**The Lua Template Project**

**Designer’s Readme**

**Why Use Lua?**

Lua has significant enhancements over the base event tools available in Civilization II Test of Time. Most things that can be dreamed of can be accomplished via this powerful language, as several scenarios have shown over the past few years. With that said, lua can be a bit overwhelming at first.

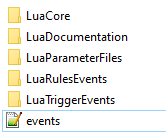
The following is intended to help break down the barrier that lua events can have to new modders who are unfamiliar with the language. This is not a technical guide to discuss all the intricacies of lua. Indeed, it is not even written by a coder. Instead, it is written by a scenario designer who has spent the past few years struggling to learn this language and who wants to make it more accessible to others.

**The Events File Structure**

Let’s suppose that you have built a scenario in game. Your units and cities are placed, your rules file is complete, all the art is in, and now you’re looking to finalize the scenario by adding events. In the base game, you would do this by creating a text file called “Events” and starting with:



Here, things are a little more complicated. Below are the five folders and base events file that make the lua template work. All of these should be within your scenario folder (for example, C:/Test of Time/Scenario/Example Scenario:



Let’s talk a bit about each of these. I’m going to start by going over what is generally in the folder and then we’ll dive into each one in more detail (or, at least the ones you need to worry about).

**LuaCore** – this folder contains several lua files – you do not make any changes to these, but will occasionally overwrite them if enhanced versions of the files are made available. Think of this folder, in conjunction with the events filed discussed below, as kind of containing all the “hardware” or “guts” that will make everything else work. Copy it into your scenario file and forget it.

**LuaDocumentation** – this folder simply contains some documentation / hints that may be useful. I suggest you read them when you have time. In fact, this might be where you ultimately found THIS file.

**LuaParameterFiles** – this is an important folder, and one that contains two .lua files, “object.lua” and “parameters.lua” Both can and should be modified by you. We’ll discuss those both in further detail later, but suffice it to say, this folder basically contains the “building blocks” that you’ll use for your scenario, and you’ll be editing it often.

**LuaRulesEvents** – this folder contains numerous .lua files, most of which are called “Settings” (e.g. “munitionSettings.”) The way Prof. Garfield describes it, “rules events are either always active or occur under regularly occurring circumstances … If you would expect to find the event in a readme, it is probably a rules event.” While we will discuss it further later, think of it this way: here, you basically set the “rules” for how certain events will play out. For example, if you are using munitions in your scenario, munitionSettings,lua would be the file you’d modify to declare which units fire what munitions and under what circumstances.

**LuaTriggerEvents** – this folder contains sub folders and different files, but you could get away with modifying only one: triggerEvents.lua. This is where you will find different triggers for situations like when a unit is killed, or if a city is captured. You will probably spend the bulk of your design time in this file and we’ll go over the other triggers further below.

**Events** – while it is counterintuitive, you don’t make any changes to this file. Prof. Garfield has already done the “behind the scenes” work for you.

So, in sum, you need to worry about the following 3 folders:

1. LuaParameterFiles
2. LuaRulesEvents
3. LuaTriggerEvents

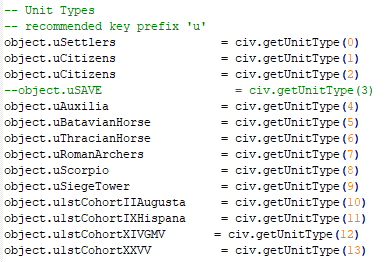
Let’s discuss those in a bit more detail, in order.

**LuaParameterFiles**

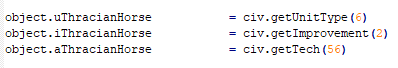
This folder contains two important .lua files, “object.lua” and “parameters.lua” Object.lua is where you store all the building blocks of your scenario (the different units, techs, cities, etc.). Parameters, while optional, can save you significant time, reduce errors, and generally make a scenario feel much more polished.

**OBJECT.LUA**

The **object** file is where you’re going to store all the different “objects” that you’re going to need in your events. For example, all units that will be part of an event should go in here, as should all cities that will have a specific event. Here’s part of the object file for Boudicca:



Note that we prefix each unit with a “u,” and that each unit corresponds to its UnitType integer (where settlers in the base game, for example is unitType 0, and the phalanx is unitTyp3 3). The “u” prefix is simply done for organization’s sake – it isn’t absolutely necessary – but it is helpful if you were to have, for example, a technology, unit, and improvement that all shared the same name.



