5
Implementação e Resultados

Este capítulo trata da implementação da arquitetura apresentada no capítulo anterior de modo a detalhar sua estruturação e validá-la. Aqui são comentados alguns detalhes de implementação dos seus componentes, a modelagem orientada a objetos utilizada (cujo diagrama de classes pode ser visto na Figura 5.1) e os padrões de projeto identificados como possíveis soluções aos problemas apresentados pelas necessidades dos diversos módulos e para ajudar a melhor descrever o *framework* [47].

5.1
Arquitetura Implementada

Esta seção apresenta a implementação de cada camada do MIAGI e seus diferentes módulos.

5.1.1
Camada Serviço

Como explicitado anteriormente, a camada de serviço trata de questões de mais baixo nível e das funcionalidades de infra-estrutura do *middleware*.

De modo a tratar da inicialização, acesso e finalização do *middleware* de IA foi necessário criar um gerenciador do *middleware* (chamado AIEngine) que é reponsável por gerenciar a execução do mesmo e deve ser inicializado uma única vez e possuir uma única instância durante toda a execução do jogo.

Esta instância deve estar acessível em qualquer parte do programa através de um ponto de acesso conhecido, isto pode ser resolvido com a utilização do padrão de projeto *Singleton*[53], que garante que uma determinada classe vai possuir uma única instância em toda a execução da aplicação. Além disso, o padrão provê um único ponto de acesso global

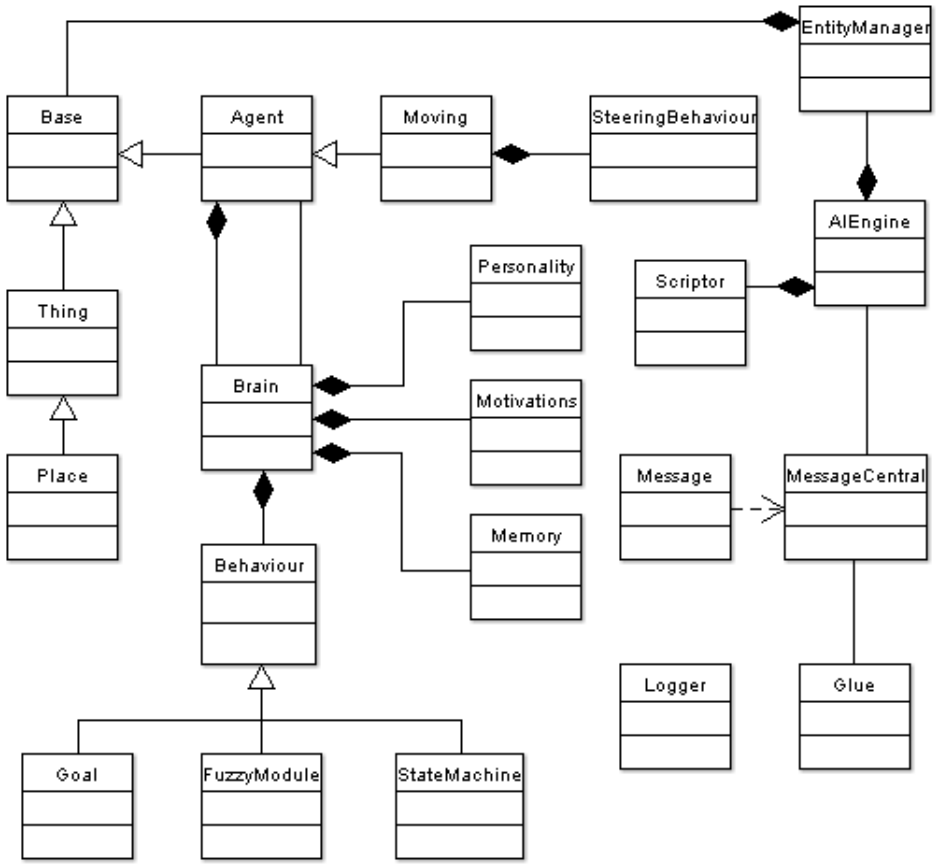

Figura 5.1: Diagrama de classes do MIAGI.

à instância de uma determinada classe, possibilitando assim que ela esteja acessível em qualquer ponto da aplicação.

De modo a implementar as funcionalidades de gerenciamento das entidades presentes no mundo de jogo e manipuladas pelo *middleware* (agentes - Agents, objetos - Things e lugares - Places), foi criado então um gerenciador de entidades (EntityManager) do qual também deve existir apenas uma instância (o que pode ser tratado com o padrão *Singleton*). Esse gerenciador (parte do AIEngine) é também responsável por funções de acesso às diversas entidades, o que pode ser modelado através do padrão *Iterator* [53] que provê acesso sequencial aos elementos abstraindo a sua representação interna.

Com relação à configuração, esta é tratada num componente em separado (chamado Scriptor) que pode ser acessado pelo AIEngine. Tal componente também existe como uma única instância e é mais um caso do padrão *Singleton*.

A seguir serão descritas as três principais preocupações tratadas na camada de serviços (relacionadas às questões de mais baixo nível da

implementação).

5.1.1.1 Balanceamento de carga

No primeiro protótipo decidiu-se usar *threads* para tratar de algumas questões do projeto, mas durante o uso do *framework* percebeu-se que uma abordagem por eventos é mais interessante no cenário de um *middleware* de IA para jogos.

Com relção ao uso de *threads*, Ousterhout [21] lista alguns dos motivos por que estas são problemáticas:

- apresentam problemas com sincronização e *deadlocks*, como dependência circular entre *locks* que é um problema difícil de localizar e tratar;

- são difíceis de depurar pois são extremamente sensíveis a dependências de temporização;

- alcançar um bom desempenho é difícil pois usar *locks* simples (como monitores, por exemplo) leva a concorrência baixa e *locks* "finos" aumentam a complexidade de gerenciá-los, o que também reduz o desempenho;

- existem poucas ferramentas para depuração, bibliotecas padrão na STL não são *thread-safe*, têm problemas de portabilidade entre plataformas, etc.

Como exemplificado, *threads* são difíceis de programar e conseqüentemente devem ser usadas apenas quando concorrência na CPU é realmente necessária. Uma vez que neste trabalho o desempenho do *middleware* não é o ponto principal, *threads* não são mais utilizadas em nenhum componente. Por outro lado, tratamento de eventos também tem suas vantagens como: possuir apenas um fluxo de execução; permitir que componentes do sistema se registrem para receber eventos e as dependências de tempo são somente entre os eventos e não dependentes do escalonamento do sistema. O componente que lida com eventos (MessageCentral) atualmente não se encontra arquiteturalmente na camada Serviço, mas sim num painel que atravessa todas as camadas do MIAGI.

5.1.1.2 Gerenciamento de Memória

A implementação de um sistema para gerenciar memória é algo muito complexo e qualquer que seja a implementação, esta não pode levar à perda de desempenho.

Se os conceitos de construtores, destrutores e escopo em C++ forem combinados, é possível criar classes que gerenciem seus próprios recursos, adquirindo-os nos construtores e liberando-os nos destrutores. Este "*idiom*" é chamado RAII (*resource acquisition is initialization*) [64] e assim os compiladores são capazes de automatizar o gerenciamento de memória. No entanto, essa é uma abordagem que depende do desenvolvedor e a maioria dos problemas com gerenciamento de memória ocorre, então, na não desalocação de objetos criados dinamicamente.

