Some simple explanations for beginners

Carsten Lynker

December 2013

1 First steps programming FlyWithLua

This is more a *cooking book* than a real in-deep tutorial. It will give some fresh ideas to beginners, helping to find the golden thread to run successful through a Lua scripting adventure. So don't forget to study the manual, especially as a reference to all the functions used in this book.

1.1 When are my scripts running?

All scripts written in Lua (the file name ends at ".lua" and the file is inside the "FlyWithLua/Scripts" directory) are executed once if you start X-Plane, if you change your plane or location - or if you force FlyWithLua to reload all scripts. This can be done by clicking the menu "Plugins" -> "FlyWithLua" -> "Reload all Lua script files".

1.2 A first Lua file

Start X-Plane and start a text editor (like Notepad++ or VIM). Then write the file "hello world.lua" into the "Scripts" directory. Fill the file with these Lua code:

```
logMsg("Hello World!")
```

Save the file with your editor and switch to X-Plane. Click on "Special" -> "Show Dev Console". You see some logging info from X-Plane. Now force FlyWithLua to reload your script and watch the developer console. You will see this (and other code from FlyWithLua):

```
FlyWithLua Info: Start loading script file Resources/plugins/FlyWithLua/Scripts/hello world.
Hello World!
FlyWithLua Info: Finished loading script file Resources/plugins/FlyWithLua/Scripts/hello wor
```

You can see, your little script does its work once during its run.

1.3 Setup start parameters

You can use this behavior to setup some start conditions. This can be done, as FlyWithLua automatically runs the scripts when you change your plane or location. So if you want to always start with cold&dark setting in X-Plane, but have the orange beacon on by default, write a script like this:

```
command_once("sim/lights/beacon_lights_on")
```

You can still use your beacon light switch, as the script is executed once and the command is executed only once as well. The Lua function "command_once()" will call an X-Plane command. You will see all the commands, if you click on "Settings" -> "Joystick & Equipment" -> "Buttons: Adv".

1.4 DataRefs

The most exiting thing programming with FlyWithLua is, that you can read and write DataRefs. These are X-Plane's internal variables, representing much more than the commands can reach. Take a look at this web page:

http://www.xsquawkbox.net/xpsdk/docs/DataRefs.html

So let's try the same procedure as before. Change your little script to this:

```
dataref("Beacons", "sim/cockpit/electrical/beacon_lights_on", "writable")
Beacons = 1
```

And again, all your action in X-Plane starts with beacon lights on (to be correct, it will start the beacon when you switch the battery on). You must be careful when using a DataRef. Not all of them can be set by a plugin. If they are writable or not can be found in the fourth column of the official DataRef listing.

1.5 Event driven programming

You can't only do things once when the script loads. There are Lua functions, that allows to react on simulator events. For example when the mouse scroll wheel is moved. Write a new script "no more zooming.lua" and fill it with this code:

```
do_on_mouse_wheel("RESUME_MOUSE_WHEEL = true")
```

Hey, this disables the zoom on the mouse wheel. In fact this little piece of software, a tiny one-liner, does nothing on a mouse wheel event, but resumes the event. So the SDK's event handler stops action and scroll wheel info never gets to X-Plane.

1.5.1 Doing some action on wheel movements

As killing the zoom only is a little bit boring, we will add some action. The mouse wheel movement is represented by the variable MOUSE_WHEEL_CLICKS. We can use this info to move the trim wheel with the mouse wheel. Use this code:

```
dataref("xp_elv_trim", "sim/flightmodel/controls/elv_trim", "writable")

function set_trim_by_mouse_wheel()
    xp_elv_trim = xp_elv_trim - MOUSE_WHEEL_CLICKS * 0.0025
    if xp_elv_trim > 1.0 then
        xp_elv_trim = 1.0
    end
    if xp_elv_trim < -1.0 then
        xp_elv_trim = -1.0
    end
    RESUME_MOUSE_WHEEL = true
end

do_on_mouse_wheel("set_trim_by_mouse_wheel()")
```

This is a little bit more complex. We first define a DataRef connection to a variable, that allows writable access. Then we define a function, to not have too many code to write inside the do_on_mouse_wheel() function. This is not really a problem, it's more a sort of ugly code.

