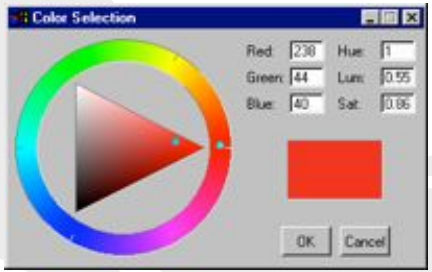
A Linguagem Lua

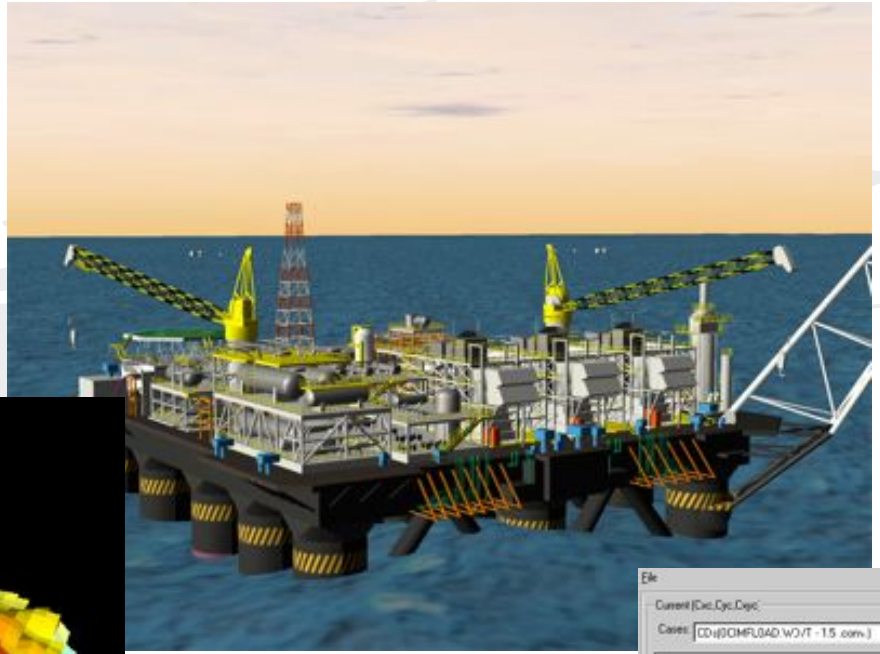
Ruan R. Martinelli
Káio C. F. Simonassi
Felipe P. C. Tavares

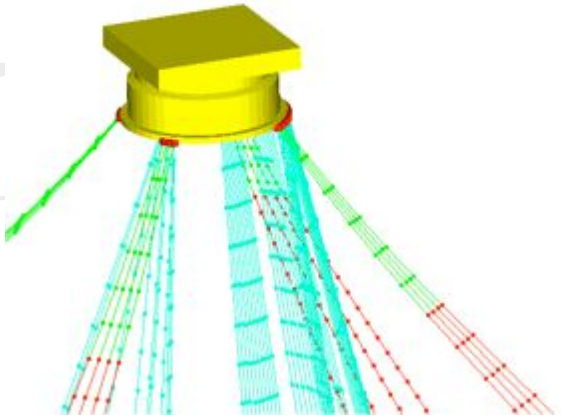
- Lua nasceu em 1993 no **Tecgraf**, na PUC-Rio;
- Criada para ser utilizada em um projeto da **Petrobras**;
- Única linguagem criada em um país em desenvolvimento a ganhar projeção mundial

