A Linguagem Lua e suas Aplicações em Jogos

Waldemar Celes, Luiz Henrique de Figueiredo, Roberto Ierusalimschy

1 Introdução

Uma pesquisa realizada em setembro de 2003 pela gamedev.net — um importante *site* para programadores de jogos — revelou que a grande maioria dos jogos (72%) é desenvolvida com o auxílio de uma linguagem de *script*.

Embora as linguagens de *script* não sejam definidas muito precisamente, elas apresentam um conjunto de características comuns típicas. Em geral, as linguagens de *script* são linguagens interpretadas, têm tipagem dinâmica e gerência automática de memória, e fornecem facilidades para construção de estruturas de dados dinâmicas e manipulação de cadeias de caracteres. Tipicamente, essas linguagens funcionam acopladas a programas hospedeiros implementados em linguagens compiladas tradicionais como C e C++. Uma outra característica importante de linguagens de *script* é que elas devem ser seguras, não sendo possível acessar serviços não autorizados do programa hospedeiro. A combinação dessas características resulta numa excelente ferramenta para o desenvolvimento de jogos.

Acoplar uma linguagem de *script* em um jogo traz vários benefícios. A linguagem de *script* pode ser usada para efetivamente implementar o *script* do jogo, para definir objetos e seus comportamentos, para gerenciar os algoritmos de inteligência artificial e controlar os personagens, e ainda para tratar os eventos de entrada e descrever a interface com o usuário. Uma linguagem de *script* também desempenha um papel importante nas etapas de prototipação, teste, depuração e análise de adequação do jogo. A escolha de uma linguagem de *script* simples permite ainda que seja dado a roteiristas e artistas acesso programável ao jogo, a fim de que eles que possam experimentar novas idéias e variações. Esses profissionais conduzem a maior parte do desenvolvimento real do jogo mas não são em geral programadores profissionais e não estão familiarizados com técnicas sofisticadas de programação.

A mesma pesquisa mencionada acima revelou que Lua é atualmente a linguagem de *script* mais utilizada no desenvolvimento de jogos (20% dos jogos são desenvolvidos com Lua, enquanto somente 7% usam Python, a segunda linguagem de *script* mais citada na pesquisa). De fato, devido ao seu pequeno tamanho, bom desempenho, portabilidade e facilidade de integração, Lua tem sido amplamente utilizada na indústria de jogos. Empresas com LucasArts, BioWare, Microsoft, Relic Entertainment, Absolute Studios e Monkeystone Games desenvolvem jogos usando Lua.

Lua é uma linguagem de *script* extensível, projetada para oferecer meta-mecanismos que possibilitam a construção de mecanismos mais específicos. Com isso, é fácil adequar Lua às necessidades da aplicação, sem comprometer as suas características básicas, mantidas desde a sua criação, tais como portabilidade, pequeno tamanho e simplicidade. Em particular, os programadores dos jogos podem fornecer abstrações adequadas para roteiristas e artistas, simplificando as tarefas desses.

Neste tutorial, discutiremos o uso da linguagem Lua no desenvolvimento de jogos. Vamos apresentar os principais mecanismos de programação oferecidos pela linguagem e a interface para a integração com programas hospedeiros escrito em C ou C++. Discutiremos ainda como estes mecanismos podem ser usados para a construção de jogos mais flexíveis, que fornecem acesso programável aos serviços implementados pelo programa hospedeiro.

2 Linguagens de extensão

Vamos classificar as linguagens de extensão ou de *script* segundo a sua complexidade:

- *Linguagens de configuração* servem para selecionar preferências e são tipicamente uma lista de valores associados a variáveis. Um exemplo típico são os arquivos `.ini` do Windows.
- *Linguagens de macros* servem para automação de tarefas e são tipicamente uma lista de ações primitivas, com muito pouco ou nenhum controle de fluxo. Um exemplo típico são os arquivos de automação de conexões internet via modem.
- *Linguagens embutidas* permitem acesso programável aos serviços da aplicação, com fluxo de controle completo e funções definidas pelo usuário a partir de primitivas exportadas pela aplicação. Essas linguagens são linguagens completas, muitas vezes variantes simplificadas de linguagens tradicionais como Lisp ou C.

Independente da complexidade da linguagem, a adoção de uma linguagem de extensão é uma técnica poderosa de desenvolvimento de *software*, pois permite que muitos aspectos da aplicação sejam controlados externamente, a partir de arquivos textos facilmente editados pelo programador ou pelo usuário, sem necessidade de recompilar a aplicação. Isso torna o desenvolvimento mais ágil, pois permite que partes importantes da aplicação sejam desenvolvidas por outros membros da equipe que não são programadores profissionais (no caso de jogos, animadores e artistas).

Além disso, a adoção de uma *mesma* linguagem para várias tarefas na aplicação permite um aumento importante de produtividade global, pois não é necessário definir, documentar e manter vários formatos diferentes. Para o usuário da aplicação, isso se traduz no re-aproveitamento automático (e inconsciente) dos conceitos aprendidos para realizar as várias tarefas, encorajando a exploração.

3 A linguagem Lua

A linguagem Lua é uma linguagem de programação poderosa e leve, projetada para estender aplicações. Isso quer dizer que ela foi projetada para ser acoplada a programas maiores que precisem ler e executar programas escritos pelos usuários. Na classificação da Seção 2, Lua é uma linguagem embutida, com sintaxe semelhante à de Pascal mas com construções modernas, como funções anônimas, inspiradas no paradigma funcional, e poderosos construtores de dados. Isso faz com que Lua seja uma linguagem de grande expressão com uma sintaxe familiar.

3.1 História

Lua foi projetada e implementada no Tecgraf, o Grupo de Computação Gráfica da PUC-Rio. A primeira versão de Lua (1.0) é de julho de 1993. A primeira versão pública (1.1) é de julho de 1994. A versão atual é a 5.0.2, lançada em março de 2004 para corrigir pequenas falhas na versão 5.0, de abril de 2003. A versão 5.1 está em desenvolvimento no momento. Este texto trata da versão oficial atual (5.0.2).

A motivação para a criação de Lua no Tecgraf foi a necessidade crescente das suas aplicações serem configuráveis externamente pelos usuários. Isso quer dizer que diversos aspectos essenciais das aplicações podem ser modificados sem recompilar a aplicação. Desde o início, nas aplicações criadas no Tecgraf, esse tipo de configuração era muito mais do que simplesmente poder escolher a cor da janela ou o tipo de fonte de texto: era necessário poder tomar decisões em tempo de execução que somente os usuários sabiam quais eram. Sendo assim, era necessário fornecer algum tipo de programação para os usuários finais. Um outro tipo de configuração era a descrição de

complexos relatórios e análises feitas pela Petrobras por encomenda ao Tecgraf. Mais uma vez, essa descrição não podia estar congelada dentro da aplicação pois cada usuário tinha uma necessidade diferente e que mudava a cada tarefa. O Tecgraf tinha portanto (e ainda tem) forte demanda para aplicações que fossem configuráveis externamente, tanto descrevendo que decisões deveriam ser tomadas quanto descrevendo quais dados seriam usados e como eles seriam usados.

