Um Framework para o Desenvolvimento de Agentes Cognitivos em Jogos de Primeira Pessoa

Ivan M. Monteiro
Universidade Federal do Rio Grande do Sul

Debora A. Santos
Universidade Federal da Bahia

Resumo

Em jogos eletrônicos é freqüente a necessidade da existência de agentes capazes de emular o comportamento humano. Tais agentes são denominados BOTs. Neste trabalho é apresentada uma arquitetura híbrida para agentes cognitivos no contexto de jogos, juntamente com um *framework* que dá suporte ao desenvolvimento desses agentes. Essa arquitetura visa simplificar o processo de modelagem dos BOTs. Já o *framework* fornece a estrutura para implementação e reutilização das partes desses agentes.

Keywords:: Inteligência Artificial, Agentes, Arquitetura de Agentes, Framework.

Author's Contact:

immonteiro@inf.ufrgs.br
abdalla@ufba.br

1 Introdução

Os jogos eletrônicos possuem um importante papel nos campos relacionados a inteligência artificial, podendo ser utilizados como laboratórios para testar tanto teorias sobre o raciocínio humano quanto métodos de resolução de problemas[van Waveren 2001]. O desenvolvimento de BOTs[1] caracteriza bem este papel, pois nos jogos, os BOTs devem atuar como jogadores virtuais dotados de inteligência artificial tentando emular[2] seqüências de comportamentos dos jogadores humanos.

Com o avanço tecnológico, tem-se aumentado a necessidade e realismo dos jogos atuais, o que torna o desenvolvimento dos BOTs muito mais complexos. Não basta o BOT cumprir sua função básica no jogo, que é interagir com os outros jogadores dentro da lógica do jogo, ele também precisa apresentar comportamentos naturais dos jogadores humanos, como ter conhecimento do ambiente em que está inserido, guardar sua história pessoal, exibir emoção e personalidade como se ele tivesse vida própria, sendo confundido com um jogador humano. Os jogadores humanos têm-se mostrado mais satisfeitos com o jogo se eles acreditam que o personagem com quem ele interage é um jogador humano ao invés de controlado por computador.

Em jogos, os personagens podem ser controlados tanto por humanos como por computador. O primeiro é visto como mais inteligente, flexível, imprevisível, com habilidade de adaptar suas estratégias ao estado dos jogos, possibilidade de raciocinar sobre suas ações e também capaz de utilizar diferentes estratégias para cada momento do jogo. Já os controlados por computador costumam ser lembrados como personagens previsíveis e com ações pré-definidas, o que torna entediante jogar com eles[de Byl 2004].

Para atingir o nível de satisfação desejado com os BOTs, diversas técnicas em computação são utilizadas, como máquina de estados finitos e *scripting*, mas é em inteligência artificial que se concentram as principais técnicas para se desenvolver um personagem convincente, tais como: buscas heurísticas, sistemas baseados em conhecimento, lógica fuzzy, redes neurais. Entretanto, as técnicas de forma isolada são insuficientes para resolver o problema. É a combinação adequada das técnicas que traz um resultado satisfatório. Neste contexto, a modelagem de um BOT como um agente, entidade que percebe e atua sobre o ambiente, é muito importante. Primeiro, porque o BOT está captando percepções do ambiente e executando ações como um agente. Segundo, porque a modelagem em agentes facilita a integração das diversas técnicas da inteligência artificial.

Uma metodologia para o desenvolvimento de BOTs baseado em agentes, além de útil, é bastante importante. Útil porque encurta o tempo do seu desenvolvimento atacando diretamente o problema com um modelo de sua solução. Importante porque permite o estudo e a avaliação da forma de desenvolver e das técnicas possíveis de serem utilizadas.

A forma de uma arquitetura de agentes está intimamente ligada à tarefa que o agente irá executar no ambiente[Nilsson 1998]. Por isso, a escolha correta da arquitetura para implementar um BOT facilita bastante o seu desenvolvimento. Além disso, esta escolha também propicia o estabelecimento de uma metodologia para o desenvolvimento desses agentes.

A utilização de uma estrutura definida, aliada a uma arquitetura de agente, servindo de suporte para a organização e o desenvolvimento de BOTs, simplifica bastante a criação desses jogadores virtuais. Esta estrutura desejada é fornecida neste trabalho através de um *framework* para a criação de agentes em jogos de primeira pessoa. A intenção desse *framework*, nomeado de IAF , é facilitar o desenvolvimento do software, que neste caso é o programa do agente. Esta estrutura se baseia na arquitetura híbrida para agentes cognitivos proposta em [Monteiro 2005] e traz funcionalidades e componentes de reúso para o desenvolvimento desses agentes.

Com isto, este trabalho simplifica o uso de jogos eletrônicos como laboratório para pesquisas e estudos voltados à computação e também amplia as possibilidades de desenvolvimento de jogadores virtuais. A área de Inteligência Artificial é bastante beneficiada com esta aproximação, uma vez que muitas técnicas presentes nesta área podem ser diretamente aplicadas ao contexto de jogos. Outro importante ponto é que o conhecimento utilizado neste contexto de jogos tem relacionamento estreito com as simulações da robótica, o que permite que os programas de agentes possam ser testados antes de serem embarcados.

A sessão 2 apresenta alguns trabalhos relacionados, a sessão 3 expõe a arquitetura[Monteiro 2005] que é utilizada como base do desenvolvimento do IAF . Na sessão 4 o *framework* é descrito, na quinta, são feitos alguns testes e avaliações do *framework* e do desenvolvimento de BOTs e na sessão 6 são feitas as considerações finais.

2 Trabalhos Relacionados

O projeto *GameBots*[Andrew Scholer 2000][Kaminka et al. 2002] busca levar o jogo *Unreal Tournament* ao domínio de pesquisa em inteligência artificial. Associado a ele existe o projeto *JavaBots*[Adobbati et al. 2001], que fornece uma *API* para o desenvolvimento de agentes para o jogo *Unreal Tournament*. Essa *API* tem como principal objetivo permitir que desenvolvedores consigam criar BOTs sem se preocuparem em lidar com aspectos específicos do protocolo do *GameBots*.

Apesar de também utilizar o GameBots como base, este trabalho vai além de uma abstração da comunicação com o jogo, como é visto no *JavaBot*. O *framework* aqui introduzido, fornece uma implementação estrutural da arquitetura de agente apresentada em [Monteiro 2005], aplicada ao jogo Unreal Tournament. Assim, este trabalho pretende estreitar ainda mais as distâncias entre o domínio

[1] A palavra BOT surgiu da simplificação da palavra ROBOT. Este termo ficou bastante conhecido na área de jogos referindo-se a personagens que não são controlados por jogadores humanos.

[2] Freqüentemente o termo emular é confundido com simular. Uma distinção breve seria que enquanto emular refere-se a o que é feito, simular está relacionado a como foi feito.