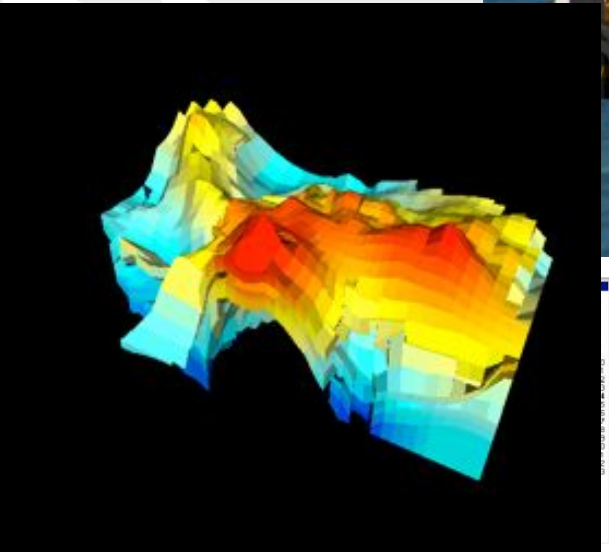
Waldemar, Roberto, Luiz

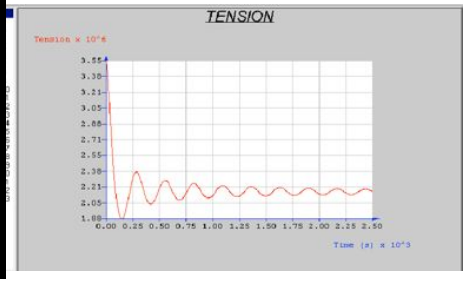
Projetos da Tecgraf

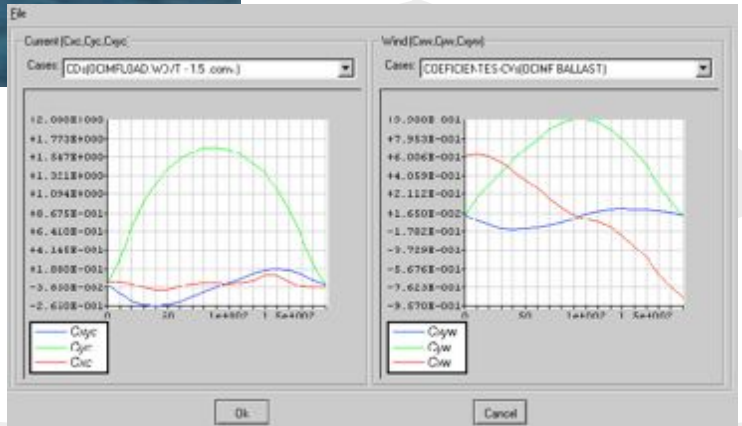












Na época, precisava-se de uma linguagem:

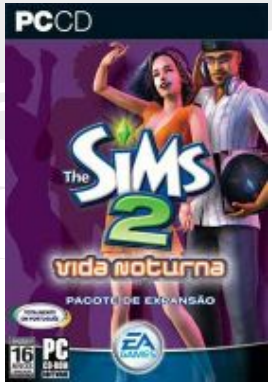
- Portável;
- Capaz de descrever dados facilmente;
- Amigável com **C**;
- Sintaxe fácil.

Por que usar Lua?

Lua é ***rápida***

Lua é ***portátil***

*Lua é **pequena***



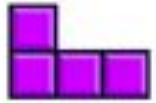
Lua é ***bem estabelecida***

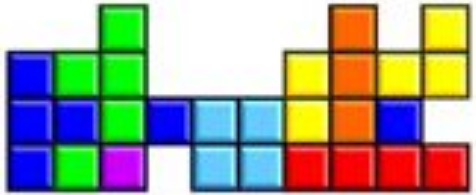
Em **jogos,** Lua é usada para:

- Implementar o script do jogo;
- Definir objetos e comportamentos;
- Gerenciar algoritmos de I.A.;
- Controlar personagens;
- Descrever a interface com o usuário;
- Testar, depurar.

De acordo com o site gamedev.net*:

- 72% dos jogos são desenvolvidos com o auxílio de uma linguagem script
- 20% dos jogos (na época) usavam Lua
- **Python** aparece com apenas 7%





*dados de 2003

Características

- Lua é uma biblioteca em **C**, podendo ser compilada em qualquer plataforma que possua compilador C (ou C++);
- Lua trabalha acoplada a uma aplicação hospedeira (**host**);
- Para a aplicação ter acesso a Lua, é **aberta uma biblioteca**. Feito isso, a aplicação pode utilizar recursos fornecidos por Lua (executar scripts e acessar dados armazenados em Lua, por exemplo)

Avaliação da LP

Legibilidade vs **Redigibilidade**:

- Como a linguagem é dinamicamente tipada ela favorece redigibilidade.

