CENTRO PAULA SOUZA

GOVERNO DO ESTADO SÃO PAULO

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise de Sistemas e Tecnologia da Informação - Jogos Digitais

A LINGUAGEM LUA NO DESENVOLVIMENTO DE JOGOS DIGITAIS

WILLIAM ROBERTO DOS SANTOS

Americana, SP
2012

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Análise de Sistemas e Tecnologia da Informação - Jogos Digitais

A LINGUAGEM LUA NO DESENVOLVIMENTO DE JOGOS DIGITAIS

WILLIAM ROBERTO DOS SANTOS

wr.sant@gmail.com

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Desenvolvimento de Jogos Digitais, sob a orientação do Prof. Antonio Alfredo Lacerda.

Área: Jogos Digitais

Americana, SP
2012

BANCA EXAMINADORA

Prof. Antonio Alfredo Lacerda

Prof. Me. Cleberson Eugenio Forte

Prof. Fernando José Ignácio

AGRADECIMENTOS

Primeiramente agradeço a Deus, pela força que me dá todos os dias, por abençoar a minha vida, me fazendo sempre trilhar o caminho do bem.

Agradeço aos meus pais, Valdir e Regina, pelo incentivo que sempre me deram, não me deixando desistir jamais. Agradeço pela educação e pelo apoio nas horas mais difíceis. À minha avó Alice, pelo seu carinho e pelas palavras de sabedoria que aprendo com a senhora a cada dia.

Agradeço a minha companheira Nany, pela sua paciência, carinho, amizade e por sempre acreditar em mim.

Aos meus professores da Fatec, que tanto me ensinaram, agradeço pela oportunidade de ter aprendido tanta coisa com essas pessoas. Ao meu orientador, Toni, que sempre acreditou no meu projeto. Agradeço também ao professor Cleberson pela paciência que teve comigo no desenvolvimento deste trabalho.

Por fim, agradeço aos meus amigos, que sempre estiveram ao meu lado me apoiando. Ao Alex, pelas tantas vezes que me acompanhou na nossa cervejinha aos sábados, aos meus colegas de trabalho Aline e Marco, ao meu parceiro Fernando Cruz, aos meus amigos da faculdade, agradeço por serem sempre prestativos.

DEDICATÓRIA

Dedico este trabalho à minha segunda mãe, Sueli, por toda a força e incentivo que me deu enquanto esteve neste mundo.

RESUMO

A linguagem Lua é uma linguagem de script desenvolvida no Brasil por uma equipe da PUC-Rio. Projetada para estender aplicações, Lua possui vantagens como simplicidade, rapidez, portabilidade e pequeno tamanho, além de permitir uma fácil integração com outras linguagens. Possui sintaxe simples, com poucos tipos de dados, porém, capaz de oferecer mecanismos para estender sua semântica. Alcançou destaque internacional sendo usada principalmente no desenvolvimento de jogos, tornando-se a linguagem de script mais utilizada nesta área. Lua também está presente em outras aplicações, como Adobe Lightroom, Wireshark e Ginga, *middleware* da TV digital brasileira. Este trabalho apresenta a linguagem Lua com ênfase no desenvolvimento de jogos.

Palavras Chave: Linguagem de Programação Lua, Desenvolvimento de Jogos, Linguagens de Script

ABSTRACT

Lua Programming Language is a scripting language developed in Brazil by a team at PUC-Rio. Designed for extend applications, Lua has advantages such as simplicity, fast execution, portability, small size and allows easy integration with other languages. It has simple syntax, with limited data types, however, it is able to offer mechanisms to extend its semantics. Achieved international highlight, especially in game development, became the most used scripting language in this area. Lua is also present in other applications, like Adobe Lightroom, Wireshark and Ginga, the Brazilian digital TV middleware. This paper presents Lua Programming Language with an emphasis on game development.

Keywords: Lua Programming Language, Game Development, Scripting Languages

SUMÁRIO

LISTA DE FIGURAS E DE TABELAS

1 INTRODUÇÃO

Com o constante crescimento no mercado de jogos digitais, desenvolvedores da área necessitam cada vez mais de ferramentas que tornem o processo mais rápido e ao mesmo tempo garantam a qualidade e bom desempenho de seu jogo. A utilização de linguagens de script vem ao longo do tempo se mostrando uma tendência entre estes desenvolvedores. Entre as linguagens mais usadas, Lua se destaca por sua simplicidade, portabilidade, economia de recursos e desempenho (IERUSALIMSCHY, FIGUEIREDO e CELES, 2007). Desenvolvida no Brasil por uma equipe da Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), é a única linguagem criada fora do eixo EUA/Europa/Japão. Passou a obter destaque internacional principalmente após ser usada no desenvolvimento de jogos de grandes empresas como a LucasArts. Por ser uma linguagem dinâmica, Lua é frequentemente utilizada para configurações e prototipagens.

Uma linguagem dinâmica se diferencia de uma linguagem estática principalmente por ser interpretada ao invés de compilada. Esta característica permite ao usuário estender funcionalidades da aplicação sem necessidade de recompilar o programa. Por essa razão, as linguagens de script estão cada vez mais sendo utilizadas, sobretudo no desenvolvimento de jogos.

O objetivo deste trabalho é abordar as características da linguagem Lua, bem como suas vantagens e as razões que contribuíram para o seu crescimento.

Este trabalho está organizado da seguinte forma: no capítulo 2 encontra-se uma breve introdução à história e evolução da linguagem, acompanhada de suas principais características. O capítulo 3 apresenta a sintaxe da linguagem, seus tipos de dados e exemplos de uso. O capítulo 4 tem como objetivo demonstrar a relação de Lua com os jogos e as principais ferramentas disponíveis no mercado. No capítulo 5 encontra-se a proposta de desenvolvimento de um jogo básico utilizando Lua. Por fim, são apresentadas as considerações finais deste trabalho.

2 HISTÓRIA

A linguagem Lua surgiu em 1993 no grupo de Tecnologia em Computação Gráfica da PUC-Rio (TeCGraf). Desenvolvida por três membros deste grupo, Roberto Ierusalimschy, Luiz Henrique de Figueiredo e Waldemar Celes, a linguagem Lua surgiu motivada pela necessidade de oferecer aplicações configuráveis pelos usuários. Neste período, a Petrobras necessitava capturar dados na perfuração de poços de petróleo e, a partir deles, os programas de simulação exibiam gráficos baseados em dados diversos, como temperatura e profundidade do poço na medida em que eram perfurados. Esse processo era cansativo e propenso a erros, uma vez que os programas precisavam de dados rigorosamente formatados. Entretanto, este procedimento poderia ser ainda mais complexo quando o engenheiro necessitava visualizar outras informações, mudar a cor, o tipo de gráfico ou a escala. Diante de situações como esta, era preciso solicitar alterações aos desenvolvedores do programa, que após modificar o software, o enviavam novamente à plataforma. Então, o TeCGraf foi solicitado junto a Petrobras para desenvolver uma solução, que era basicamente composta por diversos *front-ends* gráficos para a entrada dos dados capturados. Para simplificar o desenvolvimento, a equipe optou por uma codificação de maneira uniforme, desenvolvendo uma linguagem declarativa simples, chamada DEL (Data Entry Language).

Com a linguagem DEL era possível utilizar uma estrutura de dados chamada entity, que possuía campos com valores padrão e validação de dados. A linguagem foi sucesso tanto entre os desenvolvedores quanto entre os usuários, pois era facilmente adaptável em programas que possuíam entrada de dados. Porém, em pouco tempo os usuários começaram a exigir mais recursos na linguagem, como expressões booleanas, estruturas condicionais e *loops*. Por essa razão, ficou claro que era preciso desenvolver uma linguagem de programação completa.

Ao mesmo tempo, outro projeto estava sendo desenvolvido para a Petrobras, chamado PGM, um gerador de relatórios configurável. No projeto, toda a configuração dos relatórios poderia ser feita pelo próprio usuário e o programa seria capaz de funcionar em computadores mais simples, como um PC com MS-DOS. Surgia então a linguagem Sol (Simple Object Language), utilizada para descrição de

objetos e permitia a declaração de tipos. Foi implementada como uma biblioteca para linguagem C, sendo chamada pelo programa principal, que através de sua API, era capaz de acessar as informações de configuração.

