**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO**

# SIMULADOR DE UM AMBIENTE VIRTUAL DISTRIBUÍDO MULTIUSUÁRIO PARA BATALHAS DE TANQUES 3D COM INTELIGÊNCIA BASEADA EM AGENTES BDI

**GERMANO FRONZA**

**BLUMENAU**
**2008**

**2008/1-14**

**GERMANO FRONZA**

# SIMULADOR DE UM AMBIENTE VIRTUAL DISTRIBUÍDO MULTIUSUÁRIO PARA BATALHAS DE TANQUES 3D COM INTELIGÊNCIA BASEADA EM AGENTES BDI

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciências da Computação — Bacharelado.

Prof. Dalton Solano dos Reis, M.Sc. - Orientador

**BLUMENAU**
**2008**

**2008/1-14**

# SIMULADOR DE UM AMBIENTE VIRTUAL DISTRIBUÍDO MULTIUSUÁRIO PARA BATALHAS DE TANQUES 3D COM INTELIGÊNCIA BASEADA EM AGENTES BDI

Por

**GERMANO FRONZA**

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

| | |
|---|---|
| Presidente: | Prof. Dalton Solano dos Reis, M.Sc. – Orientador, FURB |
| Membro: | Prof. Mauro Marcelo Mattos, Dr. – FURB |
| Membro: | Prof. Paulo César Rodacki Gomes, Dr. – FURB |

Blumenau, 08 de julho de 2008

Dedico este trabalho a todos meus amigos que estiveram comigo durante a realização deste, contribuindo com idéias, críticas e principalmente com grande apoio psicológico.

## AGRADECIMENTOS

Agradeço em primeiro lugar a minha família por sempre direcionar-me no caminho dos estudos, prestando muito apoio e confiança.

Ao meu orientador e amigo Dalton Solano dos Reis por toda a paciência e suporte, mostrando-se sempre motivado pela realização e conclusão do presente trabalho.

Aos meus amigos monitores e pesquisadores do campus IV, Israel D. Medeiros, Eduardo Coelho e George R. Piva, pelas idéias cedidas que foram de grande ajuda.

Aos meus amigos de classe da FURB, especialmente os que seguiram juntos desde o inicio da graduação. Não esquecendo o meu amigo Roger E. Gaulke por toda a paciência e boa vontade no auxílio dos testes do simulador.

A todos os professores e demais colaboradores da universidade que contribuíram principalmente para meu crescimento acadêmico e também pessoal.

A todos vocês, muito obrigado.

Não digo que mudarei o mundo, mas garanto que iluminarei a mente que o mudará.

Tupac Amaru Shakur

# RESUMO

Este trabalho apresenta o desenvolvimento de um simulador de batalhas de tanques de guerra concebido como um Ambiente Virtual Distribuído (AVD) em 3D executando sobre uma rede local (*Local Area Network* - LAN). Os tanques de guerra são concebidos como agentes (comportamento autônomo e cooperativo) ou como *avatares* (controlados por usuários em tempo real). Sobre a construção do AVD, aborda técnicas que visam aproveitar de forma mais otimizada os recursos de rede, que são: largura de banda, latência e confiabilidade. Este trabalho também demonstra como a utilização do interpretador Jason facilita a programação de agentes BDI baseando-se na linguagem AgentSpeak(L). Por fim, para a construção do mundo virtual do simulador, apresenta a utilização da *engine* gráfica JMonkey Engine e o *framework* JME Physics.

Palavras-chave: Simulador. Ambiente virtual distribuído. Agentes. Arquitetura BDI. AgentSpeak(L).

# ABSTRACT

This work describes the development of a tank battle simulator conceived as a Networked Virtual Environment (NVE) in 3D running over Local Area Network (LAN). The tanks are conceived as agents (standalone and cooperative behavior) or as avatars (controled by user in real-time). About the NVE construction, it approaches techniques that try to optimize the use of network resources, such as: bandwith, latency and reliability. This work also presents how the use of Jason interpreter enables BDI agents programming based in AgentSpeak(L) language. Finally, for construction the virtual world of the simulator, demonstrates the use of graphics engine JMonkey Engine and the framework JME Physics.

Key-words: Simulator. Networked virtual environment. Agents. BDI architecture. AgentSpeak(L).

# LISTA DE ILUSTRAÇÕES

# SUMÁRIO

# 1 INTRODUÇÃO

Com o progresso das técnicas de computação, principalmente na área gráfica e de inteligência artificial, tem sido possível a construção de ferramentas que simulam determinadas atividades do mundo real. Tais simuladores possuem a responsabilidade de produzirem cenários sempre mais convincentes, para que possam ser extraídos resultados e utilizá-los, se necessário, em atividades reais.

No início, devido a complexidade e falta de recursos, as interfaces eram criadas em modelos 2D. Porém, com o avanço dos computadores e das técnicas de computação gráfica já é possível a criação de ambientes 3D que representam o mundo real. Entre essas aplicações 3D encontram-se os simuladores de ambientes virtuais.

Um ambiente virtual é um cenário 3D gerado por computador, através de técnicas de computação gráfica, usado para representar um mundo real ou fictício (PINHO et al., 1999). Estes cenários possuem a característica de se modificarem durante sua execução conforme seus integrantes interagem com o ambiente.

Uma das classificações dos ambientes virtuais vem da necessidade de suportar um ou mais usuários e permitir que estes estejam em computadores distintos. Quando surge a necessidade de um ambiente comportar vários usuários, este é denominado de Ambiente Virtual Distribuído (AVD).

Segundo Singhal e Zyda (1999, p. 2), um AVD é um sistema computacional onde vários usuários interagem entre si em tempo real, mesmo que estes usuários estejam em lugares remotos. Tipicamente, cada usuário acessa o ambiente virtual através de uma interface usando o seu próprio computador.

Conforme a popularização das *engines* gráficas[1] *freeware* e *open-source* desenvolvidas para linguagens atuais, como Java e .NET, o alto nível de abstração tem contribuído para a criação, com mais facilidade, de *Rich Interface Applications* (RIA), em português, Aplicações com Interfaces Ricas (AIR).

Entretanto, para que uma aplicação possa simular o mundo real, não é suficiente ter apenas gráficos de qualidade, é necessário também que os personagens que o compõem ajam de forma natural, pretendendo ser cada vez mais convincentes.

Neste ponto, algumas áreas da inteligência artificial podem ser muito úteis. Uma delas

[1] *Engine* gráfica é usada no texto para descrever um *middleware* para construção de aplicações gráficas 3D.

é a área de Sistemas MultiAgentes (SMA) baseados na arquitetura *Beliefs-Desires-Intentions* (BDI). Tal arquitetura provê formas de programação de agentes cujas características muito se assemelham às dos seres humanos, ou seja, baseados em crenças, desejos e intenções.

Atualmente, nas aulas de inteligência artificial das universidades, quando estudam-se assuntos referentes a SMAs, tem-se muitas dificuldades na criação de exemplos que envolvam uma grande variedade de práticas destes sistemas. Isso faz com que sejam especificados exemplos menores e pouco motivadores para aplicar tais técnicas. Todavia, acostumados com o paradigma de Programação Orientada a Objetos (POO), muitos alunos deixam de gostar da área de Programação Orientada a Agentes (POA), justo porque não vêem a aplicabilidade desta em sistemas reais, que por sua vez são mais complexos.

Concentrando-se nesses fatores, procurou-se desenvolver um simulador de um ambiente virtual distribuído em 3D, para que estudantes da área de inteligência artificial possam programar agentes (que representem os tanques de guerra no cenário) na linguagem AgentSpeak(L) do interpretador Jason. Para participar da batalha, os estudantes devem formar times de tanques, ou seja, desenvolver agentes que cooperem entre si para atingirem o objetivo de vencer a batalha.

O simulador, além de permitir o controle dos tanques de guerra através de agentes (comportamento autônomo e cooperativo), permite também que os tanques sejam controlados por usuários (através de *avatares*) durante a execução da batalha, usando teclado e mouse, conforme exemplifica a Figura 1. Com isso, os estudantes podem formar times de tanques *avatares*, controlando-os em tempo real para batalhar contra seus times programados na forma de agentes, podendo assim verificar o quão eficiente os mesmos podem ser.

Neste trabalho, desenvolveu-se um AVD com uma arquitetura híbrida (com características de arquitetura centralizada combinadas com arquitetura distribuída), composto por um servidor e vários clientes. O aplicativo servidor é responsável por interceptar determinadas mensagens enviadas pelos clientes e manipular o corrente estado do AVD. O aplicativo cliente é responsável por exibir a cena 3D, baseado em eventos do usuário, estado do AVD armazenado localmente e mensagens recebidas de outros usuários. Além disso, dependendo do tipo do cliente, pode servir de estação de entrada de comandos de controle do tanque, ou servir para executar agentes dentro do ambiente Jason.

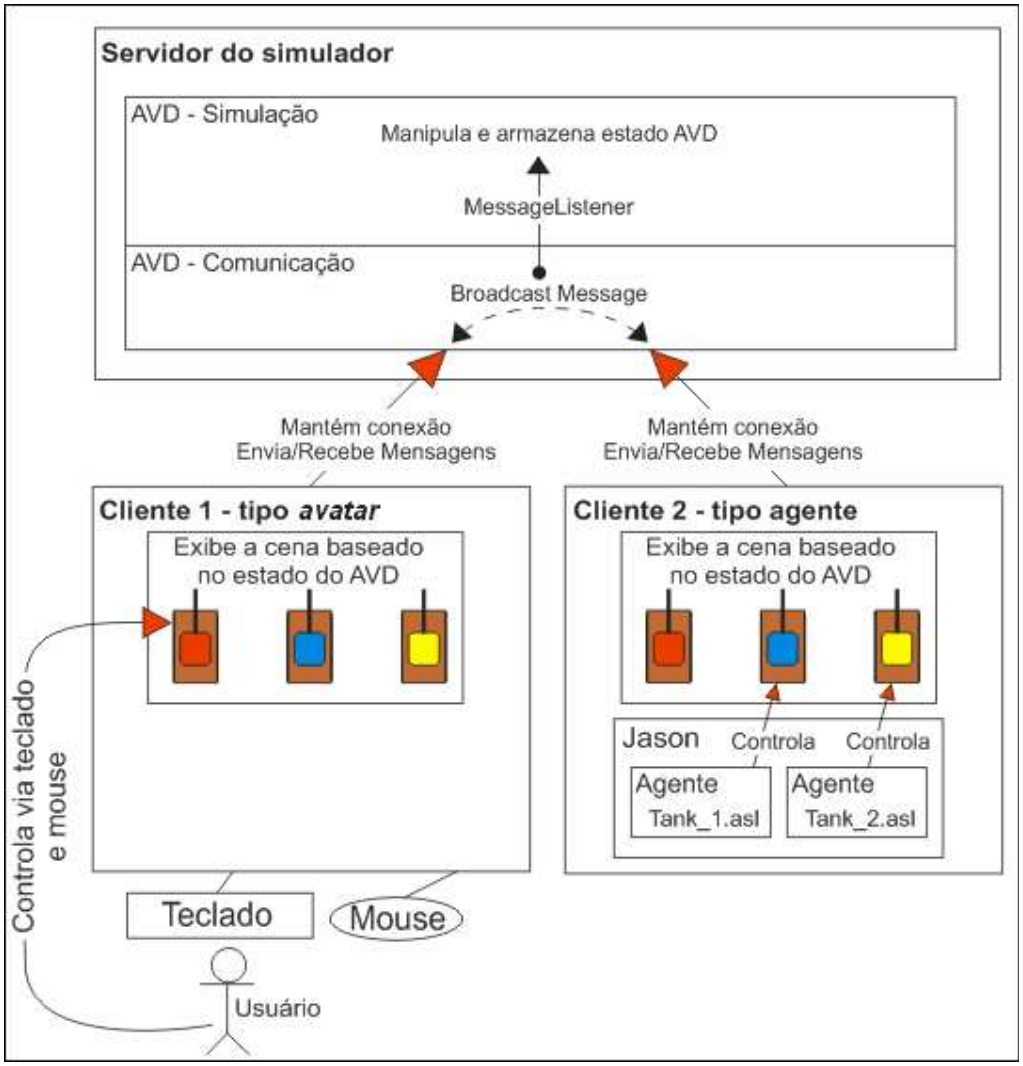



Figura 1 – Exemplo de dois clientes (tipo agente e tipo *avatar*) conectado ao servidor

## 1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um simulador 3D de um AVD sobre uma rede local, com suporte a multiusuários, para batalhas de times de tanques de guerra autônomos ou controlados por *avatares*.

Os objetivos específicos do trabalho são:

a) definir um cenário a ser utilizado no simulador (ambiente, objetos, percepções e comportamentos);

b) implementar um simulador utilizando uma *engine* gráfica (no caso, JMonkey Engine) para construção do ambiente gráfico do AVD (3D);

c) permitir que os tanques sejam controlados por usuários (*avatares*) ou tenham um

comportamento autônomo (agentes BDI).

## 1.2 ESTRUTURA DO TRABALHO

O capítulo 2 faz a fundamentação teórica, apresentando os tópicos mais importantes para o desenvolvimento do presente trabalho.

A seção 2.1 concentra-se em apresentar o conceito de AVD, bem como os primeiros projetos desenvolvidos nessa área. A seção 2.1.1 apresenta e contextualiza os dois modelos gerais de comunicação de um AVD. As seções 2.1.2, 2.1.3 e 2.1.4 tratam respectivamente de largura de banda, latência e confiabilidade, que são os principais aspectos a serem considerados na construção de um AVD. A seção seguinte (2.1.5) demonstra as principais técnicas usadas em AVDs, evidenciando as vantagens e desvantagens de cada uma delas. Na seção 2.1.6 é demonstrado o funcionamento do *middleware* de comunicação Java Game Networking (JGN), utilizado para notificar os usuários acerca de alterações no estado do AVD.

Na seção 2.2 são apresentados os conceitos de SMA e os principais aspectos de cada tipo de agente. A seção 2.2.1 aborda as características de coordenação entre os agentes de um SMA como uma vantagem para alcançar um objetivo em comum. Na seção seguinte (2.2.2) são apresentadas as linguagens de comunicação de agentes como meio de utilizar conceitos de coordenação. Na seção 2.2.3 são mostrados os conceitos da arquitetura BDI, bem como as principais características para o desenvolvimento de agentes cognitivos. A seção 2.2.4 apresenta a linguagem AgentSpeak(L) para programação de agentes baseados no modelo BDI. Na seção 2.2.5 é vista a ferramenta Jason, como um interpretador da linguagem AgentSpeak(L).

A seção 2.3 apresenta as principais características da *engine* gráfica JMonkey Engine (JME) e do *framework* JME Physics. Na seção 2.3.1 são apresentadas as principais classes e interfaces para a criação de uma aplicação JME.

Na seção 2.4 são relatados os trabalhos correlatos e as principais características que se relacionam com o presente trabalho.

O capítulo 3 apresenta a especificação e implementação do simulador.

No capítulo 4 são apresentadas as conclusões deste trabalho bem como possíveis extensões em futuros trabalhos.

# 2 FUNDAMENTAÇÃO TEÓRICA

Para a realização deste trabalho, foi feita uma revisão bibliográfica dos temas que são abordados nas seções seguintes.

## 2.1 AMBIENTE VIRTUAL DISTRIBUÍDO

Da mesma maneira que os ambientes virtuais podem ser aplicados nas diversas áreas (jogos, simulações, ciência, etc.), a forma de construção também pode variar conforme a necessidade e recursos disponíveis. Uma classificação referente à construção de tais ambientes deve-se à quantidade de usuários a serem suportados, podendo ser então monousuários ou multiusuários compartilhando o mesmo ambiente através de computadores interligados.

Segundo Singhal e Zyda (1999, p. 19), os primeiros esforços realizados para a construção de AVDs partiram de iniciativas militares, com os projetos *SIMulator NETwork* (SIMNET) e *Distributed Interactive Simulation* (DIS). Outros esforços surgiram no meio acadêmico, sendo o mais famoso deles o NPSNET[2].

Dentre os projetos anteriormente citados, o SIMNET destaca-se pelo fato de ter sido o primeiro AVD de larga escala desenvolvido para o Departamento de Defesa dos Estados Unidos. Trata-se de um simulador de batalhas militares envolvendo tanques, aviões e infantaria, para ser utilizado pelo exército dos Estados Unidos. Singhal e Zyda (1999, p. 20, tradução nossa) afirmam que “A meta do projeto SIMNET era desenvolver um AVD de baixo custo para treinar pequenas unidades (tanques M1, helicópteros AH-64, entre outros) para batalhar como um time”. Maiores informações podem ser encontradas em Shaw e Green (1993).

Os resultados extraídos desses projetos são hoje utilizados como fonte de pesquisa para os principais aspectos a serem observados no desenvolvimento de um AVD. Macedonia e Zyda (1997) afirmam que esses aspectos envolvem, desde a escolha da arquitetura geral (centralizada ou distribuída), protocolos de comunicação mais indicados para serem

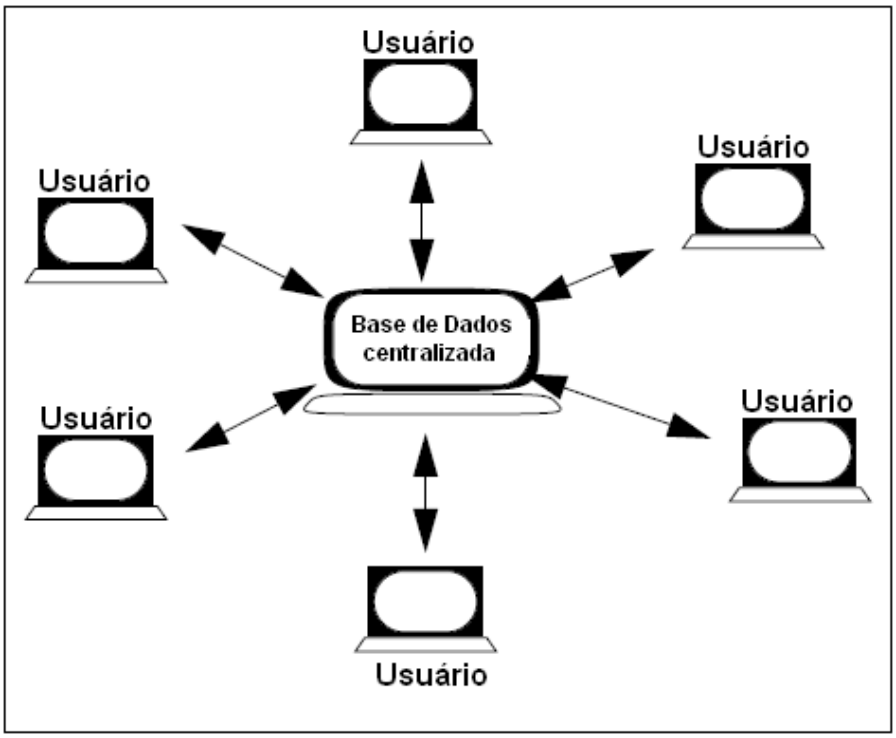
[2] Naval Postgraduate School NETwork (NPSNET) também é um AVD voltado para batalhas militares.

utilizados, até as questões mais relacionadas à infra-estrutura de rede disponível para a interconexão dos computadores, como, utilização de largura de banda, latência e confiabilidade. Nas seções seguintes esses conceitos são abordados em detalhes.

### 2.1.1 Modelos gerais de comunicação

Gossweiler et al. (1994) afirma que os dois métodos mais populares para a construção de um AVD são o modelo centralizado e o modelo distribuído.

Em um modelo centralizado, um computador central é responsável por receber todos os dados enviados pelos usuários conectados no AVD, armazenar as mudanças em uma estrutura de dados (em memória ou em uma base de dados centralizada) e então devolver os resultados das alterações para cada usuário conectado, para que os mesmos possam redesenhar a cena atualizada. A Figura 2 demonstra graficamente o funcionamento desta arquitetura.




Fonte: adaptado de Gossweiler et al. (1994).

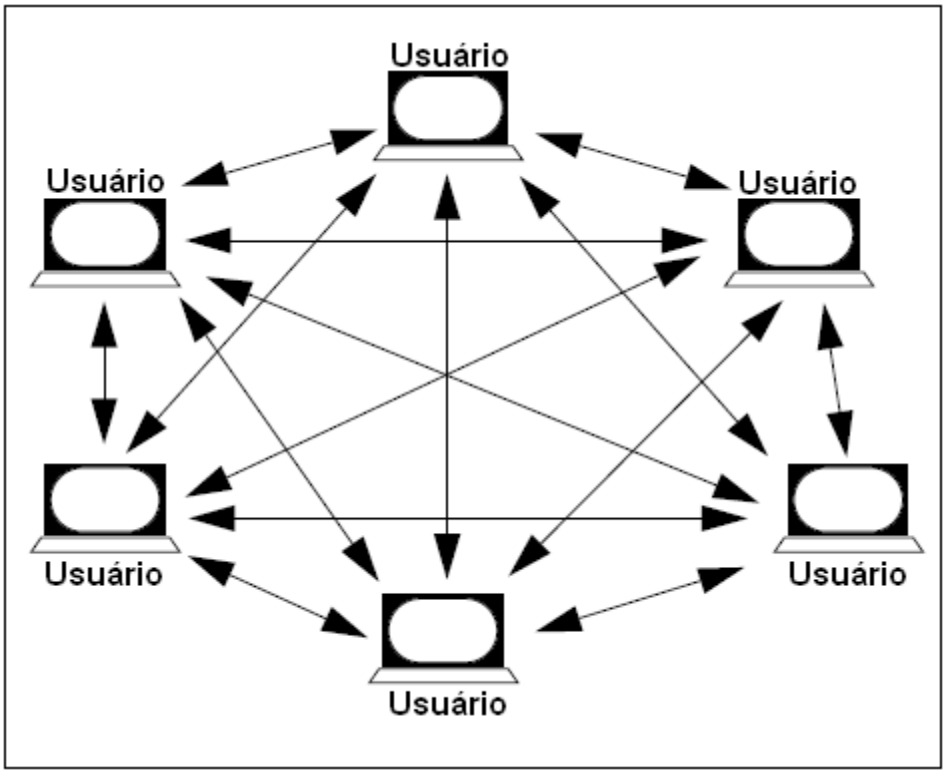
Figura 2 – Modelo de comunicação centralizado

Gossweiler et al. (1994, tradução nossa) comenta também que “Apesar deste modelo permitir ao programador desenvolver uma simples estrutura de dados para armazenar e manipular os dados, ela não é escalável.” A escalabilidade neste caso está diretamente ligada à capacidade do AVD de se manter estável conforme a quantidade de usuários conectados

cresce. Quanto mais usuários conectados ao AVD, maior é o tráfego de mensagens que o computador central tem de receber, processar e devolver, fazendo com que a velocidade de acesso diminua consideravelmente.

Como alternativa ao problema apresentado no modelo centralizado, uma abordagem mais escalável surge com o modelo distribuído. Nesse modelo, cada usuário mantém uma própria cópia local do estado do AVD, realizando ele mesmo, as tarefas de renderização, computação e animação dos objetos da cena. Dessa maneira, quando um usuário realiza alguma modificação no estado do AVD local, ele se encarrega de notificar todos os demais usuários acerca desta modificação, para que os mesmos apliquem as atualizações localmente.

A Figura 3 apresenta graficamente a interação entre os usuários em um modelo de comunicação distribuído.




Fonte: adaptado de Gossweiler et al. (1994).

Figura 3 – Modelo de comunicação distribuído

Ao mesmo tempo em que este modelo acaba com o problema de escalabilidade sobre um computador central, como é o caso do modelo centralizado, um novo problema surge, porém agora em cada usuário do AVD. Nesse modelo, o aplicativo do usuário, além de ter todo o processamento para renderizar a cena, precisa também gerenciar N-1 conexões (cada conexão com um canal de comunicação independente), onde N é o número de usuários presentes no AVD. Além das conexões, quando um usuário realiza alguma alteração no estado do AVD, o mesmo é responsável por notificar todos os demais usuários conectados a ele, gerando assim um grande número de mensagens a serem enviadas.

Com o intuito de amenizar os problemas apresentados em ambos os modelos de comunicação, o estudo dos três seguintes fatores deve ser considerado: utilização de largura de banda, latência e confiabilidade de que as mensagens são recebidas corretamente por todos os usuários do AVD.

### 2.1.2 Largura de banda

Segundo Singhal e Zyda (1999, p. 59), largura de banda é uma taxa, normalmente expressa em *Bits* Por Segundo (BPS), que relaciona a capacidade máxima de se transmitir uma determinada quantidade de informações com o tempo de entrega da mesma até o destino.

A largura de banda é normalmente o primeiro aspecto a ser considerado na construção de um AVD. É essa taxa que determina o tamanho e a riqueza de detalhes do ambiente virtual. Dentre todos os fatores na qual a largura de banda impacta, o que mais chama atenção é em relação ao número de usuários suportados pelo AVD. Dessa forma, conforme o aumento do número de usuários, maior é o tráfego na rede gerado por conta das mensagens de atualização.

Macedonia e Zyda (1997) afirmam que em redes locais (*Local Area Networks* – LANs), a largura de banda não costuma ser um ponto crítico do AVD, pelo fato da maioria operar acima dos 10 Mbps e o número de usuários ser geralmente baixo. Porém, quando o AVD opera sobre uma rede de longa distância (*Wide Area Network* – WAN), a realidade é bem diferente.

No caso do AVD utilizar uma rede de longa distância para interligar os usuários do ambiente virtual, como a internet, por exemplo, a largura de banda nem sempre é suficiente, considerando que, atualmente, em média estas operam próximo aos 1.5 Mbps e o número de usuários conectados podendo ser bem maior.

Em ambos os modelos de comunicação apresentados nas seções anteriores, a largura de banda é um aspecto importante a ser considerado. Porém quando se trata do modelo distribuído, pelo fato de existir um grande número de conexões entre os usuários e também um grande número de mensagens geradas a cada atualização, percebe-se que uma enorme largura de banda é requerida, mas que nem sempre está disponível.

### 2.1.3 Latência

Singhal e Zyda (1999, p. 56) definem latência, dentro do prisma de redes de

computadores, como a quantidade de tempo, normalmente medida em milissegundos, necessária para transmitir um pacote de um ponto a outro. Quando um AVD gera um pacote a ser transmitido, é a latência da rede que determina em qual momento a aplicação no outro ponto irá receber os dados para serem processados.

Singhal e Zyda (1999, p. 57) também comentam que a latência representa um grande desafio para os projetistas de AVDs, baseado nas seguintes razões: impacta diretamente sobre o realismo do ambiente virtual e porque os projetistas pouco podem fazer para reduzir este tempo de atraso.

