Making DCS run faster, for mission designers: Lua Best Practice #1

Hi, this will be the first in what I hope to be a long series of informative posts aimed at introducing you to the concepts I've pickup over the last few years to make DCS MP missions run more like a single player experience for the clients connected and to make dedicated servers run as efficiently and to the highest performance possible. That's not to say that these concepts do not apply to single player experiences too, because they all absolutely do.

The first post will concentrate specifically on some key Lua fundamentals and should give you enough information and examples to understand how to write your Lua more efficiently.

To introduce myself quickly and give you a bit more confidence in my advice, I'm a lone community developer and the creator of the XSAF multiplayer server. My ideas have pushed outside the boundaries of what's considered possible in DCS (live in mission warehouse manipulation, asynchronous operations (my own implementation of multi-threading), custom learning AI model etc) and over 75,000 lines of Lua and C++ code. I typically have 1,000-1,200 ground units on the map (Syria), up to 120 of them moving distances of up to 120Km at any given time, around 70 aircraft total and 20-25 clients connected and with all that and the demanding data acquisition and decision making of the AI. I usually see the CPU core (corrected for hyperthreading) sitting around 70% loaded. That might not impress you much however, I'm doing that on a server with an Intel Xeon and 1666MHz DDR3 with DCS barely squeezing on a SATAIII SSD. That's not a lot of horsepower to play with.

I'm not claiming to be an expert and indeed I'm still learning every day, but what I can tell you comes from experience and not text books, so it's sound. You might be able to build on it or even teach me how to improve my understanding, I hope!

Why should you do this

I see a lot of community posts about performance in DCS and whilst there are a huge number of variables that might contribute to a lack of performance what I don't see anybody talking about is code efficiency and as mission designers we should be aware of some key principles to ensure our missions run the best they can do. If you didn't already know, DCS's mission execution (that includes all the ED mission code running in the mission's LuaState, all your code running in the mission, all ED code running in almost every other LuaState, all your hooks and user code and most of all the C++ code that is DCS… Runs concurrently. That is, it's executing each line of code one line after another, one function after another. That means that if any line, function or method is slow, the others in line have to wait until it's complete and that can cause them to fall behind creating a snowball effect that has a surprisingly early point of no return. An underperforming function will bring the house down. Even worse if the offending code runs significantly slower than the frame time (dedicated servers have that too), clients get micro-stutters, lag, rubber banding or worst-case synchronisation issues and complete unrecoverable desync. So as designers, having some knowledge of scripting best practices, and how expensive doing a thing might be ahead of time will save you the pain later of trawling through your code later to find the bad egg/s.

Lua Best Practice #1

Ok down to the specifics of this posting, starting with an easy win, or many easy wins if you can adopt this all over your code.

Locals & Loops & Constants

In any programming language global access is bad and it's no different in Lua. Global table access is much slower than local access and this is one of those key principles I mentioned earlier. It applies to functions, variables, data structures, everything. Consider this;

```
var = 200
for I = 1, n do
   local t = var * 3.142
end
```

```
local var = 200
for I = 1, n do
   local t = var * 3.142
end
```

The code on the right runs 58% faster than the left simply because we declared that variable on the global table vs local to the loop. In certain applications we might also bring that variable inside the loop or use the loops current index as a refence and it would be even faster.

It's a similar situation when we look at anonymous functions or nested functions and that's because closures (and tables) are expensive to create. If we lift them out of the loop so they're only created once we improve the execution speed of the code, and on top of localisation the gains are night and day.

```
local var = 200
for I = 1, n do
   local t = function() return var * 3.142 end
end
```

```
local var = 200
local f = function() return var * 3.142 end
for I = 1, n do
   local t = f()
end
```

The code on the right runs 71% faster than the left.

This is not to say this is a hard and fast rule, and this is certainly only a high level introduction to the concept, and just making a huge pile of variables localising everything at the top of each file also isn't the best of idea's. For example;

```
for I = 1, n do
    local t = math.sin(var)
end
```

```
local m = math.sin
for I = 1, n do
    local t = m(var)
end
```

The code on the right runs faster for sure but only 27% faster, not as dramatic improvement. This is because the math.sin function is already a declared closure not an anonymous one created on the fly. It's also a Lua library function so even though it's a global, its already pretty fast.

