4
Arquitetura Proposta

Os capítulos anteriores apresentaram uma análise do domínio da criação de middlewares de inteligência artificial através de um levantamento de técnicas de IA usadas nessas aplicações e do estudo do estado da arte em middlewares de IA comerciais e acadêmicos.

Neste capítulo será apresentada a arquitetura do *middleware* proposto nesta dissertação, chamado MIAGI - um *Middleware* de Inteligência Artificial para *Games* Interativos. Este *middleware* apresenta uma arquitetura em camadas [56] pois as questões de nível mais alto na modelagem da IA necessitam de suporte das camadas mais baixas (técnicas e comportamentos mais simples) e é apresentado na forma de um *framework* que pode ser instanciado pelo desenvolvedor de IA em jogos digitais. De modo a ter um *design* mais robusto e bem fundado e de facilitar futuros desenvolvimentos e remodelagens, faz-se uso de padrões de projeto orientados a objeto na sua estruturação e documentação [47].

A seguir serão apresentadas a evolução dessa arquitetura e a metodologia empregada, suas diferentes camadas (e os componentes destas) e algumas considerações.

4.1
Evolução da Arquitetura e Metodologia Empregada

A idéia inicial deste trabalho surgiu no contexto do grupo de pesquisa em jogos do Centro de Informática da UFPE onde foram desenvolvidos vários trabalhos sobre a aplicação de IA em jogos digitais (tais como: o uso de atores sintéticos e modelagem de emoção e personalidade [95] e arquitetura de IA para jogos de estratégia [96]), além de trabalhos sobre a estruturação de motores de jogos via técnicas de engenharia de software (Forge V8 [99], wGEM [100], Forge 16V [135]). Tendo como inspiração o esforço de acrescentar funcionalidades básicas de IA a um desses *frameworks*, o wGEM [131].

Surgiu assim a iniciativa de se implementar um *middleware* específico para tratar as questões de IA em jogos digitais (levando em consideração as vantagens e desvantagens do uso de *frameworks* em jogos [99] e IA [97]). A abordagem inicial no desenvolvimento do primeiro protótipo desse *middleware* foi uma abordagem *bottom-up*, onde, através da implementação das funcionalidades de mais baixo nível e sua combinação, a solução final pode "emergir". Esta abordagem permitiu ter uma versão rodando logo, o que facilitou o processo de teste de novas características e funcionalidades.

Algumas das decisões de projeto deste protótipo implementado foram inspiradas pelo RenderWare AI [9], um *middleware* de IA que apresenta uma abordagem de projeto em camadas (arquitetural, serviços, agentes e decisão); a camada de agentes sendo composta de um conjunto de comportamentos que podem ser instanciados por um personagem de jogo. Outra grande influência na componentização foi o pacote Pensor, o qual era formado por um conjunto recombinável de algoritmos e técnicas; planejadores, *path-finders*, um módulo de decisão usando FSMs e regras, um módulo de percepção e um módulo de infra-estrutura que provia suporte ao gerenciamento de recursos (uso de CPU e alocação de memória).

Este protótipo inicial [142][143] foi organizado em três macro camadas, Serviço, Comportamento e Cognitiva (Figura 4.1). A camada Serviço, gerenciava detalhes de baixo nível da implementação de modo a garantir um bom desempenho do sistema além de liberar o desenvolvedor do jogo desta carga (por exemplo: usar manipulação de eventos ao invés de *polling*; centralizar a cooperação dos gerenciadores do jogo, etc.).

A camada Comportamento inicialmente implementada consistia da implementação de FSMs (pois estas alcançam bons resultados na modelagem de comportamentos simples dos personagens do jogo e são uma ferramenta familiar para os desenvolvedores de jogos) e um conjunto simples de "regras motivacionais" que permitiam a seleção do comportamento a ser executado em um dado momento. A camada Cognitiva, por sua vez, não havia sido implementada, estando ainda em estudo.

O protótipo foi desenvolvido e testado com um conjunto de jogos simples (sendo necessário apenas escrever um código cola para começar a utilizá-lo, sem grandes preocupações em torná-lo independente do gênero de jogo) e a maior parte das decisões de implementação deste protótipo são relacionadas a detalhes de mais baixo nível.

Mesmo sendo um protótipo bem simples, essa primeira versão do motor de IA já forneceu várias clarificações sobre a criação desse tipo de ferramenta. Outra vantagem desse primeiro protótipo foi permitir o seu uso

como uma ferramenta educacional exemplificando a implementação de IA em alguns jogos simples.

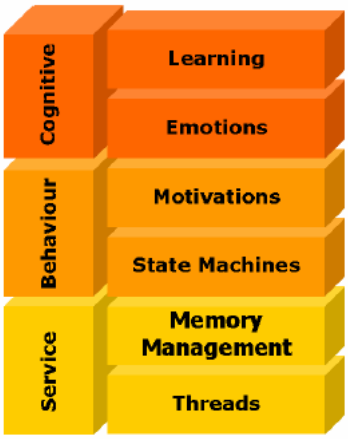

Figura 4.1: Camadas da arquitetura do protótipo do *middleware*.

