A geometria analítica no jogo Swift Space Battle

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

31 de agosto de 2014

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

Sumário

1 Introdução **5**
- 1.1 O que é o projeto? . . . 5
- 1.2 Objetivos . . . 5
- 1.3 O jogo . . . 5

2 Ferramentas **7**
- 2.1 Lua . . . 7
- 2.2 MOAI . . . 7
- 2.3 Notepad++ . . . 7
- 2.4 Adobe Photoshop CS6 . . . 8
- 2.5 LaTeX e TeXnicCenter . . . 8

3 A geometria analítica no jogo **9**
- 3.1 Vetor . . . 9
  - 3.1.1 Criação de vetores . . . 9
  - 3.1.2 Soma de vetores . . . 10
  - 3.1.3 Norma e normalização . . . 10
- 3.2 Retângulos . . . 11
  - 3.2.1 Criação de retângulos . . . 11
  - 3.2.2 Intersecção entre retângulos . . . 12
- 3.3 Áreas . . . 13
  - 3.3.1 Criação de áreas . . . 13
  - 3.3.2 Intersecção entre áreas . . . 14

4 A estrutura base do jogo **17**
- 4.1 Janela . . . 17
  - 4.1.1 Criando uma janela . . . 17
  - 4.1.2 Classe Screen . . . 18
- 4.2 Threads e Co-rotinas . . . 19
  - 4.2.1 Thread . . . 19
  - 4.2.2 Co-rotina . . . 22
- 4.3 Controles . . . 23

5 Utilização da geometria analítica **25**
5.1 Cenário . . . 25
5.1.1 Criação do cenário . . . 25
5.1.2 Movimentação do cenário . . . 27
5.2 Naves . . . 28
5.2.1 Classe Ship e movimentação da nave . . . 29
5.2.2 Classe Player . . . 31
5.2.3 Classe Enemies . . . 34
5.2.4 Classe EnemyType1 . . . 34
5.2.5 Classe EnemyType2 . . . 36
5.2.6 Classe EnemyType3 . . . 37
5.2.7 Classe EnemyType4 . . . 39
5.2.8 Classe EnemyType5 . . . 40
5.3 Classe Shot e colisão entre tiros e naves . . . 41

6 Fonte das texturas **45**

Capítulo 1

Introdução

1.1 O que é o projeto?

É uma pesquisa de Iniciação Científica, o resultado é jogo Swift Space Battle que em sua criação foram utilizados conceitos matemáticos de geometria analítica. Esta apostila também é parte do projeto, nela qualquer pessoa interessada pode encontrar detalhes e como foi utilizada a geometria analítica no desenvolvimento do jogo.

1.2 Objetivos

Ensinar o leitor os conteúdos de geometria analítica aplicados no jogo e a importância dessa área matemática em programação.

A apostila foi escrita 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 possuir conhecimentos em programação e geometria analítica.

1.3 O jogo

Swift Space Battle é um jogo casual de naves espacias. O jogador controla uma nave que fica posicionada no centro da tela e seu objetivo é eliminar as naves oponentes enquanto desvia de tiros dos inimigos. A nave pode ser controlada facilmente, fazendo com que o jogo seja simples.

Ao abater seus inimigos o jogador ganha pontos, quanto mais naves eliminar sem que o jogador seja acertado maior será o combo, ganhando cada vez mais pontos.

Ao longo do tempo a dificuldade do jogo aumenta. Conforme o jogador avança de nível as naves serão geradas mais rapidamente, o limite delas aumentam e outros tipos de naves mais difíceis surgem, aumentando o desafio para o jogador.

Capítulo 2

Ferramentas

No desenvolvimento do projeto foram utilizadas várias ferramentas para a programação do jogo e elaboração desta apostila.

Nos tópicos abaixo serão mencionadas cada uma dessas ferramentas e um breve resumo delas para o leitor conhecê-las.

2.1 Lua

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

Lua, como o próprio nome já sugere, é uma linguagem brasileira. Ela é bem recente e foi criada pela universidade PUC-Rio há mais de 20 anos. Muitas aplicações hoje trabalham com Lua, pois ela oferece uma codifição simples e poderosa para o programador, mantendo alta performance no programa.

Mais detalhes no site oficial da linguagem: http://www.lua.org/

2.2 MOAI

MOAI é a engine que sustenta o jogo, ela oferece várias funções e recursos para o desenvolvimento de um aplicativo. A linguagem necessária para trabalhar com ele é o Lua.

Embora seja voltado para desenvolvedores profissionais, em um jogo simples como o deste projeto, MOAI não é muito difícil de se trabalhar. Por ser uma engine bem recente há várias versões dela disponíveis, a deste projeto foi usada a versão 1.4p0.

Caso o leitor deseja saber mais sobre MOAI, acesse o site: http://getmoai.com/

2.3 Notepad++

Para a escrita do código foi usado o Notepad++, que é um editor simples de texto e também um dos mais sugeridos para a codificação em Lua.

Para conhecer mais sobre esse programa acesse: http://notepad-plus-plus.org/

2.4 Adobe Photoshop CS6

A ferramenta de edição na arte do jogo foi o Photoshop CS6.

Várias texturas como os sprites (desenhos) das naves são free e retiradas da internet, em seguida foram feitas edições nessa ferramenta para se adequarem mais ao jogo. Algumas artes, como o logo do título no jogo, são originais e criadas no Photoshop pelo próprio autor do jogo.

Essa ferramenta de edição é bem famosa, mas caso o leitor ainda não conhecer, veja mais sobre ela no site oficial da Adobe: https://www.adobe.com/br/products/photoshop.html

2.5 LaTeX e TeXnicCenter

Na elaboração desta apostila foi feita a codificação dos textos em LaTeX, que é uma ferramenta bem útil na confecção de artigos e livros matemáticos.

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

Saiba mais sobre cada um nos links abaixo:

http://latex-project.org/

http://www.texniccenter.org/

Capítulo 3

A geometria analítica no jogo

Neste capítulo será explicado a forma que foram definidos os conceitos de geometria analítica e cálculos em geral de matemática.

As definições das classes em Lua e outras coisas relacionadas a linguagem serão explicadas aos poucos, porém o código é bem semelhante a outras linguagens voltadas programação orientada a objetos.

3.1 Vetor

Nesta seção será mostrado como é feito o uso de vetores no jogo.

Para um vetor $\overrightarrow{V} = \overrightarrow{(x, y)}$, a classe de vetores que será mostrada irá construir e fazer operações com este vetor.

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

3.1.1 Criação de vetores

Abaixo está o código do construtor da classe Vetor:

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

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

   V.x = a
   V.y = b

   return V
end
```

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 estes 2 atributos.

A linha `local V = {}` cria uma variável local (poderia ser global, mas para evitar que variáveis fiquem alocadas sem uso, toda as funções estão criando 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 usá-las como registros, listas e vetores.

Em seguida a linha `setmetatable(V, Vector)` indica que a variável 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 e poderá usar os métodos de sua classe. A seguir serão explicados os métodos importantes para o vetor que são usados no jogo.

3.1.2 Soma de vetores

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

A função acima faz a soma de dois vetores, em que soma o x de um vetor com o de outro, análogamente para o y. É utilizado `self` para indicar que é o próprio objeto, neste caso o vetor que chamou este método, usar seus atributos na função. Em outras linguagens o `self` aparece como `this`, o código abaixo talvez seja mais fácil para o leitor compreender a função:

```
function Vector.sum2(a, b)
   a.x = a.x + b.x
   a.y = a.y + b.y
end
```

`Vector.sum2(a, b)` realiza a mesma função da anterior.

3.1.3 Norma e normalização

A norma de um vetor $\overrightarrow{V} = \overrightarrow{(x, y)}$ é aplicada seguindo a fórmula abaixo:

$$||\overrightarrow{V}|| = \sqrt{x^2 + y^2}$$

A método abaixo realiza esta operação. Nele é utilizado uma função de raiz quadrada da biblioteca `math` do Lua.