A partir disso, Roberto Ierusalimschy (PGM), Luiz Henrique de Figueiredo (DEL) e Waldemar Celes (PGM) perceberam que DEL e Sol possuíam problemas em comum e decidiram se reunir para solucioná-los combinando DEL e Sol em uma única linguagem de configuração mais completa e poderosa, e ao mesmo tempo facilmente acoplável, portátil e de sintaxe simples. Como a nova linguagem seria uma versão modificada de Sol, o nome sugerido foi Lua.

Lua herdava características de Sol e apresentava a estrutura de tabelas associativas, que ao contrário de registros e listas, sua indexação não se restringe apenas a números inteiros ou strings. Os tipos de dados disponíveis no inicio eram apenas number, string, tables, nil, userdata e functions. Lua também herdou a característica de ser implementada como uma biblioteca, seguindo o princípio “a coisa mais simples que poderia funcionar”, que posteriormente seria incorporado à metodologia *Extreme Programming* (IERUSALIMSCHY, FIGUEIREDO e CELES, 2007 apud BECK, 2000).

Atualmente, Lua está na versão 5.2, onde segundo Ierusalimschy, Figueiredo e Celes (2007) atingiu sua “maturidade”. Sua evolução se baseou sempre nos mesmos conceitos dos requisitos iniciais, como simplicidade, minimalismo e robustez. O quadro 1 demonstra a evolução da linguagem.

Versão	Principais Alterações
1.1	A linguagem possuía usuários fora do TeCGraf, manual de referência e API bem documentada.
2.1 - 2.5	Suporte para orientação a objetos e interface para depuração. Iniciou a exposição internacional através de artigos, sites e grupos de discussão.
3.0 - 3.2	Melhorias no suporte a funções
4.0	Nova API com C baseada em pilha. Além disso, todas as bibliotecas passaram a ser implementadas via API oficial.
5.0 - 5.1	Suporte a co-rotinas, sistema de módulos, máquina virtual de registradores e coleta de lixo incremental.

Quadro 1: A evolução de Lua

2.1 CARACTERÍSTICAS DE LUA

Lua foi projetada para ser uma linguagem de extensão capaz de oferecer suporte a outras aplicações. Por essa razão, não existe o conceito de programa principal em Lua: a linguagem funciona embarcada em um programa denominado hospedeiro (ou programa cliente anfitrião), capaz de executar trechos de código escrito em Lua. Segundo Ierusalimschy (2006), este tipo de integração com outra linguagem apresenta uma série de benefícios: Lua oferece estruturas dinâmicas, facilidades de testes e depuração, segurança, gerenciamento automático de memória e facilidade na manipulação de *strings*.

Lua possui características que faz com que se destaque perante outras linguagens de script. Simplicidade, portabilidade e rapidez são as principais vantagens de Lua. Seu núcleo, somado às suas bibliotecas resultam em cerca de 200k de espaço (PROJETO KEPLER, 2008).

A rapidez de Lua é um dos aspectos positivos da linguagem segundo seu site oficial, destacando que ela se mostra mais rápida diante de outras linguagens de script em vários benchmarks realizados. A portabilidade também é uma grande vantagem, pois seu pequeno pacote é compilado em todos os tipos de Unix, Windows e dispositivos móveis, como Android, iOS, Symbian e Windows Phone. A simplicidade da linguagem também se mostra vantajosa: um exemplo dessa característica é que Lua fornece mecanismos para construções ao invés de políticas definidas. Isso significa que, embora seja uma linguagem orientada a objetos, ela fornece apenas meta-mecanismos para a implementação de classes e herança, gerando assim uma economia de conceitos e deixando a linguagem pequena.

Lua permite integrações com outras linguagens, sendo facilmente embutida em qualquer aplicação escrita em C, C++, Java, C#, Fortran, e até mesmo outras linguagens de script como Perl e Ruby. Visto que Lua necessita de um programa hospedeiro para ser executada, Jung e Brown (2007) afirmam que o interpretador *standalone* de Lua pode ser considerado um hospedeiro, pois de certa forma, Lua se integra a ele no momento da execução. Lua pode ser integrada com outras aplicações de duas formas: incorporada ao programa principal ou estendida através de scripts.

Lua é distribuída sob licença MIT, sendo um software livre e de código aberto. Embora seu uso seja predominantemente voltado ao ambiente de jogos, Lua pode ser utilizada como uma linguagem de propósito geral, desde a criação de pequenos scripts para estender aplicações, como em sistemas maiores, com milhares de linhas de código. Segundo Dalmazo e Avelar (2007), diversos casos de uso podem ter Lua como parte da implementação de uma solução, seja para o domínio empresarial, científico e industrial.

Segundo Ierusalimschy (2012), Lua está presente em jogos famosos como Grim Fandango, World of Warcraft, Far Cry, Baldur's Gate, Sim City 4, The Sims 2 e Angry Birds. Lua está presente em aplicações como o Ginga, *middleware* padrão da TV digital brasileira, e Wireshark, analisador de protocolos de rede. Outro exemplo é o software Adobe Lightroom 3, que possui cerca de um milhão de linhas de código escrito em Lua. Por ser uma linguagem altamente portável, Lua está presente embarcada em dispositivos como TVs (Samsung), roteadores (Cisco), teclados (Logitech), impressoras (Olivetti), etc.

3 A LINGUAGEM

A linguagem Lua apresenta diversas características que fazem com que ela se diferencie das outras linguagens de script: programação funcional, utilização de tabelas como estruturas de dados e a comunicação com código escrito em linguagem C. Lua é uma linguagem dinâmica, e assim como outras linguagens deste tipo, apresenta as seguintes características descritas por Ierusalimschy (2009):

- Interpretação dinâmica;
- Tipagem dinâmica forte;
- Gerência automática de memória dinâmica (coleta de lixo).

Lua possui um interpretador independente (*stand-alone*) que pode ser instalado em ambiente Windows, Linux ou Mac. Após sua instalação, o interpretador estará pronto para executar qualquer código escrito em Lua podendo ser um programa escrito em um arquivo ou simplesmente informando diretamente o comando no prompt, uma vez que em Lua não existe a necessidade de uma função "main".

3.1 TIPOS DE DADOS E VARIÁVEIS

Uma das principais características de Lua, é que suas variáveis não possuem o tipo explicitamente declarado. Os tipos são dinamicamente associados aos dados da variável assim que determinado valor for atribuído à variável. Assim,

```
x = 2.33
```

armazena na variável "x" um valor numérico do tipo *number*.

Existem oito tipos de dados em Lua: *nil*, *boolean*, *number*, *string*, *userdata*, *function*, *thread* e *table*. Para saber o tipo de uma variável, utiliza-se a função *type*. Dessa forma, informando o comando

```
print(type(x))
```

será exibido na tela o resultado da função, neste caso "*number*".

As variáveis podem ter seus tipos alterados após a atribuição de um valor de tipo diferente, independentemente do valor armazenado anteriormente.

```
x = "Hello World"
```

A partir do comando acima, a variável "x" passa a ser do tipo *string*. De acordo com Ierusalimschy (2006), quando se utiliza uma única variável para dois diferentes tipos, pode aparentar um código confuso, porém, em casos específicos, tal facilidade pode ser vantajosa, como por exemplo, o uso de *nil* como retorno de função para diferenciar um valor normal de uma condição anormal.

3.1.1 Nil

Nil representa a ausência de valor em uma variável. Toda variável declarada assume o valor *nil* por padrão até que seja inicializada. De acordo com Avelar e Dalmazo (2007), é possível utilizar a palavra reservada *nil* para atribuir um valor vazio à variável, deixando-a como se não estivesse inicializada ou ainda, comparar uma variável com a palavra *nil* para testar se ela possui algum valor.

```
x = nil
if x == nil then
  print("variável não inicializada")
```

3.1.2 Boolean

O tipo *boolean* representa os tradicionais valores lógicos: *true* e *false*. Além disso, em Lua qualquer valor pode representar uma condição. Ou seja, testes condicionais, tais como os que são usados em estruturas de controle, consideram os valores *nil* e *false* sempre como falsos e qualquer outro valor como verdadeiro, inclusive zero e *strings* vazias.