Uma possível solução para o problema é utilizar *Smart Pointers* [107], objetos que gerenciam o número de referências a um outro objeto. Quando este número chega a zero, uma ação é tomada, como por exemplo, desalocar o objeto da memória.

Esta é a abordagem utilizada (em conjunto com RAII) na versão atual da implementação. Desse modo, não há um módulo explicitamente responsável pelo gerenciamento de memória. No entanto, nem todos os casos de gerenciamento de memória se encontram bem tratados e necessitam ser revistos.

5.1.1.3 Depuração/Log

Com relação à sub-camada de Depuração, foi criado um módulo de *log* com a tarefa de auxiliar a depuração de código (classe Logger).

A principal função deste módulo gerenciador de *log* é facilitar o monitoramento da aplicação através de mensagens inseridas no código fonte. Estas mensagens devem poder ser enviadas para diferentes dispositivos, como por exemplo, arquivos locais, console gráfico ou porta serial, para que possam ser posteriormente analisadas.

Tentando criar um maior grau de classificação das mensagens, também foi adaptada a idéia de criar diferentes tipos de mensagens. No MIAGI essas mensagens podem ser classificadas como: *Information*; *Warning*; *Error*; e *Fatal*.

No módulo de *log*, cada dispositivo deve informar ao gerenciador que tipo de mensagem deseja receber. Esse registro de interesses ocorre na

inicialização do sistema e é realizado quando o objeto Scriptor lê o arquivo de configuração do *middleware*, onde estão listados os dispositivos de *log* existentes junto com os tipos de mensagens que devem ser enviados a cada um deles.

O gerenciador de *log* ao receber uma mensagem deve verificar os dispositivos cadastrados e repassar as mensagens aos que desejam recebê-la; para tratar esse modelo o padrão de projeto *Observer* [53] pode ser utilizado. Este padrão é usado para definir dependências um-para-N de modo que quando um objeto sofrer modificação (no caso, a mensagem) os demais (os dispositivos) serão notificados.

Após alguns estudos, optou-se por uma implementação como a do módulo de *log* em [135], onde um controlador de *log* recebe todas as mensagens e as roteia para um conjunto de consumidores. Cada dispositivo de *log* (seja arquivo, console, porta serial, etc.) informa se deseja receber algum tipo de mensagem, herda da classe que representa os consumidores e se registra no gerenciador.

5.1.2 Camada Cognição

Com relação aos módulos de Emoção/Personalidade e de Aprendizado/Memória, emoções e aprendizado não estão sendo tratados na versão atual do MIAGI.

Personalidade, por sua vez, é implementada de maneira bem simples como valores para cada um dos cinco traços de personalidade do modelo *Big Five*; cada eixo recebendo um valor variando de -1 a 1, de acordo com o explicado no Capítulo 4. Cada um desses traços é utilizado no processo de decisão para influenciar os comportamentos que se declaram influenciáveis pelos mesmos. Os respectivos valores e seus métodos de acesso são definidos no objeto Personality que pode ou não existir para um determinado agente.

O tratamento de memória utilizado na implementação desta camada é também bem simples e consiste em apenas guardar os "eventos" ocorridos com o agente em um repositório de memórias. Esse repositório é implementado pelo objeto Memory de cada agente, que também faz uso do padrão de projeto *Iterator* [53]. Os eventos na memória de um agente consistem de um objeto representando cada agente ou item de jogo com que ele teve contato e algumas informações associadas a ele.

Estas informações podem ser utilizadas para influenciar o processo de decisão e são: seu posicionamento, o *timestamp* da última vez que a

entidade foi vista e se o último encontro foi "positivo" ou "negativo". Essa representação não tem tempo de expiração, permanecendo na memória do agente enquanto este existir.

O módulo de decisão será explicado em mais detalhes a seguir.

5.1.2.1 Tomada de Decisão

O módulo de decisão implementado no MIAGI é representado pela classe Brain e faz uso de uma pilha de comportamentos (instâncias das subclasses da classe Behaviour) para guardar os comportamentos a serem executados e sua ordem de execução. Quando um novo estado ou objetivo (ambos representando ações dentro de um comportamento) é acionado, ele é colocado no topo da pilha e quando sua execução acaba, ele é retirado e a execução volta à ação anterior.

Além disso, um comportamento "especial" sempre se encontra no fundo da pilha caso não haja outro em execução. Este comportamento é chamado Think e ativa o processo de decisão, sendo o responsável por arbitrar entre os comportamentos disponíveis para execução em um dado momento (esses comportamentos são implementados pelo desenvolvedor de acordo com as necessidades de cada jogo e seguindo os tipos das subclasses da classe Behaviour providas pelo *middleware*).

Dentre os comportamentos disponíveis, é escolhido o que possui o maior grau de desejo. Esse grau de desejo é calculado por cada comportamento e faz parte da implementação dos mesmos e diz quão importante é a sua execução (através do método desirability, do objeto Behaviour).

Como cada pessoa pesa os fatores numa situação diferentemente (pessoas percebem e raciocinam sobre o mundo de maneiras diferentes), então é natural tratar aqui outros fatores que influenciem o processo de decisão. Alguns desses fatores são a personalidade do indivíduo e suas emoções (estas sendo previstas mas não implementadas na versão atual).

Cada comportamento então é classificado de acordo com as influências que os diferentes traços de personalidade exercem sobre ele (basicamente, se positiva ou negativa e se grande ou pequena), sendo essa classificação feita na implementação do comportamento.

Além disso, também são levadas em consideração nesse processo decisório, a influência das necessidades do agente (inspiradas na hierarquia de Maslow) seguindo um abordagem parecida. Cada comportamento é

classificado como relacionado a um dos diferentes níveis da pirâmide e essa informação é aplicada como fator para aumentar o grau de desejo daquele comportamento proporcionalmente ao nível (uma motivação física é mais forte que uma motivação de segurança e assim por diante).

Dependendo da frequência necessária ao processamento da IA de um dado agente, o método `update` da classe Brain é executado a cada intervalo de tempo. Ao ser invocado, o método `update` irá varer os comportamentos disponíveis coletando seus graus de desejo e aplicará a esses graus os modificadores decorrentes das personalidades e das motivações (se essas existirem para um dado agente). Após o cálculo desses valores, o comportamento com maior grau de desejo será colocado no topo da pilha, como descrito anteriormente.

É importante lembrar que os *updates* de movimentação devem ocorrer o mais frequentemente possível de modo a apresentar um movimento realístico e suave, enquanto o processamento da IA pode ocorrer de modo bem mais discretizado. Sendo assim, as atualizações de movimentação e da tomada de decisão devem ser desacopladas, fazendo com que a carga de processamento seja aliviada e permitindo que sejam implementados níveis de LOD na IA.

5.1.3 Camada Comportamento

Nesta seção são descritas as características da implementação das diferentes técnicas na atual versão do *middleware*.

5.1.3.1 Máquinas de Estados

A implementação de FSMs no MIAGI é um meio termo entre uma máquina de Mealy (onde as ações são realizadas nas transições) e uma máquina de Moore (com as ações nos estados). Isso se dá de modo a permitir uma maior flexibilidade da implementação permitindo ao desenvolvedor escolher como utilizá-las.