```
function Vector:norm()
   return math.sqrt((self.x)^2 + (self.y)^2, 2)
end
```

Agora que temos um método que encontra a norma de um vetor, podemos fazer a normalização que é calculada em geometria analítica pela fórmula abaixo, sendo $\overrightarrow{V}$ um vetor qualquer e $\overrightarrow{W}$ o vetor normalizado.

$$\overrightarrow{W} = \frac{1}{||\overrightarrow{V}||}$$

```
function Vector:normalize()
   local norm = self:norm()
   if norm ~= 0 then
      self.x = self.x / norm
      self.y = self.y / norm
   end
end
```

O método acima faz a normalização do vetor. É checado se a norma não é nula para evitar erros quando dividir com esta variável.

Abaixo está um exemplo de uso desta classe. O resultado final no `vetorA` é 0.8 em x e 0.6 em y.

```
vetorA = Vector:new(5, 1)
vetorB = Vector:new(-1, 2)

vetorA:sum(vetorB)

vetorA:normalize()
```

3.2 Retângulos

Esta classe é útil para determinar o tamanho das naves no jogo, colisão delas com os tiros e na interface do menu do jogo.

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

3.2.1 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 parâmetros `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 espaço 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)
```

Neste exemplo os cantos do retângulo são A, B, C e D e suas posições são, respectivamente, (3, 2), (1, 2), (3, 0), (1, 0).

Nessa classe há o método `Rectangle:copy(orientation, pos)` que cria um retângulo com atributos iguais aos do original, mas a diferença é que a posição dele (o centro) será oposto na coordenada y dependendo da orientação recebida como parâmetro, assim o método é útil para inverter os hitboxes das naves caso estejam viradas para o lado oposto, isso será explicado com mais detalhes posteriormente.

3.2.2 Intersecção entre retângulos

O método abaixo realiza a intersecção de dois retângulos, retornando verdadeiro caso houver intersecção em algum ponto entre eles. A função primeiro pega 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.

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

```
                        self.center.y + self.size.y)

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

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

   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
```

No método `Rectangle:pointInside(p)` ocorre de maneira semelhante ao da intersecção acima para o caso de um retângulo ser um ponto, ou seja, seus cantos possuem valores iguais em suas respectivas coordenadas. Essa função é usada na interface para ver se o usuário selecionou uma caixa de texto, sendo o local em que houve o click do mouse é o ponto e a caixa de texto é o retângulo.

3.3 Áreas

Na verificação de colisão entre os tiros e as naves, a classe retângulos seria suficiente para fazer esse cálculo, mas os desenhos das naves no jogo não são próximas de um retângulo. Para que seja mais eficiente, são usados vários retângulos pequenos para preencher o desenho da nave.

Observe que não precisamos ficar calculando a posição de todos os retângulos de intersecção cada vez que a nave mover ou quando o alvo está muito longe. Para o primeiro caso, o objeto desta classe trabalha com seus valores constantes e cada vez que é necessária fazer uma checagem de colisão, é criado uma área igual, mas posicionada no lugar atual da nave.

No caso de o alvo estar muito longe, esta classe Área possui um retângulo grande que será sempre o primeiro a ser checado, se a intersecção não ocorre nele, os demais retângulos menores também não, mas se houver intersecção, são verificados cada um dos retângulos menores. Logo esta classe assume o papel de hitboxes. Quanto maior a quantidade desses retângulos mais perto do desenho a colisão realmente acontece, mas aumenta o processamento.

O código dessa classe está localizado em “data/math/area”.

3.3.1 Criação de áreas

Abaixo está o construtor da classe Área:

```
Area = {}
Area.__index = Area

function Area:new(size)
   local A = {}
   setmetatable(A, Area)

   A.size = Rectangle:new(Vector:new(0, 0), Vector:new(size.x, size.y))
   A.hitbox = {}

   return A
end
```

O parâmetro `size` é o retângulo maior, onde os demais retângulos menores estarão dentro.

O atributo `hitbox` é uma tabela, que em outras linguagens de programação são usadas como listas ou simplesmente vetores. Nesta tabela haverá os retângulos menores que representam as áreas de acerto e colisão com os outros objetos.

Para colocar um retângulo na tabela `hitbox`, é utilizado o método abaixo:

```
function Area:newRectangularArea(c, v)
   local rectangle = Rectangle:new(c, v)
   table.insert(self.hitbox, rectangle)
end
```

Esse método recebe o centro e tamanho do retângulo a ser criado, em seguida o insere na tabela `hitbox` do objeto pertencente a ele. Esta função de inserção em Lua faz automaticamente a alocação de memória e o posiciona em uma posição vazia e adequada na tabela, começando por padrão na posição 1 (é possível especificar a posição desejada).

3.3.2 Intersecção entre áreas

Para fazer a intersecção entre vários retângulos de dois objetos, basta percorrer a tabela `hitboxes` entre cada um deles, mas sempre verificando se naquele retângulo maior há uma intersecção, senão o custo de processamento ficaria desnecessariamente alto. Veja o código abaixo:

```
function Area:detectCollision(orientationA, posA, b, orientationB, posB)
   local sizeA = self.size:copy(orientationA, posA)
   local sizeB = b.size:copy(orientationB, posB)
   local intersection = false

   if sizeA:intersection(sizeB) then
      local i = 1
```

```
      while not intersection and i <= table.getn(self.hitbox) do
         local rectangleA = self.hitbox[i]:copy(orientationA, posA)

         local j = 1
         while not intersection and j <= table.getn(b.hitbox) do
            local rectangleB = b.hitbox[j]:copy(orientationB, posB)

            intersection = rectangleA:intersection(rectangleB)
            j = j + 1
         end
         i = i + 1
      end
   end

   return intersection
end
```

O método recebe a posição de cada objeto que está sendo analisado e suas orientações, em que como foi visto anteriormente, indica qual lado o objeto está "olhando".

Primeiramento é feita a checagem se os alvos estão próximos, ou seja, se entre os retângulos maiores deles há intersecção. Os retângulos `sizeA` e `sizeB` são esses retângulos, eles representam de fato os retângulos (apenas o maior de cada um deles) que estão cobrindo os desenhos das naves no atual memento, daí a utilidade do método `Rectangle:copy(orientationA, posA)` citado anteriormente.

Caso a colisão entre os retângulos maiores ser verdadeira, é feita em seguida várias checagens dos retângulos menores para uma detecção de colisão mais eficiente entre os alvos. É feitos então dois laços percorrendo as tabelas `hitbox` da área de cada um até encontrar alguma intersecção, e se não houver, o método retorna que as áreas em análise não colidem. A função `table.getn()` retorna o tamanho da tabela.

Capítulo 4

A estrutura base do jogo

Neste capítulo será explicado alguns conceitos importantes para entender como o jogo surge, executa e permite interação com o usuário.

Os conteúdos abordados aqui ocorrem em qualquer jogo ou aplicativo, mas cada linguagem e engine possui uma maneira diferente de trabalhar com esses conteúdos.

No MOAI há varias coisas que são bem simples de serem feitas quando comparadas a outras engines, mas também há suas desvantagens, logo algumas decisões tomadas no desenvolvimento podem parecer não muito eficientes em termos de performance para o jogo, mas a ideia foi manter a simplicidade e fazer com que o leitor consiga entender melhor o que está acontecendo no código.

