Instituto Politécnico de Lisboa

Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e Multimédia (LEIM)

Unidade Curricular: Modelação e Simulação de Sistemas Naturais (MSSN)

Projeto Final – Space War

33D

Nome	Curso	Número
Sandra Wang	LEIM	46319
José Siopa	LEIM	46338

Docente: Arnaldo Abrantes

Lisboa

19 de Janeiro de 2020

Índice

Índice de figuras

Índice de listagens

Introdução

Este trabalho tem como objetivo o desenvolvimento do jogo Space War. Para tal, foi preciso ter uma noção acerca da simulação do movimento de objetos usando as leis da física e acerca de agentes autónomos.

Um agente autómato é uma entidade que faz as suas próprias escolhas de como atuar sobre o seu ambiente, sem a influência de um plano global ou de um líder.

Numa breve explicação, este jogo consiste na existência duma nave, controlada pelo utilizador (através do teclado e do rato), que tem de fugir e/ou atacar meteoros e outros *UFOs*. Os meteoros navegam de forma aleatória (tem um comportamento de *wander*) enquanto que os outros *UFOs* perseguem a nave do jogador (comportamento de *pursuit*).

Descrição do sistema

Tal como foi referido anteriormente, este sistema é constituído por uma nave principal (a do jogador), por um número ilimitado de meteoros e outros UFO que vão aparecendo e por lasers disparados pela nave.

Para tal, foram implementados quatro *Boids* específicos: `LaserBoid`, `MeteorBoid`, `PlayerBoid` e `UfoBoid`. Estas quatro classes estendem a classe abstrata *Boid*, que por sua vez estende a classe `Mover`.

A classe `LaserBoid` representa o laser lançado da nave do jogador.

A classe `MeteorBoid` representa os meteoros que vão aparecendo ao longo do jogo, a seleção dos meteoros que aparecem é aleatória, sendo que a probabilidade de aparecerem meteoros maiores é menor por causa da condição apresentada na linha 4 da Listagem 1 (as imagens com nome *meteorBrown1*, *meteorBrown2*, *meteorBrown3*, *meteorBrown4* simbolizam os meteoros de maior porte). Foi optado ter vários tamanhos de meteoros para adicionar alguma dificuldade ao jogo.

Listagem 1 – Probabilidade de aparecimento de meteoros maiores

```
1.  public void setShape(PApplet p, SubPlot plt) {
2.      float randPercentage = (float) Math.random();
3.      int randImg;
4.      if (randPercentage < 0.6)
5.          randImg = (int) super.random(1, 5);
6.      else
7.          randImg = (int) super.random(5, 8);
8.
9.      PImage img = p.loadImage("TPFinal/Meteor/meteorBrown" + randImg + ".png");

10.     super.img = img;
11. }
```

Nesta classe é definida também a vida do meteoro, tal como o dano que ele causa à nave do jogador.

A classe `PlayerBoid` representa a nave do jogador, que tal como referido anteriormente, que pode ser movimentada com as teclas ASDW, a sua direção pode ser ajustada através do rato e é disparado um laser quando se clica no botão do rato. Esta nave é inicializada com uma vida de 100 e vai perdendo vida à medida que colide com

meteoros ou com outros *UFO*. Para além disso, é feita e apresentada no canto inferior da janela, o número de *UFO* derrotados.

A classe `UfoBoid` representa o resto dos *UFO* que perseguem a nave do jogador. Este começa com uma vida de 10 e dá 7 de dano. Em semelhança com o `MeteorBoid`, a geração destes *UFO* também é aleatória.

Main menu

O menu principal (ver Fig.2) do jogo é consistido pelo título do jogo e por dois botões *Play* e *Quit* (que por sua vez são imagens). Estes botões são criados nas classes `PlayButton` e `QuitButton` que estendem a classe abstrata `GeneralButton`, que contém os métodos para atualizar e dispor os botões.

O botão *Play* permite inicializar o jogo e o *Quit* permite sair do jogo.

É na classe `MainMenu` que são desenhados o background do jogo, é feito o display do texto dos botões, são atualizados os botões e é apresentada a última pontuação do jogador.

