Lua C API

Lua is an embedded extension library and cannot be used as a stand alone program, a program can use Lua with little input from a programmer using the provided C API. The API puts flexibility and simplicity above all else, this can sometimes make it difficult to implement for an inexperienced C developer as there is no built in type checking and no clean errors, this is down to the developer to handle [**Ierusalimschy:2013:PLT:2502646**].

A major component of the Lua C API is that new functions can be registered with the virtual machine allowing for behaviour that would be otherwise impossible to implement in pure Lua. Class like objects called userdata can also be implemented in C and used in Lua allowing for interaction between the program and Lua, in the case of a game engine, objects such as sprites could be accessed in both C and Lua and provide a developer with the ability to create high performance games in a much easier to use package than C.

An example of registering a function in the Lua vm is shown in listing **??**.

```
1
2  static int l_sin (lua_State *L)
3  {
4      double d = lua_tonumber(L, 1);  /* get argument */
5      lua_pushnumber(L, sin(d));  /* push result */
6      return 1;  /* number of results */
7  }
8  void registerSin(lua_State *L)
9  {
10     lua_pushcfunction(l, l_sin);
11     lua_setglobal(l, "mysin");
12 }
```

Listing 18: Sample function implementation in C API [**Ierusalimschy:2013:PLT:2502646**]

3 Game Engines

There are many game engines already out there covering a wide range of performance and development styles, engines such as Unreal Engine [**UE**] and Unity [**Unity**] have integrated development environments and require all development to be done through them. Other game engines, such as LÖVE [**LOVE**] can be developed in any text editor using any work flow that the developer chooses, this improves prototyping times and allows developers to get comfortable much quicker as they are using tools that they are well used to.

LÖVE [**LOVE**] provides a limited set of features and leaves the majority of the work on the internals of the engine to the game developer. This allows for very fast prototyping of small example games but when creating a larger game things can get difficult. The engine does include some things at a lower level such as physics, GLSL shaders, image loading, TCP and UDP networking. Because LÖVE doesn't manage entities there is quite a low performance ceiling especially with the addition of lighting and particles, other engines address this at the cost of development time. LÖVE is completely open source and completely free.

Unity [**Unity**] and Unreal Engine [**UE**] both manage every aspect of the game while still allowing massive extensibility. Both have unique development flows and both take a very long time to master, because of just have massive they are, development can be very slow when compared to lightweight prototyping engines. The Unity engine is not fully open source but allows source access for certain modules of the engine while the latest version of Unreal Engine is fully open source, both adopt similar payment patterns making the developer pay based on the number of sales rather than a fixed fee.

Cocos2d is an open source framework designed for creating cross platform games, apps and other GUI based programs [**COCOS2D**], it has support for scripting in both Lua and JavaScript but requires C++ code to initialize any scripts. There is an IDE created for cocos2D that provides similar workflow to the full engines while having the flexibility of text only engines such as LÖVE.

Open source allows for the collaborative development of any project and would therefore be ideal for a game engine, by allowing people to fork and extend a game engine a very wide variety of features can be implemented, enabled and disabled based on the individuals needs.