**Lua**

The scripting language used by the Witchcraft Engine is Lua, specifically Lua 5.1, as implemented by LuaJIT 2.0.5.

Every script runs in its own private environment. This means that global variables defined in one script are not visible from another. This private environment does not have access to everything in \_G. A whitelist of functions from \_G are available to a script’s environment. The following functions and values are available and have the same behavior as regular Lua 5.1:

* assert
* error
* ipairs
* next
* pairs
* pcall
* select
* tonumber
* tostring
* type
* \_VERSION
* xpcall
* coroutine.create
* coroutine.resume
* coroutine.running
* coroutine.status
* coroutine.wrap
* coroutine.yield
* math.abs
* math.acos
* math.asin
* math.atan
* math.atan2
* math.ciel
* math.cos
* math.cosh
* math.deg
* math.exp
* math.floor
* math.fmod
* math.frexp
* math.huge
* math.ldexp
* math.log
* math.log10
* math.max
* math.min
* math.modf
* math.pi
* math.pow
* math.rad
* math.random
* math.sin
* math.sinh
* math.sqrt
* math.tanh
* os.clock
* os.date
* os.difftime\*
* os.time\*
* string.byte
* string.char
* string.find
* string.format
* string.gmatch
* string.gsub
* string.len
* string.lower
* string.match
* string.rep
* string.reverse
* string.sub
* string.upper
* table.insert
* table.maxn
* table.remove
* table.sort
* table.unpack

*\*Note that while the time-related functions from the “os” table are available, it is not recommended to use these for most purposes; use the values from the “time” table (see below) instead.*

The following functions which are present in regular Lua are re-defined to suit the needs of the engine:

* print(str); correctly redirects output to ‘stdout’, ‘log.txt’, and the in-game console.
* dofile(scriptname); runs the specified script file in its own secured environment, like any other script.
* require(scriptname); as ‘dofile’, but only runs if the specified script has never been run before.

**Additional Functionality**

Besides the functionality from vanilla Lua which has been whitelisted, additional tables and functions are available.

The function ‘readonly(tbl)’ can be used to grant read-only access to a table; it is defined as:

function readonly(tbl)

return setmetatable({}, {

\_\_index = tbl,

\_\_newindex = function(tbl, key, val)

error("Attempting to write to read-only table.") return end,

\_\_metatable = false

})

end

The table ‘script’ provides read-only access to other scripts which have been loaded. This table contains a read-only table for each script which has been loaded, with a key equal to that script’s name. Values which are declared in a script’s “global scope” will be readable by other scripts in this way. If a table is defined in a script’s global scope, the contents of that table will be both readable and writable by default. Variables declared using “local” will never be visible to another script. If you want a table to be readable but not writeable, declare it using “local” and use ‘readonly’ to create a visible version of it.

The constant ‘\_SCRIPTNAME’ is available for each script. It is a string equal to the name of the currently-running script.

The table ‘time’ provides read-only access to time-related values defined by the engine:

* delta\_time; the number of seconds which have passed since the previous frame. If running during the logical update, this will always be equal to \_LOGICAL\_SECONDS\_PER\_FRAME.
* frame\_counter; the number of frames which have processed since the application started. If running during the logical update, this is the number of logical frames; if running during the display update, this is the number of display frames.
* now; the number of seconds which have passed since the application started.
* \_LOGICAL\_FRAMES\_PER\_SECOND; this constant is equal to 30
* \_LOGICAL\_SECONDS\_PER\_FRAME; this constant is equal to 1/\_LOGICAL\_FRAMES\_PER\_SECOND, or 33.333… milliseconds.

The ‘math’ table contains a handful of additional constants and functions:

