UNIVERSIDADE DE LISBOA

Faculdade de Ciências

Departamento de Informática

DEVELOP-FPS: Ferramenta de desenvolvimento de comportamentos baseados em regras para personagens virtuais

Bruno Filipe Martins Correia

MESTRADO EM ENGENHARIA INFORMÁTICA

Especialização em Sistemas de Informação

2011

UNIVERSIDADE DE LISBOA

Faculdade de Ciências

Departamento de Informática

DEVELOP-FPS: Ferramenta de desenvolvimento de comportamentos baseados em regras para personagens virtuais

Bruno Filipe Martins Correia

DISSERTAÇÃO

Trabalho orientado pelo Prof. Doutor Luis Manuel Ferreira Fernandes Moniz

e co-orientado por Paulo Jorge Cunha Vaz Dias Urbano

MESTRADO EM ENGENHARIA INFORMÁTICA

Especialização em Sistemas de Informação

2011

Agradecimentos

Devo profunda gratidão pela conclusão deste projecto à Faculdade de Ciências da Universidade de Lisboa, em especial ao Professor Doutor Luis Moniz e ao Professor Doutor Paulo Urbano pela sua orientação e conhecimento para o sucesso deste projecto.

A todos os meus colegas de laboratório (LabMAg), pelo suporte incondicional e camaradagem.

Por último mas não o menos importante aos meus pais e irmão pelo seu amor interminável e suporte, tanto emocional como monetário e à minha namorada pela sua compreensão e dedicação.

A todos os entusiastas pela inteligência artificial.

Resumo

Os videojogos não param de evoluir, fornecendo ao jogador novos desafios para que este se mantenha interessado e motivado para experimentar novas experiências. Estes desafios, normalmente, passam pela interacção do jogador com os NPCs (*Non-Player Characters*), sejam eles de combate, de cooperação ou mesmo de para troca de informação.

Com a evolução dos computadores existe oportunidade e recursos para investigar e aplicar novas técnicas de inteligência artificial no desenvolvimento de comportamentos para os NPCs, tornando-os mais complexos. Para que isto seja possível, é necessário munir os investigadores com ferramentas profissionais com capacidade para construir, testar e validar os comportamentos desenvolvidos.

Este trabalho apresenta uma arquitectura genérica para ferramentas de desenvolvimento de comportamentos escritos em *Jess* para personagens virtuais inseridas num ambiente de videojogo do género *First-Person Shooter* (*Unreal Torunament 2004*). A ferramenta desenvolvida, com base nesta arquitectura, permite realizar o *debugging* e a avaliação de comportamentos individuais ou cooperativos fornecendo capacidades para preparar cenários de teste; monitorizar e controlar todos os intervenientes; realizar experiências; testar e comparar os comportamentos.

Palavras-chave: First Person Shooter, scripts à base de regras, ferramentas de desenvolvimento, inteligência artificial.

Abstract

Video games are constantly evolving providing new challenges for the player to remain interested and motivated to try new experiences. These challenges usually involve player interaction with NPCs (Non Player Characters), they fight, cooperate to overcome obstacles, or communicate with each other to enrich the storyline.

With the growing resources available with the evolution of computers there is an increasing interest in investigating and applying new techniques of artificial intelligence in the development of ever more complex behaviors for NPCs. To make this possible, it is necessary to equip researchers with professional tools to build, test and validate the behaviors developed.

This paper presents a generic architecture development tool to achieve a NPC behavior written in Jess, inserted in an environment of the video game First-Person Shooter genre (Unreal torunament 2004). The tool developed, based on this architecture, allows debugging and evaluation of individual or cooperative behaviors, providing capabilities to prepare test scenarios, monitor and control all NPCs; conduct experiments; test and compares the behaviors.

Keywords: First Person Shooter, Rule based Scripts, development software tool, artificial intelligence.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

"*The field of Game AI is still the "Wild West." We're just scratching the surface of what's possible, and there are still many unsolved problems, which leave open huge opportunities to innovate and do cuttingedge work that really could have impact on the entire industry."* — Jeff Orkin [AiGameDev10]

(O campo de IA nos videojogos continua um "Oeste Selvagem". Estamos apenas a raspar a superfície do que é possível, e existem ainda vários problemas por resolver, que abrem grandes oportunidades para inovar e criar tecnologia de ponta que poderá ter impacto na indústria – Jeff Orkin)

Nos anos que correm, com a evolução a nível computacional, das placas gráficas, dos processadores e das memórias RAM, temos observado um grande desenvolvimento nos videojogos, principalmente na sua componente gráfica. A componente da inteligência artificial deve acompanhar esta evolução e levar aos jogadores novas técnicas e desafios de forma a enriquecer a sua experiência enquanto jogam.

A inteligência artificial pode tornar o jogo mais interessante e atraente [AIGames02], logo podendo ser, em parte, responsável pelo sucesso de um videojogo. Na verdade, já contribuiu para o sucesso do *Pacman*, com uma componente de inteligência artificial simples devido aos recursos disponíveis na altura, até ao *Black and White*, onde emprega técnicas avançadas como árvores de decisão combinadas com redes neuronais [AiGameDev07]. O *Pacman* ainda hoje é utilizado para competições [PacmanCompetition11] e as inovações introduzidas pelo *Black and White* são reconhecidas nas comunidades de inteligência artificial [AiGameDev07].

Com a evolução dos videojogos, o jogador mais experiente já não aceita comportamentos como os existentes no "*space invaders*", com padrões facilmente identificáveis. Ele espera que um videojogo consiga proporcionar experiências interessantes, diferentes e desafiantes.

1.1 Motivação

As plataformas de desenvolvimento orientadas para os videojogos fornecem novos espaços para desenvolver e aplicar novas técnicas de inteligência artificial nos jogos

comerciais. Os *Toolkits* de desenvolvimento de jogos estão a começar a fornecer suporte no desenho de comportamentos para os *non-player characters'* (NPCs), principalmente através de linguagens protegidas por direitos autorais (*UnrealScript* no *UnrealEngine* [UDK11]), *open-source* ou linguagens gratuitas (*Lua* no *World of Warcraft* [WoW10]) ou bibliotecas de comportamentos (*PandAI* no *Panda3D engine* [Panda3D11]). Alguns destes *toolkits* suportam o desenho e a integração do NPC no jogo através de soluções proprietárias e limitadas. A maioria das companhias de videojogos tem as suas ferramentas de desenvolvimento, que não são disponibilizadas à comunidade.

As alternativas de baixo custo, *open source* e *shareware* colocam os seus esforços para suportar motores de jogo, componentes gráficas, resolver problemas como simulação física, detecção de colisões e animação de personagens. As ferramentas que assistem no desenvolvimento dos comportamentos dos NPCs são normalmente omitidas.

A existência de uma ferramenta de desenvolvimento totalmente orientada para o *debug* da inteligência artificial seria importante para a indústria dos videojogos, de modo a que se possa testar os comportamentos com o intuito de os melhorar. Estas ferramentas devem fornecer funções que permitam: recriar facilmente situações para testar e comparar comportamentos; monitorizar em tempo real o estado do NPC e alterar remotamente o seu comportamento; executar o seu comportamento passo a passo fazendo o *debugging* do código, visualizando o seu estado e acedendo a mais informação executando funções sobre a percepção do agente e sobre o estado do meio-ambiente que o rodeia. Seria também útil permitir a possibilidade de poder criar situações de teste dinamicamente, em tempo real, fornecendo comandos aos NPCs para se dirigirem para uma dada posição, controlando os seus movimentos com precisão e testando os seus comportamentos passo a passo. Estas funcionalidades não existem na maioria das *toolkits* de desenvolvimento para videojogos quando precisamos de construir, testar e executar os comportamentos dos NPCs.

Várias plataformas de desenvolvimento para videojogos, nomeadamente o UDK [UDK10], Unity 3D [Unity11], e o Panda3D [Panda3D10] fornecem um conjunto de funcionalidades que promovem o *debugging*, como por exemplo um sistema de *logs* que consiste basicamente num sistema de mensagens que o programador coloca no seu *script* para acompanhar o que está a acontecer. Esta funcionalidade mostra-se muitas vezes penosa quando é gerada uma grande quantidade de *logs*.

A ferramenta que é disponibilizada para ajudar a desenvolver comportamentos é, em geral, um editor de texto. Alguns editores fornecem funções como a indentação do código e alteração da cor de algumas palavras-chaves do *script* apenas para facilitar a leitura do mesmo. Porém, alguns *toolkits* fornecem *plugins* para IDEs', que possuem sistemas de *debugging* integrados na ferramenta com funcionalidades como definição de

breakpoints, visualização de variáveis, interrupção da execução do jogo ou realizar um simples *step* linha por linha, a adição *MonoDevelop* para o *Unity* 3D [MonoDevelop10] é um exemplo destas ferramentas.

Neste tipo de ferramentas quando ocorre algum *bug*, o procedimento comum é interromper o *script* e observar o *output* produzido pelo *bug*. Existem interpretadores que fornecem mensagens identificando o tipo de erro e a sua localização no código, mas sem nenhuma ferramenta para testar e monitorizar as componentes, o programador é responsável por efectuar o *debug* e testar o seu próprio código. Por exemplo, a plataforma de desenvolvimento *UDK* [UDK10] fornece um mecanismo para o *debug* baseado em registos de mensagens produzidas no *script* para que se possa acompanhar a execução do mesmo.

A existência de ferramentas que fornecem um ambiente profissional para suportar todo o processo desenvolvimento da componente de inteligência artificial reduzirá dramaticamente o tempo despendido, libertando o programador para produzir melhor código.

Hoje em dia, os comportamentos são cada vez mais focados em acções cooperativas entre os NPCs para que estes possam ultrapassar obstáculos ou concluir certos objectivos tornando a experiência mais rica para o jogador. Devido à complexidade de gerir vários comportamentos, as ferramentas de desenvolvimento também deverão ser capazes de fornecer métodos para tornar a gestão mais fácil, como a monitorização e permitir o controlo de todos os intervenientes no jogo. Além das ferramentas de desenvolvimento, também é necessário investigar e promover novos métodos para comportamentos dos NPCs que os tornem plausíveis ao olho do jogador, oferecendo-lhe uma maior imersão[1].

Os sistemas baseados em regras [Friedman03] são frequentemente utilizados na Inteligência Artificial e já foram aplicados em vários videojogos [Cavazza00] como o *Civilization: Call to power* e o *Rainbow Six*.

Actualmente, a maior parte do código responsável pela inteligência artificial de um videojogo consiste na implementação de regras para os comportamentos para os actores intervenientes, sejam eles carros de corrida, jogadores de futebol, inimigos e por aí adiante. A implementação de regras dos comportamentos para os actores (agentes ou personagens virtuais) é requerido por todos os videojogos e é uma tarefa habitual para os programadores da componente de inteligência artificial. As regras são para a inteligência artificial do videojogo como os polígonos são para os gráficos [Wright00].

[1] Imersão é considerada como sendo a capacidade de suspender a descrença em relação ao videojogo, levando o jogador a sentir-se completamente envolvido no ambiente produzido. [Woyach08].

As linguagens de programação baseadas em regras são muito diferentes das linguagens de programação imperativas convencionais. Existem apenas factos (input), regras (o programa) e o interpretador de regras, este responável por procurar, seleccionar e activar as regra, com base no *input*. As regras activadas (activações) são colocadas numa estrutura de dados especial, a agenda, aparecendo ordenadas; a primeira activação da agenda será executada. Para a ordenação existem critérios de resolução de conflitos. Note que a mesma regra pode dar origem a várias activações.

As regras são expressões se-então ou condição-acção e podem aparecer por qualquer ordem no programa. Sempre que a condição da regra é satisfeita pela base de factos, a regra é activada e se for seleccionada, a sua parte direita é executada (diz-se também que a regra é disparada). Existe um ciclo de activação-selecção-disparo de regras (a execução do programa) até que mais nenhuma regra seja activada. O estado da base de factos antes e após o fim do disparo das regras corresponderá ao *input* e ao *output* do programa respectivamente.

Sistemas à base de regras possuem qualidades que os tornam atractivos, tais como: a inteligibilidade das regras que tem a ver com o seu aspecto declarativo, a sua modularidade e consequente facilidade de adição e remoção. A razão para a sua não adopção prende-se com a diferença em relação à execução sequencial do código imperativo tradicional - o controlo do disparo das regras é mais imprevisível e difícil. Mas, devido à sua singularidade é necessário que as plataformas de desenvolvimento, que optem por esta linguagem, possuam funcionalidades que tornem acessível o acompanhamento da execução dos *scripts* baseados em regras, promovendo o *debugging* das regras. Para isso é necessário ser capaz de parar o jogo e mandar executar passo-a passo cada ciclo de activação-selecção-disparo das regras, acompanhando o estado da base de factos, o conteúdo da agenda e os efeitos do disparo de cada regra.

Os videojogos chamados *First Person Shooter* (FPS) são jogos de acção, normalmente violentos, muito populares desde o sucesso de *Doom* em 1993 e que são jogados na perspectiva da primeira pessoa: o jogador experiencia a acção através dos olhos do protagonista.

Os FPSs envolvem um *avatar*, um ambiente 3D, e em geral, sendo um jogo de combate cujo objectivo é matar o maior número de inimigos possível, envolvem também um conjunto de diferentes armas e munições, uma variedade de oponentes. A variante dos FPS que suportam o jogo em equipa é um tipo muito popular onde equipas de 2 ou mais jogadores (não necessariamente controlados por jogadores humanos) competem para atingir os mesmos objectivos: eliminar os seus inimigos. Actualmente,

um dos desafios da inteligência artificial para videojogos está na realização de formações e estratégias para grupos de agentes, nomeadamente em videojogos do género FPS. Pretende-se que estes movimentem-se de forma coordenada e a realizem acções credíveis em conjunto perante dadas situações [AiGameDev11]. Numa equipa é preciso que haja alguma coordenação de modo a que uma equipa não seja apenas um grupo de jogadores sem uma estratégia e que não comunicam nem interagem. Assim, as ferramentas de desenvolvimento de comportamentos para videojogos de equipa terão de lidar não apenas com as percepções e acções dos jogadores mas também lidar com as suas interacções em tudo o que envolve a coordenação.

1.2 Objectivos

Este trabalho tem dois objectivos: a concretização de uma ferramenta para facilitar o desenvolvimento e testar comportamentos inteligentes realizados numa linguagem à base de regras; e o desenvolvimento de comportamentos individuais e de grupo testando as funcionalidades da ferramenta de desenvolvimento.

Propomos uma arquitectura genérica que suporta o desenvolvimento de comportamentos para personagens autónomas escritos numa linguagem à base de regras para um ambiente de videojogo do género FPS para computador. Poderia ser qualquer outro jogo mas este tipo oferece uma vantagem: não existem exemplos, pelo menos não foram encontrados, de *scripts* escritos numa linguagem de programação baseada em regras nem para controlar jogadores individuais nem em equipa.

A ferramenta de desenvolvimento irá permitir realizar experiências que possam ser repetidas o necessário número de vezes, recolher dados tanto para comportamentos individuais como para cooperativos de forma a avaliar o seu desempenho, oferecendo métodos para criar cenários de teste, monitorizar e controlar todos os agentes intervenientes na experiência.

1.3 Contribuições

Foi criada a DEVELOP-FPS que é uma ferramenta de desenvolvimento de NPCs para videojogos FPS, aplicada à plataforma UDK e usando o jogo *Unreal Tournament 2004* e que é considerada uma das melhores plataformas de desenvolvimento para videojogos [IGN09]. Na DEVELOP-FPS, os *scripts* que definem os comportamentos são escritos na linguagem de programação baseada em regras *Jess* [Friedman03].

A ferramenta fornece ao programador um conjunto de funcionalidades:

1. Monitorização do estado do jogador em tempo real – Para que o utilizador possa acompanhar tudo o que o NPC estiver a percepcionar, por exemplo se está a ver o inimigo.

2. Monitorização do estado do *Jess* – Como os comportamentos são elaborados em *Jess*, um mecanismo que permita monitorizar o seu estado é importante para que o utilizador possa acompanhar o conteúdo da base de factos, a agenda (lista ordenada de activações) e as regras que são disparadas.
3. Possibilidade de interromper a execução da lógica dos agentes, sejam todos ou apenas um – assim o utilizador pode interromper a experiência para monitorizar e controlar qualquer agente interveniente na experiência. Através de uma consola é possível enviar comandos a qualquer dos agentes de modo a fazer-lhe perguntas sobre o estado corrente do jogo, visto da sua perspectiva, e alterar o seu comportamento.
4. Criação de mapas 2D interactivos com visualização dos *waypoints* e os jogadores, que permitem ter uma panorâmica "aérea", o que é muito útil para saber como se dispõem os jogadores. Clicando num mapa particular a um agente, o agente irá encontrar um caminho para atingir o *waypoint* seleccionado e se existir uma trajectória possível para o objectivo ele irá deslocar-se para lá.
5. Preparação manual de cenários de teste, que seja de utilização fácil, dando total controlo sobre os agentes – para que o utilizador não tenha que esperar para que uma situação pretendida aconteça, evitando assim a codificação extra para que os agentes configurem a cena pretendida. Em tempo real, o utilizador pode interromper todos os agentes e através de uma interface adequada, pode colocá-los numa dada galeria.
6. Capacidade de realizar várias experiências e recolha de dados estatísticos - para avaliar e analisar vários comportamentos, que podem ser assim comparados.

Ir-se-á ilustrar e avaliar a DEVELOP-FPS para criar, testar e comparar três comportamentos individuais, do mais simples ao mais complexo. Finalmente, vai ser descrito o desenvolvimento de um comportamento de equipa, formada por 4 NPCs, que são comandados por um deles e que se movimentam num tipo de formação em forma de diamante que servirá de exemplo como aplicação da ferramenta de desenvolvimento em desafios actuais.

1.4 Estrutura do documento

Este documento está organizado da seguinte forma:

- Capítulo 2 – Trabalho relacionado

  Neste capítulo são contextualizadas algumas das técnicas de inteligência artificial aplicadas em videojogos. Também se apresentam algumas plataformas de desenvolvimento de videojogos do género FPS descrevendo as funcionalidades que oferecem para promover o desenvolvimento de comportamentos inteligentes para agentes virtuais.

- Capítulo 3 – Trabalho realizado

  Neste capítulo é feita inicialmente a descrição da DEVELOP-FS, a sua arquitectura, as várias funcionalidades que são ilustradas com vários exemplos. Fecharemos o capítulo aplicando a ferramenta que construímos no desenvolvimento dos comportamentos individuais e um e equipa.

- Capítulo 4 – Discussão e trabalho futuro

  Neste capítulo são apresentadas as conclusões do trabalho elaborado referindo os benefícios que a ferramenta desenvolvida traz para o desenvolvimento de comportamentos e propõe etapas para o trabalho futuro.

- Bibliografia

- Apêndice – Manual do utilizador

  Neste apêndice é apresentado o manual do utilizador da ferramenta desenvolvida, as regras de instalação e execução da ferramenta além de uma descrição pormenorizada de todas as funcionalidades.

Capítulo 2

Neste capítulo apresentaremos um conjunto de técnicas usadas na componente de inteligência artificial em videojogos, que se referem a três aspectos decisivos dos videojogos: i) a representação do meio ambiente; ii) o controlo da movimentação dos agentes, seja individual ou em grupo; e iii) a definição do comportamento. No que concerne à definição do comportamento serão referidas diferentes técnicas utilizadas no processo de decisão da acção, na selecção do caminho (*pathfinding*), na análise táctica e finalmente na coordenação de um grupo.

Em seguida, serão apresentadas algumas plataformas de desenvolvimento de videojogos comerciais que são representativas do que existe no mercado, nomeadamente o UDK, o Unity 3D e o Panda3D. Serão descritas as suas funcionalidades de monitorização e controlo do comportamento dos NPCS, bem como os métodos de *debugging* que disponibilizam.