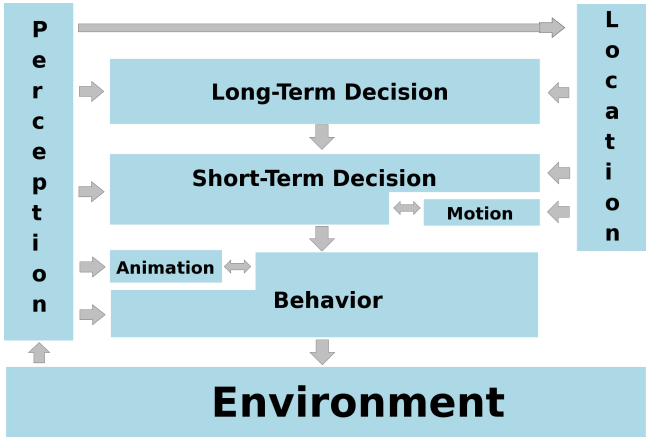

Figura 1: *A arquitetura.*

de pesquisa em inteligência artificial e sistemas multiagentes com o domínio de jogos eletrônicos.

3 A Arquitetura de Agente

Nesta sessão, é apresentada uma arquitetura modular para o desenvolvimento de agentes cognitivos em jogos de primeira e terceira pessoas[Monteiro 2005]. Essa, visa suprir uma lacuna existente na área de agentes inteligentes em jogos eletrônicos, que é a falta de arquiteturas de agente híbrido especializado ao contexto de jogos. A seguir são descritos os módulos presentes na arquitetura, juntamente com suas interações.

A arquitetura de agente cognitivo apresentada aqui é composta por sete módulos:

- **Perception -** Módulo de eventos responsável pela filtragem dos sensores do agente.
- **Location -** Centraliza informações sobre o modelo do ambiente para o agente.
- **Long-Term Decision -** Responsável pelo planejamento.
- **Short-Term Decision -** Responsável pela execução do plano atual.
- **Motion -** Manipula aspectos de roteamento, colisão e desvios de obstáculos.
- **Animation -** Módulo responsável por determinar a animação a ser exibida.
- **Behavior -** Módulo responsável pela atuação sobre o ambiente de forma reativa.

Dos sete módulos apresentados, cinco são responsáveis pelo processo de decisões do agente, são eles: *Long-Term Decision*, *Short-Term Decision*, *Motion*, *Animation*, *Behavior*. Os módulos de decisão estão agrupados em 3 camadas:

- **Camada Deliberativa -** Responsável por gerar soluções globais para tarefas complexas, composta pelo módulo *Long-Term Decision*.
- **Camada Executiva -** Responsável pelo sequenciamento do plano passado pela camada deliberativa, composta pelos módulos *Short-Term Decision* e *Motion*.
- **Camada Reativa -** Responsável por responder as percepções do ambiente, composta pelos módulos *Animation* e *Behavior*.

O fluxo de informações através dos módulos é indicado na Figura 1. Os dados captados do ambiente seguem para o módulo *Perception* que os filtra e os envia para os módulos responsáveis. *Perception*, após a filtragem, envia informações para *Location*, *Behavior*, *Animation*, *Short-Term Decision* e *Long-Term Decision*. Com a informação recebida de *Perception*, *Location* atualiza o modelo de ambiente do agente e disponibiliza, de forma síncrona com *Perception*, consultas a este modelo de ambiente para os módulos: *Long-Term Decision*, *Short-Term Decision* e *Motion*. Baseado no modelo atualizado do ambiente e nas percepções, *Long-Term Decision* utiliza sua representação interna de conhecimento para gerar planos que atendam aos objetivos do agente. Estes planos são passados para o *Short-Term Decision* que fica responsável pela execução dos mesmos. Para isto, *Short-Term Decision* utiliza as percepções e o modelo atualizado do ambiente juntamente com o módulo *Motion*, para decidir como executar cada tarefa de um dado plano. A execução das tarefas é feita trocando o comportamento reativo atual do módulo *Behavior*. Assim, *Behavior* atua sobre o ambiente baseado em suas percepções e o comportamento ativo atual. *Animation* e *Motion* são sub-módulos especializados de *Behavior* e *Short-Term Decision*, respectivamente. *Animation* comunica-se com o *Behavior* para decidir qual animação exibir e pode também receber percepções para gerar novas animações. Já o *Motion* é usado para decidir sobre a navegação do agente no ambiente.

3.1 Módulo Perception

Todos os dados captados por sensores do agente são tratados como percepções. Estes sensores podem ser dos mais variados tipos, imitando alguns dos sentidos humanos, sensores de visão, audição, tato e olfato. Cada percepção recebida deve ser repassada para os módulos apropriados. Assim, *Perception* centraliza o recebimento de percepções, cria um modelo de representação interna dessas percepções e filtra o envio dessas para cada módulo. Essa filtragem é importante para fazer que se chegue nos módulos apenas informações relevantes. Por exemplo, o som ambiente de que está ventando muito gera uma percepção auditiva que apesar de não ser muito útil para *Behavior*, pode ser interessante para *Long-Term Decision* concluir que vem chegando uma tempestade.

3.2 Módulo Location

O módulo *Location* possui uma representação do ambiente de modo a facilitar questões como determinação de rotas, navegação, desvio de obstáculos e localização de objetos no ambiente. Em jogos, é comum gerar essa representação de maneira pré-processada, diminuindo assim, o processamento em tempo de execução. Com isto, para o agente, *Location* atua como um grande oráculo sobre o ambiente, sendo atualizado diretamente por conjuntos de percepções.

3.3 Módulo Long-Term Decision

Ganhar o jogo é o mais importante objetivo para um BOT. Atingir esse objetivo significa tomar uma série de decisões que conduza o jogo ao estado de vitória para este BOT. *Long-Term Decision* é o módulo responsável pelas decisões de mais alto nível para o agente. Este módulo utiliza modelos de tomada de decisão para gerar soluções globais para tarefas complexas. A principal atribuição para este nível é o planejamento. Assim, *Long-Term Decision* fica responsável por decidir qual a meta atual e o melhor plano para atingir esta meta. Este plano é enviado para *Short-Term Decision*, que fica encarregado de executá-lo. Com isto, *Long-Term Decision* concentra-se em tipos de soluções estratégicas, delegando a execução dos planos para a camada inferior.

3.4 Módulo Short-Term Decision

Short-Term Decision atua como integrador das decisões tomadas pelo *Long-Term Decision* e as ações executadas pelo *Behavior*. Dado um plano passado para *Short-Term Decision*, este módulo fica responsável por decidir como executar o plano recebido. Este módulo também possui a liberdade de acrescentar ao plano, tarefas que otimizem o estado do agente e não desvie do objetivo do plano. A execução de tarefa dentro de um plano é feita selecionando os comportamentos dentro de *Behavior*. Dada uma seqüência de tarefas a ser executada, *Short-Term Decision* decide qual o comportamento ativar para a execução de cada uma destas tarefas. Uma tarefa pode conter um ou mais comportamentos seqüenciados. Dessa forma, *Short-Term Decision* atua sobre *Behavior* como um seletor do comportamento atual.

3.5 Módulo Motion

Motion é o responsável por entender como o agente deve se movimentar no mundo. Ele recebe informação de *Short-Term Decision* indicando para onde deve se movimentar e então decide a forma apropriada de se mover para o destino. Este módulo não executa a movimentação, apenas determina como executar. Toda decisão tomada por este módulo é computada com base nas estruturas de *Location*.

