Técnicas de Sistemas Inteligentes Aplicadas ao Desenvolvimento de Jogos de Computador

Dissertação de Mestrado

Tese de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação como parte dos requisitos para obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Engenharia de Computação

Autor: Victor Kazuo Tatai

Bolsista Fapesp Processo 00/07631-6

Orientador: Dr. Ricardo Ribeiro Gudwin

RESUMO

O uso de novas técnicas de sistemas inteligentes em jogos de computador promete ampliar a interatividade, jogabilidade e escopo dos mesmos, provendo tanto oponentes quanto parceiros mais inteligentes, tornando-os jogadores independentes mas capazes de trabalhar em grupo, seguir objetivos, e de aprenderem. Nessa tese realizamos uma análise do estado da arte no que concerne jogos de computador e sistemas inteligentes, apresentando também o trabalho feito para o controle de bots (jogadores automatizados para jogos em primeira pessoa tridimensionais) utilizando novas técnicas de sistemas inteligentes ainda não exploradas nessa área. Por outro lado, esta tese visa também apresentar uma nova ferramenta formal-computacional, as RP-Nets, demonstrando sua utilização em uma aplicação tempo real como o é um jogo de computador.

Palavras-chave: sistemas inteligentes, jogos de computador, inteligência artificial, RP-Nets, semiótica computacional.

Abstract

The use of novel intelligent systems techniques in computer games holds the promise to a whole new level in computer game interactivity and playability, widening the player´s range of possibilities and further enhancing the gaming experience. These techniques may be used to control either intelligent opponents or friends, which should be capable of working independently, following objectives, working as a group and learning from experience. In this dissertation we provide an analysis of the state of the art in computer games under the intelligent systems perspective. We also describe the work developed in the control of bots (automated players on first-person three-dimensional games) using novel intelligent system techniques still not used on this area. This dissertation also introduces a new computational tool, RP-Nets, and demonstrates its use in control of a real-time application.

Keywords: intelligent systems, computer games, artificial intelligence, RP-Nets, computational semiotics.

Agradecimentos

São muitas as pessoas a agradecer neste trabalho, culminação de vários anos de estudo. Primeiramente, gostaria de agradecer à minha família pelo suporte proporcionado em todas as horas, apoiando-me e provendo sempre um ambiente propício a realização de minhas tarefas. Também gostaria de agradecer em especial a meu orientador, por ter me aceito no programa de mestrado e pelo direcionamento proporcionado. Gostaria também de agradecer a todos meus amigos, em especial ao Antônio e ao grupo de Semiótica Computacional, que colaboraram não só em idéias mas também em companheirismo e distração nas horas de stress. Por fim agradeço ainda à FAPESP pela bolsa concedida e por acreditar em meu trabalho, e à UNICAMP por me acolher tanto durante a graduação quanto a pós-graduação durante os oito anos mais enriquecedores de minha vida.

“Just work and no play made Jack a dumb boy.”

Adágio popular americano.

Índice

Lista de Figuras

Lista de Acrônimos

BMSA Best-Match Search Algorithm

CS Counter-Strike

CTRNN Continuous Time Recurrent Neural-Network

DLL Dynamic-Link Library

FFSM Fuzzy Finite-State Machine

FSM Finite-State Machine

FPS First-Person Shooter

HL Half-Life

HUD Heads-Up Display

IA Inteligência Artificial

IDA* Iterative-Deepening A*

MMORPG Massive Multiplayer Online RPG

MOO MUD Object-Oriented

MUD Multi-User Dungeon ou Multi-User Dimension

MUSH Multi-User Shared Hallucination

NPC Non-Player Character

RP-Net Rede de Processamento de Recursos

RPG Role Playing Game

RTS Real Time Strategy

SI Sistemas Inteligentes

STL Standard Templates Library

UML Unified Modeling Language

Capítulo 1

Introdução

1.1 Motivação

Os jogos de computador / entretenimento eletrônico representam atualmente um mercado mundial de bilhões de dólares [Ass02]. Este mercado tem sido pouco explorado tanto pela indústria como pelo mundo acadêmico no Brasil, a participação brasileira neste setor está limitada principalmente à importação e adaptação de produtos oriundos do exterior. As poucas iniciativas brasileiras de desenvolvimento baseiam-se na utilização de modelos já consagrados no exterior, sem apresentar grandes inovações ou números expressivos de vendas. Através desse projeto busca-se alcançar uma capacitação inicial diferenciada na construção de IAs[1] para jogos, de forma a gerar parte do conhecimento inicial necessário ao ingresso nesse mercado mundial altamente competitivo.

Como nota Peter Molyneux, um dos maiores projetistas de jogos da atualidade, "a inteligência artificial é o futuro dos jogos". Isto se torna cada dia mais evidente pois à medida em que as plataformas utilizadas aumentam em capacidade computacional (tanto no que con-

[1] IA - Inteligência Artificial - Na indústria de jogos, o termo IA é utilizado como um jargão para denotar os módulos de software responsáveis pelo comportamento "inteligente" realizado por componentes do jogo (oponentes ou aliados) durante seu funcionamento. Apesar da conexão que se pode fazer dessa terminologia com a área de pesquisa conhecida como inteligência artificial, é importante distinguir um conjunto de metodologias dos componentes de software que se utilizam dessas metodologias. Em nosso texto, utilizaremos o termo IA de maneira intercambiável para representar tanto um conjunto de metodologias como os componentes de software que as utilizam.

cerne a parte lógica como gráfica), um maior volume de processamento torna-se disponível para a inteligência dos jogos. Vemos assim o foco tecnológico da indústria de jogos passando de um enfoque inicial maior na computação gráfica e se direcionando para as técnicas de inteligência artificial (sem perdermos de vista o foco principal, que é a jogabilidade). Nesse contexto, a utilização de recursos de inteligência artificial mais elaborados que as simples bases de regras tradicionais constitui uma grande vantagem competitiva. É interessante notarmos que embora existam exemplos isolados da utilização de técnicas mais avançadas de inteligência artificial em jogos comerciais (redes neurais, algoritmos genéticos, lógica fuzzy, aprendizado, etc.), a grande maioria dos jogos ainda têm sua inteligência baseada em métodos mais tradicionais, tais como máquinas de estado, sistemas de regras e utilização de scripts.

Do ponto de vista acadêmico, por outro lado, os jogos de computador apresentam-se como uma excelente plataforma para o teste e validação de novas metodologias e algoritmos, que encontram nos ambientes de jogos a riqueza e a complexidade de ambientes sofisticados, mas que ao contrário do mundo real são ambientes controlados, proporcionando aplicações ideais para testar estes procedimentos que embora complexos podem não estar preparados para lidar com um ambiente real. Assim, podemos explorar desde jogos tradicionais, tais como xadrez, damas, go-moku [Hef02] (entre outros), até os ambientes de realidade virtual proporcionados pelos jogos mais modernos. Tais jogos apresentam novos desafios e oportunidades sobre o ponto de vista da inteligência artificial, devido ao alto grau de interação com o usuário e por possuírem ambientes ricos e dinâmicos, aproximando-os assim do mundo real, mas mantendo-os ainda sob ambiente controlado. São estas as características procuradas por este projeto.

A validade do uso dos jogos como plataforma de testes pode ser evidenciada pelo crescente interesse da comunidade científica neste tema, interesse esse que pode ser observado em diversos trabalhos como os desenvolvidos na Carnegie Mellon University pelo grupo Oz [OZ], na Brandeis University pelo grupo DEMO [Pol] e pelo projeto Excalibur [Exc] realizado de forma conjunta por diversos institutos de pesquisa alemães, e ainda por conferências como [AAA99, AAA00]. Os jogos de computador por seu caráter multidisciplinar prestam-se a demonstrar a validade das mais diversas técnicas de inteligência artificial, desde reconhecimento de linguagem natural, modelos de cognição e interação, até mecanismos complexos de planejamento, busca e aprendizagem.

Em particular neste trabalho realizamos uma investigação sobre as técnicas de sistemas inteligentes utilizadas atualmente em jogos de computador, apresentando também nossa contribuição na forma de uma nova arquitetura para o controle de agentes em jogos de computador. Esta arquitetura está baseada em diferentes idéias colhidas na literatura, e em resultados pregressos obtidos pelo DCA / FEEC / UNICAMP na área de sistemas inteligentes, principalmente os modelos das chamadas Redes Semiônicas, desenvolvendo-os para caracterizar aquilo que chamamos aqui de **Redes de Processamento de Recursos**, ou **RP-Nets**. Essas redes foram utilizadas para sintetizar a inteligência dos agentes presentes no jogo, objeto de nosso estudo.

Assim sendo, o objetivo desta tese é duplo:

- *Primeiro*, de realizar uma *pesquisa sobre o estado da arte* no que concerne a utilização de técnicas de sistemas inteligentes (SI) em jogos de computador, e o de apresentar a **implementação de uma nova arquitetura** baseada nesse estudo;
- *Segundo*, **apresentar o conceito de RP-Nets** e demonstrar sua utilidade no contexto de uma aplicação tempo-real como é um jogo de computador.

Na seção a seguir detalhamos o processo de desenvolvimento de jogos de computador de forma a melhor situar o leitor quanto ao trabalho a ser apresentado.

1.2 Desenvolvimento de Jogos de Computador

Apresentamos aqui uma breve explicação ao leitor de como se dá o processo de desenvolvimento de um jogo de computador e sua arquitetura, situando o trabalho realizado nesta tese.

O processo de desenvolvimento de jogos de computador possui certas características distintas em relação ao processo tradicional de desenvolvimento de software, principalmente pelo fato de serem projetos razoavelmente grandes (envolvendo até dezenas de pessoas, por uma duração que pode se estender durante anos) e por constituirem até certo ponto produtos artísticos, resultantes da colaboração entre equipes multi-disciplinares. O processo de desenvolvimento envolve em linhas gerais os seguintes passos:

1. Idealização das linhas gerais do jogo a ser desenvolvido, envolvendo em alguns casos *story-boards*, arte-conceito, etc.

2. Implementação do motor do jogo (ou re-utilização de um motor já existente), e implementação de protótipo. Implementação de ferramentas para artistas / projetistas de jogos (*game-designers*).

3. Realização do projeto de fases (*level-design*), concomitantemente à realização do trabalho de arte e som. Envolve muitas vezes pessoal com pouca capacitação em computação, como artistas gráficos, projetistas de jogo (responsáveis pelo projeto lógico das fases, comportamentos automatizados e jogabilidade do jogo) e engenheiros de som.

4. Depuração, teste do jogo (*play-testing*, realizado por jogadores) e documentação.

5. Distribuição.

De forma simplificada, a arquitetura de um jogo de computador envolve os seguintes componentes (Figura 1.1):

- **Motor** (*engine*): o motor é responsável por implementar o módulo de *renderização* gráfica do jogo, freqüentemente ainda coordenando os outros componentes.

- **Rede**: o componente de rede é responsável por realizar a comunicação com jogadores externos (via rede) ou servidores dedicados.

- **Som**: componente em geral fortemente integrado ao motor, responsável por gerenciar os sons e músicas do jogo.

- **IA**: componente responsável por implementar o controle dos oponentes e aliados automatizados. Note que neste contexto nos referimos à IA como o *componente* que se utiliza de técnicas e algoritmos de inteligência artificial para efetuar o controle do jogo, e não às técnicas de inteligência artificial em si.

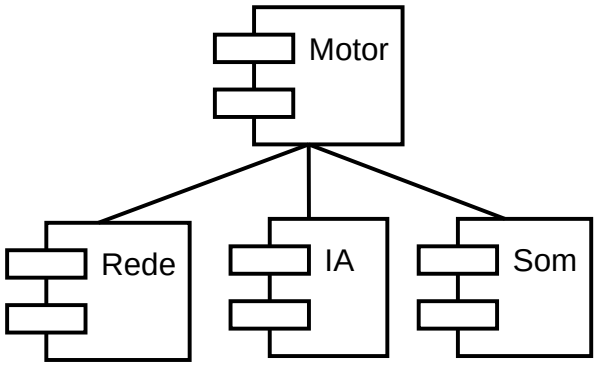

Figura 1.1: Arquitetura de um jogo de computador em UML.

Muitos jogos permitem ainda a modificação de seu comportamento através de componentes externos. Esses componentes externos são comumente denominados **mods** para um jogo (de *mod*ificação).

É interessante notar ainda o fenômeno da popularidade dos jogos *multiplayer*, que se apresentam como uma plataforma especialmente interessante à IA em jogos. Jogos multiplayer são jogos que permitem que múltiplos jogadores joguem simultâneamente em um mesmo cenário, interconectados através de uma rede local ou via Internet. Esses jogos, por permitirem que múltiplos jogadores conectem-se entre si através de mecanismos bem definidos, permitem a implementação de IAs cujos sensores e atuadores correspondam aos sensores e atuadores de um jogador humano (ao menos no que concerne ao jogo), permitindo assim a comparação de comportamento entre ambos.

De particular interesse para o trabalho aqui apresentado é o jogo Counter-Strike (CS), que é dentre os jogos multiplayer um dos mais populares. CS foi o jogo escolhido para a implementação da arquitetura apresentada no Capítulo **??**. Com a utilização de um jogo multiplayer comercial sofisticado como o CS, pudemos evitar o custo de uma implementação de outros componentes, e pudemos nos concentrar no desenvolvimento somente da IA a ele agregada. Isso foi feito por meio do desenvolvimento de *bots* para a implementação de oponentes e aliados inteligentes. Um **bot** é um termo cunhado especificamente para jogos multiplayer tridimensionais em primeira pessoa (primeira pessoa pois o foco da visão do jogador é realizado através da "visão" do seu representante no mundo virtual) que denota oponentes ou aliados automatizados.

1.3 Estrutura da tese

Esta tese encontra-se estruturada da seguinte forma:

No **Capítulo 1** efetuamos uma breve introdução e detalhamos a motivação do trabalho realizado.

No **Capítulo 2** realizamos uma apresentação das principais técnicas de sistemas inteligentes em uso atualmente nos jogos de computador.

No **Capítulo 3** apresentamos uma análise tanto do mercado quanto do ambiente acadêmico sob a ótica dos sistemas inteligentes e jogos de computador.

No **Capítulo 4** introduzimos as RP-Nets, a ferramenta formal-computacional utilizada para modelagem de sistemas inteligentes no decorrer da tese.

No **Capítulo 5** apresentamos o modelo computacional implementado.

No **Capítulo 6** apresentamos as conclusões e trabalhos futuros.

No **Apêndice A** listamos as hiper-referências para os jogos apresentados no decorrer da tese.

No **Apêndice B** apresentamos a interface utilizada para o controle do bot de Counter-Strike.

E, finalmente, a **Bibliografia** contém as referências bibliográficas citadas nos capítulos anteriores.

Capítulo 2

Sistemas Inteligentes em Jogos de Computador

Procedemos neste capítulo a uma análise das técnicas de sistemas inteligentes (SI) mais relevantes para os jogos de computador. Cabe aqui notar que o termo SI engloba tanto a inteligência artificial clássica [Nil80], baseada em lógica de primeira ordem e sistemas especialistas, quanto a inteligência computacional [ZIR94], que incorpora sistemas nebulosos, redes neurais artificiais e sistemas evolutivos.

Jogos de computador têm cativado há muito tempo o interesse da comunidade de sistemas inteligentes, especialmente os jogos chamados "tradicionais", como xadrez, damas, go-moku, dentre outros. Certamente a vitória do computador Deep Blue sobre Kasparov [SP97, Hed97] (e mesmo o mais recente empate do grão-mestre número um do mundo, Kramnik, e o computador Deep Fritz) foi de grande importância para atrair a atenção tanto da comunidade científica quanto do público em geral para a relevância da IA em jogos. Também de importância foram os grandes avanços dos computadores em outros jogos, como por exemplo damas [SLLB96]. Entretanto, tais vitórias se devem sobretudo não a inovações no campo de sistemas inteligentes mas sim ao uso do conhecimento de especialistas no projeto do sistema e a avanços de hardware que permitiram aos computadores usar de "força bruta" para superar seus oponentes humanos.

Por outro lado, os jogos modernos de computador se apresentam como uma nova plataforma

desafiadora para testes de técnicas mais avançadas de sistemas inteligentes, provendo ambientes altamente dinâmicos e complexos, com múltiplos objetivos e decisões a serem tomadas em tempo real, sendo assim muito diferentes dos jogos tradicionais. Isso torna os problemas a serem resolvidos mais próximos daqueles encontrados no "mundo real", sendo outras técnicas de SI mais apropriadas, como a inteligência computacional, a inteligência artificial distribuída, entre outras.

Atualmente muitos jogos de computador utilizam alguma técnica de SIs. Entretanto, recursos mais avançados, incorporando técnicas de inteligência computacional, só estão presentes em uma minoria dos jogos disponíveis comercialmente (alguns desses jogos serão analisados em maior detalhe no Capítulo 3). Neste capítulo procedemos assim com uma descrição das técnicas de sistemas inteligentes mais freqüentemente utilizadas em jogos de computador modernos, que são:

- Algoritmos genéticos.
- Redes neurais artificiais.
- Sistemas nebulosos.

Antecedemos à análise das técnicas de SI com uma breve introdução às técnicas de implementação de agentes em jogos mais utilizadas na atualidade, as *máquinas de estados finitos*, os *scripts* e as *estratégias de busca em árvore tradicionais*. Isso nos permitirá melhor contextualizar o uso das técnicas de SI e suas vantagens e desvantagens quando comparadas a estas estratégias mais bem estabelecidas na indústria.

2.1 Máquinas de Estados Finitos e Scripts

Ambas as técnicas são aqui apresentadas de forma conjunta pois são as mais utilizadas em jogos atualmente, freqüentemente de forma combinada. Cabe notar também que ambas as técnicas não se enquadram na categoria de sistemas inteligentes, sendo analisadas nesta seção por serem técnicas muito utilizadas em jogos de computador. Passamos assim primeiramente a descrever as máquinas de estados finitos [Cas93], passando em seguida aos scripts e sua utilização em jogos de computador associado às máquinas de estados finitos.

Formalmente, uma máquina de estados finitos é definida como sendo a seguinte tupla:

$$(E, X, f, x_0, F)$$

na qual:

- E é um conjunto finito de eventos
- X é um conjunto de estados finito
- f é uma função de transição de estado, $f : X \times E \rightarrow X$
- x_0 é um estado inicial, $x_0 \in X$
- F é um conjunto de estados finais, $F \subseteq E$

Geralmente é conveniente representar de forma gráfica uma máquina de estados finitos, o que é feito através de um diagrama de transição de estados. Esse diagrama nada mais é que um grafo direcionado no qual os nós representam os estados e os arcos representam eventos. Assim sendo um arco rotulado como e que conecta dois nós saindo de x e indo para x' representa uma transição do estado x para o estado x' quando da ocorrência do evento e.

Em jogos de computador, quando do uso de uma máquina de estados finitos, tipicamente associamos a cada estado um conjunto de ações a serem tomadas. Estas podem ser implementadas diretamente na linguagem do motor do jogo, ou na forma de uma linguagem de script. O uso de linguagens de script visa trazer maior comodidade ao processo de implementação da IA de um jogo. Com sua utilização, torna-se fácil a alteração do comportamento projetado (algumas vezes mesmo em tempo de execução), sendo uma linguagem mais amigável para se aprender e mais restrita do que uma linguagem de programação em termos de funcionalidades. Isso pode muitas vezes ser útil pois nem sempre um programador é o responsável por codificar estes comportamentos, que muitas vezes é efetuado por um projetista de jogo sem maior habilidade em programação.

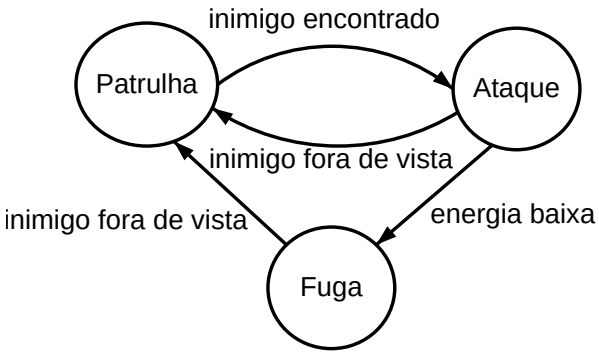

Figura 2.1: Exemplo de uma máquina de estados para um jogo fictício.

2.2 Estratégias de Busca Tradicionais

De forma genérica, as estratégias de busca tradicionais envolvem uma busca em uma árvore que descreve todos os estados possíveis a partir de um estado inicial dado. Formalmente, o espaço de busca é constituído por nós n, conectados através de arcos. A cada arco pode ou não estar associado um valor, que corresponde ao custo c de transição de um nó a outro. A cada nó temos associada uma profundidade p, sendo que a mesma tem valor 0 no nó raiz e aumenta de uma unidade para um nó filho. A aridade a de um nó é a quantidade de filhos que o mesmo possui, e a aridade de uma árvore é definida como a maior aridade de qualquer um de seus nós. O objetivo da busca é assim encontrar um caminho (ótimo ou não) do estado inicial até um estado final, explorando sucessivamente os nós conectados aos nós já explorados até a obtenção de uma solução para o problema.

As técnicas mais comuns, classificadas segundo [RN95] são:

- Busca por largura
- Busca de custo uniforme
- Busca por profundidade
- Busca bidirecional
- Busca heurística

- Busca por melhorias iterativas
- Outros

Cabe notar que as estratégias de busca tradicionais são amplamente utilizadas em jogos tradicionais (xadrez, gamão, go, etc.), especialmente porque nos mesmos o ambiente não é dinâmico. Nos jogos atuais, tais técnicas são mais utilizadas para a realização da navegação de unidades, sendo utilizadas também em alguns jogos no processo de planejamento.

A seguir comentamos resumidamente cada um dos métodos descritos acima.

2.2.1 Busca em largura

A busca em largura consiste em realizar a busca primeiramente explorando o nó raiz, e depois explorando todos os nós filhos do mesmo, e em seguida todos os filhos dos filhos, e assim sucessivamente. Assim, todos os nós de profundidade n são percorridos antes que os nós de profundidade $n+1$ sejam percorridos. O algoritmo garante encontrar assim a solução mais "rasa" (se existir), e mesmo a solução ótima caso o custo de um caminho seja monotonicamente crescente e somente dependa da profundidade do nó.

A busca em largura no entanto tem um grande problema: todos os nós explorados devem ser armazenados para futura expansão, o que pode ser extremamente custoso em termos de memória, especialmente para problemas de elevada aridade. Além disso, o custo em tempo de execução também é relativamente alto: caso tenhamos uma árvore de aridade a, e a profundidade da solução possua um valor d, o algoritmo possui complexidade de $O(a^d)$, ou seja, é exponencial com relação à profundidade da solução.

2.2.2 Busca de custo uniforme

Na busca de custo uniforme cada arco possui um custo dado por uma função $g(n)$, e sempre o nó de menor custo total acumulado até o momento é escolhido para ser explorado. Assim, a busca de custo uniforme é similiar à busca em largura, com profundidade igual ao último custo total calculado.

A busca de custo uniforme também garante que uma solução seja encontrada, e ainda

garante que esta solução será ótima, caso um caminho seja monotonicamente crescente. Entretanto, a busca de custo uniforme continua a apresentar os mesmos problemas em relação ao seu custo computacional que o algoritmo anterior.

2.2.3 Busca em profundidade

Na busca por profundidade, sempre o nó com maior profundidade é analisado. Esta estratégia garante que uma solução (se existir) será encontrada, mas não garante a solução ótima.

Em termos de requisitos de memória, essa estratégia de busca consome menos memória que a busca em largura, pois somente o caminho corrente precisa ser armazenado. Em termos de complexidade, a mesma é $O(a^d)$, aonde a é a aridade e d é a profundidade máxima das soluções. Na prática, para problemas com muitas soluções a busca em profundidade tende a ser mais rápida que a busca em largura, com a desvantagem que a busca pode ficar presa em ramos que não contenham a solução, o que pode se tornar ainda pior caso a árvore tenha profundidade infinita.

A fim de resolver o problema para árvores muito profundas, ou ainda árvores de profundidade infinita, pode-se limitar a profundidade da busca, de forma a permitir que toda a árvore até a profundidade escolhida seja explorada. Entretanto, um problema é escolher a profundidade adequada para se encontrar uma solução - problema este que pode ser aliviado permitindo que a profundidade limite aumente iterativamente. Entretanto surge então a necessidade de se definir de forma adequada o passo com o qual o limite de profundidade deve aumentar, valor este dependente dos recursos computacionais disponíveis e do problema específico sendo analisado.

2.2.4 Busca bidirecional

Na busca bidirecional a busca é realizada de forma concorrente a partir do estado inicial e do estado solução. Para cada uma das buscas sendo realizada uma diferente estratégia pode ser adotada. É interessante notar que devemos poder obter operadores que realizem tanto o caminho de ida (à partir do estado inicial) quanto de volta (partindo do(s) estado(s) final(is)).

Além disso, devemos poder verificar rapidamente se um estado já foi analisado, de forma a decidirmos quando uma solução foi encontrada. Uma desvantagem desta forma de busca é que a mesma requer que conheçamos de antemão os estados solução, informação que em alguns casos pode não estar disponível.

2.2.5 Busca heurística

Na busca heurística procura-se associar a cada arco um custo determinado por uma função heurística, que determina a distância do nó até uma solução. A estratégia mais simples baseada em heurística consiste em realizar uma busca egoísta, na qual sempre o nó que se julga mais próximo do objetivo, obtido através da função heurística $h(n)$, é o nó seguido. Entretanto, a mesma não obtém de forma garantida nem a solução ótima nem garante que uma solução seja efetivamente encontrada.

A fim de melhorar a busca egoísta descrita acima, resolveu-se aliá-la à busca por custo uniforme, que apesar de ser ineficiente, garante resultados ótimos. O algoritmo *A*[HNR68]* é o resultado dessa união, sendo eficiente, e garantindo otimalidade para funções $h(n)$ adequadas. Por função adequada entende-se que a mesma tem que ser uma *heurística admissível*, isto é, a mesma nunca pode sobre-estimar o valor de custo até a solução.

A estratégia $A*$ trabalha de forma similar à busca de custo uniforme, com a diferença que a função para determinação do próximo nó a ser explorado é dada por

$$f(n) = g(n) + h(n) \tag{2.1}$$

na qual $g(n)$ é o custo total do estado inicial ao estado corrente, e $h(n)$ é o custo estimado até um estado final escolhido. Vista de outra forma, a função $f(n)$ representa o custo estimado da solução de menor custo do estado inicial até a solução, passando pelo nó n.

