Inteligência Artificial para Jogos: Agentes especiais com permissão para matar... e raciocinar!

Fernando Osório - Gustavo Pessin - Sandro Ferreira - Vinícius Nonnenmacher

UNISINOS - Universidade do Vale do Rios dos Sinos
PIPCA – PPG de Computação Aplicada, GT-JEDi – Graduação em Jogos Digitais
São Leopoldo, RS - Brasil

Resumo

Este tutorial visa apresentar conceitos e técnicas relacionadas com a área de Desenvolvimento de Jogos e Entretenimento Digital, focando na implementação de Agentes Inteligentes (NPCIs – Non-Player Intelligent Characters). O artigo irá abordar temas que vão da aplicação de técnicas clássicas de I.A. para jogos (e.g. busca em espaço de estados, algoritmo Mini-Max) [Russell 1995], para o planejamento de trajetórias de agentes autônomos (e.g. A*, Grafo de Visibilidade, Campos Potenciais) [Russell 1995, Dudek 2000, Rabin 2002], e arquiteturas de controle de agentes autônomos individuais (e.g. flocks/boids [Reynolds 2007], arquitetura reativa, arquitetura deliberativa, arquitetura híbrida, BDI [Osório 2004, 2005, 2006]), assim como mecanismo de controle com comunicação e cooperação entre múltiplos agentes (MAS – Multi-Agent Systems). Também serão abordadas técnicas de aprendizado de máquina (ML – Machine Learning) [Mitchell 1997, Rezende 2003] e evolutivas (GA - Genetic Algorithms) [Mitchell 1996], voltadas para a evolução e aperfeiçoamento de estratégias inteligentes em agentes (NPICs) [Rabin 2002, Pessin 2007, 2007b]. Concluiremos discutindo sobre os recursos necessários de serem oferecidos em uma engine de I.A. focada no desenvolvimento de jogos, e sobre as ferramentas atualmente disponíveis no mercado nesta área.

Keywords: Inteligência Artificial, NPCs e NPCIs, Arquiteturas de Controle de Agentes, Agentes Autônomos, Agentes Inteligentes, Algoritmos de busca de caminhos (*pathfinding*), Agentes Cooperativos, Sistemas Multi-Agentes,

Authors' contact:

```
fosorio <AT> unisinos <DOT> BR,
{ fosorio, pessin sandro.s.ferreira,
vnonnenmacher } <AT> gmail <DOT> com
```

1. Introdução

Em jogos digitais, o uso de técnicas de Inteligência Artificial (I.A.) vem se fazendo cada vez mais necessário, e sendo até mesmo indispensável em certos casos. Estas técnicas podem ser aplicadas em diferentes tipos de tarefas: resolução de problemas (*puzzles*, *brain-teasers*), planejamento de tarefas, planejamento de trajetórias, controle de agentes autônomos (NPCs – *Non-Player Characters*), entre outras diversas aplicações. Para que possamos integrar técnicas de I.A. em jogos digitais, se faz necessária a inclusão de um novo componente em um motor de jogo (*engine*), além dos já tradicionais componentes destas Engines de Jogos: motor gráficos, de áudio, de rede, de física, de controle de dispositivos de E/S (interação). Este componente é o motor de Inteligência Artificial – I.A [Bittencourt 2006].

Portanto, os motores de jogos usualmente necessitam oferecer funcionalidades relacionadas ao uso de técnicas de Inteligência Artificial (I.A.), onde existe um grande número de possíveis implementações que podem ser disponibilizadas aos desenvolvedores. Focaremos nossa discussão neste tutorial em alguns tópicos mais específicos, sendo relacionados principalmente as seguintes técnicas: I.A. clássica para jogos de tabuleiro, I.A. aplicada na busca de caminhos (path finding), I.A. para o controle de personagens autônomos (agentes) tipo NPCs (non-player characters), aprendizado de máquina em jogos, e sistemas Multi-Agentes. Para aqueles que estiverem interessados em se aprofundar ainda mais na aplicação e uso de técnicas de Inteligência Artificial em jogos, sugerimos a leitura das seguintes obras AI Game Programming Wisdom [Rabin 2002, 2003], AI Game Engine Programming [Schwab 2004], AI: A modern approach [Russell 1995], Fly3D Engine [Watt 2001], AI4Games [Funge 2006] e a série Game Programming Gems[Charles-River 2006].

2. I.A. Clássica: Solução de Problemas

A I.A. clássica busca resolver problemas complexos, onde estas técnicas clássicas se baseiam usualmente em realizar uma busca no espaço de estados até encontrar a solução.

2.1. Jogos e Busca em Espaço de Estados

Uma busca de uma solução para um problema consiste basicamente em dada uma certa configuração atual (estado atual), determinar quais são as opções de ações que posso executar até chegar a uma solução. As diferentes configurações (estados) que posso alcançar a partir do da configuração inicial definem o chamado espaço de estados, ou espaço de busca. Exemplos de problemas onde posso aplicar este tipo de técnicas de busca de soluções no espaço de estados são aqueles jogos do tipo "*Puzzles*" ou "*Brain Teasers*" (Quebra-cabeças), onde podemos citar como exemplos: Resta-um, Torre de Hanói, Cubo de Rubik, Sudoku (Fig. 1).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 1: Jogo SUDOKU [Sudoku 2007]

Esta busca pode ser realizada através da "força bruta", ou, utilizando-se de "heurísticas" que visam otimizar o processo de busca no espaço de estados. A força bruta é quando realizamos uma busca cega, ou seja, sem qualquer informação a priori que nos auxilie na exploração dos estados que podemos assumir até encontrar a solução. Usualmente são empregados os métodos "*Depth-First*" ou "*Breadth-First*" para a busca da solução [Russel 1995]. No caso do exemplo da figura 1, a força bruta consiste em testar todas as combinações de possíveis arranjos dos valores de 1 a 9, dispostos nas casas que restam em aberto no tabuleiro, verificando sempre se alguma destas combinações é um tabuleiro que respeita a regra de tabuleiro correto e, portanto, é o vencedor do Sudoku.

Na busca heurística passamos a usar certas informações a priori que limitam e orientam a busca de uma solução. No caso do exemplo da figura 1, existem algumas regras heurísticas que permitem reduzir a complexidade da busca, como por exemplo: testar apenas com os números que ainda são válidos (não podem se repetir em uma mesma célula, linha ou coluna), buscar preencher primeiro as células mais completas, entre outras técnicas.

2.2. Jogos de Tabuleiro com Adversário

O uso de Agentes Inteligentes para resolver quebra-cabeças pode ser interessante, porém usualmente os Agentes Inteligentes serão aplicados em jogos "mais nobres", que envolvam adversários. Portanto os Agentes podem jogar jogos de tabuleiro, realizando uma busca no espaço de estados, mas no caso de jogos com adversários devem considerar a participação do oponente.

Estes jogos de tabuleiro colocando um computador (Agente Inteligente) contra um adversário humano vem há muito tempo servindo como um excelente desafio para a reprodução da Inteligência em máquinas. Os jogos de tabuleiro com adversário típicos são: jogo de Xadrez, jogo de Damas, jogo da Velha (*Tic-Tac-Toe*), Othello (ou Reversi), Go, Gamão, entre outros. Estes jogos apresentam um grande desafio, pois usualmente existe um grande número de jogadas possíveis a explorar, e deste modo devemos a empregar heurísticas que permitam uma redução do espaço de configurações possíveis e buscar as melhores jogadas a fazer (aquelas que levam a uma vitória). A Figura 2 apresenta um esquema (parcial) das possíveis jogadas em um jogo da velha, onde devemos aplicar técnicas que conduzam o jogo (e o oponente) em direção à configurações do tabuleiro que sejam favoráveis em relação ao oponente.

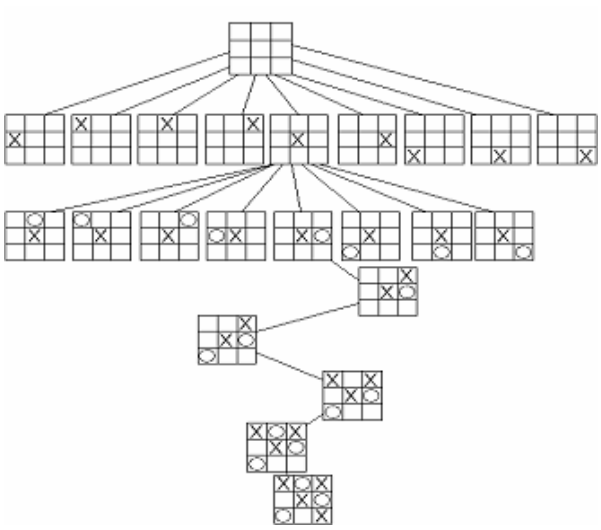


Figura 2: Árvore de estados (configurações) do tabuleiro no Jogo da Velha

No caso do jogo da velha (ou no jogo de xadrez), o que fazemos é computar um valor em relação à situação do jogo nas folhas da árvore formada por todas as possíveis jogadas (ou pelas 'n' jogadas a partir de uma certa configuração). Baseado na avaliação da situação nas folhas desta árvore (neste caso, se ganhou, perdeu ou empatou), aplicamos um algoritmo denominado Mini-Max [Russell 95, Mini-Max 2007] que irá permitir a escolha das jogadas (caminho) que com maior chance no levará a uma vitória, mesmo tendo como oponente um jogador cuja ação não pode ser totalmente prevista. O algoritmo Mini-Max [Russell 95] é conhecido como uma heurística de busca condicionada (com adversário) em espaço de estados, e no caso do jogo da velha garante com 100% de certeza um empate ou uma vitória.

2.3. Jogos de Ação: Planejando Trajetórias

Em jogos de ação, seja em um jogo como o Pac-Man ou como o DOOM, Unreal ou Half-Life, usualmente precisamos estabelecer trajetórias para guiar os NPCs (Non-Player Characters) pelo ambiente. Neste tipo de jogos se faz necessária a solução do problema da busca de caminhos (*path-finding ou path-planning)*, ou seja, devemos aplicar um algoritmo de busca, que permita encontrar um caminho, de um ponto inicial (origem) até um ponto final (destino). Portanto, este problema pertence ao mesmo grupo de técnicas da área da I.A. discutidos nesta seção, denominado de busca em espaço de estados [Russell & Norvig 95]: exploramos o espaço de configurações, dadas diversas possibilidades de ações e caminhos a seguir, buscando alcançar um determinado estado ou posição alvo (Fig. 3).

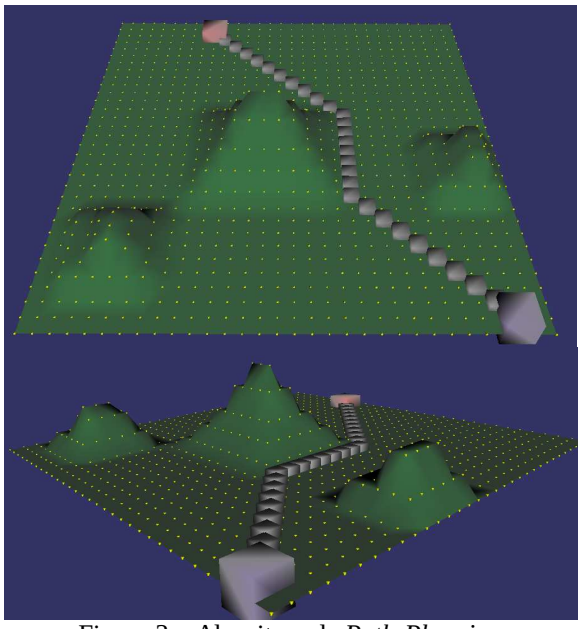


Figura 3 – Algoritmo de *Path-Planning*

Os algoritmos de *path-finding* (traçado de rotas e trajetórias) tem sido intensivamente investigados na I.A., principalmente com aplicações em áreas como a robótica autônoma (robôs móveis) e jogos digitais. Em ambientes onde possuímos um mapa descrevendo os obstáculos e a configuração do ambiente, podemos aplicar um algoritmo de busca (cega ou heurística) para traçar uma rota. No caso da busca cega, o algoritmo irá buscar explorar as diferentes possibilidades de caminhos até encontrar um que lhe permita alcançar o alvo. A **busca cega** tende a ter um **custo muito elevado**, pois realiza uma busca sobre todas as possibilidades de configurações (espaço de estados) e conseqüentemente é um algoritmo bastante pesado. Na **busca heurística** serão usadas regras que permitem **otimizar** a busca pela solução, como por exemplo, na figura 3 podemos usar uma heurística que dê uma preferência maior para a exploração das zonas planas. Assim o algoritmo de busca irá evitar passar (buscar caminhos) pelas zonas elevadas (montanhas), obtendo assim um caminho, que apesar do desvio, é melhor segundo algum tipo de critério ou informação que lhe foi passada.

O uso desta técnica de I.A. ocorre principalmente em jogos do gênero RTS (*Real Time Strategy*), onde temos um número considerável de agentes e rotas a serem traçadas. Para traçar uma trajetória entre um ponto de origem e um destino, é possível o uso de um algoritmo de busca cega (não informada) e de força bruta, testando todos os caminhos possíveis até alcançar o destino (e.g. *depth-first* or *breadth-first search* [Russell 1995]). Devido à quantidade e a complexidade desses agentes, assim como a complexidade do ambiente em que estão inseridos os agentes, pode ser necessária a otimização desta busca, utilizando-se de heurísticas que permitam acelerar a busca do caminho através do uso de algum tipo de informação e/ou regra (busca informada / busca heurística) [Winston 1992].

Um NPC em um jogo de ação pode ser um agente simples e sem inteligência, que realiza tarefas simples e repetitivas (pré-definidas). Entretanto, em um jogo o interessante é a competição dos humanos contra NPCs dotados de algum tipo de inteligência. Por exemplo, os Fantasmas do Pac-Man podem usar algoritmos baseados em regras bastante simples para se movimentar em direção ao jogador, entretanto, também é possível usar algoritmos de I.A. para traçar uma rota entre a posição do Fantasma (origem) e o Pac-Man (destino).

