# **Programming**

# **As a modder, at some point you might want to create advanced functionality for your mod. This functionality might extend beyond what’s offered in BeamNG software by default, and will require you to write custom programming code. Typically you’ll want to use Lua code for gameplay, or HTML/JS code for the user interface.**

# **The goal here is to provide some general guidance, explaining overall concepts and architecture choices that are specific to BeamNG.**

# **This means that:**

# **A detailed API documentation is outside the scope of this documentation.**

# **Tutorials about how to learn programming, web development, etc is outside the scope of this documentation. There’s plenty of better sources of education on the Internet, and we advise you to start there if you are not already a programmer.**

# **For any BeamNG-specific information that you cannot find in this documentation, such as detailed API use, etc, it’s best to either:**

# **a) Search the existing code and use it as a vague reference**

# **b) Ask for beamng-specific programming help at our** [**programming forums**](https://www.beamng.com/forums/programming.28/) **.**

# **Please check the index below to continue reading more programming documentation:**

##### Programming documentation feedback

# ***If you feel this programming documentation is too high level, too low level, is missing important topics, is erroneous anywhere, etc, please write a*** [***post at this thread***](https://www.beamng.com/threads/new-beamng-drive-documentation.77939/) ***and ping me personally by typing @stenyak so I will get notified.***

# [**Protecting your mods**](https://documentation.beamng.com/modding/programming/legal/)

# [**Languages**](https://documentation.beamng.com/modding/programming/languages/)

# [**Debugging**](https://documentation.beamng.com/modding/programming/debugging/)

# [**Extensions**](https://documentation.beamng.com/modding/programming/extensions/)

# [**Virtual Machines**](https://documentation.beamng.com/modding/programming/virtualmachines/)

# [**Improving framerate**](https://documentation.beamng.com/modding/programming/performance/)

# [**Lua Console**](https://documentation.beamng.com/modding/programming/console/)

# **Languages**

# **When programming mods for BeamNG, you’ll likely be using these languages:**

# **Lua: is the main programming language, and is typically used to calculate as much logic as possible. For example, gameplay logic, various vehicle behaviours, parts of the physics, etc.**

# **Javascript/Html: is used for displaying information in the User Interface. As a rule of thumb, we recommend it only for UI display purposes, leaving as much logic as possible to be calculated in Lua. The Lua side would then send the result of all calculations to the UI side, where Javascript/Html would merely display it. This typically helps make your code more robust.**

# **BeamNG Terminology**

# [**LUA Extensions**](https://documentation.beamng.com/modding/programming/extensions/) **: this is BeamNG’s module system. It features serialization, events, etc.**

# [**Virtual Machines**](https://documentation.beamng.com/modding/programming/virtualmachines/) **: BeamNG runs several independent Lua systems in parallel as well as multiple UI instances, which we call VMs.**

# [**Virtual Machines Queues**](https://documentation.beamng.com/modding/programming/virtualmachines/#communication) **: BeamNG’s main system to communicate between virtual machines.**

# [**Virtual Machines Mailboxes**](https://documentation.beamng.com/modding/programming/virtualmachines/#communication) **`: BeamNG’s secondary system to communicate between VLUA virtual machines.**

# **BeamNG Code Conventions**

#### **Style**

# **Folder naming example: ge/someFolderHere/lowerCamelCase/**

# **File naming example: lowerCamelCase.lua**

# **Indenting: 2 spaces (not tabs)**

# **Trim trailing spaces ON**

# **Function and variable names, camelCase**

# **"Class" names, upper camel case (PascalCase), example: MyCoolClass**

#### **Source code location**

# **Please check** [**source code location**](https://documentation.beamng.com/modding/programming/virtualmachines/#source-code-location) **for information about where your files should go.**

# **Lua language basics**

# **While you should learn how to program and how to program Lua on your own, here’s the most absolute basics of the language:**

#### **Basic Terminology**

# **Table: The basic and only type of container used in Lua. Depending on how you use a table, the table will resemble an array (a list) or a dictionary (a map).**

# **Array: A Lua table that contains only integer keys from 1 to infinity. Can be counted correctly with #tbl**

# **Dictionary (dict): A lua table that contains all kinds of keys. Cannot be counted with #tbl.**

# **Module: The traditional Lua module is deprecated. Do not use it**

# **Package: the normal** [**Lua package**](https://www.lua.org/manual/5.3/manual.html#6.3) **. Used, but please use Extensions if possible. See** [***here***](http://lua-users.org/wiki/ModulesTutorial) **and also** [***here***](http://lua-users.org/wiki/TheEssenceOfLoadingCode) **.**

#### **Syntactic Sugar**

# **For convenience, the Lua language sometimes offers different ways to identically behaving code. For example:**

# **myTable.ident is equivalent to myTable['ident']**

# **myObject:name(args) is syntactic sugar for myObject.name(myObject, args)**

# **myFunction{fields} is syntactic sugar for myFunction({fields})**

# **myFunction'string' (or myFunction"string" or myFunction[[string]]) is syntactic sugar for myFunction('string')**

# **function t.a.b.c:foobar(params) body end is syntactic sugar for t.a.b.c.foobar = function(self, params) body end**

##### Programming documentation feedback

# ***If you feel this programming documentation is too high level, too low level, is missing important topics, is erroneous anywhere, etc, please write a*** [***post at this thread***](https://www.beamng.com/threads/new-beamng-drive-documentation.77939/) ***and ping me personally by typing @stenyak so I will get notified.***

# **Debugging**

# **There are several ways to debug the code you are writing.**

### **Debugging Lua code via Console:**

# **Similar to most languages, this is the most basic way you can debug: a) add a print to console, b) reload your code c) observe the results in console.**

