



Scripting Programs with Lua

Christian Vogler

Gallaudet Research Institute



What is Scripting?

- Scripting languages are programming languages, with three key characteristics:
- Very high level
 - No fiddling with bits, bytes, and pointers
- Interpreted
 - Instant “gratification,” interactive development
- Dynamically typed
 - No need to declare variables, types are inferred from what you actually do with the variable

2



Demonstration of Scripting

- Show scripting demonstrations

3



Scripting – why do it?

- What is scripting useful for?
- Quickly allows you to change behavior of complex programs
 - “Glue” program parts together
- Makes you much more productive
 - Because of high-level and dynamic typing
- No need to wait for long compiles
- More flexible than configuration files
 - With scripting languages, you don’t need to anticipate every possible option

4



The Lua Scripting Language

- “Lua” = Portuguese word for “moon”
- Developed in Brazil, originally on a contract with Petrobras, the national oil company
- Free software (MIT license)
 - (Don't tell Darl, or SCO might get the idea that it is somehow a derivative work of System V!)

5



YASL?

- Yet Another Scripting Language???
- After all, we have Perl, Python, Guile, Ruby, Tcl, ...
- True, but choose the best tool for the job
- Lua shines at embedding, gluing together other program parts, and data description

6



Advantages of Lua

- Lua is:
 - Small
 - < 500 KB of source code, can be further stripped
 - Written in pure ANSI C
 - Compiles out of the box everywhere
 - Fast
 - At the top of the pack among scripting languages
 - Easy to learn and use
 - Simple syntax, a lot of nonprogrammers in the game industry use it
 - Powerful
 - Mechanisms for extension

7



Some projects that use Lua

- Games
 - Baldur's Gate, Escape from Monkey Island, MDK, Grim Fandango
 - Mostly used to control AI and behavior
 - This is where many designers (non-programmers) also use Lua
- Tools
 - APT-RPM, Clan Lib
 - Extend tool functionality via scripting

8



Fundamental Lua Datatypes

- Numbers, strings

```
> print(1 + 2)
3

> print('Hello World')
Hello World

> print('Hello' .. ' World')
Hello World

> print('This is a string ' .. tostring(123))
This is a string 123
```

9



Variables

- Variable assignment and type conversion

```
> x = 3
> print(x)
3
> y = 4
> print (x + y)
7

> z = 'A string: '
> str = z .. y
> print(str)
A string: 4
```

10



Tables

- The basic data structure are tables
- Can be used as arrays or lists:

```
> t = { 1, 1, 2, 3, 5, 8, 13, 21 }
> print(t[1])
1
> print(t[4])
3
> print(table.getn(t))
8
> print(t[0])
nil
```

11



Tables, continued

- Tables are also dictionaries

```
> dict = {}
> dict['SCO'] = 'Litigious Bastards'
> dict['IBM'] = "Someone you don't mess with"
> print(dict['SCO'])
Litigious Bastards
```

An alternative way to access:

```
> print(dict.IBM)
Someone you don't mess with
```

12



More on table construction

- Often tables are used to describe data or objects
- Then this construction syntax is more convenient

```
> canvas = { width=640, height=480,  
name="main", coordinates={ 100, 200, 300 } }
```

Same as:

```
> canvas = {  
> canvas.width = 640 ...  
> canvas.coordinates = { 100, 200, 300 }
```

13



Iterating on tables

- To do something with all elements in a table, we can use for loops

```
> for i = 1, table.getn(t), 1 do  
>> print(i, t[i])  
>> end
```

```
> for key, value in canvas do  
>> print(key, value)  
>> end
```

14



Functions

- Defining functions

```
function print_table(t, header)  
  local num_elements = 0  
  print(header)  
  for key, value in t do  
    print('entry:', key, 'value')  
    num_elements = num_elements + 1  
  end  
  return num_elements  
end
```

```
> num = print_table(canvas, 'The canvas')  
> print(num)
```

15



If statement

- Conditional execution of a block

```
function sgn(a)  
  if a < 0 then  
    return -1  
  elseif a ~= 0 then  
    return 1  
  else  
    return 0  
  end  
end
```

```
> print(sgn(-4))  
> print(sgn(0))  
> print(sgn(7))
```

16



Metamechanisms

- This is all good and well, but nothing fancy so far
- The real power of Lua comes from extending and customizing the behavior of tables via *metatables*
- Every table has an associated metatable that controls its behavior

17



Observing changes

- A very common application: Detecting when someone wants to create a new entry in a table
- First we create the function that should be called when we detect a new entry

```
function new_entry(t, key, value)
  print('Creating new entry in table', t)
  print('Key: ', key, 'Value:', value)
  -- rawset bypasses the metamechanism, so
  -- this function is not called over and over
  rawset(table, key, value)
end
```

18



Observing changes

- Next, we create a new metatable that overrides the default behavior for creating new entries

```
mt = { __newindex=new_entry }
```

- Now we create a new table, and associate the metatable with it

```
t = {}
setmetatable(t, mt)
```

19



Observing changes

- Let's try if it works

```
> t.name="Observed Table"
Creating new entry in table    table: 0x806afc8
Key:  name    Value:  Observed Table
```

```
> t[1]=2
Creating new entry in table    table: 0x806afc8
Key:  1    Value:  2
```

```
> t[1]=3
```

This time, the function is not called, because the entry with key 1 already exists.

