2
Inteligência Artificial em Jogos

O termo "IA em jogos" (ou "*Game AI*") é geralmente usado de forma muito ampla, variando desde a representação e controle de comportamento de personagens não controlados pelo jogador no jogo (*non-player characters* - NPCs) a problemas de controle de mais baixo nível que geralmente seriam considerados campo da teoria de controle [153]. Algumas vezes até características de modelagem física e detecção de colisão são também englobadas no rótulo "IA". Embora haja uma conexão quando da movimentação dos personagens, modelagem física é um campo em separado e melhor deixada a cargo de ferramentas específicas [2].

Os primeiros jogos que tentaram implementar uma certa "inteligência" nos personagens controlados pelo computador o faziam de maneiras bastante precárias, através de "regras" estímulo-resposta implementadas por meio de estruturas do tipo if-then-else *hardcoded* dentro do código. Esta abordagem causava grandes problemas no caso de necessidade de alterações e não permitia grande flexibilidade.

Percebeu-se rapidamente a necesidade de melhores técnicas de modelagem e para a representação e controle do comportamento desses personagens, por exemplo, permitindo aos NPCs ter objetivos e incorporarem um estado interno que permita perseguir esses objetivos.

Um primeiro passo no processo de entender as necessidades encontradas no desenvolvimento de um motor (ou *middleware*) de inteligência artificial aplicado a jogos é, então, realizar a análise do domínio através do levantamento das técnicas de IA mais utilizadas em jogos. Nas seções seguintes serão apresentadas técnicas comumente utilizadas em jogos digitais, técnicas mais avançadas e o uso de agentes em jogos.

2.1 Técnicas utilizadas

Tradicionalmente, desenvolvedores de jogos digitais fazem uso sempre de um mesmo conjunto de técnicas simples na implementação das funcionalidades de inteligência artificial em jogos, especialmente: máquinas de estado finitas (*Finite State Machines* - FSMs) e máquinas de estado *fuzzy* (*Fuzzy Finite Sate Machines* - FuSMs) [109][151], que são basicamente um conjunto de estados e transições entre estes, usadas para representar comportamentos; o algoritmo A* [109], usado para calcular caminhos; e algumas técnicas que podem ser chamadas de *Artificial Life* (A-Life), tais como comportamentos de movimentação (*steering behaviours*) [188] que podem dar maior realismo à movimentação.

Mas mesmo com esse pequeno conjunto de técnicas, é possível alcançar resultados bastante satisfatórios. Alguns jogos também fazem uso de árvores de decisão e regras de produção quando algum tipo de raciocínio sobre o mundo de jogo é necessário.

No entanto, com o avanço dos jogos digitais, técnicas mais avançadas vêm também sendo utilizadas. A seguir serão comentadas algumas das técnicas mais utilizadas na criação de jogos digitais e algumas técnicas promissoras que vêm se tornando mais populares.

2.1.1 Máquina de Estados Finita

Uma das formas de representação mais comuns (e a mais largamente utilizada) para comportamentos dos personagens de um jogo, é representá-los através do uso de máquinas de estado finitas (*Finite State Machines* - FSMs).

Uma máquina de estados é basicamente composta por um conjunto de estados e um conjunto de regras de transição entre estes estados (que usualmente refletem algum evento no mundo do jogo). A utilização desse modelo de representação e controle do comportamento de agentes consiste basicamente em representar, através dos estados, as ações possíveis ao agente, de forma que as regras de transição representem as condições que ao serem verificadas para avaliar se o agente deve mudar de estado. A Figura 2.1 mostra um exemplo de uma máquina de estados representando o comportamento de um agente inteligente simples em um jogo do gênero *First Person Shooter* (FPS).

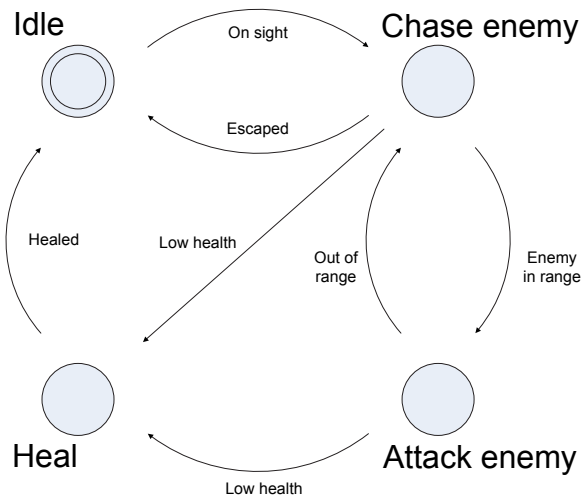

Figura 2.1: Máquina de estados simples de um personagem de um FPS.

Na execução do comportamento, a máquina de estados encontra-se inicialmente em seu estado inicial; a cada iteração, as regras das transições que deixam o estado corrente são avaliadas; se alguma delas for disparada, a transição é então realizada, e o estado de chegada desta regra se torna o novo estado corrente. As ações associadas ao estado serão então executadas pelo agente.

O uso de FSMs para definir o comportamento de agentes em jogos é bastante popular devido ao fato de necessitarem de pouco poder de processamento e ser fácil e intuitivo definir comportamentos por meio dessa abordagem, principalmente com o auxílio de ferramentas visuais.

Entretanto, as limitações da abordagem de máquinas de estados finitas no projeto de agentes inteligentes são bem conhecidas, FSMs são limitadas principalmente por explosões combinatórias (com o aumento de complexidade do ambiente, também crescem o número de estados e de transições pois a FSM tem que prever todos os casos e situações possíveis no ambiente) e pelos comportamentos potencialmente repetitivos, pois as FSMs têm um conjunto fixo de estados e transições e se uma mesma situação acontece duas vezes, o comportamento ativado será o mesmo em ambas as situações.

Uma abordagem bastante usada para mitigar o problema das explosões combinatórias é a criação de máquinas de estados finitas hierárquicas (*Hierarchical Finite State Machines* - HFSMs), onde cada estado pode ser uma nova FSM. Essa abordagem é bastante poderosa [154], permitindo uma melhor organização e modularização dos comportamentos, de modo a minimizar o impacto do crescente número de estados. Contudo, não elimina

o problema e também não trata o problema dos comportamentos repetitivos e previsíveis.

2.1.2 Sistema Baseado em Regras

Uma outra forma de representação de comportamento muito utilizada em jogos, e que também é mais flexível que a abordagem puramente estímulo-resposta, são os chamados sistemas baseados em regras (*Rule Based Systems* - RBSs). RBSs apresentam algumas vantagens como: correspondem ao modo como as pessoas normalmente pensam sobre conhecimento, são bastante expressivos e permitem a modelagem de comportamentos complexos, modelam o conhecimento de uma maneira modular (como por exemplo organizando a base por objetivos como proposto em [110]), são fáceis de escrever (e depurar, quando comparados a árvores de decisão por exemplo) e são muito mais concisos que máquinas de estados finitos.

