3
Middleware de Inteligência Artificial

3.1
Definição de middleware de IA

Um *middleware* de inteligência artificial é basicamente uma camada de software (ou um conjunto de componentes) que provê serviços para o motor do jogo realizar as funções de IA. Geralmente também chamado de "motor" de IA, o *middleware* trata do processo de produzir o comportamento desejado ou a tomada de decisão dos agentes inteligentes presentes no mundo do jogo.

O termo *middleware* é usualmente associado a ambientes de sistemas distribuídos mas é usado aqui no sentido encontrado em algumas das primeiras citações do termo nos anos 70, que definem "*middleware*" como um software de computador o qual foi trabalhado para as necessidades especiais de uma instalação. O termo comparativamente novo na época foi introduzido pois, como alguns sistemas haviam se tornado mais complexos, havia uma necessidade de melhorias ou modificações nos sistemas para atender às novas necessidades; os programas que efetivavam essas mudanças eram então chamados de '*middleware*' pois eles se encaixavam entre o sitema operacional e as aplicações.

Deste modo, *middleware* é software que intermedia software interagindo com (no nosso contexto) o motor principal do jogo. De acordo com o *Software Engineering Institute* (SEI), *middleware* é definido como uma camada de software que fica entre o sistema operacional e as aplicações, provendo interfaces de alto nível uniformes e padronizadas para os desenvolvedores de aplicações de modo que essas aplicações possam ser facilmente compostas e reusadas. Isto é, o papel do *middleware* é então tornar o desenvolvimento das aplicações mais fácil, provendo abstrações, mascarando a heterogeniedade do sistema por baixo e escondendo detalhes de programação de baixo nível; o que casa perfeitamente com nossas necessidades. Consequentemente, questões arquiteturais desempenham um

papel central no projeto de um *middleware*. Arquitetura é preocupada com a organização, estrutura geral e padrões de comunicação, tanto entre aplicações quanto com relação ao *middleware* em si.

3.2 Desafios

Como Nareyek aponta em [168], um ponto central aos cálculos e processos relacionados a AI em jogos não é somente como as ações são determinadas, mas também que informações sobre o ambiente estão disponíveis e como elas podem ser acessadas. Outra questão é como o middleware de IA vai se integrar com outros motores/middlewares existentes sem causar conflitos ou problemas do gênero. Um software com o papel de middleware de IA não precisa encontrar soluções ótimas para os problemas, mas encontrar soluções sob diversas restrições [151] e ainda assim oferecer uma IA desafiadora e em tempo real. Algumas das restrições enfrentadas são: ambientes grandes e não estáticos; estados parcialmente observáveis; ações possivelmente caras e modelos imprecisos do mundo. Estas questões aliadas à falta de literatura disponível sobre o assunto tornam o problema de desenvolver um *middleware* de IA muito maior; não é necessário apenas lidar com o projeto conceitual da IA mas também com uma pesada carga de "preocupações" de engenharia de software.

Estes são os desafios mais tecnológicos, tanto de modelagem da IA quanto da implementação, mas também é necessário realizar um maior esforço para tornar as funcionalidades de IA mais acessíveis a artistas e designers. Estes geralmente não possuem habilidades em programação e, conseqüentemente, precisam de ferramentas de alto nível para dar forma e controlar os comportamentos da IA uma vez que em jogos comerciais são responsáveis por grande parte da criação de conteúdo (além de ferramentas de mais alto nível aumentarem a produção).

3.3 Projeto e estruturação

Como exposto anteriormente, diversos dos desafios da criação de middlewares de IA estão relacionados a como projetar e estruturar tais sistemas. Duas abordagens iniciais para essa componentização poderiam ser chamadas de "Bibliotecas de Funcionalidades" e "Baseada em Agentes", onde a biblioteca de funcionalidades provê um conjunto de funções e

algoritmos, simples mas eficiente; e a abordagem baseada em agentes prove uma estrutura mais deliberativa para a implementação o que facilita a distribuição e flexibilidade. A abordagem biblioteca é usualmente considerada como uma melhor abordagem pois a partir das funcionalidades disponibilizadas é possível criar um infra-estrutura de agentes (interesse em se criar *wrappers* sobre estas funcionalidades [136]), enquanto não é possível transformar uma abordagem de agentes em um conjunto de funcionalidades que possam ser usados em separado a critério do desenvolvedor de jogos; isto sem levar em conta o fato que escolher uma abordagem que imponha uma certa metodologia ou arquitetura ao desenvolvimento de jogos digitais pode não ser viável para vários produtos, plataformas ou gêneros de jogos.