O comportamento da máquina de estados é estruturado seguindo o padrão de projeto *State* [53](que permite a um objeto mudar seu comportamento segundo um "estado interno") e foge um pouco do modelo matemático usual para máquinas de estados. Uma FSM pertencente a um agente aponta para um objeto representando um dado estado da

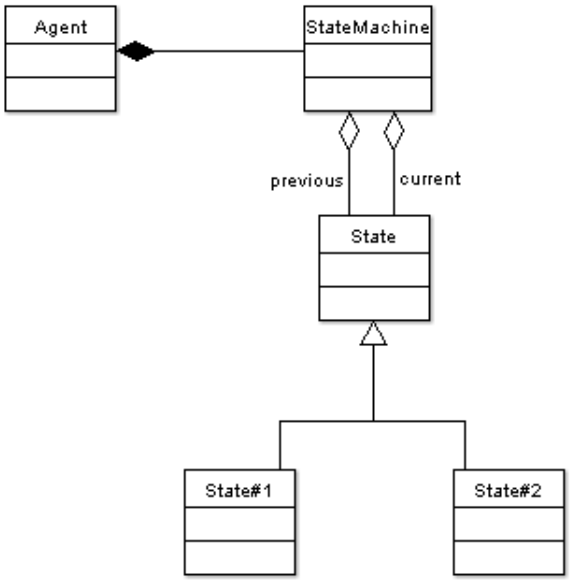

Figura 5.2: Diagrama de classes da implementação das FSMs.

máquina (e que contém as regras de transição deste), que pode ser trocado durante a execução por outros objetos estado. Usando os estados desse jeito, pode-se facilmente trocar a arquitetura da máquina de estados por outra, possibilitando a organização do comportamento em máquinas menores.

Na implementação atual (Figura 5.2), uma FSM possui um estado atual e uma referência para o estado anterior, além de poder possuir um estado global sempre em execução (conforme sugestão de Buckland em [175]). O estado anterior serve para simplificar a máquina evitando que muito mais transições tenham que ser codificadas; ao terminar a execução de um estado, caso não haja transições, a máquina volta então ao estado anterior (servindo como um autômato com pilha).

Cada estado implementa ao menos três métodos: enter(), responsável por eventuais ações na transição de outro estado para o atual e que pode usar o ponteiro para o estado anterior nesse controle; execute(), responsável pelas ações que ocorrem enquanto o estado está em voga; e exit(), responsável por eventuais ações na transição de saída.

5.1.3.2 Movimentação

Essa sub-camada se divide em duas partes, comportamentos de movimentação e procura de caminhos. Estas abordagens são independentes uma da outra, mas podem ser usadas em conjunto de modo a obter melhores resultados.

Com relação à implementação das técnicas de *steering behaviours* na sub-camada de movimentação, uma opção seria o uso da biblioteca OpenSteer [22] disponível no Sourceforge sob a licença MIT. Esta biblioteca é escrita em C++ de modo a auxiliar a construção de *steering behaviours* e usa um *framework* orientado a *plug-ins*. No entanto, de modo a facilitar a integração ao *middleware*, optou-se por utilizar uma abordagem mais simples desenvolvida anteriormente [131] e modificar esta usando algumas sugestões de Buckland em [175].

O modelo de comportamento de movimentação implementado tem como principais características um vetor velocidade e um vetor de direção normalizado; e o diagrama de classes da implementação atual pode ser visto na Figura 5.3. No método update() do agente, é chamado o cálculo da força de *steering* a cada iteração do laço de execução de raciocínio.

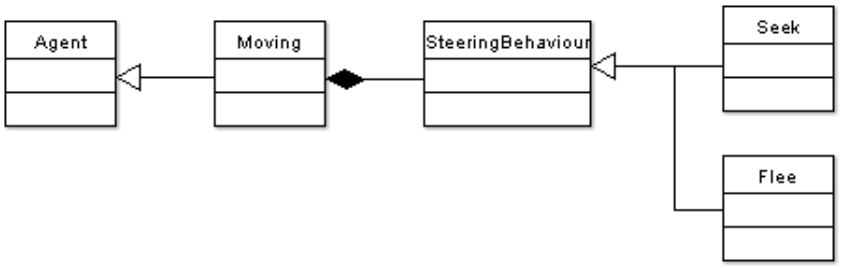

Figura 5.3: Diagrama de classes dos *Steering Behaviours*.

Foram implementados os comportamentos *Seek, Flee, Arrive, Pursuit, Evade* e *Obstacle Avoidance*, além dos comportamentos de grupo *Separation, Alignment* e *Cohesion*. Todos estes seguindo as propostas de Reynolds [187][188], além da combinação das diferentes forças de direcionamento ser calculada através dos métodos de somas ponderadas descritos no Capítulo 2.

Já com relação a *path-finding*, a abordagem utilizada é bastante similar à proposta em [156].

O algoritmo utilizado usa caminhos em linha reta que são recalculados quando essa linha intercepta algum dos objetos não inteligentes do mundo (acessados via gerenciador de entidades). O novo caminho formado por novos segmentos de reta forma um caminho até a posição alvo que não colide com nenhum dos objetos não móveis do mundo modelado. Uma vantagem desse processo é que ele não necessita de um algoritmo como o A* e de uma representação do mundo em forma de um grafo, mas é suficiente para a necessidade atual do *middleware* de implementar um módulo de descoberta

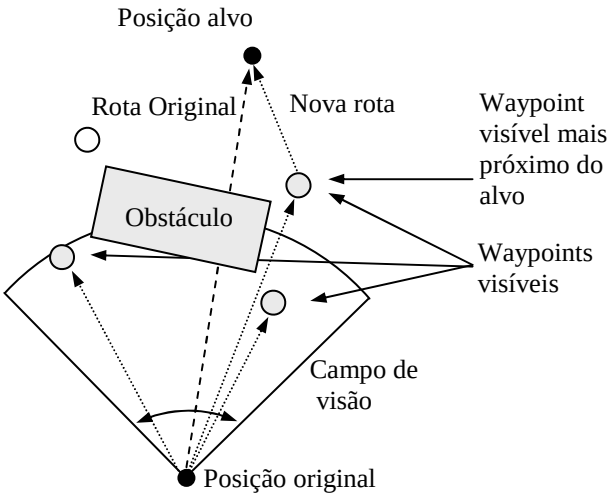

Figura 5.4: Exemplo da execução do cálculo de caminhos. Reproduzida de Pozzer [173].

de caminhos para tratar ações não atômicas como andar até uma certa posição.

A idéia básica do algoritmo é quebrar o caminho em vários segmentos retos. Deste modo o agente sempre está tentado seguir o próximo ponto de interseção entre os segmentos, mantendo a posição final a ser alcançada. Como levantado por Pozzer em [156], pode-se considerar esta abordagem como uma estratégia baseada em raciocínio sobre o terreno usando *waypoints*, pois os pontos escolhidos para desviar dos objetos (os *waypoints* usados) nada mais são que os cantos do *bounding box* do objeto deslocados pelo raio da área ocupada pelo agente.

Em mais detalhes, para se evitar a colisão com os objetos do mundo, quando um objeto entra no campo de visão do agente e o caminho atual intersecta o *bounding box* desse objeto, procura-se qual o *waypoint* (calculado como descrito acima) mais próximo dentre os visíveis ao agente. Essa seleção usa um simples cálculo de distância e vai adicionando segmentos ao redor do objeto com o qual o agente colidiria. Como visto na Figura 5.4, esse novo *waypoint* normalmente representa o menor caminho até a posição alvo e se mais de um objeto colidir com o trajeto do agente, apenas um é considerado a cada momento.

Como no momento as maiores preocupações do *framework* são estruturais e de relacionamento entre os diversos componentes, não preocupou-se com movimentos robóticos e não fluidos dando preferência a um processo de *path-finding* que evitasse colisões e fosse simples e rápido; ao qual essa abordagem essencialmente reativa se adequa perfeitamente.

