Lua Programming Language

Roberto Beauclair

rbs@impa.br

About Lua

Lua is an embeddable scripting language that aims for simplicity, small size, portability, and performance.

Unlike most other scripting languages, Lua has a strong focus on embeddability, favoring a development style where parts of an application are written in a "hard" language (such as C or C++) and parts are written in Lua.

Currently Lua is used in a vast range of applications, being regarded as the leading scripting language in the game industry.

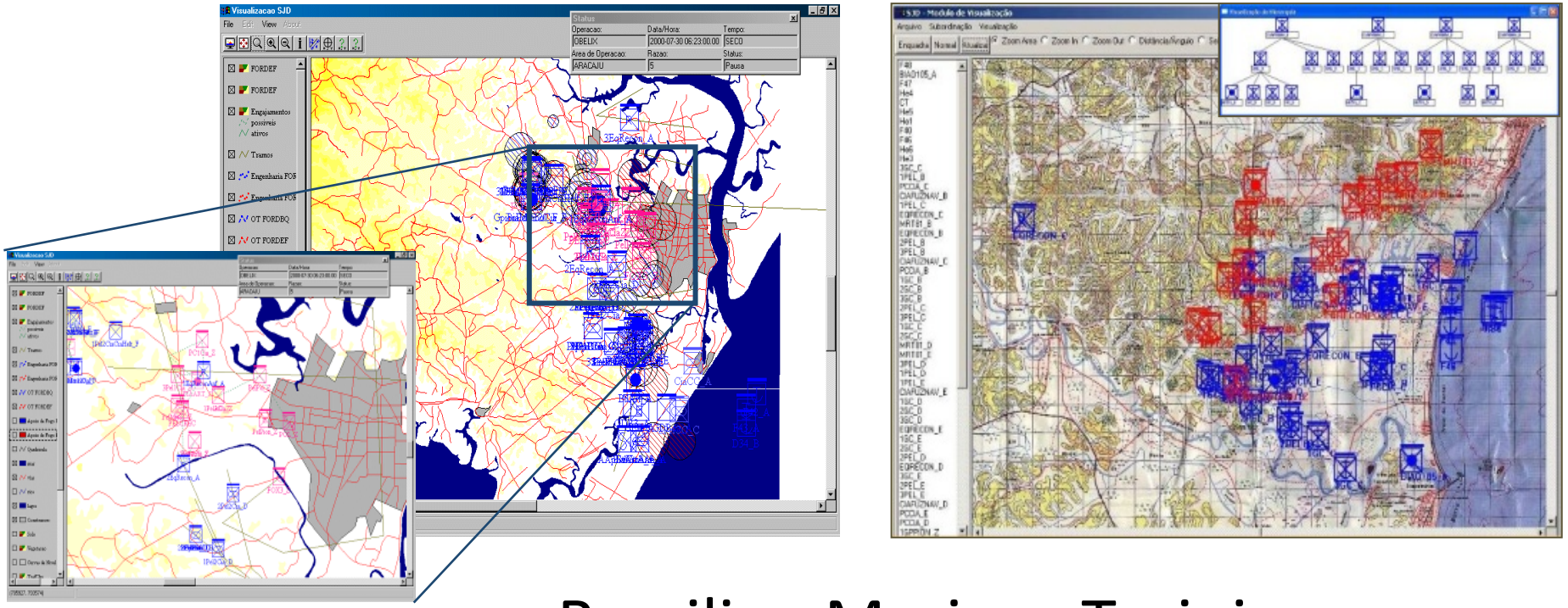
Lua poster


FAST
POWERFUL
LIGHTWEIGHT
EMBEDDABLE
SCRIPTING
LANGUAGE
&VM
Lua
Builds in all platforms with an ANSI/ISO C compiler
Fits into 128K ROM, 64K RAM per interpreter state[1]
Fastest in the realm of interpreted languages
Well-documented C/C++ API to extend applications
One of the fastest mechanisms for call-out to C
Incremental low-latency garbage collector
Sandboxing for restricted access to resources
Meta-mechanisms for language extensions,
e.g. class-based object orientation and inheritance
Natural datatype can be integer, float or double
Supports closures and cooperative threads
Open source under the OSI-certified MIT license
[1] Complete Lua SOC, practical applications in 256K ROM / 64K RAM
Designed, implemented and maintained at the
Pontifical Catholic University of Rio de Janeiro
www.lua.org

Lua is … a scripting language

- Interpreted
  - can run dynamic code
- Dynamically typed
- (Incremental) Garbage collection
- Strong support for strings
  - pattern matching
- Coroutines
- First-class functions
  - lexical scoping
  - proper tail calls

Lua is … a scripting language



Brazilian Marines Training
Simulation System

Lua is … an embeddable language

- Implemented as a library
- Offers a clear API for host applications
  - not only an implementation aspect!
  - design architecture to support it

Lua is … an embeddable language

"63% of the main Lightroom-team authored code is Lua"