Eficiência vs **Portabilidade**:

- LUA compila sem modificações no código fonte em todas as plataformas que possuem um compilador C padrão.

Avaliação da LP

Confiabilidade:

- Graças a sua tipagem dinâmica o programador pode cometer erros simples que mais tarde serão difíceis de se identificar.

Compilação

- De forma similar a **Java**, Lua ao ter seu código fonte compilado gera um **bytecode** que é interpretado por uma Virtual Machine que é baseada em registradores (daí sua grande portabilidade).

Valores e Tipos de Dados

Em Lua, as variáveis não têm tipos associados a elas: os tipos estão associados aos valores armazenados nas variáveis.

```
a = "Exemplo"    -- a armazena string
b = 1.23          -- b armazena numero
...
b = nil               -- b armazena nil
a = 3                 -- a armazena numero
```

Valores e Tipos de Dados

Em Lua, variáveis globais não precisam ser declaradas. Quando escrevemos a = 3, a variável a é, por default, uma variável global. Se desejarmos que uma variável tenha escopo local, devemos declará-la previamente usando a palavra reservada local.

- Por exemplo:

```
local a
...
a = 3
```

Valores e Tipos de Dados

Existem oito tipos básicos em Lua: *nil, boolean, number, string, function, userdata, thread e table*.

Nil

- O tipo nil representa o valor **indefinido**. Todas as variáveis ainda não inicializadas assumem o valor nil. Assim, se o código:
  a = b
  for encontrado antes de qualquer atribuição à variável b, então esta é assumida como contendo o valor nil, o que significa que a também passa a armazenar nil, independentemente do valor anteriormente armazenado em a.

Valores e Tipos de Dados

Boolean:

- É o tipo dos valores **false** e **true**. Tanto nil como false tornam uma condição falsa; **qualquer outro** valor torna a condição verdadeira.

Valores e Tipos de Dados

Number:

- O tipo number representa valores numéricos. Lua não faz distinção entre valores numéricos com valores inteiros e reais. Todos os valores numéricos são tratados como sendo do tipo number. Assim, o código

```
a = 4
b = 4.0
c = 0.4e1
d = 40e-1
```

armazena o valor numérico quatro nas variáveis a, b, c e d.

Valores e Tipos de Dados

String:

- O tipo string representa **cadeia de caracteres**. Uma cadeia de caracateres em Lua é definida por uma seqüência de caracteres delimitadas por **aspas simples** (' ') ou **duplas** ("").
  Para reproduzir na cadeia de caracteres as aspas usadas como delimitadoras, é necessário usar os caracteres de escape. Assim, são válidas e equivalentes as seguintes atribuições:

```
s = "Pau d'agua"
s = 'Pau d\'agua'
```

Em Lua, cadeias de caracteres podem conter **qualquer** caractere de 8 bits, incluindo zeros ('\0') dentro dela.

Valores e Tipos de Dados

Function:

- Funções em Lua são consideradas valores de primeira classe. Isto significa que funções **podem ser armazenadas em variáveis**, passadas como **parâmetros para outras funções**, ou **retornadas** como resultados. A definição de uma função equivale a atribuir a uma variável global o valor do código que executa a função.

```
function func1 (...)
...
end
```

que pode posteriormente ser executada através de uma chamada de função:

```
func1(...)
```

Valores e Tipos de Dados

Userdata:

- O tipo ***userdata*** permite que dados C arbitrários possam ser armazenados em variáveis Lua.

