Matemática e programação em Hex

Autor: Luan Gustavo Orlandi[1] Orientador: Leandro Fiorini Aurichi

1 de setembro de 2015

[1]Este projeto foi desenvolvido durante iniciação científica do autor, com bolsa CNPq no ICMC-USP.

Sumário

Capítulo 1

Introdução

1.1 O que é o projeto?

Durante a iniciação científica do autor foi desenvolvido o projeto "Matemática e programação em Hex", em que foi estudado como utilizar a matemática na programação de uma maneira que outras pessoas que não dominam essas áreas possam aprender essas disciplinas de uma forma diferente. O projetou levou um ano até alcançar seus resultados, ocorrendo entre o segundo semestre de 2014 até o início do segundo semestre de 2015. Durante esse período foi trabalhado a construção de um jogo eletrônico que abordasse conteúdos matemáticos usando programação.

A iniciação científica recebeu o apoio do CNPq durante todo o desenvolvimento do projeto e como resultado final o projeto conta com o jogo chamado Hex e esta apostila que apresenta conceitos das áreas de matemática e programação utilizados no desenvolvimento do projeto.

1.2 Objetivos do projeto

O resultado desse projeto é o jogo eletrônico chamado Hex e esta apostila, que contém explições de várias partes do desenvolvimento e código do jogo para que o leitor conheça toda a parte matemática e programação relevantes para o seu aprendizado.

O objetivo principal é ensinar ao leitor os conceitos de matemática que foram aplicados no jogo, bem como a importância e utilidade dessa área no desenvolvimento de jogos. Esta apostila foi confeccionada afim de alcançar esse objetvo, sua escrita foi feita da melhor forma possível para que qualquer pessoa consiga entender o seu conteúdo, porém alguns assuntos podem ser compreendidos com mais facilidade se o leitor já possuir conhecimentos das áreas abordados no jogo. Será mostrado também alguns conceitos de programação que facilitam a construção do jogo.

A seguir é descrito um pouco de cada um dos conceitos usado no jogo e que a apostila discute com mais detalhes em outros capítulos.

A geometria analítica é um dos conceitos que será discutido. No jogo são aplicados operações com vetores e manipulação de matrizes em vários momentos, mostrando como foi feito o uso na linguagem de programação.

Há uma opção de jogar contra uma inteligência artificial que utiliza uma análise para os seus turnos baseada na teoria dos jogos, em que a melhor jogada a ser feita pode ser influenciada com a reação que o oponente irá decidir em seu turno.

A inteligência artificial também precisa analisar o tabuleiro afim de calcular o quão próximos estão da vitória os dois jogadores. Esse cálculo é feito medindo distâncias por meio de um grafo. O algoritmo de Dijkstra participa nessa análise, sendo também necessária otimizações conhecidas que reduzam o tempo que leva para determinar o resultado, pois dessa maneira a inteligência artificial realiza turnos mais rapidamente.

Em todo o código é usada a programação orientada a objetos para não só organizar o código, mas também facilitar a sua codificação, sendo um dos principais conceitos utilizado no desenvolvimento de qualquer jogo. Ao longo da apostila serão mostradas partes do código que fazem o uso desse conceito frequentemente, apresentando ao leitor de uma maneira simples para que este possa compreender a funcionalidade contida no código.

1.3 O jogo

Hex é um jogo de tabuleiro baseado em turnos para ser jogado por dois jogadores. Ele foi criado há anos mas sua popularidade atualmente não é muito grande. O jogo desenvolvido no projeto tenta seguir fielmente às regras do jogo elaborando sua versão eletrônica.

1.3.1 Um pouco da história

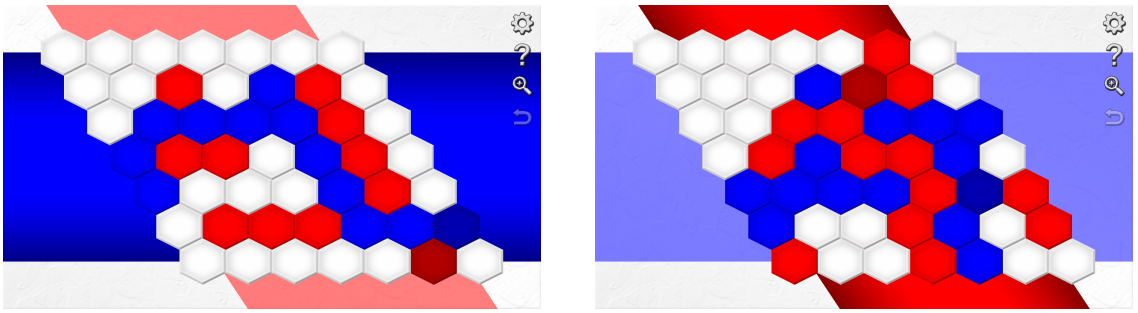
O jogo foi inventado pelo matemático dinamarquês Piet Hein, que o introduziu em 1942 no Instituto Niels Bohr. Foi independentemente re-inventado em 1947 pelo matemático John Nash na Universidade de Princeton. Tornou-se conhecido na Dinamarca sob o nome de Polygon (embora Hein o chamou de CON-TAC-TIX); Nash disse a alguns jogadores que inicialmente chamou o jogo de Nash. De acordo com Martin Gardner, alguns dos alunos da Universidade de Princeton também se referiram ao jogo como John (de acordo com algumas fontes isso foi porque eles jogaram o jogo usando o mosaico do chão do banheiro). No entanto, de acordo com a biografia de Sylvia Nasar sobre John Forbes Nash, A Beautiful Mind, o jogo ficou conhecido como "Nash"ou "John"depois de seu criador aparente. John Nash falou que pensou neste jogo, independente de Hein, durante seus anos de pós-graduação na Universidade de Princeton. Em 1952, a Parker Brothers comercializou uma versão. Chamaram sua versão "Hex", e o nome pegou.[1]

1.3.2 Como jogar

Hex é jogado com dois jogadores e tem um tabuleiro formado por hexágonos em que as bordas do tabuleiro possuem cores.

