INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

Sobrevivência – Jogo Interactivo para Estudo de Arquitecturas de Agentes Autónomos

Eduardo Miguel Gonçalves Silva

(Licenciado)

Trabalho de projecto para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Júri:

Presidente:

Professor Coordenador Doutor Manuel Barata, ISEL – DEETC

Vogais:

Arguente: Professor Adjunto Doutor Paulo Trigo, ISEL – DEETC

Orientador: Professor Adjunto Doutor Luís Morgado, ISEL – DEETC

Setembro de 2013

Resumo

Este trabalho teve como objectivo a criação de um jogo para servir como plataforma de estudo de arquitecturas de agentes autónomos. O jogo consiste na simulação de um ecossistema onde existem várias espécies de seres vivos, nomeadamente espécies animais e vários tipos de espécies vegetais que se dividem em plantas e frutos. O jogo baseia-se nos princípios dos ecossistemas, em que domina o princípio da sobrevivência do mais forte na cadeia alimentar, ou seja, alimentar-se e não servir de alimento e assim sobreviver.

Este jogo foi desenvolvido de raiz, passando por diversas fases de desenvolvimento, até chegar ao produto final. Este foi implementado com recurso a *API* do *pygame*, que fornece diversos mecanismos de criação de jogos, para além de ter disponíveis vários elementos/projectos criados por utilizadores.

No seu desenvolvimento foram implementados todos os mecanismos e as regras do jogo, que darão suporte à dinâmica do jogo, tendo em especial atenção o módulo de criação de agentes inteligentes para facilitar o uso do mesmo.

Foram utlizadas várias técnicas de inteligência artificial na implementação dos agentes inteligentes, de modo a que estes sejam usados para modelar as personagens do jogo.

Palavras-chave: jogos de computador, agentes inteligentes, inteligência artificial, *pygame*, plataforma de estudo, ecossistema

Abstract

The purpose of the work was to create a game to serve as a platform for the study of autonomous agents architectures. The game consists in the simulation of an ecosystem where there are several living species, including animal species and various types of plant species that are divided into plants and fruits. The game is based on the principles of ecosystems, where the principle of survival of the fittest in the food chain dominates, ie, feed themselves and not serving food and thus surviving.

This game was developed from scratch, going through various stages of development to reach the final product. This is implemented using pygame's API, which provides various mechanisms for creating games, in addition to having available several elements and projects created by users.

In its development it was implemented all the mechanisms and rules of the game, that will support the game dynamics, with particular focus on the module that creates intelligent agents to facilitate the use of the same.

It was used several artificial intelligence techniques in the intelligent agents implementation so that they may be used to model the characters of the game.

Keywords: computer games, intelligent agents, artificial intelligence, pygame, study platform, ecosystem

Agradecimentos

Agradeço aos meus familiares por todo o apoio e motivação que me deram durantes estes anos de estudo, que sempre me apoiaram moralmente e economicamente a minha formação académica.

Agradeço também a todos os colegas e professores que me apoiaram e pela entre ajuda na realização dos trabalhos, em especial ao Professor Doutor Luís Morgado pela sua orientação, pela sua disponibilidade para prestar esclarecimento de dúvidas e pela motivação acrescida que me deu durante a realização deste desafio.

Índice

1 Introdução 1

1.1 Motivação 2

1.2 Objectivos 3

1.3 Organização do projecto 4

1.4 Convenções de escrita 5

2 Trabalho relacionado 7

2.1 Desenvolvimento de jogos 7

2.1.1 Plano de desenvolvimento 8

2.1.1.1 Conceito do jogo e documento de visão 8

2.1.1.2 Concepção conceptual do jogo 9

2.1.1.2.1 Definição da jogabilidade central 10

2.1.1.2.2 Definição da jogabilidade contextual 10

2.1.1.2.3 Definição da história do jogo 11

2.1.1.2.4 Gestão dos recursos do jogo 11

2.1.1.3 Concepção técnica do jogo 11

2.1.1.4 Finalização do projecto 12

2.2 Plataforma de desenvolvimento 13

2.3 Modelação de personagens com base em agentes inteligentes 14

2.3.1 Conceito de agente inteligente 14

2.3.2 Realismo do sistema 14

2.3.3 Mecanismos de planeamento 15

2.3.4 Agentes reactivos 16

2.3.4.1 Modelação de comportamentos com base em máquinas de estados finitos 16

2.3.4.2 Navegação com base em campos de potencial 17

2.3.5 Agentes deliberativos 18

2.3.5.1 Mecanismos procura em espaço de estados pra planeamento automático 18

2.3.5.1.1 Navegação com base em grelha 18

2.3.5.1.2 Navegação por pontos de visão 19

2.3.5.1.3 Malha de navegação 20

2.3.5.1.4 Mapas de influência 21

2.3.6 Agentes híbridos 23

3 Especificação de requisitos do jogo 25

3.1 Visão de jogo 25

3.2 Casos de utilização...26

3.2.1 Descrição dos casos de utilização 27

3.3 Modelo de domínio...31

3.4 Especificação suplementar..32

3.5 Protótipo de requisitos 33

4 Concepção da solução 35

4.1 Definição do nível de jogo..35

4.2 Ciclo de jogo...36

4.2.1 Reaparecimento de unidades.. 37

4.2.2 Actualização da vista 38

4.3 Eventos das entidades 39

4.3.1 Eventos das entidades personagem 39

4.3.1.1 Evento de passo... 39

4.3.1.2 Evento de colisão 40

4.3.2 Eventos das entidades inanimadas 41

4.4 Controlador das entidades...41

4.5 Carregamento dinâmico de níveis..43

4.6 Comportamento das personagens 45

5 Arquitectura da solução.. 49

5.1 Desenvolvimento de jogos em pygame 49

5.2 Especificação da arquitectura da solução 51

5.2.1 Arquitectura da camada gráfica 53

5.2.2 Arquitectura da camada de motor de jogo 54

5.2.2.1 Arquitectura do modelo 56

5.2.2.2 Arquitectura da vista 57

5.2.2.3 Arquitectura do controlador 59

5.2.2.4 Módulo de abstracção da camada de motor 60

5.2.3 Arquitectura da camada de ambiente 61

5.2.3.1 Arquitectura do subsistema de objectos pré-definidos 62

5.2.3.2 Arquitectura do subsistema de acções pré-definidas 65

5.2.3.3 Arquitectura do subsistema de controlador............................. 66

5.2.3.4 Arquitectura do subsistema de sensores.................................. 68

5.2.3.5 Módulo de construção de controladores 69

5.2.4 Arquitectura da camada de personagens.. 70

5.2.4.1 Arquitectura da percepção da máquina de estados finitos 71

5.2.4.2 Arquitectura dos estados existentes 72

5.2.4.3 Módulo de construção de controladores da camada de personagens.. 74

6 Implementação da solução 75

6.1 Criação das áreas de jogo...75

6.2 Geração de eventos 75

6.2.1 Geração do evento de clique e de tecla 75

6.2.2 Geração do evento de passo e de colisão 76

6.3 Construção da percepção deliberativa 78
7 Verificação e testes.. 79
7.1 Verificação e teste ao carregamento de níveis..79
7.2 Verificação e teste da colisão entre entidades 79
7.3 Verificação e teste do reaparecimento das entidades...............................80
8 Conclusões 81
8.1 Trabalho futuro 82
8.1.1 Mapas de influência 82
8.1.2 Ferramentas de teste... 82
8.1.3 Adição de novos elementos ao jogo... 83
8.1.4 Algoritmo de intercepção... 84
8.1.5 Comportamento de grupo... 85
8.2 Considerações finais 85
9 Bibliografia... 87
Apêndice I – Uso do *Tiled* 89

Índice de figuras

Figura 1 - Maquina de estados do fantasma 16
Figura 2 – Exemplo de campos de potencial pré-processado............................ 17
Figura 3 – Exemplo de uma grelha... 19
Figura 4 – Exemplo de mapa em grafo... 20
Figura 5 – Exemplo de mapa em malha 21
Figura 6 – Mapa de influência dois contra um 22
Figura 7 – Maquete da interface gráfica 26
Figura 8 – Diagrama de casos de utilização 27
Figura 9 – Modelo de domínio 31
Figura 10 - Aspecto gráfico do protótipo 34
Figura 11 - Diagrama de sequência do ciclo de jogo.. 37
Figura 12 - Diagrama de sequência do reaparecimento das unidades 38
Figura 13 - Diagrama de sequência do evento de step das personagens 40
Figura 14 - Diagrama de sequência do evento de colisão das personagens 40
Figura 15 - Diagrama de sequência de controlador deliberativo....................... 42
Figura 16 - Diagrama de sequência do carregamento dos tiles 44
Figura 17 - Diagrama de sequência do carregamento dos objectos de jogo....... 45
Figura 18 – Especificação do comportamento do predador 46
Figura 19 - Especificação do comportamento da presa..................................... 47
Figura 20 – Especificação do comportamento do caçador 47
Figura 21 - Diagrama de sequência da MEF 48
Figura 22 - Arquitectura de camadas... 52
Figura 23 – Arquitectura das *sprites*.. 53
Figura 24 – Adaptadores.. 54
Figura 25 - *Model View Controller* 55
Figura 26 - Arquitectura do modelo 57
Figura 27 - Arquitectura da vista... 57
Figura 28 - Arquitectura do controlador.. 59
Figura 29 - *Façade* da camada de motor 60
Figura 30 - Arquitectura da camada de ambiente.. 62
Figura 31 - Arquitectura das entidades.. 63

Figura 32 – Arquitectura da barra de menu 65
Figura 33 - Arquitectura das acções das unidades 66
Figura 34 - Arquitectura do controlo 67
Figura 35 - Arquitectura do mecanismo de espaço de estados 68
Figura 36 - Arquitectura dos sensores 69
Figura 37 - Arquitectura da construção de controladores 70
Figura 38 - Arquitectura da cama de personagens 71
Figura 39 - Arquitectura da percepção da máquina de estados finitos 72
Figura 40 - Estados existentes na plataforma 73
Figura 41 - Arquitectura da MEF do predador 73
Figura 42 - Arquitectura do módulo de construção de controladores da camada de personagens 74
Figura 43 - Diagrama de sequência do evento de clique 76
Figura 44 - Diagrama de sequência do evento de passo e de colisão 77

Glossário

NPC	N*on-Player Character* (personagem não jogável)
GUI	*Graphical User Interface* (interface gráfica do utilizador)
API	*Application Programming Interface* (interface de programação de aplicações)
IA	Inteligência Artificial
MPEE	Mecanismo de Procura em Espaço de Estados
MEF	Máquinas de Estados Finitos

1 Introdução

O projecto descrito neste relatório consiste na criação de um jogo interactivo para suporte do estudo de arquitecturas de agentes autónomos. Este tem duas vertentes, a primeira, consiste na realização de uma personagem principal que seja jogada por um utilizador, na segunda, a personagem principal é autónoma, ou seja, é controlada por um agente autónomo.

O jogo consiste na simulação de um ecossistema onde existem várias espécies de seres vivos, nomeadamente espécies animais e vários tipos de espécies vegetais que se dividem em plantas e frutos. O jogo baseia-se nos princípios dos ecossistemas, em que domina o princípio da sobrevivência do mais forte na cadeia alimentar, ou seja, alimentar-se e não servir de alimento e assim sobreviver. Para atingir esse objectivo nomeadamente a sobrevivência, os animais têm de se alimentar de outros animais, ou seja, cada espécie tem os seus predadores e/ou presas, para além disso também podem comer espécies vegetais.

Como referido anteriormente o projecto tem por objectivo dar suporte ao estudo de arquitecturas de agentes autónomos. Assim sendo, a plataforma do jogo terá de fornecer um mecanismo que simplifique essa tarefa, para que quem a use com esse intuito tenha de implementar o mínimo de código possível para integrar novos agentes no jogo.

O processo de desenvolvimento de um jogo passa por várias etapas, como em qualquer processo de desenvolvimento de outro tipo de *software*. No caso do desenvolvimento de jogos, temos de dar grande ênfase na etapa de especificação de requisitos, pois é nesta fase que é determinado que tipo de jogo será desenvolvido e a sua jogabilidade, ou seja, é nesta fase que temos que definir que tipo de jogo queremos, o jogo poderá ser de acção, estratégia, ou de outro tipo.

A outra fase que temos de enfatizar é a fase da criação do motor de jogo, pois é neste que o jogo assenta. O motor do jogo, ao contrário do pensamento comum, não é apenas o aspecto gráfico do jogo, mas é também responsável pela simulação da física do mesmo. É no motor do jogo que se implementam algumas das regras do ambiente do jogo, tais como a definição das unidades que se movem e interagem com o mesmo. Pelos princípios da física pode-se entender vários aspectos como inércia, velocidade,

aceleração, gravidade, atrito e outros. Temos que analisar e reflectir sobre todas as variáveis previsíveis que possam existir no mundo real, que sejam possível, simular num jogo face à limitação dos nossos, recursos tornando-o mais rico e mais apelativo, podendo ir para além do real até ao limite da imaginação do seu criador e dos seus recursos.

Como referido anteriormente a jogabilidade de um jogo é muito importante, constituindo-se a inteligência artificial dos inimigos o seu aspecto central, é esta que influencia mais a jogabilidade, pois determina a forma como as personagens não jogáveis (*non-player character*-NPC) jogam e determinam o desafio colocado ao jogador. Nada é mais frustrante do que um NPC que tem poderes sobre-humanos, ou seja, memória e tempos de reacção humanamente impossíveis, ou que seja simplesmente demasiado fácil que não traga desafio ao jogo, em ambos os casos o jogo deixa de ser atractivo.

Assim, o grande desafio da inteligência artificial num jogo, não é o de chegar a uma solução óptima ou próxima dessa, como normalmente se faz em investigação ou noutros cenários similares de optimização de soluções, mas sim uma tentativa de simular o comportamento humano, ou seja, ter falhas e limitações que são inerentes ao ser humano patente na conhecida locução latina de “errare humanum est”.

Neste relatório será discutido o desenvolvimento de jogos, abordando as diferentes fases do seu desenvolvimento, centrados no jogo realizado, descrevendo como foi desenvolvido (a sua arquitectura, jogabilidade e aspecto gráfico), não esquecendo a introdução à plataforma usada. Iremos também falar sobre como foi implementada a inteligência artificial neste jogo e como é possível usá-la para aprender sobre a mesma.

1.1 Motivação

Com o passar dos anos, a indústria dos jogos tem vindo a crescer, sendo que o desenvolvimento dos mesmos tem-se tornado um processo cada vez mais complexo e moroso. Este desenvolvimento, acarreta riscos, pois é um processo que engloba muitos recursos humanos, financeiros, temporais e materiais, sabendo à partida que alguns jogos nunca serão lançados no mercado e que são poucos os que darão lucros às editoras.

O desenvolvimento de jogos no passado concentrou-se principalmente no desenvolvimento dos gráficos dos mesmos, seguindo a máxima em que jogos bonitos se vendem mais facilmente e dão mais lucros. Esta estratégia prevaleceu durante muito tempo e prevaleceu até há poucos anos, mas os gráficos chegaram a uma patamar tão elevado, que as suas melhorias deixaram de ter o impacto que tinham. A partir desse momento, o foco do desenvolvimento de jogos voltou-se para outro aspecto, centrou-se na inteligência artificial do jogo, com o objectivo de o tornar mais real e com melhor jogabilidade.

Perante este cenário faz todo o sentido em termos académicos tentar perceber como os jogos são desenvolvidos e como a inteligência artificial dos mesmos, os influencia.

1.2 Objectivos

Pelo atrás exposto, este projecto propõe-se a desenvolver um jogo, e que este sirva para o processo de aprendizagem de inteligência artificial. É com base nessa finalidade que foram traçados os seguintes objectivos específicos:

- Elaborar as regras do jogo, onde se determina como o jogo funciona e qual a informação que é apresentada ao utilizador;
- Determinar o seu aspecto gráfico, que torne este jogo atractivo;
- Fazer uma pequena síntese do processo de criação de jogos, explanando as várias fases de desenvolvimentos e os seus riscos;
- Fazer uma pequena síntese sobre a inteligência artificial e o seu uso em jogos;
- Criar o motor do jogo onde é implementado o funcionamento base do mesmo;
- Criar a estrutura de inteligência artificial de forma expansível e que seja relativamente fácil de alterar face a novas necessidades;
- Elaborar testes e aperfeiçoar o jogo.

1.3 Organização do projecto

Para além deste capítulo de introdução, este projecto encontra-se organizado em mais sete capítulos, perfazendo um total de oito:

Capítulo 2: Trabalho relacionado

Este capítulo contém todo o trabalho prévio à elaboração do jogo em si, ou seja, investigação e experimentação, este fala sobre o desenvolvimento de jogos, sobre a escolha da plataforma utilizada e sobre a IA nos jogos.

Capítulo 3: Especificação Requisitos

Este capítulo contém, a especificação de requisitos do jogo realizado, nomeadamente a visão do jogo e os seus requisitos na forma de casos de utilização e especificação suplementar.

Capítulo 4: Concepção da Solução

Este capítulo serve como intermédio ente a especificação dos requisitos e a criação da arquitectura da solução, esta contém realização conceptual dos casos de utilização e as opções tomadas.

Capítulo 5: Arquitectura da Solução

Este capítulo contém uma introdução ao desenvolvimento na plataforma e a arquitectura elaborada a partir da concepção da solução, dando suporte aos mecanismos estabelecidos na mesma.

Capítulo 6: Implementação da solução

Este capítulo contém os detalhes de implementação que são mais relevantes e como estes se encontram implementados.

Capítulo 7: Verificação e testes

Este capítulo contém os testes unitários realizados à solução, como são feitos e o seu resultado.

Capítulo 8: Conclusões

Este capítulo contém o resultado do trabalho efectuado, discutindo aspectos que foram positivos e aqueles que se constituem como áreas de implementação futura de

melhorias, também é efectuada uma reflexão sobre o trabalho futuro e dadas sugestões com vista a melhorar e enriquecer a solução produzida.

1.4 Convenções de escrita

No sentido de facilitar a leitura, ao longo do relatório são utilizadas as seguintes convenções de escrita:

- As traduções de termos ou expressões originalmente em língua inglesa, na sua primeira ocorrência no texto, surgem seguidas da designação original, entre parêntesis e entre aspas;
- Em relação às siglas utilizadas, dado o seu uso generalizado na comunidade técnico-científica, são mantidas tal como aparecem no original;
- O grafismo em itálico é utilizado para destacar termos em Inglês;
- O código deste projecto está todo em inglês, os diagramas irão conter as classes no seu nome original e no texto terão o seu nome traduzido, com o seu nome original entre parênteses na sua primeira ocorrência.

2 Trabalho relacionado

Neste capítulo, será explicado todo o trabalho e pesquisa efectuada, antes e durante a elaboração do projecto. Este começa com a explicação do processo de desenvolvimento de um jogo, desde uma simples ideia ao término do mesmo, passando pelas diversas fases de desenvolvimento ao longo do projecto. Passa também pela escolha da plataforma que é usada neste projecto. E para terminar, uma explicação de como a IA é usada nos jogos e as técnicas usadas nos mesmos.

2.1 Desenvolvimento de jogos

O desenvolvimento de jogos, como referido anteriormente é um desenvolvimento de *software*, mas estes também são considerados uma arte, pois para além do grafismo por vezes artístico, têm som e envolvem criatividade.

Como os jogos e a maioria do *software* vão aumentando de complexidade e de tamanho à medida que os anos passam [Blow, 2004], surge a necessidade de usar uma metodologia de trabalho de forma a lidar com essa complexidade, de forma a ser possível usar os recursos existentes de uma maneira mais eficiente, sejam estes: tempo, dinheiro ou mão-de-obra.

A metodologia passa pela criação de um plano de desenvolvimento para coordenar a equipa e gerir os diversos recursos utilizados pelo desenvolvimento do jogo, essa melhor gestão leva a que o risco de um projecto falhar seja diminuído.

Mas antes de criar um plano é necessário saber exactamente o que se irá desenvolver, ou seja, qual o tipo de jogo se trata. Existem vários factores a ter em conta quando se escolhe qual o tipo de jogo pois este tem de se adaptar ao público-alvo [Schell, 2008], ou seja, para além dos gostos pessoais existem aspectos que ajudam a determinar o gosto do público, como por exemplo os aspectos demográficos, como a idade e o género.

Cada faixa etária tem os seus gostos, por exemplo um adolescente não tem os mesmos tipos de gostos que uma criança, logo um jogo para essas faixas etárias terá de ser diferente, por exemplo uma criança gosta mais de jogos simples como os jogos da "*Disney*" e um adolescente prefere jogos complexos e com violência como o "*Grand Theft Auto*". O género do público-alvo também influencia os gostos de cada um, pois

geralmente as mulheres gostam mais de jogos sociais como o "*Sims*" e os homens gostam mais de jogos com muita acção e destruição como o "*Call of Duty*".

Na criação de um jogo, um elemento da equipa de desenvolvimento tem de ter estes aspectos em mente, pois este tem de ponderar cuidadosamente, qual o seu público-alvo e os seus interesses, um exemplo de um erro um tanto ou quanto exagerado poderia ser criar um jogo que apele às crianças, onde o seu tema seja impróprio para as mesmas, que levaria ao falhanço do jogo por completo, tendo em conta os problemas legais e éticos inerentes. Daí advém a obrigatória oposição da classificação do mesmo relativo aos destinatários.

2.1.1 Plano de desenvolvimento

Como referido anteriormente é necessário ter um plano ao desenvolver um jogo. Um dos muitos erros no desenvolvimento é passar logo à fase de implementação sem sequer ter uma visão do que se vai fazer nem como o fazer. Nesse sentido, de seguida, serão descritas as várias fases de desenvolvimento de um jogo e o papel de cada uma delas.

2.1.1.1 Conceito do jogo e documento de visão

É nesta fase que é criado o conceito do jogo, ou seja, passar de uma simples ideia e evoluir sobre a mesma. A criação do conceito do jogo não surge do nada, surge de uma conjunto de ideias, ideias essas que aparecem normalmente em sessões de "*brainstorming"*. O resultado dessas sessões é um conjunto de diversas ideias e cabe ao projectista decidir quais destas valem a pena incluir e quais são compatíveis.

Após esse processo, essas ideias tornam-se o conceito do jogo, o conceito então deve ser traduzido para um documento de visão, onde consta toda a informação relevante sobre o jogo, informação como por exemplo: que modos terá, se tem o modo de um único jogador, modo de múltiplo jogador ou até os dois; qual a plataforma alvo e qual a tecnologia a utilizar.