All three of these are different things. Had you not used “u,” “I,” or “a,” and instead simply wrote:



The event would not work properly. Thus, we highly recommend that you make use of the prefixs provided in the module:

u = unit

i = improvement

w = wonder

a = technology

c = city

x = text

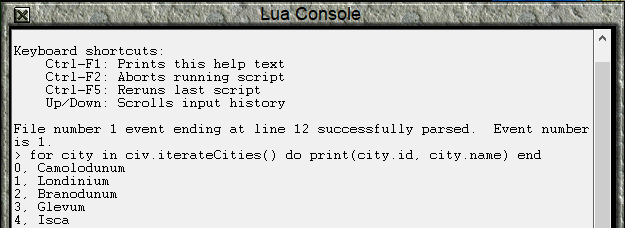
t = tribe

It would be a good idea to start building your object file once your rules file is complete and the cities are placed on the map, as you’ll have this open nearly the entire time that you’re implementing other lua code.

It may not be obvious how to gather the city ids. What you need to do is to go into your scenario save file, enter cheat mode, press ctrl + shift + f3 to bring up the lua console and then type:

for city in civ.iterateCities() do print(city.id, city.name) end

Hit enter, and this will show up – a list of all the cities and their integer.



You would then type these into the object file as so:



Note that not everything in the object file needs to start with “object.” Indeed, we suggest that you also store flags here (as discussed below—INSERT).

**PARAMETERS.LUA**

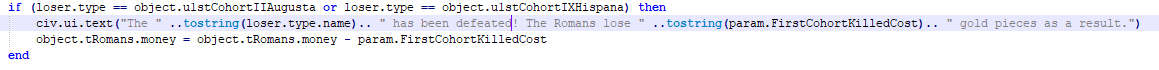
The **parameters** file is an extremely useful file to the scenario designer. Its purpose is to store different values that you are going to refer to throughout the scenario, so that they can easily be found and changed in one place.

For example, while building Boudicca, I knew that I wanted there to be a monetary penalty for losing one of the First Cohort units, but I wasn’t quite sure what that penalty should be, and would need to refine it during testing. I started by defining the penalty in the parameters file:



This assigns the value of 100 to “param.FirstCohortKilledCost.”

In the scenario, I then utilized this parameter as follows:



Let’s break this down a bit.



**“if” starts the event. This is the “trigger” from the macro language. The (loser.type == object.u1stCohortIIAugusta or loser.type == object.u1stCohortIXHispana)** tells the game that I want to run this event if either the 1st Cohort II Augusta OR 1st Cohort IX Hispana unit is killed. “then” is the action that will take place after the trigger conditions are met.



--this is the command to have text. After this, you’d have to also have (“whatever text you want”). Both the parentheses and quotation marks are necessary.



Note that “The “ is enclosed in quotation marks. Following that, we see 

What this is saying is that I want the events to supply the name of the unit. This allows me to use the same event for two different types of units, *and to have unique text for each*.



A new quotation mark continues the text, which ends at the second quotation mark (right before the next ..tostring). The second ..tostring is used to tell the game that I want the value I defined in the parameters file to also populate in the message. In this case, as you may recall, it is 100.

Basically, everything the civ.ui.txt(“stuff”) is what I want to pop up in a box.

Following that, we see:



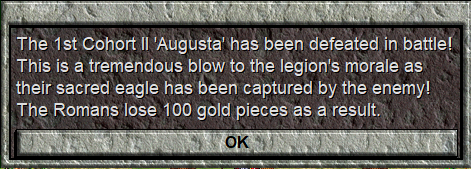
This is how you reduce the Romans money by the param.FirstCohortKilledCost, which we defined as 100 earlier. The event basically says “I want the Romans money to be what they currently have less the penalty for losing that unit.”

Note that I could have written this instead, with the same effect:



Of course, if I did that, and then in testing decided that the penalty was too high, and wanted to change it to 50, I’d have to manually find every line of it and make the change. Using the parameters file allows me to accomplish this in one place instead. I trust you can see the benefit.

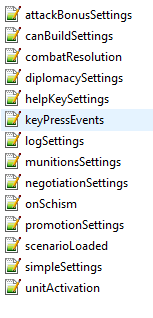
Let’s look at how this all plays out in the game (note there’s a little extra text in there that I cut out to make the pictures above of legible size):



By utilizing parameters instead of the “100,” I can change things rapidly and leverage ..tostring to ensure that my text reflects what is actually occurring, all from one place! It’s one of those things where if you set it up ahead of time, you’re really setting yourself up for an easier build and a better end user experience.