3.6 Módulo Animation

Animation é responsável por controlar o corpo do BOT decidindo qual animação será apresentada para demonstrar o estado atual de ação deste BOT. Estas animações são muitas vezes pré-geradas ou por animadores profissionais ou por captura de movimento, entretanto é possível gerar animação em tempo de execução[Grünvogel 2003]. Um exemplo de geração de animação em tempo de execução é a utilização de cinemática inversa[Eberly 2000].

3.7 Módulo Behavior

Behavior é o módulo responsável por executar a ação sobre o ambiente. Dado que *Short-Term Decision* selecionou um comportamento, este passa a ser o comportamento reativo atual. O comportamento reativo selecionado capta as percepções do ambiente e executa ações sobre o mesmo. O conjunto de ações do agente é similar às ações executadas pelo jogador humano através dos dispositivos de entrada.

4 Indigente Agent Framework

Nesta sessão, é apresentado o IAF - Indigente Agent Framework- um *framework* para o desenvolvimento de agentes cognitivos em jogos de primeira pessoa. Este *framework* fornece a estrutura para o desenvolvimento de agentes baseados na arquitetura apresentada em [Monteiro 2005]. O jogo Unreal Tournament 2004 foi escolhido como plataforma de estudo de caso devido a facilidade de interação com uma aplicação externa. A seguir é apresentada uma breve descrição do jogo e a organização do *framework* em questão.

4.1 Unreal Tournament 2004: O jogo

UT2004 é um jogo de tiro de primeira pessoa com suporte a multi-jogadores onde os personagens se enfrentam nos mais variados ambientes. Ele pode ser jogado de diversas maneiras dependendo da modalidade de jogo escolhida. Cada uma dessas modalidades oferece uma maneira distinta de jogar o mesmo jogo, mudando apenas as regras do mesmo.

As modalidades disponíveis em UT2004 são: Assault, Onslaught, DeathMatch, Capture The Flag, Team DeathMatch, Double Domination, Bombing Run, Mutant, Invasion, Last Man Standing e Instagib CTF. Dessas, duas se destacam e assumem o papel de modalidades clássicas, por estarem presentes em diversos outros jogos de tiro de primeira pessoa, são elas o DeathMatch e Capture The Flag. No formato do DeathMatch o jogador luta pela sua sobrevivência tentando ao máximo manter-se vivo enquanto destrói qualquer um que apareça em sua frente. Já no Capture The Flag, dois times de jogadores se confrontam com o objetivo de proteger a bandeira do time e sequestrar a bandeira inimiga.

Nas modalidades acima o jogo ocorre da mesma forma. O jogador tem a mesma visão que tem o personagem dentro do jogo(por isso a classificação como primeira pessoa), assim todas as percepções visuais do personagem são também as percepções dos jogadores. O personagem normalmente carrega uma arma a sua frente e tem conhecimento de seus pontos de vida e quão resistente está sua armadura. Como BOT também é um jogador, o agente modelado para implementar este BOT possuirá percepções similares a de um jogador humano.

Uma das principais vantagens de se utilizar o jogo UT2004 como laboratório é que ele é bastante personalizável. Novos módulos podem ser acrescidos utilizando uma linguagem própria que é o

Figura 2: *Tipos de Jogos Adicionados.*

Unreal Script. Devido a esta flexibilidade, surgiu , inicialmente no Instituto de Ciências da Informação da University of Southern California, o projeto *GameBOTs*[Andrew Scholer 2000][Kaminka et al. 2002] que será detalhado a seguir.

4.1.1 GameBots

O projeto *GameBOTs*[Andrew Scholer 2000][Kaminka et al. 2002] surgiu como uma iniciativa do Instituto de Ciências da Informação da University of Southern California em desenvolver uma modificação da primeira versão do jogos Unreal Tournament. A idéia deu tão certo que diversas universidades passaram a utilizar essas modificações e hoje já existem modificações para as versões mais recentes, como Unreal Tournament 2003 e UT 2004. Estas modificações funcionam como uma extensão das funcionalidades de controle do jogo, permitindo que personagens sejam controlados via socket TCP . Com isso, é possível desenvolver um agente que controla o personagem fazendo uso de qualquer linguagem de programação que tenha suporte a socket TCP.

Com a utilização da modificação, são criados quatro novos tipo de jogos. Estes novos são extensões de modos já existentes adicionando suporte a conexão via socket TCP. Os tipo são: Remote Team DeathMatch, Remote Capture The Flag, Remote Double Domination e Remote Bombing Run.

Quando o BOT está conectado ao jogo, informações sensoriais são enviadas pelo jogo através do socket de conexão e baseado nestas percepções o agente pode atuar enviando comandos através do mesmo socket. Os comandos enviandos definem como o personagem atua no ambiente, controlando movimentação, ataque e comunicação entre personagens.

Percepções e Ações A comunicação entre o agente e o jogo se dá através de troca de mensagens. O jogo envia mensagens síncronas e assíncronas com as informações sensoriais enquanto o agente atua também enviando mensagem. Uma descrição completa sobre as mensagens trocadas entre o agente e o jogo é apresentada em [Andrew Scholer 2000].

As mensagens síncronas chegam para o agente em lote num intervalo configurável. Elas incluem informações como a atualização visual do que o BOT vê e reporta o estado do próprio BOT. No início do lote das mensagens síncronas, o jogo envia uma mensagem de início do lote (BEG) e nessa mensagem contém o tempo ao qual o lote se refere. Logo após esta mensagem de início, todas as mensagens síncronas são enviadas até chegar uma mensagem de fim do lote (END). Todas as mensagens contidas no lote referem-se a um mesmo instante do tempo no jogo.

As mensagens assíncronas por outro lado, refletem os eventos ocorridos no jogo. Elas nunca aparecem entre um BEG e um END já que elas não são síncronas. Estas mensagens representam as coisas que podem acontecer a qualquer momento aleatório do jogo, tais

como: tomar dano; difusão de mensagem por algum outro personagem; bater na parede. Sempre que ocorrer um evento no jogo que gere uma mensagem assíncrona, esta mensagem estará antes ou depois de um lote de mensagem síncrona. Entretanto, não é possível garantir que a mensagem assíncrona refere-se ao mesmo tempo do jogo que o lote anterior ou posterior.

As ações dos BOTs são determinadas pelas mensagens enviadas pelo programa do agente. Essas mensagens seguem o mesmo estilo de formatação das percepções que chegam, que é um identificador seguido de zero ou mais argumentos. A maioria dos comandos possuem efeitos persistentes, o que significa que comandos como movimento e rotação, uma vez iniciados, só irão parar quando alcançar o destino, e o comando de atacar mantém o personagem atirando até que seja enviado um outro comando de parada do ataque.

No *framework*, foi desenvolvido uma camada de abstração para mensagens sensoriais e os comandos. Para toda mensagem que chega no agente é feito o parser de cada uma e instanciados objetos de mensagens que contém as informações da mensagem de fácil acesso para o *framework*. Quanto aos comandos, uma classe chamada MailBox se encarrega de enviar cada comando possível para o jogo através de chamadas de métodos. Esta mesma classe MailBox, fica responsável por receber as mensagens e encaminhá-las para que seja feito o parser.

4.2 Organização do IAF

O IAF é feito essencialmente em C++, utilizando os principais conceitos de orientação a objeto. Ele é separado em pacotes onde são agrupadas as diversas funcionalidades para dar suporte a criação de um BOT.

