Lua
Uma linguagem de extensão extensível

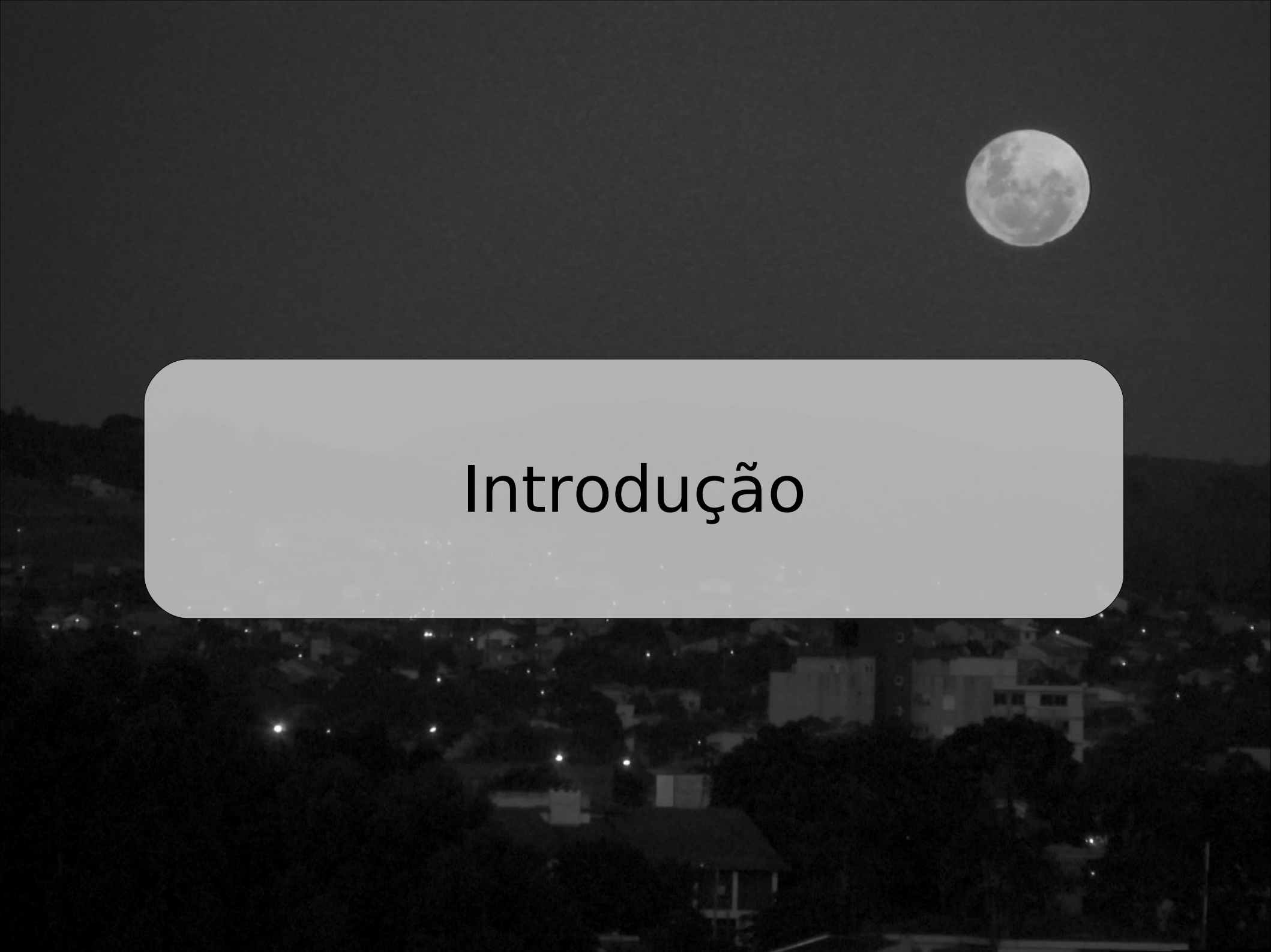
Leandro Motta Barros

lmb@stackedboxes.org
~~leandromb@unisinos.br~~

GDS – 6 de novembro de 02007
(atualizações em 30 de abril de 02009)

Sumário

- Introdução
- A Linguagem Lua
- Extendendo e Integrando
  - API C
  - tolua/tolua++
  - Luabind
  - Diluculum



Introdução

O que é Lua, afinal de contas?

