Usando MoonScript com LÖVE

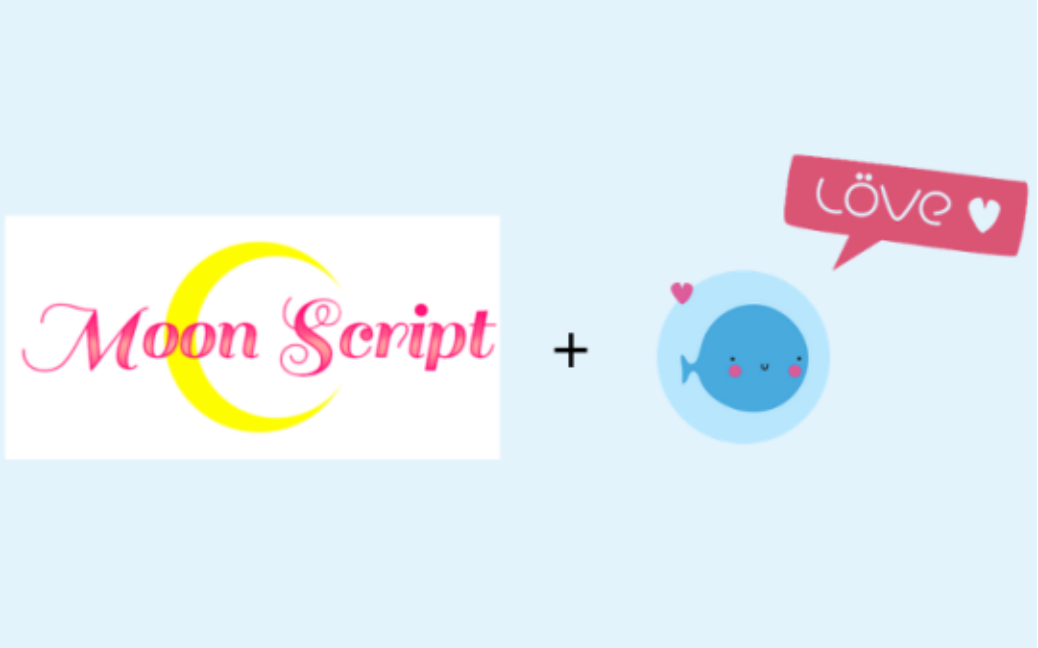
Canoi Gomes

25 de Fevereiro de 2017

Conteúdo

1 Introdução

Nesse tutorial, pretendo abordar um pouco sobre como usar o MoonScript, que é basicamente uma linguagem que é **transpilada** para Lua, com a framework LÖVE.



Mas qual a vantagem de usar uma terceira linguagem ao invés de usar somente Lua direto, se no fim tudo será convertido para ela mesmo? De fato, o uso de MoonScript se dá mais por alguns recursos que a linguagem implementa onde se fossemos fazer puramente em Lua seria necessário um trabalho a mais ou (o mais indicado) usar um biblioteca externa. Alguns exemplos como Classes, os famigerados `+=` ou `-=` .

1.1 Instalando o MoonScript

Aqui não vou me aprofundar muito nisso, mas você pode acessar o site do MoonScript para fazer o setup no seu sistema operacional. No fim você terá ba-

sicamente dois executáveis, um **moon** para rodar os códigos MoonScript, e um **moonc** responsável por compilar o código para Lua. E pra rodar é bem simples, só chamar no terminal:

```
mooncc main.moon
```

2 Criando Nosso main.moon

Bom, vamos começar criando um arquivo **main.moon**, que seria relativo ao main.lua esperado pelo LÖVE. Aqui você vai escrever as funções esperadas pela LÖVE (load, update e draw), porém usando a sintaxe do MoonScript.

```
love.load = ->
love.update = (dt) ->
love.draw = ->
```

Fácil, né? Agora roda aquele comando anterior, o moonc, para compilar nosso código. Já adiantando um pouco aqui, você também pode rodar o comando com **moonc *.moon** e ele compila todos o .moon no diretório atual.

Mas é isso, não tem muito mistério. Porém agora gostaria de mostrar um pouco de uma aplicação prática.

3 Criando uma Aplicação Prática

Então vamos brincar um pouco com os recursos do MoonScript pra fazer algo legal. Vamos criar uma classe **GameObject** :

```
class GameObject
    x: 0
    y: 0
    width: 32
    height: 32
```

```
    update: (dt) =>
    draw: () =>
```

Vamos criar tudo no main.moon mesmo, é o suficiente para esse exemplo. Bom, perceba aqui que que criei uma classe com algumas propriedades, um coisa a se atentar é que caso um desses atributos seja uma tabela, você vai precisar criar um construtor. Aqui é necessário isso devido a maneira como as tabelas se comportam dentro do Lua, que seria como ponteiros em C, então diferente de um atributo de number ou string onde os seus dados são copiados, em uma tabela somente sua referência é passada adiante.

Por exemplo:

```
class Teste
    attributes: {}
    add_value: (value) =>
        table.insert @attributes, value

obj0 = Teste!
obj1 = Teste!

print obj0.attributes[1] -- irá printar nil
print obj1.attributes[1] -- irá printar nil também

obj0\add_value 1 -- adiciona o valor 1 à tabela

print obj0.attributes[1] -- irá printar 1
print obj1.attributes[1] -- irá printar 1 também

obj3 = Teste!
print obj3.attributes[1] -- também irá printar 1
```

Note que como a referência é passada adiante, então os dados da tabela serão compartilhado por todas as instâncias da classe. Com certeza você irá querer fazer

isso em alguns casos, mas caso contrário, nós podemos utilizar um recurso das próprias classes do MoonScript, que são os construtores. Para isso é só definirmos uma função **new** dentro da nossa classe.

```
class Teste
    attributes: {}
    new: =>
      @attributes = {}
    add_value: (value) =>
      table.insert @attributes, value

obj0 = Teste!
obj1 = Teste!

print obj0.attributes[1] -- irá printar nil
print obj1.attributes[1] -- irá printar nil também

obj0\add_value 1 -- adiciona o valor 1 à tabela do obj0

print obj0.attributes[1] -- irá printar 1
print obj1.attributes[1] -- aqui irá printar nil
```

Voltando para o exemplo anterior, vamos extender nossa classe de GameObject para criar uma classe para o Player.

```
class Player extends GameObject
    update: (dt) =>
        if love.keyboard.isDown "left"
            @x -= 200 * dt
        if love.keyboard.isDown "right"
            @x += 200 * dt
    draw: () =>
        love.graphics.rectangle "line", @x, @y, @width, @height
```

Aqui não tem muito mistério, criei a classe Player que é filho de GameObject,

e implemento algumas funções dentro dela para update e draw. Agora é só criar uma instância da classe e então chamar seus métodos dentro das funções da LÖVE.

```
player = Player!

love.load = ->
love.update = (dt) ->
    player\update dt
love.draw = ->
    player\draw!
```

4 Considerações

Algumas outras coisas interessantes de fazer é executar o comando moonc no modo espectador. Ele vai ficar checando constantemente se houve alguma mudança no arquivo, e caso tenha, irá compilá-lo. Isso pode ser feito usando a opção **-w**:

```
moonc -w main.moon
```

Outra opção interessante é a **-t**, onde você pode definir o caminho para onde os arquivos compilados serão jogados. Acho bacana essa estrutura para usar no seu projeto:

```
game/
    assets/
    ~lua code~
src/
    ~moon code~
```

E daí só executar:

```
moonc -t game/ -w src/
```

E com isso todo o conteúdo necessário para rodar seu projeto estará em game/, é só empacotar tudo e distribuir.