Para fechar, será apresentada a plataforma de jogos FPS *Unreal Tournament 2004* que foi utilizada neste trabalho, bem como a interface de programação *Pogamut*, que permite a programação, na linguagem Java, dos agentes para a *Unreal Tournment 2004*. O *Jess*, também escrito em Java, que é a linguagem de programação baseada em regras, e que foi utilizada para escrever os *scripts* dos agentes, será também objecto de destaque. Finalmente, será feita uma descrição da ferramenta AGLIPS, que foi uma fonte de inspiração para este trabalho, embora para um contexto completamente diferente: o desenvolvimento de comportamentos de robôs em grupo.

2.1 Técnicas usadas em inteligência artificial para videojogos

A maioria dos videojogos segue uma arquitectura genérica (ver figura 1), onde tradicionalmente as componentes responsáveis pela física e gráficos ocupam a maioria dos recursos, quer a nível de desenvolvimento quer a nível da execução do próprio jogo. A componente de IA é apenas um dos elementos desta arquitectura que é responsável por fornecer mecanismos para que se possam construir agentes virtuais.

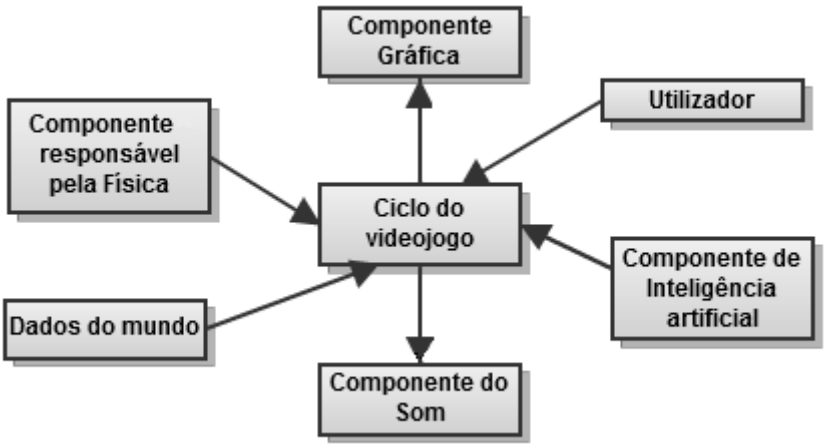

Figura 1: Arquitectura genérica de um videojogo.

A figura 1 um descreve as componentes genéricas de um videojogo em que a componente de Inteligência Artificial é apenas um dos elementos. Esta secção será dedicada à apresentação de diversas técnicas específicas à componente de inteligência artificial, as quais dividimos em três grupos: a representação do meio ambiente, a movimentação e o comportamento.

2.1.1 Representação do meio ambiente

O meio ambiente ou mundo virtual pode ser dinâmico ou estático:

- Dinâmico - envolve alterações no cenário, o que muitas vezes origina novos caminhos/ novos dados para a estrutura de dados. Por exemplo, um ambiente que contêm objectos dinâmicos (por exemplo, portas com armadilhas) que possam obstruir caminhos onde o agente poderia passar.
- Estático – o cenário, do princípio ao fim do videojogo, é inalterado. Por exemplo, um ambiente onde todos os objectos não se movimentam.

A representação corresponde à forma como o mundo virtual é percebido pelo agente e, em geral, não depende de possuir uma natureza estática ou dinâmica. A representação do meio ambiente é fundamental para calcular as trajectórias quando o agente precisa de se movimentar, evitando assim colisões com os objectos do cenário e também para obter informação sobre os locais onde se pode refugiar. [Tozour04]

Existem várias alternativas de representação de espaços virtuais para videojogos:

- **Grelhas regulares (*Regular grids*)** podem ser quadradas (ver Figura 2), rectangulares, hexagonais ou triangulares. São boas para representação de espaços de duas dimensões ou 2D, sendo uma estrutura de dados eficiente

já que o acesso aos dados de qualquer célula da grelha é imediato. Uns dos problemas é que podem dar origem a movimentos que são pouco naturais por parte do agente, visto que os caminhos são compostos por traços ortogonais.

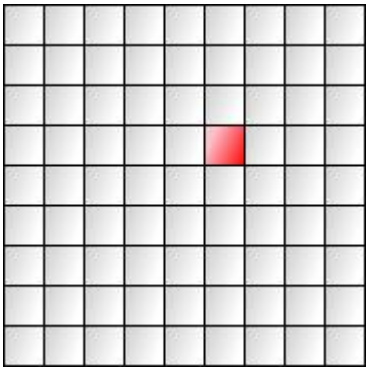

Figura 2: Grelha regular

- **Grafos de cantos (*Corner graphs*)** consistem em pontos (*waypoints*) colocados nos cantos dos obstáculos. Os pontos dos cantos são gerados automaticamente mas esta solução obriga a que o agente se movimente apenas entre estes pontos, dando a sensação que o agente anda "colado à parede".
- **Grafos de pontos** (***Waypoints graphs***) (ver Figura 3) são semelhantes aos gráficos de cantos, excepto que os pontos são usualmente inseridos no meio das salas. São bastante populares na indústria de videojogos pois consegue-se muito facilmente criar espaços em 3 dimensões (3D). Os pontos podem ser colocados manualmente ou podem ser pré-processados para serem colocados em locais ideais no mundo do jogo. Podemos considerar os grafos de cantos como um exemplo automático deste tipo de grafo. Para obter uma melhor qualidade dos caminhos é necessário ter mais nós (pontos) inseridos numa dada área. No entanto, um maior número de pontos exige naturalmente mais espaço para o armazenamento da informação e mais tempo necessário para o cálculo dos caminhos.

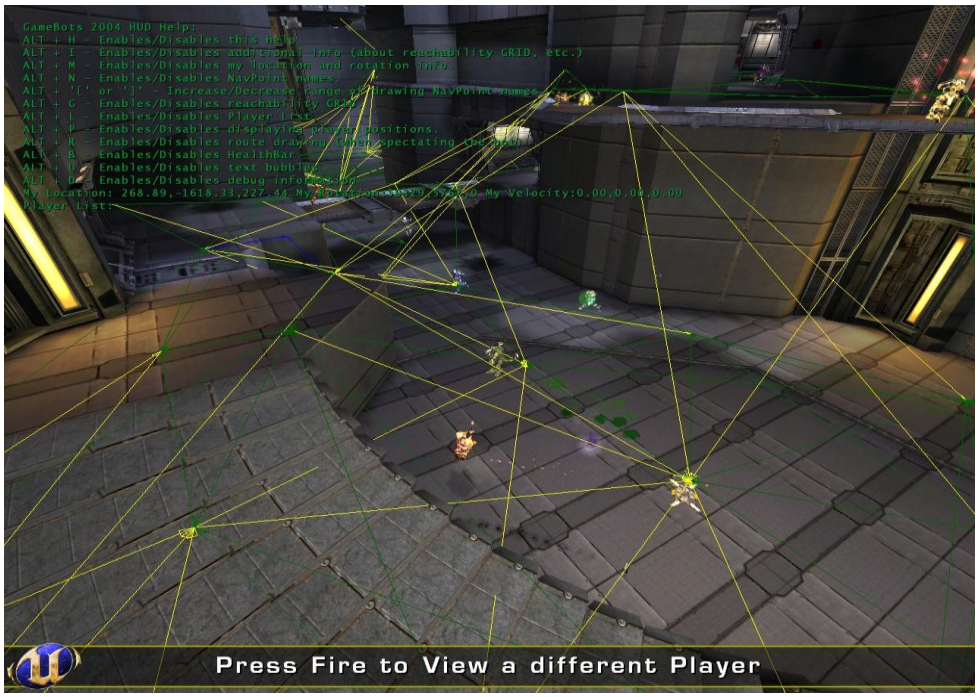

Figura 3: Visualização de grafos de pontos

- **Grafos de pontos circulares (*Circle-Based Waypoints graphs*)** são uma variação dos gráficos de pontos só mas em que é gerada uma circunferência (ver Figura 4) toda ela navegável à volta de cada ponto, fornecendo assim mais informação sobre o espaço navegável para além de um mero ponto.

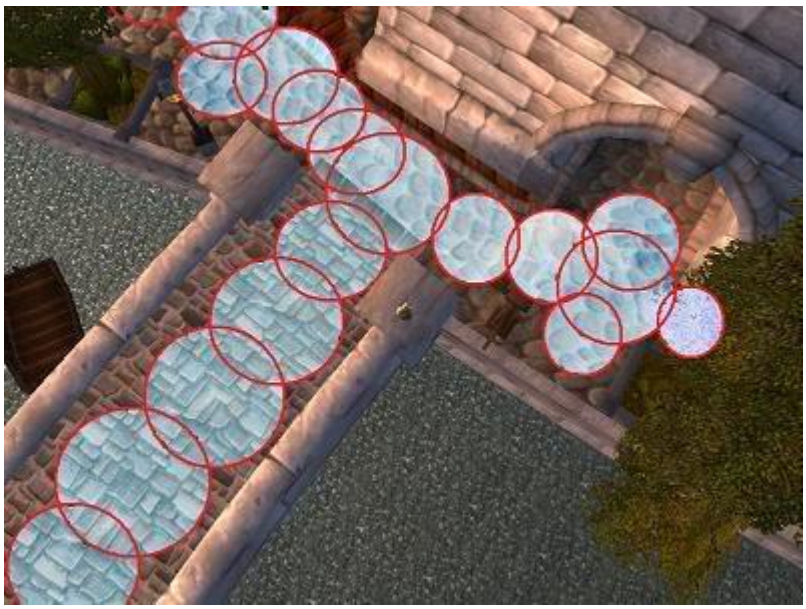

Figura 4: Visualização de grafos de pontos circulares

- **Volumes de preenchimento de espaços (*Space-filling volumes*)** são como os gráficos de pontos circulares só que em vez de usarem circunferências usam rectângulos.
- **Malhas de navegação (*Navigation Mesh ou NavMesh*)** [Snook00, White02] são representações em que se cobre as superfícies navegáveis do mundo virtual com polígonos convexos (ver Figura 5). Os polígonos convexos permitem garantir que cada agente se possa movimentar em qualquer ponto contido nesse polígono. Assim, permite procurar caminhos óptimos que não se limitem apenas à navegação entre pontos mas a todo o espaço dos polígonos. Cada polígono é armazenado como um nó na estrutura de dados. A principal desvantagem é a grande quantidade de espaço de armazenamento necessário.

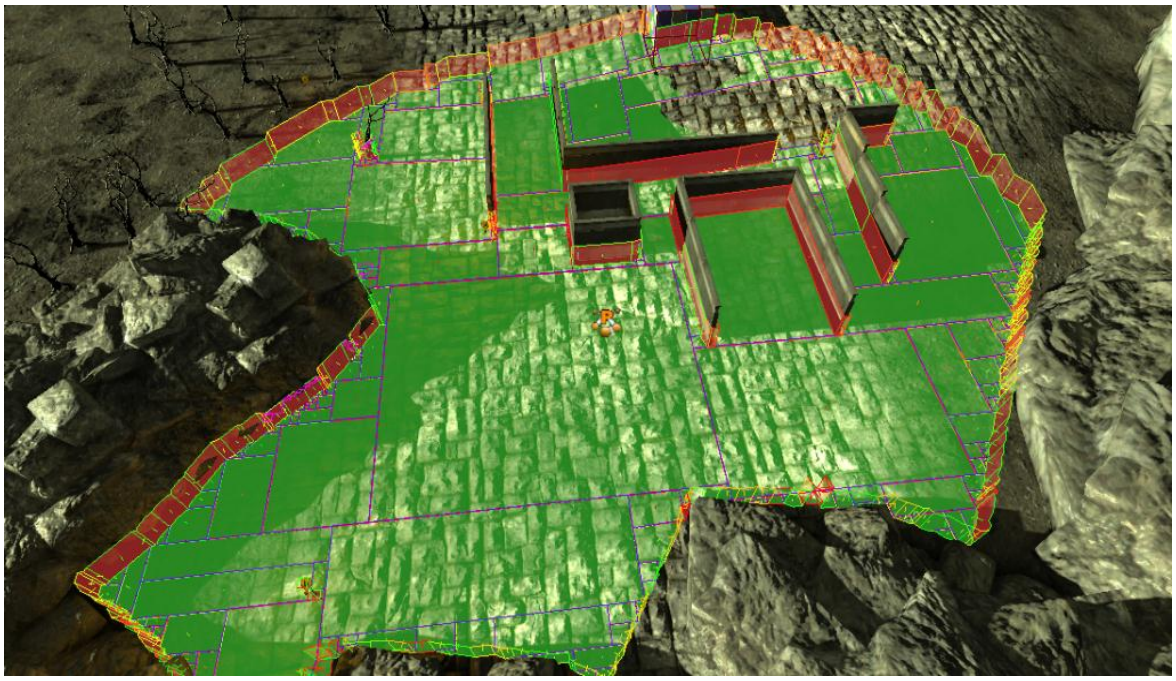

Figura 5: Malha de navegação

Na tabela 1 enumeramos as vantagens e desvantagens de cada um dos tipos de representações do mundo virtual. A escolha do tipo de representação num videojogo vai depender dos requisitos do videojogo (se o ambiente é 3D, como é que os agentes se movimentam) e do computador (se o processador é bom, se há muita memória).

Tabela 1: Vantagens e desvantagens das representações

	Vantagens	Desvantagens
Grelhas regulares	Fácil de implementação, Acesso aleatório aos dados, Lidam bastante bem com espaços 2D	Pode originar num movimento por parte do *bot* pouco atractivo.
Grafos de cantos	Não requer muito espaço de armazenamento	Limita a movimentação do *bot*.
Grafos de pontos	Criam espaços 3D muito facilmente.	Colocação, normalmente manual e pode levar caminhos erróneos
Grafos de pontos circulares	O agente pode percorrer um maior espaço	Não cobre todos os espaços navegáveis.
Volumes de preenchimento de espaços	O agente pode percorrer um maior espaço	Não cobre todos os espaços navegáveis.
Malhas de navegação	Actualmente, a melhor representação de um espaço 3D	Espaço de armazenamento, Difícil implementação

2.1.2 Movimentação

O movimento do agente é considerado um factor importante da inteligência artificial, pois um dos comportamentos essenciais para qualquer agente é a forma como ele se desloca, e para que não se pareça irreal é necessário criar um movimento adequado a fisionomia do agente.

Movimento individual

O movimento cinemático [Millington06a], usa a posição e a orientação do agente para criar movimentos simples, permitindo que a velocidade mude instantaneamente, e que o agente se desloque do ponto A para o ponto B de uma forma constante, mas no mundo real a lei de *Newton* não permite que a velocidade de um corpo mude instantaneamente, esta violação leva a que como resultado final se produzam movimentos estranhos das personagens. Já os movimentos direccionados (*steering behaviors*) [Millington06a] adicionam aceleração e rotação ao agente, que proporcionam movimentos mais realistas. A aceleração permite que a velocidade entre dois pontos vá alterando consoante a partida e a chegada.

Movimento em grupo

Além do movimento cinemático e direccionado, temos movimentos mais complexos como o *flocking* por *Craig Reynolds* [Reynolds87], cujo objectivo é simular o movimento de vários agentes ao mesmo tempo, e é baseado em três regras:

1. Separação – Faz com os agentes evitem colidir com outros agentes.
2. Alinhamento – Direcciona o agentes baseado nas direcções dos outros agentes.
3. Coesão – Faz com que o agente se mova para a posição média dos vizinhos.

O *flocking* é excelente para simular bandos de pássaros, uma manada de animais, um cardume ou mesmo, com algumas alterações, simular multidões.

Com a adição de uma quarta regra, evasão [Johnson04] , faz com que o agente evite colisão com objectos. Uma alternativa para evitar colisões é a criação de *flow fields*, cada objecto gera à sua volta um campo de fluxos para que o agente mude de direcção evitando assim a colisão com os objectos, sejam eles dinâmicos ou estáticos. [Alexander06]

O problema com o *flocking* é que é demasiado pesado computacionalmente, devido à verificação das três regras para todos os agentes, devido a isto é normalmente usado para simular dezenas de agentes. Uma alternativa ao *flocking* é o *swarming* que apenas aplica duas das três regras referenciadas no *flocking*, o alinhamento e a coesão, é orientado a simulação de centenas de agentes e origina num comportamento mais "orgânico" pode-se ser usado para simular o comportamento de baratas, ratos ou aranhas. [Scutt02]

As formações normalmente servem para que um grupo de elementos consiga obter vantagens quando ataca ou bate em retirada sobre um inimigo. Em alguns videojogos apenas são criadas para obter a imersão requerida pelo jogo ou mesmo só porque "parece bonito". [Dawson02]

As formações mais comuns são em Linha, em Coluna, Flanco direito, Flanco esquerdo, Caixa, em Cunha e em V. (Ver figura 6)

Caso o sentido de orientação do agente seja importante, por exemplo, o agente só reage quando detecta um inimigo na sua linha de visão, então existem três tipos de formações: linha *staggered*; voltados para o interior ou para o exterior. (Ver figura 6)

Para o caso de haver várias unidades de tipos diferentes, normalmente, as unidades mais fracas e de longo-alcance são posicionadas atrás das formações ou no centro, unidades mais velozes nos flancos para que tenham mais liberdade de movimento quando atacam o inimigo, ou mesmo algumas formações podem requerer unidades específicas para posições específicas.

A perda da integridade da formação durante os movimentos é a chave para o sucesso de um bom movimento incluindo as reacções para superar obstáculos, pontos de estrangulamento e combates. [Dawson02]

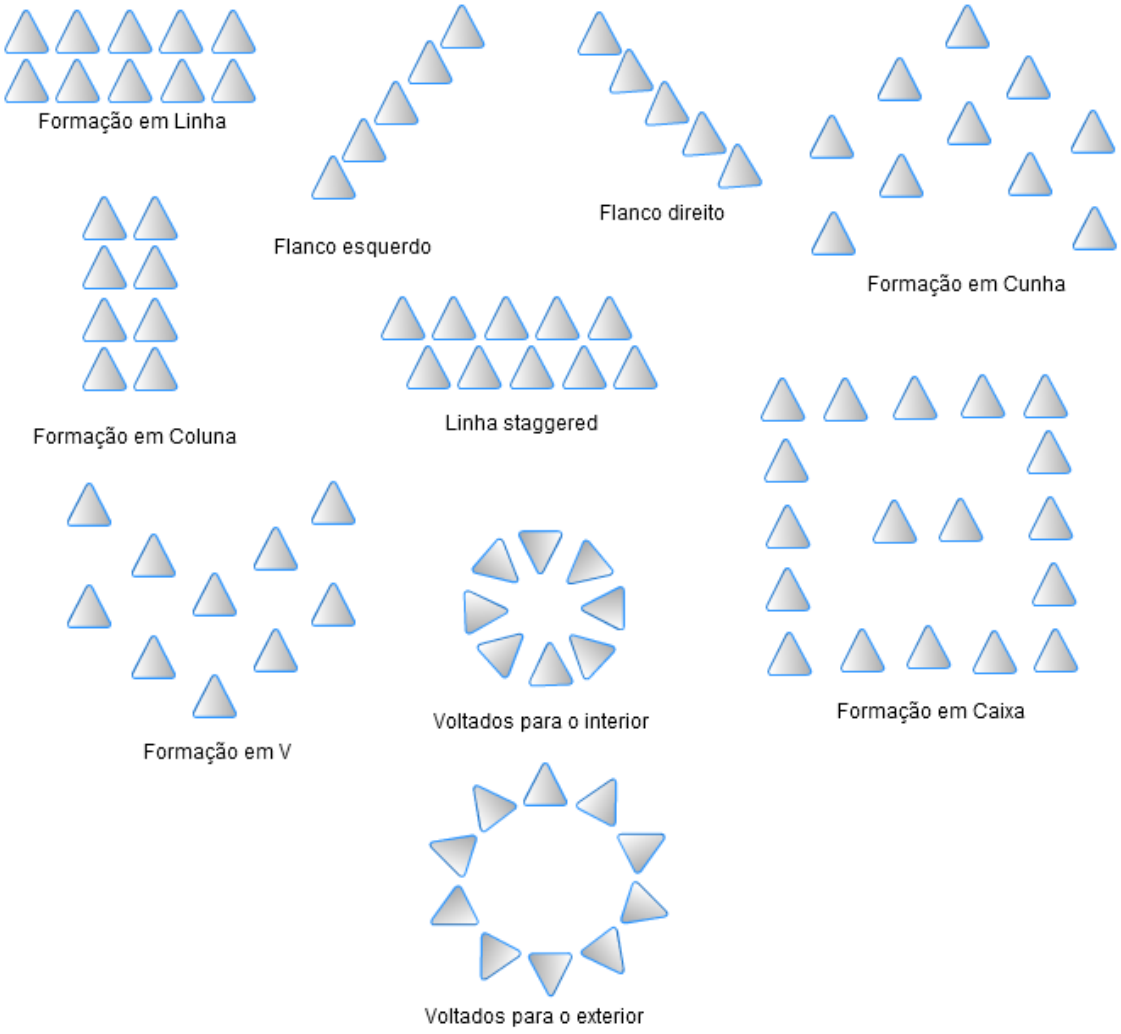

Figura 6: Formações mais comuns

Outros desafios em relação ao movimento

Nem sempre os movimentos sobre os *waypoints* se tornam os mais realistas, muitas vezes devido à sua deficiente colocação. Uma maneira de fazer com que as curvas entre os *waypoints* sejam mais suaves e credíveis, é a aplicação de um atrito na curvatura para que o agente não se vire de uma forma muito "robótica" [Pinter02].