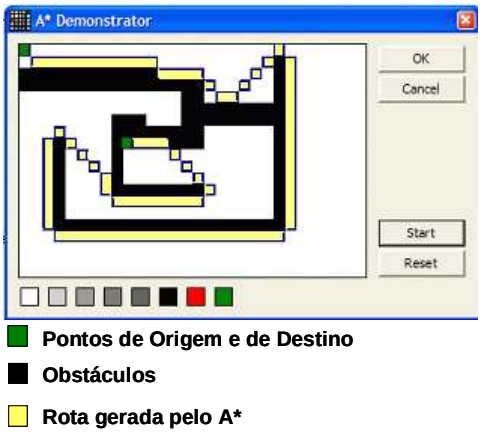


Figura 4 – Algoritmo A* de *Path-Planning* em mapa do tipo grade (matriz de ocupação) [Matthews 2003]

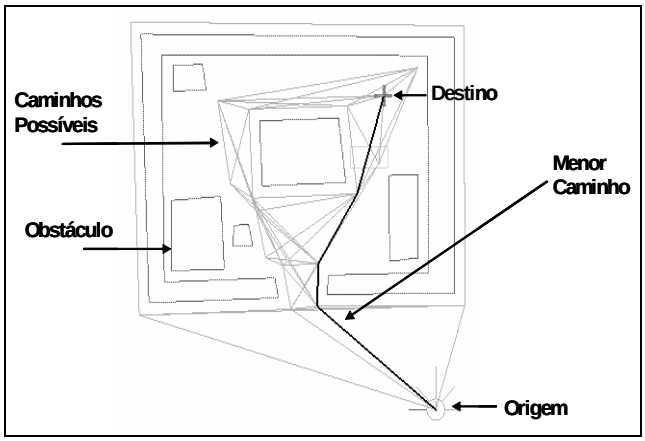


Figura 5 – Algoritmo de Dijkstra de *Path-Planning* em mapa geométrico [Osório 2004, Heinen 2000]

A busca de trajetórias pode ser realizada usando diferentes representações do mapa do ambiente, seja usando um mapa do tipo grade (mapa do tipo *occupation grid* – Figura 4), ou um mapa do tipo geométrico (Figura 5). No caso do mapa do tipo grade, cada elemento da grade representa uma posição livre, ocupada, ou mesmo um elemento com um certo custo associado à sua transposição (onde paredes tem um custo infinito). Os mapas do tipo geométrico, descrevem a geometria do ambiente, sendo que podemos adicionar nestes mapas um grafo que descreve a trajetória de um agente. Este grafo é composto por uma seqüência de pontos posicionados junto ao mapa geométrico. Cada aresta do grafo pode também ter um custo associado, o custo de ir de um ponto a outro nas extremidades desta aresta. Se for removida uma aresta entre dois pontos, isso pode indicar que não é possível passar de um ponto a outro por aquela rota. Uma aresta é removida por estar colidindo e/ou atravessando uma parede. Podemos notar na Figura 5 que a aresta que une diretamente a origem ao destino foi removida por "passar por cima" das paredes.

Existem diversos algoritmos de *path-planning* encontrados na literatura [Osório 2004, Dudek 2000, Russel 1995]. Alguns dos mais conhecidos são o A* (A Star ou A Estrela) [Russell 1995, Lester 2004, A Star 2007], e o algoritmo de Dijkstra de busca em grafos, no caso, representando o espaço de configurações e o grafo de visibilidade (caminhos possíveis) do ambiente [Osório 2004, Heinen 2000, Dijkstra 2007]. Entretanto, outros algoritmos também são encontrados na literatura, baseados em Campos Potenciais, em Diagramas de Voronoi, e em diversas outras técnicas, muitas vezes compartilhadas com aplicações e pesquisas em robótica autônoma [Jung 2005, Dudek 2000].

O algoritmo mais usado em jogos é o A Star (A* - A estrela) [Russell 95], que através de uma heurística permite identificar o melhor trajeto entre dois pontos. Usualmente o ambiente é descrito por uma grade, definindo um mapa de ocupação espacial, sobre o qual é aplicado o A*, mas também pode ser aplicado sobre uma representação de um grafo cujas arestas são ponderadas (possuem custos associados). O A* é um algoritmo de busca informada, pois considera o quanto já avançamos em direção ao destino, e uma estimativa de quanto falta para alcançar nosso objetivo. Usando esta heurística podemos explorar o espaço de possíveis configurações (trajetórias), selecionando sempre a melhor trajetória de acordo com a heurística empregada. Esta técnica será melhor detalhada na seção seguinte.

Uma outra possibilidade de se identificar o melhor caminho é usando uma descrição geométrica do ambiente e criar um grafo ponderado (pelas distâncias) que uma o ponto de partida ao destino, considerando a menor distância a percorrer, e aplicando o Algoritmo de Dijkstra. As Figuras 4 e 5 apresentam um exemplo de trajetórias geradas com cada uma das duas técnicas descritas acima.

3. Algoritmo A* (A Star)

O uso de algoritmos de busca de caminhos geralmente pode ser diferente para cada tipo de aplicação, pois muitas vezes é necessário que o agente chegue ao ponto final do modo mais rápido possível (menor distância), em outros casos é preciso que ele minimize algum custo, como combustível, dinheiro, equipamento, esforço, entre outros parâmetros.

O algoritmo A* (A Star) [Rabin 2002, cap.3], é um dos algoritmos de *path-finding* mais utilizados em jogos. Ele é baseado em grafos/árvores/grades, onde cada nó é um estado do caminho e cada estado possui transições que levam a outros estados, onde estas transições podem possuir custos diferentes. Este algoritmo encontra facilmente (de modo rápido e eficiente) a melhor rota para um único agente

3.1. A* em Detalhes

Como já foi citado anteriormente, o A* é um algoritmo baseado em grafos, que visa encontrar o caminho "ótimo" entre dois pontos. Cada nó desse grafo é um estado (ou posição) do caminho. Assim sendo, o objetivo do algoritmo de *path-finding* é analisar cada nó e decidir sobre quais nós que ele deverá passar, compondo o caminho final da origem até o destino.

O A* (A Star) [Deloura 2000] é um dos algoritmos mais populares de *path-finding*, pois ele é bastante flexível e pode ser usado em uma grande variedade de contextos, apenas mudando a heurística básica (funções de custo) que auxilia na exploração e busca no espaço de estados.

3.1.1. *Custo Estimado e Custo do Caminho Percorrido*

A idéia inicial do A* é a composição de dois custos a fim de compor a heurística aplicada na restrição do espaço de busca (direcionamento da busca), compondo uma função $f(x) = g(x) + h(x)$. A primeira parte da função, g(x), é custo propriamente dito do caminho percorrido, ou seja, é quanto custa para o agente se mover para o nó em questão. Por exemplo, na Figura 6 (reproduzida de [Winston 1992]) o custo de caminho do nó **S** para o nó **A** é de três unidades. Este custo pode estar representando qualquer propriedade consumida pelo agente, como por exemplo: tempo, equipamentos, distância, combustível gasto, etc.

Conforme mostra a Figura 7, a segunda função que compõe a heurística, h(x) é o custo estimado até o destino, que no caso desta figura está representado pela distância absoluta (em linha reta) de cada nó até o objetivo final.

Supondo que há um agente que se situa num ponto **S** (*start*), onde seu objetivo é chegar ao ponto **G** (*goal*), como pode ser visto na Figura 6. Este agente terá uma

série de possibilidades de rotas a seguir para alcançar o seu objetivo, como por exemplo: **SADEFG**, **SABEFG**, **SDABEFG**, **SDEFG**. Cada uma destas rotas consumirá uma certa quantidade de recursos do agente (e.g. tempo), ou seja, irá afetar o cálculo do custo final total da execução desta trajetória.

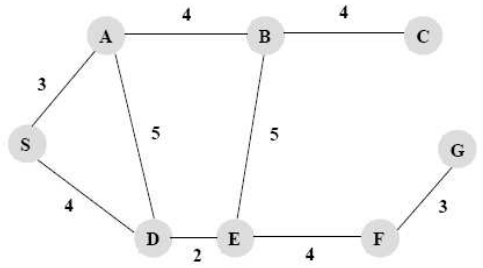


Figura 6: Custo de Caminho [Winston 1992, p.64]

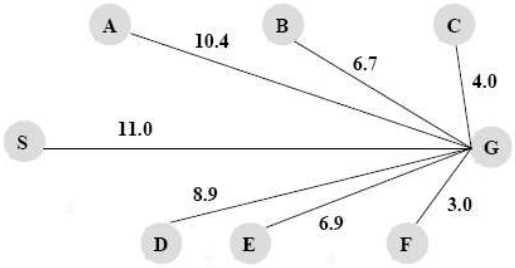


Figura7: Custo Estimado dos nós [Winston 1992, p.71]

A heurística base do A* é a soma do custo de caminho (já percorrido), com o custo estimado do caminho (que ainda falta percorrer), dando como resultado um custo total estimado para alcançar o nó em questão.

Considerando o exemplo anterior, caso o custo considerado fosse somente baseado na minimização do custo do caminho percorrido, a escolha ideal no primeiro passo deveria ser sempre mover do nó **S** para o nó **A**, pois a outra opção (nó D) teria um custo maior. Por outro lado, se a opção fosse minimizar somente o custo estimado (quanto falta) para alcançar o objetivo, a escolha ideal no primeiro passo deveria ser sempre mover do nó **S** para o nó **D**, pois esta escolha nos aproxima mais do objetivo (nó **G**). Como não é possível garantir qual das duas opções será a melhor, visto que isto dependerá muito dos caminhos possíveis de serem percorridos, ambos os custos, o custo estimado (h(x)) e o custo do caminho percorrido (g(x)) são levados em conta para a estimativa (f(x)) do custo final. Esta composição permite assim achar o caminho ótimo (considerando a heurística adotada) da origem para o destino, e por isso é conhecido como Algoritmo A "ótimo" (A Star).

3.1.2. *Path Planning*

É importante destacar, que se for adotada apenas a heurística acima para a decisão de para qual novo estado vamos passar, o agente ainda assim não terá garantias de que irá encontrar o melhor caminho. O agente poderá selecionar um caminho que irá levá-lo para um mínimo local (otimiza o custo, porém não alcança o objetivo final), e ao chegar neste ponto, se não tiver alternativas, ficará bloqueado.

O algoritmo do A*, assim como os algoritmos de busca não informada como o *depth-first* e o *breadth-first*, deve possuir uma lista de nós candidatos a serem visitados (*Open set*) e de nós já visitados (*Closed set*) [Russell 1995]. A diferença entre estes algoritmos é que o Open Set do *depth-first* é uma PILHA (LIFO – Last In, First Out), o Open Set do *breadth-first* é uma FILA (FIFO – First In, Fist Out), e o Open Set do A* está ordenado pelo custo da função heurística f(x), isto é, sempre selecionamos o nodo de menor custo desta lista.

1) Adicione o quadrado inicial à lista aberta.

2) Repita o seguinte:

   a. Procure o quadrado que tenha o menor custo de F na lista aberta. Nós referimos a isto como o quadrado corrente.

   b. Mova-o para a lista fechada.

   c. Para cada um dos quadrados adjacente a este quadrado corrente.

      i. Se não é passável ou se estiver na lista fechada, ignore. Caso contrário faça o seguinte:

         1. Se não estiver na lista aberta, acrescente-o à lista aberta. Faça o quadrado atual o pai deste quadrado. Grave os custos F, G, e H do quadrado.

         2. Se já estiver na lista aberta, confere para ver se este caminho para aquele quadrado é melhor, usando custo G como medida. Um valor G mais baixo mostra que este é um caminho melhor. Nesse caso, mude o pai do quadrado para o quadrado atual, e recalcule os valores de G e F do quadrado. Se você está mantendo sua lista aberta ordenada por F, você pode precisar reordenar a lista para corresponder a mudança.

   d. Pare quando você:

      i. Acrescente o quadrado alvo à lista fechada o que determina que o caminho foi achado, senão.

      ii. Não ache o quadrado alvo, e a lista aberta está vazia. Neste caso, não há nenhum caminho.

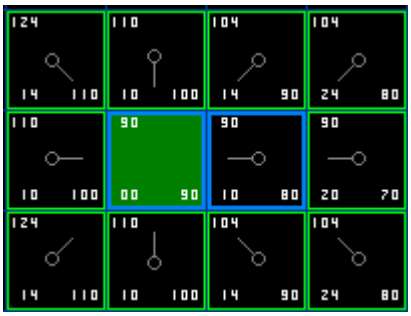
3) Salve o caminho. Caminhando para trás do quadrado alvo, vá de cada quadrado a seu quadrado pai até que você alcance o quadrado inicial. Isso é seu caminho.

Figura 8: Pseudocódigo do algoritmo A* aplicado em um mapa do tipo grade [Lester 2004]

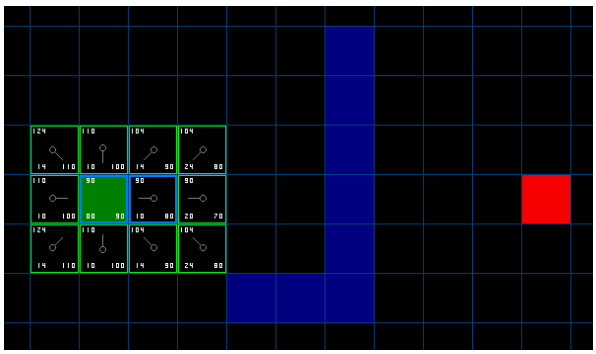
Portanto, uma outra parte do algoritmo de busca de caminhos do A* está no planejamento global de sua rota: caso ele descubra que não adotou uma boa rota (levando a um mínimo), ele poderá retornar e "se arrepender", ou seja, retirar um novo nodo da lista de nós candidatos a exploração, o *Open Set*. O agente pode então procurar por outras rotas, sempre que ficar bloqueado, buscando em outros lugares de menos custo que ainda assim possam ser alcançados a partir do ponto de origem.