Como comentado acima, o algoritmo A* garante a obtenção de uma solução ótima (para uma prova informal, vide [RN95], pgs 99-100). Além disso, o algoritmo é *ótimamente eficiente*, isto é, nenhum outro algoritmo que obtenha uma solução ótima garantidamente explora menos nós que o A*.

Entretanto o A* possui algumas limitações. Em geral, sua complexidade é exponencial, e o algoritmo pode consumir rapidamente grande volume de armazenamento uma vez

que precisa guardar todos os nós explorados. Para atenuar este problema foi criado o *IDA** [Kor85] que realiza o A* até um valor fixo de $f(n)$, valor este que aumenta iterativamente. Outro algoritmo ainda que procura corrigir alguns problemas presentes com o IDA* é o algoritmo *SMA** [Rus92]. Resumidamente, o mesmo realiza a busca de forma limitada pela memória disponível, armazenando os nós explorados até o limite da memória disponível, e eliminando-os da memória quando necessita de espaço para explorar novos nós. O IDA* pode ser demonstrado ótimo, completo e otimamente eficiente, dados certos requisitos de memória.

Cabe notar ainda que o A* e suas variantes constituem atualmente o algoritmo mais utilizado pelos jogos para resolver problemas de determinação de caminho em ambientes bidimensionais e tridimensionais.

2.2.6 Busca por melhorias iterativas

Nesta categoria enquadram-se algoritmos que realizam a busca não em uma árvore de estados até a solução, mas que procuram a melhor solução (a de maior valor) dentre um espaço de soluções n-dimensional.

Um dos algoritmos mais simples para este tipo de problema consiste em realizar a busca a partir de um estado inicial arbitrário, e escolher o próximo estado na direção que melhor otimize o estado atual, método este conhecido também como *hill-climbing*. Este método não é muito eficaz pois, apesar de sua simplicidade, tende a fornecer valores sub-ótimos quando da presença de máximos locais.

Outro método de busca, conhecido como *simulated annealing* [KJV83], escolhe o próximo estado não de forma determinística, mas aleatória. Simulated annealing realiza a escolha verificando um estado adjacente escolhido de forma aleatória; caso o estado melhore o estado atual, o estado analisado passa a ser o novo estado atual. Caso contrário, o estado analisado será o novo estado atual com uma probabilidade p, que diminui exponencialmente com a diferença de valor entre os estados atual e o analisado. p diminui também a cada iteração do algoritmo, de forma que para amplos intervalos de tempo, o algoritmo passa a se comportar aproximadamente como um *hill-climbing*.

2.2.7 Outros

Outro algoritmo utilizado comumente em jogos de computador, especialmente na determinção de caminhos em grafos, é o algoritmo de Floyd-Warshall [CLRS01]. Este algoritmo, ao contrário do algoritmo A* apresentado anteriormente, visa calcular o caminho mais curto entre todos os nós do grafo, operando sobre arcos direcionados (com pesos ou não). O mesmo recebe como entrada o conjunto dos nós do grafo e os pesos dos arcos, retornando duas matrizes, uma contendo a distância entre os nós e outra contendo o próximo nó a percorrer quando se deseja ir de um nó a a um nó b qualquer. Mais formalmente, dados:

- Um grafo $G = (V,A)$, com nós $V = \{v_0...v_n\}$ e arcos $A = \{(v_i, v_j)\}$, $A \subseteq V \times V$.
- Uma matriz $peso^{n \times n}$ contendo os pesos dos arcos de G, de tal forma que $peso(v_i, v_j)$ denota o peso do nó v_i até o nó v_j, e que $peso(v_i, v_j) = 1$ quando os nós não se encontram conectados. Note que $peso(v_i, v_j)$ não necessariamente é igual a $peso(v_j, v_i)$.

O algoritmo retorna duas matrizes:

- $custo(v_i, v_j)$ que contém o custo do menor caminho para se ir do nó v_i ao nó v_j.
- $pred(v_i, v_j)$ que contém o próximo nó pertencente ao menor caminho entre os nós v_i e v_j.

As matrizes acima são inicializadas da seguinte forma:

- $custo(v_i, v_j)_{i=1...n-1, j=0...n-1} = \infty$
- $pred(v_i, v_j)_{i=1...n-1, j=0...n-1} = \varnothing$

Definimos o operador:

- $adjacente(v_i, v_j) = verdade$ se e somente se existe um arco ligando o nó v_i ao nó v_j.

O algoritmo 1 apresenta então a resolução para o problema de determinação de caminhos mais curtos usando o algoritmo de Floyd-Warshall.

Algoritmo 1 Algoritmo de Floyd-Warshall.

```
para (v_i, v_j) ∈ A faça
    se adjacente(v_i, v_j) então
        custo(v_i, v_j) ← peso(v_i, v_j)
        pred(v_i, v_j) ← v_i
    senão
        custo(v_i, v_j) ←∞
        pred(v_i, v_j) ← ∅
para v_k ∈ V faça
    para (v_i, v_j) ∈ A faça
        se custo(v_i, v_k) < ∞ e custo(v_k, v_j) < ∞
            se custo(v_i, v_j) = ∞ ou custo(v_i, v_k) +
            custo(v_k, v_j) < custo(v_i, v_j) então
                custo(v_i, v_j) ← custo(v_i, v_k) + custo(v_k, v_j)
                pred(v_i, v_j) ← pred(v_k, v_j)
```

Cabe notar ainda que pode ser demonstrado que a complexidade deste algoritmo é de $O(|V|^3)$. Tal valor no entanto tipicamente não é relevante, pois o algoritmo é utilizado em grafos de pesos estáticos, sendo as matrizes de caminhos mínimos computadas durante o processo de inicialização do sistema. Dessa forma, determinar durante a execução o caminho mais curto entre dois pontos tem tempo constante.

Existem ainda outros algoritmos de busca e determinação de caminho em grafo, geralmente baseados nas estratégias descritas acima ou ainda em teoria de grafos. De particular interesse para os jogos de computador são os algoritmos que lidam com ambientes não totalmente conhecidos a priori, ambientes tridimensionais e ainda algoritmos que lidam com (re)planejamento para ambientes dinâmicos, o que é especialmente importante em alguns jogos de computador. Tais algoritmos não serão analisados em maior detalhe aqui, mas informações sobre o assunto podem ser encontradas em [Ste94, Jön97], ou ainda nos endereços `http://www.red3d.com/breese/navigation.html` e `http://theory.stanford.edu/~amitp/GameProgramming/MovingObstacles.html`.

2.3 Computação Evolutiva

A computação evolutiva tem como objetivo a resolução de problemas (ou a modelagem de processos evolutivos) de forma inspirada no processo biológico de propagação genética e seleção natural propostos por Charles Darwin. A computação evolutiva engloba as seguintes técnicas: algoritmos genéticos ([Hol75]), estratégias evolutivas, programação genética ([Koz92]), sistemas classificadores ([Hol75]), entre outros, sendo todos referenciados genericamente como algoritmos evolutivos.

De forma genérica, os algoritmos evolutivos se baseiam sobre uma população de "indivíduos", sendo que os indivíduos mais aptos a sobreviver em um ambiente dado reproduzem-se gerando novas gerações, e assim sucessivamente. Cada indivíduo tem suas características relevantes codificadas em um ou mais cromossomos e a pressão seletiva (responsável por determinar quais os indivíduos mais aptos a sobreviverem e reproduzirem) é modelada através de mecanismos de seleção (determinísticos ou não). Assim, os indivíduos competem entre si, sendo que o ambiente determina quais os mais aptos a sobreviverem e a se reproduzirem. A medida de aptidão de cada indivíduo pode ser determinada direta ou indiretamente - através de funções de custo, ou da capacidade de reprodução do indivíduo (que não deixa de ser também uma função de custo).

Um aspecto importante nos algoritmos genéticos é a manutenção da diversidade genotípica da população, uma vez que esta diversidade é de extrema importância no processo de exploração do espaço de busca. Por espaço de busca entendemos o espaço formado pelos parâmetros codificados nos cromossomos e a função de custo (que determina a adaptabilidade do indivíduo). A diversidade genotípica é em geral garantida pelo processo de formação de novos indivíduos, que se utiliza do genótipo de seu(s) "pai(s)" para, através de operadores genéticos não determinísticos (como *crossover* ou mutação) obter o genótipo do indivíduo "filho". Outra forma de garantir a diversidade genotípica é através dos mecanismos de seleção, que podem escolher indivíduos de forma não determinística.

Outro fator ainda de suma importância no processo evolutivo é a escolha de parâmetros para serem codificados no genótipo. Tais parâmetros influenciam diretamente na adaptabilidade do indivíduo, podendo assim limitar a capacidade da população como um todo ou ainda tornar o processo de evolução extremamente lento.

Vistos de outra forma, os algoritmos evolutivos realizam a busca em um **espaço genotípico**, sendo que a pressão seletiva incide sobre o **espaço fenotípico** (características externas), como pode ser visto na figura 2.2. Assim, a cada geração ocorre o processo de determinação das características dos indivíduos à partir do seu genótipo *g1*, num processo conhecido como **epigênese** (*f1*). A seleção (f2*)* ocorre assim sobre o espaço fenotípico, resultando na sobrevivência genotípica (*f3*) *de* alguns genótipos. Esses genótipos são então expostos aos operadores genéticos (*f4*) produzindo assim os genótipos *g2* da nova geração de indivíduos.

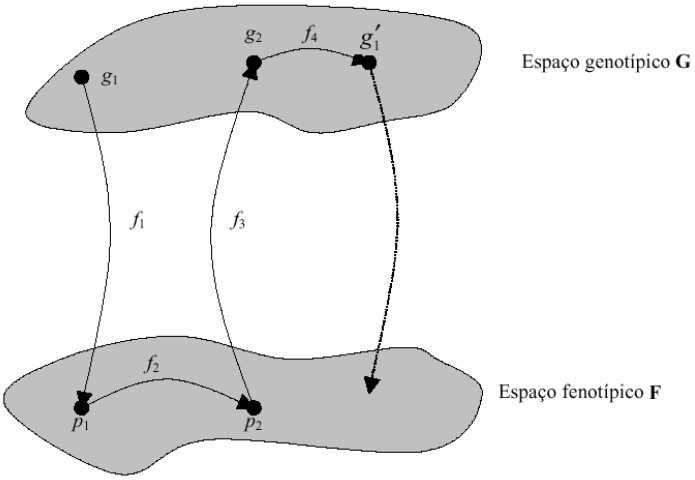

Figura 2.2: Processo de busca nos algoritmos evolutivos, visto à partir dos espaços fenotípicos e genotípicos (extraído de [Zub00]).

É interessante notar que apesar dos algoritmos evolutivos estarem baseados em processos estocásticos (o fator probabilístico está presente por exemplo na seleção de indivíduos, e nos operadores de crossover e mutação), os mesmos não podem ser caracterizados como uma mera busca aleatória pois na realidade existe um direcionamento (de certa forma uma heurística) na busca dado pelo processo de seleção.

Como última observação cabe ainda notar que os algoritmos evolutivos constituem um método *fraco* para a resolução de problemas, isto é, os mesmos se prestam a resolver problemas genéricos (não se limitando a domínios específicos) em ambientes genéricos (não necessariamente lineares, estacionários ou estáticos). Assim, devem ser utilizados preferencialmente quando da não existência de métodos *fortes* para resolução dos problemas em questão

(métodos específicos). Isso pois a utilização de métodos fracos, como algoritmos evolutivos, implica geralmente em um custo computacional muito maior.

2.4 Redes Neurais Artificiais

As redes neurais artificiais [Hay94] constituem outra técnica inspirada diretamente nos sistemas biológicos, neste caso os neurônios presentes nos seres vivos. O objetivo das redes neurais artificiais é o de construir mecanismos similares às redes neurais biológicas, possibilitando àquelas, entre outros, a tomada de decisões complexas, processamento de informações, otimização e aprendizagem. As principais áreas ligadas ao estudo de redes neurais artificiais são a estatística, os sistemas lineares e não lineares, a álgebra linear, o controle de sistemas, a otimização de funções e o processamento de sinais. Para o caso específico estudado aqui consideraremos apenas redes neurais artificiais implementadas via *software* em dispositivos computacionais de uso geral (ao contrário de redes neurais artificiais implementadas diretamente em *hardware* em dispositivos especialmente projetados).

As redes neurais artificiais são compostas por um conjunto de **neurônios artificiais** interconectados entre si, seguindo uma arquitetura (**topologia**) pré-determinada ou não. Cada neurônio possui terminais de entrada e saída, e tem seu comportamento de entrada / saída determinado por uma **função de ativação** (que pode ser a mesma para todos os neurônios da rede ou não, podendo mesmo alterar-se através de um processo de aprendizagem). A cada conexão de entrada com outro neurônio está associado um valor (**peso**) que determina a relevância da conexão entre ambos os neurônios. O resultado final obtido pelo processamento de uma rede neural depende assim da configuração assumida pelos neurônios (conexões e funções de ativações) combinado com o processo de aprendizagem.

De forma mais específica, a arquitetura da rede define previamente a organização da rede, isto é, como estarão estruturados os neurônios relativamente a suas conexões e sinais de entrada e saída. As funções de ativação determinam o comportamento individual interno a cada neurônio. Já o processo de aprendizagem determina como se dará o ajuste do valor das interconexões entre os neurônios, além de prever o conjunto de dados a ser utilizado para treinamento da rede (caso a rede passe por um período prévio de aprendizagem). O estado

inicial das interconexões neuronais pode também ser definido a priori pelo projetista da rede ou por algoritmos especializados, mas valores padrão são utilizados em grande parte dos casos, cabendo ao processo de aprendizagem o ajuste dos mesmos.

2.4.1 Aprendizagem

O processo de aprendizagem consiste na melhoria do desempenho (que pode ser quantificado de diversas formas, como por exemplo erro quadrático médio) da rede neural artificial ao longo do tempo. De forma mais específica, o processo consiste em um ajuste dos pesos das conexões entre os neurônios de forma a obter uma melhor resposta da rede como um todo ao problema em questão. Existem diversos algoritmos definidos para a realização do aprendizado em redes neurais, como máquinas de Boltzmann, lei de Hebb, algoritmo de retropropagação, entre outros.

Existem basicamente três paradigmas de aprendizagem em redes neurais: supervisionado, por reforço e não supervisionado. Cada um destes paradigmas se distingue através da resposta obtida do ambiente às respostas fornecidas pela rede neural artificial e como as respostas do ambiente influem no conjunto dos neurônios da rede.

Na **aprendizagem supervisionada**, são conhecidos conjuntos de estímulos - respostas, ou outras informações que forneçam o comportamento esperado da rede. Dessa forma a rede é ajustada de forma a obter resultados que se aproximem cada vez mais dos resultados esperados para o conjunto de padrões de treinamento.

Na **aprendizagem por reforço** o desempenho da rede é avaliado periodicamente, sendo atribuída à mesma um valor que quantifica seu desempenho em um certo período de tempo.

Na **aprendizagem não-supervisionada**, a rede é responsável por se auto-organizar em função dos estímulos do ambiente, aprendendo a diferenciar as diversas categorias de estímulos.

2.4.2 Arquiteturas

Existem diversas arquiteturas para as redes neurais artificiais, cada qual geralmente mais adaptada a determinadas categorias de problemas. As arquiteturas mais comuns são:

- Redes neurais em camadas
- Redes recorrentes
- Redes de Hopfield
- Redes de Kohonen
- Redes construtivas

As redes neurais em camadas constituem a arquitetura mais utilizada atualmente. As mesmas estruturam a rede em camadas, cada camada contendo um certo número de neurônios. Cada uma dessas camadas possui duas camadas adjacentes (com exceção das camadas de entrada e saída, responsáveis pela comunicação externa) e todos os neurônios de uma camada têm suas entradas conectadas às saídas da camada anterior e suas saídas conectadas às entradas dos neurônios da camada posterior. A forma de aprendizado mais comum para este tipo de rede é o aprendizado supervisionado, utilizando-se o algoritmo de retropropagação.

Redes recorrentes são redes que possuem realimentação, ou seja, a saída da rede (ou mesmo de qualquer um de seus neurônios) é utilizada como entrada para ela mesma. Tais redes prestam-se assim a modelar sistemas dinâmicos não-lineares, sendo em geral adequadas para tarefas como mapeamento de dados de entrada-saída, memória associativa, previsão de séries temporais, entre outros.

A **rede de Hopfield** é um caso específico de uma rede recorrente totalmente conectada, sendo geralmente utilizada para o armazenamento de padrões e a recuperação dos mesmos à partir de dados ruidosos ou incompletos. O algoritmo de aprendizado utilizado para armazenamento dos padrões é a regra de Hebb generalizada, que define os pesos da rede para armazenamento do padrão dado. A recuperação dos padrões é dada através do ajuste sucessivo das saídas dos neurônios da rede até ser atingida a estabilidade. É importante notar que a capacidade de armazenamento da rede de Hopfield é limitada tanto pelo número de neurônios da rede quanto pelas instabilidades decorrentes da não-linearidade da rede.

A **rede de Kohonen** é constituída por neurônios dispostos em vizinhanças geralmente uni-dimensionais ou bi-dimensionais, capazes de através do algoritmo de aprendizado, de se auto-organizar de forma a detectar padrões recorrentes. A rede de Kohonen realiza assim um

mapeamento entre um espaço n-dimensional (a dimensão do problema original) ao espaço de vizinhança da rede (uni ou bi-dimensional), obviamente com perda de informação. De forma simplificada, para cada ponto fornecido o algoritmo de aprendizagem determina o neurônio mais próximo ao mesmo, aproximando este neurônio e seus vizinhos ainda mais do ponto em questão. Após um certo número de iterações os neurônios tendem a agrupar-se em torno dos padrões fornecidos. A partir então das relações de vizinhança entre os neurônios podem ser determinados os padrões (*clusters*) recorrentes.

Por último, as **redes construtivas** supra-citadas não constituem propriamente uma arquitura *per se*, sendo geralmente redes em camadas diferenciadas. O que as torna distintas das redes em camadas é que as funções de ativação e a topologia das mesmas são determinadas através de um processo de aprendizagem, de forma a se adaptar ao problema e poder assim melhor representá-lo, o que implica em geral em economia de neurônios e recursos computacionais.

2.4.3 Funções de Ativação

As funções de ativação mais utilizadas são as funções do tipo sigmoidais, como a função logística ou tangente hiperbólica, principalmente por terem como vantagem em relação à função degrau serem deriváveis, facilitando sua utilização junto ao algoritmo de retropropagação. Outras funções utilizadas ainda incluem funções gaussianas e funções de base radial, ou ainda, no caso de redes construtivas, funções paramétricas ou *splines*.

2.5 Sistemas Nebulosos

Realizamos aqui uma breve introdução aos sistemas nebulosos e à lógica nebulosa [PG98]. Sistemas nebulosos são sistemas que em sua constituição lidam formalmente com conceitos vagos e/ou imprecisos na tomada de decisão, de forma similar a como seres humanos são capazes de tomar decisões levando em consideração somente conhecimentos vagos. A formalização dos sistemas nebulosos baseia-se na teoria matemática de conjuntos nebulosos (ou conjuntos *fuzzy*). De maneira análoga, a lógica nebulosa (*fuzzy*) surgiu a fim de complementar

a lógica clássica, que lidava somente com os conceitos de verdade e falsidade, incorporando a ela a incerteza. Na lógica nebulosa, um valor-verdade de uma proposição além de assumir os possíveis valores falso e verdadeiro (0 ou 1), pode assumir todo um conjunto de valores de incertezas, associados aos números reais compreendidos *entre* 0 e 1.

Um conjunto nebuloso, ao contrário de um conjunto convencional, é um conjunto em que a pertinência de seus elementos não é necessariamente absoluta. Assim, define-se um conjunto nebuloso, por meio de uma função de pertinência. Dado um conjunto *universo S,* definimos uma função de pertinência f como um mapeamento:

$$f : S \rightarrow [0,1] \tag{2.2}$$

A teoria de conjuntos nebulosos define ainda os operadores:

$$f(!x) = 1 - f(x) \tag{2.3}$$

$$f(x \wedge y) = \min(f(x), f(y)) \tag{2.4}$$

$$f(x \vee y) = \max(f(x), f(y)) \tag{2.5}$$

A popularidade da lógica nebulosa em aplicações advém do fato de que conjuntos nebulosos podem ser associados a valores para variáveis linguísticas. Por exemplo caso desejemos comprar um produto p, dado seu preço x podemos ter uma função de pertinência que expressa quão "caro" é o produto x. Um exemplo de função pode ser visto na Figura 2.3.

A lógica nebulosa permite assim expressar conceitos subjetivos, tais como "muito baixo", "baixo", "médio", "muito alto", etc., aos quais podemos associar conjuntos nebulosos, facilitando dessa forma a incorporação do conhecimento de especialistas na área ao projeto do sistema.

A lógica nebulosa está presente atualmente em diversas áreas de aplicação, sendo as principais delas o controle, a automação de processos, a recuperação de informações e os sistemas de apoio à decisão, dentre outros. As formas de utilização da lógica nebulosa também variam: existem sistemas de controle nebulosos, bases de inferência nebulosas, sistemas neuro-fuzzy, máquinas de estados finitos nebulosas, redes de petri nebulosas, etc.

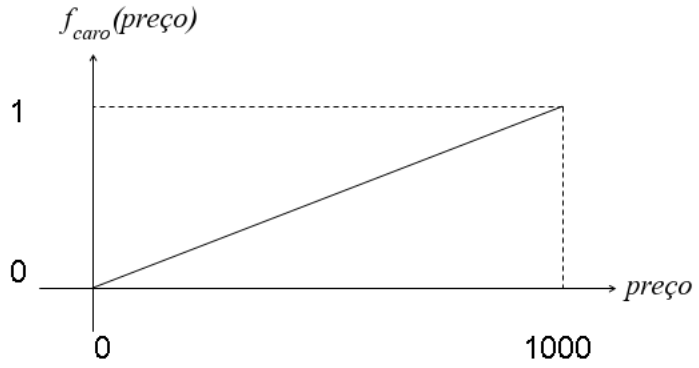

Figura 2.3: Exemplo de função de pertinência.

Das técnicas mencionadas acima, de especial interesse aos jogos de computador são as máquinas de estados finitos nebulosas (FFSM, *Fuzzy Finite-State Machine*), por serem uma evolução natural do método mais utilizado atualmente nos jogos de computador, as máquinas de estados finitos (Seção 2.1). Existem diversas implementações possíveis para as FFSM, mas geralmente elas são caracterizadas por atribuir valores nebulosos de pertinência a um determinado estado. Isso tem diversas implicações, a principal delas o fato de se ter de definir o comportamento dos valores nebulosos quando do disparo de um evento. Tipicamente um evento também é representado por um valor nebuloso, tornando assim intuitivo o uso de regras que combinem este valor com o valor do estado atual a fim de se obter o valor de pertinência do próximo estado.

Mais especificamente, se temos em S o conjunto de estados possíveis:

$$S = S_1 \dots S_i \tag{2.6}$$

Definimos como:

$$\sigma(S_i) \rightarrow [0,1] \tag{2.7}$$

o valor de pertinência do sistema em relação ao estado S_i .

Também definimos o valor de pertinência de um evento e qualquer como:

$$\rho(e) \rightarrow [0,1] \tag{2.8}$$

Assim sendo, quando da transição de um estado i para um estado j em resposta a um evento e, podemos ter por exemplo as seguintes regras:

$$\sigma(S_j) = \sigma(S_i) \wedge \rho(e) \tag{2.9}$$
$$\sigma(S_j) = \sigma(S_i) \tag{2.10}$$
$$\sigma(S_j) = \rho(e) \tag{2.11}$$
$$\sigma(S_j) = 1 \tag{2.12}$$

Notemos que o uso de regras como a primeira apresentada implicam no degradamento do sistema, resultando em valores cada vez menores de pertinência a cada próximo estado.

Tipicamente a regra a ser utilizada está associada estruturalmente à FFSM utilizada, cabendo ao projetista da mesma determinar qual regra será a mais adequada para cada evento em cada estado possível.

Podemos também utilizar as FFSMs de modo que um sistema não se encontre somente em um determinado estado, mas possa estar em vários estados com graus de pertinência distintos a cada um dos mesmos. Isso entretanto pode gerar diversas complicações, como por exemplo na determinação das ações a serem tomadas, e na compreensão do funcionamento do sistema como um todo.

2.6 Resumo

Nesse capítulo analisamos brevemente as principais técnicas de inteligência artificial utilizadas em jogos, dando especial ênfase às técnicas de sistemas inteligentes. Tal análise será muito importante tanto na compreensão do Capítulo 3, que se utiliza dos conceitos aqui apresentados para analisar o mercado atual dos jogos de computador sob a ótica dos sistemas inteligentes, quanto para melhor situar o modelo computacional apresentado neste trabalho no Capítulo 5.

Capítulo 3

Jogos de Computador: Panorama Tecnológico

Este capítulo realiza uma análise de produtos comerciais e iniciativas acadêmicas que utilizam técnicas de sistemas inteligentes, sendo analisados não somente jogos de computador, mas também aplicações de entretenimento eletrônico e realidade virtual. Procedemos inicialmente com uma análise dos produtos comerciais, passando posteriormente à análise das iniciativas de cunho acadêmico.

3.1 Produtos Comerciais

A fim de procedermos estruturadamente com a análise dos produtos comerciais decidimos dividir os mesmos em categorias amplas de acordo com seu gênero, sendo que para cada categoria analisamos as técnicas de sistemas inteligentes mais comumente utilizadas. No caso de produtos particularmente interessantes, apresentamos os mesmos individualmente. Cabe notar que apenas as categorias de jogos com exemplos representativos de utilização de técnicas de sistemas inteligentes são analisadas.

É interessante observar que a indústria de jogos tradicionalmente guarda com extremo zelo as soluções computacionais desenvolvidas, e isso se estende desde os aspectos gráficos, som, rede, até soluções de IA. Portanto, apesar de quase todos os jogos utilizarem técnicas de

sistemas inteligentes, e mesmo de alguns utilizarem técnicas mais recentes de SIs, somente uma pequena parcela possui uma descrição sobre os algoritmos e soluções utilizadas.

Assim sendo, as categorias a serem analisadas são:

- Ação primeira/terceira pessoa tridimensionais
- Estratégia
- *Role Playing Games* (*RPG*s) / Aventura
- Outros
  - *Galapagos*
  - Linha *Sim*
  - *Creatures*
  - *Black & White*

Tais categorias são as mais comumente utilizadas informalmente pela indústria de jogos de computador em geral.

Não estaremos analisando as aplicações de sistemas inteligentes em jogos tradicionais de tabuleiro (por exemplo, xadrez, go-moku, damas), uma vez que tais trabalhos fogem ao escopo deste estudo.