The function sets the DataRef with the MOUSE_WHEEL_CLICKS variable as a multiplier. As this can result in forbidden values (range from -1.0 to +1.0), we make sure that correct values are given back to X-Plane with two if statements.

1.5.2 Graphical output

We want to see the movement of the trim when we use the mouse wheel. To show something with Lua, it must be done with the do_every_draw() function. So we add the code above:

```
function draw_trim_info()
    XPLMSetGraphicsState(0,0,0,1,1,0,0)
    glColor4f(1, 1, 1, 0.5)
```

```
    glRectf(100, 100, 110, 300)
    glBegin_LINES()
    glVertex2f(90, 200 + xp_elv_trim*100)
    glVertex2f(120, 200 + xp_elv_trim*100)
    glEnd()
end

do_every_draw("draw_trim_info()")
```

This will show an info onto the screen. We use OpenGL functions, that are implemented into FlyWithLua's Lua dialect. But one thing is still stupid, the info is shown all the time. We want it to disappear after 5 seconds. Let's modify the code again:

```
dataref("xp_elv_trim", "sim/flightmodel/controls/elv_trim", "writable")

local show_trim_info_until = 0

function set_trim_by_mouse_wheel()
    xp_elv_trim = xp_elv_trim - MOUSE_WHEEL_CLICKS * 0.0025
    if xp_elv_trim > 1.0 then
        xp_elv_trim = 1.0
    end
    if xp_elv_trim < -1.0 then
        xp_elv_trim = -1.0
    end
    RESUME_MOUSE_WHEEL = true
    show_trim_info_until = os.clock() + 5
end

do_on_mouse_wheel("set_trim_by_mouse_wheel()")

function draw_trim_info()
    if os.clock() < show_trim_info_until then
        XPLMSetGraphicsState(0,0,0,1,1,0,0)
        glColor4f(1, 1, 1, 0.5)
        glRectf(100, 100, 110, 300)
        glBegin_LINES()
        glVertex2f(90, 200 + xp_elv_trim*100)
        glVertex2f(120, 200 + xp_elv_trim*100)
        glEnd()
    end
end

do_every_draw("draw_trim_info()")
```

Now we use the Lua function os.clock() to get the time in seconds the simulator is running. We add five to this value, to get the time in five seconds, and store this value in a local variable. The variable must be set (and defined as local) *before* the function to display it is defined and connected to the simulator's drawing loop. Else the first run in the drawing loop crashes the script as the **variable** is unknown.

As we didn't know if an other script set's the OpenGL graphic state wrong for our output, we set it to the default value with the XPLMSetGraphicsState() function. If not, it can end in some strange behavior, if am other script disables alpha for example.

1.6 Understanding custom commands

A very important feature of FlyWithLua are custom commands. Commands are procedures offered by X-Plane or additional elements like a plane you are using.

Commands have unique names. All commands offered by X-Plane starts with `sim/` and they are visible in the joystick or keyboard configuration menu by default. Just click on `Settings -> Joysticks & Equipment` and choose the tab `Buttons: Adv`. Then press a joystick button, that is not assigned to a command. You can see, that it is impossible to have no assignment. Instead the button is assigned to `sim/none/none`, as you can check in the little dark info area in the middle of the window and top right in a textbox, where you can copy&paste the command's name (but you can't edit it's name).

If you want access a command not starting with `sim/`, you will have to click on the little transparent square left of the textbox. Then you can choose the *first path element* – if you understand a command's name as a name with path, like a filename.

Every command has a unique name and a (not forced unique) description. The description is shown in the dark info area beneeth the command's name.

All commands provided by other software will have to use a different beginning than `sim/`. These commands are called custom commands. Many free or payware addons like additional planes deliver custom commands.

1.6.1 Creating a custom command

Now we will create a custom command to increase the OBS direction. Let's write a little script and name it `Custom_Command_Test.lua`.

```
require("radio")

create_command("FlyWithLua/testing/increase_OBS",
```

```
                "This command increases the OBS value by one degree.",
                "OBS1 = OBS1 + 1",
                "",
                "")
```

This is working, but it still has a lot of disadvantage in it.

We will examine the code first. In line no. 1 the script loads the module *radio*, if it isn't present (otherwise it will do nothing). This will define some DataRef variables like OBS1 we need here.