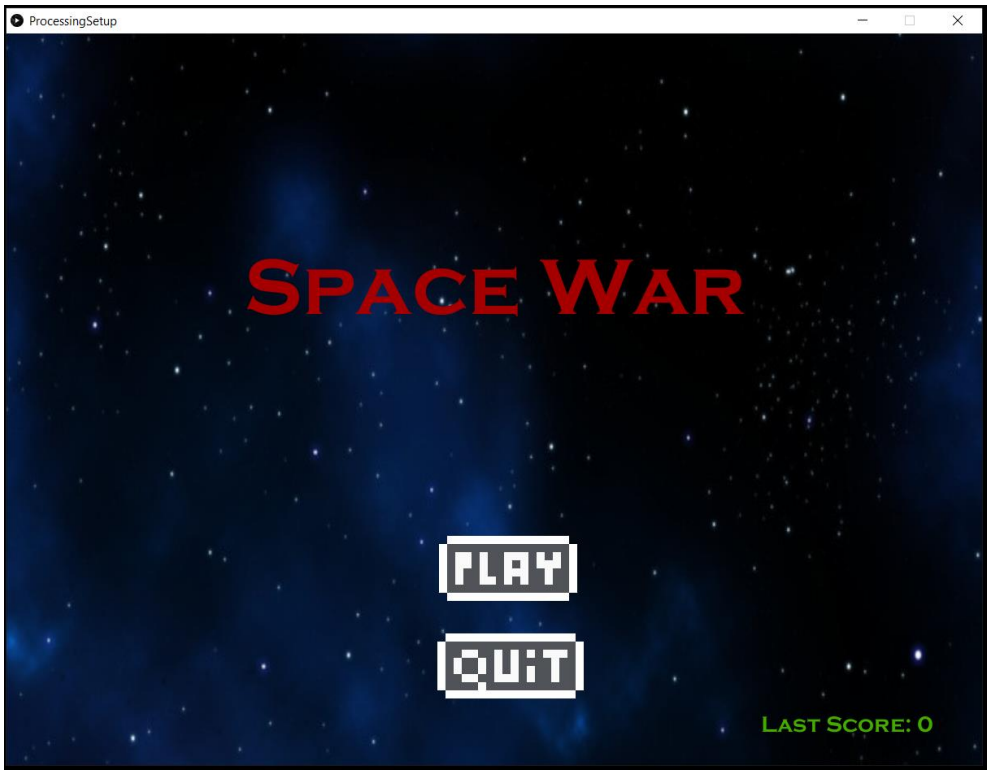

Figura 1 - Main menu do jogo

Game

A classe `Game` é aquela em que o jogo todo trabalha. Começa por inicializar a nave do jogador na posição central do ecrã (ver Fig.3) , inicializa todos os `ArrayLists` de objetos que irão aparecer durante o jogo e coloca a dificuldade a 1.