Troy Gaul, Adobe

Lua is … an embeddable language

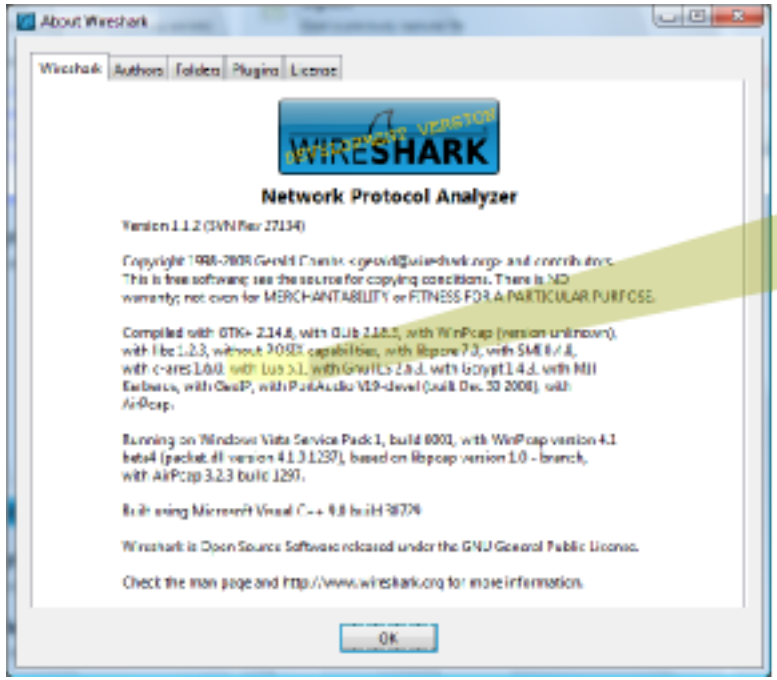
"Oil-reservoir Simulation Management"
Petrobras

Lua is … embedded in a fair share of applications

- Wireshark
- Nmap
- Snort
- Cisco SM
- Barracuda Web Server
- Chipmunk AV controller
- Olivetti printers

and many many others

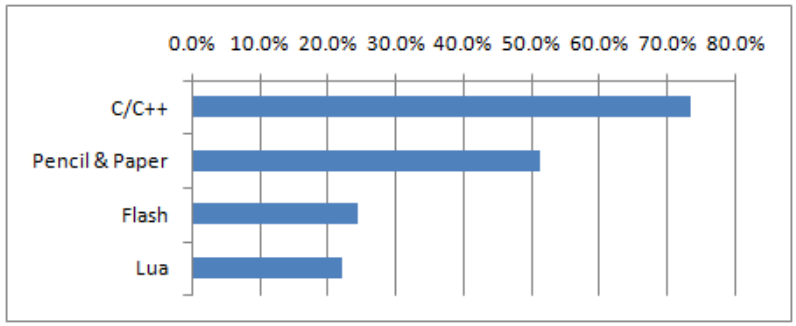
Lua is … embedded in a fair share of applications



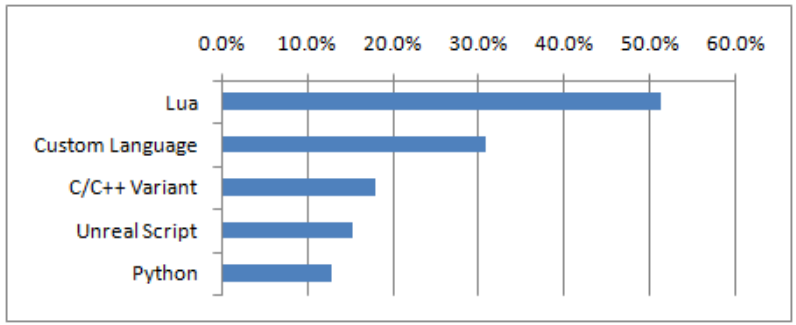
Lua is … currently the leading scripting language in games

http://www.satori.org/2009/03/the-engine-survey-general-results/

- What are you using for rapid prototyping ?