4.1.1 Metodologia de evolução de frameworks

O projeto de um *framework* é algo bastante complexo, pois é preciso chegar a uma abstração que cubra os principais problemas de um certo domínio. Além disso, um *framework* precisa ser simples para ser aprendido facilmente e ainda assim possuir funcionalidades suficientes pra poder ser utilizado na prática [57].

Para facilitar a criação de *frameworks*, Roberts e Johnson [57] propuseram uma metodologia de desenvolvimento de *frameworks* através de uma linguagem de padrões (padrões que são relacionados entre si).

De modo a criar um *framework*, é necessário encontrar abstrações para o domínio, geralmente a partir de exemplos concretos. Assim, Roberts e Johnson definiram o padrão "Três Exemplos" como o primeiro passo no desenvolvimento de um *framework*. Resumidamente, o padrão identifica que é necessário prototipar-se ao menos três aplicações em seqüência nas quais o *framework* deveria poder ser aplicado, sendo que cada aplicação deve ser um pouco diferente da outra, facilitando a identificação dos pontos de abstração.

Durante o desenvolvendo desses três protótipos, deve-se sempre ter em mente que o objetivo final é o desenvolvimento de um *framework*, por isso é essencial tornar as aplicações desenvolvidas o mais flexíveis e estensíveis possível (através de mecanismos simples de orientação a objetos como, por exemplo, herança). Como resultado desta etapa, obtém-

se um "*framework* de caixa branca" (nome dado ao segundo padrão da metodologia), pois o *framework* confiará fortemente nos mecanismos de herança e encapsulamento quando usado para desenvolver novas aplicações [58].

Uma vez que se tenha um primeiro *framework* ao final do primeiro protótipo, deve-se analisá-lo para tentar descobrir possíveis pontos de mudança e pontos estáveis no desenvolvimento das próximas aplicações teste. Sempre que for necessária uma nova classe similar a alguma já foi desenvolvida no *framework*, cria-se uma subclasse e sobrescrevem-se os métodos diferentes. Após o desenvolvimento de algumas destas subclasses, será fácil identificar os métodos que não precisam ser sobrescritos. Nesse ponto, será possível fazer o *refactoring* [76] do *framework* criando-se classes abstratas que contêm as partes comuns.

Também é importante identificar as principais funcionalidades que serão utilizadas no desenvolvimento das aplicações do domínio em questão. Por isso, é importante começar a construir uma boa "Biblioteca de Componentes" (terceiro padrão).

Uma outra questão importante que deve ser observada no desenvolvimento das demais aplicações teste é a detecção dos pontos que mudam de uma aplicação para outra. Esses pontos são conhecidos como "*Hot Spots*" (quarto padrão). Coletando-se esses pontos em objetos conhecidos, fica mais fácil o processo de reuso do *framework* e mostra aos usuários do *framework* onde os projetistas esperam que o *framework* mude. Após coletar os *Hot Spots* em objetos conhecidos, usa-se a composição ao invés da herança para se conseguir a variação necessária de uma aplicação para outra.

Para facilitar a variação de código de uma aplicação para outra, geralmente são utilizados os "Objetos Plugáveis" (quinto padrão - objetos que se plugam através de composição tomando como base uma classe abstrata). Com eles a variação só precisa ser conhecida no protocolo de inicialização, ou seja, durante o restante da execução do programa, essa variação fica transparente.

Outro padrão utilizado no tratamento de características variáveis são os "Objetos Especializados" (*Fine Grained Objects*, sexto padrão proposto). Toda vez que se encontrar uma classe que encapsule múltiplas características que podem variar de forma independente, é melhor criar múltiplas classes para encapsular cada característica, criando-se assim, uma hierarquia mais especializada.

Uma vez que a biblioteca de componentes tenha sido organizada e

especializada através da herança, usa-se a composição de componentes para a criação da aplicação. Com isso, o *framework* torna-se um "*framework* de caixa preta" (sétimo padrão), ou seja, o usuário não precisa ter pleno conhecimento da hierarquia do *framework* para utilizá-lo [58]. Para ajudar a fazer a composição dos componentes, uma ferramenta visual que permita especificar de forma gráfica quais componentes estão presentes na aplicação e como eles se relacionam, deve ser desenvolvida. A esta ferramenta dá-se o nome de "*Visual Builder*" (oitavo padrão) e ela torna a utilização do *framework* muito mais fácil. Uma vez criada essa ferramenta visual, tem-se uma linguagem visual e, como acontece com qualquer linguagem, faz-se necessária a criação de "Ferramentas da Linguagem" (nono padrão) que facilitam a depuração e a inspeção da mesma.

4.1.2 Metodologia aplicada ao middleware

Baseando-se na arquitetura modular proposta inicialmente, foi adotada uma política de desenvolvimento incremental e de aprendizagem gradativa. Sendo assim, iterações sobre o modelo original foram sendo realizadas, refinando a arquitetura.

Essa nova estrutura em camadas foi inspirada na arquitetura do protótipo, mas também recebeu influências de outras três abordagens:

a) um modelo organizacional parecido com o do cérebro onde funções são construídas umas sobre as outras (Reflexos e sobrevivência - Tronco cefálico; Cordenação motora e sensorial - Cerebelo; Funções de alto nível, emoções e aprendizado - Lóbulo frontal; Memória - Lóbulo temporal; Cortex sensorial e processamento visual - Lóbulo pariental e Lóbulo Occipital);

b) a idéia de arquiteturas com múltiplas camadas de decisão aprendida de esforços em robótica [45];

c) o conceito de IA distribuída detalhado por Schwab [151] (com as seguintes camadas: Percepção/Eventos, Comportamento, Animação, Movimento, Decisão de curto prazo, Decisão de longo prazo e informação baseada em localização).