Os pacotes do *framework* fornecem as mais variadas funcionalidades. Estas funcionalidades estão agrupadas basicamente em: rede, arquitetura, matemática e máquina de estados finitos. A seguir, são descritos os conteúdos de cada pacote juntamente com o funcionamento dos mesmos

4.2.1 Comunicação

A comunição entre agente e jogo se dá através de troca de mensagens. Devido a isto foi necessário o desenvolvimento de todo o suporte para comunição em rede. Neste suporte está incluido desde a conexão através socket, até a transformação da mensagem em um tipo abstrado de dado[3]. Desta forma, as mensagens passam a ser utilizadas pelo *framework* como informação e não mais como dados.

Uma classe MailBox é responsável por receber e enviar as mensagens. Como a quantidade de comandos é pequena, o envio deles se dá através de chamadas de método da classe MailBox. Já para o recebimento das mensagens, é utilizado uma thread com espera bloqueante que *bufferiza* os pacotes chegados formando mensagens que são enviadas para a fábrica de mensagens sensoriais.

A fábrica de mensagens sensoriais, o SensoryMessageFactory, é responsável por, dado uma mensagem em texto plano, no formato TYPE Arg1 Values1 ... ArgN ValuesN, gerar informação útil para o *framework*, ou seja, criar uma instância referente ao tipo de mensagem, com as operações necessárias para o acesso da mesma. Para isso, a fábrica realiza o parser das mensagens e decide qual instância de mensagen sensorial deve criar. As mensagens criadas são todas filhas de SensoryMessage, e elas se dividem em dois tipos básicos que são as mensagens síncronas e assíncronas.

As mensagens síncronas são recebidas pelo agente em um intervalo configurável e informam essencialmente o que o BOT vê, o estado do mesmo e a pontuação no jogo. Já as mensagens assíncronas informam eventos que podem ocorrer a qualquer instante no jogo, como: ser atingido, comunicação textual, mudança de ambiente, colisão com alguma parede ou jogador, dano tomado, entre outros.

A entrega dessas mensagens é feita pelo próprio MailBox, que mantém uma lista de inscritos interessados em mensagens chega-

[3]Tipo usado para encapsular outros tipos de dados e que possuem operações associadas

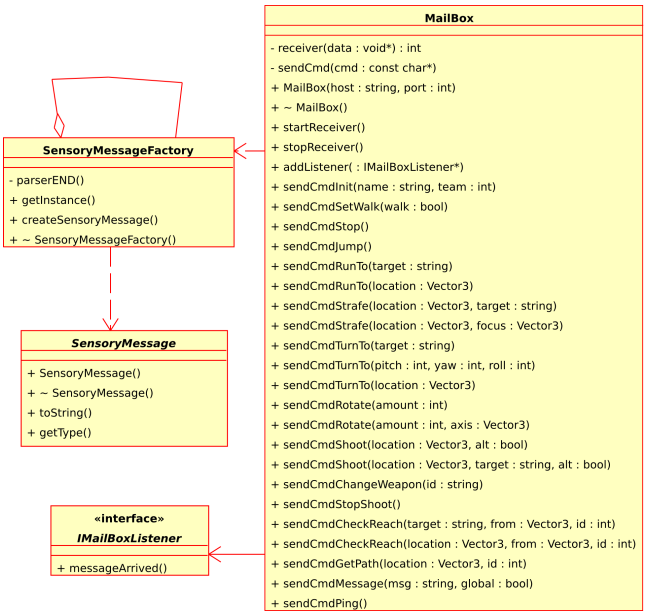

Figura 3: *Conjunto de classes e interface utilizada na comunicação.*

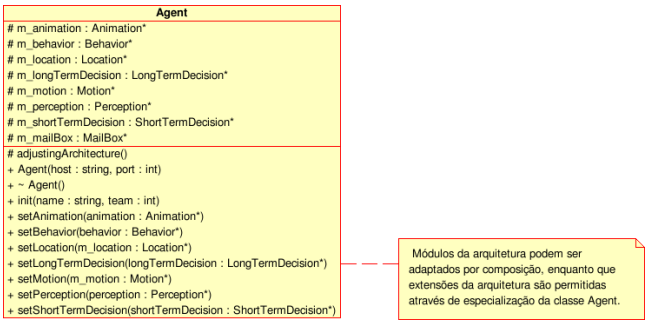

Figura 4: *Classe básica para a composição de um agente.*

das. Estes interessados precisam implementar IMailBoxListener para então se inscrever. Uma vez inscrito no MailBox, para toda mensagem chegada, será gerada um SensoryMessage para ele.

Utilização da Comunicação: A utilização dos componentes de comunição do *framework* permeia o uso da classe MailBox, que é utilizada tanto para o envio como o para o recebimento de mensagens. Novas mensagens podem ser criadas com a especialização da classe SensoryMessage, como pode ser visto na Figura 3.

4.2.2 Implementação da Arquitetura

O pacote architecture fornece a implementação da estrutura proposta pela arquitetura definida em [Monteiro 2005]. Neste pacote, são organizados cada um dos módulos da arquitetura juntamente com seus relacionamentos. Uma importante classe deste pacote é a classe Agent que contém a estrutura de um agente cognitivo que implementa a arquitetura proposta. Esta classe Agent é de fácil extensão o que permite sua reutilização também para evoluções que venham ocorrer na arquitetura. Ela implementa todas as funcionalidades necessárias para se criar um BOT, o que inclui toda a parte de comunicação com o jogo. A seguir são descritos os módulos implementados.

Utilização da Arquitetura: A classe Agent é a classe-base para o desenvolvimento de um agente no IAF . Ela define todos módulos da arquitetura e sua adaptação é feita através da composição de novos módulos, como pode ser visto na Figura 4.

Perception O módulo *Perception* tem como principal funcionalidade agrupar as informações sensoriais e entregar para os devidos módulos as informações necessárias. Ele implementa IMailBox-

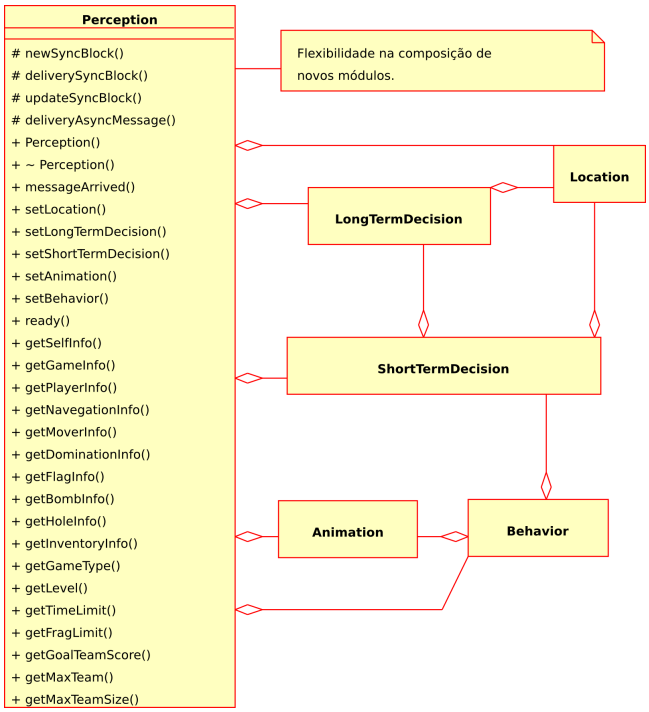

Figura 5: *O módulo perception como proxy das percepções do agente*

Listener, o que significa que ele é notificado pelo MailBox sempre que um nova mensagem chega.