- Uma biblioteca que provê uma linguagem de programação
- Uma linguagem de programação implementada com uma biblioteca

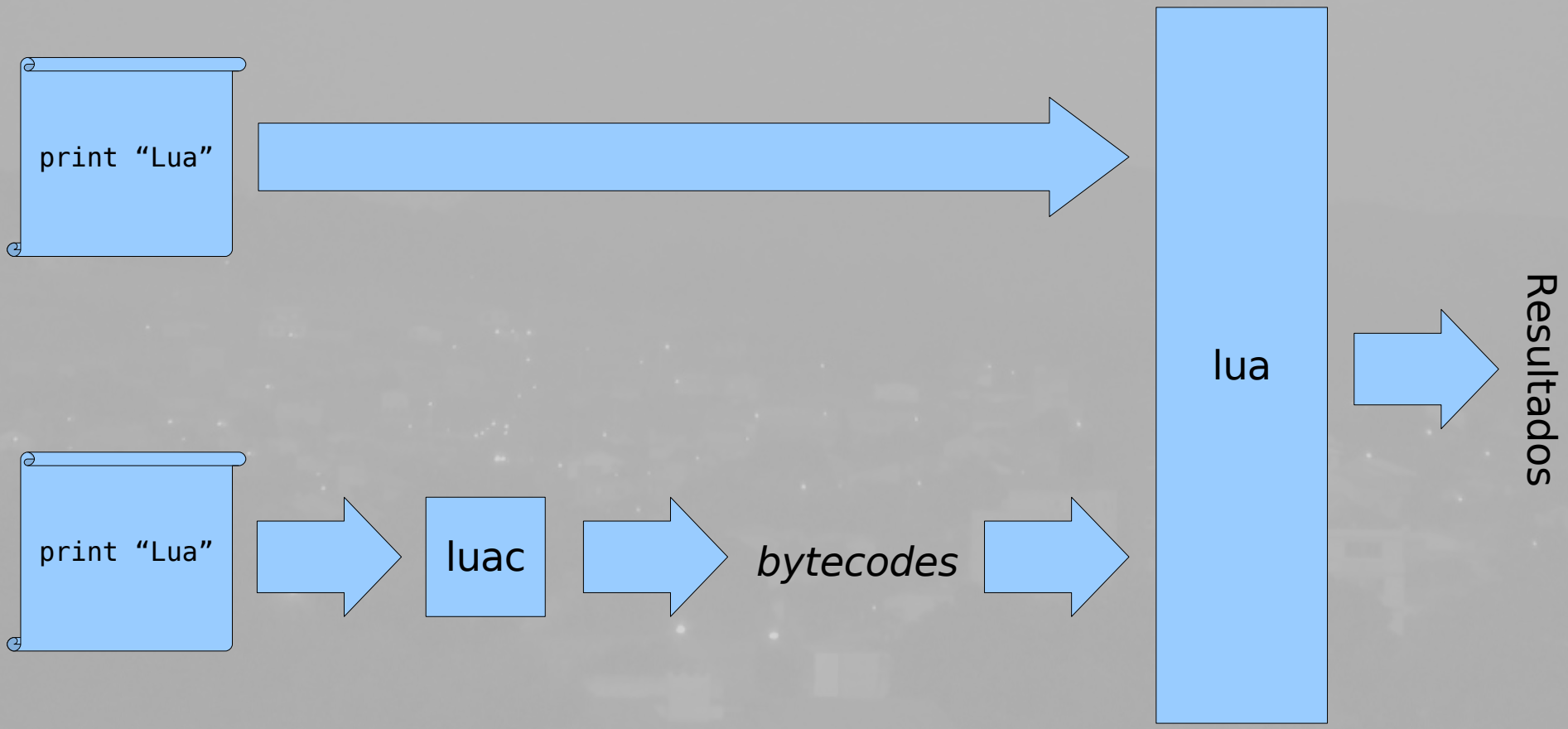
- Linguagem de extensão
- Linguagem extensível

Brevíssimo histórico

- Criada no Tecgraf da PUC-Rio, em 01993
- Usada “de verdade” desde a primeira versão
- Primeiro, em projetos do Tecgraf
- Depois, o Mundo!
- Atualmente na versão 5.1 (fevereiro de 02006)
  - Atualização [30/abril/02009]: Versão atual é Lua 5.1.4, (lançada em agosto de 02008)