A estrutura em camadas proposta é mostrada na Figura 4.2, sendo que o módulo de aprendizado ainda não se encontra implementado (contendo

apenas um simples modelo de memória) e apenas uma modelagem emocional básica (através de traços de personalidade) é utilizada.

Seguindo a metodologia para evolução de *frameworks* apresentada anteriormente, iterações sobre o novo modelo vêm sendo realizadas, identificando pontos de generalização, refactoring e aplicando padrões de projeto. Deste modo, a partir de uma primeira implementação de um *framework* de caixa branca, partiu-se para a implementação de uma "biblioteca de componentes", implementando as técnicas mais úteis/comuns usadas em IA para jogos e analisando como estas interagem entre si e com o *framework* prototipado, sempre tentando identificar padrões de projeto pra ajudar a estruturar e documentar o *framework*.

A arquitetura atual usa um sistema baseado em eventos (criado com base em experiências anteriores no mundo de sistemas embarcados) e assim evita realizar *polling* freqüente do estado dos itens do mundo e provê uma organização estrutural e de código mais modular e flexível; o sistema de eventos pode ser usado tanto para representar dados sensoriais (como uma camada entre o *middleware* e o jogo em si), como também como meio de comunicação entre as diferentes partes da arquitetura (assim como com simples alterações pode ser usado para comunicações entre os agentes). Além disso, a arquitetura atual pode ser usada tanto seguindo uma abordagem de *framework* caixa branca quanto uma abordagem de *framework* caixa preta.

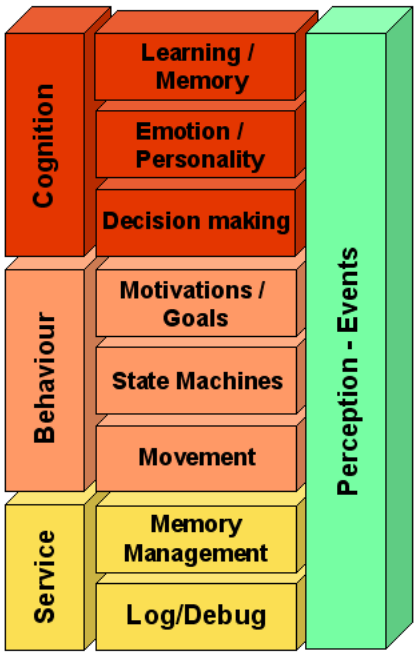

Figura 4.2: Camadas da arquitetura atual do MIAGI.

4.2 Camada Serviço

Esta é a camada responsável por gerenciar as questões de mais baixo nível da implementação do *middleware* de modo a garantir um bom desempenho do sistema.

Dentre as responsabilidades desta camada podem estar, por exemplo, balanceamento de carga, gerenciamento de memória, sistema de *log*, auxílio à depuração e módulo de infra-estrutura.

Por módulo de infra-estrutura, entende-se um módulo responsável por facilitar a centralização da cooperação entre os diversos gerenciadores parte do *middleware* (por exemplo, o gerenciador de entidades) e por prover os mecanismos pelos quais um jogo faz acesso a instância do *middleware* de IA (tratando configuração, incialização e finalização da mesma).

4.3 Camada Cognição

A camada de cognição é a camada de mais alto nível da arquitetura e é responsável pelo processo de raciocínio e decisão dos diversos agentes. Esta camada deve levar em consideração todos os fatores que possam influenciar o processo de decisão (de modo a torná-lo genérico e flexível) e ser capaz de determinar um curso de ação a ser tomado.

4.3.1 Aprendizado/Memória

Dentre as diversas técnicas de IA, provavelmente as que oferecem mais possibilidades são as de aprendizado. Estas são técnicas que podem mudar a maneira como um jogo é jogado, forçando o jogador a continuamente tentar novas táticas e estratégias, pois permitem produzir uma IA mais esperta e robusta e que consegue se adaptar a condições que não podem ser previstas.

Embora o uso de aprendizado ainda seja pouco explorado em jogos, esta técnica (ou melhor, conjunto de técnicas) pode se adequar muito bem aos mesmos [121].

Uma das técnicas mais "famosas" de aprendizado de máquina é a técnica de aprendizado por reforço [67]. Essa é uma técnica especialmente promissora com relação a sua aplicação em jogos, pois um motor que a implemente pode ser usado para tratar vários problemas não relacionados. Mas, embora genérica, ao utilizar essa abordagem podem ser encontrados

problemas em representar o mundo usando a formalização necessária. No entanto, a área de aprendizado de máquina é bastante vasta e diversos métodos podem ser utilizados, como por exemplo: aprendizado de árvores de decisão, redes neurais (já anteriormente comentado), aprendizado bayesiano, aprendizado de bases de regras, etc [59].

Alguns exemplos comerciais e acadêmicos do uso dessas técnicas são o uso de *perceptrons* e aprendizado de árvores de decisão no jogo 'Black & White' [120] e o aprendizado de regras *fuzzy* apresentado por Li [155].