  Este tipo corresponde a um bloco de memória e não tem operações pré-definidas em Lua, exceto atribuição e teste de identidade. Contudo, através do uso de metatabelas, o programador pode definir operações para valores **userdata**. Valores **userdata** não podem ser criados ou modificados em Lua, somente através da API C. Isto garante a integridade dos dados que pertencem ao programa hospedeiro

Valores e Tipos de Dados

Thread:

- O tipo thread representa fluxos de execução independentes e é usado para implementar co-rotinas. Não confunda o tipo thread de Lua com processos leves do sistema operacional. Lua dá suporte a co-rotinas em todos os sistemas, até mesmo naqueles que não dão suporte a processos leves.

Valores e Tipos de Dados

Table:

- Implementa **arrays associativos**, isto é, arrays que podem ser indexados não apenas por números, mas por **qualquer valor** (exceto nil).
  Tabelas são o **único** mecanismo de estruturação de dados em Lua; elas podem ser usadas para representar arrays comuns, tabelas de símbolos, conjuntos, registros, grafos, árvores, etc.

Valores e Tipos de Dados

- Valores do tipo table, function, thread e userdata **são objetos**: variáveis não contêm realmente estes valores, **somente referências** para eles.

- **Atribuição**, **passagem de parâmetro**, e **retorno de funções** sempre lidam com referências para tais valores; estas operações **não implicam** em qualquer espécie de **cópia**.

Palavras Reservadas

- As seguintes *palavras-chave* são **reservadas** e não podem ser utilizadas como nomes:

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

- Lua é uma linguagem que **diferencia minúsculas de maiúsculas**: *and* é uma palavra reservada, mas *And* e *AND* são dois nomes válidos diferentes. Como convenção, nomes que começam com um sublinhado seguido por letras maiúsculas (tais como _VERSION) são reservados para variáveis globais internas usadas por Lua.

Itens Léxicos

- As seguintes cadeias denotam outros itens léxicos:

```
+       -       *       /       %       ^       #
==      ~=      <=      >=      <       >       =
(       )       {       }       [       ]
 ;       :       ,       .       ..      ...
```

- *Cadeias de caracteres literais* podem ser delimitadas através do uso de aspas simples ou aspas duplas, podem conter as seguintes seqüências de escape no estilo de C: como por exemplo: '\n' (quebra de linha).

Coletor de Lixo

- Em Lua, você **não precisa se preocupar** com a alocação de memória para novos objetos nem com a liberação de memória quando os objetos não são mais necessários (a memória é gerenciada **automaticamente** usando um *coletor de lixo)*
- O coletor de lixo de Lua é do tipo **marca-e-limpa** (*mark-and-sweep*) incremental.
- É possível mudar estes números através de chamadas às funções **lua_gc** em C ou **collectgarbage** em Lua.

Tratamento de Erros

- As ações de Lua começam a partir de **código C** no programa hospedeiro
- Sempre que um erro ocorre durante a compilação ou execução, o **controle retorna para C**, que pode tomar as medidas apropriadas (tais como imprimir uma mensagem de erro).
- O código Lua pode explicitamente **gerar um erro** através de uma chamada à função *error*.

Mini-tutorial Lua

Comentários

```
-- Dois hífens realizam um comentário de uma linha

--[[
   Acrescentar dois colchetes faz um bloco
]]--
```

Variáveis e Controle de Fluxo

```
num = 42  -- Todos os números são doubles.

s = 'gandalf'

t = "aspas duplas também funcionam"
u = [[ Colchetes duplos
       para strings de
       múltiplas linhas.]]

t = nil  -- Desreferencia t; Lua tem coletor de lixo.
```

Variáveis e Controle de Fluxo

```
-- Blocos são representados por palavras-chave do/end:
while num < 50 do
  num = num + 1  -- Não existem operadores do tipo ++ ou +=.
end

-- Condições:
if num > 40 then
  print('maior 40')
elseif s ~= 'gandalf' then  -- ~= é diferente de.
  -- Checagem de igualdade é == como em Python.
  io.write('nao eh maior que 40\n')
else
```

Variáveis e Controle de Fluxo

```
-- Variáveis são globais por padrão.
  souGlobal = 5  -- Camel case é muito utilizado.

-- Para fazer uma variável ser local:
  local linha = io.read()  -- Lê a próxima linha de entrada.

-- Utiliza-se o operador .. para concatenação de strings:
  print('Winter is coming, ' .. linha)
end
```

