A Scriptable 2D Game Engine

Cyrus Hanlon

October 20, 2017

Contents

Part I

Terms Of Reference

TODO: Convert the word version to Latex

Part II

Literature review

1 Introduction

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

From Unreal Engine [2] with its complete design versus much lighter weight engines such as LÖVE [7], 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.

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

How to bind Lua?

Main languages to use C++ C# C Java objective C HTML5

Scripting languages to use https://github.com/r-lyeh/scriptorium Lua C# Pawn Squirrel Io Javascript GameMonkey AngelScript Scheme Action script TinyScheme C jit Wren Lily Python Java PHP

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. [9] Lua came from the languages SOL and DEL [8] 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 [5]. DEL is an acronym for Data Entry Language, an entity in DEL is essentially an object or record in other languages [8]. 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 widely used in games and game engines 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 [1] 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 [6]. 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 [4].

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.

if while repeat func etc

return multiple values from a function

multiple assignment local a, b = 5, 6

no bitwise operators

iterators

tables starting at 1

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 1.

```lua
--declaration of a table using numerical indices
local array = {
    "this is a string",
    105,
    function(x) print("this is a function"..x) end
}

print(array[1])             --prints out this is a string
print(array[2])             --prints out 105
array[3]("!")               --prints out this is a function!
```

Listing 1: 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. table.concat, table.insert, table.maxn, table.move, table.pack, table.remove, table.sort, table.unpack

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 2.

```lua
local tab = {
    foo = "this is a string",
    105,
    "this is another string",
    [333] = function(x) print("this is a function"..x) end
}

for k,v in pairs(tab) do
    print(k, v)
end
--[[ prints out the following:
1   105
2   this is another string
333 function: 0x1663b90
foo this is a string
]]

for k,v in ipairs(tab) do
    print(k, v)
end
--[[ prints out the following:
1   105
2   this is another string
]]
```

Listing 2: 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. [3]

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 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.

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.

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.

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 [3]. 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.

2.1.8 Sugar

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

```
--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.

```
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.

```
--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.

```
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

2.1.9 Closures

2.1.10 Proper Tail Calls

Proper tail calls to allow for recursion

2.1.11 Functional Programming

Functions are first class values

2.1.12 Dynamic Module Loading

2.1.13 Garbage Collection

2.1.14 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] *eLua*. `http://www.eluaproject.net/`.

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

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

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

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

[6] *LuaJIT*. `http://luajit.org/`.

[7] *LÖVE*. `https://love2d.org`. 2008.

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

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