The concept is simple and to drill home how much it can benefit took really no time at all, but in practice it might be a little hard to get your head around and it's even harder to apply the concept to an existing codebase if you didn't already appreciate the power of it before so here's a little cheaty list of things to think about when writing or to look out for when refactoring;

- Nesting local functions inside other functions can make your code easier to read and write, but it's not the best way to make a good mission. Take the nested functions out and put them above so that they're created on script load rather than every time you call the function. Sure it's OK if the function is rarely called, but it might be called more than you think.

  ```
  local some_func()
    local r = function(i,n) return i * n end
    local calc = r(i,n)
  end
  ```

  vs

  ```
  local r = function(i,n) return i * n end
  local some_func()
    local calc = r(i,n)
  end – 96% faster execution
  ```

- Anything that doesn't have local in front of it should pass the 'I cannot do what I want to do unless this is a global' test, which is almost never. Make it local and then make a globally accessible function which returns said thing, then localise that return elsewhere.
- Pay particular attention to loops, and especially ones that either are called often or have many cycles. If you can gain performance by refactoring any global calls to locally declared access to the same function (Group.getByName to a declared one outside the loop or indeed the function all together, local getGroup = Group.getByName) then that's an easy win. But also consider these points;

    - Making a break after a certain number of iterations and returning with a schedule to finish the job, then making the data available to others with a flag or variable, do it ahead of time that a caller might need it. As long as the data is not about to change during your break.

    - I'll touch on this a bit more in another posting but, examine your logic flow and consider re-ordering the and/or gates to essentially filter better, preventing unneeded execution.
    - Or more advanced if you're capable like thinking about if what you're trying to do might be better achieved using a small class, or even a coroutine. Both of which have been hugely beneficial to me and I hope to impart that knowledge down the line.
    - In an indexed loop the '#' operator is a little expensive, so if you can make your own index count rather than constantly relying on '#' to count for you.
    - Unused local: I'll dig into this in another post but 'for key,value in pairs' is pointless when you only ever reference the key. Use 'for key in pairs' instead.

- Tables are just the same, and there a few high-level things you can be doing along a similar theme;

    - Remove tables that are nested just like functions, why do you need to create that table each time? Move it out of the execution if it never changes and if it does and you need to amend it then do it within the function or execution for sure, but don't ever nil it, or re-assign it like so 'some_table = {}' because that's just as bad as creating a closure in some cases. Clean it and re-use it.
    - Metatables. If you don't know why, then I hope to show you.
    - Localise and return with a global access function as before.
    - Avoid copies and deep copy's as much as you can. Whilst in some situations this is convenient, it's not always cheap to do.
    - Remember if you table is local so are it's contents, so consider that functions can also live inside tables and they can also be indexes the same as they can be key's.

- Constants, I've touched on the topic indirectly thus far, but if something doesn't change then there's scope for clever ways to deal with it more efficiently, and I think it's best I give you a real-world example instead of trying to explain.

Let's then apply this (and add a bit of spice to it at the same time) in a context relating directly to DCS. The below is not made up and is a real situation that I've encountered and had a positive result. I'll name these examples for easy reference back to them later, I'm hoping that this develops and grows over time to a substantial reference guide and I certainly have a number of topics in my head I'd like to write.

Situational Example #1

I have a module (Mediator) that contains any code that would otherwise be duplicated elsewhere. All of my other modules call the functions in it should they have needed a copy themselves. Because of this most of the functions here are also overloaded, meaning because they're called from multiple locations, they might need a slightly different flavour of execution or result which adds to the execution time.

The most called function is a world search function and it's called a lot, and I mean, a lot. Sometimes faster than you're blinking right now it's been called 50 odd times… Maybe.. Inside its flow there's

also another tree of logic and a loop that essentially is like calling the function a 1000 times in one go, I've already gone to town on it and there is little left to optimise. This particular function has yielded more negative attempts than positive at making it go faster, but I really could do with it returning 0.0005 seconds faster than it already does, for reasons... And until this particular moment in time one thing stood out to me, the *hasAttribute* method, but I'd not had any idea's on how to improve it. Except there was, a super simple solution that I had overlooked, a cache, and a metatable. I'm not going to dwell on those 2 practices at all because they deserve a post of their own but know this, both are powerful.

So, here's how it went, the idea is not to go to the SSE and ask the question anymore, I'm going to cache the answer to be used thereafter, because the answer never changes. It's a constant. And if you don't know what that is, it's a variable, function, anything that never throughout your codes execution, and identifying constants can be yield big performance returns if they're not being treated so (another key principle). And in this example, it's impossible for the unit to change it's attributes once it's in the world, and using it's name as an identifier (or another case where names are recycled the *unit.id_* would be the next best bet) I'm going to find a 'lightweight' way to get them and cache them. So instead of asking the SSE for the answer, I'll ask my Mediator, and this doesn't just apply to this example function this is scalable, across the whole codebase. So how?

