
ADDETC – Área Departamental de Engenharia Eletrónica e Telecomunicações e de Computadores

LEIM -Licenciatura Engenharia informática e multimédia

# Modelação e Simulação de Sistemas Naturais

# Relatório Trabalho Final

**Turma:**

LEIM-51N

**Trabalho realizado por:**

Vasco Cruz Nº42837

Pedro Henriques Nº45415

**Docentes:**

Arnaldo Abrantes

Paulo Vieira

**Data:**24/02/2021

# Índice

# Introdução

Este relatório pretende explicar o processo de desenvolvimento do trabalho final da disciplina de Modelação e Simulação de Sistemas Naturais (MSSN).

O trabalho final de MSSN, consistiu na criação e desenvolvimento de um jogo que aplica os conceitos lecionados em aula tais como modelação de agentes (boids), forças, sistemas de partículas e sistemas complexos.

O jogo é um clone, implementado em processing do antigo jogo Space Invaders, com algumas alterações que ajudam a refletir a matéria aprendida.

Devido á natureza do jogo, não foi possível implementar explicitamente certas partes da matéria, no entanto estas poderão ser mencionadas neste relatório por terem sido utilizados como fundamento teórico de outras partes do trabalho.

## Objetivos

De um ponto de vista mais técnico, este jogo surge no culminar da unidade curricular de MSSN e como tal irá abordar temas tais como:

-Modelação baseada em agentes (inimigos e jogador serão agentes)

-Forças (aplicadas aos agentes, levam a movimento)

-Sistemas complexos. (Interação entre os agentes)

Desta forma iremos usar código elaborado por nós, classes fornecidas ao longo do semestre pelos docentes e outras partes pesquisadas online para resolver bugs pontuais que foram surgindo.

## O Jogo

Como mencionado em cima o jogo é um clone de Space Invaders. Neste jogo um “exército” de aliens ataca a Terra e um Super soldado foi criado para combater a invasão.

O nosso soldado, movimenta-se no solo e vai apanhando e disparando os ossos dos seus camaradas que morreram antes dele contra os inimigos que cada vez mais se aproximam dele e colocam em risco a sua vida e a de todos os habitantes nesse planeta.

O jogo acaba, quando o soldado vença o inimigo final (depois de destruir todos os minions), ou com a morte do jogador e consequente destruição do seu planeta.

O jogo tem de possuir forma de:

- Verificar de vitoria/derrota

- Atribuir pontos ao jogador

- Movimentar e aplicar forças a objetos e entidades

- Mostrar menus que correspondem ao estado do jogo (menu inicial, menu jogo, pausa, etc…).

- O jogador atacar

- Morrer (tanto para o jogador como os inimigos) e verificação visual do ato.

- Animações para mover os objetos para originar

- Música de fundo

# Introdução teórica

## Agentes ou Individuo

Um agente, no contexto do nosso trabalho é um elemento capaz de interagir com o ambiente e com outros agentes, com autonomia podendo ou não efetuar decisões com alguma inteligência.

No nosso trabalho, tanto o jogador como o *boss* final são agentes que tomam decisões baseados no seu ambiente. O jogador é um humano por isso a sua inteligência depende somente do utilizador. O *boss* é um agente "semi-inteligente" que tenta destruir o jogador acertando-lhe com bombas, ajustando a sua posição para lançar as mesmas em cima do jogador.

## Forças

Num ambiente não computacional os elementos movem-se seguindo leis físicas bem definidas. Nomeadamente, o movimento ocorre após a aplicação de uma ou mais forças. Tendo em conta a segunda lei de Newton em que $F = m * a$, é correto dizer que $a = F/m$. Ou seja, aplicando uma força F a um corpo de massa m o corpo irá ter uma aceleração a. Assim através das equações do movimento $v(t) = v_0 + a * t$ e $x(t) = x_0 + v_0 t + \frac{1}{2} a^2 * t$, podemos parametrizar o movimento do corpo segundo qualquer um dos eixos, após termos fornecido uma força, F.

Ou seja, se for necessário calcular qual a posição do corpo x, no instante t, é necessário resolver a equação diferencial em que $x'(t) = x(t-1) + v$ e $v'(t) = v(t-1) + a$.