Hiper-referências para os jogos aqui mencionados podem ser encontradas no Apêndice A.

3.1.1 Ação Primeira/Terceira Pessoa Tridimensionais

Os jogos de ação em primeira (FPS, de *first-person shooter*) ou em terceira pessoa em ambientes tridimensionais são os que têm apresentado maior crescimento em popularidade nos últimos anos. Isso se deve principalmente aos avanços da capacidade de processamento gráfico dos computadores pessoais aliados ao surgimento de jogos muito bem sucedidos no mercado, como a linha *Doom* e *Quake*, bem como outros tais quais *Half-Life* (e modificações), *Tribes* e *Unreal*.

Devido à grande popularidade desses jogos, formou-se uma comunidade de desenvolvedores de *bots*, que nada mais são que agentes que personificam os oponentes (ou mais raramente companheiros de equipe) do jogador. Os *bots* têm recebido cada vez mais importância nos jogos de terceira pessoa, pois a quase totalidade dos mesmos atualmente possui grande ênfase nos jogos *multiplayer* (para múltiplos jogadores) via rede (local ou Internet). Em tal ambiente se faz necessário integrar agentes cujo comportamento se aproxime ao máximo de um jogador humano, de forma a suprir a eventual falta dos mesmos ou ainda a fim de realizar tarefas pouco atrativas para um jogador humano.

Cabe notar que o comportamento dos bots em tais jogos é geralmente controlado através de máquinas de estado, tendo principalmente como preocupação a navegação no ambiente tridimensional e a utilização de técnicas básicas de combate. Entretanto, podemos notar uma tendência voltada para o desenvolvimento de bots dotados de comportamentos mais sofisticados na medida em que alguns jogos passam a dar uma ênfase maior aos aspectos estratégico e de coordenação.

Linha Doom / Quake

Esta foi a sequência de jogos que trouxe ao grande público a categoria de jogo de ação em terceira pessoa, tendo sido introduzida inicialmente com *Doom*, e tendo como sucessores *Doom 2*, *Quake*, *Quake 2*, *Quake 3*, além de numerosos outros jogos licenciados para uso do mesmo motor de jogo (como *Heretic* e *Hexen*).

Grande parte dos bots existentes atualmente são implementados para estes jogos, em especial para a linha *Quake*, uma vez que foi fornecido pelo desenvolvedor um conjunto de *API*s (*Application Programmer Interface*) que facilitam o desenvolvimento dos mesmos. A ênfase desses bots, no entanto, como já mencionado acima, é principalmente nos aspectos de navegação e comportamento individual ofensivo / defensivo.

Dentro das iniciativas de construção de bots podemos destacar o *NeuralBot* (`http://www.cyd.liu.se/~bjoli035/botepidemic/neuralbot/`) um bot experimental para *Quake 2* controlado por uma rede neural cujos pesos são evoluídos através de um algoritmo genético. A rede neural é composta por 2 - 5 camadas de até 60 neurônios cada. A entrada para a rede neural é composta pelos sensores do bot (aproximadamente 60, que informam sobre a presença de obstáculos, inimigos e ações dos mesmos, e status atual do bot). As saídas são as

ações permitidas - no total são aproximadamente 30, que instruem o bot a se locomover, a selecionar ou disparar armas, ou ainda a alterar seu foco de visão. O algoritmo genético é executado a cada 60 segundos e seleciona dois bots como pais. Esses bots são selecionados de forma proporcional ao seu *fitness*, sendo que o mesmo é calculado tendo como base o número de inimigos mortos desde a última iteração. Os pesos das conexões para cada rede neural são codificados em cromossomos ocorrendo então o crossover e mutação entre ambos, produzindo dois descendentes que substituirão os dois piores bots da população atual. No lugar da evolução dos pesos, também foi tentada a utilização de um algoritmo de retropropagação que, segundo o autor, não chegou a alcançar resultados satisfatórios devido à complexidade da rede neural.

O Neuralbot acima consegue navegar de forma satisfatória, chegando a se locomover para recuperar armas. Entretanto, os mesmos tendem a ficar presos em padrões fixos de movimento, como por exemplo *circle-strafing* (movimento lateral em padrão circular), o que levou o autor a desenvolver bots de referência responsáveis por atenuar a ocorrência desses padrões. Entretanto os bots obtidos não chegam a competir com um jogador humano, ou ainda com outros bots tradicionais.

Outro exemplo interessante de bot para Quake vem do projeto SOAR/Games, com o *SOAR QuakeBot* (`http://ai.eecs.umich.edu/~soarbot/`) ([LvL99, Lai00b, Lai00a]) desenvolvido na Universidade de Michigan. O projeto realizou a interface entre o sistema SOAR e o jogo Quake 2 (e mais tarde com outros jogos, como *Descent 3, Quake* e *Half-Life*), permitindo o controle de personagens do jogo pelo sistema SOAR. SOAR é um sistema especialista muito maduro e utilizado com sucesso em diversas aplicações, sendo utilizado pelo exército norte-americano. Existem extensões ao sistema SOAR de forma a permitir aprendizado de novas regras (utilizado um mecanismo denominado *chunking*), mas o mesmo não é utilizado pelo projeto SOAR Quakebot.

O SOAR Quakebot é composto por uma base de regras dispostas de forma hierárquica, como podemos visualizar na Figura 3.1.

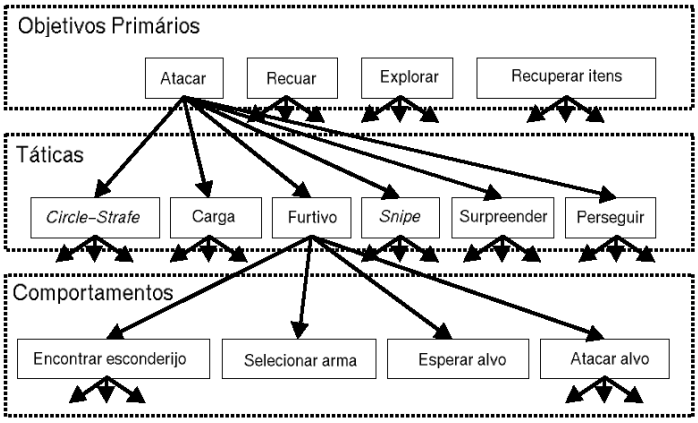

Figura 3.1: Uma parte da hierarquia de regras no sistema SOAR para um jogo de ação como Quake 2 (adaptado de [LvL99]). *Circle-strafing* se refere à tática de circular o inimigo andando lateralmente, e *Snipe* se refere ao comportamento de ataque no qual o bot usa uma arma de longo alcance para atingir inimigos a distância.

Outros

Outros jogos de ação em primeira/terceira pessoa tridimensionais em geral tendem a seguir o padrão estabelecido pela linha Quake, com alguns eventuais refinamentos. Um exemplo é a aplicação de algoritmos de *flocking* [Rey87] para coordenação dos inimigos, técnica esta presente nos jogos Unreal e Half-Life. Alguns jogos e *mod*s (modificações) vêm adotando uma linha de *multiplayer* cooperativo, no qual os jogadores devem atuar como um time caso queiram obter melhores resultados (ao contrário da linha Quake tradicional, que tem ênfase somente no aspecto de ação individual). Alguns exemplos desses jogos são *Action Quake 2*, *Half-Life Counter Strike* e *Team Fortress*, *Tribes 2*, ou ainda a série *Rainbow Six*. Tais jogos apresentam sob o ponto de vista da IA novos desafios, abordando temas como comunição inter-agentes e outras técnicas de inteligência artificial distribuída. Entretanto, apesar de já existirem alguns bots capazes de trabalhar em times respondendo a ordens simples, não existem bots capazes de tomar decisões estratégicas por si sós, mantendo-se coordenados a fim de atingir objetivos mais complexos.

3.1.2 Estratégia

Esta categoria engloba de forma ampla todos os jogos de estratégia, sejam os mesmos do tipo tempo real (*RTS*, do inglês *Real Time Strategy*) ou baseados em turno. Ainda de forma ortogonal a esta subdivisão temos crescentes níveis de realismo, variando desde jogos que simulam situações completamente fictícias (vide por exemplo *Dune 2*, *Starcraft*, *Warcraft*) até jogos que procuram simular da forma mais fidedigna possível situações de combate reais (como por exemplo *Panzer General*). Recentemente observa-se uma tendência acentuada desses jogos de passarem de uma visualização em duas dimensões para uma visualização tridimensional (por exemplo *Warcraft 3* e *Emperor: Battle for Dune*). Entretanto tal mudança raramente implica em mudanças significativas no modo de jogo do jogador ou nos sistemas de inteligência artificial utilizados (algumas exceções são *Homeworld* e *Ground Control*).

A inteligência utilizada em tais jogos é dividida basicamente em dois níveis: em um nível temos problemas como a navegação de unidades, no qual geralmente é usado um algoritmo como o A* (vide Capítulo 2), ou alguma variante do mesmo que leve em consideração replanejamento em tempo real. Em um segundo nível temos o planejamento estratégico/tático, que determina o caminho na árvore tecnológica a se seguir (se o jogo possuir uma árvore), o ritmo de produção de unidades e instalações, e a freqüência e a localização dos ataques ao jogador humano. Para isso, os métodos utilizados em geral são simplesmente máquinas de estado aliadas a sistemas baseados em regras parametrizáveis de forma a permitir o ajuste da dificuldade e jogabilidade. Em casos excepcionais, são usados sistemas de produção nebulosos, como é o caso dos jogos *Close Combat 2* e *Civilization: Call to Power*.

Dois dos jogos mais interessantes nesta categoria entretanto nunca atingiram grande reconhecimento público: *Cloak, Dagger and DNA* e *Fields of Battle*. O primeiro é um jogo de guerra similar aos jogos de tabuleiro, com a diferença de que a estratégia utilizada pelo sistema evolui através do uso de algoritmos genéticos [Hol75]. A estratégia utilizada pela máquina é codificada em um cromossomo, e a cada batalha o desempenho da mesma é avaliado, permitindo que a estratégia evolua aplicando os operadores padrões para algoritmos genéticos. Já o segundo também é um jogo de guerra, mas utiliza redes de Hopfield estocásticas para determinar a estratégia de jogo e *simulated annealling* para o movimento individual de unidades. Maiores detalhes sobre a aplicação de tais técnicas no entanto não são forneci-

dos pelos fabricantes.

Outro jogo digno de nota é *Enemy Nations*, que utiliza uma inteligência artificial "distribuída" para coordenar os diversos aspectos do jogo. Algumas das técnicas utilizadas são máquinas de estado nebulosas, mapas de influência [Zob69] e bases de dados contendo objetivos e tarefas a cumprir.

3.1.3 RPG's e Aventura

Nesta categoria se enquadram tanto os jogos de RPG quanto os jogos de aventura, não importando a interface utilizada (texto, bidimensional, tridimensional). Tais jogos possuem em comum o fato de permitirem ao jogador controlar um personagem, sua personificação em um mundo virtual, dando ênfase aos aspectos de desenvolvimento de características do personagem, resolução de quebra-cabeças e interação inter-pessoal. Como em outras categorias, nesta também existe uma tendência acentuada dos jogos de cunho comercial a passarem para uma visualização tridimensional.

Os chamados *MUD*'s (*Multi-User Dungeon*) e variantes (tais quais *MOO - MUD Object Oriented*, e *MUSH - Multi-User Shared Hallucination*) foram os jogos pioneiros a criarem ambientes virtuais para várias pessoas. São constituídos essencialmente por um servidor ao qual diversos jogadores podem se conectar, permitindo o acesso a um mundo virtual compartilhado a partir de uma interface texto. Por se tratar de ambientes relativamente simples, contendo poucas variáveis de ambiente e relativamente poucas ações, MUDs foram utilizados por diversos pesquisadores das mais diversas áreas para estudar assuntos como ensino à distância, antropologia, lingüística e sociologia [DHR97, DF98, CN94, Tur95, Rei94] ou mesmo competições para o prêmio Loebner[1] [Mau94]. Entretanto, no que concerne a aplicações de técnicas mais modernas de SIs podemos encontrar poucas aplicações, talvez justamente por causa do ambiente restrito proporcionado pelos mesmos.

Outros jogos cujas implicações ultrapassam a barreira do simples entretenimento (levantando questões pertinentes à realidade virtual e sociologia), são os jogos *MMORPG* (*Massive*

[1] O prêmio Loebner é uma competição entre programas de computador, tendo como métrica o teste de Turing. De forma simplificada, o teste de Turing determina que uma máquina é inteligente se um ser humano, através do diálogo com a máquina usando um terminal, concluir estar conversando com um outro ser humano. Para maiores informações ver `http://www.loebner.net/Prizef/loebner-prize.html`.

Multiplayer Online RPG - RPGs com mundos virtuais persistentes para jogo somente via Internet com centenas ou milhares de jogadores), sendo os exemplos mais significativos do gênero os jogos *Ultima Online*, *EverQuest* e *Asheron's Call*. Do ponto de vista dos SIs, os mesmos apresentam uma grande oportunidade ainda inexplorada, sendo os principais desafios o reconhecimento de linguagem natural e o controle de *NPC*'s (*Non-player Characteres*, personagens não controlados por jogadores humanos).

3.1.4 Outros

Galapagos

Galapagos é um jogo no qual o jogador é responsável por ensinar uma criatura chamada Mendel (similar a uma aranha com quatro patas) a se comportar de forma "inteligente", a fim de resolver diversos quebra-cabeças e avançar no mundo tridimensional de Galapagos. Os quebra-cabeças são em geral do tipo navegacional, ou seja, Mendel deve se locomover de forma sincronizada a diversos elementos do mundo para resolver o problema corrente. Mendel é capaz de aprender recebendo estímulos por parte do jogador, que indicam a Mendel se ele está se comportando de forma adequada ou não. Os sensores de Mendel são basicamente dois: um sensor "infravermelho", que lhe informa a distância dos objetos ao seu redor, e um sensor táctil.

A tecnologia utilizada em Mendel é denominada NERM *(Non-stationary Entropic Reduction Mapping*), uma tecnologia aparentemente similar a um controlador utilizando redes neurais artificiais. Infelizmente NERM é uma tecnologia proprietária e maiores informações não são fornecidas.

Linha Sim

Esta linha foi popularizada pela empresa *Maxis,* que trouxe ao grande público o conceito de jogos de computador como um laboratório virtual, contendo "seres" inseridos em mundos virtuais que podem ser manipulados pelo jogador. Nestes jogos (que inclui títulos como *El-Fish, SimEarth, SimLife, SimAnt, SimCity, The Sims, SimsVille*, entre outros) o jogador vê-se livre para alterar diversas características do jogo (como construções, terreno, etc.) e/ou controlar o comportamento dos "seres", observando o impacto de suas ações no mundo e

em seus habitantes. Esta linha de jogos surgiu originalmente em 1985 com o jogo *Little Computer People* (Figura 3.2) lançado para a plataforma Commodore 64, jogo que permitia ao usuário visualizar o interior de uma casa e seu único habitante, podendo comunicar-se com o mesmo usando frases simples ou recebendo "cartas" do mesmo.

Figura 3.2: Telas do jogo Litte Computer People.

Muitos desses jogos apresentam características interessantes sob o contexto de vida artificial, principalmente por serem justamente laboratórios nos quais formas de vida artificiais competem entre si por recursos. Especialmente interessantes são os jogos El-Fish e SimLife, por modelarem determinadas características das criaturas através de um código genético, permitindo a criação de criaturas novas e a passagem de genes através de gerações. No caso específico de El-Fish, um simulador de peixes em um aquário, foram utilizados 56 genes responsáveis por determinar as cerca de 800 características de um peixe (para maiores informações, vide `http://www.wired.com/wired/archive/1.02/maxis.html`).

De interesse também são os jogos da série SimCity, SimEarth e SimLife, que se utilizam de algoritmos baseados em autômatos celulares [Wol82] para atualizar o estado do jogo a cada instante de tempo.

Outro jogo da série que apresenta características interessantes é o jogo The Sims, no qual o jogador é responsável por gerenciar a vida de uma (ou mais) pessoas, seja controlando-a diretamente ou através do ambiente em que ela vive (como moradia, alimentos, utensílios domésticos e eletrônicos disponíveis). O jogo utiliza internamente uma máquina de estados nebulosa (capítulo 2) a fim de controlar os personagens do jogo. Cada personagem possui diversas variáveis internas que influenciam em suas escolhas, variáveis estas que representam diferentes necessidades do personagem, como fome, higiene, socialização, cansaço, entre outros (informações obitdas de [Woo]).

Creatures

Creatures [Gra97, GC98, GC99, GCM97] foi um jogo desenvolvido pela companhia Cyberlife, de sede na Inglaterra, cujo original rendeu duas continuações, Creatures 2 e Creatures 3. Neste jogo o jogador é responsável por cuidar do desenvolvimento, desde o nascimento até a morte, de pequenas criaturas chamadas *norns*. Os norns são capazes de aprender ações e associá-las a sentenças simples, demonstrando emoções básicas (por exemplo medo, alegria, tédio, fome), interagindo com outros norns e reproduzindo-se entre si, passando por estágios de desenvolvimento similares aos humanos, tais como infância, adolescência, vida adulta e velhice. O mundo no qual habitam os norns é um mundo de 2 dimensões, composto por vários níveis. O usuário interage com os norns por meio de uma pequena mão capaz de mover objetos, atrair a atenção dos norns e recompensá-los (ou repreendê-los) por suas ações. Podemos ver a tela do jogo na Figura 3.3.

Figura 3.3: Figura mostrando o jogo Creatures. Na tela podem ser vistos dois norns em dois níveis (um deles falando) e a mão controlada pelo usuário.

Entretanto, a simplicidade superficial do jogo esconde sua complexidade interior. Cada norn tem sua inteligência modelada por uma rede neural cuja implementação é diferente da tradicional. A rede neural utilizada, denominada CTRNN (de *Continuous Time Recurrent Neural-Network*), possui a capacidade de criar novas conexões, alterar os pesos para os relacionamentos e modificar as funções de definição de estado, sendo capaz de aprendizado supervisionado ou não-supervisionado. Cada neurônio ainda é alterado para em cada ati-

vação relaxar gradualmente seu estado, provendo assim um comportamento mais sofisticado capaz de refletir também a freqüência dos estímulos. Esta mesma técnica é utilizada para a aprendizagem por reforço, dando a um episódio um reforço inicial alto cujo valor diminui com o passar do tempo. Isto possibilita comportamentos complexos, como por exemplo (vide [Gra97]) que um norn que colocou a mão em um buraco e tenha recebido uma mordida de caranguejo não repita esta ação imediatamente, mas que eventualmente este episódio esteja mais enfraquecido em sua memória e ele possa executá-lo novamente, talvez dessa vez com uma resposta positiva.

Os neurônios são ainda organizados em lobos (aproximadamente camadas), sendo os dois mais importantes os de conceito e de decisão. Juntos estes são responsáveis pela generalização de episódios de acordo com os estímulos recebidos. Estes estímulos são determinados de acordo com alterações em um conjunto de *impulsos* (como por exemplo tédio, fome, cansaço).

Os impulsos de cada norn são modelados à partir de "elementos químicos" secretados por glândulas e detectados por receptores. Estes elementos químicos podem ainda combinar-se em simples reações químicas. Isto possibilita que quaisquer parâmetros sejam dependentes de receptores químicos, o que torna fácil a geração de recompensas / punições. O modelo químico ainda permite por exemplo modelar de forma intuitiva a respiração, através do uso de um elemento energético (glicose, por exemplo), proveniente da digestão, cuja concentração no sangue pode ser diminuída através da secreção de um elemento que com ele reaja, elemento este proveniente da realização de atividades físicas. Outras aplicações para o modelo químico ainda incluem a geração de hormônios e funcionamento do sistema imunológico.

É importante ainda observar que vários dos parâmetros de um norn são codificados através de genes que são transmitidos de geração em geração, sendo o código genético o resultado da combinação entre os genes parentais acrescidos de uma dose de mutação.

Fin Fin

Fin Fin é o resultado do trabalho da Fujitsu Interactive (relacionado com pesquisas no projeto Oz (`http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/oz/web/oz.html`) no desenvolvimento de um mascote virtual. Apesar da empresa Fujitsu Interactive ter fechado desde o lançamento do produto, o mesmo continua ainda a existir como TEO World (`http:`

`//www.teo-world.org/`), e é representativo de toda uma gama de mascotes virtuais, como o mais conhecido Petz (`http://petz.com/`).

O jogo consiste em interagir com uma pequena criatura meio golfinho / meio pássaro chamada Fin Fin. A interação se dá através de uma câmera de vídeo e um microfone, além do teclado (utilizado para recompensar Fin Fin). O programa entretanto não reconhece usuários nem realiza processamento de linguagem natural, baseando-se nos mecanismos acima somente para o reconhecimento da presença do usuário e determinação da altura e força da voz.

À medida que Fin Fin se acostuma com o usuário através de constante interação, ele passa a sentir-se mais à vontade e a fazer acrobacias, cantar, e realizar outros pequenos truques. Graças aos mecanismos de interação, Fin Fin sabe quando está sendo negligenciado e quando está recebendo atenção, e de que forma.

Internamente, Fin Fin é modelado através de um conjunto de estados emocionais, tais como medo, felicidade, fome, etc. A abordagem para sua arquitetura é feita por meio de camadas, que são duas: a primeira, reativa, provê o comportamento mais instintivo, e a segunda, reflexiva, providencia para um comportamento mais inteligente em intervalos de tempos maiores. Essas camadas são modeladas como sistemas especialistas cujas respostas aos estímulos estão programadas previamente para cada situação.

Fin Fin é um trabalho cujo interesse se enquadra mais especificamente na área de *believable agents*, mas cujos resultados, embora interessantes, ainda deixam por desejar.

Black & White

O jogo *Black & White* é um dos jogos mais bem sucedidos comercialmente que utiliza técnicas sofisticadas de SIs. O jogo, como outros criados por *Peter Molyneux*, consiste em moldar um mundo e seus habitantes aos desejos do jogador, que atua como um deus da antigüidade grega (daí o nome desta categoria de jogos, intitulados *God Games*). De forma mais específica, o jogador deve conseguir que os aldeões que povoam o mundo o adorem o suficiente de forma a conquistar novas tribos ou a realizar tarefas específicas. Para tal, o jogador é dotado de poderes capazes de alterar o relevo do mundo e de criar objetos, dentre

outros, além de ter um avatar[2] como seu representante.

Figura 3.4: Tela do jogo Black & White, mostrando os aldeões e o avatar (neste caso, um tigre). Figura obtida do site *Gamesdomain* (*http://www.gamesdomain.com/*).

A inteligência para as tribos de aldeões é constituída de dois níveis: existe uma inteligência global para a tribo, que decide quais tarefas cada aldeão deve executar. O segundo nível é constituído pelos aldeões, sendo que cada um possui necessidades individuais, como dormir, comer, ou reunir-se com outros aldeões.

No entanto, é no avatar que reside a maior complexidade da inteligência do jogo. O mesmo deve ser ensinado pelo jogador a fim de realizar tarefas, moldando-se de acordo com os ensinamentos do mesmo. Segundo o desenvolvedor da inteligência artificial do jogo, *Robert Evans*, disse em entrevista à revista eletrônica *Feed* (`http://www.feedmag.com/templates/default.php3?a_id=1694`), existem diferentes tipos de aprendizagem no jogo:

- Aprendizagem de fatos (por exemplo que existe uma cidade inimiga atrás do morro).
- Aprendizagem de como realizar ações (por exemplo, como pescar, como lançar magias).

[2]O termo avatar aqui é utilizado como a encarnação de uma "divindade" (o jogador) no mundo, e não como um avatar é conhecido em realidade virtual (uma representação do jogador ele mesmo no mundo).

- Aprendizagem de quais desejos dar maior prioridade (tornando a criatura egoísta, altruísta, brincalhona).
- Aprendizagem de quais fatores são mais importantes para cada desejo. Por exemplo, para o desejo de comer, vários fatores podem contar, como baixa energia, depressão, visualização de um alimento saboroso, etc.
- Aprendizagem de quais ítens são mais adequados para satisfazer determinados desejos, por exemplo, quais os melhores objetos para se comer.

O avatar em si é constituído por uma rede neural e por uma árvore de decisão, possuindo além disso vários parâmetros responsáveis por modelar as necessidades e emoções do avatar. A aprendizagem é assim realizada através de retropropagação (para a rede neural) ou de aprendizagem simbólica (para a árvore de decisão). A arquitetura interna é baseada na arquitetura *BDI* (*Belief-Desire-Intention* [RG92]), mas altamente modificada para acomodar as necessidades específicas do jogo.

3.2 Iniciativas Acadêmicas

Nesta seção procedemos com a análise dos trabalhos presentes na área acadêmica voltados à utilização de técnicas de sistemas inteligentes em aplicações de entretenimento eletrônico e ambientes virtuais.

É interessante observar que na área de inteligência artificial, ao contrário do que se pode observar na área de computação gráfica, não existe uma ativa troca de dados e experiências entre a comunidade científica e a indústria. Entretanto diversos esforços têm sido feitos no sentido de diminuir esta distância, destacando-se dentre os mesmos os simpósios de primavera promovidos pela *American Association for Artificial Intelligence* em 1999 (`http://www.cs.nwu.edu/~Ewolff/aicg99/index.html`), 2000 (`http://ai.eecs.umich.edu/people/vanlent/AAAI-SSS-2000.htm`), 2001 (`http://www.aaai.org/Symposia/Spring/2001/sss01-2.html`), além do trabalho sendo desenvolvido pelos projetos Excalibur (`http://www.ai-center.com/projects/excalibur/`), *SOAR / Games* da Universidade de Michigan, por *John Funge* (`http://www.dgp.toronto.edu/~funge/`) da Universidade de Toronto,

e pelo grupo *DEMO* (`http://www.demo.cs.brandeis.edu/`) da Universidade Brandeis liderado por *Jordan Pollack*. A importância dos jogos na pesquisa em sistemas inteligentes pode ainda ser evidenciada na palestra concedida por *Marvin Minsky* durante a *Game Developers' Conference 2001* (`http://www.gdconf.com/`).