Em um RBS, o conhecimento é definido através de um conjunto de parâmetros (variáveis) e um conjunto de regras que trabalham sobre esses parâmetros, de modo que durante a "tomada de decisão" essas regras são então processadas.

Por exemplo, supondo que o conjunto de parâmetros seja definido (usando um exemplo similar ao do jogo usado como cenário na Figura 2.1) como: {distância do inimigo, energia do agente, energia do inimigo, quantidade de munição}. E uma vez definidos os parâmetros de entrada, seja também concebido o conjunto de ações possíveis ao agente: {fugir, procurar, atirar}.

As regras de inferência têm um formato onde o antecedente (ou condição) da regra representa um conjunto de cláusulas condicionais sobre as variáveis, e o consequente (ou conclusão) representa uma ação a ser executada pelo agente. As condições são ligadas por conjunções lógicas (conectivos E e OU). Uma regra válida seria:

SE (distancia **do** inimigo $>$ 100
E energia **do** agente $>$ 50
E energia **do** inimigo $<$ 50)
ENTAO procurar

Na execução do comportamento representado por regras de inferência, primeiramente, o conjunto de regras é percorrido e para cada regra, a situação atual do mundo é avaliada para determinar se a regra será disparada (todas as condições da regra em questão são satisfeitas); se isso

acontecer, a ação a ser tomada pelo agente no jogo é aquela descrita na conclusão da regra; senão, a próxima regra é avaliada da mesma maneira.

Embora RBSs apresentem as vantagens descritas, podem necessitar de muito espaço em memória, muito poder de processamento e até, em algumas situações, se tornar extremamente difíceis de depurar (principalmente quando fazendo uso de um motor de inferência mais poderoso, como por exemplo o JEOPS [85] que traz unificação de variáveis e *forward chaining*).

2.1.3 Lógica Fuzzy

Seres humanos frequentemente analisam situações de maneiras imprecisas, isto é, fazem uso de termos como "pouca força", "muito longe" ou "bastante apertado". A lógica *fuzzy* [33] consegue então representar problemas de uma maneira similar à maneira com que humanos pensam sobre eles, pois conceitos (termos) como longe e pouco não são representados por intervalos discretos, mas por conjuntos *fuzzy* que permitem que um valor pertença a vários conjuntos com diversos graus de pertinência. Por exemplo, um certo personagem de um jogo pode ter seu estado emocial pertencente ao conjunto feliz com grau de pertinência 0.7 e ao conjunto frustrado com grau 0.5.

A lógica booleana tradicional só suporta valores de estado discretos resultando em mudanças de estado abruptas e para se conseguir respostas mais suaves em um sistema baseado em lógica booleana é necessária a adição de mais estados (o que novamente pode levar a explosões da quantidade de regras). Ao passo que lógica nebulosa evita esses problemas pois a resposta vai variar suavemente dado o grau de "verdade" das condições de entrada.

Sua versatilidade faz dela uma excelente opção para aplicações que têm um certo grau de incerteza ou que necessitam de grande flexibilidade e capacidade de adaptação. Sendo assim, jogos eletrônicos constituem um campo potencial animador para sua aplicação.

O'Brien [54] foi o primeiro a atrair a atenção dos desenvolvedores de jogos para o grande potencial desta técnica em criar sistemas de controle com transições sutis e agentes inteligentes que apresentem comportamento não tão previsível e repetitivo quanto com as técnicas geralmente utilizadas.

No entanto, poucos desenvolvedores de jogos exploraram a técnica em jogos e na literatura. Com o passar do tempo publicações introdutórias em IA para jogos (como [86]) e sobre as possibilidades de colaboração entre a

comunidade acadêmica de IA e a de game AI [101] voltaram a mostrar a adequação da abordagem a jogos digitais.

Percebeu-se que lógica *fuzzy* apresenta algumas vantagens (como síntese, adaptação, flexibilidade e versatilidade) importantes na modelagem da inteligência artificial em um jogo; apesar de algumas de suas desvantagens, como sua natureza heurística e a potencial explosão combinatória de regras e antecedentes possibilitassem problemas de depuração e de consumo exagerado de memória e processamento.

No geral, lógica *fuzzy* lida com problemas complexos de controle com um custo computacional relativamente baixo, sem sacrificar os detalhes "sutis" e seus problemas podem ser atacados com um melhor trabalho na definição de que variáveis são importantes e na criação das regras. Com base nisso, muitos pesquisadores e desenvolvedores de jogos passaram a aplicar lógica *fuzzy* na tentativa de mitigar problemas com as abordagens mais utilizadas anteriormente.

Uma tentativa de mitigar os problemas de comportamentos previsíveis e repetitivos presente nas FSMs foi a criação de *Fuzzy* FSMs (FuSMs) [102]. Essas máquinas de estados *fuzzy* são, por exemplo, FSMs que tem valores *fuzzy* nas transições de estado ou FSMs que suportam múltiplos estados ativos [151].

Alguns jogos seguiram essa "tendência" em direção ao uso de FuSMs. Estas foram usadas por exemplo no jogo 'Unreal' (gênero *First Person Shooter*) de modo a fazer os inimigos parecerem razoavelmente inteligentes [101]. Lógica *fuzzy* também foi empregada no jogo 'S.W.A.T. 2' (gênero ação/tática), sendo usada para determinar a resposta tática das unidades inimigas baseada não somente na situação mas também na "personalidade" das unidades. Outra aplicação das FuSMs pode ser visto em 'Civilization: Call to Power' (gênero estratégia) que é um jogo em que os jogadores encontram um número de diferentes grupos culturais. De modo a embutir cada grupo diferente com sua própria personalidade, os desenvolvedores implementaram FuSMs em cascata [3].

Uma outra opção seria usar sistemas *fuzzy* de inferência como proposto em [111] e [112] em contraponto a abordagens RBS. Embora sistemas *fuzzy* baseados em regras permitam que as nuances dentre as entradas sejam capturadas e então refletidas nas decisões a um custo computacional também relativamente baixo, estes são apenas superficialmente citados até mesmo em esforços de padronização de interfaces de uso de IA em jogos, devido a sua pequena utilização [171].

Um exemplo de situação em que um sistema de inferência *fuzzy* (*Fuzzy*

Inference System - FIS) teria melhor desempenho que um RBS comum (além do citado com relação às entradas) seria no caso de haver mais de uma regra disparável em um dado momento. Seriam então selecionadas aquelas cujas condições tenham sido satisfeitas, e combinadas as ações resultantes de alguma maneira (o agente poderia, ao mesmo tempo, fugir e atirar, por exemplo).