Much more important are line 3 to 7, the `create_command()` function provided by FlyWithLua. This function needs five arguments. All arguments need to be strings (Text surrounded by quotes).

The first argument is the unique command name. in this example, we name it `FlyWithLua/testing/increase_OBS`. Be aware of nameming it `sim/...` or FlyWithLua will stop with an error message.

The second argument is the description of the command. Make sure to provide a short and precise description, as the user will only have these descrition (and the name) to guess what the command is doing.

The third argument is a string containing Lua code. This code is executed once, when the command starts. You can see that the example command will increase the value by one every time you press the assigned button.

The fourth and fifth argument are strings containing Lua code, that is executed while the button (or key) is hold down (fourth argument), or when the command ends (button or key is released). In this first example both are empty. You must give empty strings to the function, if you want to do nothing, or you will get an error message about missing arguments.

The last two arguments are more important as you might guess. Let's play around with the example in deep. If you press the assigned button again and again, you will get higher values than 360°. This is not what you want. If you reach 360°, it should swap to 0° to make a clean run around the OBS instrument. Values above 360° are not usefull.

So we can use the last argument, to clean up when the command stops.

```
require("radio")

create_command("FlyWithLua/testing/increase_OBS",
                "This command increases the OBS value by one degree.",
                "OBS1 = OBS1 + 1",
                "",
                "OBS1 = OBS1 % 360")
```

The % does a modulo division and gives back the rest of the division, so 360° will result in 0° and the next turn around the instrument can start.

Okay – so far so cool, but what about clicking and clicking and clicking when changing bigger values. The first idea could be using the fourth argument. It will be executed every frame as long as the assigned button or key is hold down. So try this:

```
require("radio")

create_command("FlyWithLua/testing/increase_OBS",
               "This command increases the OBS value by one degree.",
               "",
               "OBS1 = OBS1 + 1",
               "OBS1 = OBS1 % 360")
```

Is this super clever? No!

You loose the ability to make small steps, as you can't press as short as only one frame. You will have to implement a better solution.

```
require("radio")
local obs_up_time = 1
local ops_up_value = 1

create_command("FlyWithLua/testing/increase_OBS",
               "This command increases the OBS value by one degree.",
               "obs_up_time = os.clock()\n
                obs_up_value = OBS1",
               "OBS1 = 5 * (os.clock() - obs_up_time) + obs_up_value",
               "OBS1 = OBS1 % 360")
```

Wow, what's that? First we define two local variables. This makes a script more fast and less conflicty to other scripts. We use 1 as a random value, as a value is necassary, but not known at this moment.

Then we use the Lua function os.clock() (yes, it's pure Lua, not FlyWithLua). The function `os.clock()` gives back the time in seconds the Lua engine is running.

When the command starts, we will store the actual value and the time into the local variables. Please note the special character backslash with n inside the string. It will make a new line character to seperate the two lines of code. If you let it away, FlyWithLua will promt an error message and stop working.

Then, with every frame the button or key is hold down, we will increase the value of OBS by the time in seconds multiplied by 5. You can change the value of 5 to make it fit your own needs.

Five per second is too slow? It can get even better:

```
require("radio")
local obs_up_time = 1
local ops_up_value = 1

create_command("FlyWithLua/testing/increase_OBS",
               "This command increases the OBS value by one degree.",
               "obs_up_time = os.clock() + 2\n
                obs_up_value = OBS1\n
                OBS1 = obs_up_value + 1",
               "if os.clock() > obs_up_time then\n
                  OBS1 = 10 * (os.clock() - obs_up_time) + obs_up_value\n
                end",
               "OBS1 = OBS1 % 360")
```

Now we have an if statement to check if two seconds are passed. If so, we will increase with a value multiplier of 10 (or whatever you want). The two seconds are added when the command starts (the third argument). This helps to reduce the math needed.

1.7 Smoothing an axis

If you have an axis that delivers jumping values, it may disturb your X-Plane experience. So it's time to smooth the input. Let's see how.

1.7.1 Looking for the real values

First we want to see the real values coming from the hardware. Say we have an axis that is jumping. The axis no. is 12 (see X-Plane's menu for joystick configuration to get the right axis no.).