4.1 Janela

Para que toda a aplicação seja visível precisamos de uma janela, mas não a janela do console e sim uma que seja possível desenhar os objetos e fazer o jogo aparecer para o usuário.

A seguir, será explicado como é criado uma janela no MOAI e como podemos manipular seus atributos para que tudo fique com tamanhos proporcionais independentemente das dimensões da janela.

4.1.1 Criando uma janela

Abaixo segue um código exemplo que cria uma janela:

```
MOAISim.openWindow("exemplo", 1280, 720)

viewport = MOAIViewport.new()
viewport:setSize(1280, 720)
viewport:setScale(1280, 720)

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

```
MOAIRenderMgr.pushRenderPass(layer)
```

A função `MOAISim.openWindow` abre uma nova janela com o nome e dimensões de seus 3 parâmetros.

Para a versão do MOAI usada no projeto, no programa deve haver apenas uma janela com tamanho definido em sua criação e somente uma chamada desta função pode ocorrer. Para fechar a janela é preciso encerrar o programa, por isso é necessário reiniciar para aplicar uma nova resolução escolhida nas opções.

O `viewport` ajusta a tela e determina a origem que, por padrão, fica no centro da janela, logo podemos pensar o uso das coordenadas como num plano cartesiano.

O `layer` é uma camada em que os sprites do jogo podem ser pintados e será a única no jogo a ser utilizada. A linha seguinte define para o MOAI a renderização nessa camada, poupando ao programador vários detalhes necessários para desenhar qualquer objeto na tela.

4.1.2 Classe Screen

Localizada em "data/screen/screen", a classe Screen trata as configurações da janela, veja o código do construtor dela:

```
local ratio = 16 / 9
local defaultWidth = 1280
local defaultHeight = 720

Screen = {}
Screen.__index = Screen

function Screen:new()
   local S = {}
   setmetatable(S, Screen)

   local resolution = readResolutionFile()
   S.width = resolution.x
   S.height = resolution.y

   if S.width == nil or S.width == 0 or S.height == nil or S.height == 0 then
      S.width = MOAIEnvironment.horizontalResolution
      S.height = MOAIEnvironment.verticalResolution

      if S.width == nil or S.width == 0 or
         S.height == nil or S.height == 0 then

         S.width = defaultWidth
         S.height = defaultHeight
```

```
      end
   end

   if S.height / S.width < ratio then
      S.width = S.height / ratio
   else
      S.height = S.width * ratio
   end

   S.scale = S.height / 1280

   return S
end
```

Primeiro a função tenta ler de um arquivo ("file/option.lua") a resolução da janela, caso ocorrer algum erro é feita uma chamada de funções do MOAI para ler qual a resolução do monitor que o usuário está usando. Nesta versão do MOAI, essas funções falham na leitura e retornam 0 ou nil, logo é criado uma janela de tamanho padrão, definida nas variáveis `defaultWidth` e `defaultHeight`.

A variável `ratio` indica a proporção que o jogo tem. Para ajustar a janela nesta proporção, é reduzido o tamanho da largura ou a altura, dependendo de quem for maior nesta proporção de `ratio`.

No final, é criado uma outra variável que funciona como a unidade a ser utilizada no jogo, assim todos os sprites terão uma escala relacionada a este valor.

Agora, quando o objeto da classe screen ser criado, ele tem todos os atributos para criar uma janela e auxiliar o tamanho dos próximos objetos que precisam de alguma unidade de espaço.

O método `Screen:newWindow()` faz uma nova janela, assim como mostrado na seção 4.1.1, mas agora com tamanhos que podem variar.

4.2 Threads e Co-rotinas

4.2.1 Thread

Thread é uma função que é executada simultaneamente com o restante do programa. O jogo rodando será uma thread, em que no MOAI, podemos controlar a velocidade que o laço principal do jogo será executado, ou seja, a taxa de quadros por segundo (FPS) do jogo. Por padrão essa taxa está em 60, isso indica que a cada segundo, o laço será executado 60 vezes no máximo, pois se a quantidade de tarefas for muito grande e não for possível processar tudo nesse tempo, o jogo começa a ficar lento com a sensação de travamentos.

Abaixo está o código de duas threads sendo criadas e iniciadas, localizado em "main.lua".

```
local timeThread = MOAICoroutine.new()
timeThread:run(getTime)

local introThread = MOAICoroutine.new()
introThread:run(introLoop)
```

A função `MOAICoroutine.new()` cria uma thread, que também funciona como uma corotina, e retorna essa nova thread. O método run dela inicia uma execução em paralelo com o resto programa, permitindo outras threads serem iniciadas.

A váriavel `timeThread` é uma thread que pega o tempo decorrido desde a abertura do programa e guarda esse valor em outra variável que pode ser acessada pelos objetos no jogo, como por exemplo uma nave que verifica o intervalo entre seus tiros.

A thread `introThread` mostra a introdução do jogo e quando ela termina inicia outra thread que executa o menu. No menu do jogo, se o usuário selecionar a opção de começar um novo jogo, essa thread termina e uma nova thread do jogo inicia.

A função principal do jogo que roda em uma thread é a `gameLoop()`. Abaixo está o código da função, localizado em "data/loop/ingame.lua".

```
function gameLoop()
   local blackscreen = BlackScreen:new(3)
   blackscreen:fadeOut()

   spawner = Spawner:new()

   playerData = PlayerData:new()
   playerLevel = Level:new(spawner)

   interface = GameInterface:new(playerData.lives)

   map = Stage1:new()

   player = Player:new(Vector:new(0, 0))
   playerShots = {}

   enemies = {}
   enemiesShots = {}

   deadShips = {}

   playerData.active = true

   while playerData.active do
```

```
      coroutine.yield()

      shipsMove()
      enemiesShoot()

      shotsMove()
      shipsCheckStatus()

      map:move()

      player:shoot()

      spawner:spawn()

      resumeThreads()

      if interface.gameOver  = nil then
         interface.gameOver:checkSelection()
         interface.gameOver:checkPressed()
      end
   end
end
```

Todo o código que está fora do laço `while` da função será executado apenas uma vez no jogo, pois corresponde aos objetos que inicializam o jogo, como o cenário, nave do jogador e interface.

O `blackScreen` é um objeto com uma thread que faz o efeito de escuro ao normal na janela quando o jogo inicia, a duração desse efeito é o número passado por parâmetro e o método `blackscreen:fadeOut()` é quem inicia a thread com o efeito.

O `spawner` é um gerador de naves inimigas que controla a velocidade e quais naves devem aparecer de acordo com o nível atual no jogo. A cada certo período o `spawner` cria naves mais rapidamente e de diferentes tipos, aumentando a dificuldade no jogo.

Os objetos `playerData` e `playerLevel` guardam dados sobre o jogador, como quantidade de vidas restantes, pontuação, combo e nível atual.

A `interface` é a UI (interface do usuário), que corresponde aos dados mostrados na janela para o usuário saber as informações que há no `playerData`, como por exemplo a pontuação. Nesse objeto há os sprites dos textos e desenhos dos ícones que aparecem no canto da tela, enquanto que no `playerData` há apenas os dados dessas informações.

O cenário é o objeto `map`, que possui vários sprites e atributos, como posição e velocidade. Sua classe será explicada com mais detalhes na seção 5.1.

O objeto `player` e as tabelas `playerShots`, `enemies`, `enemiesShots` e `deadShips` correspondem respectivamente à nave do jogador, tabela com os tiros da nave do jogador, tabela contendo as naves inimigas, tabela com os tiros dos inimigos e a tabela com as naves eliminadas. Essa parte sobre as naves e os tiros será explicada nas seções 5.2 e 5.3.

No loop em que o jogo executa a cada quadro, há a chamada de funções que mantém o jogo rodando. Nessa parte o nome das funções são bem sugestivas para o que está acontecendo, como por exemplo as funções com `move` faz todos os objetos se moverem, incluindo também para o cenário. Em `resumeThreads()` é resumida as animações das naves e demais coisas que acontecem junto com o jogo, como explosão da nave e alterações na interface.

A função `coroutine.yield()` será melhor explicada nessa próxima seção 4.2.2, pois apesar de ser uma thread, para o MOAI temos que dizer o momento que ela deve ceder, permitindo outras threads continuarem.

4.2.2 Co-rotina

Em Lua as co-rotinas são uma poderosa ferramenta que funcionam como threads, mas são manipuladas de maneira diferente. A função da co-rotina roda pseudo-paralela com o programa, em que o programador pode facilmente resumir e continuar a co-rotina.

No jogo a co-rotina é útil para controlar animações dos objetos que surgem durante o jogo, como o efeito de disparo do tiro e explosão das naves. Veja abaixo um exemplo de uma função para co-rotina, localizado em "data/ship/ship.lua".