No entanto o nosso programa, não consegue ser executado em todos os instantes pelo que estas equações de movimento não são as mais adequadas. É necessário proceder a uma aproximação das posições, através do Método de Euler

## Método de Euler

O método de Euler é um processo através do qual, aproximamos uma equação diferencial através de iterações sucessivas seguindo a equação.

$$x(n) = x(n-1) + \Delta t * x(n-1)$$

Adaptando ao nosso contexto a posição x(n) é dada por

$$x(n) = x(n-1) + \Delta t * \left( v_0 + \frac{1}{2} * \frac{F}{m}^2 \right)$$

Em que $\Delta t$ é o intervalo entre frames, $v_0$ é a velocidade inicial do corpo e a aceleração corresponde á força F, a dividir pela massa do corpo. Desta forma obtemos a melhor aproximação possível para o movimento dos nossos objetos e mantemos uma relação de proximidade com a realidade.

## Sistemas de Partículas

Um sistema de partículas é um objeto especial composto por outros objetos muito pequenos (partículas). O objetivo deste sistema é a simulação de comportamentos de corpos que se estejam a desfazer ou cujo movimento das suas muitas partes ofereça um custo computacionalmente elevado e como tal não seja eficiente a representação partícula a partícula.

# Desenvolvimento

## Diagrama de classes

Antes de começar a implementar o trabalho foi realizada um diagrama de classes que iria reger o processo de desenvolvimento esta classe não foi seguida a 100% pelo que existe objetos presentes no código final que não estão aqui representados, nomeadamente a classe abstrata entidade que foi criada por conveniência.

O diagrama representa por alto a maneira como o código será estruturado e quais as classes que irão ser utilizadas na sua realização. No entanto no desenrolar do desenvolvimento foi necessário a utilização de outras classes e inclusive eliminação de algumas.

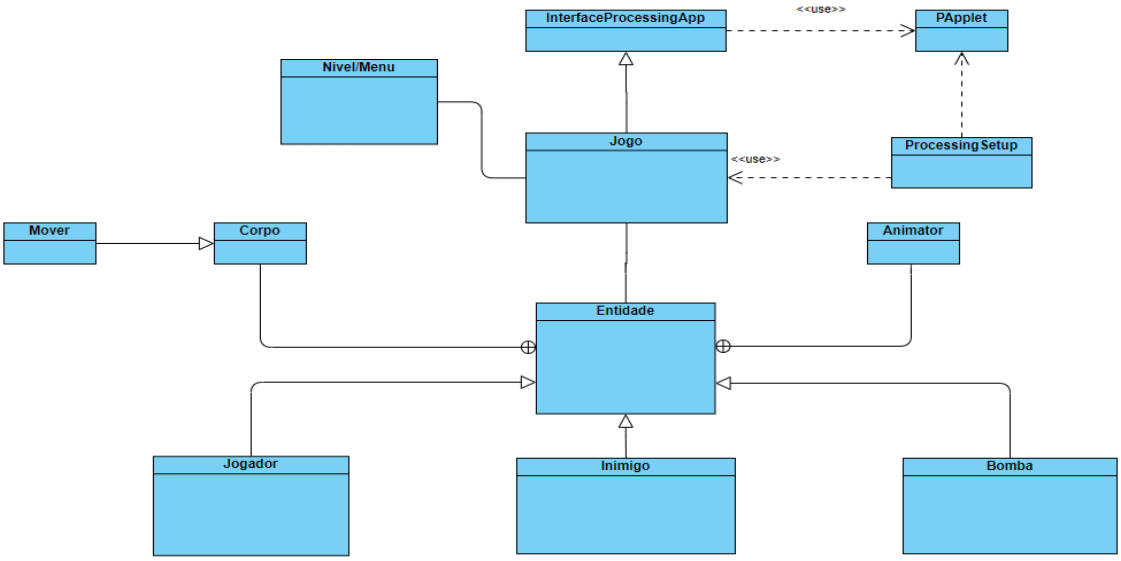



*Figura 1 - Diagrama de classes inicial*

A classe Processing Setup, tal como elaborada na aula será o ponto de partida de todo o programa. No entanto o Jogo será a parte fundamental deste.