Após projetar e usar com sucesso duas pequenas linguagens específicas para cada uma dessas tarefas, o Tecgraf decidiu investir na criação de uma linguagem única que pudesse atender a todas as necessidades de configuração das suas aplicações. Assim nasceu Lua: uma linguagem procedural simples com poderosas construções para descrição de dados. Desde então Lua tem sido usada em inúmeros projetos no Tecgraf.

A versão seguinte de Lua (2.1) foi lançada em fevereiro de 1995 e trouxe maior poder de expressão, introduzindo a noção de semântica extensível: passou a ser possível programar o que fazer em casos excepcionais, como quando um campo não existe numa tabela. Ter uma semântica extensível é desde então uma característica marcante de Lua.

A versão 2.1 também foi a primeira a ser completamente livre; as versões anteriores eram livres somente para aplicações acadêmicas. Aliada à portabilidade e à eficiência da implementação, a falta de restrições de uso foi um dos fatores importantes na adoção de Lua em inúmeros projetos no mundo todo.

A primeira notícia que tivemos do uso de Lua em jogos foi em 1997 quando a LucasArts adotou Lua como a sua linguagem de *script* no lugar de SCUMM no jogo "Grim Fandango". A adoção de Lua nesse jogo foi uma consequência direta da publicação de um artigo de divulgação sobre Lua na revista *Dr. Dobb's Journal*, em junho de 1996. Desde então, Lua tem sido cada vez mais usada em jogos, uma área longe das áreas que motivaram a sua criação!

Atualmente, Lua tem uma comunidade ativa de programadores. A lista de discussão tem mais de 750 assinantes do mundo todo. Além do *site* oficial de Lua (lua.org), há também um ativo *site* mantido por usuários (lua-users.org), uma sala de bate-papo, um *web forum* e um repositório (luaforge.net).

3.2 Características básicas

Lua é uma linguagem de extensão projetada para dar suporte à programação procedural, oferecendo facilidades para descrição de dados. No contexto da programação de jogos, isso significa que Lua possibilita combinar a descrição de objetos e a programação de seus comportamentos num mesmo contexto. Lua é uma biblioteca implementada em C, podendo ser compilada em qualquer plataforma que tenha um compilador C padrão. Lua também pode ser compilada sem alterações como uma biblioteca C++. No que se segue, toda referência a C deve ser entendida como uma referência a C++ também. Em alguns poucos lugares trataremos exclusivamente de C++.

Por ser uma linguagem de extensão, Lua trabalha acoplada a uma aplicação hospedeira (*host*). Essa aplicação pode criar e ler valores armazenados em Lua, executar funções de Lua e registrar funções C no ambiente de Lua. As funções C registradas em Lua, por sua vez, podem ser invocadas de programas Lua. Dessa forma, podemos conciliar as facilidades de uma linguagem de *script* oferecidas por Lua com a eficiência das linguagens C e C++. A distribuição da linguagem Lua inclui um programa hospedeiro, `lua.c`, que pode ser usado para executar *scripts* Lua interativamente ou em *batch*. Neste tutorial, no entanto, focaremos no uso de Lua acoplada a programas de jogos.

Para que uma aplicação tenha acesso a Lua, precisamos abrir a biblioteca conforme será discutido na Seção 3.9. Ao final, a aplicação deve fechar a biblioteca. Após a abertura da biblioteca, a aplicação pode usar os recursos oferecidos por Lua, como por exemplo, executar *scripts* Lua e acessar dados armazenados em Lua. Antes de detalharmos como isso pode ser feito dentro do código da aplicação, temos que aprender como escrever *scripts* em Lua. Programas ou *scripts* Lua são armazenados em arquivos (usualmente com extensão `.lua`) ou em *strings* da aplicação. Um arquivo ou

string com código Lua caracteriza o que chamamos de *chunk*, que é simplesmente uma seqüência de comandos Lua. Daremos a seguir uma breve introdução às principais características da linguagem Lua.

3.3 Variáveis e tipos

Em Lua, as variáveis não têm tipos associados a elas: os tipos estão associados aos valores armazenados nas variáveis. Dessa forma, uma mesma variável pode num momento armazenar um valor de um tipo e depois passar a armazenar o valor de outro tipo (naturalmente, a variável deixa de armazenar o valor inicial). O trecho de código abaixo ilustra o uso de variáveis armazenando diferentes valores:

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

(Note a forma dos comentários em Lua: começam com `--` e vão até o final da linha.)

O fato de o tipo estar associado ao valor, e não à variável, dá grande flexibilidade à linguagem. Podemos, por exemplo, criar conjuntos de valores heterogêneos naturalmente, pois o poliformismo é intrínseco à linguagem. Por outro lado, a verificação de tipo só pode ser feita em tempo de execução. Assim, se tentarmos somar duas variáveis cujos valores não são númericos, um erro será reportado, mas somente quando a soma for executada. Como pretendemos, ao acoplar Lua ao jogo, exportar serviços da aplicação, devemos prever um tratamento adequado desses erros de execução. Um usuário de um jogo pode codificar uma configuração inválida e isso deve ser tratado de maneira adequada, sem necessariamente abortar o jogo.

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 (a um bloco ou *chunk*), devemos declará-la previamente usando a palavra `local`. Por exemplo:

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

Os valores em Lua podem ser de oito tipos:

- *nil*: o valor `nil` indica ausência de valor. Variáveis não inicializadas contêm o valor *nil*. O valor *nil* também é interpretado como *falso* numa expressão booleana.
- *boolean*: valor booleano, podendo ser falso (`false`) ou verdadeiro (`true`);
- *number*: valor numérico. Lua não diferencia valor inteiro de valor real; só existe um tipo para representar números;
- *string*: valor cadeia de caracteres. Uma constante *string* pode ser delimitada por aspas duplas (`"..."`), aspas simples (`'...'`), ou duplo colchetes (`[[...]]`).
- *table*: vetor associativo (a ser detalhado na Seção 3.6);
- *function*: função escrita em Lua ou escrita em C e registrada em Lua;
- *userdata*: dado do *host*, representado por um ponteiro `void*`;
- *thread*: linha de execução, que será apresentado na Seção 3.8, quando descrevermos o uso de co-rotinas em Lua.