```
function Ship:spawnSize()
   self.sprite:moveScl(-1, -1, 0)
   local resizing = self.sprite:moveScl(1, 1, self.spawnDuration)

   while resizing:isActive() do
      coroutine.yield()
   end

   self.spawning = false
end
```

O método `Ship:spawnSize()` faz a animação de surgir com o tamanho aumentando em uma nave que acabou de ser criada no cenário. O laço fica checando a se a animação está ativa, usando a função `coroutine.yield()` para permitir o restante do jogo continuar rodando. Quando a animação acabar, a função continua até encontrar outro `coroutine.yield()` ou chegar ao final.

Para criar e manipular uma co-rotina é bem semelhante à thread. Sendo `ship` um objeto de uma nave genérica do jogo, o exemplo abaixo mostra o uso da co-rotina com o método `Ship:spawnSize()`.

```
spawn = coroutine.create(function()
```

```
   ship:spawnSize()
end)

coroutine.resume(spawn)
```

Cada vez que a função `coroutine.resume()` for chamada, a co-rotina que receber de parâmetro nessa função irá ser iniciar ou resumir de onde parou, ou seja, continuar a partir do último `coroutine.yield()` executado pela função.

4.3 Controles

Nessa versão do MOAI utilizada no projeto, é possível acessar as entradas padrões, que são o teclado e o mouse, mas com algumas limitações como as teclas de setas.

Para lidar com os controles do jogo, foi criado uma classe `Input` que usa funções callback do MOAI, essas funções são acionadas quando um determinado evento ocorre, que neste caso é quando uma tecla ou botão muda de estado (pressionado ou solto).

Abaixo está o código da classe `Input`, localizado em "data/input/input".

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

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

   I.up = false
   I.down = false
   I.left = false
   I.right = false

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

   return I
end

function Input:keyboardActive()
   MOAIInputMgr.device.keyboard:setCallback(onKeyboardEvent)
end

function Input:mouseActive()
   MOAIInputMgr.device.pointer:setCallback(onPointerEvent)
   MOAIInputMgr.device.mouseLeft:setCallback(onMouseLeftEvent)
end
```

Os atributos `up`, `down`, `left` e `right` são para as teclas de movimento da nave, quando têm o valor `true` indica que a tecla está sendo pressionada. O `pointerPressed` é para o botão do mouse e `pointerPos` guarda a posição atual do mouse na janela do jogo.

Os métodos dessa classe ativam a leitura da entrada do teclado e do mouse com as seguintes funções:

```
function onKeyboardEvent(key, down)
   if down == true then
      if key == 119 or key == 87 then input.up = true end
      if key == 115 or key == 83 then input.down = true end
      if key == 97 or key == 65 then input.left = true end
      if key == 100 or key == 68 then input.right = true end
   else
      if key == 119 or key == 87 then input.up = false end
      if key == 115 or key == 83 then input.down = false end
      if key == 97 or key == 65 then input.left = false end
      if key == 100 or key == 68 then input.right = false end
   end
end

function onPointerEvent(x, y)
   input.pointerPos.x = x
   input.pointerPos.y = y
end

function onMouseLeftEvent(down)
   if down == true then
      input.pointerPressed = true
   else
      input.pointerPressed = false
   end
end
```

Quando uma determinada tecla (ou botão) é apertada ou solta ocorre a chamada dessas funções, que atribuem ao objeto `input` a mudança de estado para a respectiva tecla.

Capítulo 5

Utilização da geometria analítica

No jogo há diversas classes que usam vetores, neste capítulo será mostrado as mais importantes. Essas classes usam as operações de vetores, retângulos e áreas que foram mostradas no capítulo 3.

5.1 Cenário

O fundo do jogo é um cenário que se movimenta constantemente. Ele possui alguns sprites que possuem velocidades diferentes e sempre que um sprite atinge um limite onde a imagem não mostra mais textura na janela, o sprite move para sua posição inicial naquele instante e seu movimento continua, tendo a impressão que o cenário nunca acaba e que as naves estão se movendo verticalmente.

Em "data/scenario" há os dois arquivos que trabalham com o cenário. No "scenario.lua" há a classe `Scenario` com atributos para um sprite e vetores que controlam a posição desse cenário, enquanto que em "stage1.lua" há a classe `Stage1` para o cenário todo, ou seja, vários atributos de objetos da classe `Scenario` que podem se mover com velocidades diferentes.

5.1.1 Criação do cenário

```
Scenario = {}
Scenario.__index = Scenario

function Scenario:new(deck, pos)
   local S =
   setmetatable(S, Scenario)

   S.sprite = MOAIProp2D.new()
   changePriority(S.sprite, "scenario")
   S.sprite:setDeck(deck)

   S.pos = pos
```

```
   S.spd = Vector:new(0, 0)

   S.startPos = Vector:new(pos.x, pos.y)

   S.size = Vector:new(screen.width / 2, screen.height / 2)
   S.limit = 2 * S.size.y

   S.sprite:setLoc(S.pos.x, S.pos.y)
   layer:insertProp(S.sprite)

   return S
end
```

O parâmetro `deck` da função acima é a imagem carregada como textura para ser usada em um sprite do jogo e `pos` é o vetor da posição inicial para aquele sprite.

A função `MOAIProp2D.new()` cria um novo sprite, ele será pintado na janela e por padrão do MOAI ficará acima de todos os sprites criados anteriormente no programa, exceto se for determinado uma prioridade para ele. A função `changePriority()` faz esse papel, mantendo para este caso o desenho do cenário sempre no fundo.

Para que o sprite mostre alguma imagem ele precisa de um deck, a função `setDeck()` estabelece a textura para o sprite.

Os atributos `pos`, `spd`, `startPos` e `size` são vetores que guardam respectivamente o valor da posição, velocidade, posição inicial e tamanho do sprite no cenário. Lembrando que a origem está no centro da janela, o tamanho que o desenho do cenário ocupa é metade da largura e altura da janela. o último atributo `limit` criado na classe guarda o valor máximo que o sprite irá se mover e daí voltar para sua posição inicial.

No final da função, o sprite criado é inserido na camada do jogo e será repintado cada vez que for necessário pelo MOAI.

