XLua

Xlua is a lightweight scripting plugin that can be added to an aircraft to create custom datarefs and commands, or automate very simple systems tasks. It can also drive custom panel and 3-d animations.

SCRIPT PACKAGING AND BASIC STRUCTURE

XLua scripts are organized into “modules” - each module is a fully independent script that runs inside your aircraft.

- Modules are completely independent of each other and do not share data -they are designed for isolation.
- You do not need to use more than one module. Multiple modules are provided to let work be copied as a whole from one aircraft to each other.
- Each module can use one .lua script file or more than one .lua script file, depending on author’s preferences.

The installation of the plugin looks like this on disk:

```
My airplane
        plugins
                xlua
                        64 <- this contains the real binaries
                                mac.xpl
                                win.xpl
                                lin.xpl
                        init.lua  <- provided by Ben
                        scripts
                                adf
                                        adf.lua
                                        adf_helpers.lua
                                stec_500
                                        stec_500.lua
```

In this example, there are two modules installed: “stec_500” and “adf”. The main lua file for a module has the same name as the module itself.

The adf module has a second lua file - for this file to be used, it must be “included” from adf.lua using dofile(“adf_helpers.lua”). Only the “main” lua script in a module is run automatically.

Sub-folders in the scripts folder are *not* allowed - all modules must be within “scripts”.

The file init.lua is part of the XLua plugin itself and should not be edited or removed.

How a Module Script Runs

When your aircraft is loaded (before the .acf and art files are loaded) the XLua plugin is loaded, and it loads and runs each of your module scripts.

When your module's script is run, all Lua code that is *outside* of any function is run immediately. Your script should use this immediate execution only to:

- Create new datarefs and commands specific to your module and
- Find built-in datarefs and commands from the sim.

All other work should be deferred until you receive additional callbacks.

Once the aircraft itself has been loaded, your script will receive a number of major callbacks - these callbacks run a function in your script if a function with the matching name is found. You do not have to implement a function for every major callback, but you will almost certainly want to implement at least some of them.

Besides major callbacks, one other type of function in your script will run: when you create or modify commands and when you create writeable datarefs, you provide a Lua function that is called when the command is run by the user or when the dataref is written (e.g. by the panel ro a manipulator).

MAJOR CALLBACKS

The non-function part of your script is run top-to-bottom when your airplane is loaded - this is the right time to create new commands and datarefs.

Most work is done in response to major callbacks - functions that are automatically called in your script during X-Plane's execution. Those names are:
aircraft_load - run once when your aircraft is loaded. This is run after the aircraft is initialized enough to set overrides.
aircraft_unload - run once when your aircraft is unloaded.
flight_start - run once each time a flight is started. The aircraft is already initialized and can thus be customized. This is always called after aircarft_load has been run at least once.
flight_crash - called if X-Plane detects that the user has crashed the airplane.
before_physics - called every frame that the sim is not paused and not in replay, before physics are calculated
after_physics - called every frame that the sim is not paused and not in replay, after physics are calculated
after_replay - called every frame that the sim is in replay mode, regardless of pause status.

These functions take *no* arguments.

GLOBAL VARIABLES PROVIDED BY XLUA

Xlua provides two global variables that are available from all callbacks:

SIM_PERIOD - this contains the duration of the current frame in seconds (so it is alway a fraction). Use this to normalize rates, e.g. to add 3 units of fuel per second in a per-frame callback you'd do fuel = fuel + 3 * SIM_PERIOD.

IN_REPLAY - evaluates to 0 if replay is off, 1 if replay mode is on.

DATAREFS

Datarefs work just like variables - that is, once you make a variable from a dataref, you just use it and change it like a regular lua variable. Lua supports 3 kinds of datarefs:

- Numbers. Integer, float and double datarefs in the X-Plane SDK all just act as "numbers" in Lua. Furthermore, if you set up a float or int array dataref but include the subscript in the name, e.g. "sim/flightmodel2/wing/aileron[2]" then you get a number-type dataref.
- Arrays. Integer and float arrays that do not have the subscripts in the name act like arrays (tables, really) in Lua. Besides supporting array access with [0] etc. the property .len tells the number of elements in the dataref. WARNING: plugin datarefs are ZERO based, so if len == 2 then only [0] and [1] are valid.
- Strings. Raw data arrays in the SDK are treated as strings in Lua.

x = find_dataref("dataref_name")

This creates a new lua variable (x) that is mapped to the named dataref that already exists. X's type (string, number or table) depends on the type of the named dataref. Use this to map X-Plane's datarefs into your plugin.