Os jogadores têm cores diferentes e cada um em seu turno escolhe um hexágono livre para sua jogada, deixando o hexágono selecionado com a sua cor.

O objetivo é completar um caminho de hexágonos adjacentes de mesma cor que liguem as bordas opostas do tabuleiro que correspondem à cor do jogador. As figuras a seguir mostram momentos finais do jogo em que algum jogador venceu.



Vitória para o jogador azul (esquerda) e para o jogador vermelhor (direita)

Em sua primeira versão, o jogo desenvolvido permite partidas locais de dois jogadores e partidas contra uma inteligência artificial. É possível escolher, além desses dois modos de jogo, o tamanho do tabuleiro, jogador que inicia o primeiro turno e alternar borda que cada jogador deve conectar.

No jogo desenvolvido há algumas opções para o usuário, como por exemplo desfazer uma joagada (note que sua utilidade é evitar erros não intencionais, logo o jogo disponibiliza a função temporariamente) e ampliar o tabuleiro, em caso do dispositivo ser pequeno para pressionar os hexágonos.

Capítulo 2

Ferramentas

No desenvolvimento do projeto foram utilizadas várias ferramentas para a programação do jogo e elaboração desta apostila. Os tópicos neste capítulo apresentam as ferramentas mais relevantes e um breve resumo de cada uma para o leitor conhecê-las.

2.1 Lua

Qualquer jogo ou programa precisa de uma linguagem de programação, neste jogo é utilizado Lua.

Lua é uma linguagem brasileira (como o próprio nome já sugere) criada pela universidade PUC-Rio em 1993. Muitas aplicações hoje trabalham com Lua, pois ela oferece uma codifição simples e poderosa para o desenvolvedor, mantendo alta performance no programa.[2]

A linguagem tem grande destaque na área de jogos, sendo que vários jogos muito famosos utilizam Lua em seus códigos, como por exemplo World of Warcraft, SimCity 4 e FarCry.[3] Alguns motores gráficos também usam Lua, como exemplo a *engine* MOAI que foi utilizada no desenvolvimento do jogo Hex.

2.2 MOAI

Criado em 2011, MOAI é o motor gráfico que sustenta o jogo realizando a renderização e efeitos necessários, também oferece várias funções e recursos para o desenvolvimento de um aplicativo. Embora seja voltado para desenvolvedores profissionais, em um jogo simples como o deste projeto, MOAI não é muito difícil de se trabalhar.

MOAI ainda é bem recente, mas já existem vários títulos que usam a *engine*, como por exemplo Broken Age.[4]

2.3 LaTeX e TeXnicCenter

Na elaboração desta apostila foi feita a codificação dos textos em LaTeX, que é uma ferramenta na produção de textos com alta qualidade muito popular na confecção de artigos e livros matemáticos de médio e grande porte.[5]

Para trabalhar com o LaTeX, foi usado o TeXnicCenter. Esse programa é um editor voltado para textos em LaTeX.

Capítulo 3

Programação orientada a objetos

Para facilitar a leitura da apostila, primeiramente é importante entender o que é a programação orientada a objetos e como ela foi utilizada na linguagem Lua.

3.1 O que é programação orientada a objetos?

A orientação a objetos é um modelo de análise, projeto e programação de sistemas de *software* baseado na composição e interação entre diversas unidades de software chamadas de objetos.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de *software*. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.[6]

É muito importante não confundir programação orientada a objetos (muitas vezes resumida pelas siglas POO) com programação estruturada. Na programação estruturada é possível acessar qualquer dado em todo o código, enquanto que em POO existe um encapsulamento dos dados, mantendo organização no código.

Há vários conceitos que constituem esse modelo de programação, nas seguintes seções será explicado os principais conceitos para entender o uso de POO no jogo desenvolvido.

3.1.1 Classe

É a definição de algo que possui características e ações.

Para exemplificação, vamos definir a classe de um carro. A classe Carro possui propriedades que todos os carros devem ter, como por exemplo velocidade, cor e peso. Um Carro também pode realizar ações, como "andar", virar rodas e abrir as portas. Essas características e ações que uma classe possui são definidas como atributos e métodos, respectivamente.

3.1.2 Objeto

Em poucas palavras, objeto é a instância de uma classe.

Continuando com o exemplo do carro, uma vez que temos definido sua classe, podemos criar vários objetos do tipo Carro, sendo que cada um possui atributos diferentes, como por exemplo 2 ou 4 portas. O objeto pode interagir com os outros objetos que há no código, não necessariamente com apenas os que são de sua classe.