Para tentar atingir um nível de reuso maior e minimizar os riscos de problemas, a indústria de jogos começou cada vez mais a fazer uso de *frameworks* orientados a objetos no desenvolvimento de jogos. Um *framework* pode ser visto como um conjunto de blocos de software pré-fabricados que podem ser usados como base de desenvolvimento para novas aplicações [73] ou como um projeto reusável de todo ou parte de um sistema de software, descrito por um conjunto de classes abstratas e pela forma como as instâncias dessas classes se relacionam entre si [57].

Uma abordagem então seria tentar criar um *framework* que se encaixe entre as abordagens descritas anteriormente e possa ser personalizado pelo desenvolvedor se adequando ao perfil que este ache necessário e possibilitando a utilização de parte da infra-estrutura e não necessariamente do modelo todo.

Alguns requisitos destes *frameworks* para jogos foram levantados por [135] e cinco deles também se aplicam ao desenvolvimento de middlewares de IA. Um *framework* de IA tem então que ser de fácil utilização por parte dos desenvolvedores, ajudar na melhoria da jogabilidade, prover suporte a *scripts* e ao gerenciamento de objetos e claro, como o próprio nome diz, dar suporte a técnicas de IA.

Várias questões são de grande importância e devem ser examinadas ao tentar criar um *middleware* de IA. Por exemplo, em um jogo de estratégia, é necessário aplicar IA num nível estratégico a grupos de unidades, mas também é necessário aplicar técnicas de IA às unidades individualmente de modo a permitir que estas tenham uma certa autonomia quando não estiverem sobre o controle direto do jogador. Detalhes de baixo nível também são importantes [92][104] para garantir um bom desempenho da implementação do sistema assim como livrar o desenvolvedor da carga de ter que sempre implementar essas funcionalidades. Algumas características

desejáveis num *middleware* de IA são, por exemplo: usar gerenciamento de eventos ao invés de *polling* (o ato de ficar constantemente perguntando por um recurso até que este esteja disponível), tentar centralizar a cooperação entre os gerenciadores do jogo e executar os procedimento de IA com a menor freqüência possível de modo a reduzir a carga de CPU, usar IA com nível de detalhe (*level of detail* - LOD, deste modo dedicando menos poder de processamento a entidades cujo comportamento atual não afete diretamente o usuário), fazer uso do máximo de pré-processamento possível (o que também tem o benefício adicional de facilitar a separação entre dados e o projeto em si) e dar preferência a comportamentos emergentes ao invés de usar soluções por meio de *scripts*.

Também é importante sempre atentar para a separação clara entre o subsistema de IA e o resto do jogo, pois o objetivo é criar um *middleware* de inteligência artificial flexível e não a criação de uma solução para um problema específico.

Um *middleware* de IA geralmente apresenta um ciclo de execução com a seguinte estrutura [72]:

1. Sentir: Acessar a informação sensorial do mundo

2. Pensar: Processar o conhecimento adquirido (informações sensoriais), levar em consideração o estado do agente e do mundo e escolher um curso de ação

3. Agir: Executar as ações

De modo a alcançar este objetivo, a interface com o mundo de jogo é um ponto crucial, e deve seguir dois princípios simples: ser tão próxima quanto possível da disponível a um ser humano (ou das que estariam disponíveis às entidades sendo modeladas) e ter acesso direto às estruturas de dados, evitando a complexidade desnecessária associada à "simulação" dos sentidos (de modo a alcançar esses objetivos e ainda ter as funcionalidades de IA separadas do jogo em si, é necessário escrever algum "código cola", código que vai acessar as estruturas nativas e convertê-las para um formato que o módulo de IA possa entender). Por exemplo, para que um monstro sinta a presença do personagem do jogador, o monstro deve ver o jogador, a interface aqui é o sentido de visão, mas não é necessário pegar o modelo da tela e baseado em técnicas de processamento de imagem tentar reconhecer o jogador, deve ser suficiente acessar a estrutura de dados relevante que mostre se o jogador se encontra na frente do monstro ou não e

converter essa informação em uma estrutura reconhecível pelo *middleware* de inteligência artificial.