Uma outra questão mais indiretamente relacionada a aprendizado é como realizar a representação de conhecimento na memória dos agentes de modo que estas informações possam ser usadas para influenciar o comportamento dos mesmos.

Uma possível proposta nessa área seria os agentes fazerem uso da abordagem de memória esparçamente distribuída (*sparse distributed memory* - SDM) como sua representação de memória associativa [144]. SDM é um modelo matemático para um tipo de memória endereçável por conteúdo, e pode ser um mecanismo bastante interessante para modelar memórias de longo prazo [40]. O fato de ser endereçável por conteúdo significa que itens na memória podem ser resgatados por meio de parte de seu conteúdo, sem que seja necessário saber um "endereço" ou identificador do item na memória nem varrer toda a memória a procura de um item.

Embora ambas as áreas (aprendizado e memória) apresentem características interessantes, um estudo mais amplo tem que ser realizado sobre sua utilização e integração dentro do *middleware* de IA. Uma possibilidade seria fazer uso da Torch [132], uma biblioteca para aprendizado de máquina disponível sob uma licença BSD [19].

Na versão atual da arquitetura, apenas o impacto de uma representação de memória simples é levado em consideração no processo de raciocínio e decisão dos agentes de jogo.

4.3.2 Emoção/Personalidade

De modo a conseguir personagens mais cativantes e com maior realismo, é necessário, além de um comportamento inteligente, dar-lhes capacidade de expressar emoções, personalidade e comportamento orientado a objetivos. A área que trata desse tema em modelagem de agentes inteligentes é chamada de *Synthetic Actors* (SA) ou *Believable*

Agents e vários modelos têm sido propostos para tratar do problema [60][61][179][49][105].

Personalidade é geralmente considerada como uma característica crucial para fazer com que agentes se comportem de maneira mais "viva", já que é expressada em termos de padrões de comportamento diferenciados que distinguem um indivíduo dos demais [32]. Emoção, também crucial, é uma ampla área de estudos e pode ser abordada via diversas áreas: neurologia [50], ciência cognitiva [37][68] e psicologia [62].

Para que os atores sintéticos atinjam os objetivos apresentados, é necessário que cumpram uma série de requisios [20]:

Personalidade - personalidade deve influenciar suas ações. As personalidades podem ser caricaturadas (como é comum no caso de desenhos animados) para se atingir o efeito desejado.

Emoção - agentes devem exibir emoções e, idealmente, responder às emoções dos outros de acordo com a sua personalidade. As emoções também podem ser exageradas para se melhorar a experiência fornecida ao usuário/jogador.

Auto-motivação - agentes não só reagem ao mundo, eles têm seus próprios desejos e objetivos, os quais devem perseguir.

Evolução - agentes evoluem e mudam com o tempo de uma maneira consistente com a sua personalidade.

Relações sociais (atitudes) - é através destas que o agente se relaciona com outros agentes. Tal relação pode mudar como resultado da interação entre eles.

Com relação a emoções, um modelo bastante popular é o OCC [41]. Este é um modelo descritivo de emoções e resulta de três tipos de avaliações subjetivas: da importância de eventos com respeito aos objetivos do agente, da aprovação das ações do próprio agente ou mesmo de outros com respeito ao conjunto de padrões de comportamento, e dos gostos e preferências do agente em respeito a objetos. Este modelo define vinte e quatro tipos de emoção, alguns primitivos e outros que resultam da combinação destes primeiros. OCC é usado, por exemplo, em [6][182][98].

Embora a frase dita por Minsky [38] "*The question is not whether intelligent machines can have emotions, but whether machines can be intelligent without any emotions*" resuma bem a importância da

representação de emoção, neste trabalho apenas personalidades serão tratadas, sendo emoções deixadas para trabalhos futuros.

Como dito por Bandura [63], um espaço multi-dimensional de características (tais como sociabilidade ou extroversão) é visto como o fator determinante de como alguém irá agir na sociedade. Conseqüentemente, essa é uma possível abordagem para a modelagem de personalidades em personagens para jogos digitais.

Uma iniciativa nessa linha, e a modelagem de personalidade mais utilizada atualmente, é o modelo *Big Five* (também chamado OCEAN) [48]. Este modelo consiste basicamente em representar as personalidades como um ponto num espaço cinco-dimensional, onde cada eixo representa um traço de personalidade. Os cinco traços são: Abertura (*Openness*), Conscienciosidade (*Conscientiousness*), Extroversão (*Extraversion*), Amabilidade (*Agreeableness*), Neuroticismo (*Neuroticism*).

Tabela 4.1: Exemplos de personalidades associadas aos traços do OCEAN.

	Traços com alto grau	**Traços com baixo grau**
Abertura	Criativo, Curioso, Complexo	Convencional, Mente Estreita, Não Criativo
Conscienciosidade	Confiável, Bem Organizado, Disciplinado, Cuidadoso	Desorganizado, Descomprometido, Negligente
Extroversão	Sociável, Amigável, Falante, Divertido	Introvertido, Reservado, Inibido, Quieto
Amabilidade	De Boa Índole, Simpático, Benevolente, Cortês	Crítico, Rude, Duro, Calejado
Neuroticismo	Nervoso, Hipertenso, Inseguro, Preocupado	Calmo, Relaxado, Seguro, Robusto

Abertura significa uma pessoa criativa e mente-aberta. Essa abertura a experiências descreve a quantidade, profundidade, originalidade e complexidade da vida de um indivíduo.