#### **a) Add a log**

# **To print to console, you can use:**

# **log("E", "context", "My message: "..dumps(myContext)): this will generate a log message with a severity level of Error (use E for error, W for warning or I for info).**

# ***Note: you can leave an empty context if that’s not useful for your purposes, e.g. log("I", "", "My message")***

# **dump("My message", myContext): this generates a log with the highest severity level. It converts all passed object to string with calls to dumps().**

# **print("My message: "..dumps(myContext)): this will generate a log message with the highest severity level**

# **Note: as a general rule, all log messages you intend to publish with your mod should have at least one variable to provide appropriate context. If there is no variable, and the log is simply a hardcoded string, there’s a high chance the log alone won’t be useful enough when you need to solve issues in the future.**

### **b) Reload your code**

# **To reload the Lua VM, see** [**virtual machines reloading**](https://documentation.beamng.com/modding/programming/virtualmachines/#reloading) **.**

# **In some rare cases you may need to completely shut down the program and launch it again from scratch. This can happen if you’ve previously tried buggy code that has left the entire reloading system in a failing, unrecoverable state. Our reloading system tries as best as it can to survive possible programming errors - but it can only do so much.**

### **c) Check the console**

# 

# **To open the Console press the ~ (*tilde*) key on keyboards with US layout.**

# ***Note: For different keyboard layouts, verify which key is assigned to the Toggle System Console action in the Options > Controls > Bindings menu, under the General Debug section***

### **Debugging Lua code using a debugger:**

# **(TODO)**

##### Programming documentation feedback

# ***If you feel this programming documentation is too high level, too low level, is missing important topics, is erroneous anywhere, etc, please write a*** [***post at this thread***](https://www.beamng.com/threads/new-beamng-drive-documentation.77939/) ***and ping me personally by typing @stenyak so I will get notified.***

# **Extensions**

### **Description**

# **Lua BeamNG extensions are an easy and scalable way to extend functionality. BeamNG extensions allow to mesh multiple pieces of code together, without needing to modify any official files (which would likely create issues after each official update, and would make your mod inadmissible in the** [**official BeamNG mod repository**](https://www.beamng.com/resources) **).**

# **For example: you could write a mod that reacts when the slow-motion system is used by the player, without needing to edit the official slow-motion source files to insert explicit calls to your mod.**

# **This is achieved using the “hook” functionality, where extensions can subscribe to various events, and can trigger events themselves too.**

### **Show me the code**

# **A BeamNG Lua extension is essentially a Lua file that returns a table. For example:**

# ***local* M = {}**

# **M.myData = 10**

# **M.myFunction = function() print("hello world") end**

# **return M**

# 

# **Once your mod is packed into a zip file, the extension files should land somewhere in:**

# **(someMod.zip) /lua/common/extensions/**

# **(someMod.zip) /lua/ge/extensions/**

# **(someMod.zip) /lua/vehicle/extensions/**

# 

# **Your choice will dictate in which** [**Lua virtual machine**](https://documentation.beamng.com/modding/programming/virtualmachines/) **the BeamNG extension can be used. See** [**source code location**](https://documentation.beamng.com/modding/programming/virtualmachines/#source-code-location) **for additional context.**

### **Using an extension**

# **An extension you create won’t be loaded until you write explicit code for it to be loaded. For example:**

# **extensions.load("myMod\_myExtension") *-- this loads extensions/myMod/myExtension.lua***

# 

# **Once a BeamNG extension has been loaded, you can access its table:**

# **myMod\_myExtension.myFunction()**

# **myMod\_myExtension.myData = 20**

# 

# **You can also run a function in all the extensions that are currently loaded in this Lua virtual machine:**

# **extensions.hook("myCustomEvent") *-- this will call myMod\_myExtension.myCustomEvent(), and it will myCustomEvent() in all other loaded extensions***

# 

# **When you no longer require an extension, you must unload it:**

# **extensions.unload(...)**

# 

# **Keeping extensions loaded when they aren’t necessary will waste computer resources and lower the framerate, so avoid it.**

# **Note 1: There’s a syntax in the form of extensions.myMod\_myExtension.foo(). Avoid this syntax, because a) it will lower frame rates compared to myMod\_myExtension.foo(), and… b) it automatically loads the extension if it was not loaded. As a general rule, you shouldn’t need to automatically load an extension, you should do it explicitly.**

# **Note 2: Your extension may explicitly define which other extensions it depends on. This is done with:**

# **M.dependencies = { "foo\_bar", "baz\_qux", ...}**

# 

# **This will tell the BeamNG extension system to load/unload these dependencies for you. It will also organize the hook function calls, so they follow the tree of dependencies in the appropriate order. In other words, a hook will run in the dependencies first, then in your own extension, and so on.**

### **Common extension functions/data**

# **In addition to your own really specific “myFunction”, “myData” members, your table can contain more common members.**

# **For example:**

# **M.onExtensionLoaded = function() ... end *-- called when the extension is loaded***

# **M.onUpdate = function() ... end *-- called once per GFX frame***

# **M.onGuiUpdate = function() ... end *-- called once per UI frame***

# **M.state = { foo=1, bar=2, ... } *-- may be used to save+load extension state during a reload of the Lua virtual machine***

# **M.onSerialize = function() ... end *-- may be used to save+load extension state during a reload of the Lua virtual machine using a custom serializing function***

# **M.dependencies = { "core\_camera", ... } *-- may be used to automatically load other extensions***

# **etc**

# 

# **There’s no centralized list with those special functions and variables; the source code IS the documentation(tm). For example, you may research the code inside lua/ge/main.lua, lua/common/extensions.lua, etc. See who’s calling extension hooks, or observe how other extensions have been written, etc.**

##### Programming documentation feedback

# ***If you feel this programming documentation is too high level, too low level, is missing important topics, is erroneous anywhere, etc, please write a*** [***post at this thread***](https://www.beamng.com/threads/new-beamng-drive-documentation.77939/) ***and ping me personally by typing @stenyak so I will get notified.***