O mundo do jogo é o simulador mestre e é reponsável por gerenciar o estado do jogo e prover todos os serviços relacionados a ele. Estes serviços são exatamente os sensores e atuadores, os sensores permitem receber eventos ou realizar *polling* (tentar acessar alguma informação) se necessário, e os atuadores representam as ações (ações de curta duração, tais como pular ou disparar uma arma) e os efetuadores (ações de maior duração e que estão sujeitas a mudanças no mundo do jogo, como por exemplo, caminhar até uma certa posição). Outro aspeto importante é a maneira como o mundo do jogo é modelado, pois dependendo da estrutura do mundo do jogo, interfaces diferentes (ou código cola) podem ser necessárias, assim como a cooperação com outros módulos do jogo, como por exemplo, o motor de modelagem física (que alguns consideram parte do motor de IA) de modo a traduzir um comportamento de movimentação em movimento real.

Outra opção relacionada à interface entre o *middleware* e o ambiente do jogo seria estabelecer interfaces padronizadas para um conjunto bem definido de funcionalidades (*path-finding*, por exemplo). Definir estas interfaces também tem seus problemas como por exemplo: De que forma caracterizar conceitos como *path-finding* ([136])? Em que nível definir essas interfaces, em um nível básico como gerenciamento de uma lista de nós na implementação do algortimo A* ou em um nível de interfaces completas para problemas como *path-planning* (que usa *path-finding* como sub-sistema)?

3.4 Estado da arte

Nas próximas seções serão apresentados diversos *middlewares* de IA comerciais e acadêmicos como parte de um levantamento do estado da arte na área. Serão discutidas as abordagens que estes empregam para tratar o problema de criar um motor de IA para jogo, além de sua estrutura geral e componentização.

3.4.1 DirectIA

Desenvolvido pela Mathematiques Appliquees S.A. (Masa), este *middleware* segue uma abordagem diferente da maioria dos demais pois usa uma abordagem biológica/evolucionária [74]. DirectIA [7] provê ferramentas

e suporte à criação de agentes inteligentes, com comportamentos variando de reações básicas a análise mais profunda do estado do mundo. DirectIA é uma ferramenta centrada no conceito de agentes, a qual apresenta considerações significativas sobre como e porque um determinado personagem toma uma decisão, traz inspiração da modelagem de comportamentos animal e humano e usa um modelo motivacional, i.e., um mecanismo de seleção de ações que imita o mecanismo utilizado por animais [55]. O mecanismo empregado pelo DirectIA [169][174] manipula estímulos, emoções, estados (interno e externo), motivações, comportamentos e as ações do agente.

Tal sistema pode ser visto como um conjunto de motivações que compete até que se decida qual será aplicada a uma determinada situação, dados os estados internos e externos, resultando em um comportamento emergente. Os agentes inteligentes podem pesar os diferentes *tradeoffs*, aprender a partir das próprias experiências e apresentar comportamentos não especificamente programadas pelos desenvolvedores.

DirectIA, provê diferentes componentes agrupados de forma compacta, um motor motivacional para modelar as emoções e necessidades dos agentes inteligentes, um motor comportamental para modelar os processos de decisão dos agentes, um motor de comunicação que dá suporte a comunicação entre agentes, um motor de percepção, um motor de ações e um motor de conhecimento para guardar a representação do mundo de cada agente. Mas a arquitetura do Direct IA também pode ser considerada uma arquitetura em camadas, pois as funcionalidades de alto nível provêem uma abordagem baseada em agentes e as funcionalidades de baixo nível provêem suporte a técnicas comumente usadas como *path-finding* e *steering*.

DirectIA oferece um processo de decisão em tempo real e ferramenteas para modelagem de comportamento de ações, na forma de um pacote de ferramentas de alto nível (que suporta agentes complexos ou reativos) e um pacote de ferramentas de baixo nível.

Um ambiente gráfico de interface com o usuário (GUI) para testes também é disponibilizado pelo *middleware* [126], mas o *toolkit* como um todo é voltado para programadores e o desenvolvedor tem que criar várias classes em C++ de modo a poder utilizá-lo. Também é possível definir diferentes comportamentos através de *scripting* e arquivos de configuração que são inicializados e carregados pelos vários *engines* do DirectIA em tempo de execução.