Ainda com relação à aplicação de FISs a jogos digitais, pouco existe na literatura sobre sua real aplicação. Os dois exemplos encontrados[1] foram um jogo simples de combate entre duas naves [113] (e a aquisição automática de suas regras [137]) e um trabalho mais detalhado sobre o processo de criação do jogo 'BattleCity.net' (*remake* do jogo 'BattleCitty' da Namco de 1985) e da criação e teste de suas bases de regras [155].

2.1.4 Path-finding

Mover-se de um lugar a outro utilizando um caminho razoável, ao mesmo tempo em que se desvia de obstáculos, é um requisito fundamental para qualquer entidade que queira demostrar algum sinal de inteligência em um jogo. Um dos aspectos mais importantes relacionados à implementação de funcionalidades de IA em jogos, e de impacto visual mais óbvio, é então a determinação de caminhos (*path-finding*).

De modo a tratar esse problema, a abordagem geralmente utilizada é executar um algoritmo de busca sobre os dados da cena de modo a encontrar um caminho entre a posição de origem e a posição de destino.

Esse é um ponto em que a IA para jogos aproveita bem as soluções da IA clássica, especificamente na forma do algoritmo de busca A* [109]. Tal algoritmo é relativamente fácil de implementar e rapidamente encontra o caminho com custo mínimo entre dois pontos no mapa, se um caminho existir [114]. Normalmente o mapa é organizado como um grupo de nós, que são estruturas que representam posições. Uma vez que a busca é realizada sobre esse grafo, o caminho resultante é uma lista de nós a serem percorridos para se chegar ao destino.

Era de se esperar que, uma vez que esse é um problema tão importante e que o melhor algoritmo de busca é aplicado a ele, este fosse um problema já bem tratado. No entato, ainda é comum encontrar situações em que

[1] Há também um trabalho onde o autor dessa dissertação faz um estudo da aplicação de lógica *fuzzy* em um *remake* do jogo 'Pacman' lançado pela Namco em 1980 [172].

personagens fiquem presos em cantos ou vagando perdidos pelo mundo do jogo.

Embora seja um algoritmo robusto e utilizado em diversos jogos [115][81], implementar o A* num contexto de jogos digitais requer diversas melhorias/adaptações [116][87]. Uma dessas adaptações, por exemplo, seria realizar um balanceamento entre qualidade do caminho encontrado e tempo de processamento uma vez que o A* é um algoritmo que pode apresentar elevado custo computacional (especialmente quando aplicado a dezenas de entidades simultaneamente). Além disso, o A* precisa de uma boa estrutura para armazenar e manipular os caminhos e tem dificuldade em lidar com mudanças e objetivos dinâmicos no cenário.

O algoritmo em si é apenas parte do problema, de modo a melhorar seu desempenho é necessário (principalmente) tentar simplificar o espaço de busca. Existem diversas maneiras de se representar esse espaço, dentre as quais se destacam: representação em grade; grafo de esquinas (*corner graph*); grafo de *waypoints*; grafo de *waypoints*-círculo (*circle-based waypoints*); *space filing volumes* (similar aos *waypoints*-círculo, mas com retângulos no lugar dos círculos e conexão entre os mesmos); pontos de visibilidade; e malhas de navegação (*navigation meshes*) [88].

Uma maneira de facilitar o cálculo de caminhos é fazer uso de *waypoints*. *Waypoints* são marcações no ambiente que ajudam os NPCs a se mover no ambiente, já que eles representam os nós no grafo (facilitando a representação do mapa, como discutido anteriormente) com todos os caminhos possíveis no mundo do jogo [117]. Outra vantagem do uso de *waypoints* é que estes podem ser utilizados para sinalizar pontos de interesse para os NPCs (como por exemplo os *waypoints* que represetam as entradas dos prédios em [156]).

Uma outra abordagem bastante flexível, e que merece destaque, é a de malhas de navegação que podem ser descritas como basicamente um conjunto de polígonos convexos que descreve as superfícies caminháveis do mundo. Essa abordagem é bastante intuitiva, pois permite criar uma "planta baixa" do cenário de maneira completamente automatizada e se adequa a qualquer tipo de ambiente 3D [118] (desde que o mesmo não sofra mudanças drásticas em tempo de execução).

Além do cálculo de caminhos e de evitar obstáculos no cenário, há ainda a necessidade dos personagens estarem atentos ao ambiente, seja para desviarem de objetos dinâmicos (como evitar colidir com outro personagem) quanto para poderem escolher melhor seus caminhos (por exemplo, evitar passar por uma área perigosa). Estes motivos fazem com que o problema

também seja referido como planejamento de caminhos (*path-planning*).

Para realizar esse planejamento de caminhos, geralmente tomam-se as informações extras como alterações nos pesos dos caminhos a serem analisados pelo *path-finder*. No entanto, também existem outras abordagens para o problema.

Uma abordagem para esse "planejamento" é o uso dos chamados mapas de influência [157], uma representação espacial do conhecimento do mundo que contém informações codificadas espacialmente (por exemplo: grau de perigo da região, quantidade de unidades, capacidade ofensiva e defensiva, quão aberto é o local) que podem também ser representadas em diversas camadas. Outra abordagem é a de análise de terreno [158], onde através de processamento automático é possível ao agente analisar o ambiente para se posicionar melhor, identificar se se encontra vulnerável a fogo inimigo, encontrar cobertura, etc.

Praticamente todas estas técnicas podem ser executadas em uma etapa de pré-processamento, durante a carga dos diferentes níveis de jogo ou dinamicamente com diferentes graus de flexibilidade. Sendo algumas vezes necessário combiná-las em diferentes camadas. Uma outra questão que também merece atenção é a geração progressiva de caminhos, começando com um caminho aproximado e refinando-o aos poucos.

2.1.5 Comportamentos de Movimentação

Como comentado anteriormente, um ponto importante em jogos é a maneira com que os objetos e personagens se comportam e em particular como eles se movimentam no mundo do jogo. Os personagens no ambiente do jogo têm que ter uma certa autonomia na sua movimentação, não só relacionada a como se deslocar de um ponto a outro, mas também a como se comportar durante esse trajeto.

Com relação a esse comportamento de movimentação, grande parte das questões sobre movimento de unidades em jogos gira em torno de minimizar a carga da CPU versus maximizar a (aparente) inteligência do movimento. Outra preocupação é a de implementar comportamentos inteligentes bem definidos e de uma maneira que possam ser reutilizados de modo que possam ser úteis para melhorar a experiência do jogador e facilitar o desenvolvimento [4].