# **Virtual Machines**

# **Introduction**

# **A virtual machine is each of the environments in which code can be executed. There’s 3 types of virtual machines:**

# **Vehicle Lua VM (VLUA): runs code specific to a vehicle (such as turbo boost calculations or AI steering inputs).**

# **GameEngine Lua VM (GELUA): runs general code that’s not specific to a vehicle (such as career mode, traffic vehicles orchestration or replays).**

# **User Interface VM (UI): runs javascript/html to generate a visual interface (such as the main menu).**

# **The GELUA and Main Menu UI VMs always exist; however the rest of VMs depend on which vehicles you’re using (and in some cases even maps, or mods in general).**

#### **Example**

# **Let’s imagine you have spawned a modern taxi and a ’90s hatchback. You might have the following VMs:**

# **General VMs:**

# **GELUA**

# **Main Menu UI**

# **Taxi VMs:**

# **VLUA**

# **Dashboard UI**

# **GPS navigator UI**

# **Infotainment UI**

# **Hatchback VMs:**

# **VLUA**

# **That would be a total of 7 different VMs running in parallel.**

### **Source code location**

# **Folder structure:**

# **/ui/: UI VM code (javascript and html).**

# **/lua/vehicle/: VLUA VM code.**

# **/lua/ge/: GELUA VM code.**

# **/lua/common/: libraries that can be used by both VLUA and GELUA.**

# **The libraries found in /lua/common/ are a convenient way to avoid code duplication, however they are not a way to share or send information between different VMs. Please check** [**communication**](https://documentation.beamng.com/modding/programming/virtualmachines/#communication) **if you want to share information.**

### **Threading**

# **Each VM runs in its own isolated environment:**

# **UI vms: they run in a separate process each.**

# **VLUA vms: they run in a separate thread each.**

# **GELUA VM: it runs in the main thread.**

### **Update rate**

# **Each VM can update at a different rate, depending on its purpose. As a general rule:**

# **VLUA: can run at physics frequency (2000Hz, see lua/vehicle/main.lua::onPhysicsStep()), or graphics frequency (see lua/vehicle/main.lua::onGraphicsStep()).**

# **GELUA: can run at graphics frequency (the current “framerate”, see lua/ge/main.lua::update()), or UI frequency (see lua/ge/main.lua::onGuiUpdate event).**

# **Main UI: runs at a variable rate. Ideally 60 FPS, but often 30 FPS or less. This depends on the available computer resources.**

# **Secondary UIs: run at a variable rate, often lower than the Main UI update rate.**

# **On top of that, you could conceivably implement your own fixed update rate. One way you can do this, is by hooking your code to the graphics update() call, and then manually skipping ticks based on basic time tracking. This way your code will run no more often than the graphics rate, typically at the fixed rate you have decided. However, we recommend against using fixed-rate logic due to performance reasons. Follow the link below for more information.**

# ***Note: we understand that writing math that can work under extreme rate variations is hard. For this reason, the reported graphics “update rate” is guaranteed to be a minimum of 20Hz. When the computer is unable to reach 20fps, then the simulation will slow down as needed. This is a guarantee that helps you to program your math with a safe baseline rate.***

# **Picking the correct update rate for your code is essential to ensure your mod can work in a variety of computers. See** [Improving Framerate](https://documentation.beamng.com/modding/programming/performance/#update-rate) for more information.

### **Reloading**

# **During development of your mod, you may want to modify the source code that runs in a particular VM. To improve iteration times, BeamNG allows you to reload the VM of your choice on the fly, in order to test your new changes immediately without restarting the program:**

# **Main UI: press F5.**

# **GELUA: press Ctrl-L.**

# **VLUA: press Tab to the desired vehicle, then press Ctrl-R.**

# **VLUA (all of them): press Ctrl-Shift-R.**

# **These reloads will start with a clean slate, spinning up an entirely new VM from scratch and delete the old one; however BeamNG has a convenient feature: it will attempt to recover/resume where you left, by storing the old state, and loading it when the new VM is running.**

# **The idea is that you can continue development exactly where you were previously rather than, say, having to load your Career saved game again each time you edit some code.**

### **Communication**

# **Every virtual machine is running in parallel to the rest, and fully isolated from them all.**

# **If you want a piece of information from one VM to be available in other VM, you need to explicitly send the information from one VM to another VM.**

# **We offer these asynchronous communication methods:**

# **Asynchronous queues in VLUA and GELUA: you enqueue a string of source code in the target LUA VM. This code contains the information you want. For example, you might enqueue the string "input.event('throttle', 0.6)" into a VLUA VM.**

# **Asynchronous events in UI: you enqueue an event that javascript will be notified about, and various listeners can react to it. For example, you might enqueue the event "MenuHide".**

# **Asynchronous mailboxes in VLUA: VLUA vms can also read data from a virtual mailbox storage system: the sender writes data to the mailbox, and the recipient(s) can check what’s the last information available in that mailbox. This is a more performant way to communicate than queues, but you cannot run arbitrary code: instead, the recipient VM must manually check their mailbox in order to notice the information has been updated, and act upon it however they see fit.**

# ***Note: if you were wondering about traditional communication methods (such as mutex, barrier…), those are intentionally not made available, for robustness and simplicity.***