Conscienciosidade significa que a pessoa é responsável, ordeira e confiável (*dependable*). O grau desse traço descreve controle social, que facilita o comportamento orientado a tarefas e objetivos, tais como pensar antes de agir, seguir regras, planejar e priorizar tarefas.

Extroversão significa uma pessoa falante, social e assertiva, implicando numa abordagem energética ao mundo e inclui sociabilidade, atividade e ser emocionalmente feliz.

Amabilidade significa uma pessoa de boa natureza, cooperativa e confiável (*trusting*) e inclui traços como altruísmo, confiança e modéstia.

Neuroticismo significa uma pessoa ansiosa, predisposta a depressão e preocupada, contrastando estabilidade emocional e boa índole com emoções negativas como ansiedade, nervosismo e tristeza.

Este é o modelo utilizado para influenciar o processo de decisão dos agentes no MIAGI. Na Tabela 4.1 podem ser vistos alguns exemplos de personalidade que podem ser usados como base para determinar quais os traços que representam um certo personagem.

4.3.3 Tomada de Decisão

Observando a hierarquia desenhada por Funge et al. em [77] (vide Figura 4.3), podemos ver que as duas camadas mais baixas foram tratadas cedo em computação gráfica usando modelos geométricos e cinemática inversa e tomadas como certas em jogos. As duas camadas seguintes (física e comportamental) também já se encontram bem tratadas em CG e jogos (com um pequeno auxílio de IA). Já a camada superior, ainda é um problema que pode ser considerado em aberto e é um dos motivos principais da utilização de IA em jogos.

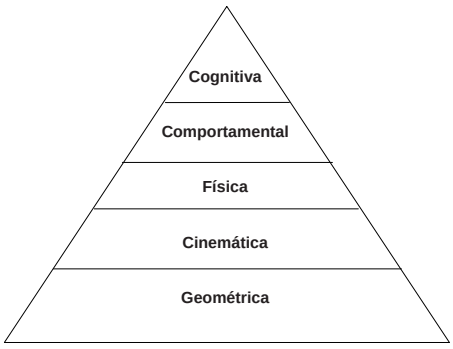

Figura 4.3: Hierarquia de modelagem de CG de Funge [77].

Deste modo o processo de decisão estabelecido no MIAGI tem que levar em consideração diversas das características que compõem um processo decisório real, tais quais memória, emoções, personalidade e motivações (mencionadas anteriormente) como sugerido em [106][78][63].

Nesse modelo de estado mental, cada comportamento é associado aos traços de personalidade que o afetam e as necessidades a que se referem. Estas necessidades são referentes à hieraquia de motivações de

Maslow [34] (Figura 4.4), uma representação de motivações amplamente utilizada desde gerência de projeto a psicologia. Além disso, a seleção de um determinado comportamento (ou sua implementação interna) pode fazer uso de informações presentes na memória no sentido de se explicitar se a experiência passada com uma certa entidade foi positiva ou negativa. Assim as camadas superiores exercem influência nas decisões do agente.

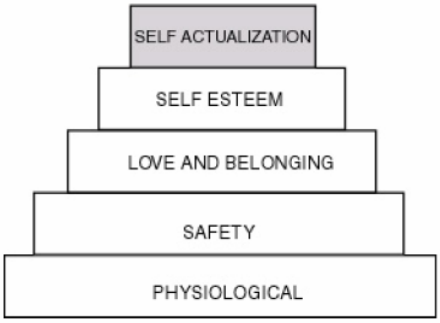

Figura 4.4: Pirâmide de necessidades de Maslow [34].

Essa abordagem permite que um mesmo comportamento resulte em personagens com atitudes diferentes, por exemplo, um ogro pode ser mais cauteloso enquanto outro é mais agressivo.

4.4 Camada Comportamento

A camada de comportamento é a camada responsável por englobar os diversos módulos que tratam das técnicas de implementação e controle dos comportamentos das entidades inteligentes mantidas pelo *middleware*.

A seguir, serão detalhados os requisitos das técnicas sugeridas para representar e implementar estes comportamentos.

4.4.1 Máquinas de Estados

Máquinas de estados finitas são a técnica de IA mais comumente utilizada em jogos por serem simples, intuitivas e de fácil implementação. No entanto, diversas questões devem ser tratadas ao se projetar a implementação dessa técnica de maneira flexível e a se integrar num *middleware* de IA.

Um dos principais pontos é escolher uma boa maneira de estrutrar a implementação dessa técnica de modo que fique flexível e de fácil

manutenção e adaptabilidade. Também é um requisito bastante importante tornar essa implementação o mais *data-driven* possível, de modo a deixar a definição dos componentes ser especificada de modo independente do código do *middleware* ou do jogo em si. Fazer uso de uma linguagem de *script* para modelar os comportamentos é um exemplo de abordagem mais *data-driven*.

Uma outra questão de *design* relacionada a estrutura e codificação das FSMs é como se dá a interação das mesmas com o mundo jogo, mas esse aspecto é tratado na arquitetura aqui proposta na camada de percepção/eventos, com a qual a camada de comportamento se comunica.