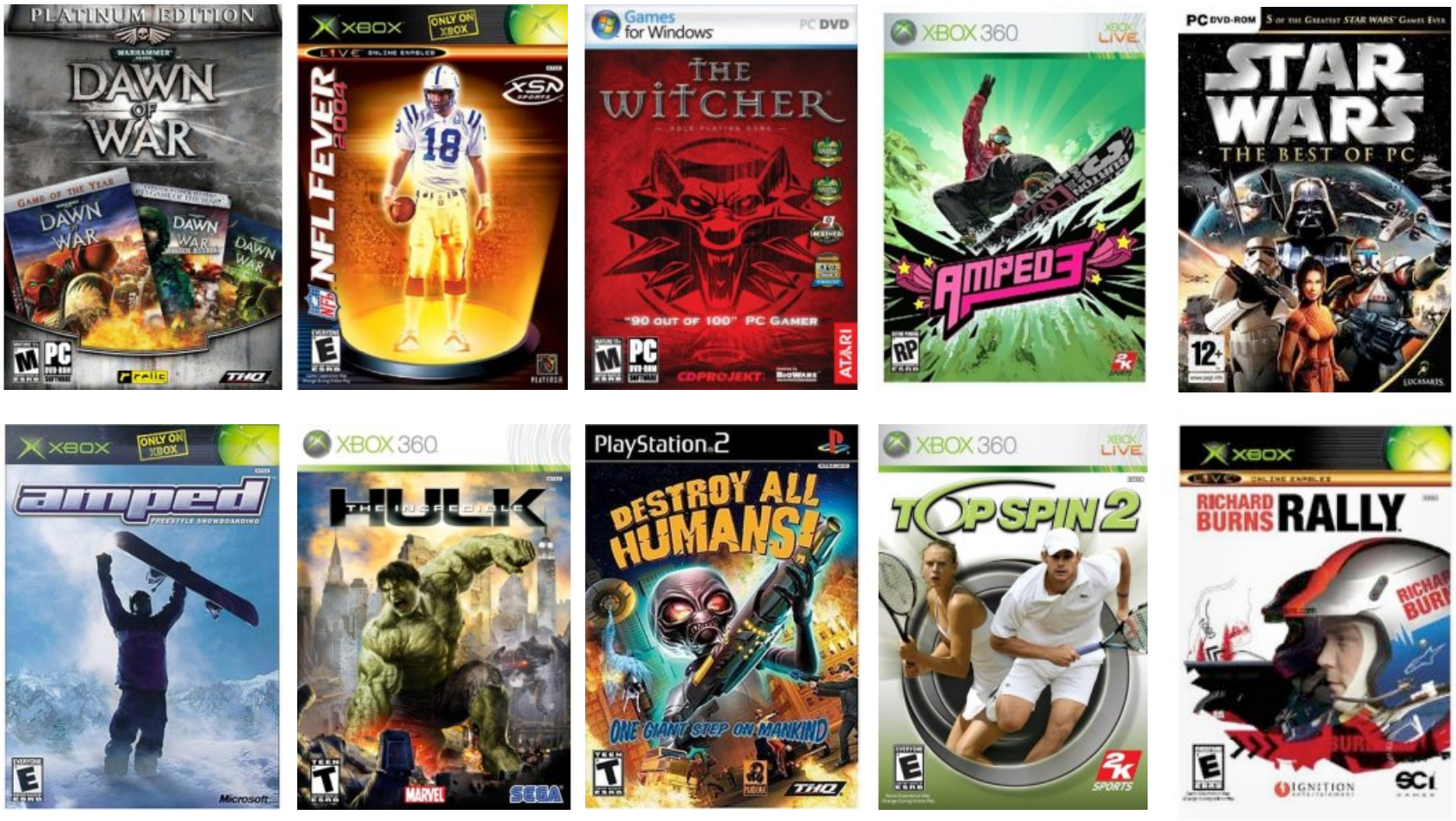


- What are you using as script language ?



Lua is … currently the leading scripting language in games

Lua is … currently the leading scripting language in games



Why Lua ?

- Embeddability
- Portability
- Simplicity
- Small Size
- Quite Efficient
- Free Open-Source Software

Why Lua ?

Embeddability

- Provided as a C library
  - stand-alone interpreter is a client
- Simple API
  - simple types
  - low-level operations
  - stack model
- Embedded in C/C++, Java, Fortran, C#, Perl, Ruby, Ada, etc.
  - language designed with embedding in mind
  - bi-directional
    - host calls Lua and Lua calls host
    - most other scripting languages focus only on calling external code

Why Lua ?

Portability

- Runs on most machines we ever heard of
  - Unix, Mac, Windows, Windows CE, Symbian, Palm, Xbox, PS2, PS3, PSP, etc.
  - embedded hardware
- Written in ANSI C / ANSI C++
  - avoids #ifdefs
  - avoids dark corners of the standard
  - development for a single and very well documented platform: ANSI C
- Lua has “two parts”: *core* and *library*
  - *core* moving towards a free-standing implementation
    - no direct dependencies on the OS

Why Lua ?

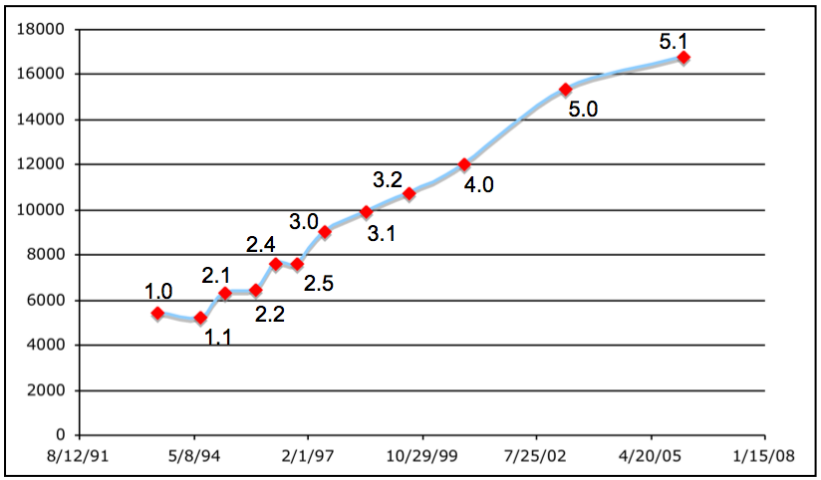
Simplicity

- Just one data structure: *tables*
  - associative arrays
  - *t.x* for *t["x"]* (sugar syntax)
  - all efforts to simplicity and performance
- Complete manual with 100 pages
  - core, libraries (standard, auxiliary) and API
- Paradigm: mechanisms instead of policies
- Non-intimidating syntax
  - For non-programmers users like engineering, geologists, mathematicians, etc

Why Lua ?