Reynolds [188] propôs que para tratar deste problema de autonomia e gerenciamento de movimento, pode-se dividir o gerenciamento em três níveis

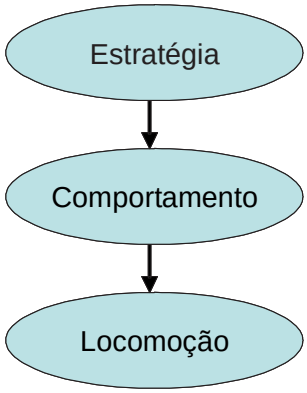

Figura 2.2: Hierarquia de comportamentos de movimentação [188].

(como observados na Figura 2.2): estratégia, comportamento e locomoção.

Nesta abordagem, a camada de locomoção é a camada de baixo nível responsável por representar o corpo do objeto, convertendo sinais do modelo de comportamento na movimentação propriamente dita, sendo essa sujeita às restrições do modelo físico do corpo (por exemplo, velocidade máxima). Esta camada geralmente é implementada através de modelagem física (com aceleração, velocidade, colisões, etc.) e é a base de qualquer movimento.

A camada de comportamento (*steering*) refere-se à forma como o agente adquire informação do ambiente e é focada em que ações tomar (cálculo da trajetória e determinação de ações/sub-objetivos) para que seja possível chegar ao local determinado, lidando com possíveis obstáculos encontrados no mundo. Alguns exemplos desses comportamentos seriam: perseguir ou fugir de um determinado objeto, seguir o líder, evitar colisões, andar em grupo, seguir um fluxo, etc.

Por sua vez, a camada de estratégia (seleção de ações) é a camada de mais alto nível, onde se realiza o planejamento de que ação o agente vai realizar, que objetivos se deseja alcançar. Embora não exatamente pertencentes a essa camada, combinações de comportamentos básicos podem também servir para implementar alguma estratégia ou alcançar um objetivo.

Os esforços de pesquisas sobre essa área têm relacionamento e implicações em robótica, IA e vida artificial, sendo geralmente voltadas para animação, ou seja, modelagem de comportamento está localizada na interseção entre animação e os três campos citados.

O método mais utilizado para implementar as características necessárias descritas anteriormente (e conseqüentemente, útil a essas várias áreas) é chamado *Steering Behaviors* [188] (Comportamentos de

Movimentação em português) e consiste em usar regras simples que juntas dão a grupos de agentes autônomos uma forma mais realística de movimentação por unidade e de comportamento em grupo. É uma técnica que não preserva estado, economiza memória e reage em tempo real, o que se encaixa nos requisitos encontrados em jogos digitais. Além do que, combinações de *steering behaviors* podem ser usadas (como sugerido anteriormente) para se alcançar objetivos de mais alto nível (por exemplo, atravessar uma região evitando obstáculos).

Reynolds previamente propôs vários comportamentos de movimentação para controlar a movimentação de criaturas autônomas em ambientes virtuais [188]. Estes comportamentos produzem movimentação "interessante" para estas entidades autônomas e consistem de melhorias sobre o modelo de BOIDS original [187].

Com relação à coordenação de movimento, serão descritos alguns dos modelos de comportamento propostos por Reynolds de modo a exemplificar a utilidade da abordagem de comportamentos de movimentação e as inter-relações entre comportamentos básicos e sua capacidade de reuso.

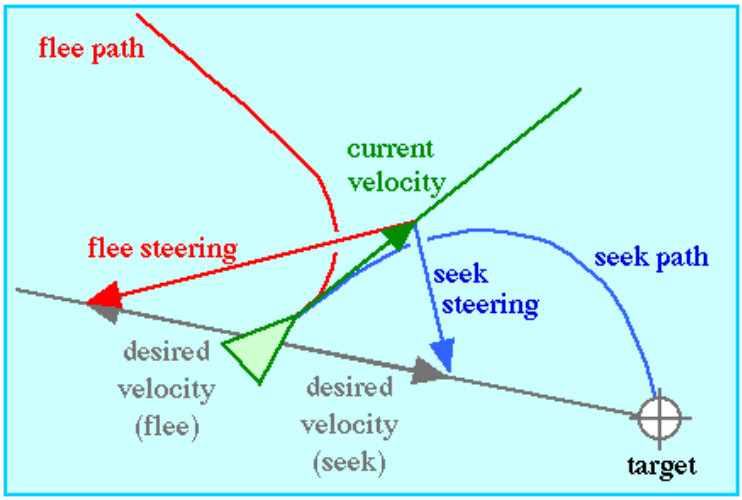

Figura 2.3: Exemplo dos comportamentos *Seek* e *Flee*. Reproduzida de Reynolds [188].

Um dos comportamentos básicos apresentados por Reynolds é o comportamento *Seek* (Figura 2.3). *Seek* funciona "puxando" o personagem em direção a um ponto específico no mapa. Esse comportamento ajusta a velocidade do personagem de modo que fique radialmente alinhada em direção ao alvo. A "velociade desejada" é um vetor na direção do objeto para o alvo e o módulo deste vetor pode ser expresso pela velocidade máxima, ou pela velocidade atual dependendo da aplicação. O vetor de *steering* é a

diferença entre essa velocidade desejada e a velocidade atual do objeto. Se o objeto sempre segue o comportamento *seek*, ele eventualmente vai passar pelo alvo, dar a volta e se aproximar novamente, produzindo um movimento semelhante ao de uma mosca.

Outro comportamento proposto é o comportamento *Flee* (Figura 2.3) que é simplesmente o inverso do comportamento anterior e "puxa" o objeto de modo que sua velocidade é radialmente alinhada para longe do alvo e a velocidade desejada é na direção oposta.

Pursuit é outro dos comportamentos proposto e é bastante similar ao *Seek*, exceto pelo fato de que o alvo agora é um objeto móvel. Uma perseguição efetiva requer predição da posição futura do alvo. A idéia aqui é simplesmente prever a nova posição do alvo e reavaliar essa previsão a cada passo. A predição corresponde a assumir que o alvo não vai mudar de direção durante o intervalo de predição. Isso quase sempre não é o que acontece, mas como esse resultado só será utilizado num curto espaço de tempo isso não é tão prejudicial e poupa o sistema de problemas relacionados a performance, que surgiriam no caso de se tentar utilizar predições mais elaboradas. *Pursuit* é então praticamente aplicar *Seek* ao alvo previsto.

A chave do *Pursuit* é o método usado para estimar o intervalo de predição. Idealmente, esse intervalo deveria ser o tempo até a interceptação, mas esse valor não é calculável pois o alvo pode fazer manobras arbitrárias e imprevisíveis. Uma abordagem seria tratar esse tempo como constante, pois embora ingênua, ainda apresentaria um comportamento melhor que um simples *Seek*. Para uma perfomance razoável o tempo deveria ser proporcional à distância do alvo, grande quando longe do alvo e pequeno quando perto.