Em muitos mundos virtuais existem obstáculos que originam problemas de navegação, como portas, elevadores, saltos, extremidades e escadas. Estes tipos de problemas têm soluções com uma leitura mais rica sobre a representação do meio envolvente [Hancock02] [Reed04].

2.1.3 Comportamento

Os comportamentos são criados para que o agente possa atingir um objectivo ou tarefa proposta. Há várias técnicas para que o agente possa seleccionar o melhor comportamento em cada situação ou estado, entre um leque de possibilidades.

Esta secção estará divida em quatro partes, que tratam de diferentes tipologias de selecção do comportamento, nomeadamente:

- **Processos de decisão** – são processos em que, como o nome indica, dado o estado de um agente geram o próximo comportamento.
- ***Pathfiding*** – Um dos comportamentos essenciais. Calcula o caminho para que o agente se possa mover da posição actual para uma determinada posição objectivo.
- **Análise táctica** – É um mecanismo que permite que o agente, a partir da informação a que tem acesso, gere uma estratégia ou um comportamento.
- **Grupos e comportamentos coordenados** – Técnicas para comportamentos coordenados.

Processos de decisão

Na maioria dos videojogos, o processo de decisão do comportamento do agente é suportado por uma máquina finita de estados (FSM) [Millington06c] [Pittman05]. Esta consiste num número finito de estados, definidos pelo programador, em que o agente só muda de estado se alguma transição for activada. Cada um destes estados corresponde normalmente a um comportamento que o agente pode executar.

As FSM são uma estrutura de decisão de fácil implementação e apresentam bons resultados. A sua principal limitação deve-se ao facto de ser muito difícil cobrir todas as situações a que o agente pode estar submetido num mundo virtual dinâmico.

Uma solução mais elegante é o *Goal-Oriented Action Planning* (GOAP) [Orkin04a] [Pittman05], que dado um objectivo, é gerado um plano de acções para que o agente

possa executar de forma a atingir esse objectivo. As acções são atómicas, representando comportamentos básicos, como por exemplo saltar. As acções têm pré-condições que precisam de ser satisfeitas e pós-condições que podem despoletar outras acções. Este tipo de técnica pode originar comportamentos diferentes para objectivos idênticos. Em mundos virtuais dinâmicos esta solução torna-se demasiada dispendiosa, pois pode haver alterações no cenário durante o planeamento da sequência de acções ou durante a sua execução, tornando o plano obsoleto, obrigando a um novo planeamento. A rede hierárquica de tarefas (HTNs) oferece uma solução para este problema, consiste em várias camadas de planeamento, caso uma acção se torne inválida, gera um novo plano nessa camada e não é necessário que todo o plano seja cancelado [Wallace04].

Pathfiding

Existe vários algoritmos de pesquisa em profundidade, em largura, *Dijkstra*, e o A* que servem para obter um caminho entre dois pontos, evitando colisões com os objectos do cenário.

O A* devido à sua função heurística consegue obter uma excelente performance para encontrar caminhos óptimos e por este mesmo motivo é o algoritmo mais utilizado na indústria de videojogos [Tozour04] [Hart72].

No entanto, como já foi dito antes, a performance da pesquisa é afectada com a escolha da estrutura de dados para a representação do mundo virtual, por isso, muitas vezes a escolha do algoritmo de pesquisa depende da estrutura de dados aplicada.

Além destes algoritmos de pesquisa em cima referenciados, existem alternativas como a criação de tabelas de pesquisa [Dickheiser04] [Sterren04], em que o acesso aos dados torna-se directo, consequentemente mais rápido. Também existem alternativas ao A*, que invés de procurar o melhor caminho em tempo de execução usam informação pré-processada para calcular o melhor [Surasmith02].

Análise táctica

Com a informação contida nos *waypoints*, (normalmente cada *waypoint* contém a informação dos *waypoints* vizinhos visíveis) e com a informação relativa à posição do inimigo (normalmente obtida apenas quando o agente o vê), é possível criar planos para atacar, flanquear ou bater em retirada [Lidén02].

Para uma avaliação da posição táctica podemos considerar quatro factores: Posições perto do agente; Posições que os inimigos vêem; Posições que fornecem cobertura (estas posições normalmente são inseridas manualmente); e em caso de múltiplos inimigos, a Posição que oferece linha-de-ataque para o inimigo que o agente prefere atacar [Straatman06].

Podemos também ter em conta alguns aspectos definidos por *Jeff Orkin* para tornar o agente mais credível, caso este simule um ser humano, e os aspectos são Propriedade,

Dependência, Relevância, Responsabilidade, Prioridade, Consciência, Estado esperado, Presença de outros [Orkin04b].

Os agentes também podem aprender com as suas acções, utilizando mapas de ocupação, que geram informação acerca do mundo baseada na percepção do agente, o que ouve e vê. Por exemplo, e se o agente deixa de ver o inimigo? Ele deve ir para a última posição onde o viu? E se ele lá não estiver? Vai à procura? O agente deverá então avaliar a situação para poder aprender, e pode avaliá-a da seguinte maneira:

- Confirma, se o comportamento era o esperado
- Refuta, se o comportamento não era o esperado
- Não faz nada, caso não consiga confirmar nem refutar o comportamento

Com esta informação, deve-se proceder à criação de uma grelha com campos de probabilidade à volta dos objectos inseridos no mundo, representando as áreas onde o objecto poderá estar com o passar do tempo [Isla06].

Grupos ou comportamentos coordenados

Os agentes podem estar inseridos numa equipa, o que pode originar a outros problemas de decisão e comportamentos que o agente deve tomar. Uma equipa pode optar por uma organização centralizada, onde existe um líder que toma as decisões de como a equipa deve proceder, ou descentralizada, onde cada elemento comunica aos outros a sua intenção de actuar perante uma dada situação e no fim todos decidem o que devem executar.

Baseado numa organização descentralizada temos o *Squad AI* [Sterren02a], uma camada que processa qual o melhor comportamento que a equipa deve tomar, com base nas intenções que os agentes comunicam. Como a camada *Squad AI* acaba por tomar as decisões para a equipa, pode ser vista também como centralizada e temos assim um misto dos dois tipos de organizações. É capaz de produzir comportamentos emergentes, comportamentos baseados na interacção com os elementos do grupo para que os elementos não executem acções independentes.

No *Company of heroes* [Jurney08], os esquadrões são compostos por três elementos: Núcleo, Flanco esquerdo e Flanco direito. Cada esquadrão tem um líder e o líder normalmente encontra-se no elemento núcleo. As ordens dadas pelo jogador (utilizador) são executadas pelo líder, ele é responsável por procurar o caminho até ao objectivo, os outros *bots* prevêem a futura localização do líder (mais ou menos com 2 segundos de avanço) e movem-se para essa posição mais um deslocamento definido. Usam a formação em cunha para áreas abertas, cunha mais apertada para áreas limitadas e coluna *staggered* para estradas.

A melhor escolha de uma acção para uma equipa executar pode ser baseada nas observações dos elementos da equipa, nos dados históricos e na experiência. É portanto necessário a criação de uma "imagem" da situação para fazer manobras planeadas.

Em alguns casos, usa-se *Fuzzy logic* para calcular o *fitness* da manobra. Na tabela 2 apresenta-se um exemplo de uma função de cálculo do fitness.

Tabela 2: Exemplo de uma função para calcular o fitness de uma manobra

Fitness (pullback) = weaker (line-of-fire ratio) ∩ weaker (position quality) ∩ mediumshorter(range)

Há exemplos de técnicas em que o resultado das manobras bem sucedidas é guardado ("aprendido") para apoio na escolha de futuras manobras [Sterren02b].

Podem também utilizar-se comandos hierárquicos para simular uma interacção mais credível e semelhante às dos humanos, os postos mais altos na hierarquia dão ordens à hierarquia abaixo, e os postos mais baixos transmitem informação à hierarquia superior [Reynolds02].

Existem técnicas simples que facilitam os comportamentos coordenados. Para o caso de querermos proporcionar comportamentos diferentes em dois ou mais agentes com o mesmo objecto, podemos reservar caminhos já calculados para que outros agentes calculem caminhos diferentes. Até mesmo para evitar que uma área se torne demasiada lotada pode-se usar uma *flag* para marcar a área e assim os próximos agentes que tentem entrar nessa área ficam na periferia [Orkin04c].

Uma noção de equipa é também importante para mostrar inteligência por parte dos elementos da equipa, pois torna-se fácil criar comportamentos como evitar a linha-de-fogo de outros agentes aliados [Reynolds04].

2.2 Ferramentas de desenvolvimento para videojogos do género FPS

Existem várias plataformas para o desenvolvimento de videojogos do género FPS, nesta secção apenas abordamos algumas baseadas nos seguintes critérios; licença gratuita, capacidade de realizar um videojogo FPS, que já tenha pelo menos um jogo comercial, que ofereça algumas técnicas de inteligência artificial e contenha mecanismo para o *debugging*. Com base nestes critérios deparamo-nos com o UDK, Unity3D e o Panda3D.

Estas plataformas oferecem um conjunto de técnicas comuns para o *debugging* como o sistema de *logs*, funções de suspender, retomar e acompanhar passo a passo o fluxo de execução do processo, e ainda a monitorização de variáveis num dado instante (ver tabela 3).

Os sistemas de *logs* são compostos por instruções colocadas no meio do *script* de forma a rastrear o processo e são úteis para indicar o fluxo de execução de um processo.

As funções como suspender, retomar e passo a passo são úteis para interromper o fluxo do processo num dado instante e assim oferecem a possibilidade de monitorizar o estado do processo como também permitem acompanhá-lo passo a passo para detectar anomalias de forma rápida.

Tabela 3: Técnicas de *debugging* mais comuns

Técnicas usadas **Plataformas de desenvolvimento**	**Sistema de *Logs*'**	**Funcionalidades como Retomar, Suspender, e passo a passo**	**Monitorização de variáveis**
UDK	Tem	De momento não. [UDKDebugging10]	Com o uso do sistema de *Logs*'
Unity 3D	Tem	Através do IDE (MonoDevelop)	Através do IDE
Panda3D	Tem	Através do IDE (Microsoft Visual C++)	Através do IDE

Além das técnicas de *debugging*, estas plataformas também oferecem algumas técnicas de inteligência artificial como comportamentos que servem como base para construção de comportamentos mais complexos (ver tabela 4).

Tabela 4: Técnicas e comportamentos de inteligência artificial

	Pathfinding	Comportamentos já implementados
UDK	**Navigation mesh e A***	**Movimento entre os waypoints.**
Unity 3D	*	*
Panda 3D (Panda AI)	**Navigation Mesh e A***	**Seek** **Flee** **Pursue** **Evade** **Wander** **Flock**

* O Unity 3D por si só não tem nenhuma biblioteca, ou alguns métodos para desenvolver comportamentos inteligentes, mas existem vários *plugins* que fornecem métodos de inteligência artificial, como o UniSteer. [UnitSteer09]

As técnicas de *debugging* mostram-se pouco eficazes quando pretendemos procurar anomalias nos comportamentos construídos. Quando se constrói comportamentos é importante que exista uma funcionalidade que facilite a criação de cenários para que se possa testar o comportamento. Tanto o Unity 3D como o UDK oferecem este tipo de funcionalidade com a manipulação dos objectos presentes nos cenários através do editor mas só o é permitido quando o jogo não está em execução, impossibilitando assim rápidas correcções nos cenários de teste. A monitorização de variáveis que o Unity 3D e o Panda3D disponibilizam é só para quando o jogo está suspenso, a monitorização destas várias em tempo real pode levar a uma detecção mais rápida de anomalias no comportamento.

Tendo em conta a dificuldade em encontrar plataformas de desenvolvimento para *scripts* baseados em regras é também natural não encontrar mecanismos de *debugging* para esse tipo de sistemas.

É necessário uma ferramenta que consiga cobrir as lacunas das plataformas apresentadas, de forma a facilitar o desenvolvimento de comportamentos para videojogos do género FPS, que possam ser utilizados com qualquer linguagem de *scripts*.

2.3 Ferramentas utilizadas

Esta secção descreverá as ferramentas que foram escolhidas e utilizadas para a concretização da DEVELOP-FPS, a ferramenta que assiste no desenvolvimento de comportamentos escritos em *Jess* para agentes inseridos no videojogo *Unreal Tournament 2004*.

2.3.1 Unreal Tournament *2004*

Unreal tournament 2004 (UT*2004*) é um jogo multijogador para computador do género FPS, *First Person Shooter*, [Wikipedia11] onde vários jogadores competem entre si em várias arenas. A figura 7 ilustra o que um jogador humano pode observar enquanto joga.

Figura 7: Perspectiva que o jogador humano tem sobre o ambiente proporcionado pelo videojogo Unreal Tournament *2004*.

Este género de videojogo tem dois tipos de jogador, jogador humano ou virtual também conhecido por *bot*. O jogador é colocado em vários cenários ou ambientes em três dimensões para explorar de forma a ir ao encontro do objectivo proposto. Em particular este videojogo proporciona vários objectivos para que o jogador possa

progredir na história, alguns destes objectivos são *Deathmatch*, *Team Deathmatch* e *Capture the Flag*.

- *Deathmatch* - O objectivo é chegar a um número determinado de inimigos* para aniquilar, ou mesmo num determinado espaço de tempo eliminar o máximo de inimigos para que o número de inimigos aniquilados seja superior aos do adversário.
- *Team Deathmatch* - O mesmo que o *deathmatch* só que os inimigos mortos por cada jogador da mesma equipa são somados e resulta no número de pontos que a equipa tem.
- *Capture the flag* - O objectivo aqui é, capturar a bandeira da base da equipa inimiga e retornar a bandeira à base da própria equipa, perfazendo um número estipulado de bandeiras capturadas, ou fazer um número máximo de bandeiras capturadas num determinado espaço de tempo para que este número seja superior ao do adversário.

* - Os inimigos podem ser humanos ou virtuais (agentes).

Para que o jogador possa ir de encontro aos objectivos propostos, cada personagem do jogo, no seu estado inicial, está munida de uma arma de longo alcance capaz de infligir dano ao inimigo, mas esta arma contém pouco poder de dano. O jogador pode explorar o cenário para adquirir mais armas com superior poder de dano de forma a ganhar vantagem sobre o adversário. Armas essas que estão distribuídas em pontos fixos pelo cenário todo. Todas as armas à excepção de uma, que inflige dano corpo-a-corpo, necessitam de munições, que também se encontram espalhadas pelo cenário. Existem munições específicas para cada arma.

Além das armas e das munições, também existem itens que adquiridos pela personagem, recuperam e/ou adicionam mais pontos de energia a quem os adquiriu, isto é, o estado inicial de energia de cada jogador é de 100 pontos e 0 pontos de escudo, o escudo serve para absorver o impacto das armas. Há itens de energia que apenas aumentam até aos 100 pontos e há outros que passam o limite dos 100 pontos até aos 199. O sistema de pontos do escudo é bastante semelhante à energia, existem vários itens que adicionam pontos ao escudo até perfazer os 99 pontos e normalmente existe apenas um item no mapa que adiciona mais 100 pontos ao escudo fazendo um máximo de 199 pontos. Resumindo, tanto o escudo como a energia têm dois limites de pontuação, e existem itens para regenerar ambos. O jogador tem então um imenso leque de estratégias que pode formar conforme o objectivo proposto.

2.3.2 Pogamut V.2

O *Pogamut* é uma API, uma interface de programação que permite a programação de agentes, na linguagem Java, para o videojogo *Unreal Tournament 2004*. Usa a interface *GameBots* para poder comunicar com o jogo *Unreal Tournament 2004*.

O *GameBots* (ver Figura 8) providencia um protocolo de rede baseado em texto que permite conectar com o UT*2004* e controlar os agentes em jogo através de mensagens [Pogamut09] [Gemrot11].

O *Pogamut*, além de facilitar a programação dos *bots* para o jogo *Unreal Tournament 2004* em JAVA, também já foi usada como ferramenta de implementação de comportamentos inteligentes [Burkert] [Arrabales09] [Brom09]. É considerada como um excelente ponto de partida para a programação de *bots* e para pesquisa de comportamentos multiagentes [Hindriks11] [Kaminka02].

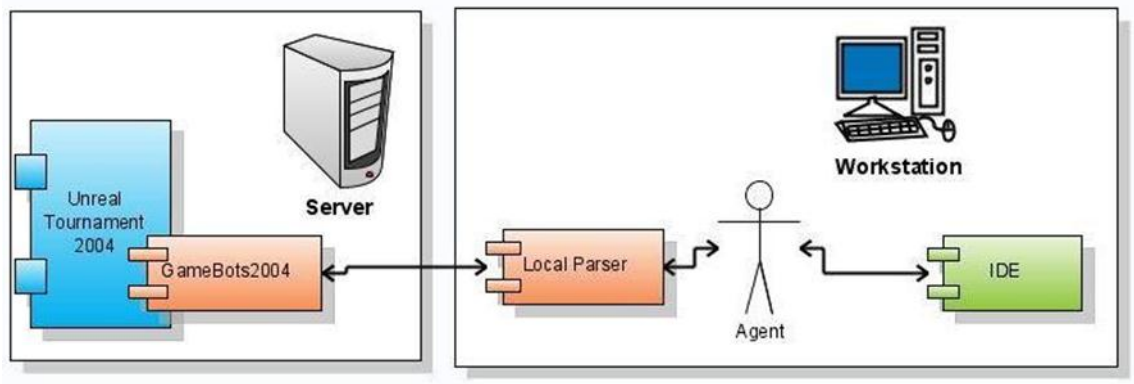

Figura 8: Arquitectura Unreal Tournament *2004* / Pogamut. O IDE (Ambiente integrado de desenvolvimento, facilita/ajuda na programação) em questão é o *NetBeans* que é o requerido pelo Pogamut v2. [Pogamut09]

2.3.3 Sistemas de regras

A maioria das linguagens de programação que usamos são imperativas ou orientada a objectos, em que "dizemos", ao computador, o que deve fazer, como deve fazer e por que ordem. Não são ideais para escrever código baseados em regras para os comportamentos dos agentes virtuais.

O código da componente de inteligência artificial num videojogo, em contraste com o outro código do videojogo, tende a ser rico em condições, precisando de muitos testes para a transição de estados so agente virtual.

A programação com base em regras, descreve o que o computador deve fazer mas omite a maioria das instruções de como se faz [Friedman03] e, ao contrário de uma linguagem de programação convencional, não é necessário declarar variáveis, criar ciclos ou sub-rotinas apenas é necessário criar cláusulas condição-acção, conhecidas por regras.