3.2 Programação orientada a objetos em Lua

Embora Lua não seja uma linguagem puramente orientada a objetos, ela fornece meta-mecanismos para a implementação de classes e herança. Os meta-mecanismos de Lua trazem uma economia de conceitos e mantêm a linguagem pequena, ao mesmo tempo que permitem que a semântica seja estendida de maneiras não convencionais.[2]

A uso de programação orientada a objetos pode ser exemplificada no código abaixo:

```
Carro = {}
Carro.__index = Carro

function Carro:novo(posicao, numeroDePortas, cor)
   local C = {}
   setmetatable(C, Carro)

   C.cor = cor
   C.numeroDePortas = numeroDePortas

   C.pos = posicao
   C.vel = 0

   return C
end
```

As duas primeiras linhas formam a maneira em que podemos definir uma classe em Lua, criando uma tabela que pode conter dados.

Em seguida temos o construtor da classe, o principal método pois podemos criar os objetos da classe ao utilizar essa função. `C` é uma tabela para incluirmos os dados que o carro possui. Como se trata do construtor, os dados iniciais que um carro possui são generalizados ou definimos por meio argumentos recebimos na chamada do método.

A cor, número de portas e posicao são atribuídas aos atributos do classe, enquanto que a sua velocidade é definida como 0, assim todo carro ao ser criado terá essas propriedades determinadas pelo construtor da classe.

No final, a tabela `C` é retornada para instanciar a classe. O código abaixo exemplifica a criação de alguns carros:

```
carro1 = Carro:new(0, 4, ‘‘cinza’’)
```

```
carro2 = Carro:new(5, 4, ‘‘preto’’)
carro3 = Carro:new(12, 2, ‘‘branco’’)
```

Da mesma maneira que definimos o construtor, podemos definir os demais métodos da classe. Abaixo estão alguns métodos que a classe poderia ter de uma maneira exemplificada, e em seguida a utilização desses métodos:

```
function Carro:acelerar(ace)
   self.vel = self.vel + ace
end

function Carro:mover()
   self.pos = self.pos + self.vel
end

carro1:mover()
carro2:acelerar(4)
carro2:mover()
```

O `self` faz com que o objeto a ser usado seja aquele que chamou o método, em outras palavras `self` é um objeto passado por parâmetro. Em outras linguagens, como Java, a escrita é feita com `this`, porém a funcionalidade é a mesma.

O código abaixo é uma maneira mais explícita (pois é possível ver o objeto sendo passado por argumento e a qual classe ele pertence) que pode ser usada ao chamar os métodos, porém no restante da apostila será mostrada como visto anteriormente.

```
Carro.mover(carro1)
Carro.acelerar(carro2, 4)
Carro.mover(carro2)
```

Capítulo 4

Interface gráfica

O jogo precisa fornecer ao usuário as funcionalidades disponíveis, como por exemplo desfazer uma jogada e exibir as regras do jogo. Essas funções são acessadas através de uma interface gráfica, que é composta pelos botões que o jogador pode interagir.

Este capítulo mostra como foi feito a interface gráfica utilizando vetores e geometria analítica, bem como a leitura dos dispositivos de entrada (no caso o *mouse* e a tela de toque).

4.1 Vetores e geometria analítica

Nesta seção será mostrado como é foi feita a representação de vetores no jogo e a operação básica de soma de vetores.

O vetor $\overrightarrow{V} = \overrightarrow{(x, y)}$ em $\mathbb{R}^2$ será definido como um objeto da classe `Vector`. O código da classe está localizado em "data/math/vector.lua".

4.1.1 Criação de vetores

```
Vector = {}
Vector.__index = Vector

function Vector:new(a, b)
   local V = {}
   setmetatable(V, Vector)

   V.x = a
   V.y = b

   return V
end
```

Apesar de já ter sido discutido no capítulo 3.2 como é a definição de uma classe em Lua, será explicado novamente mas agora com uma classe que há no jogo.

As duas primeiras linhas definem em Lua que `Vector` é uma classe. A função `Vector:new(a, b)` é um construtor que cria um novo vetor com coordenadas x em a e y em b, sendo que a e b são recebidos por parâmetros, retornando o objeto com esses 2 atributos.

A linha `local V = {}` cria uma variável local (poderia ser global, mas para evitar que variáveis fiquem alocadas sem uso, todas as funções criam variáveis locais) que recebe uma tabela.

Tabela em Lua é a forma que um objeto pode ser representado quando construído nesta linguagem. É possível também usar tabelas para representar registros, listas e vetores.

Em seguida a linha `setmetatable(V, Vector)` indica que a tabela criada é da classe `Vector`, assim o objeto pode ter acesso aos métodos de sua classe. As linhas seguintes criam os atributos para o objeto, que neste caso são as coordenadas x e y do vetor.

Por fim, a tabela `V` é retornada pela função. Quem receber o retorno desta função será um objeto com os atributos `x` e `y`.

4.1.2 Soma de vetores

```
function Vector:sum(b)
   self.x = self.x + b.x
   self.y = self.y + b.y
end
```

O método acima faz a soma de dois vetores, somando as coordenadas x e y (no caso os atributos dos objetos) de um vetor com o de outro. É utilizado `self` para indicar que é o próprio objeto, neste caso o vetor que chamou este método, usar seus atributos na função.

4.2 Retângulos

Com os vetores definidos, podemos representar retângulos usando vetores.

O código da classe está localizado em "data/math/rectangle.lua".