Quando o algoritmo estiver verificando um nó de onde não é possível alcançar o objetivo, ou, de onde o custo final seja mais alto do que o de outros nós disponíveis no Open Set, é necessário se arrepender e seguir analisando por outro caminho, obtido da lista *Open set*. A Figura 8 mostra o pseudocódigo do algoritmo A* [Lester 2004], e um exemplo que demonstra o A* sendo executado é visto na Fig. 9, onde este programa pode ser obtido no site do GameDev [Lester 2007].

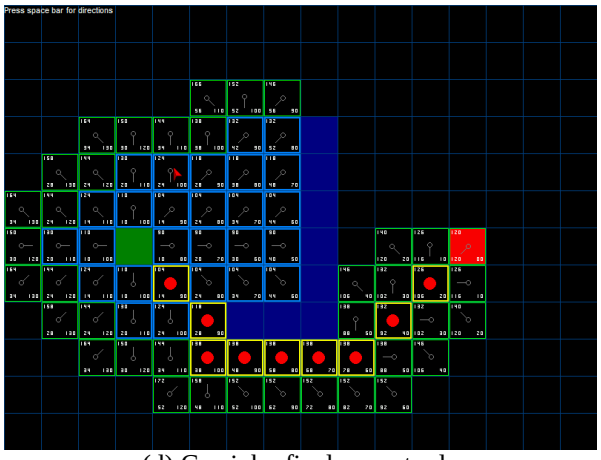
(a) Distância em linha reta: 9 casas (custo estimado 90)



(b) Avaliação da função heurística f(x) = g(x) + h(x). Valor acima: custo total f(x). Valor abaixo à esquerda: custo do caminho percorrido g(x). Valor abaixo à Direita: custo estimado do caminho restante h(x). Casas em verde: Open Set. Casas em azul: Closed set.



(c) Início da exploração do caminho



(d) Caminho final encontrado

Figura 9: Tela do Software de Demonstração do Path-Finding baseado em A* [Lester 2004, 2007]

A Figura 9 demonstra a execução do algoritmo A* para um mapa simples, onde são apresentadas as funções de custo (estimado – Fig. 9(a) e percorrido – Fig. 9(b)). Podemos perceber nesta figura que os valores são representados por inteiro, evitando assim cálculos de ponto flutuante e acelerando o processamento. O custo de avançar uma casa é 10 em linha reta, e se for em diagonal será 14, equivalente à distância euclidiana da diagonal, ou seja, SQRT(1+1) ~ 1.4 (representado pelo valor 14, em notação inteira). O algoritmo explora a matriz e o caminho final é apresentado na Fig. 9(d) – linha de pontos vermelhos.

3.2. A* para Jogos

As heurísticas e algoritmos até agora apresentados se aplicam ao A* padrão. Este algoritmo conforme apresentado permite que um único agente defina sua trajetória de um ponto de origem até uma posição de destino, desviando dos obstáculos conhecidos (estáticos) que estão presentes no mapa do ambiente.

Porém uma das características interessantes do algoritmo A* é que podemos alterar seu comportamento apenas modificando ou adicionando heurísticas e/ou informações extras junto ao algoritmo original. Podemos alterar a matriz de custos dinâmicamente e assim influenciar no resultado final obtido pelo algoritmo de planejamento de trajetórias.

Essa característica do A* de alterar seu comportamento baseado nas funções de custo pode ser muito útil em um jogo, permitindo a criação de novos tipos de trajetórias e de comportamentos. Além de planejar trajetórias entre 2 pontos, o A* pode ser usado para criar trajetórias que evitem passar próximo a um inimigo (fuga), simplesmente alterando a matriz de custos. Também é possível identificar regiões do mapa que desejamos evitar (pontos de emboscada e pontos mais perigosos) e comportamentos particulares, onde cada agente usa seu próprio mapa de custos (regiões preferenciais de atuação). Também é possível realizar a cooperação entre agentes através da troca de informações sobrepostas ao mapa do ambiente. Estes comportamentos serão detalhados no item 3.3 (próxima seção). O tipo de comportamento do agente vai depender da função a ser exercida pelo agente no jogo e da necessidade de velocidade de jogo, que permita assim obter um bom *gameplay* [Gameplay 2007].

O balanceamento entre a velocidade e a precisão também pode ser explorado para deixar um jogo mais rápido. Para muitos jogos pode não ser necessário obter o melhor caminho entre um ponto e outro, mas sim algo aproximado. O mapa do A* pode ser dimensionado com diferentes “níveis de resolução”, onde a grade de ocupação pode representar uma visão mais macro ou mais detalhada do ambiente. Também podemos dividir o ambiente e assim facilitar (acelerar) o processamento do planejamento de uma trajetória. Por exemplo, podemos planejar uma trajetória que vai até uma ponte (obrigando o agente a usar esta ponte), e depois da ponte até o seu objetivo final, ao invés de planejar toda a trajetória.

Por fim, apesar do mapa de ocupação (grid) ser plano, podemos adicionar elementos com custos diferentes, e assim representar elevações no terreno (montanhas e encostas) que teriam um custo de transposição mais elevado. Isto levaria o agente a dar preferência aos terrenos planos. Também podemos adicionar custos que tornem mais “cara” a passagem por terrenos pantanosos, pela água, ou por qualquer tipo de terreno que apresente uma indicação de dificuldade de transposição, fazendo com que o agente

evite a escolha de trajetórias que passem por estas áreas. Por outro lado, também podemos alterar o mapa para privilegiar certas rotas, forçando que o agente dê uma melhor pontuação e preferência para certas trajetórias. Todos estes custos podem ser adicionados sendo sobrepostos ao mapa.

3.3. Comportamentos com o A*: Regiões Importantes

Definir regiões importantes é muito interessante quando se joga um jogo de estratégia. O agente deve ser capaz de analisar os caminhos que são comuns a outros agentes, e desse modo identificar, por exemplo, rotas perigosas e possíveis pontos de emboscada. Primeiramente é necessário verificar qual é a prioridade dos caminhos escolhidos pelos agentes, ou seja, em qual tipo de terreno os agentes inimigos ou amigos estão agregando baixo custo.

Supondo que os agentes estejam apenas considerando o A* padrão, onde são considerados os custos estimados mais o do caminho percorrido. O agente poderá analisar os terrenos em que está passando, e verificar se há uma área de baixo custo em meio a uma grande área de alto custo, como por exemplo, um vale em meio à montanhas. Para efeitos de um jogo, pode-se supor que aquela área é um caminho perigoso (ideal para emboscadas), por que vários agentes irão considerar aquele caminho como sendo um caminho ótimo (de baixo custo).

Uma outra forma usada para encontrar essas "áreas de baixo custo" é executar o algoritmo A* várias vezes partindo de lugares diferentes, porém com o mesmo destino, e analisar quais os nós que foram escolhidos pelo algoritmo A* e por onde os agentes irão passar seguidamente. Se um número considerável de caminhos planejados pelo A* fizer um percurso parecido, passando sempre por uma mesma zona, pode-se considerar esta região onde os caminhos coincidiram, como uma região de baixo custo, e portanto será uma área de maior tráfego para os agentes. Esta informação poderá ser usada do ponto de vista estratégico para o jogo, e não apenas para a movimentação dos agentes.

Outro modo de definir regiões de possível tráfego intenso de agentes é executar o algoritmo várias vezes estabelecendo como origem um mesmo lugar e como destino também um mesmo ponto. Porém, para cada rota definida deve-se definir aquela área como inalcançável, ou seja, caminhos já percorridos passam a ter um custo muito alto. Assim ao final da geração de trajetórias teremos várias áreas marcadas sobre o mapa, e caso essas áreas fiquem muito próximas umas das outras, poderemos agrupar estas regiões como sendo regiões de alto tráfego e de relativo baixo custo. Caso essas áreas fiquem muito dispersas, poderemos constatar então que essas regiões provavelmente farão parte de rotas alternativas de outros agentes, e portanto, que existem diversas alternativas (rotas de fuga) possíveis. A Figura 10 apresenta estas regiões de maior tráfego, identificadas com o auxílio do A*.

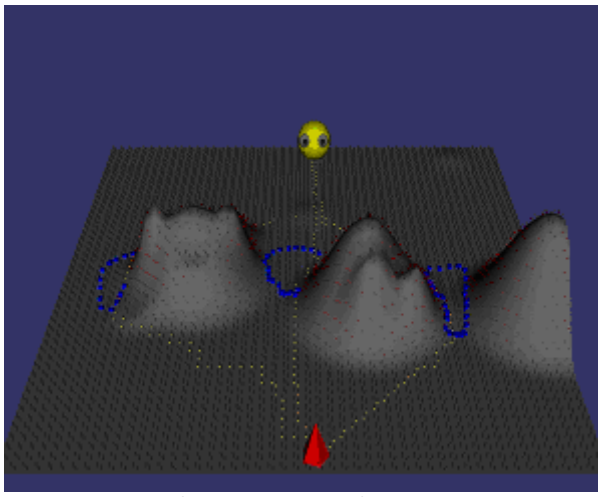


Figura 10: Definindo regiões de fluxo de agentes - As montanhas são áreas de alto custo e as zonas tracejadas indicam as regiões de maior tráfego (vales).

Para que esta tarefa não fique muito cara computacionalmente, deve-se dar essa missão de mapear as regiões para um único agente ou para um grupo bastante limitado de agentes. Durante um certo tempo do jogo, são coletadas as informações resultantes da aplicação do algoritmo de planejamento de trajetórias. Após esta coleta de dados, os agentes passam a usar estas informações para a definição de estratégias de defesa e/ou de ataque, considerando rotas usuais e rotas alternativas existentes em uma dada região. Esses agentes fariam parte de um time de agentes exploradores, comuns em jogos de estratégia.

3.4. Comportamentos com o A*: Fuga

Em um jogo do tipo predador-presa (e.g. Pac-Man), um comportamento usual é o de fuga. O primeiro passo para obter um comportamento de fuga é definir prioridades em relação ao caminho a ser seguido, mas também o caminho a ser evitado. Inicialmente devemos definir o ponto de destino do agente: a solução mais simples e rápida é determinar o ponto mais afastado (em linha reta) em relação ao(s) predador(es). Posteriormente iremos discutir outras opções, mas assume-se inicialmente que uma posição de destino é estabelecida previamente, podendo ser adotado o ponto extremo de maior distância em relação aos demais agentes que participam da perseguição.

Após ser definido o destino, a próxima prioridade é não passar por perto do(s) perseguidor(es). Para que isso aconteça, devemos aumentar o custo de todos os nós que estejam próximos a um dos inimigos. Desta forma iremos criar uma camada dinâmica (*layer*) com custos que serão sobrepostos sobre os custos do mapa do ambiente, somando assim os custos referentes aos obstáculos estáticos definidos no mapa do ambiente original, junto com os custos que são determinados de modo dinâmico e que possuem uma relação direta com a proximidade em relação aos inimigos.

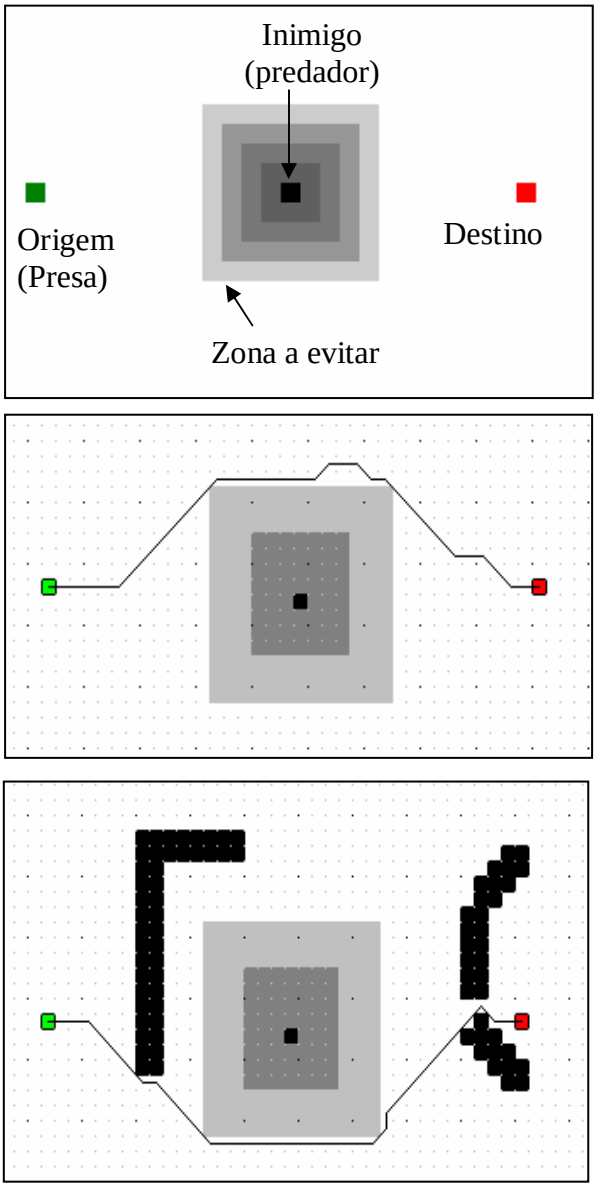


Figura 11: Área de influencia, trajeto do agente sem obstáculos, trajeto do agente com obstáculos.

Cada inimigo ou objeto que o agente deseja desviar deve possuir definido um "raio de influência". Este raio de influência irá determinar uma região onde será aplicado um custo adicional aos nós consultados pelo algoritmo A*, estabelecendo um mínimo de distância que é seguro para o agente passar em relação a um inimigo. Desta forma o agente irá gerar uma trajetória que evita passar dentro do raio de influência do objeto a evitar, reduzindo os riscos de ser capturado (Fig. 11). Quanto maior o raio de influência, mais longe o agente irá passar do um inimigo.

Uma vez que também podemos definir regiões com um possível tráfego mais intenso de agentes (ver item 2.1), é interessante que o fugitivo possa avaliar as informações disponíveis sobre estas áreas, que poderão ser protegidas ou não por aliados, ou então, possuir uma maior chance de ter inimigos a espreita preparando uma emboscada.