Como as informações sensoriais podem ser síncronas ou assíncronas, *Perception* dá um tratamento diferenciado para cada um dos dois tipos.

As mensagens assíncronas são enviadas contendo as informações do evento que disparou, desta forma, cada evento contém a sua mensagem assíncrona equivalente. Já para as mensagens síncronas, o que as disparam é o temporizador, mas neste caso o temporizador não dispara apenas uma mensagem contendo a informação necessária, mas sim, um conjunto de mensagens delimitado por mensagens de BEGIN e END. O módulo *Perception* é quem se encarrega de agrupar essas mensagens síncronas e só entregar para os outros módulos o bloco completo após o recebimento do END. Além disso, é possível fazer a seleção de quais mensagens sensorias devem ser passadas como percepção para cada módulo.

Uma outra funcionalidade do *Perception* é manter o histórico centralizado de percepções. Como a arquitetura é bem modular, evita que cada módulo tente manter esse histórico a fim de realizar planejamento ou tomadas de decisões. As mensagens são armazenadas em uma fila e o tamanho dessa fila é configurável.

Utilização de Perception: Todas as percepções passam pela classe *Perception* através de troca de mensagens.O método messageArrived define o que deve ser feito para cada mensagem recebida. Neste método é que são separadas as mensagens síncronas de assíncronas para entrega no demais módulos. A estrutura de *Perception* é apresentada na Figura 5.

Location O módulo *Location* utiliza a representação interna do ambiente através de *WaypointSystem* e ele pode ser facilmente extendido para utilizar mais alguns outros tipos de representação, como campos potenciais[Tozour 2004] ou mapas de influências[Schwab 2004].

O *WaypointSystem* é um grafo não orientado que é atualizado inserindo-se um novo nó a cada novo marco encontrado no jogo, conforme a figura 9. Sobre este grafo são permitidos, além das operações de construção do grafo, a consulta sobre o caminho mínimo entre os dois marcos e a verifação de nós já visitados.

Location tem importante papel no processo cognitivo, servindo

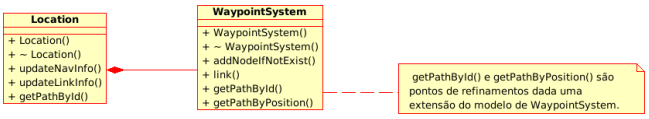

Figura 6: *Location utilizando po padrão o WaypointSystem.*

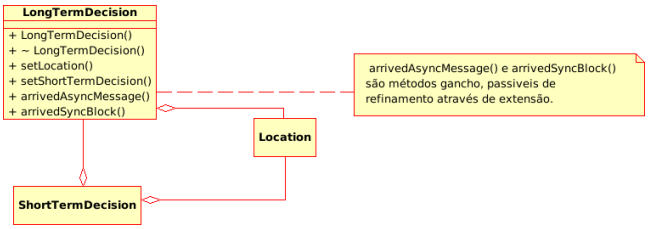

Figura 7: *Os relacionamentos de LongTermDecision e seus hots-pots.*

como uma representação interna do ambiente. O auxílio no planejamento de trajetória é apenas umas das importantes funcionalidades fornecidas por esse módulo. Informações sobre os nós visitados e as características desses nós e suas vizinhanças auxiliam bastante no processo decisório do agente.

Utilização de Location: Cada nova mensagem de navegação é tratada pelo método *updateNavInfo*, que por padrão atualiza o *WaypointSystem*. Já para as mensagens de informação de caminho que chegam, o método *updateLinkInfo* é chamado. Na implementação padrão, este último método atualiza as ligações entre os nós do *WaypointSystem*. O método *getPathById* é também outro ponto de flexibilidade permitindo que especializações de *Location* definam a forma como o caminho no ambiente é gerado. A estrutura de *Location* pode ser vista na Figura 6.

Long-Term Decision *Long-Term Decision* é o módulo utilizado para implementar a camada deliberativa do agente. Sua implementação no *framework* é apenas estrutural, deixando a flexibilidade de extensão para a fase de projeto do agente. Sua estrutura facilita integração com outras arquiteturas baseadas em conhecimencimento como o SOAR[Laird et al. 1987]. A idéia principal desse módulo é gerar planos que devem ser executados pelo *Short-Term Decision*.

Utilização de LongTermDecision: Permite a especialização de uma camada deliberativa para o agente através dos métodos gancho *arrivedAsyncMessage* e *arrivedSyncBlock* , como pode ser visto na Figura 7. Os métodos *arrivedAsyncMessage* e *arrivedSyncBlock* são chamados por *Perception* com a chegada das mensagens sensoriais.

Short-Term Decision O módulo *Short-Term Decision* é o responsável pelo seqüenciamento das reações do agente, gerando um comportamento complexo de alto nível. Sua implementação no *framework* é apenas estrutural, deixando em aberto a forma como o seqüenciamento dos comportamentos é feito. Uma forma simples da sua implementação é utilizando máquinas de estado finito, entretanto, neste aspecto as máquinas possuem um grande inconviniente que é o da previsibilidade das transições de estado.

Utilização de ShortTermDecision: Os métodos *arrivedAsyncMessage* e *arrivedSyncBlock* são métodos gancho, ou seja, permitem a adaptação de *ShortTermDecision* através de sua sobrescrita numa especialização. A Figura 8 apresenta a estrutura de *ShortTermDecision*.

Motion O módulo *Motion* é utilizado principalmente para o planejamento de trajetória. Apesar do jogo UT2004 já fornecer suporte interno para o planejamento de trajetória, este suporte é limitado no que se refere a interferência no processo de planejamento. Com ele, não é possível, por exemplo, escolher o caminho com melhor utilidade ao invés do menor caminho. Essa deficiência é superada

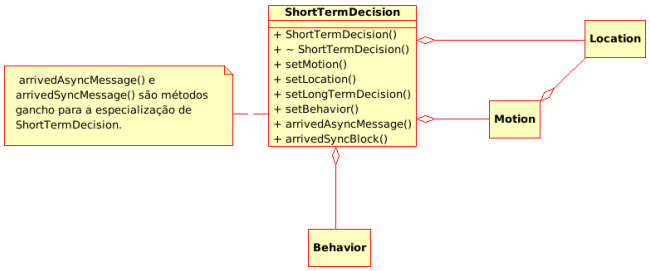

Figura 8: *ShortTermDecision e seus relacionamentos.*

Figura 9: *Marcos visualizados pelo BOT, representado como cilindros preto quando habilitado o modo de debug.*

devido a criação do modelo interno do ambiente, que é a fonte de dados para o módulo *Motion*.

Motion implementa essencialmente o algorimo A* sobre a representação em grafos de *Location*. Entretanto, com a extensão de *Motion*, é possível gerar novas formas de planejamento de trajetória, o que não é permitido com a representação fixa oferecida pelo jogo.

Utilização de Motion: *Motion* oferece o método *getPath* que pode ser sobrescrito para atender a novos modelos de ambiente. A Figura 10 apresenta a estrutura de *Motion*.