Os trabalhos analisados, escolhidos por terem maior relevância com o trabalho desenvolvido nesta tese, são:

- Blumberg
- Funge
- Reynolds
- Tu e Terzopoulos

3.2.1 Blumberg

Blumberg em sua dissertação de doutorado [Blu96] propõe um sistema para controle de criaturas artificiais inspirado na etologia e na animação clássica. Tal sistema foi implementado em diversos sistemas computacionais, como ALIVE [MDBP96] e Hamsterdam [Blu94], tanto na forma de pequenos hamsters como de cães virtuais que podiam ser "adestrados" (vide Figura 3.5). A etologia vem a contribuir em seu sistema com modelos de comportamentos que visam explicar: - a tomada de decisões por parte de animais; - a organização de suas diferentes necessidades e objetivos; - o aprendizado e adaptação dadas as mais diferentes situações e estímulos. Já da animação clássica Blumberg busca meios de eficientemente transmitir estados emocionais e de dar assim às suas criaturas digitais maior aparência de "vida" e maior apelo visual.

O modelo computacional utilizado por Blumberg baseia-se numa arquitetura dividida em três partes: o sistema de decisão, o sistema motor e o sistema geométrico. O sistema de decisão é o responsável pela tomada de decisões, contando para isso com um mecanismo de sensoriamento para obter informações acerca do mundo externo. Já o sistema motor e o geométrico são responsáveis por progressivamente refinar as decisões tomadas pelo sistema

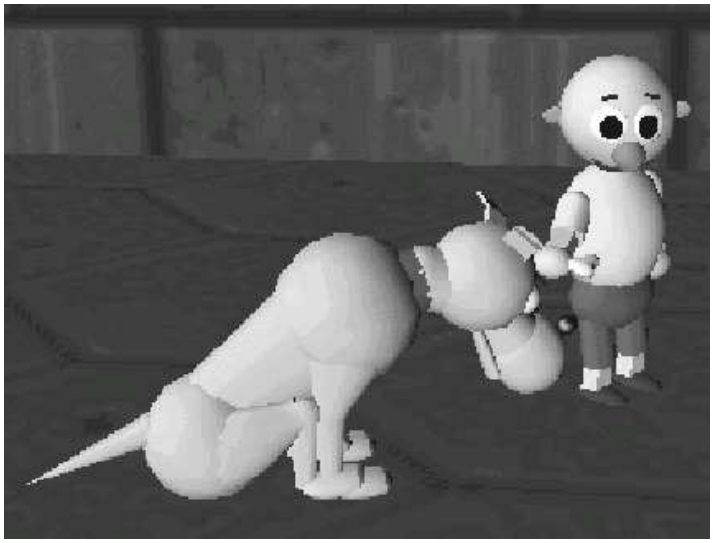

Figura 3.5: O cão Silas T. Dog e seu treinador (extraído de [Blu94]). O treinador é controlado pelo usuário que pode assim "adestrar" o cachorro.

de decisão de forma a produzir o efeito externo desejado (por exemplo, andar, correr, latir, etc.).

O sistema de decisão é baseado na idéia de **comportamentos**, que são entidades direcionadas a um objetivo específico. Estes comportamentos organizam-se de forma hierárquica, de forma que podem assumir diversos níveis de granularidade, por exemplo desde "reduzir fome" até "mastigar". A cada passo da simulação os comportamentos competem por ativação, começando em um nível superior, progredindo nos níveis inferiores até a geração de uma ação motora. Cada comportamento é responsável por computar sua própria importância baseado em variáveis internas e externas. Os dados externos que chegam até os comportamentos são filtrados por **mecanismos de disparo** (MD), que são dispositivos que determinam o grau de relevância do estado do ambiente para um dado comportamento. Um MD é constituído tipicamente por uma **função de filtro** (geralmente uma função gaussiana ou similar), que determina a relevância de um objeto externo dada a distância do agente ao mesmo. Além disso, os mecanismos de disparo são responsáveis pela alocação de **pronomes**, que são objetos que podem ser de interesse aos comportamentos associados ao mecanismo (um pronome assume o papel do foco de atenção). Os pronomes permitem o reuso de comportamentos,

permitindo que um comportamento do tipo "comer" possa ser associado por exemplo tanto a um osso quanto a uma porção de ração.

Além disso a arquitetura conta com **variáveis internas** responsáveis por modelar o estado interno da criatura, como por exemplo fome, sede, medo, etc. Estas variáveis não são no entanto estáticas - elas automaticamente aumentam ou diminuem com o tempo, ou ainda são influenciadas por ações da criatura (comer por exemplo reduz a variável fome) ou pela ativação de mecanismos de disparo (a detecção de uma aranha por exemplo altera a variável medo). As variáveis internas funcionam assim tanto na modelagem de estados fisiológicos da criatura como de estados emocionais, assumindo um papel importante também nos mecanismos de aprendizado.

A competição entre comportamentos se dá de forma hierárquica - a cada nível existem grupos de comportamentos, sendo que no interior de cada grupo somente um comportamento pode ser ativado. Os comportamentos em cada grupo competem através de um mecanismo de inibição mútua, de forma que ao final da competição somente um comportamento é o vencedor (Figura 3.6). De forma genérica, os comportamentos no topo da hierarquia representam objetivos de mais alto nível da criatura - por exemplo comer, brincar, acasalar, etc. Já os comportamentos de mais baixo nível representam diferentes estratégias para se atingir objetivos propostos por comportamentos de mais alto nível, e assim sucessivamente até o nível de comandos motores.

Outro aspecto importante na arquitetura proposta por Blumberg concerne o mecanismo de aprendizagem. Basicamente a aprendizagem busca resolver três problemas (obtidos a partir da etologia):

- um comportamento pode levar a alcançar um objetivo, em uma situação nova;
- novos mecanismos de disparo antecipam ou incrementam o valor de mecanismos de disparo já existentes;
- adaptação dos valores dos mecanismos de disparo existentes.

A aprendizagem tem como chave as variáveis internas da criatura. Quando uma variável interna passa por uma mudança significativa, ela gera um "grupo de descobrimento", que nada mais é que um conjunto de mecanismos de disparo. Estes mecanismos de disparo no entanto

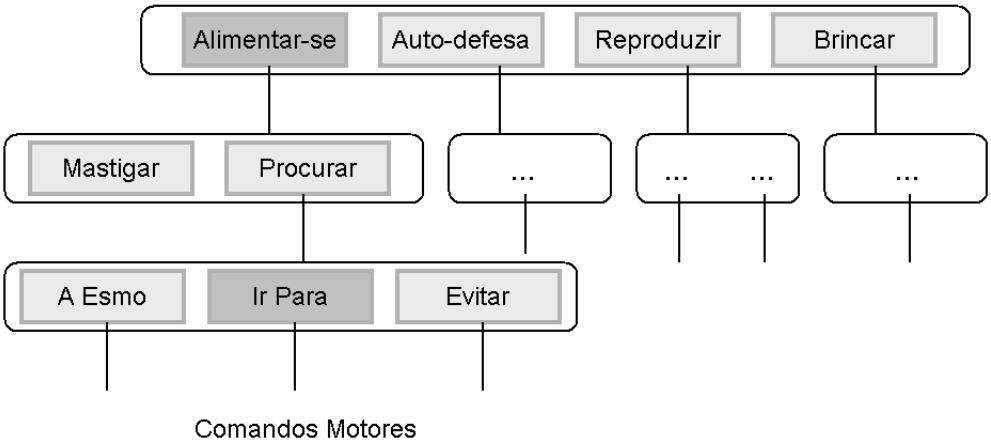

Figura 3.6: Hierarquia de comportamentos apresentando o disparo do comportamento vencedor.

não são acionados somente por variáveis externas, mas são formados como uma combinação dos últimos objetos de interesse para a criatura e os últimos comportamentos ativos, levando ainda em consideração as diferenças temporais entre as ativações dos comportamentos. Para obter tais dados, a criatura possui uma memória de curto termo que armazena os três últimos objetos de interesse (pronomes) bem como os cinco últimos comportamentos ativos. Isso possibilita à criatura associar comportamentos a alterações em variáveis externas, alterando o valor desses mecanismos de disparo quando necessário (por exemplo, a variável interna "fome" pode associar "sentar" a "comida", e mais ainda, associar o comando "sentado!" a ação "sentar"). A criatura possui ainda níveis de interesse associados ao aprendizado, que ditam quanto tempo a mesma deve esperar para obter uma recompensa, impedindo que a criatura fique inativa por longos períodos de tempo.

A arquitetura proposta ainda aborda diversos outros aspectos relacionados tanto ao sensoriamento quanto à implementação de movimentos por parte da criatura. De forma simplificada, existem três formas simplificadas de sensoriamento (para o sistema ALIVE):

- sensoriamento do mundo real (permitindo a interação do usuário com as criaturas);
- sensoriamento direto (a criatura interroga diretamente propriedades de objetos e outras criaturas);

- visão computacional (utilizada principalmente para a navegação).

No que concerne o controle motor da criatura, o mesmo tem como base "graus de liberdade" que controlam o acesso à geometria subjacente. Cada habilidade motora necessita de um conjunto de graus de liberdade para executar (por exemplo, andar requer acesso aos graus de liberdade correspondentes às pernas e patas de um cachorro). Isso permite que comportamentos não vencedores possam executar ações motoras, contanto que não interfiram com ações comandadas pelo comportamento vencedor (permitindo por exemplo que um cachorro que se encontra "feliz" possa andar balançando seu rabo).

3.2.2 Reynolds

Reynolds foi o pioneiro no desenvolvimento da área denominada animação comportamental com seu trabalho [Rey87]. Neste trabalho Reynolds introduz os *boids*, pequenas entidades capazes de se movimentar em grupo em direção a um objetivo, comportando-se de forma similar a revoadas de pássaros ou cardumes de peixes. Em sua forma mais básica cada boid de Reynolds é regido por três comportamentos simples:

- distanciar-se de outros boids vizinhos de forma a evitar colisões;
- alinhar o sentido de movimento com os boids vizinhos;
- aproximar-se da posição média dos boids vizinhos de forma a manter um grupo coeso.

O comportamento final é dado através da soma vetorial de cada um dos comportamentos individuais. Cada um desses comportamentos é regulado por parâmetros que ditam a importância do mesmo, permitindo assim ao grupo assumir comportamentos distintos de acordo com o desejado (por exemplo, maior alinhamento e proximidade dos boids, ou maior distanciamento e movimento mais caótico).

Além destes comportamentos, Reynolds introduziu outros que possibilitam aos boids seguir em direção a um alvo pré-determinado e desviar de obstáculos presentes no ambiente.

Cabe notar ainda que os boids de Reynolds são a forma de animação comportamental mais utilizada até o presente instante, podendo ser vista em diversos filmes, como nos morcegos e pingüins do filme "Batman: O Retorno".

3.2.3 Tu e Terzopoulos

O trabalho de Tu e Terzopoulos [TT94] visou simular de forma realista o comportamento de peixes. No entanto, diferentemente de Reynolds, seu trabalho inclui modelos detalhados sobre o processo de locomoção dos peixes, bem como sistemas sensoriais e comportamentais bem mais refinados. A animação do movimento dos peixes se dá através de um complexo sistema de molas e controladores motores, interagindo com o ambiente através de regras bem definidas de física e de hidrodinâmica. Já o sistema sensorial dos peixes recebe dados como informações geométricas, distâncias, cores e materiais, além de poder interrogar diretamente ao sistema informações como a velocidade dos objetos. O mesmo é capaz ainda de sensoriar a temperatura e o grau de luminosidade ambientes.

O sistema comportamental dos peixes é controlado essencialmente por um mecanismo denominado "gerador de intenções", responsável por determinar qual comportamento o peixe deve adotar a cada instante. Esse mecanismo toma como base para suas decisões, além do meio ambiente, três variáveis de estado interno: fome, libido e medo. Além disso, os peixes possuem preferências por determinadas características ambientais, tais como luminosidade e temperatura. Basicamente o gerador de intenções funciona em etapas: primeiramente verificando se existem obstáculos por perto, em seguida verificando a presença de predadores, ou então computando seu desejo de fome ou libido a fim de procurar comida ou um(a) parceiro(a) adequado(a). Ainda caso nenhuma dessas alternativas possa ser ativada passam a valer as preferências do peixe, que ditam se ele procura sair do ambiente em que se encontra a fim de ir para locais mais ou menos iluminados ou mais ou menos aquecidos. Dessas escolhas resulta um comportamento para o peixe, sendo que existem oito comportamentos básicos: evitar obstáculo estático, evitar peixe, comer, acasalar, abandonar local, passear, fugir e agrupar-se.

Após um comportamento ser ativado pelo gerador de intenções, um segundo mecanismo é acionado. Este mecanismo é responsável por determinar o foco de atenção do peixe, de forma a restringir a interação do mesmo apenas ao elemento mais importante (o obstáculo mais próximo, o predador de maior perigo, o parceiro mais atraente). Os peixes possuem ainda uma memória de curto termo de forma a evitar que a atenção dos mesmos se desvie muito rapidamente caso a execução de uma ação seja temporariamente interrompida por um

comportamento mais importante (por exemplo desviar de um obstáculo enquanto fugindo de um predador). Somente o último comportamento (e informações associadas) são armazenados.

Através ainda de pequenas modificações no gerador de intenções foram criados diferentes tipos de peixes, como presas, predadores ou ainda pacifistas (cujo único objetivo era acasalar-se), gerando uma gama muito variada de comportamentos e provendo uma animação impressionante, até certo ponto similar ao que esperaríamos de peixes reais.

3.2.4 Funge

O trabalho de Funge [FTT99, Fun98] visa aprimorar o trabalho de Tu e Terzopoulos através da "modelagem cognitiva". Tal modelagem visa definir uma linguagem (denominada *cognitive modeling language*, ou CML) para o controle de criaturas em ambientes artificiais e animações. CML é baseada em cálculo situacional, e como tal é composta por bases de regras do tipo "se (situação S) então execute (ação A)" e "(ocorrência O_1) implica na (ocorrência de O_2)". CML possui ainda primitivas não determinísticas, que possibilitam à criatura assumir comportamentos distintos em situações similares.

O fato de CML se embasar em uma linguagem matemática permite a utilização de várias ferramentas, facilitando principalmente o planejamento de ações por parte das criaturas. Basicamente o planejamento se dá através da construção de uma árvore de possibilidades e do corte heurístico de ramos não interessantes.

O trabalho de Funge inclui ainda um complexo sistema motor para controle de criaturas, largamente baseado no trabalho de Tu e Terzopoulos, levando em consideração diversos aspectos de computação gráfica, como detecção de colisão, modelagem geométrica e texturização.

Vários aplicativos interessantes foram desenvolvidos utilizando-se da CML, entre eles dinossauros exibindo comportamento predador-presa e um mundo submerso compostos por sereias e tubarões. Entretanto CML possui as mesmas deficiências apresentadas por outros sistemas baseados em regras, não apresentando soluções relativas à aprendizagem e adaptação.

3.3 Resumo

Neste capítulo analisamos sob a ótica dos sistemas inteligentes os principais produtos comerciais e as principais iniciativas acadêmicas em jogos, animação e entretenimento eletrônico. Assim, utilizando do conhecimento apresentado no Capítulo 2, construímos uma base que permita melhor situar e compreender as idéias apresentadas no Capítulo 5.

Capítulo 4

RP-Nets

Neste capítulo apresentamos os conceitos básicos envolvendo as Redes de Processamento de Recursos (*Resource Processing Networks* - ou RP-Nets), uma ferramenta formal-computacional derivada das Redes de Petri, e que foi utilizada para o desenvolvimento de nosso trabalho.

4.1 Definição

Uma rede de processamento de recursos (RP-Net) é uma ferramenta matemática que se destina a modelar sistemas que envolvem o processamento de recursos de qualquer natureza, transformando e/ou consumindo esses recursos de acordo com a dinâmica do sistema. Esse processamento de recursos pode ocorrer de maneira paralela e concorrente, podendo ser realizada por múltiplos agentes processadores simultaneamente. Além disso, esses agentes processadores podem envolver alguma forma de aprendizagem e/ou adaptação, podendo inclusive modificar-se a si mesmos de maneira recursiva. Assim sendo, podemos classificar uma RP-Net como um modelo formal-computacional de sistemas a eventos discretos. Tanto a estrutura como o funcionamento de uma RP-Net podem ser visualizados de forma gráfica, o que facilita seu entendimento e uso. Como os recursos processados por uma RP-Net podem ser de ordem genérica, é lícito assumir que eles possam representar conhecimentos, idéias, signos e outros artefatos comumente utilizados em sistemas inteligentes. Assim, podemos categorizar o uso de uma RP-Net também como uma técnica de sistemas inteligentes. A

seguir apresentaremos as noções básicas envolvendo RP-Nets.

Uma RP-Net é constituída por um conjunto de **recursos** situados em **lugares** conectados por **arcos**, formando um tipo especial de grafo direcionado. A seguir descrevemos cada um desses elementos.

Um **recurso** representa qualquer conceito / elemento concreto ou abstrato que tenha as seguintes características:

- Cada recurso é único e identificado pelo seu nome. Um nome pode ser realocado para outro recurso uma vez que o recurso deixe de existir, mas um recurso nunca pode trocar de nome durante sua existência.
- Um recurso pode possuir funções de transformação. Um recurso que possui tais funções é denominado um **recurso ativo**, caso contrário é denominado um **recurso passivo**.
- Um recurso ativo é capaz de consumir e/ou gerar outros recursos.

Recursos passivos podem tanto ser recursos materiais como recursos de informação. Exemplos de recursos materiais poderiam ser: peças, matérias-primas, componentes, dinheiro, etc.. Exemplos de recursos de informação poderiam ser textos, documentos, diagramas, dados, planilhas, tabelas, etc..

Recursos ativos, ao contrário de recursos passivos, que são meramente entidades manipuladas, são recursos capazes de processar outros recursos, ou seja, executam atividades de processamento de recursos. Esses recursos processados por um recurso ativo podem ser tanto recursos passivos, como recursos ativos. Um recurso ativo pode inclusive processar a si próprio.

Recursos ativos podem executar seu processamento com ou sem tomada de decisão. Vamos explicar mais aprofundadamente esta qualidade. Uma vez que um certo conjunto de recursos necessários para um determinado processamento esteja disponível, o recurso ativo pode então processá-los, sem distinguir entre diferentes instâncias desses recursos (caso elas existam). Neste caso, dizemos que esse processamento é um processamento sem tomada de decisão. Um exemplo de um processamento desse tipo ocorre quando temos uma máquina montando uma peça A, que necessita de duas sub-peças B e C para sua confecção. A máquina é o recurso ativo. As sub-peças B e C são os recursos necessários para que o

processamento da peça ocorra. Neste caso, havendo diversas sub-peças dos tipos B e C, a máquina não faz nenhuma diferença entre duas ou mais peças do tipo B, ou duas ou mais peças do tipo C. Qualquer peça de um mesmo tipo é equivalente para fins de processamento. Recursos ativos deste tipo são chamados de **mecânicos**. Entretanto, nem todos os recursos ativos em um sistema de processamento de recursos serão sempre mecânicos. Recursos ativos podem ser pessoas (trabalhadores), e suas atividades podem envolver algum tipo de tomada de decisão com relação ao processamento sendo efetuado. Assim, caso hajam múltiplas instâncias de um recurso necessário ao processamento, um recurso ativo pode avaliar essas múltiplas instâncias, e decidir qual delas seria mais adequada para o processamento. No caso do recurso ativo ser capaz de efetuar mais de um tipo de processamento, ele também poderia decidir qual processamento seria executado. Um exemplo de um recurso ativo nessas circunstâncias poderia ser o caso de um professor orientador que forma alunos de pós-graduação, transformando-os em professores. Nesse caso, os professores são recursos ativos e os alunos (enquanto alunos) são recursos passivos. Um professor deve escolher dentre os alunos disponíveis, aqueles que estão mais aptos, selecionando-os como seus orientados. Após o processamento (que pode durar alguns anos!), esses alunos são transformados em professores, e irão realimentar o processo, passando a processar seus próprios alunos. Os professores podem ter também diferentes metodologias de orientação, de acordo com o tipo de aluno (por exemplo, alunos com mais ou menos iniciativa, mais ou menos habilidade de trabalho em grupo, gosto por leitura, habilidades de programação, etc). Nesse caso, o professor deverá decidir qual metodologia aplicar em cada aluno selecionado, tornando o processo tão complexo quanto se desejar. Verifica-se neste tipo de recurso ativo a existência de um elemento de tomada de decisão que é determinante na dinâmica do sistema. Recursos ativos deste tipo são chamados de **inteligentes**.

De maneira mais completa (englobando tanto recursos *mecânicos* quanto *inteligentes*), podemos definir um **recurso ativo** como um tipo de recurso especial que tem o seguinte comportamento: - é capaz de selecionar outros recursos, dentre um conjunto dado, dos quais pretende extrair o conteúdo; - é capaz de destruir ou preservar esses recursos escolhidos; - uma vez de posse desses recursos, realiza operações internas, de forma a gerar outros recursos; - é capaz de gerar novos recursos, a partir da assimilação e transformação do conteúdo dos recursos selecionados previamente; - realiza incessantemente esta atividade de escolha,

assimilação e geração de novos recursos, de maneira autônoma e independente; - pode possuir diferentes maneiras de escolher e processar os recursos em seu ciclo de vida.

No mundo real, recursos (tanto ativos quanto passivos) ocupam um lugar no espaço (seja esse espaço o espaço físico, como a memória de um computador no caso de recursos de informação), de onde podem executar seu processamento (no caso de recursos ativos), ou de onde podem ser manipulados (no caso de recursos passivos).

Assim, de maneira formal, um **lugar** (ou *place*) é definido como uma entidade passiva capaz de armazenar zero ou mais recursos quaisquer. No mundo real, o conceito de lugar está associado a uma idéia contínua (uma região do espaço n-dimensional), pois os lugares se ligam continuamente de maneira topológica criando um *continuum* de lugares que chamamos em sua totalidade de *espaço*. Para nossos propósitos, torna-se necessário que discretizemos esse conceito de conexão entre lugares. Assim, ao invés de uma teia contínua conectando lugares, definiremos a conexão entre lugares por meio de arcos conectando um conjunto de lugares discretos.

Dessa maneira, em nosso modelo, um lugar pode se conectar a outro lugar por meio de **arcos** direcionados, conforme podemos visualizar na Figura 4.1.

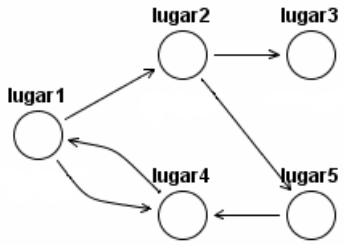

Figura 4.1: Topologia de uma RP-Net.

Os pontos de contato dos arcos com os lugares são chamados de **portas** dos lugares. Uma porta pode ser uma porta de entrada ou de saída, dependendo da direção do arco associado à mesma. Um recurso ativo armazenado em um determinado lugar somente poderá se servir dos recursos que estejam em lugares conexos ao lugar em que se encontram. Dessa forma, um sistema de processamento de recursos constitui-se de uma rede de lugares conectados por arcos, onde recursos ativos e passivos interagem de acordo com a conexão entre os lugares.

Dissemos anteriormente que recursos ativos executam ações sobre recursos passivos. Es-

sas ações são determinadas através da definição de **funções de transformação**. À cada função estão associadas portas de entrada ou de saída, através das quais a função ou obterá os recursos de entrada ou colocará os recursos resultantes do processamento. Uma função deve ser capaz tanto de realizar a seleção dos recursos de entrada, avaliando sua importância de acordo com o conteúdo dos mesmos, como de realizar as operações necessárias sobre os recursos, gerando os recursos de saída. Em virtude deste comportamento, as funções demandam a definição de 2 conjuntos de operadores especiais: os **operadores de avaliação** e os **operadores de transformação**. Os operadores de avaliação determinam, para um conjunto de recursos candidato a ser processado por uma dada função, qual é o interesse do recurso ativo em selecionar cada um desses recursos. Esse interesse é representado por um valor de **utilidade** da função. Outro valor determinado pelos operadores de avaliação é o **modo de acesso** com o qual a função deseja acessar determinada porta. Este modo de acesso determina como o recurso acessível através da porta deve ser utilizado - se por exemplo o recurso deve ser acessado exclusivamente por esta função, se o acesso pode ser compartilhado, ou ainda se acesso implica no consumo do recurso em questão. Já os operadores de transformação tomam os recursos selecionados previamente e realizam as operações necessárias para gerar zero, um ou mais novo(s) recurso(s).

A seguir apresentamos as definições formais dos conceitos delineados anteriormente.

Definição 4.1: Lugar

Um lugar é uma entidade passiva p_j, $j > 0$ que agrega um ou mais recursos. O conjunto de todos os lugares é dado por $P = \{\ p_1, p_2, ..., p_n\ \}$.

Definição 4.2: Portas de um Lugar

Cada lugar possui um conjunto finito de portas de entrada e de portas de saída. Assim, $g_e(p,n)$, $n > 0$ denota a n-ésima porta de entrada do lugar p. Da mesma forma $g_s(p,n)$, $n \in \mathbb{N}$, $n > 0$ denota a n-ésima porta de saída do lugar p.

Definição 4.3: Arco

Definimos um arco $a = (g_s(p_j, m), g_e(p_i, n))$ como sendo o arco conectando a m-ésima porta de saída do lugar p_j à n-ésima porta de entrada do lugar p_i. O conjunto de todos os arcos é $A = \{ a_1, a_2, ..., a_q \}$.

Definição 4.4: Recursos

Definimos o conjunto de recursos de uma rede como:
$R = \{ r_1, r_2, ..., r_q \}$
onde cada r_i, $i \in \mathbb{N}$, $i > 0$ corresponde a um recurso.

Definição 4.5: Marcação

Definimos uma marcação M como uma função $M : R \rightarrow P$, que associa recursos a lugares.

Definição 4.6: RP-Net

Uma RP-Net é uma ênupla $RPN = \{P, A, M\}$ na qual:
$P = \{ p_1, p_2, ..., p_m \}$ é o conjunto finito de lugares,
$A = \{ a_1, a_2, ..., a_n \}$ é o conjunto finito de arcos e
M é a marcação corrente da rede.

Uma RP-Net como acima definida é uma entidade estática que representa apenas um estado atual da rede. A seguir definimos como se dá a dinâmica operacional da rede.

Definição 4.7: Recurso Ativo

Um recurso $r \in R$ é chamado ativo quando é capaz de executar atividades de criação, transformação e destruição sobre outros recursos. A atuação dos recursos ativos em uma RP-Net é que confere à mesma seu comportamento dinâmico.

Em função da atuação dos recursos ativos, a marcação de uma RP-Net varia no tempo. Essa atuação envolve para cada recurso ativo etapas de decisão e ação. Na etapa de decisão

o recurso ativo avalia cada outro recurso disponível nos lugares conexos ao lugar em que se encontra e decide quais irá utilizar. Na etapa de ação ele efetivamente utiliza esses recursos (destruindo-os ou não) e criando novos recursos. Esse comportamento é por demasiado complexo para ser definido de maneira generalizada, sendo portanto sumarizado pelo algoritmo DA apresentado no Algoritmo 2.