3.4.2 Ai.implant

Criado pela BioGraphic Technologies, a ferramenta AI.implant [8] (anteriormente chamada ACE) é um exemplo do uso de árvores de decisão. O ACE era originalmente um motor de controle de animações que foi projetado de modo a introduzir inteligência artificial no processo de desenvolvimento de personagens para jogos de computador e vídeo. Essencialmente, AI.implant provê controle autônomo dos personagens ao motor do jogo ou de animação através de vários comportamentos pré definidos (tais como "Evitar Obstáculos" por exemplo) que são atribuídos a um "agente" para implementar um desejado conjunto de ações (evitar que um guarda ande para cima de uma parede). Sensores são criados para o "agente", permitindo que este perceba eventos do mundo do jogo e, baseado nesses eventos, usa uma árvore binária de decisão (*Binary Decision Tree* - BDT) para selecionar o curso de ação. A BDT pode ser usada para criar comportamentos complexos de profundidade arbitrária. É até possível construir uma máquina de estados finitos usando a BDT de modo apropriado, além dos agentes poderem ter comportamentos padrão (*default*) atribuídos a eles para que sejam executados no caso da ausência de um comportamento determinado pela BDT. Como o AI.implant trabalha de maneira muito próxima a pacotes de modelagem 3D (através de uma interface de *plug-ins* para o Maya da Alias e o Autodesk 3ds Max), ele também provê funcionalidades para o posicionamento visual de *waypoints* no terreno do cenário criado de modo a ajudar os personagens a realizar *path-finding*.

Também é disponibilizado um *kit* de desenvolvimento (SDK) em C++ para permitir chamar as funções do AI.implant de dentro do código do jogo. Com relação a *path-finding*, o pacote também dá suporte a *path-finding* hierárquico (onde os caminhos vão sendo refinados aos poucos) e através do uso dos *plug-ins* pode realizar automaticamente a criação de uma rede de *waypoints* que pode ser posteriormente modificada se necessário [139].

3.4.3 Renderware AI

Ainda outro *middleware* de inteligência artificial comercial é o RenderWare AI (também chamado RWAI) [9], parte do pacote Renderware da Criterion Inc. (recentemente comprada pela Eletronic Arts - EA), que apresenta um *design* mais puramente em camadas e focado nos

desenvolvedores, oferecendo uma camada arquitetural, uma camada de serviços, uma camada de agentes (na verdade, uma camada de comportamentos) e uma camada de decisão [127].

A camada *Agents* consiste de um conjunto de comportamentos que podem ser instanciados por um personagem do jogo (comportamentos como "fuga", "ataque a distância", etc.). Por sua vez, a camada *Decisions* provê um "objeto cérebro" e cada entidade pensante tem associado a ela um objeto desse tipo e é este cérebro o responsável pelo processo de decisão.

A camada *Services* basicamente consiste de diferentes gerenciadores (gerenciador de caminhos, gerenciador de entidades, gerenciador de fontes de som, etc.) e a camada *Architecture* lida com a inicialização, atualização e finalização das camadas do RWAI sob o controle do motor do jogo. Com relação a *path planning*, RWAI também provê um analisador dinâmico de topologia que ajuda os NPCs a identificar obstáculos, inimigos, localizar sons, etc.

Renderware AI oferece código fonte aberto e totalmente modificável, um conjunto de classes em C++ que os desenvolvedores podem estender e ajustar/configurar para criar seus próprios agentes. RWAI oferece ainda as opções de FSMs e de redes neurais para serem utilizadas no processo de decisão dos personagens.

Como praticamente todos os *middlewares* disponíveis, ele divide os objetos do jogo em duas categorias, entidades pensantes e entidades passivas para auxiliar sua representação interna do mundo do jogo.

Uma das vantagens do RWAI é que ele pode ser utilizado em conjunto com os demais components da plataforma RenderWare (como motor gráfico, motor de som, etc.) evitando problemas de integração entre diferentes pacotes de software, o que, aliado à disponibilidade do código fonte, o torna um opção bastante atraente para os desenvolvedores de jogo. Devido a essas características e a sua maturidade em relação aos demais *middlewares*, o RenderWare AI é o *middleware* de IA mais usado em jogos digitais atualmente.

3.4.4 Simbionic

SimBionic [10], desenvolvido pela Stottler Henke, apresenta outra abordagem (supostamente mais poderosa que as usuais) no projeto conceitual dos softwares de suporte a funcionalidades de IA provendo um *framework* para a definição de objetos que exibem comportamentos dentro

do mundo do jogo (uma abordagem que se encaixa como um meio termo entre FSMs e RBSs). Este *framework* é bastante orientado a estados, possibilitando a criação de complexos sistemas hierárquicos de estados.