# **Here’s a quick visual guide with the code needed to communicate between APIs:**

# 

# **And here’s the same information explained with plausible code examples:**

# **Sending events to UI:**

# **guihooks.trigger("myEvent") (from VLUA)**

# **guihooks.trigger("myEvent") (from GELUA)**

# **$scope.$emit("myEvent") (from UI)**

# **Listening for events in a UI:**

# **$scope.$on("myEvent", function() { myJavascriptCode() } )**

# **Enqueing code to GELUA:**

# **bngApi.engineLua('myLuaCode()') (from UI)**

# **obj:queueGameEngineLua('myLuaCode()') (from VLUA)**

# **Enqueuing code to one particular VLUA:**

# **bngApi.engineLua("be:getPlayerVehicle(0):queueLuaCommand('myLuaCode()')") (from UI)**

# **be:getPlayerVehicle(0):queueLuaCommand('myLuaCode()') (from GELUA)**

# **Enqueuing code to all VLUA:**

# **bngApi.engineLua("be:queueAllObjectLua('myLuaCode()')") (from UI)**

# **be:queueAllObjectLua('myLuaCode()') (from GELUA)**

# **BeamEngine:queueAllObjectLua('myLuaCode()') (from VLUA)**

# **Enqueuing code to all VLUA except one:**

# **BeamEngine:queueAllObjectLuaExcept('myLuaCode()', exceptObjectID) (from VLUA)**

# **Leaving information in a VLUA mailbox:**

# **be:sendToMailbox("myMailboxAddress", myData) (from GELUA)**

# **Reading information left in a VLUA mailbox:**

# **local myData = obj:getLastMailbox("myMailboxAddress")**

# **Note: reading data from a mailbox does not delete the information. It’s still available to be read in the future and by any other VLUA VM.**

# **We recommend you look at existing official code to get a better idea of the possibilities and to see working examples instead of the pseudoc-code listed above.**

### **Performance considerations**

# **Please refer to** [**Improving Framerate**](https://documentation.beamng.com/modding/programming/performance/#avoid-communication-between-virtual-machines) **for details on how to reduce the negative impact that VM communications can have.**

# **Full architecture**

# **The sections above cover all the basics you need to know when programming your mods.**

# **However, if you want additional detail, there are additional components that link all these VMs together:**

# **C++ Game Engine: the “real” high-performance engine, written mostly in C++, and implementign features such as frame rendering, input handling, VR, audio generation, etc.**

# **C++ Physics Core: while VLUA VMs compute accessory parts of the physics that relate to vehicles (such as powertrain, etc), there’s also the core physics engine. This computes the soft-body physics, friction models, node-grabbing physics, etc. It also orchestrates the execution and synchronization of all VLUA VMs.**

# **TS VM: this component has not been completely removed yet, so remnants of the old TorqueScript VM still exist in our engine. It runs on the main thread and shouldn’t be used for any new development or mods.**

# **This is a detailed overview of all VMs, all conceptual sections of the engine, and which threads/processes each runs in:**

# 

# **As you can see, in addition to the multiple threads from spawned vehicles and the different UI vms, the C++ Game Engine itself is internally multithreaded too (for tasks such as Vulkan rendering, audio tasks, etc). However, for modding purposes, you can ignore them all, and focus only on UI/VLUA/GELUA VMs.**

##### Programming documentation feedback

# ***If you feel this programming documentation is too high level, too low level, is missing important topics, is erroneous anywhere, etc, please write a*** [***post at this thread***](https://www.beamng.com/threads/new-beamng-drive-documentation.77939/) ***and ping me personally by typing @stenyak so I will get notified.***

# **Improving framerate**

# **As a programmer, you should keep in mind the performance implications of your code.**

# **This is not just about optimizing code, but also about consciously choosing code architectures that favour performance.**

# **Important: it doesn’t matter if things work “*fine*” in your computer. Your code could end up running in wildly different computers, where the performance patterns may not match those of your computer. Even if you develop using a slow computer, its slowness is not necessarily a good reference that you can use: a different combination of mods, a different combination of hardware, of drivers, OS updates, etc can all lead to a worse performance than what your low-spec computer exhibits.**

# **Basic optimization tips**

# **This document assumes you already have basic knowledge of optimization. Some examples:**

# **Knowing how optimization is done in general: measure, then tweak, then measure again to evaluate the validity of your tweaks.**

# **Never assuming you know how your code will behave. Always measure. Measure as many times as needed to ensure validity of the numbers. Dedicate explicit effort to determine if your measurements are reliable or if, on the contrary, they are incorrectly biased by external factors (hot vs cold caches, external programs, thermal throttling, etc).**

# **Knowing basic optimization techniques: what big O notation means and how its used. What are cache systems and knowing their possible tradeoffs (such as memory vs cpu use). Taking advantage of data locality to use the available RAM bandwidth more efficiently. Knowing when the bottleneck is I/O (such as storage or networks) and how to best work around those limitations. Etc.**

# **If you don’t already have basic optimization knowledge, it’s advisable that you first dedicate some time to learning about it. The rest of this document is a collection of some of optimization techniques that are specific to BeamNG software, or which are harder to find information about elsewhere.**

# **IMPORTANT: Do not blindly assume you know which part of code is the main bottleneck, always measure to identify where your optimization efforts can be the most efficient.**

# ***For example, there’s no point shaving 0.25 milliseconds in a function, if you are not planning to optimize the next function which is bleeding 3 milliseconds per frame with an avoidable O(N^2) loop.***

# **LuaJIT optimization: BeamNG tips**

# **As part of the BeamNG Lua ecosystem, we use a few tools to do measurements. As a mod programmer, these will be helpful for you too:**

# **timeprobe() function: measures the time between 2 consecutive runs.**

# **gcprobe() function: measures the increase in GC workload between 2 consecutive runs (see** [**Avoid garbage collection**](https://documentation.beamng.com/modding/programming/performance/#avoid-garbage-collection) **section below).**

# **lua/common/luaProfiler.lua class: allows you to split your code into multiple sections, including sections inside repeated executions (such as loops), and measure both GC load and time on each.**

# **lua/common/luaProfiler.lua class: also allows to detect performance spikes (stutter) and show the GC load and time measurements that led to it.**