Algoritmo 2 Algoritmo DA.

Primeiramente definimos uma ação *ac* como sendo uma estrutura composta por uma função de transformação, um conjunto ordenado de recursos de entrada, e um valor de utilidade. A função de transformação f é composta pelos operadores de transformação e avaliação, e pelos conjuntos ordenados de portas de entrada e saída associadas à mesma. O algoritmo DA procede então da seguinte forma:

$AC \leftarrow \emptyset$
para cada recurso ativo r localizado em um lugar p
    para cada função de transformação f de r

        criar ação ac contendo a função de transformação f, nenhum recurso e valor utilidade 0.
        $n \leftarrow$índice da primeira porta de entrada associada à função f
        $AC \leftarrow obtemCombinacoes(ac, g_e(p,n))$
para cada $ac \in AC$

    obter valor de utilidade u de ac aplicando o operador de avaliação de ac sobre os recursos de entrada de ac
$AC \leftarrow resolveRestricoes(AC)$
para cada ação $ac \in AC$
    executa operador de transformação associado a ac

O algoritmo *resolveRestricoes* recebe como parâmetro um conjunto de ações e retorna outro conjunto de ações. Este algoritmo é responsável pela resolução de restrições, selecionando dentre todas as ações aquelas que podem ser executadas concorrentemente, priorizando aquelas com maior valor de utilidade. Um exemplo de um algoritmo de resolução de restrições, o *BMSA*, é apresentado no Capítulo **??**.

Apresentamos a descrição de *recebeCombinacoes* no Algoritmo 3.

Algoritmo 3 Algoritmo recebeCombinacoes.

O algoritmo *recebeCombinacoes* recebe como parâmetros de entrada uma ação ac e uma porta de entrada $g_e(p,n)$, e resulta em um conjunto de ações AC.

$AC' \leftarrow \emptyset$
se a $g_e(p,n) = \emptyset$
    $AC' \leftarrow ac$
caso contrário
    para cada recurso de entrada r acessível através de e

        criar ação ac' contendo a função f de ac, todos os
            recursos de entrada de ac acrescidos do recurso r
        $g_e(p,n') \leftarrow$ próxima porta de entrada de f depois
            de $g_e(p,n)$ ou $\emptyset$ se não existem mais portas
        $AC' \leftarrow AC' + obtemCombinacoes(ac', g_e(p,n'))$
retorna AC'

Definição 4.8: Sequência de Ocorrência

Uma sequência potencialmente infinita de ocorrências de uma RP-Net é uma sequência de marcações descritas da seguinte forma:

$M_1[DA_1] > M_2[DA_2] > M_3 ...$

Na qual DA_i corresponde à aplicação do algoritmo DA no passo i. Assim, de maneira genérica, temos $M_i[(DA_i)] > M_{i+1}$ para todo $i \in \mathbb{N}$, $i > 0$ sendo que M_1 é denominado a marcação inicial da rede.

4.2 Funcionamento

Inicialmente, após a definição da topologia da rede (lugares e arcos), distribuímos os recursos nos lugares de uma RP-Net a partir de uma marcação inicial definida na estrutura da

rede. No caso especial de recursos ativos, para cada argumento de uma função de transformação deve haver no lugar onde este recurso é inserido uma porta de entrada e, da mesma maneira, para cada saída da função de transformação, deve haver uma porta de saída. Uma vez que a rede entre em operação, os recursos ativos iniciam sua atividade, selecionando dentre os recursos disponíveis dos lugares aos quais estão conectados a partir de uma porta de entrada, e disponibilizando novos recursos nos lugares pelos quais estão conectados a partir de portas de saída. Esse comportamento, denominado de **disparo** de um recurso ativo, pode ser ilustrado na Figura 4.2.

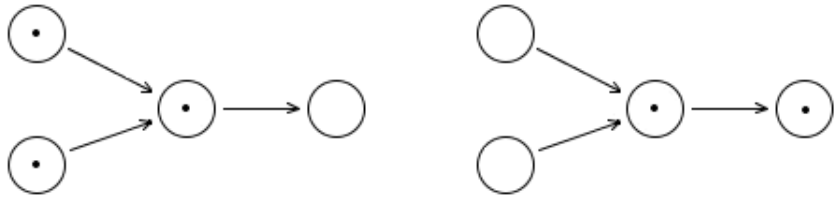

Figura 4.2: Disparo de um recurso ativo.

O funcionamento interno de uma rede (dado pelo algoritmo DA) se dá da seguinte forma: em um primeiro instante, para cada função, **ações potenciais** são geradas. Uma ação potencial é uma combinação possível entre os recursos acessíveis através das portas de entrada associadas à função. Por exemplo, caso uma função receba a entrada de 2 portas, sendo que a primeira está conectada a um lugar que possua 10 recursos, e a segunda porta está conectada a outro lugar que possua 5 recursos, 5 x 10 = 50 ações potenciais são geradas. De posse dessas ações potenciais, o funcionamento da rede procede da seguinte forma: - para cada ação potencial, o operador de avaliação da função correspondente é acionado a fim de se obter a utilidade da ação e o modo de acesso dos recursos; - dentre estas ações algumas são selecionadas para execução através de um algoritmo de resolução de restrições, eliminando ações conflitantes e prioritizando ações com maior utilidade; - os operadores de transformação das ações selecionadas são executados.

4.3 Análise e Aplicações

RP-Nets têm um comportamento muito parecido com o de redes de Petri[Mur89]. Sistemas de processamento de recursos poderiam perfeitamente ser modelados por Redes de Petri, principalmente Redes de Petri Estendidas, tais como Redes de Petri Coloridas ou Redes de Petri Orientadas a Objeto. O uso de RP-Nets na modelagem de sistemas de processamento de recursos possui algumas vantagens, entretanto. Em Redes de Petri, os recursos ativos seriam parcialmente modelados por transições, parcialmente por tokens (que guardariam informações a respeito do processamento dos recursos). Apesar disso ser viável do ponto de vista computacional, o modelo via RP-Nets supõe-se que seja mais conveniente, por explicitar os recursos ativos e facilitar a criação de redes onde a compreensão de como os recursos são processados torna-se mais simples. Outra vantagem importante das RP-Nets é a possibilidade de se modelar processos de aprendizagem e adaptação (Figura 4.3) o que com dificuldade poderia ser modelado em redes de Petri. Com esta capacidade, as RP-Nets permitem a representação de uma vasta gama de comportamentos sofisticados, o que se consegue considerando-se que a noção de um recurso ativo é uma abstração bastante eficiente em termos de modelagem. A possibilidade de se criar recursos ativos dinamicamente, em tempo de execução, traz uma riqueza de comportamentos que precisa ainda ser explorada em todas as suas potencialidades.

Recursos podem representar partes quaisquer de um sistema, podendo ser por exemplo estruturas de representação de conhecimento, sinais sensoriais, sinais de atuação, bases de regras, etc. Recursos ativos podem ser elementos dinâmicos quaisquer, tais como neurônios, redes neurais inteiras, máquinas de inferência fuzzy, operadores evolutivos de crossover ou mutação, etc. Por exemplo, na figura 4.3, temos a representação de uma rede neural aplicada em um ambiente de controle. No lugar A se encontra um sensor, que envia dados sensoriais ao lugar B. Esses dados sensoriais vão alimentar uma rede neural colocada em C, gerando dados de atuação que são enviados ao lugar E. Por fim, o atuador em F coleta esses dados de atuação e transforma-os em atuações no ambiente. O recurso em D é responsável pela aprendizagem da rede neural contida em C. Ele toma os dados sensoriais em B, atualizando a própria estrutura da rede neural em C.

Uma das vantagens de se representar redes neurais, algoritmos evolutivos e sistemas fuzzy

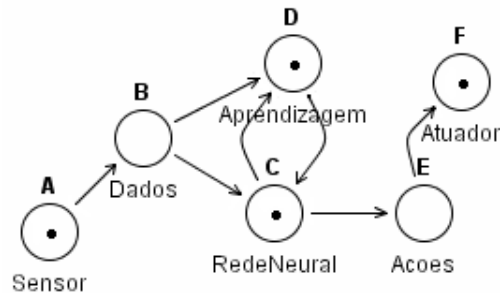

Figura 4.3: Exemplo de um sistema com adaptação e aprendizagem.

por meio de RP-Nets é a possibilidade de se integrar diferentes partes desses paradigmas, gerando sistemas neuro-fuzzy-evolutivos de caráter híbrido, mantendo a inteligibilidade de seu funcionamento. Pode-se ao mesmo tempo encapsular comportamentos - um recurso tanto pode representar um único neurônio como uma rede neural completa, um único cromossomo como um algoritmo genético completo, uma única regra fuzzy, como uma base de regras completa, ou ainda ser constituído por uma outra rede de processamento de recursos qualquer. Essa integração é possível, sem se perder todas as potencialidades que estes paradigmas podem trazer, individualmente. De uma maneira geral, RP-Nets apresentam-se como um paradigma muito adequado para o desenvolvimento de sistemas inteligentes, tanto por sua expressividade quanto pela facilidade em visualizar graficamente o comportamento do sistema. Por estas facilidades foram a ferramenta adotada para o desenvolvimento desse trabalho, como explicitado mais tarde na Seção 5.4.2.

É interessante observar também a similitude entre a estrutura e a dinâmica de funcionamento das RP-Nets e os trabalhos apresentados anteriormente no Capítulo 3, em especial [Blu94, Mae89, KZ02, Lai00a]. Em linhas gerais, todos estes trabalhos são baseados em comportamentos estruturados hierarquicamente, comportamentos estes que podem ser mapeados em recursos ativos ocupando lugares estratégicos em uma RP-Net, e competindo pela ação. Cada comportamento é capaz de auto-avaliar sua importância de acordo com as leituras dos sensores e o estado interno do agente. Esse valor de importância é mapeado diretamente em uma RP-Net para o valor de utilidade de uma função de transformação. Isso permite que os comportamentos realizem hierarquicamente uma competição por ativação que é resolvida concorrentemente e transparentemente pelo algoritmo de soluções de restrições da RP-Net.

Por exemplo, para a hierarquia presente na Figura 3.6 do Capítulo 3 podemos ter uma RP-Net que a implementa conforme a Figura 4.4.

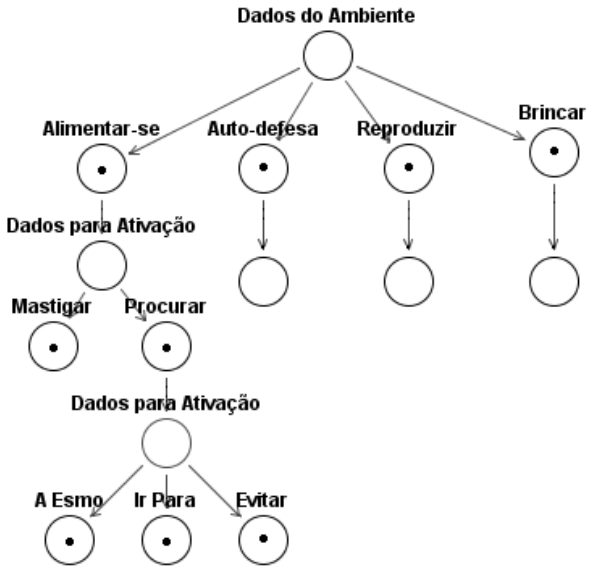

Figura 4.4: Hierarquia de comportamentos representada em uma RP-Net.

Uma Ferramenta Anterior: O SNToolkit

Para uma melhor compreensão da dimensão do trabalho aqui desenvolvido, é necessário que apresentemos também uma ferramenta já desenvolvida anteriormente pelo grupo de pesquisa e que foi a base do desenvolvimento da ferramenta utilizada para esta tese: o SNToolkit. O SNToolkit é uma ferramenta para o desenvolvimento das chamadas Redes Semiônicas, bem como suas precursoras, as Redes de Agentes ([Gue00, Gom00]). Podemos encarar uma Rede Semiônica como um caso especial de uma RP-Net. Supondo-se que os recursos sendo manipulados pela RP-Net são signos, o funcionamento da rede pode ser equiparado ao processo de semiose proposto na semiótica Peirceana [Gud02]. O SNToolkit possibilita tanto o projeto de redes quanto sua simulação e depuração. O mesmo foi desenvolvido na plataforma Java pelo Grupo de Semiótica Computacional, e está em sua versão 3. Originalmente, pretendia-se utilizar o SNToolkit no desenvolvimento da arquitetura de cont-

role dos bots. Por uma série de motivos, que serão apresentados posteriormente no Capítulo 5, entretanto, essa idéia acabou sendo abandonada, e o SNToolkit foi utilizado somente no projeto visual da RP-Net, devido às semelhanças topológicas entre uma RP-Net e uma Rede Semiônica. No entanto, toda a experiência no desenvolvimento do SNToolkit (no qual o autor desta dissertação colaborou ativamente), serviu como base para a construção do **RPNToolkit**, que foi a ferramenta efetivamente usada no decorrer deste trabalho.

O SNToolkit é composto por diversos módulos, que são os seguintes:

- SNE (*Semionic Network Engine*): módulo responsável por executar uma rede. Uma rede é descrita através de um modelo interno definido através de um conjunto de classes e interfaces Java denominado SNM *(Semionic Network Model*). Não possui interface gráfica.

- SNDesigner (*Semionic Network Designer*): módulo responsável por prover uma interface gráfica para a criação e edição de redes, gerenciando o SNM e realizando a geração de código SNM a ser executado pelo SNE.

- SNDebugger (*Semionic Network Debugger*): módulo que permite a execução de modelos SNM pelo módulo SNE com visualização gráfica para depuração.

O processo de edição de uma rede do ponto de vista do aplicativo consiste nos seguintes passos: inicialmente a rede é editada usando-se o SNDesigner. A partir do modelo da rede é então gerado um conjunto de classes Java em formato fonte (.java) que representam este modelo. Estes arquivos são então compiladas pelo compilador Java (ainda utilizando-se da ferramenta SNDesigner). Os arquivos resultantes dessa compilação podem então ser executados pelo SNE, podendo a rede ser depurada através do uso do SNDebugger conectado ao SNE. Podemos ver na Figura 4.5 este fluxo de forma resumida.

O SNDesigner e o SNDebugger procuram facilitar para o usuário a realização das diversas tarefas descritas acima. A edição é feita através de uma interface gráfica modular (Figura 4.6), na qual são definidas de forma gráfica todos os elementos da rede e editados os trechos de código Java para execução pelas funções de transformação. O SNDesigner permite ao usuário ainda salvar o modelo, que utiliza um formato em padrão XML (*eXtensible Markup Language*) para serialização da rede, permitindo assim posterior re-edição da

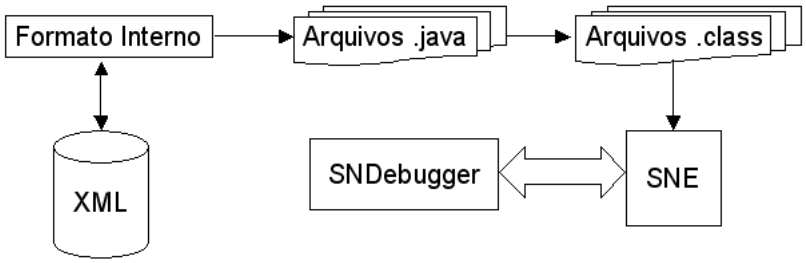

Figura 4.5: Fluxo de edição de uma rede semiônica com a ferramenta SNTool.

rede. Opcionalmente também o usuário pode realizar uma verificação semântica do modelo SNM, verificação esta responsável pela detecção de inconsistências no modelo (como nomes duplicados ou inválidos, referências inválidas, entre outros). O SNDesigner permite ainda ao usuário gerar o código Java correspondente à rede e compilá-lo. Uma vez realizada com sucesso a compilação, pode-se executar o SNDebugger (Figura 4.7), que permite a visualização gráfica da rede e de seus recursos enquanto a mesma é executada pelo motor SNE. O SNDebugger permite a visualização passo-a-passo de cada iteração, podendo realizar a visualização em tempo real do conteúdo dos lugares e recursos.

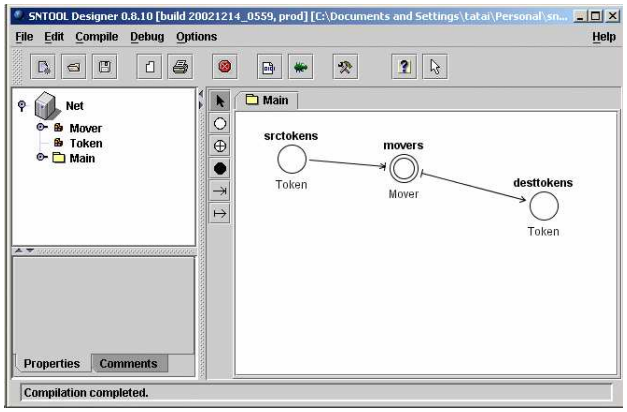

Figura 4.6: Tela ilustrando a execução do SNDesktop.

Ao contrário do SNToolkit, no RPN-Toolkit, o código referente à rede não é gerado automaticamente, por meio de geração de código. O modelo adotado no RPN-Toolkit foi o de um framework de suporte à construção de RP-Nets. Esse modelo mostrou ser mais amigável

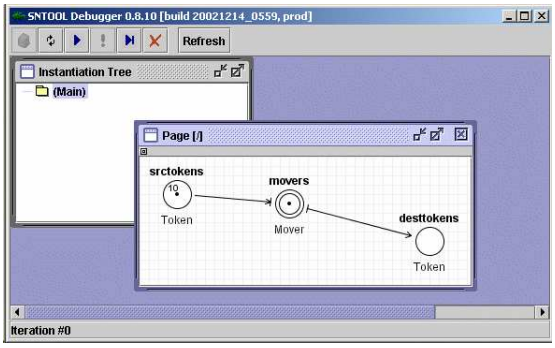

Figura 4.7: Tela ilustrando a execução do SNDebugger.

para a construção de redes mais sofisticadas, o que no SNToolkit demandava um certo esforço adicional por parte do programador que foi minimizado na nova implementação.

4.4 Resumo

Este capítulo apresentou uma nova ferramenta formal-computacional para a modelagem de sistemas inteligentes, as RP-Nets. Além disso, apresentou também de forma sucinta o modelo das Redes Semiônicas, modelo este que serviu como base para a formulação das RP-Nets. Juntamente à apresentação das Redes Semiônicas foi apresentada a ferramenta computacional usada para modelá-las, o SNToolkit, ferramenta esta que foi de grande importântica quando da implementação do RPN-Toolkit, implementação esta apresentada em maiores detalhes no Capítulo 5.

Capítulo 5

Modelo Computacional

Neste capítulo descrevemos a plataforma computacional implementada. Optamos por utilizar um jogo comercial já de grande aceitação no mercado, o jogo Half-Life Counter-Strike (CS), como plataforma junto à qual uma implementação computacional da ferramenta formal RP-Net é aplicada. CS é um jogo multiplayer, em primeira pessoa, tridimensional. Fizemos esta escolha devido aos seguintes fatores:

- O esforço de desenvolvimento de um jogo completo é extremamente alto, mesmo com o uso de toolkits gráficos disponíveis no mercado, como por exemplo CrystalSpace (`http://crystal.sourceforge.net/`) ou Ogre (`http://ogre.sourceforge.net/`). Tais toolkits foram devidamente avaliados mas descartados como opção.
- CS conta com uma comunidade bem estabelecida de desenvolvedores, o que provê maior suporte ao processo de desenvolvimento.
- CS é um jogo muito conhecido na comunidade de jogos em geral.

Em particular, optamos por realizar o controle de bots no jogo, que são jogadores de CS automatizados. Os bots são para os outros jogadores, ao menos por suas características físicas visíveis, indistingüíveis de jogadores humanos, sendo o objetivo mais comum no desenvolvimento dos mesmos o de prover aliados (ou inimigos) que tornem o jogo mais interessante ou mesmo possível (quando só existe um jogador humano por exemplo).

Nas seções a seguir começamos por introduzir alguns requisitos e peculiaridades do desenvolvimento de arquiteturas para o controle da IA de jogos de computador. Introduzimos depois de forma detalhada o jogo Counter-Strike, e por último apresentamos a solução computacional desenvolvida.

5.1 Considerações Preliminares

Existem diversos fatores que devem ser levados em consideração no projeto e desenvolvimento de uma arquitetura para o controle de uma IA de um jogo de computador. O fator central no entanto que sempre deve nortear os demais é o conceito de **jogabilidade**. Jogabilidade aqui é entendida simplesmente como um fator subjetivo determinado através da interação dentre todos os aspectos do jogo (gráficos, som, projeto de nível, IA). A jogabilidade determina o quanto um jogador sente-se compelido a jogar o jogo, e quão imersivo e quão divertido é o mesmo para o jogador. Tendo como foco a jogabilidade, vemos que a estratégia adotada para o controle da(s) IA(s) do jogo pode ser de extrema importância, especialmente na medida em que os jogos apresentam-se em ambientes virtuais cada vez mais complexos e desafiadores.

Um aspecto subjetivo determinante à jogabilidade de um jogo diz respeito ao que comumente é chamado de *suspension of disbelief*. *Suspension of disbelief* se refere ao fenômeno que leva os usuários de um jogo a "suspenderem" seu descrédito com relação à realidade fictícia que lhes é apresentada, efetivamente possibilitando que o jogador dê credibilidade aos eventos que acontecem, com base em seu senso comum e nas regras apresentadas pelo mundo imaginário. Tal fenômeno é importante não somente em jogos mas também no domínio da animação, tendo sido previamente apresentado em [TJ81].

De forma mais detalhada, uma lista não exaustiva dos principais aspectos a serem considerados no desenvolvimento de IAs para jogos de computador são:

- **Desempenho computacional**: apesar de nos anos recentes mais e mais ciclos de execução terem sido dedicados para as IAs dos jogos (fator este acentuado com o surgimento de placas gráficas dedicadas), o desempenho da arquitetura ainda é um fator importante. Isso se acentua ainda mais em jogos em tempo real, como FPSs e RTSs.

- **Reatividade**: uma IA deve responder em tempo adequado aos estímulos provenientes do ambiente. Poucos jogos permitem IAs "filosóficas", que levam muito tempo ponderando sobre a ação a realizar.

- **Depuração**: um motor de IA deve ser de fácil depuração, o que em muitas situações significa a eliminação de comportamentos não-determinísticos ou a capacidade de torná-los determinísticos (e facilmente reproduzíveis) somente para depuração.

- **Estados internos**: uma IA é mais facilmente aceita pelos jogadores e contribui para a credibilidade do jogo (e assim para o fenômeno de *suspension of disbelief*) se possui os meios de expressar (e de agir de acordo com) "estados internos" facilmente identificáveis pelo jogador, tais como desejos, emoções e necessidades. Isto permite ao jogador "antropomorfizar" o comportamento da IA, tornando-a mais interessante como oponente ou parceiro.

- **Defeitos**: para manter o fenômeno de *suspension of disbelief*, é preferível ter uma IA mediana mas consistente do que ter uma IA brilhante com uma ou duas falhas sérias.

- **Adaptabilidade**: uma IA capaz de se adaptar a novas situações e de responder de forma diferente em situações já apresentadas permite ao jogo ser utilizado repetidamente de forma interessante (*replayability*), além de prover ao jogador parceiros mais interessantes, podendo mesmo eventualmente se adequar ao nível de habilidade do jogador.

5.2 O Jogo Counter-Strike

Nesta seção procedemos com uma análise do jogo Counter-Strike [Teab, Teaa], em especial os aspectos relevantes na construção de bots.

O jogo Counter-Strike é um jogo em primeira pessoa, tridimensional, que permite ao jogador controlar seu avatar em um ambiente virtual de forma a eliminar seus oponentes ou a atingir objetivos pré-determinados. De forma resumida, os jogadores dividem-se em dois times, os **terroristas** (*terrorists*) e os **contra-terroristas** (*counter-terrorists*), sendo que o objetivo de cada time é de ou eliminar todos os membros do outro time ou completar um

objetivo específico. A duração do jogo é dividida em **sessões**, cada sessão tendo tipicamente no máximo 5 minutos de duração. Cada sessão se passa dentro de um **mapa**, mapa este que determina os objetivos específicos da sessão e que geralmente permanece o mesmo durante várias sessões. Os objetivos específicos de cada cenário podem ser:

- Para os terroristas:
    1. Armar bomba em local pré-terminado.
    2. Impedir o resgate de reféns.
    3. Assassinar o VIP (*Very Important Person*, um personagem especial controlado por um jogador).
    4. Escapar para uma área de fuga pré-determinada.
- Para os contra-terroristas:
    1. Desarmar bomba (quando armada pelos terroristas).
    2. Resgatar reféns.
    3. Proteger o VIP (ou, caso o jogador seja o VIP, escapar para uma zona de resgate pré-determinada).
    4. Impedir que os terroristas cheguem a uma área de fuga.

Em cada mapa, somente um dos quatro objetivos acima são válidos, sendo o tipo de objetivo mais utilizado nos cenários o primeiro (armar / desarmar bomba). Podemos visualizar na Figura 5.1 screenshots de alguns dos mapas e objetivos específicos mais populares.

O jogo de Counter-Strike torna-se ainda mais interessante pois permite que cada jogador utilize quatro tipos de armamentos (armamento primário, armamento secundário, armamento terciário, granadas), além de equipamentos diversos (como lanterna, equipamento de visão noturna, alicate para desarmar bomba, coletes e capuzes de kevlar). Além disso, diversos armamento possuem um modo secundário de fogo (por exemplo uma visão de zoom para rifles com mira telescópica). Os armamentos possuem quantidade limitada de munição.

Figura 5.1: Fotos do jogo mostrando, a partir do canto superior esquerdo, no sentido horário, respectivamente: VIP no cenário *as_oilrig*; reféns no mapa *cs_office*; área de fuga de terroristas no mapa *es_trinity*; ponto de plantar bomba em *de_dust*.

É interessante notar também que os danos aos jogadores são localizados, isto é, um tiro na cabeça causa uma dano muito maior ao jogador do que um tiro por exemplo na perna. É importante ainda notar que cada jogador possui uma certa quantidade de **armadura**, armadura esta responsável por absorver parte do dano impingido por um jogador (a quantidade de armadura cresce de acordo com a compra de colete e capuz de kevlar mencionados acima e diminui à medida que o jogador sustenta danos). A armadura absorve cerca de 1/4 do dano que seria causado a um jogador, valor este que depende de fatores como localização do dano e tipo de arma.