Estes sistemas de estados possuem diversos componentes [128] que podem ser classificados como descritores e declarações. Descritores são identificadores de tipos que podem ser usados para representar objetos e comportamentos que existem no mundo do jogo, assim como relacionamentos hierárquicos entre eles possibilitando um certo poliformismo aos objetos.

SimBionic também descreve os atributos de um objeto como descritores, por exemplo, um descritor Arma pode ter dois descritores filhos Revolver e Rifle, Rifle por sua vez pode ter um filho rifle, e assim adiante; e os atributos de Arma (como peso e munição) também podem ser representados por descritores. Declarações são associações simbólicas usadas por um projeto criado no SimBionic.

Estas associações consistem de ações, predicados (funções *built-in* que provêem serviços de acesso e avaliação), comportamentos e variáveis. Um comportamento implica na seleção de um outro comportamento ou de uma ação, e é responsável pela "tomada de decisão" no SimBionic.

O motor de execução (*runtime engine*) do SimBionic provê APIs simples em C++ e Java de modo que o desenvolvedor basicamente precisa apenas notificar o *middleware* de cada evento do relógio (*clock tick*) e incluir alguns arquivos de código (cabeçalhos e arquivos de código C++) no projeto do jogo e tudo estará pronto para rodar.

SimBionic oferece também um depurador interativo e um editor visual para os comportamentos, o que aumenta bastante a produtividade (além de serem bem amigáveis a não programadores) uma vez que possibilita a autoria dos comportamentos sem praticamente nenhuma programação.

Além disso, o *toolkit* também permite que as entidades comuniquem dados e informação de status entre si, seja usando mensagem por grupo ou o conceito de *blackboards*, uma característica não explorada na maioria dos *middlewares*.

3.4.5 Soar

Algumas abordagens usam motores de inferência para "concluir" qual o melhor curso de ação num determinado momento. Um exemplo de abordagem nesse gênero, inicialmente usada em simulações militares,

é a arquitetura Soar [11][36] que combina a reatividade dos sistemas estímulo resposta e os comportamentos sensíveis ao contexto passíves de representação em sistemas que fazem uso de FSMs or *scripting*.

O *design* do Soar é baseado na hipótese de que todo comportamento deliberativo baseado em objetivos pode ser definido com a seleção ou aplicação de um operador de estado. Nessa arquitetura, o conhecimento é representado então por uma hierarquia de operadores. Cada nível nessa hierarquia representa um nível de decisão mais específico no comportamento do agente inteligente. Os operadores que se encontram no nível do topo representam os objetivos e modos de comportamento do agente. Os operadores do segundo nível (de cima para baixo) representam as táticas de alto nível utilizadas para alcançar os objetivos da camada superior. Os operadores das camadas mais baixas representam os passos (ou sub-passos) necessário para o agente implementar e executar as táticas. A arquitetura Soar também permite a persistência dos operadores escolhidos assim como da memória interna dos agentes, o que possibilita aos agentes inteligentes reagir levando em consideração o contexto do ambiente em que se encontram.

Soar usa memórias separadas (e representações diferentes) para a descrição da situação atual e para o conhecimento de longo prazo. Dados da situação atual (que incluem dados dos sensores, resultados intermediários das inferências, objetivos ativos, etc.) são guardados numa memória chamada memória de trabalho. O "conhecimento de longo-prazo", que especifica como responder às diferentes configurações dessa memória de trabalho, pode ser visto como um programa para o Soar. Este conhecimento é organizado como regras de produção usando as funções de seleção e aplicação de operadores (cada uma das funções é descrita em detalhes no manual [12]). Essa representação é praticamente uma linguagem de programação especializada em cima de uma teoria de quais são as primitivas base para raciocínio, aprendizado e planejamento [43] e o Soar provê um editor visual (chamado *Visual Soar*) para ser usado no desenvolvimento destes programas.