Com o uso do A* e de mapas compostos por camadas (estática e dinâmica) de custos, podemos assim dar prioridade a outras questões referentes à estratégia de fuga. Concluindo, o agente será capaz de combinar em seu comportamento a geração de trajetórias que evitem obstáculos (mapa original), que evitem os predadores (camada dinâmica adicional) e que busque alcançar pontos específicos mais seguros do ambiente.

3.5. Comportamentos com A*: Coordenação e Cooperação de Multi-Agentes

O A* é usualmente aplicado para definir trajetórias de um agente único, porém pode ser usado também em sistemas multi-agentes, visando obter um comportamento coordenado. Em um sistema com múltiplos agentes o objetivo de cada agente pode ser compartilhado entre eles (e.g. capturar uma presa). Portanto, caso se deseje criar um sistema multi-agente cooperativo, isto implica que as rotas deverão ser planejadas em conjunto, para que um resultado coletivo possa ser alcançado. Se o objetivo dos agentes for o de cercar alguma posição-alvo, eles devem tomar caminhos diferentes, sendo os diversos caminhos planejados de forma conjunta por dois ou mais agentes, compartilhando as informações de mapa e trajetórias, a fim de que o resultado final seja mais adequado.

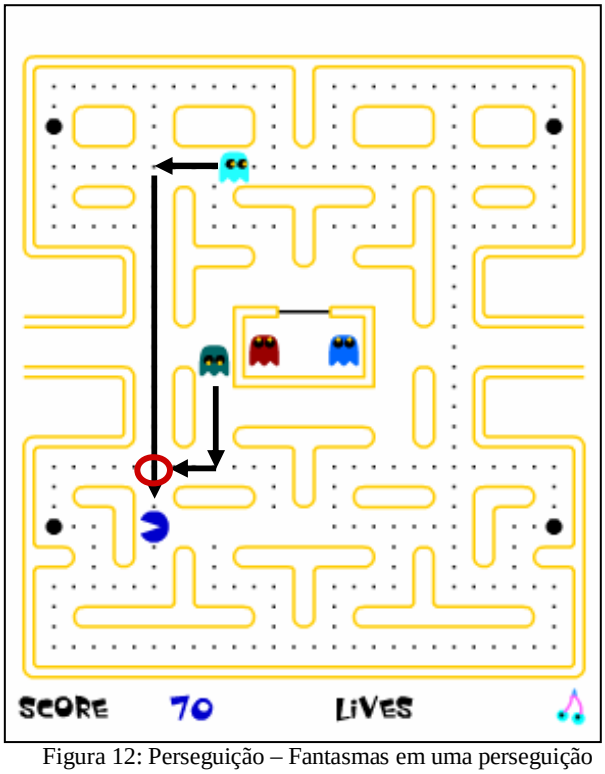


Figura 12: Perseguição – Fantasmas em uma perseguição ***não coordenada*** ao Pac-Man [PacMan 2007]

Em um jogo como o Pac-Man, a perseguição pode se dar de forma individual com cada fantasma (predador) planejando sua própria trajetória de forma independente dos demais. A Figura 12 mostra um exemplo destas trajetórias independentes onde não existe uma coordenação entra as ações do fantasma.

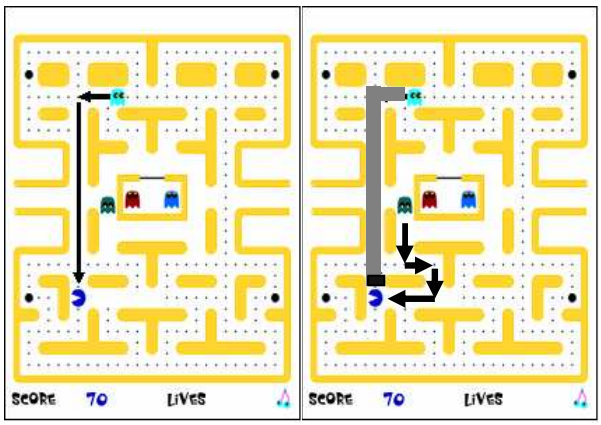


Figura 13: Perseguição – Fantasmas em uma perseguição ***coordenada*** ao Pac-Man [PacMan 2007]

Em uma perseguição coordenada, podemos traçar primeiramente a trajetória de um dos fantasmas até o seu destino (presa). Esta trajetória é apresentada na Figura 13 no lado esquerdo, onde o fantasma que está mais acima determina uma rota de ataque ao PacMan. Em seguida, esta rota recebe um custo adicional (marcação em cinza) que é adicionado aos custos do mapa usado pelo A*. Além disto, marcamos que a passagem imediatamente superior ao PacMan está bloqueada, pois já será atacada pelo primeiro fantasma (indicado pelo retângulo preto acima do PacMan). A partir deste novo mapa dinamicamente modificado e considerando a rota do primeiro fantasma, aplicamos o A* para calcular a trajetória do segundo fantasma, apresentada na Figura 13 no lado direito. Podemos perceber que o segundo fantasma buscará uma rota alternativa, atacando o PacMan por um outro lado.

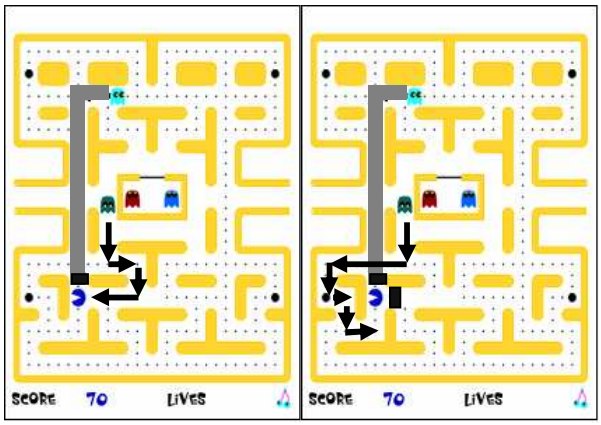


Figura 14: Perseguição – Fantasmas em uma perseguição ***coordenada*** ao Pac-Man, com adição de bloqueios dinâmicos

É importante destacar que, como podemos ter custos diferenciados no mapa, podem ser adicionados bloqueios que impedem completamente a passagem do fantasma (retângulos pretos nas figuras 13 e 14), ou elementos que apenas aumentam o custo da passagem do fantasma, mas não vão impedir completamente que ele use este caminho. Este é o caso da marcação cinza da figura 14, onde queremos que o segundo fantasma evite a mesma rota do primeiro, mas se for necessário ele poderá compartilhar alguma parte desta rota, apenas considerando um custo mais elevado por ter feito esta escolha.

Além das situações acima citadas, existem muitas outras situações que poderiam ser consideradas, onde constata-se que devido às particularidades de cada situação é necessário buscar otimizações específicas do algoritmo A*, para cada uma delas. A maioria dessas técnicas faz uso de heurísticas aplicadas sobre o algoritmo original do A*, o que pode ser feito através da manipulação dos custos dos nós do grafo usado na busca do caminho.

3.6. Comportamentos com A*: Cercar um Alvo

Em jogos, muitas vezes é necessário se obter uma estratégia mais inteligente, como por exemplo, cercar uma área importante ou um inimigo. Este tipo de estratégia é aplicado em um campo de batalha, onde os NPCs devem se posicionar ao redor do alvo. Vamos discutir aqui, mais uma outra forma de cooperação e coordenação de multi-agentes, baseada no A*, e que permite executar este tipo de tarefa: cercar um alvo.

Para se fixar o destino de cada agente, é definido um círculo ao redor do alvo a ser cercado. Este círculo será dividido, em partes iguais de modo que os agentes fiquem distribuídos ao redor do alvo. O raio será dado pela abrangência da área ao redor do alvo, exceto em casos mais específicos onde se deseja chegar a uma distância específica do ponto-alvo.

Para que os agentes cerquem o alvo vindos de diferentes regiões, é feito um novo círculo ainda maior da região alvo. Primeiramente a região alvo é dividida em quadrantes, e utilizando-se destes mesmos quadrantes podemos atribuir custos para distribuir o envio de agentes para cada um deles. Assim podemos aumentar o custo (camada dinâmica) para os quadrantes onde não se quer que um ou mais agentes passem. Desse modo o agente não passará pelos quadrantes que não foram atribuídos a ele, indo diretamente para o quadrante que lhe foi especificado, e desta forma cercando o alvo (Fig. 15).

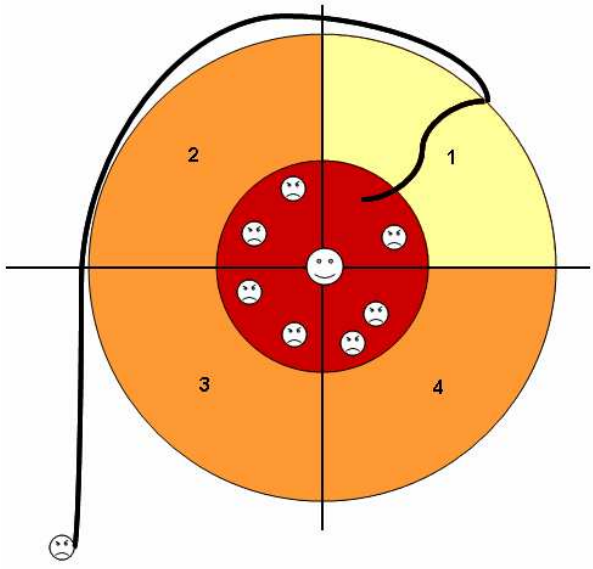


Figura 15: Agentes cercando o alvo

Na figura 15 pode-se ver claramente, o exemplo de um agente contornando os quadrantes que não são de sua responsabilidade, até chegar ao seu destino. Desse modo um grupo de agentes irá cercar o alvo, de modo coordenado, e possivelmente poderá evitar melhor que o agente alvo possa fugir.

Deve ser salientado que, além da trajetória também teremos que depois controlar o deslocamento dentro desta trajetória. Em tarefas como cercar o alvo, a velocidade individual de deslocamento no ambiente deve ser também planejada de modo que os diversos agentes se posicionem "no tempo certo" ao redor do alvo.

Concluindo, além do planejamento de trajetórias com o A*, o deslocamento e o re-planejamento a ser feito quando é detectada uma mudança significativa no estado do ambiente, são também problemas importantes a serem abordados. Estes aspectos, relativos ao A*, não serão abordados neste tutorial, pois iremos dar um maior destaque as arquiteturas de controle dos agentes, usadas no planejamento e execução de suas ações, conforme será discutido na seção seguinte.

4. Arquiteturas para o Controle de Agentes (NPCs)

Nas seções anteriores deste tutorial, tratamos os agentes como sendo baseados em técnicas clássicas de I.A., visando a busca de soluções através da exploração do espaço de estados, seja através de uma busca cega, ou no caso do A*, com uma busca heurística. O planejamento de trajetórias é uma das tarefas mais usuais em jogos, onde o A* é tipicamente o algoritmo mais usado. Entretanto, existem aplicações onde podemos adotar outros tipos de algoritmos para o planejamento e execução das ações dos agentes. Um termo mais genérico é usado para definir o módulo que irá gerenciar a "inteligência" do agente, sendo denominado de: Arquitetura de Controle dos Agentes.

As arquiteturas de controle podem ser usualmente organizadas em categorias [Osório 2004]:

- **Arquiteturas Deliberativas:** Planejam antecipadamente (deliberam) as ações dos agentes, baseadas em informações e conhecimentos sobre o ambiente, utilizando usualmente regras (heurísticas) que controlam o comportamento do agente. O A* adota uma arquitetura deliberativa.
- **Arquiteturas Reativas:** São arquiteturas simples de controle adotadas em agentes que possuem a capacidade de perceber o ambiente, pois são dotados de sensores. Os agentes são capazes de perceber e reagir à estas percepções, através de uma integração sensorial-motora (reativa).
- **Arquiteturas Hierárquicas e Híbridas:** São arquiteturas que integram as características das arquiteturas deliberativa e reativa, visando obter sistemas de controle mais robustos.

A seguir vamos detalhas cada uma destas diferentes arquiteturas usadas para o controle de agentes autônomos, os NPCs em jogos.

4.1. Arquitetura Deliberativa

Uma arquitetura é dita deliberativa (ou cognitiva) quando a escolha da ação a ser executada pelo agente é realizada a partir de um modelo simbólico do ambiente e de um plano de ações. Essa arquitetura está fundamentada na produção de uma seqüência de ações (planos) para alcançar um determinado objetivo.

As ações de um agente deliberativo estão baseadas nas hipóteses de que o agente possui um conhecimento do ambiente, e em certos casos, sobre os outros agentes. Para isto, é mantida uma representação explícita do conhecimento sobre o ambiente, bem como podemos manter um histórico das ações passadas. Entretanto, a arquitetura cognitiva é tipicamente incapaz de agir (ou reagir) de forma rápida e adequada perante situações não previstas. Este modelo adota a hipótese de que as condições do mundo permanecem estáticas enquanto o agente estiver executando as suas ações ou processando alguma informação para deliberar sobre as ações.

Os agentes deliberativos raciocinam e decidem sobre quais objetivos devem alcançar, que planos seguir e quais ações devem ser executadas em um determinado momento. Deste modo, um agente executa uma ação inteligente quando, possuindo um certo objetivo e o conhecimento de que uma certa ação o conduzirá a este objetivo, seleciona esta ação. Os algoritmos de busca em espaço de estados (cega ou heurística) são uma das técnicas que permitem a implementação de agentes deliberativos.

O A*, tratado na seção anterior, é um exemplo de algoritmo que implementa um controle deliberativo: a partir do mapa do ambiente, contendo os obstáculos, ponto inicial e o ponto de destino, o algoritmo estabelece um plano de ações para alcançar seu objetivo. O algoritmo clássico do A* não permite que o agente possa tratar de situações imprevistas (bloqueio no meio do caminho), pois todas suas ações são previamente definidas por um plano, que estabelece a trajetória. Se algo mudar, deve ser feita uma atualização do mapa e um novo plano deve ser gerado.