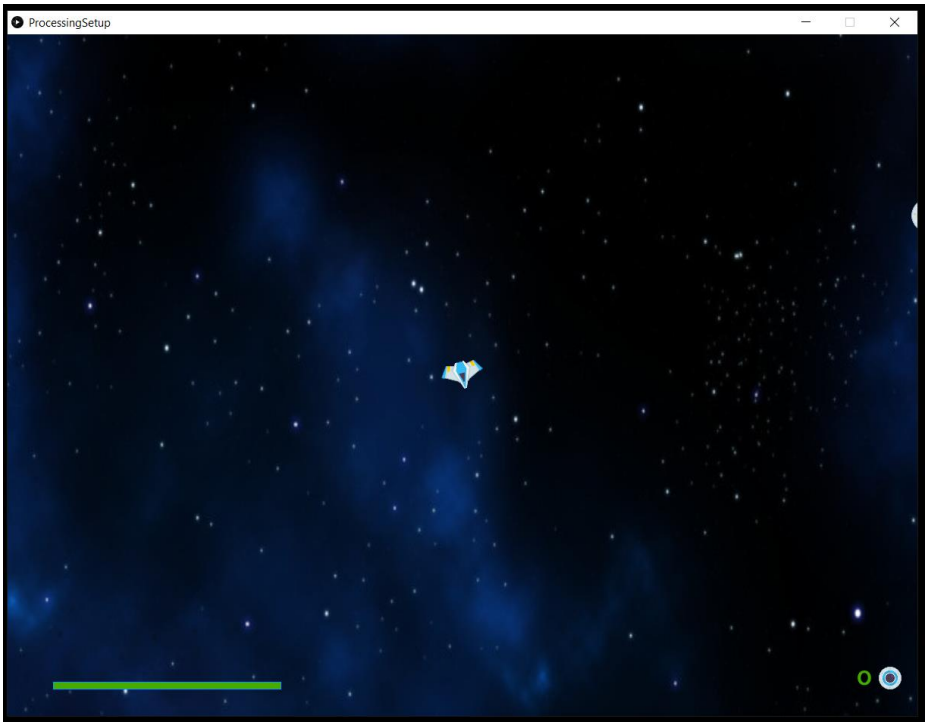

Figura 2 - Frame inicial do jogo

De seguida, no método `draw`, que se executa múltiplas vezes por segundo, é verificado se cada um dos lasers que o jogador dispara colide com outro objeto, utilizando o método `checkCollisions` (explicado mais à frente) que irá tratar desencadear uma ação nesse objeto caso colida. Como os projéteis não podem fisicamente atravessar um objeto sólido de grandes dimensões e massa (como rochas), o laser que colidir com algo desaparece.

Percorrem-se agora os `ArrayLists` dos objetos inicializados no construtor (lista de *UFO*s e lista de meteoros). Na primeira, são desenhados todos os *UFO*s da lista, aplica-se uma força a cada um deles, tendo essa força como ponto de chegada a nave do jogador, fazendo assim com que todos os *UFO*s persigam o jogador independentemente da sua posição e verifica-se se estes colidiram. Caso afirmativo, acrescenta-se um à pontuação.

Na lista de meteoros, a cada meteoro vai ser aplicada uma força em que ponto de chegada vai ser o simétrico das coordenadas onde aparece, dando a ilusão que este está a viajar no espaço. São também verificadas as colisões de cada um e se colidir com o jogador, é removido da lista.

Para finalizar este método, são chamados os métodos de display da nave de jogador, da sua barra de vida e dos pontos, além do método que vai executar ações conforme o passar do tempo, timerEventTrigger.

Os métodos que se seguem verificam quando o jogador clica no rato, que irá chamar o método do playerBoid, shoot, que irá tratar de disparar um laser na direção em que o rato está posicionado (ver Fig.3) e os métodos que verificam quais as teclas que o jogador premiu, movimentando a nave conforme a tecla pressionada.