Para que o cenário do AVD consiga atingir um elevado nível de imersão junto ao usuário, é preciso que a entrega de pacotes de atualização tenha uma latência mínima para garantir que as imagens 3D do ambiente virtual sejam enviadas aos usuários em uma freqüência que garanta a ilusão de realidade. Essa freqüência, Macedonia e Zyda (1997) afirmam ser entre 30 e 60 *Frames* Por Segundo (FPS), aproximadamente.

Da mesma forma que ocorre com a largura de banda, em LANs a latência não apresenta problemas significativos, já que o atraso costuma ser inferior a 5 ms. O grande problema é quando se trata de redes WANs. Quando o AVD opera sobre esse tipo de rede, é comum ter-se latência acima de 50 ms, pois cada pacote precisa percorrer um longo caminho, passando por diversos equipamentos de interconexão de rede, até chegar ao destino.

Cecin e Trinta (2007) afirmam que para AVDs na área de jogos no estilo de estratégia, a latência média tolerada é de até 500 ms. Porém em jogos de primeira pessoa, a latência deve se manter abaixo dos 100 ms para garantir um nível de ilusão de realidade aceitável.

### 2.1.4 Confiabilidade

A confiabilidade é o aspecto relacionado à garantia que um AVD tem de que todas as mensagens de atualização no ambiente enviadas pelos usuários sejam recebidas no destino de forma integra.

O problema é que qualquer rede de interconexão de computadores pode apresentar diversos tipos problemas durante a sua utilização. Um dos principais problemas a ser considerado é em relação à mídia usada na rede, ou seja, se é cabo par trançado, fibra ótica ou até mesmo *WI-FI*[3]. Essas questões físicas da rede estão sempre sujeitas à falhas de

[3] WI-FI trata-se de uma rede de computadores sem a utilização de fios, também chamada de rede 802.11.

interferências, tornando-a muito instável (SINGHAL; ZYDA, 1999, p. 59).

Para que um AVD garanta o aspecto de confiabilidade, considerando os possíveis problemas de rede, significa fazer uso de protocolos de comunicação orientados a conexão, como é o caso do protocolo *Transport Control Protocol* (TCP), que utilizam mecanismos de confirmação de recebimento (comando *acknowledgment*) e recuperação quando da ocorrência de erros ou completo não recebimento de um pacote.

Mais uma vez, no caso do AVD operar sobre uma rede WAN, o uso de protocolos como o TCP, apesar de em um primeiro momento parecerem recursos ideais para comunicação, podem não ser os mais aconselháveis, pois os mecanismos mencionados anteriormente combinados com outros que controlam o fluxo de pacotes causam um considerável atraso na comunicação entre os usuários.

Macedonia e Zyda (1997) afirmam que a confiabilidade provida pelo protocolo TCP afeta diretamente o desempenho de um AVD em tempo real por causa do tratamento de retransmições, sendo que em simulações em tempo real, é impossível voltar no tempo. Quando um pacote se perde, ou seja, não chega ao usuário destinatário, a entidade que enviou precisa obter uma cópia do pacote enviado e retransmiti-lo. Principalmente em redes WANs, o tempo entre a primeira tentativa de envio e o recebimento com sucesso do pacote no destinatário pode ser tão grande que no final a informação pode não ter mais um valor semântico para o ambiente virtual daquele usuário.

Em contrapartida ao uso do protocolo TCP, torna-se possível também o uso de protocolos não orientados a conexão, como o *User Datagram Protocol* (UDP). Esse protocolo, por não ser orientado a conexão, não possui nenhum mecanismo de garantia de entrega de pacotes, porém logicamente provendo maior desempenho na comunicação.

É importante considerar que nenhum dos dois protocolos citados apresenta-se como solução ideal para todos os casos. É necessário se ter em mente o quão importante é o aspecto de confiabilidade em um determinado AVD e quando que o não recebimento de alguns pacotes não impacta diretamente no normal funcionamento do mesmo. Na seção seguinte são apresentadas técnicas que visam contornar a perda de pacotes ou pelo menos diminuir o seu impacto sobre o ambiente virtual.

### 2.1.5 Principais técnicas usadas em AVDs

Até o momento foram vistos os diferentes modelos de comunicação de um AVD, bem

como os principais aspectos a serem considerados. Tendo em vista melhorar o desempenho e reduzir os problemas até agora notados, as próximas seções apresentam as principais técnicas criadas para esta finalidade.

#### 2.1.5.1 Formas básicas de comunicação entre usuários de um AVD

Segundo Kurose e Ross (2005, p. 301) existem três formas básicas de comunicação em redes de computadores e que também podem ser aplicadas na construção de AVDs:

a) *unicast*: nessa forma, um pacote sempre tem um único destinatário. Sendo assim quando um usuário faz alguma alteração no AVD, ele precisa enviar uma mensagem para cada um dos demais usuários do ambiente virtual. Esse tipo de comunicação também é conhecido por ponto-a-ponto;
b) *broadcast*: nessa forma, a camada de rede provê um serviço de entrega de pacotes enviado de um nó fonte à todos os outros nós da rede. Sendo assim quando o usuário efetua alguma alteração no AVD, basta ele submeter uma única mensagem à rede que todos os outros usuários recebem normalmente;
c) *multicast*: nessa forma, a camada de rede provê um serviço de entrega de pacotes enviado de um nó fonte à um determinado subgrupo de nós da rede. Sendo assim quando o usuário efetua uma alteração do AVD, basta ele submeter uma única mensagem para o subgrupo de rede na qual ele está contido.

Macedonia e Zyda (1997) afirmam que cada uma das formas de comunicação possui vantagens e desvantagens, sendo que essas estão diretamente relacionadas ao número de usuários suportados pelo AVD.

A forma de comunicação *unicast*, apesar de ser, em um primeiro momento, mais fácil de implementar (fato é visto como uma vantagem), pode se tornar um método impraticável quando muitos usuários estão conectados ao AVD, tanto se tratando de um modelo de comunicação centralizado, como de um modelo distribuído. Pelo fato de que é necessário ter um canal de comunicação diferente para cada usuário conectado, quando ocorre uma mudança no ambiente virtual, é gerado uma nova mensagem para cada usuário. Nesse momento, não é difícil perceber que nesse método, recursos de hardware são desperdiçados, afetando principalmente a já apresentada largura de banda (SINGHAL; ZYDA, 1999, p. 69).

Com a intenção de amenizar o problema do alto número de conexões e mensagens submetidas à rede, tem-se a forma de comunicação *broadcast*. Nesse método, basta que uma

única mensagem seja enviada para o endereço de *broadcast* da rede na qual os usuários estão conectados, que estes “ouvem” a mensagem como se tivesse sido enviada diretamente a eles.

A principal desvantagem do *broadcast*, é que esse método de comunicação geralmente faz uso de UDP *broadcasting*, onde as mensagens são enviadas para o endereço de *broadcast* da rede local utilizando o protocolo UDP. Sendo assim, as mensagens ficam restritas à rede local. Para AVDs que operam sobre redes WANs, essa forma não pode ser utilizada. Nesse caso, tem-se como opção o uso de *multicast*. Outro problema notado com o uso de *broadcast* é que, como toda mensagem é enviada a todos os usuários do AVD, pode que algum usuário receba e processe alguma mensagem que não lhe interessa, logo um processamento extra é gerado (*overhead*), atrapalhando o processamento de mensagens que realmente interessam (SINGHAL; ZYDA, 1999, p. 70).

A terceira e ultima forma de comunicação é o *multicast*, também identificado pelo nome *IP multicast*. Nesse método, uma vantagem evidente, se dá pelo fato de uma mensagem poder ser entregue a um seleto grupo de usuários do AVD. Essa característica é importante, pois soluciona um dos problemas que o *broadcast* apresenta, que é o de gerar *overhead* por conta de mensagens que não interessam determinados usuários. Uma desvantagem do método IP *multicast* é que nem todos os roteadores existentes no mercado são compatíveis com tal técnica, logo não pode ser usada incondicionalmente na internet.

A Figura 4 demonstra graficamente as três formas de comunicação citados.

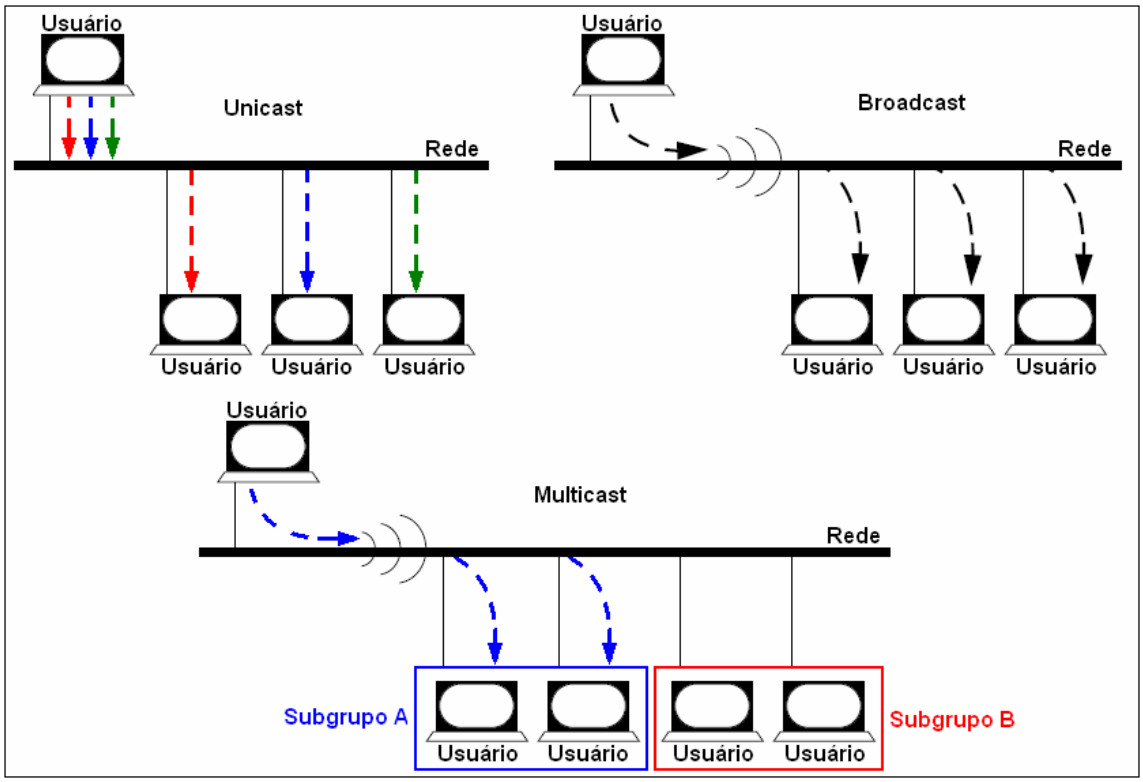



Figura 4 – Envio de mensagens com *unicast*, *broadcast* e *multicast*

#### *2.1.5.2 Dead Reckoning*

A técnica de *Dead Reckoning*, na construção de AVDs, é utilizada para reduzir significativamente o número de mensagens de atualização de estado do AVD geradas por cada usuário. Segundo Ferreira (1999), sem o uso desse tipo de técnica, um usuário do AVD precisa, a cada atualização de *frame*, enviar mensagens aos demais informando completamente todos os atributos de cada entidade que ele controla no ambiente virtual. Em um jogo distribuído de primeira pessoa, por exemplo, quando um usuário movimenta seu personagem, a cada nova posição ocupada por ele no cenário, uma nova mensagem é enviada aos demais usuários. Em um AVD onde dezenas de usuários constantemente movimentam suas entidades no cenário do ambiente virtual, fica fácil perceber que o tráfego de dados na rede torna-se muito intenso.

A idéia central da técnica de *Dead Reckoning* é transmitir mensagens apenas quando ocorrem determinadas atualizações no estado corrente de uma entidade (FERREIRA, 1999). Entre as atualizações, cada *host* prevê os atributos da entidade baseando-se nos dados recebidos anteriormente e armazenados localmente. Quando uma nova mensagem chega, o *host* então atualiza os dados da entidade. Se houverem divergências entre o novo estado e o estado previsto, é feita uma convergência entre eles para corrigir a inconsistência.

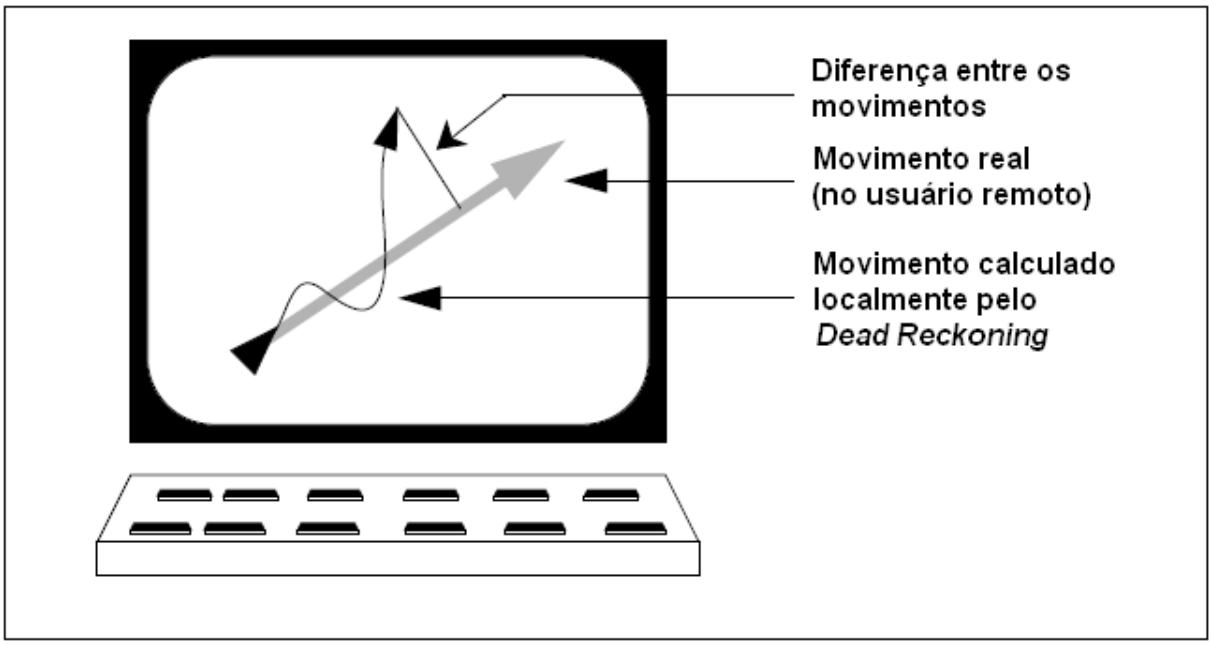
A técnica pode ser dividida em duas etapas: predição e convergência. A fase de predição refere-se ao modo como o estado corrente da entidade é calculado baseado em mensagens previamente recebidas. A fase de convergência refere-se ao ato de corrigir os valores dos atributos da entidade previamente estimados, quando uma nova mensagem é recebida.

A etapa de predição, normalmente é baseada em conceitos da matemática e física, tal como o Movimento Uniformemente Variado (MUV). Com esse conceito, por exemplo, é possível prever a nova posição de um personagem no cenário tendo como informação a direção e velocidade do mesmo. Quando um personagem se move no ambiente virtual, basta o usuário enviar a direção e velocidade, que a técnica de *Dead Reckoning* nos usuários remotos se encarrega de prever a movimentação localmente.

Além da técnica de *Dead Reckoning* ser executada nos usuários que recebem a mensagem, é também executada no usuário que enviou a mensagem. Com isso é possível saber se a movimentação real do personagem está de acordo com a movimentação calculada pela técnica. Caso a divergência entre os movimentos ultrapassar um valor aceitável, definido

pelo projetista do AVD, então uma nova mensagem é enviada aos demais, para que a técnica atualize a real posição do personagem.

A Figura 5 ilustra um exemplo da divergência gerada entre o movimento real de um personagem no mundo virtual (personagem controlado por um usuário remoto do AVD) e o movimento calculado pela técnica de *Dead Reckoning*.




Fonte: adaptado de Gossweiler et al. (1994).

Figura 5 – Divergência entre o movimento real e o movimento calculado pela técnica de *Dead Reckoning*

Como também pode ser visto na Figura 5, a técnica de *Dead Reckoning* sacrifica a consistência do estado do AVD, em função da redução do tráfego de mensagens e conseqüente aumento da taxa de FPS, suportando AVDs com mais participantes.

#### 2.1.5.3 *Heartbeats*

Segundo Gossweiler et al. (1994), uma forma de amenizar os problemas apresentados por um sistema de comunicação não confiável e para notificar rapidamente o corrente estado do AVD aos novos usuários, cada usuário já conectado, periodicamente (a cada cinco segundos, por exemplo), envia uma mensagem chamada *heartbeat*, informando os demais a respeito do corrente estado das entidades controladas por ele. Com isso, quando um usuário conecta-se ao AVD, facilmente ele pode criar sua própria representação do ambiente virtual.

Essa técnica, além de contribuir com os novos usuários que conectam-se ao AVD, também auxilia no processo de atualização por *Dead Reckoning*. Por exemplo, caso aconteça

de um usuário não receber alguma mensagem anterior a respeito da movimentação de um determinado personagem, nos próximos ciclos ele recebe um novo *heartbeat*, podendo então atualizar a posição do mesmo.

### 2.1.6 Java Game Networking (JGN)

Até o momento foram vistos conceitos e técnicas envolvendo a construção de AVDs. Porém, a implementação desse tipo de ambiente não está associado a uma tecnologia ou linguagem específica, ou seja, qualquer uma que contenha os recursos básicos necessários pode ser utilizada. A tecnologia Java apresenta-se como uma solução *freeware* para desenvolvimento de aplicações, atendendo desde dispositivos móveis, até sistemas corporativos de alto nível (JAVA EVERYWHERE, 2008). Além de ser uma tecnologia extremamente abrangente, possui uma grande comunidade de desenvolvedores, fazendo com que muitas extensões (bibliotecas, *frameworks* ou *middlewares*) sejam criadas, normalmente utilizando licenças *open-source*.

Para a construção de aplicações distribuídas, como é o caso dos AVDs, o *kit* de desenvolvimento nativo do Java provê um pacote específico chamado `java.net`. Nesse pacote, estão incluídas principalmente classes de baixo nível, ou seja, as que utilizam recursos mais próximos do sistema operacional. É importante ter em mente que as classes desse pacote não implementam, de fato, a comunicação entre diferentes máquinas, elas somente delegam esse serviço para o sistema operacional em questão. É justamente esse fato, que permite ao Java ser considerada uma tecnologia multiplataforma, ou seja, ser executada em diferentes sistemas operacionais e diferentes dispositivos. A Quadro 1 apresenta as principais classes do pacote `java.net`.

| **CLASSE** | **PROTOCOLO** | **UTILIDADE** |
|---|---|---|
| `Socket` | TCP | Conectar-se a um *host* remoto |
| `ServerSocket` | TCP | Aceitar conexões de *sockets* clientes |
| `DatagramSocket` | UDP | Enviar e receber pacotes UDP |
| `MulticastSocket` | UDP | Lidar com grupos *multicast* |

Fonte: adaptado de Java Plataform SE 6 (2007).

Quadro 1 – Principais classes de baixo nível do pacote `java.net`

Utilizando as classes de *sockets* do Java, para que uma mensagem seja enviada através da rede, a aplicação deve primeiro converter os dados em um *array* de *bytes*, para que então esse possa ser “escrito” no canal de comunicação e de fato enviado. Esse processo de converter os dados da mensagem em um *array* de *bytes*, muitas vezes, deixa o processo de

envio de mensagens, por parte da aplicação, extremamente trabalhoso e sem nenhum grau de abstração.

Com a intenção de prover maior abstração à aplicação em nível de envio de mensagens e tratamentos de conectividade em geral, surge a necessidade da utilização de um *middleware* entre a camada da aplicação e a camada baixo nível de *sockets*, como mostra a Figura 6.

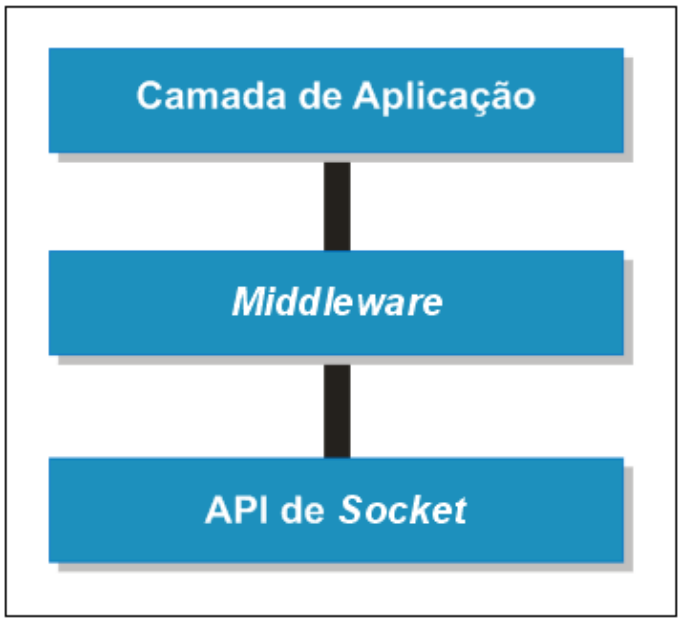



Figura 6 – Organização da aplicação utilizando um middleware de comunicação

Dentre vários *middlewares* de comunicação disponíveis para a construção de aplicações distribuídas, especialmente na subárea de jogos multiusuários, tem-se a Java Game Networking (JGN). Trata-se de um *middleware freeware* e *open-source*, distribuído sob a licença *Berkeley Software Distribution* (BSD), utilizado para escrever jogos com arquitetura cliente-servidor.

A JGN tem como principal objetivo encapsular as operações sobre *sockets*, deixando a camada de aplicação livre de operações baixo nível. Utilizando esse *middleware*, as mensagens são concebidas como objetos. Isso faz com que, quando a aplicação necessita enviar uma mensagem através da rede, basta instanciar um objeto que represente a mensagem, povoar seus atributos e por fim submeter ao *middleware*. Este por sua vez serializa o objeto e envia através do *socket*.

Para atender os diferentes ambientes de rede os jogos multiusuários podem executar, a JGN possibilita tanto o uso de *socket* TCP quanto UDP. Possibilita ainda utilizar os dois tipos de *sockets* e a decisão de quando utilizar um ou outro fica por conta das mensagens que são enviadas. Se uma classe de mensagem herdar da classe `CertifiedMessage` e o estiver usando um *socket* TCP, então essa mensagem é enviada através desse canal, caso contrário é enviada pelo *socket* UDP.

No Quadro 2 são apresentadas as principais classes e interfaces da JGN, bem como as funcionalidades de cada uma delas.

| **CLASSE/INTERFACE** | **UTILIDADE** |
|---|---|
| `JGNServer` | Implementar o servidor do jogo |
| `JGNClient` | Implementar um cliente do jogo |
| `JGN` | Registrar as classes de mensagens e também iniciar as *threads* do servidor e dos clientes |
| `JGNConnectionListener` | Detectar eventos de conexão e desconexão de um cliente ou do servidor |
| `MessageListener` | Detectar evento de mensagens |
| `Message` | Classe base para mensagens a serem enviadas através do *middleware* |
| `PlayerMessage` | Classes de mensagens que permitem acessar o Id do *player* e do destinatário |
| `SynchronizationManager` | Gerenciar as sincronizações dos objetos do cenário (no aplicativo cliente) |
| `GraphicalController` | Interligar o módulo de comunicação com a *engine* gráfica utilizada para *renderizar* a cena do jogo |
| `SyncObjectManager` | Implementações dessa interface são utilizadas pelo SynchronizationManager para criação/deleção de objetos da cena do jogo |

Quadro 2 – Relação das principais classes e interfaces da JGN

Para a implementação do aplicativo servidor de um jogo multiusuário, as principais classes utilizadas são: `JGNServer`, `JGN`, `JGNConnectionListener`, `MessageListener` e `Message`. O Quadro 3 demonstra, passo a passo, como é desenvolvido um servidor com JGN utilizando essas classes.