3.1.3 Number

Lua usa o tipo *number* para representar valores numéricos reais (ponto flutuante) e inteiros. Diferente de outras linguagens, não há um tipo específico para inteiros, pois segundo Ierusalimschy (2006), é desnecessário e, além disso, processadores mais modernos podem realizar cálculos com pontos flutuantes mais rápidos do que com números inteiros.

Sendo assim, os seguintes valores são válidos para o tipo *number*:

```
x = 4
x = 0.4
x = 4.57e-3
x = 0.3e12
x = 5e+20
```

3.1.4 String

O tipo *string* é utilizado para armazenar sequências de caracteres. São delimitados por aspas simples ou duplas, caso o valor seja apenas uma linha, e colchetes duplos, para sequências com mais de uma linha. Lua aceita os mesmos caracteres de escape da linguagem C, como por exemplo, \n para quebra de linha, \" para aspas duplas e \\ para barra invertida.

Lua fornece conversões automáticas entre os tipos *number* e *string*. Sendo assim, ao informar o comando

```
print("10" + 1)
```

será exibido na tela o número 11, resultado da conversão da *string* "10" para o tipo *number* e a soma do valor a 1. Além desse recurso, Lua oferece as funções tonumber( ) e tostring( ) para realizar as conversões de tipos.

3.1.5 Table

Lua apresenta um único tipo de estrutura de dados chamado *Table*. Este tipo implementa os *arrays* associativos, permitindo ser indexado não apenas por números, mas por qualquer outro tipo de dado (exceto *nil*). Além disso, as tabelas não possuem tamanho fixo, sendo possível adicionar os elementos dinamicamente quando for preciso. Ierusalimschy (2006) afirma que é possível imaginar o tipo *table* como um objeto de alocação dinâmica, sendo que o programa manipula apenas referências (ou ponteiros).

Para utilizar uma varável do tipo *table*, é preciso usar a expressão construtora composta por chaves (`{}`). A partir da construção, é possível adicionar novos "campos" à tabela utilizando colchetes (`[]`) para informar o índice do valor.

```
T = {}
T["x"] = 10
```

O uso de *tables* não se restringe apenas a criação de vetores. Pode-se utilizar uma variável do tipo *table* para representar tabelas de símbolos, listas, registros, filas, árvores, entre outras estruturas de dados.

3.1.6 Function

Segundo Ierusalimschy (2006), o tipo *function* (função) é um valor de primeira classe. Isto significa que um valor do tipo *function* pode ser armazenado em outras variáveis, passados como parâmetros para outras funções, e, por fim, serem retornadas como resultados de outras funções.

Lua pode chamar uma função escrita tanto em Lua como em C. As funções da biblioteca padrão de Lua são todas escritas em C, e são usadas para manipulação de strings, manipulação de tabelas, operações de E/S, funções matemáticas, etc.

Uma função escrita em Lua pode ser declarada da seguinte forma:

```
function nomeDaFuncao(arg1, arg2, ..., arg_n)

end
```

Onde as variáveis "arg" representam os parâmetros da função. Uma função pode ter vários ou nenhum parâmetro, da mesma forma que pode retornar nenhum ou vários valores, separados pelo símbolo de vírgula. (PROJETO KEPLER, 2008)

```
function somaProduto(a, b)
  local x = a or 1 -- recebe o valor 1 quando a não é informado
  local y = b or 1
  return x + y, x * y
end

s, p = somaProduto(3, 4) -- chama a função declarada acima
print(s, p) -- imprime na tela o resultado da soma e do produto
```

O comando *return* pode receber múltiplos argumentos ou simplesmente ser usado para encerrar a execução da função.

3.1.7 Userdata

O tipo *userdata* permite armazenar dados escritos em C em uma variável de Lua. Este tipo se trata de um bloco de memória e possui apenas as operações de atribuição e teste de identidade. Variáveis com valores do tipo *userdata* não podem ser criadas ou alteradas em Lua. Estas operações são permitidas apenas através da API C, com o objetivo de garantir a integridade dos dados contidos no programa hospedeiro.

3.1.8 Threads

O tipo *thread* representa fluxos de execução independentes, utilizado para implementar co-rotinas. Lua oferece suporte a co-rotinas em todos os sistemas operacionais, inclusive naqueles que não oferecem suporte a *threads*, conforme afirmam Ierusalimschy, Figueiredo e Celes (2006).

3.2 ESTRUTURAS DE CONTROLE

Lua oferece um pequeno conjunto com as típicas estruturas de controle, composto pela estrutura *if*, para condições e *while*, *repeat* e *for* para iterações. É importante ressaltar que para estruturas condicionais, Lua considera o resultado como verdadeiro para qualquer resultado diferente de *false* e *nil*.

3.2.1 If then else

O comando *if* testa a condição informada e executa o bloco a partir de *then* ou *else*. O bloco *else* é opcional. Para utilizar *if*s aninhados, utiliza-se o comando elseif, evitando assim uma quantidade demasiada de *end*s.

```
if x < 0 then
  print(“x é positivo”)
else
  print(“x é negativo”)
```

3.2.2 While

A instrução *while* é um laço de repetição com tomada de decisão no início. Isto significa que primeiramente, uma condição é testada para que em seguida, ocorra a execução do trecho de código contido no corpo do *loop*, caso a condição seja verdadeira. Ao contrário, o *loop* é encerrado.

```
local i = 1
while i < 10 do
 print(i)
 i = i + 1
end
```

3.2.3 Repeat

Ao contrário de *while*, a instrução *repeat until* é uma estrutura com tomada de decisão no final, ou seja, o trecho contido no corpo da instrução é executado e em seguida, a condição é testada, e caso seja verdadeira, ocorre a execução do trecho novamente. Isso significa que o bloco será executado ao menos uma vez, independentemente da condição ser verdadeira.

```
repeat
  print ("Digite um número:")
  line = io.read()
until tonumber(line) ~= nil
```

3.2.4 For

De acordo com Ierusalimschy (2006), em Lua, a instrução *for* apresenta duas formas variantes: *for* numérico e *for* genérico.

O *for* numérico apresenta a seguinte estrutura:

```
for var=exp1,exp2,exp3 do

end
```

onde o *loop* executará seu conteúdo para cada valor de “var” desde exp1 até exp2 usando exp3 para incrementar var, sendo esta última opcional. Quando não informada, Lua pressupõe que o valor de incremento seja 1.

O *for* genérico percorre todos os valores retornados por funções de iteração. Os principais iteradores oferecidos por Lua são *pairs*, para percorrer as chaves de uma tabela, *ipairs*, para percorrer *arrays*, e *io.lines*, para percorrer arquivos de texto. (PROJETO KEPLER, 2008)

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

O exemplo acima utiliza o *for* genérico para percorrer e imprimir na tela todos os elementos do *array* “a”, onde a variável “i” armazena o índice e “v” armazena seu valor.

3.2.5 Break e Return

As instruções *break* e *return* são utilizadas para sair de um bloco. *Break* é utilizado para encerrar uma estrutura de repetição, fazendo com que a execução do programa continue no ponto logo após o *loop*, enquanto *return* é utilizado em funções, retornando possíveis resultados ou, simplesmente encerrando a função. Embora seja mais comum escrever estes comandos como a última instrução do bloco, às vezes pode ser útil a inserção de tais comandos no meio do bloco. Ierusalimschy (2006) cita como exemplo uma situação de depuração que se queira evitar a execução de determinado trecho em uma função. Neste caso, adiciona-se a instrução *do return end*.

```
function test()
  do return end
  <trecho de código não executado>
end
```

4 LUA NOS JOGOS

4.1 O INÍCIO

Em junho de 1996, após a publicação de um artigo acadêmico no periódico Dr. Dobb's com o título "Lua in Software: Pratice & Experience", a linguagem começou a adquirir reconhecimento internacional. Os desenvolvedores recebiam diversas mensagens a respeito de Lua. Uma delas se destaca por ser de Bret Mogilefsky, programador-chefe do jogo Grim Fandango, lançado pela LucasArts em 1997.

> "Olá...
>
> Depois de ler o artigo de Dr. Dobbs sobre Lua, eu estava muito ansioso para conferir, e até agora superou minhas expectativas em todos os sentidos! Sua elegância e simplicidade me surpreenderam. Parabéns pelo desenvolvimento tão bem pensado da linguagem.
>
> Algumas informações: Eu estou trabalhando em um jogo de aventura para a LucasArts Entertainment Co., e eu quero tentar substituir nossa velha linguagem de script, SCUMM, por Lua." (Ierusalimschy et al., 2001, p. 8, tradução nossa).

