Developer Information
on the Lua-API of AllTheHaxx

by xush', Henritees & FuroS

Disclaimer

Please note that this is not a 100% complete Lua tutorial with anything you need to know in order to become a professional programmer, this is simply some basic stuff for people who are interested in creating some scripts for the AllTheHaxx-Client. You will find some general tutorials on the Lua language, tutorials how the API works, how to implement your own scripts and a lineup of all available Lua commands for our Client.

WE ARE NOT RESPONSIBLE IF YOU GET BANNED FROM ANY SERVER BY A RAGING ADMIN BECAUSE OF AN ANNOYING SCRIPT YOU ARE USING!

This tutorial is not finished yet, so always check for updates if you need it. If you find a mistake or if something is unclear to you which you would like to see explained better, please contact the modderteam.

Further support & fastest contact:
IRC: irc.quakenet.net → Channel: #AllTheHaxx

or simply use the IRC Api in the client
(F5 → Join)

Version 0.2-rev1 April 17 2016 for Client *beta12*

0) Getting started : FAQ

What is Lua?

Lua is a dynamic scriptlanguage which can be embedded in other discrete programs. It allows to run external Lua-Scripts in the program through an embedded Just-In-Time-Compiler (because we use LuaJIT, normal Lua only offers an interpreter).

What do I need?

Through downloading the AllTheHaxx client the first step is already done.
We suggest to use a Lua-capable editor (notepad++, any IDE editor, no standard windows editor!!11eleven). Also reading this tutorial is really helpful (if not vital; experienced programmer can read through it fast tho)

What can I do with Lua?

We implemented Lua in our script in a way that you can do as many stuff as possible.
You can mess around in the LuaConsole (F6) or write your own functions, scripts etc...

How is the system set up?

You can write your own functions or scripts in .lua files and run them through our Lua-GUI in the settings or you can write local functions in the LuaConsole (they won't be saved on closing the client!).
Each .lua file has their own "LuaSpace", that means it doesn't clash with other scripts in any way.
In our system you can simply bind your written Lua-Functions to an Ingame-Event, such as OnTick, which is called every 50 times per second by the client.
Please note : We, the devs, must implement all Ingame-Functions to the Lua-API, so it might occur that there is no specific function for your needs provided, in that case contact the devs through skype or our QuakeNet Channel. Anyway, we try to implement as much updates with new necessary Lua-Functions as possible.

Positive:

No recompilation needed
You don't need to know C++ or any Teeworlds related coding stuff
Your written Lua-Stuff won't (usually) crash your client; error messages are printed in Teeworld's console

Negative:

You won't be able to rewrite the whole client to a new thing, you can only change the things that are offered to change and add minor new functionalities to it.
You have to read through this documentation.
You have to read through this documentation accurate if you can't programm at all, and we also recommend to checkout another Lua as this one is far from complete (there are great videos!).

1) Basic Lua Programming

In this chapter I will explain you the basic Lua-Syntax (no API-Stuff yet!); if you are new to programming in general you should start reading from here; the rest can just look at the syntax and continue :).

1.0 Comments

A comment in lua starts with two dashes ("`--`"). Everything behind will be ignored during execution. If you want to outcomment a whole block of code, you can do so by putting `--[[` at the start of your comment, and `]]` at the end.

1.1 Variables

Variables are used to store (and read) data in any way. That's all, Lua is dynamically typed, this means there are no datatypes etc., so you don't need to mind anything.

Variables must start with a letter and should indicite what they are storing by their name.

```
x = 10        → the number 10 is now stored in the variable x :)
y = "Text"    → y stores now the string "Text"
x = y         → x has now the same value as y     →    x is now "Text"
```

In General you can store anything you want in variables in Lua, even functions.

Valid operators for variables (and general)

Mathemathical: +, - , *, /, =

1.2 Functions

Functions are a seperate part of a script, they only do their stuff if you call them. Else they won't do anything.
Functions can get some input-variables (so called 'parameter') and can return one value or in internal in Lua multiple values (but this is usually not needed).

The structure of a function looks like this:

```
function Name (Parameters)
    -- Do stuff here
    return something –optional, also multireturn is possible!
end
```

Example:

```
function Sum(number1, number2)
      numbersum = number1 + number2
      return numbersum
end
```

As stated, this won't do anything on its own: You need to call the function. This can be achieved like this:

```
x = 5
y = 10
sum = Sum(x, y)    → the function is called, sums both numbers and returns 15; this
                     number then is stored in the variable 'sum'.
```

```
function helloworld() --no parameters!
    print("Hello world!")
end
```

This simply prints "Hello world!" into the Teeworlds-Console.

Call looks like this: `helloworld()`

1.3 The if-statement

This statement is important when you want something to be done based on a condition.