## Classe Jogo

A classe Jogo foi concebida com o intuito de possuir toda a parte logica deste, e delegar às classes que implementem um nível (ou menu), e às entidades, o desenho na própria GUI dos seus elementos. Desta forma obtemos um código altamente modular, pouco propenso a erros e caso existam alguns erros, fáceis de corrigir e identificar.

Esta classe Jogo ficaria também responsável por toda a parte logica do mesmo, tais como instanciar inimigos, processar eventos do jogador, e as condições de vitoria/derrota.

A classe Jogo foi implementada de maneira sequencial tendo em conta as necessidades desta e utilizando sempre o melhor possível as suas classes derivadas. Esta classe possui um método draw onde os métodos draw das implementações de Nivel e Entidade são chamados, sempre nesta ordem de modo a sobrepor os agentes ao fundo, (Nivel).

Este método draw, foi nos muito conveniente pois é chamado pelo Processing, com o executar de cada frame, desta forma tínhamos um ponto no código que sabíamos sempre que seria executado pelo que a logica de jogo foi também implementada dentro deste.

## Fluxo da aplicação

A aplicação possui um conjunto de menus e eventos que fornecem alguma logica ao jogo, sendo que cada evento possui uma ação que lhe corresponde e finalmente o sistema avança para o próximo estado. De notar que no contexto da aplicação não existe nenhum fim proposto, no entanto é possível sair do sistema fechando a aplicação.

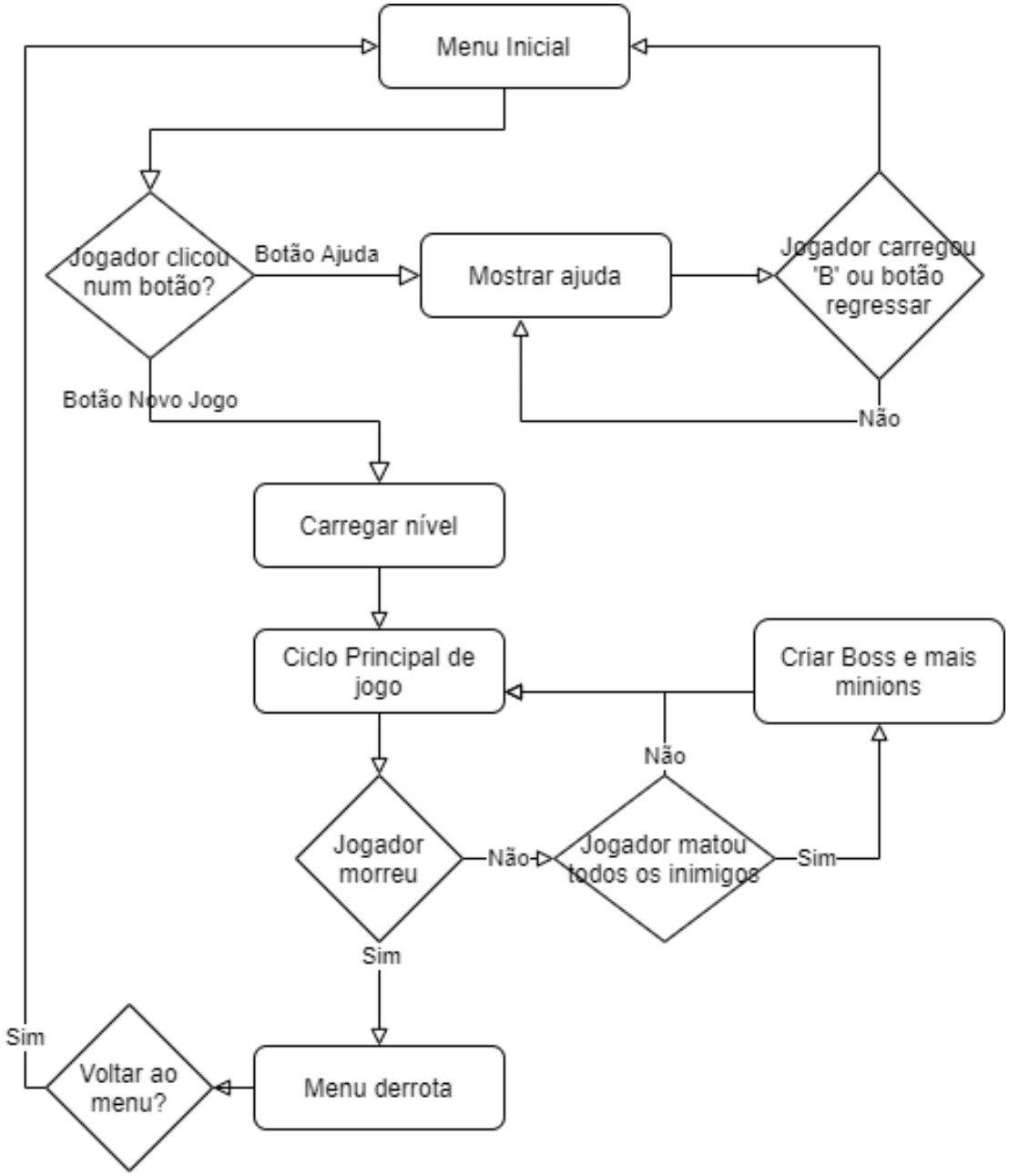
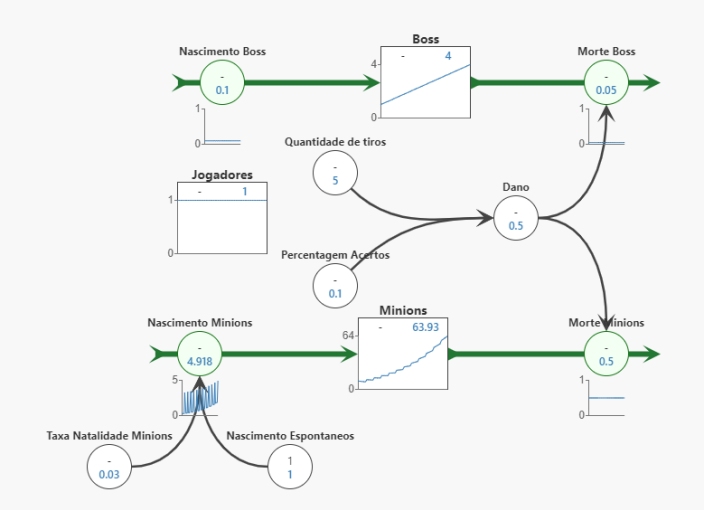