Um sistema baseado em regras normalmente é composto: por uma memória de trabalho, considerada com a base de dados dos factos; um conjunto de regras, que contem todas as regras definidas pelo utilizador; o motor ou máquina de inferência, que é composto por um identificador de padrões, responsável por identificar quais as regras que podem ser executadas no ciclo corrente, e uma agenda que contém uma lista de todas as regras possíveis de serem activadas cujo primeiro elemento da lista é a regra que é activada, resultado do tratamento de conflitos; e por fim um motor de execução que é responsável por executar o que está definido na regra [Vogt08]. (Ver figura 9)

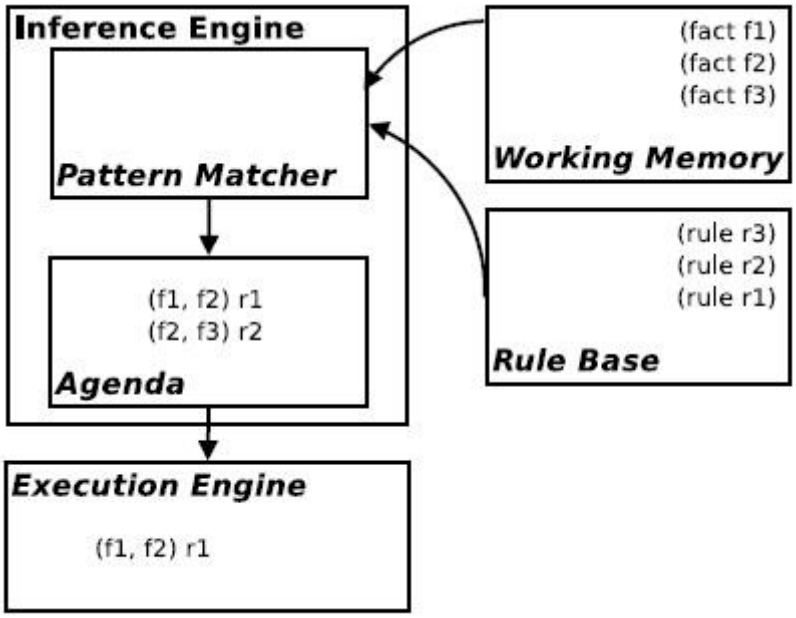

Figura 9: Arquitectura típica de um sistema baseado em regras. [Friedman03]

Uma abordagem com base em regras, comparada com a programação imperativa traz muitas vantagens:

- O sistema de produção (máquina de inferência e motor de execução) automaticamente verifica e activa as regras, por esse motivo, o programador apenas escreve as regras, mas não o código que as invoca, as verifica ou as executa.
- Não é necessário escrever ciclos no código em múltiplas instâncias.
- O código das regras é separado do restante código, facilitando a leitura, a alteração, o teste e a reutilização.
- O raciocínio para as regras é mais fácil e natural que o raciocínio para algoritmos com vários ciclos e condições.
- Regras individuais podem ser inseridas, alteradas e removidas sem que seja necessário mudar a lógica de todo o *script*.

As regras podem ser comparadas com as declarações "*if-then*" das linguagens tradicionais de programação, e são compostas por uma parte esquerda (*if*), também consireda com a parte com a condição; e uma parte direita (*then*), parte da acção.

Os sistemas à base de regras já foram utilizados para implementar comportamentos em videojogos como: o RC++ [Wright00a] [Wright00b], que facilita o desenho de comportamentos para a consola de videojogos *PlayStation 2*; e o SOAR [Lent99], que serve como motor de inferência para agentes inteligentes para os videojogos *Descent 3* e *Quake II*.

Contudo, geralmente um programa à base de regras consome mais recursos, tanto em tempo de processamento como em espaço, comparado com as linguagens imperativas convencionais.

Jess

O *Jess*, linguagem de programação baseada em regras, totalmente escrito em *JAVA*, apresenta uma arquitectura idêntica à da figura 9 é composto essencialmente por: factos, considerado o conhecimento que damos como adquirido; regras, cláusula composta por uma condição seguida de uma acção; funções, compostas por um conjunto de instruções; e módulos, que agrupam um conjunto de regras, permitindo a modularidade. Em termos de eficiência, o *Jess* usa o algoritmo *rete* para indetificar os padrões, e foi provado efeciente na selecção das regras [Follek03].

2.3.4 AGLIPS

A AGLIPS [Moniz03] é uma ferramenta para o desenvolvimento de grupos de robôs baseados em comportamentos e que foi uma das inspirações para a DEVELOP-FPS. A AGLIPS baseia-se em dois componentes principais: a plataforma multiagente AGLETS [Lange98] e o ambiente de simulações robóticas *Player/Stage* [Vaugham01]. A AGLIPS liga estes dois ambientes heterogéneos numa única plataforma, fornecendo uma ferramenta para a construção de agentes e um ambiente para os experimentar e avaliar. Uma das funcionalidades da AGLIPS é fornecer ao utilizador monitorização e controlo, em tempo real, sobre cada robô individual, podendo também fazêlo em termos de grupo, através de um sistema de envio de mensagens (*broadcasting*). Esse acesso permanente aos robôs permite parar e resumir a execução dos robôs, tanto individualmente como em bloco, acedendo aos estados dos diversos robôs e permitindo alterar os seus comportamentos em tempo real, tanto em bloco como individualmente. Por outro lado, permite repetir um conjunto de experiências, partindo de situações iniciais configuráveis e heterogéneas e recolher dados importantes para avaliar e comparar posteriormente o desempenho de diversos tipos de comportamentos.

Capítulo 3

Este capítulo agrega os tópicos referentes ao trabalho realizado na investigação das vantagens de uma ferramenta de desenvolvimento de comportamentos para personagens virtuais, especificamente movimentação em grupo e comportamentos de ataque no género FPS.

Começamos por descrever a ferramenta DEVELOP-FPS, referindo a sua arquitectura que é composta por quatro componentes: a consola global; a consola individual; a componente *Jess*; e o servidor do jogo *Unreal Tournament 2004*. As funcionalidades da consola global, controlo e monitorização sobre todos os intervenientes na experiência, são descritas na secção seguinte. Na mesma secção, é descrita a consola individual e a sua importância para manipular individualmente cada agente que intervém na experiência. No que diz respeito à utilização da linguagem *Jess* para construção de comportamentos, são referidas as características da arquitectura adoptada.

São apresentados também dois cenários criados para o videojogo *Unreal Tournament 2004*, bem como o intuito das experiências a realizar nestes mesmos cenários,por exemplo, pontos de estrangulamento submetidos ao movimento das equipas.

Por fim será apresentada a construção de um comportamento, mais concretamente a formação em diamante com quatro elementos, como um exemplo da aplicação da ferramenta para o desenvolvimento de comportamentos colectivos.

Dado que o videojogo é do género FPS, os NPCs são considerados *bots*, doravante qualquer referência a *bot* pode ser considerada como NPC ou agente virtual.

3.1 DEVELOP-FPS: Ferramenta de apoio à construção de comportamentos inteligentes

A ferramenta DEVELOP-FPS é uma evolução da arquitectura AGLIPS, totalmente escrita em *JAVA.* Esta ferramenta é desenhada especificamente para executar *scripts* escritos em *Jess* no videojogo *Unreal Tournament 2004*, oferecendo funcionalidades que permitem ao utilizador fazer *debug* e testar os seus comportamentos. É composta por duas componentes principais, a consola individual e a consola global, que possuem algumas funcionalidades inspiradas em programas como o *NetLogo* [NetLogo11] [TurtleKit09] e em plataformas de desenvolvimento de videojogos,que foram descritas anteriormente na secção 2.2 (ver tabela 5).

Tabela 5: Funcionalidades DEVELOP-FPS.

Técnicas usadas
Sistema de *logs*
Retomar, Suspender e passo-a-passo
Criação de cenários de teste
Execução de funções definidas pelo utilizador, em tempo real
Monitorização de variáveis, em tempo real
Realização de experiências
Sistema de avaliação de comportamentos
Capacidade para gerir um grupo de agentes

As três últimas técnicas da tabela 5, não se encontram nas mais recentes plataformas de desenvolvimento de videojogos.

Num videojogo existem vários níveis de dificuldade, com sistemas que permitem realizar experiências e avaliar comportamentos. Podemos testar e comparar qual o melhor comportamento e a que nível de dificuldade se adequaria, o que é impossível com as plataformas actuais. A capacidade de gerir grupos de agentes é útil para promover comportamentos colectivos entre agentes e jogadores.

3.1.1 Arquitectura e respectivas funcionalidades

A arquitectura da DEVELOP-FPS baseia-se em quatro componentes: o *script* (em *Jess*) que define o comportamento do *bot*; a consola individual responsável pela monitorização e controlo individual de cada *bot*; a consola global responsável pela execução, controlo e monitorização de todos os *bots* intervenientes na experiência; e o servidor do videojogo *Unreal Tournament 2004* que permite aos agentes exercer os comportamentos (ver figura 10).

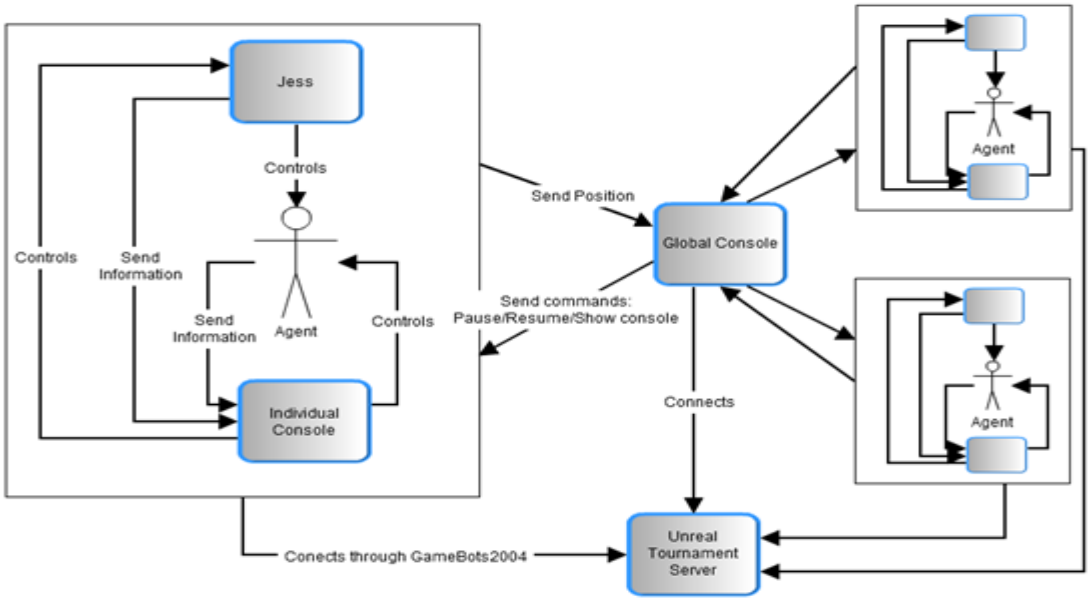

Figura 10: Arquitectura genérica da DEVELOP-FPS.

Consola Global

A consola global (ver figura 11), que pode ser considerada como o centro das operações, é responsável por recolher os parâmetros e iniciar as experiências, monitorizar e oferecer controlo sobre todos os intervenientes na experiência, e construir e apresentar toda a informação proveniente da experiência.

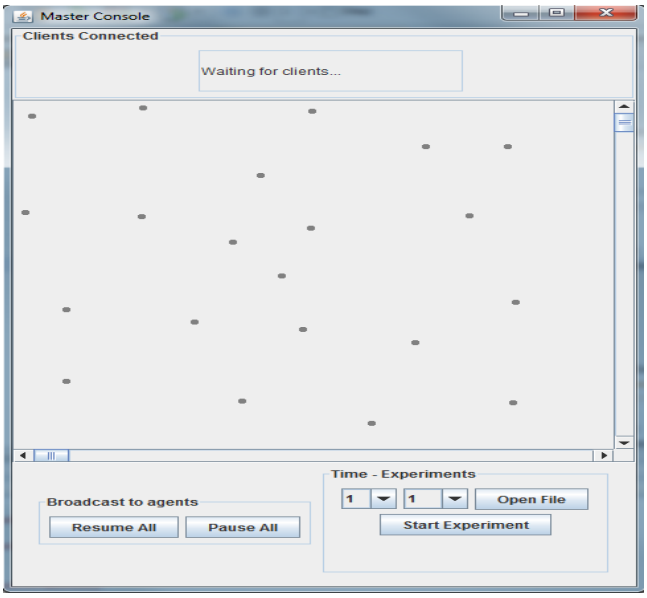

Figura 11: Consola Global. No topo fica situada a lista de *bots* conectados, no centro podemos observar o mapa 2D com os *waypoints* fornecendo uma vista de topo do cenário, no canto inferior esquerdo temos os botões para interromper e retomar a lógica dos *bots*, no lado direito os argumentos necessários para começar uma experiência.

De uma forma mais detalhada, as funcionalidades oferecidas por esta consola são as seguintes:

1. Executar experiências e disponibilizar os resultados, com o objectivo de avaliar os comportamentos testados;
2. Fornecer uma vista de topo sobre o cenário, mostrando os *waypoints* e a posição dos *bots* para uma melhor percepção onde estão situados;
3. Permitir a interrupção e a retoma da execução do jogo para que se possa ou adquirir informação, ou controlar um *bot* num dado instante;
4. Disponibilizar o acesso à consola individual de cada agente interveniente na experiência, permitindo o controlo e monitorização individual do *bot* seleccionado a qualquer altura.

Estas funcionalidades permitem testar comportamentos colectivos, porque permitem acompanhar e controlar todos os intervenientes numa experiência. Para mais detalhes ver a secção 3.3.4 - Aplicação da DEVELOP-FPS para comportamentos colectivos.

De forma a satisfazer a primeira funcionalidade, começamos por definir seis parâmetros, nomeadamente: o comportamento que o *bot* deve executar; o nome a que o comportamento está associado; o número de *bots* intervenientes na experiência; o estado inicial do *bot* (estes quatro parâmetros são recolhidos através de um ficheiro XML (ver tabela 6)); o número de iterações; e a duração de cada iteração. (Estes dois últimos parâmetros são adquiridos através da interface da consola global)

Estes parâmetros permitem realizar várias iterações sobre uma experiência com múltiplos agentes, permitindo uma avaliação probabilística de vários comportamentos, para uma análise mais detalhada.

O parâmetro "estado inicial do *bot*" tem como objectivo definir se o *bot* deve ou não começar parado, isto permite que o utilizador possa preparar um cenário de teste sem que o *bot* execute os seus comportamentos.

Tabela 6: Ficheiro XML, com os parâmetros necessários para a experiência. O ficheiro contém as seguintes TAGs: <bot> representa um bot que vai ser inserido na experiência e contém as sub TAGs <name> nome do bot, <fileName> directoria onde está o ficheiro que contém o comportamento desejado e <pause> se deve ou não começar parado.

```
<?xml version="1.0"?>
<!-- There is only two team, team 0 and team 1 -->
<experiment>
   <bot>
         <name>Moniz</name>
         <team>0</team>
         <fileName> ~behaviorDirectory\\hunter.clp</fileName>
         <pause>true</pause>
   </bot>
   <bot>
         <name>Urbano</name>
         <team>0</team>
         <fileName> ~behaviorDirectory\\dodgeBehavior.clp</fileName>
         <pause>true</pause>
   </bot>
   <bot>
         <name>CTT</name>
         <team>1</team>
         <fileName> ~behaviorDirectory\\dummy.clp</fileName>
         <pause>false</pause>
   </bot>
</experiment>
```

As condições que definem a conclusão de cada iteração da experiência são o tempo estipulado pelo utilizador e a condição de vitória. Se for um cenário de todos contra todos (*DeathMatch*), a iteração termina quando existir um *bot "*vivo" (cada *bot* tem apenas uma vida). Quando temos um cenário de jogo de equipa (*TeamMatch*), a condição de paragem é quando todos os elementos de uma equipa são eliminados. Porém, existem dois casos particulares que ocorrem quando o utilizador apenas coloca um elemento ou só elementos na mesma equipa, nesta situação a iteração só termina com o tempo estipulado pelo utilizador. Estas situações podem ocorrer caso o utilizador pretenda testar o *bot* ou *bots* sem intervenção de inimigos, por exemplo, se quiser testar a movimentação dos *bots.*

Optamos por recolher os seguintes dados provenientes de cada iteração da experiência: o tipo de jogo (*All vs All* ou *TeamMatch*) para destacar os comportamentos individuais e cooperativos; o nome e o comportamento de cada *bot* interveniente na experiência,que servem como identificação; o estado final (vivo ou morto) e o nível de

energia, de forma a obter o desempenho do *bot*; e qual o tempo que durou a iteração, útil para classificar a eficácia dos comportamentos (ver tabela 7).

Tabela 7: Informação apresentada sobre cada iteração da experiência

Type of game	All vs All!
AgentName;File(Behavior);	Urbano_4;\dodgeBehavior.clp;
Status;EnergyLeft	Still Alive;72
Time elapsed	Time elapsed: 55 s

Com o intuito de proporcionar a vista de topo (segunda funcionalidade) sobre o cenários e os *bots* intervenientes, a consola global, quando inicializada, cria uma ligação com o servidor do jogo *Unreal Tournament 2004* para obter as coordenadas dos *waypoints* presentes no cenário. Quando a localização dos *waypoints* é obtida, e devido à diferença entre os sistemas de eixos do jogo e da consola global, é realizada uma translação e uma mudança de escala das coordenadas dos *waypoints* para as coordenadas do mapa da consola global. As posições dos *bots* são obtidas quando estes criam uma ligação com a consola global, permitindo assim, o envio da sua posição.

Os botões "*Pause All"* e *"Resume All",* respectivos à terceira funcionalidade, sempre que pressionados enviam uma mensagem de interrupção ou de retoma, a todos os *bots* conectados, interrompendo ou retomando toda a lógica comportamental.

O acesso à consola individual de cada *bot* é obtido através da selecção do *id* respectivo da lista de *bots* conectados. Ao seleccionar o *id* é enviada uma mensagem para o respectivo *bot* de modo a tornar a sua consola individual visível ao utilizador. Este tipo de selecção faz com que apenas possa haver uma consola individual visível, o que implica que o utilizador para observar várias consolas, terá de seleccioná-las de forma sequencial, satisfazendo assim a quarta e última funcionalidade.

Consola individual

A consola individual (ver figura 12) oferece funcionalidades para a visualização do estado e controlo do *bot*, de modo a proporcionar mecanismos que facilitem o *debugging* dos comportamentos. Estas funcionalidades passam por: adquirir e mostrar o estado do *bot*, dados intrínsecos (por exemplo, nível de energia), dados sensoriais (se vê algum inimigo), e dados sobre os comportamentos realizados (que comportamento foi escolhido dado o estado do *bot*)*;* e permitir o controlo do *bot*, através de funções que realizam deslocamento do *bot* para qualquer parte do cenário e de mecanismos que permitem o utilizador executar as suas funções para um tipo de controlo mais específico, por exemplo, uma função que faça com que o *bot* salte.

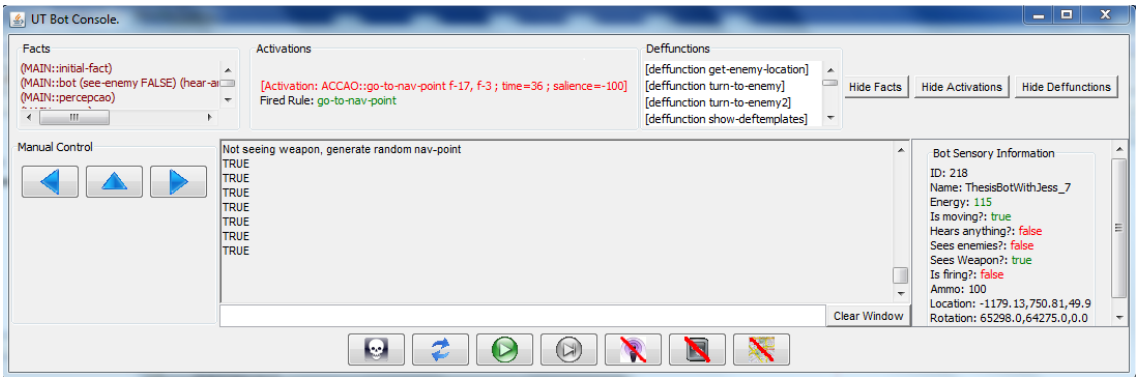

Figura 12: Exemplo da consola individual.