Variáveis e Controle de Fluxo

```
-- Variáveis que não foram definidas retornam nil.

-- Isso não causa erro:
foo = variavelAleatoria  -- Agora foo = nil.

umValorBool = false

-- Apenas nil e false são "falsos"; 0 e '' são true!
if not umValorBool then print('eh falso') end

-- 'or' e 'and' são curto-circuitados.
```

Variáveis e Controle de Fluxo

```
somaJoao = 0
for i = 1, 100 do  -- Início e fim do loop são passados.
  somaJoao = somaJoao + i
end

-- Use "100, 1, -1" para contagem regressiva:
somaMaria = 0
for j = 100, 1, -1 do somaMaria = somaMaria + j end

-- Em geral, a ordem de passagem do for é
-- início, final[, incremento].
```

Funções

```
function foo()
  local x, y = algumacoisa(4, 5) -- Váriáveis locais.
  return x ^ y
end

function algumacoisa(x, y)
  local a = (x * y) ^ 2
  local b = (x - y) ^ 2
  return a + b, a * b -- Pode-se retornar mais de 1 valor.
end
```

Funções

```lua
function bar(a, b, c)
  print(a, b, c)
  return 4, 8, 15, 16, 23, 42
end

x, y = bar('shoryuken')  --> Imprime "shoryuken  nil nil"
-- Agora x = 4, y = 8, valores 15..42 são descartados.
```

Funções

```
-- Chamadas com parâmetros de uma string não precisam de
parenteses:
print 'hello'  -- Funciona.
```

Tabelas

```lua
-- Tabelas = Única estrutura de dados complexa presente
--           em Lua; são arrays associativos.
x = 5
a = {} -- Tabela vazia.
b = { chave = x, outraChave = 10 } -- Strings como chaves.
print(b.outraChave) -- Imprime 10.

-- Atribuições:
a[1] = 20
a["foo"] = 50
a[x] = "bar"
```

Metatabelas

```lua
f1 = {a = 1, b = 2}  -- Representa a fração a/b.
f2 = {a = 2, b = 3}

-- Não funciona: s = f1 + f2

metafracao = {}
function metafracao.__add(f1, f2)
  soma = {}
  soma.b = f1.b * f2.b
  soma.a = f1.a * f2.b + f2.a * f1.b
  return soma
end
```

Metatabelas

```
-- Método setmetatable faz a ligação de uma tabela com sua
-- metatabela.

setmetatable(f1, metafracao)
setmetatable(f2, metafracao)

s = f1 + f2  -- chama __add(f1, f2) da metatabela.
```

Metatabelas

```lua
-- Sobrecarga na metatabela usando o metamétodo __index:

favoritos = {animal = 'gorila', comida = 'banana'}
meusFavoritos = {comida = 'pizza'}
setmetatable(meusFavoritos, {__index = favoritos})
quemComeu = meusFavoritos.animal  -- retorna 'gorila'.
```

Metatabelas

```
-- Alguns metamétodos:
-- __add(a, b)                          for a + b
-- __sub(a, b)                          for a - b
-- __mul(a, b)                          for a * b
-- __div(a, b)                          for a / b
-- __mod(a, b)                          for a % b
-- __pow(a, b)                          for a ^ b
-- __concat(a, b)                       for a .. b
-- __len(a)                             for #a
-- __index(a, b)  <fn or a table>       for a.b
-- __newindex(a, b, c)                  for a.b = c
-- __call(a, ...)                       for a(...)
```

Referências Bibliográficas

- Ierusalimschy, Roberto; Figueiredo, Luiz Henrique de; Celes, Waldemar. *Lua Reference manual*. Rio de Janeiro: Lua.org, 2006. 103 p. ISBN 85-903798-3-3
- Tyler Neylon, http://tylerneylon.com/a/learn-lua/
- Nova Fusion, http://nova-fusion.com/2012/08/27/lua-for-programmers-part-1/