3.4 Operadores e controladores de fluxo

A linguagem Lua oferece os operadores comumente encontrados em linguagens de programação. Os operadores aritméticos são os usuais: `+` (adição), `-` (subtração), `*` (multiplicação), `/` (divisão), `^` (exponenciação) e `-` unário (negação). Os operadores relacionais resultam num valor booleano e incluem: `<` (menor que), `>` (maior que), `<=` (menor ou igual que), `>=` (maior ou igual que), `==` (igualdade), `~=` (diferença). Os operadores lógicos servem para combinar valores booleanos e são dados por: `and` (e), `or` (ou), `not` (negação). Existe ainda o operador `..` para concatenação de *strings*.

Os operadores lógicos `and` e `or` são úteis também na avaliação de expressões. Esses operadores combinam duas expressões e fazem a avaliação da segunda expressão apenas quando necessário. Além disso, o resultado de um `and` ou `or` é último valor usado na sua avaliação. Dessa forma, é válido e muito útil usar construções como as seguintes:

```
x = v or w
y = a and b or c
```

Na primeira atribuição acima, `x` passa a armazenar o valor de `v`, se esse for diferente de falso (`false` ou `nil`); caso contrário, passa a armazenar o valor de `w`. A segunda atribuição, que deve ser lida como `(a and b) or c`, é equivalente à expressão `y = a ? b : c` em C, desde que `b` não resulte em *falso*.

A linguagem Lua oferece os controladores de fluxo comuns às linguagens procedurais. As construções para tomada de decisão são as variantes usuais de `if ... then ... else`:

```
if expr then
   ...
end
```

```
if expr then
   ...
else
  ...
end
```

```
if expr1 then
   ...
elseif expr2 then
   ...
else
  ...
end
```

As construções para execução iterativa podem ter o seu teste no início (`while`) ou no fim (`repeat`):

```
while expr do
   ...
end
```

```
repeat
   ...
until expr
```

Lua oferece ainda a construção de laços com `for`. O `for` numérico tem a seguinte forma:

```
for var = expr_inicial, expr_final, expr_incremento do
   ...
end
```

Nessa construção, a variável `var` é automática e local ao laço, isto é, não precisa ser explicitamente declarada e só existe dentro do laço. As expressões inicial, final e de incremento são avaliadas uma única vez, antes do início da execução do bloco de comandos do laço. A expressão de incremento, se omitida, vale `1`.

Um laço de `for` pode ainda aparecer na sua forma genérica, que permite a construção de diversos tipos de iteradores especializados. O trecho de código abaixo, por exemplo, usa um `for` genérico para ler e imprimir cada linha do arquivo de entrada corrente, usando funções pré-definidas na biblioteca de entrada e saída (uma discussão de como se constrói iteradores foge do escopo desse tutorial).

```
for line in io.lines() do
   io.write(line,"\n")
end
```

A execução de laços `while`, `repeat` e `for` pode ser interrompida usando o comando `break`.

3.5 Funções

Funções em Lua são valores de primeira classe. Isso significa que, como qualquer outro valor, uma função pode ser criada, armazenada em uma variável (local ou global) ou campo de tabela e passada adiante como parâmetro ou valor de retorno de uma outra função. Uma função pode receber zero ou mais valores. A lista de parâmetros é especificada da maneira usual: entre os parênteses que seguem o nome da função. Como exemplo simples (tradicional, mas útil somente para fins didáticos), consideremos a definição da função recursiva abaixo para o cálculo do fatorial de um número inteiro:

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

As funções em Lua não têm nome; elas são sempre *anônimas*. O código acima é apenas uma maneira conveniente de definir uma função e atribuí-la a uma variável global, e é equivalente a

```
fat = function (n) ... end    -- função anônima atribuída à variável fat
```

Para testar essa função, podemos usar a função `print` da biblioteca padrão de Lua que imprime um valor na tela (nesse caso, 120):

```
local a = 5
print(fat(a))
```

Lua permite atribuições múltiplas. É válido, por exemplo, escrever `x,y = y,x`, que troca os valores de `x` e `y` sem a necessidade de usar uma variável temporária, da mesma forma que é válido escrever `a,b = 1,2`. Ainda de forma análoga, é válido escrever `a,b = f()` — os valores retornados por `f` serão atribuídos às variáveis `a` e `b`. (Sim, Lua permite que uma função retorne múltiplos valores, escrevendo `return expr1, expr2, ...`. Isso evita em grande parte a necessidade de passagem de parâmetros por referência; em Lua todos os parâmetros são passados por valor.)

Se o número de variáveis numa atribuição múltipla for maior que o número de valores resultantes à direita do sinal de igualdade, as variáveis excedentes recebem o valor *nil*. Se o número de valores for maior, os valores excedentes são descartados. Esse mesmo *ajuste* ocorre ao se chamar funções: argumentos ausentes recebem o valor *nil*; argumentos extras são ignorados (exceto quando a função aceita um número variável de argumentos).

Funções podem ser criadas localmente dentro de outras funções, e depois retornadas ou armazenadas em uma tabela. Uma função pode ainda acessar variáveis locais do escopo acima. Considere por exemplo o trecho de código abaixo:

```
function counter ()
   local i = 0
   return function ()
             i = i+1
             return i
          end
end

local c = counter()
print(c())
print(c())
```

A variável `c` armazena uma instância da função anônima criada (e retornada) em `counter`. Essa função usa a variável local `i` declarada em `counter`. Assim, cada vez que executamos a função armazenada em `c`, recebemos o valor da variável `i` incrementado de uma unidade. Se fizermos um paralelo com C, a variável da função `counter` funciona como uma *variável estática* para a função `c`. Se executarmos a função `counter` novamente, teremos como retorno uma *outra* função que, se chamada, retornará da primeira vez o valor 1, depois 2, e assim por diante. Essa segunda função é diferente da primeira: embora ambas façam a mesma coisa, elas o fazem de maneira independente. É importante observar que a função anônima é única e seu código é gerado uma única vez. A função `counter` retorna o que chamamos de *closure*, que guarda uma referência para a função anônima e uma lista com os valores das variáveis do escopo superior usadas, entre outras coisas. A existência de *closures* com escopo léxico (acesso a variáveis do escopo superior) permite que Lua seja usada como linguagem funcional, o que dá grande flexibilidade de programação.