A consola individual é composta pelos seguintes componentes: os botões de controlo, observáveis na zona inferior da consola; a mini-estação de comando ao centro, representada por uma linha de comandos; uma janela, localizada acima da linha de comandos, que é responsável pelo *output*; a informação sensorial sobre o *bot*, localizada na parte direita; os botões responsáveis pelo controlo manual do *bot*, na parte esquerda; e a representação do estado da máquina de inferência do *Jess* na parte superior.

Botões de controlo

Os botões de controlo permitem que o utilizador controle o *bot* de uma forma simples e rápida. Estes apresentam um conjunto de funcionalidades, que iremos detalhar na ordem em que os botões são apresentados, da esquerda para a direita, na figura 13:

Figura 13: Botões de controlo.

***Kill agent*:** O *bot* é eliminado e desaparece do jogo. Esta acção é útilse quisermos, por exemplo, testar como uma equipa reage se o líder for eliminado.

***Reload Logic*:** Este botão tem como finalidade alterar o estado do *Jess* com o *script* previamente modificado pelo utilizador.

***Play/Pause*:** A execução do *bot* pode ser interrompida e retomada. Desta maneira pode-se interromper a lógica do *bot* para poder monitorizá-lo com mais detalhe ou mesmo controlá-lo num determinado instante.

***Step*:** Executa uma unidade de comportamento. Promove o acompanhamento da sequência de comportamentos que o *bot* vai executando ao longo do videojogo (Sobre uma unidade de comportamento ver secção 3.1.2).

***Show/Hide Agent State*:** A informação sobre o estado do *bot* pode ser omitida ou revelada.

***Show/Hide Jess State*:** A informação sobre o estado da máquina de inferência do *Jess* pode ser omitida ou revelada.

Show/Hide Map: Mostra ou omite o mapa.

Informação intrínseca e sensorial do *bot*

Para uma boa monitorização sobre a execução de um comportamento é fundamental ter acesso à informação interna, tais como o nível de energia do *bot*, a sua posição e rotação, se está parado, etc.

A informação interna é responsável pela transição dos comportamentos, por exemplo, se a energia do *bot* estiver abaixo de 20 unidades, o *bot* começa a procurar itens que possam recuperar a energia. Do mesmo modo é igualmente importante aceder à informação sensorial, como por exemplo, se vê um inimigo ou uma arma, se ouve alguma coisa, etc.

A janela do estado do *bot*, apresenta o ID e nome do agente seguido da informação anteriormente descrita. (Ver figura 14)

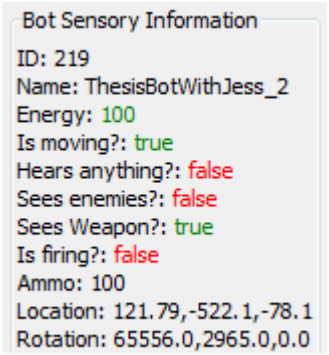

Figura 14: Informação sensorial sobre o bot.

Esta foi a informação que considerámos mais importante para ser exibida, mas como iremos explicar, existe um método para monitorizar outro tipo de informação usando a mini-estação de comando.

Monitorização do *Jess*

Para que se possam suportar *scripts* escritos em *Jess*, é bastante útil poder monitorizar: os factos contidos na memória de trabalho, para ser possível consultar o estado da máquina de inferência do *Jess*; a agenda, sendo possível acompanhar as regras activadas e a que é activada a cada iteração da lógica do *bot*; e as funções definidas pelo utilizador, pois isto ajuda-o a procurar o nome correcto da função que deseja executar.

A informação acima descrita está disponível na zona respectiva ao estado do *Jess*, como se pode observar na figura 15.

Figura 15: Informação acerca do estado do Jess. Os factos contidos na memória de trabalho à esquerda, as regras activadas e a disparada no centro e a lista de funções definidas pelo utilizador à direita.

Controlo manual

Sempre que o *bot* tem a sua lógica interrompida, o utilizador pode assumir o controlo manual para o levar a locais mais específicos. Locais a que não tenha acesso através dos *waypoints* e que sejam necessários para testar o comportamento.

O controlo manual encontra-se na parte esquerda da consola individual. Ao clicar nos botões das setas esquerda e direita, o *bot* fará uma rotação de 45 graus no sentido horário e anti-horário, respectivamente. O botão da seta direccionada para cima fará com que o *bot* se mova ligeiramente para onde está voltado.

Mapa 2D individual

O mapa, além de mostrar a posição do *bot*, os *waypoints*, o caminho que o *bot* tomou e a posição das armas disponíveis no cenário representados por círculos de diferentes cores (como se pode observar na figura 16), é também responsável por disponibilizar ao utilizador uma outra função de controlo, isto é, ao clicar em qualquer *waypoint* visível no mapa, o *bot* irá calcular o caminho da posição onde se encontra ao *waypoint* escolhido e se houver um caminho possível, o *bot* irá deslocar-se até ao mesmo.

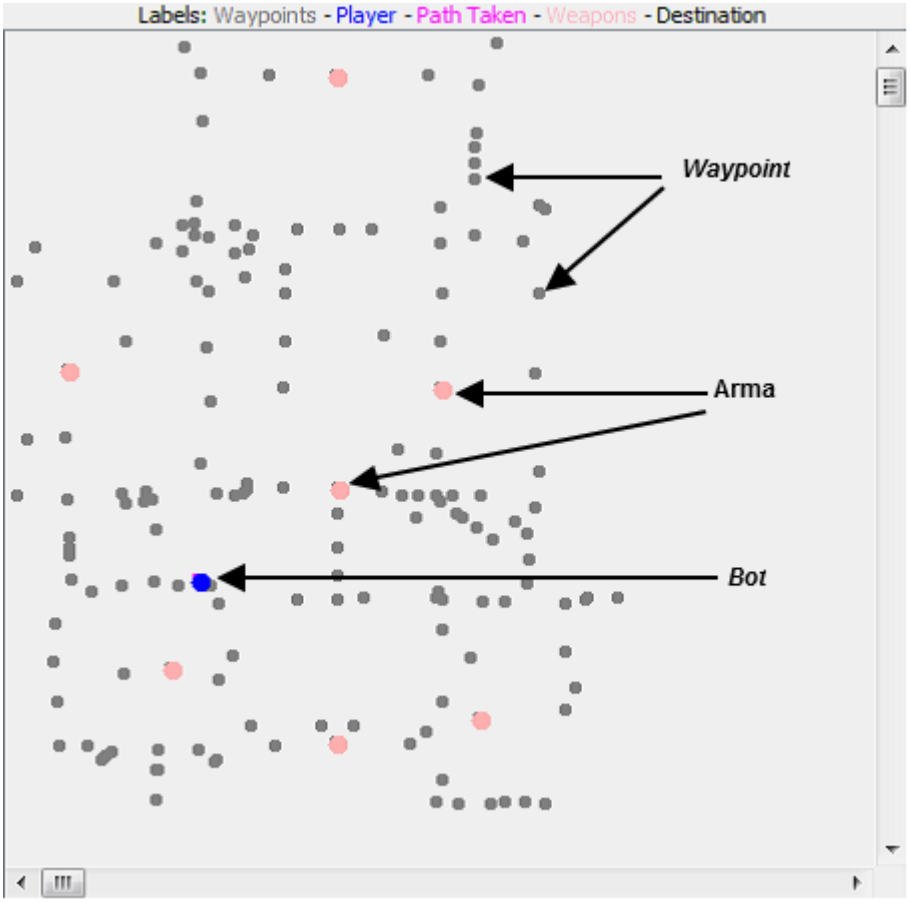

Figura 16: Mapa 2D com vista aérea para os *waypoints*, armas e posição do *bot* no cenário. No topo temos a legenda sobre as cores contidas no mapa.

Com apenas as funcionalidades descritas sobre monitorização (Informação intrínseca, sensorial e do *Jess*) e controlo (Botões de controlo, Controlo manual e Mapa 2D), podemos então facilitar a criação de cenários de teste, de modo a promover o *debugging* de comportamentos.

Ao invés do utilizador esperar para que um comportamento seja despoletado para verificar se este contém erros, poderá acelerar o processo com a criação de cenários de teste. Por exemplo, se quisermos testar um cenário em que sempre que um *bot* vê um inimigo começa a disparar, ao interrompermos a lógica do *bot* (Botões de controlo) podemos conduzi-lo para perto de um inimigo (Controlo manual), e quando o seu estado sensorial detectar um inimigo (Monitorização sensorial), poderemos observar se o comportamento escolhido (Monitorização do *Jess*), retomando a lógica, fará com que dispare contra o inimigo.

Assim, com as funcionalidades presentes na consola individual, podemos identificar, de uma forma mais rápida, se o comportamento construído tem erros.

Mini-estação de controlo

A mini-estação proporciona o controlo e monitorização do *bot* através de funções definidas pelo utilizador e está disponível através de uma linha de comandos e uma janela para o *output* presentes na consola individual.

O utilizador tem a possibilidade de executar qualquer comando/função em *Jess*. Estes comandos ou funções podem realizar um comportamento, uma mudança de estado do *bot*, ou até mesmo a monitorização de variáveis. Esta componente é importante para explorar e fazer *debugging* ao *script* criado, sendo também bastante útil para preparar situações de teste. O utilizador pode ainda criar funções em tempo-real e executá-las e como o *Jess* é concretizado em *JAVA*, o utilizador tem acesso completo à API do *JAVA*. Como exemplo, imaginemos que queremos testar uma regra criada que é disparada sempre que um *bot* encontra um inimigo e como resultado o *bot* começa a disparar contra ele. Corremos o jogo, pausamos os *bots* e colocamos um perto do outro. Para direccionar um *bot* para o inimigo, escolhemos a função definida em *Jess* (*turn-to-enemy*) que faz com que o *bot* se direccione para o inimigo, executamo-la na linha de comandos e de seguida podemos observar a lista de regras activadas na janela acima da linha de comandos (janela do *output* do *Jess*) para verificar se a regra disparada é a esperada. Poderá simplesmente retirar o *bot* do estado de pausa e verificar se a regra que disparou foi a esperada, sendo também possível esperar que o *bot* alcançasse o estado pretendido para observar se o comportamento era disparado.

3.1.2 *Jess* para comportamentos inteligentes

De forma a explicar melhor a ligação do JAVA com o *Jess*, começamos por explicar a componente *agent* presente na arquitectura *Pogamut* (ver figura 8, secção 2.3.2).

A componente *agent* é uma classe da biblioteca *Pogamut* que é responsável pela execução de um *bot* no videojogo *Unreal Tournament 2004*. Esta classe disponibiliza métodos para manipular e adquirir informação intrínseca e sensorial de um *bot.*

Com o intuito de passar toda responsabilidade da componente *agent* para o *Jess*, para que este fique inteiramente responsável pelos comportamentos do *bot*, é necessário passar a sua referência para o *Jess.* Assim, este pode invocar todos os métodos do *bot*, tornando-o responsável pela escolha e execução dos comportamentos.

A lógica optada para criar os comportamentos inteligentes em *Jess*, utilizando os mecanismos disponíveis na linguagem, é a seguinte:

- Nos factos – a informação sensorial e intrínseca do agente, por exemplo, o que vê, se ouve alguma coisa, quantas munições tem, o nível de energia, etc.
- Nas regras – a reacção, composta por factos e funções, por exemplo, se o agente vê um inimigo então dispara contra o inimigo.

- Nas funções – as acções que o agente pode executar, normalmente invocadas quando alguma regra é escolhida.
- Nos módulos – Neste projecto foram usados três módulos, *Init*, *Perception* e *Action*. Os módulos correspondem a um grupo de regras, sempre que um módulo obtém o foco apenas essas regras podem ser disparadas.
    - *Init* – Este módulo é responsável por executar todas a funções necessárias para a inicialização do *bot*.
    - *Perception* – Este módulo é responsável pela criação dos factos com a informação sensorial e intrínseca do *bot*, ou por informação que o utilizador ache relevante para a activação das acções.
    - *Action* – Este módulo é responsável por definir as acções que o *bot* pode tomar dependendo dos factos criados pelo módulo percepção.

Estas escolhas foram tomadas com base no esquema básico do funcionamento perceptivo-motor [Barreiros05].

Devido às restrições impostas pela arquitectura *Pogamut* e com os módulos definidos (*Init*, *Perception* e *Action*), a classe *agent* fica obrigatoriamente responsável por executar a máquina do *Jess* (ver tabela 8).

A execução dos dois módulos, Perception e Action, é considerada uma unidade de comportamento.

Com a unidade de comportamento estipulada, a funcionalidade passo-a-passo (*Step*) da consola individual é responsável pela execução dos dois módulos.

Tabela 8: Método responsável pela execução do Jess. Corresponde também a uma unidade de comportamantto.

```
void doLogic() {
    jessEngine.eval("focus Action")
    jessEngine.eval("focus Perception")
    if(onlyDoesOneTime)
            jessEngine.eval("focus Init")
    jessEngine.run()
}
```

Sempre que o utilizador optar por criar um *bot* para ser testado na DEVELOP-FPS, terá que ter pelo menos um módulo *Init*, um *Perception* e um módulo *Action* no seu *script* escrito em *Jess*. A tabela 9 mostra um simples comportamento com a arquitectura requerida.

Tabela 9: Dummy.clp, um simples comportamento que faz com que o bot esteja sempre a saltar.

```
(defmodule INIT)

(defrule init
        =>
        (assert (perception))
        (assert (action))
        (store RegraDisparada init)
        (return)
)

(defmodule PERCEPTION)

(defrule perception
        ?f <- (perception)
        =>

        (retract ?f)
        (assert (perception))
        (store RegraDisparada perception)
        (return)
)

(defmodule ACTION)

(defrule jump
        ?a <- (action)
        =>
        (retract ?a)
        (assert (action))

        (jump)
        (store "RegraDisparada" jump)
        (return)
)
```

O comando (*return*) assegura que mais nenhuma regra é accionada no módulo respectivo: depois de um (*return*) numa regra do módulo PERCEPTION, o controlo é transferido para o módulo ACTION, e depois de um (*return*) numa regra do módulo ACTION, o controlo é novamente retornado ao JAVA, criando o ciclo da lógica do *bot*.

3.1.3 Aplicação da DEVELOP-FPS para comportamentos individuais

Além do comportamento apresentado na tabela 9, foram criados outros dois mais complexos: o *hunter*, que anda aleatoriamente pelo cenário e sempre que vê um inimigo persegue-o e ataca-o; e o *dodgeBehavior*, que é uma evolução do comportamento *hunter* e sempre que encontra um inimigo, além de o atacar, também se tenta desviar do ataque proveniente do inimigo.

Após a escrita dos *scripts* em *Jess*, realizada num editor de texto, que definem os três comportamentos, estes necessitam de ser testados para verificar se correspondem ao que foi pensado e posteriormente programado.

O procedimento normal para testar os comportamentos, dados os mecanismos comuns de *debugging*, pode passar pela criação de *logs* por todo o script de forma a obter a sequência de execução de código. Este tipo de procedimento pode levar a uma quantidade elevada de informação, tornando difícil a sua interpretação.

Um outro procedimento pode passar pelo uso de *breakpoints* e funções de *step* que, ao interromperem o jogo, permitem a observação do estado do comportamento nesse dado instante.

O problema das abordagens anteriores é que pode levar muito tempo para que o *bot* atinja o estado desejado para se proceder à verificação de erros. Para superar esta barreira muitas das vezes os programadores recorrem a programação extra para colocar o *bot* no estado pretendido.

A realização de testes de comportamento na DEVELOP-FPS é mais simples e eficaz, visto que a ferramenta oferece métodos para criar cenários de teste, monitorizar e controlar o estado do *bot* mais simples que os métodos comuns. Como exemplo, iremos testar se o comportamento do *hunter* funciona correctamente, isto é, quando vê um inimigo (*dummy*) começa a disparar. O primeiro passo é lançar os dois comportamentos, configurando os parâmetros da experiência, através da DEVELOP-FPS. Após serem lançados e com o objectivo de criar um cenário de teste, interrompemos a sua lógica, através dos botões de controlo da consola global, e observamos no mapa global as suas posições para identificar qual o *waypoint* mais perto do *dummy*. Através do mapa da consola individual do *hunter* colocamo-lo perto do *dummy* para que possamos testar se, ao ver o inimigo, começa a disparar. Com a chegada do *hunter* ao destino, podemos observar se vê o seu inimigo, através da informação sensorial contida na sua consola. Caso não o veja, devido a estar de costas para o alvo, podemos aplicar pequenas rotações, com o controlo manual, de modo a obter o estado de visualização do inimigo. Quando o *hunter* tem no seu estado "vê o inimigo", o próximo passo será retomar a lógica do *bot* e verificar se a regra responsável por disparar contra o inimigo foi activada, bastando para isso retomar a lógica do *bot* e observar, monitorizando o estado do *Jess*, se a regra foi activada.

Estes testes servem para identificar alguns problemas comuns como: o comportamento esperado não é o escolhido, ou o comportamento escolhido não executa devidamente o que lhe foi instruído.

Continuando com o exemplo, se após o resumo da lógica do *bot* observarmos que o comportamento disparado não era o previsto, podemos interpretar este resultado de duas maneiras: ou deriva de um problema de sintaxe no *script*, que pode ser facilmente detectado executando na linha de comandos, da consola individual, o comando "*(rules)"* (que mostra todas as regras contidas no sistema), e, se verificarmos que a regra não está presente, poderá ser devido à falta de um parêntese algures na regra (*bug*

sintáctico); ou deriva de um problema na definição das condições da regra. Este problema é também facilmente identificável, já que, ao interrompermos novamente a lógica do *hunter*, podemos verificar se na lista de factos presente na sua consola individual, estão os factos que satisfazem as condições para que a regra dispare. Se não estiverem é porque foram mal inseridos pela lógica do *hunter*.

Quando observamos que a regra pretendida foi disparada e esta não executa correctamente o comportamento, por exemplo o *hunter* não disparava, o problema deve-se à programação das funções contidas na mesma. Com o problema restringido à acção dessa regra, basta então executar as suas funções, através da linha de comandos, para averiguar qual delas é que está a provocar o mau funcionamento do comportamento. Utilizando o mesmo exemplo atrás descrito, a regra do *hunter* teria uma função do género *(fire-to-enemy (enemy-position))* na parte direita (acção) da regra. Ao executar esta função na linha de comandos, poderíamos observar se o *bot* disparava ou não contra o inimigo, detectando assim se o problema é da função em questão, para proceder então à reparação da mesma.

Para que a DEVELOP-FPS seja mais eficaz a identificar problemas, o desenho dos comportamentos deve ser modular, isto é, é aconselhável existir uma regra para cada acção (uma regra para mover, para disparar, para comunicar, etc.). Deste modo é mais simples restringir o problema para posteriormente proceder à sua reparação.

A DEVELOP-FPS, além de permitir o teste dos comportamentos, também os avalia. Os três comportamentos construídos foram submetidos a uma experiência com três repetições de forma a obter os seus desempenhos, que podemos consultar na tabela 10.

Tabela 10: *Output* de uma experiência decorrida com três iterações.

```
All vs All!
Moniz_3;\hunter.clp;Died;0
Urbano_4;\dodgeBehavior.clp;Still Alive;72
CTT_5;\dummy.clp;Died;0
Time ellapsed: 26 s
Moniz_6;\hunter.clp;Died;0
Urbano_7;\dodgeBehavior.clp;Still Alive;79
CTT_8;\dummy.clp;Died;0
Time ellapsed: 42 s
Moniz_9;\hunter.clp;Died;0
Urbano_10;\dodgeBehavior.clp;Still Alive;58
CTT_11;\dummy.clp;Died;0
Time ellapsed: 55 s
```

Podemos concluir, como se pode observar em cada iteração da experiência, que o comportamento *dodgeBehavior.clp* (Urbano_?) manteve-se vivo onde os outros dois foram vencidos, sendo este o melhor comportamento entre os três.

3.2 Arenas/Cenários para experiências

Para que certos comportamentos possam ser devidamente testados, é necessário que os cenários onde os *bots* estão inseridos ofereçam condições para a activação destes mesmos comportamentos.