*Figura 2 - Fluxo do jogo aplicação*

O diagrama, presente na figura 2, representa o fluxo de atividades que um jogador experiência aquando da execução da aplicação. Este ciclo corresponde por traços grossos ao fluxo do programa tendo em conta os pressupostos mencionados em cima. Não inclui detalhes tais como o desenrolar do jogo, que apenas se encontra representado pela ação “ciclo principal de jogo” pois é impossível prever todas as ações que todos os agentes irão tomar e qual o passo seguinte do sistema. Desta forma apenas representamos a condição de entrada e saída deste macro estado.

*Reprodução dos Inimigos (Stocks e Flows)*

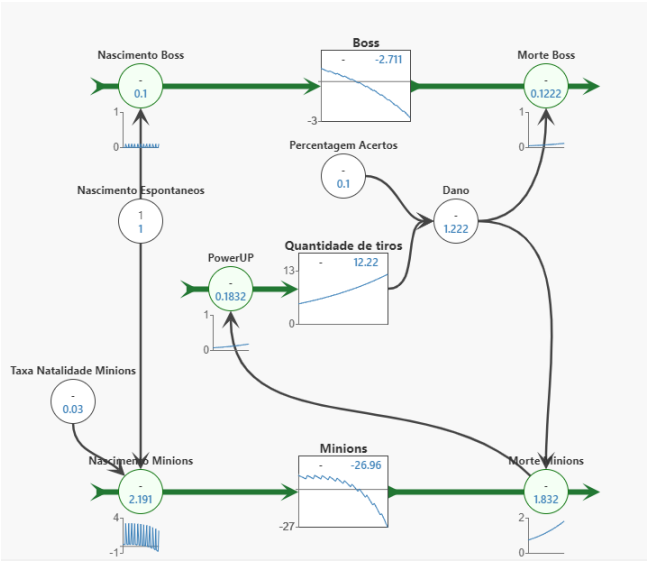
De um ponto de vista muito grosseiro, podemos dizer que os inimigos são presas e o jogador um predador. Assim sendo podemos modelar o comportamento do sistema através de um diagrama de stocks e flows. Diagrama este muito útil para tentar equilibrar a complexidade do jogo, mantendo o desafio, mas não permitindo uma explosão na população de inimigos.