Por fim, considere o trecho de código abaixo:

```
a = 2

function f ( )
   ...
end

local b = 3

function g ( )
   local x = b
   ...
end
```

Todo *chunk* representa o corpo de uma função anônima que é retornada quando o *chunk* é carregado. No exemplo acima, atribui-se três variáveis globais (`a`, `f` e `g`) e declara-se uma variável local `b`. Fazendo uma analogia com programação em C, temos que a variável `b` funciona como uma variável estática do módulo. Na verdade, em Lua, `b` é uma variável local da função anônima caracterizada pelo *chunk*, acessada de dentro da função que foi armazenada na variável global `g`.

3.6 Tabelas e objetos

O tipo *table* representa um vetor associativo, implementado internamente com o uso de uma eficiente combinação de *array* e *hash* (tabela de dispersão). As tabelas são a única forma de estruturação de dados em Lua. Todas as estruturas de dados comumente encontradas em programação (tais

como vetores, listas, filas, conjuntos e *hash*) podem ser eficientemente (e facilmente) implementadas com o uso de tabelas. Uma tabela em Lua é criada pela expressão `{ }`. Se uma variável armazena um valor do tipo tabela, essa variável pode ser indexada por qualquer outro valor (exceto *nil*). O valor armazenado em cada índice da tabela também pode ser de qualquer tipo (incluindo *nil*). O valor associado a um índice da tabela não inicializado tem o valor *nil*. O trecho de código abaixo ilustra a criação de uma tabela e a atribuição de alguns campos:

```
local t = {}   -- cria nova tabela
t[1] = 4       -- armazena 4 no índice 1
t[2] = "alo"   -- armazena "alo" no índice 2
t["alo"] = 5   -- armazena 5 no índice "alo"
t[t[2]] = 0    -- armazena 0 no índice "alo" (sobrescrevendo)
```

Lua oferece uma sintaxe simplificada quando o índice é uma *string* simples (desde que a *string* não seja uma palavra reservada na sintaxe de Lua). Assim, a atribuição acima `t["alo"] = 5` pode ser escrita simplesmente por `t.alo = 5`. Lua permite ainda que campos da tabela sejam inicializados na criação. Dessa forma, as três primeiras linhas do código acima podem ser substituídas por:

```
local t = {4,"alo"; alo=5}
```

A biblioteca padrão de Lua oferece duas funções que permitem iterar sobre os elementos armazenados na tabela. A função `ipairs` itera sobre todos os índices numéricos armazenados na tabela. Assim,

```
for i,v in ipairs(t) do ... end
```

itera sobre os pares `(1,t[1])`, `(2,t[2])`, ..., até que o primeiro índice com valor associado igual a *nil* seja encontrado.

A função `pairs` permite iterar sobre todos os pares armazenados na tabela, independente do tipo associado à chave:

```
for k,v in pairs(t) do ... end
```

Nesse caso, a ordem em que os pares `k,v` são reportados é indefinida.

Como índices (chaves) e valores de uma tabela podem ser valores quaisquer, podemos naturalmente usar tabelas como índices e valores. Com isso, podemos ter tabelas aninhadas (inclusive com ciclos). Considere a tabela abaixo, que especifica os parâmetros para a criação de uma janela de diálogo numa aplicação gráfica. Observe que funções podem também ser armazenadas em tabelas.

```
local w = {
   width = 640,
   height = 480,
   menu = {
           {label="Load",action=function () ... end},
           {label="Save",action=function () ... end},
   },
   ...
}
```

Lua permite ainda a especificação de um "construtor" na criação da tabela. Consideremos, por exemplo, a especificação de um ponto em 3D, dado pelas suas coordenadas. Em Lua, podemos escrever:

```
local p = Point{x=3.0,y=1.3,z=3.2}
```

O código acima é equivalente a

```
local p = Point({x=3.0,y=1.3,z=3.2})
```

isto é, uma tabela é criada e chama-se a função `Point` passando a nova tabela como parâmetro. Essa função pode ser o construtor do objeto sendo criado. Por exemplo, podemos usar a função para validar e inicializar campos do objeto.

```
function Point (self)
   self.x = tonumber(self.x) or 0.0
   self.x = tonumber(self.y) or 0.0
   self.x = tonumber(self.z) or 0.0
   return self
end
```

Assim, se na criação do objeto não forem especificados valores das coordenadas (ou se forem especificados valores não numéricos), a função inicializa esses valores com zero.

Lua oferece ainda um eficiente mecanismo para estendermos a sua semântica através do uso de *eventos*. Esse mecanismo permite, por exemplo, adotarmos uma programação orientada a objetos. Para estender a semântica de um objeto (tabela), devemos associar a ele uma outra tabela, chamada de *metatable*. Na *metatable*, podemos programar a ação que deve ser tomada quando ocorre um determinado evento. Por exemplo, a operação de soma não é especificada para tabelas; no entanto, podemos fazer com que dois objetos do nosso tipo `Point` acima possam ser somados, gerando um terceiro novo objeto do tipo `Point`. Para isso, devemos primeiro criar uma *metatable* com o comportamento da operação de soma definido:

```
local Point_metatable = {
   __add = function (p1,p2)
              return Point(p1.x+p2.x,p1.y+p2.y,p1.z+p2.z}
           end
}
```

Devemos reescrever o construtor de `Point` para definir a *metatable* de cada objeto criado:

```
function Point (self)
   self.x = tonumber(self.x) or 0.0
   self.x = tonumber(self.y) or 0.0
   self.x = tonumber(self.z) or 0.0
   setmetatable(self,Point_metatable)
   return self
end
```

Assim, definimos um objeto `Point` e podemos usá-lo de maneira transparente:

```
local p = Point{x=3.0,y=1.3,z=3.2}
local q = Point{x=4.2,y=1.0}
local r = p+q                      -- r.x=7.2, r.y=2.3, r.z=3.2
```

Além de *add*, podemos (re-)definir o comportamento quando da ocorrência dos seguintes eventos de operação aritmética: *sub* (subtração), *mul* (multiplicação), *div* (divisão), *pow* (exponeciação), *unm* (negação), *concat* (concatenação), *eq* (igualdade), *lt* (menor que), *le* (menor ou igual que).

Basta criar o campo adequado na *metatable*. (O nome do campo é o nome do evento precedido de `__`.)