20



Metamechanisms

- There are many other allowed metamethods, among them:
 - Arithmetic operations: +, -, *, /, exponents
 - Comparisons: equality, less than
 - Function calls (a table can be treated as a function)
 - Reading from a table

21



Use of metamechanisms

- They form the basis for extending Lua to fit the application's paradigm
 - e.g. Object oriented, data driven, ...
- They allow binding to and manipulating C/C++ structures and objects
- They allow using tables to control complex operations
 - e.g., use a table field to control the orientation of a robot's arm

22



Lua for data description

- Lua is ideally suited to describe data and configuration files
- In fact, it started this way in 1994
- XML is all the rage, but Lua has three advantages over it:
 - Much easier to use than an XML parser
 - Much smaller than the monstrous XML libraries out there
 - The third one will become apparent in this talk

23



Data description

- We want to describe a 3D model of the human face, as in the demo
- The model consists of “faces” and “nodes”
- Each face specifies a triangle for three particular nodes
- Each node has a position that can be computed

24



Example: a square

```
n0 = Node(0, 0, 0)
n1 = Node(0.5, 0, 0)
n2 = Node(0.5, 0.5, 0)
n3 = Node(0, 0.5, 0)
```

```
Face(n0, n1, n2)
Face(n2, n3, n0)
```

25



Binding to C or C++

- Next, we have to create the two functions for Node and Face in C, which do the actual construction of the object
- We can use the Lua API for this
- Take a look at `lua5-readline/facesnodes.c`
- Compared to using XML, this is pretty painless
- The real meat ...

26



Lua API for C and C++

```
int C_Node(lua_State *L)
{
    int id = next_node_id++;

    /* 1st, 2nd, and 3rd argument to function */
    nodes[id].x = lua_tonumber(L, 1);
    nodes[id].y = lua_tonumber(L, 2);
    nodes[id].z = lua_tonumber(L, 3);

    /* return node id to Lua */
    lua_pushnumber(L, id);
    return 1;
}

lua_register(L, "Node", C_Node);
```

27



Tables in data description

- Of course, we can also pass and manipulate tables from C.
- So, you will often find data description like:
 - `JediKnight{lightsaber='red', speed=moderate, ranking=JustBelowYoda, ... }`
- See the example `tables.c`
- We can also generate bindings automatically with the `tolua` tool

28



More fancy stuff

- But remember, Lua is a full-blown programming language
- Data description files can be simple, like before ...
- ... or they can be full-blown programs
- Back to the square example:
- Let's make a circle instead
- Defining all nodes and faces by hand is nasty
- So we write our description file as a program

29



Generating a circle

- See `circle.lua`

```
radius = 10
step = 2 * math.pi / 10
center = Node(0, 0, 0)
first_node = Node(radius * math.cos(0), radius *
                  math.sin(0), 0)
previous_node = first_node
for angle = step, 2 * math.pi, step do
    local new_node = Node(radius * math.cos(angle),
                          radius * math.sin(angle), 0)
    local new_face = Face(center, previous_node,
                          new_node)
    previous_node = new_node
end
```

30



Data as programs, conclusion

- This is really the whole trick to defining the complex face model with all the deformations
- The basic idea is:
 - Have a base model of the face at rest
 - Define areas that are affected by a particular muscle
 - Such as eyebrows or lips
 - The closer a node is to the center of that area, the more it is affected by the deformation

31



More info

- Christian.Vogler@gallaudet.edu
- <http://www.lua.org>
- <http://www.lua-users.org>

32



Lua vs Python – pro Python

- Python has more high-level data structures, such as lists, arrays, dictionaries, tuples
- Python has explicit syntax for classes and inheritance, whereas Lua users build their own with metamethods
- Python has a huge standard library, whereas for Lua, most of these are 3rd party addons
- Python has a larger user community

33



Lua vs Python – pro Lua

- Lua is smaller, easier to distribute, easier to embed, and easier to port to different platforms
- Lua is generally faster
- Lua uses garbage collection, not reference-counting (so no worries about cyclic data structures)
- Lua has coroutines and is ready for multiple threads, whereas Python uses one Big Lock
- Lua has simple lexical scoping rules and full-blown closures

34