Nesse *framework*, o ciclo de execução pode ser descrito como entrada, percepção, elaboração, decisão e saída. Percepção ocorre na fase de entrada e após as percepções serem processadas, estas são enviadas à memória de trabalho. Durante a fase de elaboração, regras de produção são casadas ao conteúdo da memória de trabalho e disparam em paralelo até que não haja mais regras a ser disparadas. As regras disparadas durante esta fase não alteram a memória de trabalho, apenas criam referência a mudanças

e geram quais seriam os efeitos atuadores necessários. Estes comandos para os atuadores serão enviados ao motor do jogo durante a fase de saída. Durante a fase de decisão, as referências criadas são avaliadas e se decide que mudanças serão realizadas na memória de trabalho, e qual o operador a ser aplicado ao contexto atual. Uma vez que a fase de decisão acaba, o ciclo de raciocínio recomeça. Em resumo, embora o Soar use regras de produção como sua linguagem de representação de baixo nível, os processos empregados (como uma teoria cognitiva e a representação alto nível do conhecimento) em conjunção com essas regras fazem com que seu comportamento difira bastante de outro RBSs.

Embora seja uma abordagem acadêmica e extremamente pesada, o Soar tem sido aplicado com êxito em jogos gerando agentes que conseguem antecipar o comportamento do jogador no jogo Quake da id software [93] e também (e talvez mais importante) na criação de uma base de conhecimento sobre o comportamento inteligente de um agente para jogos do gênero FPS [72].

3.4.6 Emotion AI

Um novo *middleware* é o Emotion AI [13] desenvolvido pela Neon AI; ele tenta prover um novo *framework* para a criação de IA de personagens em jogos através da simulação de comportamentos emocionais até certo ponto humanos. Apenas emoções não cognitivas estão disponíveis na versão atual do motor [94] mas o modelo de agentes já é capaz de simular uma gama de diferentes necessidades motivacionais, e estas necessidades podem ter seus níveis de "satisfação" ajustados.

O *Emotion AI Engine* conceitualmente divide as emoções em três camadas de comportamento. Na camada superior estão as "reações" ou "emoções momentâneas"; estes são os comportamentos que uma pessoa apresenta brevemente durante a reação a eventos. O nível intermediário representa os "humores" (*moods*), estes são estados emocionais prolongados causados pelos efeitos cumulativos das emoções momentâneas, na forma de sinais de punição e recompensa (i.e., são gerados em resposta a eventos ocorridos no mundo e decaem em taxas dependentes da personalidade). Como base para essas duas camadas e sempre presente está a camada de "personalidade"; estes são os comportamentos que geralmente são exibidos quando nenhuma emoção temporária ou humor os sobrepõe.

O modelo de cérebro proposto neste *middleware* é também afetado por "necessidades motivacionais" (inspiradas na hierarquia de necessidades de Maslow [34]) que são usadas como uma faixa de necessidades dos agentes que requer mais atenção em um dado momento. Por exemplo: necessidades que requerem a atenção mais urgente, necessidades que se satisfeitas vão propiciar a maior recompensa, necessidades (que se não satisfeitas) vão causar as maiores punições (recompensa negativa) [94].

Os eventos do ambiente do jogo são enviados ao modelo cerebral, são influenciados pela lista de necesidades, pelo modelo cerebral em si e então se decide que ações serão realizadas e as mudanças a serem efetuadas no estado atual do cérebro. Com este modelo, o SDK do *Emotion AI Engine Middleware* consegue prover personagens com comportamentos bem elaborados, similares aos apresentados pelos personagens do jogo Black & White da Lionhead.

3.4.7 Outros Middlewares

Existem ainda outros *middlewares* acadêmicos e comerciais de inteligência artificial, mas ou têm utilização muito pequena ou ainda se encontram em fase de desenvolvimento. No entanto, a seguir serão apresentados ainda mais três motores de IA que também foram analisados durante o estudo da arte dos softwares deste tipo disponíveis e que acabaram se revelando muito simples (BEHAVIOR) ou não são mais comercializados (Pensor).

3.4.7.1 BEHAVIOR

Um exemplo de ferramenta simples é o *toolkit* SOFTIMAGE/BEHAVIOR [14], um *toolkit* focado em animação por computador que faz uso de FSMs hierárquicas de modo a permitir melhor representação e controle de sistemas em tempo-real grandes, como multidões complexas por exemplo, permitindo que o animador "dê" inteligência a sua multidão e a personagens individuais dentro dela. Esta ferramenta também provê uma ferramenta visual para a criação e edição de HFSMs graficamente e suporta uma linguagem de *script* para aumentar sua flexibilidade. Usando o BEHAVIOR, pode-se também depurar visualmente o progresso através das HFSMs e dos *scripts* quando de sua execução. Outro serviço que esse *toolkit* provê é o serviço de *path-*

planning, o qual automaticamente gera *waypoints* e sobre o qual é possível definir comportamentos aos personagens (o comportamento pré-definido de evitar obstáculos é um exemplo). Esta ferramenta já foi usada em vários jogos digitais comerciais e está disponível há algum tempo, mas é muito focada em animação e na maioria das vezes é usada apenas em situações relacionadas à produção de vídeos para os jogos (*cut scenes*).