Com um intuito de testar um caso específico na realização de comportamentos cooperativos, procedemos à criação de um cenário (ver figura 17), que consiste num corredor onde apenas há espaço para um *bot* passar. Este cenário é útil para testar comportamentos em que equipas de *bots*, que se movem em formação, deparam-se com um ponto de estrangulamento. Uma abordagem para este problema é explicada na secção 3.3 – Tratamento e detecção de colisões. O cenário descrito foi criado pois não era possível replicar o caso específico com os cenários providenciados pelo videojogo *Unreal Tournament 2004*.

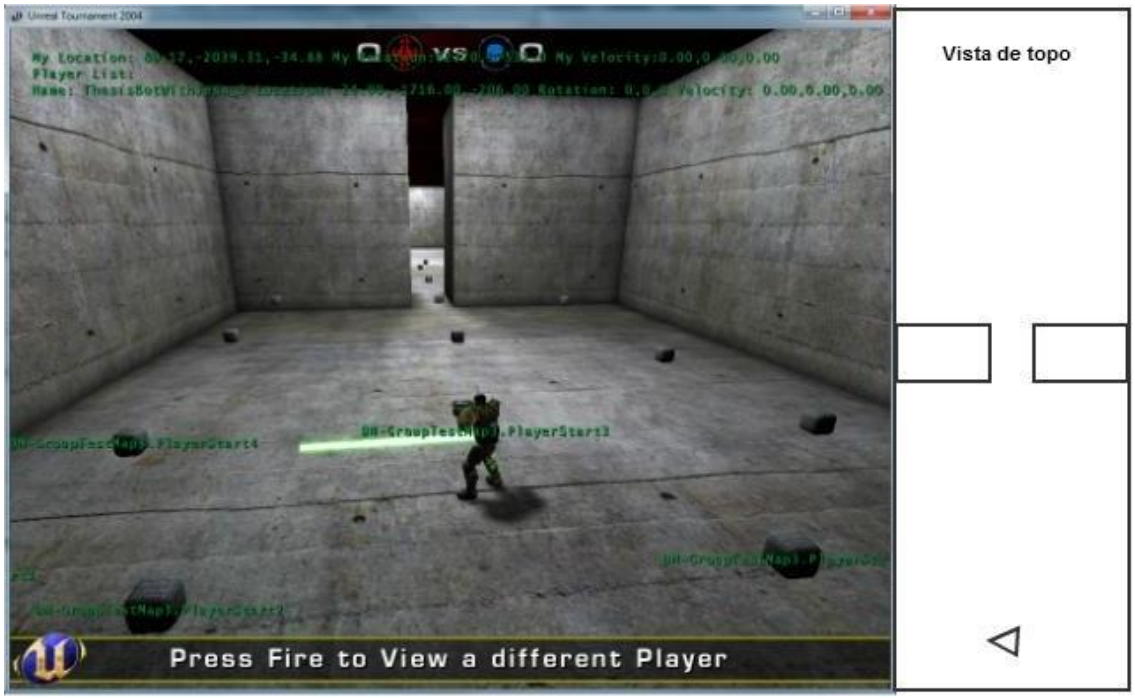

Figura 17: Mapa com corredor para promover o estrangulamento.

Criamos um novo cenário (ver figura 18) para que os *bots* fossem iniciados uns perto dos outros, de modo a testar mais rapidamente alguns comportamentos mais simples tal como os de ataque.

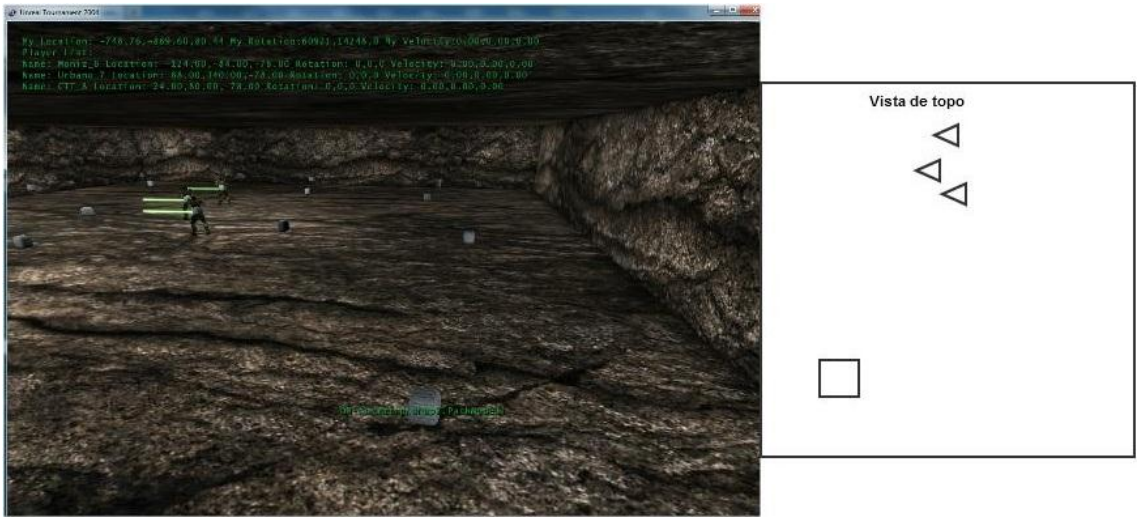

Figura 18: Mapa amplo, com poucos obstáculos, para realizar testes mais simples.

3.3 Comportamento em equipa

Na construção do comportamento em equipa, a movimentação deve ser o primeiro comportamento a ser elaborado, pois servirá como base para comportamentos mais complexos.

No movimento em equipa optámos por realizar uma formação entre 4 elementos da mesma equipa, devido aos *bots* do *Unreal Tournament 2004* serem limitados em certos aspectos, como por exemplo terem um campo de visão de 180 graus e por isto não conseguem detectar inimigos que se encontrem nas suas costas. De forma reduzir esta limitação, a equipa desloca-se em formação diamante (ver figura 19) de modo a que cubram todos os ângulos de visão para uma completa detecção do inimigo.

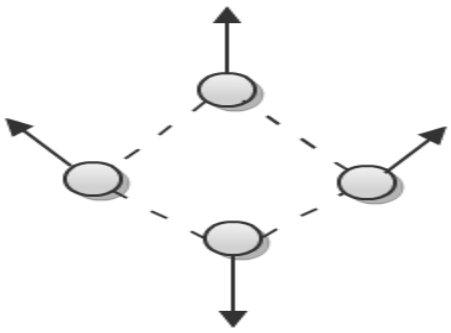

Figura 19: Formação em diamante para quatro elementos. Decidimos que as equipas deveriam ser compostas por quatro elementos para facilitar a sua programação.

Uma equipa de *bots* pode optar por duas abordagens para que possam andar em formação: ou é nomeado um líder dos quatro elementos e os restantes três movem-se a partir da posição e orientação do mesmo (inspirado no jogo *Company of Heroes*); ou utilizamos técnicas simples que facilitam os comportamentos coordenados, isto é, os *bots* com o mesmo objectivo podem reservar caminhos para que outros *bots* da mesma equipa calculem caminhos diferentes e fiquem perto um dos outros. Neste trabalho optámos por seleccionar um líder e aplicar movimentos cinemáticos para realizar na íntegra a formação em diamante, o que era impossível com a outra abordagem.

Uma equipa que tenha um líder leva a vários problemas, tal como a escolha do líder; quais são as funções do líder e dos restantes elementos; e o que acontece quando o líder "morre".

A selecção do líder, neste caso, é realizada com a entrada dos *bots* no jogo, o primeiro a entrar no jogo é considerado o líder, isto porque não existem propriedades diferentes entre os *bots* e é um mecanismo de implementação simples.

A distinção do líder para os restantes elementos é feita através da atribuição de um ID para cada elemento, o líder tem o ID zero, e os restantes elementos têm o ID sequencial a partir do ID do líder.

Na atribuição das funções considerámos que a equipa de *bots* é um comando[2] e que o líder é o comandante. O comandante, como numa formação militar, é responsável por definir e comunicar uma estratégia perante uma situação de risco. Como a decisão só passa por um elemento normalmente é mais eficaz na realização de uma ordem. Por isso, simulando uma situação de comando, o nosso líder também ficará responsável por decidir o que é que a unidade deverá fazer perante certas situações.

Consequentemente o líder terá que comunicar à equipa instruções para que esta se movimente utilizando a formação em diamante.

3.3.1 Sistema de comunicação

O sistema de comunicação, oferecido pelo *Unreal Tournament 2004*, tem dois canais, um canal (*global chat*) que permite que cada jogador presente na arena/cenário possa enviar mensagens de texto para todos os jogadores presentes na arena, e um canal (*team chat*) que permite o envio de mensagens entre todos os elementos da mesma equipa. Este sistema não permite enviar mensagens privadas.

---

[2] Um comando, na metodologia militar [Wiki10], é considerado como uma organização de unidades em que apenas uma, o comandante ou oficial, é responsável por toda a unidade.

Recorrendo ao sistema de mensagens só para equipas do *Unreal Tournament 2004*, foi criado um formato de mensagem para que os *bots* possam enviar e interpretar as mensagens.

idDoTipoDeMensagem ; *idDoBotQueEnviou* ; *idDoBotDestinatário* ; *conteúdo*

A razão pela qual este tipo de formato de mensagem foi adoptado é a seguinte: o parâmetro *idDoTipoDeMensagem* é útil para filtrar qual é o tipo de conteúdo que o *bot* enviou para os restantes. Assim, sempre que um *bot* recebe uma mensagem, sabe que informação é que estará na secção *conteúdo*, o que torna o tratamento desse mesmo conteúdo mais fácil; o parâmetro *idDoBotQueEnviou* é utilizado para identificar quem enviou a mensagem, sendo útil para quando algum elemento envia uma mensagem ao líder e este precise de responder ao mesmo; o parâmetro *idDoBotDestinatário* serve para criar um sistema de mensagens privadas, pois só o *bot* a quem a mensagem se destina a interpreta; o *conteúdo* representa qualquer informação que seja necessária para que o *bot* ou *bots* possam recolher informação para executar uma acção. Quando referimos *bots*¸ é porque este sistema também permite fazer divulgação a todos os *bots* (da mesma equipa) bastando simplesmente nomear um *id* para o *idDoBotDestinatário* em que todos possam interpretar a mensagem.

Os pontos e vírgulas “;”, servem como caracter especial de divisão dos componentes descritos da mensagem.

Com o sistema de comunicação elaborado, o líder pode comunicar com a equipa para que esta realize instruções ou vice-versa.

3.3.2 Movimentação em equipa

Para que a equipa se possa movimentar em formação diamante, aplicamos a movimentação cinemática a todos os elementos excepto o líder.

A movimentação do líder, neste projecto, é incerta, isto é, o líder escolhe aleatoriamente um *waypoint*, calcula, com o algoritmo A*, o caminho até ao destino e executa-o.

Os restantes elementos da equipa, para executar uma movimentação cinemática de forma a realizar a formação em diamante, necessitam da rotação e da orientação do líder. Como só o líder é que contém esta informação, terá que enviar uma mensagem para que os restantes elementos se possam movimentar. Exemplo da mensagem para os elementos se movimentarem:

0;0;9;*posiçãoLíder*;*rotaçãoLíder*

O líder transmite a mensagem, todos os segundos, de modo a cada elemento, quando recebe esta mensagem (tipo zero), saber que no conteúdo terá a posição e rotação do líder e procederá à movimentação. A movimentação é calculada através do sistema que se segue:

$$Angulo_{Posicao} = ((180 - Angulo_{Lider}) \pm Angulo_{Desvio})$$

$$X \begin{cases} X_{Lider} + |Offset \times \cos(Angulo_{Posicao})|, \text{ se } 90 \leq Angulo_{Posicao} \leq 270 \\ X_{Lider} - |Offset \times \cos(Angulo_{Posicao})|, \text{ caso contrário} \end{cases}$$

$$Y \begin{cases} Y_{Lider} + |Offset \times \sin(Angulo_{Posicao})|, \text{ se } 0 \leq Angulo_{Posicao} \leq 180 \\ Y_{Lider} - |Offset \times \sin(Angulo_{Posicao})|, \text{ caso contrário} \end{cases}$$

O ângulo de desvio corresponde ao grau de deslocamento a que queremos que o *bot* esteja perante a direcção do líder (ângulo do líder). A partir desta informação calculamos o ângulo onde supostamente, dada uma distância em relação ao líder (*Offset*), o *bot* irá posicionar-se. A figura 20 ilustra a relação dos ângulos.

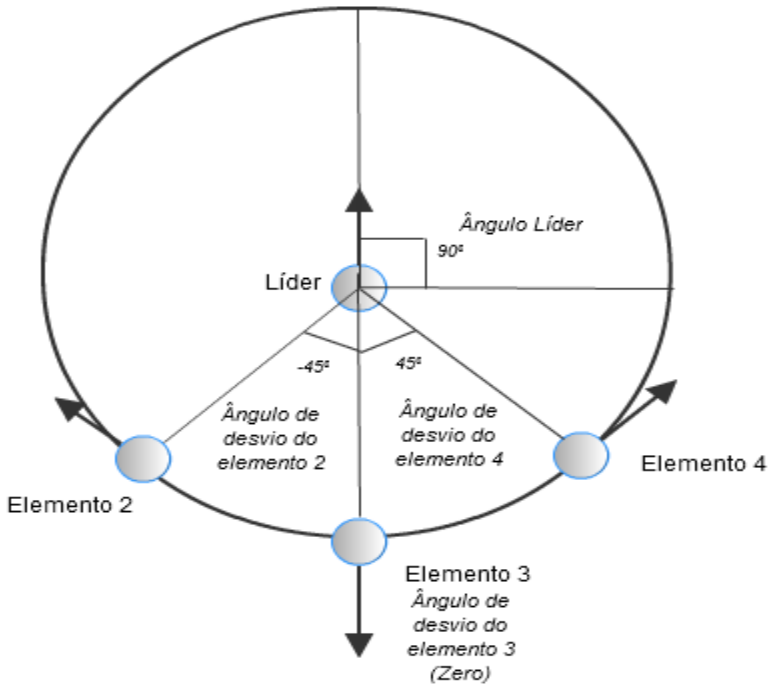

Figura 20: Circulo trigonométrico aplicado para realizar a formação diamante.

Depois de realizados e aplicados os cálculos, os elementos da equipa mantêm uma formação em diamante, e o movimento é realizado quando o líder se move. (ver tabela 11)

Tabela 11: Código *Jess* executado quando um elemento da equipa recebe uma mensagem do líder para acompanhar o movimento da formação.

```
(defmodule PERCEPTION)

(defrule perception
   ?f <- (perception)
   ?x <- (bot) ; ESTE BOT é um def-template contendo todas
               as váriáveis de estado do bot
   =>
   (retract ?f)
   ;SE RECEBE UMA MENSAGEM DO TIPO 9 (Mover em formação)
   (if (and (eq (get-receiver-team-id-from-message) 9))
      then     (bind ?var  (select-place-on-diamond-formation
                              (get-location-from-message)
                              (get-rotation-from-message)))
               ;ATRIBUI AO SEU ESTADO O DESTINO
               (modify ?x (nav-target ?var))
               ;E A ROTACAO FUTURA
               (modify ?x (rot-target
                      (select-rotation-on-diamond-formation ?var)))
   )
   (assert (perception))
   (store RegraDisparada perception)
   (return)
)

(defmodule ACTION)

(defrule go-to-destination
   ?a <- (action)
   ;SE TEM UM DESTINO E UMA ROTAÇÃO
   ?x <- (bot nav-target ?target&~nil) (rot-target ?rot))
   =>
   (retract ?a)
   (assert (action))
   ;PROCEDE AO MOVIMENTO
   (go-to-target ?target ?rot))
   (store "RegraDisparada" go-to-destination)
   (return)
)
```

3.3.3 Tratamento de detecção de colisões

Com a componente da movimentação elaborada, um dos problemas que daí advém são as colisões. O líder não terá esse problema, pois o algoritmo A* apenas pesquisa o caminho entre os *waypoints* (*Pathfinding*), fazendo com que o líder se movimente de *waypoint* em *waypoint* evitando a colisão com os objectos da arena, mas o mesmo já não acontece com os restantes elementos, pois estes movimentam-se fora dos *waypoints*.

Existem duas abordagens para contornar as colisões, ou realizamos um algoritmo que evita a colisão, ou detectamos a colisão e realizamos um comportamento credível.

Existem vários métodos para evitar as colisões, como o *Pathfinding*, mas dificilmente são aplicados para realizar a formação em diamante na íntegra. Para isso, teríamos que ter todos os *waypoints* numa disposição de diamante (losango).

Para que se consiga aplicar a formação em diamante na íntegra, detectar a colisão por impacto do *bot* num obstáculo e proceder a um movimento credível será o mais adequado.

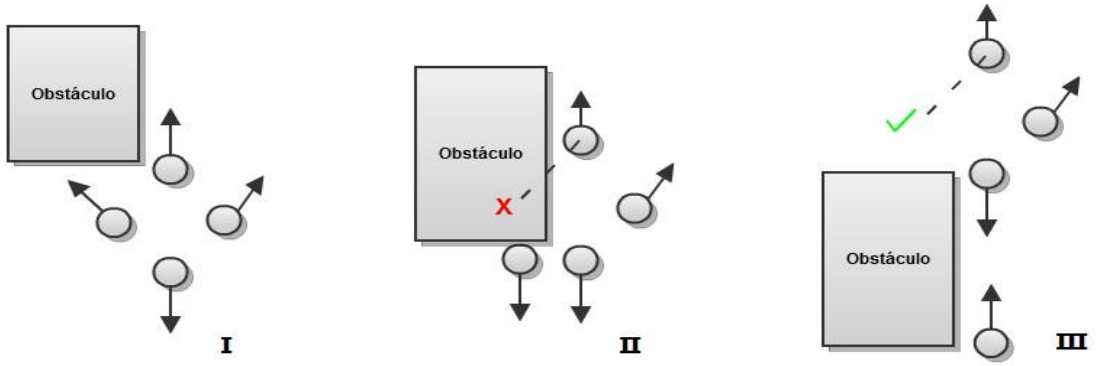

Figura 21: Exemplo de como um *bot* na formação se comporta quando detecta um obstáculo.

De forma a proporcionar um comportamento credível quando o *bot* sofre de um impacto com um obstáculo, este realiza uma rotação de 180 graus (ver figura 21-I) para proporcionar uma sensação em que se encostou à parede para cobrir os restantes elementos da equipa enquanto estes avançam. Para que o *bot* bloqueado possa retomar a formação, o líder fica responsável por verificar quando é que a posição, na formação, do elemento "bloqueado" está livre. Esta verificação é feita através de um sistema de "traços", isto é, o líder executa um espécie de "traço", com origem na sua posição e destino na posição na formação do elemento bloqueado, que tem como objectivo detectar colisões com obstáculos.

Quando o líder, ao executar um "traço", não detecta nenhuma colisão, então a posição do elemento "bloqueado" está livre. O líder envia, a esse elemento, uma mensagem para que possa retornar a sua posição, executando o caminho que o líder realizou para superar o obstáculo (ver figuras 21- II/III, 22 e 23).

O tratamento de colisões fica resolvido, promovendo uma movimentação em formação capaz de superar obstáculos e pontos de estrangulamento.