Animation O módulo *Animation* possui apenas sua implementação estrutural e um dos principais motivos para isto é que o jogo UT2004 não fornece uma maneira direta de controlar a animação do personagem. Esta animação fica a cargo do prórpio jogo. Mesmo assim, *Animation* pode ser extendida para suportar os diversos modos de se animar um personagem. Como é bastante comum a utilização da técnica de keyframe para animação, uma extensão de *Animation* junto a uma máquina de estado seria uma alternativa imediata.

Utilização de Animation: O controle da animação pode ser definido pelas mensagens que chegam através do *arrivedAsyncMessage* e *arrivedSyncBlock*. A Figura 11 apresenta a estrutura de *Animation*.

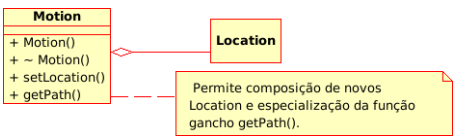

Figura 10: *Modulo de acesso à entidade Location.*

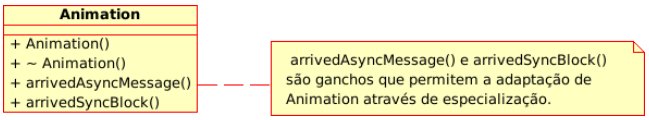

Figura 11: *Implementação estrutural do módulo responsável pela animação dos BOTs.*

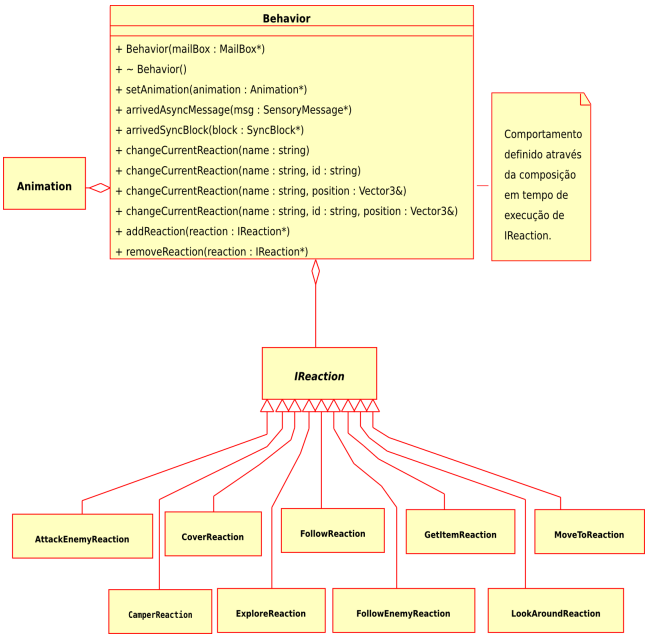

Figura 12: *Modulo behavior com adaptações através de composição.*

Behavior O módulo *Behavior* atua diretamente com o ambiente, enviando os comandos básicos definidos pelo jogo. Ele possui uma lista de comportamentos na qual apenas um é ativado por vez. Estes comportamentos são ativados pelo módulo *Short-Term Decision*, e uma vez ativado, ele passa a responder por todas as percepções que chegem a *Behavior*. Apesar da possibilidade de implementação dos comportamentos de diferentes formas, o *framework* pré-disponibiliza alguns comportamentos implementados como máquinas de estados finitos. Esses comportamentos são utilizados como reações elementares pelo *Short-Term Decision*, que forma comportamentos complexo através de composição.

Os comportamentos implementados por padrão são específicos para a atuação nos jogos UT, UT2003 e UT2004. Uma importante observação quanto as máquinas de estados é que elas podem compartilhar estados com outras máquinas. Desta forma, um comportamento de "Tocaia"que necessite de um estado de "Aguardando Inimigo", pode compartilhar esse mesmo estado com um outro comportamento de "Protegendo Item", diminuindo o esforço com a implementação.

Utilização de Behavior: O funcionamento de *Behavior* é dependente das *IReaction* que o compõe. As *IReaction* são selecionadas como o comportamento ativo através do método *changeCurrentReaction*. Desta maneira, os métodos *arrivedASyncMessage* e *arrivedSynBlock* são repassados para o recebimento de mensagens sensoriais da *IReaction* ativa. Novos comportamentos podem ser definidos através da especialização de *IReaction* que também disponibiliza os *hotspots arrivedAsyncMessage* e *arrivedSyncBlock*. A Figura 12 apresenta a estrutura de *Behavior*.

4.2.3 Funcionalidades Matemáticas

Funcionalidades referentes a matemática são necessárias em diversos momentos da implementação de um BOT, principalmente as funcionalidade referentes ao cálculo vetorial. Isto porque, constantemente o BOT necessita calcular qual sua distância em relação a algum ponto, a nova posição de um determinato objeto, ou até quais

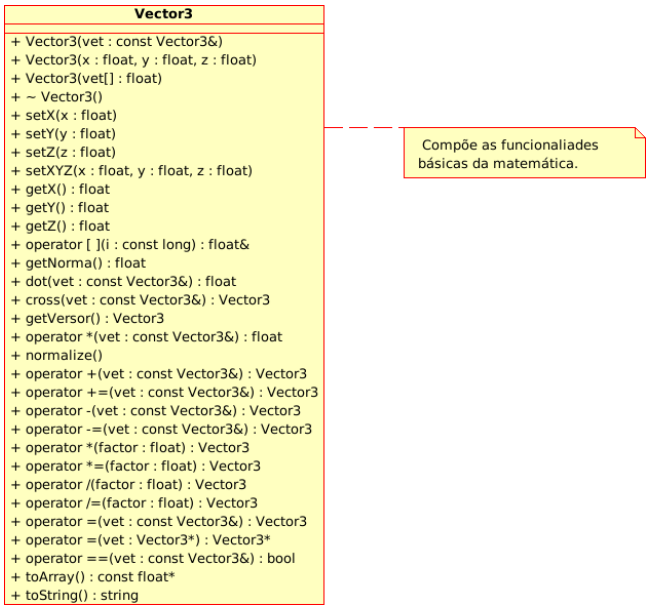

Figura 13: *Funcionalidades do Cálculo Vetorial.*

direções é possível se movimentar para desviar de um projétil a caminho. No intuito de suprir esse importante papel, está presente no pacote *Math* a classe *Vector3*.

Vector3 é uma classe que agrega grande parte das operações possíveis no cálculo vetorial para um vetor de três dimensões. Além das operações básicas de soma e adição de vetores, é permitido também multiplicação e divisão por escalar, produto vetorial, normalização, obtenção da norma e do versor. Tudo isso com o máximo de sobrecarga de operadores, tornando mais natural a utilização desse tipo.

Utilização de Math: As funcionalidades básicas do cálculo vetorial são disponibilizadas na classe Vector3, como mostrado na figura 13.

4.2.4 Máquina de Estados Finitos

A utilização de máquinas de estados finitos é a forma mais difundida de implementação de comportamentos para BOT em jogos de primeira pessoa[Schwab 2004]. Como citado anteriormente, suas desvantagens podem ser superadas pelo uso racional deste recurso. Aliado a isto, a implementenção no *framework* da estrutura para máquinas de estados é bastante flexível, explorando bem os recursos de orientação a objetos.