5.1.3.3 Planejamento de Ações

A versão atual do MIAGI faz uso de uma implementação simples baseada em planejamento na forma de tomada de decisão orientada a objetivos (*goal-directed decision making*), i.e., por meio da seleção de ações a partir de uma base de planos pré-definidos criados pelo desenvolvedor de cada jogo.

Humanos selecionam objetivos de alto nível baseados em suas necessidades e desejos; um agente no jogo então será capaz de simular esse processo e, após selecionado um objetivo, vai tentar segui-lo até o fim decompondo-o em sub-objetivos, satisfazendo esses um de cada vez e continuando até o objetivo de alto nível ter sido saftisfeito ou falhar (alguma mudança no mundo impedir de ele seja alcançado).

Para implementar esse comportamento, em cada ciclo de atualização/execução do comportamento, se examina o estado do jogo e um grupo de planos ou estratégias que satisfaz o "desejo" mais forte é selecionado (com maior *desirability*).

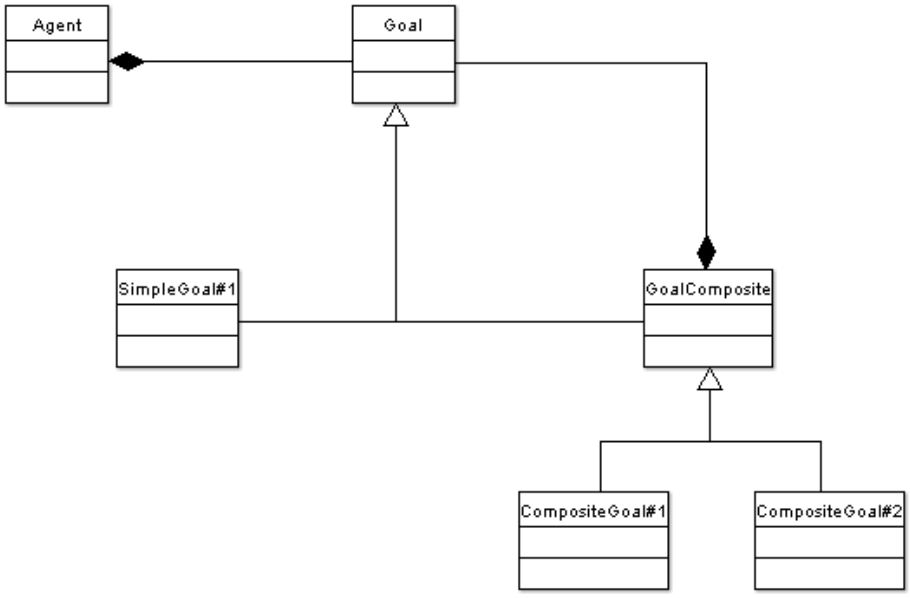

Figura 5.5: Diagrama de classes dos objetivos.

Os diversos objetivos são organizados de forma hierárquica podendo ser objetivos atômicos ou objetivos compostos, sendo os atômicos os objetivos nos nós da hierarquia. Esse tipo de comportamento se enquadra no padrão de projeto *Composite* [53], que permite lidar de maneira uniforme com objetos pertencentes a hierarquias parte-todo, sejam eles objetos individuais ou composições de objetos (Figura 5.5).

Os objetivos de mais alto nível podem ser vistos como objetivos em um nível de estratégia que vão sendo decompostos até que se chegue nas ações básicas. Um objetivo de alto nível sempre presente, e executado quando nenhum outro é possível, é o objetivo de explorar o mundo.

De maneira a ilustrar a utilização, pode-se imaginar o algoritmo de *path finding* implementado como um plano onde o objetivo de alto nível do agente é mover-se até uma determinada posição final, os sub-objetivos são os *waypoints* intermediários do caminho e as ações são a real movimentação até estes pontos.

Cada sub-objetivo necessário na decomposição do objetivo pai é então inserido no topo da pilha de comportamentos (descrita na camada de decisão).

Um objetivo implementa os seguintes métodos: activate(), utilizado para checar se sua execução é possível; process(), que tenta executá-lo e retorna se este se encontra inativo, ativo, completo ou falhou; e terminate(). O método activate() pode ser invocado mais de uma vez de modo a executar um "replanejamento" daquele objetivo, um *reset*. Cada objetivo composto chama o método process() de todos os seus sub-objetivos, assim removendo todos os que estiverem completos ou que tenham falhado da pilha.

A abordagem aqui descrita permite ao agente interagir melhor com o ambiente (por exemplo, no caso do objetivo "abrir porta do labortório", o agente ao chegar na porta insere na pilha um comportamento "passar crachá" e, após a execução deste, volta ao comportamento "abrir porta" que agora é passível de execução). Outra vantagem é que essa abordagem permite criar uma pilha de comandos para os agentes, o que pode ser usado para definir um roteiro de ações que um agente deve executar em seqüência e pode ser útil na implementação de determinadas partes de um jogo.

5.1.3.4 Lógica Fuzzy

Uma grande motivação para a implementação de um módulo de inferência *fuzzy* a ser usado em jogos digitais foi o fato da abordagem *fuzzy* parecer bastante promissora para esse tipo de aplicação; tanto pela relativa facilidade de projetar o sistema que controla a IA do agente, quanto pelo fato de que regras e variáveis (graças às suas definições lingüísticas) são facilmente interpretáveis e muito intuitivas em sua criação.

Um outro fator de interesse é que esta não é uma técnica presente mesmo no estado-da-arte dos *middlewares* de IA comerciais [149], mas que

se enquadra perfeitamente como parte de um sistema desse tipo.

Zarozinski em [111] apresenta a *Free Fuzzy Logic Library*, uma biblioteca relativamente completa para inferência *fuzzy* em C e disponível no Sourceforge [23] sob uma licença BSD; no entanto foi tomada a decisão de não se utilizar essa biblioteca. Dentre os motivos que levaram a essa decisão se encontram sua relativa complexidade (de código, uso e especificação de regras) e, sobretudo, a necessidade de integrar o módulo *fuzzy* ao *middleware* em desenvolvimento. Optou-se então por implementar um módulo mais simples para se conhecer melhor o problema e facilitar essa integração (a Figura 5.6 mostra o diagrama de classes simplificado em UML da implementação).