4.2.1 Criação de retângulos

```
Rectangle = {}
Rectangle.__index = Rectangle

function Rectangle:new(c, v)
   local R = {}
   setmetatable(R, Rectangle)

   R.center = Vector:new(c.x, c.y)
```

```
   R.size = Vector:new(v.x, v.y)

   return R
end
```

Os argumentos `c` e `v` são vetores que serão os atributos do centro e tamanho do retângulo. O atributo `center` indica a posição do centro do retângulo no plano e `size` é a distância do centro até um de seus cantos (metade de sua diagonal), com esses valores é possível encontrar os demais cantos do retângulo. Veja o exemplo:

```
centro = Vector:new(2, 1)
tamanho = Vector:new(1, 1)

r = Rectangle:new(centro, tamanho)

A = Vector:new(r.center.x + r.size.x, r.center.y + r.size.y)
B = Vector:new(r.center.x - r.size.x, r.center.y + r.size.y)
C = Vector:new(r.center.x + r.size.x, r.center.y - r.size.y)
D = Vector:new(r.center.x - r.size.x, r.center.y - r.size.y)
```

Os cantos do retângulo `r` são `A`, `B`, `C` e `D` e suas posições são, respectivamente, (3, 2), (1, 2), (3, 0), (1, 0).

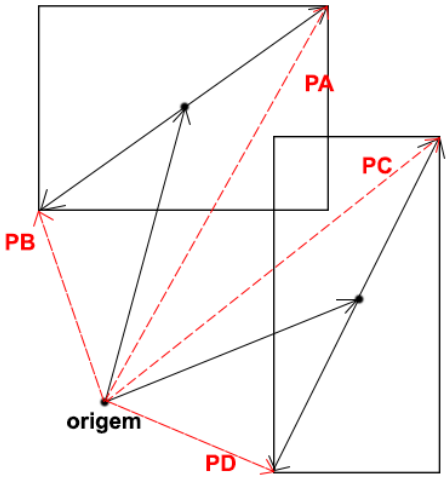
4.2.2 Interseção entre retângulos

O método abaixo realiza a interseção de dois retângulos, retornando verdadeiro caso houver interseção em algum ponto entre eles.

```
function Rectangle:intersection(b)
   local PA = Vector:new(self.size.x, self.size.y)
   PA:sum(self.center)

   local PB = Vector:new(-self.size.x, -self.size.y)
   PB:sum(self.center)

   local PC = Vector:new(b.size.x, b.size.y)
   PC:sum(b.center)

   local PD = Vector:new(-b.size.x, -b.size.y)
   PD:sum(b.center)

   if PA.x > PD.x and PB.x < PC.x then
      if PA.y > PD.y and PB.y < PC.y then
         return true
      end
```

```
    end

    return false
end
```

Primeiramente a função calcula 2 pontos opostos de cada retângulo e compara os pontos mais próximos entre os retângulos: se o maior ponto de um com o menor do outro é maior, e também os pontos mais distantes entre eles: se o menor ponto de um com o maior do outro é menor, em suas respectivas coordenadas.



Interseção entre retângulos usando vetores

A figura acima ilustra cada vetor criado pelo método e o resultado com as operações realizadas. Nesse exemplo a verificação retorna verdadeiro, pois `PA.x > PD.x(true)`, `PB.x > PC.x(true)`, `PA.y > PD.y(true)` e `PC.y > PB.y(true)`.

4.2.3 Ponto dentro de um retângulo

No método `Rectangle:pointInside(p)` ocorre de maneira semelhante ao da interseção entre dois retêngulos vista em 4.2.2, a diferença é que um dos retângulos é um ponto, ou seja, seus cantos possuem valores iguais em suas respectivas coordenadas. Podemos facilmente adaptar o método `Rectangle:intersection(b)` para verificar uma interseção com um ponto.

```
function Rectangle:pointInside(p)
    local PA = Vector:new(self.size.x, self.size.y)
    PA:sum(self.center)

    local PB = Vector:new(-self.size.x, -self.size.y)
```

```
   PB:sum(self.center)

   if PA.x > p.x and PB.x < p.x then
      if PA.y > p.y and PB.y < p.y then
         return true
      end
   end

   return false
end
```

O argumento `p` é o ponto que será verificado se está dentro ou não do retângulo, sendo mais direta a verificação das condições, pois somente é necessário determinar quais são os pontos de um retângulo, diferentemente do método `Rectangle:intersection(b)`.

A classe `Rectangle` é utilizada nos botões da interface e a seleção desses botões é através de um *click/tap* (um ponto) dentro de um retângulo (oculto ao usuário) que constituí a área de seleção do botão.

4.3 Leitura dos dispositivos de entrada

Dispositivos de entrada são os meios que jogo usa como controle para jogar, no caso o *mouse* e o *touch screen*. A leitura desses dispositivos permite identificar o estado atual e a posição onde houve a ação, por exemplo um *click* no centro da tela é definino pelo pressionamento do botão esquerdo do *mouse* e a posição no centro da tela onde o cursor estava naquele momento. Com essas informações é possível determinar qual ação a aplicação deve tomar.

A classe `Input`, localizada em "data/input/input.lua", descreve a leitura dos dipositivos de entrada.

```
Input = {}
Input.__index = Input

function Input:new()
   I =
   setmetatable(I, Input)

   I.pointerPos = Vector:new(0, 0)
   I.pointerPressed = false

   I.selection = nil

   I.pointerPressedCancel = false
```

```
   return I
end
```

No atributo `pointerPos` será armazenado a posição do cursor, como um vetor. Em todo o jogo é definida a origem no centro da tela, entretanto nesse caso a origem é localizada no canto superior esquerdo da janela. Note que na tela sensível ao toque o valor da posição `pointerPos` somente altera quando há um toque na tela.