**LuaRulesEvents**

This folder has a lot going on inside of it. As of writing, all the following are contained therein, with the possibility that more may be added later.



I mentioned above that the rulesEvents are made to work in the background and aren’t dependent on specific triggers. Indeed, many of them (such as negotiationSettings) can work in conjunction with triggers found in the appropriately named triggerEvents. There are a few files (like unitActivation) which are basically triggers in and of themselves, but this was never meant to be a perfect division. Let’s do a little highlight over each one for now, and then we’ll get into some details about how you can modify each to do your bidding.

**attackBonusSettings** – This file will modify how a unit can acquire an attack bonus under certain settings. For example, in *Boudicca* (and *Napoleon* before it, where the concept was pioneered) “leader” units such as generals provide a bonus to other units that are activated while sharing the same square.

**canBuildSettings** – An extremely useful file that allows you to decide which cities can build which units, improvements, or wonders, and under what circumstances. This allows for considerable

**combatResolution** – Cheat death with this file (well, for your units anyway). Essentially, you can decide if certain units live or die in combat and under what circumstances. This could be useful for a “sword in the stone” situation where you only want one type of unit to be able to kill others, or could be used so that you could *both* have a “proper” partisan unit in the partisan slot *and* have a “flamethrower” type unit that can easily kill 0 attack units (a design trick is that units in the partisan slot have an 8x combat advantage against 0 attack units—this used to mean you had to choose if you wanted partisans or these special units – that is no longer an issue).

**diplomacySettings** – provides different options for diplomatically giving units and cities away to different civs. Particularly useful in multiplayer scenarios (*The Cold War 1947-1991* makes extensive use of it) but also helpful for supporting AI allies.

**helpKeySettings** – As scenarios are becoming increasingly complicated, having a help function is quite important. This file allows you to provide some data that will be given to the end user. In Over the Reich, we used it to provide information on the nearest friendly airfields so you could calculate if your aircraft could return to base or not.

**keyPressEvents** – Here you can define what happens when players press certain keys. Since *Caesar’s Gallic Wars*, “k” has been the key most designers have chosen for ranged attacks, but you could use others.

**logSettings** – One drawback to multiplayer scenarios is that it can take a lot of time to write down everything that happened and then report it to your friend. This file enables you to set up settings for a reporting log that will detail important events that occurred in the scenario so that the next player can simply press a key and have all that information at their fingertips. This was used extensively in *Over the Reich*.

**munitionSettings** – This file is where you would input all the details for any units that have a ranged attack, or simply spawn different units. So far the scenarios that have used this concept have used it to represent munitions, but you could also have a “Gremlins” unit spawn other gremlins if they were on a water tile using the same theory.

**negotiationSettings** – This is where you can define if the human and AI can talk to each other/negotiate.

**onSchism** – Out of the box, prevents a civilization from splitting in two when its capital is taken. No modification is necessary unless you *want* the split to occur.

**promotionSettings** – This module allows for units to promote or demote to other units. For example, in *Over the Reich* and *Midway*, certain units (like B-17s or aircraft carriers) demoted to damaged versions of themselves after first being “killed” in combat. Likewise, you could have units *promote* instead.

**scenarioLoaded** – A place to store different events for when the scenario is first loaded. In *Cold War*, I used this to warn players what events file they had loaded (single player or multiplayer) so that they wouldn’t play with the wrong ones.

**simpleSettings** – Kind of a catch all for some basic settings that don’t fit elsewhere. For example, one can disabled “We love the king day” celebrations here, to control rampant city growth. This is also where air protected stacks can be modified to remove that “feature.” In *Over the Reich* we did that so that ground forces couldn’t “hide” under air units.

**unitActivation** – in my mind, this one is almost more of a trigger event as a unit activating is a trigger. This is a file where you’d place events that you want to run when certain units are activated. Of particular use, you should realize that an AI unit “activates” *every* tile that it moves. Note that this does not work if a player manually activates a unit within the city screen:



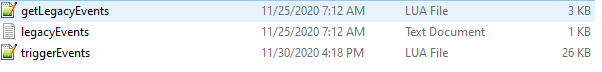
It *will* work if the game naturally *cycles* to a unit that is within the city.

