A Scriptable 2D Game Engine

Cyrus Hanlon

October 31, 2017

Contents

1 Introduction

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

From Unreal Engine [4] with its complete design versus much lighter weight engines such as LÖVE [12], 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 easily 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 Pawn Squirrel Io Javascript GameMonkey AngelScript Scheme TinyScheme Wren Lily Python

point out useful techniques and algorithms that will be employed in the solution.

How to bind Lua?

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. [14] Lua came from the languages SOL and DEL [13] 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 [8]. DEL is an acronym for Data Entry Language, an entity in DEL is essentially an object or record in other languages [13]. Lua was created as a unique combination of the two more primitive languages, it was designed to avoid strange syntax so that anyone could use it effectively and for the language to have built in type checking.

Lua is written in C and so, is very cross platform [10], it is used in projects ranging from Cheat Engine [1] running on computers to Android and iOS games using something such as Cocos2d [2]. 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 especially widely used in games It is also used in many non game settings such as in embedded systems using eLua [3] thanks to its low memory footprint.

2.1.1 Running Lua

Lua is not interpreted directly but is instead compiled into bytecode first, this is then run on the Lua virtual machine. Because of this Lua is a fast language especially with the use of LuaJIT [11]. Lua executes code by first compiling it for a virtual machine, 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 [7].

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 [5], this allows

the language to be used almost anywhere on 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 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 extra keywords such as then for if statements quite unlike C, which the language is written in. It is, however, renowned for being very easy to write with its simple syntax and extremely powerful tables.

Lua has support for all of the major flow of control statements. The most common style in similar languages is the C style using curly brackets to dictate the block and parenthesis to dictate the boolean statement however in Lua the parenthesis are not required and a then end is used instead of curly brackets, this is quite a strange syntax choice and closely resembles that of pascal with its begin and end. 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 1. 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 1, the other is the generic `for` that uses an iterator and allows for the traversal of all values, they will be discussed later. 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` [9] 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 [9] has an excellent set of examples as shown in listing 2.

```
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 [9])

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

multiple assignment local a, b = 5, 6

bitwise operators

.. concatenation

iterators

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 3: Syntactic Sugar Sample 1

In listing 3 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 4: Syntactic Sugar Sample 2

We can see how to declare a member function using some syntactic sugar in listing 4. Which is equivalent to listing 5.

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

tab:func(25)
```

Listing 5: 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 6: Syntactic Sugar Sample 4

the # operator on tables

2.1.3 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 7.

```
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 7: 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 [9].

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 8.

```
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 8: 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 starting at the first index and stopping when it hits the first nil value. [6]

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 is 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.

2.1.4 Metatables

meta tables

2.1.5 Object Oriented Programming

prototyping and whatever

Classes do not exist in Lua but object oriented programming is possible through the use of first class value functions in conjunction with tables. Multiple inheritance is possible through the use of metatables.

examples

2.1.6 Coercion

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.

examples

2.1.7 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 [6]. 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.

examples

2.1.8 Functions

Lua allows for a function to return multiple values, this is quite a strange 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 9.

```
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 9: Functions Sample 1

Functions, in Lua, are first class values with proper lexical scoping [6]. 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.

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 [6]. An example of a Lua tail call is shown in listing 10.

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

Listing 10: Functions Sample 2

Without proper tail calls recursion could overflow the stack however with proper tail calls this is impossible and fits in perfectly with a high level scripting language such as Lua.

2.1.9 Dynamic Module and Library Loading

Intro

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.

example of a simple module being created

2.1.10 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 [6]. 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.

examples of why you would manually do gc stuff

2.1.11 The Lua C API

3 Game Engines

Performance of Love2D Unreal Engine Unity Cry Engine Source (Garrysmod) Game Studio id Tech (5/6?) MonoGame Cocos OGRE

References

[1] *Cheat Engine*. http://www.cheatengine.org.

[2] *COCOS2D*. http://cocos2d-x.org/.

[3] *eLua*. http://www.eluaproject.net/.

[4] Epic Games. *Unreal Engine*. https://www.unrealengine.com. 1998.

[5] Luiz Henrique de Figueiredo. *Libraries and tools for Lua*. http://webserver2.tecgraf.puc-rio.br/~lhf/ftp/lua. 2017.

[6] Roberto Ierusalimschy. *Programming in Lua, Third Edition*. 3rd. Lua.Org, 2013. ISBN: 859037985X, 9788590379850.

[7] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. "The Implementation of Lua 5.0". In: (2005).

[8] Leslie Lamport. *Latex: A Document Preparation System*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. ISBN: 0-201-15790-X.

[9] *Lua 5.3 Manual*. https://www.lua.org/manual/5.3/manual.html. 2017.

[10] *Lua source code*. https://github.com/lua/lua.

[11] *LuaJIT*. http://luajit.org/.

[12] *LÖVE*. https://love2d.org. 2008.

[13] Roberto Ierusalimschy and Luiz Henrique de Figueiredo and Waldemar Celes. In: *IN PROCEEDINGS OF V BRAZILIAN SYMPOSIUM ON PROGRAMMING LANGUAGES*. 2001.

[14] Roberto Ierusalimschy and Luiz Henrique de Figueiredo and Waldemar Celes. "Lua – an extensible extension language". In: *Software: Practice & Experience 26*. 1996, pp. 635–652.