Além dos algoritmos de busca de caminhos (A*), existem outras técnicas que são correntemente utilizadas em jogos para o controle deliberativo do comportamento dos NPCs, notadamente as mais importantes são: Autômatos Finitos (FSA – *Finite State Automatas*) e Controle baseado em Regras (RBS – *Rule Based Systems*). Vamos descrever brevemente a seguir estas técnicas.

4.1.1. *Autômatos Finitos* (FSA)

Os NPCs podem ter um comportamento que simula inteligência, mas que na realidade é bastante fixo e pré-definido, obtido através do uso de um autômato. Esta é uma das técnicas de controle deliberativo bastante utilizadas nos jogos na atualidade. Os autômatos definem uma seqüência de estados e de condições para passagem de um estado a outro, fazendo com que o NPC aparentemente execute ações coerentes. Veja um exemplo de um autômato na Figura 16.

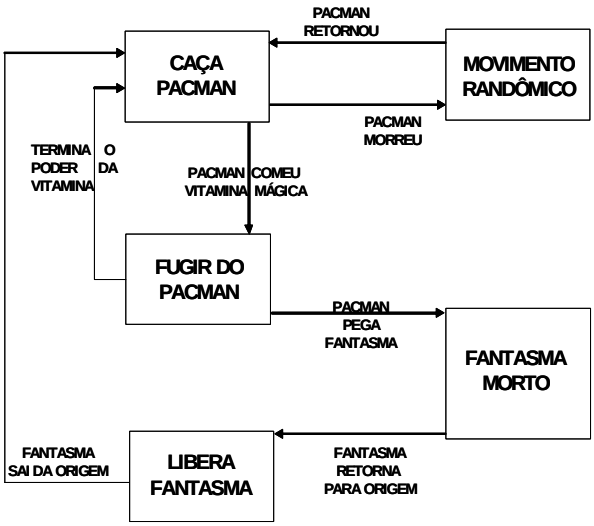


Figura 16: Exemplo de um autômato (FSA) usado para o controle do Fantasma no jogo do Pac-Man [adaptado de Schwab 2004]

Uma descrição detalhada de como pode ser implementado um controle de NPCs baseados em FSA é encontrado no *Game Programming Gems* [Dybsand 2000], e no *AI Game Programming Wisdom* [Rabin 2002 – Cap. 6.5 e 6.6].

4.1.2. *Controle Baseado em Regras* (RBS)

Nos sistemas de controle deliberativo baseado em regras – RBS (*Rule Based Systems*) [Rabin 2002 – cap. 6.4], um NPC pode ser controlado por um conjunto de regras, como por exemplo, se Jogador próximo e NPC forte então NPC atira; se Jogador próximo e NPC fraco então NPC foge. Na realidade um FSA pode ser alternativamente descrito por um conjunto de regras, sendo portanto, um mecanismo similar ao uso dos autômatos, apenas descrito na forma de regras. Note que é possível fazer uso de um sistema baseado em regras mais complexo, onde muitas vezes podemos compor um verdadeiro sistema especialista para controlar um jogo, incluindo uma base de conhecimentos e um mecanismo de inferência que atua sobre os fatos e regras de produção definidas. Além de regras usuais, também é possível usar um sistema de regras baseado na lógica nebulosa (Fuzzy Logic), que pode ser bastante útil para que se possa definir regras com elementos como "forte", "fraco", "próximo", "distante", citados no exemplo logo acima. Um exemplo do uso de regras nebulosas em jogos é apresentado por Bittencourt [BIT 2002a, 2002b].

Agentes dotados de uma arquitetura deliberativa podem realizar tarefas de alto nível, como planejar suas trajetórias, deliberar sobre qual a ação é a mais adequada em relação a uma dada situação, mas no entanto, muitas vezes não é necessária uma arquitetura de controle tão sofisticada para controlar um NPC. Uma alternativa é a adoção de uma arquitetura bastante simples, a arquitetura reativa. Este tipo de arquitetura possui vantagens sobre a deliberativa, por ser mais simples e por permitir que sejam introduzidos sensores junto ao modelo de controle, que detectam e reagem a situações não previstas em um plano inicial pré-estabelecido. No próximo item, 4.1, abordaremos a arquitetura de controle reativa.

4.2. Arquitetura Reativa

Uma arquitetura é denominada reativa ou não-deliberativa quando a escolha da ação a ser executada está relacionada de forma direta com a ocorrência de eventos percebidos no ambiente. Nessa arquitetura, o controle das ações do agente é realizado a partir de um comportamento do tipo situação – ação (ou estímulo – resposta). O agente age em um espaço de tempo, com base em uma pequena quantidade de informação, no instante em que recebe ou percebe algum sinal ou estímulo do ambiente. O agente é dotado de sensores, que percebem o ambiente, e de atuadores que permitem que este possa agir. O agente decide como agir baseado em seus sensores, recebendo de volta um "*feed-back*"de suas ações através de uma nova leitura de seus sensores (laço de controle). A Figura 17 apresenta uma representação deste esquema sensorial-motor.

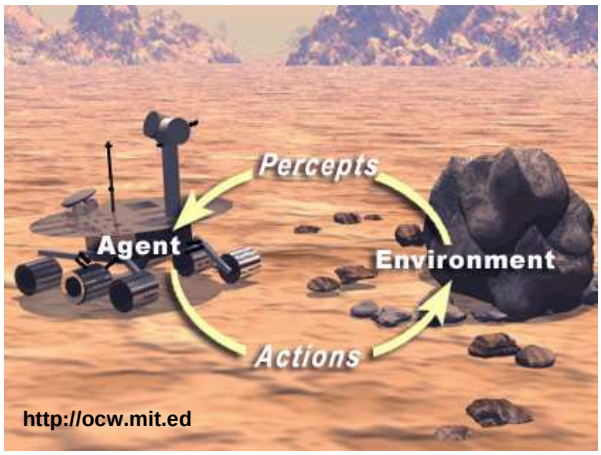


Figura 17: Arquitetura Reativa – Percepção e Ação [MIT 2007]

Nesse tipo de arquitetura não há representação explícita do conhecimento sobre o ambiente. O conhecimento dos agentes é implícito e manifestado através de comportamentos, o que pode restringir a autonomia do agente e sua capacidade de executar tarefas mais complexas. O agente reage a situação do momento, não planejando suas ações em termos de um médio ou longo prazo. Outra característica marcante dessa arquitetura é a ausência de memória das ações passadas, sendo que o resultado de uma ação passada não exerce influência direta sobre as ações futuras.

Os agentes modelados a partir de uma arquitetura reativa, denominados reativos ou não-deliberativos, não possuem capacidade de raciocínio e planejamento, e por isso, são consideradas entidades mais simples do que os agentes deliberativos. São agentes baseados em comportamento, definidos a partir da situação atual do ambiente e do conhecimento atual que possuem do ambiente (fornecido pela sua entrada sensorial). As ações destes agentes são realizadas em resposta a estímulos oriundos do ambiente.

Em relação ao desenvolvimento de NPCs para jogos, é importante destacar que nem sempre todos os jogos irão usar algoritmos mais sofisticados como o A*. Um jogo como o Pac-Man pode ser implementado usando: ***(i)*** um comportamento reativo, onde o fantasma detecta o PacMan pela proximidade e se move na direção onde está o PacMan (sem considerar o mapa); ***(ii)*** um conjunto de regras simples (e.g. se está abaixo do PacMan (presa), move para cima; se está a esquerda, move para a direita, etc); ***(iii)*** um NPC que se desloca adotando uma trajetória pré-definida, apenas cumprindo uma "ronda" que passa por uma rota previamente definida; ***(iv)*** um comportamento deliberativo baseado no A* (e nas variações deste como foi apresentado na seção 3). Estas diferentes formas de controlar o agente (fantasma) podem dar resultados bem satisfatórios, criando jogos com uma boa interação e *GamePlay*.

Entretanto, está claro que a possibilidade de definir rotas ótimas de modo a evitar bloqueios e usar comportamentos inteligentes, capazes de inclusive reagir a situações imprevistas, dará melhores resultados. É por isso que as arquiteturas deliberativa e reativa são modelos usados mais para se criar uma descrição formal do controle de agentes. Do ponto de vista mais prático, a arquitetura de um agente "realmente" inteligente deve integrar estes dois modelos, compondo assim uma arquitetura híbrida.

4.3. Arquitetura Híbrida

Os agentes reativos usualmente não possuem um raciocínio mais elaborado, ao contrário da arquitetura de controle deliberativa, onde os agentes deliberam (raciocinam) sobre suas ações, planejando estas com antecedência. As arquiteturas reativa e deliberativa possuem cada uma suas deficiências e limitações, sendo assim, usualmente são adotadas arquiteturas modulares do tipo híbrido ou hierárquico, combinando assim diferentes módulos de controle [Osório 2004, 2005].

A arquitetura híbrida é portanto aquela em que a escolha da ação é realizada usando uma combinação entre as técnicas utilizadas em arquiteturas deliberativa e reativa. Essa arquitetura foi proposta como alternativa para solucionar as deficiências principais das duas arquiteturas anteriores. A arquitetura deliberativa é tipicamente incapaz de reagir rápida e adequadamente perante situações não previstas. Na arquitetura reativa, o agente é incapaz de descobrir alternativas para o seu comportamento quando a situação do mundo diverge bastante de seus objetivos iniciais. Além disso, o agente não possui capacidade de raciocínio e planejamento.

O objetivo é construir um agente composto por dois subsistemas: (i) o sistema cognitivo, que contém um modelo simbólico do mundo (e.g. mapa), utilizado no planejamento e na tomada de decisões; e (ii) o sistema reativo, capaz de reagir a eventos que ocorrem no ambiente. Os agentes híbridos são normalmente projetados através de uma arquitetura hierárquica (em camadas – Fig. 18). Os níveis mais baixos representam o sistema reativo e são usados para a aquisição de informações do ambiente, de outros agentes ou de outras fontes. Os componentes deliberativo-cognitivos, são responsáveis pelo planejamento e determinação de objetivos, sendo usados nos níveis mais altos.

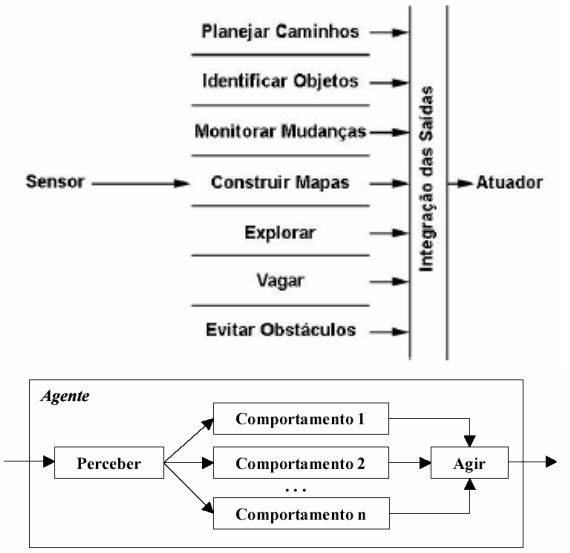


Figura 18: Arquiteturas Híbridas Hierárquicas

Um Autômato Finito pode ser adaptado para considerar as entradas dos sensores (internos e externos) do agente e assim realizar a mudança de estados. Um exemplo desta arquitetura é apresentado na figura 19 abaixo.

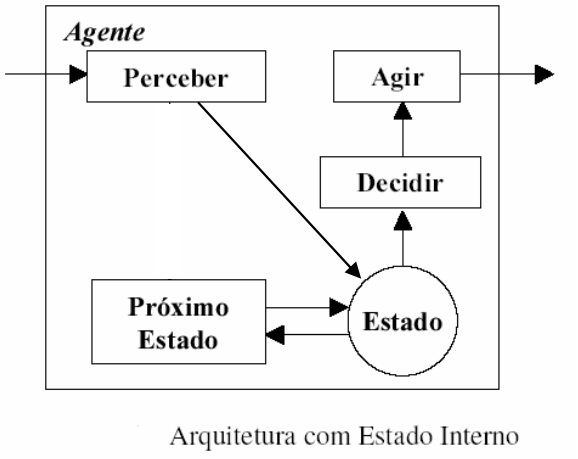


Figura 19: Arquitetura Híbrida tipo FSA

Uma arquitetura de controle híbrida muito usada é baseada na arquitetura BDI (*Belief-Desire-Intention*) [Bratman 1999, Wooldridge 2000], onde os agentes possuem explicitadas suas crenças (incluindo suas percepções), seus desejos e intenções, a partir dos quais realizam o planejamento de suas ações.

Arquitetura BDI é baseada em estados mentais. A idéia básica de uma arquitetura baseada em estados mentais está em descrever o processamento interno de um agente utilizando um conjunto básico destes estados, tais como crenças, desejos, intenções, expectativa, entre outros. A arquitetura BDI (*Belief, Desire, Intention*) é um exemplo de arquitetura baseada em estados mentais, apesar de ser considerada por alguns autores uma estrutura deliberativa, pelo fato de manter uma representação simbólica do ambiente, expressa em crenças, desejos e intenções.

Segundo Rao e Georgeff (1995), *Belief* (crença), *Desire* (desejo) and *Intention* (intenção) representam, respectivamente, a informação (e.g. mapas, percepções), a motivação e o estado deliberativo de um agente. De modo intuitivo, as crenças correspondem à informação que o agente possui sobre seu ambiente, desejos representam opções de estados futuros disponíveis ao agente (motivação) e intenções são os estados futuros que o agente escolheu e com os quais comprometeu-se (deliberação).

4.4. Arquiteturas de Controle: Exemplo de Aplicação em Jogos

Uma interessante aplicação, criada sob a forma de um jogo, e que permite testar diferentes métodos de controle de agentes autônomos é o *Robocode* [Robocode 2007]. Este jogo foi inicialmente desenvolvido pelos pesquisadores da IBM e atualmente é mantido como uma iniciativa de software livre (disponibilizado junto ao *sourceforge*).