* tau; Equal to 2 \* PI.
* deg2rad; Equal to PI / 180. Multiply with an angle in degrees to convert it to radians.
* rad2deg; Equal to 180 / PI. Multiply with an angle in radians to convert it to degrees.
* clamp(val,min,max); If ‘val’ is less than ‘min’, returns ‘min’. If ‘val’ is greater than ‘max’, returns ‘max’. Otherwise, returns ‘val’.
* linstep(val, min, max); Returns a value between 0 and 1 corresponding to how far between ‘min’ and ‘max’ ‘val’ is. Returns 0 if ‘val’ is less than ‘min’, and 1 if ‘val’ is greater than ‘max’.
* lerp(from,to,alpha); Returns a linear interpolation between ‘from’ and ‘to’, according to ‘alpha’ ((from \* (1-alpha)) + (to\*alpha)). ‘alpha’ is expected to be within the range ‘[0,1]’ but this is not strictly enforced.
* sign(val); Returns -1 if ‘val’ is less than 0, or 1 if ‘val’ is greater than or equal to 0.
* angle\_difference(a,b); Returns the shortest angle difference between the two angles. Inputs and outputs are measured in degrees.
* radians\_difference(a,b); As ‘angle\_difference’, but measured in radians.
* spring(current,velocity,target,tightness,delta\_time); Used to push a value (‘current’) towards ‘target’ in a spring-like fashion, according to ‘tightness’ and ‘delta\_time’. Returns ‘current’ and ‘velocity’ which should be fed back into this function in future frames. ‘velocity’ should be zero the first time this function is used. ‘delta\_time’ should be the amount of time passed (in seconds) since the last time this function was called.
* spring\_angle(current,velocity,target,tightness,delta\_time); As ‘spring’, but with respect to angles (measured in degrees).
* spring\_radians(current,velocity,target,tightness,delta\_time); As ‘spring\_angle’, but measured in radians.

**Events**

When a script is attached to an entity, either by an area being loaded, an entity being spawned, or a script being explicitly attached to an existing entity, the engine will look in that script’s “global scope” for a function named ‘on\_init(id,params)’ and execute it. The first argument is the ID of the entity, the second is a table containing parameters for the script (this table might be nil).

Entities do not automatically store any state regarding scripts; this state will need to be stored explicitly. The following is the recommended way to do so:

local state = {}

function on\_init(id, params)

state[id] = { parameters = params }

end

When a script is removed from an entity, either by an area being unloaded, an entity being deleted, or a script being explicitly removed from a live entity, the engine will look in that script’s “global scope” for a function named ‘on\_destruct(id)’ and execute it with the ID of the entity. This function returns ‘nil’ if the entity’s state is not persistent; if it is persistent, this function should return a table containing the values which should be passed to this entity’s ‘on\_init()’ the next time it is created; this table should not contain any tables or userdata types (including entity IDs), as only numbers, strings, and booleans are saved.

function on\_destruct(id)

local mystate = state[id]

state[id] = nil

return mystate.parameters

end

‘on\_init()’ and ‘on\_destruct()’ are the only functions in a script which are automatically called. The table ‘events’ provides access to various events which are executed by the engine and allows scripts to create new events which other scripts can register for. Each event has a table, and each table contains the following values:

* eventid; A string consisting of the name of the event.
* listen(id,script,func); Allows a script to listen to an event. ‘id’ is the ID of the entity listening to this event, ‘script’ is the name of script (so multiple scripts attached to the same entity can listen to the same event), and ‘func’ is a function which accepts one or more argument and has no return value.
* unlisten(id,script); Causes the script to stop listening to an event. ‘id’ and ‘script’ are the same as passed to ‘listen’.
* execute\_global(…); For every entity which has registered for this event, it calls the ‘func’ as passed to ‘listen’, with the entity’s ID followed by ‘…’ as arguments.
* execute\_local(id, …); As ‘execute global’, but only for the specified entity.

The following events are present within the ‘events’ table, giving scripts access to events which are called explicitly by the engine:

* logical\_update; This event executes during every logical frame; that is, at a fixed frequency of 1/\_LOGICAL\_FRAMES\_PER\_SECOND. \_LOGICAL\_FRAMES\_PER\_SECOND is a constant equal to 30, so this update runs 30 times per second, or every 33.333… milliseconds. Script functionality which is expected to run “every frame” should go here. No arguments other than the entity’s ID are passed.
* display\_update; Executes immediately before the visible frame is drawn to the screen. Its frequency is undefined, as display updates occur as fast as the user’s hardware allows it. This event should only be used for special functions which directly interface with the renderer, such as debug drawing. Besides the entity’s ID, a number between 0 and 1 is passed, indicating the interpolation between logical frames that this display frame occurs at.
* animation\_event; Executes only for entities which have an “animator” component, and runs when an animation event is encountered. This is used to notify the script regarding when to play sounds, spawn particles or projectiles, etc. Besides the entity’s ID, a string describing the animation event is passed as an argument.
* entity\_deletion; Runs for every entity whenever any entity is deleted. This event should be listened to if your script keeps references to other entities, so that it can clean up those references. Passes both the listener’s entity ID and the deleted entity’s ID as arguments.

In addition, the ‘events’ table contains the function ‘create\_generic\_event(name)’. This function allows scripts to define their own events which other scripts can listen to. It returns a table containing all of the functionality seen in pre-defined events.