O comportamento *Evasion* é análogo a *Pursuit*, exceto que *Flee* é utilizado ao invés de *Seek* para calcular a nova direção do objeto.

Arrival funciona identicamente a *Seek* enquanto o objeto está longe do alvo. Mas ao invés de ter o problema de passar direto pelo alvo, o objeto vai freando a medida que se aproxima do seu alvo, eventualmente parando na posição desejada. A distância onde se deve começar a frear (ou raio de ação) é um paramêtro do comportamento.

Reynolds ainda propõe diversos outros comportamentos, dentre os quais podem-se destacar *obstacle avoidance e hide* como dois dos mais aplicáveis em jogos.

Além dos comportamentos individuais, também é possível definir comportamentos que levam em consideração alguns dos seus vizinhos (criando grupos de objetos) [187] para definir o comportamento do personagem.

Alguns desses comportamentos podem ser, por exemplo, os comportamentos apresentados a seguir: *Separation*, *Cohesion* e *Alignment* (Figura 2.4).

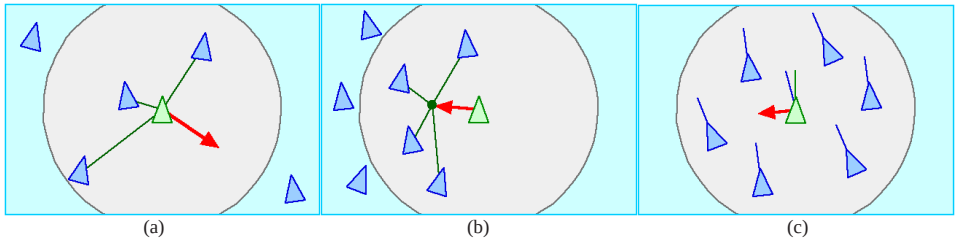

Figura 2.4: Exemplos de comportamentos de movimentação em grupo. Reproduzidas de Reynolds[188].

Separation (Figura 2.4(a)) oferece a um personagem a habilidade de manter uma certa separação dos outros personagens próximos a ele. Esse comportamento pode ser usado para impedir que diversos personagens se aglomerem. Para computar a força do comportamento, é feita uma busca pelos seus objetos vizinhos. Para cada objeto próximo, uma força repulsiva é calculada subtraindo-se a posição do objeto da do seu vizinho, normalizando, e então se aplicando um peso. Essas forças repulsivas são somadas e a resultante é a força do comportamento.

Cohesion (Figura 2.4(b)), por sua vez, permite ao objeto se aproximar e formar um grupo com outros objetos próximos. A força de *steering* desse comportamento pode ser computada encontrando os vizinhos de maneira similar à anterior e calculando a posição média destes (centro de gravidade do grupo). A força de coesão será então aplicada na direção dessa posição, ou esta pode ser usada como alvo para o comportamento *Seek*.

Finalmente, *Alignment* (Figura 2.4(c)) provê a habilidade do sentido de movimento de um personagem ser alinhado (apontar para a mesma direção) ao dos outros personagens próximos. E pode ser calculado encontrando-se os objetos na vizinhança e tirando a média das direções de suas velocidades. Essa média é a direção desejada, então a força será calculada como a diferença entre a velocidade atual e esta direção de velocidade desejada. Essa força tende a "virar" o objeto para a mesma direção para a qual os demais estão voltados.

Os comportamentos descritos anteriormente, além de modelarem um comportamento específico, podem também servir como blocos para definir novos padrões de comportamento mais complexos. Para se conseguir comportamentos interessantes e críveis, é necessário selecionar e combinar estes componentes individuais, uma vez que a não ser que o "objeto" exista num mundo extremamente simples, não faz sentido ele apresentar sempre o

mesmo comportamento. Por exemplo, juntos, os comportamentos de grupo descritos na seção anterior criam o efeito de *flocking* pelo qual o trabalho de Reynolds tornou-se inicialmente conhecido.

A combinação de comportamentos pode ser conseguida unificando as forças de direcionamento (*steering*) dos vários comportamentos básicos de várias maneiras. As duas maneiras mais simples são usando uma soma ponderada truncada ou uma soma ponderada truncada com priorização.

A soma truncada é a abordagem mais simples e consiste em calcular as forças dos comportamentos individuais e somá-los, possivelmente com algum fator de peso aplicado a cada uma das forças, e depois truncar o valor resultante pelo da força máxima permitida. Essa combinação linear simples funciona razoavelmente bem, mas também apresenta problemas: não é computacionalmente eficiente (é custoso calcular sempre as diversas forças), é extremante difícil configurar e ajustar os pesos e mesmo com o ajuste dos pesos, as forças dos comportamentos podem se anular em momentos inoportunos.

A combinação através da soma ponderada truncada com priorização melhora essa situação e apresenta um bom compromisso entre velocidade de processamento e precisão. Essa abordagem consiste em definir prioridades aos diversos comportamentos e realizar a soma das forças usando essas prioridades para impor uma ordem a esta soma (os comportamentos com maior prioridade são processados primeiro e os de menor prioridade serão tratados por último). Deste modo, a força resultante vai acumulando as diferentes forças dos comportamentos agrupados até que o valor agregado seja próximo ao máximo de *steering* possível, o que faz com que as demais forças sejam desprezadas.

Como discutido, comportamentos de movimentação são uma abordagem relativamente leve e que permite uma boa autonomia para a movimentação dos personagens. Uma outra vantagem da abordagem é que ela pode ser facilmente combinada às abordagens de *path-finding* e assim pode lidar melhor com problemas como evitar obstáculos dinâmicos durante o percorrimento de um caminho. Também é possível estender os *steering behaviours* e combiná-los com técnicas de planejamento de caminhos [159].

2.1.6 Outras técnicas

Além dessas técnicas comumente utilizadas em IA para jogos, a seguir algumas técnicas promissoras e que vêm recebendo um crescente interesse dos desenvolvedores também são apresentadas.

2.1.6.1 Redes Neurais

Uma dessas abordagens promissoras, e que vêm recebendo crescente interesse dos desenvolvedores de jogos [89], é a de redes neurais (RN), pois provê benefícios de aprendizado por exemplos e pode ser implementada de maneira relativamente simples [160].

Uma rede neural artificial é uma simulação de um modelo simplificado do cérebro e é composta por unidades chamadas neurônios (nós da rede) e conexões com pesos entre esses nós. A RN adquire conhecimento do mundo e o mantém, representando-o nos pesos de suas conexões. Estes pesos são refinados através de uma fase de treinamento de modo que a rede "aprenda" a se comportar em um dado cenário.