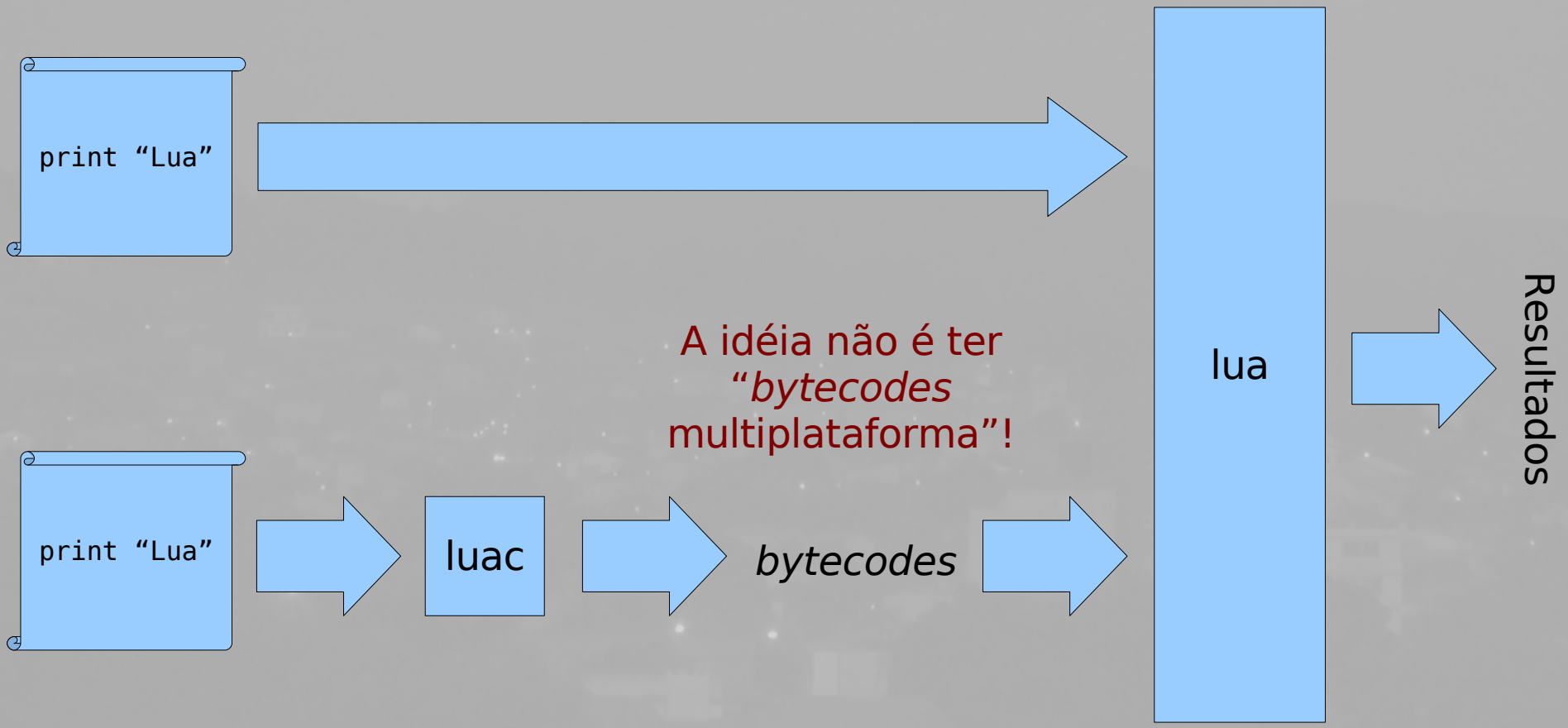
Algumas características

- Interpretada (compilada para *bytecodes*)



Algumas características

- Interpretada (compilada para *bytecodes*)



Algumas características

- Boa para “descrição de dados”

```
Hero = {
   name = “Freddy Hardest”,
   speed = 3,
   strength = 12,
   intelligence = 7,
   model = “freddy.dae”,
   inventory = { “LaserGun”, “Boots”, “AstroSuit” }
}
```

Algumas características

- Boa para “descrição de dados”

```
Hero = {
    name = “Freddy Hardest”,
    speed = 3,
    strength = 12,
    intelligence = 7,
    model = “freddy.dae”,
    inventory = { “LaserGun”, “Boots”, “AstroSuit” }
}
```

Algumas características

- Simples
  - Sintaxe clara
  - Uso por não programadores
  - Simplicidade tem ótimos efeitos colaterais!

```
-- Define uma função
function  Testa (a, b)
   if a > b then
      print (“Maior!”)
   else
      print (“Menor!”)
   end
end

-- Chama a função
Testa (10.4, 38)
```

Algumas características

- Poderosa
  - Funcionalidades criteriosamente incluídas

Algumas características

- Poderosa
  - Funcionalidades criteriosamente incluídas

Um dia, alguém perguntou algo como:

“Why doesn't Lua have ⟨some feature⟩?”