Outra característica interessante do jogo Counter-Strike está no fato que cada jogador possui uma certa quantidade de **dinheiro**. No início de cada mapa é alocado a cada jogador uma certa quantia fixa de dinheiro, quantia esta que é renovada a cada sessão ou durante o jogo. O valor do dinheiro ganho é influenciado por três fatores:

1. Resultado da sessão anterior (se o time do jogador ganhou ou perdeu).
2. Eliminação de um inimigo.
3. Execução de um objetivo específico (por exemplo, resgatar um refém).

Este dinheiro pode ser gasto durante o início de cada sessão em áreas específicas (assinaladas através da presença de um carrinho de supermercado na tela do jogador), podendo ser usado para que o jogador possa adquirir armamentos, munições ou equipamentos.

O jogador de Counter-Strike possui diversas formas de interagir com o jogo. A primeira delas, a mais óbvia, é através da tela do jogo, que apresenta não só o mapa e outros jogadores no campo de visão do jogador, bem como outras informações relevantes ao jogo através do uso de um HUD (*Heads-Up Display*). Podemos ver uma foto da tela do jogador na figura 5.2.

Além disso, o jogador interage com o jogo fornecendo diversos comandos para controlar seu avatar, tipicamente usando-se para tal do teclado e do mouse. Estes comandos dividem-se em:

- **Movimento e mira**: permitem ao jogador se movimentar para frente, para trás ou para os lados, alterar o sentido de movimento, e alterar a inclinação e sentido da mira. Além disso, permitem ao jogador pular ou agachar-se.

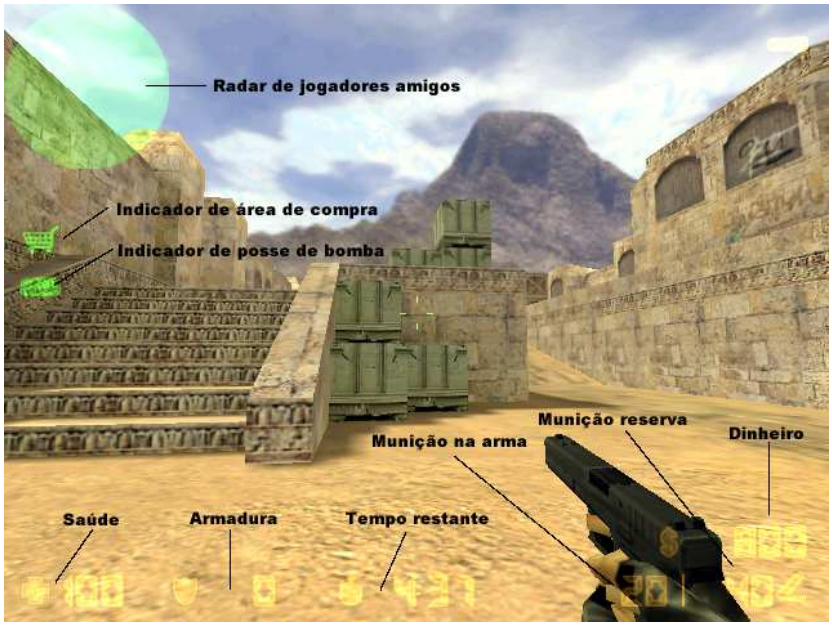

Figura 5.2: Tela do jogador mostrando seu HUD (Heads-Up Display).

- **Armas e equipamentos**: permitem ao jogador selecionar o armamento ou equipamento a ser utilizado, usá-los ou ainda ativar modos secundários. Permitem ainda ao jogador acionar dispositivos do mapa do jogo.

- **Comunicação**: permitem ao jogador enviar tanto mensagens pré-definidas quanto mensagens de texto qualquer aos outros jogadores. É possível ainda aos jogadores comunicarem-se entre si através do uso de microfones.

- **Compras**: permite ao jogador comprar itens quando localizado em área apropriada. Uma foto da tela mostrando o menu de compras pode ser vista na Figura 5.3.

Finalmente, uma outra forma de interação com o jogo se dá através dos sons emitidos durante o jogo. Os jogadores emitem sons ao andar (com ruídos distintos para diferentes tipos de terreno), no ato de acionar dispositivos, interagir com o mapa (quebrar uma janela por exemplo emite o som característico de vidro sendo quebrado), ou ainda disparar armas (armas distintas possuem sons distintos). Também o recebimento de mensagens pré-definidas se dá através da emissão de vozes pré-gravadas.

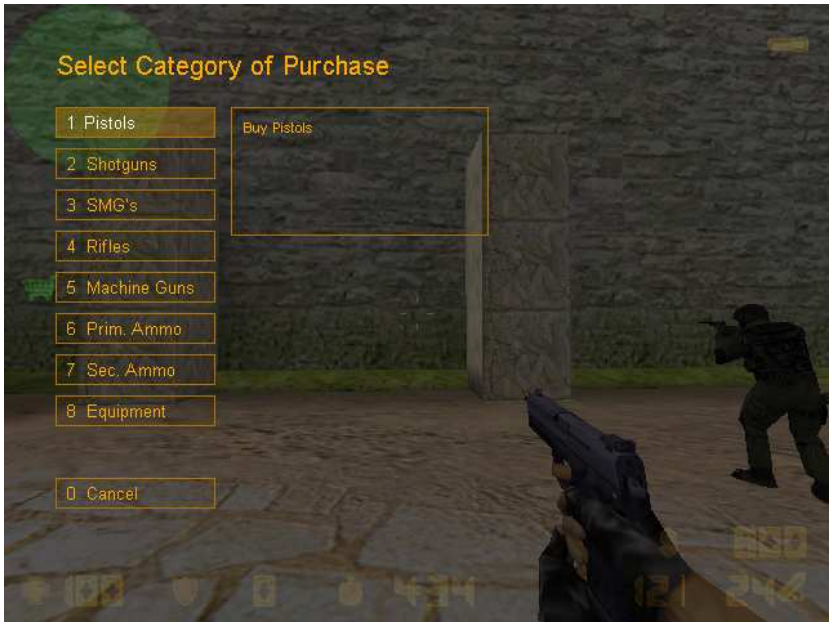

Figura 5.3: Menu de compra.

5.3 Desenvolvimento de Bots em Counter-Strike

O desenvolvimento de bots engloba uma série de peculiaridades intrínsecas ao motor de jogo e a comunicação do mesmo com o bot. Primeiramente, é interessante notar que um bot possui características distintas de interação com o jogo do que um jogador humano, uma vez que por exemplo a implementação de um sistema completo de visão computacional para interpretar a tela do jogo seria um trabalho extremamente complexo (e desnecessário para as necessidades deste trabalho). Assim sendo, a interação de bots com o jogo se dá tipicamente diretamente acessando estruturas providas pelo motor do jogo, o que permite ao bot ter acesso e manipular todos os dados diretamente. Ações de movimento e mira, gerenciamento de armas e equipamento se tornam assim razoavelmente simples. A fim de resolver o problema de visualização de características do mapa e de outros jogadores o bot acessa diretamente as estruturas correspondentes aos mesmos, podendo ainda realizar *raytracing* (traçado de raio, utilizado para determinar se existem entidades entre dois pontos dados). É claro que tal método possui seus limites, não permitindo por exemplo ao bot ver detalhes da textura do

terreno. Além disso, gerenciar o processo de navegação dentro dos mapas é extremamente difícil, razão pela qual a grande maioria dos bots se utiliza de *waypoints*, que são pontos pré-definidos de navegação inseridos por um jogador humano para facilitar a navegação do bot (mais sobre os mesmos logo abaixo), facilitando assim o planejamento das ações do bot. Por último, a interpretação de mensagens pré-definidas também é razoavelmente simples, uma vez que podemos manipulá-las através de identificadores bem definidos, não tendo que processar o som em si. É interessante observar que tanto a emissão / interpretação de mensagens arbitrárias de texto e de voz apresentam um grande desafio (abrangendo áreas da inteligência artificial como reconhecimento e geração de fala e processamento de linguagem natural), não sendo portanto formas de interação utilizadas na quase totalidade dos bots.

Segue abaixo a lista dos principais sensores e atuadores disponíveis no desenvolvimento de um bot CS:

- Sensores
  - Outros jogadores em posicionamento absoluto dentro do mapa.
  - Raytracing (em qualquer direção e altura a partir do bot).
  - Quantidade de saúde.
  - Quantidade de armadura.
  - Munição para arma selecionada.
  - Armas disponíveis para uso.
  - Armas disponíveis para compra.
- Atuadores
  - Movimentação para frente e para trás (comando efetuado a cada iteração do motor do jogo, valores inteiros usados para se delimitar o passo, valores negativos representam um passo para trás).
  - Movimentação para os lados (comumente chamada de *strafe*, comando efetuado a cada iteração do motor do jogo, valores inteiros usados para se delimitar o passo, valores positivos representam um passo para esquerda e negativos para direita).

  - Mudança de ângulo horizontal de mira (para direita e para esquerda, influencia a direção de movimento)
  - Mudança de ângulo vertical de mira (para cima e para baixo).
  - Seleção de armas.
  - Usar arma selecionada.
  - Comprar armas.
  - Recarregar arma atual.
  - Agachar.
  - Pular.

Um problema constantemente enfrentado em jogos de computador e na construção de sistemas autônomos é o de navegação no meio ambiente. Um bot de CS, em especial, precisa não somente saber como navegar em um mapa (não ficando preso em cantos, sabendo passar por obstáculos, etc), mas também como encontrar pontos favoráveis para a realização de suas ações. Isto diz respeito não somente aos objetivos, como por exemplo localizar um ponto para plantar bomba, mas também aos comportamentos individuais, como por exemplo, localizar um ponto de fuga protegido, localizar um ponto para *sniper* (ataque a distância, parado, com arma de longo alcance), etc. A fim de facilitar a navegação optamos por adotar a estratégia mais comumente utilizada em jogos tridimensionais, o uso de *waypoints*. Waypoints são marcas especiais posicionadas no mapa do cenário com o objetivo de delinear os caminhos que o bot pode seguir. Os waypoints fornecem não só esses caminhos, mas podem também indicar pontos especiais no mapa. Estes pontos especiais são:

- Elevador.
- Porta acionada por botão.
- Local para sniper.
- Ponto de fuga.
- Ponto para plantar bomba.

- Ponto de localização de reféns.
- Ponto de resgate de reféns.
- Local de fuga para VIP.
- Outros (customizados).

O uso dos waypoints só se faz possível pois os mapas de CS são predominantemente estáticos, o que permite que uma pessoa qualquer os insira aonde julgar adequado para a navegação dos bots. Por outro lado, a grande desvantagem do uso de waypoints é justamente o fato de se ter esses waypoints pré-definidos para cada mapa - a ausência de waypoints significa que o bot passa a simplesmente se locomover sem rumo, freqüentemente batendo em paredes e objetos e não conseguindo realizar nenhum objetivo que dependa de informações navigacionais, como por exemplo plantar uma bomba. Uma alternativa ao uso de waypoints seria o desenvolvimento de um mecanismo "autônomo" de navegação capaz de reconhecer e memorizar as características do ambiente de forma a permitir ao bot se locomover adequadamente e cumprir os objetivos. Tal mecanismo no entanto é demasiado complexo diante do escopo deste trabalho e assim foi adotada a navegação via waypoints.

Em segundo lugar é importante também compreender como se dá em termos computacionais a integração entre o jogo CS e os bots. Em linhas amplas, a comunicação entre um bot e um jogo qualquer (não necessariamente CS) pode se dar de duas maneiras:

1. Através da emulação de um cliente comum conectado através da rede (usando o protocolo de comunicação de rede do jogo, tipicamente usando sockets TCP/IP).
2. Através de um ponto de extensão do motor do jogo usando-se bibliotecas dinâmicas.

No caso de CS somente a opção número 2 é viável atualmente, pois o protocolo de comunicação entre cliente e servidor é fechado e de difícil compreensão (tendo sido deliberadamente encriptado para coibir a ação de jogadores desonestos). Ainda assim, uma característica que dificulta um pouco mais o desenvolvimento dos bots é que CS não consitui um jogo *per se*, mas sim em um *mod* para o jogo Half-Life (HL). Esse mod é um componente de software empacotado na forma de uma biblioteca de ligação dinâmica, ou DLL (*dynamic-link library*),

que é acessada pelo motor do jogo, permitindo que o comportamento do mesmo seja customizado. Entretanto o desenvolvimento dos bots também se dá através do uso de DLL´s , o que gera um problema pois o jogo prevê o uso de apenas uma DLL de customização (e o código de ambos HL e CS são fechados - não permitindo modificações). Esse problema é resolvido então inserindo a DLL do bot entre o motor do jogo e a DLL de CS. Assim, as chamadas relevantes aos bots são interceptadas e outras chamadas são repassadas à DLL de CS. Chamadas de *callback* são tratadas de forma similar, podendo também ser interceptadas pela DLL dos bots (figura 5.4).

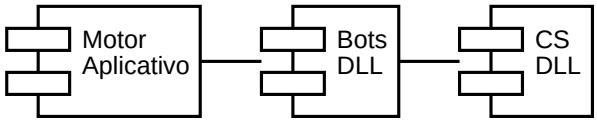

Figura 5.4: Integração dos bots e CS.

5.4 Implementação

A implementação da arquitetura de controle do bot implicou na realização de duas implementações distintas. Uma primeira implementação teve como objetivo utilizar a plataforma existente SNToolkit para o controle do bot. Esta implementação resultou em uma série de problemas técnicos que motivaram uma mudança significativa e com isso levaram a uma segunda implementação. Esta implementação efetuou uma reimplementação do SNToolkit incorporando os conceitos das RP-Nets, e a definição de uma nova arquitetura de controle.

O desenvolvimento deu-se através do uso das ferramentas Microsoft Visual C++ 6.0 e J2SDK 1.3. O código do bot foi baseado no código encontrado em [Bot], que é uma versão adaptada do código fornecido pela Valve para DLLs (Valve é o desenvolvedor de Half-Life), e a versão de Counter-Strike utilizada para desenvolvimento foi a versão 1.0.0.3.

5.4.1 Primeira Arquitetura

Inicialmente, nosso interesse era utilizar a tecnologia já disponível em nosso grupo de pesquisa - o SNToolkit para implementar o controle dos bots, pois se previa que o uso da mesma como ferramenta de depuração e projeto proveria muitas facilidades na implementação da arquitetura, acelerando o processo de desenvolvimento. Para testar a viabilidade disso, o primeiro passo foi tentar implementar o controle de um bot por meio de um programa em Java. Caso isso fosse viável, o próximo passo seria a utilização do SNToolkit, cuja implementação é em Java. Assim, procedemos a implementação do que chamamos aqui de **primeira arquitetura**, cuja finalidade era então verificar a possibilidade de implementarmos um controlador de bot em Java. Para isso, foi desenvolvida uma infraestrutura de integração ao jogo com interfaces em Java. Em um primeiro instante foi implementada uma interface para o controle genérico de bots, permitindo realizar todas as operações associadas ao ciclo de vida de um bot - como por exemplo, criar bot, remover bot, locomover um bot, utilizar e selecionar armas, etc. Esta interface encontra-se no apêndice B. Em seguida foi desenvolvida uma arquitetura de controle em linguagem Java, com o propósito de testes e de prova de conceito. Como podemos ver na Figura 5.5, a arquitetura desse sistema divide-se em três subsistemas:

- O subsistema servidor, contendo aproximadamente 16000 linhas de código, (muitas delas adaptadas a partir de [Bot]), implementado em C/C++, é responsável pela interface direta com o motor do jogo.
- O susbsistema de mensagens, implementado em C++, contendo aproximadamente 1000 linhas de código, é responsável por isolar o bot Java do subsistema servidor, permitindo a interação assíncrona entre ambos.
- O subsistema do bot, implementado em Java. Uma versão protótipo foi implementada, contendo aproximadamente 1500 linhas de código.

Posteriormente, tentamos substituir o subsistema bot em um experimento simples integrando o SNToolkit com a infraestrutura descrita acima (substituindo o subsistema do bot), somente para testes. Esta integração no entanto foi abandonada, pois diversos problemas foram detectados - problemas estes que serão explicitados mais adiante na Seção 5.4.2.

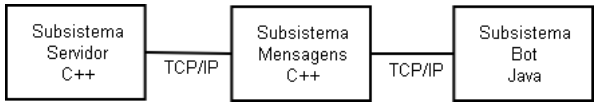

Figura 5.5: Arquitetura do sistema.

Subsistema Servidor

O subsistema servidor é responsável por realizar a interface dos outros subsistemas com o motor do jogo Counter-Strike. Este subsistema é a DLL acessada pelo motor do jogo através do uso de um conjunto de chamadas bem definidas, permitindo que as operações básicas relativas aos bots sejam realizadas.

Subsistema de Mensagens

O subsistema de mensagens é responsável por realizar a troca de mensagens entre os subsistemas servidor e bot. Isto é realizado através de um mecanismo simples de troca de mensagens, de forma assíncrona. De forma genérica, cada cliente do serviço é alocado um ID e dado uma lista de mensagens recebidas. Um cliente pode assim enviar mensagens para outros clientes ou ainda obter as mensagens para ele enviadas, armazenadas em um mecanismo de fila FIFO (*First-In First-Out*). É interessante notar que em ambos os clientes usados (nos outros dois subsistemas) a comunicação com o subsistema de mensagens é abstraída através da utilização de classes de comunicação genéricas, no caso a classe *SocketsMessenger* que possuia uma implementação tanto em C++ quanto Java. Esta classe encapsula assim o protocolo de comunicação, acrescentando as informações necessárias para que a mensagem seja corretamente enviada e lida pelo destinatário, de certa forma acrescentando mais uma camada à pilha de protocolos TCP/IP.

As mensagens enviadas ao subsistema de mensagens são simples *strings* de caracteres, sendo os parâmetros da mensagem delimitados por um caracter arbitrário (atualmente é utilizado espaço em branco). Os tipos de mensagens que são aceitos são:

- **Conectar**: conecta um novo cliente no sistema, gerando um novo ID e criando uma nova fila de mensagens.
- **Desconectar**: desconecta um cliente de ID dado, removendo sua fila de mensagens.

- **Postar mensagem**: posta uma mensagem para um cliente de ID fornecido.
- **Obter mensagem**: obtém a primeira mensagem do topo da fila de mensagens do cliente de ID dado.

Cabe notar ainda que o subsistema de mensagens foi implementado na linguagem C++ utilizando-se de sockets TCP [Ste90] em arranjo *multithreaded*.

Subsistema do Bot

O subsistema do bot é responsável por realizar o controle do bot. Foi implementado em Java, seguindo a arquitetura proposta em [Blu96]. Não implementamos o mecanismo de aprendizagem e os comportamentos utilizados constituem o suficiente para demonstrar a validade da arquitetura do sistema como um todo. Algumas modificações foram acrescentadas no entanto à arquitetura original:

- O conceito de variáveis internas foi complementado pelo conceito de uma "corrente sanguínea" composta por diversos elementos que podem interagir entre si, de forma similar ao que ocorre em [Gra97]. Isso dá uma maior flexibilidade ao sistema, tornando procedimentos comuns mais fáceis de serem implementados, e mais intuitivos. Como por exemplo, um elemento indicando a "fome" pode reagir com outro elemento produzido com a ingestão de um alimento, ou por exemplo o "nível de stress" de uma criatura tende a um nível padrão com o tempo.
- Pequenas modificações foram realizadas devido a restrições de projeto e implementação. Estas no entanto não impactam de forma significativa o funcionamento do subsistema.

A seguir analisamos individualmente cada um dos pacotes Java (*packages*) que compõem o subsistema.

- Pacote *messaging*: responsável por encapsular o serviço de mensagens, contendo somente a classe *messaging.SocketsMessenger*.

- Pacote *client*: pacote que utiliza o pacote *messaging* para abstrair a comunicação com o servidor Counter-Strike, abstraindo os comandos específicos ao jogo (como por exemplo mover bot, atirar, etc). Possui as seguintes classes principais:

  - *client.base.CSClient*: Principal classe responsável por abstrair a comunicação com o servidor, gerenciando um só bot. Possui métodos para conectar e desconectar ao servidor, criar uma conta no servidor de mensagens, e criar, remover e utilizar os sensores e atuadores do bot (por exemplo, correr, virar, atirar, obter inimigos, gerenciar waypoints, etc).
  - *client.app.CommandLineApp*: Classe que provê um aplicativo de linha de comando a fim de testar todas as funcionalidades da classe *client.base.CSClient*. Permite o uso de todas as funcionalidades de um bot.

- Pacote *chemistry*: contém as classes responsáveis por gerenciar a química interna do bot, permitindo o gerenciamento de um número arbitrário de elementos e decaimento linear automático. Essa química é o cerne do controle "emocional" do bot, permitindo ao mesmo aparentar emoções como medo, raiva, etc.

- Pacote *bot*: pacote responsável por realizar o controle do bot. Conforme mencionado anteriormente, implementa parcialmente o modelo descrito em [Blu96]. Suas classes mais importantes são:

  - *bot.Behavior*: Classe que representa um comportamento, um dos principais elementos do controle do bot. Um comportamento é caracterizado por estar associado a mecanismos de disparo (MD[1]) (que determinam o valor de saída do comportamento), a variáveis internas (tanto para entrada quanto a serem modificadas), a atuadores, e por possuir um grupo de comportamentos (GC) filho deste comportamento. O valor de saída de um comportamento é determinado da seguinte forma:

[1] Relembrando, um MD (conforme descrito no Capítulo 3), é um mecanismo responsável por determinar a relevância de um comportamento dadas as informações do meio ambiente, por exemplo a relevância de um bot inimigo a uma determinada distância.

1. O valor de todos os MDs é obtido somando-os em $\sum MD$.
2. Requisita ao MD de maior valor que armazene seu pronome.
3. Obtém os valores das variáveis internas (de *CirculatorySystem)* relevantes, somando-os em $\sum VI$
4. Combina $\sum MD$ e $\sum VI$ para obter o valor de saída. Esta combinação pode ser atualmente tanto uma soma quanto uma multiplicação.

Já o disparo de um comportamento implica na realização dos seguintes passos, em ordem:

1. Atualização das variáveis internas.
2. Disparo dos atuadores.
3. Disparo do próximo GC associado a este comportamento.

– *bot.BehaviorGroup*: Classe que representa um grupo de comportamentos (GC). Um GC armazena um conjunto de comportamentos, dentre os quais durante um ciclo de execução um (ou nenhum, caso nenhum comportamento atinja um valor mínimo) poderá ser disparado. Quando um GC é disparado, as seguintes ações são executadas:

1. Os valores de todos os comportamentos são obtidos.
2. Para o comportamento de maior valor, se o valor for maior que um valor mínimo arbitrário, dispara o mesmo. Caso contrário, nada é feito.

– *bot.BotController*: Classe que estende *Controller*, realizando o ajuste do ambiente, a "construção" do bot e sua execução. Especificamente a arquitetura implementada contém:

* Dois GCs, cada um com:
  · Comportamento *Attack* behavior.
  · Comportamentos *MoveTo* e *ShootAt*.
* MDs para os comportamentos acima.
* Um sensor de jogadores (*bot.sensors.PlayerSensor*).
* Três atuadores, *Move, Turn, Shoot*.

  - Um *ChemicalSystem.*

– *bot.RMOperator*: Classe que armazena constantes que determinam o tipo de operador a ser usado em um mecanismo de disparo (MD). Possui as seguintes variáveis:

  - *ANY*: O MD disparará se qualquer um dos requisitos for verdadeiro.
  - *ALL*: O MD disparará se todos os requisitos forem verdadeiros.
  - *ONE*: O MD disparará se e somente se um dos requisitos for verdadeiro.
  - *ALWAYS*: O MD disparará sempre.

– *bot.ReleasingMechanism*

  Classe que representa um mecanismo de disparo (MD). Juntamente com *bot.Behavior* representa o cerne do sistema de controle do bot. O processo de obtenção do valor de saída de um MD segue os seguintes passos:

  1. Determinar se este MD deve disparar. Isto é feito verificando se a entrada associada ao MD possui um *InputObject*:
     (a) Em caso negativo, o MD não disparará.
     (b) Caso contrário, os atributos pertinentes (dependendo do tipo de operador determinado em *bot.RMOperator*) são verificados. Se satisfeitos, o MD disparará. Caso contrário o MD não será disparado.
  2. Caso o mecanismo deva disparar, a saída para este MD é calculada usando-se a função de filtro atribuída ao mesmo.

– *bot.LinearFilterFunction*: Classe que implementa uma função de filtro do tipo:

$$f(x) = \begin{cases} 0 & (x \leq a_0) ou (x \geq a_2) \\ \begin{cases} \frac{x-a_0}{a_1-a_0} & a_0 \leq x \leq a_1 \\ \frac{a_2-x}{a_2-a_1} & a_1 \leq x \leq a_2 \end{cases} \end{cases} \tag{5.1}$$

  Aonde x é o valor de entrada proveniente do MD, e a_0, a_1 e a_2 valores determinados pelo MD.

– *bot.StepFilterFunction*: Classe que implementa uma função de filtro do tipo:

$$f(x) = \begin{cases} 0 & (x < a_0) ou (x \geq a_1) \\ 1 & a_0 \leq x < a_1 \end{cases} \tag{5.2}$$

Aonde x é o valor de entrada proveniente do MD, e a_0 e a_1 valores determinados pelo MD.

– *bot.Pronome*: Classe que representa um pronome, responsável por armazenar um InputObject que foi o último foco de atenção. Implementa *bot.Sensor*, funcionando assim como um "sensor interno" do bot.

O diagrama de classes para este pacote pode ser vista nas Figuras 5.6, 5.7, 5.8 e 5.9.