Now that we’ve gone over things at a high level, let’s dive into each one by one using examples from *Boudicca*.

[INSERT EXAMPLES FOR EACH]

**LuaTriggerEvents**

This folder has two subfolders (ContextTriggerEvents and UniversalTriggerEvents) that we will get into later. For now, let’s focus on the other three files:



**getLegacyEvents** – you do not need to modify this file. What it does is allow you to leverage your current knowledge of the macro.txt events and still have a lua-based scenario. Simply write the events as you normally would in the **legacyEvents.txt** file, and **getLegacyEvents** will read them and incorporate them into your scenario. You can then add different lua-specific functionality as needed. This can be a useful steppingstone to creating some cool new scenarios.

**triggerEvents** – This is one of the main files that you’ll be working on. Indeed, you may spend most of your design time in here. In it you will find different triggers that define situations that will elicit some response or action. As of writing, the following triggers are in place:

**onTurn** – this trigger fires as the name implies, on a given turn. Just as with the macro.txt, if there are text events that fire on turn, they will fire as the first civ (usually white, unless it was destroyed) plays its turn. In multiplayer, that means no other civ will see the event (for that, you’ll want to use afterProduction as described below).

**unitKilledInCombat** – Used for when a unit is killed in combat, vs. some other circumstance. This is pretty much the one you’d want to use if you were expecting macro.txt functionality.

Of note, you can do considerably more with this than you can in the macro.txt. Of usefulness, you can choose to only have the event fire if the unitType is on a particular square. In *Boudicca*, one unit (the Legion Standard) has a different effect depending on *where* it is killed. Thus, a macro.txt classic like Techumseh’s *Operation Market Garden* would not need multiple bridge sapper units, with one for each bridge. Instead, one could be used, and a different event could fire dependent on its location.

**unitDefeated** – This will run if a unit is defeated in combat, or via events (for example via the **CombatResolution** module discussed above.

**unitDeath** – this fires when a unit dies, in or out of combat, just so long as it isn’t replaced by something else.

**unitDeleted** – this happens if a unit is deleted via combat or some other event (but not via disbanding).

**onCityProduction** – Fires when a city produces something. In *Over the Reich* we used it to transfer certain units (15th AF and 332nd Fighter Group) that was “produced” in England over to Italy instead.

**afterProduction** - This is quite a useful trigger because it allows “onTurn” events to fire on a specific *tribe’s* turn, after one of their units has activated. This is great for multiplayer games because it allows introduction text, for example, to be viewed by all players and not simply the first player (white civ). This also allows you to ensure that the AI has the initiative via certain events, even if they play their turn after the human civ. If you want to create units for the AI, and don’t want the human player to destroy them before they have a chance to move, you should place them here.

**IMPORTANT:** this will not fire unless there is a unit that can activate, so it may be inappropriate for sparsely populated scenarios, or ones where you feel all units are likely to be fortified. I’ve been befuddled more than once by events that would not fire because I was testing them in situations where all my units were on sentry.

**function triggerEvents.onCityTaken** – A trigger that fires when a specific city as defined in the object file is captured.

**function triggerEvents.onCentauriArrival** – A trigger that will fire an event when a civ has successfully launched a spaceship.

**function triggerEvents.onCityDestroyed** – Fires when a particular city is actually *destroyed* as opposed to simply captured.

**function triggerEvents.onBribeUnit** – A trigger that can modify what happens after a unit is bribed. For example, perhaps you don’t want to consider it a “bribe” so much as a payment, and you want the former owner to receive compensation. Or, perhaps you want to ensure that the two nations don’t war with one another. This can all be accomplished here.

**function triggerEvents.onGameEnds** – A trigger like the macro.txt where one can have certain things occur when the game ends (usually text boxes, I’d imagine).

**function triggerEvents.onCityFounded** – This trigger fires when a city is founded. You can use this to set the city’s name, to add improvements (or units) to it immediately, or even to delete the city if you don’t want them to be founded.

**USEFUL FUNCTIONS / TIPS & TRICKS**

Below I intend to give a few examples of how different things can be achieved via lua… Will have random things we do in Boudicca that don’t quite fit elsewhere:

-transferring city ownership/unit ownership

-polygons/occupation zones

-flags

-counters

**HELPFUL THREADS / POSTS**

-Will probably link a few of the helpful threads/posts/articles/etc.