Fu e Houlette em [181] também levantam outras questões a serem levadas em consideração durante o projeto da implementação desta técnica, tais como: eficiência; balanceamento de carga; extensões ao modelo (como adição de pilha, polimorfismo, tratamento de mensagens) e suporte a depuração. A utilização dessas propostas aumenta ainda mais a utilidade dessa técnica e deve ser considerada desde o início do projeto.

4.4.2 Movimentação

Como discutido anteriormente, a maneira como uma entidade se move no mundo do jogo é crucial para uma boa impressão de sua inteligência. Deste modo, é necessário que este módulo implemente técnicas que representem essa movimentação de maneira realística e que se adaptem ao cenário de jogos digitais.

Alguma técnicas englobadas nesse módulo podem ser comportamentos de movimentação e *path-finding* de modo a, combinando-se, poderem tratar as fraquezas uma das outras.

4.4.3 Planejamento de Ações

Conforme discutido no Capítulo 2, o uso de técnicas baseadas em planejamento na modelagem do comportamento de agentes permite comportamentos menos repetitivos e previsíveis e maior variedade de comportamentos, além de permitir ao agente adaptar suas ações a condições específicas de uma determinada situação.

Outra vantagem da implementação de um sistema desse tipo é que este simplifica a representação do comportamento do agente e permite uma maior modularização do código.

Conseqüentemente, ao menos um sistema de tomada de decisão orientada a objetivos (sistema de "escolha" de objetivos simples baseado em planejamento, mas não formal) deveria ser implementado.

4.4.4 Lógica Fuzzy

Devido à versatilidade oferecida pela lógica *fuzzy*, esta é uma opção bastante interessante para aplicações que necessitam lidar com um certo grau de incerteza ou que necessitam de grande flexibilidade e capacidade de adaptação, o que é o caso de jogos digitais.

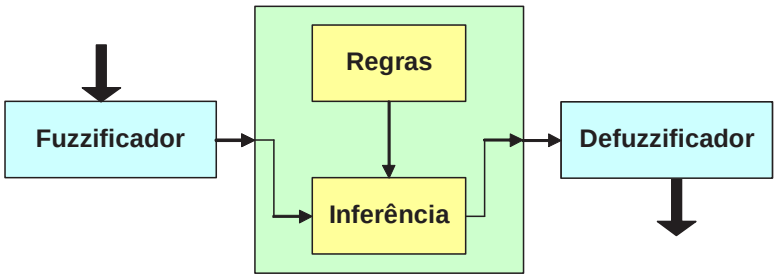

Figura 4.5: Diagrama dos módulos de um sistema de inferência *fuzzy*.

Optou-se então por implementar um sistema de inferência *fuzzy* (*Fuzzy Inference System* - FIS) pois este é capaz de realizar as mesmas funções de um RBS simples e apresenta a vantagem de possibilitar o uso de lógica *fuzzy* quando necessário. Além do fato desta abordagem ser bastante promissora em jogos, embora muito pouco utilizada atualmente.

Sistemas *fuzzy* baseados em regras (ou sistemas de inferência *fuzzy*) [52] são a forma mais usada de lógica *fuzzy* em geral. Um sistema de inferência *fuzzy* é geralmente composto de quatro componentes (como pode ser visto na Figura 4.5):

1. o fuzzificador converte as entradas *crisp* em conjuntos *fuzzy*;

2. a base de regras contém as relações entre as entradas e as saídas desejadas, expressadas como uma coleção de regras de implicação (`IF-THEN` *rules*);

3. o módulo de inferência que processa a base de regras, decide que regras são ativadas e combina seus resultados;

4. o defuzzificador que converte o resultado da inferência de volta a valores *crisp*.

As regras na base de regras são codificadas no formato `IF` antecedente `THEN` consequente. É feito então um `AND` das diferentes regras. Os antecedentes são formados por entradas associadas às variáveis do problema e termos que representam um conjunto fuzzy dentro de uma variável linguística *fuzzy*. Isto é, uma variável linguística *fuzzy* é a composição de vários *fuzzy sets* pra representar um conceito, e.g. Velocidade = Devagar, Média, Rápida. E um antecedente seria representado no formato "X É Devagar", sendo X o valor da velocidade num dado momento.

Além disso, podem ser aplicados a um antecedente funções modificadoras (*hedges*) de modo a modificar o grau de pertinência retornado por uma função de pertinência, e.g. "X É muito(Devagar)", onde muito() é um modificador. No caso de múltiplas variáveis de entrada, é ideal que todas as possibilidades de combinação entre todos os termos definidos em cada variável linguística estejam presentes na base de regras. Este requisito de completude leva inevitavelmente a uma explosão combinatória de regras (em um sistema com X entradas e Y termos cada, são necessários X^Y regras).