# **getPlayerVehicle(), getAllVehicles(), vehiclesIterator(), activeVehiclesIterator() functions in** [**GELUA side**](https://documentation.beamng.com/modding/programming/virtualmachines/#introduction) **: retrieves vehicle objects with zero GC overhead. Prefer this over be:getPlayerVehicle() and similar calls, which will reduce the framerate (since they need to retrieve data all the way into C++ territory and then back to LUA, and in addition they also increase the GC workload).**

# **LuaJIT optimization: Generic tips**

# **In addition to those BeamNG-specific tips, there’s also generic Lua and LuaJIT optimizations you should try to follow.**

#### **LuaJIT: Loops**

# **As a general rule, for i,n loops will be faster than ipairs() loops, which in turn are faster than pairs() loops.**

# **Always pick the faster loop type if you have no good reason to pick a slower variant. If you can trivially (re)design your code to work with arrays, instead of arbitrary key-value tables, then that will allow the use of ipairs(), which will be faster than pairs(), all else being equal.**

# **As usual, if you are unsure or don’t have a lot of practice doing optimizations, you probably want to verify by measuring the improvements instead of assuming your changes are okay.**

#### **LuaJIT: Local symbols**

# **The location where a function or variable is defined has a performance impact.**

# **Accessing a variable that’s nested deep inside some table structure will be slow, for example:**

# ***-- very slow access, AVOID THIS:***

# **foo(myTable[4]["foobar"][myIndex][50])**

# **bar(myTable[4]["foobar"][myIndex][50])**

# **baz(myTable[4]["foobar"][myIndex][50])**

# 

# ***-- faster access with a cache, DO THIS:***

# ***local* myVar = myTable[4]["foobar"][myIndex][50]**

# **foo(myVar)**

# **bar(myVar)**

# **baz(myVar)**

# 

# **In the same way, the scope of a variable can have the same performance effects. This makes sense once you know that a file-local symbol is stored in a file-specific Lua table. And that a global symbol is contained in a global Lua table of variables.**

# **Whenever you use some symbol, the LuaJIT interpreter will begin by checking that local table, then the parent table, until reaching the global table of symbols. Each of those table accesses cost performance. So from the point of view of pure performance, local variables are preferable to global variables.**

# **For example: this is why the** [**BeamNG LUA Extensions**](https://documentation.beamng.com/modding/programming/extensions/) **system will make your extension available as myMod\_myExtension, rather than as myMod.myExtension (saving one or more table accesses). It’s also why you’ll find local max = math.max in several official BeamNG files, as it saves one table access. Etc.**

# **The gains may not be huge, and any impact in code maintainability/readability is always something to consider. Sometimes it’s better to have readable code than a slightly faster code. In other cases, such as very commonly used libraries or functions, and which rarely are modified, performance will probably take a front seat, sacrificing code maintainability in the name of framerate.**

#### **LuaJIT: References**

# **Here’s an assorted list of links with information about how the LuaJIT interpreter works, as well as numerous optimization tips:**

# [**https://raw.githubusercontent.com/MethodicalAcceleratorDesign/MADdocs/master/luajit/luajit-doc.pdf**](https://raw.githubusercontent.com/MethodicalAcceleratorDesign/MADdocs/master/luajit/luajit-doc.pdf)

# [**https://0xbigshaq.github.io/2022/08/22/lua-jit-intro/**](https://0xbigshaq.github.io/2022/08/22/lua-jit-intro/)

# [**https://nickcano.com/hooking-luajit/**](https://nickcano.com/hooking-luajit/)

# [**https://mrale.ph/talks/vmss16/#/**](https://mrale.ph/talks/vmss16/#/)

# [**https://gitspartv.github.io/LuaJIT-Benchmarks/**](https://gitspartv.github.io/LuaJIT-Benchmarks/)

# [**https://percona.community/blog/2020/04/29/the-anatomy-of-luajit-tables-and-whats-special-about-them/**](https://percona.community/blog/2020/04/29/the-anatomy-of-luajit-tables-and-whats-special-about-them/)

# [**https://kipp.ly/jits-impls/**](https://kipp.ly/jits-impls/)

# [**https://www.freelists.org/post/luajit/How-to-call-functions-from-a-static-library-in-Luajit,13**](https://www.freelists.org/post/luajit/How-to-call-functions-from-a-static-library-in-Luajit,13)

# [**https://www.freelists.org/post/luajit/Few-questions-about-LuaJIT-internals-on-64-bit,6**](https://www.freelists.org/post/luajit/Few-questions-about-LuaJIT-internals-on-64-bit,6)

# [**http://brrt-to-the-future.blogspot.com/2019/03/reverse-linear-scan-allocation-is.html**](http://brrt-to-the-future.blogspot.com/2019/03/reverse-linear-scan-allocation-is.html)

# [**https://piotrduperas.com/posts/nan-boxing**](https://piotrduperas.com/posts/nan-boxing)

# [**https://pwparchive.wordpress.com/2012/10/16/peeking-inside-luajit/**](https://pwparchive.wordpress.com/2012/10/16/peeking-inside-luajit/)

# **Load-time vs run-time performance**

# **When following various optimization techniques, you may find yourself having to choose between making the mod faster to start up, versus making the framerate higher once the mod has loaded.**

# **You should use common sense when choosing this balance. Normally the choice is to move complexity to the loading times, if you can then get better performance afterwards.**

# ***For example, a high GC load (see*** [***Avoid garbage collection***](https://documentation.beamng.com/modding/programming/performance/#avoid-garbage-collection) ***section below) during startup is acceptable if you can later manage to have zero GC load while the simulator is running. On the other hand, if certain optimization makes the loading time 5min longer while gaining only a 0.5% of framerate in exchange, that might not be a worthwhile tradeoff.***

# **Update rate**

# **When writing your code, you’ll need to think about how often your code will run. Should it run once per graphics frame? Maybe once per physics tick? Once per User Interface refresh? Maybe a fixed rate of 15Hz? Etc.**

# **Note: To learn more about the fundamental update rates available, please check the** [Virtual Machine’s Update Rate](https://documentation.beamng.com/modding/programming/virtualmachines/#update-rate) section.