Com essa classe que cria um cenário que possui um sprite, temos a outra classe `Stage1` que em seus atributos há objetos da classe `Scenario` que podem se mover com velocidades diferentes. Veja abaixo o código que contém o construtor da classe `Stage1`.

```
local littleStarsDeck = MOAIGfxQuad2D.new()
littleStarsDeck:setTexture("texture/scenario/littleStars.png")
littleStarsDeck:setRect(-screen.width / 2, -screen.height / 2,
                        screen.width / 2, screen.height / 2)

local bigStarsDeck = MOAIGfxQuad2D.new()
bigStarsDeck:setTexture("texture/scenario/bigStars.png")
bigStarsDeck:setRect(-screen.width / 2, -screen.height / 2,
                     screen.width / 2, screen.height / 2)
```

```
Stage1 = {}
Stage1.__index = Stage1

function Stage1:new()
   local S = {}
   setmetatable(S, Stage1)

   S.name = "Stage1"

   S.scenarios =

   local bigSpd = (-2) * screen.scale
   local smallSpd = bigSpd * 0.95

   local downSmaller = Scenario:new(littleStarsDeck, Vector:new(0, 0))
   downSmaller.spd.y = smallSpd
   table.insert(S.scenarios, downSmaller)

   local upSmaller = Scenario:new(littleStarsDeck,
                                  Vector:new(0, screen.height))
   upSmaller.spd.y = smallSpd
   table.insert(S.scenarios, upSmaller)

   local downBigger = Scenario:new(bigStarsDeck, Vector:new(0, 0))
   downBigger.spd.y = bigSpd
   table.insert(S.scenarios, downBigger)

   local upBigger = Scenario:new(bigStarsDeck, Vector:new(0, screen.height))
   upBigger.spd.y = bigSpd
   table.insert(S.scenarios, upBigger)

   return S
end
```

O atributo `scenarios` é uma tabela em que será inserido nela cada objeto da classe `Scenario` a ser criado, são eles `downSmaller`, `upSmaller`, `downBigger` e `upBigger`. Como os nomes já sugerem, eles são os cenários com as estrelas posicionadas acima ou abaixo, maiores ou menores. Observe que são usados dois decks diferentes, para as estrelas menores o deck `littleStarsDeck` e para as maiores o deck `bigStarsDeck`.

5.1.2 Movimentação do cenário

Para mover um sprite do cenário, o método `Scenario:move()` usa a soma de vetores para mudar sua posição de acordo com sua velocidade cada vez que essa função ser chamada.

```
function Scenario:move()
```

```
   if self.pos.y > self.startPos.y - self.limit then
      self.pos:sum(self.spd)
   else
      self.pos.y = self.startPos.y
   end

   self.sprite:setLoc(self.pos.x, self.pos.y)
end
```

Enquanto o movimento não chegar no limite, a posição do cenário será o resultado da soma entre os vetores posição e velocidade. Quando atingir seu limite, o sprite volta a sua posição inicial.

A função `setLoc()` é onde de fato o sprite altera sua posição visualmente no jogo, os atributos guardam esses valores da posição atual para que não sejam perdidos.

Para mover todos os cenários que há no objeto da classe `Stage1`, o método abaixo percorre a tabela dos cenários e chama a função de mover para cada um deles.

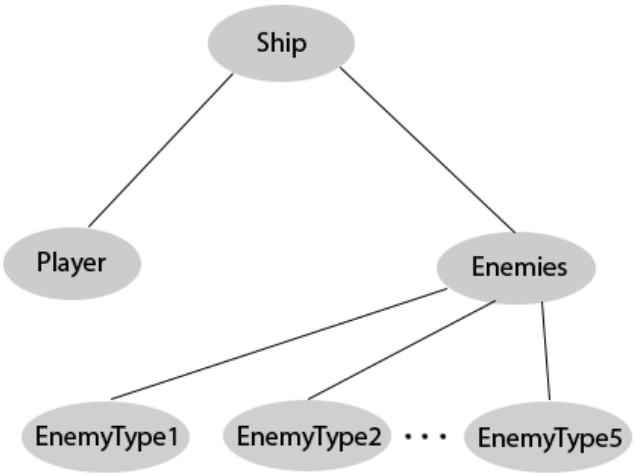
```
function Stage1:move()
   for i = 1, table.getn(self.scenarios), 1 do
      self.scenarios[i]:move()
   end
end
```

A operação `for` em Lua é inclusive para o início e fim do seu laço.

5.2 Naves

As naves do jogo possuem várias semelhanças entre si, mas cada tipo tem alguns atributos que diferem as classes. Para isso, há um sistema de heranças entre essas classes, em que vários métodos podem ser executados por qualquer nave, como por exemplo mover, destruir, e atirar.

A classe `Ship`, localizada em "data/ship/ship.lua", cria uma nave genérica, em que as classes que herdarem ela devem alterar seus atributos e em alguns casos criar novos atributos para o novo tipo de nave. Abaixo está uma figura representando a herança entre as naves.



`Ship` e `Enemies` são como classes abstratas, pois apenas auxiliam a criação e uso das outras classes de naves.

5.2.1 Classe Ship e movimentação da nave

Pelo fato de a classe ser muito grande, será comentado apenas os atributos e métodos que mais interessam na apostila. O código abaixo foi retirado do construtor da classe, nele há os atributos que todas as naves usam para movimentar e checar colisões.

```
S.pos = pos
S.spd = Vector:new(0, 0)
S.acc = Vector:new(0, 0)
S.maxAcc = 0.1 * screen.scale
S.dec = S.maxAcc / 3
S.maxSpd = 2 * screen.scale
S.minSpd = S.maxAcc / 5
S.area = Area:new(Vector:new(0, 0))
```

Os atributos `pos`, `spd`, `acc` são vetores que indicam a posição, velocidade e aceleração atuais da nave. Os limites para a velocidade e acelaração estão em `maxSpd` e `maxAcc`. Há também o atributo `minSpd` para velocidade mínima, pois após várias contas com a velocidade o valor dela pode ficar em um número "quebrado" próximo de zero, sendo necessário um mínimo de velocidade para o valor ser válido.

O atributo `area` é o tamanho da nave e os hitboxes que ela possui. As demais classes que herdarem, devem atribuir seus valores adequados nesse atributo. A área será útil para checar a colisão entre os tiros, que será melhor explicado na seção 5.3.

Com esses atributos sobre a velocidade e posição é possível fazer a movimentação da nave semelhante ao cenário, mas a diferença é que a velocidade não é constante e há uma aceleração

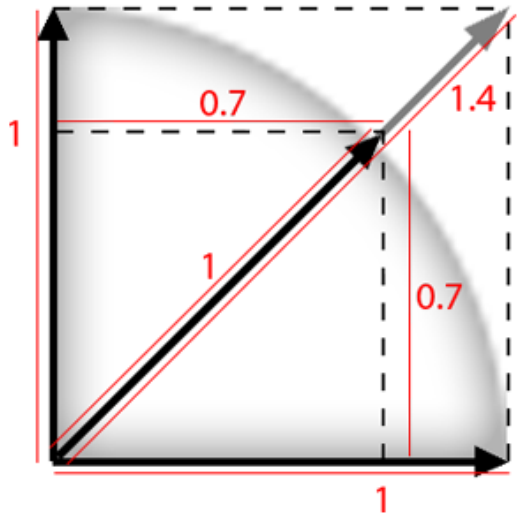
que pode variar a qualquer momento. O método `Ship:move()` realiza essa movimentação, veja o código abaixo:

```
function Ship:move()
   if self.acc.x == 0 and self.spd.x > 0 then self.acc.x = -self.dec end
   if self.acc.x == 0 and self.spd.x < 0 then self.acc.x = self.dec end
   if self.acc.y == 0 and self.spd.y > 0 then self.acc.y = -self.dec end
   if self.acc.y == 0 and self.spd.y < 0 then self.acc.y = self.dec end

   if self.acc:norm() > self.maxAcc then
      self.acc:normalize()
      self.acc.x = self.acc.x * self.maxAcc
      self.acc.y = self.acc.y * self.maxAcc
   end

   self.spd:sum(self.acc)

   if self.spd.x < self.minSpd and
   self.spd.x > -self.minSpd then self.spd.x = 0 end

   if self.spd.y < self.minSpd and
   self.spd.y > -self.minSpd then self.spd.y = 0 end

   if self.spd:norm() > self.maxSpd then
      self.spd:normalize()
      self.spd.x = self.spd.x * self.maxSpd
      self.spd.y = self.spd.y * self.maxSpd
   end

   self.pos:sum(self.spd)

   Ship.moveLimit(self)

   self.sprite:setLoc(self.pos.x, self.pos.y)
end
```

Apesar do jogo ser no espaço, as naves possuem uma desaceleração, assim o movimento fica mais fácil de ser controlado, principalmente para a nave do jogador. A primeira parte do método verifica se a aceleração é nula e a nave se move naquela direção, criando a desaceleração.