Existem ainda dois eventos especiais cujos comportamentos podem ser programados: *index*, gerado quando tentamos acessar um índice não existente na tabela, e *newindex*, gerado quando tentamos atribuir um valor a um índice ainda não existente na tabela. Esses eventos podem ser usados para programar diferentes comportamentos. Por exemplo, podemos usar o evento *index* para delegar a uma outra tabela a busca do valor associado ao índice. Dessa forma, podemos programar nosso próprio mecanismo de herança. Se o objeto não tem o campo, retornamos o campo associado à sua "classe":

```
local Point_methods = {
    Print = function (self)
               print(self.x, self.y, self.z)
            end,
    ...
}
```

Na *metatable*, associamos a tabela acima ao campo `__index`:

```
local Point_metatable = {
   __index = Point_methods,
   __add = function (p1,p2)
              return Point(p1.x+p2.x,p1.y+p2.y,p1.z+p2.z}
           end
}
```

Podemos então acessar o "método" `Print` de nosso tipo:

```
local p = Point{x=3.0,y=1.3,z=3.2}
local q = Point{x=4.2,y=1.0}
local r = p+q
r.Print(r)
```

Para facilitar o uso e dar clareza ao código, a última linha do código acima pode ser escrita `r:Print()`, como se espera de uma chamada de método em C++ ou Java. Em Lua, a chamada de função da forma `t:meth(...)` é equivalente a `t.meth(t,...)`.

Se a delegação não for direta, podemos atribuir ao campo `__index` da *metatable* uma função que deve ser executada quando o evento ocorrer. Isto nos dá flexibilidade para, por exemplo, buscarmos o valor do campo num objeto em C!

3.7 Biblioteca padrão

A distribuição oficial de Lua inclui um conjunto de bibliotecas que implementam diversas funções importantes para a construção de programas. Com exceção das funções que pertencem ao que chamamos de biblioteca básica, as funções de cada biblioteca são agrupadas em tabelas. Assim, a tabela `string` agrupa as funções para manipulação de *strings*, a tabela `table` agrupa as funções para manipulação de tabelas e assim por diante. Listamos abaixo as bibliotecas padrão incluídas na distribuição. O manual de referência contém uma descrição detalhada das funções oferecidas.

Além da biblioteca básica, que oferece funções básicas para a programação em Lua (como `print`, `setmetatable`, `pairs`, que usamos acima), a distribuição inclui as seguintes bibliotecas:

- `string`: oferece funções para manipulação de *strings*. Destacamos o poderoso mecanismo de casamento de padrões (*pattern matching*) oferecido através das funções `string.find`, que

permite buscar a ocorrência de um padrão numa *string*, e `string.gsub`, que permite substituirmos ocorrência de um padrão por uma sequência de caracteres dentro de uma *string*.

- `table`: oferece funções para manipulação de tabelas, tais como funções para inserir um novo elemento (associado a um índice numérico) na tabela (`table.insert`), remover um elemento da tabela (`table.remove`) e ordenar os elementos armazenados em íncides numéricos de uma tabela (`table.sort`).

- `math`: oferece funções semelhantes às funções oferecidas pela biblioteca matemática de C, tais como `math.sqrt`, `math.sin`, `math.log`, etc.

- `io`: oferece funções para operações de entrada e saída, tais como abertura (`io.open`), fechamento de arquivos (`io.close`), leitura (`io.read`) e escrita (`io.write`). A biblioteca de `io` trabalha com o conceito de objeto. Um arquivo aberto é um objeto ao qual temos associado métodos. Assim, após o comando `f = io.open("entrada.txt","r")`, a variável `f` contém um objeto do tipo *arquivo*. De posse do objeto, podemos usar funções (`io.read(f,...)`) ou métodos (`f:read(...)`) para manipularmos o arquivo.

- `os`: oferece funções relacionadas ao sistema operacional, também análogas às funções oferecidas pela biblioteca C, tais como `os.clock`, `os.date`, `os.execute` (análoga a `system` de C).

- `debug`: oferece funções para depuração de códigos Lua. As funções oferecidas permitem, por exemplo, consultar o estado corrente da pilha de execução de Lua e os valores de variáveis locais em todos os níveis da pilha. Essa biblioteca oferece ainda mecanismos para cadastrar ações a serem tomadas a cada execução de uma linha de código, a cada chamada de função, etc., viabilizando a construção de interfaces de depuração. Assim, em vez de oferecer uma ferramenta de depuração, Lua oferece mecanismos para que tais ferramentas sejam facilmente construídas, direcionadas para o domínio da aplicação em questão.

Lua oferece ainda a biblioteca de co-rotinas, que discutiremos na próxima seção, dada a sua especial importância para a programação de jogos.

3.8 Co-rotinas

Co-rotinas são um poderoso mecanismo de programação para jogos. Uma co-rotina é semelhante a um *thread* num sistema de *multithreading*, no sentido de que temos uma linha de execução com seu próprio ambiente local (pilha de execução) compartilhando o ambiente global com outras co-rotinas. A grande diferença entre uma co-rotina e uma função é que a execução de uma co-rotina pode ser suspensa e retomada posteriormente (no ponto em que foi suspensa). A diferença entre co-rotinas e *threads* é que, conceitualmente, diferentes *threads* executam simulataneamente, enquanto que num sistema com co-rotinas, apenas uma co-rotina executa por vez.

As funções que manipulam co-rotinas estão agrupadas na tabela `coroutine`. Criamos uma co-rotina passando uma função (em geral, anômina) para a função de criação, que retorna um valor do tipo `thread`:

```
local c = coroutine.create(function () ... end)
print(type(c))      --> "thread"
```

Uma co-rotina pode estar em três diferentes estados: suspensa, executando e inativa. Imediatamente após a sua criação, uma co-rotina está no estado "suspensa". Para executar uma co-rotina, invocamos a função `coroutine.resume`. A execução de uma co-rotina começa pela execução da função passada como parâmetro na sua criação. Dentro do código da co-rotina, podemos suspender sua execução invocando a função `coroutine.yield`. Ao executar essa função, o controle volta

para o código que tinha dado `coroutine.resume` na co-rotina, restaurando todo o ambiente local. A co-rotina pode voltar a ser executada com uma outra chamada de `coroutine.resume`, e a execução é retomada logo após o último comando `coroutine.yield` executado. Do ponto de vista da co-rotina, uma chamada a `coroutine.yield` retorna quando a execução da co-rotina é retomada (via `coroutine.resume`). Lua oferece um mecanismo simples e versátil para troca de dados (mensagens) entre co-rotinas. Os argumentos de uma chamada a `coroutine.yield` são passados como valores de retorno da chamada a `coroutine.resume`. Simetricamente, os argumentos de `coroutine.resume` são passados como valores de retorno da função `coroutine.yield`.

Co-rotinas são muito úteis quando queremos implementar um procedimento de maneira incremental. Em jogos, onde temos um tempo limitado para executarmos nossas simulações, podemos implementar as simulações de forma incremental, executando os passos que são possíveis entre quadros da animação. Para ilustrar, vamos considerar um exemplo hipotético: um jogo tem que fazer a simulação do comportamento de diversos personagens. Para não favorecer um personagem em relação aos outros, podemos pensar em implementar a simulação de forma incremental, criando co-rotinas e suspendendo sua execução após um passo da simulação. Podemos prever então uma função que gerencia a execução das diversas simulações, executando cada uma passo a passo. Note que o uso de co-rotinas aqui é muito apropriado, pois cada simulação pode ser retomada a qualquer instante — a linguagem garante a restauração do seu ambiente local.