Em `pointerPressed` indica o estado de pressionamento (`true` para pressionado e `false` para não pressionado). O objeto selecionado será armazenado em `selection`, podendo ser um hexágono ou um botão. O atributo `pointerPressedCancel` auxilia na seleção, fazendo com que se houve um *click/tap* num lugar vazio, não é possível selecionar nenhum objeto a partir desse estado, enquanto que se houve uma seleção em um objeto ainda é possível trocar a seleção feita naquele estado movendo o *cursor*.

```
function Input:mouseActive()
   if MOAIInputMgr.device.pointer then
      MOAIInputMgr.device.pointer:setCallback(onMouseMoveEvent)
      MOAIInputMgr.device.mouseLeft:setCallback(onMouseLeftEvent)
   end
end

function Input:touchActive()
   if MOAIInputMgr.device.touch then
      MOAIInputMgr.device.touch:setCallback(onTouchEvent)
   end
end
```

Os métodos `Input:mouseActive()` e `Input:touchActive()` ativam, respectivamente, a leitura dos dispositivos de entrada para o *mouse* e o *touch screen*. Essas leituras são feitas por meio de funções "callback", que consistem em chamar outras funções quando um determinado evento ocorre, no caso é quando o estado de alguma entradada sofrer mudanças.

```
function onMouseMoveEvent(x, y)
   input.pointerPos.x = x
   input.pointerPos.y = y
end

function onMouseLeftEvent(down)
   input.pointerPressed = down
end

function onTouchEvent(event, idx, x, y, tapCount)
   if event == MOAITouchSensor.TOUCH_MOVE then
      input.pointerPos.x = x
      input.pointerPos.y = y
```

```
   end

   if event == MOAITouchSensor.TOUCH_DOWN then
      input.pointerPressed = true
      input.pointerPos.x = x
      input.pointerPos.y = y
   end
if event == MOAITouchSensor.TOUCH_UP or
      event == MOAITouchSensor.TOUCH_CANCEL then

      input.pointerPressed = false
      input.pointerPos.x = x
      input.pointerPos.y = y
   end
end
```

Por último, temos o método `Input:dealWithPointerPressed()` que lida com a mudança de estado no pressionamento do botão do *mouse* ou na tela de toque. Por ser muito grande, o código desse método não será exibido aqui. Sua funcionalidade é importante, toda vez que o usuário pressiona na tela verifica se houve seleção em algum botão ou num hexágono, ou para o caso de já ter um objeto selecionado e ter soltado o toque/*click* é ativada a funcionalidade daquele objeto.

4.4 Interface

Graphical User Interface (GUI) ou Interface Gráfica do Usuário trata-se da utilização de técnicas de design e usabilidade voltadas para a interação entre homens e máquinas, por meio de softwares, sistemas ou aplicações. Esta modalidade consiste em aperfeiçoar o modo como os usuários utilizam, um equipamento ou um sistema de computação.[7]

O jogo usa uma interface para permitir o jogador interagir com a aplicação. Esse tópico foca a interação usando a parte matemática desenvolvida no projeto. Inicialmente será discutido apenas os botões e em seguida os hexágonos do tabuleiro.

A classe `Interface`, localizada em "data/interface/interface.lua", cria uma nova camada de renderização dos objetos, que serão os botões, evitando que se misturem com os demais objetos na tela que podem ser interagidos como botões, como por exemplo as faixas, dessa maneira ao aplicar uma ampliação na tela pela funcionalidade de *zoom*, esses objetos continuam em suas posições e não sofrem mudanças.

```
Interface = {}
Interface.__index = Interface

function Interface:new(viewport)
   local I = {}
```

```
   setmetatable(I, Interface)

   I.layer = MOAILayer2D.new()
   I.layer:setViewport(viewport)

   I.gameInterface = nil

   return I
end
```

O atributo `layer` é a nova camada cirada pela função do MOAI, `viewPort` é um elemento da janela onde a camada precisa ser renderizada (apesar de ser importante, não tem muita relevância para ser discutido os detalhes) e `gameInterface` é um outro objeto da interface (porém mais específico ao jogo) que será criado posteriormente assim que outros objetos do jogo são criados.

Os métodos seguintes da classe são os mais importantes. Há vários outros métodos, porém, como já foi esclarecido, o objetivo é explicar somente aquilo que é mais relevante ao leitor.

```
function Interface:createGameInterface()
   self.gameInterface = GameInterface:new()
end

function Interface:getButton(pos)
   local newPos = Vector:new(pos.x, -pos.y)
   local windowCorner = Vector:new(-window.resolution.x/2,
                                   window.resolution.y/2)

   newPos:sum(windowCorner)

   if self.gameInterface  = nil then
      return self.gameInterface:getButton(newPos)
   end
end
```

No `Interface:createGameInterface()` é criado um novo objeto da classe `GameInterface`, que conterá os botões e vários detalhes que ajustam proporcionalmente os botões no canto da tela. Esse objeto será melhor discutido em breve.

Em `Interface:getButton(pos)` é feita uma busca por qual botão foi pressionado pelo usuário. O método é chamado pela função `Input:dealWithPointerPressed()` na leitura do dispositivo de entrada para encontrar o botão selecionado.