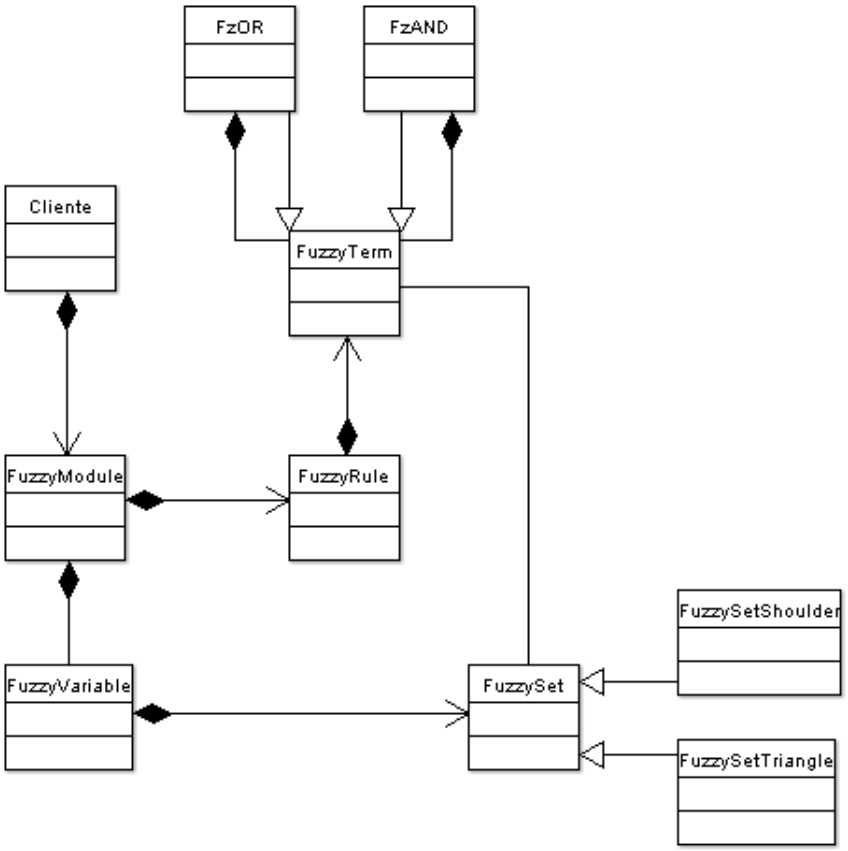

Figura 5.6: Diagrama de classes do sistema de inferência *fuzzy*.

De modo a utilizar esse módulo no ambiente de jogos digitais interativos e devido a restições de tempo na implementação desse protótipo, diversas decisões de simplificação foram tomadas. Uma destas restrições é que na implementação atual os conjuntos *fuzzy* apenas podem ser definidos através de funções "ombro" (Figura 5.7(a)), "triângulo" (Figura 5.7(b)) e "trapézio" (Figura 5.7(c)). Uma outra restrição é que as regras na base de regras não podem conter o conectivo lógico `NOT`.

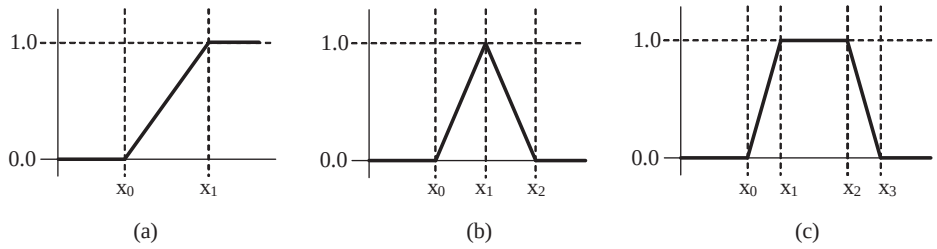

Figura 5.7: Funções de pertinência implementadas.

O componente *fuzzy* implementado faz uso do padrão de projeto *Composite* [53] para representar as regras utilizando uma hierarquia parte-todo (ou seja, uma regra pode ser composta de sub-regras ou termos). Além disso, usa como operadores na agregação, a função min para o conectivo `AND` e a função max para o conectivo `OR` e usa a função max para a composição dos resultados da inferência. Foi ainda escolhida a implicação de engenharia (min) [52] para implementar a implicação. Estas decisões se deram tanto por questões de simplicidade de implemtação quanto eficiência.

O sistema implementado provê ainda dois modificadores, "muito(x)" e "pouco(x)" implementados como x^2 e $\sqrt{x}$ respectivamente.

No componente de defuzzificação, a implementação atual suporta o método MoM (*Mean of Maximum*), que usa somente o termo com maior grau de validade e o método CoM (*Center of Maximum*), que usa mais de um termo da saída no cálculo do resultado. A implementação do método de centro de massa foi postergada devido a sua maior complexidade.

Um outro ponto de simplificação da atual implementação está na codificação das regras que atualmente tem que ser feita em código dentro da classe que possui a referência ao módulo *fuzzy*.

Com base na experiência da modelagem e implementação do módulo de inferência *fuzzy* [172], pode-se concluir que lógica *fuzzy* é uma técnica de inteligência artificial que pode ser claramente usada para melhorar o desempenho do "raciocínio" em diversos cenários em jogos digitais, pois consegue lidar com complexos problemas de controle a custos computacionais baixos e representar intuitivamente comportamentos de agentes de software, ambos sem sacrificar detalhes sutis dos mesmos.

Como trabalho para um futuro imediato, a primeira iniciativa será implementar um módulo para que as regras e as diversas funções de pertinência possam ser definidas usando a linguagem Lua em arquivos em separado. De modo a permitir um *design* mais orientado a dados (*data-driven*) e deste modo permitir que as regras e conjunto sejam definidos sem

que se tenha que modificar o código do módulo *fuzzy* ou se implementar uma nova classe que herde de uma classe pertencente ao *middleware*.

Além disso deve ser implementado um editor gráfico para as regras e funcões de pertinência de modo a evitar erros e a facilitar a criação de novos comportamentos; tal editor também suportará o uso do método de Combs para diminuir a base de regras caso seja do interesse do usuário criador das regras.

5.1.3.5 Scripting

A implementação das funcionalidades de *scripting* poderia fazer uso do padrão de projeto *Interpreter* [53], como, por exemplo, no uso de uma gramática simples para ler palavras chave e parâmetros de um arquivo de configuração. No entanto, como a arquitetura do MIAGI prevê um esquema de *scripting* relativamente complexo, decidiu-se partir para o uso de uma linguagem já existente de modo a evitar os diversos problemas que podem surgir durante o desenvolvimento de uma nova linguagem para programação.

A linguagem escolhida na implementação foi Lua [145]. Tal escolha se deve ao fato desta ser uma linguagem amplamente utilizada para *scripting* em jogos (por exemplo, nos jogos 'Baldur's Gate', 'Grim Fandango' e 'Far Cry') e pelo fato de Lua apresentar diversas características interessantes em um cenário como jogos. As principais vantagens consideradas de Lua são: a implementação de seu interpretador é rápida e ocupa pouco espaço em memória; programas em Lua podem ser lidos em forma de *script* ou como *bytecode* compilado; e Lua provê um mecanismo de *garbage collection*.

De modo a permitir a ligação entre o código C++ e código desenvolvido em Lua, é utilizada a biblioteca Luabind. Esta é uma biblioteca disponível no Sourceforge sob a licença MIT [24] e tem a habilidade de expor as funções e classes escritas em C++ para o ambiente Lua.

Luabind é implementada utilizando técnicas de meta-programação por *templates* [107] e, por sua vez, necessita que a biblioteca Boost (um conjunto de bibliotecas C++ portáveis que faz uso da mesma técnica) [25] esteja instalada no sistema. Com esse ambiente configurado, Luabind gera código para fazer a ligação dependendo apenas de informações disponíveis em tempo de compilação e esta ligação é feita de modo a abstrair o desenvolvedor do mecanismo de pilha utilizada para comunicação entre as duas linguagens.

O procedimento para expor uma classe C++ é bastante simples, não sendo necessário especificar os tipos de retorno nem os parâmetros das funções e métodos. Um exemplo de uso é mostrado a seguir para a classe Skeleton.

Supondo o código C++ da classe mostrado na Figura 5.8, utilizando Luabind o registro dessa classe tem a forma apresentada na Figura 5.9.

```
class Skeleton {

private:
   int id;
   int health;

public:
   Skeleton(int i, int h): id(i), health(h) {}
  ~Skeleton();

   void decHealth(int dec) {
      this->health-=dec;
   }
}
```

Figura 5.8: Exemplo de código C++ da classe *Skeleton*.

```
module(pLua) [
   class_<Skeleton>("Skeleton")
    .def(constructor<int, int>())
    .def("decHealth", &Skeleton::decHealth)
];
```

Figura 5.9: Exemplo de *binding* da classe *Skeleton* para Lua.