O Robocode é um jogo que visa criar os controladores de robôs autônomos simulados (tanques de batalha), que são depois colocados em uma arena de batalha (Figura 20). O usuário deve programar em Java a "inteligência" do controlador, e assim fazer o seu robô lutar em uma batalha contra outros robôs autônomos disponibili-zados por outros usuários. Existem jogos comerciais similares ao Robocode, onde o objetivo também é a criação do controle do robô, e em alguns casos, como no *Roboforge* [Roboforge 2007], o objetivo é configurar a estrutura do robô, seus atuadores e sensores, construindo assim um robô e o seu sistema de controle.

O Robocode é uma plataforma interessante de estudo de arquiteturas de controle de NPCs, pois oferece a possibilidade de implementar comportamentos reativos (baseados em reações sensório-motoras), comportamentos deliberativos (seqüências de ações baseadas em regras) e comportamentos híbridos, combinando sensores e regras de atuação.

Figura 20: Tela de abertura [Robocode 2007]

Entretanto, o Robocode oferece apenas uma arena simples, onde o ambiente é uma arena sem obstáculos e onde comportamentos deliberativos mais sofisticados (como o A*), acabam não sendo explorados. A integração de técnicas de controle de agentes inteligentes, com capacidade sensorial, motora, e cognitiva (deliberativa), é uma das áreas de estudo da I.A. aplicada a jogos que vem sendo muito pesquisada e desenvolvida recentemente.

Atualmente os grandes desafios da I.A. junto aos jogos vem sendo a criação de sistemas multi-agentes, com NPCs capazes de criar estratégias, de se comunicar e coordenar a execução de tarefas, de cooperar a fim de alcançar um objetivo comum. Outro grande desafio é a integração de técnicas de Aprendizado de Máquina (*Machine Learning*) junto aos jogos e aos agentes inteligentes. Estes tópicos serão abordados nas seções seguintes.

5. Sistemas Multi-Agentes: Comunicação Estratégia, Coordenação, Cooperação

Os jogos onde temos vários NPCs que interagem entre si e com o ambiente devem possuir um controle que leve em consideração os seguintes elementos: comunicação, coordenação e cooperação. Existem diversas técnicas de I.A. que buscam promover este tipo de comportamento coletivo, onde podemos destacar os sistemas de *ant colonies*, *flocks* e *swarms* [Reynolds 2007].

Podemos imaginar um conjunto de naves espaciais que buscam atacar o inimigo como sendo um "enxame de abelhas" ou um "bando de pássaros", buscando assim, através de algumas regras e comportamentos pré-estabelecidos, reproduzir estes comportamentos de forma organizada e coordenada. O comportamento coletivo em jogos é um tema bastante complexo e exige muitas vezes soluções específicas, que incluem diferentes técnicas de coordenação, organização e cooperação entre os agentes.

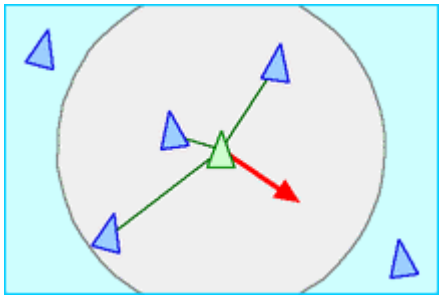
Podemos citar como referências de base nesta área a obra *Multiagent Systems* [Weiss 1999], o livro Sistemas Inteligentes [Rezende 2003 – Cap.11], as bibliotecas *OpenSteer* e *Boids* criadas por Reynolds [Reynolds 2007], e os estudos sobre Simulação de Multidões - *Crowd Simulation*, de Thalmann e Musse [Thalmann 2007].

Não temos a intenção de fazer aqui uma revisão ampla e/ou completa da área de sistemas multi-agentes aplicada a jogos, pois isto demandaria um tutorial específico apenas sobre este tema. Abordaremos aqui apenas dois exemplos de aplicações que podem servir de "inspiração" para o desenvolvimento de aplicações baseadas em multi-agentes: Boids e Robombeiros.

5.1. Boids

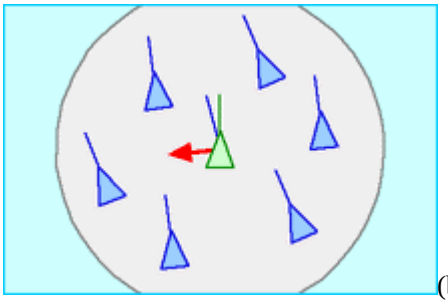
Segundo o criador desta técnica [Reynolds 2007], movimentos complexos podem ser modelados a partir de um conjunto de regras simples, associadas a cada agente, tais como manter uma distância mínima de obstáculos e uma certa velocidade e orientação de movimentação. Os comandos de direcionamento e controle dos Boids são chamados de "*steering*".

O modelo básico de grupos (*flocking*) é composto por 3 regras simples (ver Figura 21) de direcionamento (*steering*), que descrevem como um indivíduo do grupo deve se comportar, em termos de sua posição, orientação e velocidade em relação aos seus companheiros de grupo.



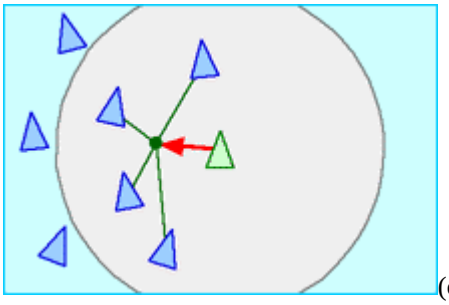
(a)

Separation: Direcionar o movimento dos agentes para evitar um aglomeramento local dos membros do grupo.



(b)

Alignment: Direcionar o movimento de modo a seguir o alinhamento médio (*heading*) dos membros do grupo.



(c)

Cohesion: Direcionar o movimento de modo a ir em direção a posição central média dos membros do grupo.

Figura 21: *Boids* - Regras de controle [Reynolds 2007]

Cada Boid possuirá uma velocidade e orientação que são constantemente ajustadas em função da combinação das 3 regras citadas anteriormente. Além disto, os Boids possuem um raio fixo que define quem será considerado como parte de seu grupo. Usando regras simples como estas, Reynolds criou simulações de pássaros e de grupos que são extremamente simples, ao mesmo tempo, que permitem gerar comportamentos de grupo bastante interessantes.

5.2. Robombeiros

Robombeiros [Pessin 2007, 2007a, 2007b] é um sistema de simulação de combate a incêndios em florestas, baseado em um esquadrão de robôs bombeiros autônomos. Este sistema vem sendo desenvolvido visando o estudo de técnicas de controle e cooperação dos robôs.

Os robôs, assim como foi descrito na seção 3, buscam cercar um incêndio que está se propagando na floresta. Para realizar esta tarefa, um coordenador define a estratégia-macro, direcionando os robôs para suas posições-chave de combate ao incêndio. Cada robô irá executar uma parte desta estratégia coordenada, mas também irá possuir uma certa autonomia de navegação. A estratégia-local de navegação permite que o robô siga para o seu destino pré-especificado, porém é usado um algoritmo baseado em Redes Neurais Artificiais [Mitchell 1997] para controlar o robô localmente, evitando colisões contra obstáculos imprevistos. Os Robombeiros são um exemplo de uma estratégia de controle híbrida, onde temos uma rota (plano) pré-estabelecido, conjugado com um comportamento reativo que evita colisões e direciona o robô para o seu destino. A figura 22 apresenta cenas geradas durante a simulação.

Figura 22: Robombeiros – Simulação Virtual em ambiente 3D (OSG+ODE) [Pessin 2007c]

Concluindo, as estratégias multi-agentes têm um papel muito importante na implementação de jogos, onde muitas vezes estas são definidas previamente através de regras fixas. Algumas aplicações, como os Boids e os Robombeiros, permitem dotar os agentes de um certo grau de autonomia. O interessante é que se possa adotar métodos de aprendizado para assim obter de modo automático controladores inteligentes para os agentes autônomos, estejam estes agentes operando de modo individual ou coletivo.

6. Aprendizado de Máquina em Jogos

A criação de Agentes Inteligentes para jogos que façam uso de técnicas de Sistemas Adaptativos e com Aprendizado de Máquina (*Machine Learning*) [Mitchell 1997, Rezende 2003] é um grande desafio na atualidade.

Os agentes autônomos de um jogo também podem ser controlados baseados em técnicas de *Machine Learning* estudadas na Inteligência Artificial, como por exemplo: Raciocínio baseado em Casos - CBR (*Case Based Reasoning*) [Wangenheim 2003, Kolodner 1993], Redes Neurais - ANN (*Artificial Neural Networks*) [Haykin 2001, Braga 2000], Algoritmos Genéticos - GA (*Genetic Algorithms*) [Mitchell 1996, Rezende 2003], RL (*Reinforcement Learning*) [Mitchell 1997, Sutton 1998], IDT (*Induction of Decision Trees*) [Quinlan 1993, Rezende 2003] e Raciocínio Probabilista (*Bayesian Networks*) [Mitchell 1997].

Entretanto, a integração de técnicas de *Machine Learning* em Jogos é bastante dependente do tipo de problema ou aplicação em que se está inserindo este tipo de técnicas. Podemos ter o aprendizado de máquina em jogos visando, por exemplo: aprender a "imitar" o comportamento humano em um jogo (aprendizado supervisionado), aprender a criar estratégias eficientes de jogo (algoritmos evolutivos para geração de estratégias), aprender o perfil do usuário de modo a melhor adaptar o jogo ao usuário (adaptação de interface e de níveis de dificuldade), aprender a reconhecer uma seqüência de movimentos ou gestos (interfaces gestuais, baseadas no uso de controles do tipo WiiMote [Wiimote 2007] ou mesmo com visão artificial), entre muitas outras possíveis aplicações. Na seção 6.2 iremos abordar alguns exemplos de aplicações de aprendizado de máquina aplicadas em jogos e entretenimento.

6.2. Aprendizado de Máquina em Jogos: Considerações sobre Motores de Jogos

Usualmente os motores de jogos não incluem estas técnicas e ferramentas de aprendizado de máquina, sendo que atualmente podemos encontrar diversas ferramentas e bibliotecas para a implementação destas técnicas, mas usualmente implementadas em pacotes específicos e separados. Por exemplo, os algoritmos genéticos podem ser implementados com o uso da **GALib**[1], as redes neurais artificiais com o uso do **SNNS**[2] e as árvores de decisão com o **C4.5**[3]. Algumas poucas implementações, como o **WEKA**[4] reúnem diversos modelos de aprendizado, entretanto por ser implementada em Java e com fins de uso mais para a pesquisa, sua aplicação em jogos pode acabar ficando um pouco mais restrita. Os motores de jogos que incluem ferramentas de I.A. têm usualmente focado na implementação de técnicas de *path-planning* (como o A*), e de arquiteturas de controle de agentes (reativas, baseadas em regras, baseadas em FSA). Poucas são as *engines* de jogos que incluem módulos de aprendizado de máquina.

Em conseqüência do exposto acima, a maioria das aplicações de aprendizado de máquina para jogos são desenvolvidas de forma específica e customizada para uma determinada aplicação.

6.2. Aplicações de Aprendizado de Máquina em Jogos e de Entretenimento

Vamos apresentar aqui alguns exemplos de aplicações de aprendizado de máquina em jogos, onde não temos a pretensão se fazer uma revisão completa da área, mas sim demonstrar algumas possíveis aplicações deste tipo de técnicas.

6.2.1. *Aprendendo Comportamentos*

Uma maneira de criar comportamentos inteligentes é através do aprendizado supervisionado [Rezende 2003, cap. 4]. Inicialmente um usuário irá jogar um jogo por um certo tempo, onde são registrados os dados referentes ao seu comportamento. Este "arquivo de log" deve conter o estado do jogo (ambiente, agentes, sensores) e a ação realizada pelo jogador (tecla de comando, movimento executado). De posse destas informações podemos criar um agente inteligente que irá aprender a reproduzir o comportamento humano, conforme o esquema apresentado na Figura 23.

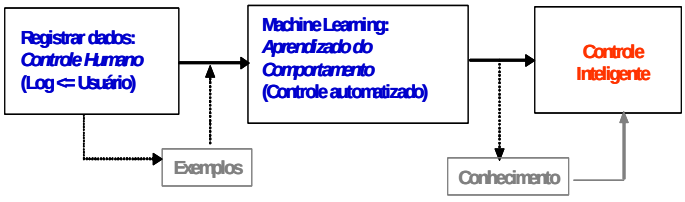


Figura 23: Aprendendo comportamentos através de exemplos

A ferramenta de aprendizado pode ser uma rede neural, ou árvores de decisão, ou qualquer outro método capaz de, a partir dos exemplos de comportamento, criar uma controlador do agente que reproduza este comportamento que lhe foi apresentado.

---

[1] **GALib** - http://lancet.mit.edu/ga/

[2] **SNNS** e **JavaNNS** - http://www-ra.informatik.uni-tuebingen.de/SNNS/

[3] **C4.5** - http://www.rulequest.com/Personal/

[4] **WEKA** - http://www.cs.waikato.ac.nz/ml/weka/

O aprendizado de comportamentos reativos, através deste tipo de abordagem, é uma tarefa extremamente simples. Por exemplo, podemos pegar o jogo do PacMan como aplicação-alvo, e aplicar esta metodologia para criar um PacMan autônomo. O princípio seria bem simples, inicialmente o jogador vai realizar um certo número de partidas contra o computador, onde estará sendo registrado o seu comportamento: Visão do PacMan sobre o ambiente (proximidade dos fantasmas, vitaminas, seu estado atual, etc) e ação do usuário (qual comando o usuário digitou, determinando a direção de deslocamento do PacMan). Uma vez composta a base de exemplos, esta base é fornecida para um algoritmo, uma Rede Neural Artificial, por exemplo, que será responsável por obter um modelo (generalização) do comportamento do usuário. A rede treinada é então colocada no sistema e irá decidir através dos estados de entrada, qual a ação que o PacMan deverá realizar, imitando as decisões e o modo de agir do humano que a treinou.