Small Size

- Entire distribution (tar.gz) has 209 KB
- Binary less than 200 KB
- <18 KB lines of source code
- *core* + *library*
  - strong separation
  - clear interface
  - core has less than 100 KB
  - easy to remove library
  - easy to add new libraries



Why Lua ?

Quite Efficient

- Not compared with C/C++, but Perl, Python, …
- Several independent benchmarks show Lua as the most efficient in the real of dynamically-typed interpreted languages
- Efficient in real code, too
- Smart implementation
  - register-based virtual machine
    - Java is stack-based
  - novel algorithm for tables
  - small and simple (!)

Why Lua ?

Free Open-Source Software

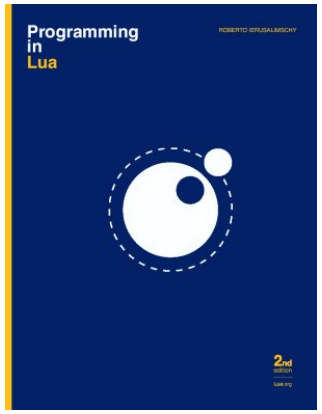
- Lua is free open-source software, distributed under a very liberal license (the well-known MIT license). It may be used for any purpose, including commercial purposes, at absolutely no cost. Just download it and use it.

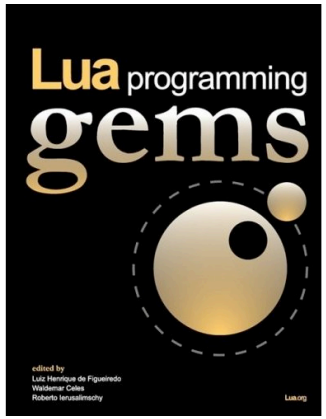
No more talk!!

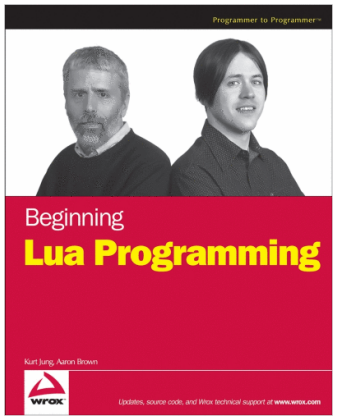
Code! Code! Code! Code! Code! Code!

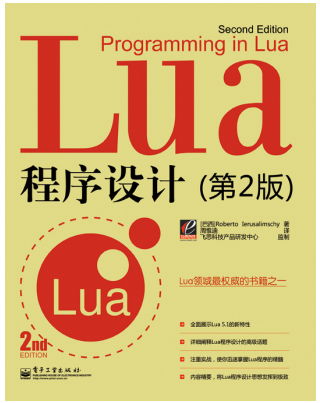
http://www.lua.org/demo.html

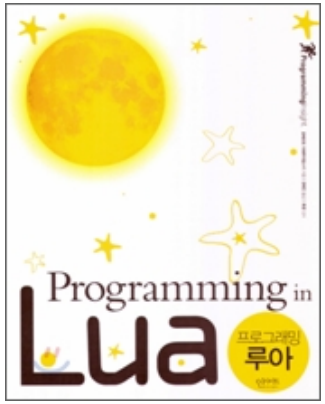
If you need a book to start …











Hello, World!

```
-- A comment in Lua starts with a double-
   hyphen
-- and runs to the end of the line.
--[[ Multi-line strings & comments
     are adorned with double square brackets. ]]

print("Hello, World!")
```

Lua Types

- *nil*: no defined value
- *boolean*: true ou false
- *number*: double float (by default)
- *string*: array of characters
- *table*: associative array
- *function*: may be C ou Lua function

- *userdata*: user defined data (void *)
- *thread*: corotine execution code

```
print(a)   -- nil value
…
a = true   -- boolean
…
a = 45.1   -- number
a = 67.2e-8
…
a = “Mathrice”   -- string
…
a = {}     -- empty table
…
a = function (x) … end
function a(x) … end
…
a = nil    -- undefine a
…
local b = 2011
```

Lua Control Structures

if <*expr*> then <*block*> else <*block*> end

if <*expr*> then <*block*> elseif <*expr*> else <*block*> end

while <*expr*> do <*block*> end

do <*block*> end -- just to control scope of variables

repeat <*block*> until <*expr*>

for <*var*>=<*exp1*>,<*exp2*>,<*exp3*> do <*block*> end

for + iterator function

-- print all values of array ‘a’

```
for i,v in ipairs(a) do print(v) end
```

ipairs: a handy iterator function to traverse a array. For each step in that loop, *i* gets an index, while *v* gets the value associated with this index

-- print all keys of table ‘t’

```
for k in pairs(t) do print(k) end
```

pairs: similar to *ipairs*, iterates over all elements of a table except that the iterator function is the *next* function

First class functions

```
local oldprint = print          -- Store current print function as oldprint
function print(s)               -- Redefine print function
  if s == "foo" then
    oldprint("bar")
  else
    oldprint(s)
  end
end
```

Any future calls to *print* will now be routed through the new function, and thanks to Lua's lexical scoping, the old *print* function will only be accessible by the new, modified *print*.