A informação mais relevante sobre o jogo é o que se pretende fazer e como é feito, a isso é chamado análise de requisitos.

Para elaborar os requisitos, inicialmente teremos que nos focar nos aspectos principais do jogo, qual será o papel do jogador, que acções o mesmo pode fazer, como será o mundo à sua volta e assim em diante até que haja certeza que nenhum aspecto

tenha sido esquecido. Em seguida teremos de organizar os mesmos em grupos, por exemplo acções do utilizador e acções do mundo.

É importante que nesta fase os requisitos estejam focados só nos aspectos fulcrais do jogo, sendo necessário verificar se a ideia não ficou demasiado dispersa. Assim a delimitação temática é fundamental, ou seja, é necessário verificar se foram inseridas ideias que não sejam muito compatíveis, pois por vezes existe uma tentativa de criar vários jogos num, o que geralmente acaba em desastre pois causa uma falta de foco naquilo que é importante.

2.1.1.2 Concepção conceptual do jogo

É nesta fase que é criado o documento de especificação conceptual do jogo, este documento detalha todas as personagens do jogo, os seus níveis/áreas, o mundo à sua volta, menus e outros. Este documento é construído com base no conceito do jogo criado na fase anterior, a criação do mesmo não é simples, pois é impossível criar um documento de especificação conceptual do jogo acabado no início do mesmo, e um projectista não deve ficar com a ideia de criar um documento de *design* acabado pois este irá ficar condicionado ao mesmo, ou mesmo ficar relutante à introdução de mudanças no mesmo.

O documento de especificação conceptual do jogo nunca está acabado, porque este estará sempre em constante evolução à medida que o jogo irá sendo implementado, pois novas ideias surgem, alguma são descartadas e outras modificações serão certamente introduzidas para equilibrar a dinâmica do jogo.

Este documento é útil, pois permite ao projectista ter uma noção real do que será feito, ajudando-o a perceber quanto tempo cada um dessas ideias demorará a ser implementada, quais as que dependem de outras e quais aquelas que não são exequíveis, tendo em conta os recursos à sua disposição. Este documento também serve como guia orientador e lembrete de todos os detalhes do mesmo, facilitando a comunicação das suas ideias aos engenheiros de *software* que irão implementá-las.

[Bethke, 2003] Divide a explicação do conteúdo do documento de especificação conceptual do jogo em diferentes partes sendo que a mesma estratégia utilizada de seguida.

2.1.1.2.1 Definição da jogabilidade central

Nesta secção a discrição base do jogo será expandida, dando espaço a mais detalhes sobre o mesmo. Para começar, é necessário estabelecer a visão base do jogo, será este um jogo 2D ou 3D, que tipo de visão terá o jogador visão de topo, em primeira pessoa, terceira pessoa ou um subconjunto das três, o jogo será passado dentro de quatro paredes ou em espaço aberto e qual será a sua distância de visão. Estes são um exemplo dos aspectos a ter em conta na criação da visão base do jogo.

Posteriormente, é necessário determinar qual será o papel do jogador no jogo em si, será um comandante do um exército, será um assassino a soldo ou outros. É também necessário determinar como este interage com o mundo, como se movimenta e que acções pode realizar. Com as acções do jogador sobre o mundo em mente é útil criar um diagrama do controlador utilizado no mesmo seja este um rato, um teclado ou um *gamepad*, em que tenha a indicação de que cada tecla/botão tenha a sua acção associada.

Sendo adicionalmente necessário determinar qual a informação apresentada ao utilizador, alguns exemplos disso são: a sua pontuação, o número de vidas, o tempo que passou desde o início do jogo ou do nível, o nível, se tem um mini-mapa orientador. Após esse processo é útil criar uma maquete que defina onde e como essa informação é disponibilizada ao utilizador.

2.1.1.2.2 Definição da jogabilidade contextual

Nesta secção irão ser detalhados todos os aspectos da jogabilidade que eram demasiado específicos para estarem na chamada jogabilidade central. Muitos jogos têm submenus onde o jogador poderá criar a sua personagem, escolher o seu equipamento, ir às opções de jogo, entre outros. É necessário então determinar como o utilizador irá navegar sobre os mesmos, para isso é útil criar um diagrama de fluxo para os menus.

Outros aspectos a serem detalhados nesta secção são os intervenientes do jogo, ou seja, as personagens do jogo. É necessário determinar quais são os tipos de personagens que existirão no jogo, quais os seus atributos, ou seja, a sua vida, o seu dano, a sua velocidade e outros. Estes aspectos normalmente são modificados à medida que o jogo é aperfeiçoado.

2.1.1.2.3 Definição da história do jogo

Muitos jogos hoje em dia têm uma história que os enriquece, normalmente existe a história do mundo evolvente e a história de cada personagem. A história do mundo é a definição do mesmo e os eventos que antecedem o início do jogo e os que se desenrolam ao longo do mesmo.

No caso das personagens, para além da sua história passada que normalmente modela a sua personalidade, também é nesta fase que se realiza uma maquete da personagem.

Os níveis, missões ou áreas também fazem parte da história de jogo, estes podem ser por exemplo as masmorras do "*Diablo"* ou as pistas do "*Grande Turismo*". Mas é importante que em todos os casos se detalhe bem o nível em termos de cor, atmosfera, desafio, acontecimentos do mesmo e como é que o jogador navega pelos diferentes níveis ou áreas, (se tem só um caminho ou vários). Com a evolução dos jogos veio-se a optar mais por ter vários caminhos entre as áreas de jogo para poder dar mais liberdade ao jogador.

2.1.1.2.4 Gestão dos recursos do jogo

Os diversos recursos do jogo têm de ser enumerados e organizados, sejam estes os modelos das personagens, níveis/áreas/missões, vozes, música, efeitos sonoros e outros. Isto serve para que no desenvolvimento do jogo nenhum recurso seja esquecido, perdido ou desenquadrado. No caso dos níveis/áreas/missões, é necessário atribuir uma prioridade e hierarquia de desenvolvimento do jogo, pois o primeiro nível deve ser elaborado primeiro que o segundo ou terceiro, e os níveis de bónus devem ser deixadas para último pois estes em caso de falta de tempo podem ser cortados sem que o jogo seja muito afectado, dado estas missões complementares não afectarem o desenvolvimento da narrativa do jogo.

2.1.1.3 Concepção técnica do jogo

Nesta fase é escrito o documento de especificação técnica de jogo. Este será o que os engenheiros de *software* irão usar para desenvolver o jogo, para além de descrever o que é necessário desenvolver, descreve como é para fazer, reduzindo assim a entropia no desenvolvimento do jogo. Este deve ser acompanhado de um bom processo de desenvolvimento por parte da equipa, sejam estes processos ágeis ou não.

O documento passa por várias fases técnicas de desenvolvimento: captura de requisitos, análise de requisitos, arquitectura, plano de desenvolvimento, plano de testes e o plano de transições.

2.1.1.4 Finalização do projecto

Com a finalização do projecto este passa por várias fases começando pelo primeiro jogável, passando pela alfa e pela beta terminando no candidato final.

O primeiro jogável é um grande marco no desenvolvimento de um jogo, pois é possível pela primeira vez jogar o jogo e ver se este é divertido ou precisa de mudanças para o tornar divertido, ou simplesmente necessita que lhe seja adicionado mais elementos ao mesmo.

A alfa é a fase em que os programadores deixam de implementar novos recursos ao jogo, e começam a limpar o seu código. A forma de determinar se o jogo já não necessita de novos recursos é comparando a ultima versão executável do jogo com o documento de especificação conceptual do jogo, mesmo que estes esteja por vezes finalizado, os testadores e a equipa pode achar que o jogo ainda não está suficientemente apelativo e necessitar de novas características.

Entre a fase beta e alfa os desenvolvedores terão de resolver os erros pendentes do jogo tornando-o estável, evitando entrar logo na fase beta, para evitar uma chuva de relatórios de erros por parte da equipa de testes. O uso da equipa de testes deve ser gradual, ou seja, como um jogo na fase de beta ainda tem muitos erros não será necessário uma equipa muito grande para os detectar, isso só iria criar relatórios duplicados, mas à medida que os erros vão desaparecendo torna-se mais difícil encontrar erros e é aí que essa equipa de testes deve aumentar.

Um cuidado a ter na fase de beta é que quando estamos com restrições de tempo é necessário priorizar a resolução de erros, resolvendo os de maior importância e ir descendo nessa hierarquia à medida que os mesmos são resolvidos.

A passagem para a fase de candidato final é uma decisão complexa. Existem vários factores a ter em conta, por exemplo se este ainda tem bugs, se tiver é necessário verificar se estes são relevantes, e decidir se será lançado uma correcção (*patch)* para os resolver. Muitos dos jogos que são lançados dos quais foram lançados *patches*, já eram conhecidos os bugs antes do lançamento. Acontece muitas vezes um jogo ser lançado e pouco tempo depois ser lançado o *patch*, isto é devido ao facto de enquanto o jogo entra

no processo de lançamento, produção da cópia física e distribuição das mesmas, existe tempo necessário para corrigir esses erros.

2.2 Plataforma de desenvolvimento

Uma das escolhas a realizar quando se pretende elaborar um jogo é a plataforma em que este irá ser desenvolvido, o mercado oferece várias soluções desde as de mais baixo nível às de mais alto nível.

De mais baixo nível temos o *DirectX* e o *openGL*, mas este geralmente são usados por empresas que querem criar o seu jogo de raiz e depois vender a licença do seu motor a terceiros como o *UnrealEngine*. A não ser que seja esse o propósito, não é aconselhável seguir este caminho. É possível desenvolver um jogo que assente sobre um motor de jogo como o *UnrealEngine*, mas tal evolve licenças caras e uma complexidade grande na criação do mesmo.

Face à opção de criar um jogo simples, sem o tempo excessivo necessário para aprender a usar, quer o DirectX quer o *UnrealEngine*, então será mais viável usar *APIs* em que o seu uso não implique assim tanto tempo como o *XNA*, *pygame*, *Slick2D*, *Nplay*, e outros.

Como trabalho exploratório foram investigadas as seguintes plataformas o *XNA*, *pygame*, *Slick2D* e *Nplay*, dessas quatro plataformas foram desenvolvidos pequenos protótipos somente em dois, o *XNA* e o *pygame*.

Feitos estes protótipos verificou-se que ambos eram um pouco mais abaixo do nível de abstracção esperado e que a sua dificuldade de uso era muito similar. Embora estivesse mais familiarizado com o C#, linguagem usada no XNA, esse fato também é um factor negativo pois o C# só funciona em ambientes Windows e como este jogo tem por objectivo servir como plataforma para facilitar a aprendizagem de inteligência artificial, convém que este seja disponibilizado para o maior público possível, tenham estes máquinas um ambiente Windows ou não. Com esses fatos em conta optou-se por não usar XNA. Restando uma só alternativa, o *pygame*, mas este com a vantagem da sua linguagem de suporte o *python* ser multiplataforma, ou seja, funciona em vários ambientes.

2.3 Modelação de personagens com base em agentes inteligentes

As personagens do jogo serão controladas por agentes inteligentes neste subcapítulo irá ser explicado o que é um agente inteligente e os mecanismos associados à criação dos mesmos.

2.3.1 Conceito de agente inteligente

Um agente é uma entidade que percepciona o seu ambiente através de sensores e actua sobre esse ambiente através de actuadores.

Uma agente inteligente é um agente que é capaz de adquirir conhecimento e determinar "acção correcta" de acordo com o conhecimento adquirido, ou seja, é um agente racional.

Uma das características de um agente é a reactividade que é a capacidade do agente adquirir e atempadamente responder aos estímulos que advêm do ambiente. De modo a garantir a reactividade o agente terá um limite de tempo para executar as respectivas acções, sendo necessário ter cuidado na criação de sistemas com maior nível de inteligência e de comportamento rico mas que demoram muito tempo a realizar o processamento interno, umas vez que o tempo de processamento elevado compromete a reactividade do agente.

No caso dos jogos é necessário controlar o tempo de reacção para que este não seja demasiado rápido, ou seja, muito mais rápido que um humano. Este tempo de reacção não altera a reactividade do mesmo. Para reduzir o tempo de reacção de um agente é possível controlar o intervalo de tempo entre a obtenção da percepção e raciocínio para determinar a acção. Ou até mesmo o intervalo de tempo entre o raciocínio e o despoletar da acção. Este tipo controlo do tempo de reacção de um agente depende muito do tipo de jogo e da dificuldade do mesmo.

2.3.2 Realismo do sistema

A inteligência artificial (IA) de um jogo tem que tomar decisões e elaborar acções sobre as mesmas de forma inteligente, tal como um humano faria. Mas existe um problema, um computador por si só é demasiado preciso e rápido, por exemplo um jogo em que o computador tem precisão de tiro de 100% e tempo de reacção muito

superiores ao de um humano, não parece muito real. Então é necessário tornar a IA da persongame mais humana ou seja ter falhas e limitações, normalmente estas falhas determinam o nível de dificuldade de um jogo, diminuído a medida que a dificuldade aumenta.

2.3.3 Mecanismos de planeamento

Como referido anteriormente os agentes necessitam da percepção do ambiente que os envolve, para que estes possam efectuar a suas decisões. Como a percepção influencia as acções dos agentes, existe um esforço para que esta seja o mais realística possível. Imaginemos o caso de a uma personagem ser atribuída visão e audição, esta tem de ter limitação, tal como os humanos, limitação de visão a 180º e uma audição que não é perfeita. Então, não deve ser possível a uma personagem detectar uma outra personagem que se aproxime pelas "costas" silenciosamente, isto não seria real, e só aconteceria se este tivesse a informação global do jogo e não só esses dois tipos de percepção.

O sistema de percepção da IA da personagem terá vários tipos e níveis de percepção, é necessário saber quando cada uma é actualizada e como reduzir o impacto computacional de cada uma.

Existem vários tipos de percepção e com tempos de actualização diferentes, as de tipo estático, por exemplo mapa do labirinto que não necessita ser actualizado, ou as do tipo dinâmico, em que tem a posição de cada unidade do mapa, que precisa de ser constantemente actualizada, porque estas movem-se.

Cada nível de percepção pode ter um intervalo de actualização, por exemplo uma unidade que tenha dois tipos de percepção, um em que tem conhecimento do que se encontra perto de si e outro do mapa todo, o primeiro irá ter de ser actualizado mais frequentemente pois é o que mais o influencia, porque este necessita de saber que se um inimigo está próximo dele e se não for actualizada frequentemente pode ser tarde demais quando esta for actualizada novamente.

É necessário ter cuidados na frequência com que as percepções são actualizadas, pois algumas têm um custo computacional elevado, e devemos evitar que estas sejam actualizadas a todos os passos de jogo, devendo ser actualizadas com menos frequência.

2.3.4 Agentes reactivos

Os agentes reactivos determinam a acção a partir de um estímulo com pouco ou nenhum raciocínio. Estes agentes são geralmente caracterizados por um conjunto de comportamentos reactivos como por exemplo, regras de estímulo resposta tal como os reflexos humanos.

2.3.4.1 Modelação de comportamentos com base em máquinas de estados finitos

As máquinas de estados finitos (MEF) é uma das técnicas mais usadas na modelação de personagens para jogos.

Existem dois modelos de máquinas de estados o modelo de Mealy e o de Moore. No modelo de Moore as acções estão associadas aos estados, e no de Mealy estão associados às transições de estado.

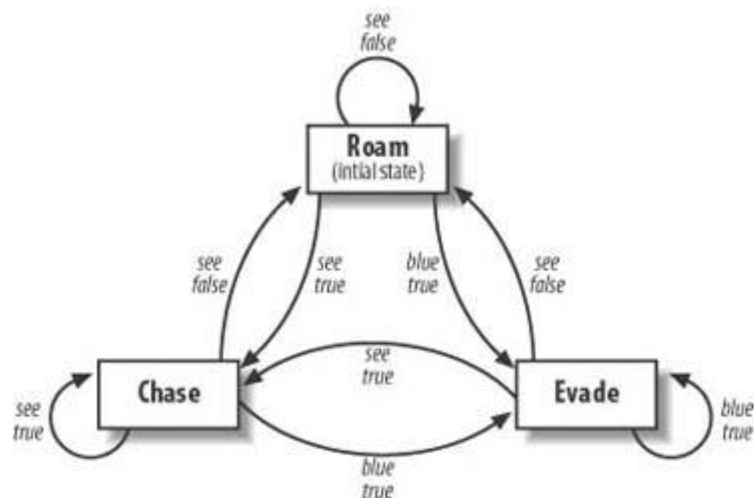

Figura 1 - Maquina de estados do fantasma [Bourg, Seeman, 2004]

Na Figura 1 podemos ver um exemplo de MEF que modela um comportamento similar ao de um fantasma do jogo "*Pac Man"*, como se pode observar este tem três estados perseguir (*Chase*), escapar (*Evade*) e deambular (*Roam*), e duas condições de mudança de estado, se o jogador está visível e se o fantasma está a azul.

O comportamento do fantasma é simples, este persegue o jogador enquanto o vê. Quando deixa de o ver, começa a deambular voltando a persegui-lo quando este voltar a vê-lo, quando o fantasma fica azul significa que este terá de fugir do jogador para evitar ser "comido" pelo jogador e quando deixa de estar azul volta a perseguir o jogador.

Esta solução é elegante pois para criar um fantasma com um comportamento diferente basta mudar um pouco a máquina de estados e sem mais código ter uma unidade com comportamento único, por exemplo o fantasma poderá continuar a seguir pela mesma direcção de onde viu o jogador e não passar logo para o estado de deambular.

Como qualquer técnica, esta técnica tem os seus prós e contras. Os prós são o facto de esta técnica ser de simples implementação após a definição da máquina de estados. É só implementar os estados e as suas transições, é simples de expandir e alterar, o seu teste é simples pois a suas transições bem definidas. Para slém disso, básicas, esta técnica é genérica podendo ser usada em muitos cenários.

Os contras são a demasiada simplificação que pode levar a complicações futuras, à medida que o jogo evolui e mais comportamentos são adicionados a máquina de estados tendendo esta ficar demasiado complexa. Ao longo dos anos foram criadas variantes deste método, para colmatar os seus pontos fracos e melhorar a sua eficiência.

2.3.4.2 Navegação com base em campos de potencial

Na solução baseada em campos de potencial, o processo passa por criar uma grelha para representar o ambiente, onde cada quadrado irá ter um valor que irá repelir ou atrair a personagem. Em certos cenários em que o mundo é estático este pode ser pré-processado levando a soluções óptimas, um exemplo disso encontra-se ilustrado na Figura 2, mas na maioria dos casos este é dinâmico tendo este de ser processado à medida que o jogo avança.

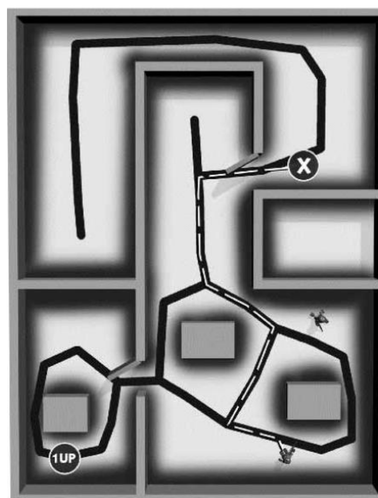

Figura 2 – Exemplo de campos de potencial pré-processado [Schwab, 2004]

Na Figura 2 encontra-se um mapa de jogo com vários quartos e unidades, em que o X representa a posição do agente no mesmo, a preto é possível o campo de potencial gerado, bem como os trajectos de navegação a partir do campo de potencial, para navegar no ambiente

2.3.5 Agentes deliberativos

Os agentes deliberativos têm a capacidade de deliberar sobre as suas acções, ou seja, têm a capacidade de definir antecipadamente um plano de acção (conjunto de acções ou intenções), usando o conhecimento para criar uma solução que consiste num conjunto de acções para a resolução de um determinado problema.

2.3.5.1 Mecanismos procura em espaço de estados pra planeamento automático

Os algoritmos de procura em espaços de estados realizam o processamento com base em estados, operadores. Os estados representam o mundo e todos os seus constituintes num dado momento, os operadores representam as acções que podem ser efectuadas para mudar de estado. O seu funcionamento parte do estado inicial que é inserido na árvore e é expandido, ou seja, são gerados todos os operadores possíveis para esse estado, determinando assim, os estados sucessores do mesmo, fazendo o mesmo para os nós sucessores, até que seja encontrada a solução.

Assim, o planeamento envolve conhecer o estado inicial, estado corrente, o estado final e assim descobrir uma sequência de operadores que leve do estado inicial ao estado final. Na procura de um trajecto os operadores são todos operadores de movimento (andar, usar um portal ou outro tipo de transporte), neste caso existe um grafo no mapa onde será usado por exemplo o algoritmo A* para encontrar o trajecto.

2.3.5.1.1 Navegação com base em grelha

Na solução baseada em grelha, o mundo é dividido em quadrados ou hexágonos formando uma grelha como ilustrado na Figura 3, isto é acompanhado por um algoritmo de procura de trajecto. Geralmente neste tipo de mapa cada quadrado tem uma propriedade que diz se este é um obstáculo ou não, este tipo de mapas contêm pouca informação sobre o mesmo, ou seja, um terreno a subir que é mais difícil de percorrer e

mais lento tem o mesmo valor que um terreno a descer, não permitindo que as decisões tomadas possam ser as melhores, pois pode ser mais rápido dar a volta a montanha do que atravessá-la.

O que normalmente é feito, é tornar este mapa mais rico inserindo propriedades nos quadrados, por exemplo, o declive e o tipo do terreno, isto permite por exemplo inserir unidades que não possam andar em certos tipos de terreno.

O problema deste tipo de sistemas, é o facto da grelha ocupar um grande espaço e tempo de processamento a calcular, sendo também necessário determinar a resolução da mesma. Se esta for muito baixa, perde-se o detalhe, mas se for muito alta, o algoritmo procura de caminho irá demorar muito mais tempo. Existe no entanto, uma técnica para reduzir esses problemas, que é ter uma solução hibrida ou seja ter baixa resolução nas zonas onde o mapa não varia ou varia pouco e alta resolução nas fronteiras ou onde este possa variar mais.

Figura 3 – Exemplo de uma grelha [Schwab, 2004]

Na Figura 3 vemos um exemplo de um mapa em grelhas, dividida por quadrados usada pelo agente para evitar obstáculos, como por exemplo as paredes

2.3.5.1.2 Navegação por pontos de visão

Na solução baseada em grafo de pontos de visão consiste em inserir pontos no mapa de modo a que cada nó do grafo tenha linha de visão para pelo menos um outro ponto, esta inserção normalmente é manual, usualmente feito pelos desenhadores

quando criam o mesmo, em que inserem esses pontos e as conexões entre os mesmos como ilustrado na Figura 4. Por ser manual, este tipo de estratégia pode consumir imenso tempo para mundos complexos.