Segundo Ierusalimschy, Figueiredo e Celes (2001), o uso de Lua em Grim Fandango despertou o interesse pela linguagem em outros desenvolvedores de jogos pelo mundo todo, adquirindo popularidade nos grupos de discussão sobre o assunto. A partir desse momento, Lua passou a ser utilizada por outras empresas de desenvolvimento de jogos, entre as mais importantes, destaca-se a canadense BioWare, desenvolvedora do jogo Baldur's Gate. Posteriormente, a linguagem seria homenageada em outro jogo da LucasArts que também usava Lua: Escape from Monkey Island. Neste jogo, os desenvolvedores renomearam o bar anteriormente chamado de SCUMM Bar para Lua Bar, referenciando a substituição da antiga linguagem por Lua.

Figura 1: Lua no jogo *Escape from Monkey Island* (IERUSALIMSCHY, 2008)

Dalmazo e Avelar (2007) destacam que outras empresas como Microsoft, Relic Entertainment, Absolute Studios e Monkeystone Games também utilizam Lua no desenvolvimento de jogos. Ierusalimschy, Celes e Figueiredo (2001), afirmam que isso contribui para que programadores que detenham o conhecimento da linguagem obtenham destaque no mercado através deste diferencial. Embora, estime-se que grande parte dos programadores em Lua estejam envolvidos no desenvolvimento de jogos, não há como ser específico em estatísticas como esta, pois existem muitos segredos nas empresas de jogos, como por exemplo a LucasArts, que utiliza uma versão adaptada de Lua cujos detalhes são de propriedade da própria empresa.

4.2 RAZÕES PARA UTILIZAR LUA NOS JOGOS

Utilizar uma linguagem de extensão no desenvolvimento de jogos, assim como no desenvolvimento de qualquer software, permite o controle externo de partes da aplicação sem que haja necessidade de recompilar todo o restante. Esta técnica apresenta vantagens para programadores e usuários, que podem editar arquivos localizados fora do software conforme a sua necessidade. Nos jogos, esta prática contribui para o aumento da produtividade no desenvolvimento, fazendo com

que outros profissionais que não sejam programadores (designers, animadores, etc.) desenvolvam trechos do jogo sem interferir em outras partes previamente concluídas.

Lua foi projetada para estender outras aplicações, fornecendo ao usuário final a possibilidade de escrever pequenos programas ou trechos de código para executá-los no programa principal. Além disso, há uma série de razões que contribuíram para o crescimento da popularidade da linguagem no ambiente de desenvolvimento de jogos. Segundo Ierusalimschy, Figueiredo e Celes (2001), seu pequeno tamanho, boa performance, portabilidade e facilidade de integração são alguns fatores que contribuíram para este crescimento. Particularmente nos jogos, é possível encontrar diversos benefícios para usar uma linguagem de script como Lua: pode-se usar na definição de *sprites* e física dos objetos, gerenciamento de Inteligência Artficial, manipulação de dispositivos de entrada, etc.

Segundo uma pesquisa realizada pela *gamedev.net* em 2003, 72% dos jogos são desenvolvidos utilizando uma linguagem de script. Os dados coletados revelaram que Lua era utilizada por 20% dos jogos, enquanto a segunda linguagem mais comum neste meio era *Python*, com apenas 7% do total. De acordo com Ierusalimschy, Figueiredo e Celes (2004), um levantamento feito por Marcio Pereira de Araujo, aluno da PUC-Rio, comprovou que a linguagem pode ser aplicada de diferentes maneiras no desenvolvimento de jogos, conforme descrito no Quadro 2.

Jogo	Desenvolvedor	Aplicação da linguagem Lua
Grim Fandango	LucasArts	Utiliza uma versão modificada de Lua 3.1 como linguagem de script.
Escape from Monkey Island	LucasArts	Utiliza uma versão modificada de Lua 3.1 como linguagem de script.
Psychonauts	Double Fine	Toda a lógica do jogo é implementada em Lua; Jogo controlado por entidades com scripts Lua.
Baldur's Gate	Bioware	Utiliza scripts Lua em todo o jogo; Prompt Lua adicionado para debug em tempo real.
Impossible Creatures	Relic	Controle de Inteligência Artificial; Aparência de efeitos e elementos gráficos; Determinação das regras do jogo; Edição dos atributos dos personagens; Debug em tempo real.
FarCry	CryTek	Controle de Inteligência Artificial; Interfaces; Edição de cenas e atributos em tempo real.

Quadro 2: Aplicações da linguagem Lua em jogos

Conforme outra pesquisa realizada pelo site *gamasutra.com*, foi constatado que mais da metade dos desenvolvedores de jogos preferem utilizar Lua como linguagem de script. Os dados coletados na pesquisa tiveram a distribuição ilustrada na figura abaixo:

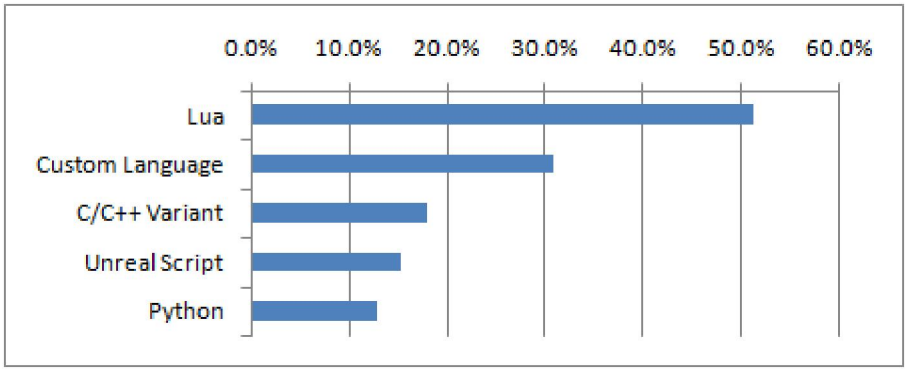

Figura 2: Qual linguagem de script é a mais usada nos jogos (*Gamasutra*)

4.3 COMO UTILIZAR LUA NOS JOGOS

Ierusalimschy, Celes e Figueiredo (2004) propõem que a linguagem seja utilizada no desenvolvimento de jogos de três formas diferentes: linguagem de configuração, extensão e controle.

O uso de Lua como linguagem de configuração, se resume basicamente a uma forma de relacionar valores a variáveis, resultando em uma sequência de atribuições, sem controle de fluxo ou funções, podendo ser comparado aos arquivos de configuração do Windows (.ini). Embora esta seja a maneira mais simples de utilização da linguagem, oferece grande flexibilidade ao jogo, permitindo que o usuário realize alterações na aplicação editando o arquivo desejado. Restaria ao programador do jogo apenas carregar e tratar o arquivo.

A vantagem de se utilizar uma linguagem de script para realizar a atribuição de variáveis é que ao contrário de um simples arquivo de texto, uma linguagem oferece a possibilidade de comentários, linhas em branco, identação, aspas e expressões, como por exemplo, concatenação de strings, cálculos, etc. Um exemplo pode ser o código abaixo:

```
NAME = "JOHN"
AGE = 25
LEVEL = 5
SCORE = 0.5 * LEVEL
MESSAGE = "Olá " .. NAME .. ", sua pontuação é " .. SCORE
```

A segunda forma de utilização da linguagem é como uma linguagem de extensão, usufruindo da facilidade que Lua oferece para estruturação de dados em tabelas. Um típico exemplo é quando se deseja descrever um conjunto de elementos compostos por atributos separados por categorias. Ierusalimschy, Figueiredo e Celes (2004) citam um exemplo para definição de armas que o personagem pode utilizar no decorrer do jogo:

```
Armas {
      Faca {
            agressividade = 0.3
            alcance = 0.5
            precisao = 1.0
      },
      Espada {
            agressividade = 0.5
            alcance = 1.5
            precisao = 0.8
      }
}
```

Desta forma, o uso de Lua oferece a possibilidade de extensão do jogo, fornecendo ao usuário a possibilidade de adicionar novas armas ou alterar os valores de seus atributos por exemplo. Porém, assim como no exemplo anterior, o desenvolvedor do jogo deverá se preocupar com o tratamento de possíveis erros causados pelo usuário, podendo inclusive trata-los no próprio script.