Send a chatmessage when someone says something in the chat, quit the game if you died 5 times etc. etc....

General syntax:

```
if Statement then -- "Statement" can for example be 5 ~= 3 (always true)
    -- do stuff...
end
```

Statements are usually comparisons of variables, numbers, texts, returnvalues from functions or anything else.

Operators:

`==` this operator in an statement returns true if the data left and right of the operator have the same value, thus are equal!

```
x = 10              → x is now assumed to be 10 in any other examples!

if x == 5 then
    print("x is 5!")
end
```

→ this is a false statement, because x is declared as 10 and 10 is not 5, so the print function will not be called.

In contrast:

```
if x == 10 then
    print("Yey, x is 10!")
end
```

→ this is a true statement because x is declared as 10 and 10 is equal to 10, so you will find a nice message in your console by the print function :)

`if x == "text" then` works also equally

`~=` → this operator is the opposite of ==, it will return true if the equation is not equal.

```
if x ~= 50 then
      ...
end
```

→ this return true, because x is declared as 10 and 10 is not 50.

`>` → returns true if the left argument is higher than the right

if x > 5 then → true, because 10>5

`>=` → returns true if the left argument is higher or equal to the right argument

`<`, (`<=`) → opposite of the `>` operator, return true if the right argument is higher (or equal)

Additional keywords:

not : This will negate the next statement, so if the next statement is true, it will now be false; a false statement will become true!

if not x == 10 then
→ this will return false, because x = 10, so the inner statement is true and thus, negated, will become false; the code in the scope won't be executed.

and : This is used to stick some statements together. Every part-statement must be true to get the code in the scope executed.

if x > 5 and x < 15 then
→ this will return true altogether, because 10>5 and also 10<15

or: This is also used to stick statements together but now only one of them must be true (the rest doesn't matter at all!)

if x == 10 or x == 100 then
→ this will return true altogether because x == 10 is true, so it doesn't matter if x is also 100

1.4 Loops

After understanding some statements which can be true or false, we can switch over to another important topic: loops.

There are 3 types of loops:

for :

This loop runs a specific amount of times, e.g. 5, 10, 50, 10000000000.

Syntax :

```
for CounterVariable=Start, End, Stepsize do
    --stuff...
end
```

CounterVariable is a variable which contains the actual counted iteration. You can use this variable in this loop, not outside!

Start is the start number, most of the times 0, but you can start higher or lower.

End is the final iteration. If CounterVariable == End then this loop is finished and the script moves over to the next commands.

StepSize : This number specifies the stepsize of the CounterVariable, so if StepSize is 1 then the CounterVariable will be increased by 1 after each iteration.

do : This is a necessary keyword to symbolize Lua the start of this loop. Such as end represents the endpoint of the loop.

=>
```
for i = 0, 15, 1 do
    --stuff...
end
```

while:

This loop runs as long as the statement is true. If the statement gets false, then this loop is finished.

Syntax:

```
while(Statement)
      --stuff...
end
```

Statement can be anything as in chapter 1.3.

Please note : With this loop (and in for) it is possible to create a (wanted or unwanted) infinite loop. Our system runs Lua in normal routines, so you should not create any infinite loops or any loop which has a long time to work on or else the client will hang itself in this loop => You need to restart your client.

do:

This is mostly the same as the while-loop and is usually not used, so this loop is skipped. Sry.

Additional loopkeywords :

break:

→This keyword ends the actual loop instantly. Logically it should appear after an if-statement, or else the loop doesn't work how it should :D

continue:

→This keyword skips the rest of the actual iteration and thus, the next iteration is started now. This means, a CounterVariable in for gets increased etc.

1.5 Additional Lua-intern commands & libraries

print : This function prints anything you give it as parameters in Teeworld's console.
(It prints in the LuaConsole directly if used in the LuaConsole).

```
print("Hi")  → prints Hi
print(5)      prints 5
print(x)      prints the value of x (in this case 10)
```

string-library :

This library provides some basic functions to use on strings ("textlines").

I won't tell you that much about it, only the most needed stuff.

Usage :

You can use string.command(Variable to be used, Parameter...)

or

Variable:command(Parameter).

Often needed commands for Teeworlds:

Text = "Teeworlds in Lua rocks!"

`Text:find("Lua")` → this returns the position of "Lua" in Text(and thus true for the keyword if).

`Text:gsub("Teeworlds", "Worldtee")` → this replaces simply all "Teeworlds" with "Worldtee"

For further information visit : http://lua-users.org/wiki/StringLibraryTutorial or Google :P

2) Setting up a script

2.1 – General

Now you know what a function is, what variables are and other useful stuff. Now we can start writing our scripts.

In general a script is made out of 2 parts :

1. One or multiple functions which do the main work of the script.

2. Variables which are used for all functions of that script outside of any function, usually at the top/start of the script
   + some functions needed to initialize the script.