Esta solução, tal como solução baseada em grelha é acompanhada por um algoritmo procura de trajecto, mas este tem a vantagem de ter um número mais reduzido de estados, permitindo uma procura mais eficiente e rápida. Mas este tem um problema, não lida bem com alvos dinâmicos, sendo a solução para este cenário seria a inserção de um sistema de evitação de obstáculos.

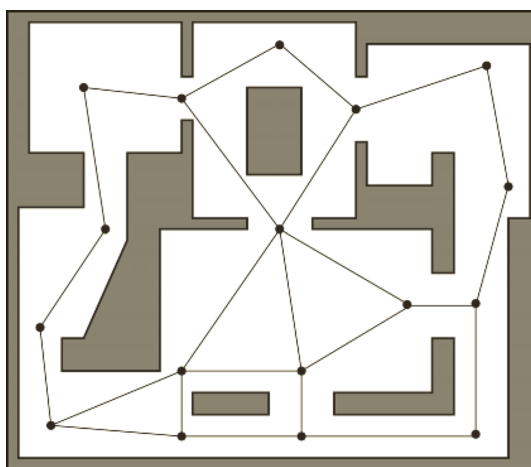

Figura 4 – Exemplo de mapa em grafo [Buckland, 2005]

Na Figura 4 é possível ver um mapa em grafo usando pontos de visão, e é possível ver que os pontos que se encontram no mapa que passam por todas as zonas do mesmo e que têm linha de visão para pelo menos um outro ponto.

2.3.5.1.3 Malha de navegação

Este sistema foi criado de forma a ter as vantagens do sistema em grafo, mas sem se ter a preocupação da criação e manutenção dos mesmos. Este sistema, por sua vez gera automaticamente o grafo a partir dos polígonos do mapa, os quais contêm normalmente um atributo de permissão ou negação de passagem. O grafo criado neste caso difere do anterior pois não consiste em pontos mas em espaços convexos.

No entanto este método tem um problema quando os mapas, por exemplo, têm elevadores, pois não é possível gerar automaticamente as ligações respectivas, tendo o criador do mapa de defini-los manualmente. Deste modo, não faz muito sentido o seu

uso nestes cenários, pois mistura o mecanismo automático e a adição de pontos por parte dos desenhadores.

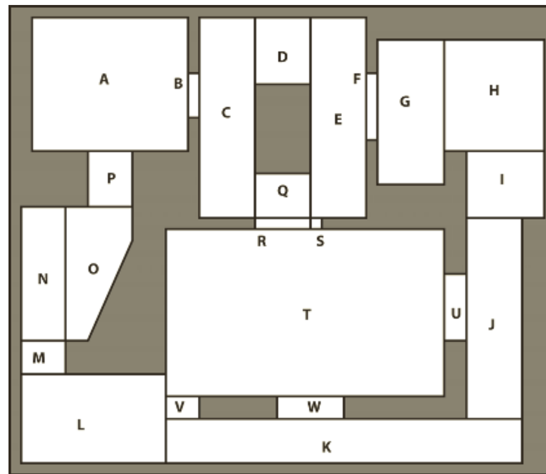

Figura 5 – Exemplo de mapa em malha [Buckland, 2005]

A Figura 5 ilustra um exemplo de um mapa com espaços convexos em que a sua área de jogo navegável se encontra dividida em espaços convexos.

2.3.5.1.4 Mapas de influência

Os mapas de influência (MI) é uma técnica muito usada nos jogos por ser muito genérica e poder ser usada em múltiplos cenários, um MI é uma matriz em que cada posição do mesmo tem a informação relativa a uma determinada posição do mundo [Pottinger,2013]. A resolução do mapa irá depender muito do detalhe que se quer ter sobre a informação do mundo, infelizmente existe um compromisso entre a resolução e a memória utilizada, quanto maior a resolução maior será o matriz do MI e mais memória irá ser utilizada, quanto menor a resolução menor será o matriz e menos memória irá ser utilizada.

Por vezes é necessário ter uma grande resolução. Mas, criar um MI de alta resolução para o mundo todo seria muito dispendioso ao nível da memória, pelo que a solução usual passa por criar vários níveis de MI. Por exemplo, tendo um nível de baixa resolução onde cada posição indica uma área e os recursos que esta tem. Este tipo de mapa, permite a uma personagem saber onde por exemplo criar a sua base principal, para que esta esteja o mais perto do máximo de recursos possível, mantendo alta

resolução em certas áreas onde por exemplo existe uma batalha e é necessário saber onde as unidades se encontram para criar uma estratégia de combate.

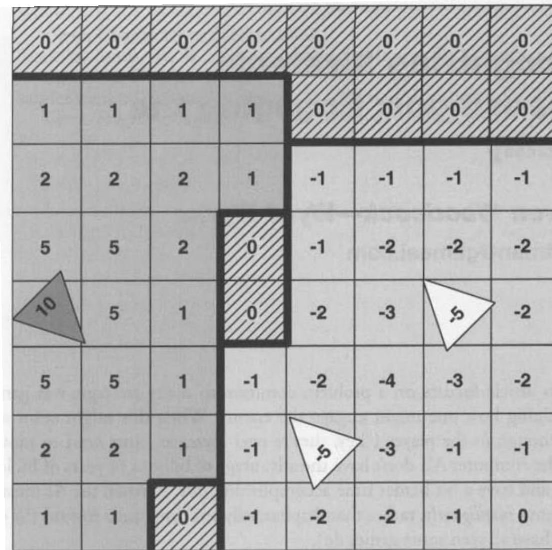

Figura 6 – Mapa de influência dois contra um [Rabin, 2002]

Na Figura 6 está ilustrado um MI de baixa resolução, este ignora qualquer limitação do terreno estando só concentrado na influência de cada unidade. Cada unidade tem um valor que representa a sua influência e esta é espalhada pelo mapa reduzindo-a para metade por cada quadrado de distância do mesmo.

Isto resulta num MI que tem quadrados com valor zero que significa que não pertencem a nenhuma unidade, os quadrados de valor positivo pertencem a unidade aliada e os quadrados com valores negativos às unidades inimigas, daí resultam duas áreas distintas uma a cinzento-escuro pertencente a unidade aliada e outra a cinzento-claro pertencente às unidades inimigas. As linhas a preto definem as fronteiras entre as áreas de domínio.

Este mapa de influência ajuda na decisão, pois informa quais as posições mais perigosas, por exemplo a posição que se encontra entre os dois inimigos tem a influência mais baixa devido ao facto de se encontrarem dois inimigos adjacentes à mesma, e como ambos os inimigos se encontram em pé de igualdade é mais lógico ir de encontro ao que se encontra em baixo pois está mais próximo.

2.3.6 Agentes híbridos

Como o nome sugere os agentes híbridos combinam as características dos dois tipos de agentes anteriores. Um agente reactivo gera respostas rápidas mas baseadas em comportamentos de baixa complexidade. Os agentes deliberativos são dotados de mecanismos de deliberação por vezes complexos, essa deliberação leva o seu tempo, sendo que por vezes é incapaz de dar resposta imediata aos estímulos do exterior.

O agente híbrido contém então dois módulos de decisão o reactivo e o deliberativo, tal como um humano quem tem os seus reflexos que são accionados no caso de existir necessidade de resposta imediata e a sua capacidade de raciocínio para efectuar decisões

A integração destes dois módulos pode ser feita de forma hierárquica em que o sistema reactivo tem prioridade sobre o módulo reactivo, de modo a permitir uma rápida resposta por parte do agente quando é necessário, por exemplo um projéctil que irá atingir o agente.

3 Especificação de requisitos do jogo

Este capítulo apresenta a especificação de requisitos do sistema realizado. Esta especificação de requisitos irá conter a visão do jogo, acompanhada da especificação gráfica do mesmo, os casos de utilização, o modelo de domínio e a especificação complementar.

3.1 Visão de jogo

O jogo consiste na simulação de um ecossistema onde existem várias espécie de seres vivos, nomeadamente espécies animais e vários tipos de espécies vegetais que se dividem em plantas e frutos. O jogo baseia-se nos princípios dos ecossistemas, em que domina o princípio da sobrevivência do mais forte na cadeia alimentar, ou seja, alimentar-se e não servir de alimento e assim sobreviver. Para atingir esse objectivo nomeadamente a sobrevivência, os animais têm de se alimentar de outros animais, ou seja, cada espécie tem os seus predadores e/ou presas, para além disso também podem comer espécies vegetais.

Os animais terão uma barra de fome que enche à medida do tempo e diminui quando este come. No mundo do jogo existirão vários tipos de terrenos: vegetação, terra, água, floresta e rochedos. Cada espécie poderá andar por certos tipos de terreno. Os animais, plantas e frutas após serem comidos, terão um tempo de reaparecimento e uma zona para o fazer, estas zonas serão denominadas de zonas férteis, por exemplo zona de vegetação, existirão também zonas inférteis onde não é possível reaparecer, por exemplo, zona onde há terra.

O jogador irá assumir o papel de um animal no jogo, esse poderá ser jogado por um humano ou por um agente autónomo, a pontuação do jogador é determinada por aquilo que come, quando tal acontece recebe pontos e quando estes pontos chegam a um determinado valor, o jogo passa de nível, a mudança de nível envolve a mudança de mapa e das espécies, bem como a mudança do animal do jogador. Contudo, o jogador perde o jogo quando este é comido, não reaparecendo.

O jogo terá de permitir pausar e resumir o jogo, reiniciar o nível e iniciar um novo jogo.

O jogo irá ter de apresentar a pontuação do jogador, o mapa de jogo e as espécies do mesmo, e os dados da espécie seleccionada, que são: nome, imagem, fome e velocidade. Essa informação estará apresentada como indicado na maquete presente na Figura 7.

A IA das personagens deve ser implementada de modo a que esta seja simples e extensível, permitindo uso fácil da mesma para quem queira aprender sobre a mesma.

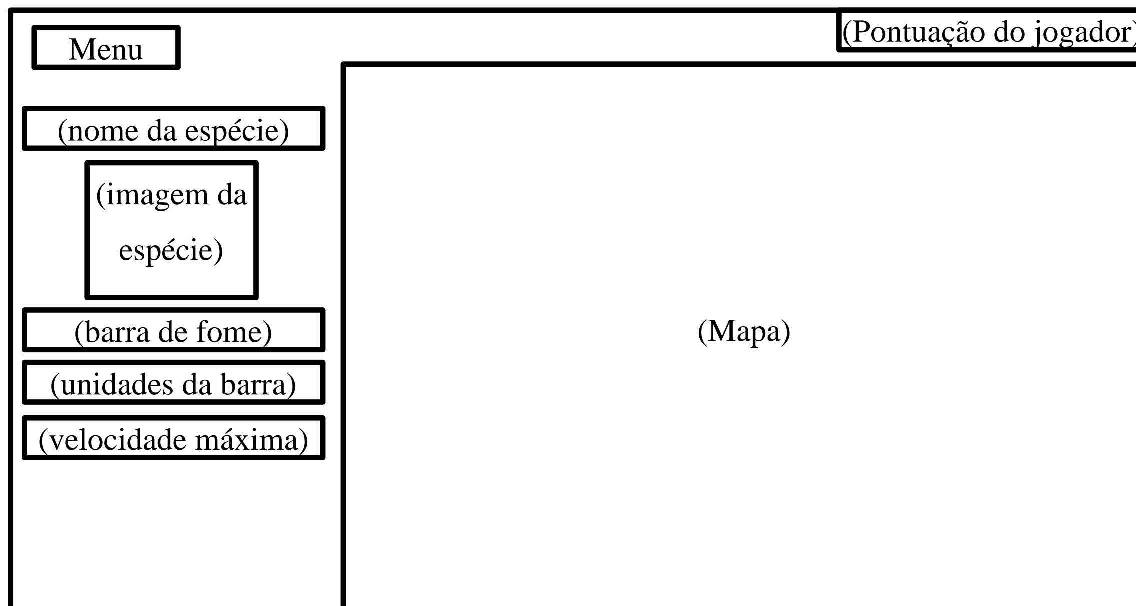

Figura 7 – Maquete da interface gráfica

3.2 Casos de utilização

Com base no texto da visão do jogo foi elaborado o diagrama de casos de utilização presentes na Figura 8, onde estão presentes as funcionalidades principais do sistema de jogo.

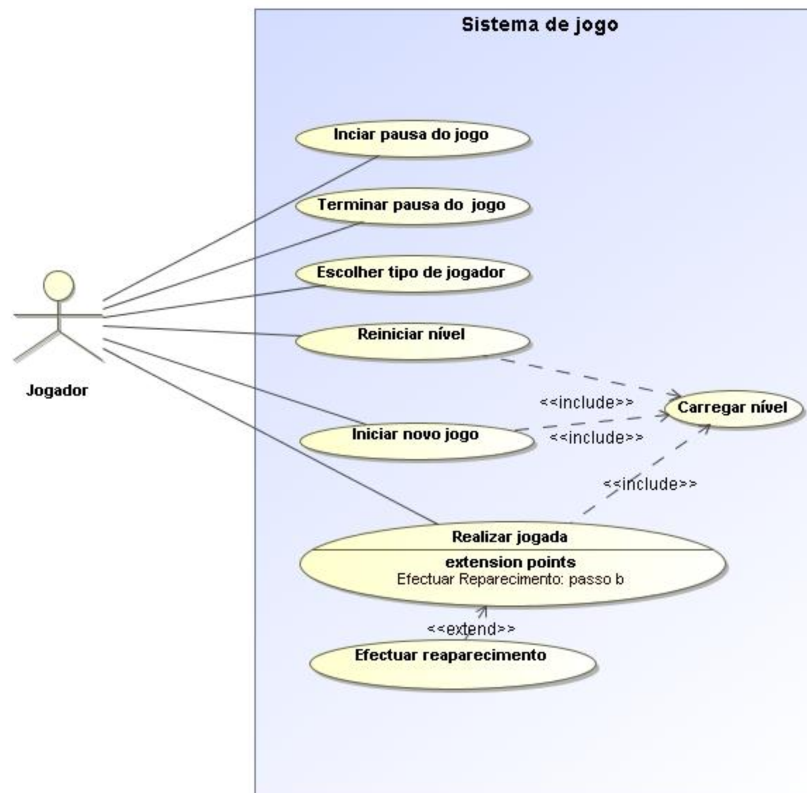

Figura 8 – Diagrama de casos de utilização

3.2.1 Descrição dos casos de utilização

Com base no diagrama de casos de utilização foram realizadas as descrições dos mesmos.

Nome	Iniciar pausa do jogo
Resumo	Permite o jogador iniciar a pausa do jogo
Actor	Jogador
Cenário Principal	1. O caso inicia-se quando o jogador pretende iniciar a pausa o jogo 2. O jogador carrega no botão de iniciar pausa 3. As unidades do jogo deixam de se mover

Nome	Iniciar pausa do jogo
Resumo	Permite o jogador iniciar a terminar a pausa do jogo
Actor	Jogador
Cenário Principal	1. O caso inicia-se quando o jogador pretende terminar a pausa o jogo 2. O jogador carrega no botão de terminar a pausa 3. As unidades do jogo deixam de se mover

Nome	Escolher o tipo de jogador
Resumo	Permite o jogador escolher o tipo de jogador que irá jogar
Actor	Jogador
Cenário Principal	1. O caso inicia-se quando o jogador pretende escolher o tipo de jogador 2. O jogador escolhe o tipo de jogador humano 3. O jogador indica ao sistema que o personagem respectiva irá ser controlada por um humano
Cenário alternativo 1	1. No passo 2 do cenário principal o jogador escolhe o tipo de jogador automático 2. O jogador indica ao sistema que o animal respectivo irá ser controlado por um agente

Nome	Reiniciar nível
Resumo	Permite o jogador reiniciar o nível corrente do jogo
Actor	Jogador
Cenário Principal	1. O caso inicia-se quando o jogador pretende reiniciar o nível do jogo 2. O jogador carrega no botão de reiniciar o nível 3. Os pontos voltam ao valor que estavam no início desse nível 4. O sistema indica que o nível corrente necessita de ser carregado 5. Incluir Carregar nível

Nome	Iniciar novo jogo
Resumo	Permite o jogador iniciar um novo jogo
Actor	Jogador
Cenário Principal	1. O caso inicia-se quando o jogador pretende iniciar um novo jogo 2. O jogador carrega no botão de iniciar novo jogo 3. Os pontos são postos a zero 4. O sistema indica que o primeiro nível necessita de ser carregado 5. Incluir Carregar nível

Nome	Realizar Jogada
Resumo	Permite o jogador realizar uma jogada
Actor	Jogador
Pontos de extensão	Cenário alternativo 2, passo 3: efectuar reaparecimento
Cenário Principal	1. O caso inicia-se quando o jogador pretende realizar uma jogada 2. O jogador indica o movimento que quer realizar 3. A posição do seu animal é actualizada
Cenário alternativo 1	1. No passo 3 do cenário principal ao actualizar a posição a personagem colide com o predador 2. O jogo termina
Cenário alternativo 2	1. No passo 3 do cenário principal ao actualizar a posição a personagem colide com uma presa 2. O sistema reduz a fome do animal 3. O sistema mata a presa 4. Aumenta os pontos do jogador
Cenário alternativo 3	1. No passo 4 do cenário alternativo 2 ao actualizar os pontos do jogador, os pontos atingirem a o valor necessário para a passagem para o próximo nível 2. O sistema indica o próximo nível como nível a ser carregado 3. Incluir carregamento de nível

Nome	Carregar nível
Resumo	Permite carregar um dado nível quando este é indicado para tal
Actor	
Cenário Principal	1. O caso inicia-se quando um nível é indicado para carregamento 2. São carregados todas as unidades e o mapa constituintes desse nível 3. As unidades e o mapa substituem os existentes no jogo

Nome	Efectuar reaparecimento
Resumo	Permite carregar o reaparecimento de uma unidade, passado um determinado tempo após ser morta
Actor	
Cenário Principal	1. O caso inicia-se quando uma unidade é morta 2. É esperado o tempo de reaparecimento estipulado 3. É determinado o sítio onde esta irá reaparecer 4. Esta unidade é colocada no sítio

3.3 Modelo de domínio

Após a elaboração do texto de visão de jogo e dos casos de utilização foram obtidos os principais elementos constituintes do mesmo e foi realizado o diagrama de domínio apresentado na Figura 9.

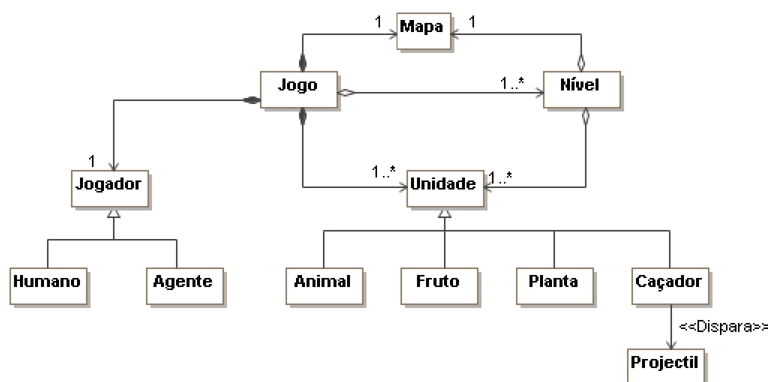

Figura 9 – Modelo de domínio

3.4 Especificação suplementar

Número do requisito	Descrição do requisito
1	O jogo contém: 1.1. Animais; 1.2. Plantas; 1.3. Frutos; 1.4. Caçadores.
2	Os caçadores disparam projecteis
3	Os animais e caçadores perseguem presas e fogem dos predadores
4	Os animais têm uma barra de fome que diminui ao longo do tempo
5	O jogo contém vários tipos de zonas que são: 5.1. Vegetação; 5.2. Terra; 5.3. Água; 5.4. Floresta; 5.5. Rochedos.
6	Em cada área só podem andar certos tipos de animais: 6.1. Animais aquáticos só podem andar sobre água 6.2. Animais terrestres só podem andar sobre vegetação e terra 6.3. Animais anfíbios só podem andar sobre água, vegetação e terra 6.4. Animais aéreos podem andar em qualquer tipo de terreno
7	Zonas férteis são: 7.1. Vegetação; 7.2. Água.

8	Zonas não férteis são: 8.1. Terra; 8.2. Floresta; 8.3. Rochedos;
9	As espécies devem ter como atributos: 9.1. A sua fome; 9.2. Unidade que alimenta; 9.3. Nome; 9.4. Imagem; 9.5. Ícone; 9.6. Velocidade.
10	A informação de jogo será apresentada de acordo com a maquete
11	A IA das personagens deve ser simples de usar e extensível
12	A execução do jogo não pode exceder o tempo do ciclo de jogo
13	As balas disparadas são destruídas após impacto
14	As personagens não podem movimentar-se para lá do limite do mapa de jogo

3.5 Protótipo de requisitos

Para perceber melhor os requisitos, foi elaborado um protótipo de requisitos. Um protótipo de requisitos é uma implementação parcial do sistema, para ajudar os elementos da equipa de desenvolvimento, utilizadores e clientes a perceberem os requisitos do sistema. Este deve ser o mais simples possível, gastando o menor número de recursos possíveis.

Existem vários tipos de protótipos de requisitos, os que implementam várias funcionalidades, os que implementam poucas funcionalidades, (mas já com boa qualidade) e os de interface gráfica [Leffingwell, Widrig, 1999].

O protótipo elaborado neste caso, não implementa uma funcionalidade em si, mas um protótipo de validação da interface gráfica e do conceito do jogo. Para isso, foi

usada a aplicação Torque, esta aplicação serve para criar jogos rapidamente e com recurso a poucas linhas de código.

Figura 10 - Aspecto gráfico do protótipo

Na Figura 10 encontra-se o resultado do protótipo do jogo, onde se encontra o mapa de jogo, a informação apresentada ao jogador, as personagens incluindo a personagem controlada pelo jogador. Tendo a personagem controlada pelo jogador, foi possível verificar qual seria a melhor forma de interacção do jogador com a personagem e a forma como esta se movimenta e interage com o ambiente.

Este, também serviu para testar algumas regras de jogo e determinar como os animais se moviam, e assim testar se o conceito do jogo era válido e se os requisitos faziam sentido e as possíveis melhorias, a adicionar ao mesmo.