# While picking a high update rate for your code is easy from a developer perspective, this will not only have a negative impact on frame rates, but it will also lead to greater chances of stutter and of unstable frame rates.

# The rule of thumb here is to pick the lowest possible rate that you can get away with, while still making sense for your particular application.

# For most “gameplay” purposes (such as keeping track of a score, or other similar high level concepts), following the User Interface update rate is probably enough. Or alternatively, following the graphics update rate.

# Only in extremely rare cases will you need to resort to a physics update rate. Any code that is run at physics update rate will need to be written extremely carefully to avoid a heavy impact in framerate for people running a computer with the minimum hardware specs. You’ll need to apply all the knowledge included in this document, and more. You’ll also want to include only the absolute minimum code in the physics update, moving everything that’s non-essential to the graphics update.

# You may notice that our official code only uses physics rates as our very last resort, when nothing else can possibly work from the point of view of mathematics.

# You will also notice that, unlike what many game development guides advice, at BeamNG we avoid fixed rate calculations as much as possible. We understand that fixed rate updates can make your life easier as a programmer: it’s easier to write stable math for a stable rate, than to write stable math for a constantly variable rate. However, the downside is that a fixed rate workload will not scale up nor down according to the available computer resources. A fixed rate means that you’ll need to settle with a suboptimal compromise, where low-end hardware will suffer an unnecessarily high computing cost, while the high-end hardware will be unnecessarily missing the extra detail that it could be calculating. With that in mind, running on a variable rate (such as graphics framerate or user-interface rate) means that you can provide higher fidelity in high-end computers, while also being friendly to low-end computers.

# ***Note: Writing math that can work under extreme rate variations is hard: for this reason, the reported graphics “update rate” is guaranteed to be a minimum of 20 Hz. When the computer is unable to reach 20 FPS, then the simulation will slow down as needed. This is a guarantee that helps you to program your math with a safe baseline rate.***

# **A useful tool you should use when writing code that works on a variable update rate, is the Options > Display > Limit framerate slider. You can set it to 20 FPS to test your math under the conditions of a worst-case-scenario (20 Hz updates, if your code is hooked to graphics updates), and you can disable this limiter together with Options > Graphics > Lowest to try to reach as high a framerate as possible. A good place to achieve a high framerate is the Grid, Small, Pure level while using no traffic vehicles.**

# **Avoid garbage collection**

# **If you are a programmer of a language featuring garbage collection (such as Lua or Javascript), and you are not familiar with what a garbage collector (GC) is, then please search the internet for information and learn the basics about them before continuing.**

# **GC is a convenient feature that some high-level languages provide, but they can have a large impact when used in a performance-intensive environment, such as real time simulators. The GC will hide new/delete from you, but in exchange it will take a toll in two ways:**

# **Lower framerate: the GC has to run to do its work, and this garbage bookkeeping workload is going to rob some framerate.**

# **Variable framerate: the GC workload may not be evenly spread over time, but might be bunched up cyclically. This can lead to both stutter (negative spikes in framerate), as well as rubber banding (the framerate being high for a second, low for another second, then high again, etc) which will lead to an undesirable effect of slowmotion/fastmotion.**

# **To reduce the GC load, first you need to be able to measure it. In the case of Lua, we offer two features for this:**

# **gcprobe() function: run it before/after a piece of code, and it will tell how many bytes of garbage that code has generated.**

# **ctrl-shift-f > Tools menu > Log gelua profile: this will log how many bytes of garbage each GELUA extension has generated (use ~ to see the logs) during the last graphics frame.**

# **Once you know how much GC load your code is generating, you need to find ways to reduce it. As a general rule, this means avoiding the creation (the allocation) of new objects.**

# **Try to re-use objects across multiple consecutive calls to your extension hooks. For example, you might want to have a parent-scope Lua variable that gets reused, rather than generating a new object from scratch on each function call.**

# **Use APIs that reduce the GC load. For example, you can reassign myVector = vec3(5,4,2) with the zero-garbage alternative myVector:set(5,4,2). Same goes for favouring setAdd and similar APIs that we offer with this exact purpose of GC reduction.**

# **Use APIs that fully eliminate the GC load. For example, favour using X,Y,Z tuples (such as our functions that end in ....XYZ()) instead of vec3.**

# **Etc.**

# **There’s no fixed rule about this, and many optimizations are a tradeof between performance versus code maintainability/readability. Sometimes you may want to sacrifice short-term performance in favour of making the code easier to work with (which might in turn enable higher-level optimizations in the longer-term, thanks to the code being more understandable).**

# **Avoid trigonometry**

# **Working in terms of angles typically leads to using sin(), cos(), tan() and all variants of such functions. These functions can be really slow, and should be avoided when possible.**

# **Instead, consider the use of dot product, cross product and other basic vector operations. These simpler math tools can often simplify your code, completely eliminating the need for explicit use of “angles”.**

# **It’s relatively common for programmers to be very familiar with angles, but unfamiliar with dot/cross products. So the appeal of traditional trigonometry is understandable, but it doesn’t mean it’s the best approach from a performance coding perspective.**

# **Trigonometry functions are typically used to transform geometric concepts into angles, so the programer can then operate in angles; only to eventually transforming it all back to vectors or quaternions. Therefore, if you learn to work with vectors directly, you can skip those unnecessary back-and-forth conversions. Which typically leads to simpler code, and faster too.**