Uma observação importante a ser feita é que o tempo de compilação para o arquivo que faz o registro do mapeamento aumenta consideravelmente, então é recomendado que todos os registros sejam realizados no mesmo arquivo de código. No caso das FSMs, por exemplo, só é necessário adicionar os registros dos métodos changeState, setCurrentState e getCurrentState ao arquivo C++ de *bind*. O que torna possível a criação das máquinas de estado em Lua sem precisar modificar mais nenhum arquivo de código fonte em C++. Qualquer outro *bind* entre C++ e Lua deveria ser definido no mesmo arquivo usado para o *bind* da FSM adicionando-se mais uma cláusula module(pLua) como a da Figura 5.9.

A classe chamada Scriptor (vide Figura 5.1), descrita anteriormente, é responsável por lidar com todas as questões relacionadas a *scripting*. Ela é responsável pela leitura da configuração do sistema a partir de um

arquivo Lua e também por expor as interfaces das máquinas de estados e do mecanismo de objetivos presente no MIAGI. A exposição dessas interfaces permite que esses comportamentos possam ser implementados totalmente em Lua, alcançando assim o requisito de propiciar uma abordagem de desenvolvimento mais *data-driven*.

5.1.4 Comunicação/Eventos

Como este módulo não deve depender nem da implementação das técnicas no *framework* nem do motor do jogo e o conjunto de funcionalidades que oferece é bem definido (envio e recebimento de mensagens), pode-se aplicar o padrão de projeto *Façade* [53] que provê uma interface alto nível e unificada para um sub-sistema, tornando-o de mais fácil utilização.

Outro aspecto importante da implementação desse painel de comunicação é que ele é responsável por redirecionar todos os tipos de eventos do sistema, sejam vindos do módulo cola, dos demais módulos do sistema e seus respectivos manipuladores de eventos, usados como mecanismo de gerenciamento de carga ou comunicação entre agentes. De modo a resolver o problema desse tratamento pode-se utilizar o padrão de projeto *Chain of Responsability* [53]. Este padrão evita o acoplamento do módulo de comunicação aos demais módulos e permite que mais de um objeto tenha a chance de tratar um determinado evento.

Outra caracterítica desse módulo, quando da sua utilização como camada de percepção, é que deve notificar o resto do sistema de modificações ocorridas - o que pode ser resolvido com o padrão de projeto *Observer* [53]. Os módulos que desejem ser notificados de mudanças no estado devem então se cadastrar na central de mensagens através dos métodos registerForMessage() e unregister(), ambos para o tipo de mensagem desejada.

Durante cada laço de execução, o MessageCentral checa os tempos das mensagens enfileiradas e decrementa seus contadores, entrega as mensagens da vez ou destrói as mensagens velhas e não lidas.

5.2 Ambiente de Testes

O cenário utilizado para testes durante o desenvolvimento do MIAGI representa um ambiente semelhante ao apresentado por um jogo e tem como

objetivo servir para a simulação da interação entre os personagens, objetos e lugares sem que seja necessário se preocupar com todos os detalhes de implementação ou de comunicação com um jogo complexo inteiro.

Esse cenário de testes é uma "caixa de areia" onde, basicamente, o universo do jogo é um ambiente aberto que usa uma representação de terreno com relevo [186] e uma biblioteca para modelagem e efeitos do céu [178], de modo a torná-lo mais realístico.

Nesse ambiente, co-existem os personagens (que possuem ações comuns em jogos como andar, atacar e pular), objetos e lugares (estes com nome e posição únicos usados na locomoção e raciocínio dos agentes), além de estarem implementados aspectos físicos, como distâncias, velocidade de deslocamento e obstáculos. Deste modo é possível representar e testar os personagens e seus comportamentos.

A representação gráfica do cenário de teste usa um grafo de cena para representar o ambiente. Este grafo suporta a representação de fontes de luz, câmeras, texturas, a representação do terreno, a representação do céu e representações de modelos 3D em três formatos: modelos MD2 [183], formato de animação de personagens utilizado no jogo 'Quake 2' da id software, modelos 3DS gerados pelo 3D Studio Max da Autodesk (uma das ferramentas mais populares para modelagem 3D) e modelos POZ (formato proposto por Pozzer em [173]). Tal cenário de testes é implementado em C++ e faz uso de OpenGL [146] para manipulação gráfica de personagens e objetos.

5.3 Cenário Geral de Uso do MIAGI

O desenvolvedor de um jogo que deseje fazer uso do MIAGI deve escrever inicialmente o código cola (dependente do jogo em si, geralmente escrito em C/C++) de modo a implementar métodos sensores e atuadores no mundo do jogo.

Os sensores serão responsáveis por detectar eventos no jogo que sejam relevantes para o comportamento inteligente desejado para o agente, por exemplo, "inimigo entrou no campo de visão", "fui atingido", "estou com pouca energia", etc. e por enviar esses eventos para o MIAGI. Os atuadores por sua vez são responsáveis por ler do *middleware* possíveis ações a serem tomadas pelo agente no jogo e converter esses eventos nas chamadas de métodos correspondentes no mundo do jogo.

Após a criação do código cola, é necessário inicializar o MIAGI através da instanciação da classe AIEngine, responsável por representar o *middleware* e disponibilizar suas funções de acesso.

Ao instanciar a AIEngine, esta inicializará seus componentes principais: EntityManager, MessageCentral e Scriptor. O Scriptor é responsável então pela leitura do arquivo de configuração do sistema e registra os dispositivos de *log* disponíveis.

Após o processo de inicialização é necessário cadastrar as entidades do jogo que devem ser consideradas durante o processamento dos comportamentos dos agentes pelo *middleware*. Essas entidades podem ser de quatro tipos: Agent, Moving, Thing e Place. Agent é um agente, ou seja, uma entidade que "pensa"; Moving é uma classe derivada de Agent e representa um agente que se move; Thing representa um objeto a ser levado em consideração pelo middleware; e Place, por sua vez, representa um lugar no mundo do jogo. Todos estes quatro tipos estendem a classe abstrata Base (vide Figura 5.1), que representa uma entidade básica do MIAGI.

Esse cadastro acontece enviando-se para o AIEngine (através da invocação do método addEntity) um identificador da entidade, seu tipo (um dos quatro listados) e seus comportamentos (se a entidade for um agente). Esses comportamentos podem ser escritos em classes C++ ou em Lua.

Para cada agente cadastrado no EntityManager será criado um objeto do tipo Brain (responsável pelo processo de decisão do agente) e a ele serão atribuídos os comportamentos.

Estes comportamentos devem implementar um dos esquemas de modelagem de comportamento providos pelo MIAGI que são: listas de objetivos (Goal), base de regras de inferência *fuzzy* (FuzzyModule) ou máquinas de estados (StateMachine). Cada um desses modelos implementa a interface definida na classe abstrata Behaviour e os comportamentos dos agentes devem então herdar destas classes.

Além da definição dos comportamentos de um agente, é possível definir se este agente tem personalidade e motivações. Para definir a personalidade de um agente é necessário que sejam definidos os valores para cada um dos cinco traços de personalidade apresentados anteriormente. Estes valores devem então ser informados ao MIAGI via uma chamada ao método setPersonality do AIEngine, que recebe o identificador da entidade e os parâmetros da personalidade. Para definir as motivações (i.e. os graus de influência dos níveis da hierarquia de Maslow - Figura 4.4) segue-se um procedimento semelhante, mas chamando o método setMotivations.