Já para a implementação do aplicativo cliente, as principais classes utilizadas são: `JGNClient`, `JGN`, `MessageListener` e `Message`. O Quadro 4 demonstra, passo a passo, como é desenvolvido um cliente com JGN utilizando essas classes.

```
import java.net.InetAddress;
import java.net.InetSocketAddress;

import com.captiveimagination.jgn.JGN;
import com.captiveimagination.jgn.clientserver.JGNConnection;
import com.captiveimagination.jgn.clientserver.JGNConnectionListener;
import com.captiveimagination.jgn.clientserver.JGNServer;
import com.captiveimagination.jgn.event.MessageListener;
import com.captiveimagination.jgn.message.Message;

public class Servidor {
    /** Construtor da classe Servidor */
    public Servidor() {
        // obtém uma referência para o endereço de rede local
        InetAddress local = InetAddress.getLocalHost();
        // cria um endereço de socket para o servidor confiável (TCP)
        InetSocketAddress endTCP = new InetSocketAddress(local, 10);
        // cria um endereço de socket para o servidor rápido (UDP)
        InetSocketAddress endUDP = new InetSocketAddress(local, 11);
        // cria um servidor JGN para o jogo
        JGNServer servidor = new JGNServer(endTCP, endUDP);
        // cria um listener para “escutar” eventos de conexão
        servidor.addClientConnectionListener(
            new JGNConnectionListener() {
                public void connected(JGNConnection con) {
                    // executado quando um usuário conecta- se
                }

                public void disconnected(JGNConnection con) {
                    // executado quando um usuário desconecta-se
                }
            }
        );
        // cria um listener para “escutar” eventos de mensagem
        servidor.addMessageListener(
            new MessageListener() {
                public void messageCertified(Message message) {
                    // executado quando uma mensagem é recebida com
                    // sucesso no destinatário
                }
                public void messageFailed(Message message) {
                    // executado quando uma mensagem não é entregue com
                    // sucesso no destinatário
                }
                public void messageSent(Message message) {
                    // executado quando uma mensagem é enviada
                }
                public void messageReceived(Message message) {
                    // executado quando uma mensagem é recebida
                }
            }
        }
        // cria uma nova thread para o servidor ser executado
        JGN.createThread(servidor).start();
    }
}
```

Quadro 3 – Implementação de um simples servidor utilizando JGN

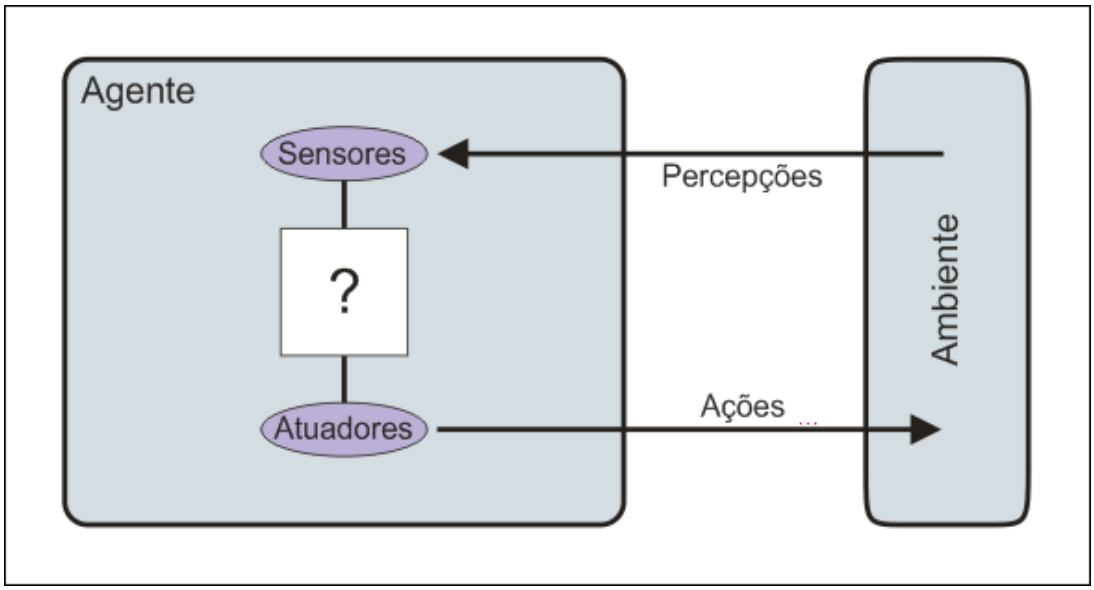
```
import java.net.InetAddress;
import java.net.InetSocketAddress;

import java.lang.InterruptedException;

import com.captiveimagination.jgn.JGN;
import com.captiveimagination.jgn.clientserver.JGNConnection;
import com.captiveimagination.jgn.clientserver.JGNClient;
import com.captiveimagination.jgn.event.MessageListener;
import com.captiveimagination.jgn.message.Message;

public class Cliente {
    /** Construtor da classe Cliente */
    public Cliente () {
        // obtém uma referência para o endereço de rede do servidor
        InetAddress endServidor = InetAddress.getByName("192.168.0.1");
        // cria um endereço para acessar o socket TCP do servidor
        InetSocketAddress endTCP = new InetSocketAddress(endServidor, 10);
        // cria um endereço para acessar o socket UDP do servidor
        InetSocketAddress endUDP = new InetSocketAddress(endServidor, 11);

        // obtém uma referência para o endereço de rede local pro cliente
        InetAddress local = InetAddress.getLocalHost();
        // cria um endereço de socket para o cliente confiável (TCP)
        InetSocketAddress cliEndTCP = new InetSocketAddress(local, 100);
        // cria um endereço de socket para o cliente rápido (UDP)
        InetSocketAddress cliEndUDP = new InetSocketAddress(local, 101);
        // cria um cliente JGN
        JGNClient cliente = new JGNClient (cliEndTCP, cliEndUDP);
        // cria um listener para "escutar" eventos de mensagem
        cliente.addMessageListener(
            new MessageListener() {
                public void messageCertified(Message message) {
                    // executado quando uma mensagem é recebida com
                    // sucesso no destinatário
                }
                public void messageFailed(Message message) {
                    // executado quando uma mensagem não é entregue com
                    // sucesso no destinatário
                }
                public void messageSent(Message message) {
                    // executado quando uma mensagem é enviada
                }
                public void messageReceived(Message message) {
                    // executado quando uma mensagem é recebida
                }
            }
        }
        // cria uma nova thread para o cliente ser executado
        JGN.createThread(cliente).start();
        try {
             // conecta-se com o servidor esperando no máximo 5 segundos
             cliente.connectAndWait(endTCP, endUDP, 5000);
        } catch (InterruptedException ie) {
             // entra nesse bloco caso ocorra timeout na conexão.
        }
    }
}
```

Quadro 4 – Implementação de um simples cliente utilizando JGN

## 2.2 SISTEMAS MULTIAGENTES (SMA)

"Os sistemas multiagentes são sistemas compostos por múltiplos elementos computacionais interativos, conhecidos como agentes." (WOOLDRIDGE, 2002, tradução nossa). Seguindo essa definição, é possível inferir que um SMA é composto por vários agentes autônomos que cooperam entre si para atingir um objetivo comum.

Um agente, segundo Alvares e Sichman (1997), é um elemento computacional com comportamento autônomo, contido em um ambiente real ou virtual, podendo perceber, agir e se comunicar com outros agentes. A Figura 7 ilustra, de forma abstrata, como um agente está inserido no ambiente.




Fonte: adaptado de Russell e Norvig (2004, p. 34).

Figura 7 – Agentes interagindo com o ambiente através de sensores e atuadores

Como pode ser visto na Figura 7, agentes são capazes tanto de perceber mudanças que ocorrem no seu ambiente, como também atuar sobre ele baseado em um repertório de ações pré-determinadas (BORDINI; HÜBNER; WOOLDRIDGE, 2007, p. 2). Sensores são como os sentidos, por exemplo, um agente humano pode conter olhos, ouvidos, nariz e outros sentidos. Um agente robótico pode ter câmeras, detectores infravermelhos e outros dispositivos capazes de perceber o ambiente. Os atuadores também são como os sentidos, porém utilizados para agir sobre o ambiente, por exemplo, um agente robótico pode conter rodas, braços mecânicos, esteiras e outros dispositivos capazes de atuar sobre o mundo em que estão contidos.

O ambiente que os agentes ocupam pode ser virtual ou real. Ambientes virtuais são concebidos como softwares, escritos em qualquer linguagem de programação. Esse tipo de ambiente pode contemplar desde cenários do tipo *grid*, até cenários mais complexos em 3D,

como ilustra a Figura 8.

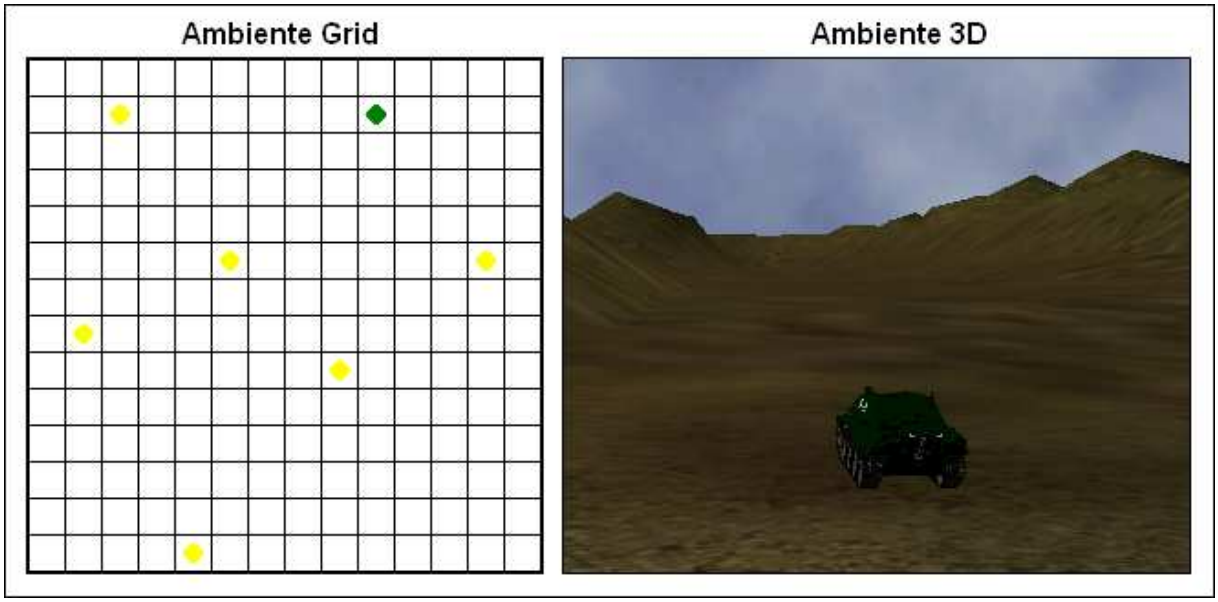



Figura 8 – Ambiente tipo *grid* e tipo 3D

Existe também a possibilidade de um SMA utilizar um ambiente real. Nesse caso, os agentes comumente controlam robôs ou outro tipo de hardware. Um exemplo real de uso desse tipo de SMA pode ser encontrado no RoboCup. O RoboCup consiste em um projeto de inteligência artificial e robótica, onde robôs inteligentes formam times de futebol para competir em uma liga. Mais informações sobre esse projeto podem ser vistas em RoboCup (2008).

Além dos agentes fazerem parte de um ambiente, Wooldridge e Jennings (1995 apud BORDINI; HÜBNER; WOOLDRIDGE, 2007, p. 3) afirmam que os agentes também podem possuir as seguintes propriedades:

a) autonomia: um agente é considerado autônomo quando ele é capaz de agir baseado somente no conhecimento que ele próprio adquire durante sua vivência no ambiente. Russell e Norvig (2004, p. 38) dizem também que, na prática, raramente um agente é completamente autônomo, principalmente no início. É comum o projetista do SMA, prover ao agente um certo conhecimento inicial, para que a partir desse o agente possa seguir por si só;
b) pró-atividade: um agente é considerado pró-ativo quando ele exibe um comportamento dirigido a metas. Dessa forma, um agente não age simplesmente em resposta às percepções do ambiente, mas sim de acordo com um propósito que ele almeja;
c) reatividade: um agente é considerado reativo quando ele possui um comportamento estímulo-resposta, ou seja, age simplesmente baseado em

percepções do ambiente ou mensagens de outros agentes;

d) habilidade social: um agente possui essa propriedade quando ele apresenta habilidade de coordenar e cooperar atividades com outros agentes, tendo como intenção, atingir suas metas. Bordini, Hübner e Wooldridge (2007, p. 5) afirmam ainda que, para um agente ter essas habilidades, é imprescindível que eles contem com recursos de comunicação de alto nível, para que possam trocar informações em nível de metas, crenças e planos.

Com base nessas propriedades, é possível classificar os agentes em reativos e cognitivos. Os agentes reativos, como o próprio nome sugere, possui somente a propriedade de reatividade. Possuem comportamentos simples, não tendo nenhum modelo a respeito do mundo onde estão contidos. Os agentes cognitivos, por sua vez, possuem comportamentos complexos onde deliberam e negociam ações, metas, e demais informações com outros agentes. Ao projetar esse tipo de agente, é comum utilizar-se conceitos de coordenação de ações e metas. A seção seguinte aborda esse assunto com mais detalhes.

### 2.2.1 Coordenação

Segundo Jennings (1996, p. 187), coordenação, no contexto de SMA, é o processo no qual um agente raciocina sobre suas ações locais e ações de outros agentes, a fim de garantir que a comunidade funcione de maneira coerente. Trata-se de trabalho em equipe, onde os agentes agem para atingir um objetivo em comum.

Jennings (1996, p. 188) afirma ainda que existem três principais razões para que os agentes de uma comunidade coordenem suas ações. Sendo elas:

a) existem dependências entre as ações dos agentes. Essa interdependência ocorre quando as metas a serem alcançadas por cada agente individualmente estão relacionadas. Também porque decisões locais feitas por um agente têm um impacto sobre as decisões dos outros membros da comunidade. Por exemplo, um grupo de robôs móveis construindo uma casa, decisões sobre tamanho e localização das salas impactam também no projeto de fiação e encanamento, por isso as decisões precisam ser tomadas em conjunto;

b) existe a necessidade de cumprir restrições globais. Essas restrições existem quando a solução que está sendo desenvolvida por um grupo de agentes precisa satisfazer certas condições. Por exemplo, um time de robôs construindo uma casa tendo

disponível um orçamento de, no máximo, cento e cinqüenta mil reais;

c) individualmente não se tem competência, recursos e informações suficientes para resolver o problema por completo. Por exemplo, ao carregar um objeto muito pesado ou tocar uma sinfonia. Somente coordenando as ações o objetivo final é atingido com sucesso.

Na prática, uma das formas dos agentes coordenarem suas ações é fazendo uso de algum protocolo e uma linguagem de comunicação. A seção seguinte apresenta duas linguagens de comunicação de agentes consideradas linguagens em “nível de conhecimento” (do inglês, *knowlegde-level communication*).

### 2.2.2 Linguagens para comunicação de agentes: KQML e FIPA

A linguagem *Knowledge Query and Manipulation Language* (KQML) foi a primeira tentativa de definir uma linguagem prática e de alto nível para comunicação de agentes de um SMA. KQML é uma linguagem de troca de mensagens essencialmente em nível de conhecimento. Quando um agente deseja enviar uma mensagem a outro, ele simplesmente expressa o conhecimento em uma mensagem, bem diferente de linguagens de comunicação baseada em invocação de métodos (BORDINI; HÜBNER; WOOLDRIDGE, 2007, p. 26).

A linguagem KQML possui um conjunto de primitivas na qual são usadas para determinar explicitamente a intenção do agente enviando a mensagem. Por exemplo, a primitiva `tell` é usada com a intenção de alterar as crenças do agente receptor. Já a primitiva `achieve` é usada com a intenção de alterar as metas do receptor. Existe ainda a primitiva `ask-one` utilizada para fazer perguntas a outros agentes.

Segundo Bordini, Hübner e Wooldridge (2007, p. 27), a *Foundation for Intelligent Physical Agents* (FIPA) possui um componente chamado FIPA *suit*, liberado em 2002. Trata-se de um componente que contempla uma coleção de padrões relacionados à comunicação de agentes. Na verdade, esses padrões são customizações de primitivas e outras questões semânticas da linguagem KQML, com a intenção de deixar a comunicação entre os agentes mais simples e abstrata.

### 2.2.3 Arquitetura BDI

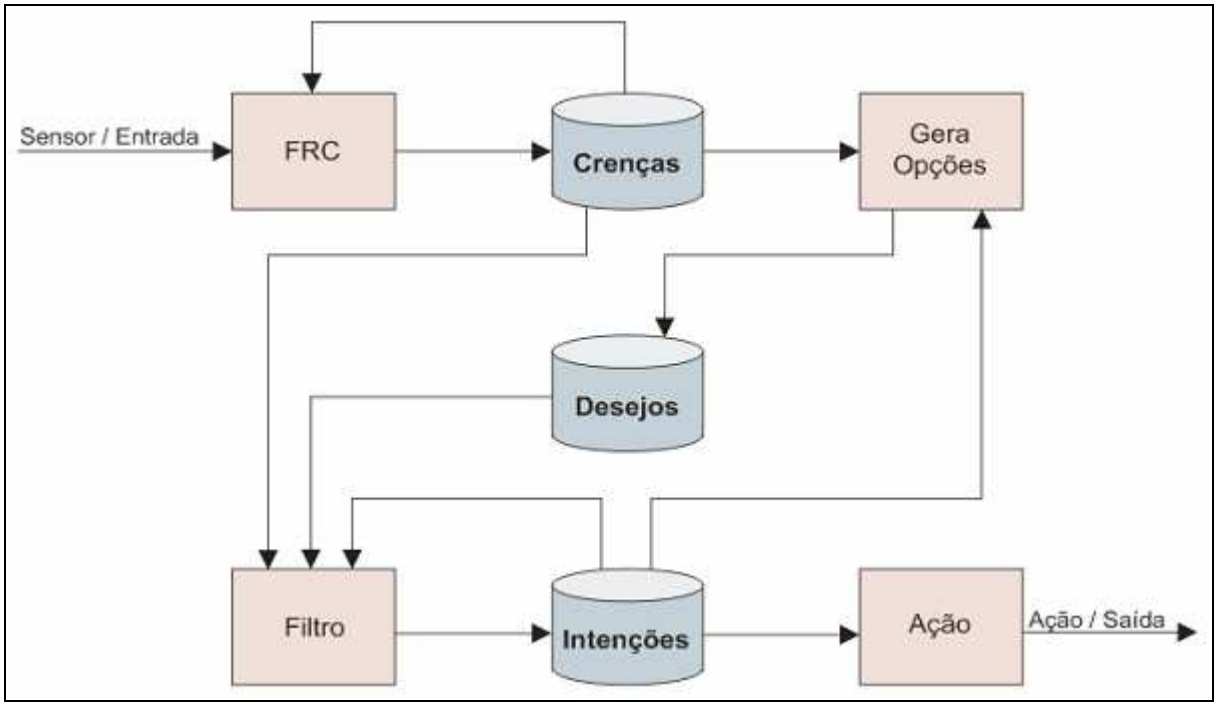
Existem diversas abordagens que propõem diferentes tipos de atitudes mentais e seus relacionamentos. Entre elas, destaca-se a arquitetura BDI originalmente proposta por Bratman em 1987 como uma teoria filosófica do raciocínio prático. Esse modelo, baseado em comportamentos humanos, é caracterizado por três atitudes mentais, que são as crenças, os desejos e as intenções (BORDINI; HÜBNER; WOOLDRIDGE, 2007, p. 15).

As crenças, contidas na memória do agente, representam tudo o que ele conhece a respeito do ambiente, dos demais agentes e de si mesmo. Bordini, Hübner e Wooldridge (2007, p. 15), para facilitar o entendimento desse conceito, fazem uma analogia de crenças com variáveis em linguagens de programação convencionais.

Os desejos representam todos os estados possíveis que o agente quer alcançar. Bordini, Hübner e Wooldridge (2007, p. 16) afirmam ainda que ter um desejo, entretanto, não significa que o agente passa a agir diretamente para alcançá-lo. Desejos são vistos somente como opções para os agentes.

As intenções representam todo o conjunto de ações que o agente deve realizar para atingir sua meta. A atualização do conjunto de intenções do agente é baseada nas suas crenças e nos seus desejos.

A Figura 9 demonstra graficamente o fluxo de uma arquitetura BDI genérica.




Fonte: adaptado de Wooldridge (1999).

Figura 9 – Fluxo de uma arquitetura BDI genérica

A Função de Revisão de Crenças (`FRC`), através dos sensores de entrada, recebe as informações do ambiente, podendo ler e se necessário atualizar a base de crenças. A função `Gera Opções` verifica quais estados devem ser atingidos, baseando-se no estado atual e nas intenções que o agente já está comprometido. A função `Filtro` atualiza o conjunto de intenções do agente com base nas crenças e nos desejos que ele possui. Por último, a função `Ação` representa a escolha de uma determinada ação para ser executada sobre o ambiente.

### 2.2.4 Linguagem AgentSpeak(L)

Até o momento foram vistos os principais conceitos relacionados à SMA e também um modelo genérico da arquitetura BDI. Porém, ao pensar em como projetar um SMA é necessário ter em mente qual linguagem utilizar para escrever, de fato, o código-fonte dos agentes. Dentre as linguagens utilizadas para programação de agentes cognitivos, destaca-se a AgentSpeak(L), inicialmente proposta por Rao em 1996.

Segundo Bordini e Vieira (2003), a linguagem AgentSpeak(L) é uma extensão natural e alto nível de programação em lógica para a arquitetura de agentes BDI. Representa um modelo abstrato para a programação de agentes e tem sido o modelo mais utilizado para a implementação de agentes inteligentes (também chamados de agentes “racionais”). Um agente especificado através da linguagem AgentSpeak(L) é composto por um conjunto de crenças iniciais, metas e planos.

O Quadro 5 apresenta um trecho de código-fonte de um simples agente especificado na linguagem AgentSpeak(L). Cada item desse código-fonte é explorado em detalhes nas seções seguintes.

```
1
2   /** Crenças iniciais */
3   likes(orquestraSinfonica).
4   likes(robertoCarlos).
5   likes(coralDeNatal).
6
7   /** Planos */
8   +concert(A, V) : likes(A)
9        ← !book_tickets(A, V).
10
11  +!book_tickets(A, V) : ¬busy(phone)
12       ← call(V);
13         ...;
14         !choose_seats(A, V).
15
```

Quadro 5 – Exemplo de agente AgentSpeak(L)

#### 2.2.4.1 Crenças

Conforme Bordini, Hübner e Wooldridge (2007, p. 32), a primeira questão que deve ser compreendida sobre a linguagem AgentSpeak(L) é como representar as crenças dos agentes. Como já foi mencionado na seção anterior, um agente possui uma base de crenças (*belief base*) que consiste em uma coleção de literais, como na tradicional programação de lógica.

Em linguagens de programação inspiradas em lógica (como o Prolog), as informações são representadas por predicados, por exemplo, `tall(john)`. Esse predicado expressa que certo indivíduo (nesse caso "John", que é expresso pelo termo `john`) é alto (*tall*). Para representar relacionamentos entre dois ou mais objetos, pode-se utilizar um predicado como `likes(john, music)`. Nesse exemplo, claramente percebe-se que está sendo dito que John gosta de música.

No Quadro 5 as linhas 3, 4 e 5 representam declarações de crenças iniciais do agente. Está sendo dito que o agente gosta de Orquestra Sinfônica, Roberto Carlos e do Coral de Natal.

#### 2.2.4.2 Metas

Segundo Bordini, Hübner e Wooldridge (2007, p. 40), na programação de agentes racionais, a noção de meta é fundamental. A linguagem AgentSpeak(L) distingue dois tipos de metas: metas de realização (*achievement goals*) e metas de teste (*test goals*). Metas de realização e teste são predicados, tais como crenças, porém com operadores prefixados "!" ou "?", respectivamente. Uma meta de realização expressa o que o agente deseja alcançar e uma meta de teste retorna a unificação do predicado de teste com uma crença do agente, ou falha caso não seja possível unificar com nenhuma crença (BORDINI; VIEIRA, 2003).

No Quadro 5, o predicado `!book_tickets(A, V)` (linha 9) trata-se da execução de uma meta de realização. Significa que o agente tem como objetivo reservar ingressos para assistir o concerto do artista A no local V (do inglês *venue*). O Quadro 5 não apresenta nenhum exemplo de meta de teste, porém um uso desse tipo de meta pode ser visto com o predicado `?likes(A)`. Nesse caso, será unificado na variável `A` o valor da primeira crença `likes`, que no código-fonte do Quadro 5 é o termo `orquestraSinfonica` (linha 3).

Crenças e metas são duas atitudes mentais importantes que podem ser expressas no código-fonte de um agente. Outra construção essencial de um agente BDI são os planos. Segundo Bordini, Hübner e Wooldridge (2007, p. 41), eles representam o *know-how* do agente.

#### 2.2.4.3 Planos

Em AgentSpeak(L) um plano é composto por um evento de disparo (*triggering event*), contexto do plano (*context*) e um corpo (*body*). O evento de disparo e o contexto do plano, juntos, são chamados de cabeça do plano (*head of the plan*). As três partes do plano são sintaticamente separadas por "`:`" e "`<-`", como mostra o Quadro 6.

```
triggering_event : context <- body.
```

Fonte: Bordini, Hübner e Wooldridge (2007, p. 42).

Quadro 6 – Sintaxe de um plano AgentSpeak(L)

Evento de disparo é o nome de identificação do plano. É através desse predicado que um plano é selecionado para ser executado. O contexto do plano é utilizado para testar se o plano é aplicável na hora que o agente seleciona o evento para ser executado. Ou seja, se o contexto resultar no valor booleano *false*, o corpo do plano não é executado (sendo considerado não-aplicável). É possível ter mais de um plano com o mesmo nome de evento de disparo, porém as condições do contexto do plano devem ser diferentes, caso contrário o agente não tem como filtrar o plano correto a ser executado.

O corpo do plano é uma seqüência de ações básicas, objetivos ou atualizações de crenças. As ações básicas que os agentes podem executar podem ser classificadas como:

a) ações internas: são ações prefixadas por "`.`" e não alteram o ambiente. Por exemplo: `.somar(X, Y, Resultado)`; essa ação efetua a soma dos dois primeiros argumentos e unifica o resultado na variável `Resultado`.
b) ações externas: são ações que alteram o ambiente. Por exemplo: `mover(1, 2)`; essa ação move o personagem para a posição x e y passada como argumento.

A execução de um plano pode originar a partir de um evento interno ou externo. Eventos internos correspondem à alteração das crenças ou metas (+b, -b, +!g, -!g, +?g, -?g) do agente. Eventos externos correspondem à percepção do ambiente ou ao receber uma mensagem de outro agente.

No Quadro 5, as linhas 8 e 11 correspondem a planos do agente. O primeiro deles (`concert`) corresponde a um plano que é invocado a partir de um evento externo, mais especificamente de uma percepção. Para o corpo desse plano ser executado, a condição do contexto precisa ser verdadeira (valor booleano *true*). Nesse caso, o agente precisa conter a crença `likes(A)`, sendo que `A` é uma variável já preenchida no evento de disparo do plano (`concert(A, V)`). O plano da linha 11 é invocado a partir de um evento interno, mais especificamente de uma meta de realização. Essa meta, no Quadro 5, é executada na linha 9 como sendo um subplano do plano `concert`.

### 2.2.5 Interpretador Jason

A ferramenta Jason, desenvolvida por Hübner e Bordini, compreende um interpretador de uma versão estendida da linguagem AgentSpeak(L). Sendo desenvolvida sobre a plataforma Java, possui todos os benefícios de rodar sobre a Java Virtual Machine (JVM), sendo inclusive multiplataforma (BORDINI; HÜBNER, 2008). Outra importante característica do interpretador Jason em comparação com outras ferramentas para programação de agentes BDI, é que ele é *open-source*, sendo distribuído sob a licença GNU LGPL[4].

Segundo Bordini e Hübner (2008), além de possuir as características comuns da linguagem AgentSpeak(L) original, o interpretador Jason também contempla os seguintes recursos:

a) negação forte (*strong negation*), portanto é possível construir sistemas que consideram mundo-fechado (*closed-world*) e mundo-aberto (*open-world*);
b) tratamento de falhas de planos;
c) comunicação inter-agentes baseada em atos de fala (*speech-act*);
d) anotações em identificadores de planos, que podem ser utilizados na elaboração de funções personalizadas para seleção de planos;
e) suporte para desenvolvimento de ambientes (que normalmente não é programado em AgentSpeak(L));
f) possibilidade de executar o SMA distribuidamente em uma rede (usando o SACI[5]);

[4] Mais informações sobre essa licença de software podem ser obtidas em: http://www.gnu.org/licenses/lgpl.html

[5] Disponível em http://www.lti.pcs.usp.br/saci/

g) possibilidade de especificar (em Java) funções de seleção de planos, as funções de confiança e toda a arquitetura do agente (percepção, revisão de crenças, comunicação e atuação);

h) possui uma biblioteca de ações internas básicas;

i) possibilita a extensão da biblioteca de ações internas.