A classe *FSMState* é uma classe básica para qualquer estado que venha ser inserido na máquina de estado. Assim, a criação de um estado se dá através da herança de *FSMState*. Os métodos de *FSMState* são chamados pela máquina que contém a classe, de maneira que para implementar um novo estado basta codificar o que faz cada evento. É lançado um evento sempre que um estado é inicializado, quando se entra nele e também quando se sai dele. Um método de *update* é chamado sempre que uma nova entrada chega para o *FSMState*.

A classe *FSMMachine* representa a máquina de estado. Ela tem uma interface simples e é facilmente extendida para se criar um comportamento. Para a implementação de um novo comportamento basta determinar a função de transição de estados, pois todo o restante já é implementado em *FSMMachine*.

Um ponto importante da implementação de máquina de estados finitos no *framework* é que ela suporta máquina de estados hierárquica. Ou seja, uma máquina de estado pode conter estados que também são máquinas de estado. Esta abordagem reduz bastante no número de transições entre os estados e facilita o entendimento da modelagem para um comportamento complexo[dos Santos 2004].

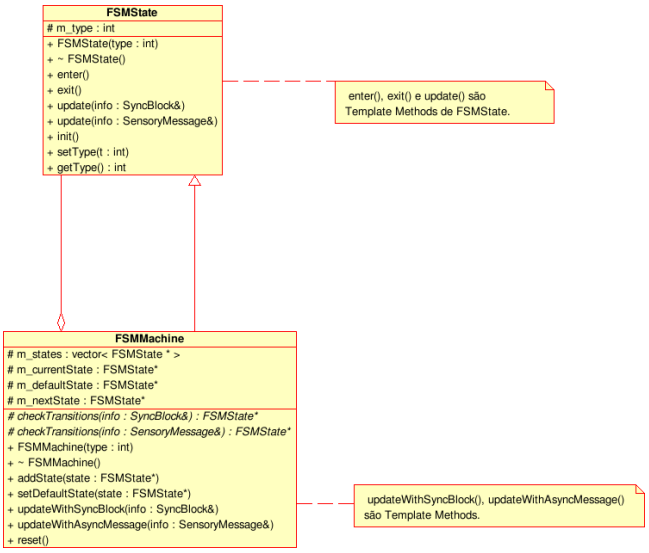

Figura 14: *Estrutura para o desenvolvimento de máquinas de estado.*

Utilização da Máquina de Estados: A Classe *FSMMachine* oference as operações definidas para funcionamento de uma máquina de estado e mantém dois *hotspots*, *updateWithSyncBlock* e *updateWithAsyncMessage*, que avaliam a transição de estados. Ela é uma especialização de *FSMState*, o que a permite atuar como estado numa máquina de estado hierárquica. A classe *FSMState* também tem o papel de Template Methods e disponibiliza os *hotspots*, *enter*, *exit* e *update*, chamados respectivamente na entrada do estado, na saída para outro estado e na atualização do estado. A Figura 14 apresenta a estrutura das máquinas de estados.

5 Estudo de Caso

Para avaliar a instanciação de BOTs utilizando o IAF , dois cenários foram montados. O primeiro envolveu o desenvolvimento de um BOT puramente reativo, ou seja, utilizando apenas um comportamento e sem fazer uso do *Short-Term Decision* e do *Long-Term Decision*. O segundo cenário foi a implementação de um agente cognitivo simples, que possuia uma meta padrão e uma representação do seu ambiente. Nestes dois casos, foi possível perceber não só a simplicidade de implementação dos BOTs, com poucos *hotspots* para serem especializados, como também a eficiência obtida pelos jogadores virtuais nos combates.

É importante caracterizar o ambiente no qual o agente é inserido. Isto porque a maneira como o ambiente é classificado influi diretamente na forma como o agente é projetado e também possibilita uma melhor compreensão das dificuldade envolvidas no processo de desenvolvimento do BOT. O jogo apresenta o ambiente com as seguintes características: **parcialmente observável**, porque o agente, através de suas percepções, não consegue acessar todos os estados relevantes ao ambiente; **estocástico**, porque o estado seguinte do ambiente não depende exclusivamente das ações do agente; **seqüencial**, porque a qualidade das ações do BOT no estado atual depende das decisões nos estados anteriores; **dinâmico**, porque entre o agente perceber e agir o ambiente já mudou de estado; **contínuo**, porque existe um número ilimitado e não enumerável[4] de como as percepções são apresentadas; **multiagente**, porque existe tanto interação cooperativa como competitiva entre os agentes[Russel and Norving 2004].

O estilo de jogo Team DeathMatch foi utilizado para simplificar

[4]Apesar de toda representação computacional ser recursivamente enumerável, a informação que este modelo computacional respresenta não é enumerável. Um exemplo é que as posições dos objetos são passadas em coordenadas cartesianas com cada componente pertencente ao conjunto dos reais. Apesar do modelo computacional suportar uma representação parcial dos reais, o conjunto dos números reais são não enumeráveis. Por isso é possível dizer que as percepções são não enumeráveis.

os testes. Neste estilo, dois grupos travam combate e vence o grupo que primeiro alcançar a pontuação de objetivo da partida. A pontuação é contada com o número de baixas do inimigo. Cada personagem possui sua pontuação própria que se soma a dos outros integrantes para formar a pontuação da equipe. A avaliação dos BOTs é baseada exclusivamente em sua pontuação dentro da partida.

Nos dois cenários foram utilizados as seguintes configurações da partida:

- **GameType:** Remote Bot Team DeathMatch
- **Bot Skill:** Skilled
- **Goal Score:** 50 no primeiro ensaio e 100 no segundo
- **Time Limit:** 40
- **Map:** DM-Rustatorium

A única diferença existente na configuração da partida para o primeiro e o segundo cenário foi a pontuação de objetivo, que no primeiro foi 50 e no segundo foi 100. A habilidade dos demais BOTs estava configurada como "Skilled"que é um nível a mais da habilidade que vem configurada como padrão. As habilidades para os BOTs do jogo disponíveis são: Novice, Average, Experienced, Skilled, Adept, Masterful, Inhuman e Godlike. O mapa escolhido foi o DM-Rustatorium por ser um mapa amplo e de fácil navegação, o que evitaria que o BOT reativo ficasse preso em um mínimo local[5] por não possuir uma representação do ambiente.

No primeiro cenário, para implementação do BOT foi utilizado apenas um dos comportamentos padrões, o de exploração. O comportamento de exploração já é definido como o inicial dentro do módulo *Behavior*, o que significa que a instanciação da classe Agent já cria um agente reativo pronto para responder aos estímulos do jogo. Este agente, por não possuir cognição associada, é considerado incompleto aos propósitos do IAF , que é o desenvolvimento de agente cognitivo. Para alcançar tal objetivo são necessárias especializações de alguns módulos da arquitetura. O agente reativo implementado, chamado de *SimpleExplorer*, apenas percorria o mapa buscando os marcos existentes no ambiente e combatendo seus adversários. Em uma partida real de jogo, a classificação final foi a seguinte:

Time A		**Time B**	
BOTs	Pontuações	BOTs	Pontuações
SimpleExplorer	**14**	Cannonball	12
Tamika	14	Kaela	10
Prism	13	Arclite	8
Jakob	9	Subversa	3
-	-	Gorge	4

A participação dos demais BOTs na partida foi aleatória. O que significa que não houve seleção do grupo que iria compor o time do *SimpleExplorer*. Mesmo com um grupo em desvantagem numérica, ele obteve um resultado muito bom comparado aos BOTs originais do jogo, superando inclusive o esperado para um BOT puramente reativo.