Luiz Henrique de Figueiredo respondeu:

“Lua evolves by answering 'why?' not 'why not?'”

Algumas características

- Não força políticas
  - Provê funcionalidades básicas
  - Vide programação OO em Lua
    - LOOP: http://oil.luaforge.net/loop

Algumas características

- Pequena
  - Meu `liblua.a` tem 208 kB
  - E isso inclui todos os “opcionais”, como a biblioteca padrão e o compilador!
- Rápida
  - Para os padrões de linguagem interpretada
- Altamente portável
  - Implementada em ANSI/ISO C

Algumas características

- Robusta
  - Diversas aplicações rodando muito bem
- “*Software* aberto com desenvolvimento fechado”
  - Licença MIT

Alguns usuários

- LucasArts: Substituindo o SCUMM

Alguns usuários

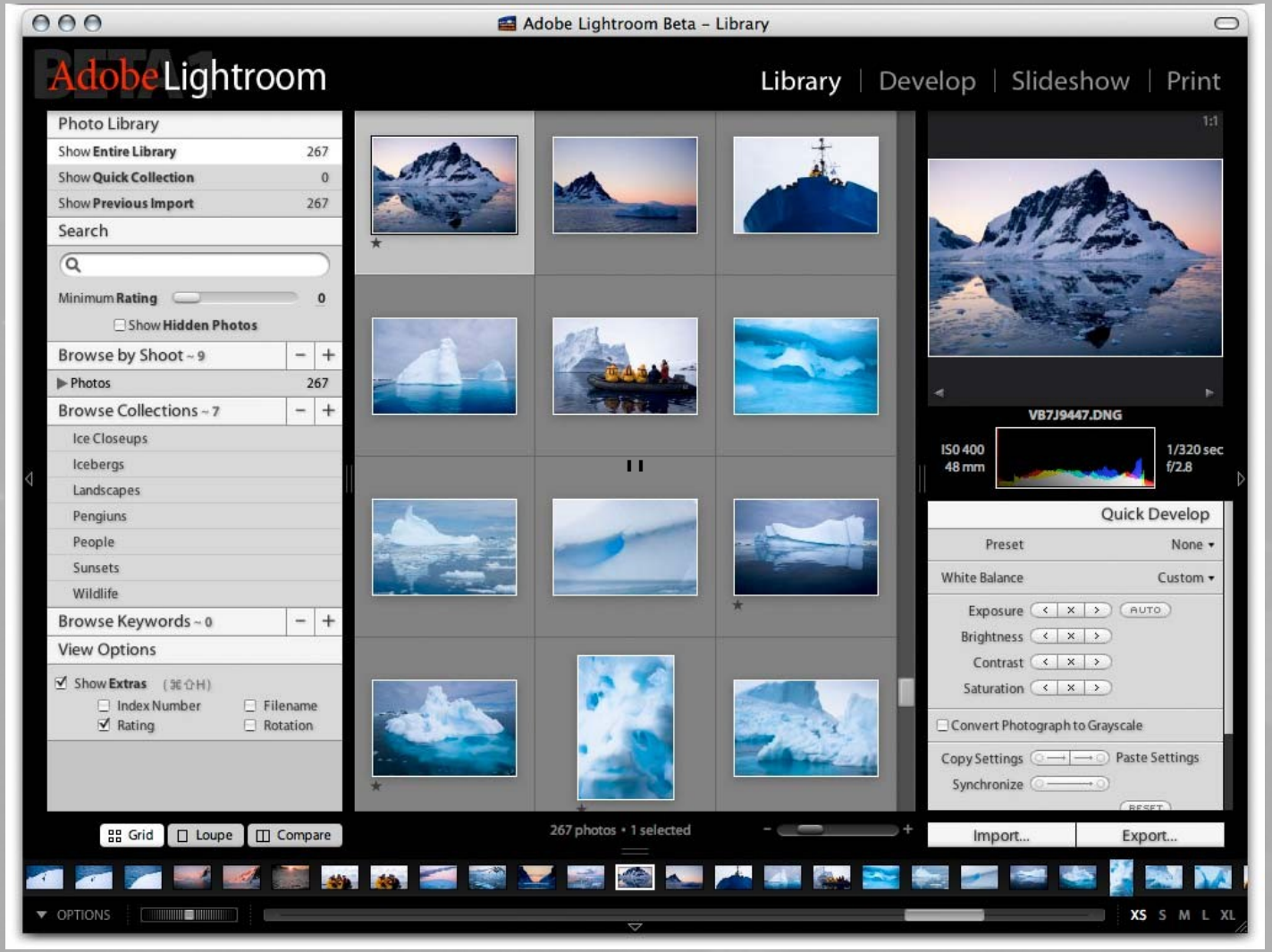
- Blizzard: World of Warcraft

Alguns usuários

- Crytec: Far Cry, Crysis

Alguns usuários

- Adobe: Lightroom



Alguns usuários

- Olivetti: Configuração de impressoras

Alguns usuários

- Eu :-)

A Linguagem Lua

Primeiro programa

- Como não podia deixar de ser...

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

Primeiro programa

- Como não podia deixar de ser...

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

- Variações...

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

print "Hello, World!"

print 'Hello, World!'

print [[Hello,
World!]]

print ([[Hello, World!]]);
```

Comentários

```
-- Isso é um comentário de uma linha


--[[
E isso é um comentário
feio de múltiplas
linhas
--]]
```

Variáveis

- Por padrão, são globais!

```
local message = "Hello, World!"

print (message)
```

Variáveis

- Variáveis não têm tipo, valores têm

```
-- 'var' contém uma string
local var = "Blá, blá, blá"
print (var)

-- agora um número
var = 3.1415
print (var)

-- e agora um booleano
var = true
print (var)
```

Tipos

- Nil
  - Tipicamente, representa a ausência de um valor útil
  - Somente `nil` tem tipo Nil
  - `nil` é diferente de qualquer outro valor
  - Variáveis não inicializadas têm valor `nil`