Example:

```
x = 0       --we initialize x for all functions

a()   --we initialize the script here by calling the function a()

function a()      --do something with x
    x = x+5
    b()
end

function b()
    print(x)
end
```

In the end the following will happen :

`x = 0` → function a is called, it increases x by 5 (x is 5 now) and calls b().
b() then simply prints 5 in the console.

Though neither a() nor b() have gotten the variable x as parameters they can acces it because x is available for both, a() and b() (and any other function in this file...).
=> x is global for both functions.

Extra : The keyword 'local'

This keyword sets a variable or a function as local for this file. Because our system already load each file in it's own memory space and because EventFunctions are accessed by their name (read further in the next chapter) you should not use this keyword for functions, otherwise eventcalls won't work.

2.2 – Teeworlds Events in ATH

This is an important chapter as it's the core of our API.

To understand it we should face the following problem :

You write your script with its variables, functions etc.
But how does Teeworlds know when to call them?
=> That's where events come in

Teeworlds (as many other games, too) has intern events. These are functions in Teeworlds that are called after a specific thing happened, and they handle this event. We built our API in a way that allows you to bind your own written Lua-functions to a specific event that you need. If this event happens in Teeworlds, your function will be called.

Following events are available in Lua (for now) :

`OnTick()`

no parameters

Maybe the most important event. A tick function, in game-programming, is the main function of a game as it's called a specific number of times per second and handles all important stuff which happens ingame (moving characters, creating new entities like bullets, check for player collision, checking if a player is hit by a bullet etc.)

In Teeworlds, the function OnTick is called 50x per second, so every 20ms.
You should register a function to this event if you need to do something each game step, or simply if there is no other event you can bind to to get the script work.

Important: Make sure that you don't put heavy stuff into your OnTick function, or it will significantly slow the whole game down!

`OnScriptInit()`

no parameters

Called when the script is initialized.

Important: Make sure to return true, otherwhys the loading will fail!

`OnScriptUnload()`

no parameters

Called when the script is unloaded.

Hint: Reset stuff here

`OnChat(ID, Team, Message)`

ID: The ID of the player who sent the chat or -1 if it's a server message
Team: Is 1 if the chatmessage is teamchat (the green one), otherwise 0
Message: The chat message itself, in plain text.

This event is called when a chatmessage comes in, either from a player or as serverchat (the yellow one) from the server.
It passes the ID of the player who sent the message (more on this later), the team (allchat or teamchat (the green one)) and the message itself.

`OnEnterGame()`

no parameters

This event is called when you enter (join) a server.

`OnKill(Killer, Victim, Weapon)`

Killer: ID of the player who scored the kill
Victim: ID of the player who got killed
Weapon: The weapon which was used my the killer

This event is called when a player dies (watch your killmessages on the top right of the screen).
Hint: if Killer is equals victim, then it was a self kill (suicide)

You can see all events that exist in the file *data/luabase/events.lua*, but DO NEVER EDIT this file, or the whole lua api won't work anymore!!

X) Lua-API Commandlist

Preamble: Claiming special permissions for you script – Only for advanced users!

Be aware that scripts do not have access to all parts of native lua by default for security reasons.. If you need special things like writing files, getting the system time or even shutting down the computer, you need to open those libs for you script. Do the following:

```
--[[#!
    #io         ←reading and writing files
    #os         ←various operating system functionalities
    #ffi        ←execution of native c code from within lua
    #debug      ←debugging shit, usually not needed
]]--#package    ←package library
```

for further information on the libraries itself, consult the Lua Reference Manual:
http://www.lua.org/manual/5.1/manual.html#5

// TODO: rewrite this better :D

Note:

Everything that ends with a : (colon) is a function and needs arguments, at least ()

Everything with a . (dot), e.g. Game.Chat.Mode is treated like a variable.

If there is a '-- *only get*' note, you can only read from it, but not write into it:

var = Game.Chat.Mode => Correct
Game.Chat.Mode = var => Wrong, because you try to write in the Var.

Game

Game.Chat

Game.Chat:Say((int)Team, (string)Message)
Game.Chat.Mode --only get

Game.Console

Game.Console:ExecuteLine(Line, ID)
Game.Console:LineIsValid(Line)
Game.Console:Print((int)OutputLevel, (string)From, (string)Line, (bool)Highlighted)

Game.ServerInfo --only get (for all)

Game.ServerInfo.GameMode
Game.ServerInfo.Name
Game.ServerInfo.Map
Game.ServerInfo.Version
Game.ServerInfo.IP
Game.ServerInfo.Latency
Game.ServerInfo.MaxPlayers
Game.ServerInfo.NumPlayers

Game.Emote

Game.Emote:Send(EmoteID)
Game.Emote:EyeEmote(EyeEmoteID)
Game.Emote.Active --only get

Game.HUD

Game.HUD:PushNotification(Text, (vec4)Color)

Game.Menus

Game.Menus.Active --only get
Game.Menus.ActivePage --get & set
Game.Menus.MousePos --get & set; vec2 !