Para uma nave movendo em uma única direção (vertical ou horizontal) e com aceleração de 1 unidade, o resultado será mais 1 unidade na velocidade cada vez que a função ser chamada. Quando a nave mover em duas direções ao mesmo tempo, com aceleração de 1 unidade na vertical e 1 unidade na horizontal o resultado será uma velocidade na diagonal de $\sqrt{1^2 + 1^2} = \sqrt{2} \simeq 1.4$ unidade, fazendo com que a nave mova-se mais rapidamente.

Esse problema é causado quando a norma é maior que a aceleração máxima, para corrigir isso usamos a normalização do vetor. O resultado para o exemplo acima seria um vetor aceleração com 0.7 unidade. Veja a figura abaixo que representa o exemplo:



Com a aceleração calculada, podemos encontrar a nova velocidade atual da nave. Para isso usamos uma simples soma de vetores, verificando em seguida se a velocidade é quase nula, impedindo que a nave se mova lentamente quando deveria estar parada, pois pelo fato do método que cria o efeito de desaceleração, a soma dos vetores dificilmente pode resultar numa velocidade igual a zero.

Em seguida é feito a mesma verificação para o caso da velocidade ser maior que o limite, aplicando a normalização se necessário.

Agora com a velocidade calculada, podemos fazer a soma dos vetores e encontrar a posição final da nave, que é da mesma maneira feita para o cenário, utilizando uma soma dos vetores posição e velocidade. Mas antes de posicionar o sprite na sua posição, ainda há um outro problema que precisa ser tratado, a nave não pode sair da janela, pois a câmera é fixa. O método `Ship:moveLimit()` verifica se a posição da nave ultrapassa a borda direita ou esquerda da janela, ocorrendo uma translação para a borda oposta.

5.2.2 Classe Player

Nessa classe, localizada em "data/ship/player.lua", há a criação de uma nave específica para o jogador, que possui alguns métodos exclusivos que dependem dos controles do jogador para fazer a nave mover e atirar.

Assim como na classe `Ship`, será explicado apenas algumas partes do código da classe `Player`. Abaixo está o início do construtor de `Player`.

```
local P = {}
```

```
P = Ship:new(deck, pos)
setmetatable(P, Player)
```

Nessas 3 linhas, a classe `Player` cria um objeto que recebe herança da classe `Ship`, logo a tabela `P` possui todos os atributos criados de `Ship` e pode acessar os métodos dessa classe. Em seguida, no construtor de `Player`, ocorre todas as mudanças necessárias para adaptar um objeto de `Ship` para a nave do jogador.

Veja abaixo como fica a movimentação para a nave do jogador.

```
function Player:move()
   if self.spawned then
      if input.left == true then
         self.acc.x = -self.maxAcc
      else
         self.acc.x = 0.0
      end

      if input.right == true then self.acc.x = self.maxAcc end

      if input.up == true and
      not self.rot:isActive() and self.aim.y == -1 then
         self.rot = self.sprite:moveRot(180, 0.8)
         self.aim.y = 1
      end

      if input.down == true and
      not self.rot:isActive() and self.aim.y == 1 then
         self.rot = self.sprite:moveRot(180, 0.8)
         self.aim.y = -1
      end
   else
      self.acc.x = 0
      self.acc.y = 0
   end

   Ship.move(self)
end
```

O atributo `spawned` é uma variável que auxilia em alguns momentos do jogo, como por exemplo impedir que a nave receba dano enquando surge. No caso da movimentação, `spawned` é usado para impedir que o jogador mova a nave enquanto ela explode.

O objeto `input`, como já foi visto na seção 4.3, contém os controles do jogo. Quando uma tecla de movimentar ser pressionada a nave recebe aceleração na direção correspondente ao

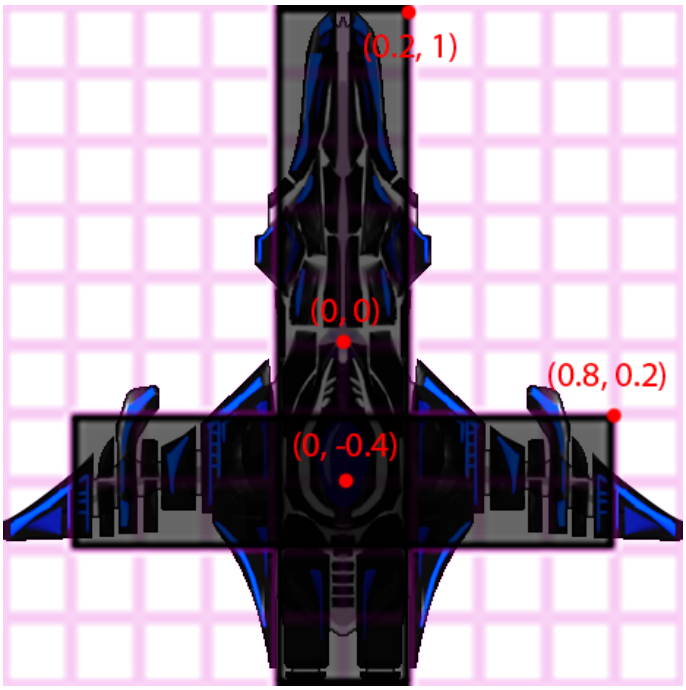
lado que jogador deseja move-lá. Os atributos `input.down` e `input.up` são para rotação da nave, verificando sempre se a nave já está girando ou apontando para o lado desejado.

No final, a nave tem a aceleração para aquele momento e é chamado o método de mover da classe pai, calculando sua velocidade e posição da nave no jogo.

As demais classes de naves têm o método de movimentar semelhante ao explicado acima, a diferença é que a aceleração será calculada a partir de uma inteligência artifical, e não dos controles do jogador.

Voltando ao construtor de `Player`, há uma parte importante para o jogo que a classe pai não completou, apenas inicializou, é o atributo `area`. Esse atributo, como já foi comentado, é um objeto que possui o tamanho e hitboxes da nave, que são dados importantes para detectar colisão entre as naves e tiros. Abaixo está o código que cria a área ocupada pela nave e uma figura que mostra o resultado dos hitboxes na nave.

```
P.area.size = Rectangle:new(Vector:new(0, 0), deckSize)
P.area:newRectangularArea(Vector:new(0, -0.4 * deckSize.y),
                          Vector:new(0.8 * deckSize.x, 0.2 * deckSize.y))
P.area:newRectangularArea(Vector:new(0, 0),
                          Vector:new(0.2 * deckSize.x, 1.0 * deckSize.y))
```



O vetor `deckSize` é o tamanho da imagem carregada para o sprite, esse vetor é usado como referência para criar os retângulos. Lembrando que o primeiro parâmetro dos métodos usados para criar os retângulos é o centro, no segundo parâmetro será o tamanho dele, ou seja, um vetor do centro ao canto do retângulo.