A configuração do SMA, em um projeto Jason, é feita através de um arquivo de texto com extensão ".mas2j". Nesse arquivo, várias parametrizações são possíveis, porém as principais a serem citadas são a infra-estrutura do sistema, o ambiente onde os agentes estão situados e os agentes, de fato. Até o presente momento, o Jason contempla as infra-estruturas "Centralised", "Saci" e "Jade", sendo que as duas últimas servem para SMA distribuídos. O Quadro 7 apresenta um exemplo de arquivo de configuração de projeto Jason.

```
MAS robos_aspiradores {
    infrastructure: Centralised
    environment: ambiente.Casa
    agents:
        bob;
        maria;
}
```

Quadro 7 – Arquivo de configuração de projeto Jason

A programação dos agentes AgentSpeak(L) é feita em arquivos com extensão ".asl" (AgentSpeak *Language*). No exemplo do Quadro 7, o projeto contém os agentes bob e maria, logo o diretório desse projeto Jason deve conter os arquivos "bob.asl" e "maria.asl".

A implementação do ambiente do SMA é feita na linguagem Java. O parâmetro *environment* do arquivo ".mas2j" deve conter o nome completo da classe[6] que implementa esse ambiente. Na prática, essa classe deve herdar de `jason.environment.Environment` e opcionalmente sobrescrever seus métodos. O Quadro 8 exemplifica a implementação de um ambiente escrito na linguagem Java.

Atualmente, para a criação de um projeto SMA no Jason, existem duas possibilidades. A primeira delas (a mais utilizada pela grande maioria de usuários do Jason) é utilizando o *plug-in* para a ferramenta jEdit[7]. A outra possibilidade, que recentemente foi criada pelo mesmo autor desse trabalho, é utilizando o *plug-in* para o ambiente de desenvolvimento Eclipse. Ambas as ferramentas de apoio provem facilidades para criação e manutenção de projetos de SMA Jason.

---

[6] Nome completo da classe subentende-se pela combinação do nome do pacote e o nome da classe.

[7] jEdit é um editor de textos open-source baseado em plug-ins para ser utilizado por programadores.

```
package ambiente;

import jason.environment.Environment;
import jason.asSyntax.Structure;
import jason.asSyntax.Literal;
import java.util.List;

public class Casa extends Environment {

    @Override
    public void init(String[] args) {
        // aqui podem ser feitas inicializações do ambiente.
    }

    @Override
    public void executeAction(String agName, Structure actionTerm) {
        // esse método é executado toda vez que um agente solicitar a
        // execução de um ação externa.
        // normalmente aqui é feito um “chaveamento” para descobrir qual
        // ação deve ser executada. Para obter o nome da ação, o código é:
        // String actionName = actionTerm.getFunctor();
    }

    @Override
    public List<Literal> getPercepts(String agName) {
        // esse método é executado toda vez que o interpretador Jason
        // for notificar as percepções de um agente.
        // essa é uma forma de enviar percepções aos agentes, a outra
        // forma é utilizando um dos métodos:
        //   - addPercept(String agName, Literal per);
        //   - addPercept(Literal per);
    }
}
```

Quadro 8 – Ambiente de um SMA programado em Java

### 2.2.5.1 Componentes personalizados pelo usuário

A ferramenta Jason é distribuída com um conjunto básico de funcionalidades, como algumas citadas nas seções anteriores. Entretanto, em alguns casos os programadores precisam de componentes com comportamentos diferentes dos já implementados por padrão dentro da ferramenta. As seções subseqüentes apresentam os principais componentes que podem ser personalizados e como isso é feito.

#### 2.2.5.1.1 Ações internas

Ações internas criadas por usuários devem ser organizadas em bibliotecas específicas. Em AgentSpeak(L), uma ação interna é acessada pelo nome da biblioteca, seguida por “.”, seguida pelo nome da ação (BORDINI; HÜBNER; WOOLDRIDGE, 2007, p. 140). Por

exemplo, uma ação interna chamada `distance` situada em uma biblioteca chamada `math`, pode ser usada tanto no contexto como no corpo de um plano de um agente. O Quadro 9 mostra um trecho de código AgentSpeak(L) utilizando uma ação interna.

```
+event : true <- math.distance(2, 3, 5, 5, D); ...
+event : math.distance(2, 3, 5, 5, D) & D > 30 <- ...
```

Fonte: adaptado de Bordini, Hübner e Wooldridge (2007, p. 140).

Quadro 9 – Utilização de ações internas em um plano AgentSpeak(L)

As bibliotecas são definidas por pacotes Java e cada ação interna dela é uma classe Java. Um detalhe importante a respeito das classes de ações internas no Jason, é que essas devem começar com uma letra minúscula, diferente do padrão de nomenclatura padrão das classes Java.

Todas as classes de ações internas devem implementar a interface InternalAction. No Jason existe também uma implementação padrão dessa interface, chamada DefaultInternalAction, que simplifica a criação de novas ações internas por parte do usuário. O Quadro 10 demonstra a implementação de uma ação interna básica.

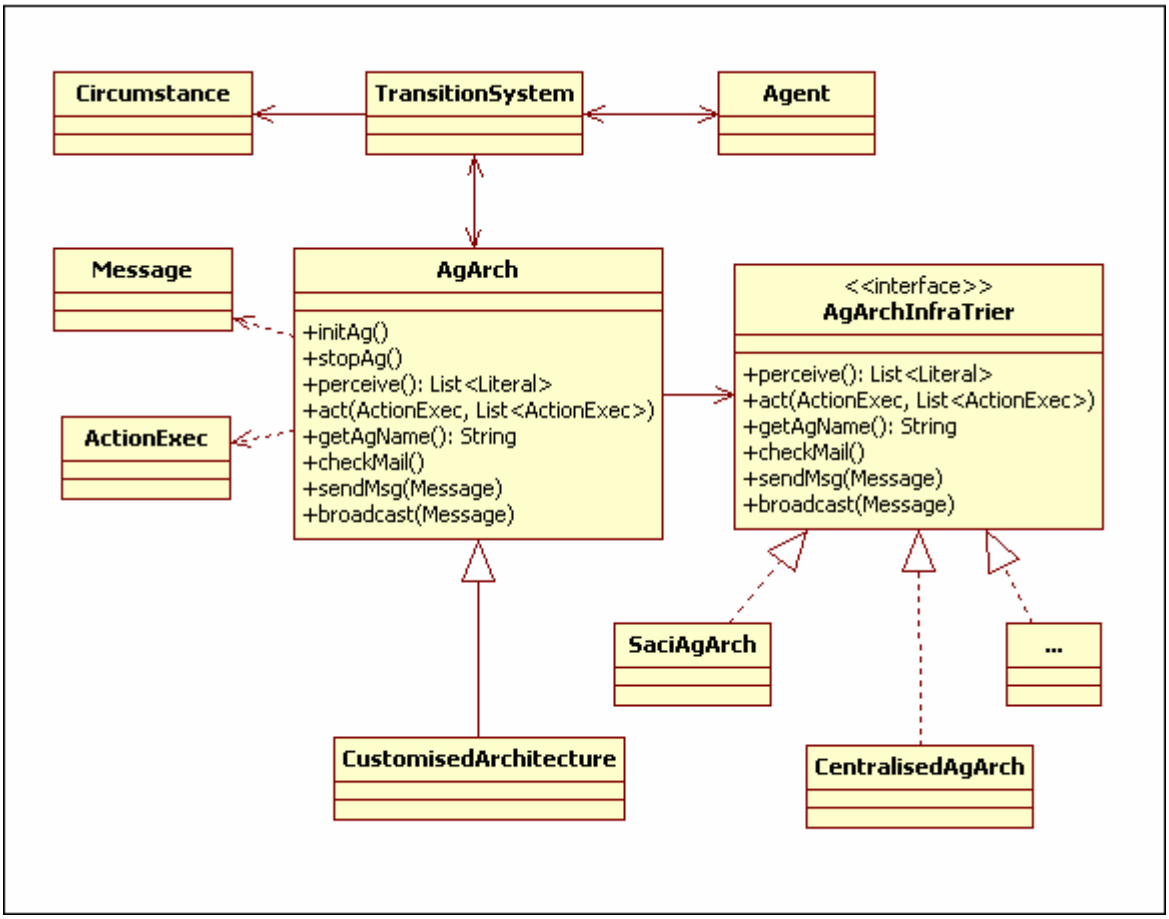
```
package math;

import jason.asSemantics.DefaultInternalAction;
import jason.asSemantics.TransitionSystem;
import jason.asSemantics.Unifier;
import jason.asSyntax.Term;

public class distance extends DefaultInternalAction {

    @Override
    public void execute(TransitionSystem ts, Unifier un, Term[] args) {
        // o código que implementa essa ação interna vai aqui.
    }
}
```

Fonte: adaptado de Bordini, Hübner e Wooldridge (2007, p. 140).

Quadro 10 – Ação interna programada em Java

#### 2.2.5.1.2 Classe de arquitetura do agente

A classe de arquitetura do agente é o componente utilizado para interligar a mente do agente (código AgentSpeak(L)) com a camada de infra-estrutura do SMA. A implementação padrão dessa arquitetura é a classe `jason.architecture.AgArch`. A Figura 10 ilustra como a classe de arquitetura está inserida no Jason.




Fonte: adaptado de Bordini, Hübner e Wooldridge (2007, p. 152).

Figura 10 – Diagrama de classe da arquitetura de um agente

A personalização da classe de arquitetura do agente é feita criando uma nova classe que estenda a classe `AgArch` e pode ser útil para "interceptar", principalmente, os métodos `act` e `perceive`. Podendo sobrescrever esses métodos, tem-se a total liberdade de integrar os agentes com ambientes externos ao SMA. Por exemplo, quando um agente AgentSpeak(L) precisa controlar um robô (no mundo real), os métodos `act` e `perceive` podem ser sobrescritos para interagir diretamente com o hardware.

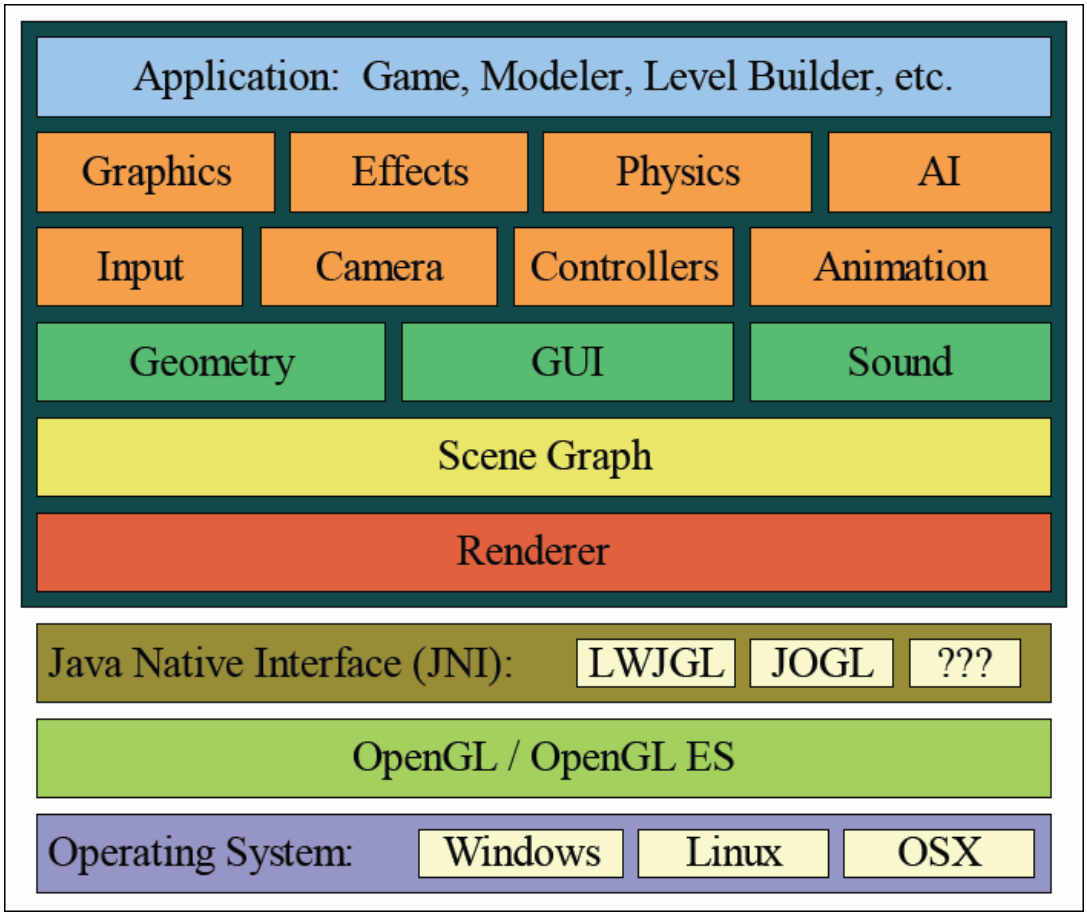
Outra personalização possível na classe de arquitetura é em relação à comunicação entre os agentes do SMA. A implementação padrão da arquitetura somente delega o serviço para os respectivos métodos da infra-estrutura (implementações da interface `AgArchInfraTrier`).

## 2.3 JMONKEY ENGINE (JME)

A JME é uma *engine* gráfica 3D *freeware* e *open-source*, escrita na linguagem Java,

utilizada para construir aplicações gráficas de alta performance. Trata-se de uma *engine* que contempla diversas construções para prover o máximo de abstração possível ao desenvolvedor, fazendo com que este não se preocupe com questões de baixo nível, por exemplo, a chamada de funções da própria biblioteca nativa de *renderização*.

A Figura 11 apresenta graficamente as camadas da arquitetura da *engine* JME. Os parágrafos subseqüentes visam explorar cada camada e suas respectivas responsabilidades.




Fonte: Patton (2004).

Figura 11 – Arquitetura da *engine* JMonkey Engine

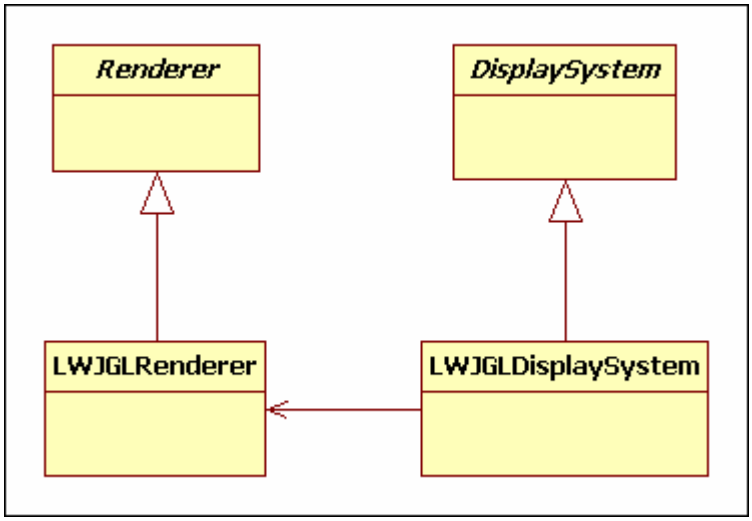
A primeira camada (e de mais baixo nível) a ser comentada é a de sistemas operacionais suportados. Por ser escrita cem por cento em Java, do ponto de vista dessa camada, a JME é considerada multiplataforma. Porém, como é visto adiante, outras camadas são importantes para definir a portabilidade da *engine* por completo.

A segunda camada trata-se da biblioteca nativa de *renderização*. Atualmente a única biblioteca suportada é a OpenGL. Consiste em uma biblioteca de alta performance, escrita na linguagem de programação C, para construção de aplicações gráficas interativas 2D ou 3D (OPENGL OVERVIEW, 2008).

A interface nativa Java (Java *Native Interface* – JNI) é a camada que faz o acoplamento entre a biblioteca nativa de *renderização* e a *engine* JME de fato. Através da

JNI, é possível que um código escrito em Java utilize a implementação de uma biblioteca escrita em C/C++, Assembler, Pascal ou outra linguagem qualquer, desde que essa possa gerar bibliotecas nativas (arquivos binários com extensão “.dll” ou “.so”). Para a biblioteca OpenGL, as implementações de JNIs mais conhecidas são a *Light Weight Java Game Library* (LWJGL) e a JOGL, sendo que atualmente a JME opera somente com a primeira opção.

A camada de *Renderer* é responsável por transformar os dados da cena representados por coordenadas de mundo para coordenadas de espaço. Essa é a primeira camada que faz parte efetivamente da *engine* JME, interagindo diretamente com a biblioteca JNI de OpenGL para desenhar na tela os dados transformados em coordenadas de espaço. Outra responsabilidade dessa camada é prover dois tipos de otimizações. O primeiro deles, denominado *culling*, é o processo de desconsiderar os objetos que não estão correntemente visíveis na perspectiva de visão do observador (câmera). O segundo tipo de otimização, chamado de *clipping*, é o processo de dividir um objeto em pequenas porções para desconsiderar as que não estão correntemente visíveis na perspectiva de visão do observador. A Figura 12 ilustra a especialização *renderizador* LWJGL.



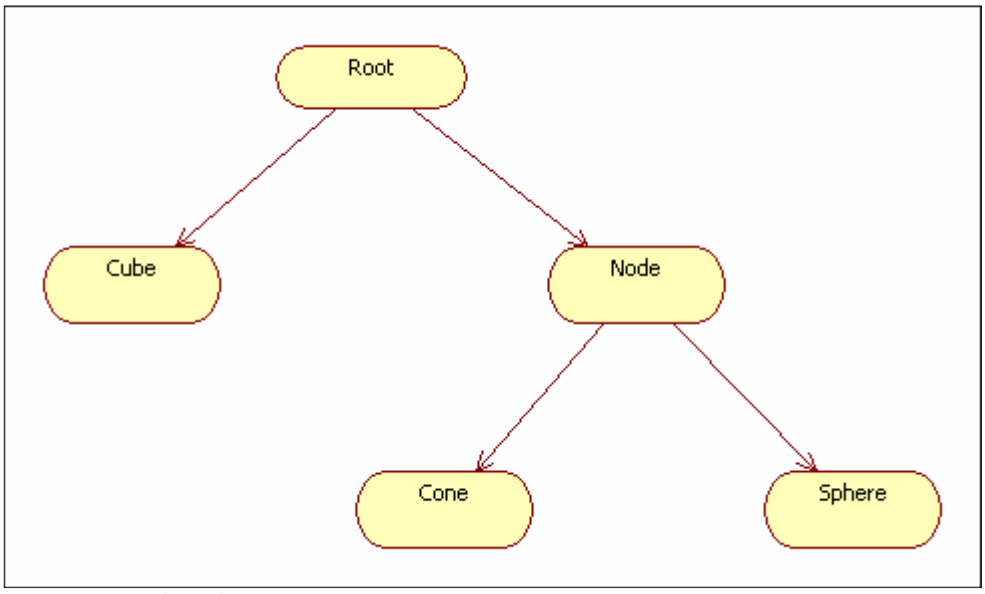

Fonte: adaptado de Patton (2004).

Figura 12 – Diagrama de classe da especialização do *renderizador* LWJGL

Através das classes abstratas `Renderer` e `DisplaySystem` que o fraco acoplamento entre a *engine* JME e a camada de JNI é garantido. Dessa forma, quando um novo *renderizador* precisar ser suportado pela JME, basta criar as respectivas especializações das classes abstratas. Mais adiante é mostrado como e em qual momento essas classes concretas são instanciadas.

A camada de grafo de cena (do inglês, *scene graph*) provê uma maneira de representar os dados do ambiente virtual de forma hierárquica. Patton (2004) afirma que utilizar estruturas de grafos de cena simplifica o gerenciamento de estruturas globais. Por exemplo,

para adicionar um objeto de iluminação a um cenário, basta associá-lo ao sub-nó do grafo de cena desejado, que ele e todos os seus sub-nós são afetados pelo objeto de iluminação. A Figura 13 apresenta graficamente um exemplo de grafo de cena, envolvendo objetos simples como cubos, cones e esferas. Ainda na Figura 13, os nós denominados `Root` e `Node` são considerados "nós de agrupamento", sendo que não são objetos visuais.




Fonte: Patton (2004).

Figura 13 – Exemplo de grafo de cena

A camada seguinte, composta por Geometria (Geometry), Interface de Usuário (GUI) e Sonorização (Sound), provê o penúltimo nível de abstração para o desenvolvedor da aplicação. O componente de geometria, representado pela classe base `Geometry`, define um nó folha utilizado para representar objetos geométricos no grafo de cena. No diagrama de classes da Figura 14, podem ser vistas as classes estendidas de `Geometry` que acompanham a JME por padrão. O componente de interface de usuário é a implementação padrão da JME para a criação de janelas, botões, *inputs*, entre outros componentes padrões de interface. Por último, o componente de sonorização funciona da mesma forma que a camada de renderização (*Renderer*), ou seja, conectada a uma biblioteca nativa. Atualmente esse componente da JME opera com as bibliotecas LWJGL (a mesma de *renderização*) e a JOAL[8].

A última camada de abstração é composta por oito componentes. Os componentes de gráficos e de efeitos são responsáveis, respectivamente, pela importação de modelos de ferramentas externas e tratamento de texturas, e pela criação e gerenciamento de sistemas de partículas, tal como efeitos de fogo, fumaça e explosão. O componente *Input* é responsável

[8] JOAL é uma implementação JNI para utilização da biblioteca nativa OpenAL. Mais informações podem ser vistas em JOAL (2008).

por prover uma interface dos comandos de teclado e mouse com a aplicação. A câmera é utilizada para definir o campo de visão do usuário. O componente controlador é utilizado para gerenciar objetos animados, que freqüentemente são importados de ferramentas externas, tal como 3D Studio Max, MilkShape 3D e Blender.

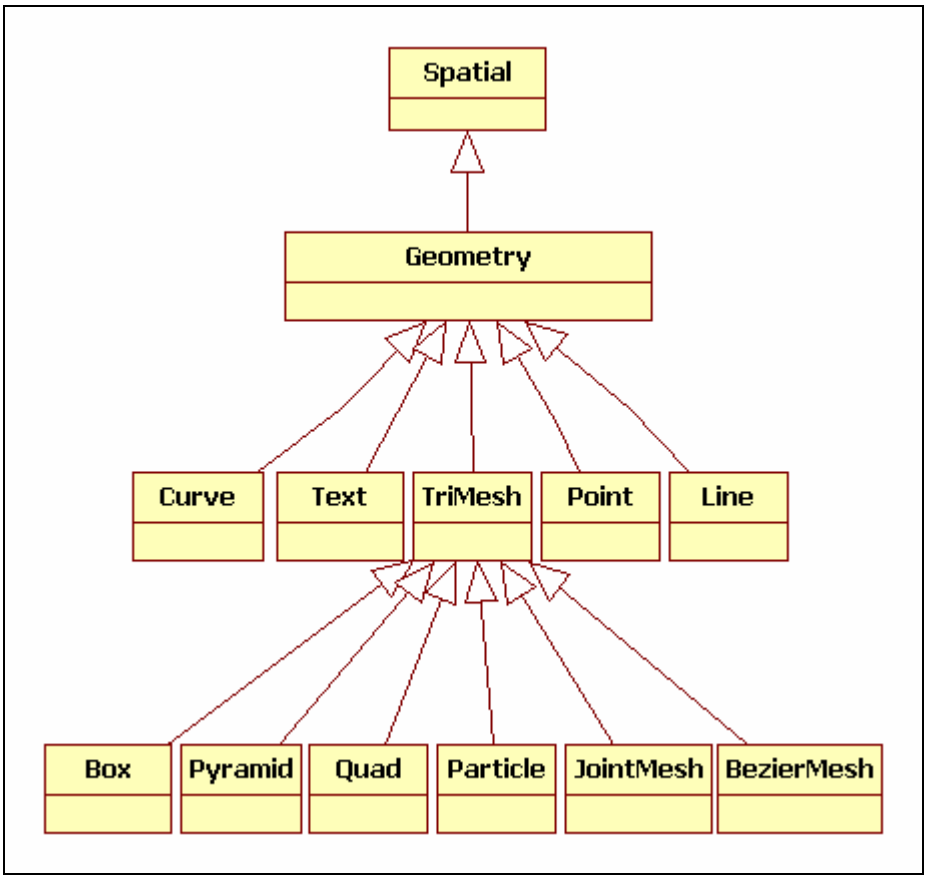



Figura 14 – Diagrama de classes do componente de geometria da JME

Ainda tratando-se da última camada de abstração, o componente mais relevante é o que controla a física dos objetos de uma aplicação JME. Questões de física não são diretamente tratadas pela *engine* JME, essa tarefa é delegada a um *framework* de física específico, chamado JME Physics. Esse *framework* é responsável por prover uma interface de alto nível entre a *engine* JME e a Open Dinamics Engine (ODE), sendo essa uma biblioteca de alta performance escrita na linguagem de programação C, utilizada para simulação de corpos rígidos dinâmicos.