# **Avoid euler angles**

# **Very often, euler angles are used as an intermediate format, before eventually being converted back into quaternions or matrices (for consumption by the core engine). It’s advisable to avoid euler angles completely, and use quaternions or matrices.**

# **Doing so means you can skip those back-and-forth conversions of rotations into (and then out of) euler format. This simplifies your code, and as a bonus makes it faster too.**

# **In addition to the performance cost of such temporary conversions, they can also lead to bugs (such as losing numerical precision due to unnecessary operations), and lead to less maintainable/readable code (for example, there’s many variants of euler angles, and you might not be sure which exact euler format is accepted by each function).**

# **The only exception where Euler angles might be acceptable, is for display to end-users, for example in a level editor UI, or similar content-creation tools:**

# **If you need to show angles to an artist/modder in the UI, always operate with quaternions, and convert to euler only at the very end of your data pipeline, at the exact moment you need to render values on the screen.**

# **If you can, show the euler values as a read-only values, not an editable text field. Consider offering an interactive 3D gizmo to apply rotations with the mouse/keyboard (and which will not be using euler internally, but quaternions), rather than offering a text field with 3 numbers that the user can type into.**

# **If you absolutely, truly need to show a read-write text field with Euler angles on your editing tool UI, review your code under the scenario of multiple consecutive saves: do the Euler angles slowly drift away from the original value without the user editing the value? If so, you need to review your code pipeline to find out the source of numerical drift, and find a solution for it.**

# **Avoid communication between virtual machines**

# [**Sending data between virtual machines**](https://documentation.beamng.com/modding/programming/virtualmachines/#communication) **can negatively affect performance, particularly in the form of latency, lower framerate, and unnecessarily varying framerate. The combination of which is typically called “lag” by end-users nowadays (even if it’s not limited to latency).**

# **Some advice to preserve performance as much as possible:**

# **Avoid communication between VMs altogether if you can.**

# **Use** [VLUA mailboxes](https://documentation.beamng.com/modding/programming/virtualmachines/#communication) if that fits your requirements.

# Reduce the frequency of communications to the minimum. Sending data each gfx frame is really bad, consider sending pre-computed data once per-event, or once-per-minute, etc if that’s possible.

#### **Example of VM communication optimization**

# **Let’s assume you are writing a mod that rates how good your burnout is. This will analyze the wheelspinning patterns, and show an amount of points in the UI.**

# **In your initial implementation, you are sending the wheelspin information from VLUA to GELUA each graphics frame using obj:queueGameEngineLua(). Once in GELUA, all the data is analyzed, and you generate a numeric rating value. You then send this rating to the UI on each frame using guihook.trigger().**

# **While this approach works, it’s pretty bad in terms of performance, and there’s plenty of margin for improvement:**

# **First of all, don’t compute the burnout rating in GELUA. Instead calculate it in VLUA, where the data already exists. This avoids sending all that physics information from VLUA to GELUA.**

# **Then, consider if you want to show a burnout only at the end-screen:**

# **Only at end-screen? Then you only need to communicate between virtual machines a single time, at the end of the burnout.**

# **As a real-time indicator in the UI?**

# **Then you may want to only update this indicator (to communicate between VMs) if it has changed since the last time.**

# **And depending on the visual indicator, you will want to optimize further:**

# **Is it progress bar? Then on top of that, you may want to send it only on UI update frames onGuiUpdate() rather than on graphics frames onUpdate().**

# **Is it a text label showing a numeric value? Then on top of that, you may want to only update this value once per second (so the user can actually read the text before it changes again)**

##### Programming documentation feedback

# ***If you feel this programming documentation is too high level, too low level, is missing important topics, is erroneous anywhere, etc, please write a*** [***post at this thread***](https://www.beamng.com/threads/new-beamng-drive-documentation.77939/) ***and ping me personally by typing @stenyak so I will get notified.***