5.2.3 Classe Enemies

As naves inimigas no jogo apontam para o jogador e dependendo do lado em que surgirem, elas devem rotacionar. A classe `Enemies`, localizada em "data/ship/enemies.lua", faz esse papel de girar as naves inimigas.

```
Enemy = {}
Enemy.__index = Enemy

function Enemy:new(deck, pos)
   local E = {}
   E = Ship:new(deck, pos)
   setmetatable(E, Enemy)

   if E.pos.y > player.pos.y then
      E.sprite:moveRot(180, 0)
      E.aim.y = -1
   end

   return E
end
```

Se a posição da nave que está sendo criada estiver acima do jogador, ocorre uma rotação instântanea.

As classes do inimigos seguintes que herdam de `Enemies` irão criar uma nave dessa classe e não de `Ship`, como foi feito para o jogador. Durante o desenvolvimento do projeto elas não receberam nenhum nome e ficaram com nomes genéricos, porém em cada uma será explicado a qual se referem.

5.2.4 Classe EnemyType1

A classe `EnemyType1` corresponde a nave vermelha. Esse inimigo tenta ficar numa região próxima ao jogador e atira quando seu alvo estiver próximo. Abaixo está o método de mover da nave, localizado em "data/ship/EnemyType1.lua".

```
function EnemyType1:move()
```

```
   if player.spawned then
      if self.pos.x >= player.pos.x and self.acc.x >= 0 then
         if self.pos.x < player.pos.x + self.moveRange then
            self.acc.x = self.maxAcc
         else
            self.acc.x = -self.maxAcc
         end
      else
         if self.pos.x > player.pos.x - self.moveRange and
            self.acc.x <= 0 then
            self.acc.x = -self.maxAcc
         else
            self.acc.x = self.maxAcc
         end
      end
   else
      self.acc.x = 0
      self.acc.y = 0
   end

   Ship.move(self)
end
```

As condições nessa função para determinar a direção do movimento da nave são:

- se a nave estiver fora do seu limite de movimentação (atributo `moveRange`), a aceleração da nave fica em direção ao jogador;
- se a nave está dentro do seu limite de movimentação, ela faz um movimento de ida e volta dentro desse limite.

Para a nave atirar há um método que checa se a nave está próxima do alvo, o código na condição fica semelhante ao de mover dessa classe.

```
function EnemyType1:shoot()
   if player.spawned then
      local prob = math.random(1, 100)

      if prob <= 5 then
         if self.pos.x > player.pos.x - self.shotRange and
            self.pos.x < player.pos.x + self.shotRange then
            Ship.shoot(self, enemiesShots)
         end
      end
```

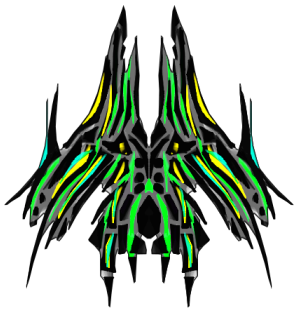
```
   end
end
```

A variável `prob` é um número sorteado para checar se a nave deve atirar, como esse número está entre 1 e 100, podemos considerar isso como a probabilidade da nave disparar naquele momento.

Há também uma condição de que a nave deve estar próxima ao jogador, assim como na movimentação, mas neste caso o alcance é menor.

Se essas condições permitirem a nave atirar, a chamada do método `Ship.shoot(self, enemiesShots)` faz a nave atirar, fazendo antes algumas verificações, como por exemplo checar se a nave já atirou recentemente.

5.2.5 Classe EnemyType2



A classe `EnemyType2` corresponde a nave verde, ela é a mais simples e fácil de ser eliminada. A nave faz um movimento de ida e volta entre as bordas da janela e atira em momentos aleatórios. Abaixo está o método de mover e atirar da nave, localizado em "data/ship/EnemyType2.lua".

```
function EnemyType2:move()
   if self.pos.x > self.moveRange or self.acc.x == 0 then
      self.acc.x = -self.maxAcc
   else
      if self.pos.x < -self.moveRange then
         self.acc.x = self.maxAcc
      end
   end

   Ship.move(self)
   end

function EnemyType2:shoot()
   local prob = math.random(1, 100)
```

```
   if prob <= 1 then
      Ship.shoot(self, enemiesShots)
   end
end
```

Essas funções de mover e atirar são bem parecidas com as funções da classe `EnemyType1`, elas fazem com que a nave desse tipo tenha uma dificuldade baixa para ser abatida e por isso elas são as primeiras a surgir durante o jogo.

5.2.6 Classe EnemyType3

A classe `EnemyType3` corresponde a nave azul claro, ela fica parada esperando a nave do jogador se aproximar do seu alcance, olhando não só a posição do alvo, mas também a velocidade. Abaixo está o método de mover da nave, localizado em "data/ship/EnemyType3.lua".

```
function EnemyType3:move()
   if player.spawned then
      if self.fireLast < gameTime - self.waitToMove then
         if self.pos.x < player.pos.x then
            self.acc.x = self.maxAcc
         else
            self.acc.x = -self.maxAcc
         end
      else
         self.acc.x = 0
      end
   else
      self.acc.x = 0
      self.acc.y = 0
   end

   Ship.move(self)
end
```

As condições nessa função verificam se a nave ficou esperando muito tempo para disparar, se isso ocorrer ela faz a nave mover em direção ao jogador, até que a nave atire novamente.

O método de atirar é o grande diferencial dessa nave em relação as outras. A nave do tipo `EnemyType3` somente atira quando souber que o seu tiro acertará o seu alvo, porém isso não faz com que o tiro dela seja impossível de esquivar, pois a nave desse tipo conhece apenas a posição e velocidade do jogador, mas não a aceleração, sendo possível alterar o movimento antes que o tiro atinja a nave.

```
function EnemyType3:shoot()
   if player.spawned then
      if player.spd.x == 0 and
         near(self.pos.x, player.pos.x, self.playerSize) then
         Ship.shoot(self, enemiesShots)
      else
         if player.spd.x  = 0 then
            local d1 = math.abs(self.pos.y) / self.shotSpd
            local d2 = (self.pos.x - player.pos.x) / player.spd.x

            if near(d1, d2, math.abs(self.shotRange / player.spd.x))then
               Ship.shoot(self, enemiesShots)
            end
         end
      end
   end
end
```

A primeira condição verifica se a nave inimiga está próxima ao jogador e a velocidade do alvo é nula, ou seja, se a nave do jogador está parada na sua frente, disparando caso isso ocorrer. A função `near()` verifica se 2 números (os primeiros passados por parâmetros) estão próximos, mas dentro de um certo limite (terceiro parâmetro).

Se as condições acima forem falsas, então o alvo está longe do alcance do tiro nave. Caso a velocidade do alvo ser nula, ou seja, a nave está parada, significa que o disparo não acertará o alvo, logo a nave não tenta atirar.

Nas linhas seguintes a nave assume que o seu alvo está fora do alcance e em movimento, calculando se o disparo vai acertar supondo uma velocidade constante para o alvo. As variáveis `d1` e `d2` correspondem a uma medida do tempo que levará para, respectivamente, o tiro atingir o local que a nave está mirando e o jogador movimentar-se para a posição final do movimento. Se essas medidas calculadas estiverem próximas e dentro do limite esperado, a nave irá disparar, acertando o jogador caso sua nave não alterar a velocidade.

5.2.7 Classe EnemyType4

A classe `EnemyType4` corresponde a nave rosa, ela se movimenta sempre em direção ao jogador, semelhante a nave da classe `EnemyType1` e atirando sempre que seu alvo estiver na mira. O detalhe nessa nave é que ela possui uma habilidade de trocar de lado no campo.