Uma unidade neurônio básica possui um conjunto de entradas, um conjunto de camadas internas e um conjunto de saídas. Durante o processo de treino de um neurônio, os sinais são conectados às suas entradas e um concentrador multiplica o valor das entradas pelos respectivos pesos e passa este sinal para uma função de ativação correspondente à unidade, e esta se disparada propaga o sinal para o próximo neurônio. Ao término desse processo, é calculado o erro das saídas em relação ao universo de exemplos de teste e os pesos são ajustados se necessário. O processo de testes só é finalizado quando a taxa de erro atinge um certo limiar ou após um dado número de ciclos de execução.

Embora RNs possam ser utilizadas como um módulo fechado (caixa preta), são um técnica complexa e é necessário ter conhecimento de como funcionam internamente para poder tirar proveito do que podem oferecer. Alguns pontos que devem ser observados são como escolher as entradas da rede (modelando o ambiente com cuidado), que precauções tomar durante o processo de treinamento e como definir a estrutura da rede.

Champandard [119] sugere dividir as abordagens da aplicação de redes neurais em jogos digitais em dois ramos, reconhecimento (que pode ser visto como um processo de decisão) e controladores robóticos (redes para controlar o comportamento físico das entidades usando o método de

regressão, como em [46]) e alega que utilizar essa diferenciação no projeto conceitual traz grandes benefícios mesmo o funcionamento interno da rede sendo igual.

Para problemas que podem ser classificados como reconhecimento, Champandard recomenda que uma boa abordagem inicial seria limitar o número de camadas em três e representar a camada interna com uma quantidade de nós igual ao dobro do número de entradas, pois essa configuração é suficiente para classificar qualquer padrão [119] e apresenta bom desempenho com o uso de *back-propagation*.

Já no caso do uso como controlador, deve-se tentar diminuir ao máximo o número de entradas da rede e deve-se utilizar ao menos quatro camadas internas com número de nós por camada também igual a duas vezes o número de entradas. Essa configuração torna a rede mais flexível e a diminuição do número de entradas contribui para amenizar o problema de treinar a rede, deste modo tornado-a mais adequada ao problema de controle que faz uso intensivo da rede.

Poucos desenvolvedores de jogos têm utilizado essa técnica até o momento e geralmente aplicado apenas o modelo de rede mais popular e fácil de utilizar, o *multilayer perceptron* [103][160]. Mas, embora aplicadas na maioria das vezes de maneira muito simples, redes neurais têm-se mostrado adequadas aos ambientes encontrados em jogos digitais e cenários similares.

Alguns exemplos de sua aplicação com sucesso em jogos são: para que agentes jogadores de futebol aprendam a interceptar a bola durante uma partida [65]; para modelar o comportamento mais alto nível de agentes como em [120] e [161] ou para ensinar aos NPCs a dirigir um carro como no jogo 'Colin McRae Rally 2.0' da Codemasters [121]. A técnica de redes neurais pode até mesmo ser implementada por meio de programação em placa gráfica [162] de modo a liberar mais tempo de processamento para outras tarefas se necessário.

2.1.6.2 Algoritmos genéticos

Outra abordagem para permitir criar entidades inteligentes (ou agentes inteligentes) capazes de apresentar comportamentos interessantes, tomar decisões, e se adaptar ao mundo do jogo é a utilização dos chamados algoritmos genéticos.

Algoritmos genéticos usam uma abordagem inspirada no proceso de seleção natural e tentam imitar esse processo para evoluir até encontrar uma

solução próxima à ótima para o problema. Esta abordagem aos poucos vem chamando a atenção dos desenvolvedores de jogos e ganhando popularidade tanto por permitir a modelagem do comportamento de agentes através de sua evolução como pré-processamento quanto para modelar esta mesma evolução durante o jogo [122][163].

Ao utilizar algoritmos genéticos para "evoluir" um personagem, certas propriedades do agente podem ser modeladas como sendo os genes e cada possível valor do gene é chamado alelo (por exemplo, sendo o gene cor dos olhos, os valores azul e verde seriam dois alelos). O conjunto dos genes que definem um personagem forma um cromossomo que representa um indivíduo.

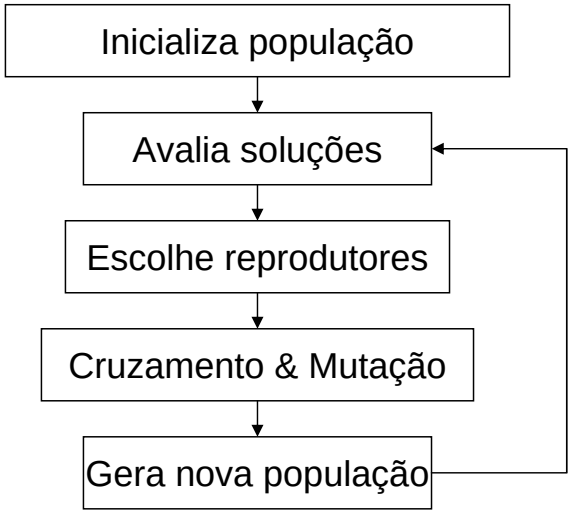

Figura 2.5: Processo básico de execução de um algoritmo genético.

De modo geral, o processo de execução segue os passos apresentados na Figura 2.5. Inicialmente é criada a população inicial, geralmente preenchendo-se os valores dos cromossomos de forma aleatória. Em seguida, é realizada uma avaliação da população e se o objetivo da evolução foi alcançado o processo para. Caso o critério de parada não tenha sido ainda atingido, é realizado um processo de cruzamento para criar uma nova população de agentes. Nesse processo de cruzamento uma parte da população original é transferida para a próxima população e casais de indivíduos são selecionados para combinar seus cromossomos e dar origem a um indivíduo da nova população. Além desses cruzamentos, introduz-se um certo fator de mutação no processo, de modo a aumentar a diversidade da população. Após o cruzamento, repete-se o passo de avaliação da população.

Existem diversas abordagens para especificar como se realiza a combinação dos cromossomos, que indivíduos vão reproduzir, que indivíduos serão transferidos para a nova população, como definir os critérios de avaliação da população, como evitar convergência muito rápida da população, etc. mas estas se encontram fora do escopo deste trabalho, mais detalhes nesse sentido podem ser encontrados em [122][164].

Algoritmos genéticos são úteis quando é difícil prever as interações entre o conjunto de parâmetros que regula o comportamento dos agentes e o mundo do jogo. No entanto, é necessário um bom esforço de tempo em modelagem e principalmente simulação para alcançar bons resultados.

Esta abordagem evolucionária se adequa a tratar problemas em diversos níveis, seja modelando um troll em um cenário simples [122], evoluindo os comportamentos de um time de robos jogadores de futebol na categoria de simulação da Robocup [66] até o aprendizado de regras em um sistema de inferência *fuzzy* para um jogo de combate entre tanques [155].

