Speed of Lua embedded in C++

Patrick Schuur

# Introduction

When you want users to make additional content for your software like HUD improvements in your MMORPG or functionality in your sandbox game you don’t want them to feel overwhelmed doing it. To prevent this you would make an API that the user can interact with instead of the pure underlying code.

Maybe you would even go as far as making it possible for the user to work in a wholly different (usually interpreted) programming language than the original software was made in. You could do this because the other language is easier for novice programmers to understand or because it makes quick scripting easier because the user doesn’t have to think about the things that would come into play with a compiled language. This is called language embedding.

Tooling like this is often used in big title games like World of Warcraft for addons and in Roblox for game functionality. In both these cases the scripting language Lua is used, Lua is interpreted, dynamically typed, and has automatic memory management.

Embedded languages can also be used during game development to make it easier for designers to implement functionality so programmers can focus on other tasks. In this paper I explore how the scripting language Lua compares in speed to native C++ code and whether the speed difference is significant enough to not use it in game development.

# Implementation

I implemented Lua in C++ using the Lua 5.4.4 source files. Lua is a dynamically typed language which doesn’t translate well to C++ which is statically typed. To circumvent this everything works via a stack which you can pull information off and into.

# References

***Programming in Lua (first edition). (n.d.).***[***https://www.lua.org/pil/contents.html***](https://www.lua.org/pil/contents.html)

***javidx9. (2019, March 24). Embedding Lua in C++ #1 [Video]. YouTube.*** [***https://www.youtube.com/watch?v=4l5HdmPoynw***](https://www.youtube.com/watch?v=4l5HdmPoynw)

***Standard library header (C++11) - cppreference.com. (n.d.).*** [***https://en.cppreference.com/w/cpp/header/chrono***](https://en.cppreference.com/w/cpp/header/chrono)

***Iterate through Lua Table. (n.d.). Stack Overflow.*** [***https://stackoverflow.com/questions/6137684/iterate-through-lua-table***](https://stackoverflow.com/questions/6137684/iterate-through-lua-table)

***Calling C++ Functions from Lua. (n.d.).***[***http://gamedevgeek.com/tutorials/calling-c-functions-from-lua/***](http://gamedevgeek.com/tutorials/calling-c-functions-from-lua/)

# Method

For my testing I used the chrono time library from the C++ standard template library to make a simple timer to record the time my tests took to complete. I opted to record the time in nanoseconds as many C++ tests returned a duration of 0 because it took too little time.

Text

Description automatically generated

I do the same test multiple times to get an average duration, sometimes however the first and last result can skew the results a lot, so I left them out of the equation.

Text

Description automatically generated

# Palindromes

For my first test I wanted to test the speed of finding all the palindromes up until a million. Even though finding palindromes isn’t very useful for game development it does help with show how fast Lua is at string manipulation, looping and table manipulation.

*Text

Description automatically generated*Text

Description automatically generated

*Function in Lua Function in C++*

Graphical user interface, text

Description automatically generated

*CPP Test*

Text

Description automatically generated

*Lua Test*

I decided to run the test 3 different times: the first I would run 10 times, the second I would run 100 times and the last one 1000 times.

For the graphs I rounded all the values to 3 decimal places to get seconds instead of nanoseconds.

The last test seems to have a significantly higher average for Lua than the other tests, I first thought this had to with me doing other things on my laptop during the test as the test took around 20 minutes, but this doesn’t explain the fact that C++ doesn’t have this issue.

I can however conclude from these tests that Lua is significantly worse for these kinds of tasks although I think that is fine as that isn’t really the purpose of the language.

# Objects

For my second test I wanted to use object-oriented programming in Lua as it is very common in game development. In this test I made a function that makes a set number of instances of a player object which has a number variable. In a game engine the underlying C++ code would have to have access to certain information stored in an object in Lua to use it, so I also took into account the reading of the number variable from C++.

I decided to run the test 4 different times: the first I would run 10 times, the second I would run 100 times, the third one 1000 times and the last one 10000 times. I also ran one to a million but that one encountered stack overflow issues.

Text

Description automatically generated Text

Description automatically generated

*Player class with creating function in Lua Player class in C++*

Text

Description automatically generated

*CPP Test*

Text

Description automatically generated

*Lua Test, comments to keep track of the stack*

Taking the average from the 10000 times test as that one should be the most accurate, we can see that running the Lua test takes about 6.4 times as much time as the C++ test on average.

Lua takes about .13 seconds over something that does takes C++ about 0.02 seconds to do.  
Now this could be very significant in big simulations, but I think in the context of scripting functionality for games this doesn’t matter that much as we’re talking about 10000 objects.

If we look at the 100 objects, which a more reasonable scale I would say for the use case, we get a duration of 0.000238 seconds for the C++ test and a duration of 0.0014778 seconds for the Lua test which shouldn’t matter that much.

# C++ Functions

For this last test I wanted to see how fast Lua can call C++ functions, I do this because calling underlying engine functions is crucial for making functionality. For this I decided to keep it simple and not return anything and only say how many times the function should be called. I’ll stick around 1000 times as I think most use cases will fall within that amount. I will run the test itself 100000 times to get a relatively accurate average.