Programming in Lua – Embedding Lua

Fabio Mascarenhas

http://www.dcc.ufrj.br/~fabiom/lua

Configuration language

- Lua started as a *configuration language*, a nice language for writing configuration files, and this is still one of its main uses
- In this unit, we will take a simple game written using the SDL library where everything is fixed at compile time, and gradually expose the game settings to a Lua configuration file
- Our configuration file will have not only simple atomic values
- We can use tables to describe more complex initialization data
- We can use functions to actually implement part of the game logic in Lua

Reading simple scalars

- Our game has several simple settings in a include file that would be good to expose to the configuration file:

```
#define WIN_WIDTH 800
#define WIN_HEIGHT 800
#define PADDLE_SPEED 7   /* Number of pixels paddle moves per frame. */
#define BALL_SPEED 6   /* Number of pixels ball moves per frame */
```

- The configuration file will look like this:

```
-- window size
width = 1000
height = 600
-- speeds
paddle = 10
ball = 10
```

- On startup, the game will try to load a “config.lua” file

Loading a simple config file

- Assuming we have moved the configuration settings from macros to global variables, we can use the following functions to load the configuration file:

```
void load_config() {
    lua_State *L = luaL_newstate();   /* opens Lua */
    luaL_openlibs(L);                 /* opens standard libraries */
    if(luaL_loadfile(L, "config.lua") || lua_pcall(L, 0, 0, 0)) {
        fprintf(stderr, "error in config file: %s", lua_tostring(L, -1));
        goto close;
    }
    load_ivar(L, "width", &WIN_WIDTH);
    load_ivar(L, "height", &WIN_HEIGHT);
    load_ivar(L, "paddle", &PADDLE_SPEED);
    load_ivar(L, "ball", &BALL_SPEED);
close:
    lua_close(L);
}
```

- The heavy lifting of bringing the values from Lua to the C side is done by the `load_ivar` function, in the next slide

Reading global variables

- Function load_ivar uses lua_getglobal to push the value of a global variable, then checks its type and extracts the value if it is a number:

```c
void load_ivar(lua_State *L, const char *name, int *var) {
    lua_getglobal(L, name);
    if(lua_type(L, -1) != LUA_TNUMBER)
        fprintf(stderr, "'%s' should be a number", name);
    else
        *var = lua_tointeger(L, -1);
}
```

- The error handling in this application is very simple: we just log the problems we had when reading the configuration file

- Any unset variables will keep their default values

More configuration for free

- We do not load the configuration file in a sandbox, it has access to all Lua standard libraries, and even external libraries (using `require`)

- This means the user can write a more sophisticated configuration without needing to change a single line of code:

```
local platform = require "platform"
local W, H = platform.screen_size()

width = W * 0.8
height = H * 0.8
paddle = height / 90
ball = width / 150
```

- If we have a library that lets us query the screen dimensions, we can set the width and height to cover a part of the screen; we also can let the speed of the paddle be whatever is necessary to cross the height of the screen in 90 frames, and the speed of the ball what is necessary to cross the width in 150 frames

Using tables

- We have just four configuration variables, but suppose we now want to add several more: the RGB values for the color of each paddle

- Having one global variable for each individual value would be cumbersome! We can use tables to better structure the configuration file:

```
window = {
  width = 1000,
  height = 600
}
paddle = {
  speed = window.height / 90,
  left_color = { r = 255, g = 0, b = 0 },
  right_color = { r = 0, g = 0, b = 255 }
}
ball = {
  speed = window.width / 150,
}
```

Manipulating tables

- The `lua_gettable` function indexes a table; it takes the stack index of the table, pops the key, and then pushes the value:

```
void load_ifield(lua_State *L, int index, const char *name, int *var) {
    lua_pushstring(L, name);
    lua_gettable(L, index);
    if(lua_type(L, -1) != LUA_TNUMBER)
        fprintf(stderr, "'%s' should be a number", name);
    else
        *var = lua_tointeger(L, -1);
    lua_pop(L, 1);
}
```

table

field
name

C variable

- Indexing a table with a string is so common that there is a specialized function for that, `lua_getfield`:

```
lua_getfield(L, index, name);
```

instead of

```
lua_pushstring(L, name);
lua_gettable(L, index);
```

Setting fields

- We can offer some pre-defined colors to the user, populated from a static structure:

```
struct Color {
    char *name;
    unsigned char red, green, blue;
} colors[] = {
    {"WHITE", 255, 255, 255},
    {"RED",   255,   0,   0},
    {"GREEN",   0, 255,   0},
    {"BLUE",    0,   0, 255}
    {NULL,      0,   0,   0}
};
```

- `lua_settable` is analogous to `lua_gettable`: it takes the index of the table and pops first the value and then the key (so you should push the key and then the value before calling `lua_gettable`)

- There is also `lua_setfield`, specialized for string fields, and it only pops the value

Creating a color

- Creating a new color is straightforward:

```
void new_color(lua_State *L, struct Color *color) {
    lua_newtable(L);
    lua_pushinteger(L, color->red);
    lua_setfield(L, -2, "r");
    lua_pushinteger(L, color->green);
    lua_setfield(L, -2, "g");
    lua_pushinteger(L, color->blue);
    lua_setfield(L, -2, "b");
    lua_setglobal(L, color.name);
}
```

- Now we just need to add two lines to load_config, right before loading the configuration file:

```
for(int i = 0; colors[i].name != NULL; i++)
    new_color(L, &colors[i]);
```

Calling Lua functions

- Suppose we want to add an option to the paddle configuration variable, to replace either the left or the right paddle (or both!) by a “robot”, instead of user input
- We can model this as a Lua function in the configuration script:

```
paddle = {
  speed = window.height / 90,
  left_ai = function (ball_x, ball_y, paddle_y, direction)
              if direction < 0 then
                if paddle_y > ball_y then
                  return -1     → move up
                else
                  return 1     → move down
                end
              else
                return 0
              end
            end
}
```

lua_pcall

- To call a function, we first have to push it and any arguments we want to pass to the function
- Then we can use `lua_pcall`: the second argument to `lua_pcall` is the number of arguments we are passing, and the third argument is how many results we want from the function
- If the function succeeds the returned values (adjusted to how many you want) will be on the top slots of the stack
- If the number of results is `LUA_MULTRET` `lua_pcall` will push all of the return values; in this case it is a good idea to save the stack top before `lua_pcall`, so you know how many results the function has!

“lua_xpcall”

- The last arguent of lua_pcall lets it act like xpcall instead of pcall; if it is not 0 then it should be the stack index of an error handler function, in the manner of xpcall

- Like xpcall, lua_pcall will also catch errors in the error handler function; the error code returned by lua_pcall tells what happened:

  - LUA_ERRRUN: a regular error in the called function

  - LUA_ERRERR: error in the error handler

  - LUA_ERRMEM: memory allocation error

  - LUA_ERRGCMM: error in a *finalizer*, we will cover the later

Quiz

- Suppose we want to add player names to the configuration file:

```
names = { "John", "Mary" }
```

- What is the problem with the `load_player` function below? How can we fix it?

```
void load_player(lua_State *L, int index, int player, const char **var) {
    lua_pushinteger(L, player);
    lua_gettable(L, index);
    if(lua_type(L, -1) != LUA_TSTRING)
        fprintf(stderr, "player %d should be a string", player);
    else
        *var = lua_tostring(L, -1);
    lua_pop(L, 1);
}
```

→ string should now be at -1

→ pop the string

→ strncpy(*var, lua_tostring(L, -1));