# **Lua Console**

# 

# **You can run Bin64/console.x64.exe as nice little (interactive) lua console.**

# **Command line arguments**

# **console.x64.exe cmd <lua string to execute>**

# **console.x64.exe <lua file to execute>**

# 

# **For example to execute a lua string:**

# **console.x64.exe cmd "print(1+1)"**

# 

# **For example to execute a lua file:**

# **console.x64.exe test.lua**

# 

# **For an interactive shell start without arguments:**

# **console.x64.exe**

# 

# **Below is a short documentation of its very simplistic API**

# **BeamNG console Lua API Documentation**

## **Classes**

### **HighPerfTimer**

# **High-performance timer for accurate time measurements.**

# ***local* timer = HighPerfTimer()**

# 

#### **Methods**

# **reset() - Resets the timer to zero**

# **elapsed() - Returns elapsed time in seconds since creation or last reset**

# **stopAndReset() - Returns elapsed time and resets timer**

# **stop() - Alias for elapsed() (deprecated)**

### **BeamEngine**

# **Main engine interface.**

#### **Methods**

# **update(dtReal, dtSim) - Updates the engine simulation**

# **spawnObject2(...) - Spawns an object in the simulation**

# **deleteAllObjects() - Removes all objects from the simulation**

# **setDynamicCollisionEnabled(enabled) - Enables/disables dynamic collision**

# **instabilityDetected() - Checks if instability was detected in simulation**

### **BeamObject**

# **Represents a physical object in the simulation.**

#### **Methods**

# **getNodeCount() - Returns the number of nodes in the object**

# **getBeamCount() - Returns the number of beams in the object**

### **VFS**

# **Virtual file system interface.**

# **the VFS can be accessed with the global variable FS**

#### **Methods**

# **fileSize(filename) - Returns size of file in bytes**

# **fileExists(filename) - Checks if file exists**

# **directoryExists(directory) - Checks if directory exists**

# **getFileRealPath(virtualPath) - Gets real filesystem path from virtual path**

# **getGamePath() - Gets root game path**

# **openFile(filename, mode) - Opens a file**

# **copyFile(src, dst) - Copies a file**

# **directoryCreate(name, recursive) - Creates a directory**

# **findFiles(path, pattern, recursiveLevels, multiMatch, includeDirs) - Finds files matching pattern**

# **hashFile(filename) - Gets file hash**

# **hashFileSHA1(filename) - Gets file SHA1 hash**

# **stat(filename) - Gets file stats**

# **removeFile(filename) - Deletes a file**

# **renameFile(oldName, newName) - Renames a file**

# **mount(source, target, priority) - Mounts a directory or archive**

# **mountList(mountList) - Mounts multiple paths**

# **unmount(path) - Unmounts a path**

# **isMounted(path) - Checks if path is mounted**

## **Global Functions**

# **initBeamEngine(steps) - Initializes the BeamNG engine**

# **destroyBeamEngine() - Shuts down the BeamNG engine**

# **reloadLua() - Reloads the Lua environment**

# **log(level, origin, msg) - Logs a message**

# **level: Log level (I=Info, W=Warning, E=Error, D=Debug)**

# **origin: Source of the log**

# **msg: Message to log**

# **updateLuaCore() - Updates the Lua core system**

# **profilerPushEvent(name) - Starts profiling an event**

# **profilerPopEvent() - Ends profiling the current event**

# **setPowerPlanMaxPerformance() - Sets system power plan to maximum performance**

# **restorePowerPlan() - Restores original power plan**

## **Global Variables**

# **hw - Table with hardware information**

# **hw.bits - CPU architecture (32 or 64 bit)**

# **args - Command line arguments**

# **beamng\_platform - Current platform (“windows”, “linux”, etc)**

# **Creating an app**

### **Introduction**

The user interface and apps of BeamNG.drive are written using the AngularJS (1.5.8) framework.

Some familiarity with the framework is useful, although not necessary. In this context, apps are just simple directives in the module beamng.apps.

The 3 important parts of each app are:

* An app.js file with all the code used by the app
* An app.json file that contains information about the app
* An app.png image file that shows up in the app selector

#### **The app.js file**

A BeamNG app directive generally follows this structure:

angular.module('beamng.apps')

.directive('myApp', ['StreamsManager', function (StreamsManager) {

return {

template: '[Some HTML Content]',

replace: true,

restrict: 'EA',

link: function (scope, element, attrs) {

// An optional list of streams that will be used in the app

var streamsList = [/\* streams here \*/];

// Make the needed streams available.

StreamsManager.add(streamsList);

// Make sure we clean up after closing the app.

scope.$on('$destroy', function () {

StreamsManager.remove(streamsList);

});

scope.$on('streamsUpdate', function (event, streams) {

/\* Some code that uses the streams' values \*/

});

}

};

}]);

The example above is an example of app that uses the vehicles' streams (which is by far the most common case). However this is not always the case, the apps can be used for just about anything. Again, the code must be included in the link function of the app.js file, for example:

angular.module('beamng.apps')

.directive('myApp', ['StreamsManager', function (StreamsManager) {

return {

template: '<button ng-click="hello()">Click Me</button>',

replace: true,

restrict: 'EA',

link: function (scope, element, attrs) {

scope.hello = function () {

// do something here.

};

}

};

}])

Sometimes, apps need to store some data. In this case, an extra file settings.json with this data can be added to the app folder. In order to use it, the directive must be slightly modified, like in this example:

angular.module('beamng.apps')

.directive('myApp', ['StreamsManager', function (StreamsManager) {

return {

template: '[Some HTML Content]',

replace: true,

restrict: 'EA',

// [1] we "require" the bngApp parent controller

require: '^bngApp',

// [2] the controller is available as 4th argument of the link function

link: function (scope, element, attrs, ctrl) {

var streamsList = ['sensors'];

StreamsManager.add(streamsList);

// [3] Use a variable to keep the settings

var appSettings = null;

// When DOM is ready and controllers are set up, get the stored settings.

element.ready(function () {

// [4] Call the getSettings() function of the controller

ctrl.getSettings()

.then(function (settings) {

appSettings = settings;

})

});

scope.$on('$destroy', function () {

StreamsManager.remove(streamsList);

// [5] Optionally save the (possibly modified) app settings when done

ctrl.saveSettings(appSettings);

});

scope.$on('streamsUpdate', function (event, streams) {

/\* Some code that uses the streams' values \*/

});

}

};

}])

#### **The app.json file**

This file is really simple and just holds information about the app, useful for loading it. However, one must be very careful with the fields’ names, as typos or missing fields might affect the proper execution of the app. The file contents have the form:

{

"name" : "My App",

"author": "Me",

"version": "0.1",

"description": "Just a tutorial app",

"directive": "myApp",

"domElement": "<my-app></my-app>",

"css": { "width": "150px", "height": "150px", "top": "200px", "left": "200px" },

"preserveAspectRatio": true

}

The required fields are:

Field

Description

name

The display name for the app

author

The authors of the app

version

The version number of the app

description

A small description of the app

directive

The name of the directive (the same as in the app.js file)

domElement

The DOM element that will actually hold the app. The domElement is determined from the directive name, and is just a conversion from *camelCase* to *lisp-case*, so in our example it would be from myApp to <my-app></my-app>

css

The default CSS attributes to be used when the app is first launched. These are width and height and a top/bottom, left/right property that also states the screen corner to which the app will be aligned

#### **The app.png file**

This is the image that shows up in the app selection view. Recommended size is 250x120px.

### **Using your app**

In order to make your app visible from the game, move the folder with the app’s files to your userfolder’s UI directory (<Userfolder>\ui\modules\apps).

The app will then be available from the selection list. You might need to close and open the list once.