Por fim, Lua ainda pode ser utilizada como linguagem de controle. Isto significa que Lua passa a ser o controlador do jogo, enquanto a aplicação funciona como uma espécie de servidor. Sendo assim, os programadores são responsáveis por codificar algoritmos de estruturação do jogo, enquanto o script escrito em Lua determina o comportamento geral do jogo. Isso torna o desenvolvimento mais rápido, já que é uma divisão natural da equipe, ou seja, na maioria das vezes o responsável pela programação não é o roteirista, e é comum que estes não possuam conhecimento em programação. Dessa forma, enquanto o programador prepara as *engines* do jogo, o roteirista pode escrever scripts necessários para determinar qual arma usar ou qual som tocar, por exemplo.

4.4 FRAMEWORKS

Atualmente, existem diversas ferramentas no mercado que auxiliam no desenvolvimento de jogos. Os motores de jogos, originalmente chamados de *game engines*, são responsáveis por abstrair os detalhes comuns no desenvolvimento de um jogo, como renderização, física, animações e sons para que os desenvolvedores possam se dedicar a outras tarefas voltadas diretamente ao próprio jogo. Dessa forma, os motores de jogos ficam encarregados de gerenciar a animação de modelos, detectar colisões entre objetos e gerenciar partes da inteligência artificial, enquanto os desenvolvedores são responsáveis pela criação de elementos que compõem a essência do jogo, como a maneira em que os objetos interagem com o cenário, por exemplo (Ward, 2008).

4.4.1 Corona SDK

O Corona SDK é uma ferramenta que permite o desenvolvimento de jogos em plataformas iOS, Android, Kindle Fire e NOOK utilizando o mesmo código fonte. O software é mantido pela empresa Ansca Mobile (também conhecida como Corona Labs Inc.) e distribuído através de pagamento de licença anual. As licenças variam de acordo com as plataformas escolhidas, ou seja, caso o desenvolvedor queira trabalhar somente em plataforma iOS, deverá adquirir a versão *Indie*. Caso contrário, há a versão *Professional*, que abrange todas as plataformas. É possível ainda utilizar a versão *trial*, porém, caso o desenvolvedor queira publicar o jogo desenvolvido, é necessário adquirir a licença.

Segundo os desenvolvedores, é possível reduzir em dez vezes o tempo de desenvolvimento de uma aplicação usando o Corona SDK. Isso se deve graças ao uso da linguagem Lua, cuja sintaxe simplificada oferece menor complicação ao desenvolvedor quando comparada com Objective-C ou Java, usados em aplicações para iOS e Android respectivamente. Na documentação do Corona SDK é citado o exemplo do procedimento para carregar uma imagem dentro da aplicação. Utilizando uma das outras duas linguagens citadas, o resultado obtido é um trecho com várias linhas de código, contendo rotinas para tratar a imagem e suas propriedades, como redimensionamento e posição. Em contrapartida, no ambiente Corona, o mesmo procedimento pode ser obtido utilizando o seguinte código:

```
display.newImage("Imagem.png", 20, 50);
```

A linha acima mostra como carregar uma imagem e posicioná-la nas coordenadas informadas.

Um projeto desenvolvido no Corona SDK necessita de apenas um arquivo chamado main.lua. Porém, existem arquivos opcionais que podem ser adicionados ao projeto como o config.lua, responsável por outras configurações, como o tamanho da tela, por exemplo. A execução da aplicação pode ser acompanhada utilizando o simulador que acompanha o Corona SDK.

Figura 3: Tela principal do Corona SDK (FERNANDES, 2012)

4.4.2 Moai SDK

Moai é uma plataforma de desenvolvimento de jogos criada pela empresa Zipline Games, que combina um framework cliente com serviços em nuvem. O Moai SDK suporta desenvolvimento em linguagem Lua e C++, podendo ser utilizado para escrever o jogo todo em linguagem Lua ou para estender classes escritas em C++. Os jogos desenvolvidos em Moai podem ser executados nas plataformas iOS, Android, navegadores Chrome, Windows, Macintosh e Linux. É distribuído sob licença Open Source, porém oferece serviços pagos como armazenamento em nuvem, banco de dados e conteúdos adicionais para jogos. É destinado a

desenvolvedores mais experientes que desejam incorporar Lua em jogos mobile ou tenham necessidade de serviços em nuvem.

O Moai SDK inclui bibliotecas e serviços para renderização, animações, sons, física, textos, entre outros efeitos. Está amplamente voltado ao desenvolvimento de jogos cujo estilo dominam o mercado atual, como *arcade*, cassino, estratégia e *puzzle*. Além de proporcionar rapidez no desenvolvimento e portabilidade ao jogo criado, o Moai SDK apresenta a vantagem de possuir código aberto, possibilitando ao desenvolvedor realizar modificações no SDK de acordo com a sua necessidade.

Combinado ao framework, o serviço em nuvem oferecido pela empresa é utilizado para hospedar lógicas de jogo através de *web services*. O Moai Cloud fornece opções de controle na web através de um painel (*dashboard*), onde é possível gerenciar os serviços contratados e visualizar estatísticas. Os serviços on-line oferecidos incluem análises, testes de marketing, notificações de atualização, compartilhamento em redes sociais, disponibilização de conteúdos para download, contas de usuário e serviços *multiplayer*.

Portanto, a ferramenta completa é dividida em três partes:

- Moai SDK – O *framework* para desenvolvimento rápido capaz de trabalhar com Lua e C++;

- Moai Cloud – Utilizada para disponibilizar serviços on-line, como lógica e dados para a aplicação;

- Moai Direct Service – Disponibiliza serviços predefinidos para a aplicação.

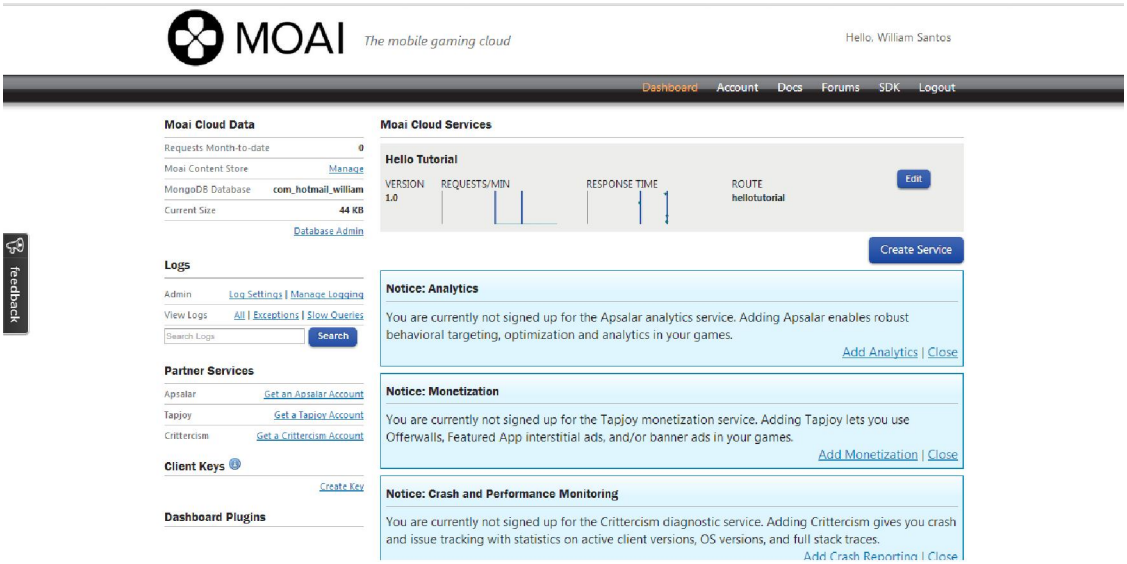

Figura 4: Painel de controle do Moai

4.4.3 Löve 2D

Löve é um framework destinado a programação de jogos 2D em linguagem Lua. É distribuído totalmente sem custos e pode ser utilizado livremente para quaisquer fins. Funciona em plataforma Windows, Linux e Mac OS, porém já existem projetos experimentais para distribuições web e *mobile*. A distribuição de jogos feitos no Löve2D é feita simplesmente zipando a pasta com o conteúdo do jogo (código fonte, imagens, etc) substituindo a extensão “.zip” por “.love”. É necessário observar que o usuário final deverá ter o Löve2D instalado para executar o jogo. Por essa razão, recomenda-se distribuir o jogo acoplado ao executável do Löve em um único pacote.