Factorial function example

The factorial is an example of a recursive function:

```
function factorial(n)
  if n == 0 then
    return 1
  else
    return n * factorial(n - 1)
  end
end
```

Boolean evaluation

The second form of the factorial function originates from Lua's short-circuit evaluation of boolean operators, in which Lua will return the value of the last operand evaluated in an expression:

```
function factorial(n)
  return n == 0 and 1 or n * factorial(n - 1)
end
```

Multiple Results

- An unconventional, but quite convenient feature of Lua is that functions may return multiple results, by listing them all after the *return* keyword.
- Lua always adjusts the number of results from a function to the circumstances of the call.
  - When we call a function as a statement, Lua discards all results from the function;
  - When we use a call as an expression, Lua keeps on the first result;
  - We get all results only when the call is the last (or the only) expression in a list of expressions
- These lists appear in four constructions in Lua
  - Multiple assignments, arguments to function calls, table constructors, and return statements

Multiple Results

```
function foo0() end                    -- return no results
function foo1() return 'a' end         -- returns 1 result
function foo2() return 'a', 'b' end    -- returns 2 results

x,y = foo0()          -- x=nil, y=nil
x,y = foo1()          -- x='a', y=nil
x,y = foo2()          -- x='a', y='b'
x = foo2()            -- x='a', 'b' is discarded
x,y,z = foo2()        -- x='a', y='b', z=nil
i,j,x,y=j,i,foo2()    -- swap i and j values, x='a', y='b'
```

Closure

```
function makeaddfunc(x)
  -- Return a new function that adds x to the argument
  return function(y)
    -- When we refer to the variable x, which is outside of the current
    -- scope and whose lifetime is longer than that of this anonymous
    -- function, Lua creates a closure.
    return x + y
  end
end
plustwo = makeaddfunc(2)
print(plustwo(5))        -- prints 7
```

Tables

Tables are the most important data structure (and, by design, the only complex data structure) in Lua, and are the foundation of all user-created types.

The table is a collection of key and data pairs (known also as hashed heterogeneous associative array), where the data is referenced by key. The key (index) can be of any data type except *nil*. An integer key of 1 is considered distinct from a string key of "1".

Tables are created using the {} constructor syntax

Tables

```
a_table = {}   -- Creates a new, empty table
-- Tables are always passed by reference
-- Creates a new table, with one associated entry. The string x mapping to
-- the number 10.
a_table = {x = 10}
-- Prints the value associated with the string key, in this case 10.
print(a_table["x"])      -- Prints 10.
b_table = a_table
b_table["x"] = 20        -- The value in the table has been changed to 20.
print(b_table["x"])      -- Prints 20.
-- Prints 20, because a_table and b_table both refer to the same table.
print(a_table["x"])      -- Prints 20.
```

Tables

Extensible semantics is a key feature of Lua, and the metatable concept allows Lua's tables to be customized in powerful and unique ways. The following example demonstrates an "infinite" table. For any n, fibs[n] will give the n^{th} Fibonacci number using dynamic programming and memoization.

```
fibs = { 1, 1 }                    -- Initial values for fibs[1] and fibs[2]
setmetatable(fibs, {               -- Give fibs some magic behavior
  __index = function(name, n)      -- Call this function if fibs[n] does not
    exist
    name[n] = name[n - 1] + name[n - 2]  -- Calculate and memorize fibs[n]
    return name[n]
  end
}
```

Tables as structure

Tables are often used as structures (or objects) by using strings as keys. Because such use is very common, Lua features a special syntax for accessing such fields.

```
point = { x = 10, y = 20 }     -- Create new table
print(point["x"])              -- Prints 10
print(point.x)                 -- Has exactly the same meaning as line above
```

Tables as namespace

By using a table to store related functions, it can act as a namespace.

```
Point = {}
Point.new = function (x, y)
  return {x = x, y = y}
end
Point.set_x = function (point, x)
  point.x = x
end
```

Object-Oriented Programming

Although Lua does not have a built-in concept of classes, they can be implemented using two language features: first-class functions and tables. By placing functions and related data into a table, an object is formed. Inheritance (both single and multiple) can be implemented via the metatable mechanism, telling the object to lookup nonexistent methods and fields in parent object(s).

Lua provides some syntactic sugar to facilitate object orientation. To declare member functions inside a prototype table, one can use function table:func(args), which is equivalent to function table.func(self, args). Calling class methods also makes use of the colon: object:func(args) is equivalent to object.func(object, args).