4 Concepção da solução

Neste capítulo irá ser explicado o processo de realização dos casos de utilização, ou seja, utilizar os casos de utilização e trabalhá-los, explorando-os e determinando assim os mecanismos necessários para os implementar, criando um ponto de partida para a criação da arquitectura da solução, que será discutida no capítulo seguinte.

4.1 Definição do nível de jogo

Como qualquer jogo, este tem de ter um nível ou um ambiente onde este decorre, com base na especificação do jogo o nível de jogo tem os seguintes constituintes: as entidades que representam os animais, plantas, frutas e os caçadores; as áreas que representam os terrenos do jogo e o mapa de jogo. As duas primeiras são simples objectos de jogo, mas no caso do mapa existem diversos tipos de mapas de jogos que influenciam a maneira como o jogo é construído e a interacção com o mesmo.

Desse problema surgiram várias soluções que irão ser discutidas de seguida, analisando os seus pontos fortes e pontos fracos.

A solução inicial passou pela possibilidade do mapa se tratar de uma imagem, esta constituída por áreas de várias cores, cada cor representando um tipo de terreno, por exemplo, cor verde para a área de vegetação, facilitando assim, um possível algoritmo de mapeamento do mapa para uma representação interna. Mas, esta solução tornaria o jogo pouco atractivo pois seria muito fraco em termos gráficos.

Com esses factos em conta, houve a tentativa de arranjar uma solução que tornasse o jogo mais atractivo. A solução seguinte seria melhorar o aspecto da imagem em si, ou seja, em vez de uma área ter uma cor esta seria uma imagem, por exemplo, a área de vegetação teria uma imagem de vegetação, mas logo se percebeu que a passagem do mapa para uma representação interna seria algo complexa, sendo que o esforço para implementar a solução seria demasiado elevado para o objectivo a ser alcançado.

Foi então pensada uma solução que pudesse ser mais equilibrada, ou seja, ser agradável visualmente e ter um mapeamento simples. Após alguma pesquisa optou-se por um *tiledMap*, ou seja, mapa constituído por *tiles* (pequenos azulejos), mapa este que é de simples processamento e com um aspecto gráfico agradável.

4.2 Ciclo de jogo

Como referido anteriormente, num jogo existe sempre um ciclo principal, que se encarrega de gerir o compasso do jogo, seja em termos gráficos ou da acção do mesmo. Isto envolve a gestão de vários elementos do jogo e de várias acções do mesmo. Na Figura 11 é apresentado o diagrama de sequência que descreve a sequência de acções que ocorrem durante o ciclo de jogo ao nível do motor de jogo.

Durante o ciclo de jogo, o *looper*, objecto responsável pelo ciclo principal do jogo, começa por obter os objectos de jogo que são as entidades e a barra de menu. Após obter os objectos de jogo, é necessário gerar os eventos para os mesmos que são os seguintes:

- Evento de botão esquerdo do rato pressionado – é accionado quando o botão esquerdo do rato é pressionado;
- Evento de colisão – é accionado quando um objecto de jogo colide com uma área ou uma unidade;
- Evento de tecla pressionada – é accionado quando tecla é pressionada
- Evento de passo – é accionado a cada compasso de jogo.

É de notar que neste diagrama não se encontra o evento de colisão pois este encontra-se englobado na geração do evento de passo, pois é após o passo, ou seja, existência de movimento das unidades, que irá existir colisões.

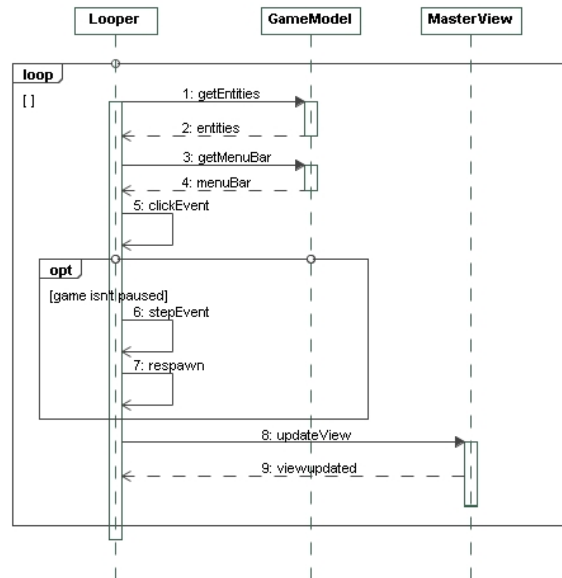

Figura 11 - Diagrama de sequência do ciclo de jogo

A geração de eventos começa pela geração do evento de clique e de tecla, que gera os eventos para ambos os tipos de objectos de jogo. Em seguida, é necessário verificar se o jogo está em pausa, pois não existe passo enquanto o jogo está parado. Não estando em pausa é necessário gerar o evento de passo somente para as entidades, pois são estas que se movimentam. Logo após, é necessário fazer reaparecer as entidades que já se encontram novamente vivas. Após estas alterações nos dados é necessário reflecti-las na interface gráfica, actualizando a vista.

4.2.1 Reaparecimento de unidades

Quando uma unidade é morta, esta tem um tempo até voltar à vida e voltar a ser colocada de volta no mundo de jogo. Quando uma entidade é morta ela é indicada como estando morta e é-lhe atribuído um tempo de reaparecimento, que quando chega ao fim, esta volta à vida.

Para isso, é efectuado a sequência de acção ilustrada na Figura 12. Quando o tempo acaba é necessário então gerar a nova posição no mundo onde a unidade irá reaparecer de forma aleatória, após gerar a nova posição é necessário verificar as colisões nessa nova posição, pois nesta nova posição ela poderá estar em colisão com outra unidade, ou com alguma área. No caso da área o caso é mais complexo, pois a

entidade não pode reaparecer numa área infértil, ou que esta não possa atravessar, por exemplo uma unidade aquática não pode reaparecer numa floresta.

Existindo uma colisão então é novamente gerada uma nova posição e verificadas as colisões, processo que se repete até que não existam colisões, e nesse caso essa entidade volta a vida. Se o tempo não terminou, este é simplesmente decrementado.

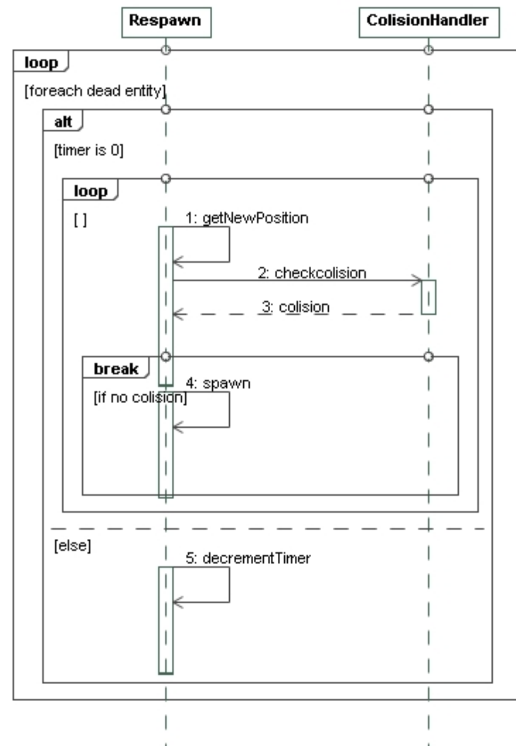

Figura 12 - Diagrama de sequência do reaparecimento das unidades

4.2.2 Actualização da vista

A actualização da vista divide-se em duas partes, o carregamento da actualização do mapa de jogo e da actualização dos restantes elementos dinâmicos. Como o mapa, que é constituído pelos *tiles*, não se altera durante o percorrer de um nível e é só actualizado quando um novo nível é carregado, então não existe necessidade de o actualizar a cada ciclo de jogo.

Pelo contrário, os elementos dinâmicos, que são as entidades, o menu e a barra lateral alteram-se durante o decorrer do nível necessitando assim de ser actualizados a cada ciclo.

Esta estratégia permite reduzir o custo computacional de actualizar a vista, pois que só actualiza os elementos que necessitam de ser actualizados retirando o custo de actualização dos restantes

4.3 Eventos das entidades

O comportamento das unidades como referido anteriormente é definido, pelo que estas fazem nos seus eventos. Neste trabalho, esse comportamento é definido da seguinte forma, o comportamento das entidades de personagem, que são móveis e podem ser usadas por um jogador, e o comportamento das entidades inanimadas, sendo que nestas existe uma variação de comportamento para as entidades inanimadas móveis e para o comportamento da barra de menu.

4.3.1 Eventos das entidades personagem

As entidades personagem são as que têm mais lógica pois são estas que representam os animais e o caçador do jogo, e são estes que se movimentam, têm fome, e outros. Sendo então nestes, que assentam algumas das regras que definem o jogo. Em seguida irá ser descrito a sequência de acção para este tipo de entidade e para os eventos relevantes.

4.3.1.1 Evento de passo

No evento de passo a entidade necessita de decidir qual a sua próxima acção a realizar. Para isso, elabora a sequência de acções ilustrada na Figura 13. Em primeiro lugar necessita de recorrer ao seu controlador, o controlador por usa vez determina a sua próxima acção e devolve-a de seguida.

Ao receber a acção do controlador, este verifica qual o seu tipo, neste caso só existem dois, o mover e o disparar e efectua essa acção passando o vector recebido pelo controlador.

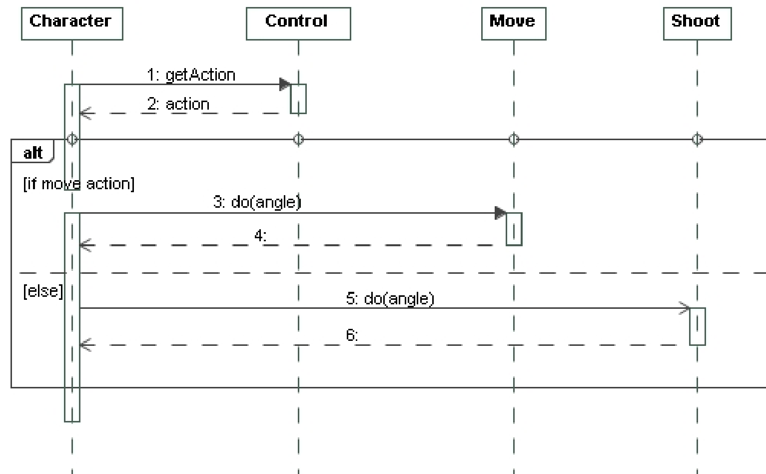

Figura 13 - Diagrama de sequência do evento de step das personagens

4.3.1.2 Evento de colisão

Quando uma entidade personagem colide com outra entidade, é necessário saber se esta é predadora, de forma a saber se a pode comer, nesse caso a unidade colidida é morta. Ao comer uma entidade, a sua fome é reduzida conforme as unidades que a unidade ingerida alimenta. No caso da unidade que come ser do jogador, a sua pontuação também é actualizada conforme as mesmas unidades. Esta sequência de acções encontra-se ilustrada na Figura 14.

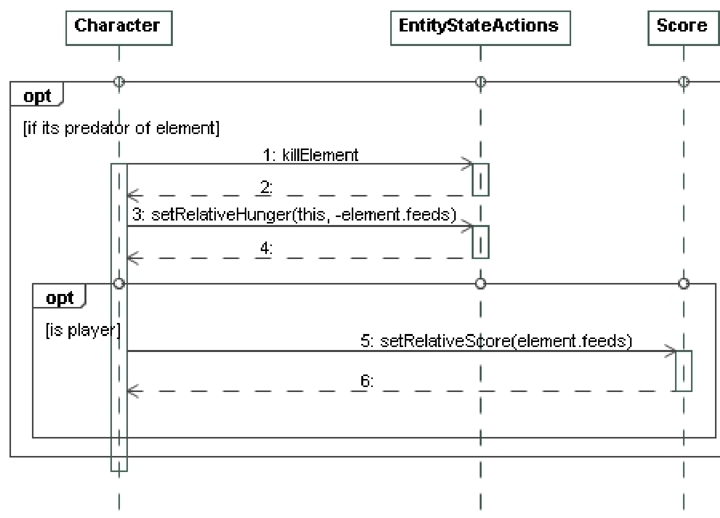

Figura 14 - Diagrama de sequência do evento de colisão das personagens

4.3.2 Eventos das entidades inanimadas

Os eventos das unidades não animadas são pouco ricos, ou seja, estas não têm um comportamento complexo, sendo que nestas só dois dos eventos são relevantes, o de colisão e o de passo (step) para as unidades inanimadas móveis.

O evento de colisão, evoca o evento colisão do dono dessa entidade, ou seja, certas entidades irão ter um dono tal como a bala que é disparada por um caçador, em que o seu dono é o caçador e quando a bala colide com uma entidade, esta não irá comer a entidade mas sim o seu caçador. Ao colidir com uma entidade, a bala irá destruir-se, ou seja, vai matar-se a si própria, mas esta não irá reaparecer como as restantes unidades.

O evento passo só funciona para as unidades móveis que se movimentam num padrão, tal como a bala que tem um padrão de movimento rectilíneo.

Esta abordagem permite que sejam criados novos objectos com donos, sejam eles móveis ou não, como por exemplo armadilhas, que não são móveis.

4.4 Controlador das entidades

Como referido anteriormente o controlador das unidades determina a acção a ser realizada. Este divide-se em dois, o controlador de humano e o controlador de IA, que se subdivide em controlador reactivo e em controlo deliberativo.

O controlador humano é muito simples, obtém as teclas de movimento que foram carregadas desde o último ciclo e determina a acção a ser efectuada através das mesmas.

Os controladores de IA começam por actualizar a sua percepção, pois necessitam da mesma para efectuar decisões. O controlador reactivo simplesmente, irá usar a percepção para determinar se a sua entidade mais próxima é predador ou presa/alimento, afastando-se ou aproximando-se das mesmas, pois este mecanismo é só percepção e resposta, sem deliberação tal como os reflexos de um humano.

O controlador deliberativo é mais complexo, pois este tem a capacidade de deliberar sobre as acções que irá realizar e para isso irá usar um mecanismo de procura em espaços de estados (*SSSMecanism*).

A criação desse plano e a geração da acção a realizar encontra-se ilustrada na Figura 15.O controlo este começa por obter a entidade mais próxima da entidade controlada e em seguida gera os operadores de entidade controlada.

Em seguida, é necessário criar um objectivo, existem dois tipos de objectivo, objectivo atractivo (*SSSObjective*), ou seja, uma entidade da qual se quer aproximar, por exemplo uma presa/alimento e o objectivo repulsivo (*RepulsiveObjective*), ou seja, uma entidade da qual se quer afastar, por exemplo um predador, assim é verificado o tipo da entidade mais próxima e é criado o objectivo correspondente.

Para finalizar, após ter os operadores e o objectivo, estes são usados para criar um plano com uso ao mecanismo de procura em espaço de estados e a partir do plano obter a próxima acção a ser realizada.

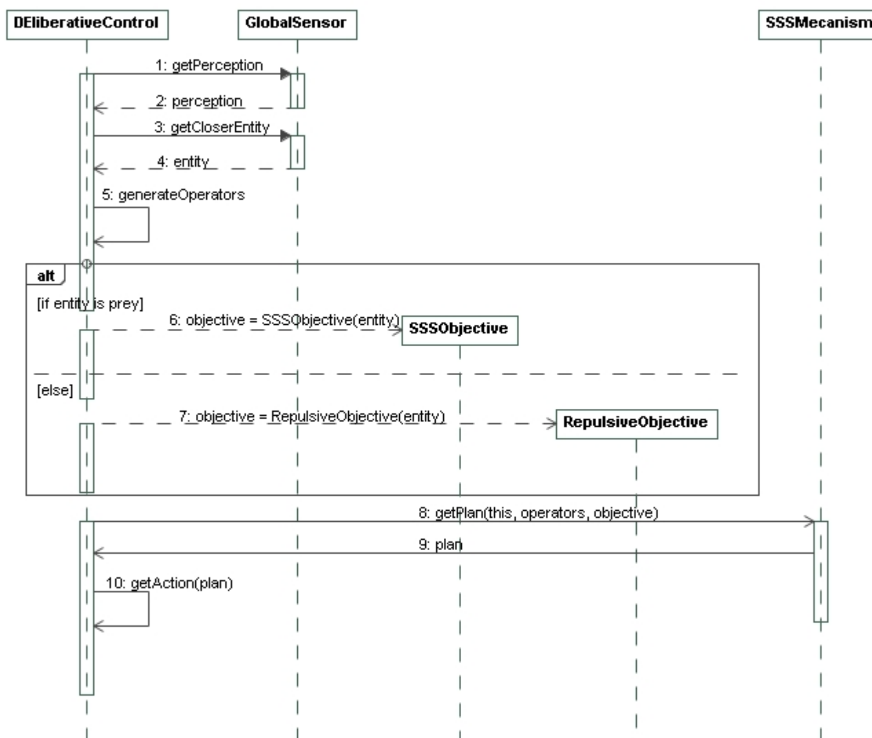

Figura 15 - Diagrama de sequência de controlador deliberativo

4.5 Carregamento dinâmico de níveis

O jogo em si terá vários níveis, após alguma reflexão chegou-se à conclusão que seria vantajoso permitir a criação de novos mapas após o término do projecto, a criação de tais níveis permitiria ao utilizador criar mapas para testar as suas implementações em certos cenário, por exemplo para testes de algoritmos reactivos este poderia começar por criar um nível em que não existissem predadores e existissem somente alimentos que são estáticos, testando assim o seu algoritmo sem entidades móveis para além da personagem utilizada.

Para tal seria necessário criar carregamento de níveis dinâmico e num mecanismo simples de criação de novos níveis.

A solução inicial passou por criar um ficheiro com uma classe que derivasse de uma interface que continha uma matriz bidimensional com os códigos do *tile* de cada posição e uma lista de entidades do jogo e as respectivas posições, mas esta solução para além de morosa é um pouco ingrata. Pois, para fazer um nível o seu criador, teria de saber os códigos do *tile* que queria para cada posição, algo que dificulta imenso a definição de um novo nível, o que levaria a que o mecanismo fosse pouco ou nada utilizado.

Após alguma pesquisa foi encontrada uma ferramenta de criação de *tiledMaps* denominada *Tiled*[Tiled], esta ferramenta é de simples utilização, criando um *tiledMap* e os objectos de jogo devidamente colocados nas suas posições, guardando-o num formato proprietário *.tmx que a aplicação terá de interpretar.

Tendo o mapa criado é necessário obter o nível de jogo a partir do ficheiro, após nova pesquisa foi encontrada uma API elaborada em *python* com recurso ao *pygame*, que interpreta esse tipo de ficheiro denominada *TmxLoader*.

O carregamento de um nível começa pela chamada da *API* do TmxLoader indicando o nome do ficheiro, que por sua vez devolve um objecto no qual se encontram os *tiles* e os objectos de jogo, que por sua vez é usado para carregar os *tiles* e os diversos objectos do jogo para a plataforma.

O carregamento dos *tiles* encontra-se ilustrado na Figura 16, cada tile começa por obter a imagem e a informação respectiva e usa-a para criar a *sprite*, usando a *sprite* para criar o próprio *tile* e adiciona-o à colecção de *tiles* do mapa.

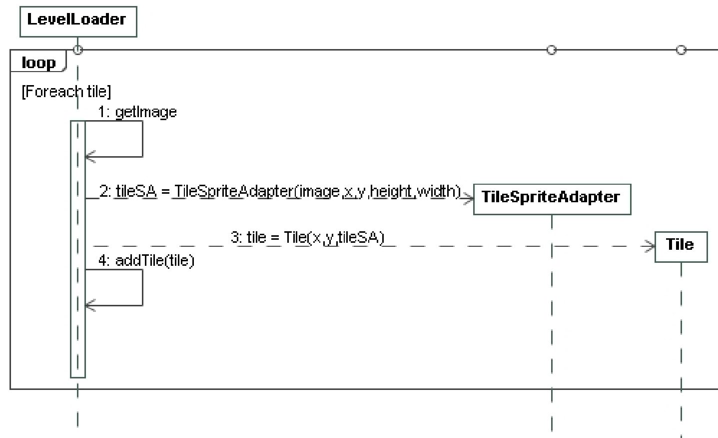

Figura 16 - Diagrama de sequência do carregamento dos tiles

O carregamento dos objectos de jogo é mais complexo, pois existem três tipos de objectos de jogo, as áreas, a pontuação, e as entidades que são identificadas pelos seus atributos.

O carregamento é então dividido em três como ilustrado na Figura 17. No caso de um objecto de jogo ter o atributo *point*, este significa que é uma área pois a área é um polígono que é representado por pontos, em seguida são obtidos esses mesmos pontos e o tipo da área e esta é então criada.

O carregamento da pontuação é simples, bastando identificar se o objecto é do tipo Score (pontuação), esta pontuação representa a pontuação necessária para a passagem de nível, bastando obter o valor do objecto.

No carregamento das entidades, começa-se por obter o seu nome e a partir do mesmo é obtido o seu construtor e é criada a entidade e em seguida é posta na sua posição.

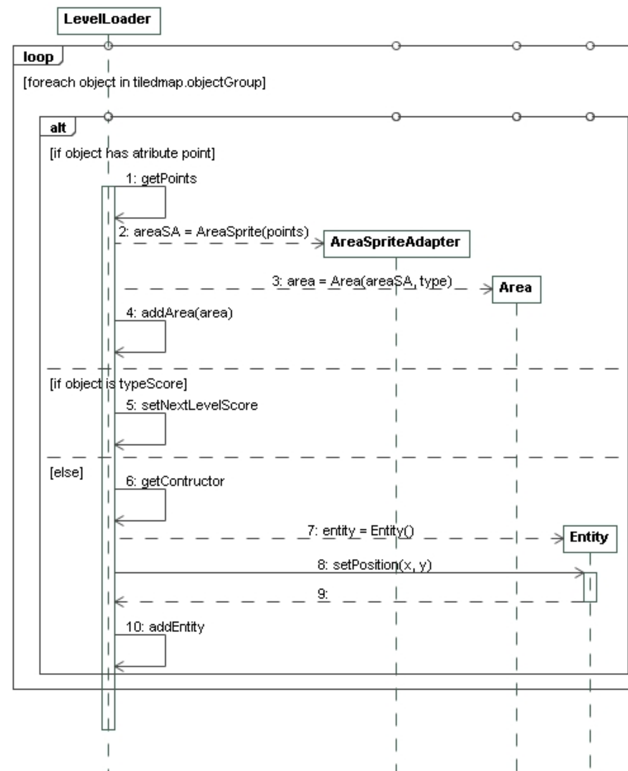

Figura 17 - Diagrama de sequência do carregamento dos objectos de jogo

4.6 Comportamento das personagens

O comportamento das personagens encontra-se definido sobre a forma de máquinas de estados finitos (MEF), sendo assim necessário definir os seus estados e as suas transições na forma de um diagrama de máquina de estados finitos.