Este modelo tem algumas variáveis fixas, nomeadamente os nascimentos espontâneos e a taxa de natalidade, ambos os valores incluídos no código, apesar de não explicitamente. Como o nosso projeto é um jogo, o predador não foi incluído no diagrama apesar de estar representado com a quantidade de tiros e a percentagem de acertos, em média que o jogador acerta.

Através deste diagrama é possível ver facilmente que se nada mudar, depressa o número de inimigos aumenta exponencialmente, chegando eventualmente a despoletar a condição de perda por “perda do planeta”.

Assim sendo, para equilibrar o jogo, decidimos que seria uma boa ideia implementar power ups para o jogador, dando um sentido de progresso e possibilitando um equilíbrio dos números numa fase mais tardia do jogo. Power ups este incluído no diagrama seguinte.




Desta forma somos capazes de controlar a quantidade de dano debitada pelo jogador e assim levar a uma vitoria mais rápida. De notar que a quantidade de tiros é também um stock, que é influenciado

pela quantidade de mortes de *minions*. Ou seja, cada vez que o jogador mata um *minion* existe uma pequena probabilidade de o *minion* largar um *power up* que aumente a quantidade de tiros, e assim aumenta a quantidade de dano dada por este que contraria a tendência de crescimento da população de minions, equilibrando o jogo e dando uma probabilidade de vitoria maior.

O boss é um inimgo que nasce a intervalos regulares de aproximadamente 5 segundos, e que possui uma vida superior aos minions, desta forma através do modelo de stocks e flows podemos ver que apesar de estes serem repostos a cada 5 segundos o jogador á partida é mais que capaz de os eliminar constantemente, entrando num estado em que apenas minions estão presentes no jogo.

Finalmente a variável percentagem de acertos é um valor determinado empiricamente entre os dois autores do projeto, e que resultou em gráficos que permitem interpretar o resultado pretendido, a percentagem de acertos do leitor, pode ser maior ou menor dependendo da sua habilidade.

## Entidades (Agentes)

A classe abstrata entidade fornece métodos e atributos que correspondem a um agente que pode ou não ter inteligência. Um agente interage com o ambiente de jogo e com os outros agentes e leva á simulação de agentes complexos.

Todos os agentes são responsáveis pelo seu próprio desenho e pela criação do seu corpo (Body). A classe Animador e a classe Body são partes constituintes destes

### Agentes autónomos

Um agente autónomo é um agente inteligente que atua tendo em conta um conjunto de instruções previamente definidos com um elevado grau ou completa independência de quem criou esse agente. Pode apresentar visão e conhecimento do seu mundo, e tomar decisões baseadas em funções heurísticas (funções matemáticas que atribuem valores ao estado do mundo para aproximar o melhor possível o agente ao estado de destino).

No contexto do nosso trabalho os agentes possuem dois grandes comportamentos, a auto preservação (tentar desviar das balas) e o patrol (seguir caminho previamente definido).

### Boid

Um boid é um modelo que representa vida artificial, nomeadamente o comportamento de pássaros. Estes boids são agentes autónomos que em grupo apresentam comportamentos interessantes, tais como flocking, Alinhamento, patrol, entre outros comportamentos que possam ser relevantes.

No nosso trabalho utilizamos boid com corportamento de waypoint para definir um traçado a seguir pelos inimigos de modo a otimizar a conquista do planeta, enquanto tentam eliminar o jogador.

### Modelo baseado em agentes

Um modelo baseado em agentes é um modelo computacional utilizado para simular as ações e interações de agentes autónomos que procura estudar o impacto que esses agentes possuem nesse sistema. No nosso trabalho os agentes não têm grande impacto no ambiente e realmente interagem pouco com este, sendo a única interação a condição de perda do jogo por perda do mundo derivado á conquista do mundo pelos invasores.