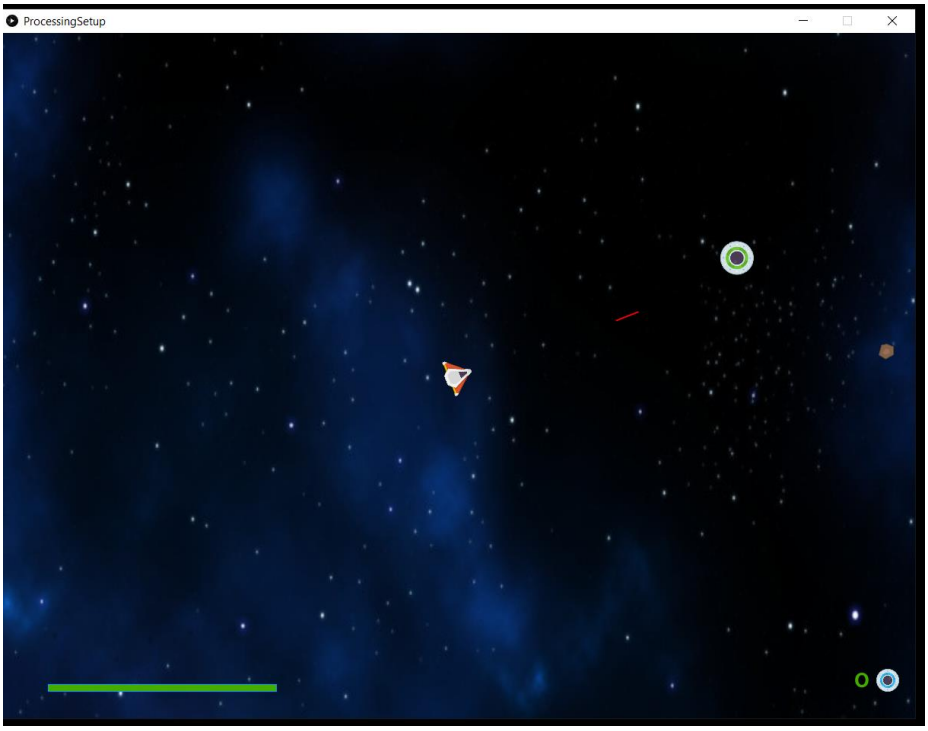

Figura 3 - Nave a disparar laser

Segue-se então o método timerEventTrigger, que irá adicionar à lista de UFOs um novo UFO, inicialmente aos 3 segundos de jogo. Valor esse será subtraido com a dificuldade do jogo, que aumenta 200 milisegundos de 10 em 10 segundos (por exemplo, aos 50 segundos de jogo, o próximo UFO irá aparecer 2 segundos depois). São também adicionados meteoros a cada 1,5 segundos.

Prosseguindo, tem-se o método checkCollisions (ver Listagem 2) que é executado por todos os elementos das listas de lasers, de UFOs e de meteoros. Na primeira comparação, verifica se o objeto em questão não é um laser e chama a função collided desse objeto, irá executar o código presente e retorna verdade se colidiu com a nave do jogador. Se a condição se verificar, adiciona um sistema de partículas conforme o objeto.

Listagem 2 – Método checkCollisions

```
1.  private void checkCollisions(Boid b) {
2.      if (!b.getClass().equals(TPFinal.GameEntities.LaserBoid.class) && b.collid
    ed(playerBoid)) {
3.          gameParticles.add(b.createNewParticles());
4.      } else if (b.getClass().equals(TPFinal.GameEntities.LaserBoid.class)) {
5.          for (UfoBoid ufo : ufos) {
6.              if (b.collided(ufo)) {
7.                  ufo.setHealth(ufo.getHealth() - b.getAttackDamage());
8.              }
9.          }
10.         if (ufos.removeIf(ufo -> ufo.getHealth() <= 0))
11.             ufosDestroyed++;
12.
13.         for (MeteorBoid meteor : meteors) {
14.             if (b.collided(meteor)) {
15.                 meteor.setHealth(meteor.getHealth() - b.getAttackDamage());
16.             }
17.         }
18.         meteors.removeIf(meteor -> meteor.getHealth() <= 0);
19.
20.         gameParticles.add(b.createNewParticles());
21.     }
22. }
```

Se o objeto pertencer à classe do laser, vão se percorrer todos os UFOs e todos os meteoros, verificar se o laser colidiu com algum deles. Se sim, aplica-se o dano do laser à vida do objeto que este colidiu, removendo esse objeto caso a sua vida seja igual ou inferior a 0. É acrescentado 1 ponto à pontuação caso seja um UFO a ser eliminado.

Para finalizar, tem-se as funções gameEnded, que retorna verdade quando a vida do jogador é igual ou menor que 0 e getScore, que retorna a pontuação do jogador.

Boid

É utilizada a classe `Boid` dada em aula para fazer a nave do jogador, os lasers, os meteoros e os *UFOs*, mas fizemos algumas alterações a ela. Começamos por tornar esta classe abstrata porque seria mais apropriado à nossa metodologia. Apesar de estarem dentro da classe diversos métodos, apenas foram utilizados o `applyForce`, `seek` e `pursuit`, O primeiro aplica ao *boid* uma força dada como argumento, a segunda o *boid* dirige-se para um dado ponto de coordenadas e o terceiro o *boid* persegue outro *boid* dado como argumento.

Foram acrescentadas as funções `genRandPosOutside` que retorna um `PVector` de coordenadas fora dos limites do campo de batalha, para criar a ilusão que os meteoros e os UFOs estão a viajar pelo espaço.

Segue-se a função `collided` que irá verificar se o *boid* que chamou esta função colidiu com o *boid* dado como argumento. Para tal, utilizam-se cálculos (ver Listagem 3) para saber se estes se sobrepõem. Caso isso aconteça, será retirada à vida do *boid* dado como argumento o dano de ataque do *boid* atual.

Listagem 3 – Método collided

```
1.  public boolean collided(Boid b) {
2.      float distSq = (pos.x - b.pos.x) * (pos.x - b.pos.x) + (pos.y - b.pos.y) *
     (pos.y - b.pos.y);
3.      float radSumSq = (radius + b.radius) * (radius + b.radius);
4.      if (distSq <= radSumSq) {
5.          b.setHealth(b.getHealth() - this.getAttackDamage());
6.          return true;
7.      }
8.      return false;
9.  }
```

A função `createNewParticles` irá criar na posição em que está o *boid* atual um sistema de partículas, dependendo da classe do *boid*.