Começamos pela programação da simulação de cada personagem (ou grupo de personagens) encapsulada por co-rotina. Agrupamos as co-rotinas numa tabela e passamos essa tabela para um gerenciador das simulações. O gerenciador chama uma co-rotina por vez. Conforme ilustrado abaixo, o gerenciador pode, por sua vez, ser uma co-rotina gerenciada por um controle externo.

```
local simulators = {
 coroutine.create(function ()  ... end),   -- simulação 1
 coroutine.create(function ()  ... end),   -- simulação 2
 coroutine.create(function ()  ... end),   -- simulação 3
 ...
}

function manager ()
   while true do
      for i,v in pairs(simulators) do
         coroutine.resume(v)
      end
      coroutine.yield()  -- repassa para controlador externo
   end
end
```

3.9 Interface com C

Como Lua é uma linguagem para estender aplicações, ela não tem somente uma sintaxe e uma semântica: ela tem também uma API para comunicação com a aplicação. Essa API está descrita em `lua.h` e é formada por aproximadamente 80 funções C. (Não se assuste com esse número! A API é razoavelmente simples.)

O primeiro conceito na API é o *estado Lua*: a execução de um programa Lua e a comunicação de C com Lua se dão através de um estado Lua, que contém todas as variáveis e seus valores correntes. A aplicação pode criar mais de um estado Lua. Eles são todos completamente independentes uns dos outros. Por isso, cada função da API recebe como primeiro parâmetro o estado Lua sobre o qual ela deve operar. A única exceção à essa regra é a função `lua_open`, que cria um estado novo.

Um estado Lua existe até que ele seja fechado, com `lua_close`. Nesse momento, toda a memória usada pelo estado é liberada, e suas variáveis e valores desaparecem.

O principal mecanismo de comunicação entre Lua e C é uma *pilha virtual*. Nela, C põe valores a serem usados por Lua e vice-versa. A pilha pode armazenar valores Lua de qualquer tipo (nil, booleano, número, *string*, tabela, função, *userdata* e *thread*). Há portanto funções da API para por na pilha valores de cada um desses tipos. Há também funções da API para consultar o tipo de um valor que está na pilha e para convertê-lo para um valor C, quando isso faz sentido. (Não faz sentido converter uma tabela Lua para C porque C não tem tabelas. Mas faz sentido converter um número ou *string* para C.)

Como Lua tem coleta automática de lixo, é necessário estar atento para não usar valores obtidos de uma pilha inativa. O erro mais comum é guardar um *string* Lua em C como simplesmente o ponteiro que Lua retorna via `lua_tostring`: quando a pilha ficar inválida, esse ponteiro pode não mais apontar para o *string* correspondente (nem para nenhum outro *string* ou qualquer área válida). A pilha fica inválida quando a função C retorna ou quando o estado é fechado.

4 Uso de Lua em jogos

Nesta seção, discutiremos o uso de Lua em jogos, desde um nível mais simples até um nível sofisticado.

4.1 Lua como linguagem de configuração

Como discutimos na Seção 2, no nível mais simples uma linguagem de configuração é uma maneira de associar valores a variáveis. Não há controle de fluxo nem funções definidas pelo usuário, somente uma sequência de atribuições. Um exemplo típico é:

```
-- começar no meio do jogo, usando Mickey...
LEVEL = 13
HERO = "Mickey"
```

Mesmo uma linguagem simples como essa já dá uma grande flexibilidade à aplicação, pois permite ao usuário controlar a aplicação externamente, bastando editar um arquivo texto.

Vejamos como usar Lua nessa situação do ponto de vista do programador da aplicação. Estamos portanto agora falando de código C. (Do ponto de vista do usuário da aplicação, para usar a linguagem de configuração basta ler a documentação da aplicação para saber que variáveis existem, quais os seus possíveis valores e o que eles significam para a aplicação. O usuário nem precisa saber que está escrevendo na verdade um programa Lua.)

A primeira coisa é carregar essa configuração de dentro da aplicação. Antes disso, é preciso inicializar Lua, criando um estado, que vai existir até ser fechado:

```
#include "lua.h"
#include "lauxlib.h"
...
lua_State *L=lua_open();
...
lua_close(L);
```

Uma vez criado um estado Lua, podemos carregar um arquivo de configuração, digamos `init.lua`:

```
luaL_loadfile(L,"init.lua");
lua_pcall(L,0,0,0);
```

Note que a carga da configuração é feita em dois passos: leitura com `luaL_loadfile` e execução com `lua_pcall`. Isso permite o tratamento separado de erros de sintaxe e erros de execução. Entretanto, o código acima não trata erros. Na prática, usa-se o código abaixo ou algo parecido:

```
if (luaL_loadfile(L,"init.lua") || lua_pcall(L,0,0,0)) error(lua_tostring(L,-1));
```

onde `error` é uma função que trata erro. A mensagem de erro vinda de Lua está no topo da pilha e portanto é obtida com `lua_tostring(L,-1)`.

Assumindo que não houve erros na carga da configuração, a execução de `init.lua` criou no estado `L` as variáveis com seus valores dados em `init.lua`. É hora portanto da aplicação usar esses valores. Note que os valores estão em Lua, mas ainda não em C; é necessário lê-los de Lua para C. Tipicamente, a aplicação está interessada nos valores de algumas variáveis específicas, como `LEVEL` no exemplo inicial. Podemos ler o valor de `LEVEL` com

```
lua_getglobal(L,"LEVEL");
```

Isso lê o valor da variável `LEVEL` de Lua e deixa esse valor na pilha, que é o mecanismo de comunicação entre Lua e C e vice-versa. Basta agora copiar essa valor para uma variável C:

```
level=lua_tonumber(L,-1);
```

assumindo claro que `level` esteja declarada corretamente em C. Note que não há nenhuma relação entre a variável C e a variável Lua. Nesse exemplo, elas nem têm o mesmo nome, somente um nome parecido. Mas mesmo que tivessem o mesmo nome, seriam variáveis em mundos separados, sem nenhuma relação automática entre elas. (É possível *programar* uma tal relação automática entre os mundos C e Lua usando mecanismos avançados de Lua.)