Após a inicialização do *middleware* e os registros das entidades e de

suas características, basta, a cada laço de execução do jogo, enviar ao MessageCentral os eventos relevantes ocorridos no mundo do jogo (feito pelos métodos sensores do código cola) e ler os eventos de resposta do MIAGI para enviá-los ao jogo (feito pelos métodos atuadores do código cola). O MessageCentral se encarrega de encaminhar os eventos às respectivas entidades e de encaminhar as ações de volta ao código cola.

5.4 Estudo de Caso como Framework

De modo a realizar uma prova de conceito da utilização do *middleware* com conhecimento da hierarquia de classes do *framework*, foi utilizada uma abordagem caixa branca em vários cenários usando o ambiente de testes descrito. Dentre estes cenários de uso podemos destacar, a título de exemplificação, os seguintes:

1) Uma situação comum em jogos tipo FPS é a de dois personagens se procurando e entrando em combate quando se encontram. Tal situação foi implementada utilizando-se FSMs e o *path finding* simples implementado.

2) Situação onde um personagem se movimenta até um local no mapa que representa um mina (onde ele "trabalha") e depois volta pra casa (usando *path-finding*) ao mesmo tempo em que evita colisão com os outros personagens (usando *steering behaviours*). Como a camada de movimento não interfere nas camadas superiores da arquitetura e estas apenas definem um possível destino à procura de caminhos (i.e. o algoritmo de *path-finding* escolhe um *waypoint* a ser atingido e insere na pilha de comportamentos uma ação "ir até posição (x,y)"), este cenário testa a integração entre as duas sub-camadas de movimentação de forma a poderem ser usadas em qualquer outro cenário.

3) Para utilizar algumas funcionalidades das camadas de mais alto nível, foi criado um cenário onde um personagem que, ao explorar o mapa, "decide" ir para casa para comer e depois voltar a explorar. Neste caso, foi usada a abordagem de objetivos onde existiam os objetivos explorar, matar e comer. Sendo a *desirability* de comer e matar inversamente proporcionais à eneriga do personagem (que decrescia com o tempo). A diferença básica entre matar e comer é que o último se declarava uma necessidade fisiológica, o que sempre fazia com que o personagem apenas comesse e explorasse o mundo.

Estes cenários exemplificam a aplicabilidade e adaptação do *middleware* a vários problemas. No entanto, seria bastante interessante e útil a sua aplicação aos diferentes personagens de um jogo complexo, o que não foi possível realizar devido a restrições de escopo e tempo.

Um exemplo da implementação de um personagem fazendo uso mais extenso da hierarquia de classes seria criar um personagem Skeleton, onde a classe que o implementa estende Moving que, por sua vez, estende a classe básica de agente (Agent). A classe Agent possui um objeto Brain, responsável pelo processo de decisão e que, por sua vez, possui os comportamentos. Sabendo que os vários tipos de comportamento herdam de Behaviour, o desenvolvedor pode estender a classe StateMachine para implementar alguma outra funcionalidade, ou até mesmo criar seu próprio comportamento (por meio de herança da classe Behaviour). Outra opção é o desenvolvedor alterar a modelagem de personalidade implementada, simplesmente criando uma subclasse de Personality, atribuindo-a ao agente Skeleton e realizando as modificações correspondentes no cálculo do grau de desejo na arbitragem dos comportamentos.

Independentemente do uso do *framework* ser feito como caixa branca ou caixa preta, o código é disponível, pois uma lição aprendida vinda da experiência dos programadores Unix através de décadas de mudanças constantes é que código fonte dura, código objeto não. Executáveis binários fechados não se adaptam a mudanças e prendem os desenvolvedores a suposições feitas por quem os construiu [147]. Além disso, a disponibilização do código ajuda a mitigar o problema do "não inventado aqui" descrito anteriormente e facilitar adaptações do *middleware* por parte do usuário.

Um cenário mais voltado para um *framework* de caixa preta é apresentado a seguir, sendo ainda mais complexo pois o isolamento dos componetes é ainda maior.

5.5 Estudo de Caso como Biblioteca

Um outro estudo de caso foi implementado como prova de conceito da utilização do *middleware* sem que seja necessário o conhecimento sobre a estrutura de classes da ferramenta e sem que seja necessário o uso de herança.

Para testar essa possibilidade, foi utilizado como base o ambiente Robocode, originalmente criado pela IBM [26][27] e disponível como código aberto no Sourceforge [28]. O Robocode é um simulador em Java para

um jogo de batalha entre robôs em que o desenvolvedor se encarrega de criar um agente tanque e o ambiente instancia esse agente (junto com os oponentes escolhidos) e gerencia a execução das regras do jogo. Cada agente é notificado pelo simulador sobre os eventos ocorridos durante a execução do jogo e pode então tomar suas decisões e agir em resposta.

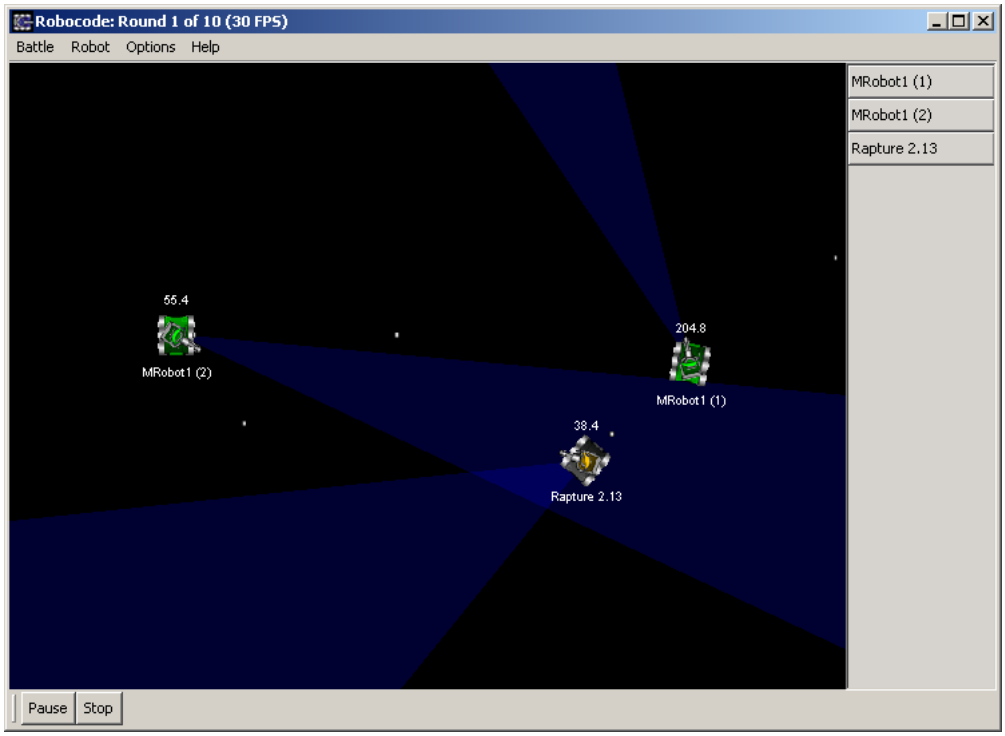

Figura 5.10: Exemplo de execução de um combate entre agentes no simulador Robocode.