Object-Oriented Programming

```
Vector = {}                       -- Create a table to hold the class methods
function Vector:new(x, y, z)      -- The constructor
  local object = { x = x, y = y, z = z }
  setmetatable(object, { __index = Vector })  -- Inheritance
  return object
end
function Vector:magnitude()       -- Another method (member function)
  -- Reference the implicit object using self
  return math.sqrt(self.x^2 + self.y^2 + self.z^2)
end

vec = Vector:new(0, 1, 0)         -- Create a vector
print(vec:magnitude())            -- Call a member function using ":"
print(vec.x)                      -- Access a member variable using "."
```

Standard Libraries

- Basic
- String
- Table
- Math
- IO (input/output)
- OS (operating system)
- Debug
- Coroutine (threads)

Strings – pattern matching

- The most powerful functions in the string library are *find*, *match*, *gsub* (global substitution), and *gmatch* (global match)

.	All characters	%p	punctuation characters
%a	letters	%s	space characters
%c	control characters	%w	alphanumeric characters
%d	digits	%x	hexadecimal digits
%l	lower-case letters	%u	upper-case letters

Strings – pattern matching

```
 s = "hello world from Lua"
for w in string.gmatch(s, "%a+") do
      print(w)
end


t = {}
s = "from=world, to=Lua"
for k, v in string.gmatch(s, "(%w+)=(%w+)") do
    t[k] = v
end
```

Strings – pattern matching

x = string.gsub("hello world", (%w+)", "%1 %1")

→ x="hello hello world world"

x = string.gsub("hello world from Lua", "(%w+)%s*(%w+)", "%2 %1")

→ x="world hello Lua from"

date = "Today is 2/6/1966"

d,m,y = string.match(date, "(%d+)/(%d+)/(%d+)")

print(d,m,z)

→ 2 6 1966

test = "int x; /* x */ int y; /* y */"

print(string.gsub(test, "/%*.-%*/", "<comment>")

→ int x; <comment> int y; <comment> 2

Community Libraries

http://www.lua.org

http://luaforge.net

http://lua-users.org

- http://lua-users.org/wiki/LuaDirectory
- http://lua-users.org/wiki/LuaAddons

http://www.tecgraf.puc-rio.br/iup

http://www.tecgraf.puc-rio.br/cd

http://www.tecgraf.puc-rio.br/im

IUP – Graphical User Interface

IUP is a portable toolkit for building graphical user interfaces.

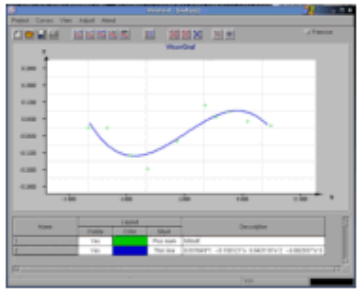
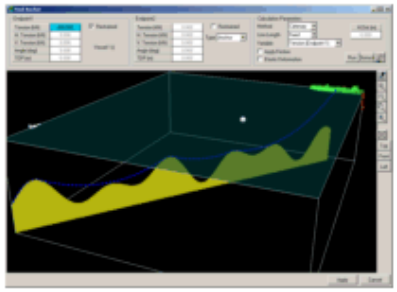
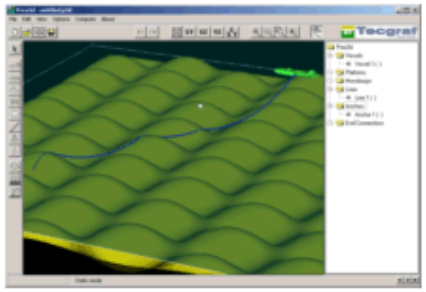
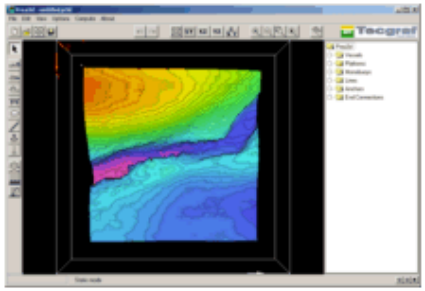
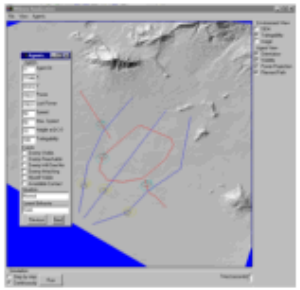
IUP's purpose is to allow a program to be executed in different systems without any modification, therefore it is highly portable. Its main advantages are:

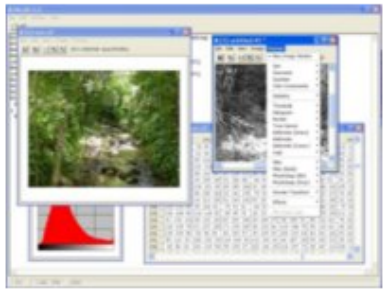
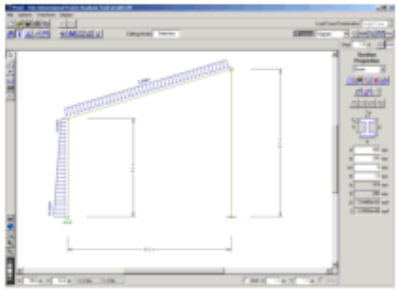
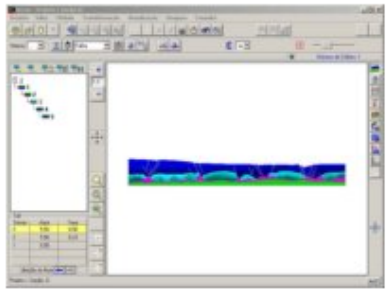
- high performance, due to the fact that it uses native interface elements.
- fast learning by the user, due to the simplicity of its API.