X = create_dataref("name","type")

This creates a new read-only dataref "name". Type must be one of:
"number" - creates a single number dataref.
"string" - creates a string dataref.
"array[5]" - creates an array - you specify the number of elements.

Note that this dataref **can** be changed by your script, e.g. with x = 5. But it cannot be changed by X-plane or other plugins.

X = create_dataref(“name”,”type”,my_func)

This creates a writeable dataref; the function “my_func” (which takes no arguments) is called each time a plugin OTHER than your script writes to the dataref. It is okay to have the function do nothing.

COMMAND CLASSES

Commands are objects that you can act on. You start by finding or created a command, e.g.

pause_cmd = find_command("sim/operation/pause_toggle")

You can then act on it, e.g.

pause_cmd:once()

Will pause the sim. Command objects have three methods:
once() - runs the command exactly once.
start() - starts holding down the command.
stop() - stops holding down the command.

Make sure every call to start() is balanced by a stop()!!!

You can also create your own commands:

C = create_command(name, description, function)

The name is the permanent name, e.g. plugin/engine/do_rocket_goo
The description is a human readable name, e.g. “This fires the photon torpedos”.
The function is a lua callback that is called when the command is run:

function command_handler(phase, duration)
end

The phase is an integer that will be 0 when the command is first pressed, 1 while it is being held down, and 2 when it is released. For any command invocation, you are guaranteed exactly one begin and one end, with one or more “held down” in the middle. But note that if the user has multiple joysticks mapped to the command you could get a second down while the first one is running.

The duration is how long the command has been held down in seconds, starting at 0.

create_command returns a command object but you can ignore that object if you just need to make a command and not run it yourself.

You can customize existing commands in two ways:

C = replace_command(name, handler)

This takes an existing command (e.g. one of X-Plane's commands) and **replaces** the action the command does with your handler - the handler has the same syntax as the custom command handler - it takes a phase and duration.

C = wrap_command(name, before_handler, after_handler)

This takes an existing command (e.g. one of X-Plane's commands) and installs two new handlers - the before handler runs before X-Plane, and the after handler runs after. You can use this to "listen" for a command and do extra stuff without losing X-Plane's capabilities. You must provide both functions even if one does nothing.

TIMERS

You can create a timer out of any function. Timer functions take no arguments. Several commands are provided:

run_at_interval(func,interval)

Runs function every interval seconds, starting interval sections from the call.

run_after_time(func,delay)
Runs func once after delay seconds, then timer stops.

run_timer(func,delay,interval)

Runs function after delay seconds, then every interval seconds after that.

stop_timer(func)

This ensures that func does not run again until you re-schedule it; any scheduled runs from previous calls are canceled.

is_timer_scheduled(func)

This returns true if the timer function will run at any time in the future. It returns false if the timer isn't scheduled or if func has never been used as a time.

DEBUG FACILITIES

1. Script reload
2. Lua output
3. Interactive command line

TABLE DESIGN

The globals table that the user script runs in (e.g. using setfenv on the hunk returned from loadfile before running it) contains meta-methods in its meta-table to look-up data in _two_ sub-tables:

1. A variables table contains direct raw data, used for all newly added variables, so, like, globals work.
2. A properties table contains property objects, each of which contain a read and write function. The property objects can optionally also contain internal state.
3. The global namespace has a global namespace method to create a property from a table. By “installing” the property into the properties table, we get operator= control routed properly.
4. The global namespace can also look to its parent scope as last resort for already-defined variables.

DATAREFS

- Script supports string, float, and array datarefs.
- Strings and floats backed by properties.
- Arrays are direct variables, using meta-tables to do item access.
- Float/int is type-coerced in the runtime.

COMMANDS

- Command object is a thin wrapper to the C API

NATIVE API

NATIVE API - the obj system is a thin lua wrapper around straight C function calls:
dref = XPLMFindDataRef(name)
    Note: if name contains an array, automatic index mapping takes effect
value = XPLMGetData(dref)
XPLMSetData(dref,value)
    Note: automatic type coercion between int, float double, byte array etc.
value = XPLMGetDatav(dref, idx)
XPLMSetDatav(dref,idx,value)
    Note: automatic type coercion between int array, float array
dref = XPLMCreateDataref(name, type,read-write,notifier)
    Type is a string, e.g. “float”, “int[20]”, “string”
    Read-write is a string, e.g. “read-only”, “read-write”