Game.Voting --only get (for all)

Game.Voting.CallvoteSpectate(ClientID, Reason)
Game.Voting.CallvoteKick(ClientID, Reason)
Game.Voting.CallvoteOption(OptionD, Reason)
Game.Voting.Vote(v) -- 1 = yes, -1 = no
Game.Voting.SecondsLeft
Game.Voting.IsVoting
Game.Voting.TakenChoise -- what did the player vote?

Game.Input

Game.Input.Direction
Game.Input.Fire
Game.Input.Hook
Game.Input.Jump
Game.Input.WantedWeapon
Game.Input.TargetX
Game.Input.TargetY --all get & set until here
//Messis Keysachen da

Game.Collision

//do the functions as property first

Game.Ui

Game.Ui:DoLabel(UIRect rect, Text, (float)Size, Align, MaxWidth, (char)Highlight)

Game.LocalTee --only get!

Game.LocalTee.Pos --vec2
Game.LocalTee.Vel --vec2
Game.LocalTee.Hook
Game.LocalTee.Collision
Game.LocalTee.HookPos --vec2
Game.LocalTee.HookDir --vec2
Game.LocalTee.HookTick
Game.LocalTee.HookState

Game.LocalTee.HookedPlayer
Game.LocalTee.Weapon
Game.LocalTee.Direction
Game.LocalTee.Angle
Game.LocalTee.Jumps
Game.LocalTee.JumpeTotal
Game.LocalTee.Jumped

Game.LocalCID --simply a number variable

(ID goes from 0 – 63 and represents the ingame ID of a player (see Scoreboard)).

Game.Players(ID) --ID = player's ID (in scoreboard etc)
Game.Players(ID).ColorBody --only getter for all, too
Game.Players(ID).ColorFeet
Game.Players(ID).UseCustomColor
Game.Players(ID).SkinID
Game.Players(ID).SkinColor
Game.Players(ID).Country
Game.Players(ID).Team
Game.Players(ID).Emote
Game.Players(ID).Friend --not sure if this works :/
Game.Players(ID).Foe
Game.Players(ID).Name
Game.Players(ID).Clan
Game.Players(ID).SkinName

Game.Players(ID).Tee --only getter
Game.Players(ID).Pos --vec2 nop, not yet xD
Game.Players(ID).Vel --vec2
Game.Players(ID).Hook
Game.Players(ID).Collision
Game.Players(ID).HookPos --vec2
Game.Players(ID).HookDir --vec2
Game.Players(ID).HookTick
Game.Players(ID).HookState
Game.Players(ID).HookedPlayer
Game.Players(ID).Weapon
Game.Players(ID).Direction
Game.Players(ID).Angle
Game.Players(ID).Jumps
Game.Players(ID).JumpeTotal
Game.Players(ID).Jumped

Game.Client
Game.Client.Tick --important getter
Game.Client.LocalTime --time in seconds since the client runs
Game.Client:Connect(Address)
Game.Client:Disconnect()
Game.Client:DummyConnect()
Game.Client:DummyDisconnect(Reason)

Game.Client:DummyConnected() --do this as property
Game.Client:DummyConnecting() --this too
Game.Client:SendInfo((bool)Start) --usually pass *false*
Game.Client:RconAuth(Name, PW) --?
Game.Client:RconSend(Command)
Game.Client:RconAuthed() --do this as property
Game.Client.FPS --only get
Game.Client.State --only get
Game.Client:DemoStart(Filename, (bool)WithTimestamp, (int)Recorder)
Game.Client:DemoStop((int)Recorder)
Game.Client:Quit()

Graphics

Graphics.Engine

Graphics.Engine:QuadsBegin()
Graphics.Engine:QuadsEnd()
Graphics.Engine:LinesBegin()
Graphics.Engine:LinesEnd()
Graphics.Engine:SetRotation(Angle)
Graphics.Engine:SetColor(r, g, b, a)
Graphics.Engine:BlendNone()
Graphics.Engine:BlendNormal()
Graphics.Engine:BlendAdditive()
Graphics.Engine:MapScreen(TopLeftX, TopLeftY, BottomRightX, BottomRightY)
Graphics.Engine.ScreenWidth
Graphics.Engine.ScreenHeight

Config

Config.PlayerName
Config.PlayerClan
Config.Skin
Config.PlayerCountry
Config.PlayerColorBody
Config.PlayerColorFeet
Config.PlayerUseCustomColor