All the IUP functions are available in Lua, with a few exceptions.

We call it **IUPLua**. To use them the general application will do require"iuplua", and require"iupluaxxxx" to all other secondary libraries that are needed.

IUP – Graphical User Interface

IUP – Graphical User Interface

```
-- Dialog Example in IupLua
-- Creates a simple dialog.
require( "iuplua" )
vbox = iup.vbox {   iup.label {title="Label"},
                    iup.button { title="Test" } }
dlg = iup.dialog{vbox; title="Dialog"}
dlg:show()
iup.MainLoop()
```

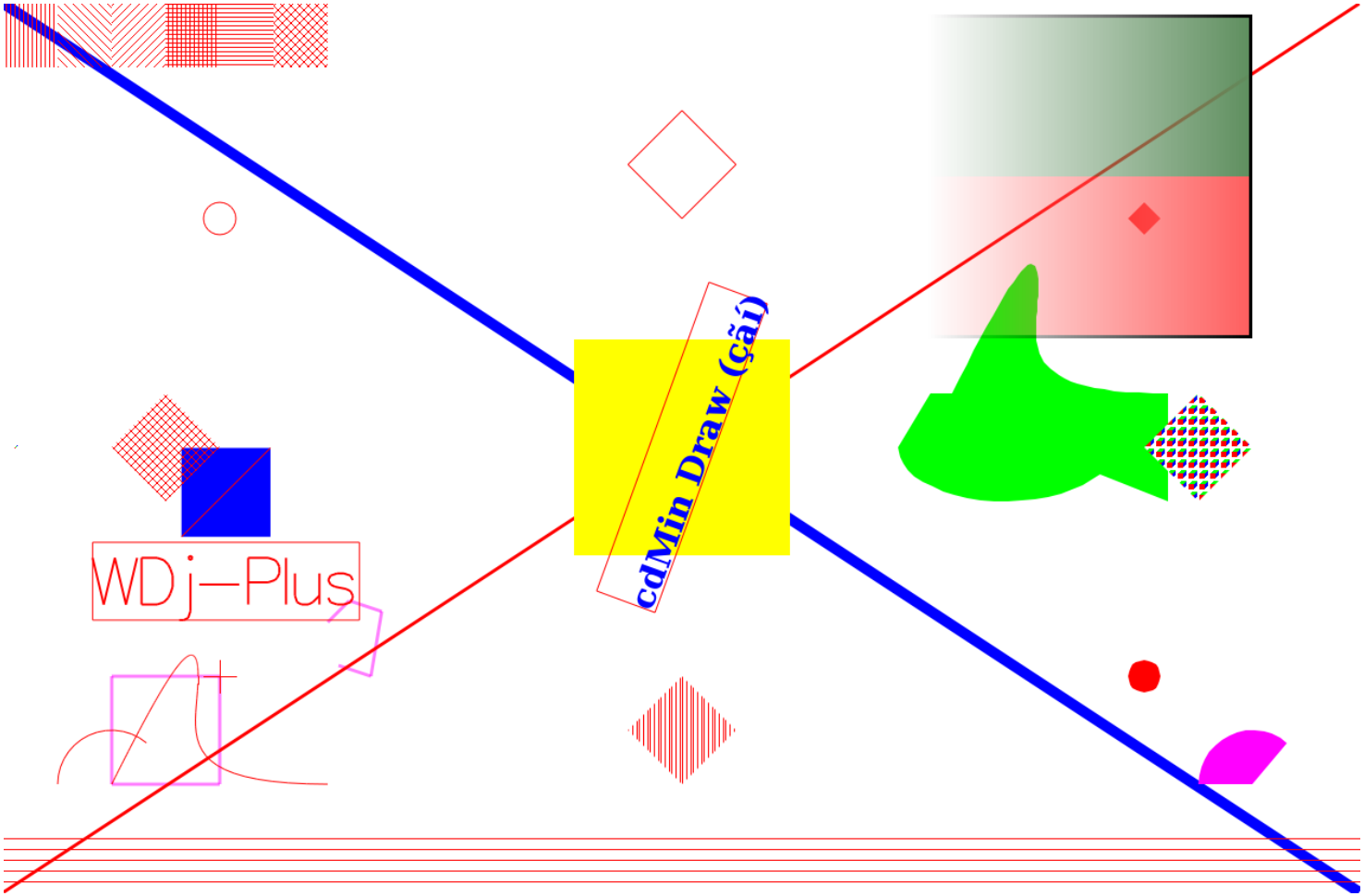
CD – Canvas Draw

CD is a platform-independent graphics library. It is implemented in several platforms using native graphics libraries: Microsoft Windows (GDI) and X-Windows (XLIB).

The library contains functions to support both vector and image applications, and the visualization surface can be either a window or a more abstract surface, such as Image, Clipboard, Metafile, PS, and so on.