O Löve2D provê suporte a sons, imagens, simulações de física entre corpos, *threads*, eventos, entre outros. Além disso, existem as funções *callback*, usadas para tarefas como carregamento, desenho, ou quando determinadas teclas são pressionadas. A documentação é descrita em uma *wiki*, na qual são descritas as principais funcionalidades da ferramenta em diversos idiomas, inclusive português.

4.4.4 Gideros Studio

Criado pela empresa Gideros Mobile, o Gideros Studio é um ambiente de desenvolvimento de aplicações *mobile* para iOS e Android. É distribuído através de planos de assinatura anual, porém é possível utilizar a versão gratuita sem limitações perante as demais. Nesta versão, uma tela é adicionada na aplicação desenvolvida e exibida sempre antes de seu carregamento.

Diferente das ferramentas citadas anteriormente, o Gideros Studio é uma IDE completa que oferece um amplo conjunto de ferramentas composto por console de saída, simulador com diferentes resoluções, ferramentas para imagem e fontes, além do editor de código com realce de sintaxe e autocomplemento. Possui suporte a depuração de código e possibilidade de realizar testes instantâneos em dispositivos reais através de conexão Wi-Fi.

No Gideros Studio há suporte para simulações físicas, sensores de toque e acelerômetro, sons, sprites, entre outros. Possui ampla documentação em seu próprio site, onde também são disponibilizados tutoriais e modelos básicos de aplicações para download.

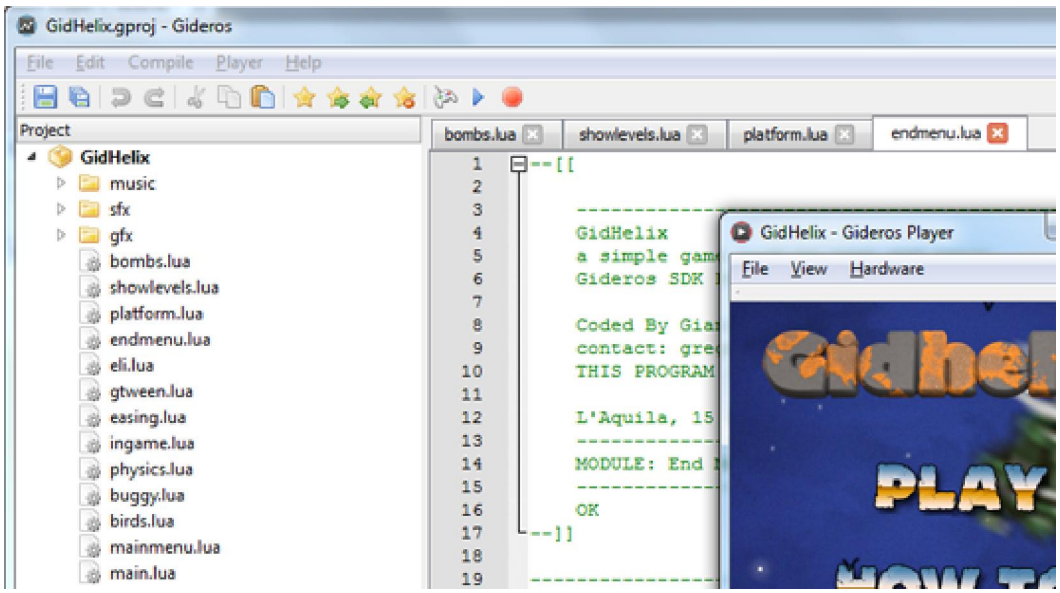

Figura 5: Tela do Gideros Studio

4.4.5 Comparação

Analisando as funcionalidades oferecidas pelas quatro ferramentas, o site *Game From Scratch* disponibiliza um quadro comparativo com os principais recursos de cada uma delas.

	Corona	Gideros	Löve	Moai
Preço	US$ 199 por ano para iOS ou Android; US$ 349 por ano para ambas as plataformas	US$ 149 por ano na versão Indie US$ 449 por ano na versão Pro	Sem custos	Sem custos
Limitações de versão gratuita	Impossibilidade de publicação de aplicações em lojas	Tela inicial imposta antes do carregamento da aplicação	N/A	N/A
Plataformas Alvo	iOS Android	iOS Android (Mac e Windows em desenvolvimento)	Windows Mac Linux	iOS Android Windows Mac Linux Chrome
Plataformas de desenvolvimento	Windows Mac	Windows Mac	Windows Mac Linux	Windows Mac Linux
Suporte	Forum Suporte pago	Forum	Forum	Forum Suporte pago
Código Aberto	Não	Não	Sim	Sim
Outros detalhes	Geração da build nos servidores da Corona Labs	Possui uma IDE própria		Oferece serviços em nuvem
Jogos publicados	Go Ninja The Lorax	Joustin Beaver Cerberus: The Puppy	Não divulgado	Crimson Steam Pirates Strikefleet Omega

Quadro 3: Matriz de características de *engines* (Game From Scratch, 2012, tradução nossa)

Conforme o Quadro 3, cada uma das ferramentas possuem seus pontos fortes e fracos, cabendo ao desenvolvedor escolher a que melhor se aplica ao projeto que deseja desenvolver. O Corona SDK apresenta a melhor documentação e disponibilidade de material para aprendizado, além de possuir vários títulos de sucesso publicado, porém seu custo é elevado e não possui código aberto. Moai apresenta menor volume de documentação e o programador pode encontrar maiores dificuldades em trabalhar com esta ferramenta quando comparada com as demais. Porém, é mais flexível e oferece os serviços em nuvem, sendo este um

grande diferencial. Assim como o Corona, o Moai possui várias publicações de jogos de sucesso. O Löve2D é a ferramenta mais adequada para iniciantes, graças a sua sintaxe amigável e a grande quantidade de material de referência. Porém está mais relacionada a desenvolvimento “por hobby”, visto que não possui títulos de sucesso publicados. Ao contrário das outras três, o Löve apresenta a desvantagem de ainda não possuir suporte para jogos mobile. Por fim, o Gideros Studio apresenta vantagens como IDE robusta, boa documentação disponível e facilidades durante a programação, sendo indicado para desenvolvedores com pouca experiência. Porém, assim como o Corona, o valor de sua licença é elevado.

5 IMPLEMENTAÇÃO DE UM JOGO BÁSICO

Com o objetivo de demonstrar o desenvolvimento de um jogo utilizando a linguagem Lua, foi escolhido utilizar o *framework Löve2D*. Para a escolha desta ferramenta, foram levados em consideração os seguintes pontos a se destacar:

- Licença gratuita para quaisquer fins;
- Grande quantidade de documentação disponível, inclusive em língua portuguesa.
- Sintaxe amigável e auto intuitiva;
- Facilidade de distribuição do jogo desenvolvido.

Embora ainda não haja suporte nativo para dispositivos *mobile*, é possível encontrar projetos experimentais para tal tarefa. Porém, para este trabalho, o jogo será desenvolvido apenas para plataforma Windows, utilizando a versão 0.8.0 do Löve2D. Para a escrita do código fonte é possível utilizar qualquer editor de texto. Para este projeto foi escolhido utilizar o Notepad++, que possui suporte para linguagem Lua.

5.1 CONCEITOS BÁSICOS

O Löve2D pode ser baixado diretamente do site do desenvolvedor, onde o usuário pode escolher a versão de acordo com seu sistema operacional. O diretório de instalação do programa contém o arquivo love.exe (na versão Windows), responsável por executar uma aplicação desenvolvida no Löve2D. A execução pode ser realizada de duas formas: através de uma pasta ou de um arquivo zipado com a extensão .love. Em ambos os casos, basta arrastá-los para o arquivo love.exe.

Qualquer aplicativo desenvolvido no Löve2D deve obrigatoriamente conter um arquivo principal chamado main.lua, responsável por carregar a aplicação no Löve2D. Caso o desenvolvedor queira definir o tamanho da tela, alterar o título da

janela, ou outras configurações adicionais, estas informações devem estar contidas no arquivo conf.lua.