### 2.3.1 Principais classes e interfaces

Além das classes já citadas, a *engine* JME e o *framework* JME Physics provem uma gama de classes e interfaces que visam facilitar o desenvolvimento de aplicações gráficas

escritas em Java, assim como pode ser visto no Quadro 11.

| **CLASSE/INTERFACE** | **DESCRIÇÃO/FUNCIONALIDADES** |
|---|---|
| `StandardGame` | Classe central de uma aplicação JME. Contém o *loop* principal da aplicação. |
| `DisplaySystem` | Interface para a criação do sistema. Através dela pode ser obtida a referência para o *renderizador*. |
| `StatisticsGameState` | Provê um *game state* com estatísticas de *frames* por segundo, número de vértices e número de triângulos. |
| `GameStateManager` | Gerenciador de todos os *game states* da aplicação. Através dele que novos *game states* são anexados para serem *renderizados*. |
| `GameTaskQueueManager` | Provê um ponto central para adicionar tarefas a serem executadas dentro da *Thread* principal da aplicação (*thread* que faz uso direto da biblioteca JNI do OpenGL). Por questões de sincronismo, todo processamento que faz uso (mesmo que internamente) das APIs do OpenGL, precisa ser executado dentro da *thread* principal da aplicação. |
| `PhysicsGameState` | Provê um *game state* responsável pelo controle da física dos objetos estáticos e dinâmicos. |
| `PhysicsSpace` | Classe central do *framework* JME Physics. Utilizado para criar objetos estáticos e dinâmicos. |
| `Node` | Provê um nó de agrupamento para o grafo de cena. |
| `DynamicPhysicsNode` | Mesmas funcionalidades da classe `Node`, porém essa é controlada pela física. Contém tratamento de colisão e é afetada pela gravidade do mundo. |
| `StaticPhysicsNode` | Mesmas funcionalidades da classe `DynamicPhysicsNode`, porém essa não é afetada pela gravidade do mundo. |
| `Vector3f` | Provê métodos para manipulação de uma coordenada 3D. |
| `PointLight` | Provê um ponto de luz para ser adicionado ao grafo de cena. |
| `Camera` | Interface que encapsula todo o gerenciamento da *viewport*. Representa a visão do observador. |
| `InputHandler` | Utilizado para manipular eventos de teclado e mouse. |
| `ChaseCamera` | Classe herdada de `InputHandler`. Câmera utilizada para seguir um determinado objeto da cena. Possui eventos de mouse e teclado pré-definidos para rotação e zoom. |
| `FirstPersonHandler` | Classe herdada de `InputHandler`. Possui eventos de mouse e teclado pré-definidos para movimentar, rotacionar e dar zoom na câmera. |
| `PhysicsUpdateCallback` | Interface usada para implementação de *callbacks* de física. A cada atualização do Game State de física os métodos `beforeStep` e `afterStep`. |
| `Texture` | Classe que representa uma textura na *engine* JME. |
| `TextureManager` | Provê métodos estáticos para criação e carga de texturas. |
| `SkyBox` | Provê uma caixa que possibilita a adição de texturas nas faces internas. Utilizada comumente para criação do céu do cenário. |
| `ImageBasedHeightMap` | Classe utilizada para gerar um “mapa de alturas” a partir de uma imagem em escala de cinza. |
| `TerrainPage` | Classe utilizada para gerar malha de polígonos para terrenos, baseado em um “mapa de alturas”. |
| `ProceduralTextureGenerator` | Provê um gerador de texturas procedural. Utilizada para combinar diferentes texturas. |

Quadro 11 – Principais classes e interfaces da *engine* JME e do *framework* JME Physics

A classe `StandardGame` é normalmente utilizada como superclasse da classe principal

de uma aplicação Java utilizando a *engine* JME. Como mencionado no Quadro 11, essa classe é responsável pelo controle do *loop* principal da aplicação, que consiste basicamente nos passos:

a) atualização do grafo de cena:
   - executa as tarefas de atualização enfileiradas através da classe `GameTaskQueueManager`,
   - atualiza os *game states* através da classe `GameStateManager`,
   - atualiza o sistema de áudio através da classe `AudioSystem`;

b) *renderização* do grafo de cena:
   - limpa as correntes estatísticas do *renderizador*,
   - limpa os buffers do *renderizador*,
   - executa as tarefas de *renderização* enfileiradas através da classe `GameTaskQueueManager`,
   - *renderiza* os *game states* através da classe `GameStateManager`;

Um *game state* é utilizado para encapsular um determinado estado da aplicação. Exemplos comuns de *game states* são os estados "em jogo" (*in game state*) ou "menu do jogo" (*game menu state*).

O diagrama de atividades da Figura 15 apresenta sinteticamente o processo de inicialização de uma aplicação Java utilizando a *engine* JME.

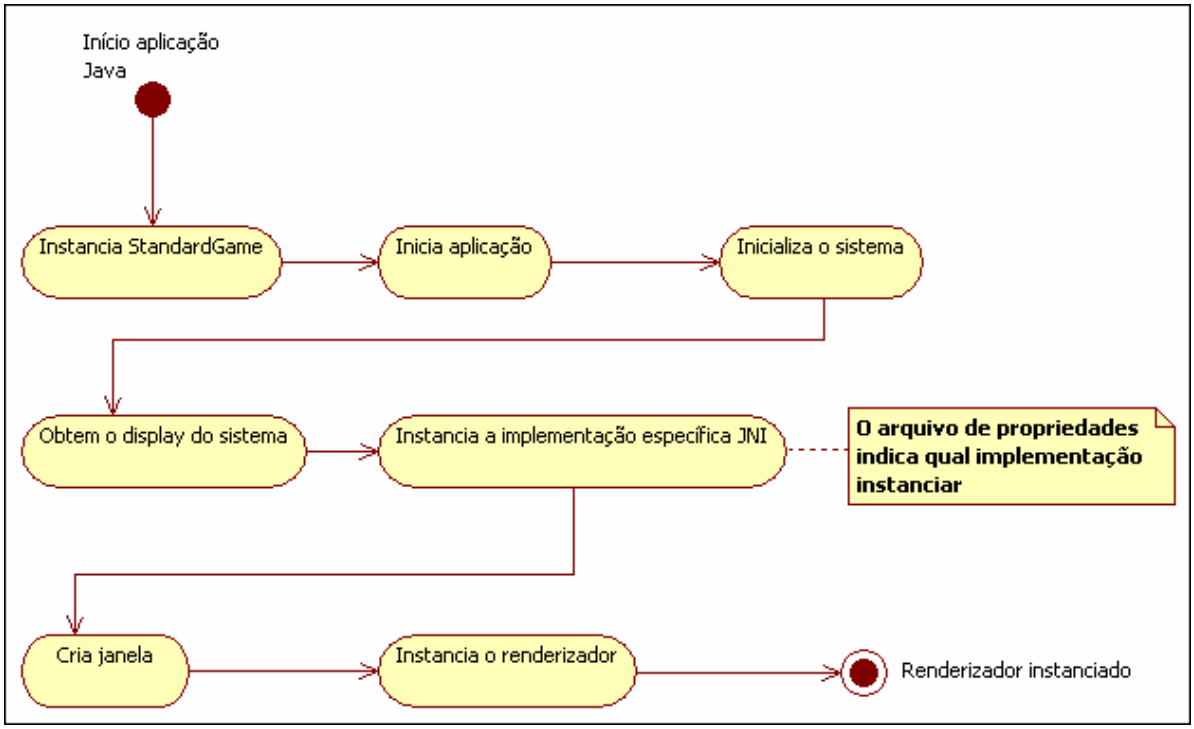



Figura 15 – Diagrama de atividades da inicialização de uma aplicação JME

## 2.4 TRABALHOS CORRELATOS

Existem alguns projetos semelhantes ao atual trabalho desenvolvido. Dentre eles, foram escolhidos três cujas características se enquadram nas três principais áreas de estudos desse trabalho. Foram selecionados os projetos: Robocode (ROBOCODE, 2007), “Sistema Multiagentes Utilizando a Linguagem AgentSpeak(L) para Criar Estratégias de Armadilha e Cooperação em um Jogo Tipo PacMan” (APPIO, 2004) e o “Protótipo de um Ambiente Virtual Distribuído Multiusuário” (EDUARDO, 2001).

### 2.4.1 Robocode

O Robocode é um projeto originalmente desenvolvido por Mathew A. Nelson entre 2001 e 2005, atualmente mantido por Flemming N. Larsen (HELP ROBOCODE, 2005). Trata-se de um jogo de tanques de guerra onde os usuários programam seus tanques, utilizando a linguagem Java, para batalharem em uma arena.

Neste jogo, a anatomia de um *bot*[9] é composta por: corpo, arma e radar. Cada componente do tanque possui uma função específica a ser tratada pelo programador.

Através de uma *Application Programming Interface* (API) bem definida e documentada, os usuários são capazes de programar os *bots*. Basicamente, deve-se criar uma classe Java que herde da classe `robocode.Robot`, `robocode.AdvancedRobot`, ou `robocode.TeamRobot`, e sobrescrever o método `run` sem argumentos. Essas classes do pacote `robocode` são na verdade, classes que implementam a interface `java.lang.Runnable`, isso significa que cada *bot* é uma *thread*.

Sua interface é bastante simples, contempla somente gráficos 2D e possui um cenário sem nenhum obstáculo, ou seja, só composto pelos próprios *bots*. Outra limitação do jogo é caracterizada pelo fato de não suportar a visualização da batalha em computadores distintos.

[9] No contexto do Robocode, *bot* é sinônimo de robô de tanque de guerra.

### 2.4.2 Sistema multiagentes utilizando a linguagem AgentSpeak(L) para criar estratégias de armadilha e cooperação em um jogo tipo PacMan

Trata-se de um SMA que atua em um jogo do tipo PacMan, onde os agentes, nesse contexto, representam os personagens fantasmas. O objetivo dos agentes neste SMA é criar estratégias de cooperação para a execução de armadilhas com o intuito de dificultar a vitória do personagem come-come, que é controlado por um usuário em tempo real.

Para a implementação do SMA, o modelo de arquitetura utilizado é o BDI, a linguagem para implementação dos agentes é a AgentSpeak(L) e a ferramenta utilizada para interpretar essa linguagem é o Jason.

### 2.4.3 Protótipo de um ambiente virtual distribuído multiusuário

Apresenta aspectos importantes a serem observados ao projetar e implementar um ambiente virtual, focando mais na construção de AVDs. São demonstradas e testadas técnicas para manter consistente e sincronizado o estado do AVD. Também são demonstradas técnicas e algoritmos que procuram melhor aproveitar os recursos de rede disponíveis.

Para a construção do mundo virtual do protótipo utilizou-se a API do Java3D, que é uma *engine* gráfica 3D desenvolvida pela Sun Microsystems para construção de aplicações gráficas. Também utilizou-se a API do DIS-Java-VRML que contribuiu principalmente para tratar os mecanismos de controle de comunicação entre os usuários e para a representação gráfica do cenário utilizando a linguagem Virtual Reality Modelating Language (VRML).

# 3 DESENVOLVIMENTO

Baseado no que foi definido e estudado no capítulo 2, esse capítulo apresenta como foi desenvolvido o simulador de batalha de tanques, denominado TankCoders.

## 3.1 REQUISITOS PRINCIPAIS DO SIMULADOR

O simulador deverá:

a) permitir a escolha do mapa onde a batalha de times de tanques será executada (Requisito Funcional – RF);
b) possibilitar a entrada de novos modelos de mapas (RF);
c) disponibilizar os dois modelos de tanques de guerra: um mais pesado, com menos mobilidade, com maior poder de fogo no canhão e uma metralhadora, e outro mais leve, com mais mobilidade, com menor poder de fogo no canhão e uma metralhadora (RF);
d) fornecer um conjunto de ações, percepções e crenças para os agentes que representam os tanques (RF);
e) permitir que os tanques sejam controlados por usuários (*avatares*) ou tenham um comportamento autônomo (agentes BDI) (RF);
f) prover um AVD com arquitetura híbrida (parte centralizada e parte distribuída) que rode sobre uma rede local (LAN) (RF);
g) tratar no máximo seis tanques de guerra em cada time. Pois foi escolhida a arquitetura híbrida (parte centralizada e parte distribuída) para o AVD, desta forma se existirem muitos usuários conectados ao servidor, a tendência é diminuir significativamente a taxa de FPS, conseqüentemente diminuindo o nível de imersão no mundo virtual por parte do usuário (Requisito Não-Funcional – RNF);
h) garantir que nenhum usuário se conecte ao servidor após o início da batalha (RNF);
i) ser implementado na linguagem de programação Java Standart Edition (JSE) 6.0, utilizando a *engine* gráfica JMonkey Engine, o *framework* JME Physics, o *middleware* JGN e o interpretador Jason (RNF).

## 3.2 ESPECIFICAÇÃO

Para o desenvolvimento desse trabalho, inicialmente especificou-se uma arquitetura híbrida para o AVD, ou seja, com características de arquitetura centralizada combinadas com características de arquitetura distribuída. A partir dessa especificação inicial, especificou-se também o aplicativo servidor e o aplicativo cliente, como apresentado na Figura 16 e descrito nas seções subseqüentes.

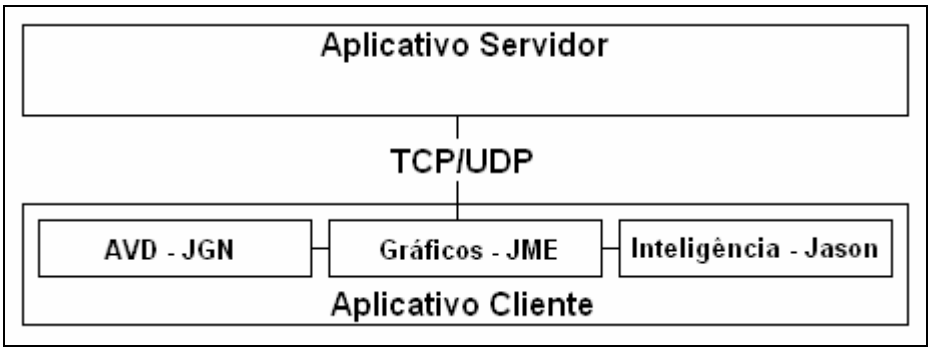



Figura 16 – Arquitetura geral do simulador TankCoders

### 3.2.1 Especificação da arquitetura híbrida do AVD

A arquitetura do simulador foi especificada baseando-se no paradigma de orientação a objetos, usando a *Unified Modeling Language* (UML). A ferramenta StarUML foi usada para a elaboração dos diagramas de classes, de estados e grafo de cena.

Baseado nos modelos gerais de comunicação estudados na seção 2.1.1, que são o modelo centralizado e o distribuído, vários problemas são notados quando utilizados separadamente. Tendo em vista que a principal vantagem do modelo centralizado é o fato de ter um computador central, onde informações gerais a respeito do AVD e de seus usuários podem ser armazenadas, pode-se rapidamente perceber a facilidade de implementação e manutenção do estado do AVD. Porém, em um modelo centralizado propriamente dito, todo o estado do AVD deve ser mantido no aplicativo servidor, sendo que esse tem o dever de enviar para cada cliente somente imagens *raster* quadro-a-quadro, tornando o cliente um simples "terminal burro". Por outro lado, no modelo distribuído, cada usuário mantém uma cópia local do AVD, sendo que as alterações efetuadas localmente devem ser notificadas à todos os usuários (comumente através de mensagens *broadcast*). Uma desvantagem desse modelo é

em relação à entrada de novos usuários, pois não se tem um computador central armazenando todo o estado do AVD e técnicas auxiliares precisam ser implementadas.

O simulador proposto nesse trabalho define que a partir do momento que a batalha é iniciada, nenhum usuário pode se conectar ao servidor, pois esse entra no estado “em jogo”, não aceitando mais requisições de conexão.

A arquitetura híbrida desenvolvida nesse trabalho é baseada principalmente no modelo centralizado. A única característica proveniente do modelo distribuído é que cada usuário possui sua própria representação do estado do AVD. O simulador possui um aplicativo servidor e um cliente, como é descrito e detalhado nas seções seguintes.

#### 3.2.1.1 Especificação do aplicativo servidor

Como a arquitetura escolhida segue principalmente o modelo centralizado, o papel de um aplicativo servidor apresenta-se como um componente necessário para a composição da infra-estrutura do simulador por completo. A função desse aplicativo depende diretamente do corrente estado no qual ele se encontra, sendo que os estados possíveis são “Parado” (*Stopped*), “Esperando por Conexões” (*WaitingConnections*) ou “Em Jogo” (*InGame*). A Figura 17 ilustra esses estados e como ocorre a transição entre eles.

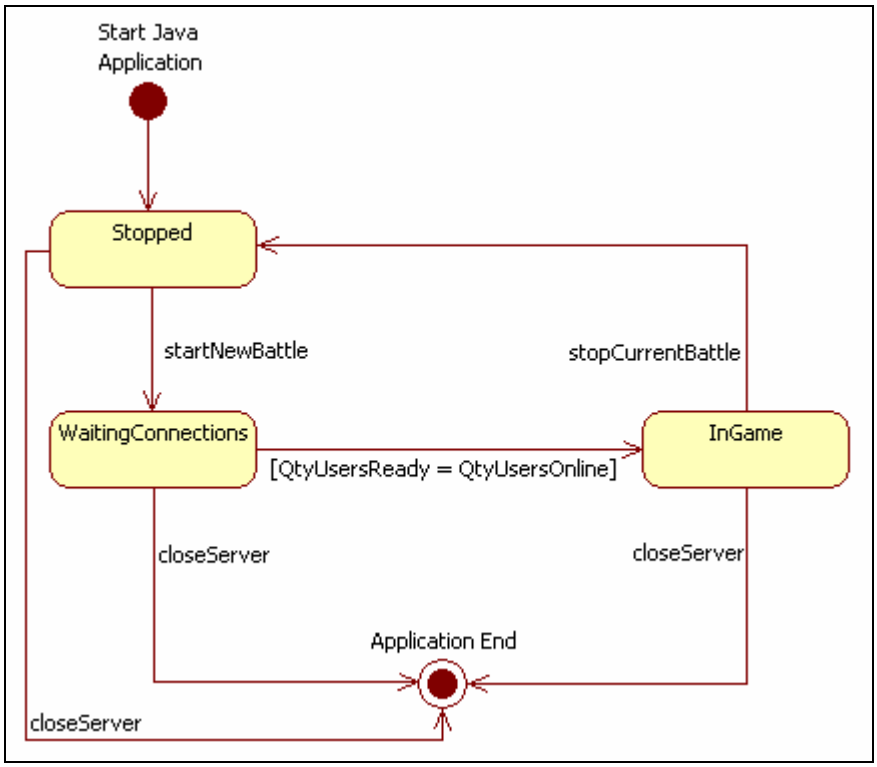



Figura 17 – Estados que o aplicativo servidor pode assumir

No início da aplicação, o estado *Stopped* é diretamente definido. Nessa etapa, o aplicativo servidor não precisa executar nenhuma tarefa, é esperado somente que o usuário informe os parâmetros da batalha de tanques de guerra e inicie-a quando desejado (evento `startNewBattle`).

Quando a batalha é iniciada, o servidor assume o estado *WaitingConnections*. Essa etapa compreende o aguardo de requisições de conexão por parte dos usuários. No momento que todos os usuários conectados possuem o estado "Pronto para Jogar" (*Ready*), o servidor passa a assumir o estado *InGame*.

O estado *InGame* tem como primeiro objetivo enviar uma mensagem para cada usuário conectado a respeito do início da batalha. Tendo feito isso, o servidor fica aguardando o recebimento de mensagens de controle da batalha, como é visto adiante na seção 3.3, que trata da implementação do simulador.

A Figura 18 apresenta a especificação das classes do servidor, relacionando-as com classes da API do Java e do *middleware* JGN. Em seguida é feita uma reflexão a respeito da responsabilidade de cada uma delas.

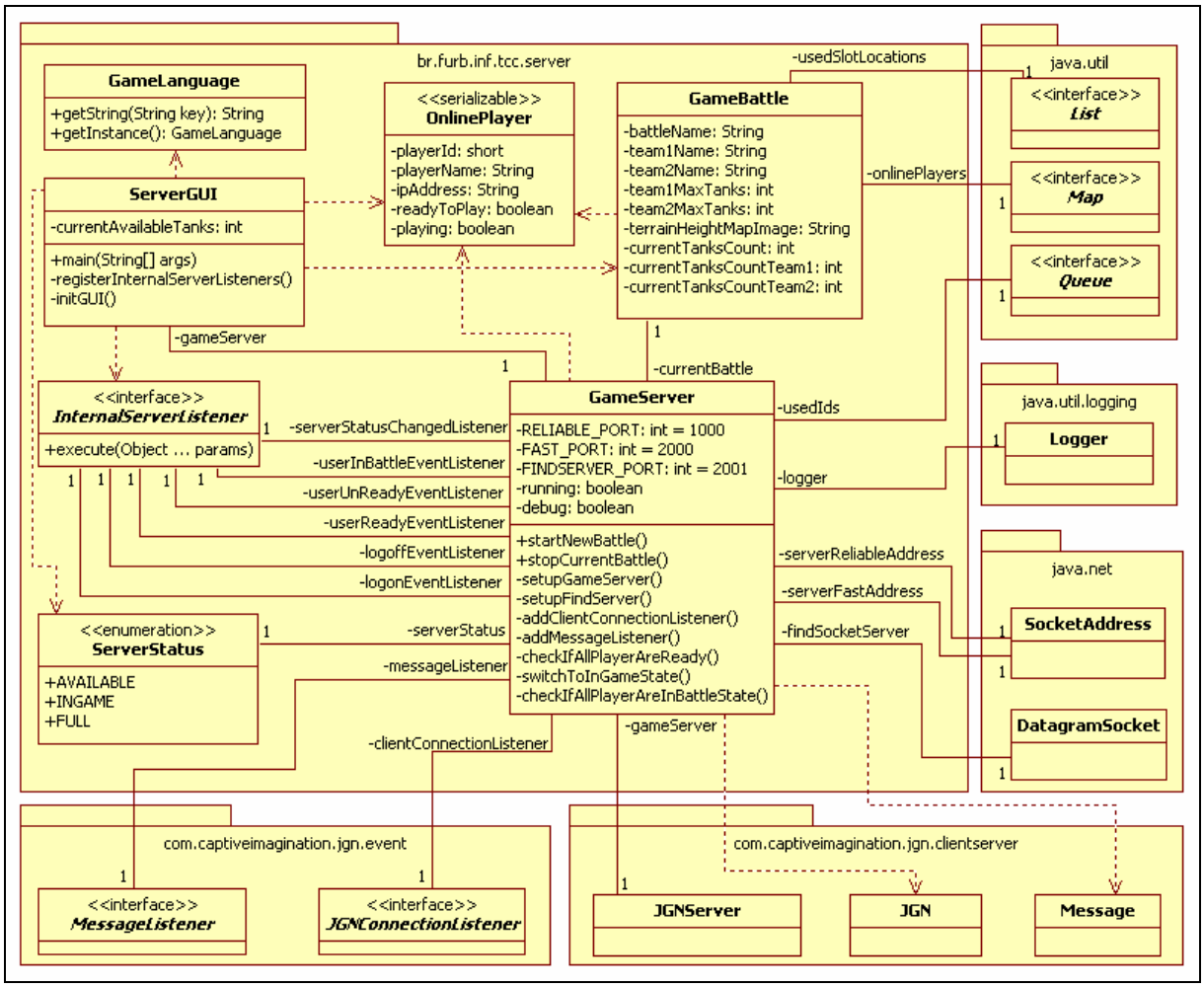



Figura 18 – Diagrama de classes do aplicativo servidor

O início da aplicação é dado através da classe `ServerGUI`, pois é nessa que está contido o método principal (método publico e estático chamado `main`). Essa classe é responsável por exibir uma tela de criação de batalhas, nela são informados o nome da batalha, o nome dos dois times, a quantidade máxima de tanques em cada time e por último o terreno onde a batalha deve ser executada.

A classe `GameLanguage` provê suporte à todas as classes de interface com o usuário do simulador. Seguindo o padrão de projetos (do inglês, *design pattern*) *Singleton*, a única instância dessa classe só pode ser obtida através do método `getInstance`, fazendo com que seja simples obtê-la em qualquer ponto da aplicação. A responsabilidade dessa classe é prover uma internacionalização dos textos que aparecem nas telas. Nessa abordagem, os textos (*strings*) são colocados em arquivos de propriedades, ou seja, cada linha contendo uma chave e um valor, separados por "=". Para cada língua que a aplicação suporta, existe um arquivo de *strings*, sendo que somente um pode estar associado à aplicação de cada vez.

A classe `GameBattle` é utilizada pela classe `ServerGUI` para a criação de novas batalhas. Nela é possível atribuir todos os campos da tela de criação de batalha e passar para a instância da classe `GameBattle`, que é criada no construtor da classe `ServerGUI`.

`GameServer` é a principal classe do aplicativo servidor. Nela são implementados os mecanismos de comunicação com os clientes através do *middleware* JGN e da API de *sockets* do Java. O funcionamento dessa classe, como apresentado na Figura 17, depende do estado correntemente definido, sendo que cada estado também já foi abstratamente comentado nos parágrafos anteriores. Quando ocorre algum evento específico nessa classe, por exemplo, uma conexão ou saída de um usuário, isso precisa ser notificado à tela (`ServerGUI`), para isso faz-se uso da interface `InternalServerListener`. Como ilustrado no diagrama de classes da Figura 18, os eventos que podem ocorrer no servidor são:

a) *serverStatusChangedListener*: quando um jogador troca de status;
b) *userInBattleEventListener*: quando um jogador entra na batalha;
c) *userReadyEventListener*: quando um jogador define que está pronto para jogar;
d) *userUnReadyEventListener*: quando um jogador define que não está pronto para jogar;
e) *logonEventListener*: quando um jogador conecta-se ao servidor;
f) *logoffEventListener*: quando um jogador desconecta-se do servidor.

Através da interface `InternalServerListener`, dentro da classe `ServerGUI` pode-se

implementar e registrar uma classe interna anônima[10] (*anonymous inner class*) para cada um desses eventos que podem ocorrer no servidor. Utilizando essa solução, o acoplamento entre a classe de interface de usuário (`ServerGUI`) e o modelo do servidor (`GameServer`) torna-se fraco, fazendo com que possíveis refatorações (*refactoring*) no código de ambas as classes seja mais ágil e seguro. Outra característica da interface `InternalServerListener`, é que ela segue o padrão de projetos *Command*, sendo que esse define que uma classe ou interface possua um método principal, comumente chamado de `execute`. Essa interface serve como uma abstração e também como um contrato para quem utiliza o recurso, sendo que a implementação de fato não precisa ser conhecida pelo utilizador.

A enumeração `ServerStatus` serve para determinar qual a corrente situação do servidor. Essa informação é utilizada principalmente no momento que um cliente faz uma requisição de procura de servidores (através de mensagem *broadcast* na rede). Quando essa requisição é feita, o servidor responde ao cliente, juntamente com outras informações, o seu corrente status.

Uma instância da classe `OnlinePlayer` é criada pelo `GameServer` quando um jogador conecta-se ao servidor. Quando isso ocorre, também é enviada ao jogador uma lista de todos os jogadores já conectados, ou seja, enviando instâncias da classe `OnlinePlayer`. Mais adiante, na especificação do aplicativo cliente, é visto que existe uma classe denominada `Player`, sendo que essa poderia ser utilizada ao invés de ter a `OnlinePlayer`. Porém, a classe `Player` possui um alto nível de acoplamento com o aplicativo cliente, sendo que várias informações seriam transmitidas através da rede sem a real necessidade. Por esse motivo, baseado principalmente no padrão de projetos da plataforma Java EE denominado *Transfer Object*, a classe `OnlinePlayer` é utilizada para transmitir um objeto somente com as informações necessárias.

#### 3.2.1.2 Especificação do aplicativo cliente

Além do aplicativo servidor, o outro componente apresentado na Figura 16 é o aplicativo cliente. Esse componente é responsável por exibir graficamente o estado do AVD (armazenado localmente em cada instância de aplicativo cliente), bem como interagir com o aplicativo servidor para mantê-lo atualizado acerca das modificações nos demais jogadores.

[10] Mais informações sobre as classes internas anônimas podem ser encontradas em The Java Tutorials (2008).