É importante destacar que neste tipo de aplicação o comportamento final é uma generalização do comportamento do usuário. Este comportamento só poderá ser adequado se: (i) a percepção do agente reproduzir de certa forma a percepção do usuário sobre o estado do jogo; (ii) as ações do usuário forem coerentes e com um bom desempenho. Note que se o usuário jogar mal, o agente irá aprender a imitar este jogador, ou seja, possivelmente também irá jogar mal.

6.2.2. *Aprendendo Estratégias*

Em certas aplicações é interessante que o aprendizado possa ocorrer de modo independente do usuário, onde o agente "descobre" ou "evolui" sozinho uma estratégia de como se comportar. A evolução de estratégias pode ser obtida com a aplicação de algoritmos evolutivos (e.g. Algoritmos Genéticos), onde é fornecido ao agente apenas um *feedback* de seu desempenho final (e.g. pontuação, tempo de vida, nro. de inimigos capturados). Neste caso, não é apresentado ao agente "como" ele deve se comportar, mas apenas indicamos uma medida que fornece a aptidão do agente (*fitness*) em realizar uma certa tarefa.

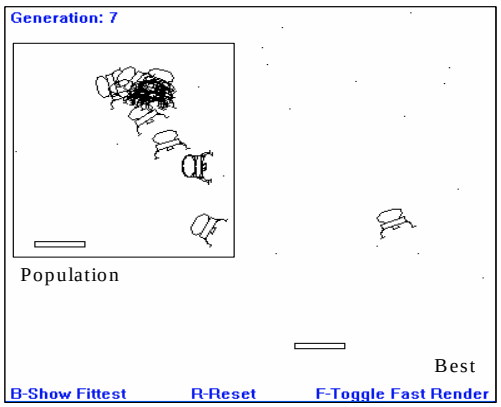


Figura 24: Aprendizado por evolução – Lunar Lander [Buckland 2002]

Um exemplo deste tipo de comportamento é apresentado por Buckland [Buckland 2002] no jogo "*Lunar Lander*", onde o comportamento do módulo lunar é aprendido através da evolução do sistema de controle deste módulo. O objetivo é evoluir uma seqüência de comandos de controle do motor e de deslocamentos que permitam ao módulo lunar pousar correta e suavemente em uma base. Após várias gerações (tentativas), com a seleção dos mais aptos, chegamos a um controlador que possuirá em seu "código genético" a boa seqüência de controle.

Este jogo demonstra bem o uso de Algoritmos Genéticos para evoluir um controlador, de modo não supervisionado, baseado apenas no *score* final obtido por cada um dos elementos da população. Este tipo de técnica tem sido usada para evoluir comportamentos e estratégias em agentes inteligentes, mas entretanto é relativamente custosa, pois exige que sejam "jogadas" muitas partidas para que seja feita a avaliação da pontuação, seleção e evolução dos agentes que possuem o melhor "código interno" (representando uma solução para o problema tratado) e conseqüentemente a melhor aptidão (*fitness*) frente ao problema.

6.2.3. *Aprendendo Perfis*

Uma outra possibilidade de aplicação de técnicas adaptativas é o uso de técnicas da área de sistemas de recomendação e análise de perfil do usuário. Atualmente muitas empresas adotam sistemas que analisam o comportamento do usuário, como por exemplo, o site da "*amazon.com*", determinando as preferências e hábitos de seus clientes. A partir deste perfil do usuário, é feita uma adaptação do conteúdo das páginas Web, com sugestões e recomendações específicas para um determinado usuário. Este tipo de conceito de adaptação de páginas Web pode ser transposto para os jogos, onde também podemos monitorar os jogadores, constituir um perfil dos mesmos, e oferecer um conteúdo adaptado a cada jogador.

Um exemplo de aplicação que permite a adaptação de um ambiente tridimensional de um jogo, através da inserção, remoção e reposicionamento de objetos (e.g. munição, bônus, armadilhas, etc), considerando o perfil de comportamento de cada jogador, foi proposto baseado no modelo AdapTIVE [Santos 2005]. Este modelo apresenta uma forma de adaptar ambientes 3D, conforme o perfil do jogador, onde este perfil é construído pelo monitoramento das ações do jogador durante uma rodada do jogo.

De um modo geral, podemos classificar os sistemas como o AdapTIVE, como sendo técnicas de aprendizado e adaptação baseadas no perfil do usuário. Este tipo de aplicações ainda é pouco explorado em jogos, mas assim como ocorre na área de *e-commerce*, a área de jogos vem cada vez mais se preocupando em proporcionar um melhor entretenimento/satisfação aos seus usuários. A adaptação baseada no perfil é uma das técnicas que certamente irá contribuir neste sentido.

6.2.4. *Aprendendo a Reconhecer Movimentos*

O reconhecimento de movimentos recebeu uma maior atenção desde o lançamento da console Wii da Nintendo que possui o WiiMote [Wimote 2007], um controle capaz de registrar movimentos. O WiiMote possui um sensor que registra deslocamentos no espaço 3D, e assim, podemos ter jogos onde o gesto se torna um componente importante do jogo, como por exemplo: jogar tênis (controle de movimento da raquete), jogar golfe ou baseball (controle do movimento do taco), jogar boliche ou baseball (controle do arremesso da bola), entre muitas outras opções.

A inclusão de interfaces capazes de capturar movimentos e gestos em jogos, abriu caminho para a implementação de sistemas capazes de reconhecer o movimento. Algumas empresas, como a AILive [AI Live 2007], tem apostado na criação de ferramentas, que incluem técnicas de I.A., para auxiliar na criação de jogos com funcionalidades de reconhecimento de movimentos.

Mas não é apenas a console Wii que permite explorar este tipo de recurso. Um programa de demonstração, bastante simples, que permite o reconhecimento de "desenhos" feitos com o mouse, é apresentado por Buckland [Buckland 2002] em seu livro "AI techniques for game programming" (que inclui programas demonstrando esta funcionalidade). Através do uso de uma Rede Neural Artificial, é feito um treinamento para reconhecer certos movimentos de traçado de desenhos muitos simples feitos com o mouse (e.g. seta para cima, para baixo, traços e quadrados). Este tipo de reconhecedor é muito simples, porém permite ver o potencial que os algoritmos de aprendizado de máquina (como as Redes Neurais) possuem para criar ferramentas de reconhecimento de movimentos e gestos.

O sucesso da console Wii é uma prova de que a inovação em termos de interfaces com o usuário pode revolucionar a área de jogos. Junto com as novas interfaces vem novos desafios de programação, como o reconhecimento de gestos, e sem dúvida as técnicas de aprendizado de máquina possuem um papel muito importante nesta área que apenas começa a ser explorada.

As aplicações de Aprendizado de Máquina (*Machine Learning*) em Jogos não se esgotam nestes poucos exemplos citados aqui. Existem muitas outras possibilidades já exploradas e ainda a serem exploradas. Podemos constatar a importância desta área pela sua recente expansão, onde um dos exemplos deste crescimento é visto em conferências como o IEEE CIG – *Symposium on Computational Intelligence in Games.* Nesta conferência temos inclusive competições de *Machine Learning* para criar agentes inteligentes usados para o controle autônomo de carros, aviões ou PacMans!

Por fim, a questão da performance também é de grande importância quando abordamos a implementação de sistemas adaptativos e com aprendizado em jogos. Este é ainda um tema de pesquisa atual na área, com poucas implementações realmente funcionais (jogos comerciais) que se usam de modo prático o aprendizado de máquinas. Entretanto, devemos destacar que em um horizonte de curto prazo os jogos deverão cada vez mais integrar este tipo de ferramentas, seja pelo aumento da performance das CPUs, ou mesmo pela inclusão de processadores adicionais e dedicados a I.A. Deste modo podemos prever que o comportamento dos NPCs e o funcionamento dos agentes inteligentes vão se tornar o cada vez mais realistas.

7. Considerações Finais

Neste tutorial foram apresentados métodos e algoritmos de I.A. aplicados a jogos. Podemos constatar a grande variedade de técnicas e possibilidades de aplicação da I.A. junto aos jogos digitais. Apesar dos motores de I.A. para jogos estarem atualmente sendo desenvolvidos e aperfeiçoados continuamente, ainda não contamos com uma grande variedade de ferramentas e soluções nesta área.

Existem algumas soluções comerciais, como o DirectIA e o Dark A.I., e iniciativas de código livre, como o OpenAI, entretanto todas são usualmente focadas para fins mais específicos. Estas soluções não implementam uma ampla gama de técnicas de I.A., fornecendo apenas um sub-conjunto mais restrito de técnicas, como algumas das descritas neste tutorial.

O DirectIA [DirectIA 2006] oferece um kernel para a implementação de comportamentos autônomos e adaptativos, podendo ser integrado em aplicações através do uso de um SDK. O Dark A.I. [Dark IA 2006] é um módulo de extensão da ferramenta RAD (*Rapid Application Development*) para jogos, o DarkBasic Pro. Esta ferramenta oferece funções para a criação de trajetórias com desvio de obstáculos, usando o A*, além de oferecer também funções para a implementação de comportamentos inteligentes em agentes (e.g. comportamento reativo). O OpenAI [OpenAI 2006] é uma iniciativa de código aberto, que visa oferecer ferramentas e implementações de técnicas de Inteligência Artificial. O OpenAI oferece atualmente implementações de Redes Neurais Artificiais, Algoritmos Genéticos e Autômatos Finitos.

A maioria dos motores profissionais de jogos apresenta alguma funcionalidade que implementa funções de controle de trajetórias ou comportamento de NPCs, entretanto podemos afirmar que ainda não existe disponível alguma solução mais completa de I.A. que integre desde a I.A. clássica, passando pelos algoritmos de planejamento de trajetória, controle de comportamento de agentes, aprendizado e adaptação, assim como comportamentos inteligentes de grupos de

agentes. O desenvolvedor que buscar ter acesso a tais ferramentas terá que integrar soluções de diferentes origens ou desenvolver sua própria solução.

Agradecimentos

Os autores gostariam de agradecer a UNISINOS, CNPq e FAPERGS pela concessão de bolsas (IC) e de auxílios financeiros que permitiram o desenvolvimento deste trabalho. Gostaríamos também de agradecer as múltiplas contribuições de nossos colegas de trabalho, Farlei Heinen, Milton Heinen, João Ricardo Bittencourt, Soraia Musse e Cássia Trojahn dos Santos, que ajudaram a consolidar mais ainda este trabalho.

Referências

AILIVE – LiveMove Product [online], Disponível em: http://www.ailive.net/liveMove.html [Acessado 10 de Setembro de 2007].

A-STAR WIKIPEDIA - *A-Star Search Algorithm* [online], Disponível em: http://en.wikipedia.org/wiki/A*_search_algorithm [Acessado 5 de Agosto de 2007].

BITTENCOURT, JOÃO R.; OSÓRIO, F. S. GNU MAGES: UM AMBIENTE PARA SIMULAÇÃO DE MÚLTIPLOS AGENTES AUTÔNOMOS, COOPERATIVOS E COMPETITIVOS. IN: ANAIS DO III WORKSHOP SOBRE SOFTWARE LIVRE. PORTO ALEGRE, 2002. (BITTENCOURT 2002A)

BITTENCOURT, JOÃO R.; OSÓRIO, F. S. FUZZYF - FUZZY LOGIC FRAMEWORK: UMA SOLUÇÃO SOFTWARE LIVRE PARA O DESENVOLVIMENTO, ENSINO E PESQUISA DE APLICAÇÕES DE INTELIGÊNCIA ARTIFICIAL MULTIPLATAFORMA. IN: ANAIS DO III WORKSHOP SOBRE SOFTWARE LIVRE. PORTO ALEGRE, 2002. (BITTENCOURT 2002B)

BITTENCOURT, J. R. ; OSÓRIO, FERNANDO S. . MOTORES DE JOGOS PARA CRIAÇÃO DE JOGOS DIGITAIS - GRÁFICOS, ÁUDIO, INTERFACE, REDE, INTELIGÊNCIA ARTIFICIAL E FÍSICA. IN: V ESCOLA REGIONAL DE INFORMÁTICA DE MINAS GERAIS, BELO HORIZONTE. ANAIS DA V ERI-MG SBC. BELO HORIZONTE : PUC MINAS, 2006. V. 1. P. 1-36. 2006. (TUTORIAL) WEB: HTTP://OSORIO.WAIT4.ORG/PUBLICATIONS/BITTENCOURT-OSORIO-SIC-BOOK2002.PDF (LOGIN: GUEST / UNISINOS)

BRAGA, ANTÔNIO; LUDERMIR, TERESA; CARVALHO, ANDRÉ. REDES NEURAIS ARTIFICIAIS: TEORIA E APLICAÇÕES. LTC, 2000. 262P.

BRATMAN, M. E. (1999). INTENTION, PLANS, AND PRACTICAL REASON. CSLI PUBLICATIONS. ISBN 1-57586-192-5.

BUCKLAND, MAT. AI TECHNIQUES FOR GAME PROGRAMMING. PREMIER PRESS, GAME DEVELOPMENT SERIES. 2002. 481 P.

CHARLES-RIVER. GAME PROGRAMMING GEMS BOOK SERIES – CHARLES RIVER MEDIA. DISPONÍVEL EM: HTTP://WWW.GAMEPROGRAMMINGGEMS.COM ACESSO EM: 10 JUL. 2006.

DARK A.I. - DARKBASIC PRO. THE GAME CREATORS. WEB: HTTP://DARKBASICPRO.THEGAMECREATORS.COM/?F=DARK_AI ACESSO EM: 10 JUL. 2006.

DIRECTIA – MASA SCI. DISPONÍVEL EM: HTTP://WWW.MASA-SCI.COM/DIRECTIA.HTM ACESSO EM: 10 JUL. 2006.

DECHTER, R,. AND PEARL , J.,1985, *Generalized Best-First Search Strategies and the Optimality of A**. University of California.

DELOURA MARK. (ED.) . GAME PROGRAMMING GEMS (VOL.I). CHARLES RIVER MEDIA ED. 2000. 550 P. ISBN 1584500492.

DIJKSTRA WIKIPEDIA [ONLINE], DISPONÍVEL EM: HTTP://EN.WIKIPEDIA.ORG/WIKI/DIJKSTRA%27S_ALGORITHM [ACESSADO 10 DE SETEMBRO DE 2007].