5.2 FUNÇÕES CALLBACK

Durante o desenvolvimento, podem ser utilizadas funções especiais chamadas funções *callback*. Cada função é responsável por uma tarefa e são todas opcionais.

- love.load: esta função é responsável por carregar recursos, inicializar variáveis e outras definições em geral. É chamada apenas uma vez, ao inicializar a aplicação;

- love.update: esta função é encarregada por realizar cálculos em função da variável chamada dt (delta time), sendo esta o número de segundos entre uma chamada e outra da função. É chamada continuamente durante a execução e geralmente contém a lógica para realizar movimentos, animações, etc.

- love.draw: utilizada para desenhar objetos na tela, como imagens, textos e formas por exemplo. Assim como a função anterior, é chamada continuamente.

- love.keypressed e love.keyreleased: são funções chamadas ao pressionar e soltar uma tecla respectivamente. Recebem a tecla como parâmetro e a partir disso podem tomar decisões durante a execução.

- love.mousepressed e love.mousereleased: semelhantes à função anterior, porém, são chamadas ao clicar e soltar um botão do mouse. Recebem como parâmetro o botão e as coordenadas do clique.

- love.focus: esta função é chamada quando a janela da aplicação obtém ou perde o foco. Por exemplo, no caso de estar executando um jogo e o usuário clique fora da janela ou abra outro aplicativo, esta função será chamada, podendo determinar que o jogo fique pausado.

- love.quit: esta função é chamada quando a aplicação encerra a sua execução. Pode ser utilizada para salvar o progresso do jogo, por exemplo.

5.3 API DO LÖVE2D

A API do Löve2D é dividida em módulos, organizados em um único módulo principal chamado love. De acordo com a documentação do *framework*, o quadro abaixo descreve cada um dos módulos e suas funcionalidades:

Módulo	Descrição
love.audio	Oferece recursos para criação e manipulação de sons
love.event	Gerencia eventos, como tecla pressionada
love.filesystem	Oferece uma interface para gerenciar arquivos
love.font	Permite trabalhar com fontes
love.graphics	Responsável por gerenciamento de imagens, animações, etc.
love.image	Permite decodificar imagens codificadas através de sua interface
love.joystick	Provê uma interface com o joystick
love.keyboard	Provê uma interface com o teclado
love.mouse	Provê uma interface com o mouse
love.physics	Permite simulações de física entre corpos 2D
love.sound	Permite decodificar arquivos de som
love.thread	Oferece recursos para trabalhar com threads
love.timer	Provê uma interface para o relógio

Quadro 4: Os módulos do Löve2D

Os módulos oferecem funções específicas para tarefas relacionadas com a categoria. Por exemplo, pra desenhar um retângulo na tela, basta chamar a função `love.graphics.rectangle(mode, x, y, width, height)`, substituindo os parâmetros por valores desejados.

5.4 INICIANDO UM PROJETO

Conforme anteriormente citado, o arquivo main.lua é o ponto de partida de uma aplicação escrita no Löve2D. O jogo desenvolvido para este trabalho tem como objetivo abordar as principais funções *callback* e parte da API do *framework*. Será desenvolvido um jogo baseado no clássico *Space Invaders*, cujo objetivo é destruir naves invasoras.

O arquivo main.lua terá inicialmente a função abaixo responsável por carregar o personagem principal do jogo, codificado utilizando uma tabela em Lua.

```
function love.load()
  nave = {}        -- tabela para armazenar propriedades da nave
  nave.x = 300     -- coordenadas x e y
  nave.y = 400
  nave.w = 42      -- definições do tamanho da nave
  nave.h = 40
  nave.vel = 150   -- velocidade na qual a nave se movimentará
  nave.imagem = love.graphics.newImage("nave.png")  -- imagem da nave
end
```

Na função love.load uma nova tabela é criada com o nome de “nave”. Desta forma é possível armazenar propriedades neste objeto, como coordenadas x e y, largura e altura, velocidade e a imagem que será exibida na tela. É interessante notar que ao contrário da maioria das linguagens, em Lua é possível definir as propriedades fora da própria tabela.

Após carregar o objeto nave na função anterior, será preciso definir a maneira em que o personagem vai se movimentar na tela. Esta definição será codificada na função love.update.

```
function love.update(dt)
  if love.keyboard.isDown(“left”) then
    nave.x = nave.x – nave.vel * dt
  elseif love.keyboard.isDown(“right”) then
    nave.x = nave.x + nave.vel * dt
  end
end
```

A função acima testa se o usuário pressionou a tecla seta para esquerda ou seta para direita e, a partir disso, subtrai ou acresce o valor definido como velocidade multiplicado pela variável “dt”. Isso faz com que o objeto movimente-se pelo eixo x da tela.

Para que o objeto seja desenhado na tela, é preciso utilizar a função love.draw. Nesta função também será desenhado um retângulo verde utilizado para representar o chão. É importante observar o uso da função responsável por seleção de cores. Após escolher uma cor, todos os objetos a seguir serão desenhados utilizando esta mesma cor até que um novo valor seja definido.

```
function love.draw()
  love.graphics.setColor(0,255,0)   -- Cor verde para desenhar o chão
```

```
  love.graphics.rectangle("fill", 0, 465, 800, 150)

  love.graphics.setColor(255,255,255) -- Selecionando a cor branca
  love.graphics.draw(nave.imagem, nave.x, nave.y)
end
```

Na função acima, são utilizadas três funções. A função love.graphics.setColor recebe como parâmetros valores inteiros que vão de 0 a 255 para cada cor, na sequência vermelho, verde, azul e alfa (RGBA). O valor de alfa pode ser utilizado opcionalmente para definição de transparência da imagem. A função love.graphics.rectangle é responsável por desenhar um retângulo na tela, recebendo como parâmetros o modo de desenho (preenchido ou contornado), as coordenadas x e y, e por fim, os valores de largura e altura. A função love.graphics.draw é semelhante à anterior, porém é utilizada para desenhar objetos na tela.

Neste ponto, é possível executar o jogo e visualizar a nave movimentando-se de acordo com as teclas de direção. A próxima etapa do desenvolvimento será a criação dos invasores. Para esta tarefa, será criada uma tabela chamada "inimigos", na qual serão armazenados objetos do tipo "invasor" utilizando um laço de repetição *for*. O código abaixo deverá estar contido dentro da função love.load.

```
inimigos = {}        -- tabela para armazenar os invasores

for i=0, 6 do
  invasor = {}       -- tabela para armazenar propriedades do invasor
  invasor.w = 54
  invasor.h = 42
  invasor.imagem = love.graphics.newImage("invasor.png")
  invasor.x = i * (invasor.w + 60) + 30
  invasor.y = invasor.h + 100
  table.insert(inimigos, invasor)    -- função para inserir na tabela
end
```

O trecho de código acima cria a tabela "inimigos" e adiciona 7 elementos do tipo "invasor" nesta tabela. A tabela "invasor" por sua vez, armazena as propriedades do objeto como coordenadas, tamanho e imagem. Em seguida, os invasores podem ser desenhados na tela através da função love.draw:

```
love.graphics.setColor(255,255,255)  -- Seleciona a cor branca
for i,v in ipairs(invasores) do      -- laço for genérico de Lua
  love.graphics.draw(v.imagem, v.x, v.y)
end
```

O uso do *for* genérico é utilizado para percorrer todos os elementos de uma tabela em Lua. No caso do trecho de código acima, a função irá iterar sobre os

pares (1, invasores[1]), (2, invasores[2]) sucessivamente até o fim da tabela. Em seguida, a imagem armazenada na tabela “invasor” é desenhada utilizando o valor “v” retornado pela função.