A solução poderia passar por criar um comportamento individual para cada tipo de personagem, ou seja, o urso ter um comportamento diferente do coelho. Sendo o urso mas agressivo e o coelho mais esquivo, embora isto pudesse trazer mais riqueza ao jogo o esforço seria demasiado grande para o benefício que traria. Com esses factos em conta optou-se por criar três tipos de comportamento, um para os predadores, um para as presas e outro para o caçador.

Isto é devido ao facto de todos os predadores terem o mesmo tipo de comportamento, perseguir as presas e evitar predadores, as presas por seu lado terem como comportamento, buscar alimento e evitar predadores e o caçador perseguir as presas até que eles estejam no seu alcance e disparar, como este não tem predador não necessita de evitar os mesmos.

Na Figura 18 encontra-se ilustrado o comportamento dos predadores, estes são caracterizados por três estados, o estado de deambular em que o animal anda aleatoriamente, o estado de perseguir em que este persegue a presa e o estado de escapar em que o animal escapa do seu predador.

No esquema, podemos ver que o predador quando fica com fome, passa para o estado de perseguição para perseguir a sua presa e satisfazer a sua fome, podemos ver que o estado de escapar tem conexão de todos os outros estados, isto porque o objectivo máximo do predador é sobreviver, ou seja, não ser comido sendo essa a sua prioridade principal. Ao ver esta troca entre o perseguir e o escapar, surge uma ideia para poder enriquecer o comportamento das personagens, por exemplo quando um predador está quase a morrer de fome o seu comportamento pode ser modificado para que arrisque mais e tente comer a presa mesmo com o seu predador próximo, pois se este não comer num curto espaço de tempo irá morrer logo não existe grande vantagem em evitar o predador e morrer de fome.

Um aspecto a ter em conta é o facto do estado de deambular existir, pois o predador poderia passar o tempo a perseguir a sua presa mesmo que não necessitasse de se alimentar, ou mesmo escapar do predador quando não encontra presas. Mas este comportamento não seria muito real pois um animal não persegue a sua presa quando não necessita, nem tão pouco evita o predador quando este não está próximo, pois gasta energia ao fazê-lo e ao deambular pelo mundo pode então explorá-lo e assim obter mais informações sobre o mesmo de modo a poder tomar melhores decisões.

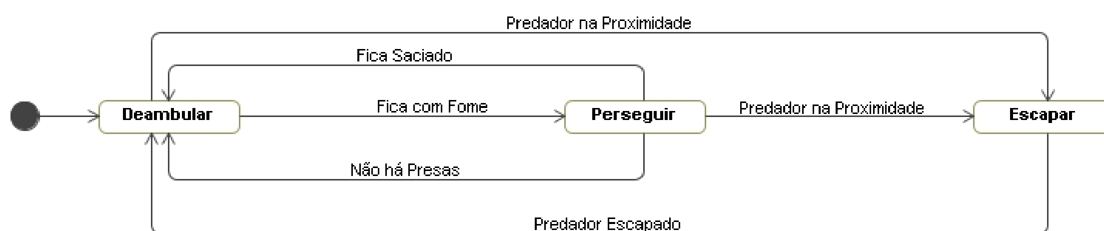

Figura 18 – Especificação do comportamento do predador

O comportamento da presa ilustrada na Figura 19 é praticamente idêntico ao do predador, visto que estes têm o mesmo princípio de sobreviver que implica fugir de

predador, só que no caso das presas estas não perseguem outros animais, em vez disso buscam o seu alimento, que ao contrário das presas é imóvel, não existindo perseguição.

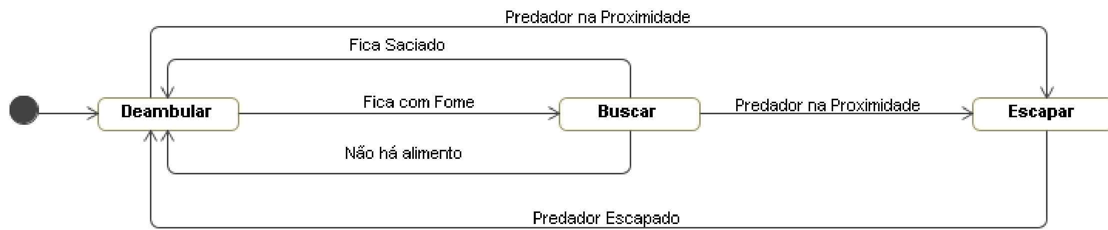

Figura 19 - Especificação do comportamento da presa

Na Figura 20 encontra-se ilustrada o diagrama de transição de estados correspondente ao caçador, como podemos ver o caçador não tem o estado de escapar, isto acontece porque o caçador não tem predadores, o perseguir foi substituído pelo aproximar pois o caçador não precisa de entrar em contacto directo com a presa para a comer, em vez disso este tem de se aproximar para quando a presa estiver ao seu alcance, disparar.

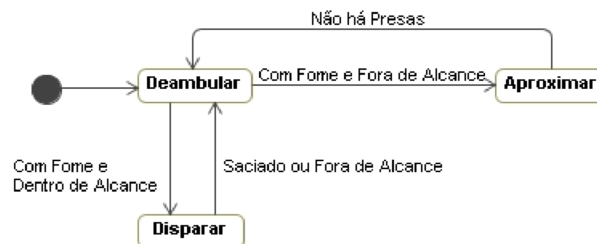

Figura 20 – Especificação do comportamento do caçador

Após a definição do comportamento das personagens é necessário estabelecer como estes comportamentos irão ser suportados pela MEF e sua ligação com os controladores existentes.

A interligação com os controladores é feita através do controlador MEF, que quando activado despoleta a actualização da máquina de estados, estando o seu comportamento ilustrado na Figura 21. Este começa por obter a identificação do estado corrente, em seguida verifica se existem transições de estado, as quais devolvem a

identificação do estado objectivo e em seguida verifica se estas identificações são diferentes, no caso de o serem, significa que existe uma transição de estado.

No caso de existir uma transição, é necessário verificar se o estado existe mesmo. No caso de existir, é necessário sair do estado corrente, em seguida o novo estado passa a ser o corrente e entra-se no novo estado. No caso de não existir o estado, o estado corrente mantém-se. No final o estado é actualizado e devolve a acção correspondente ao mesmo, que por sua vez é devolvido ao controlador MEF.

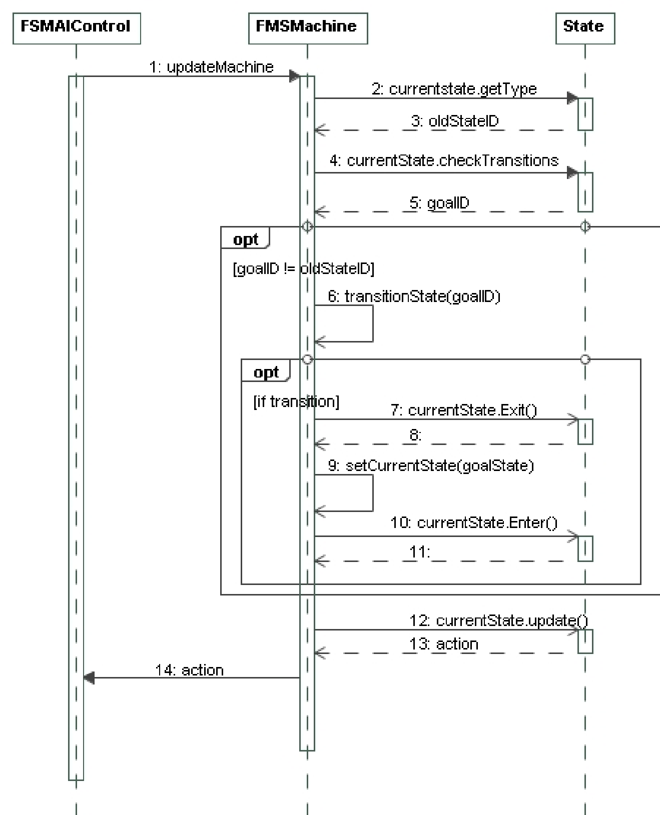

Figura 21 - Diagrama de sequência da MEF

5 Arquitectura da solução

Neste capítulo irá ser explicada a arquitectura da solução que foi desenvolvida com base na concepção da mesma, estabelecida no capítulo anterior.

A explicação começa por uma introdução aos elementos do *pygame*, como estes funcionam e como são utilizados, pois o funcionamento dos mesmos influencia a arquitectura criada e prosseguido por uma introdução aos aspectos gerais da arquitectura do projecto, onde será detalhado cada aspecto da mesma, de uma forma progressiva em maior detalhe.

5.1 Desenvolvimento de jogos em pygame

Para desenvolver um jogo em *pygame* é necessário primeiro saber quais os elementos base e como usá-los para criar um jogo. Irei listar alguns desses elementos e explicar o seu papel e como estes funcionam.

- *Surface* - representa uma superfície sobre a qual são desenhados os vários tipos de elementos do *pygame,* incluindo outras superfícies
- *Display* – representa janela de jogo, é responsável por a iniciar devolvendo a superfície da mesma e da actualização da mesma
- *Image* – responsável por carregar ou guardar uma imagem
- *Transform* – responsável pelas transformações de uma superfície, ou seja, rodar ou redimensionar uma imagem. É necessário ter cuidado pois estas transformações podem sofrer perca de pixéis, logo imagens que sofram várias transformações, estas alterações devem ser efectuadas sobre a original e não sobre a mesma evitando percas sucessivas
- *Draw* – que permite desenhar várias figuras geométricas sejam estas, rectângulos, círculos elipses, polígonos ou linhas numa determinada superfície
- *Rect* – representa áreas rectangulares e é responsável pela manipulação das mesmas
- *Font* – responsável por criar uma superfície onde se encontra escrito o texto desejado na fonte escolhida

- *Sprite* – representa os objectos de jogo visíveis, esta contem dois atributos que têm de ser definidos image e rect, que representam a superfície e a área (Rect) da mesma respectivamente. A Sprite por si só não pode ser desenhada numa superfície
- *SpriteGroup* – representa um grupo de *Sprites*, as *Sprite* que desejem ser desenhadas têm de ser adicionadas a um grupo, porque só é possível desenhar um grupo
- *Clock* – permite supervisionar o tempo de jogo, e gerir a *framerate* do jogo
- *Event* – representa todos os eventos do jogo

Para criar um jogo terá certamente que criar um ciclo de jogo, e como referido anteriormente a classe Clock permite gerir a *framerate* do jogo, ou seja, a frequência do ciclo de jogo por exemplo: chamando o método Clock.tick(40) o ciclo de jogo ocorrerá quarenta vezes por minuto.

A um ponto do jogo será necessário detectar colisões entre objectos do jogo, para isso o *pygame* tem vários mecanismos de colisão, um para as *Sprites* e para *SpriteGroups* e um para *Rect*.

O mecanismo de colisão de *Sprites* tem duas vertentes, duas são muito básicas, a por rectângulo e a outra pelo círculo das mesmas, em que a do rectângulo detecta se os rectângulos que encapsulam a *Sprite* se interceptam e a *circle* que segue a mesma lógica só que este usa o atributo *radius*, ou seja, o raio para criar os círculos que por definição são de raio suficiente para englobar o rectângulo da Sprite, e o terceiro que é a *mask*, este método é o mais preciso pois este irá verificar o bitmap das imagens das duas *Sprites* para ver se existe colisão. Este último é o mais ineficiente, mas é o mais preciso. Esta precisão cria um mecanismo mais realista pois os outros mecanismos detectavam colisões que o utilizador não interpretaria como tal. Mas, existe uma forma de atenuar o custo dessas colisões usando um mecanismo hibrido, ou seja, usar o mecanismo de colisão por rectângulo que é mais eficiente, pois se este detectar a colisão é feito um teste mais profundo usando o mecanismo por máscara.

O *Rect* oferece dois tipos de colisões entre rectângulos, similar à versão das *Sprites* e entre um ponto e o rectângulo, verifica se um ponto está contido no rectângulo.

Após a criação da interface gráfica será necessário obter o *input* do utilizador, iremos então explicar como o fazer para os dois principais dispositivos de input, o teclado e o rato.

No caso do rato é necessário verificar se o evento que aconteceu é de botão do rato e qual o botão do mesmo que foi pressionado. De seguida, é necessário obter a posição do rato, agora é necessário saber aonde foi clicado, para isso é necessário verificar se esta posição corresponde a algum elemento do jogo, para isso é necessário usar o mecanismo de colisão do *Rectangle* com o ponto, que é a posição do rato e o rectângulo dos elementos a serem testados.

No caso do teclado é necessário verificar se o evento que aconteceu é de tecla pressionada ou liberta, o *pygame* oferece dois mecanismos de leitura. O primeiro mecanismo é usando o método key.get_pressed() que devolve uma listas de booleanos das teclas pressionadas, para que as teclas pressionadas apareçam varias vezes quando continuam pressionadas, é necessário usar o método key.set_repeat(delay, interval). Só que este método tem dois problemas pois, quando existem duas teclas pressionadas e um delas é libertada, a que ficou pressionada deixa de ser detectada e também é impossível saber qual a ordem em que essas teclas foram pressionadas, oque impossibilita a escrita por parte do utilizador. O segundo mecanismo é usando o evento de tecla event.key que devolve a tecla que foi pressionada ou libertada, uma forma de o fazer é criar o dicionário de teclas, com o estado corrente das mesmas e afectá-lo com usando o event.key para saber a tecla e ver o tipo de evento para saber se este foi de libertar tecla ou de pressionar tecla.

5.2 Especificação da arquitectura da solução

A arquitectura desta solução baseia-se numa arquitectura de camadas, ilustrada na Figura 22. Este tipo de arquitectura permite uma solução com menor acoplamento entre subsistemas, na medida em que cada camada tem uma única responsabilidade separando a lógica de negócio de cada uma. Também foi criada uma camada de abstracção entre as camadas para reforçar tal separação, esta camada tem por base o padrão *façade*. Facilitando assim a manutenção e as alterações de cada uma das

camadas, pois permite a alteração de uma das camadas sem que as restantes tenham de ser modificadas.

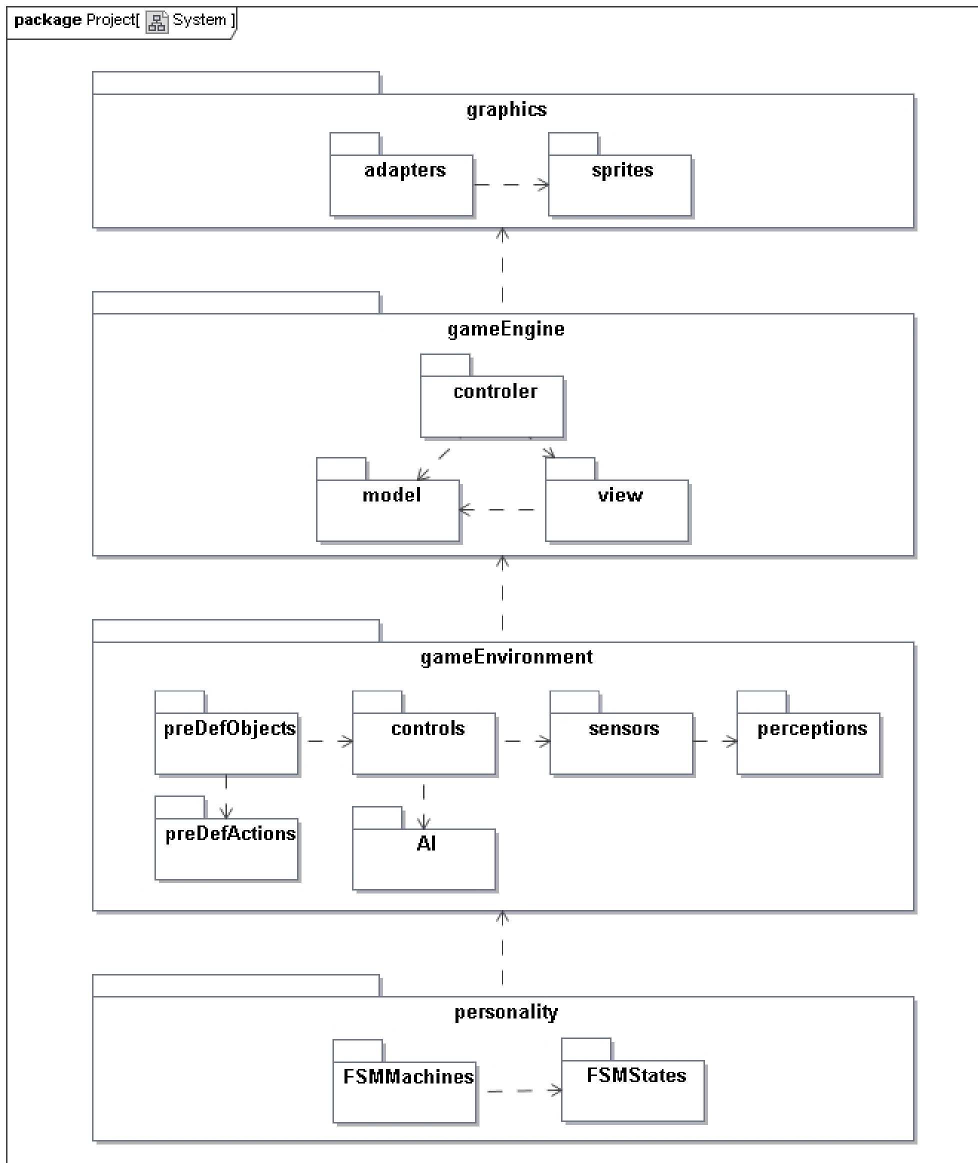

Figura 22 - Arquitectura de camadas

Como referido anteriormente a arquitectura é dividida em camadas e cada uma com a sua responsabilidade. As camadas encontram-se organizadas tal como apresentado na Figura 22, estando estas organizadas das de mais baixo nível para as de mais alto nível.

A camada gráfica (*graphics*) tem os elementos base do aspecto gráfico do jogo, ou seja, tem os elementos base necessários para a criação da interface gráfica.

A camada de motor de jogo (*gameEngine*), como o nome indica é a camada onde o motor de jogo se encontra, ou seja, esta camada é aquela que trata de todos os aspectos base que dão funcionamento ao jogo, seja a sua manutenção de estado e a evolução do mesmo, ou a visualização desse mesmo estado por parte do utilizador. Para isso são usados os elementos da camada gráfica e ciclo de jogo e a leitura das acções efectuadas pelo utilizador.

A camada de ambiente (*gameEnvironment*) é responsável por criar o ambiente de jogo, ou seja, esta é responsável pela definição das personagens do jogo, pela aplicação das regras do mesmo, é também responsável pela inicialização, evolução do jogo (passagem de níveis e do carregamento dos mesmo) e pelo término do mesmo. Para isso, utiliza a camada de motor de jogo e a interface que esta disponibiliza. Para além destes aspectos, esta camada disponibiliza mecanismos de controlo das personagens seja por controlo do utilizador ou por agentes autónomos.

A camada de personagens (*personality*) é a camada onde são definidos os comportamentos de cada personagem, ou seja, esta usa os mecanismos de controlo de personagens disponibilizados na camada de ambiente utilizando esses mecanismos para dar um comportamento específico a cada personagem.

5.2.1 Arquitectura da camada gráfica

Como referido anteriormente, esta camada tem os elementos base do aspecto gráfico do jogo. Para começar, é necessário criar uma *Sprite* para cada um dos elementos base do jogo que são as entidades, as áreas, os *tiles* e as imagens do jogo. Para isso foram criadas *Sprite* para os mesmos que são as *EntitySprites*, *AreaSprite*, *TileSprite* e *ImageSprite* respectivamente, como ilustrado na Figura 23.

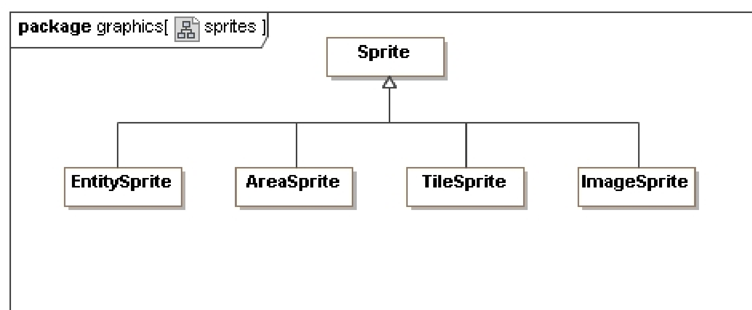

Figura 23 – Arquitectura das *sprites*

Como referido anteriormente o *pygame* oferece outros elementos para além das *Sprites* que dão suporte à visualização gráfica. Com o uso desses vários elementos temeu-se que a solução estaria a ficar demasiado acoplada ao *pygame* o que poderia afectar a evolução da solução futuramente, ou seja, caso fosse necessário existir uma evolução do projecto em que o uso de *pygame* deixasse de ser uma solução viável. Isto obrigaria à utilização de outra API gráfica que se adequasse a esta evolução, devido a este acoplamento a alteração da mesma seria complexa.

Surgiu então a solução de usar adaptadores para os diversos elementos do *pygame* usados no jogo, ilustrados na Figura 24. Tal solução permite que seja alterada a API gráfica com um esforço muito mais reduzido que a solução anterior, por exemplo no caso da classe *EntitySprite* ser alterada basta alterar o seu adaptador à classe *EntitySpriteAdapter* para que use a nova classe ou a classe modificada, não propagando a sua alteração nas camadas superiores, desde que o esqueleto do adaptador não seja modificado.