## Classe Entidade

Como mencionado em cima tanto os inimigos como o jogador e por conveniência até as bombas largadas por inimigos são entidades. Entidades estas afetadas pelo ambiente (força gravítica e colisões entre elas). Estas entidades desenham-se a si próprias e detetam se outro corpo colidiu consigo através do seu atributo corpo. Assim sendo, é nos conveniente que a classe entidade seja uma classe abstrata que delega ás suas implementações a implementação de métodos que sejam necessário.

Uma entidade é independente de todas as outras entidades, devendo ser instanciada dentro da própria classe jogo, de modo a fornecer um ambiente comum entre todas estas. Ao criar uma entidade os objetos Animador e Body são criados dentro do próprio construtor através da chamada a um método abstrato implementado pelas classes que extendam de entidade. Desta forma cada classe possui uma forma de se controlar a si própria obtendo o *sprite* próprio e sem necessitar de complicar a situação para a classe entidade.

Ou seja, sem heranças e polimorfismo teríamos de declarar cada classe como independente, criando e garantindo manualmente a correta implementação dos métodos, copiando e colando código entre classes, levando á introdução de diversos pontos de falha, dificultando a compreensão do código, e posterior modificação deste. Desta forma temos código robusto, pouco propenso a erros, e fácil de modificar, mantendo ainda assim a modularidade e garantindo as diferenças entre cada uma destas classes.

O comportamento de cada uma destas entidades é garantido pelo input do teclado (jogador), ou através da própria classe Jogo que garante o movimento e interação de todas as entidades. Uma alternativa melhor seria a implementação de comportamentos diretamente através de uma interface que seria implementada pela classe Entidade. No entanto esta solução não foi explorada por falta de tempo.

### *Classe Animador*

A classe Animador é a classe responsável por representar em modo gráfico uma entidade. Esta classe faz recurso de sprites para representar o movimento e o estado das entidades.

Um sprite é uma imagem PNG que se move na tela sem deixar rasto da sua passagem. No nosso caso o sprite, é uma imagem composta por muitas imagens pequenas que correspondem a cada um dos estados da animação. Assim sendo uma única frame corresponde uma pequena parte da imagem.

Através da utilização de uma única imagem e correspondente corte, somos capazes de carregar apenas uma imagem, reduzindo o custo de memoria, mas aumentando um pouco o custo computacional. Assim apenas carregamos a imagem uma única vez, e vamos selecionando a parte que nos é conveniente.

*Figura 3-Sprite com animações do jogador*

A animação surge assim através da sobreposição de imagens sucessivas com poucas diferenças entre elas que dá a ideia de movimento.

Dependendo da posição do jogador a animação do movimento selecionada corresponde ás linhas presentes na figura acima. Caso seja necessário efetuar uma animação relativa á mudança de direção, podemos obter a animação nas colunas.

Através destes sprites somos capazes de ter animações visualmente apelativas, mas com pouco custo de memoria, ou seja, não é necessário carregar uma imagem por cada frame e sim escolher partes desta imagem. Esta animação é implementada pela classe Animador, sendo a imagem selecionada através da classe SpriteDef. Esta classe SpriteDef, tem em conta o movimento da Entidade, e a intervalos regulares troca a frame que é escolhida utilizando um ficheiro JSON que contem as informações da animação.

Um ficheiro JSON é uma representação de objetos em Java script, que permite a troca de dados de maneira rápida e simples entre diferentes linguagens de programação. O JAVA possui funcionalidades capazes de ler os ficheiros e assim apenas é necessário obter as informações contidas neste ficheiro. Cada Sprite possui um ficheiro JSON correspondente que inclui informações sobre como cortar a imagem em segmentos e qual a sequência de segmentos para originar a animação.

### *Body*

A classe body implementa uma espécie de RigidBody do Unity, lecionado noutra disciplina do mesmo curso. Este corpo, possui métodos para colidir com outros corpos, definir as suas dimensões (regra geral iguais á da frame atual). Através desta classe e como está contida dentro de uma entidade somos capazes de definir com precisão a posição da entidade e do seu corpo. Todas as operações de movimento serão aplicadas nesta classe tais como forças, alterar a posição manualmente e posteriormente comunicar a alteração á entidade que ira depois mudar comunicar com o animador a alteração e possivelmente até atualizar a sua animação.