O método abaixo, localizado em "data/ship/EnemyType4.lua", mostra a movimentação da nave.

```
function EnemyType4:move()
   if player.spawned then
      if not near(self.pos.x, player.pos.x, self.closeToTarget) then
         if self.pos.x < player.pos.x then
            self.acc.x = self.maxAcc
         else
            self.acc.x = -self.maxAcc
         end
      else
         self.acc.x = 0
      end
   else
      self.acc.x = 0
      self.acc.y = 0
   end

   self:checkBlink()

   Ship.move(self)
end
```

O código é bem parecido em relação as classes anteriores, mas a diferença é na chamada do método `self:checkBlink()` no final. Essa função é verifica se a nave pode usar a habilidade especial dela, para ativar deve haver algum tiro do jogador próximo a ela.

Há um tempo de recarga para que a habilidade possa ser usada novamente e ela não será ativada se o tiro passar ao lado da nave. Veja abaixo os métodos que fazem essas verificações:

```
function EnemyType4:checkBlink()
   if self.blinkLast <= gameTime - self.blinkCd and not self.blinkActive and not
self.spawning then
      if self:shotClose() then
         local blinkThread = coroutine.create(function()
            self.spawned = false
            self:blink()
            self.spawned = true
         end)
         coroutine.resume(blinkThread)
         table.insert(self.threads, blinkThread)
      end
   end
end

function EnemyType4:shotClose()
   for i = 1, table.getn(playerShots), 1 do
      if near(self.pos.y, playerShots[i].pos.y, self.blinkShotClose) then
         if near(self.pos.x, playerShots[i].pos.x, self.shotRange) then
            return true
         end
      end
   end

   return false
end
```

Em `EnemyType4:shotClose()` é percorrido a tabela dos tiros do jogador e verificado se algum deles está na região em que a habilidade deve ser ativada.

Logo esse tipo de nave pode ser eliminada de duas maneiras: atirando em suas laterais ou atirando quando sua habilidade está recarregando (isso inclui quando ela surge).

5.2.8 Classe EnemyType5

A classe `EnemyType5` corresponde a nave vermelha que se movimenta com uma trajetória em curvas, lembrando uma função de seno. Apesar de ser uma nave rápida e difícil de acertar, ela não possui nenhhum método específico para atirar, assim ela dispara em intervalos constantes. Sua classe está em "data/ship/EnemyType5.lua" e o código abaixo mostra a movimentação desse tipo de nave.

```
function EnemyType5:move()
   if self.pos.x > self.moveRange.x and self.acc.x >= 0 then
      self.acc.x = -self.maxAcc
   else
      if self.pos.x < -self.moveRange.x and self.acc.x <= 0 then
         self.acc.x = self.maxAcc
      end
   end

   if self.pos.y > self.startPos.y + self.moveRange.y and
      self.acc.y >= 0 then
      self.acc.y = -self.maxAcc
   else
      if self.pos.y < self.startPos.y - self.moveRange.y and
         self.acc.y <= 0 then
         self.acc.y = self.maxAcc
      end
   end

   Ship.move(self)
end
```

O seu movimento de ida e volta na horizontal é igual ao da nave verde (classe `EnemyType2`) e na vertical há uma verificação semelhante ao da horizontal, a diferença é que o limite para se mover na vertical é menor. O resultado é um movimento em curvas.

A normalização do vetor é importante para essa nave, pois ela está frequentemente com velocidade nas duas dimensões, podendo resultar no problema comentado na seção 5.2.1, em que a nave recebe uma velocidade maior do que deveria. Tratando com a normalização esse problema faz com que a nave não tenha esses movimentos mais rápidos em diagonais.

5.3 Classe Shot e colisão entre tiros e naves

Os tiros se movem igualmente ao cenário, com velocidade constante e sempre na mesma direção. Abaixo está o código, localizado em "data/shot/shot.lua", de uma função que realiza a movimentação dos tiros.

```
function shotsMove()
   local s = 1
   while s <= table.getn(playerShots) do
```

```
      playerShots[s]:move()
      if not shotCheckDistant(playerShots, s) then
         s = s + 1
      end
   end

   s = 1
   while s <= table.getn(enemiesShots) do
      enemiesShots[s]:move()
      if not shotCheckDistant(enemiesShots, s) then
         s = s + 1
      end
   end

   if not player.spawning and player.spawned then
      shotsCheckCollision(enemiesShots, player)
   end

   for i = 1, table.getn(enemies), 1 do
      if not enemies[i].spawning and enemies[i].spawned then
         shotsCheckCollision(playerShots, enemies[i])
      end
   end
end
```

Nessa função há outras duas funções que são chamadas dentro dela que são importantes, que são `shotCheckDistant()` e `shotsCheckCollision()`.

Ao percorrer as tabelas que dos tiros das naves, é chamado a função `shotCheckDistant()` que verifica se os tiros que não atingiram nenhum alvo estão longe da parte visualizável do jogo, onde não há mais inimigos.

Para evitar uso de memória e processamento desnecessário, esses tiros distantes são removidos por essa função.

```
function shotCheckDistant(shots, s)
   if shots[s].pos.x > screen.width or
      shots[s].pos.x < -screen.width or
      shots[s].pos.y > screen.height or
      shots[s].pos.y < -screen.height then

      Shot.destroy(shots[s])
      table.remove(shots, s)

      return true
```

```
   end

   return false
end
```

O limite onde os tiros ainda existem é definido pela largura e altura da janela, mas como a origem está no centro, há uma região não visível no jogo de metade da tela onde os tiros ainda existem, para evitar que os tiros sejam apagados enquanto ainda podem ser vistos.

Na função `shotsCheckCollision()` é o momento que verifica se um tiro colidiu com uma nave.

```
function shotsCheckCollision(shots, ship)
   local s = 1
   local hit = false

   while s <= table.getn(shots) and not hit do
      local shot = shots[s]

      if shot.area:detectCollision(Vector:new(0, 1),
         shot.pos, ship.area, ship.aim, ship.pos) then
      hit = true
         ship:damaged(shot.dmg)

         Shot.destroy(shot)
         table.remove(shots, s)
      else
         s = s + 1
      end
   end
end
```

Lembrando o que foi comentado na seção 3.3 sobre áreas, um objeto dessa classe possui o método `Area:detectCollision(orientationA, posA, b, orientationB, posB)`, em que os parâmetros são as duas áreas que buscamos a intersecção, suas posições e orientações (para o caso da nave estar posicionada no outro lado do campo).

A função percorre a tabela dos tiros e verifica para cada um deles se algum alvo foi atingido.

No atributo `area` do tiro, a área é o próprio tamanho da textura, pois pelo fato de ser pequeno e seu formato próximo de um retângulo, não é necessário vários retângulos que representem o seu desenho.

Caso a detecção indicar que houve colisão, a nave que estava sendo analisada recebe o dano causado (se não possuir vida suficiente ela é destruída) e o tiro é removido do jogo.

Capítulo 6

Fonte das texturas

As imagens encontradas no jogo são de fontes livres. Abaixo estão os locais de onde foram retiradas e a data de acesso:

Sprite das naves `http://millionthvector.blogspot.com.br/` em 04/2014.

Sprite de estrelas `http://effextures.com/bright-star/` em 03/2014.

Sheet de explosão `http://percsich.hu/tgmforum/index.php?topic=446.0` em 04/2014.

Imagens de fundo `http://www.hdwallpapers.in/` em 03/2014.

Fontes de texto `http://www.ajpaglia.com/` e `http://typodermicfonts.com/` em 05/2014.