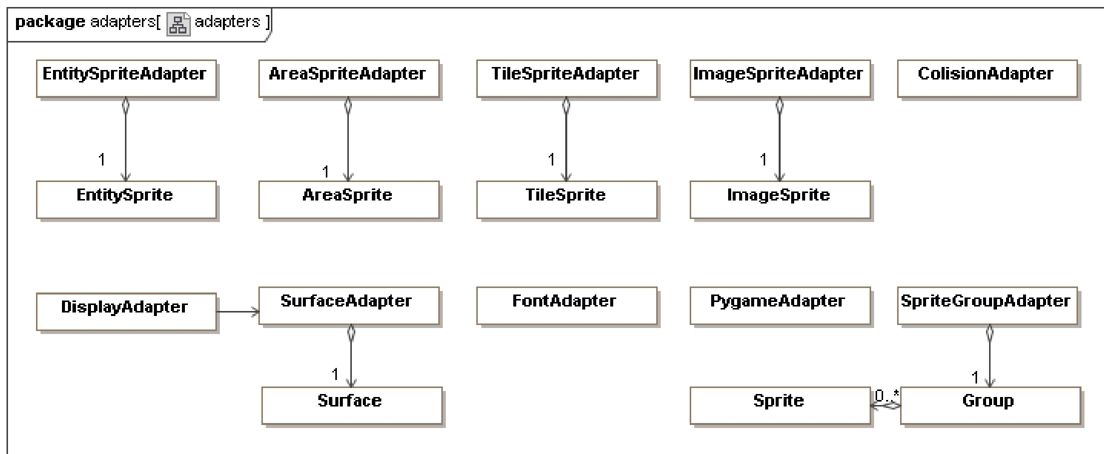

Figura 24 – Adaptadores

5.2.2 Arquitectura da camada de motor de jogo

A arquitectura raiz desta camada usa o padrão MVC (*Model View Controller*) [Reenskaug, 1979], pois este é o mais usado em projectos que tenham interface gráfica e também na maioria dos jogos do género. O uso deste padrão trás a vantagem de cada

módulo ter a sua responsabilidade. O modelo contém toda a informação necessária para o problema, a vista é a representação do modelo, responsável por escolher qual a informação a ser apresentada. Esta obtém a informação pedindo-a ao modelo, por sua vez o controlador serve como ligação entre as acções do utilizador e o sistema, traduzindo essas acções em evocação de métodos.

Esta separação tem a vantagem de levar a que a alteração de um desses módulos, reflicta uma mudança mínima nos restantes e isto permite que cada módulo seja desenvolvido separadamente e em paralelo, sendo condição que as interfaces de comunicação entre eles estejam definidas, permitindo assim um desenvolvimento mais rápido. Esta separação permite também que um dos módulos seja alterado sem que os restantes sintam tal mudança, por exemplo um sistema em que a sua vista assenta sobre *Windows Forms* pode evoluir para *Windows Presentation Foundation* sem que os restantes módulos sejam alterados. Assim sendo, a solução deste modo fica mais independente da plataforma gráfica usada.

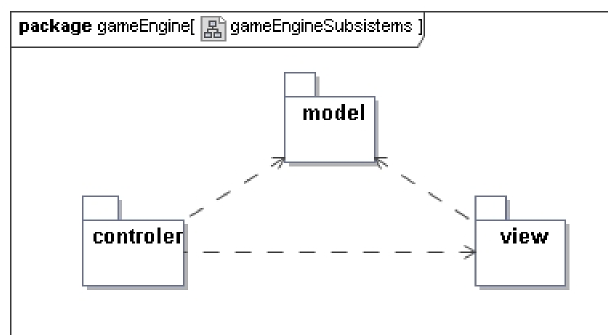

Figura 25 - *Model View Controller*

O padrão MVC tem várias vertentes e optou-se pela vertente demonstrada na Figura 25, ou seja, o Controlador (*Controller*) comanda o Modelo (*Model)* alterando o seu estado, após essa mudança de estado, o Controlador indica à Vista (*View*) que se actualize. A Vista trata de actualizar a visualização do estado do jogo, para isso acede ao Modelo para ir buscar esse estado e após obtê-lo, actualiza a *GUI* com o estado respectivo. Esta vertente tem a vantagem de que a camada de negócio, ou seja, o Modelo não dependa de ninguém, isto permite que o Modelo seja implementado e

testado separadamente, até porque é este que envolve mais tempo e recursos no seu desenvolvimento.

5.2.2.1 Arquitectura do modelo

O jogo é constituído por três tipos de elementos base: as entidades que representam os animais, plantas e frutos do jogo; as áreas que definem as áreas de água, floresta, deserto e outras; os os *tiles* que constituem o mapa do jogo. Cada uma destas tem um *SpriteAdapter*, ou seja, tem associada a sua representação gráfica. Para além dessas três classes existe a classe *MenuBar* que representa uma barra de menu.

A classe *GameModel* é a classe central do modelo, esta guarda o estado actual do jogo, seja este representado pelos objectos que existem no jogo, seja pela informação referente ao mesmo.

As classes *Entity* e *MenuBar* são objectos de jogo, ou seja, estes têm eventos associados aos mesmos, o uso destes eventos nos objectos de jogo permite que facilmente seja definido um comportamento a cada objecto para cada um desses eventos com grande facilidade, permitindo assim a implementação de objectos distintos mantendo a implementação base. Esta implementação é de fácil expansão, pois facilita a adição de futuros tipos de objectos de jogo e fácil expansão dos existentes, por exemplo, no caso de ser necessário adicionar um novo tipo de evento de futuro ao objecto de jogo, basta adicionar a sua interface e implementar o mecanismo de disparo para esse mesmo evento.

Para além de permitir comportamentos distintos, também é possível criar comportamentos comuns entre os objectos, bastando criar uma classe que seja um objecto de jogo que implementaa o comportamento comum entre os objectos a serem implementados, em que os objectos a serem implementados são uma extensão da classe com o comportamento comum.

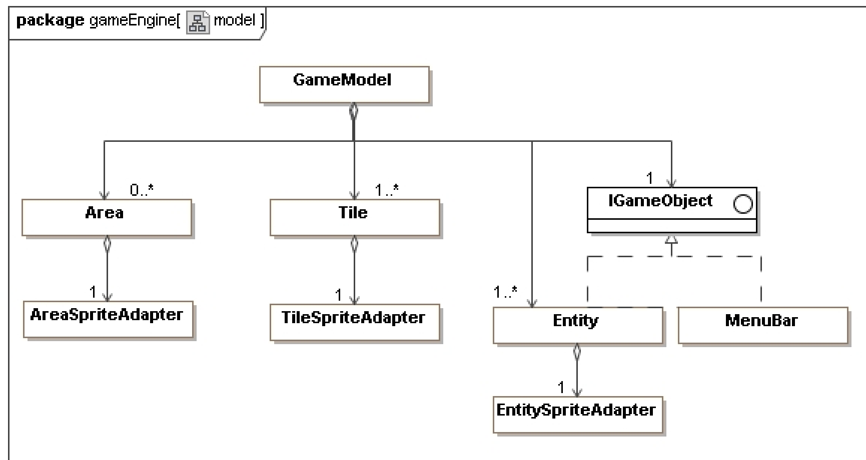

Figura 26 - Arquitectura do modelo

Na Figura 26 é possível constatar que Área e Tile não são objectos de jogo, inicialmente tudo levaria a pensar que estes seriam objectos de jogo porque fazem parte do jogo, mas depois de alguma reflexão concluiu-se que estes não deveriam ser objectos de jogo pois estes são objectos que não têm interacção, pois são estáticos, logo seria ineficiente estar a gastar recursos para gerar eventos para esses objectos.

5.2.2.2 Arquitectura da vista

A vista como referido anteriormente, trata dos aspectos gráficos e para isso terá de utilizar vários elementos da API *pygame*, seguindo a estratégia anterior de usar adaptadores, como ilustrado Figura 27.

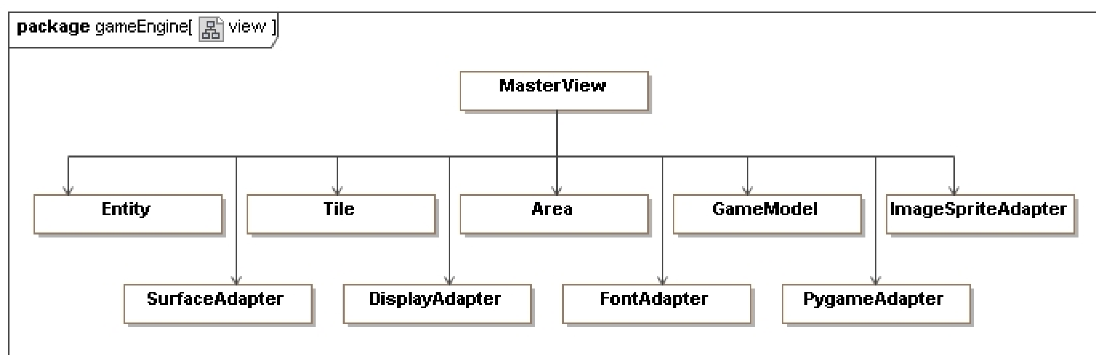

Figura 27 - Arquitectura da vista

A vista está dividida em duas partes, a parte de carregamento do ambiente de visualização que actualiza todos os intervenientes na visualização do jogo, como por exemplo o mapa de jogo onde são obtidos todos os tiles do mapa e é construído o mesmo. A segunda parte actualiza somente os intervenientes que mudam de estado, como por exemplo as entidades onde a sua posição é alterada. Esta solução tem a vantagem de só actualizar certos elementos, somente quando é necessário poupando assim o custo de os actualizar constantemente.

5.2.2.3 Arquitectura do controlador

O controlador é responsável pelo ciclo do jogo, recorrendo a várias classes para dar suporte ao mesmo, como ilustrado na Figura 28. Mas, antes de iniciar o ciclo do jogo inicia o Modelo e a Vista. Existem várias políticas de ciclo, a primeira em que a cada ciclo é actualizado o estado do modelo e o estado da visualização. Na segunda política, existem dois ciclos, um para actualização do estado do modelo e outro para a actualização do estado da visualização.

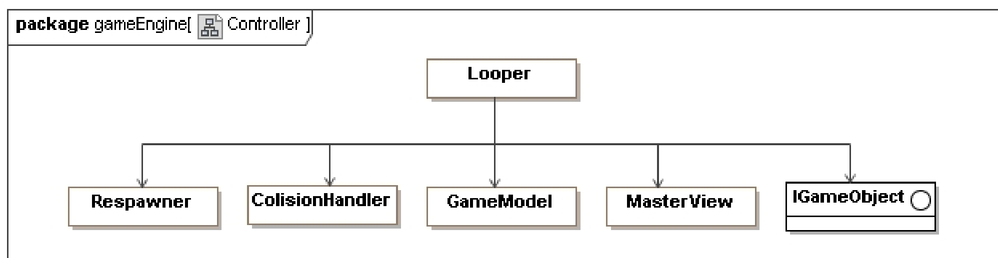

Figura 28 - Arquitectura do controlador

A primeira estratégia, tem o inconveniente de a actualização dos dois estados ser feita ao mesmo tempo, sendo necessário encontrar um ponto de equilíbrio que satisfaça as duas actualização não podendo ser muito rápida, (pois o jogo torna-se demasiado rápido) e não podendo ser muito lenta, (pois quebra o fluxo da interface gráfica), provocando perda ou latência das actualizações das mesmas. Este tipo de ponto de equilíbrio por vezes é difícil ou até impossível de encontrar. Já na segunda estratégia, isso não é necessário, pois existem dois ciclos separados podendo assim garantir a frequente actualização da componente gráfica. Desta forma, a taxa de actualização em *fps* (*frames per second*), a uqal normalmente tem um valor de 60 *fps*, e a velocidade do jogo podem ser adequadas ao jogo em questão, com a velocidade pretendida para o mesmo, podendo esta ainda variar por nível. Tendo esses factos em mente, optou-se pela segunda politica sendo que tem grandes vantagens em relação à primeira e pela implementação de um segundo ciclo ser considerada trivial.

A classe *ColisionHandler* verifica se existe colisão entre as entidades que estão vivas e a colisão entre as entidades com as áreas, caso haja colisão, a entidade é movida para a posição imediatamente adjacente à entidade que colidiu. Também verifica se a entidade sai dos limites do mapa, não a deixando trespassá-los e retorna à lista de colisões existentes.

Como referido anteriormente os objectos de jogo têm associados eventos e é o controlador que é responsável por os disparar. Sempre que uma tecla é premida, este dispara o evento de tecla passando a tecla que foi premida, sempre que existe um clique por parte do utilizador este verifica se este foi sobre um objecto de jogo e dispara o evento de clique do mesmo. A cada ciclo de jogo, o controlador dispara o evento de passo e por último, a cada ciclo de jogo o controlador usa a classe *ColisionHandler* para detectar as colisões existente e dispara o evento de colisão dos objectos envolvidos na mesma.

A classe *Respawner* verifica quais as unidades que podem voltar à vida e gera uma nova posição para as mesmas no mapa e volta-as à vida.

5.2.2.4 Módulo de abstracção da camada de motor

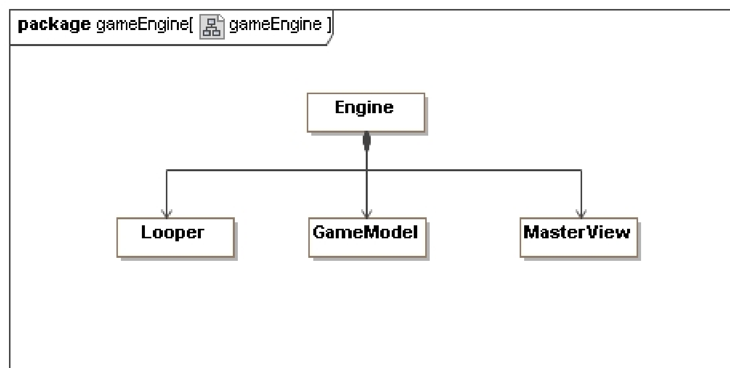

Figura 29 - *Façade* da camada de motor

Para implementar o modulo de abstracção da camada de motor recorreu-se a implementação do padrao *façade*, como ilustrado na Figura 29, o uso desta cama de abstracção permite que as camadas superiores usem esta camada sem saber os detalhes da mesma e uma melhor manutenção pois esta camada só tem um ponto de acesso em vez de serem três, como esta serve como ponto único de acesso ao motor de jogo pelas várias classes das restantes camadas, a classe *Engine* é *singleton* o que permite que o motor seja acedido de forma simples e sempre sobre a mesma instância.

5.2.3 Arquitectura da camada de ambiente

Como referido anteriormente, a camada de ambiente, é responsável por criar o ambiente de jogo. Definir e gerir o ambiente de jogo, envolve várias tarefas distintas, tais como a definição das personagens do jogo, a aplicação das regras do mesmo, é também responsável pela inicialização e pelo fluxo do jogo.

Para cumprir as várias tarefas, a arquitectura do ambiente foi dividida em vários subsistemas, cada um com tarefas distintas como ilustrado na Figura 30, tornando o sistema modular e de fácil manutenção.

No subsistema *PreDefObjects* (objectos pré-definidos), encontram-se todos os objectos de jogo existentes e toda lógica que os eventos têm sobre os mesmos, definindo assim o comportamento de cada tipo de objectos, despoletando assim as repercussões que os mesmos têm sobre o jogo fazendo evoluir o estado do mesmo. De modo a superar a logica, foi criado o subsistema de *PreDefActions* (acções pré-definidas), onde estão todas as acções que envolvem a evolução do jogo por exemplo a mudança de nível e as acções que as entidades efectuam.

Como referido anteriormente, o controlo das entidades de jogo poderá ser feito por parte do jogador ou de um agente autónomo, sendo necessário o sistema tratar desta ambiguidade, que é o controlo das unidades. Para isso, foi construído o subsistema *Control* que trata de traduzir o input de utilizador ou a IA do agente.

Mas o agente necessita de uma percepção do mundo para poder determinar a acção que a entidade deve fazer. É aí que entra o subsistema *Sensors* (sensores), onde se encontram os vários tipos de sensores que o agente usa para obter essa percepção que necessita para determinar a acção seguinte.

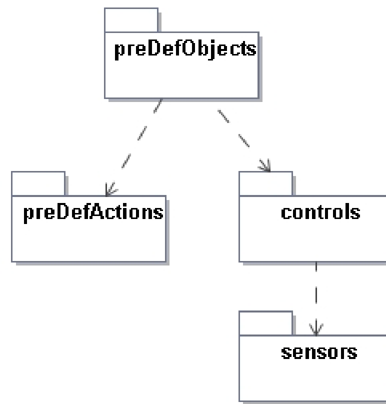

Figura 30 - Arquitectura da camada de ambiente

5.2.3.1 Arquitectura do subsistema de objectos pré-definidos

Como referido anteriormente o subsistema de objectos pré-definidos contém todos os objectos de jogo existentes. Existem vários tipos de objectos sejam eles frutas, animais ou até a barra de menu.

Foi criada a arquitectura ilustrada na Figura 31, de modo a minimizar a complexidade e o código necessário para criar um objecto de jogo, mas principalmente para poder adicionar novas entidades ao jogo, para que no futuro, caso seja necessário adicionar novos tipos de entidades com comportamentos similares ou diferentes das existentes.

Inicialmente as entidades são separadas em dois grupos. Os *Character* (personagens), que representam todas as unidades que o jogador pode controlar, sejam estas animais ou caçadores e os *Unanimated* (inanimados) que representam todas as entidades que o jogador não pode controlar e que não se movem ou movem-se num padrão fixo, por exemplo em linha recta. A diferença entre esses dois grupos não acaba aqui, pois as personagens têm fome, podem morrer de fome e podem comer outras entidades. Os objectos inanimados por sua vez, não o podem fazer podendo somente ser comidos.

Os objectos inanimados são posteriormente divididos em dois grupos distintos, pois como referido anteriormente existem os que se movem em padrões que são representados pelos *Mobile* (móveis), e pelos que não se movem, que são representados pelos *Static* (estáticos). Os estáticos foram divididos em dois grupos as frutas e plantas.

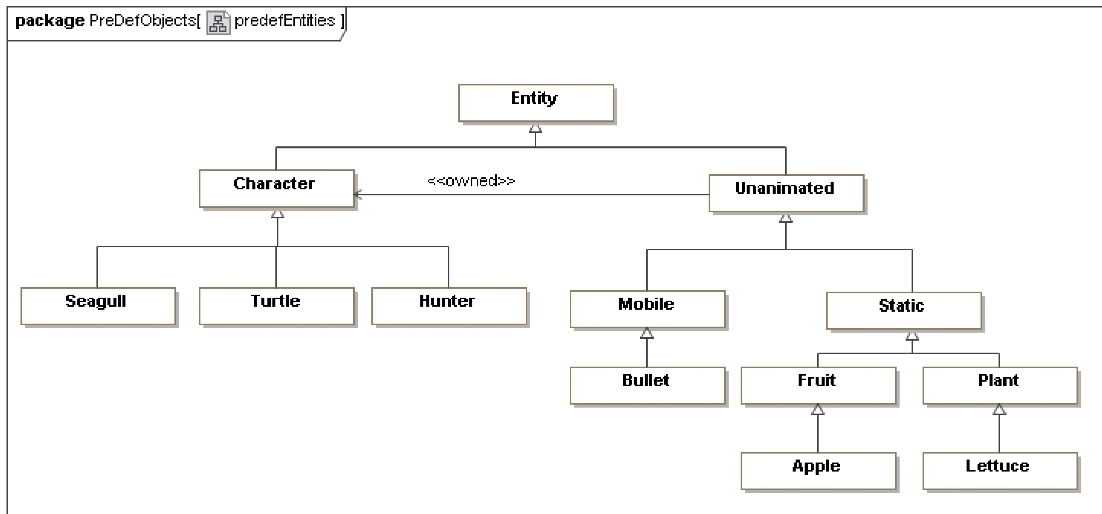

Figura 31 - Arquitectura das entidades

Como referido anteriormente as entidades têm os eventos a partir dos quais é definido o comportamento de cada entidade ou grupo de entidades com comportamento comum.

O comportamento dos personagens está definido da seguinte forma:

- Evento de botão esquerdo do rato pressionado – selecciona a unidade;
- Evento de colisão – verifica se colidiu com uma presa e se for o caso, reduz a fome do personagem e mata a presa, mas se a personagem for controlada pelo jogador este actualiza os seus pontos. Se os pontos atingirem o patamar de mudança de nível este provoca mudança de nível. No caso do personagem, a cada passo do jogo, a fome do mesmo aumenta a um ritmo pré estabelecido. Caso a fome chegue ao máximo, o personagem morre. No caso dos personagens quando existe uma colisão, este verifica se colidiu com uma presa e se for o caso reduz a fome do personagem e mata a presa. Mas, se o personagem for controlado pelo jogador este actualiza os seus pontos e se os pontos atingirem o patamar de mudança de nível este provoca mudança de nível. No caso do personagem, a cada passo do jogo a fome do personagem é reduzida a um ritmo pré estabelecido, caso a fome chegue ao máximo o personagem morre;

- Evento de tecla pressionada – é verificado se a personagem é controlada por um jogador, nesse caso é necessário traduzir essa tecla em acções sobre a mesma chamando o seu controlador;
- Evento de passo – no passo é necessário saber qual a acção que a personagem irá realizar, para isso o personagem usa o controlador associado que iremos abordar posteriormente quando o subsistema *control* for abordado.

O comportamento dos objectos inanimados está definido da seguinte forma:

- Evento de botão esquerdo do rato pressionado – selecciona a unidade;
- Evento de colisão – como indicado na Figura 31, existe uma relação de *owner* (dono) entre os objectos inanimados e os personagens, isto representa os objectos que pertencem a um personagem, como é o caso da bala, pois caso a bala colida com outra entidade despoletará o comportamento de colisão do seu dono, pois é o caçador que irá comer a unidade e não a bala em si;
- Evento de tecla – não existe qualquer comportamento;
- Evento de passo – não existe qualquer comportamento.

O comportamento dos objectos móveis está definido da seguinte forma:

- Evento de botão esquerdo do rato pressionado – herdado dos objectos inanimados;
- Evento de colisão – herdado dos objectos inanimados;
- Evento de tecla – herdado dos objectos inanimados;
- Evento de passo – a unidade move-se de acordo com o padrão de movimento associado, no caso da bala é em linha recta.

Com esta arquitectura implementada é simples adicionar um novo animal, bastando herdar do tipo personagem e definir os seus atributos, podendo alterar o seu comportamento de acordo com a necessidade, tal como é simples a adição de outro qualquer tipo de entidade.

Mas no jogo não existem só objectos do tipo entidade existe também o *MenuBar* criada a partir da *MenuSystem* [josmiley, 2012], foi então criada uma versão padrão da mesma denominada *StandartMenuBar* (barra de menu padrão) ilustrada na Figura 32. Esta permite que o utilizador seleccione as seguintes acções: começar um novo jogo, resumir e pausar o jogo, seleccionar o tipo de jogador e sair do jogo. Este só implementa o evento de clique e neste verifica qual a opção que foi seleccionada e despoleta essa acção.