A mesma coisa se aplica para `HERO`, exceto que agora queremos um *string*:

```
lua_getglobal(L,"HERO");
hero=lua_tostring(L,-1);
```

E isso é tudo. A aplicação não precisa mais de Lua e pode agora fazer o que ela tem que fazer, usando os valores de `level` e `hero` fornecidos pelo usuário. Um código completo seria então algo como:

```
#include "lua.h"
#include "lauxlib.h"

static int level=0;
const char* hero="Minnie";
...
int main(void)
{
 lua_State *L=lua_open();
 luaL_loadfile(L,"init.lua");
 lua_pcall(L,0,0,0);
 lua_getglobal(L,"LEVEL");
 level=lua_tonumber(L,-1);
 lua_getglobal(L,"HERO");
 hero=lua_tostring(L,-1);
 play(level,hero);
 lua_close(L);
 return 0;
}
```

Note que não podemos fechar o estado Lua antes de chamar `play`, pois `play` usa `hero`, que é um *string* obtido de Lua. Para poder fechar o estado Lua antes de chamar `play`, seria necessário duplicar o valor de `hero` antes.

Mais uma vez, o código acima não trata erros. Isso é feito somente para simplificar a exposição. Na prática, o tratamento de erros é obrigatório (como em toda aplicação de qualidade), principalmente quando se carrega arquivos escritos por usuários: não se pode exigir que os usuários não cometam enganos! (A aplicação também precisa se proteger contra usuários mal intencionados...)

O uso de Lua nessa situação simples pode parecer um exagero. É código demais para ler dois valores fornecidos pelo usuário. Seria bem mais simples lê-los da linha de comando ou mesmo de um arquivo, mas sem a necessidade de nomes de variáveis. Na prática, são necessários muito mais do que somente dois valores. De qualquer modo, note como usar uma linguagem tem grandes vantagens: comentários, linhas em branco, indentação, aspas e espaços dentro de aspas são todos tratados automaticamente e funcionam da maneira como o usuário espera inconscientemente que eles funcionem. Fazer isso manualmente na aplicação seria sim uma grande complicação!

Esse nível simples de configuração também permite coisas mais complicadas, como definir variáveis em função de outras:

```
-- começar no meio do jogo, usando Mickey...
LEVEL = 13
HERO = "Mickey"
GREET = "Bom dia " .. HERO .. "! Como vai"
SCORE = 1.2 * LEVEL
```

Embora o arquivo continue sendo uma lista de atribuições de valores a variáveis, é possível usar *expressões* do lado direito das atribuições. Entender e executar expressões é uma das tarefas principais de uma linguagem de programação. Note aqui a vantagem de termos uma linguagem embutida completa!

O usuário pode não saber que é possível fazer isso, mas assim que ele souber ou descobrir, vai provavelmente usar atribuições complicadas sem ter que pensar muito na sua forma, pois a sintaxe das expressões em Lua é a mesma da maioria das linguagens (com a possível exceção do operador de combinação de *strings*).

4.2 Lua como linguagem de extensão

O uso de Lua como linguagem de configuração mostrado na seção anterior ainda é muito simples. Lua oferece facilidades para estruturação de dados que podemos explorar quando descrevemos os objetos de um jogo. Para ilustrar a discussão, vamos considerar que precisamos descrever diferentes armas que podem ser usadas por nossos personagens. Para cada arma, devemos informar seu "fator de agressividade", "alcance de ataque" e "precisão". O conjunto de armas pode ser agrupado numa tabela, onde os elementos especificam as características de cada arma:

```
weapons = {
   knife = {
            aggression = 0.3,
            attackrange = 0.5,
            accuracy = 1.0,
           },
   sword = {
            aggression = 0.5,
            attackrange = 1.5,
            accuracy = 0.8,
```

```
        },
      ...
}
```

Com os dados estruturados, é fácil estender o jogo, incluindo, por exemplo, um novo tipo de arma. Dessa forma, a "precisão" de uma espada é obtida consultando o valor de `weapons.sword.accuracy`. De C, assumindo que `weapons` é uma variável global de Lua, esse valor seria obtido pelo seguinte trecho de código:

```
double accuracy;
lua_getglobal(L,'weapons');      /* push weapons on stack */
lua_pushstring(L,'sword');       /* push string 'sword' */
lua_gettable(L,-2);              /* get weapons.sword */
lua_pushstring(L,'accuracy');    /* push string 'accuracy' */
lua_gettable(L,-2);              /* get weapons.sword.accuracy */
accuracy = lua_tonumber(L,-1);   /* convert value to C */
lua_pop(L,2);                    /* restore Lua stack */
```

Conforme mencionado na seção anterior, é fundamental que tenhamos verificação de erros. A verificação de erros em C seria tediosa. Fica bem mais simples escrever código Lua que faça a verificação de erros nos *scripts* escritos pelos usuários (roteiristas, artistas, programadores, ou os próprios usuários finais dos jogos). Uma maneira simples de fazer a verificação de erro é incluir construtores de tabelas. No exemplo acima, podemos incluir o construtor `Weapon` para cada arma descrita:

```
weapons = {
   knife = Weapon{
              aggression = 0.3,
              attackrange = 0.5,
              accuracy = 1.0,
           },
   sword = Weapon{
              aggression = 0.5,
              attackrange = 1.5,
              accuracy = 0.8,
           },
   ...
}
```

O construtor `Weapon` pode então verificar erros e preencher valores *defaults*:

```
funciton Weapon (self)
   if not self.aggression then
       self.aggression = 0.5       -- default aggression value
   elseif self.aggression < 0.0 or self.aggression > 1.0 then
       ReportError("Invalid aggression value")
   ...
   return self
end
```

Podemos ir mais longe, por exemplo, especificando o comportamento dos personagens. Em Lua, como funções são tratadas como valores de primeira classe, esses comportamentos e ações

podem ser facilmente integrados na descrição de tabelas. Como exemplo, vamos imaginar o momento em que o personagem encontra uma nova arma. As características da arma encontrada podem enriquecer o diálogo:

```
weapons = {
   knife = Weapon{
               aggression = 0.3,
               attackrange = 0.5,
               accuracy = 1.0,
               getit = function (person)
                           if person:HasEnoughWeapon() then
                              person:Speak("Não preciso dessa faca")
                              return false
                           else
                              person:Speak("Essa faca me será muito útil")
                              return true
                           end
                       end,
           },
   ...
}
```

4.3 Lua como linguagem de controle

Nesse terceiro nível de utilização da linguagem Lua em jogos, invertemos os papéis: Lua passa a ser o controlador do jogo, o cliente, e o código C funciona apenas como servidor, implementando de forma eficiente os serviços demandados por Lua. Nesse caso, ganhamos uma grande flexibilidade com o uso de uma linguagem de *script*. Os programadores C ficam responsáveis por implementar algoritmos eficientes e os "programadores" Lua ficam responsáveis por criar o roteiro, a história, o comportamento dos personagens, etc. Dessa forma, em C codificamos as *engines* do jogo (estruturação e *rendering* de cenas, simulação física, algoritmos de inteligência artificial, gerenciamento de sons, etc.) e, em Lua, escrevemos o *script*, decidindo que arma usar, que som tocar, que algortimo de inteligência artificial usar, etc. Essa é uma divisão natural para o desenvolvimento dos jogos. A vantagem de se usar uma linguagem como Lua é que os profissionais envolvidos com a programação do roteiro não são, em geral, profissionais com experiência em programação. No entanto, aprender a programar um ambiente Lua onde os erros são automaticamente verificados é muito simples. Além disso, como não há necessidade de compilação da aplicação — que pode ser demorada — o desenvolvimento do jogo fica muito mais rápido.