Também é responsável por interagir com o interpretador Jason, a fim de prover uma maneira de controlar os tanques de guerra através de agentes inteligentes seguindo a arquitetura BDI. As seções subseqüentes descrevem em detalhes a especificação de cada uma dessas responsabilidades citadas.

#### 3.2.1.2.1 Representação gráfica do estado do AVD

Esse módulo é responsável por tratar todas as questões gráficas do aplicativo cliente, envolvendo desde a criação do menu de preparação da batalha, até a criação do cenário 3D. A Figura 19 apresenta a especificação das classes para a composição do *game state* "menu" (`GameMenuState`) e início do *game state* "em jogo" (`InGameState`).

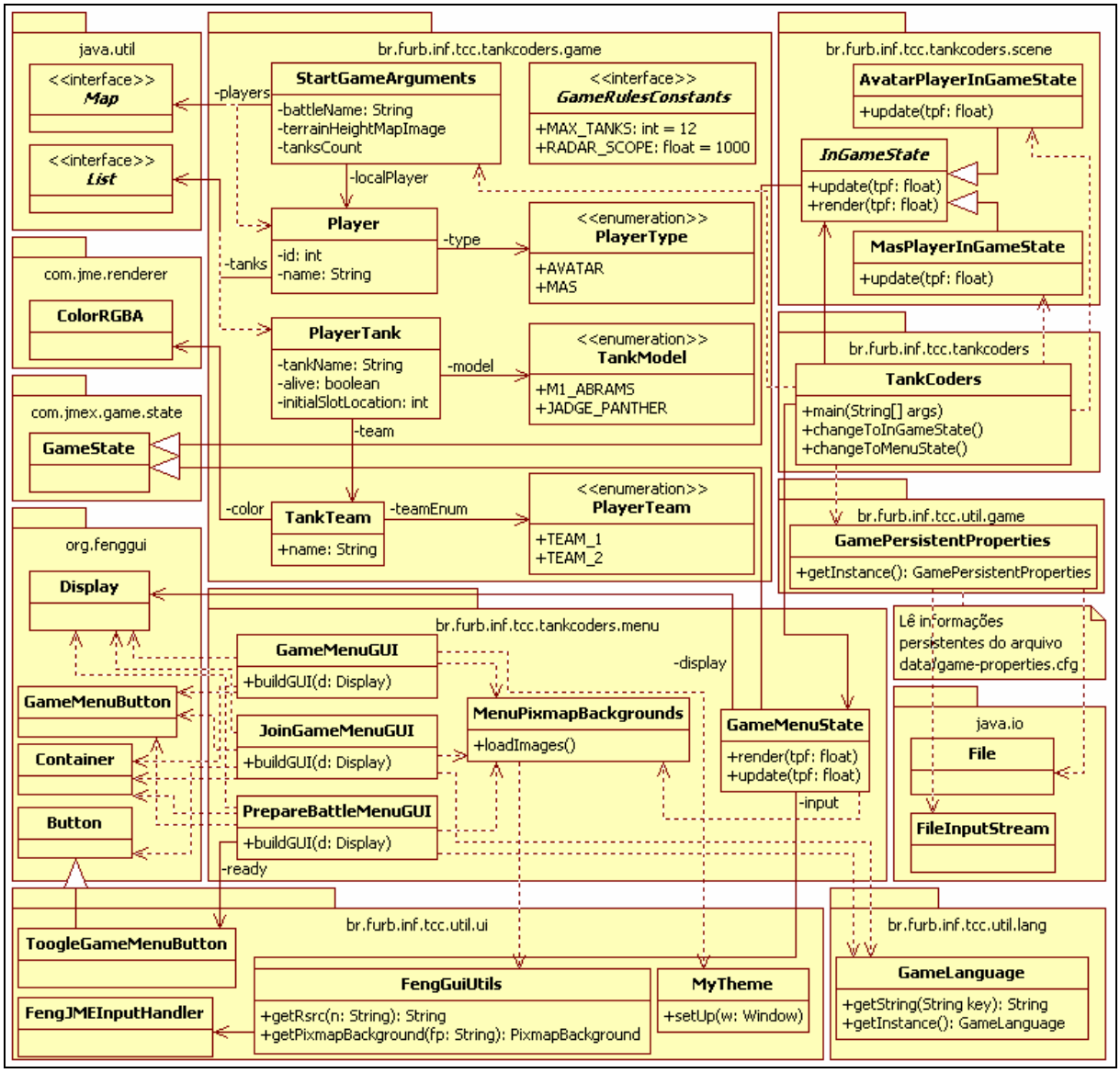



Figura 19 – Especificação de classes relacionadas ao AVD do aplicativo cliente

O início do aplicativo cliente é dado através da classe `TankCoders`. Ela é responsável por criar o *game state* "menu", que é representado pela classe `GameMenuState`. Esse *game state* é responsável por encapsular todos os componentes gráficos do menu do aplicativo cliente, que envolve as páginas:

a) menu principal (classe `GameMenuGUI`): apresenta as opções do menu principal, que são "Jogar", "Instruções", "Opções", "Créditos" e "Sair". As opções são representadas através de objetos da classe `GameMenuButton`, que é um componente de interface gráfica provido pela biblioteca FengGUI[11];
b) seleção de servidor (classe `JoinGameMenuGUI`): apresenta uma tabela contendo todos os servidores ativos do simulador TankCoders na rede local. Nesse passo jogador deve informar também um apelido para representar seu(s) tanque(s);
c) preparação da batalha (classe `PrepareBattleMenuGUI`): apresenta uma tabela contendo todos os jogadores *online* no servidor selecionado. Nessa página, o jogador pode alterar o modelo e o time de cada um de seus tanques. Quando o jogador decide que está pronto para batalhar, ele clica no botão "*Ready*" (representado pelo componente `ToogleGameMenuButton`, que é uma customização do componente `Button` da biblioteca FengGUI).

`MenuPixmapBackgrounds` é uma classe que carrega para a memória, todas as imagens utilizadas nos menus do aplicativo cliente. A carga das imagens é feita logo na criação do *game state* "menu", ou seja, no construtor da classe `GameMenuState`. Dessa maneira, a transição entre as páginas do menu ocorre de forma mais rápida, pois todos os recursos necessários já estão em memória.

No aplicativo cliente, a classe `GameLanguage` também é utilizada para prover o aspecto de internacionalização dos textos (*strings*) utilizados nos menus. Essa abordagem é bastante utilizada em jogos, simuladores e demais sistemas computacionais, tendo em vista podem ser utilizados por usuários do mundo todo.

A classe `GamePersistentProperties` é utilizada para carregar informações contidas no arquivo de configuração do simulador, denominado "game-properties.cfg". As principais informações que este arquivo contempla são utilizadas pela *engine* JME para a inicialização dos componentes do sistema. Outras informações consideradas por este arquivo são em relação à língua a ser usada no simulador (no aplicativo cliente e servidor). Também contém informações a respeito da integração com o interpretador Jason, como é visto na seção que

[11] FengGUI é uma biblioteca para criação de interfaces gráficas dentro de aplicações JME.

aborda este assunto.

As classes do pacote "`br.furb.inf.tcc.tankcoders.game`" representam as estruturas de dados utilizadas para a criação de batalhas. A interface `GameRulesConstants` é utilizada para definir constantes de determinadas regras do jogo. As enumerações `PlayerType`, `TankModel` e `PlayerTeam`, são utilizadas para representar, respectivamente, os tipos de jogadores, modelos de tanques disponíveis e times disponíveis aos jogadores.

A classe `Player` representa um jogador conectado ao servidor. A enumeração `PlayerType` é utilizada para definir se o jogador é do tipo *avatar* (controlado pelo jogador via teclado e mouse) ou do tipo Sistema MultiAgentes (*MultiAgent System* – MAS) (controlado por agentes BDI do interpretador Jason).

A classe `PlayerTank` concebe uma estrutura de dados para um tanque de guerra. Através dela, é possível definir uma lista de todos os tanques pertencente a um jogador. Jogadores do tipo *avatar* possuem somente um tanque, entretanto jogadores do tipo MAS podem possuir diversos deles. Um objeto da classe `TankTeam` é associado para determinar qual o time que o tanque do jogador compõe.

No momento que a batalha deve ser iniciada (processo que é explicado em detalhes na seção seguinte), através do método `changeToInGameState` da classe `TankCoders`, um objeto da classe `AvatarPlayerInGameState` ou `MasPlayerInGameState` (que são especializações da classe `InGameState` para um determinado tipo de jogador) é criado para representar o *game state* "em jogo". A partir desse momento, o cenário 3D da batalha é montado, dando inicio de fato ao combate. O grafo de cena apresentado na Figura 20 ilustra a hierarquização dos elementos que compõem o cenário do simulador TankCoders.

O nó `RootNode` apresenta-se como o nó principal do grafo de cena. É o elemento passado ao *renderizador* a cada atualização de *frame* para que a cena seja desenhada. A partir desse nó, são anexados os nós `Sky`, `Terrain`, `Hud` e pelo menos duas ocorrências de tanques, representados pelos nós `TankM1Abrams` e `TankJadgePanther`.

O nó `Sky` é responsável por encapsular as configurações da *sky box* do cenário, tal como a definição das texturas a serem aplicadas. Também é responsabilidade desse nó, a atualização posicionamento da *sky box* baseado no corrente posicionamento da câmera. Esse nó, como o próprio nome sugere, provê a visualização de um céu para o cenário. Sem a utilização dele, ao redor do terreno só seria visto um quadro de cor preta, fazendo com que a cena fique com um nível de realidade muito baixo.

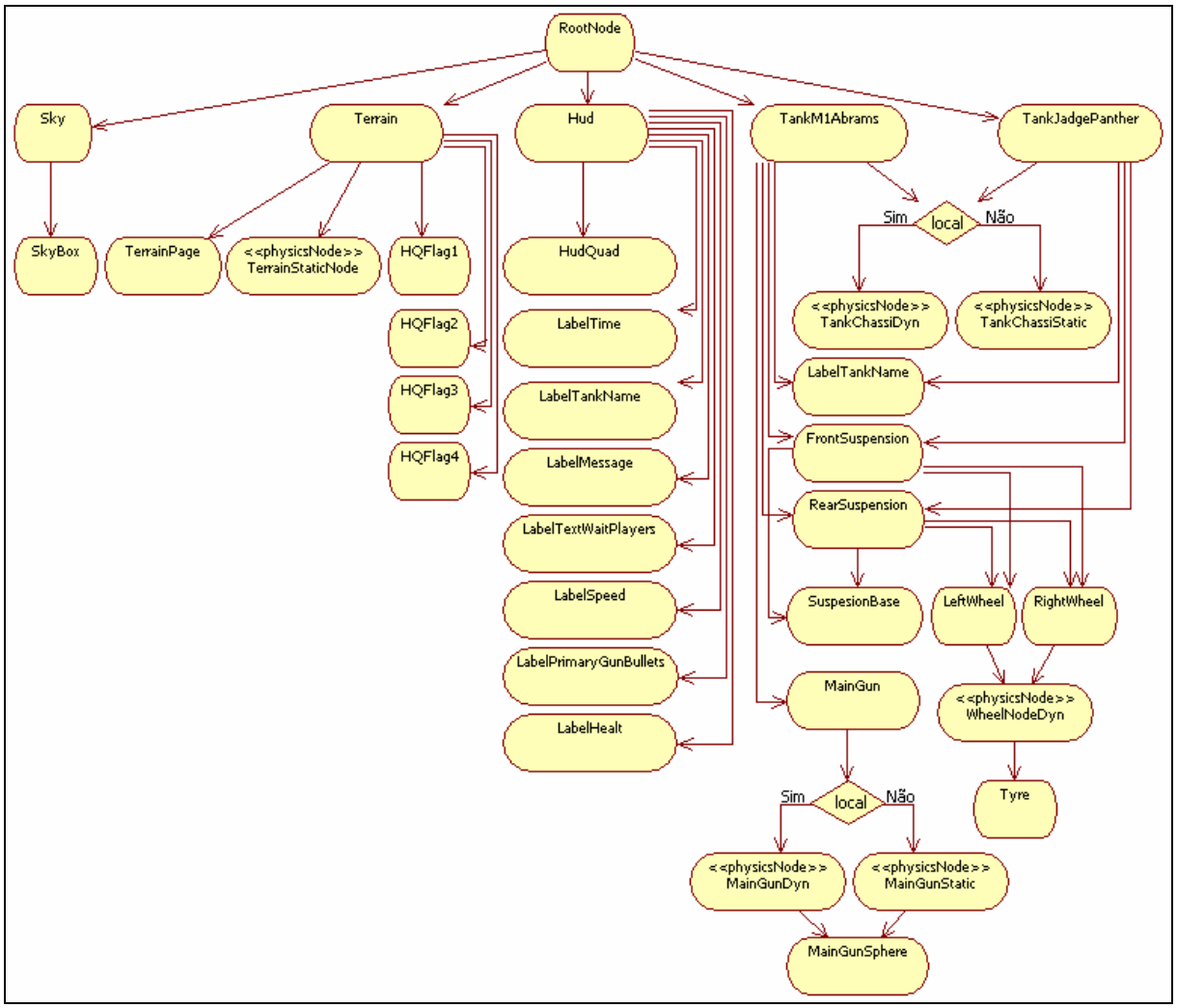



Figura 20 – Grafo de cena do simulador

`Hud` é o nó responsável por exibir um painel de informações a respeito da batalha e do corrente tanque sendo "seguido" pela câmera. `HudQuad`, o primeiro nó filho de `Hud` visto no grafo de cena, é responsável por exibir um quadrado escuro transparente na parte inferior da tela, onde as informações são sobrepostas. Os rótulos (*labels*) `LabelTime`, `LabelTankName`, `LabelMessage`, `LabelTextWaitPlayers`, `LabelSpeed`, `LabelPrimaryGunBullets` e `LabelHealth` são os elementos usados para exibir as respectivas informações no painel (*hud*).

O nó `Terrain` é usado para encapsular a criação física e visual do terreno. O nó filho `TerrainPage` contém a malha de triângulos que representa a topografia do terreno. Nele são aplicadas as texturas, fazendo com que o terreno torne-se visível ao jogador. O nó filho `TerrainStaticNode` (que contém o estereótipo *physicsNode*) é responsável por prover a representação física da topografia do terreno. Dessa forma, os demais objetos controlados pelo *framework* de física são sujeitos a ter colisão com o terreno, garantindo o aspecto de realidade no cenário 3D. Com auxílio do diagrama de classes da Figura 21, os parágrafos seguintes descrevem a composição do terreno e como ele é criado.

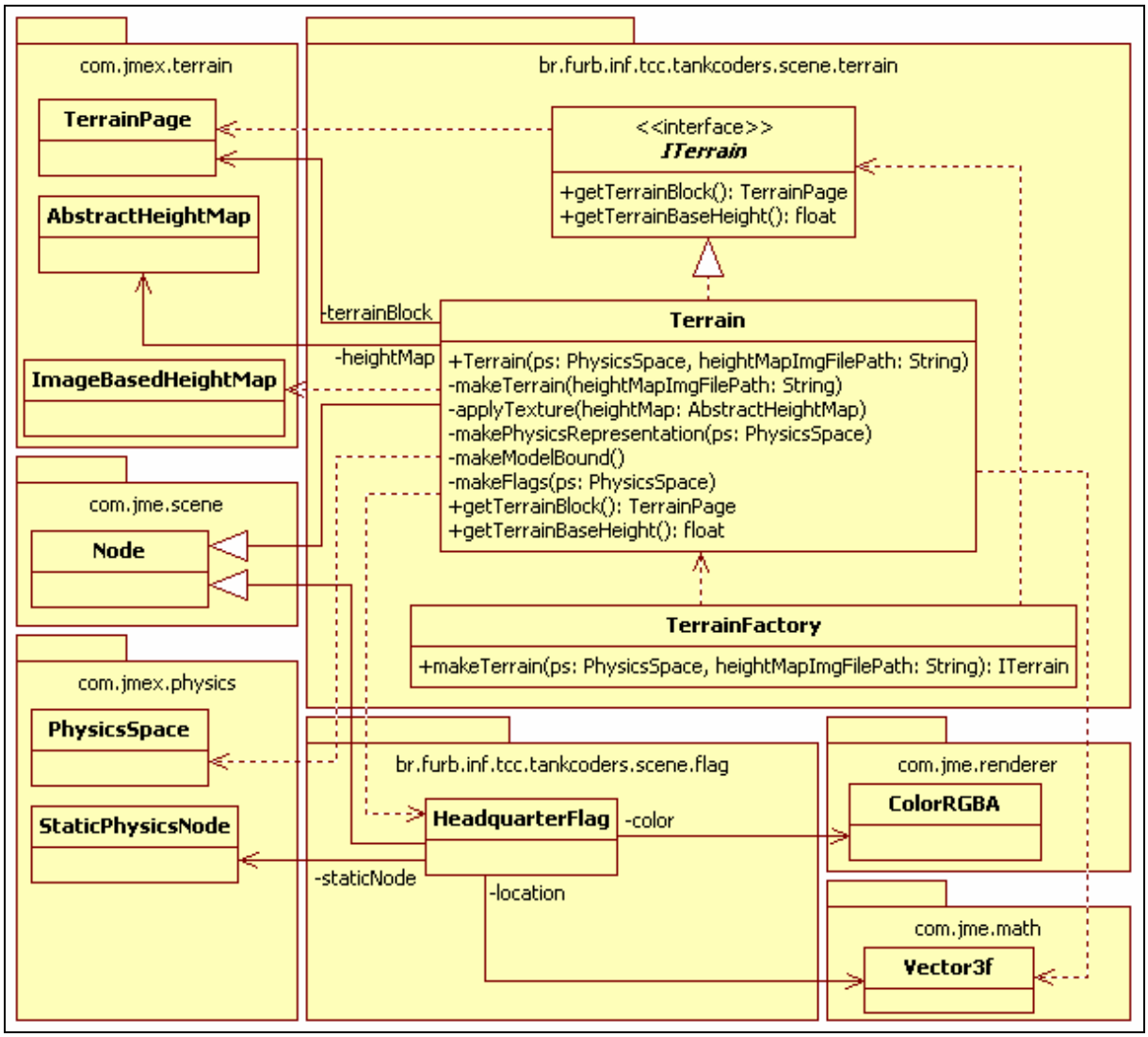



Figura 21 – Diagrama de classes envolvendo a criação do terreno

A criação do terreno é feita através da classe `TerrainFactory`, que segue o padrão de projeto *Factory Method*. Essa classe, juntamente com a interface `ITerrain`, provê uma abstração em relação à criação da classe concreta que implementa de fato o terreno (classe `Terrain`). Somente a classe `TerrainFactory` possui dependência direta da classe `Terrain`, nos demais pontos do código somente é disponibilizado a interface dela, que é a interface `ITerrain`. Dessa forma, no momento em que uma nova implementação de terreno for criada, basta alterar a classe que é instanciada na fábrica de objetos de terreno, que ela já passa a ser considerada em todos os demais pontos do código.

A classe `HeadquarterFlag` representa a bandeira do quartel general de um determinado time. O terreno representado pela classe `Terrain` contém quatro bandeiras, duas azuis e duas vermelhas, ou seja, duas para cada time.

Na Figura 20, o último elemento a ser comentado e detalhado é o tanque de guerra, representado pelos nós `TankM1Abrams` e `TankJadgePanther`. Os parágrafos seguintes,

juntamente com a Figura 22, descrevem a composição de cada modelo de tanque, mostrando também como eles são criados.

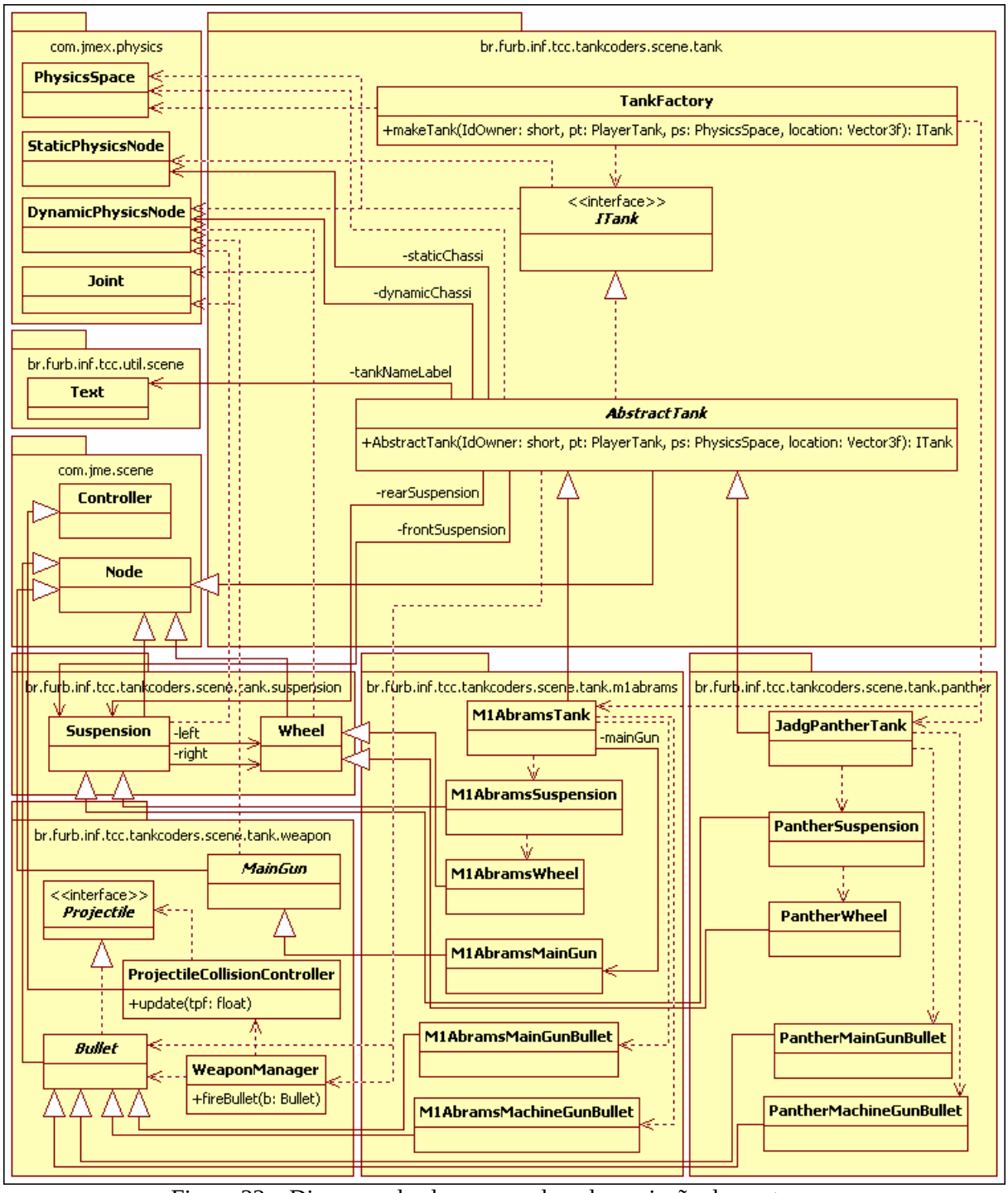



Figura 22 – Diagrama de classes envolvendo a criação de um tanque

Também seguindo o padrão de projeto *Factory Method*, a criação de tanques de guerra é feita através da classe `TankFactory` e da interface `ITank`. A fábrica de objetos de tanque (`TankFactory`), além de servir para a correta instanciação do modelo de tanque indicado pela enumeração `TankModel`, é usada para garantir a integração com o interpretador Jason, como é visto na seção 3.2.1.2.3.

A classe abstrata `AbstractTank` é usada para encapsular a criação dos elementos comuns entre os dois modelos de tanques considerados pelo simulador. Também é responsável por encapsular comportamentos comuns dos modelos, tal como acelerar, frear, girar para um lado e atirar.

Dentre os elementos comuns dos modelos de tanques (Figura 22) destacam-se:

a) `dynamicChassi`: representa o chassi do tanque através de um nó dinâmico controlado pelo *framework* de física. Esse nó é associado ao tanque quando ele pertencer ao jogador da instância local do simulador, ou seja, quando for controlado localmente;
b) `staticChassi`: representa o chassi do tanque através de um nó estático controlado pelo *framework* de física. Esse nó é associado ao tanque quando ele não pertencer ao jogador da instancia local do simulador, ou seja, controlado por um jogador remoto;
c) `frontSuspension` e `rearSuspension`: representam o esquema de suspensão do tanque. São adicionados somente quando o tanque for controlado pelo jogador da instância local do simulador;
d) `tankNameLabel`: representa o rótulo com o nome do tanque. Só é adicionado caso o tanque não seja o alvo da câmera (atributo `cameraAtThisTank`) no momento.

As suspensões são representadas pela classe abstrata `Suspension`, sendo que para cada modelo de tanque existe uma especialização específica. Cada suspensão é composta por duas bases (representados por nós dinâmicos) e em cada base é anexada uma roda, que é representada por uma classe concreta derivada de `Wheel`, dependendo do modelo do tanque.

Existe uma classe concreta para cada elemento e modelo de tanque, pois as classes abstratas contêm métodos abstratos que requisitam informações específicas de cada modelo que somente as classes concretas podem prover. Por exemplo, na classe abstrata `Wheel`, durante a sua configuração, é necessário saber qual massa aplicar no nó que representa a esfera da roda. Para isso, as classes `M1AbramsWheel` e `PantherWheel` sobrescrevem, entre outros métodos, o método `getWheelMass` que retorna um valor de ponto flutuante representando a massa da roda.

A classe `M1AbramsTank`, primeira classe concreta de tanque a ser comentada, representa o modelo indicado pela enumeração “`TankModel.M1Abrams`”. Esse modelo, além dos elementos comuns especificados na classe `AbstractTank`, possui uma arma principal (*main gun*) móvel. A arma principal desse modelo de tanque é representada pela classe

concreta `M1AbramsMainGun`, que por sua vez deriva da classe abstrata `MainGun`. O nó da arma principal, como também pode ser visto no grafo de cena da Figura 20, contém um nó dinâmico ou estático para representá-lo fisicamente. O nó dinâmico, da mesma forma que ocorre com o chassi do tanque, é anexado quando o tanque pertencer ao jogador da instância local do simulador. Caso o tanque pertença a um jogador remoto, é anexado o nó estático.

O segundo modelo de tanque, representado pela classe `JadgePantherTank`, consiste em um tanque mais simples, pois não possui uma arma principal móvel. A arma principal desse modelo é acoplada ao chassi, dessa maneira, não é necessária uma nova classe para representá-la.