O argumento `pos` é o vetor correspondente à posição do *cursor* ou onde há a mudança de estado no toque da tela, note que especifcamente esse vetor tem como origem o canto superior esquerdo da tela, enquanto que o restante do jogo lida com os vetores em origem no centro da tela. Para tratar esse problema, é criado o novo vetor `newPos`, contendo a informação da

posição e origem no centro (em relação à origem no canto) somado ao vetor da posição do *cursor*, porém note que o valor da coordenada y cresce quanto mais "para baixo" o ponto está (em relação à origem no canto), por isso é colocado com negativo (`newPos.y = - pos.y`). desta forma é possível usar esse vetor final nas operações com os demais vetores do jogo.

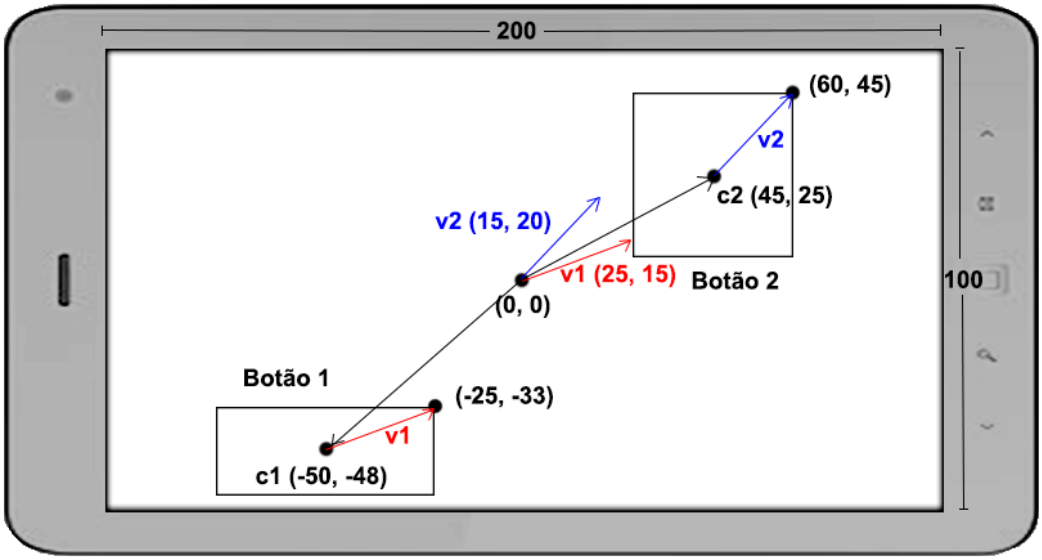
4.4.1 Botões

Com os objetos `Interface` e `GameInterface` criados são gerados vários botões na tela pertencentes a classe `Button`, ficando armazenados como atributo em `GameInterface`. Pelo fato dessas classes serem compridas, será mostrado essa parte com exemplos mais simples, sendo que a aplicação feita no jogo segue a mesma ideia, mas com vários botões e detalhes sobre eles.

```
c1 = Vector:new(-50, -48)
v1 = Vector:new(25, 15)
area1 = Rectangle(c1, v1)

gameInterface.botao1 = Button:new(c1, area1, "acao1", desenho1)

c2 = Vector:new(45, 25)
v2 = Vector:new(15, 20)
area2 = Rectangle(c2, v2)

gameInterface.botao2 = Button:new(c2, area2, "acao2", desenho2)
```

A criação do botão tem seu código localizado em "data/interface/button.lua". Os parâmetros necessários para definir o novo botão são a posição, área de seleção, nome de sua ação (irá determinar o que ocorre quando o jogador aperta o botão, a classe possui uma lista das possíveis ações programadas que o botão pode usar) e seu desenho. Este último representa a textura que o objeto recebe no momento de ser renderizado na tela, é muito comum chamar esse desenho de *sprite* na área de jogos.

O objeto `gameInterface`, que no jogo é um atributo da classe `Interface`, agora contém atributos de objetos do tipo `Button`. A imagem a seguir ilustra como é representado os botões na interface usando os vetores e retângulos usados de exemplo.



Representação dos botões numa tela de resolução 200x100

Logo, dado uma posição `pos` onde o usuário fez um *click/tap* podemos usar o método `getButton(pos)` que irá percorrer os botões disponíveis na interface e checar pelo método `checkSelect(pos)` em cada um deles para saber em que botão o jogador pode ter pressionado.

```
function Button:checkSelect(pos)
   return self.area:pointInside(pos)
end
```

Note que a `pos` e `self.area` representam, respectivamente, um vetor posição do *click/tap* e um retângulo contendo o *sprite* do botão onde o usuário deve pressionar para fazer a ação do botão.

4.4.2 Hexágonos

Além dos botões, o jogador pressiona nos hexágonos da tela para realizar o turno. A forma como feita a seleção para botões não seria muito boa nos hexágonos do tabuleiro, pois são muitos hexágonos e essa quantidade varia conforme o tamanho do tabuleiro, sendo muito lenta verificações um a um hexágonos.

O método `getHexagon(pos)` da classe `Board` é quem realiza a seleção dos hexágonos, o código está localizado em "data/game/board.lua". A função é comprida, mas cada detalhe é importante ser discutido, exceto partes que usam a tela do jogo ampliada. Para facilitar o entendimento será considerado que a tela está sempre do mesmo tamanho, não havendo ampliação, logo, partes do código da função a ser exibido foram omitidos, afim de simplificar a leitura.