```
if hero == nil then
  hero = InitializeHero()
end

Display (hero)
```

Tipos

- Booleanos
  - `true` e `false`

```
local x, y = true, false -- atribuição múltipla

if not ((x and y) or y) then
   print "blá"
end

if x == y then
   print "blé"
end

if x ~= y then
   print "bli"
end
```

Tipos

- Booleanos
  - Em testes, apenas `nil` e `false` são considerados falsos
    - *String* vazia ("") e zero (0) são considerados verdadeiros

Tipos

- Números
  - Equivale a um double (por padrão)
  - Sem problema para representar inteiros

```
for i = 1, 10 do
   if i % 2 == 0 then
      print (tostring(i).." é par.")
   else
      print (tostring(i).." é ímpar.")
   end
end
```

Tipos

- *Strings*
  - Comprimento e conteúdo arbitrário
  - Conversão automática entre números e *strings*

```
local str1 = "As armas "
local str2 = "e os barões assinalados"
local concatenada = str1..str2

print ("1.234" + 4.321) --> 5.555
print ("Valor: "..171) --> "Valor: 171"
```

Tipos

- Tabelas (*tables*)
  - O único tipo “composto” em Lua
  - *Array* associativo: valores de qualquer tipo indexados por chaves de qualquer tipo
  - Variáveis contém apenas referências para tabelas!

Tipos

- Tabelas (*tables*)

```
local t = {
   nome = "Fulano",
   idade = 33,
   [5] = "cinco",
   [false] = "blá",
   pais = { pai = "Ivo", mae = "Ana"}
}

print (t["nome"]) --> "Fulano"
print (t.nome) --> "Fulano"
print (t[5]) --> "cinco"
print (t[false]) --> "blá"
print (t["pais"]["pai"]) --> "Ivo"
print (t.pais.mae) --> "Ana"
```

Tipos

- Tabelas (*tables*)

```
local t = {
   nome = "Fulano",
   idade = 33,
   [5] = "cinco",
   [false] = "blá",
   pais = { pai = "Ivo", mae = "Ana"}
}

print (t["nome"]) --> "Fulano"
print (t.nome) --> "Fulano"
print (t[5]) --> "cinco"
print (t[false]) --> "blá"
print (t["pais"]["pai"]) --> "Ivo"
print (t.pais.mae) --> "Ana"
```

Tipos

- Tabelas (*tables*)
  - Usadas como listas/*arrays*: índice inteiro é implicitamente adicionado