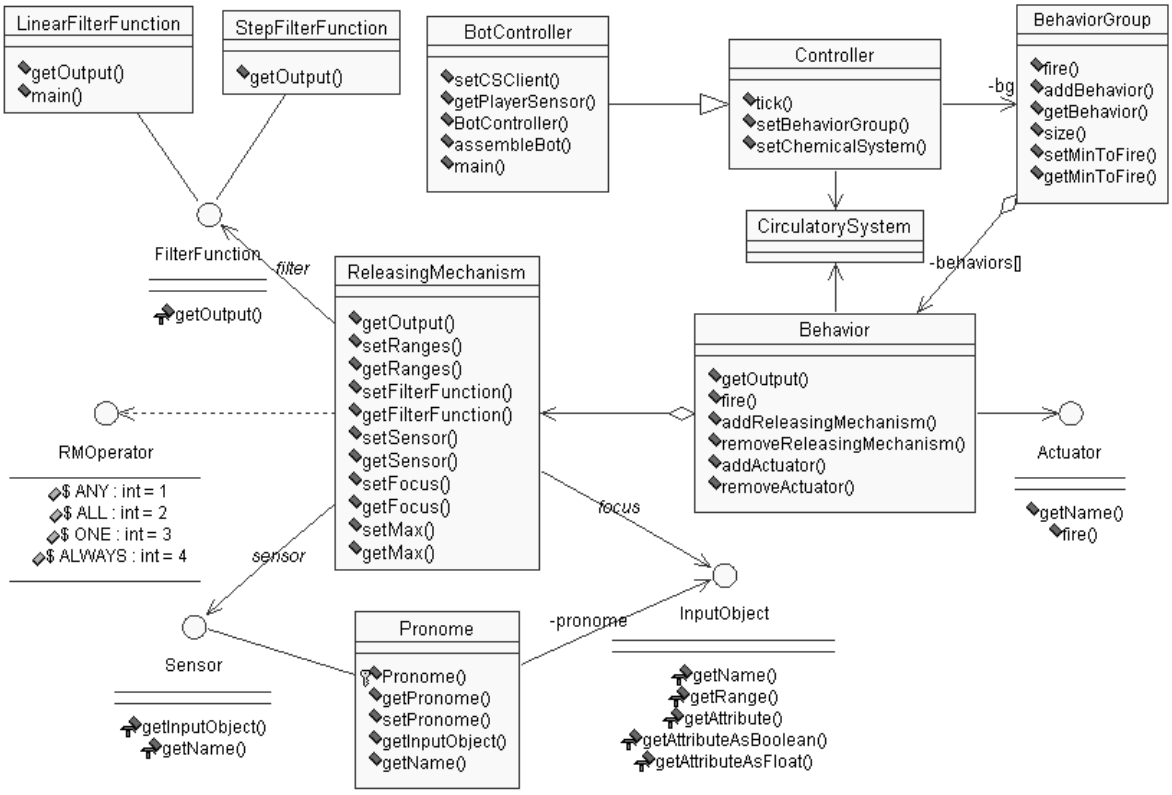

Figura 5.6: Diagrama de classes UML para as principais classes do pacote.

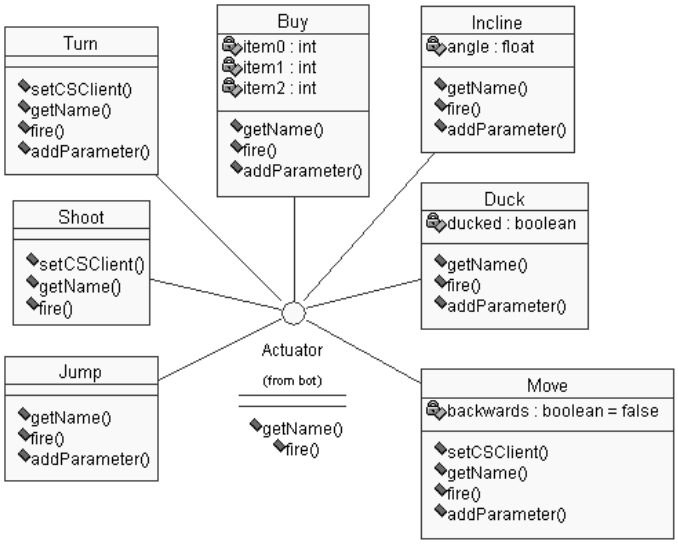

Figura 5.7: Diagrama de classes UML para os atuadores.

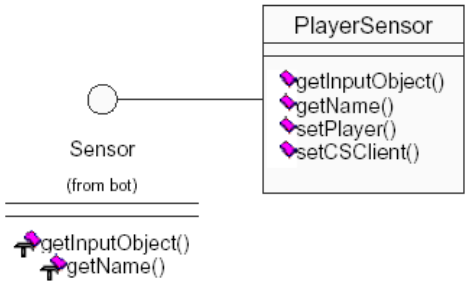

Figura 5.8: Diagrama de classes UML para os sensores.

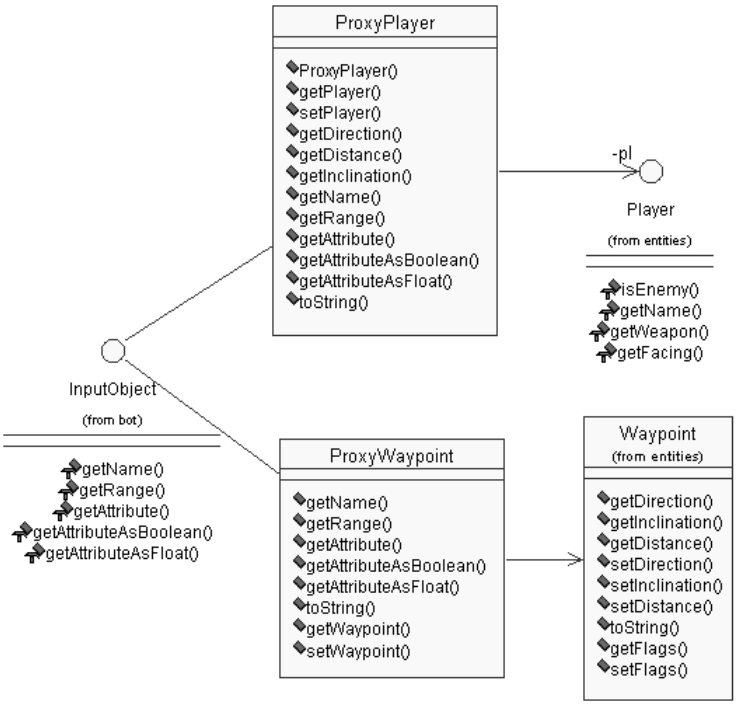

Figura 5.9: Diagrama de classes UML para as entidades do bot.

Esta primeira arquitetura, apesar de extremamente modular, apresentou uma série de dificuldades de ordem prática o que levou a implementação de uma segunda arquitetura detalhada a seguir.

5.4.2 Segunda Arquitetura

Completada a implementação da primeira arquitetura descrita anteriormente, diversos problemas foram detectados, que impediam nossa pretensão de utilizar o SNToolkit para o controle dos bots. Dentre outros, os seguintes problemas foram levantados:

- **Desempenho insatisfatório**: dois fatores contribuíram para isso:
  - O núcleo do jogo CS (de código fechado) impõe sérias restrições ao acesso de recursos externos por DLL´s, entre elas o fechamento de conexões de rede abertas. Tal fato impacta em muito o desempenho da arquitetura, uma vez que a cada passo de execução do jogo precisamos abrir uma nova conexão com o servidor.
  - O núcleo operacional do SNToolkit tem uma eficiência questionável para aplicações de tempo real, utilizando soluções de código pouco eficientes (tais como o uso excessivo de *threads* e de outras técnicas que fazem um uso despreocupado de recursos computacionais), o que limita seu uso em aplicações críticas em relação ao tempo, tais como jogos de computadores.

- **Dificuldades na edição de código**: O suporte do SNToolkit a edição de código é deficiente sob diversos aspectos, não provendo facilidades comumente encontradas em outras IDEs (*Integrated Development Environment*, ambiente de desenvolviment integrado), como identação, completamento automático de código, colorização, teclas de atalho, ambiente integrado de edição-compilação-depuração (para o código Java), dentre outras.

- **Restrições na compilação e depuração**: O mecanismo de compilação e depuração de código do SNToolkit possui uma série de restrições, não permitindo por exemplo o uso de ferramentas tradicionais de compilação e depuração pelo fato de existir um gerador

de código responsável por gerar o código Java a partir da rede modelada visualmente (conforme detalhado no Capítulo 4). Além disso, apesar do ambiente permitir a visualização da rede e execução passo-a-passo, não permite a inserção de breakpoints no interior do código Java editado, nem o acompanhamento instrução a instrução da execução do código Java.

É importante notar que sem dúvida o SNToolkit é uma ferramenta poderosa no que concerne a visualização e edição da arquitetura de alto nível do sistema, podendo também muito bem ser utilizada como uma ferramenta de integração de sistemas. Na prática, entretanto, a ferramenta mostrou uma séria deficiência para o desenvolvimento de projetos com volume razoável de código a ser implementado.

Detectadas estas deficiências da arquitetura utilizada, optamos por uma total reimplementação do motor do SNToolkit, o SNE. Partimos assim para a análise, projeto e implementação de uma nova versão do SNToolkit, agora já utilizando o conceito de RP-Nets em substituição ao conceito de Redes Semiônicas utilizadas no SNToolkit. A esta nova plataforma de desenvolvimento (que implementa os conceitos das RP-Nets) denominamos de **RPN-Toolkit**. Os principais requisitos para esta ferramenta são seu uso em aplicações críticas em relação ao desempenho, e maior facilidade no processo de compilação e depuração, permitindo o uso da IDE já utilizada no desenvolvimento da DLL do bot (Visual C++). Além disso esta ferramenta visou integrar novos requisitos particulares às RP-Nets não presentes em seu antecessor, as redes semiônicas. Cabe notar ainda que dentre os requisitos desta ferramenta não se incluem nem a interface para o projeto de modelos de RP-Nets (SNDesigner) nem o módulo de depuração visual (SNDebugger) uma vez que a reimplementação de ambos em C++ implicaria em um esforço muito grande de desenvolvimento e estaria fora das necessidades deste trabalho. Assim sendo a ferramenta SNToolkit continuou a ser utilizada para o projeto da nova arquitetura de controle do bot, não se prestando no entanto ao controle do sistema em si.

O trabalho foi organizado então em duas fases distintas: em uma primeira etapa, realizamos o projeto, implementação e testes do RPNToolkit, e em uma segunda etapa, realizamos a integração do RPNToolkit ao motor do jogo CS, procedendo ao projeto, implementação e testes da nova arquitetura de controle. Analisaremos nas subseções a seguir o

resultado de cada uma dessas etapas. Os processos de projeto e modelagem para o sistema em C++ foram realizados utilizando-se da ferramenta CASE Rational Rose 2000, e a IDE utilizada para edição e compilação de código C++ foi o Visual C++ 6.0. A plataforma de desenvolvimento foi Windows 2000 Professional. Para a modelagem visual de RP-Nets foi utilizado o SNToolkit 3. Cabe aqui notar que apesar da plataforma de desenvolvimento ter sido a plataforma Win32, a linguagem utilizada foi o C++ padrão juntamente a bibliotecas de uso disseminado (STL [Str97]) o que torna o código facilmente portável para outras plataformas, como por exemplo Linux. O desenvolvimento total implicou na implementação de aproximadamente 15.300 linhas de código.

O RPN-Toolkit

A implementação do RPN-Toolkit foi realizada na linguagem C++ e implicou no desenvolvimento de cerca de 2300 linhas de código distribuídas em aproximadamente 26 classes (podemos visualizar um diagrama UML contendo as principais classes do sistema na Figura 5.10). A fim de facilitar a integração e distribuição do RPNToolkit o mesmo foi empacotado no formato de uma biblioteca de ligação estática Windows.

O projeto do RPNToolkit inspirou-se no projeto do SNToolkit [Gom00, Gue00]. É interessante observar que apesar do novo sistema diferentemente do SNToolkit não possuir interface gráfica, a nova ferramenta implementada apresenta diversas facilidades em relação ao mesmo, tais como: - desempenho computacional superior; - maior flexibilidade no ciclo de edição - compilação - depuração de código; - mais fácil integração com sistemas externos, acessando nativamente C e C++. Tais vantagens advém do fato que o RPN-Toolkit foi projetado desde sua concepção não como uma plataforma integrada de desenvolvimento, mas como um framework / ferramenta genérica para ser integrada em outros sistemas.

O cerne da implementação do RPNToolkit é constituído pelas classes *Engine* e *BMSASolver*. BMSASolver é responsável por implementar o algoritmo de resolução de conflitos BMSA (*Best-Match Search Algorithm*) descrito inicialmente em [GGG99]. Este algoritmo é o responsável pela resolução dos conflitos resultantes da geração das diversas ações dos recursos ativos, conforme descrito no Capítulo 4. A seguir apresentamos o funcionamento em linhas gerais do algoritmo.

O algoritmo BMSA é responsável por, dentre um conjunto de ações possíveis de serem

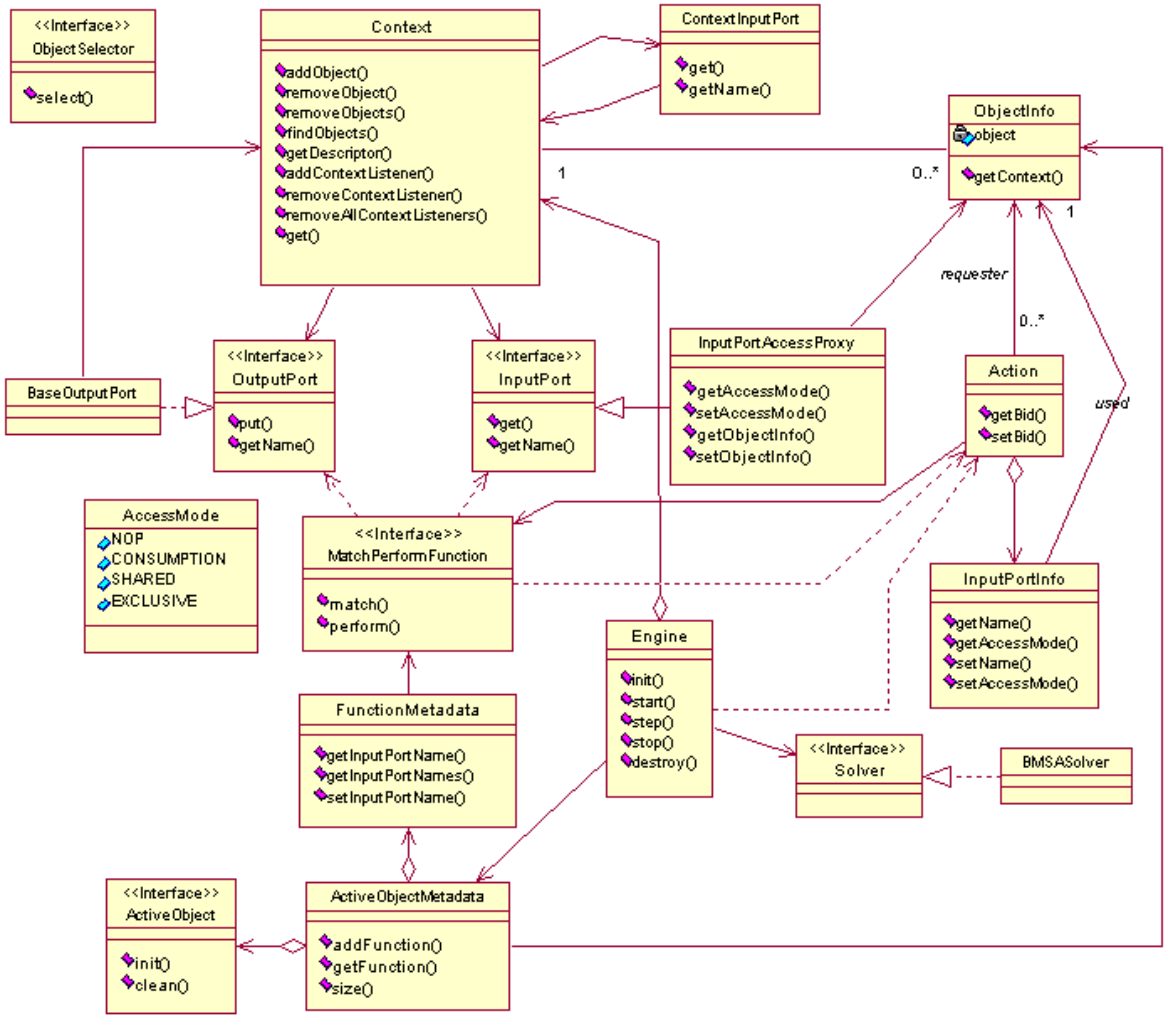

Figura 5.10: Diagrama de classes UML para o RPNToolkit.

executadas por todos os recursos ativos, de escolher aquelas que podem ser executadas tendo como base o grau de utilidade da função. O algoritmo funciona resumidamente da seguinte forma:

1. Dentre todas as ações passíveis de serem executadas, a ação de maior utilidade é escolhida para execução; caso existam várias ações com mesmo grau de utilidade, uma delas é escolhida de forma aleatória. Esta ação é retirada do conjunto de ações a serem selecionadas e passa ao conjunto de ações a serem executadas.

2. Para esta ação escolhida, todo o conjunto restante é verificado para se avaliar a viabilidade de coexistência entre esta e as outras ações. Uma ação pode coexistir com outra caso não correspondam a uma mesma função, e, caso acessem um mesmo recurso, o acesso por parte de ambas seja do tipo compartilhado.

3. O algoritmo retorna ao passo 1 até que não existam mais ações a serem selecionadas.

Uma particularidade deste algoritmo é que nunca há a necessidade de se realizar *backtracking* (retrocesso), ou seja, de se realizar um retorno a um estado prévio.

Voltando nossa atenção à classe *Engine*, a mesma é responsável por amalgamar todas as estruturas que compõem uma RP-Net, controlando todo o ciclo de execução. Este ciclo consiste em gerar ações, selecioná-las (através da classe *BMSASolver*) e executá-las.

Cabe aqui realizarmos algumas observações pertinentes à implementação do RPNToolkit, tanto como forma de facilitar a implementação como a incorporar novos conceitos das RP-Nets:

- O engine não se utiliza de um gerador de código, como no SNToolkit. A implementação de uma rede particular se dá diretamente em código C++, utilizando-se para tal da classe *EngineFacade*, que provê um conjunto de métodos que facilitam tanto o processo de criação da rede (por exemplo, auxiliando na criação de recursos passivos e ativos, lugares e funções) quanto o processo de execução da rede. Isso facilita o processo de implementação e depuração integrada a um sistema externo.

- Um lugar (correspondente à classe *Context*) é na realidade um container genérico de recursos, não sendo associado a nenhuma classe específica. Isso permite uma maior

flexibilidade ao sistema, não sendo necessário que uma classe seja "importada" ao RPNToolkit para que seja utilizada.

- O estado interno de um recurso, antes rigidamente controlado pelo SNE (antecessor do RPNToolkit), é completamente dissociado do motor do RPNToolkit. Isso provê maior facilidade e flexibilidade na implementação dos recursos ativos, uma vez que seus estados não precisam ser previamente declarados, podendo mudar arbitrariamente com o tempo.

Arquitetura de Controle

A arquitetura de controle implementada consiste em uma RP-Net constituída por 35 lugares (Figura 5.13), sendo que metade dos mesmos possui recursos ativos. O modelo da rede implementada é composto por basicamente três tipos de recursos ativos: **sensores**, **atuadores** e **comportamentos**. A rede não comporta aprendizagem, sendo no entanto extremamente reativa e eficiente, possuindo comportamento razoavelmente similar a um jogador humano nas tarefas realizadas (a similitude de comportamento foi atestada apenas de forma informal junto a alguns jogadores de CS, não tendo sido devidamente aferida em testes formais). De forma também a limitar o esforço de implementação, a rede é capaz somente de executar um tipo de objetivo de CS, o de plantar bombas, o que é o caso mais típico em CS.

Resumidamente, um ciclo de execução da arquitetura ocorre da seguinte forma: inicialmente, variáveis internas e do ambiente são captadas pelos sensores. Com base nesses valores os comportamentos são responsáveis por determinar seu valor de utilidade. Os comportamentos são estruturados hierarquicamente, em uma árvore pré-determinada de comportamentos. Apenas um nó filho de um dado nó pode disparar por vez, sendo o comportamento vencedor determinado de forma transparente através do motor da RP-Net. Dessa forma os comportamentos são disparados através da hierarquia até chegarem aos nós folhas, os quais encontram-se conectados aos atuadores, responsáveis por executar as ações do bot. Uma observação importante é que a RP-Net utiliza waypoints pré-definidos para a navegação do bot e para determinar pontos para plantar bomba e vigiar bomba. O caminho através dos waypoints é determinado utilizando-se o algoritmo de Floyd-Warshall descrito no Capítulo 2.

Podemos visualizar nas Figuras 5.11 e 5.12 o bot sendo controlado pelo RPN-Toolkit.

Figura 5.11: Bot seguindo waypoint (representado pela barra azul na tela).

Figura 5.12: Bot imediatamente após armar a bomba.

É interessante notar também que a rede opera assim de forma completamente síncrona, ou seja, a cada iteração (ou a um número pré-determinado de iterações) do motor do jogo CS a rede executa um ciclo completo de sensoriamento - decisão - atuação.

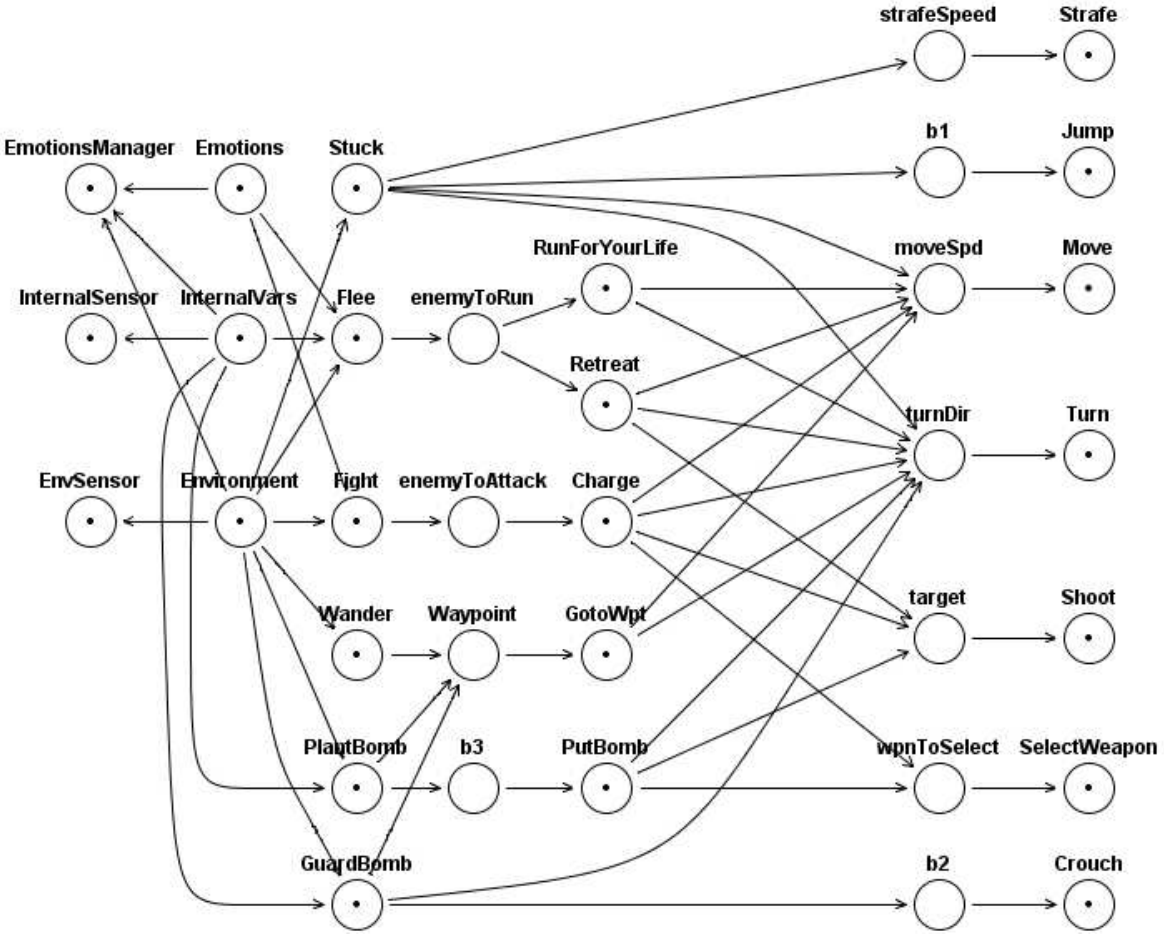

Figura 5.13: Rede usada para o controle do bot.

No cerne do mecanismo de decisão está uma estrutura baseada em **comportamentos**. A idéia de comportamentos é oriunda das ciências cognitivas e da etologia, tendo sido utilizada anteriormente em outros trabalhos como por exemplo [Mae89, KZ02, Blu96]. Um comportamento representa qualquer decisão que leve a execução de uma ação com base nos sensores e/ou estado interno, sendo que os comportamentos organizam-se de forma hierárquica com

os comportamentos de mais alto nível no topo da hierarquia, e os comportamentos mais específicos sucessivamente mais abaixo, de forma similar à hierarquia observada na Figura 3.1. Um comportamento é responsável por determinar seu valor de utilidade de acordo com suas entradas, fornecidas tanto por outros comportamentos quanto por sensores. Exemplos de comportamentos são Atacar, Fugir, Esconder-se, etc. Por serem mapeados diretamente a conceitos abstratos quotidianamente utilizados pelos seres humanos, o uso de comportamentos torna o projeto do sistema de controles extremamente intuitivo, facilitando a um projetista organizar as ações do bot de acordo com o estado do ambiente e do estado interno do bot.

De forma mais específica, cada comportamento, sensor e atuador é mapeado em recursos ativos na RP-Net de controle do bot. Esses recursos ativos são:

- **EnvSensor**: sensor responsável por obter informações do ambiente externo, provendo ao bot todas as informações necessárias a sua tomada de decisão, como jogadores amigos e inimigos no campo de visão, obstáculos, waypoints, etc.
- **InternalSensor**: sensor responsável por sensoriar o estado interno do bot, mantendo atualizados por exemplo os valores de energia, armadura, munição e armas do bot.
- **EmotionsManager**: este recurso ativo atualiza os estados emocionais do bot de acordo com o modelo descrito mais adiante. Atualmente somente uma emoção é emulada, o *medo*, mas mais emoções podem ser adicionadas facilmente se necessário.
- **Stuck**: comportamento responsável por controlar o bot em situações nas quais o mesmo se encontra preso em paredes ou em outros bots.
- **Flee**: comportamente que determina se um bot deve fugir de um (ou mais) inimigos. É determinado em parte pela emoção medo.
- **RunForYourLife**: um comportamento filho de Flee, determina que o bot deve virar as costas e fugir de seu inimigo.
- **Retreat**: também um comportamento filho de Flee, representa uma fuga ordenada do inimigo (com a face voltada para o mesmo).
- **Fight**: comportamento que determina quando o bot deve engajar-se em combate.