## Níveis e menus

Um nível ou menu corresponde ao estado do jogo, assim sendo é uma interface fundamental para dar ao utilizador informação visual sobre qual a posição em que este se encontra relativamente ao fluxograma da aplicação presente na figura 2.

Estes menus, podem possuir ou não botões conforme necessário que serão ativos através do event handler do Processing, este método é chamado cada vez que existe um clique na janela e através da posição do clique identificamos se e qual o botão que foi clicado e tendo em conta o botão qual o próximo passo que a aplicação irá tomar. Podendo seguir para um outro menu que altera a apresentação da aplicação mostrando imagens e afins. Ou iniciando o jogo propriamente dito.

Um dos menus, o menu principal, possui botões bem definidos cada um com um fundo específico, estes botões são desenhados com as coordenadas especificas dentro da janela, sendo que o clique é detetado pela sobreposição do clique com o espaço ocupado pelo botão. Caso tal sobreposição seja verdadeira, o clique é detetado como sendo naquele botão em específico.

Um menu pode possuir também um fundo, este fundo ocupa toda a janela (ou múltiplos fundos que ocupam diferentes partes da janela, no caso do menu de jogo). Fundo este que será desenhado antes do desenho das entidades de modo a que estas estejam sempre por cima do fundo.

## Áudio

Na sequência da implementação dos menus foi introduzido áudio de fundo ao jogo. Tanto no menu principal como nos sequentes menus, existe sons de fundo. A classe Áudio é responsável por reproduzir sons ou músicas conforme seja necessário. De notar a utilização das capacidades do JAVA em prol das capacidades implementadas pelo Processing para reproduzir áudio. Tal deveu-se á possibilidade de utilizar ficheiros de maior qualidade, maior controlo e uma melhor documentação existente nas classes incluídas com o JAVA. Caso tivéssemos optados por utilizar as funções fornecidas pelo Processing possivelmente obteríamos o mesmo resultado, no entanto não foi necessário.

Um clip de áudio corresponde a um ficheiro WAV, escolhemos este tipo de ficheiro devido á sua elevada qualidade de som e possibilidade de manipulação.

Vale mencionar que, por exemplo cada tiro possui um efeito sonoro, reproduzido cada vez que um tiro é disparado, através da existência de uma instância de áudio presente na classe Osso.