Para testar o uso do *middleware* como uma biblioteca foi criado um novo agente para o Robocode, cujo processamento consiste apenas em instanciar o motor de IA, repassar para este os eventos ocorridos no jogo e receber um evento contendo a ação a ser tomada. Além disso, a máquina de estados correspondente ao comportamento do agente é especificada usando a linguagem Lua de modo a evitar a manipulação direta do código do MIAGI.

Como os agentes são desenvolvidos utilizando-se a linguagem Java e o *middleware* foi desenvolvido em C++, foi necessário criar uma interface entre os dois componentes do agente. Para realizar essa integração, Java provê uma API chamada JNI (Java Native Interface) [184] que realiza o mapeamento das chamadas de métodos Java em chamadas ao código nativo contido numa DLL (biblioteca de ligação dinâmica).

Utilizando essas características, foi realizada uma bateria de testes fazendo uso de duas instâncias do agente criado em batalha contra um agente mais esperto. Os dois agentes (representados em verde na Figura

5.10 como Mrobot1 e Mrobot2) compartilham uma mesma instância do *middleware* de IA e, embora sua máquina de estados seja bastante simples, conseguem vencer o oponente aproximadamente 65% das vezes. O que mostrou a viabilidade da utilização do *middleware* como um componente fechado separado da implementação das entidades inteligentes do jogo.

5.6 Aplicação a Outros Gêneros de Jogos

O cenário de testes apresentado representa bem jogos tipo *First Person Shooter* (FPS) e alguns jogos de estratégia em tempo real (*Real Time Strategy* - RTS), no entanto o middleware foi desenvolvido de modo a poder ser aplicado a diversos gêneros de jogos.

Alguns exemplos de jogos que podem ser implementados utilizando a arquitetura proposta são:

Jogos tipo The Sims da Eletronic Arts: No jogo The Sims os personagens são modelados utilizando critérios que definem suas personalidades e necessidades. Tais necessidas podem ser, por exemplo: fome, sede, necessidade de contato social. Além disso, os objetos presentes no ambiente de jogo propagam a informação sobre que necessidades eles suprem. Dependendo dos vários níveis de necessidades apresentados pelos personagens, estes, ao receberem as informações dos objetos, tentarão utilizá-los para supri-las.

Tal cenário poderia ser implementado utilizando o MIAGI uma vez que o *middleware* provê uma modelagem de personalidade baseada no modelo *Big Five* e esses fatores de personalidade influenciam no processo de decisão do agente, o qual também leva em consideração as necessidades do agente. As informações propagadas pelos objetos, por sua vez, podem ser capturadas pelo módulo de cola e convertidas em eventos enviados ao *middleware* através da camada de percepções. Esta camada notifica então os agentes próximos sobre aquele evento e assim o agente pode levá-la em consideração ao decidir que comportamento assumir para suprir suas necessidades.

Jogos de luta: de acordo com a literatura [148], este gênero de jogos necessita na sua implementação de máquinas de estados e sistemas de regras, além de se beneficiar bastante de uma maior imprevisibilidade [29]. Tal gênero pode então ser implementado utilizando o sistema de inferência fuzzy e o esquema de FSMs implementados no MIAGI.

Esta abordagem cobre os requisitos do gênero pois o FIS pode ser usado para representar as regras sem grandes modificações e, também, permite usar lógica *fuzzy* nas regras para conseguir um maior grau de imprevisibilidade do comportamento dos NPCs.

Uma outra possibilidade de utilização do *middleware* é em conjunto com o sistema para a geração e dramatização interativa de histórias Logtell [177]. A aplicação do MIAGI como camada de apoio ao sistema permitiria uma maior variedade de comportamentos aos diversos personagens (coadjuvantes ou não) da história.

Essa integração pode se dar de maneira relativamente simples, pela substituição do *action manager* (responsável pela escolha da próxima ação a ser executada) pelo processo de decisão implementado no *middleware*. Além desta substituição, seriam necessários mais dois passos. Primeiramente, alterar a comunicação com o *drama manager* de modo a permitir que esse insira objetivos de maior importância na pilha dos agentes (pilha de comportamentos do objeto Brain), o que não era previsto no *middleware*. Em seguida, é necessário implementar um histórico das ações passadas, previsto no Logtell, e isto pode ser feito de maneira trivial no MIAGI usando a memória do agente e guardando os nomes das ações executadas após sua finalização.

É importante ressaltar que a integração com a camada de mais baixo nível do Logtell (a camada de planejamento que faz uso do IPG [180]) está fora do escopo de aplicação do MIAGI.

5.7 Conclusões e Observações

Validar um *framework* não é uma tarefa fácil. Uma possível abordagem para este problema seria a de realizar duas implementações para um único jogo. Uma utilizando o motor e outra não. Assim seria possível comparar os dois projetos e verificar o desempenho, tempo de desenvolvimento e as facilidades e limitações do uso do *framework*.

Contudo, esta abordagem possui um número muito grande de variáveis e é muito complexa, o que a torna não aplicável dentro do escopo e tempo disponíveis para este trabalho.

Foram então realizados testes do *middleware* contra um cenário de testes tipo jogo que replique as principais caracteísticas deste tipo de ambiente. Além disso, de modo a testar o MIAGI também como

biblioteca (uma abordagem de *framework* caixa preta) foi testado um outro cenário inclusive com separação por causa das linguagens de programação empregadas. Em ambos, o *middleware* apresentou comportamento bastante satisfatório, o que indica que se encontra em um estágio com bom grau de flexibilidade. Embora sua utilização como *framework* de caixa preta ainda necessite de uma melhor definição das interfaces, antecipando um conjunto mais amplo de casos de uso potenciais.

Outra iniciativa realizada para tentar analisar a flexibilidade e generalidade do *middleware*, foi uma pequena análise de como se daria a implementação de jogos de outros gêneros que não os cobertos pelo cenário de teste e baseado nas conclusões desta, pode-se concluir também que o conjunto de funcionalidades disponibilizado pela versão atual da implementação já permite sua utilização em diversos tipos de jogos digitais.

Tabela 5.1: MIAGI comparado aos diferentes motores de IA analisados.

	Suporte a decisão	**Serviços**	**Comportamentos**	**Código fonte**	**Ferramentas**
MIAGI	FSMs, FIS, GOAP	*Path-finding* simples e comportamentos de movimentação	Criados pelo usuário com personalidades e necessidades	Sim	*Scripts*
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

Pode-se concluir que a implementação atual atende aos requisitos citados anteriormente. É de relativa fácil utilização, como sugerido pelos cenários de teste; ajuda a melhorar a jogabilidade ao prover diferentes

maneiras de modelar os comportamentos dos NPCs; auxilia a criação de jogos também ao prover essas diferentes técnicas de IA, um modelo de agentes utilizável e ao explicar sua estruturação; apresenta suporte a linguagem de *script* de modo a permitir um desenvolvimento mais *data-driven*; suporta o gerenciamento dessas entidades com comportamento e claramente suporta funcionalidades de IA, uma vez que este é seu principal propósito (um resumo das características do MIAGI pode ser visto na Tabela 5.1).

Embora o MIAGI atinja esses diversos objetivos, não foram implementadas ferramentas de apoio à criação dos comportamentos e devido a restrições de escopo e tempo, problemas de eficiência não foram estudados. Além disso, a utilização do *middleware* em diferentes jogos completos pode trazer ainda mais clarificações sobre os vários problemas da criação deste tipo de sotware, em especial os relacionados ao componente cola.