- **Charge**: comportamento que realiza um ataque frontal.
- **Wander**: comportamento que faz o bot andar a esmo pelo mapa.
- **GotoWpt**: comportamento que leva o bot a um determinado waypoint.
- **PlantBomb**: comportamento que determina se o bot deve procurar armar a bomba.
- **PutBomb**: comportamento que arma a bomba uma vez o bot estando na localização adequada.
- **GuardBomb**: comportamento que leva o bot a guardar a bomba uma vez a mesma tendo sido armada.
- **Strafe**: atuador que faz o bot andar lateralmente.
- **Jump**: atuador que faz o bot pular.
- **Turn**: atuador que faz o bot girar o corpo.
- **Move**: atuador que faz o bot andar para frente e para trás.
- **Shoot**: atuador que faz o bot atirar.
- **SelectWeapon**: atuador que permite ao bot selecionar uma arma (ou um item qualquer a sua disposição).
- **Crouch**: atuador que permite ao bot agachar-se.

Outro aspecto digno de nota da arquitetura utilizada é a presença de variáveis internas para denotar o estado emocional do bot. Cabe notar aqui que nosso objetivo não é o de *emular* completamente emoções, mas sim tão somente de *simulá-las* para que o comportamento do bot aparente ser mais "humano" aos olhos do jogador (demonstrando assim estados internos do bot tais como medo, curiosidade, etc).

É interessante ressaltar que o uso de "emoções" em agentes computacionais, especialmente em agentes que interagem com humanos, é cada vez maior. Isso se deve ao fato do uso de emoções contribuir para a riqueza de interações em interfaces homem-computador, e

também por pesquisas indicarem a emoção como fator essencial no processo de cognição e tomada de decisões [Dam96, Pic97]. É interessante notar que a utilização de emoções possibilita ao bot (ou a um agente qualquer em ambientes virtuais) ter maior credibilidade frente ao jogador ([Bat97]), além de poder ajudar no processo de identificação e comunicação do jogador com o jogo. Tal fato se torna patente caso analisemos por exemplo o domínio da animação tradicional, que tão bem se utiliza de emoções em seus personagens de forma a se comunicar com sua audiência e aumentar o fenômeno de *suspension of disbelief*.

Mais especificamente, neste trabalho utilizamos uma arquitetura simples para a modelagem de emoções, sendo que arquiteturas muito mais complexas existem atualmente (como por exemplo [Vel97, MF98, Slo01]). Adotamos aqui um modelo discreto de emoções (inspirado em [Ekm99] e [Pic97]), sendo cada emoção codificada na forma de um número real - mais especificamente uma variável interna ε para cada emoção. Tal variável é caracterizada por possuir um valor de descanso e saturação, taxa de decaimento e função de crescimento. Tais valores são parametrizáveis, provendo assim bots mais ou menos "covardes", mais ou menos "curiosos", mais ou menos "sociáveis" ou "obedientes", etc. Atualmente somente uma "emoção" é utilizada, o *medo*, a qual afeta a chance do bot fugir frente a um inimigo. Mais emoções podem ser adicionadas facilmente para dotar o bot de comportamentos mais interessantes.

Mais formalmente, o valor de uma emoção ε é atualizado a cada interação j de acordo com a equação 5.3. Na mesma o valor constante a_0 representa a taxa de decaimento da emoção (linear neste caso por simplicidade), o valor constante b_0 representa a taxa de aproveitamento da função de crescimento e $\beta(\gamma)$ representa a função de crescimento.

$$\varepsilon_{j+1} = \varepsilon_j - a_0 + b_0\beta(\gamma) \tag{5.3}$$

Se $\varepsilon_{j+1} < l_0$ então $\varepsilon_{j+1} \leftarrow l_0$, o valor fixo l_0 sendo o valor de descanso da emoção. Também se $\varepsilon_{j+1} > s_0$ então $\varepsilon_{j+1} \leftarrow s_0$, o valor fixo s_0 sendo o valor de saturação da emoção.

A função de crescimento pode ser vista na Função 5.4.

$$\beta(\gamma) = \frac{1}{1 + e^{-c_0\gamma + d_0}} \tag{5.4}$$

A função de crescimento caracteriza-se por ser uma função sigmoidal, tendo como en-

trada o valor γ calculado caso a caso para cada emoção. No caso específico da emoção medo, $\gamma = e_j - e_{j+1}$, na qual o valor e é o valor da variável interna que armazena o valor de energia do bot. Caso assim ocorra um decréscimo súbito grande na energia do bot (decorrente por exemplo de ele ter sido atingido por um tiro à queima-roupa) o valor do medo sobe. A Função 5.4 é regulada por um conjunto de valores constantes: c_0 indica a inclinação da curva sigmoidal; e d_0 indica o ponto de início da sigmóide, funcionando como um valor limite a partir do qual o valor de γ passa a afetar o estado emocional do bot. Na Figura 5.14 podemos visualizar a função de crescimento para a emoção medo, com valores $c_0 = 0,4$ e $d_0 = 10$.

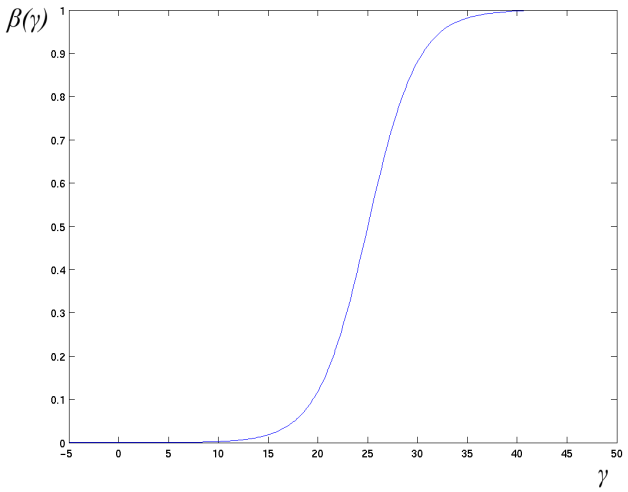

Figura 5.14: Função de crescimento para a emoção medo.

5.4.3 Resultados

A arquitetura de controle implementada mostrou-se promissora tanto no que diz respeito ao seu consumo de recursos computacionais quanto ao seu desempenho no jogo de CS. O comportamento do bot apresenta-se coerente com o ambiente e com seu estado interno, não apresentando ainda no entanto um comportamento similar ao de um jogador humano, principalmente pelo pequeno número de comportamentos implementados - mais comportamentos de ataque especialmente são necessários para tornar o bot mais “humano”. Observamos tam-

bém que o bot não está capacitado a executar todos os tipos de objetivos disponíveis, somente sendo capaz de realizar o objetivo de armar/desarmar bomba. Entretanto, tendo em vista que a infraestrutura básica encontra-se devidamente preparada, a implementação de outras funcionalidades e sua integração no sistema deve ser razoavelmente fácil.

5.5 Resumo

Este capítulo apresentou o modelo computacional utilizado para o controle de agentes em um jogo comercial, mais especificamente no jogo Half-Life Counter-Strike. O modelo aqui apresentado é uma evolução das idéias apresentadas no Capítulo 2, mas tem suas principais raízes nos modelos baseados em comportamentos apresentados no Capítulo 3, utilizando como base o conceito das RP-Nets apresentado no Capítulo 4. É um modelo que apresenta muitas das características propostas no início do capítulo, tais quais reatividade, estado interno, facilidade de compreensão e depuração (graças ao RPN-Toolkit) e mais importante, é um modelo que acrescenta ao efeito de *suspension of disbelief* tão fundamental a um jogo de computador, principalmente graças à incorporação de "emoções" junto ao mesmo.

Capítulo 6

Resultados e Trabalhos Futuros

Este trabalho teve dois objetivos principais: primeiro, realizar uma análise do estado da arte no que concerne a jogos de computador e sistemas inteligente e apresentar a utilização de novos conceitos de sistemas inteligentes em uma nova arquitetura para agentes em jogos de computador; segundo, apresentar e demonstrar a validade da ferramenta RP-Net no controle de aplicações, mais especificamente no controle de jogos de computador. A seguir detalhamos estas contribuições, apresentando também perspectivas de trabalhos futuros para as mesmas.

Neste trabalho introduziu-se uma nova ferramenta formal para sistemas de processamento de recursos, as *RP-Nets*, que se baseiam em um modelo matemático-computacional similar a uma Rede de Petri mas com algumas vantagens quando o objeto de modelagem é um sistema complexo de processamento de recursos. Dentre outras vantagens uma RP-Net explicita quais são os recursos ativos do sistema, além de permitir a modelagem de sistemas com adaptação e aprendizagem. Além disso, demonstrou-se a viabilidade do uso das RP-Nets no controle efetivo de um sistema complexo, altamente dinâmico e muito sensível em relação ao desempenho / utilização de recursos computacionais - um jogo de computador. O controle de um bot neste jogo provou ser uma plataforma desafiadora, tanto nos aspectos mais abstratos abrangendo o projeto da rede quanto em termos de esforço de implementação computacional, o que evidencia a potencialidade das RP-Nets como ferramenta genérica para o projeto de sistemas inteligentes, da mesma forma que em trabalhos anteriores a este ([Gud96, Sua00]). Detectou-se deficiências na plataforma computacional anterior-

mente disponível ao grupo (SNToolkit), sendo que uma nova ferramenta computacional, o RPNToolkit, foi desenvolvida. Esta não está vinculada ao projeto de jogos de computador, podendo vir a ser utilizada por quaisquer outros projetos com requisitos semelhantes aos encontrados neste.

As perspectivas de continuação deste trabalho são muitas. Primeiramente, podemos vislumbrar a evolução da ferramenta desenvolvida, o RPNToolkit, para que possa integrar-se ao SNToolkit, aproveitando-se do esforço de desenvolvimento aplicado sobre o mesmo. Além disso, a ferramenta não trata conceitos mais recentes presentes nas Redes Semiônicas, como as redes modulares [Gom00] ou as redes com campos [GGG98], conceitos que poderiam vir a ser implementados futuramente à plataforma RPNToolkit.

Outra contribuição deste trabalho foi a integração de um projeto acadêmico a um produto comercial. Esta integração torna-se ainda mais interessante considerando-se que o produto comercial é um jogo de computador, um tipo de integração pouco explorada tanto no âmbito nacional quanto internacional - o que atesta o pioneirismo da iniciativa. De forma mais específica, a plataforma computacional desenvolvida junto ao jogo CS pode vir a ser facilmente integrada em outros projetos que necessitem de um ambiente dinâmico e interativo como uma plataforma de testes / validação de conceitos.

Finalmente, também contribuímos para a área de jogos de computador levantando um panorama dos jogos de computador mais interessantes sob a ótica dos SI, apresentando também as técnicas mais relevantes e sua forma de utilização nos jogos. Introduzimos também uma nova arquitetura de controle para agentes autônomos que pode ser utilizada em outros jogos do mesmo gênero (podendo mesmo ser adaptada a outros gêneros de jogos, ou aplicações de entretenimento eletrônico / realidade virtual). A arquitetura aqui apresentada, se não adaptativa, é extremamente reativa, computacionalmente eficiente, de fácil projeto, compreensão e riqueza de comportamentos (fato este ainda reforçado pela arquitetura de emoções utilizada). Esta arquitetura, apesar de utilizar-se da ferramenta RP-Net, poderia também facilmente ser abstraída de forma a tornar-se um framework independente para controle de agentes em jogos de computador.

As possibilidades de trabalhos futuros subsequentes são várias. Primeiramente, não foram realizados testes mais formais da arquitetura (na forma dos bots) junto a grupos de jogadores, testes estes que poderiam prever formulários para avaliar o desempenho dos bots. Poderíamos

ainda realizar experimentos similares ao teste de Turing nos quais os jogadores seriam responsáveis por tentar distingüir em uma sessão de jogo quais as entidades que são bots e quais são jogadores humanos. Tais informações, com certeza, poderiam nortear futuros desenvolvimentos na arquitetura e levar a muitas melhorias na mesma. Além disso, podemos vislumbrar a aplicação de mecanismos adaptativos à arquitetura, primeiramente através do aprendizado dos valores-utilidade dos comportamentos, e depois com a criação de novos comportamentos, de forma similar ao que ocorre em [Blu96]. Poderíamos ir além ainda e considerarmos bots cujo comportamento inicial fosse apenas composto por comportamentos "básicos", que equivaleriam ao conteúdo filogenético do bot ([SS96]), sendo que todos os comportamentos mais complexos seriam adquiridos através de aprendizagem. Isto inevitavelmente acarretaria modificações também junto aos aspectos teóricos das RP-Nets, de forma a permitir a criação de novos lugares por recursos ativos, gerando RP-Nets adaptativas. Também podemos vislumbrar o uso de um algoritmo genético no processo de aprendizagem dos bots, codificando os comportamentos em genótipos que seriam passados através de gerações de bots vencedores.

Esperamos, com este trabalho, ter trazido uma contribuição efetiva à área de jogos de computadores bem como à área de sistemas inteligentes.

Apêndice A

Sites WWW para Jogos Referenciados

Jogo	Hiper-referência
Action Quake 2	`http://action.action-web.net/`
Asheron's Call	`http://www.microsoft.com/games/zone/asheronscall/`
Black & White	`http://bwgame.com/`
Close Combat 2	`http://www.atomic.com/`
Civilization: Call to Power	`http://www.activision.com/games/civilization/`
Cloak, Dagger and DNA	`http://www.gamesdomain.com/directd/481.html`, obtenção do *shareware*
Creatures	`http://www.creaturelabs.com/`
Doom	`http://www.idsoftware.com/`
Emperor: Battle for Dune	`http://westwood.ea.com/`
EverQuest	`http://everquest.station.sony.com/`
Falcon 4	`http://www.falcon4.com/`
Freespace 2	`http://www.freespace2.com/`

Jogo	Hiper-referência
Galapagos	http://anarkmedia.com/Galapagos/
Ground Control	http://sierrastudios.com/games/groundcontrol/
Half-Life	http://www.sierrastudios.com/games/half-life/
Half-Life Counter-Strike	http://www.counter-strike.net/
Heretic	http://www.idsoftware.com/killer/heretic.html
Hexen	http://www.idsoftware.com/killer/hexen.html
Panzer General	http://www.panzergeneral.com/
Quake 1, 2, 3	http://www.idsoftware.com/
Rainbow Six	http://www.redstorm.com/rainbow_six/
SimCity	http://simcity.ea.com/us/guide/
SimsVille	http://simsville.ea.com/
Starcraft	http://www.blizzard.com/starcraft/
Team Fortress	http://sierrastudios.com/games/tf1.5/
The Sims	http://www.thesims.com/
Tribes 2	http://sierrastudios.com/games/tribes2/
Ultima Online	http://www.uo.com/
Warcraft	http://www.blizzard.com/war1/
WarCraft 3	http://www.blizzard.com/war3/

Os jogos *Dune* 2, *El-Fish*, *Fields of Battle*, *SimEarth* e *SimLife* não possuem *sites* oficiais.

Apêndice B

Interface para Controle de Bots

Apresentamos abaixo as funções utilizadas para o controle do bot. Note que estas são as funções chamadas pelos sensores e atuadores do bot.

```
int createBot(string name, string team, string uniform);
void removeBot(int bot);
void setSpeed(int bot, float speed);
void setSidewaysSpeed(int bot, float speed);
void turn(int bot, float angles);
void pitch(int bot, float angles);
void jump(int bot);
void duck(int bot, bool status);
void shoot(int bot, bool keep = false);
void secShoot(int bot, bool keep = false);
void selectItem(int bot, string item);
void use(int bot, bool keep = false);
void reload(int bot);
void buy(int bot, string item, string subitem);
float getObstacleFront(int bot, float range, float right, float height);
float raytrace(int bot, float angle, float range, float height);
float fastRaytrace(int bot, float front, float right, float height);
float utilGetLeastTurnAngle(int bot);
RelativePosition getClosestEnemy(int bot);
RelativePosition getClosestFriend(int bot);
vector<RelativePosition> getEnemies(int bot, float range);
vector<RelativePosition> getFriends(int bot, float range);
```

```
vector<WptRelativePosition> getWaypoints(int bot, float range);
WptRelativePosition getClosestWaypoint(int bot);
WptRelativePosition getClosestObjectiveWaypoint(int bot);
void resetWaypoints(int bot);
int getHealth(int bot);
int getMoney(int bot);
int getArmor(int bot);
void sendMessage(string str);
bool hasBomb(int bot);
void plantBomb(int bot);
void addRoundStartListener(RoundStartListener* listener);
void removeRoundStartListener(RoundStartListener* listener);
```

Referências Bibliográficas

[AAA99] The 1999 aaai symposium on computer games and artificial intelligence, 1999. http://www.cs.nwu.edu/ wolff/aicg99/.

[AAA00] The 2000 aaai symposium on computer games and artificial intelligence, 2000. http://www.cs.nwu.edu/ wolff/AIIE-2000.html.

[Ass02] Interactive Digital Software Association. Essential facts about the the video game and computer industry, 2002.

[Bat97] J. Bates. The role of emotion in believable agents. *Communications of the ACM*, 37(7):122–125, 1997.

[Blu94] Bruce Blumberg. Action selection in hamsterdam: Lessons from ethology. In *Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, volume 3 of *From Animals to Animats*. MIT Press, 1994.

[Blu96] Bruce Blumberg. *Old Tricks, New Dogs: Ethology and Interactive Creatures*. PhD thesis, Massachusetts Institute of Technology, 1996.

[Bot] Botman. Botman's bots. Visitado em 30/09/2006. http://botman.planethalflife.gamespy.com/.

[Cas93] Christos G. Cassandras. *Discrete Event Systems: Modeling and Performance Analysis*. Aksen Associates Incorporated Publishers, 1993.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 2 edition, 2001.

[CN94] P. Curtis and D. Nichols. Muds grow up: Social virtual reality in the real world. In *Digest of Papers, Spring COMPCON 94*, pages 193–200. IEEE Computer Society Press, 1994.

[Dam96] Antônio Damásio. *O erro de Descartes: Emoção, Razão e o Pensamento Humano*. Companhia das Letras, 1996.

[DF98] A. Dieberg and A. U. Frank. A city metaphor for supporting navigation in complex information spaces. *Journal of Visual Languages and Computing*, pages 597–622, 1998.

[DHR97] Patrick Doyle and Barbara Hayes-Roth. Guided exploration of virtual worlds. Technical report, Knowledge Systems Laboratory, 1997.

[Ekm99] Paul Ekman. Basic emotions. In T. Dalgleish and T. Power, editors, *The Handbook of Cognition and Emotion*, pages 45–60. John Wiley & Sons, Ltd., 1999.

[Exc] Excalibur: Adaptive constraint-based agents in artificial environments. http://www.ai-center.com/projects/excalibur/.

[FTT99] John Funge, Xiaoyuan Tu, and Demetri Terzopoulos. Cognitive modeling: Knowledge, reasoning and planning for intelligent characters. In *Proceedings of the SIGGRAPH 99*, August 1999.

[Fun98] John Funge. *Making Them Behave: Cognitive Models for Computer Animation*. PhD thesis, University of Toronto, 1998.

[GC98] Stephen Grand and D. Cliff. Creatures: Entertainment software agents with artificial life. *Autonomous Agents and Multi-Agent Systems*, 1:39–57, 1998.

[GC99] Stephen Grand and D. Cliff. The creatures global digital ecosystem. *Artificial Life*, 5:77–94, 1999.

[GCM97] S. Grand, D. Cliff, and A. Malhotra. Creatures: Artificial life autonomous software agents for home entertainment. In W. Lewis Johnson, editor, *Proceedings of The First International Conference on Autonomous Agents (Agents'97)*, pages 22–29, Marina del Rey, California, USA, 1997. ACM Press.

[GGG98] R. Gonçalves, Fernando Gomide, and Ricardo Gudwin. Fielded object networks as a framework for computer intelligence. In *Proceedings of ISAS'98 - Intelligent*

Systems and Semiotics - International Conference, pages 210–214, Gaithersburg, USA, September 1998.

[GGG99] J. A. S. Guerrero, A. S. Gomes, and R. R. Gudwin. A computational tool to model intelligent systems. In *Anais do IV Simpósio Brasileiro de Automação Inteligente - SBAI'99*, pages 227–232, 1999.

[Gom00] Antônio S. R. Gomes. Contribuições ao estudo de redes de agentes. Master's thesis, DCA/FEEC/UNICAMP, June 2000.

[Gra97] Stephen Grand. Creatures: an exercise in creation. *IEEE Expert*, pages 19–24, July/August 1997.

[Gud96] Ricardo Ribeiro Gudwin. *Contribuições ao Estudo Matemático de Sistemas Inteligentes*. PhD thesis, DCA/FEEC/UNICAMP, May 1996.

[Gud02] Ricardo Ribeiro Gudwin. Semiotic synthesis and semionic networks. *S.E.E.D. Journal (Semiotics, Evolution, Energy, and Development)*, 2(2):55–83, August 2002.

[Gue00] José A. S. Guerrero. Rede de agentes - uma ferramenta para o projeto de sistemas inteligentes. Master's thesis, DCA/FEEC/UNICAMP, February 2000.

[Hay94] S. Haykin. *Neural Networks, A Comprehensive Foundation*. Macmillan, New York, NY, 1994.

[Hed97] Sara Hedberg. Smart games: beyond the deep blue horizon. *IEEE Expert*, 12(4):15–18, July 1997.

[Hef02] Katie Hefner. In an ancient game, computing's future. *The New York Times*, August 2002. http://www.nytimes.com/2002/08/01/technology/circuits/01GONE.html?8ict.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimal cost paths. *IEEE Transactions on Sustems, Science and Cybernetics*, pages 100–107, 1968. SSC-4(2).

[Hol75] John H. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, 1975.

[Jön97] F. M. Jönsson. An optimal pathfinder for vehicles in real-world digital terrain maps. 1997.

[KJV83] S. Kirpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[Kor85] R. E. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[KZ02] A. Khoo and R. Zubek. Applying inexpensive ai techniques to computer games. *IEEE Intelligent Systems Magazine*, 4(17):48–53, 2002.

[Lai00a] John E. Laird. Creating human-like synthetic characters with multiple skill levels: A case study using the soar quakebot. In *Proceedings of the AAAI 2000 Fall Symposium Series: Simulating Human Agents*, November 2000.

[Lai00b] John E. Laird. It knows what you're going to do: Adding anticipation to a quakebot. Technical report, AAAI 2000 Spring Symposium Series: Artificial Intelligence and Interactive Entertainment, 2000.

[LvL99] John E. Laird and Michael van Lent. Developing an artificial intelligence engine. In *Proceedings of the Game Developer's Conference*, pages 577–588. CMP Media LLC, March 1999.

[Mae89] Pattie Maes. How to do the right thing. *Connection Science Journal*, 1(3):291–323, 1989. Special Issue on Hybrid Systems.

[Mau94] M. Mauldin. Chatterbots, tinymuds, and the turing test: Entering the loebner prize competition. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. AAAI, 1994.

[MDBP96] Pattie Maes, T. Darrell, Bruce Blumberg, and A. Pentland. The alive system: Wireless, full-body interaction with autonomous agents. *ACM Special Issue on Multimedia and Multisensory Virtual Worlds*, 1996.

[MF98] Lee McCauley and Stan Franklin. An architecture for emotion, 1998.

[Mur89] Tomohiro Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Nil80] N. Nilsson. *Principles of artificial intelligence*. Tioga, Palo Alto, CA, 1980.

[OZ] Cmu scs oz project homepage. http://128.2.242.152/afs/cs.cmu.edu/project/oz/web/oz.html.

[PG98] Witold Pedrycz and Fernando Gomide. *An Introduction to Fuzzy Sets: Analysis and Design*. MIT Press, 1998.

[Pic97] Rosalind W. Picard. *Affective Computing*. MIT Press, 1997.

[Pol] Jordan B Pollack. Demo: Dynamical and evolutionary machine organization. http://www.demo.cs.brandeis.edu/index.html.

[Rei94] Elizabeth M. Reid. Cultural formation in text-based virtual realities. Master's thesis, University of Melbourne, January 1994.

[Rey87] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. In *Proceedings of the SIGGRAPH 87*, pages 25–34, 1987.

[RG92] A. Rao and M. Georgeff. An abstract architecture for rational agents. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 1992.

[RN95] S. J. Russell and Peter Norvig. *Artificial Intelligence – A Modern Approach*. Prentice Hall, 1995.

[Rus92] S. J. Russell. Efficient memory-bounded search methods. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 1–5, 1992.

[SLLB96] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook: The man-machine world checkers champion. *AI Magazine*, 17(1):21–29, 1996.

[Slo01] Aaron Sloman. Beyond shallow models of emotion. In *Cognitive Processing*, volume 2, pages 177–198, 2001.

[SP97] Jonathan Schaeffer and Aske Plaat. Kasparov versus deep blue: the re-match. *ICCA Journal*, 20(2):95–102, 1997.

[SS96] Lee Spector and Kilian Stoffel. Ontogenetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 394–399, Stanford University, CA, USA, 1996. MIT Press.

[Ste90] W. Richard Stevens. *Unix Network Programming*. Prentice Hall, primeira edition, 1990.

[Ste94] Anthony Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, terceira edition, July 1997.

[Sua00] Lizet Liñero Suarez. Conhecimento sensorial - uma análise segundo a perspectiva da semiótica computacional. Master's thesis, DCA-FEEC-UNICAMP, December 2000.

[Teaa] The CS Team. Counter-strike manual. http://www.counter-strike.net/manual.html.

[Teab] The CS Team. The official counter-strike web site. http://www.counter-strike.net/.

[TJ81] Frank Thomas and Ollie Johnson. *the illusion of life*. Hyperion, 1981.

[TT94] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of ACM SIGGRAPH 94*, pages 43–50, 1994.

[Tur95] Sherry Turkle. *Life on the Screen: Identity in the Age of the Internet*. Simon and Schuster, 1995.

[Vel97] Juan D. Velásquez. Modeling emotions and other motivations in synthetic agents. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, Providence, RI, 1997. American Association for Artificial Intelligence, AAAI/MIT Press.

[Wol82] Stephen Wolfram. Cellular automata as simple self-organizing systems. Technical report, Caltech, 1982.

[Woo] Steve M Woodcock. The game ai page. Visitado em 30/09/2006. http://www.gameai.com/.

[ZIR94] Jacek M. Zurada, Robert J. Marks II, and Charles J. Robinson. *Computational intelligence: imitating life*. IEEE Press, 1994.

[Zob69] A. L. Zobrist. A model of visual organisation for the game go. In *Proceedings of the Spring Joint Computer Conference*, volume 34, pages 103–112, 1969.

[Zub00] Fernando J. Von Zuben. Computação evolutiva: Uma abordagem pragmática. In *Anais da I Jornada de Estudos em Computação de Piracicaba e Região (1a JECOMP)*, volume 1, pages 25–45, 2000.

Índice Remissivo