```
inventory = { "LaserGun",
              "Boots",
              "AstroSuit" }

print (inventory[1]) --> "LaserGun"
print (inventory[3]) --> "AstroSuit"

print (inventory[0]) --> nil
print (inventory[4]) --> nil

print (#inventory) --> 3
```

Tipos

- Tabelas (*tables*)
  - Também possuem *synctatic sugar* quando usadas como único parâmetro de uma função

```
-- Isso:
ProcessItems ({"LaserGun", "Boots", "AstroSuit"})

-- ...é o mesmo que isso:
ProcessItems {"LaserGun", "Boots", "AstroSuit"}
```

Tipos

- Funções

```
function Soma (a, b)
    return a + b
end

print (Soma(10, 20)) --> 30


function SomaSubtrai (a, b)
    return a + b, a - b
end

a, b = SomaSubtrai (10, 20)
print (a, b) --> 30, -10
```

Tipos

- Funções
  - São um tipo como outro qualquer

```
hero = {
   name = "Freddy Hardest",
   strength = 13,
   attack = function()
               print "I'm attacking!!!"
            end
}

hero.attack()
```

Tipos

- Outros
  - *Threads*: não é *bem* o que você está pensando
    - Mas veja, entre outros, o Lua Lanes http://kotisivu.dnainternet.net/askok/bin/lanes
  - *Userdata*: tipo definido pelo usuário, em C/C++

Iteradores e o `for` genérico

- Já vimos um exemplo do "`for` padrão" antes...

```
for i = 1, 10 do
   if i % 2 == 0 then
      print (tostring(i).." é par.")
   else
      print (tostring(i).." é ímpar.")
   end
end
```

- Mas como, por exemplo, fazer um laço que percorra todos os elementos de uma tabela?

Iteradores e o `for` genérico

- Iteradores!
  - Basicamente, uma função que retorna o próximo elemento de uma seqüência
  - Trabalha junto com o `for` genérico

```
hero = {
   name = "Freddy Hardest",
   speed = 3,
   strength = 12,
   intelligence = 7
}

for k, v in pairs (t) do
   print (k, "-->", v)
end
```

Iteradores e o `for` genérico

- `ipairs()`
  - Para percorrer apenas índices numéricos (de 1 até o primeiro buraco)

```
inventory = {
   "LaserGun",
   "Boots",
   "AstroSuit"
}

for k, v in ipairs (inventory) do
   print ("Item: "..v)
end
```

Iteradores e o `for` genérico

- Outros iteradores na biblioteca padrão
  - `io.lines ("some_file.txt")`
  - `string.gmatch (some_string, regexp)`

Iteradores e o `for` genérico

- Nós também podemos escrever nossos próprios iteradores
  - Como iterar apenas pelas armas?

```
inventory = {
   { name = "LaserGun", type = "weapon" },
   { name = "Boots", type = "armor” },
   { name = "AstroSuit", type = "armor” },
   { name = "Chainsaw", type = "weapon" },
}

for k, weapon in weapons (inventory) do
   print ("Weapon: "..weapon.name)
end
```

Metatabelas

- Vetores 3D em Lua...

```
local v1 = { 1, 2, 3 }
local v2 = { 4, 5, 6 }

print (v1)

local v3 = v1 + v2
```

Metatabelas

- Se tentamos fazer algo que Lua não sabe fazer, obtemos um erro
  - A menos que ensinemos Lua fazer o queremos!
- Uma *metatable* é uma tabela associada a outra tabela
  - Indica o que fazer quando tentamos realizar alguma operação que de outra forma seria inválida
    - Por exemplo, como somar ou imprimir um vetor 3D

Metatabelas

- Uma metatabela é uma tabela normal...
  - Com metamétodos!