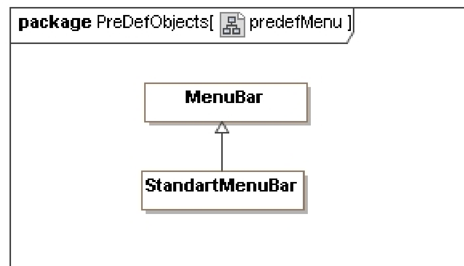

Figura 32 – Arquitectura da barra de menu

5.2.3.2 Arquitectura do subsistema de acções pré-definidas

As acções pré-definidas estão divididas em dois grupos, as acções do jogo que controlam o estado do jogo e acções das entidades que controlam as acções da entidade.

As acções que controlam o estado do jogo dividem-se em quatro tipos de acções que são as acções da pontuação que alteram a pontuação do jogo, as de nível que permite passar para o próximo nível, recomeçar nível e recomeçar o jogo, as acções do estado do jogo que resumem ou pausam o jogo, seleccionam uma entidade e seleccionam o tipo de jogador e por fim as acções sobre as entidades que afectam a fome da unidade, matam a unidade e ressuscitam a unidade.

As acções das unidades são somente duas, como é possível ver na Figura 33 mover e disparar, estas acções só recebem o ângulo da acção, este tipo de abordagem simplista tem a vantagem das acções serem tratadas da mesma forma, permitindo que novas acções sejam adicionadas sem necessitar proceder a alterações no sistema.

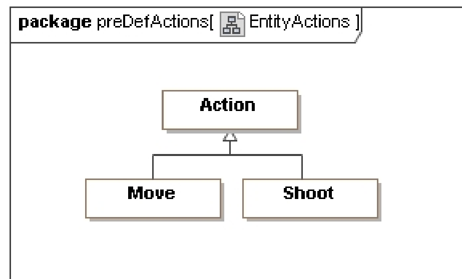

Figura 33 - Arquitectura das acções das unidades

5.2.3.3 Arquitectura do subsistema de controlador

O controlador tem como propósito descodificar as teclas que o utilizador pressiona e de as converter em acções sobre a entidade do jogador caso este seja controlado por um humano. É também responsável por determinar a acção de uma entidade que seja controlada pelo computador, ou seja, pela inteligência da personagem.

Como ilustrado na Figura 34, o controlador está dividido em dois tipos *Human* que como o nome indica traduz as acções do humano sobre uma entidade e quando utilizado produz uma acção de acordo com as teclas pressionadas. O segundo tipo, o *AutoControl* que como o nome indica representa o controlo automático das unidades que a cada passo determina as acções das entidades automaticamente. Este controlo automático não seria muito rico se determinasse as acções de forma aleatória ou determinística, por isso é necessário adicionar inteligência artificial ao mesmo. Para isso, foi criado o *IAControl*, e como referido anteriormente a IA necessita de uma percepção do jogo e é esta classe que é responsável por usar os sensores para obter essa percepção e actualizar a mesma quando necessário.

A IA neste momento só tem duas implementações que são o *ReactiveControl* que implementa mecanismos reactivos da IA, ou seja, este simplesmente recebe o estímulo que é a sua percepção e produz a sua resposta que é uma acção e o *DeliberativeControl* que implementa mecanismos deliberativos da IA, ou seja, que ao contrário do mecanismo reactivo que é mais simples, quando recebe um estímulo produz um plano de acção para atingir o objectivo pretendido. Mas, para obter esse plano usa mecanismos de procura em espaço de estados que será explicado em seguida.

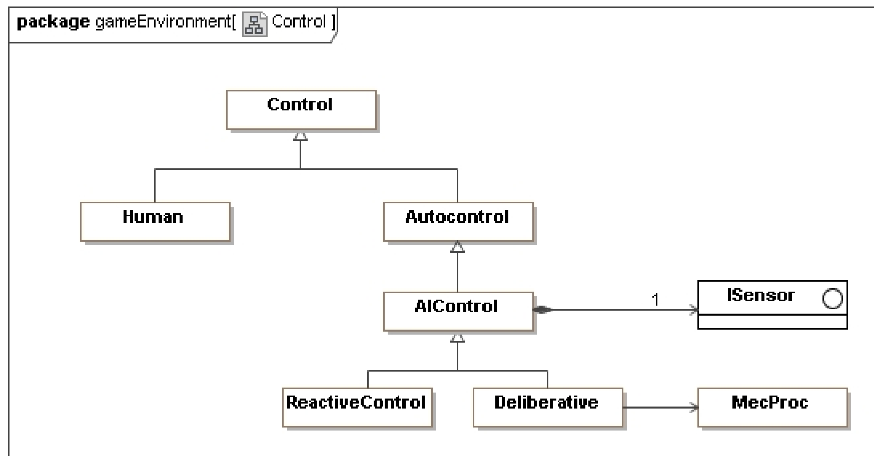

Figura 34 - Arquitectura do controlo

A procura em espaço de estados, é um mecanismo já muito explorado e desenvolvido na área da IA, este mecanismo usa uma representação do mundo e produz um plano de acções para atingir um objectivo final. Existem vários mecanismos implementados que são a procura em profundidade, largura, custo uniforme, sôfrega e por último e mais usado o A*. Estes estão organizados como ilustrado na Figura 35. É necessário referir que esta arquitectura foi construída no âmbito da cadeira de Complementos de Inteligência Artificial (CIA) e posteriormente adaptada a este projecto. Este sistema tem disponível um conjunto de estatísticas para comparação dos algoritmos para que seja possível a quem a utilizar comparar qual o algoritmo mais eficiente, o mais rápido, ou aquele que encontra uma melhor solução.

O uso deste mecanismo é simples, de forma a ser somente necessário definir os operadores de transição de estado, estado inicial e estado final, nos mecanismos *BestFirstMec* (melhor primeiro), é necessário criar um objectivo que contém o estado final e a heurística a utilizar.

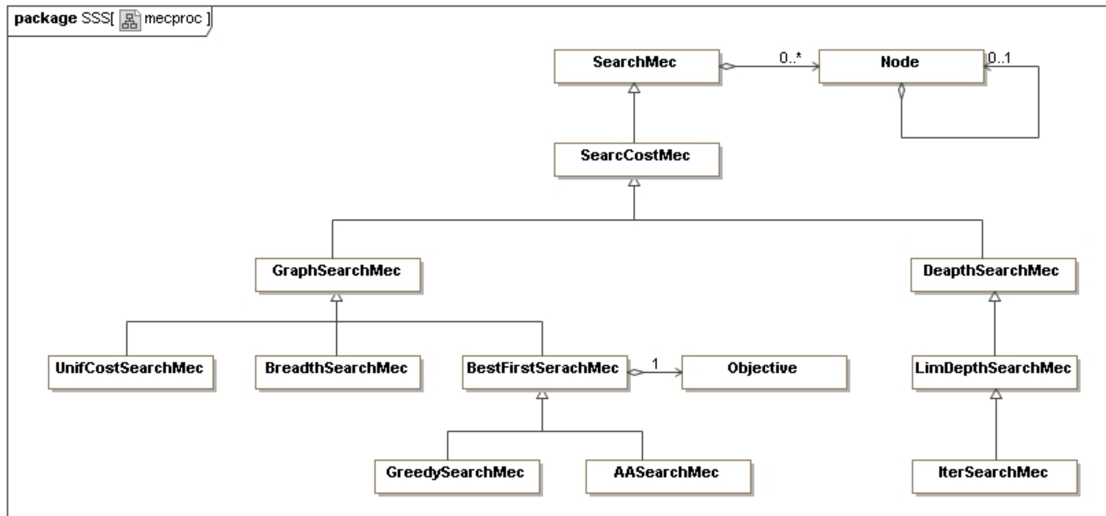

Figura 35 - Arquitectura do mecanismo de espaço de estados

5.2.3.4 Arquitectura do subsistema de sensores

Como referido anteriormente, pra suportar a IA das personagens é necessário precisa de uma percepção do mundo, geralmente não existe uma única percepção do mundo, por exemplo o controlo reactivo não precisa do mesmo tipo de percepção que o controlo deliberativo, até por que o funcionamento dos mesmos é muito diferente. Como a percepção é obtida através dos sensores, conclui-se que é necessário existir um tipo de sensor para cada tipo de percepção.

A existência de vários sensores e de várias percepções leva a que não haja um consenso, levando à diferenciação entre a sua utilização que leva a mais código e a uma manutenção mais complexa, daí surgiu a arquitectura ilustrada na Figura 36. Esta arquitectura tem um sensor global (*GlobalSensor*), contendo nele todos os tipos de sensores como ilustrado na Figura 36, sendo que quando é pedida a sua percepção este utiliza os vários sensores e obtém as suas percepções e com essas, cria a percepção global que é o conjunto de percepções, nesta solução existem dois sensores um sensor de proximidade que mede a proximidade de uma entidade com a outra com dois pontos de referência e o sensor deliberativo que obtém a informação do mapa, guardando essa percepção nas devidas classes.

Esta abordagem permite uma homogeneidade na utilização dos sensores, e permite com relativa facilidade inserir novos tipos de sensores e percepções sem alterar

a forma como estes são utilizados. Permite também que o controlo de IA utilize vários tipos de percepções no seu algoritmo, para ter um nível de detalhe sobre o mundo diferente tendo assim mais informação. Mas, esta solução tem a desvantagem de obter todas as percepções quando o controlo de IA só necessita de uma, ou quando uma percepção não necessita de ser actualizada com menos frequência. Esta questão deverá ser melhorada no futuro e uma das soluções será permitir ao controlo de IA indicar quais as percepções que quer e a frequência das suas actualizações.

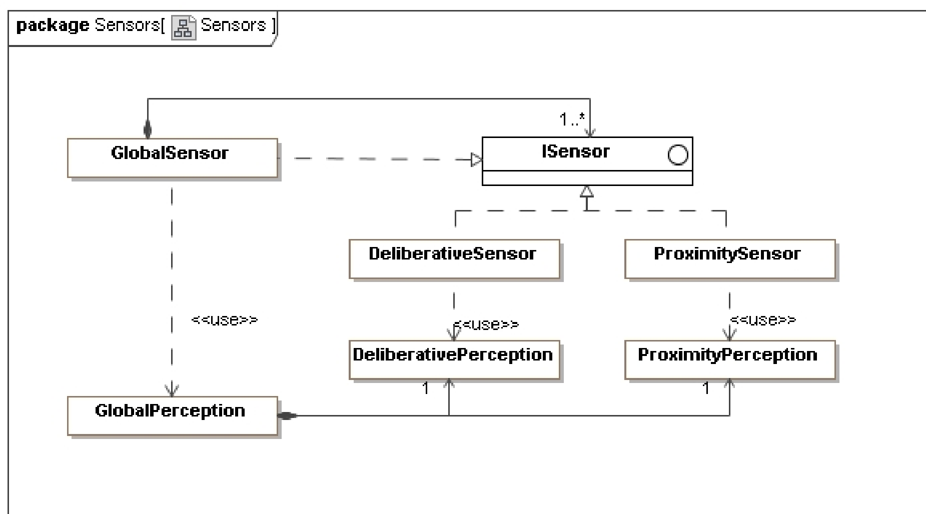

Figura 36 - Arquitectura dos sensores

5.2.3.5 Módulo de construção de controladores

Como referido anteriormente o jogo contém carregamento dinâmico de níveis e durante este carregamento é atribuído às entidades o seu controlador. Este carregamento inicialmente estava "*hardcoded*", ou seja, a entidade era atribuído um controlador predefinido na classe que carregamento de nível, sendo que esta solução necessitava que fosse alterada a classe responsável pelo carregamento de nível de cada vez que houvesse necessidade de mudar de controlador.

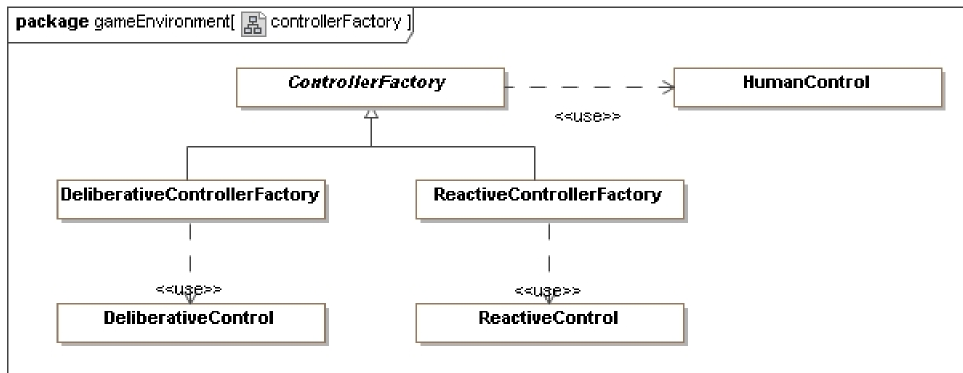

Figura 37 - Arquitectura da construção de controladores

Com este cenário em mente, foi elaborada uma solução Figura 37, que usa o padrão *factory*, ou seja, a classe de carregamento de nível recebe uma classe que cria o controlador, isto permite que ao iniciar a plataforma seja possível atribuir uma classe que cria o controlador pretendido, seja este controlador por um humano ou por um agente, no último este o seu controlador é escolhido de acordo com o *factory* escolhido.

5.2.4 Arquitectura da camada de personagens

A camada de personagens, é a camada onde são definidos os comportamentos de cada personagem, e para isso esta usa os mecanismos de controlo de personagens disponibilizados na camada de ambiente. É o mecanismo mais usado nos jogos para definir o comportamento das personagens.

O comportamento das personagens é implementado com recurso a máquinas de estados finitos (MEF), sendo que estas contemplam três aspectos: todos os seus estados, as condições de *input* e as transições de estado que servem como conexão entre os estados.

Segundo [Schawab, 2004] uma MEF pode ser implementada de duas maneiras, da forma clássica e da forma modular. Ele apresenta esta estratégia como tendo a vantagem da máquina não deter toda a lógica, sendo os estados simples estruturas de dados. Em vez disso a sua estratégia passa pelos estados serem módulos independentes que têm a sua lógica e a sua transição de estados, isto permite que sejam adicionados novos estados sem grande dificuldade.

Com estes factos em mente, optou-se por usar a estratégia de máquina de estados modular, seguindo o modelo de Moore em que as acções dos estados estão nos mesmos.

Daí surgiu a arquitectura ilustrada na Figura 38. Esta arquitectura, foi baseada na arquitectura presente no livro [Schawab, 2004]. Como referido anteriormente a camada de personagens assenta sobre a camada ambiente, isto é feito através do controlador dessa camada, como podemos ver na Figura 38 foi criado um controlo para a máquina de estado finita (*FSMAIControl*) que estende o controlo *AIControl* existente na camada de ambiente. A responsabilidade deste controlo é responsável por obter da percepção as variáveis de input, que a MEF necessita para verificar se existem transições.

Como é possível verificar o *FSMAIControl* tem uma MEF associado ao mesmo, vendo o diagrama surge a dúvida de a máquina de estado ser também um estado. Isto tem por objectivo permitir que a máquina de estados possa ser usada como estado composto numa outra MEF, de modo a criar um comportamento mais complexo e modular.

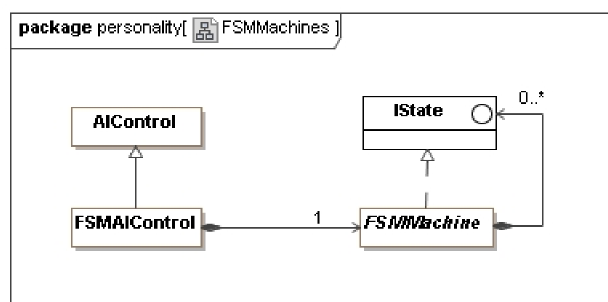

Figura 38 - Arquitectura da cama de personagens

5.2.4.1 Arquitectura da percepção da máquina de estados finitos

Como referido anteriormente o controlador da MEF, necessita de uma percepção, a solução mais óbvia seria criar uma percepção específica para esta sem ligação às percepções existente na camada de ambiente, mas isto criaria uma divergência na utilização das percepções.

Sendo assim optou-se por estender a percepção global existente, daí surgiu a arquitectura definida na Figura 39,esta arquitectura cria uma solução homogénea permitindo o uso desta percepção sem qualquer alteração da forma como as percepções são usadas.

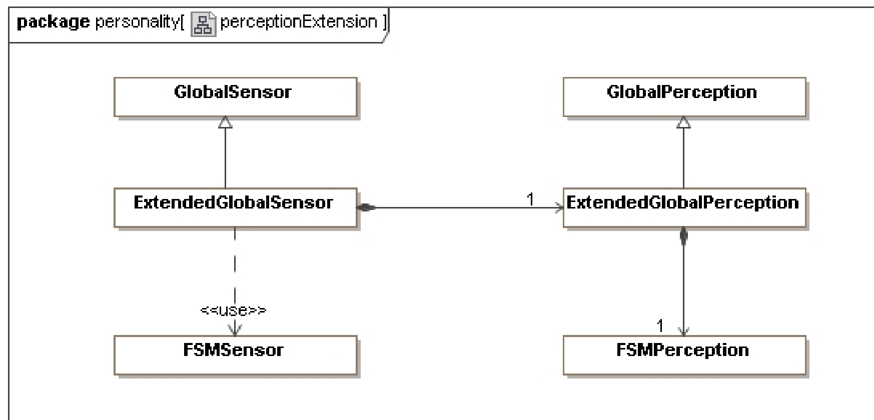

Figura 39 - Arquitectura da percepção da máquina de estados finitos

5.2.4.2 Arquitectura dos estados existentes

Tendo a arquitectura base da MEF estabelecida, é necessário então implementar os comportamentos das personagens estabelecidos no capítulo anterior. Começando pela criação dos estados, que se encontram ilustrados na Figura 40, como estados de deambular (*Wander*) e escapar (*Escape*), os quais são usados por vários comportamentos, sendo o comportamento o mesmo nos vários estados, só mudando as transições de estado.

Para reaproveitamento, criou-se então um estado de escapar genérico, onde está implementado o comportamento de evasão, mas deixando as transições em aberto, permitindo assim criar uma evasão específica para cada personagem, como por exemplo escapar do predador (*PredatorEscape*), na qual este implementa somente as suas transições.

Isto permite que sejam inseridos futuramente novos comportamentos que usem tais estados com pouco esforço.

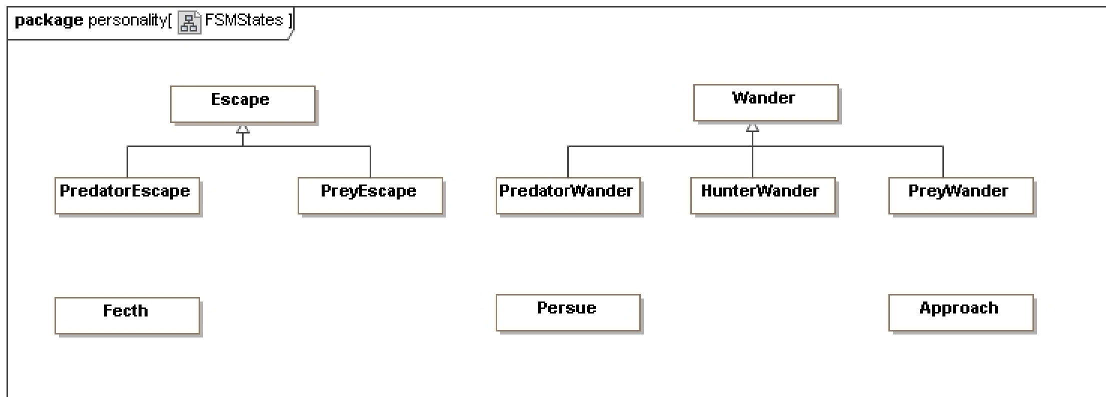

Figura 40 - Estados existentes na plataforma

A criação de comportamentos passa por criar uma máquina de estado para cada comportamento que exista, para facilitar a implementação dos mesmos. O tipo *FSMMachine* contém toda a lógica base da máquina de estados.

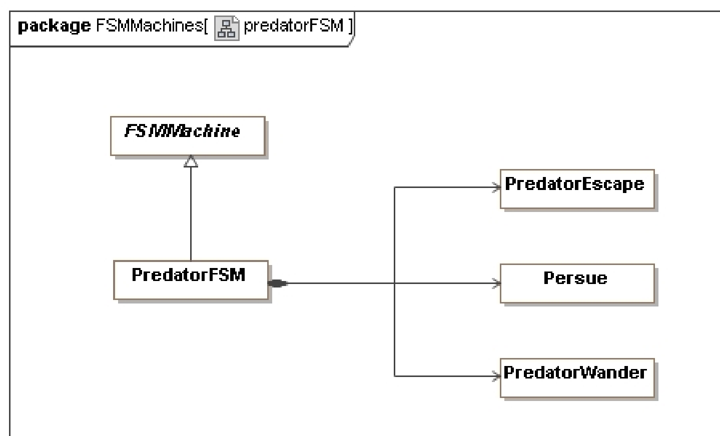

Figura 41 - Arquitectura da MEF do predador

Para criar uma máquina de estados é somente necessário criar uma MEF que estenda a mesma e que contenha os estados pela ordem correcta, como exemplificado para a MEF do predador, que se encontra ilustrada na Figura 41.

5.2.4.3 Módulo de construção de controladores da camada de personagens

O módulo de construção de controladores é mais complexo que na camada ambiente, pois este terá de ciar o controlador e atribuir a MEF especifica, ou seja, este terá de verificar qual o tipo de unidade (predador, presa e caçador) e obter a MEF que implementa o seu comportamento específico dando origem à arquitectura ilustrada na Figura 42.

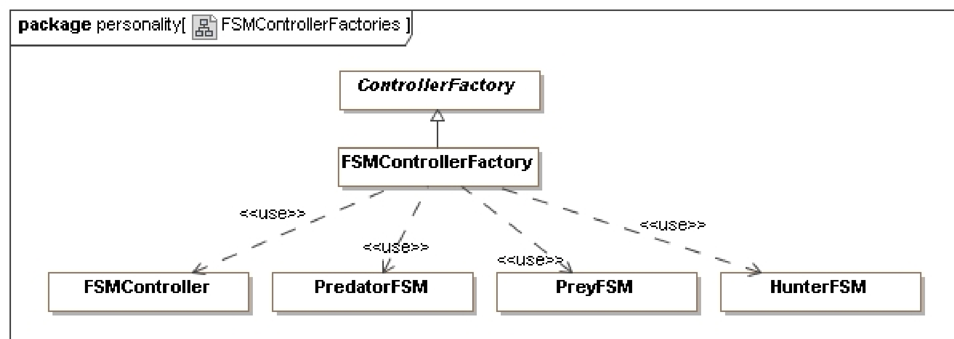

Figura 42 - Arquitectura do módulo de construção de controladores da camada de personagens

6 Implementação da solução

Neste capítulo são explicados pormenores específicos da implementação da solução, nomeadamente a criação das áreas e jogo, a geração de eventos e construção da percepção deliberativo.