A seguir será discutido aos poucos o código esse método.

```
function Board:getHexagon(pos)
   local hex = nil

   local hexPos = Vector:new(0, 0)
```

Nessa primeira parte recebe de argumento o vetor posição `pos` onde o jogador fez o *click/tap*, assim como nos botões mostrado em 4.4. Note que mais uma vez esse vetor tem a origem no canto superior esquerdo.

A variável `hex` será o objeto da classe `Hexagon` que representa o hexágono selecionado (se de fato foi selecionado), e `hexPos` será o vetor auxiliar para encontrar o hexágono.

```
   hexPos.y = pos.y - window.resolution.y/2 + self.start.y
                                            + self.hexagonSize.y/2
   hexPos.y = hexPos.y / (0.75 * self.hexagonSize.y)

   local row = math.floor(hexPos.y)

   hexPos.x = pos.x - window.resolution.x/2 - self.start.x
              + self.hexagonSize.x/2 - ((row - 1) * self.hexagonSize.x/2)
   hexPos.x = hexPos.x / self.hexagonSize.x

   local column = math.floor(hexPos.x)
```

Alguns detalhes importantes nessa parte:

- `window.resolution` é um vetor que indica o tamanho da janela do jogo (resolução em *pixels*), como uma diagonal da tela, sendo que o canto superior direito é dado pelas coordenadas `window.resolution.x/2` e `window.resolution.y/2`, assim como a coordenada inferior esquerda em `- window.resolution.x/2` e `- window.resolution.y/2`;
- de maneira análoga à `window.resolution`, `o self.hexagonSize` (atributo de `Board`) é o tamanho do hexágono em *pixels*;
- `self.start` é o vetor posição do primeiro hexágono, o mais superior da esquerda, no plano;

Observando o tabuleiro como uma matriz, podemos ver o tabuleiro construído tem suas colunas variando conforme a linha, pois o formato do tabuleiro não é um quadrado ou retângulo, mas sim algo próximo de um paralelogramo. Então devemos encontrar a linha (coordenada y) e depois a coluna (coordenada x).

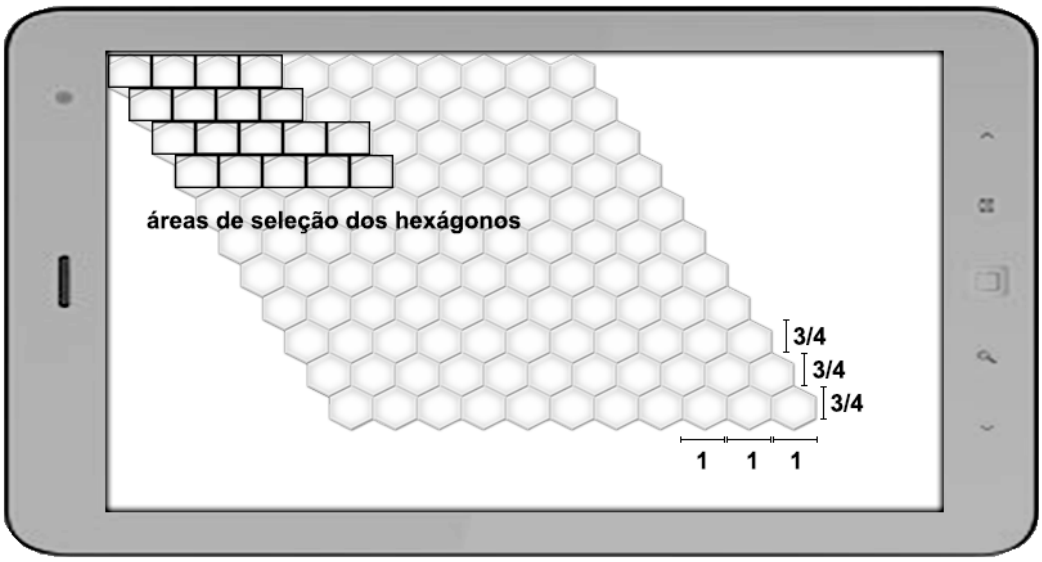
Primeiramente ajustamos o tabuleiro para que esteja iniciando (primeiro hexágono) no canto da janela, facilitando as contas, pois no jogo ele está centrado no meio da tela. Subtraindo o tamanho da tela `window.resolution.y` e somando posição inicial `self.start.y`

(pois lembrando que `pos.y` cresce quanto mais "para baixo") resulta o centro do primeiro hexágono, localizado no canto esquerdo superior, ficar exatamente nesse canto, mas ainda temos que somar um pedaço do tamanho do hexágono para que o jogador consiga selecionar a partir da sua lateral, isto é, somamos com `self.hexagonSize.y/2`. Em seguida, somando com a posição `pos.y`, temos a localização onde foi feito o *click/tap*.

O passo seguinte é determinar a linha que foi selecionada. Note que cada linha possui 3/4 do tamanho (na vertical) do hexágono, logo dividimos por esse valor (ou multiplica por 0.75) para encontrar a linha.

Essa mesma conta é feita para a coordenada x, porém subtraímos em `self.hexagonSize.x/2`, pois `pos.x` cresce "para a direita", e subtraímos `self.hexagonSize.x/2 - ((row - 1) * self.hexagonSize.x/2)`, pois a cada linha, cada coluna avança em metade do tamanho do hexágono. Por fim dividimos pelo tamanho (na horizontal) inteiro do hexágono para determinar sua coluna. A operação *floor* é aplicado porque as variáveis `row` e `column` fazem o acesso na matriz do tabuleiro, sendo permitido somente valores inteiros.