DUDEK, G.; JENKIN, M. COMPUTATIONAL PRINCIPLES OF MOBILE ROBOTICS. CAMBRIDGE: CAMBRIDGE UNIVERSITY, 2000. 280 P. ISBN 0-521-56876-5

DYBSAND, ERIC. A FINITE-STATE MACHINE CLASS. IN: DELOURA MARK. (ED.), GAME PROGRAMMING GEMS (VOL.I). CHARLES RIVER MEDIA ED. 2000. 550 P. (CAP. 3.1).

FUNGE, J. D. AI FOR GAMES AND ANIMATION: A COGNITIVE MODELING APPROACH. NATICK, MA: AK PETERS, 1999. 212 P.(WEB: HTTP://JFUNGE.GOOGLEPAGES.COM/)

FUNGE, JOHN. AI4GAMES. DISPONÍVEL EM: HTTP://WWW.AI4GAMES.ORG/ ACESSO EM: 10 JUL. 2006.

GAMEPLAY WIKIPEDIA – *Gameplay* [online], Disponível em: http://en.wikipedia.org/wiki/Gameplay [Acessado 14 de Agosto de 2007].

HAYKIN, Simon. Neural Networks: A Comprehensive Foundation. Prentice-Hall. 2nd Ed, 1999. 842p. (Tradução: Neural Network: Princípios e Prática. Bookman, 2001).

HEINEN, F. (2000) "ROBÓTICA AUTÔNOMA: INTEGRAÇÃO ENTRE PLANIFICAÇÃO E COMPORTAMENTO REATIVO". EDITORA UNISINOS. DISPONÍVEL NA WEB EM: HTTP://NCG.UNISINOS.BR/ROBOTICA/LIVRO.HTM.

JUNG, C. R; OSÓRIO, F. S.; KELBER, C.; HEINEN, F. COMPUTAÇÃO EMBARCADA: PROJETO E IMPLEMENTAÇÃO DE VEÍCULOS AUTÔNOMOS INTELIGENTES. JAI - JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, CONGRESSO DA SBC 2005. SÃO LEOPOLDO: RS. (JAI05 – TUTORIAL). WEB: HTTP://INF.UNISINOS.BR/~OSORIO/PALESTRAS/JAI2005.HTML

KOLODNER, JANET. CASE-BASED REASONING. SAN MATEO: MORGAN KAUFMANN, 1993. 668 P. ISBN 1-55860-237-2

LESTER, P., 2004. *A* Pathfinding for Beginners* [online], Disponível em: http://www.policyalmanac.org/games/aStarTutorial_port.htm [Acessado em 11/08/2007].

LESTER, PATRICK. *A* PATHFINDING FOR BEGINNERS*. [online] GAMEDEV–WEB: HTTP://WWW.GAMEDEV.NET/REFERENCE/ARTICLES/ ARTICLE2003.ASP [Acessado em 14/08/2007]

MATTHEWS, JAMES. [ONLINE], DISPONÍVEL EM: HTTP://WWW.GENERATION5.ORG/CONTENT/2000/ASTAR.ASP A* DEMONSTRATOR SOFTWARE [ACESSADO 10 MARÇO DE 2003]

MINI-MAX WIKIPEDIA [ONLINE], DISPONÍVEL EM: HTTP://PT.WIKIPEDIA.ORG/WIKI/MINIMAX (SIMPLES) OU HTTP://EN.WIKIPEDIA.ORG/WIKI/MINIMAX [ACESSADO 10 DE SETEMBRO DE 2007].

MIT OCW - TECHNIQUES IN ARTIFICIAL INTELLIGENCE [ONLINE]. OPEN COURSEWARE MIT AVAILABLE AT WEB: HTTP://OCW.MIT.EDU/OCWWEB/ELECTRICAL-ENGINEERING-AND-COMPUTER-SCIENCE/6-825TECHNIQUES-IN-

ARTIFICIAL-INTELLIGENCEFALL2002/COURSEHOME/ [ACESSADO 10 DE SETEMBRO DE 2007]

MITCHELL, MELANIE. AN INTRODUCTION TO GENETIC ALGORITHMS. MIT PRESS, 1996. 209P.

MITCHELL, T. M. MACHINE LEARNING. NEW YORK: MCGRAW-HILL. SERIES IN COMPUTER SCIENCE, 1997. 414P.

OPENAI. DISP. EM: HTTP://OPENAI.SOURCEFORGE.NET/ ACESSO EM: 10 JUL. 2006.

OSÓRIO, F.S. ; MUSSE, S.R. ; ET AL. AMBIENTES VIRTUAIS INTERATIVOS E INTELIGENTES: FUNDAMENTOS, IMPLEMENTAÇÃO E APLICAÇÕES PRÁTICAS. IN: A.M.S. ANDRADE; A.T. MARTINS; R.J.A. MACEDO(ORG.). JAI - JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, CONGRESSO DA SBC 2004. SALVADOR: TECART ED., 2004, V.2, P.239-288. (JAI04 – TUTORIAL). WEB: HTTP:// INF.UNISINOS.BR/~OSORIO/PALESTRAS/JAI04-AVII.HTML

OSÓRIO, FERNANDO S. ; MUSSE, SORAIA RAUPP ; ET AL. INTELLIGENT VIRTUAL REALITY ENVIRONMENTS (IVRE): PRINCIPLES, IMPLEMENTATION, INTERACTION, EXAMPLES AND PRACTICAL APPLICATIONS. IN: XAVIER FISCHER. (ORG.). VIRTUAL CONCEPT (PROCEEDINGS - TUTORIALS). 1 ED. BIARRITZ, FRANÇA: ESTIA - IEEE - SPRINGER-VERLAG, 2005, V. 1, P. 1-64. VC05 – TUTORIAL. WEB: HTTP://OSORIO.WAIT4.ORG/PUBLICATIONS/PAPERS-OSORIO.HTM

OSÓRIO, FERNANDO S. ; MUSSE, SORAIA RAUPP ET AL. INCREASING REALITY IN VIRTUAL REALITY APPLICATIONS THROUGH PHYSICAL AND BEHAVIOURAL SIMULATION. IN: X. FISCHER. (ORG.). PROCEEDINGS OF THE VIRTUAL CONCEPT CONFERENCE 2006. 1 ED. BERLIM: ESTIA - VIRTUAL CONCEPT - SPRINGER VERLAG, 2006, V. 1, P. 1-45. (VC-SS06 – TUTORIAL). WEB: HTTP://OSORIO.WAIT4.ORG/PUBLICATIONS/PAPERS-OSORIO.HTM

PACMAN (AVAILABLE AT IGOOGLE) [ONLINE]. DISPONÍVEL EM: HTTP://WWW.SCHULZ.DK/PACMAN.HTML?LANG=EN [ACESSADO 10 DE SETEMBRO DE 2007].

PESSIN, G. ; OSORIO, F. S. ; MUSSE, S. ; NONNEMMACHER, V. ; FERREIRA, S. S. . SIMULAÇÃO VIRTUAL DE AGENTES AUTÔNOMOS PARA A IDENTIFICAÇÃO E CONTROLE DE INCÊNDIOS EM RESERVAS NATURAIS. IN: SYMPOSIUM ON VIRTUAL AND AUGMENTED REALITY, 2007, PETRÓPOLIS. IX SYMPOSIUM ON VIRTUAL AND AUGMENTED REALITY, 2007. V. 1. P. 136-245. WEB: HTTP://PESSIN.GOOGLEPAGES.COM/

PESSIN, G. ; OSORIO, F. S. ; MUSSE, S. ; NONNEMMACHER, V. ; FERREIRA, S. S. . UTILIZANDO AGENTES AUTÔNOMOS COM APRENDIZADO PARA A IDENTIFICAÇÃO E COMBATE DE INCÊNDIOS EM ÁREAS FLORESTAIS. IN: VII SIMPÓSIO DE INFORMÁTICA DO PLANALTO MÉDIO (SIPM), PASSO FUNDO, RS. VII SIMPÓSIO DE INFORMÁTICA DO PLANALTO MÉDIO (SIPM), 2007. WEB: HTTP://PESSIN.GOOGLEPAGES.COM/ (PESSIN 2007A)

PESSIN, G. PROPOSTA DE DISSERTAÇÃO DE MESTRADO DO PIPCA: EVOLUÇÃO DE ESTRATÉGIAS E CONTROLE INTELIGENTE EM SISTEMAS MULTI-ROBÓTICOS ROBUSTOS. UNISINOS, MAIO 2007. (PESSIN 2007B)

PESSIN, G. ; OSORIO, F. S. ; MUSSE, S. ; NONNEMMACHER, V. ; FERREIRA, S. S. . DESENVOLVIMENTO DE UM SISTEMA MULTI-ROBÓTICO COM CONTROLE INTELIGENTE APLICADO NA IDENTIFICAÇÃO E COMBATE DE INCÊNDIOS EM ÁREAS FLORESTAIS. IN: OBSERVATÓRIO 2007 – UNOCHAPECÓ.. (PESSIN 2007C). WEB: HTTP://PESSIN.GOOGLEPAGES.COM/

POTTINGER, D.C., 2000. *Terrain Analisys in RealTime Startegy Games* [online], Disponível em: www.gamasutra.com/features/gdcarchive/2www.gamasutra.com/features/gdcarchive/2000/pottinger.doc000/pottinger.doc [Acessado 14 de Agosto de 2007].

QUINLAN, J. R. C4.5: Programs for machine learning. San Mateo: Morgan Kaufmann Publishers, 1993. 302p

RABIN, STEVE (ED.). AI GAME PROGRAMMING WISDOM 2. CHARLES RIVER MEDIA; DEC. 2003, 732 P.

RABIN, STEVE (ED.). AI GAME PROGRAMMING WISDOM. CHARLES RIVER MEDIA; 2002, 672 P. (WEB: HTTP://WWW.AIWISDOM.COM/)

RABIN, STEVE (ED.). *AI GAME PROGRAMMNG WISDOM*. CHARLES RIVER MEDIA ED. 2002. 672P.

RABIN, STEVE. AI GAME PROGRAMMING WISDOM. CHARLES RIVER MEDIA, 2002. 672P.

RAO, A. AND GEORGEFF, M. (1995) "BDI AGENTS: FROM THEORY TO PRACTICE". PROCEEDINGS OF THE 1ST INTERNATIONAL CONFERENCE ON MULTI-AGENTS SYSTEMS, SAN FRANCISCO, CA, JUNE.

REZENDE, SOLANGE (ED.). SISTEMAS INTELIGENTES : FUNDAMENTOS E APLICAÇÕES. BARUERI: EDITORA MANOLE, 2003. 525 P.

REYNOLDS - CRAIG W. REYNOLDS – BOIDS, FLOCKS AND STEERING (OPENSTEER). AVAILABLE IN THE WEB SITE: HTTP://WWW.RED3D.COM/CWR/ (ACESSADO EM AGOSTO/2007)

ROBOCODE GAME [ONLINE]. PROGRAMMING AGENT AI. WEB: HTTP://ROBOCODE.SOURCEFORGE.NET/ [ACESSADO 10 DE SETEMBRO DE 2007].

ROBOFORGE GAME [ONLINE]. LIQUID EDGE GAMES LTD. WEB: HTTP://WWW.ROBOFORGE.NET/ [ACESSADO 10 DE SETEMBRO DE 2007].

RUSSEL, R.; NORVIG, P. *ARTIFICIAL INTELLIGENCE: A MODERN APPROACH* ENGLEWOOD CLIFFS, PRENTICE HALL, 1995. 932P.

SANTOS, CÁSSIA TROJAHN DOS ; OSÓRIO, FERNANDO S. APPLYING THE ADAPTIVE MODEL IN GAMES. IN: SBGAMES / WJOGOS, 2005, SÃO PAULO. ANAIS DO SBGAMES 2005 / WJOGOS. SÃO PAULO : USP - SBC, 2005. V. 1. P. 264-269. (SBGAMES/WJOGOS 2005 - ARTIGO).

SCHWAB, BRIAN. AI GAME ENGINE PROGRAMMING. CHARLES RIVER MEDIA. 2004. 624 P.

SUDOKU WIKIPEDIA [ONLINE], DISPONÍVEL EM: HTTP://PT.WIKIPEDIA.ORG/WIKI/SUDOKU [ACESSADO 10 DE SETEMBRO DE 2007].

SUTTON, RICHARD S. AND BARTO, ANDREW G. REINFORCEMENT LEARNING: AN INTRODUCTION. MIT PRESS (A BRADFORD BOOK), CAMBRIDGE, MA, 1998.

THALMANN, DANIEL, MUSSE, SORAIA RAUPP (EDS). CROWD SIMULATION. SPRINGER-VERLAG. (COMPUTER IMAGING, VISION, PR AND GRAPHICS) 2007. 270 P., ISBN: 978-1-84628-824-1.

WANGENHEIM, CHRISTIANE GRESSE VON; WANGENHEIM, ALDO VON. RACIOCÍNIO BASEADO EM CASOS. BARUERI: MANOLE, 2003. 293 P. ISBN 85-204-1459-1

WATT, ALAN; POLICARPO, FABIO. 3D GAMES: REAL-TIME RENDERING AND SOFTWARE TECNOLOGY. ADDISON-WESLEY, 2000, 800 P.

WEISS, GERHARD. MULTIAGENT SYSTEMS: A MODERN APPROACH TO DISTRIBUTED ARTIFICIAL INTELLIGENCE. MIT PRESS, 1999. 643 P.

WOOLDRIDGE, M. (2000). REASONING ABOUT RATIONAL AGENTS. THE MIT PRESS. ISBN 0-262-23213-8.

WIIMOTE WIKIPEDIA [ONLINE], DISPONÍVEL EM: HTTP://EN.WIKIPEDIA.ORG/WIKI/WIIMOTE [ACESSADO 10 DE SETEMBRO DE 2007].

WINSTON, PATRICK H. *ARTIFICIAL INTELLIGENCE.* (3RD. EDITION) ADDISONS-WESLEY PUBLISHING, 1992, 737P.