3.4.7.2 Pensor

A maior parte dos motores de inteligência artificial (ou dos pacotes de software que assim se classificam) pode ser categorizada como pacotes de funcionalidades relacionadas, abordagens em camada ou componentes semi-independentes. Alguns desses pacotes de funcionalidades seriam: planejadores, *path-finding*, tomada de decisão, gerenciamento de ações, sensores, infra-estrutura, aprendizado, comunicação, emoções, motivações, coordenação e análise tática. Um exemplo de *middleware* de IA que é organizado como conjuntos de funcionalidades relacionadas é o Pensor [15], que é composto de um conjunto de implementações de algoritmos/técnicas/tecnologias de certa forma recombináveis em cada novo projeto.

Pensor tem como componentes seis planejadores, três deles sendo *path-finders* otimizados (*path-planners*) [129] que levam em consideração informação de análise do terreno quando calculam os caminhos; um módulo de decisão que implementa FSMs, *fuzzy* FSMs, regras e um *shell* [130]; um módulo atuador, que implementa modelagem física básica e vários comportamentos de movimentação; um módulo de percepção que provê diversos sensores; um módulo de infra-estrutura responsável por prover serviços de suporte a priorização de tarefas e gerenciamento dos recursos necessários a elas e também um interpretador para uma linguagem de *script*.

A empresa Mindlathe foi a desenvolvedora da tecnologia que formam a base do Pensor, mas atualmente este pacote parece não mais estar disponível.

3.4.7.3 FEAR

FEAR [16] é um projeto código aberto que provê uma especificação, um *framework* (com funcionalidades para suportar sistemas de IA) e vários

módulos implementando técnicas comumente utilizadas (i.e. árvores de decisão, FSMs, lógica *fuzzy*, *perceptrons* e um RBS simples).

Um ponto interessante deste pacote é que ele usa modelos XML para geração de código, o que pode ser bastante útil para facilitar a vida dos desenvolvedores. A partir dessa abordagem, a maior parte do "código cola" ("*glue code*" - código que une o pacote de IA ao motor do jogo) é automaticamente gerada através do uso de *templates* C++ e meta-programação com *templates*. O que, junto ao fato de ser (por enquanto) muito voltado para o motor do jogo Quake 2 da id software, acaba fazendo com que não possa ser tão facilmente aproveitado.

3.5 Outras considerações

Além das abordagens mencionadas, não se deve esquecer das possibilidades apresentadas ao se fazer uso de técnicas de aprendizado de máquina integradas ao middleware de inteligência artificial de modo a obter novos (possivelmente emergentes) comportamentos [75].

Uma outra consideração importante a ser feita é a importância de dedicar atenção especial à base de conhecimento que contém os objetivos, táticas e comportamentos independentes do jogo (mas não necessariamente do gênero), segundo Laird [72], esta é a característica mais importante numa infra-estrutura de IA. Sem essa base, o motor de inferência (ou, de modo mais geral, o componente responsável por tomar as decisões de fato), não vai ser muito útil. Uma outra consideração de igual importância, levantada por Leonard [140], é o valor incalculável de se construir um bom conjunto de sensores de modo a melhorar a experiência do jogador quando interage com o mundo do jogo, possibilitando a criação de ambientes muito mais ricos pelos designer e desenvolvedores.

Além destas questões, uma das mais importantes questões de design durante o projeto de um middleware de IA é definir quem será o público alvo da ferramenta. Pacotes como o SimBionic e o AI.implant, por exemplo, tem como alvo animadores visuais e game designers (especialmente o AI.implant), apresentando ferramenteas visuais fáceis de usar e muitas vezes integradas a pacotes de software para modelagem 3D e que utilizando abordagens que não requerem conhecimentos de linguagens de programação. Por outro lado, ferramentas como o DirectIA e o Renderware AI são claramente focadas nos desenvolvedores de jogos (especialmente o Renderware AI) exigindo que os desenvolvedores tenham um bom

entendimento das implementações em baixo nível no jogo e muitas vezes oferecendo o código fonte para ser modificado pelos desenvolvedores (embora esse não seja o caso do DirectIA).