A imagem a seguir ilustra essa parte do código.



Tabuleiro posicionado no canto para o cálculo da seleção do hexágono

O resultado é uma caixa retangular de seleção em cada hexágono, porém temos que aprimorar essa seleção para que fique hexagonal. A próxima parte do código desse método explica como isso foi tratado.

```
hexPos.y = hexPos.y % 1
```

```
if hexPos.y < 1/3 then
   -- y - y0 = m(x - x0)
   local y0 = 1/3
   local x0
   local side

   hexPos.x = hexPos.x % 1

   if hexPos.x > 0.5 then
      -- lado direito
      hexPos.x = hexPos.x % 0.5
      side = -1
      x0 = 0
   else
      -- lado esquerdo
      side = 1
      x0 = 0.5
   end

   local m = side * ((1/3) / 0.5)

   hexPos.y = 1/3 - hexPos.y

   -- y >= y0 + m(x - x0)
   if hexPos.y >= y0 + (m * (hexPos.x - x0)) then
      row = row - 1

      if side == -1 then
         column = column + 1
      end
   end
end
```

Nesse parte é usado o conceito de equação da reta em geometria analítica. Podemos representar a área de seleção de um hexágono como um função num gráfico, veja a imagem a seguir:

Canto do hexágono representado numa função

A primeira linha do código mostrado faz com que consideremos área de seleção do hexágono com tamanho 1. Como `hexPos` representa a coordenada do hexágono numa matriz, fazendo a operação de módulo 1 (em Lua %) resulta no ponto onde houve o *click/tap* supondo tamanho 1 e nova origem relativa para a caixa de seleção do hexágono.

O caso em que pode haver erro na hora selecionar o hexágono é quando o ponto está na parte de cima da área de seleção, mais específicamente a 2/3 acima. Lembrando que a coordenada y do vetor posição do `click/tap` cresce quanto mais "para baixo". Então a verificação `hexPos.y < 1/3` nos coloca para analisar esse caso da seleção.

$$y - y0 = m(x - x0)$$

Dada a equação da reta acima podemos representá-la nessa situação, porém consideramos cada metade do hexágono com equações diferentes. A variável `side` auxilia a identificar qual lado usar para a equação, sendo que o lado direito é quando `hexPos.x > 0.5` e o esquerdo `hexPos.x < 0.5`.

Em seguida calculamos o coeficiente angular $m = \tan\alpha$, em que α é o ângulo que determina a reta no plano cartesiano. Pela conta `m = side * ((1/3) / 0.5)` encontramos o coeficiente, que dependendo do lado que está sendo analisado.

No que os valores -1 e 1 de `side` foram propositais, pois pelo fato de a tangente de um ângulo ser o oposto da tangente de seu ângulo suplementar (ângulos em que a soma resulta em 180 graus), `side` auxilia no cálculo da tangente resultando no valor oposto para lado direito do hexágono.

Logo, para o caso em que $y >= y0 + m(x - x0)$ o ponto está acima (fora do triângulo formado no gráfico) do hexágono, e portanto indica a seleção de um outro hexágono. Para

o lado esquerdo, esse outro hexágono está uma linha acima da matriz, ou seja, `row = row - 1`, enquanto que para o lado direito não só está uma linha acima, mas também está a uma coluna a direita, ou seja, `column = column + 1`.

```
   if row > 0 and column > 0 then
      if row <= self.size.x and column <= self.size.y then
         hex = self.hexagon[row][column]
      end
   end

   return hex
end
```

Na última parte do código é atribuído à variável `hex` criada no início no método com o hexágono que foi selecionado pelas contas.

Num tabuleiro de tamanho 11x11 a matriz dos hexágonos possui linhas e colunas de 1 à 11. Essa matriz é armazenada em `board.hexagon`. Para acessar o elemento de linha 3 e coluna 7, usamos `hex = board.hexagon[3][7]`.

Se foi selecionado um hexágono, ou seja, suas coordenadas estão dentro dos valores de 1 à 11 inclusive, a linha `hex = self.hexagon[row][column]` atribuí a `hex` o hexágono selecionado. Caso essas condições não satisfazem, é retorna uma variável com valor nulo, indicando que o vetor inicial `pos` não selecionou nenhum hexágono no tabuleiro.

Capítulo 5

Referências

[1] Wikipédia, Hex (jogo). Disponível em: <https://pt.wikipedia.org/wiki/Hex_(jogo)> Acesso em 16 de agosto de 2015.

[2] Lua, A Linguagem de Programação Lua. Disponível em: <http://www.lua.org/portugues.html> Acesso em 16 de agosto de 2015.

[3] lua-users, Lua Uses. Disponível em: <http://lua-users.org/wiki/LuaUses> Acesso em 16 de agosto de 2015.

[4] MOAI, Mobile Games Made with MOAI. Disponível em: <http://getmoai.com/made-with-moai.html> Acesso em 16 de agosto de 2015.

[5] LaTeX, An introduction to LaTeX. Disponível em: <http://latex-project.org/intro.html> Acesso em 16 de agosto de 2015.

[6] Wikipédia, Orientação a objetos. Disponível em: <https://pt.wikipedia.org/wiki/Orientação_a_objetos> Acesso em 22 de agosto de 2015.

[7] Romanino, Interface Gráfica do Usuário. Disponível em: <http://www.romanino.com.br/servicos/interface-grafica-do-usuario/> Acesso em 25 de agosto de 2015.