```
VectorMetatable = {
    __tostring =
      function(v)
         return "("..v[1]..", "..v[2]
                ..", "..v[3]..")"
      end,

    __add =
      function(v1, v2)
         local t = { v1[1]+v2[1], v1[2]+v2[2],
                     v1[3]+v2[3] }
         setmetatable (t, VectorMetatable)
         return t
      end
}
```

Metatabelas

- A mágica acontece quando associamos a metatabela a uma tabela que representa um vetor 3D

```
v1 = { 1, 2, 0 }
v2 = { 5, 2, 3 }

setmetatable (v1, VectorMetatable)
setmetatable (v2, VectorMetatable)

print (v1 + v2)
```

Rudimentos de OO em Lua

- Antes de mais nada, como OO é implementada em uma linguagem como C++?

Rudimentos de OO em Lua

- Sem querer ser repetitivo, mas...
  - Lua não tem suporte “nativo” a OO
  - Lua não força políticas
  - Várias formas diferentes de OO podem ser implementadas!
- Aqui, veremos apenas uma forma simples, básica e incompleta
  - Que nem pode ser chamada de OO de verdade
    - Não veremos herança, por exemplo!

Rudimentos de OO em Lua

- Em Lua
  - Objetos são tabelas
  - Atributos são valores armazenados na tabela
  - Métodos são funções armazenadas na tabela
    - Parâmetro `self` (equivalente ao ponteiro `this` no C++) deve ser passado explicitamente
  - Chamada de métodos: mais *synctatic sugar*

Rudimentos de OO em Lua

```
hero = {
   name = "Freddy",
   strength = 12,
   sayHello =
      function (self)
         print ("Hi, my name is
"..self.name.." and my strength is "

..tostring(self.strength).."."")
      end
}

hero.sayHello (hero)

hero:sayHello()
```

Rudimentos de OO em Lua

```
hero = {
   name = "Freddy",
   strength = 12,
   sayHello =
      function (self)
         print ("Hi, my name is
"..self.name.." and my strength is "

..tostring(self.strength).."."")
      end
}

hero.sayHello (hero)
```

```
hero:sayHello()
```

Rudimentos de OO em Lua

- Até aqui, não temos o conceito de classe
- Classes podem ser implementadas através de metatables
  - Campo `__index`

Estendendo e Integrando

Possibilidades

- Arquivos de configuração/dados em Lua
  - Comentários, funções, `for`, `os.getenv()`...
- Permitir que o usuário estenda, modifique, personalize nosso aplicativo
- Prover novas funcionalidades para *scripts* em Lua

A API C de Lua

- Interpretador Lua é baseado numa pilha
- Mas com acesso “livre”
  - Índices positivos: posição absoluta (1 representa o primeiro item *push*ado)
  - Índices negativos: similar, mas contando do topo (-1 é o elemento no topo da pilha)
- Pseudo-Índices
  - LUA_GLOBALSINDEX

A API C de Lua

- Executando uma “*string* Lua”

```
lua_State* ls = luaL_newstate();
luaL_openlibs (ls);

luaL_dostring ("print ('Bla!')");

lua_close (ls);
```

A API C de Lua

- Implementando uma função em C
  - Implementa uma `lua_CFunction`
    - `int Func (lua_State* ls)`
  - Troca parâmetros e valores de retorno com o "lado Lua" via pilha
  - Retorna o número de valores de retorno
  - `lua_push*(), luato*()`
- Exemplo
  - Função que recebe dois parâmetros numéricos e retorna a sua soma e produto

A API C de Lua

```
int soma_prod (lua_State* ls)
{
   lua_Number a = lua_tonumber (ls, -1);
   lua_Number b = lua_tonumber (ls, -2);

   lua_pushnumber (ls, a + b);
   lua_pushnumber (ls, a * b);

   return 2;
}
```

A API C de Lua

- É preciso ainda “registrar” a função C num `lua_State`.

```
lua_pushstring (ls, "SomaProd");
lua_pushcfunction (ls, soma_prod);
lua_settable (ls, LUA_GLOBALSINDEX);
```

- A função C `soma_prod()` é atribuída ao índice `SomaProd` da tabela de globais
  - Ou seja: em Lua, temos uma nova função global chamada `SomaProd()`

A API C de Lua

- E para usar “arquivos de configuração em Lua”?
  - Na mesma linha
    - `lua_gettable()`

Ferramentas...

- Usar a API C de Lua não é coisa de outro mundo...
- Mas a vida pode ser melhor!
  - (Ou pior! Tudo depende do ponto-de-vista!)
- Ferramentas podem auxiliar na integração de C/C++ com Lua

LuaBind

- Na linha de Boost.Python
  - *Templates*, *templates*, *templates*...
  - Sem necessidade de uma ferramenta externa
  - Complexo
    - Projeto órfão: apenas Lua 5.0 suportada
    - Atualização [30/abril/02009]: Parece que o projeto está ativo de novo!
- http://luabind.sourceforge.net

LuaBind

- Exemplo básico

```
void hello()
{
   std::cout << "Hello, World\n!";
}

// ...
using namesace luabind;
open (ls);

module(ls)
[
   def ("Hello", hello)
];
```

LuaBind

- Uma classe em C++...