2.1.6.3 Planejamento

Planejamento de ações e sistemas de planejamento não são uma abordagem nova na área de inteligência artificial, contudo, raramente têm sido usados para modelar o comportamento de NPCs em jogos digitais. Um sistema que usa planejamento trás uma arquitetura modular e pode facilitar muito a implementação e compartilhamento de comportamentos.

Vários jogos recentemente começaram a fazer uso de tomada de decisão orientada a objetivos (*goal-directed decision making*)[165], o que consiste basicamente em ter uma biblioteca de planos e escolher qual o mais adequado para um dado momento. O jogo *'No One Lives Forever 2: a Spy in H.A.R.M.'s Way'* da Monolith é um jogo que utiliza essa abordagem. NOLF2 utiliza um sistema de objetivos e *SmartObjects* (objetos que emitem a informação de que necessidades suprem) de modo a permitir que os NPCs tenham comportamentos variados e verossímeis sem que seja necessário, por exemplo, ter que codificar todos os possíveis estados e transições do comportamento.

Planejamento pode ainda ser usado de uma maneira mais flexível e poderosa. Através de planejamento de ações orientado a objetivos (*goal oriented action planning* - GOAP) é possível permitir aos NPCs decidir não só o que fazer em um certo momento, mas também como fazer [166]. O personagem formula seus próprios planos, tem comportamentos menos

repetitivos e menos previsíveis, além de poder adaptar suas ações a condições específicas da situação corrente.

Para explicar como funciona um sistema de planejamento, são necessárias algumas definições. Nareyek et al [171] define ação, plano e objetivo como:

Ação: comportamento atômico (isto é, não pode ser subdividido) que contribui para a satisfação de um ou múltiplos objetivos. Ações têm um número variado de efeitos e pré-condições. Os efeitos definem como a execução da ação modifica o estado do mundo e as pré-condições definem aspectos do estado do mundo que devem ser verdade no momento da execução da ação. Uma ação pode ser tanto uma ação simples (como atirar) ou um comportamento simples (como mover-se até uma certa posição).

Objetivo (*Goal*): sub-estado do ambiente do jogo (acoplado ao personagem ou ao mundo) que um plano tenta alcançar e que quando alcançado com sucesso encerra o comportamento iniciado para alcançá-lo.

Plano: seqüência válida de ações a ser executada.

Assim, um sistema de planejamento recebe como entrada o estado atual do mundo do jogo, objetivos a serem satisfeitos e o conjunto das ações que podem ser executadas e gera um plano, que é uma seqüência válida de ações que quando executada sobre o estado atual do mundo satisfaz os objetivos dados.

A idéia é então criar um plano em tempo real passando um ou mais objetivos para o planejador, o planjeador realiza uma busca no espaço de ações e caso um caminho do estado atual até o objetivo (passando pelas ações permitidas) seja encontrado, esse é retornado ao agente. Com o plano, o agente pode seguí-lo até o fim, até que o plano seja invalidado por mudanças no mundo ou até o surgimento de outro objetivo de maior prioridade ocasionar a criação de um novo plano.

Alguns benefícios da abordagem GOAP [166][167] são:

- as ações do NPC podem ser adaptadas à situação atual e aos arredores de onde o agente se encontra; o NPC pode encontrar soluções alternativas para um mesmo problema dinamicamente;

- o código da implementação de um sistema desse tipo fica mais simples pois evita que qualquer pequena mudança (como um passo a mais)

numa seqüência de ações invalide todo o conjunto de planos e ocasione a necessitade de editá-los novamente;

- maior corretude já que como os planos são gerados pelo planejador, evita erros cometidos pelo programador, como definir uma seqüência de ações não válida; e

- permite também uma maior variedade de comportamento pois, com um *pool* de ações possíveis, um mesmo personagem pode resolver um mesmo problema de várias maneiras diferentes.

Além da aplicação de planejamento nos comportamentos e ações dos personagens, uma outra possibilidade bastante interessante é aplicar esta técnica na geração da própria história do jogo [180][173].

2.1.6.4 Redes Bayesianas

No caso de jogos de gerenciamento, como 'SimCity', onde é necessária a simulação de sistemas complexos, aquisição de conhecimento, incerteza e grande número de variáveis, o uso das técnicas anteriores se mostra inadequado.

Redes bayesianas [35][109], por sua vez, conseguem lidar bem com a maioria dessas questões. Sintaticamente, uma rede deste tipo é um grafo acíclico direcionado (dag) onde cada nó representa uma variável de estado com estados independentes e mutualmente exclusivos. As arestas direcionadas representam a influência do nó pai no nó filho. Para cada nó filho, uma tabela de probabilidades define como seus estados são afetados pelas combinações de estados do nó pai e assim os efeitos são codificados probabilisticamente na definção da BBN (*Belief Bayesian Network*)[39].

A idéia principal por trás dessa abordagem é o fato de que a maioria dos eventos que ocorrem são independentes entre si e consequentemente a interação entre eles não precisa ser considerada, o que favorece uma representação mais local e concisa da distribuição das probabilidades. A maior vantagem dessas redes é a de se alcançar conclusões a partir de raciocínio sobre incertezas (com conhecimento incompleto do mundo), uma vez que esse método captura a dependência dinâmica entre variáveis aleatórias com conhecimento de outros eventos. Uma outra vantagem é que, mesmo para redes muito grandes, é provado que realizar cálculos necessários é um problema tratável [42].

Redes bayesianas permitem uma representação concisa e limpa do conhecimento e permitem ajustes finos das diferentes probabilidades de modo a alcançar melhores resultados. No entanto, como inicializar as probabilidades de transição de modo a representar bem as relações de causa e efeito para cada jogo é um problema difícil [123] e fora do escopo desta dissertação.

Embora seja uma técnica pouco usada em jogos, se adequa bem a ambientes onde a modelagem seja o aspecto mais importante do jogo [124], como mostra o jogo 'FutSim' [5], um jogo de gerenciamento de times de futebol criado pela Jynx Playware. Essa técnica é bastante promissora uma vez que possibilita a previsão das prováveis consequências de uma certa ação ou selecionar uma ação "ótima" dada uma certa configuração e começa a atrair a atenção dos desenvolvedores de jogos [125].

2.1.7 Agentes em Jogos

O termo agente é utilizado de forma bastante genérica, não havendo uma definição exata do termo aceita por todos. Russel e Norvig definem um agente como sendo uma entidade que pode perceber um ambiente através de sensores e agir através de atuadores. Já Nareyek, em [84], define o termo agente como uma entidade que têm objetivos, que pode sentir propriedades do ambiente e executar ações sobre este. De forma geral, um agente é uma entidade que está situada em um ambiente sobre o qual tem sensações e pode agir, idealmente mantendo seus objetivos e prevendo os efeitos de suas ações no mundo (por exemplo através de um estado mental como nas arquiteturas BDI [44] e HAL [185]). Essa diversidade na definição do termo agente, surge do fato de que, para diferentes domínios de aplicação, os atributos associados ao conceito têm diferentes níveis de importância [51].