## PowerUPS

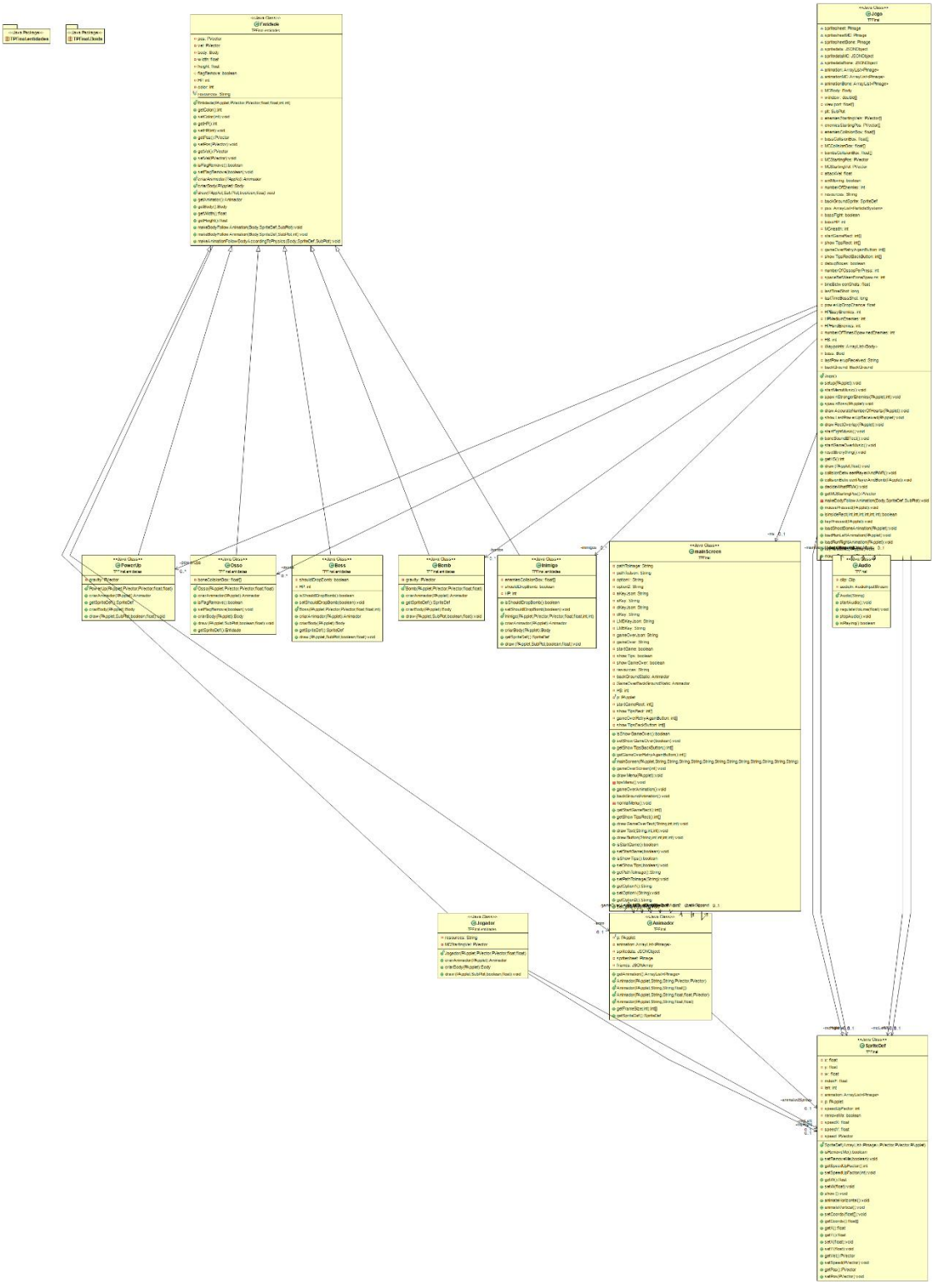
Um power up altera o fluxo do jogo de modo a fornecer equilíbrio numa parte posterior e assim controlar a população de inimigos de uma maneira mais equilibrada.

Quando um inimigo morre, existe uma pequena probabilidade da bomba que ele larga se transformar num power up. Desta forma com o aumento de mortes de inimigos a quantidade de dano debitada pelo jogador aumenta controlando assim a população e pode levar á vitoria. Assim sendo como é necessário a colisão do jogador com o *power UP* para detetar se o jogador o apanhou ou não, foi-nos conveniente definir o power up como uma entidade e aplicar-lhe uma força igual á força gravítica.

Existem diversos tipos de powerups implementados, nomeadamente maior velocidade de disparo, disparos múltiplos e maior velocidade dos tiros. Existe também um power up que fornece vidas ao jogador, permitindo a prolongação da vida deste, mesmo que este esgote as suas vidas iniciais.

# Diagrama de classes final

Tal como mencionado na introdução, foi elaborado um diagrama de classes que nos serviu como guia para a realização do trabalho, no entanto aquando da implementação algumas coisas não foram respeitadas e outras foram alteradas. Assim sendo o diagrama final é o seguinte.



# Conclusão

Este trabalho surgiu no culminar da cadeira de MSSN. Com o terminar do relatório esperamos ter explicado sucintamente toda a matéria lecionada e como ela foi aplicada no desenrolar do projeto.

Havia partes que poderiam ter sido mais bem implementadas, no entanto devido a restrições de tempo o projeto não foi devidamente planeado e como tal houve coisas que foram sendo pensadas conforme iam surgindo. No entanto implementamos todas as funções que pretendíamos nos objetivos e mais algumas, tivemos uma aplicação visualmente apelativa e a funcionar sem bugs de maior importância.

Abordamos temas como programação por agentes, boids, animações, forcas e movimentos e esperamos ter atingido os objetivos propostos para a conclusão da unidade curricular.