LuaUsage

This page documents the principles of lua developement.

- darktable uses lua version 5.2, see Lua reference manual
- for a documentation of the lua API see LuaAPI
- for user provided scripts see LuaScripts14 (darktable version 1.4)
- for scripts for version 1.6 and above, see the separate github repository

A note about beta

The lua API is still in beta. This has multiple implications you should be aware of when writing lua code

- The API might change without notice. In particular If I find out that the API is not practical.
- The API is incomplete. If you find something missing please bug me (boucman) on IRC, the mailing list or open a feature request.
- Please assign all lua bugs/FR/PR to boucman
- The general limitation of the lua API is the lighttable view. In other word we aim at being able to do with lua everything that can be done in the lighttable, but anything that can only be done in darkroom mode is off-limit

Basic principles

At startup, darktable will run two lua scripts

- a script called *luarc* in */usr/share/darktable*
- a script called *luarc* in the user's configuration directory

This is the only time darktable will run lua scripts by itself. Scripts can register callbacks to perform actions. See the corresponding section in LuaAPI

Debugging lua code

enabling log

the first step to debugging lua is to enable lua logging. this is is done by enabling the lua logdomain with the command line argument -d lua

- messages printed by a lua script using darktable.print will be visible on the standard out
- in case of lua error, the whole traceback (and not just the error message) will be printed to the console

analyzing data

Darktable provides some debugging helpers. Below is an example

```
dt_debug = require "darktable.debug"
dt_debug.debug.debug = true

print(dt_debug.dump(_G,"Global environement")
```

the main function provided is darktable.debug.dump(object,name) This function returns a string that describes object. The object can be anything and this function will use its knowledge of the DT API to be smart about the type of objects

the boolean variable darktable.debug.debug is a boolean initialized to false. When this variable is true, dt_debug.dump will also dump metatables for objects. You usually only want to see this to debug the internals of darktable's lua API

debugging unnprotected calls

when experimenting with lua, darktable might crash with the following message :

```
PANIC: unprotected error in call to Lua API (some message here)
```

this is always a bug in the lua API. Please open a bug report and attach the script that caused it to the bug report

Handling scripts

darktable will look for lua modules in the system provided lua path, but it will also look into the following places

- $(CONFIG_DIR)/lua/?.lua
- $(USER_DIR)/lua/?.lua

In other words, if you place a file called c.lua in the directory ~/.config/darktable/lua/a/b/ then require "a.b.c" will find your script.

The normal way to install a script is to copy it in ~/.config/darktable/lua/ then adding a require line to the file ~/.config/darktable/luarc

Yielding from lua code

Lua code in DT is allowed to yield (see the yield system call in the lua documentation)

Lua code that yields allows other lua callbacks to run, be aware of that

It is important to yield whenever you have some code that you expect to block to allow other lua code to run.

In particular you should always yield when calling an external program, particularly if it's a long, image processing task.

see LuaAPI for details about the syntax of the yield call in Darktable