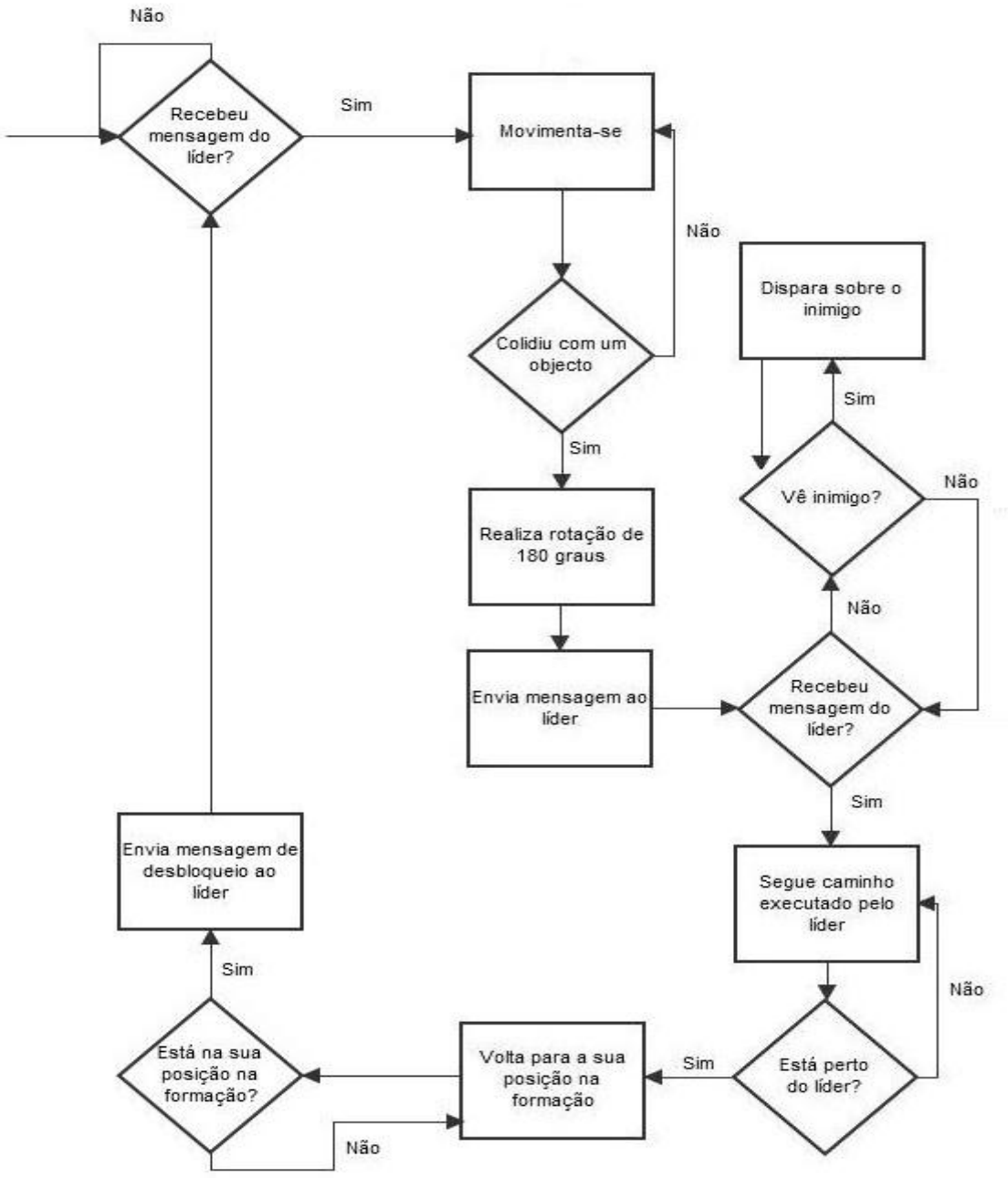

Figura 22: Fluxograma dos elementos da equipa após uma ordem para se moverem em formação.

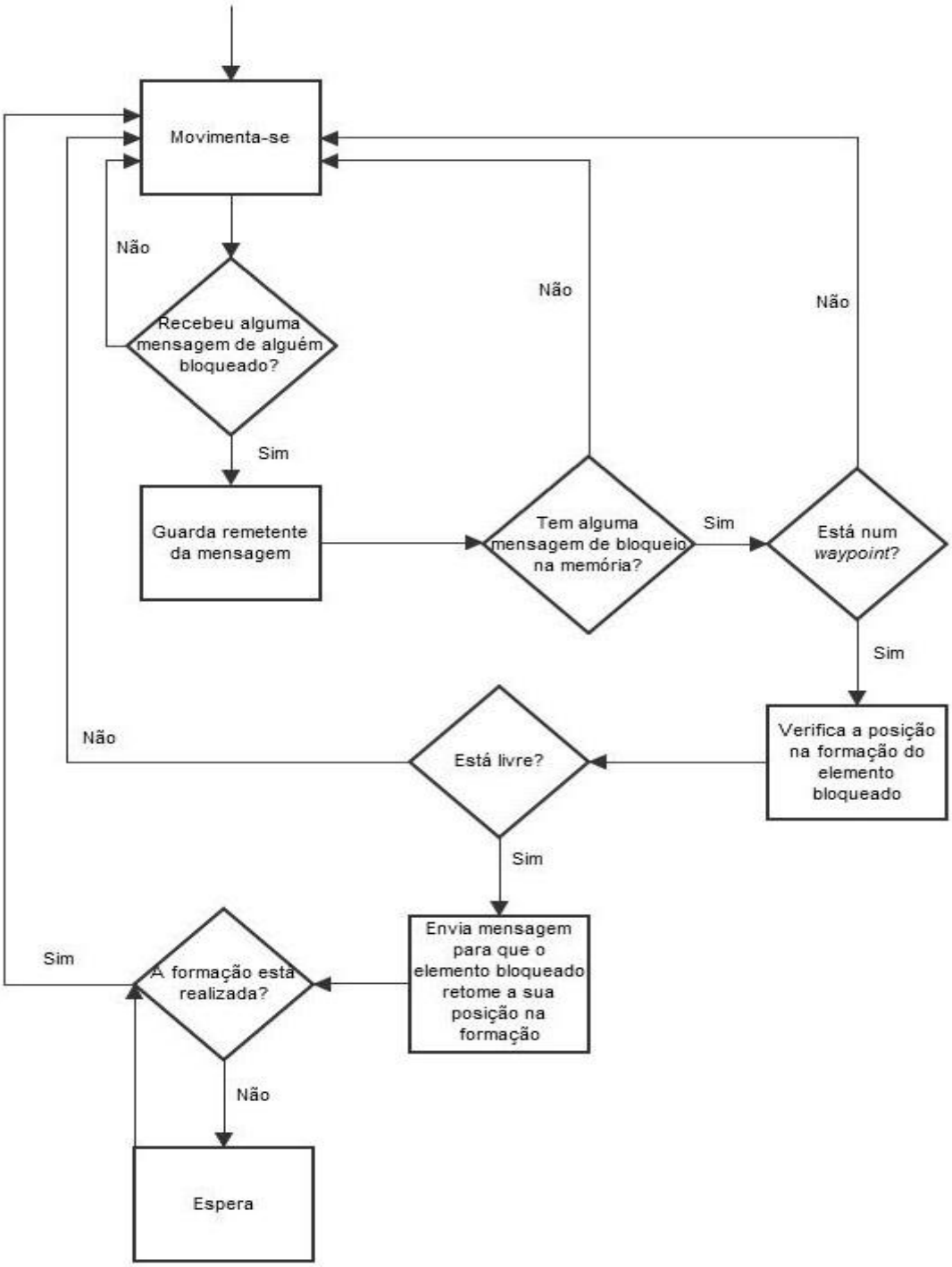

Figura 23:Fluxograma do líder após a movimentação.

3.3.4 Aplicação da DEVELOP-FPS para comportamentos cooperativos

A DEVELOP-FPS foi usada para assistir no desenvolvimento do comportamento cooperativo construído, nomeadamente a formação em diamante entre 4 elementos.

Com os mecanismos actuais de *debug*, torna-se difícil poder controlar e monitorizar um grupo de agentes. Imaginemos um sistema de *logs* a criar informação sobre o estado de todos os *bots* intervenientes na formação, o programador iria ter dificuldades para interpretar toda essa informação.

A DEVELOP-FPS oferece mecanismos para, em tempo real, monitorizar todos os *bots* intervenientes no jogo, que é útil para se consultar o estado e verificar se o *bot* está a "comportar-se" devidamente; e controlá-los um a um, caso seja necessário proceder à criação de um cenário de teste ou mesmo rectificar algum comportamento.

Por exemplo, um dos problemas que obtivemos na realização da formação foi uma descoordenação inicial do movimento entre os elementos, já que estes não se moviam de modo a formar um losango (formação diamante). Este tipo de problema foi facilmente identificado pela observação do mapa global (não era visível um losango entre os 4 elementos). Ao interrompermos a lógica de todos os *bots*, através dos comandos da consola global, verificamos o estado de cada *bot* naquele instante. Acedendo às suas consolas individuais para monitorizar o estado do *Jess*, averiguámos que a regra responsável pela movimentação em diamante era disparada, restringindo assim o problema às funções presentes na parte direita (acção) da regra (Esta regra é responsável por calcular o destino do *bot* e aplicar a movimentação). Procedendo à execução de ambas as funções, na mini-estação de controlo, detectámos que os cálculos que definiam o destino estavam mal implementados.

Sem a DEVELOP-FPS para a detecção do problema, com os métodos tradicionais, teríamos levado bastante mais tempo a identificar o erro, sendo este tempo útil na melhoria dos comportamentos.

Capítulo 4

4.1 Conclusão

Neste documento foi apresentada uma arquitectura genérica, que suporta o desenvolvimento de ferramentas que assistem no desenho, *debug* e execução de *non-player characters* (NPCs ou *bots*) escritos numa linguagem à base de regras (*Jess*) em ambientes de um videojogo do género FPS (*Unreal Tournament 2004*).

A DEVELOP-FPS fornece diversas funcionalidades para o *debugging* de *scripts* declarativos, baseados em regras, para os videojogos FPS, nomeadamente a: facilidade de criação e de configuração de cenários de teste; e a capacidade de interromper e retomar um jogo monitorizando e controlando os jogadores em tempo real. Oferece também um sistema que permite realizar várias experiências para obter, a partir do *output* gerado, uma avaliação sobre os comportamentos testados.

A ferramenta em questão foi utilizada para casos práticos, como a construção e avaliação de vários comportamentos individuais escritos em *Jess* e comprovou-se que consegue identificar rapidamente os erros no *script* e avaliar qual o melhor dos comportamentos. Este tipo de avaliação pode ser útil para definir o nível de dificuldade do comportamento.

Procedemos à construção de comportamentos colectivos com o objectivo de aplicar a DEVELOP-FPS para provar que está apta para o desenvolvimento de vários agentes. Com o desenrolar do desenvolvimento, a DEVELOP-FPS, mostrou-se útil na detecção de problemas nos *scripts*¸ poupando tempo para melhorar os comportamentos.

É correcto concluir que este tipo de ferramentas é muito útil para o processo de construção e avaliação de comportamentos para NPCs, e que devem ser utilizadas em futuros trabalhos que envolvam a programação agentes virtuais para videojogos.

4.2 Trabalho futuro

A DEVELOP-FPS ainda está em estado de desenvolvimento podendo ser estendida com um conjunto de funcionalidades úteis. Uma extensão útil seria adicionar uma mini-estação de controlo à consola global para propagar a todos os *bots*, ou mesmo só a uma equipa, comandos em *Jess* que poderiam ajudar a preparar cenários de teste e a gerir equipas numerosas.

Um futuro melhoramento nos dados reportados relativos a cada experiência seria importante para que o utilizador tivesse uma melhor leitura sobre a experiência ou uma avaliação mais completa sobre os comportamentos dos *bots*. Dados como as armas utilizadas, a variação dos danos infligidos, os caminhos que seguiu ou os *waypoints* que visitou, e que inimigos matou. Pode também ser interessante reportar dados relativos às regras, como por exemplo, a frequência com que foram executadas.

A versão utilizada da API *Pogamut* (V2.4.1) não é perfeita, porque com o desenrolar do projecto foram detectados vários problemas na API, principalmente com métodos que promoviam a aquisição de informação para formar o resultado da experiência.

Por exemplo, o método *killedByWho*() chamado sempre que um *bot* era destruído devolvia identificadores de agentes que não constavam no jogo. A actualização desta API, actualmente na versão 3.2.0 [Pogamut11], seria bastante benéfica em dois aspectos: primeiro, oferece uma biblioteca de desenvolvimento para várias plataformas, nomeadamente para o UT*2004*, para o Unreal Development Kit (UDK) e para mundos DEFCON; segundo, apresenta uma biblioteca mais rica e com os erros mais graves corrigidos. Este processo de actualização não foi realizado porque esta nova versão apenas surgiu numa fase já avançada do projecto e não é compatível com a versão anterior.

Com o comportamento base implementado para uma equipa de agentes, o movimento em formação diamante, existe agora espaço para aplicar a DEVELOP-FPS e desenvolver comportamentos mais sofisticados para equipas, como o contacto com o inimigo, a aquisição de armas e munições, a resolução de objectivos como a captura da bandeira inimigo, e por aí adiante.

Bibliografia

[AiGameDev07] Top 10 Most Influential AI Games (http://aigamedev.com/open/highlights/top-ai-games/), Last visited 06/2011

[AiGameDev10] Todays AI Game Development (http://aigamedev.com/open/upcoming/plus-coming-shortly/), Last visited, 06/2011

[AiGameDev11] Open Challenges in First-Person Shooter (FPS) AI Technology (http://aigamedev.com/open/editorial/open-challenges-fps/), Last visited, 06/2011

[AIGames02] Artificial Intelligence in Games document (http://www.cs.rochester.edu/~brown/242/assts/termprojs/games.pdf), Last visited 07/2011

[Alexander06] Alexander, B. 2006. Flow Fields for Movement and Obstacle Avoidance. In Rabin, S., ed., AIGame ProgrammingWisdom 3.Charles RiverMedia. 159–172.

[Arrabales09] Arrabales, R., Ledezma, A., Sanchis, A.: Towards Conscious-like Behavior in Computer Game Characters. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games, pp. 217–224 (2009)

[Barreiros05] Barreios, J. Percepção e Acção: perspectivas teóricas e as questões do desenvolvimento e da aprendizagem. Faculdade de Motricidade Humana. Universidade Técnica de Lisboa. (http://www.fmh.utl.pt/Cmotricidade/dm/textosjb/texto_7.pdf), Last visited 07/2011

[Brom09] Brom, C., Bída, M., Gemrot, J., Kadlec, R., Plch, T.: Emohawk: Searching for a "Good" Emergent Narrative. In: Iurgel, I.A., Zagalo, N., Petta, P. (eds.) ICIDS 2009. LNCS, vol. 5915, pp. 86–91. Springer, Heidelberg (2009)

[Burkert] Burkert, O., Brom, C., Kadlec, R., Lukavský, J.: Timing in Episodic Memory: Virtual Characters in Action. In: Proceedings of AISB workshop Remembering Who We Are – Human Memory For Artificial Agents, Leicester, UK (to appear)

[Cavazza00] Cavazza, M. AI in computer games: Survey and perspective. Virtual Reality 5 (4), 223-235. (2000)

[Dawson02] Dawson, C. 2002. Formations. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 272–281.

[Dickheiser04] Dickheiser, M. *2004*. Inexpensive Precomputed Pathfiding Using a Navigation Set Hierarchy. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 103–113.

[Follek03] Follek, R.: A Rule-Based System For Playing Poker. Master's thesis, School of Computer Science and Information Systems at Pace University, 2003.

[Friedman03] Friedman-Hill, E. 2003. JESS in action. Manning Publications.

[Gemrot11] Gemrot, J., Brom, C., and Plch, T.: A Periphery of Pogamut: From Bots to Agents and Back Again. F. Dignum (Ed.): Agents for Games and Simulations II, LNAI 6525, pp. 1-18, 2011

[Hancock02] Hancock, J. 2002. Navigation Doors, Elevators, Ledges, and Other Obstacles. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 193–201.

[Hart72] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1972). "Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"". SIGART Newsletter 37: 28–29.

[Hindriks11] Hindriks, K., Riemsdijk, B., Behrens, T., Korstanje, R., Kraayenbrink, N., Pasman, W., and Rijk, L.: Unreal Goal Bots: Conceptual Design of a Reuseble Interface. F. Dignum (Ed.): Agents for Games and Simulations II, LNAI 6525, pp. 1-18, 2011

[IGN09] The 10 Best Game Engines of This Generation (http://uk.pc.ign.com/articles/100/1003725p2.html), Last visited 06/2011

[Isla06] Isla, D. 2006. Probabilistic Target Tracking and Search Using Occupancy Maps. In Rabin, S., ed., AIGame ProgrammingWisdom 3.Charles RiverMedia. 379–387.

[Jess08] Jess Rule Engine (http://www.jessrules.com/), Last visited, 06/2011

[Johnson04] Johnson, G. *2004*. Avoiding Dynamic Obstacles and Hazards. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 161–170.

[Jurney08] Jurney, C. 2008. Company of Heroes Squad Formations Explained, S., ed., AIGame ProgrammingWisdom 4.Charles RiverMedia. 61–69.

[Kaminka02] Kaminka, G., Veloso, M., Schaffer, S., Sollito, C., Adob, pp. bati, R., Marshall, A., -Scholer, A., and Tejada, S.: A flexible test bed for multiagent team research. Communications of the ACM 45(1), 43-45 (2002)

[Lange98] Lange, B. Programming and Deploying Mobile Agents with Java Aglets. Peachpit Press (1998).

[Lent99] Lent M., Laird J., Buckman J., Hartford J., Houchard S., Steinkraus K., Tedrake R.: Intelligent Agents in Computer Games (1999).

[Lidén02] Lidén, L. 2002. Strategic and Tactical Reasoning with Waypoints. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 211–220.

[McLean04] McLean, A. *2004*. Hunting Down the Player in a Convincing Manner. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 151–159.

[Millington06a] Millington, I. 2006. Artificial Intelligence for Games. Morgan Kaufmann. 51-57

[Millington06b] Millington, I. 2006. Artificial Intelligence for Games. Morgan Kaufmann. 57-99

[Millington06c] Millington, I. 2006. Artificial Intelligence for Games. Morgan Kaufmann. 318-343

[Moniz03] Moniz, L., Urbano, P. and Helder, C. AGLIPS: An educational environment to construct behaviour based robots. In Proc. of the International Conference on Computational Intelligence for Modelling, Control and Automation – CIMCA (2003).

[MonoDevelop10] MonoDevelop documentation (http://unity3d.com/support/documentation/Manual/HOWTO-MonoDevelop.html), Last visited 07/2011

[NetLogo11] Official NetLogo website (http://ccl.northwestern.edu/netlogo/), Last visited 07/2011

[Orkin04a] Orkin, J. *2004*. Applying Goal-Oriented Action Planning to Games. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 217–227.

[Orkin04b] Orkin, J. *2004*. Constraining Autonomous Character Behavior with Human Concepts. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 189–197.

[Orkin04c] Orkin, J. *2004*. Simple Techniques for Coordinated Behavior. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 199–206.

[PacmanCompetition11] Ms Pac-Man Competition (http://dces.essex.ac.uk/staff/sml/pacman/PacManContest.html), Last visited 07/2011

[Panda3D10] Official Panda3D website (http://www.panda3d.org/), Last visited 06/2011

[Panda3D11] Lang, Christoph: Panda3D 1.7 Game Developer's Cookbook. Packt Publishing (2011).

[Pinter02] Pinter, M. 2002. Realistic Turning between Waypoints. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 186–192.

[Pittman05] Pittman, D.: Practical Development of Goal-Oriented Action Planning AI. Master's thesis, Faculty of The Guildhall at Southern Methodist University, 2005.

[Pogamut09] Pogamut v2 (https://artemis.ms.mff.cuni.cz/pogamut/tiki-index.php?page=homepage). Last visited, 05/2011

[Pogamut11] Official Pogamut website (http://diana.ms.mff.cuni.cz/main/tiki-index.php), Last visited 06/2011

[Reed04] Reed, C. and Geisler, B. *2004*. Jumping, Climbing, and Tactical Reasoning: How to Get More Out of a Navigation System. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 141–150.

[Reynolds02] Reynolds, J. 2002. Tactical Team AI Using a Command Hierarchy. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 260–271.

[Reynolds04] Reynolds, J. *2004*. Team Member AI in an FPS. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 207–215.

[Reynolds87] Reynolds, C. W., "Flocks, Herds and Schools: A Distributed Behavioral Model," *Computer Graphics*, 21(4), SIGGRAPH '87 Proceedings, pp. 25-34, 1987.

[Scutt02] Scutt, T. 2002. Simple Swarms as an Alternative to Flocking. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 202–208.

[Sterren02a] van der Sterren, W. 2002. Squad Tactics: Team AI and Emergent Maneuvers. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 233–246.

[Sterren02b] van der Sterren, W. 2002. Squad Tactics: Planned Maneuvers. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 247–259.