Para que de Lua tenhamos acesso aos serviços oferecidos por C temos que exportar as funcionalidades de C para Lua. Isso pode ser feito utilizando a API de Lua diretamente ou através de ferramentas que fazem o mapeamento de forma automática. No *site* de Lua, encontram-se disponíveis algumas dessas ferramentas.

A disponibilização dos serviços implementados em C para "programadores" Lua pode ser feita em duas etapas: mapeamento direto das funções e classes, e código de interface em Lua.

Na primeira etapa, usando a API ou uma ferramenta de mapeamento, obtemos um código C que exporta as funções e métodos para Lua. Em geral, isso é feito escrevendo-se funções C que, usando a API de Lua, recebem os parâmetros de Lua, chamam as funções e métodos de C e mapeiam os valores retornados para Lua.

Na segunda etapa, podemos encapsular as chamadas das funções e métodos de C através de construtores e funções escritos em Lua, elevando o nível de abstração para acesso aos serviços das

engines. Dessa forma, fazemos com que a programação em Lua seja feita de forma simples, facilitando o acesso programável a artistas, roteiristas, etc.

Para exemplificar, vamos considerar a implementação em C++ da classe 'CPerson' que estrutura as características de uma personagem do jogo. A cada personagem associamos uma série de atributos: nome, energia inicial, listas das armas que sabe manusear, etc. Em C++, esses atributos são definidos através de chamadas de métodos. Podemos também prever a implementação de ações simples como andar, correr, pular, atacar. A interface da classe em C++ poderia ser dada então por:

```
class CPerson {
   ...
public:
   CPerson (char* model_file);
   void SetName (char* name);
   void SetEnergy (double value);
   AddSkill (Weapon* w);
   double GetEnergy ();

   Walk ();
   Run ();
   Jump ();
   Attack ();
   ...
};
```

Com o uso de uma ferramenta (ou fazendo o mapeamento diretamente via API), podemos ter acessos a esses métodos em Lua. No entanto, não queremos que o roteirista do jogo tenha que fazer chamadas de métodos de C++. O quanto possível, devemos dar preferências a interfaces descritivas, como já vinhamos fazendo nas seções anteriores. Um roteirista poderia, por exemplo, instanciar um novo personagem de forma descritiva:

```
Hero = Person {
          name = "Tarzan",
          model = "models/tarzan.mdl",
          energy = 1.0,
          skills = {knife, axe}
       }
```

O construtor, previamente codificado em Lua, seria responsável por instanciar o objeto em C++ e definir seus atributos iniciais (além de fazer verificação de erros, que será omitida aqui):

```
function Person (self)
  local cobj = CPerson:new(self.model)       -- create instance
  cobj:SetName(self.name)
  cobj:SetEnergy(self.energy)
  for i,v = ipairs(self.skills) do
     cobj:AddSkill(v)
  end
  return cobj
end
```

Numa segunda etapa, o roteirista pode programar as ações associadas ao personagem:

```
...
if Hero:GetEnergy() > 0.5 then
   Hero:Attack()
else
   Hero:Run()
end
...
```

5 Conclusão

A linguagem Lua tem sido amplamente utilizada no desenvolvimento de jogos. A Lucasarts, por exemplo, usou a versão 3.1 de Lua para desenvolver os títulos "Grim Fandango" e "Scape from Monkey Island". A versão 3.1 de Lua foi por eles modificada para tratar co-rotinas. Hoje, como vimos, suporte para co-rotinas está presenta na versão 5.0.

Double Fine utilizou Lua em "Psychonauts" para controlar toda a lógica do jogo. Basicamente, a *engine* carrega um mundo estático e os *scripts* em Lua tomam o controle, dando vida e interatividade às cenas. Em "Baldur's Gate", Bioware usou Lua para prover uma ferramenta de depuração em tempo-real. Relic utilizou Lua em "Impossible Creatures" para controlar a IA, as aparências dos objetos e personagens, para definir as regras do jogo e também como ferramenta de depuração em tempo-real. Em "FarCry", Crytek também utilizou Lua para controlar diversos aspectos do jogo e para permitir a criação de modificadores através da codificação de *scripts* Lua.

6 Agradecimentos

Os usos de Lua em jogos listados na conclusão foram levantados por Marcio Pereira de Araujo como parte do trabalho da disciplina "Linguagem Lua", ministrada por Roberto Ierusalimschy, oferecida nos programas de graduação e pós-graduação do Departamento de Informática da PUC-Rio.

7 Referências

Para saber mais sobre Lua, leia o livro "Programming in Lua", o manual de referência e os artigos abaixo. Todos esses documentos e muitas outras informações estão disponíveis no *site* de Lua (lua.org).

- R. Ierusalimschy, *Programming in Lua*. Lua.org, December 2003. ISBN 85-903798-1-7.
- R. Ierusalimschy, L. H. de Figueiredo, W. Celes. "Lua 5.0 Reference Manual". Technical Report MCC-14/03, PUC-Rio, 2003.
- R. Ierusalimschy, L. H. de Figueiredo, W. Celes. The evolution of an extension language: a history of Lua, *Proceedings of V Brazilian Symposium on Programming Languages* (2001) B-14–B-28.
- R. Ierusalimschy, L. H. de Figueiredo, W. Celes. Lua—an extensible extension language. *Software: Practice & Experience* **26** #6 (1996) 635–652.
- L. H. de Figueiredo, R. Ierusalimschy, W. Celes. Lua: an extensible embedded language. *Dr. Dobb's Journal* **21** #12 (Dec 1996) 26–33.
- L. H. de Figueiredo, R. Ierusalimschy, W. Celes. The design and implementation of a language for extending applications. *Proceedings of XXI Brazilian Seminar on Software and Hardware* (1994) 273–83.