Para que os invasores se movimentem em direção ao chão, será preciso adicionar o seguinte trecho de código na função love.update:

```
for i,v in ipairs(invasores) do
  v.y = v.y + 30 * dt    -- invasores vão descendo em direção ao chão

  if v.y > 465 then      -- caso atinja o limite do chão
                         -- o jogador perde.
  end
end
```

O próximo passo é fazer com que a nave seja capaz de disparar contra os invasores. Para iniciar este procedimento, será necessário criar uma tabela que armazene os tiros disparados pela nave. Esta tabela deverá estar armazenada dentro da própria tabela de definições da nave. Na função love.load, logo abaixo da linha que define a imagem da nave, deverá ser adicionado o seguinte código:

```
nave.tiros = {}             -- armazena os tiros disparados pela nave
```

Em seguida, deve ser criada a seguinte função:

```
function atirar()
  local disparo = {}      -- cria uma tabela para armazenar o disparo
  disparo.x = nave.x + nave.w / 2     -- posição de x no meio da nave
  disparo.y = nave.y                  -- posição de y no topo da nave
  table.insert(nave.tiros, disparo)     -- insere o disparo na tabela
end
```

O tiro deverá ser disparado assim que o jogador pressionar uma tecla. Neste caso será utilizada a barra de espaço. Portanto, a função love.keyreleased será implementada da seguinte forma:

```
function love.keyreleased(key)   -- função chamada ao soltar a tecla
  if (key == " ") then           -- caso pressionou a barra de espaço
    atirar()                     -- chama a função atirar
  end
end
```

O trecho de código acima poderia estar localizado na função love.update, conforme a rotina de movimentação foi implementada. Porém, ao utilizar a função love.keyreleased o tiro será disparado apenas quando o usuário soltar a tecla,

impedindo que ele possa manter a tecla pressionada e continuar disparando contra os inimigos.

Na função love.update será adicionado um trecho de código responsável por atualizar o movimento dos tiros disparados e checar se houve colisão com o inimigo. Uma nova função deverá ser criada para detectar se houve colisão entre dois objetos, onde serão passados como parâmetros as coordenadas e dimensões dos objetos a e b, retornando um valor booleano true caso seja verdadeiro. Esta função pode ser encontrada na própria documentação do Löve2D

```
function CheckCollision (ax1,ay1,aw,ah, bx1,by1,bw,bh)
  local ax2,ay2,bx2,by2 = ax1 + aw, ay1 + ah, bx1 + bw, by1 + bh
  return ax1 < bx2 and ax2 > bx1 and ay1 < by2 and ay2 > by1
end
```

O jogo será atualizado na medida em que os tiros sejam disparados e os inimigos sejam atingidos. Desta forma, duas novas tabelas locais serão criadas para armazenar quais tiros e inimigos devem ser removidos da tela. Os tiros serão removidos na medida em que sua posição do eixo y for menor do que zero ou quando atingirem um inimigo. Os inimigos serão removidos assim que uma colisão com o tiro for detectada. Para que isso ocorra, o código abaixo deverá estar na função love.update:

```
local removeInimigo = {}
local removeTiro = {}

for i,v in ipairs(nave.tiros) do        -- percorre a tabela de tiros
  v.y = v.y - 100 * dt                  -- movimenta o tiro para cima

  if v.y < 0 then                    -- caso o tiro não esteja visível
    table.insert(removeTiro, i)          -- então o tiro será removido
  end

  if ii, vv in ipairs(inimigos) then        -- checa se houve colisão
    if CheckCollision(v.x,v.y,2,5,vv.x,vv.y,vv.w,vv.h) then
      table.insert(removeInimigo, ii)      -- o inimigo será removido
      table.insert(removeTiro, ii)            -- o tiro será removido
    end
  end
end

for i,v in ipairs(removeInimigo) do    -- remove os inimigos marcados
  table.remove(inimigos, v)
end

for i,v in ipairs(removeTiros) do         -- remove os tiros marcados
  table.remove(nave.tiros, v)
end
```

Por fim, é necessário acrescentar na função love.draw o trecho de código responsável por desenhar os tiros na tela. Os tiros armazenados na tabela correspondente são percorridos utilizando o *for* genérico, e para cada um deles, um pequeno retângulo de 2px x 5px é desenhado na tela para representar o disparo.

```
for i,v in ipairs(nave.tiros) do        -- percorre a tabela de tiros
  love.graphics.rectangle("fill", v.x, v.y, 2, 5)
end
```

O resultado do projeto desenvolvido é exibido na imagem abaixo:

Figura 6: Jogo desenvolvido utilizando o Löve2D

6 CONSIDERAÇÕES FINAIS

Não por acaso Lua alcançou o reconhecimento por grande parte dos desenvolvedores. Com a constante evolução e aprimoramentos obtidos no decorrer dos anos, a linguagem se estabeleceu no mercado como uma importante ferramenta para programação, indo além do ambiente de criação de jogos. Lua possui listas de discussão, boa documentação e diversos livros publicados, porém, pouco material em língua portuguesa.

A popularidade de Lua no desenvolvimento de jogos é evidenciada através das pesquisas realizadas pelo site gamedev.net, o que surpreende inclusive seus próprios desenvolvedores, uma vez que a linguagem foi projetada para outros fins. Entretanto, este fato também não aconteceu por acaso. No desenvolvimento de jogos, a escolha da linguagem deve considerar pontos importantes como portabilidade, simplicidade, desempenho, facilidade de integração e robustez. Lua oferece este conjunto de vantagens sem qualquer custo.

Embora seja uma linguagem desenvolvida no Brasil, ainda é pouco conhecida no país, o que explica a pouca quantidade de publicações em idioma nativo. Espera-se que este cenário mude com a contribuição dos profissionais da área, na divulgação desta linguagem em meios profissionais e acadêmicos.

7 REFERÊNCIAS BIBLIOGRÁFICAS

DALMAZO, Bruno Lopes; AVELAR, Francisco Tiago. **Estudo sobre a linguagem de programação Lua**. 2007. Disponível em: www.infovisao.com/arquivos/lua_doc.pdf. Acesso em 14 fev. 2012.

FERNANDES, Michelle M. **Corona SDK Mobile Game Development Beginner's Guide**. Birmingham: Packt Publishing Ltd, 2012.

GAMASUTRA, **The Engine Survey: General results**. Disponível em: http://www.gamasutra.com/blogs/MarkDeLoura/20090302/581/The_Engine_Survey_General_results.php. Acesso em 18 set. 2012

GAME FROM SCRATCH. **Battle of the Lua Game Engines: Corona vs. Gideros vs. Love vs. Moai**. Disponível em: http://www.gamefromscratch.com/post/2012/09/21/Battle-of-the-Lua-Game-Engines-Corona-vs-Gideros-vs-Love-vs-Moai.aspx. Acesso em 05 nov. 2012.

IERUSALIMSCHY, Roberto. **About Lua** – Foster City, 2012.

IERUSALIMSCHY, Roberto. **Programming in Lua** – 2nd ed. – Rio de Janeiro, 2006.

IERUSALIMSCHY, Roberto. **Uma Introdução à Programação em Lua**. 2009. Disponível em: http://www.lua.org/doc/jai2009.pdf. Acesso em 29 fev. 2012.

IERUSALIMSCHY, Roberto. **A Evolução de Lua** – Rio de Janeiro, 2008. Disponível em: http://www.inf.puc-rio.br/~roberto/talks/luapyconf.pdf. Acesso em 17 set. 2012.

IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique; CELES, Waldemar. **The evolution of an extension language: A history of Lua** – Curitiba, 2001. Disponível em: http://www.lua.org/history.html. Acesso em 27 mar. 2012.

IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique; CELES, Waldemar. **The evolution of Lua**. 2007. Disponível em: http://www.lua.org/doc/hopl.pdf. Acesso em 04 nov. 2012.

IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique; CELES, Waldemar. **A Linguagem Lua e suas Aplicações em Jogos**. 2004. Disponível em: http://www.lua.org/doc/wjogos04.pdf. Acesso em 06 fev. 2012.

IERUSALIMSCHY, Roberto; FIGUEIREDO, Luiz Henrique; CELES, Waldemar. **Lua 5.1 Reference Manual**. 2006. Disponível em: http://www.lua.org/manual/5.1. Acesso em 29 fev. 2012

JUNG, Kurt; BROWN, Aaron. **Beginning Lua Programming**. Indianapolis: Willey Publishing, 2007.

PROJETO KEPLER. **Lua - Conceitos Básicos e API C**. 2008. Disponível em: http://www.keplerproject.org/docs/apostila_lua_2008.pdf. Acesso em 13 set. 2012.

PUC-RIO, **Lua**. Disponível em: http://www.lua.org. Acesso em 18 set. 2012

WARD, Jeff. **What is a Game Engine**. 2008. Disponível em: http://www.gamecareerguide.com/features/529/what_is_a_game_.php. Acesso em 13 out. 2012