Neste trabalho, um agente inteligente seria uma entidade de software que se encontra em um meio termo entre a definição simples em [109] (mais próxima de agentes reativos) e a que possui seus próprios objetivos, sensores, mecanismo de raciocínio e reações (mais próxima de agentes deliberativos). Numa abordagem aplicada a jogos digitais, os NPCs seriam então modelados como entidades "vivas" e "pensantes" em diferente graus. A utilização dessa abordagem tem como vantagens o fato de que esta arquitetura permite uma mais fácil criação de um sistema de comportamentos complexos e o fato dos agentes poderem interagir entre si, se necessário, usando linguagens de comunicação.

Uma outra vantagem dessa abordagem de modelagem de agentes é que além de um comportamento inteligente, muitas vezes é necessário que os personagens do jogo apresentem algo mais. Modelar as personalidades e emoções destes personagens de modo que interfiram em suas ações pode então tornar seus comportamentos ainda mais realísticos e atrativos (para tratar destas questões, pode-se recorrer também mais uma vez à academia na forma das linhas de pesquisa em atores sintéticos, ou *believable agents* [6]).

O uso de agentes inteligentes permite a criação de criaturas com comportamentos emergentes, o que (como já citado) melhora a experiência do usuário e é um dos objetivos que se busca alcançar com novas técnicas de IA. Uma das mais populares abordagens para essa modelagem de agentes deliberativos baseadas em estados mentais é a chamada arquitetura BDI [44], onde a sigla significa *Belief-Desire-Intention*. Ela pode ser descrita como um conjunto de crenças, desejos e intenções.

Crenças ou *beliefs* são a representação do conhecimento que o agente possui sobre o seu ambiente atual e sobre si. As crenças de um agente podem ser vistas como o provável estado do ambiente, alguma crença pode ter se tornado inválida mas o agente ainda "crê" nela e a utiliza no seu processo de raciocínio já que não teve contato com essa mudança. Um agente pode ter crenças sobre o mundo, sobre outros agentes, sobre interações com outros agentes e, até mesmo, crenças sobre suas próprias crenças.

Desejos ou *desires*, descrevem os objetivos do agente (um estado do sistema que o agente quer alcançar) e representam estados desejáveis que o sistema poderia apresentar. Os desejos motivam o agente a agir de forma a alcançar estes objetivos. As ações finais são definidas através das intenções criadas pelos desejos.

Intenções ou *intentions*, podem ser interpretadas como as ações que o agente selecionou para alcançar um certo objetivo, ou um plano a ser seguido. Se um agente após seu processo de decisão escolhe tentar alcançar um objetivo, então este objetivo cria uma intenção. As intenções determinam a parte mais "prática" do processo de raciocínio pois determinam as ações (ou seqüência de ações) a serem realizadas.

Ao projetar um agente baseando-se no modelo BDI, são especificados suas crenças e desejos iniciais e a escolha das intenções e, conseqüentemente, das ações a serem tomadas, fica a cargo do agente (dentro do espaço de ações possíveis) o que acarreta em problemas bastante similares à abordagem de planejamento discutida anteriormente.

Este modelo é geralmente adaptado quando da criação de agentes

deliberativos para jogos de computador. Deste modo, no começo de cada ciclo o agente "sente" o ambiente e atualiza seu conjunto de crenças. Como consequência desse passo um desejo pode ser selecionado/disparado como resposta a algum evento recém recebido. O agente então seleciona as melhores ações a serem executadas (através de algum esquema de arbitragem ou planejamento). Com o fim do ciclo, a intenção/ação que atende o desejo mais urgente é escolhida e executada.

Esse ciclo é chamado Ciclo de Execução do Agente e pode ser representado resumidamente como um conjunto de três passos: *Sense*, onde ocorre o acesso às informações sensoriais vindas do mundo do jogo; *Think*, onde se processa o conhecimento adquirido, levando em conta o estado do mundo; e *Act*, onde se executam as ações escolhidas.

2.1.8 Conclusões e observações

Desenvolvedores de jogos digitais necessitam de soluções entre FSMs simples e complexos modelos cognitivos. Primeiramente, é necessário evitar a necessidade de atualizações muito frequentes e lidar com um modelo baseado em eventos. Em segundo lugar, essa nova "solução" precisa ser capaz de apresentar comportamentos não repetitivos, exibir uma variedade de reações e ainda ser crível (apresentar comportamentos que ao menos pareçam inteligentes e lógicos). E, por último, tal solução deve prover um *framework* simples sobre o qual se possa trabalhar e que permita a criação de outras soluções altamente "customizadas"; este *framework* deve ainda possibilitar uma boa escalabilidade ao desenvolvimento.

Uma possível abordagem que apresenta muitas dessas características é o uso de planejamento, em especial o uso de "*anytime planners*" [90], planejadores que melhoram o plano a cada iteração, se há pouco tempo de processamento disponível, o sistema ainda assim consegue ser reativo, e o plano é refinado se houver uma maior fatia de tempo de processamento, além de permitir que o plano seja adaptado/atualizado no caso de mudanças ocorrerem no mundo do jogo. Ou mesmo, técnicas de "planejamento representacional" [138] que poderiam ser usadas sobre os mecanismos de inteligência artificial já utilizados, que manipulam uma representação do mundo de modo a possibilitar a escolha do melhor comportamento para uma dada situação. Mas, infelizmente, abordagens de planejamento são pouco conhecidas e utilizadas no desenvolvimento de jogos por computador (embora a situação venha se modificando, como mostrado anteriormente).

A área de agentes inteligentes também tem diversas aplicações a jogos pela representação dos personagens aos olhos dos jogadores como entidades que pensam, mas também como inspiração a partir da modelagem acadêmica de suas propriedades.

Prover essas diversas possibilidades de representação de comportamento desde as mais simples (como FSMs simples) até as mais complexas (como agentes BDI e planejamento em geral) é essencial, uma vez que nem todos os personagens em um jogo necessitam de muita inteligência.

Algumas outras áreas que merecem atenção especial com relação ao uso de IA em jogos digitais e que devem apresentar um grande crescimento no futuro próximo são aprendizado automatizado (como a iniciativa KnoMic-*Knowledge Mimic* [72], por exemplo), uma maior interação entre os personagens do jogo [91] (através de comunicação e da modelagem de seu comportamento social, outra área que pode se beneficiar das pesquisas em agentes inteligentes) e iniciativas de narrativas inteligentes, sejam orientadas a personagens [189] ou a história em si [173] de modo a melhorar ainda mais a experiência do usuário e a jogabilidade.