Como discutido anteriormente, essa explosão combinatorial de regras apresenta um grave problema para a aplicação de controle *fuzzy* no *design* de jogos digitais, pois complica o processo de projeto e consome recursos preciosos da CPU. Para aliviar esse problema, Combs propôs a chamada *Union Rule Configuration* (URC) [69][70]. Uma típica URC é expressada abaixo:

```
    SE (X1 = A11) ENTAO Y = B1
OR SE (X1 = A12) ENTAO Y = B2
OR SE (X2 = A21) ENTAO Y = B2     (1)
OR ...
OR SE (Xi = Aij) ENTAO Y = Bk
```

onde Xi são as variáveis de entrada, Y a variável de saída, e Aij e Bk os termos definidos sobre as variáveis *fuzzy*. Ao invés de apresentar a relação entre os antecedentes conectados logicamente e o consequente, a URC relaciona cada entrada diretamente à variável de saída, assim reduzindo o número total de regras necessárias a XY (i.e., ao invés de ter uma base de regras no formato SE condição 1 E condição 2 E ... ENTAO Y = conjunto *fuzzy* 3, todas conectadas por AND, usa-se lógica propositional para se manipular as regras gerando uma única regra (1), composta de uma série de mapeamentos Xi ENTAO Y = Bk juntos pela união - OR). Ou seja, tem-se um crescimento linear e não mais exponencial.

Embora os fundamentos teóricos da URC ainda estejam sob debate [79][80], há evidências da sua aplicabilidade em um número relativamente grande de situações. Podendo servir como uma alternativa prática e econômica para outras abordagens (tais quais *clustering* e sistemas hierárquicos) para resolver o problema da explosão do número de regras [155]. Também há evidências de que, embora possa alterar um pouco o resultado da inferência *fuzzy*, é bastante aplicável ao universo de jogos digitais [112][175]. Li, em [155], inclusive testa as regras tradicionais (27 regras) e as regras na configuração URC (9 regras) e conclui que ambos os conjuntos se comportam de maneira similar.

4.4.5 Scripting

De modo a permitir que pessoas de fora do time de desenvolvimento (como os *game designers*) possam mais facilmente criar e testar os comportamentos das entidades do jogo sem que seja necessário recompilar o mesmo, é necessário isolar essas definições do código do *middleware*.

A abordagem mais utilizada para tal é utilizar uma linguagem de *script*. Esta abordagem tem diversas vantagens como:

Data-driven design: *scripts* representam os comportamentos isoladamente da base de código do *middleware* e do jogo e podem então ser tratados como dados;

Desenvolvimento rápido: linguagem de *script* são geralmente mais simples, consequentemente causando menos erros na implementação, além de permitirem testar novos comportamentos ou correções sem recompilar o jogo ou necessariamente necessitar de alguém da equipe de programacão;

Extensibilidade: apenas alterando os *scripts* é possível estender os comportamentos e até mesmo implementar novas características não previstas anteriormente aos mesmos; e

Editores desacoplados: permitem a criação de editores de comportamentos que não necessitam de acoplamento forte ao *middleware*.

Uma linguagem de *script* tem que ser de fácil utilização pois *designers* geralmente não têm conhecimentos em programação. Uma possibilidade é criar uma linguagem específica para esse fim, como é o caso da UnrealScript

do jogo 'Unreal' da Epic Games. Outra abordagem é utilizar uma linguagem de script genérica já existente, tal como Python, Lua, Ruby ou Perl; ou ainda, usar uma linguagem de programação completa mesmo (como no caso do uso de Java no jogo 'Vampire: the Mascarade - Redemption' da Nihilistic).

A opção de desenvolver uma linguagem própria implica na necessidade de cuidados extras para evitar diversos problemas comuns [133], como por exemplo: adição de muitas *features* e perda do controle do crescimento e complexidade da linguagem; projetar uma linguagem ser uma tarefa não trivial; depurar é difícil (faltam de ferramentas de suporte); e problemas de desempenho.

Embora o uso de linguagens de *script* prontas também sofra de problemas como desempenho e falta de suporte a depuração, não criar uma linguagem própria e usar uma genérica descarta os vários problemas encontrados apenas quando se projeta uma linguagem nova.

Com base nesses fatores, decidiu-se utilizar um sistema de *scripting* em Lua que permite a criação de FSMs, regras *fuzzy* e objetivos. Além dessa abordagem permitir que este sistema seja utilizado também para tratar arquivos de configuração do sistema.

Um cuidado que deve ser tomado ao se fazer uso de um sistema de *script* é que a roteirização excessiva das ações dos agentes pode levar o jogo a ter comportamento muito amarrado e linear, o que na maioria das vezes compromete a jogabilidade. Uma possível solução é criar a IA de maneira autônoma e prover funcionalidades de tomar o controle desta quando necessário.

4.5 Comunicação/Eventos

Devido ao fato de arquiteturas de jogos geralmente serem baseadas em eventos e levando em consideração experiências passadas em desenvolvimento embarcado, decidiu-se implementar um sistema baseado em eventos para tratar dos problemas de comunicação entre os componentes da arquitetura se necessário.

Além disso, Schwab [151] sugere que as percepções dos agentes só necessitam ser atualizadas de tempos em tempos, não mudam tanto e são muito "caras" para serem calculadas a todo ciclo. Tal cenário se mostra adequado a um sistema baseado em eventos, o *input* diz o que mudou e o sistema de percepção nota a diferença. Decidiu-se então flexibilizar o sistema

de eventos de modo a tratar também da comunicação de dados sensoriais entre o motor do jogo e o *middleware*.