3.6 Conclusões e observações

A maioria dos *middlewares* e motores de IA comerciais atuais pode ser considerada como abordagens em camadas ou em componentes semi-independentes e propõem suas próprias maneiras de tratar o problema, apenas cobrindo um sub-conjunto relativamente pequeno das capacidades e funcionalidades de IA. Uma tabela comparativa das características discutidas dos principais *middlewares* de IA disponíveis pode ser vista na Tabela 3.1.

Conforme apresentado, várias soluções para *middlewares* de IA estão florecendo, mas pouco ainda se sabe sobre a possibilidade real de se criar um conjunto de interfaces padronizadas para facilitar a criação de *middlewares* de inteligência artificial e como esse padrões vão se relacionar com as questões de implementação. Cooperação entre a indústria de jogos digitais e a academia é crucial neste esforço pela criação de tais padrões mesmo se os objetivos de ambas as comunidades não forem os mesmos.

Muito mais esforço terá que ser dedicado (e está sendo dedicado atualmente) à questão da padronização de interfaces para os *middlewares* de IA em jogos digitais. Foi com isso em mente que a *International Game Developers Association* (IGDA) formou o *AI Interface Standards Committee*, uma inicativa conjunta de desenvolvedores de IA para jogos, representantes das empresas de *middleware* e representantes da academia [152][171]. É esperado que com o avanço destes padrões, os *middlewares* de IA possam realmente alcançar aceitação generalizada.

Mesmo com essas questões em aberto, inteligência artificial em jogos vai ser uma crescente prioridade no processo de desenvolvimento de jogos digitais por muito tempo, principalmente por ser uma área ainda relativamente pouco explorada mas já ter mostrado que pode trazer grandes avanços em game design e jogabilidade. *Middlewares* de IA serão tão essenciais ao sucesos dos jogos quanto os motores gráficos o são atualmente.

No entanto, a síndrome do "não inventado aqui" (que é muito comum na indústria dos jogos digitais), o receio de não ter completo controle sobre o jogo e a possível perda de desempenho que pode ocorrer com o uso de um *middleware*, "motor" ou biblioteca de IA são alguns dos obstáculos para

a adoção maciça de *middlewares* de IA pelas empresas desenvolvedoras de jogos. Alguns alegam que *middlewares* que não oferecam licenciamento de código fonte adiciona grandes riscos ao projeto e este tem sido um grande fator em não se utilizar alguns pacotes de software [17]. Atenção também tem que ser dada para evitar que a curva de aprendizado para implementar e integrar o *middleware* de IA ao jogo seja muito elevada.

Mas pelo lado positivo, motores de IA já são uma realidade e estão sendo empregados mais e mais. Os desenvolvedores têm que realmente analisar de forma consciente as necesidades dos jogos e o que os middlewares de IA supostamente podem prover. Este é com certeza o passo mais importante a ser dado, pois não entender os requisitos do jogo muito provavelmente vai levar a tomada de decisões ruims [141], seja de usar ou não um certo produto.

Tabela 3.1: Uma comparação entre diferentes *middlewares* de Inteligência Artificial (inspirada em [18]).

	Suporte a decisão	**Serviços**	**Comportamentos**	**Código fonte**	**Ferramentas**
DirectIA	MDGs	*Path-finding*	*Templated*	Não	*Scripts*, *templates*, GUI
AI.implant	BDTs	Geração automática de caminhos, *path-finding*	*Pre-packaged*	Algum	*Plug-ins* p/ Maya e 3DS
RWAI	FSMs e NNs	Geração automática de caminhos, *path-finding*, outros	*Pre-packaged*	Algum	Editor e *debugger* visuais
SimBionic	*FSM-like*	Comunicação entre agentes	Criados pelo usuário	Algum	Editor e *debugger* visuais
Emotion.AI	Modelo cérebro / emoção	N/A	Personalidades / emoções	N/A	N/A
Pensor	FSMs e regras	*Path-finding* e física básica	*Pre-packaged*	Não mais disponível	N/A
Behavior	FSMs	*Path-finding*	N/A	Não	Editor e *debugger* visuais