Já para o desenvolvimento do segundo agente, o *Explorer*, foi necessário especializar apenas uma classe, a *ShortTermDecision*, sobrescrevendo os métodos-gancho *arrivedAsyncMessage* e *arrivedSyncBlock*. Essa classe especializada passou a fazer parte da instância de Agent através de composição. Dado que os comportamentos padrões já são implementados. Neste segundo cenário, o agente já fazia uso de sua representação interna. Ele decidia se seguiria um marco, se perseguiria um personagem ou se pegaria um item ao chão. Em sua partida, o BOT teve o seguinte resultado:

[5]O agente reativo escolhe seu próximo nó avaliando apenas suas percepções, então numa tentativa de explorar o mapa, a sua escolha pode ser expressa como um função de avaliação onde sua percepção só lhe permite achar a sua melhor escolha local, se sua percepção não lhe permitir visualizar além de suas opções locais, é possível que a escolha da melhor opção se repita indefinidamente. Desta forma dizemos que o agente ficou preso em um mínimo local.

Time A		**Time B**	
BOTs	Pontuações	BOTs	Pontuações
Explorer	**46**	Ambrosia	30
Skakauk	24	Enigma	22
Guardian	20	Remus	15
Satin	10	Reinha	11
-	-	Vírus	5

Apesar do foco deste trabalho ser o desenvolvimento do *framework* e não dos BOTs, a instanciação desses BOTs ajuda a consolidar a utilidade do *framework* para a criação desses jogadores virtuais. É possível perceber com estas implementações é que a simplificação no processo de desenvolvimento não implica em uma baixa eficiência do agente. Os resultados ajudam a reforçar isso, e uma explicação plausível para tal é que é possível concentrar esforços no que o agente irá fazer ao invés de como ele irá fazer.

6 Considerações Finais

A utilização de jogos eletrônicos como laboratório para testar tanto teorias sobre o raciocínio humano quanto métodos de resolução de problemas vem crescendo ao longo do tempo. Os desafios apresentados pelos jogos eletrônicos atuais requerem um grande esforço para o desenvolvimento de personagens virtuais. Devido a isto, a produção de ferramentas que colaborem com este processo ganha cada vez mais apoio tanto do meio acadêmico quanto da indústria de jogos.

A arquitetura apresentada aqui é baseada no modelo híbrido de três camadas, o que oferece melhor aproximação com as seqüências de ações dos humanos e com o seu raciocínio. A divisão da arquitetura em sete módulos quebra o processo de desenvolvimento do agente em partes menores, simplificando o gerenciamento de cada parte e reduzindo a complexidade total do problema. Além disso, a especialização no contexto de jogos é o que permite que esta arquitetura de agente cognitivo obtenha melhores resultados em relação a outras arquiteturas híbridas.

A utilização de uma estrutura definida, aliado a uma arquitetura de agente, servindo de suporte para a organização e o desenvolvimento de BOTs, simplifica bastante a criação desses jogadores virtuais. Neste trabalho, a estrutura fornecida através do Indigente Agent Frameworkcumpre com o objetivo de simplificação do processo de desenvolvimento de agentes cognitivos para jogos. Isto porque, o IAF utiliza as vantagens apresentadas pela arquitetura de agente, disponibiliza diversas funcionalidades necessárias para a criação de BOTs, e fornece um estrutura extensível para a organização dessas funcionalidades.

A avaliação dos resultados, tanto da atuação quanto do desenvolvimento, permite observar que além de simplificar o processo de desenvolvimento de agentes, é possível obter facilmente bons resultados devido ao nível de abstração oferecido pela arquitetura. Com isto, é dado mais um passo na busca pela criação de jogadores virtuais convincentes.

6.1 Trabalhos Futuros

Apesar dos grandes avanços conseguidos com o IAF , algumas atividades ainda podem ser desenvolvidas no intuito de aperfeiçoá-lo. A implementação do suporte a utilização de linguagem de script como LUA[Ierusalimschy 2006] é uma delas. O suporte a linguagem de script permite que o *framework* seja reconfigurado sem a necessidade de recompilar seu código fonte. Esta configuração vai desde parametrização de atributos iniciais, passando por redefinição dos comportamentos, até a especialização de diversas estruturas presentes no *framework*.

Extensões da arquitetura serão os passos seguintes, adequando esta a novos contextos. Há em vista uma extensão para viabilizar o planejamento multiagente, pois hoje isto fica por conta da mesma estrutura responsável pelo planejamento local. Extensões da arquitetura refletirão em incremento do *framework*. Com isto a intenção é que o IAF passe a dá suporte ao desenvolvimento de agentes cognitivos especializados para cada novo domínio.

Referências

ADOBBATI, R., MARSHALL, A. N., SCHOLER, A., TEJADA, S., KAMINKA, G. A., SCHAFFER, S., AND SOLLITTO, C. 2001. Gamebots: a 3d virtual world test-bed for multi-agent research. *Proceedings of 2nd International Workshop on Infrastructure, MAS and MAS Scalability*.

ANDREW SCHOLER, G. K., 2000. Gamebots. Último acesso em 01 de dezembro de 2006.

DE BYL, P. B. 2004. *Programming Believable Characters for Computer Games*. Charles River Media.

DOS SANTOS, G. L. 2004. *Máquinas de Estados Hierárquicas em Jogos Eletrônicos*. Master's thesis, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

EBERLY, D. H. 2000. *3D Game Engine Design*. Morgan Kaufmann.

GRÜNVOGEL, S. M. 2003. Dynamic character animation. *International Journal of Intelligent Games and Simulation. 2*, 1, 11–19.

IERUSALIMSCHY, R. 2006. *Programming in Lua*. Lua.Org; 2nd edition.

KAMINKA, G. A., VELOSO, M. M., SCHAFFER, S., SOLLITTO, C., ADOBBATI, R., MARSHALL, A. N., SCHOLER, A., AND TEJADA, S. 2002. Gamebots: a flexible test bed for multiagent team research. *Commun. ACM 45*, 1, 43–45.

LAIRD, J. E., NEWELL, A., AND ROSENBLOOM, P. S. 1987. Soar: an architecture for general intelligence. *Artif. Intell. 33*, 1, 1–64.

MONTEIRO, I. M. 2005. Uma arquitetura modular para o desenvolvimento de agentes cognitivos em jogos de primeira e terceira pessoa. In *Anais do IV Workshop Brasileiro de Jogos e Entretenimento Digital*, 219–229.

NILSSON, N. J. 1998. *Artificial Intelligence - A New Synthesis*. Morgan Kaufmann.

RUSSEL, S., AND NORVING, P. 2004. *Inteligência Artificial - Tradução da segunda edição.* Editora Campus.

SCHWAB, B. 2004. *AI Game Engine Programming*. Charles River Media.

TOZOUR, P. 2004. Search space representations. In *AI Game Programming Wisdom 2*, Charles River Media, 85–102.

VAN WAVEREN, J. M. P. 2001. *The Quake III Arena Bot*. Master's thesis, Delft University of Technology, Delft, Netherlands.