A interface `Projectile` define as operações (métodos) de um projétil indefinido. Ela foi criada, pois futuramente podem ser criados outros elementos considerados projéteis que não são necessariamente balas de uma arma. Já a classe `Bullet`, que consiste em uma implementação da interface `Projectile`, representa de fato uma bala. Como pode ser visto no diagrama de classes da Figura 22, `Bullet` deriva da classe `Node`, logo pode ser anexada ao grafo de cena.

Existem dois tipos de balas implementadas no simulador. A primeira delas serve para ser disparada pela arma principal e é representada pelas classes concretas `M1AbramsMainGunBullet` e `PantherMainGunBullet`. O segundo tipo de bala serve para ser disparado pela arma secundária, chamada de metralhadora. As classes concretas que representam esse tipo são `M1AbramsMachineGunBullet` e `PantherMachineGunBullet`.

Qualquer bala, não importando o modelo do tanque, sempre é disparada através do método estático `fireBullet` da classe `WeaponManager`. Essa classe provê um ponto central para o lançamento de qualquer projétil que parta da arma de um tanque. A classe `ProjectileCollisionController` é utilizada para encapsular o processo de detecção de colisão com outros objetos dinâmicos ou estáticos do simulador.

O diagrama contido na Figura 23 apresenta as classes utilizadas para o controle de um tanque e da câmera via teclado. Tratam-se de classes que implementam a interface `InputActionInterface` para que possam ser associadas à eventos de teclado do componente *input*, contido na classe `InGameState`.

A classe `ActionAccelerator` é utilizada para mover o tanque para frente e para trás. Já as classes `ActionDirection` e `ActionMainGunDirection` são usadas, respectivamente para definir a direção do tanque e direção da arma principal. Sendo que a classe da arma principal (`ActionMainGunDirection`) só pode ser utilizada com o modelo *M1Abrams*.

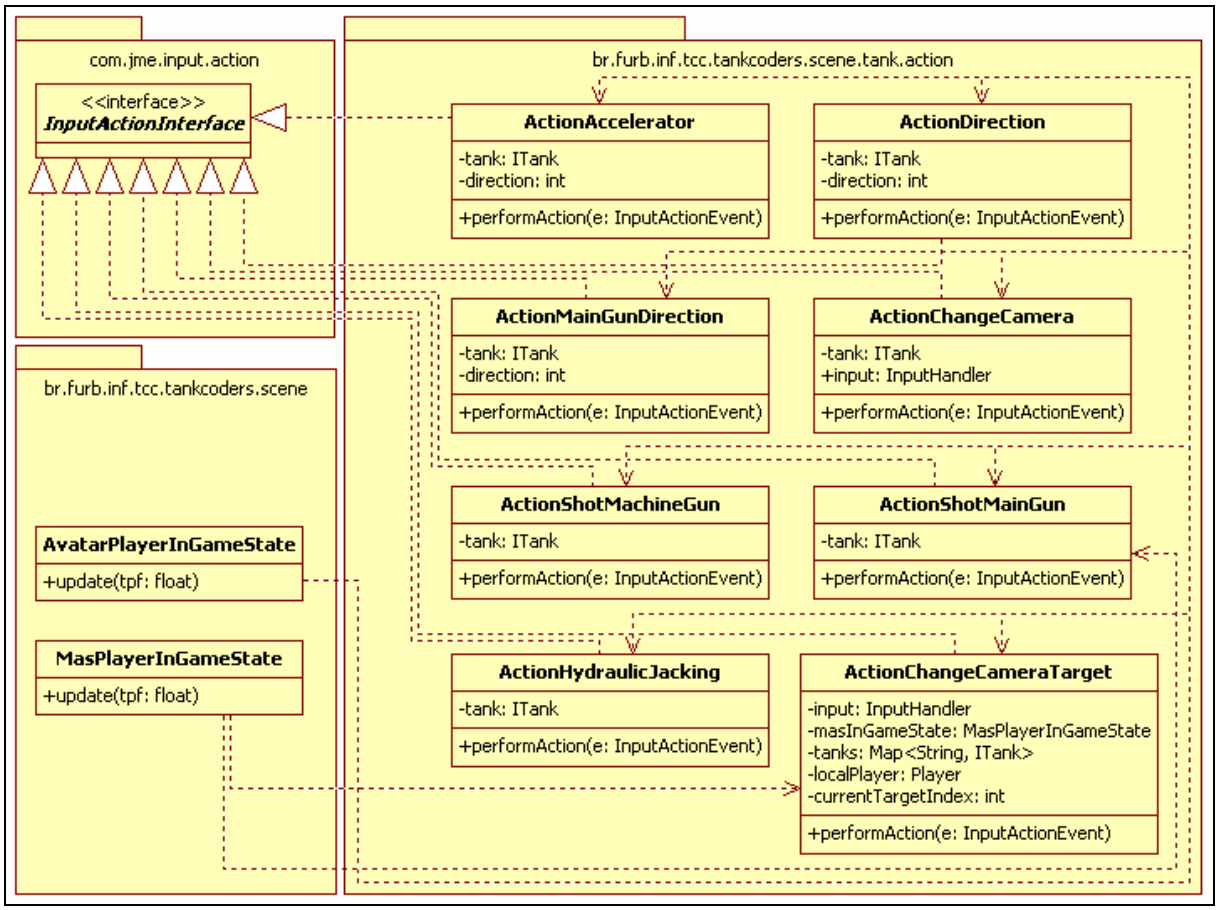



Figura 23 – Diagrama de classes das ações de controle do tanque e da câmera

As classes `ActionShotMainGun` e `ActionShotMachineGun` são utilizadas para disparar tiros com a arma principal e com a metralhadora, respectivamente. A ação representada pela classe `ActionHydraulicJacking` é responsável por ligar e desligar o dispositivo denominado "macaco hidráulico". Esse dispositivo consiste na adição de uma força ao chassi do tanque, fazendo-o flutuar (a adição dessa força é semelhante à força aplicada no disparo de uma bala, detalhes são apresentados na seção 3.3.2.2.1). Pode ser útil quando o tanque leva um tiro e permanece com as rodas viradas para cima, sem poder se movimentar.

Todas as classes de ações citadas até o momento servem somente para o *game state* do tipo de jogador *avatar* (`AvatarPlayerInGameState`), pois tratam-se de controles disponibilizados ao jogador via teclado.

A classe `ActionChangeCamera` é utilizada para alternar o alvo da câmera entre o chassi e a arma principal. Essa ação pode ser utilizada pelos dois tipos de *game state* "em jogo", pois consiste somente na troca do corrente alvo da câmera.

A classe `ActionChangeCameraTarget`, utilizada somente pelo *game state* representado pela classe `MasPlayerInGameState`, é responsável por alternar o alvo da câmera entre todos os tanques de um time de agentes.

#### 3.2.1.2.2 Interação com o aplicativo servidor

Outra responsabilidade do aplicativo cliente é manter a sua cópia local do AVD atualizada em relação às modificações que ocorrem nos jogadores remotos. Essas modificações são classificadas de acordo com o *game state* que elas ocorrem, podendo ser modificações de "menu" (`GameMenuState`) ou "em jogo" (`InGameState`).

As modificações consistem em classes de mensagens derivadas das classes bases do *middleware* JGN. Todas as classes de mensagens do simulador, com exceção das mensagens de sincronização dos tanques (explicadas em detalhes adiante), derivam da classe `RealtimeMessage` e implementam a interface `PlayerMessage`. Dessa forma, as mensagens são enviadas através do canal de comunicação com protocolo UDP. Caso houvesse a necessidade de enviar mensagens através do protocolo TCP, bastaria adicionar a interface `CertifiedMessage` à classe de mensagem.

O Quadro 12 apresenta uma relação de todas as classes de mensagens do simulador, definindo a classificação e o sentido de cada uma delas.

| **CLASSE** | **CLASSIFICAÇÃO** | **SENTIDO** |
|---|---|---|
| *AbstractGameMessage* | Ambos | Ambos |
| AllPlayersAreInGameState | Em jogo | Servidor -> Cliente |
| AnotherPlayerChangeModelResponse | Menu | Servidor -> Cliente |
| AnotherUserLogonRespose | Menu | Servidor -> Cliente |
| ChangeModel | Menu | Cliente -> Servidor |
| ChangeTeam | Menu | Cliente -> Servidor |
| InvalidChangeTeamResponse | Menu | Servidor -> Cliente |
| PlayerChangeTeamResponse | Menu | Servidor -> Cliente |
| PlayerInBattle | Em jogo | Cliente -> Servidor |
| PlayerLeftGame | Em jogo | Servidor -> Cliente |
| StartBattle | Menu | Servidor -> Cliente |
| Synchronize3DMessage | Em jogo | Cliente -> *broadcast* |
| SynchronizeArticulatedObject3DMessage | Em jogo | Cliente -> *broadcast* |
| SynchronizeCreateTankMessage | Em jogo | Cliente -> *broadcast* |
| TankActionShotMachineGun | Em jogo | Cliente -> *broadcast* |
| TankActionShotMainGun | Em jogo | Cliente -> *broadcast* |
| TankBulletHit | Em jogo | Cliente -> *broadcast* |
| TankDead | Em jogo | Cliente -> *broadcast* |
| TeamLeftGame | Em jogo | Cliente -> *broadcast* |
| UserLogoff | Menu | Cliente -> Servidor |
| UserLogon | Menu | Cliente -> Servidor |
| UserLogonFailedResponse | Menu | Servidor -> Cliente |
| UserLogonResponse | Menu | Servidor -> Cliente |
| UserReady | Menu | Cliente -> Servidor |
| UserUnready | Menu | Cliente -> Servidor |

Quadro 12 – Relação de classes de mensagens do simulador

A classe abstrata `AbstractGameMessage` consiste em uma classe base para todas as mensagens do simulador. É essa classe que deriva de `RealtimeMessage` e implementa a interface `PlayerMessage`. Todas as classes concretas de mensagens do simulador, com exceção das mensagens de sincronização dos tanques, derivam de `AbstractGameMessage`.

As mensagens que ocorrem durante o *game state* "menu", no aplicativo cliente, são recebidas na classe `PrepareBattleMenuGUI`. Quando um jogador conecta-se ao servidor, a primeira mensagem enviada é a `UserLogon`, levando consigo o nome, o tipo e os tanques (instâncias de `PlayerTank`) do jogador. Ao receber essa mensagem, o aplicativo servidor pode responder com a mensagem `UserLogonResponse`, abrigando consigo uma lista de todos os jogadores já conectados. Para que esses jogadores já conectados tomem conhecimento do novo jogador conectado, é enviada a mensagem `AnotherUserLogonRespose`, carregando as mesmas informações que a mensagem `UserLogon`. Porém o servidor também pode responder com a mensagem `UserLogonFailedResponse`, sendo que esta consiste em uma falha na entrada do jogador na batalha, carregando consigo o motivo do problema.

Quando o usuário fecha o jogo ou decide sair do menu de preparação da batalha, a mensagem `UserLogoff` é enviada ao servidor, fazendo com que o usuário saia da lista de jogadores conectados. Quando isso ocorre, internamente o *middleware* JGN envia um evento de desconexão de jogador a todos os demais jogadores. Esse evento é capturado através do *listener* representado pela interface `JGNConnectionListener`.

Um jogador pode também decidir trocar o modelo dos seus tanques. Quando isso ocorre a mensagem `ChangeModel` é enviada levando consigo o nome do tanque e o novo modelo a ser usado. Ao receber essa mensagem, o aplicativo servidor envia a todos os demais jogadores a mensagem `AnotherPlayerChangeModelResponse`, carregando as mesmas informações da mensagem `ChangeModel`.

Além de trocar o modelo, o jogador pode também trocar o time que cada um de seus tanques pertence. Para realizar isso, a mensagem `ChangeTeam` é enviada abrigando o nome do tanque e o novo time que ele deve fazer parte. Essa tarefa precisa ser aprovada pelo servidor, pois existe um limite máximo de tanques de guerra em cada time, que é configurado no servidor antes de iniciar a batalha. Caso o servidor aprove a troca de time, é enviada ao jogador a mensagem `PlayerChangeTeamResponse`, carregando consigo a nova posição (*slot*) inicial a ser ocupada pelo tanque no campo de batalha. Porém se a troca de time não for aprovada, a mensagem `InvalidChangeTeamResponse` é enviada ao jogador.

Quando o usuário termina de configurar o modelo e o time de seus tanques, é

necessário notificar o servidor que o jogador está pronto para batalhar. Isso é feito através do envio da mensagem `UserReady`. O usuário pode também determinar que ainda não está pronto para jogar, para isso a mensagem `UserUnready` é enviada ao servidor.

Toda vez que o servidor recebe a mensagem `UserReady`, ele verifica se por acaso todos os jogadores já enviaram essa mensagem, ou seja, todos prontos para batalhar. Quando isso ocorre, a mensagem `StartBattle` é enviada à todos os jogadores para que eles possam dar início ao processo de carregamento do cenário 3D, entrando no *game state* "em jogo".

Ao entrar no *game state* "em jogo", as mensagens não são mais recebidas na classe `PrepareBattleMenuGUI`, mas sim na classe `NVEHandler`. A Figura 24 apresenta a especificação das classes que envolvem o tratamento das mensagens que ocorrem durante o *game state* "em jogo".

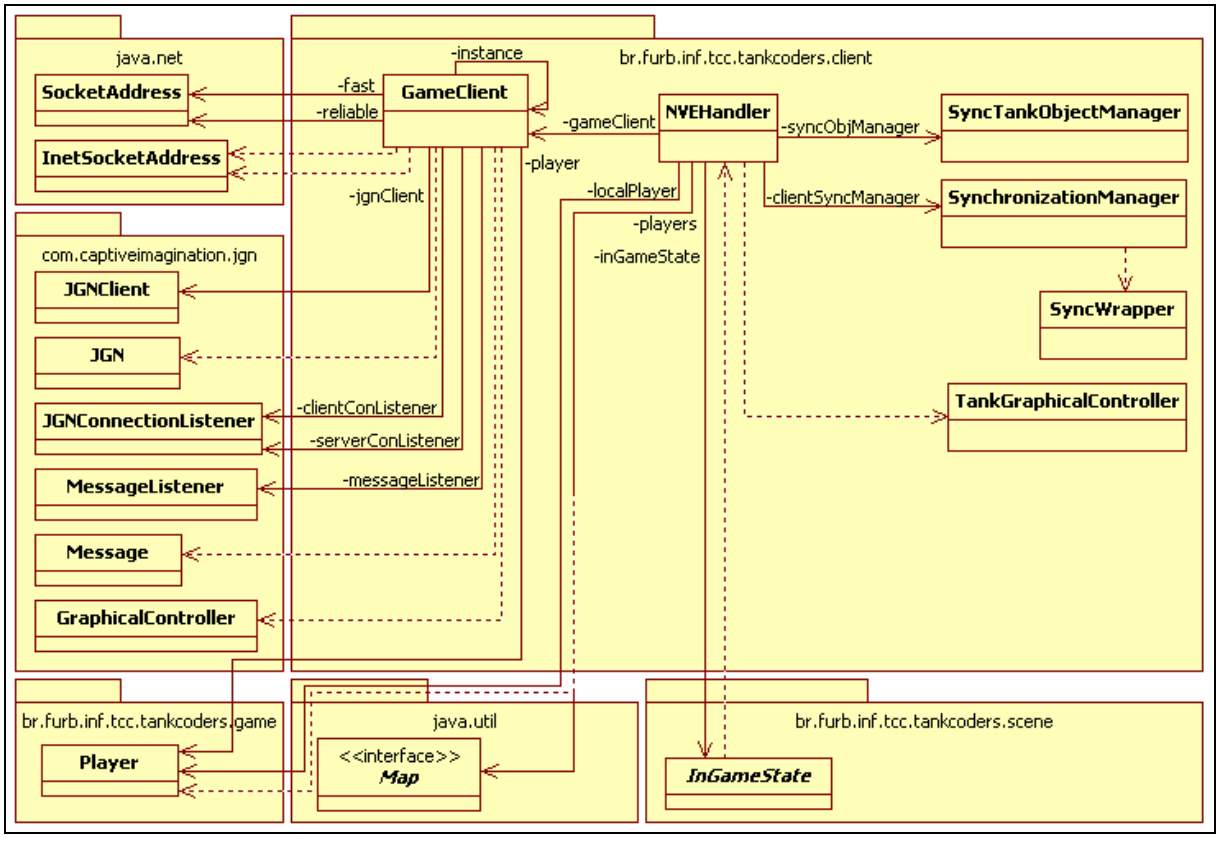



Figura 24 – Diagrama de classes do pacote que trata mensagens do `InGameState`

A classe `GameClient`, já utilizada no *game state* menu, é utilizada para interagir diretamente com o *middleware* JGN, provendo maior abstração no envio e recebimento de mensagens. Essa classe também segue o padrão de projetos *Singleton*, ou seja, somente uma instância dessa classe é criada durante a aplicação. A instância é obtida através do método `getInstance`, que segue o padrão de "instanciação preguiçosa" (do inglês, *lazy instantiation*). Esse padrão determina que uma nova instância somente seja criada, caso a corrente instância

ainda seja nula, garantindo o padrão *Singleton*.

A classe `NVEHandler` representa o ponto central para realizar a notificação de modificações ocorridas no estado local do AVD. Todos os métodos dessa classe são estáticos, tornando a utilização deles mais prática e rápida em qualquer ponto do código. Também é responsabilidade dessa classe, registrar os *listeners* que tratam os eventos de conexão (com o servidor e os demais clientes) e eventos de mensagem. Como pode ser visto na Figura 24, a classe `NVEHandler` contém um atributo do tipo `InGameState` e também um mapa (interface `Map`) referenciando os tanques de guerra da batalha. Dessa maneira, ao receber uma mensagem, a classe `NVEHandler` possui todos os recursos necessários para processá-la.

A classe `SynchronizationManager` é uma implementação customizada da classe `SynchronizationManager` padrão do *middleware* JGN. Ela é responsável por gerenciar a sincronização dos objetos registrados, que no caso do simulador são os tanques de guerra. Essa classe implementa também a interface `Updatable`, que define um método chamado `update`. Dessa forma, um objeto dessa classe é adicionado a uma *thread*, que a cada ciclo invoca o método `update` sobrescrito.

A classe `SyncWrapper`, que também consiste em uma customização da classe `SyncWrapper` padrão da JGN, é responsável por criar as mensagens de sincronização de objetos e enviá-las através da técnica de *broadcast*. Cada objeto (tanque de guerra) a ser gerenciado pela classe `SynchronizationManager` possui uma respectiva instância de classe `SyncWrapper`.

`SyncTankObjectManager` é a classe responsável por retornar as instâncias dos tanques dos jogadores remotos. No inicio da batalha, todos os tanques (representados pelas classes concretas `M1AbramsTank` e `JadgePantherTank`) são instanciados, independentemente do proprietário do tanque (se é o jogador da instância local ou um jogador remoto). Porém, como é visto adiante, cada tanque de guerra do jogador da instância local é registrado na classe `SynchronizationManager`, fazendo com que uma nova mensagem (representada pela classe `SynchronizeCreateTankMessage`) seja enviada à cada jogador notificando o registro de cada tanque. Ao receber essa mensagem, o `SynchronizationManager`, através do método `create` da classe `SyncTankObjectManager`, retorna a instância do tanque para que possa ser associada internamente. Tendo todos os tanques remotos associados internamente, a classe `SynchronizationManager` consegue aplicar nas instâncias de tanques locais as modificações ocorridas nos tanques remotos, fazendo uso da classe `TankGraphicalController`.

A classe `TankGraphicalController` é utilizada por dois propósitos. O primeiro deles

é prover instâncias de mensagens de sincronização dos atributos de um determinado tanque. As mensagens de sincronização são representadas pelas classes `Synchronize3DMessage` (translação e rotação do tanque) e `SynchronizeArticulatedObject3DMessage` (translação e rotação do tanque, juntamente com a rotação da arma principal articulada), sendo que a primeira é usada com o modelo de tanque "Jadge Panther" e o segunda com o modelo "M1 Abrams". O segundo propósito, como comentado no parágrafo anterior, é para aplicar os atributos de um determinado tanque remoto (recebido como uma mensagem) na respectiva instância do tanque local.

Quando o aplicativo cliente termina de carregar o cenário 3D e configurar a classe `NVEHandler`, é enviada ao servidor a mensagem `PlayerInBattle`. A cada mensagem desse tipo recebida, o servidor verifica se todos os jogadores já a enviaram. Se isso ocorrer, o servidor envia a todos os jogadores a mensagem `AllPlayersAreInGameState`, fazendo com que os controles do tanque sejam liberados para o início da batalha.

Durante a batalha, é possível atirar com a arma principal ou com a metralhadora de um tanque. Para notificar esse evento, é enviada a todos os demais jogadores (via *broadcast*) a mensagem a `TankActionShotMainGun` ou `TankActionShotMachineGun`, dependendo do tipo da arma. Ao receber uma dessas mensagens, o aplicativo cliente dispara um tiro através da arma utilizada do tanque cujo nome é informado no corpo da mensagem recebida. Um tiro remoto, originado pelo recebimento de uma mensagem, somente é disparado por efeitos de visualização, pois não afetam os tanques. O controle de colisão da bala com os demais elementos do cenário ocorre somente na instância do simulador do jogador que disparou o tiro. Dessa forma, para notificar a colisão da bala com outro tanque de guerra, é enviada ao jogador proprietário a mensagem `TankBulletHit`, levando consigo o nome do tanque acertado e o poder de destruição da bala. Ao receber essa mensagem, o aplicativo cliente pode atualizar os pontos de vida (saúde) do tanque acertado baseado no poder de destruição da bala. Caso os pontos de vida do tanque seja inferior ou igual a zero, esse é considerado "morto". Quando isso ocorre, é enviado aos demais jogadores a mensagem `TankDead`.

Durante a batalha, também é possível que um jogador abandone a batalha. Caso isso ocorra, a mensagem `PlayerLeftGame` é enviada aos demais jogadores para que todos os tanques do jogador sejam eliminados do cenário. Além disso, cada vez que um jogador abandona a batalha, o servidor verifica se ainda resta algum tanque nos dois times. Caso algum time fique sem nenhum tanque, a mensagem `TeamLeftGame` é enviada aos jogadores para que a batalha seja interrompida, notificando o time vencedor.

#### 3.2.1.2.3 Integração com o interpretador Jason

Essa seção trata da última grande subdivisão do aplicativo cliente, que consiste na integração com o interpretador Jason versão 1.1.1. Essa integração é possível, devido à possibilidade de se estenderem determinados componentes pré-implementados na API do Jason. Os componentes personalizados na integração em questão são relacionados à classe de arquitetura (`AgArch`) e à classe de ambiente (`Environment`).

A Figura 25 apresenta o diagrama das classes que representam a integração com o interpretador Jason. Em seguida é feita uma reflexão a respeito das responsabilidades de cada uma delas.

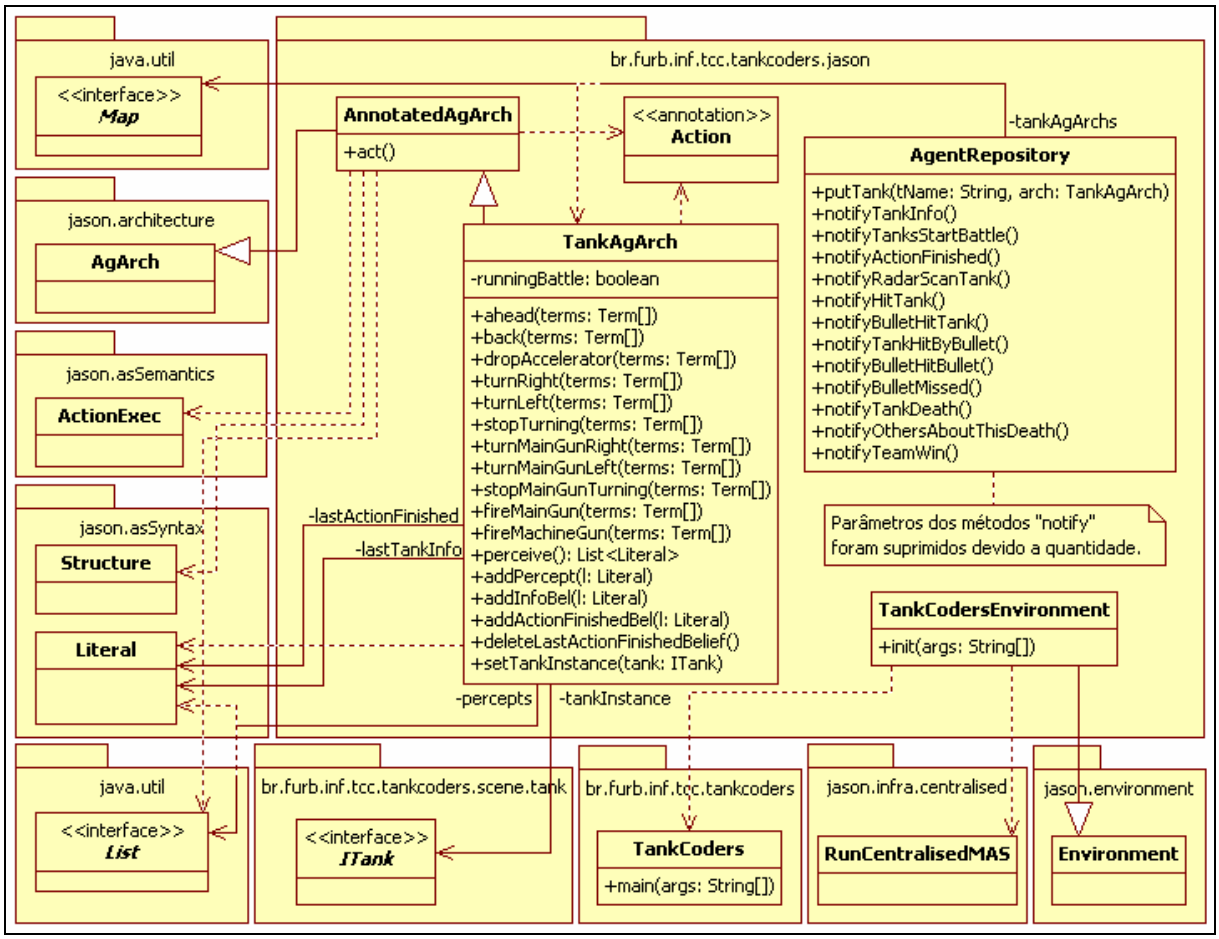



Figura 25 – Diagrama de classes da integração com o interpretador Jason

A anotação[12] `Action` é utilizada para definir quais métodos da classe de arquitetura do simulador são ações que podem ser invocadas pelos agentes durante a execução do SMA.