6.1 Criação das áreas de jogo

As áreas de jogo são definidas através de um polígono, ou seja um conjunto de pontos, este polígono tem de ser desenhado numa superfície sendo que não é possível saber sem quaisquer cálculos adicionais a dimensão do polígono, então a dimensão da superfície terá de contemplar o pior caso, ou seja, o polígono ter o mesmo tamanho que o próprio mapa de jogo. A inserção das áreas criadas desta forma gerou problemas de desempenho, ou seja, o tempo de execução do jogo era superior a do ciclo de jogo atrasando o mesmo e baixando a taxa de actualização de imagem.

Esta quebra de desempenho deveu-se à verificação de colisões entre as entidades e as áreas, pois como o método usado para colisão era o da máscara e cada área ocupava a área do mapa todo, estas tinham de verificar as colisões com todas as áreas *pixel* a *pixel*.

Para resolver tal problema de desempenho foi implementado um algoritmo que calculava a dimensão do polígono, criando assim uma superfície com a dimensão estritamente necessário reduzindo o custo do cálculo de colisões e removendo o atraso do jogo.

6.2 Geração de eventos

Como referido anteriormente durante o ciclo de jogo existe geração dos diversos eventos do jogo, essa geração implica alguma implementação específica que será explicada em seguida

6.2.1 Geração do evento de clique e de tecla

O evento de clique começa pela geração do evento de clique da barra de menu, pois esta já possui a implementação da detecção do clique sobre a mesma e da opção

escolhida, bastando assim evocar esse método para saber se foi activado o evento respectivo e gerar o evento de clique sobre a barra.

No caso das entidades, como são várias, é necessário verificar uma a uma se existe clique, como a implementação das *Sprites* do *pygame* não têm qualquer detecção de clique, este é feito da seguinte forma: quando é invocado o método de clique da entidade, esta recebe a posição actual do rato e cria um ponto e em seguida verifica se esse ponto colide com a sua *Sprite*, no caso de colidir significa que o clique foi efectuado sobre a mesma, após detectado o clique, é gerado o evento.

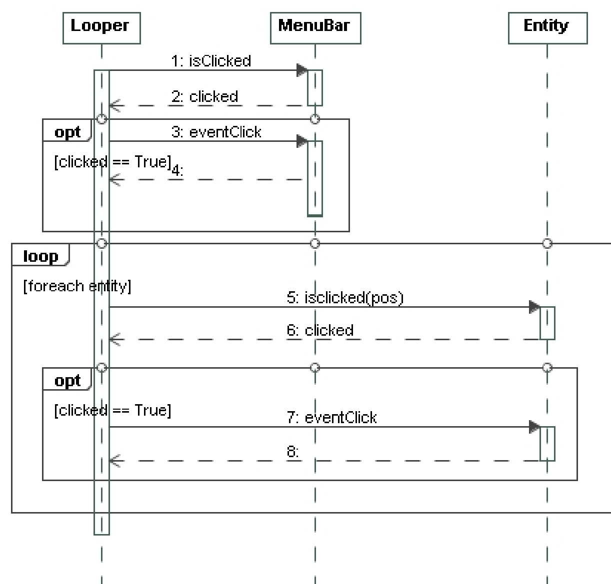

Figura 43 - Diagrama de sequência do evento de clique

O evento de teclado, como referido no subcapítulo de desenvolvimento no *pygame*, quando da leitura de teclado traz alguns problemas. Para dar a volta aos mesmos foi criada uma lista com todas as teclas e a cada ciclo é verificado o estado das mesmas, ou seja se a tecla se encontra pressionada ou não. Afectando a lista com os estados das mesmas, esse estado permite verificar quais as que se encontram pressionadas e essas são envidas no evento de tecla das entidades.

6.2.2 Geração do evento de passo e de colisão

A geração do evento de passo é relativamente simples e está ilustrada na Figura 44. Sabendo que o jogo não está em pausa, é somente necessário saber se aquele ciclo

de jogo corresponde a um ciclo de acção de jogo, nesse caso é somente necessário gerar o evento de passo das entidades que irá aumentar a sua fome e calcular a próxima posição das mesmas e em seguida actualizar a sua posição.

Em seguida, com a nova posição é necessário verificar as colisões existentes de modo a gerar o evento de colisão, inicialmente, essa colisão só era verificada quando as entidades se encontravam na sua posição final, mas esta solução tinha vários problemas, que provoca perda de colisões, por exemplo duas entidades em que na sua trajectória colidissem mas que na sua posição final não colidissem, essa colisão era perdida. Para evitar esse cenário as colisões são verificadas ao longo da trajectória e não só na sua posição final, assim foi evitado esse problema mas foi gerada mais carga computacional devido ao facto de existirem mais verificações de colisão.

No final é verificado se a entidade já atingiu o limite de fome, ou seja, se morre de fome ou não, isto não é feito logo no início quando se aumenta a fome pois neste último passo esta entidade eventualmente poderá comer mesmo no limite.

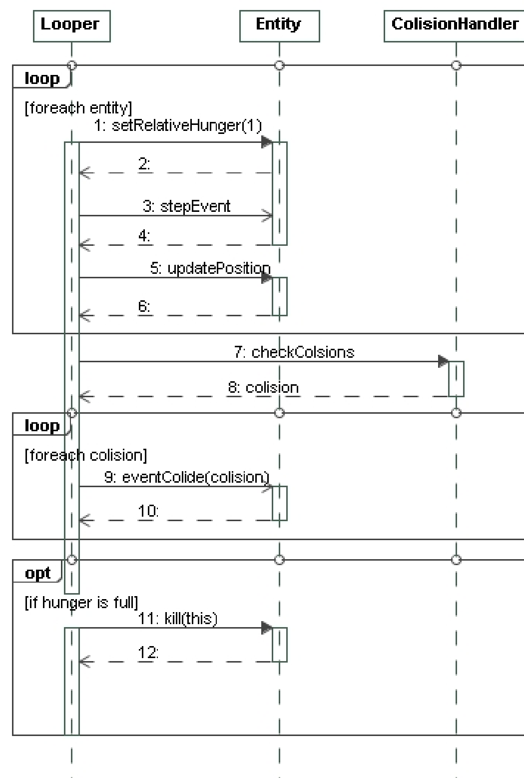

Figura 44 - Diagrama de sequência do evento de passo e de colisão

6.3 Construção da percepção deliberativa

A construção da percepção deliberativa começa pela criação de uma representação do mapa de jogo, essa representação é constituída por uma grelha constituída por quadrados de uma determinada dimensão, cada quadrado da mesma contém a sua posição central e os tipos de terrenos engloba.

Para determinar as áreas de um dado quadrado do mapa, é criada uma área correspondente a esse quadrado e posteriormente é verificado quais as áreas que esta colide.

A informação sobre as áreas de cada quadrado serve para determinar se uma entidade se pode deslocar para tal quadrado, por exemplo uma entidade aquática não pode deslocar-se para um quadrado onde existe terreno de terra.

Este também permite obter a posição da entidade e dos predadores e presas/alimentos mais próximos.

7 Verificação e testes

Neste capítulo são discutidos alguns testes e verificações feitas a plataforma.

7.1 Verificação e teste ao carregamento de níveis

Para suportar os testes do carregamento de níveis foi elaborado um nível que contém duas áreas, uma área de floresta e outra de água e duas entidades, uma gaivota que se trata de uma espécie voadora e uma tartaruga uma espécie anfíbia.

Para começar o teste foram criadas as entidades e áreas como é suposto ficarem após o carregamento, juntamente com a dimensão do mapa e pontuação de passagem para o próximo nível.

O nível de testes é então carregado, e é nessa altura que é verificado se as entidades criadas são iguais às carregadas pelo sistema, ou seja, se se encontram na posição correcta se é a espécie correcta, etc. Nas áreas é verificado se os pontos dos vértices são os mesmos e que dimensão do mapa e a pontuação carregadas correspondem as estabelecidas.

Embora o carregamento dos tiles seja uma parte importante do carregamento do nível, este não se encontra presente o teste de carregamento, isto é devido ao facto de a API *pyTmx* ao carregar um *tile* devolve a imagem correspondente ao *tile*, que é obtida a partir do *tileSet* do mapa não tendo imagem para possível comparação sendo necessário carregar essa mesma imagem a partir dos mesmo, mas verificou-se que não era possível comparar imagem no pygame, ou seja, era possível verificar a posição e o numero dos *tiles* mas não era possível verificar se a imagem do *tile* era a correcta, sendo que o teste poderia dar positivo e os *tiles* do mapa estarem trocados.

7.2 Verificação e teste da colisão entre entidades

Para verificação das colisões foi carregado o mapa de teste e em seguida as entidades são postas numa posição e numa trajectória de modo a que durante o seu movimento elas colidissem.

Após o movimento é verificado se existe colisão durante o percurso das mesmas verificando as colisões devolvidas pelo mecanismo de colisões que neste caso terá de ter

duas colisões, ou seja, a colisão entre a primeira entidade e a segunda, e entre a segunda e a primeira.

7.3 Verificação e teste do reaparecimento das entidades

Para verificação do reaparecimento das entidades foi carregado o mapa de teste e em seguida a entidade que representa a tartaruga é morta, para facilitar o teste o tempo de reaparecimento foi diminuído para 1, ou seja, a entidade permanecerá morta durante um ciclo.

É efectuada a chamada ao método de reaparecimento e é verificado se a entidade continua morta, pois como a entidade tinha como tempo de reaparecimento 1, esta não volta a vida e em vez disso o tempo de reaparecimento é decrementado.

É efectuada a chamada ao método de reaparecimento, é então verificado se a entidade se encontra viva e com a sua nova posição dentro dos limites do mapa e numa área em que a entidade possa andar, neste caso como se trata de uma tartaruga, um animal anfíbio, esta só não pode andar na área de floresta, logo é verificado se na sua nova posição esta colide com alguma área de floresta.

8 Conclusões

Desde os primórdios da criação de jogos, estes evoluíram desde o simples jogo "*Pong*" até aos jogos com a complexidade dos nossos dias. Essa evolução, levou a um acréscimo de complexidade, que determinou ser necessário organizar o seu desenvolvimento de forma a reduzir riscos, pois com o acréscimo de complexidade também aumentam os custos de desenvolvimento.

Como o mercado dos jogos é um mercado muito competitivo e arriscado, e dado existirem muitas companhias e muitos jogos a saírem todos os meses, existe a necessidade constante de superar os competidores, seja criando algo novo ou algo melhor e assim tornar o jogo mais atractivo e criando mais vendas. Um dos aspectos que tornam um jogo apelativo é a realidade da sua jogabilidade, que nos leva ao tema da inteligência artificial nos mesmos, pois é esta que mais influencia a jogabilidade hoje em dia.

A IA nos jogos sofreu grande evolução ao longo dos anos, desde o uso de padrões, ao uso de vários outras técnicas como planeamento para criar o plano de acção de um agente, ao uso de máquina de estados finitos, para definir comportamentos mais complexos e modulares, técnicas essas que foram utilizadas e implementadas durante este projecto. Para além destas técnicas existem muitas outas que são usadas em conjunto para criar uma IA o mais realista possível, para fornecer ao jogador um maior e mais real desafio possível, aumentando assim a qualidade de jogo.

Ao longo do projecto foi possível criar uma síntese do processo de desenvolvimento de jogos, com recursos a várias ferramentas de engenharia de *software* como o uso de casos de utilização, que também foram utilizados neste trabalho para obter uma melhor caracterização do que era necessário elaborar, dando assim um melhor conhecimento de como proceder quando se desenvolve jogos, seja num cenário empresarial, seja num cenário de projecto individual.

Usando o conhecimento adquirido através da investigação efectuada. Foi então elaborado o jogo em questão, e assim atingidos os objectivos iniciais do projecto que era a criação dum jogo interactivo para aprendizagem de IA, começando de raiz passando por várias fases de desenvolvimento e tendo sempre em mente que esta deveria ser o mais expansível e modular possível para facilitar o seu uso futuro.

8.1 Trabalho futuro

Como em qualquer projecto que tenha um tempo limitado, existe sempre algum aspecto que tenha de ser sacrificado para que este possa ser concluído dentro do prazo e com uma boa qualidade.

Estes aspectos passam por várias funcionalidades que possam ter surgido durante o desenvolvimento do projecto, mas postas de parte para futuras implementações devido às restrições do mesmo e do seu carácter não crucial para o seu funcionamento.

Para além das funcionalidades adicionais existe sempre espaço para melhoria das funcionalidades implementadas. Ambos os aspectos serão sintetizados de seguida.

8.1.1 Mapas de influência

Como referido anteriormente os mapas de influência são um mecanismo usado para fornecer informação adicional aos algoritmos de planeamento, criando assim um melhor planeamento e agentes mais inteligentes.

A implementação dos mesmos passaria por criar um tipo de sensor que possivelmente usaria os existentes para obter a informação do mundo e assim criar uma percepção com base em mapas de influência.

8.1.2 Ferramentas de teste

O suporte de teste existente na plataforma é muito básico comtemplando apenas algumas mensagens na consola, isto é pouco intuitivo e de difícil compreensão e uso por parte de terceiros, levando a acréscimo de tempo no desenvolvimento e na aprendizagem do uso da plataforma.

Para melhorar esse aspecto surgiu a ideia de incorporar um modo de teste na plataforma, em que este tivesse um mecanismo de teste visual e de controlo temporal, ou seja, expor ao utilizador uma maneira intuitiva de ver o que a plataforma está a fazer e escolher o ritmo que esta o faz.

No teste visual passaria pela criação de uma nova área na tela de jogo dedicada ao teste, das seguintes formas:

- Percepção – este iria mostrar o ambiente da forma que o agente o vê permitindo verificar qual a informação que o agente possui sobre o ambiente para melhor compreensão sobre as suas decisões.
- Acções – este iria mostrar as acções que o controlador determinou para o agente em questão, no caso do controlador deliberativo, para além de mostrar a acção seguinte, mostra também o plano, ou seja, mostra a sequência de acções, e a representação das mesmas é feita em conjunto com a percepção.
- Máquinas de estados finitos (MEF) – este iria mostrar o diagrama da MEF de modo visual, pondo em evidência o estado actual e as transições efectuadas, isto permite verificar se a máquina de estados funciona correctamente, ou seja, se as transições são bem efectuadas, permite também verificar se o comportamento do agente é adequando para aquele estado.
- Mapa de influência (MI) – com a adição de MI ao projecto será também necessário um teste visual para verificar se este é bem construído e se o agente usa essa informação adicional correctamente, visto que este é aliado da percepção do agente, e estas devem ser mostradas em conjunto.

O teste visual será efectuado sobre a unidade seleccionada, podendo o utilizador ter a liberdade visual de mudar de unidade que analisa durante o decorrer do ciclo de jogo, permitindo assim, a análise de vários comportamentos/algoritmos.

O teste visual pouco vale se o utilizador não tiver tempo de o analisar pois o ciclo de jogo é demasiado curto para que o utilizador consiga concluir o que quer que seja sobre o que está a ver, por isso é necessário implementar um controlo temporal para que este possa controlar o fluxo de jogo de forma a analisar as decisões dos agentes, dando a opção de controlar o ritmo de jogo acelerando ou abrandando, podendo ainda corrê-lo passo a passo.

8.1.3 Adição de novos elementos ao jogo

Embora o jogo já tenha alguns elementos que promovem comportamentos algo diversos, é sempre possível adicionar mais, para enriquecer os comportamentos existentes e criar novos, fazendo com que a plataforma seja mais rica. Alguns exemplos

dos mesmos são os seguintes: a adição de caçadores que coloquem armadilhas, e assim poder ter o comportamento de atrair ou afugentar a presa em direcção do mesmo, contendo assim um algoritmo de previsão, para tentar determinar a acção da presa para a poder levar para algum sítio.

A adição de abrigos para as presas, ou seja, sítios em que as presas possam estar em segurança enquanto não têm um determinado patamar de fome, isto permite o aparecimento de dois comportamentos distintos, a presa como tem um lugar seguro irá tentar não se afastar muito do abrigo a não ser que seja absolutamente necessário e no caso de não encontrar alimento não proximidades. O predador no caso de não ter muita fome poderá aprender que aquele sítio tem uma presa e poderá ficar a espera que ela saia, durante um determinado tempo, pois é-lhe desvantajoso estar lá sempre correndo o risco da presa não sair da toca num espaço de tempo útil, em vez de estar a caçar.

Durante projecto, para facilitar o comportamento das personagens não foram implementados animais omnívoros, isto devido ao facto da decisão sobre gastar energia ao perseguir a presa ou ir buscar alimento não ser simples, pois cada um tem as suas vantagens e desvantagens, enquanto a presa é mais difícil de apanhar esta alimenta mais enquanto o alimento é mais acessível, mas alimenta menos, dificultando a criação da MEF dos mesmos.

8.1.4 Algoritmo de intercepção

No algoritmo de perseguição implementado o predador persegue a presa indo sempre atrás dela, isto não é necessariamente a melhor solução pois caso o predador não seja mais rápido que a presa este não conseguirá apanhá-la, uma solução melhor seria em vez de o predador perseguir a presa dirigir-se para o ponto de intercepção.

O facto do predador se dirigir para o ponto de intercepção permite que este possa apanhar a presa mesmo que esta não seja mais lenta que o mesmo, o algoritmo de intercepção tem alguma complexidade pois é necessário ter em conta a distância, direcção e velocidades de ambos para determinar o mesmo.

Este também é útil para o caçador pois este não deve disparar para onde a presa se encontra mas para onde ela irá se encontrar, porque quando a bala lá chegar a presa poderá já lá não estar, deve então calcular o ponto de intercepção da bala com a presa e disparar nesse sentido.

8.1.5 Comportamento de grupo

É usual na natureza elementos de certas espécies andarem em grupo para se protegerem, por exemplo de um rebanho. Este tipo de comportamento embora não tenha grande riqueza por si só, traz algum realismo ao jogo, porque retracta melhor a realidade.

Permite implementar o cenário da necessidade de ter de haver mais do que um predador para que seja possível caçar um elemento de um desses grupos, o que nos leva a necessidade de existir cooperação entre predadores e na criação de agentes cooperativos que é algo complexo mas interessante do ponto de vista da IA.

8.2 Considerações finais

Os jogos e a inteligência artificial continuam em constante evolução, com o uso de novas técnica e o melhoramento das existentes, na busca de um jogo que consiga ser praticamente real e acompanhado por uma verdadeira inteligência artificial, ou seja, uma inteligência que não seja possível distinguir de um humano.

Este projecto serviu para fazer uma análise sobre o processo de desenvolvimentos de jogos e de algumas técnicas de IA usadas nos mesmos, usando esse conhecimento para criar uma plataforma de ensino de inteligência artificial.

9 Bibliografia

[Bethke, 2003] Erik Bethke, Game Development and Production. Wordware Publishing, Inc. 2003

[Blow, 2004] J. Blow, Game Development: Harder Than You Think. ACM Queue, 2004

[Bourg, Seeman, 2004] David M. Bourg, Glenn Seeman. AI for Game Developers, O'Reilly, 2004

[Buckland, 2005] Mat Buckland, Programming Game AI by Example. Wordware Publishing, Inc. 2005

[Leffingwell, Widrig, 1999] Dean Leffingwell, Don Widrig, *Managing Software Requirements*. Addison Wesley, 1999

[Lindeijer, 2013] Thorbjørn Lindeijer, Tiled Map Editor 0.9.0. http://www.mapeditor.org/, 2013

[josmiley, 2012] josmiley, MenuSystem 2.2. http://www.pygame.org/project-MenuSystem-2031-.html, 2012

[McGugan, 2007] Will McGugan, Beginning Game Development with Python and Pygame From Novice to Professional. Apress, 2007

[Pottinger, 2013] Dave C. Pottinger, Terrain Analysis in Realtime Strategy Games, http://astarimpl.googlecode.com/svn/trunk/pathfinding%20-%20samples/pottinger.doc, 2013

[Rabin, 2002] Steve Rabin, AI Game Programming Wisdom. Charles River Media, 2002

[Reenskaug, 1979] Trygve Reenskaug, MODELS-VIEWS-CONTROLLERS, http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf , 1979

[Russel, Norvig, 2010] Stuart J. Russell, Peter Norvig, Artificial Intelligence A Modern Approach (3rd Edition), Prentice Hall, 2010

[Schell, 2008] Jesse Schell, The Art of Game Design. Elsevier Inc. 2008

[Schwab, 2004] Brian Schwab, AI Game Engine Programming. Charles River Media, Inc. 2004

[Sweigart, 2012] Albert Sweigart, Making Games with Python & Pygame. http://inventwithpython.com/pygame. 2012

Apêndice I – Uso do *Tiled*

Como referido anteriormente o jogo possibilita a criação de novos níveis, tal é feito através da aplicação *Tiled Map Editor*. Os níveis são constituídos por um *tiledMap* e os objectos de jogo.

A criação de um novo mapa começa pela escolha de tamanho do mesmo, quantos tiles tem de altura e de largura e o tamanho de cada *tile*, os *tiles* fornecidos têm dimensões de 32x32 pixéis. Em seguida é necessário ter acesso aos *tiles* em si, denominado *tileset*, para adicionar o *tileset* basta ir a opção *Map->addexternal tileset* e escolher o ficheiro "*tileset.tsx*".

Tendo o mapa em branco e o *tilset* disponível, para criar o mapa do jogo basta elaborá-lo carregando no tile e em seguida carregar na posição desejada. É necessário a criação dos objectos de jogo. Os objectos de jogo são as áreas, as entidades e a pontuação. Para adicionar objectos é necessário adicionar uma camada de objectos indo à opção *Layer-> add object layer*, após criada a camada é necessário inserir os objectos na mesma.

Para criar uma área é necessário criar um polígono no mapa e em seguida dar-lhe um tipo, por exemplo um polígono de uma floresta tem o tipo "*florest*". Para criar uma entidade é necessário criar um rectângulo na posição desejada e como tipo a sua espécie, por exemplo a gaivota tem como tipo "*seagull*", adicionalmente se a entidade for a entidade do jogador esta terá como nome "*Player*" e finalmente o score é um rectângulo no qual a sua posição não é relevante, bastando que tenha como tipo "score" e no seu nome a pontuação necessária para a passagem de nível.