[Sterren04] van der Sterren, W. *2004*. Path Look-Up Tables – Small Is Beautiful. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 115–129.

[Straatman06] Straatman, R. and Beij, A. 2006. Dynamic Tactical Position Evaluation. In Rabin, S., ed., AIGame ProgrammingWisdom 3.Charles RiverMedia. 389–403.

[Surasmith02] Surasmith, S. 2002. Preprocessed Solution for Open Terrain Navigation. In Rabin, S., ed., AIGame ProgrammingWisdom .Charles RiverMedia. 161–170.

[Tozour04] Tozour, P. *2004*. Search space representations. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 85–102.

[TurtleKit09] Beurier, G., Michel, F.:The TurtleKit Simulation Platform: A Multi-Agent Tool for Designing and Studying Complex Systems.: In AAMAS (2009)

[UDK10] Unreal Development Kit (http://www.udk.com/), Last visited 06/2011

[UDK11] UnrealEngine and UnrealScript official web page (http://www.unrealengine.com/), Last visited, 06/2011

[UDKDebugging10] Official UDK development website, see Debugging section (http://udn.epicgames.com/Three/DevelopmentKitFirstScriptProject.html), Last visited 07/2011

[UnitSteer09] UnitySteer – Steering componentes for unity (http://www.arges-systems.com/articles/35/unitysteer-steering-components-for-unity/), Last visited 06/2011

[Unity09] Goldstone, Will. Unity Game Development Essentials. Packt Publishing; (2009)

[Unity11] Unity game development tool official web page (http://unity3d.com), Last visited, 06/2011

[UT04] Unreal Tournament *2004* official web pager (http://www.unrealtournament2003.com/ut*2004/*), Last visited 06/2011

[Vaugham01] Vaugham R., Stoy K. Player Robot Server, version 0.8c user manual edition, 2001.

[Vogt08] Vogt, J.: Jess to JADE Toolkit (J2J): A Rule-Based Solution Supporting Intelligent and Adaptive Agents. Master's thesis, Software Engineering Group Department of Informatics at University of Fribourg, 2008.

[Wallace04] Wallace, N. *2004*. Hierarchical Planning in dynamic worlds. In Rabin, S., ed., AIGame ProgrammingWisdom 2.Charles RiverMedia. 229–236.

[Wiki10] Command (military formation) (http://en.wikipedia.org/wiki/Command_(military_formation)), Last visited, 06/2011

[Wikipedia11] FPS Game description (http://en.wikipedia.org/wiki/First-person_shooter), Last visited 06/2011

[WoW10] Whitehead II, J., Roe, R.: World of Warcraft Programming: A Guide and Reference for Creating WoW Addons. Wiley; (2010)

[Woyach08] Immersion through Video Games (http://illumin.usc.edu/article.php?articleID=103), Last visited, 06/2011

[Wright00a] Wright I., Marshal J.: RC++: a rule-based language for game AI. Sony Computer Entertainment Europe; (2000)

[Wright00b] Wright I., Marshal J.: The execution kernel of RC++: Rete*, a faster Rete with treat as special case. Department of Earth Science and Engineering at Imperial College London; (2000)

Anexo – Manual de utilizador

Requisitos recomendados

Processador: Dual Core. 3.00GHz

Memória RAM: 4 GB DDR2

Placa Gráfica: ATI Radeon HD 4550 (Ou equivalente)

Espaço em disco: 7 GB de memória livre.

Instalação utilizador

Para poder usufruir, tanto da consola individual com da consola global para assistir no desenvolvimento de *bots* escritos em *Jess* para o videojogo *Unreal Tournament 2004* terá de instalar os seguintes programas (todos os ficheiro necessários estão presentes no cd que acompanha esta dissertação):

1. ***Unreal Tournament 2004***
    a. Duplo clique no executável e basta seguir os passos descritos no instalador.
2. **JAVA Runtime v6** (ou superior) (Pode fazer o download aqui: http://www.java.com/pt_BR/download/index.jsp)
    a. Duplo clique no executável e basta seguir os passos descritos no instalador.
3. ***Pogamut* v 2.4.1**

Na instalação terá que colocar a directoria onde o videojogo *Unreal Tournament 2004* foi instalado.

Instalação programador

Se for para realizar alguma alteração às consolas como à própria arquitectura do projecto, então deverá proceder à instalação dos seguintes programas:

1. ***Unreal Tournament 2004***
    a. Duplo clique no executável e basta seguir os passos descritos no instalador
2. **JAVA SDK v6** (ou superior) (Pode fazer o download aqui: http://www.oracle.com/technetwork/java/javase/downloads/index.html)

    a. Duplo clique no executável e basta seguir os passos descritos no instalador.
3. ***NetBeans* v6.0.1** (versão Java SE, 26MB suffices) - IDE requerido pelo *Pogamut* v2.4.1, para programar os *bots* para o *Unreal Tournament 2004*.
    a. Para Windows 7 apenas a distribuição ZIP funciona.
4. ***Pogamut* v 2.4.1** – Biblioteca que disponibiliza o acesso à programação de *bots* para o videojogo *Unreal Tournament 2004*
    a. Na instalação terá que colocar a directoria onde o videojogo *Unreal Tournament 2004* foi instalado.
    b. Também terá de colocar a directoria onde o *NetBeans* foi instalado
5. ***Jess* API** – Incluir as bibliotecas "Jess.jar" e "jsr94.jar" no projecto do *NetBeans*.

Para usufruir do trabalho realizado basta aceder ao projecto, este também contido no cd que acompanha esta dissertação.

Correr a aplicação

1. Executar o *Unreal Tournament 2004* como servidor em modo *GameBots*. Existem já uns ficheiros "*.bat*", no cd, que correm alguns exemplos de servidores.
2. Na consola do Windows (Cmd)
    a. Ir até à directoria onde está o ficheiro "ThesisBotWithJess.jar"
    b. Executar o seguinte comando: "Java –jar ThesisBotWithJess.jar".

Como usar a aplicação

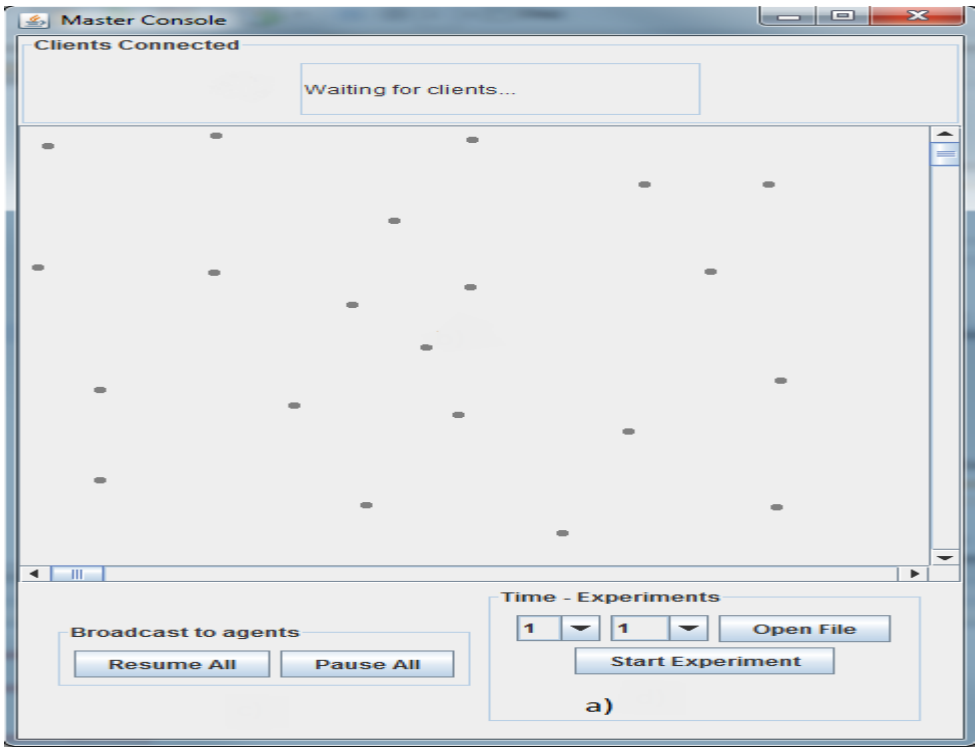

Figura 24: Anexo - Consola global

Iniciar experiência

1. Primeiro configurar um ficheiro XML (ver tabela 12), com os intervenientes na experiência.
   a. Existem uns ficheiros com uns comportamentos básicos que podem ser experimentados. (hunter.clp, dodgeBehavior.clp e dummy.clp)
2. Abrir o ficheiro previamente configurado. (Figura 24 a) “Open File”)
3. Definir quanto tempo deverá durar cada experiência. (Figura 24 a) “Time”)
4. Definir quantas iterações deve correr. (Figura 24 a) “Experiments”)
5. Correr experiência. (Figura 24 a) “Start Experiment”)

Tabela 12: Anexo - Ficheiro XML para configuração da experiência.

```
<?xml version="1.0"?
<!-- There is only two team, team 0 and team 1 -->
<experiment>
  <bot>
        <name>Moniz</name>
        <team>0</team>
<!--Colocar a directoria onde o ficheiro está, directorias separadas com "\\"--
>
        <fileName>. \\hunter.clp</fileName>
<!—Se começam ou não pausados, útil para preparar cenários de teste-->
        <pause>true</pause>
  </bot>
  <bot>
        <name>Urbano</name>
        <team>0</team>
        <fileName>.\\dodgeBehavior.clp</fileName>
        <pause>true</pause>
  </bot>
  <bot>
        <name>CTT</name>
        <team>1</team>
        <fileName>.\\dummy.clp</fileName>
        <pause>false</pause>
  </bot>

</experiment>
```

Funcionalidades da consola global

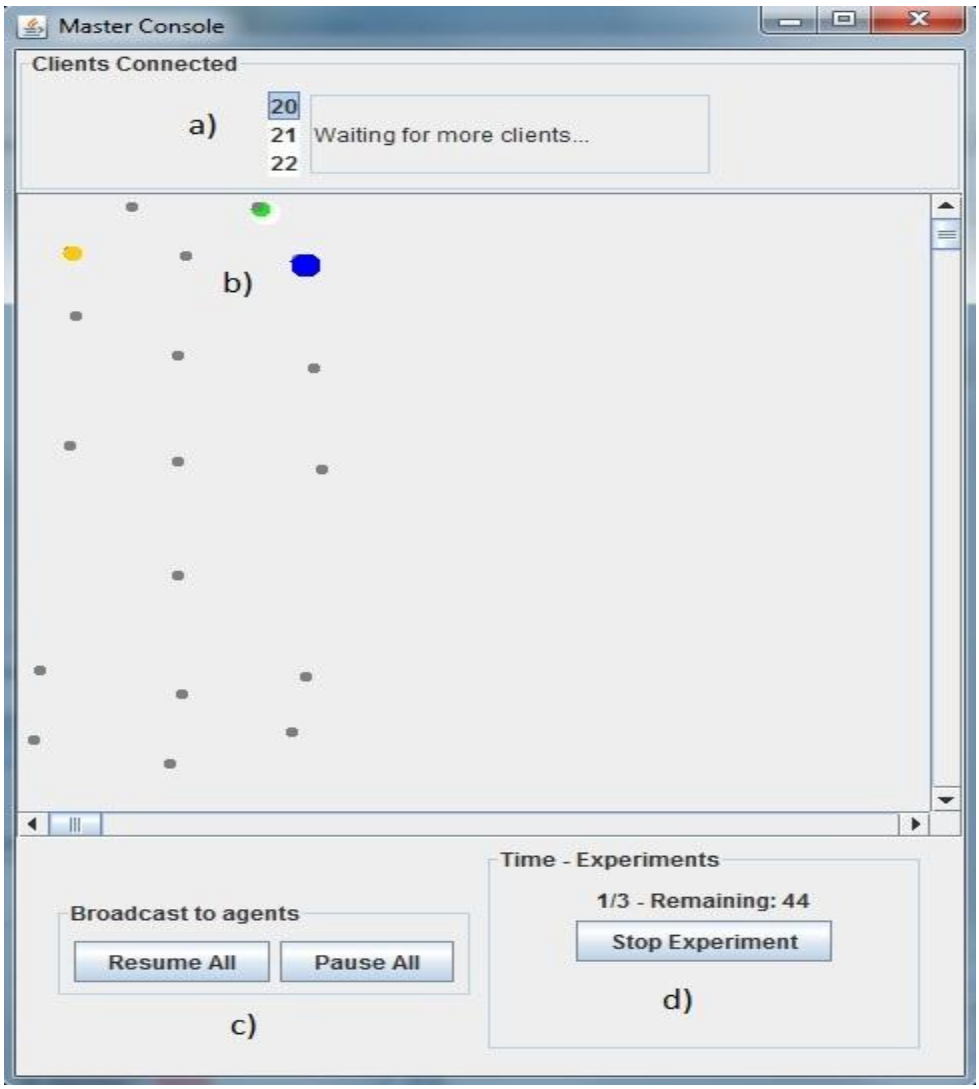

Figura 25: Anexo - Consola global detalhada.

a. Lista de Ids dos *bots*, contidos na experiência. Serve para escolher o *bot* pretendido e dá acesso à sua consola individual.
b. Mapa com vista aérea do cenário, com os *bots* a cor e os *waypoints* a cinzento.
c. Botões para resumir e pausar a lógica de todos os agentes.
d. Monitorização do decorrer da experiência, "1/3" (Número da iteração corrente / Número de iterações), "*Remaining*: 44" (Tempo que falta para terminar a iteração corrente em segundos), e botão "*Stop Experiment*" para terminar a qualquer altura a experiência.

Funcionalidades consola individual

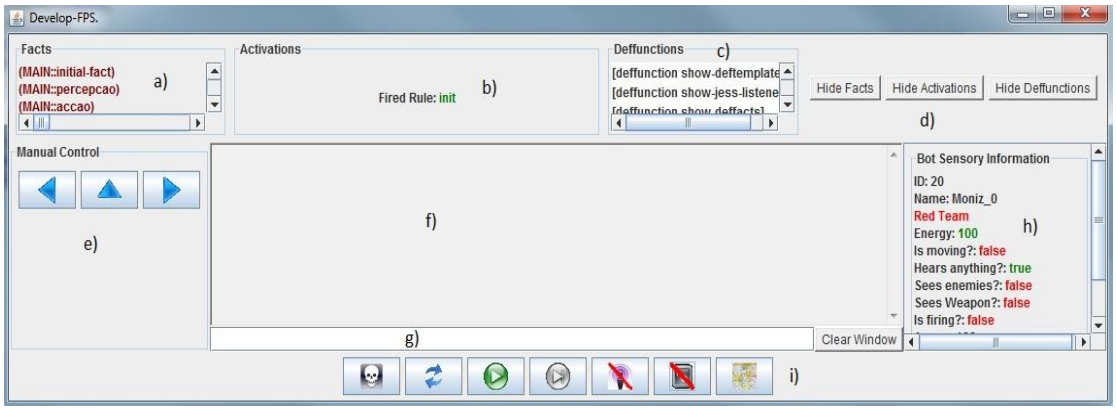

Figura 26: Anexo - Consola individual.

a. Visualização dos factos contidos na memória de trabalho do *Jess*
b. Lista de activações e a regra disparada.
c. Lista de funções definidas pelo utilizador e algumas utilitárias do *Jess*
d. Botões para omitir as informações anteriormente descritas.
e. Controlo manual do *bot* (apenas é possível se o *bot* estiver em pausa), com rotação para a esquerda e direita (botão da esquerda e da direita respectivamente) e movimento para onde está voltado (botão para o meio).
f. Janela de *output* do *Jess*.
g. Linha de comandos para exercer qualquer função sobre *Jess*.
h. Informação sensorial em *runtime* do *bot*.
i. Botões de controlo (da esquerda para direita)
    1. Destrói do *bot*.
    2. Reload. Destrói toda a informação contida no *Jess* e substituía pela nova informação contida no *script* do comportamento em questão.
    3. Resume e pausa o *bot* de exercer os seus comportamentos
    4. Executa um *step*
    5. Omite/Mostra a informação sensorial h) do *bot*
    6. Omite/Mostra a informação do *Jess* a), b), c), d)
    7. Omite/Mostra o mapa individual. (Ver figura 27)

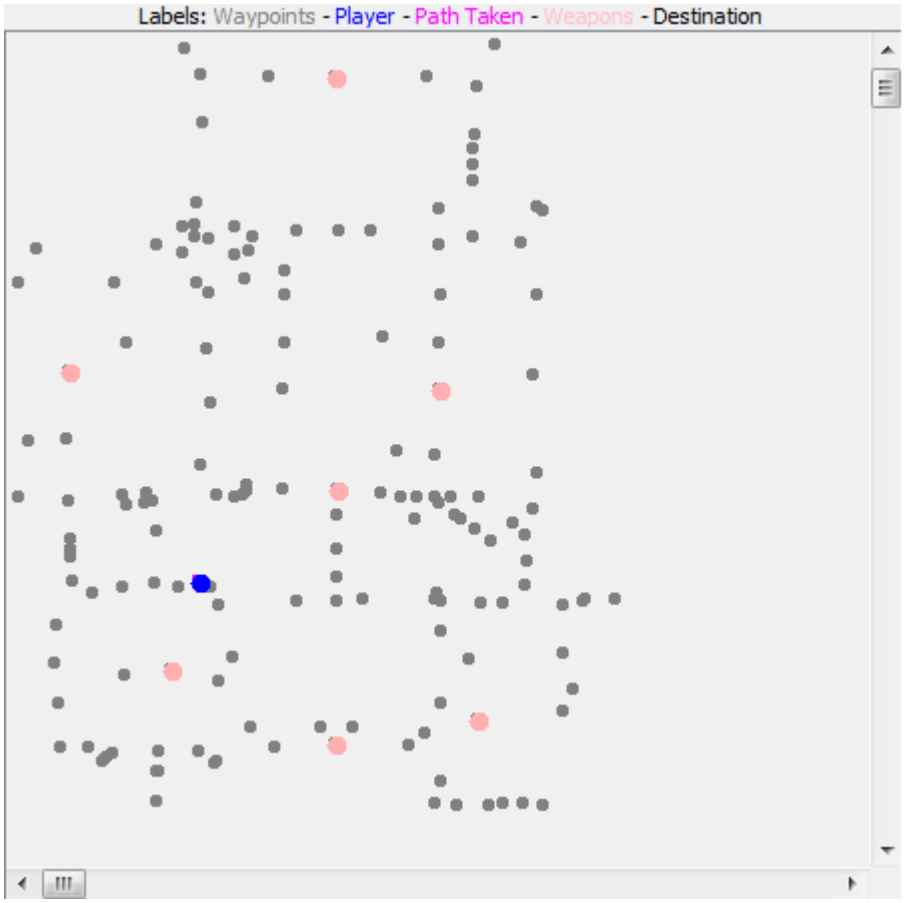

Figura 27: Anexo - Mapa 2D

O mapa individual além de mostrar as armas, *waypoints* e a posição do *bot*, tem também como funcionalidade colocar o *bot* num *waypoint* seleccionado. Para isso basta que o utilizador clique num *waypoint* pretendido situado no mapa.

Aviso: Nem sempre é possível executar o caminho desde a localização do *bot* até ao *waypoint* seleccionado.

Construir *bots* em *Jess*

Para construir comportamentos em *Jess* terá que seguir o seguinte modelo:

Tabela 13: Anexo - Arquitectura Jess

```
(deffacts SETUP
  (percepcao)
  (accao))
(defmodule INIT)
(defrule init
  =>
  (store RegraDisparada init)
  (return)
)
(defmodule PERCEPCAO)
(defrule perception
  ?f <- (percepcao)
  =>
  (retract ?f)
  (assert (percepcao))
  (store RegraDisparada perception)
  (return)
)
(defmodule ACCAO)
  (defrule jump
  ?a <- (accao)
  =>
  (retract ?a)
  (assert (accao))
  (store "RegraDisparada" jump)
  (return)
  )
```

Existem, também no cd, mais dois comportamentos exemplos para usufruir. (hunter.clp, dodgeBehavior.clp)