The Cache :: *Class*

I want to create a class to handle the caching of the data, and the requests for information, but I don't really need to scale up to a massive class so, just a simple table to hold the data;

local has_attribute = {} – localised in the 'Mediator'

And rather than writing a massive class with methods and constructors, or even just some small functions, I'm going to get this table to do it all, by giving it a metatable and tell it that if it's called, be a function not a table. Like so;

```
setmetatable(has_attribute, { __call = function (self, object, attribute)
  local object_name = object:getName()
  if self[object_name] and self[object_name][attribute] ~= nil then
    return self[object_name][attribute]
  elseif not self[object_name] then
    self[object_name] = {}
  end
  self[object_name][attribute] = object:hasAttribute(attribute) or false
  return self[object_name][attribute]
end })
```

I'm not going to dive into why that's so slick because I want to cover metatables later but it essentially takes a unit object, and the attribute that needs to be evaluated and looks in its self to see if it can be answered, if so returns the answer else it is the one that asks the SSE for the answer and before giving it to the caller it saves it in the cache. To improve on this, the only time the SSE is called will be for a dynamically spawned unit that was not there at the start. That's because I also made a global access to the localised function like so;

aiMed.cache = function() return has_attribute end

This way when I first load the mission I can get the complete attribute list from everything I bring back through my persistence code and take a single hit at that point in time. So now I already know

most of the answers. Also, now any other module that call the *hasAttribute* method can now have access to the cache like so;

Local has_attribute = aiMed.cache()

And they can ask for answers like this;

has_attribute(unit_object, attribute)

That will return the answer from the cache which is localised code elsewhere, and we called that through a global once to the place it was needed in future and now, they have a 'copy'.

The Results

So, what does that get me? I'll break it down step by step in this example, so you can see. And line it up with the leanings of the post. I'll use this as the benchmark, get the unit, as for the attribute and I'll put this in a loop just like it is in the implementation of the example.

Group.getByName('Skynet3'):getUnit(1):hasAttribute('Planes')

If I use the global call to my cache, and ask for the result;

aiMed.cache()(Group.getByName('Skynet3'):getUnit(1),'Planes')

On the first try I get a similar result, even though I'm asking for a global not baked into the DCS code like the Group class, so I know this is going to be positive. I ask for it again and I get an improvement of 8%, not great but lets localise the cache first;

```
local has_attribute = aiMed.cache()
has_attribute (Group.getByName('Skynet3'):getUnit(1),'Planes')
```

Now it's 11% faster than benchmark, of course I shouldn't be asking for the unit object like that as in a real world scenario I would already have it as a result from *world.searchObjects* so;

```
local has_attribute = aiMed.cache()
local unit_object = Group.getByName('Skynet3'):getUnit(1)
has_attribute (unit_object,'Planes')
```

And there it is, 95% faster than benchmark, result, except there's a hole in that profiling example because my benchmark wasn't a real-world example either. So, to be sure let's set this as our benchmark and give the SSE a chance by letting it have the same localised advantages…

```
local unit_object = Group.getByName('Skynet3'):getUnit(1)
unit_object:hasAttribute('Planes')
```

This implementation is now 65% slower, each and every time. So, we've just gained a 65% performance boost on one method which in my case is called thousands of times every second. That's a nice result for very little coding. It took me longer to replace the 100 or so examples of the method being used across my codebase, but it was worth every minute spent doing so.

In Summary

I've tried to make this easy to read and understand but I'm conscious that I've used phrases, names, processes, mentioned methods and concepts that I've not explained and I don't expect all of the things to be understood by all. This is a high-level introduction to some core concepts that I feel are fundamental to writing good code and making a optimised mission. If anything in here needs clarifying then please don't hesitate to ask questions. My goal with this is to break the stigma that the MP experience is any less awesome than playing alone, because done in the right way, it is not. If the learnings I can impart are not well know then it uplifts us all and is only a positive for the Multiplayer community.

Keys things I hope you take away;

- Localise where possible, global access is not your friend.
- Remove anonymous functions and constants from loops or functions that are called often.
- Functions called often with large loops are low hanging fruit.
- Tables are not just data structures, and they are also expensive to create.
- Metatables are awesome.

I hope that got you thinking or even better I hope that gets you a result or two. I have a whole bunch of post's planned and honestly I actually had this massive article planned but, this has taken me months to get to finishing so it's going to be easier for me to get this out there and then build on the thread with all the other ideas I have over time. It would be great to get your feedback too before I do actually commit the time, was this valuable to you? To complex? Not complex enough? Can you improve on it? (I warn you though I'm not showing you all my cards) or feel free to suggest topics to cover.