All the CD functions are available in Lua, with a few exceptions. We call it **CDLua**. To use them the general application will do require"cdlua", and require"cdluaxxxx" to all other secondary libraries that are needed.

CD – Canvas Draw



CD – Canvas Draw

```
require("cdlua")
require("cdluapdf")
canvas = cd.CreateCanvas(cd.PDF, "test.pdf")
canvas:Foreground (cd.RED)
canvas:Box (10, 55, 10, 55)
canvas:Foreground(cd.EncodeColor(255, 32, 140))
canvas:Line(0, 0, 300, 100)
canvas:Kill()
```

IM – Digital Image

IM is a toolkit for Digital Imaging. IM is based on 4 concepts: Image Representation, Storage, Processing and Capture.

The main goal of the library is to provide a simple API and abstraction of images for scientific applications.

The most popular file formats are supported: TIFF, BMP, PNG, JPEG, GIF and AVI. Image representation includes scientific data types. About a hundred Image Processing operations are available.

All the IM functions are available in Lua, with a few exceptions. We call it **ImLua**. To use them the general application will do require"imlua", and require"imluaxxxx" to all other secondary libraries that are needed.

IM – Digital Image

```
require("imlua")
require("imlua_process")
local image = im.ImageCreate(500, 500, im.RGB, im.BYTE)
im.ProcessRenderRandomNoise(image)
image:Save("noise.tif", "TIFF")
```

LuaSQL

LuaSQL is a simple interface from Lua to a DBMS. It enables a Lua program to:

- Connect to ODBC, ADO, Oracle, MySQL, SQLite and PostgreSQL databases;
- Execute arbitrary SQL statements;
- Retrieve results in a row-by-row cursor fashion.

LuaSQL: PostgreSQL example

```
-- load driver
require "luasql.postgres"
-- create environment object
env = assert (luasql.postgres())
-- connect to data source
con = assert (env:connect("luasql-test"))
-- reset our table
res = con:execute"DROP TABLE people"
res = assert (con:execute[[
  CREATE TABLE people(
    name  varchar(50),
    email varchar(50)
  )
]])
```

```
-- add a few elements
list = {
  { name="Jose das Couves",
      email="jose@couves.com", },
  { name="Manoel Joaquim",
      email=”m.joaquim@fundo.com", },
  { name="Maria das Dores",
      email="maria@dores.com", },
}
for i, p in pairs (list) do
  res = assert (con:execute(string.format([[
    INSERT INTO people
    VALUES ('%s', '%s')]], p.name, p.email)
  ))
end
```

LuaSQL

```
-- retrieve a cursor
cur = assert (con:execute"SELECT name, email from
      people")
-- print all rows, the rows will be indexed by field names
row = cur:fetch ({}, "a")
while row do
  print(string.format("Name: %s, E-mail: %s",
       row.name, row.email))
  -- reusing the table of results
  row = cur:fetch (row, "a")
end
-- close everything
cur:close()
con:close()
env:close()
```

Output:

Name: Jose das Couves, E-mail: jose@couves.com

Name: Manoel Joaquim, E-mail: m.joaquim@fundo.com

Name: Maria da Dores, E-mail: maria@dores.com

LuaSOCKET

LuaSocket is a Lua extension library that is composed by two parts: a C core that provides support for the TCP and UDP transport layers, and a set of Lua modules that add support for functionality commonly needed by applications that deal with the Internet.

- The core support has been implemented so that it is both efficient and simple to use. It is available to any Lua application once it has been properly initialized by the interpreter in use. The code has been tested and runs well on several Windows and Unix platforms.

- Among the support modules, the most commonly used implement the SMTP (sending e-mails), HTTP (WWW access) and FTP (uploading and downloading files) client protocols. These provide a very natural and generic interface to the functionality defined by each protocol. In addition, you will find that the MIME (common encodings), URL (anything you could possible want to do with one) and LTN12 (filters, sinks, sources and pumps) modules can be very handy.

LuaSOCKET: TCP example

```
local socket = require("socket")                -- load namespace
-- create a TCP socket and bind it to the local host, at any port
local server = assert(socket.bind("*", 0))
local ip, port = server:getsockname()           -- find out which port the OS chose for us
print("Please telnet to localhost on port " .. port)  -- print a message informing what's up
print("After connecting, you have 10s to enter a line to be echoed")
while 1 do                                      -- loop forever waiting for clients
  local client = server:accept()                -- wait for a connection from any client
  client:settimeout(10)                         -- make sure we don't block waiting for this client's line
  local line, err = client:receive()            -- receive the line
  if not err then client:send(line .. "\n") end -- if there was no error, send it back to the client
  client:close()                                -- done with client, close the object
end
```

LuaSOCKET: UDP example

```
-- change here to the host an port you want to contact
local host, port = "localhost", 13
local socket = require("socket")   -- load namespace
-- convert host name to ip address
local ip = assert(socket.dns.toip(host))
local udp = assert(socket.udp())   -- create a new UDP object
-- contact daytime host
assert(udp:sendto("anything", ip, port))
-- retrieve the answer and print results
io.write(assert(udp:receive()))
```

Questions ?