Para finalizar esta classe, o método `setHealth` e `setAttackDamage` permite alterar o valor da vida e do dano de ataque do *boid* e as funções `getHealth` e `getAttackDamage` retornam o valor da vida e do dano de ataque do mesmo.

Space shooting

É na classe `SpaceShooting` que se insere a música de fundo do jogo e é nesta que se desenha o menu principal e o jogo em si, através de `switch case`.

Inicialmente é criado um enumerado `gameStates` com dois estados, `MainMenu` e Game. De seguida é criada uma variável `gameState` do tipo `gameStates`, que é utilizada nos `switch case` presentes nos outros métodos.

No método `draw` se o `gameState` for `MainMenu` então é desenhado o Main Menu, caso seja `Game` então é desenhado o jogo em si e se verifique que o jogo acabou `(game.gameEnded())` então é apresentada no Main Menu a última pontuação do jogador e após um delay de 2000ms o jogo é inicializado outra vez, voltando para o Main Menu.

No método `mousePressed(PApplet p),` se o `gameState` for `MainMenu` e caso o botão "play" for pressionado então o estado passa a `Game`. Neste estado, caso o botão do rato for pressionado, a nave do jogador dispara um laser.

Diagrama UML das relações entre as classes

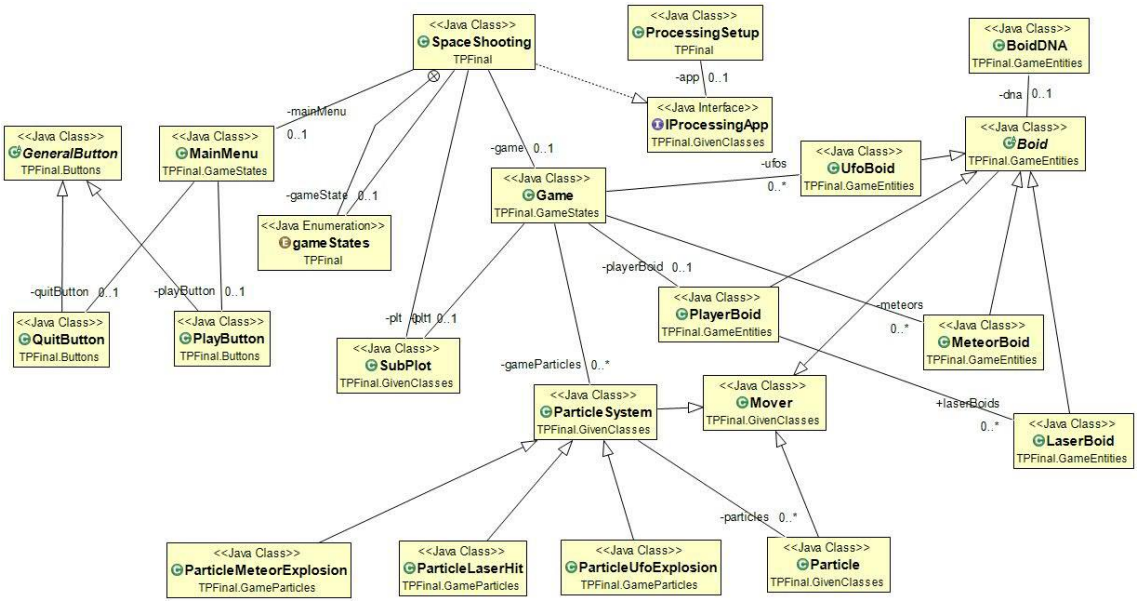

Figura 4 - Diagrama UML das relações entre as classes

O diagrama UML acima permite entender melhor e de uma forma mais prática as relações entre todas as classes utilizadas no desenvolvimento deste trabalho.

Conclusões

Tendo em conta o objetivo do jogo e os resultados obtidos, pode-se concluir que o jogo foi bem executado, e os objetivos atingidos. No entanto, poderiam ter sido implementados mais métodos para melhorar os efeitos gráficos, nomeadamente as colisões entre a nave e os meteoros/*Ufos.*