De modo a cobrir essas necessidades, cada evento possui os identificadores de qual entidade o enviou e para a qual é destinado, um tipo (associado ao sensor ou ao módulo da arquitetura), *time stamps* do momento de criação da mensagem e de quando esta deve ser entregue e um campo extra para *payload*. Asim, entidades interessadas em receber notificações sobre eventos de seu interesse devem registrar-se nesse componente de modo que este saiba como lhes passar os eventos corretos para manipulação.

Schwab sugere que deve-se utilizar um manipulador de percepções centralizado e que um sistema centralizado de percepções funciona bem num ambiente baseado em eventos [151], o que também se enquadra na proposta deste componente.

Essa estruturação permite, por exemplo, modelar tempo de reação dos agentes, onde um evento lido no componente cola é inserido no gerenciador de eventos com um tempo de entrega um pouco atrasado, o que pode ser interessante na implementação da jogabilidade. Pode-se também utilizar essa infra-estrutura de mensagens para gravá-las e usar essa gravação posteriormente para um *replay* de modo a examinar o comportamento do sistema em um dado momento (repetindo exatamente o mesmo fluxo de eventos). Outra vantagem desta modelagem é que as únicas interfaces do mesmo são as de mensagens entrando e mensagens saindo, o que facilita sua independência do resto do MIAGI e do motor do jogo.

4.6 Componente Cola

O chamado componente cola representa o código que deve ser implementado pelo desenvolvedor de jogos para poder fazer uso do *middleware*. Este componente consiste basicamente em implementar sensores e atuadores de modo a possibilitar a comunicação de dados entre a representação do mundo do jogo e a eventual representação do mesmo mundo no lado do *middleware*.

No caso da aquitetura aqui descrita, este módulo é responsável pela integração com o motor do jogo, sensores então lêem o que acontece no jogo e transformam essas informações em eventos a serem inseridos na camada de percepções e os atuadores pegam eventos na camada de percepção e executam a ação correspondente no mundo do jogo. Esse código cola deve ser implementado na linguagem utilizada no desenvolvimento do jogo que

fará uso do MIAGI (a grande maioria dos jogos digitais com esse perfil são escritos em C/C++) e deve também ser capaz de trocar dados com a linguagem utilizada na implementação do *middleware* (C++).

Um exemplo da utilização desse componente pode ser interpretar os eventos de visão do agente do lado do *middleware* sempre da mesma maneira, mas do lado do motor do jogo (no componente cola) podem ser usados diferentes mecanismos para detectar se há algum personagem visível como: diferentes cones de visão, grau de iluminação da região, presença de movimento e tempo de exposição (exemplos do jogo 'Thief: The Dark Project' [140]).

4.7 Conclusões e Observações

Seguindo o processo de evolução de *frameworks* proposto por Roberts e Johnson [57], procedeu-se à construção de um *framework* de caixa branca, em seguida partiu-se para a implementação de uma "biblioteca de componentes", implementando as técnicas mais úteis/comuns usadas em IA para jogos e então analisando como estas interagem entre si e com o *framework* prototipado. Durante a implementação da biblioteca de componentes, tentou-se identificar padrões de projeto (*design patterns*) [53] que pudessem ser aplicados no desenvolvimento de jogos (tanto na implementação das técnicas de IA quanto na estruturação do *middleware*) e assim ajudar a evitar erros/enganos comuns, contribuir para tentar formalizar a linguagem (através de um catálogo desses padrões de projeto) aproximando inteligência artificial e engenharia de software, e também prover um *design* de fácil construção e utilização que futuros desenvolvedores pudessem facilmente entender.

Na versão atual da arquitetura foram contempladas FSMs, comportamentos de movimentação, *path-finding* bem simples, um sistema de inferência *fuzzy*, um sistema de seleção de objetivos a partir de uma base de planos predefinidos, um sistema para tratar das percepções e que tem contato com todas as camadas e um simples sistema de arbitragem de objetivos. Tanto as FSMs quanto os objetivos podem ser usados de maneira hierárquica de modo a aumentar sua flexibilidade. Devido à natureza do *framework*, foi também implementado um sistema de *script* que faz uso da linguagem Lua [145] para permitir que FSMs, objetivos e regras *fuzzy* sejam especificadas fora do *middleware*. Decidiu-se não implementar árvores de decisão neste momento pois esta técnica não tem sido muito bem recebida na

comunidade de desenvolvimento de jogos digitais [170][152] (embora alguns jogos façam uso dela).

A implementação do protótipo original e a especificação da arquitetura atual (além da implementação descrita no Capítulo 5) permitiram a visão mais clara de várias questões de projeto na criação de tais ferramentas de maneira genérica para se adequar a diversos gêneros de jogos. No seu estado atual, a arquitetura proposta também serve como base para o desenvolvimento de comportamentos e sistemas cognitivos de mais alto nível. Tendo, inclusive, um artigo sobre a mesma sido publicado no *ACM SIGCHI International Conference on Advances in Computer Entertainment Technology 2005* [176].

O desenvolvimento deste tipo de software ainda enfrenta várias questões em aberto mas é essencial para propiciar uma melhor jogabilidade. De um ponto de vista de engenharia de software, se continuará tentando aumentar a flexibilidade do *middleware* através da identificação de novos *hot-spots* que possam ser generalizados e trabalhar para torná-lo mais próximo de um *framework* de caixa preta.