```
class TestClass
{
   public:
      TestClass(const std::string& s)
         : string_(s)
      { }

      void printString()
      {
         std::cout << string_ << "\n";
      }

   private:
      std::string string_;
};
```

LuaBind

- ...exportada para Lua

```
Module (ls)
[
   class_<TestClass>("TestClass")
      .def(constructor<const std::string&>())
      .def("print_string", &TestClass::printString)
];
```

tolua e tolua++

- Ferramenta externa
- Processa um arquivo “ .h limpo”
- Gera o código C que faz a exportação
- Adiciona um mecanismo OO a Lua
- http://www.tecgraf.puc-rio.br/~celes/tolua
- tolua++
  - Versão com algumas melhorias (em particular, sabe o que são `std::strings`)
  - http://www.codenix.com/~tolua

tolua e tolua++

- Exemplo
  - `Character`, `hero`, “My name is Freddy”...
    - De novo!

tolua e tolua++

- Para “arquivos de configuração em Lua”
  - Nenhuma funcionalidade específica
  - Mas até que dá prá se virar com algumas coisas fáceis de implementar

```
DoString ("_number_ = GetFavoriteNumber('lmb')");
double favoriteNumber = GetGlobal("_number_");
```

Diluculum

- Foco em “arquivos de configuração em Lua”
  - Incluindo bom suporte a chamada de funções Lua a partir de C++
- Suporte à exportação de funções e classes C++ para Lua
  - Intrusivo
  - Limitado
  - Com implementação, ahm, simples (macros)
- http://www.stackedboxes.org/Projects/Diluculum

Diluculum

- Principais classes
  - `Diluculum::LuaState`
    - Um interpretador Lua
  - `Diluculum::LuaValue`
    - Pode conter uma *string*, número, tabela, função...
  - `Diluculum::LuaVariable`
    - Referência a uma variável dentro de um `LuaState`

Diluculum

- Arquivos de configuração em Lua

```
Diluculum::LuaState ls;
ls.doFile ("config.lua");

double favePi =
   ls["FavoritePiApproximation"].value().asNumber();

std::string color =
   ls["FavoriteColor"].value().asString()

if (ls["UserInfo"]["Name"].value() != "lmb")
{ ... }
```

- Variáveis Lua têm acesso de leitura e escrita

Diluculum

- Exportando funções para Lua

```
Diluculum::LuaValueList
Func (const Diluculum::LuaValueList& params)
{ /* ... */ }

// ...

DILUCULUM_WRAP_FUNCTION (Func);

Diluculum::LuaState ls;

ls["MyFunction"] = DILUCULUM_WRAPPER_FUNCTION (Func);

ls.doString ("a, b, c = Func (4.5)");
```

Diluculum

- Exportando classes para Lua
  - Limitações similares
  - Construtor recebendo um `LuaValueList`
  - Métodos exportados devem receber e retornar um `LuaValueList`

```
DILUCULUM_BEGIN_CLASS (MyClass);
   DILUCULUM_CLASS_METHOD (MyClass, myMethod);
DILUCULUM_END_CLASS (MyClass);

DILUCULUM_REGISTER_CLASS (ls["MyClass"], MyClass);

ls.doString ("obj = MyClass.new (3)");
ls.doString ("obj:myMethod()");
```

Diluculum

- Exportando objetos para Lua
  - Permite, em Lua, chamar métodos de um objeto instanciado no lado C++
  - Objeto não será “limpado” por Lua

```
MyClass obj (params);

DILUCULUM_REGISTER_OBJECT (ls["obj"], MyClass, o);

ls.doString ("o:myMethod()");
```

Diluculum

- Módulos em bibliotecas dinâmicas
  - Em Lua, `require("Modulo”)`

```
DILUCULUM_BEGIN_MODULE (MyFineModule);
   DILUCULUM_MODULE_ADD_CLASS (ValueBox, "ValueBox");
   DILUCULUM_MODULE_ADD_FUNCTION(
      DILUCULUM_WRAPPER_FUNCTION(MyFunction),
      "MyFunction");
DILUCULUM_END_MODULE();
```

Referências

Referências

- http://www.lua.org
  - Manual
  - Programming in Lua
  - Lista de discussão
  - Wiki