We will use this code to display the real values. A visible string position is more convenient that looking at naked values in DRE.

```
dataref("real_axis_12", "sim/joystick/joystick_axis_values", "readonly", 12)

-- show the axis
do_every_draw('draw_string(1500 * real_axis_12, 20, "hardware value", "red")')
```

In the code we position the text to an x-value of 0 up to 1500, as the values delivered by X-Plane are float values from 0.0 to 1.0. You may change the value of 1500 depending on your X-Plane window's width.

1.7.2 Simple Moving Avarage

To smooth the signals coming from the axis, we use a simple moving avarage calculation. The following code should be added:

```
local values_axis_12 = { }
for i = 1, 10 do
    values_axis_12[i] = real_axis_12
end
axis_12 = real_axis_12

function calculate_axis_12()
    axis_12 = real_axis_12
    for i = 2, 10 do
        axis_12 = axis_12 + values_axis_12[i]
        values_axis_12[i-1] = values_axis_12[i]
    end
    values_axis_12[10] = real_axis_12
    axis_12 = axis_12 / 10
end

do_every_frame("calculate_axis_12()")

-- show the smooth value
do_every_draw('draw_string(1500 * axis_12, 40, "smoothed value", "green")')
```

On the screen we see a more stable text "smoothed value", as it's position is calculated as the mean of the last 10 values. If you use one pysics calc run on one graphical run, then this is about half a secound when FPS is 20. If you choose more physic calculations or use a faster hardware (CPU/GPU), then the time period for the mean calculation may be shorter.

1.7.2.1 Creating a helper module As the method above isn't very smart code, we now create a module to be used by a script. The module file must start with this line:

```
module(..., package.seeall);
```

We place the module file into the directory `.../FlyWithLua/Modules/` and name it `SMA_smoothing.lua` (a pure Lua module ends in ".lua").

As Lua has no classes we can create instances from, we need a function that creates functions by executing strings. Call it some kind of pure-mans-OOP.

```
function create_SMA(axis_number, samples)
    -- make samples an optional argument
    samples = samples or 10

    -- make sure that axis_number is a string
    axis_number = tostring(axis_number)

    -- create the code
    local code = 'dataref("real_axis_' .. axis_number
    code = code .. '", "sim/joystick/joystick_axis_values", "readonly", ' .. axis_number ..
    code = code .. "local values_axis_" .. axis_number .. " = { }\n"
    code = code .. "for i = 1, " .. samples .. " do\n"
    code = code .. "    values_axis_" .. axis_number .. "[i] = real_axis_" .. axis_number ..
    code = code .. "end\n"
    code = code .. "axis_" .. axis_number .. " = real_axis_" .. axis_number .. "\n\n"
    code = code .. "function calculate_axis_" .. axis_number .. "()\n"
    code = code .. "    axis_" .. axis_number .. " = real_axis_" .. axis_number .. "\n"
    code = code .. "    for i = 2, " .. samples .. " do\n"
    code = code .. "        axis_" .. axis_number .. " = axis_" .. axis_number .. " + values
    code = code .. "        values_axis_" .. axis_number .. "[i-1] = values_axis_" .. axis_n
    code = code .. "    end\n"
    code = code .. "    values_axis_" .. axis_number .. "[" .. samples .. "] = real_axis_" .
    code = code .. "    axis_" .. axis_number .. " = axis_" .. axis_number .. " / " .. sampl
    code = code .. "end\n\n"
    code = code .. 'do_every_frame("calculate_axis_' .. axis_number .. '()")\n'

    -- execute the code
    assert(loadstring(code))()
end
```

You find the code as an example delivered with FlyWithLua.

The script itself shrinks to this:

```
require("SMA_smoother")

SMA_smoother.create_SMA(12, 10)

-- show the axis
do_every_draw('draw_string(1500 * real_axis_12, 20, "hardware value", "red")')

-- show the smooth value
do_every_draw('draw_string(1500 * axis_12, 40, "smoothed value", "green")')
```

The modules's function `create_SMA()` will create the dataref variable `real_axis_` plus the number of the axis given by the first argument. It will also

create the function to calculate the smoothed value and calls it every frame loop. So everything you have to do after this line:

```
SMA_smoother.create_SMA(12, 10)
```

Is to use the value of the new created variable `axis_` plus the number, as shown in the last line of the example script.