[12] Anotação é uma construção que surgiu na versão 1.5 da linguagem Java. São utilizadas para prover informações (de tempo de compilação ou tempo de execução) a respeito de um atributo, método ou classe Java. Mais informações sobre essa construção podem ser vistas em Annotations (2008).

A classe `AnnotatedAgArch` é a base para a classe final de arquitetura do simulador (`TankAgArch`). Sua responsabilidade é invocar o respectivo método que representa uma ação invocada por um agente do SMA.

A classe `TankAgArch` é, de fato, a classe que representa a arquitetura de um agente que controla um tanque de guerra no simulador. Ela é responsável por interagir diretamente com a instância do tanque (objeto do tipo `M1AbramsTank` ou `JadgePantherTank`). Quando um agente executa uma ação, a classe `TankAgArch` age diretamente sobre o tanque que ela controla (através do atributo `tankInstance`). A cada ciclo interno, o interpretador Jason requisita a cada agente, através da respectiva classe de arquitetura (método `perceive`), uma lista de percepções, que são eventos específicos que ocorrem no ambiente e que devem ser repassados aos agentes. Os eventos específicos são adicionados como percepções na classe de arquitetura através da classe `AgentRepository`.

A classe `AgentRepository` representa um ponto central para adicionar-se percepções e crenças aos agentes. Ela provê um método para cada diferente tipo de evento que pode ocorrer durante uma batalha. Os métodos para esse propósito contidos nessa classe são:

a) `notifyTankInfo`: usado para notificar as correntes informações a respeito de um determinado tanque de guerra. Essa crença é adicionada periodicamente dentro da classe abstrata `AbstractTank`;
b) `notifyTanksStartBattle`: usado para notificar o inicio da batalha à todos os agentes do SMA;
c) `notifyActionFinished`: usado para notificar o término da execução de uma ação no ambiente;
d) `notifyRadarScanTank`: usado para notificar que um tanque foi avistado no radar;
e) `notifyHitTank`: usado para notificar que um tanque colidiu com outro;
f) `notifyBulletHitTank`: usado para notificar que uma bala acertou outro tanque;
g) `notifyTankHitByBullet`: usado para notificar que o tanque foi acertado por uma bala. Pode ser um tiro disparado por um inimigo ou um aliado;
h) `notifyBulletHitBullet`: usado para notificar que uma bala do tanque acertou outra bala;
i) `notifyBulletMissed`: usado para notificar que uma bala do tanque se perdeu, ou seja, sem ter colisão com algum objeto do cenário;
j) `notifyTankDeath`: usado para notificar que o tanque “morreu”;
k) `notifyOthersAboutThisDeath`: usado para notificar a “morte” de um tanque de

guerra aos demais tanques.

l) `notifyTeamWin`: usado para notificar que o time do tanque venceu a batalha;

Por último, a classe `TankCodersEnvironment`, que consiste na personalização da classe de ambiente padrão do Jason (classe `Environment`), é responsável por dar início ao aplicativo cliente do simulador (classe `TankCoders`). Ao iniciar o aplicativo cliente em modo MAS[13], é passada uma lista contendo o nome de todos os agentes do SMA que utilizam a classe de arquitetura `TankAgArch`, sendo que essa lista é obtida através da classe `RunCentralisedMAS` da API do interpretador Jason.

Como a integração com o interpretador Jason faz uso da classe `RunCentralisedMAS` para obter a lista de agentes do SMA, significa que a infra-estrutura do projeto Jason do jogador tipo MAS deve ser sempre centralizada (infra-estrutura *Centralised*). Essa limitação é necessária pelo fato de que uma classe de arquitetura interage diretamente com a instância de tanque de guerra. Dessa forma, se uma infra-estrutura distribuída (através do *SACI* ou *JADE*) fosse utilizada, não seria possível obter as instâncias da classe de arquitetura para associar às instancias dos tanques, pois elas estariam possivelmente em máquinas distintas.

## 3.3 IMPLEMENTAÇÃO

Nessa seção são apresentados os aspectos a respeito da implementação do simulador, bem como as técnicas e ferramentas utilizadas. Por último, é descrita a operacionalidade da implementação através de um estudo de caso.

### 3.3.1 Técnicas e ferramentas utilizadas

Para a implementação do simulador (de ambos os aplicativos especificados) na linguagem Java, foi utilizado o ambiente de desenvolvimento integrado Eclipse 3.3. Também foi usada a ferramenta Blender 2.43 para modelagem dos tanques de guerra 3D. Para a geração das texturas utilizadas no terreno do cenário 3D foi usada a ferramenta Adobe

[13] Iniciar o aplicativo cliente do simulador em modo MAS significa dizer que o jogador da instância local é do tipo MAS, ou seja, tendo seus tanques controlados por agentes BDI do interpretador Jason.

Photoshop CS.

Foi utilizada também a ferramenta Assembla[14] para gerenciar o andamento do projeto. Trata-se de uma ferramenta WEB com acesso gratuito, que provê utilitários como Wiki, Trac, repositório SVN, controle de metas (*milestones*) e controle de tarefas (*tickets*).

Por último, a ferramenta Apache Ant, juntamente com a linguagem de marcação XML, foi utilizada para implementar uma forma prática de executar as seguintes tarefas:

a) geração do arquivo Java ARchive (JAR) do simulador;
b) execução do aplicativo cliente do simulador;
c) execução do aplicativo servidor do simulador;
d) geração do Javadoc do código-fonte do simulador;
e) geração do arquivo ZIP de distribuição do simulador.

O Quadro 13 apresenta um trecho do documento XML (arquivo build.xml) que implementa as tarefas citadas acima.

Durante o desenvolvimento do simulador, foram utilizados quatro padrões de projetos. Sendo que três deles são padrões definidos pela Gangue dos Quatro (*Gang Of Four* – GOF) e o outro é um padrão definido pela Sun Microsystems na especificação da plataforma Java EE. Os padrões utilizados foram *Singleton* (GOF), *Command* (GOF), *Factory Method* (GOF) e *Transfer Object* (Java EE *core patterns*). O Quadro 14 apresenta exemplos de códigos desenvolvidos no simulador usando os padrões citados.

[14] Disponível em http://www.assembla.com/.

```
...
<target name="jar">
	<jar jarfile="${tankCodersClientJar}" manifest="${basedir}/MANIFEST.MF">
		<fileset dir="${build.dir}">
			<include name="**/*.class" />
		</fileset>
	</jar>
</target>

<target name="runClient">
	<java dir="bin" classname="br.furb.inf.tcc.tankcoders.TankCoders"
	    fork="yes">
		<arg value="-debug"/>
		<jvmarg value="-Djava.library.path=../lib/native"/>
		<classpath refid="project.classpath" />
	</java>
</target>

<target name="runServer">
	<java dir="bin" classname="br.furb.inf.tcc.gui.ServerGUI" fork="yes">
		<classpath>
			<pathelement location="lib/looks-2.1.4.jar"/>
			<pathelement location="lib/jgn.jar"/>
			<pathelement location="lib/jme.jar"/>
			<pathelement location="./bin"/>
			<pathelement path="${java.class.path}"/>
		</classpath>
	</java>
</target>

<target name="javadoc">
	<javadoc access="public"
	   author="true"
		destdir="doc/api"
		doctitle="TankCoders API documentation"
		nodeprecated="false"
		nodeprecatedlist="false"
		noindex="false"
		nonavbar="false"
		notree="false"
		packagenames="*"
		source="1.6"
		sourcepath="src/server;rsrc;src/client"
		splitindex="true"
		use="true"
		version="true">
		<classpath refid="project.classpath"/>
	  </javadoc>
    </target>

<target name="zip">
	<delete dir="${dist.dir}"/>
	<mkdir dir="${dist.dir}"/>

	<zip destfile="${tankCodersDistZip}">
		<fileset dir="${basedir}">
			<exclude name=".settings"/>
			<exclude name=".project"/>
			<exclude name=".classpath"/>
			<exclude name="dist"/>
			<exclude name="bin/**/*"/>
		</fileset>
	</zip>
</target>
```

Quadro 13 – Trecho do documento build.xml para execução de tarefas via Ant

```
////////////////////////////// Singleton ////////////////////////////////
public class GamePersistentProperties {
    ...
    // singleton instance.
    private static GamePersistentProperties instance;

    // construtor privado para ser acessado somente dentro da classe.
    private GamePersistentProperties() {
        ...
    }

    public static GamePersistentProperties getInstance() {
        // lazy instantiation.
        if (instance == null) {
            instance = new GamePersistentProperties();
        }
        return instance;
    }
    ...
}
/////////////////////////////////////////////////////////////////////////

////////////////////////////////// Command //////////////////////////////////
public interface InternalServerListener {

    public void execute(Object ... params);

}
/////////////////////////////////////////////////////////////////////////

///////////////////////////// Factory Method ////////////////////////////
public class TerrainFactory {

    public static ITerrain makeTerrain(PhysicsSpace pSpace,
                                       String heightMapImageFilePath) {
        // instancia a classe concreta de terreno.
        return new Terrain(pSpace, heightMapImageFilePath);
    }
}
/////////////////////////////////////////////////////////////////////////

///////////////////////////// Transfer Object ////////////////////////////
public class OnlinePlayer implements Serializable {

    private short playerId;
    private String playerName;
    private PlayerType playerType;
    private String ipAddress;
    private PlayerTank[] tanks;
    private boolean readyToPlay;
    private boolean playing;

    ... // getters e setters suprimidos.
}
/////////////////////////////////////////////////////////////////////////
```

Quadro 14 – Trechos de código utilizando os padrões *Singleton*, *Command*, *Factory Method* e *Transfer Object*

### 3.3.2 Implementação do simulador

Seguindo as mesmas divisões de seções contidas na especificação do simulador, essa seção demonstra primeiramente como foi implementado o aplicativo servidor. Em seguida é descrito como foi implementado o aplicativo cliente, que por sua vez é subdivido em implementação da representação gráfica do AVD, implementação da interação com o aplicativo servidor e por último a implementação da integração com o interpretador Jason.

#### 3.3.2.1 Implementação do aplicativo servidor

Como comentado na especificação, o início do aplicativo servidor é dado pela classe `ServerGUI` através do método estático `main`. O Quadro 15 apresenta trecho de código que realiza a inicialização do aplicativo servidor.

No método `main`, através da chamada `SwingUtilities.invokeLater`, a *thread* representada pela classe interna anônima que implementa a interface `Runnable`, é executada assincronamente dentro da *thread* de eventos AWT. Ao executar a *thread*, a primeira instrução executada é a instanciação da classe `ServerGUI`. Dentro do construtor dessa classe, é invocado o método que inicializa os componentes de interface gráfica (método `initGUI`). Em seguida é instanciada a classe `GameServer`, cuja implementação é detalhada na seqüência. Por último são registrados os objetos (*listeners*) responsáveis por tratar os eventos que podem ocorrer durante a execução da batalha no simulador. O método `setLogonEventListener` da classe `GameServer` recebe por parâmetro um objeto de uma classe que implementa a interface `InternalServerListener`. Uma implementação dessa interface pode ser vista na classe interna anônima implementada e instanciada dentro da lista parâmetros na chamada do método `setLogonEventListener`. O método `execute` dessa classe interna anônima é executado quando um jogador conecta-se ao servidor, sendo que o exato momento da invocação dessa método é visto mais adiante na descrição da classe `GameServer`.

O método `initGUI` da classe `ServerGUI`, além de criar e posicionar os componentes na tela, também registra os objetos (*listeners*) responsáveis por capturar eventos deles. Dentre os principais eventos que podem ocorrer na tela, o mais importante consiste no evento do botão que inicia ou encerra uma batalha (botão representado pelo atributo `bStartServer`). O Quadro 16 apresenta o trecho de código que é executado quando o evento ocorre.

```
public class ServerGUI {

    ... // atributos dos componentes de interface gráfica Swing.

    private GameServer server;
    private int currentAvailablePlayers = GameRulesConstants.MAX_TANKS;

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                 ServerGUI inst = new ServerGUI();
                 inst.setLocationRelativeTo(null);
                 inst.setVisible(true);}
             });
    }

    public ServerGUI() {
        super();
        initGUI();
        server = new GameServer();
        registerInternalServerListeners();
    }

    private void initGUI() {
        ... // código responsável pela criação e posicionamento dos componentes
        // da interface gráfica Swing.
    }

    private void registerInternalServerListeners() {
        // on player login.
        server.setLogonEventListener(
            new InternalServerListener() {

                public void execute(Object... params) {
                     // add player in the table.
                     OnlinePlayer op = (OnlinePlayer)params[0];
                     DefaultTableModel dataModel =
                             (DefaultTableModel)tOnlinePlayers.getModel();
                     dataModel.addRow(new String[]{
                                       String.valueOf(op.getPlayerId()),
                                       op.getPlayerName(),
                                       String.valueOf(op.getTanks().length),
                                       op.getIpAddress(),
                                      "Normal"});
                     setLastEvent(
                            new Formatter().format(
                                  GameLanguage.getString("server.playerLoggedIn"),
                                  op.getPlayerName()).toString()
                            );
                 }
            }
         );

        server.setLogoffEventListener( ... );

        server.setUserReadyEventListener( ... );

        server.setUserUnReadyEventListener( ... );

        server.setServerStatusChangedListener( ... );

        server.setUserInBattleEventListener( ... );
    }
    ...
}
```

Quadro 15 – Trecho de código da inicialização do aplicativo servidor

```
...
try {
    if (server.isRunning()) {
        server.stopCurrentBattle();

        changeEnableFormControls(true);
        clearFormControls();

        bStartServer.setText(GameLanguage.getString("server.startServer"));
        setTitle(APP_TITLE + " - " + GameLanguage.getString("server.stopped"));
        setLastEvent(GameLanguage.getString("server.serverStopped"));
    }
    else {
        GameBattle battle = new GameBattle();
        battle.setBattleName(tfBattleName.getText());
        battle.setTeam1Name(tfTeam1Name.getText());
        battle.setTeam2Name(tfTeam2Name.getText());
        battle.setTeam1MaxTanks((Integer)sMaxTanksTeam1.getValue());
        battle.setTeam2MaxTanks((Integer)sMaxTanksTeam2.getValue());
        battle.setTerrainHeightMapImage((String)cbTerrain.getSelectedItem());
        server.startNewBattle(battle);

        changeEnableFormControls(false);

        bStartServer.setText(GameLanguage.getString("server.stopServer"));
        setTitle(APP_TITLE + " - " +
                        GameLanguage.getString("server.waitingForConnections"));
        setLastEvent(GameLanguage.getString("server.serverStarted"));
    }
} catch (IOException e1) {
    e1.printStackTrace();
}
...
```

Quadro 16 – Trecho de código referente ao evento de inicio e encerramento de batalha

Como também pode ser verificado no Quadro 15, no construtor da classe `ServerGUI` é instanciada a classe `GameServer`. O fonte mostrado no Quadro 17 ilustra o construtor que é chamado, bem como todos os atributos que compõem a classe `GameServer`.

Ao instanciar a classe `GameServer`, nota-se que é associado ao atributo `serverStatus` a enumeração `ServerStatus.AVAILABLE`. Porém, no momento da instanciação o servidor ainda não está disponível, de fato, para ser acessado. Essa atribuição só ocorre para que o atributo `serverStatus` não represente um valor nulo. Durante a execução do servidor, a troca de status do servidor ocorre sempre através do método `setServerStatus`.

O Quadro 18 ilustra um trecho da implementação da enumeração `ServerStatus`, que representa todos os status possíveis do servidor. Na sua definição, cada literal de enumeração recebe por parâmetro uma *string* contendo a chave para acessar a mensagem no arquivo de internacionalização (obtida através da classe `GameLanguage`). Essa chave é armazenada no atributo `name` da enumeração.

```
public class GameServer {
    // Ports contants
    public static final int RELIABLE_PORT = 1000;
    public static final int FAST_PORT = 2000;
    public static final int FINDSERVER_PORT = 2001;

    // sockets
    private SocketAddress serverReliableAddress;
    private SocketAddress serverFastAddress;

    // servers
    private JGNServer gameServer;
    private DatagramSocket findServerSocket;

    // internal controls
    private boolean running;
    private ServerStatus serverStatus;
    private GameBattle currentBattle;

    // listeners
    private InternalServerListener logonEventListener;
    private InternalServerListener logoffEventListener;
    private InternalServerListener userReadyEventListener;
    private InternalServerListener userUnReadyEventListener;
    private InternalServerListener userInBattleEventListener;
    private InternalServerListener serverStatusChangedListener;

    // current listeners
    private JGNConnectionListener clientConnectionListener;
    private MessageListener messageListener;

    // used to control the IDs of syncronization managers of clients.
    private Queue<Short> used;

    private static Logger logger;
    private static boolean debug;

    public GameServer() {
        running = false;
        serverStatus = ServerStatus.AVAILABLE;
        debug = true;

        prepareLogger();
    }

    ...
```

Quadro 17 – Atributos e construtor da classe `GameServer`

```
public enum ServerStatus {

      AVAILABLE("server.status.available"),
      INGAME("server.status.inGame"),
      FULL("server.status.full");

      private final String name;

      ...
}
```

Quadro 18 – Enumeração `ServerStatus`

No Quadro 16 pode ser visto também que, caso não tenha nenhuma batalha em execução, uma nova batalha é criada. Para isso, é instanciado um objeto da classe *JavaBean* `GameBattle` e em seguida são povoados os seus atributos para que seja invocado o método `startNewBattle` da classe `GameServer`. No Quadro 19 é mostrado o código-fonte desse método que é responsável por dar início a uma nova batalha, fazendo com que o aplicativo servidor entre no estado "*WaitingConnections*".

```
public void startNewBattle(GameBattle battle) throws IOException {
    running = true;
    setCurrentBattle(battle);
    setServerStatus(ServerStatus.AVAILABLE);

    setupFindServerSocket();
    setupGameServer();
}
```

Quadro 19 – Método `startNewBattle` da classe `GameServer`

O método `setupFindServerSocket` é responsável por criar um *socket* cujo objetivo é "escutar" uma porta específica, definida pela constante `FINDSERVER_PORT`, no protocolo UDP. Como é visto na descrição do aplicativo cliente, esse *socket* é utilizado para localizar os servidores ativos na rede local, através de mensagens *broadcast* UDP. A criação desse *socket* é feita dentro de uma nova *thread* no método `setupFindServerSocket`. O código do método `run` dessa *thread* é mostrado no Quadro 20.

```
...
findServerSocket = new DatagramSocket(FINDSERVER_PORT);
while (isRunning()) {
    logger.info("Waiting FindServer requests...");

    DatagramPacket question = new DatagramPacket(new byte[12], 12);
    try {
        findServerSocket.receive(question);
    } catch (SocketException e) {
        logger.info("Socket closed by user");
        return;
    }
    String questionStr = new String(question.getData());
    if (questionStr.equals("ServerStatus")) {
        ... // cria e define as variáveis a serem incorporadas ao buffer.
        byte[] buffer = (currentBattle.getBattleName() + "@@" +
                         statusStr + "@@" + t1Name + "@@" +
                         t2Name + "@@").getBytes();
        DatagramPacket dgp = new DatagramPacket(buffer,
        buffer.length, question.getAddress(), question.getPort());
        findServerSocket.send(dgp);
    }
}
...
```

Quadro 20 – Método `run` da *thread* que cria o *socket* de procura de servidores

Ainda no método `startNewBattle` visto no Quadro 19, após a execução do método `setupFindServerSocket`, é executado o método `setupGameServer`. Esse método é responsável por configurar e iniciar o servidor do simulador, fazendo uso do *middleware* JGN, como exibido no Quadro 21.

```
private void setupGameServer() throws IOException {
    serverReliableAddress = new InetSocketAddress(InetAddress.getLocalHost(),
                                                  RELIABLE_PORT);
    serverFastAddress = new InetSocketAddress(InetAddress.getLocalHost(),
                                              FAST_PORT);
    gameServer = new JGNServer(serverReliableAddress, serverFastAddress);

    addClientConnectionListener(
        new JGNConnectionListener() {
            public void connected(JGNConnection conn) {}
            public void disconnected(JGNConnection conn) {
                ... // código referente ao tratamento da desconexão de um jogador
            }
        }
    );

    addMessageListener(
        new MessageListener() {
            public void messageCertified(Message message) {}
            public void messageFailed(Message message) {}
            public void messageSent(Message message) {}
            public void messageReceived(Message message) {
                ... // código referente ao tratamento de mensagem recebida
            }
        }
    );

    JGN.createThread(gameServer).start();
    JGNRegisterUtil.registerTankCodersClassMessages();
}
```

Quadro 21 – Método `setupGameServer` da classe `GameServer`

Esse método configura um servidor JGN que cria dois canais de comunicação, um para o protocolo TCP (*reliable*) e outro para o UDP (*fast*). Nesse momento, são registrados também os objetos (*listeners*) que capturam eventos de conexão de jogadores e eventos de mensagem. O objeto que trata eventos de mensagem considera somente os eventos que ocorrem durante o estado "*WaitingConnections*". A partir do momento que o aplicativo servidor entrar no estado "InGame", outro objeto tratador de eventos de mensagem é registrado. Já o objeto que trata mensagens de conexão, considera eventos de ambos os estados, pois o procedimento a ser executado é praticamente o mesmo, como visto adiante.

Antes de finalizar a execução do método `setupGameServer`, é criada uma nova *thread* para o servidor JGN e em seguida já iniciada. Também são registradas as mensagens que podem ser recebidas ou enviadas através do servidor. Esse registro é necessário para que o servidor JGN só receba mensagens previamente informadas pelo desenvolvedor.

O Quadro 22 apresenta o código-fonte do método `disconnected` que foi suprimido no

Quadro 21 para melhorar a legibilidade. Esse método é responsável por retirar a instância do jogador de dentro da instância da batalha (`GameBattle`) e caso estiver no estado “InGame”, verifica se é necessário enviar mensagem aos clientes.

```
OnlinePlayer op = currentBattle.getOnlinePlayerById(conn.getPlayerId());
if (op != null) {
    currentBattle.removeOnlinePlayerById(op.getPlayerId());
    logoffEventListener.execute(op);

    if (serverStatus == ServerStatus.INGAME) {
        AbstractGameMessage m = null;
        if (currentBattle.getCurrentTanksCountTeam1() == 0) {
            m = new TeamLeftGame();
            ((TeamLeftGame)m).setTeam(PlayerTeam.TEAM_1);
        }
        else if (currentBattle.getCurrentTanksCountTeam2() == 0) {
            m = new TeamLeftGame();
             ((TeamLeftGame)m).setTeam(PlayerTeam.TEAM_2);
        }
        else {
            m = new PlayerLeftGame();
            m.setPlayerId(conn.getPlayerId());
        }

        gameServer.sendToAll(m);
    }
}
```

Quadro 22 – Método `disconnected`

Outro método suprimido no Quadro 21 por questões de legibilidade do código, foi o `messageReceived` da classe interna anônima passada ao método `addMessageListener`. O fonte desse método é mostrado no Quadro 23.

```
if (message instanceof UserLogon) {
    processUserLogonMessage((UserLogon)message);
}
else if (message instanceof UserLogoff) {
    processUserLogoffMessage((UserLogoff)message);
}
else if (message instanceof UserReady) {
    processUserReady((UserReady)message);
}
else if (message instanceof UserUnready) {
    processUserUnready((UserUnready)message);
}
else if (message instanceof ChangeTeam) {
    processChangeTeam((ChangeTeam)message);
}
else if (message instanceof ChangeModel) {
    processChangeModel((ChangeModel)message);
}

// just log the received message.
logger.info("Received message: " + message);
```

Quadro 23 – Método `messageReceived` referente ao estado “*WaitingConnections*”

Nota-se que nesse método é feito um “chaveamento” para saber qual mensagem foi recebida, podendo assim saber como tratá-la adequadamente. Na seção 3.2.1.2.2, onde foi apresentada a especificação da interação com o aplicativo servidor, é explicado em detalhes

como é feito o tratamento de cada mensagem pelo cliente e pelo servidor. Por esse motivo, o código-fonte de todos os métodos de processamento de mensagens por parte do aplicativo servidor não é exibido.

No método `processUserReady`, além de trocar o *status* do jogador para "*pronto para jogar*" (ativando o atributo booleano `readyToPlay` da classe `OnlinePlayer`), é invocado o método `checkIfAllPlayerAreReadyToPlay`, cujo fonte é mostrado no Quadro 24.

```
private void checkIfAllPlayerAreReadyToPlay() {
    Collection<OnlinePlayer> playersOnline = currentBattle.getAllOnlinePlayers();
    int qtyReady = 0;

    for (OnlinePlayer onlinePlayer : playersOnline) {
        if (onlinePlayer.isReadyToPlay()) {
            qtyReady++;
        }
    }

    if (playersOnline.size() == qtyReady) {
        serverStatus = ServerStatus.INGAME;
        switchToInGameState();

        // notify object listener
        serverStatusChangedListener.execute(ServerStatus.INGAME);

        // send message to all clients to start battle.
        StartBattle sb = new StartBattle();
        sb.setTerrainHeightmapImage(currentBattle.getTerrainHeightMapImage());
        gameServer.sendToAll(sb);
    }
}
```

Quadro 24 – Método `checkIfAllPlayerAreReadyToPlay`

Nesse método pode ser visto que, caso todos os jogadores estejam prontos para jogar, o simulador entra no estado "*InGame*" (executando o método `switchToInGameState`). Após isso, é invocado o método `execute` do objeto que trata o evento de alteração de *status* do servidor (representado pelo atributo `serverStatusChangedListener`, definido através do método `setServerStatusChangedListener` pela interface gráfica `ServerGUI`, como visto no Quadro 15). Por último, é enviada a todos os jogadores a mensagem `StartBattle`, levando consigo o caminho da imagem que representa o terreno do cenário 3D.

A mudança para o estado "*InGame*" resulta basicamente na criação de um novo objeto que trata os eventos de mensagem. A partir desse momento, as mensagens tratadas durante o estado "*WaitingConnections*" passam a ser desconsideradas, inclusive não aceitando mais a conexão de novos jogadores (através de mensagens `UserLogon`). As mensagens tratadas diretamente pelo aplicativo servidor durante permanência nesse estado são `PlayerInBattle`, `SynchronizeRequestIDMessage` e `TankDead`, como é mostrado no Quadro 25.

```
if (m instanceof PlayerInBattle) {
    OnlinePlayer op = currentBattle.getOnlinePlayerById(m.getPlayerId());
    op.setPlaying(true);

    // notify object listener.
    userInBattleEventListener.execute(op);

    checkIfAllPlayersAreInBattleState();
}
else if (m instanceof SynchronizeRequestIDMessage) {
    SynchronizeRequestIDMessage request = (SynchronizeRequestIDMessage)m;
    if (request.getRequestType() == SynchronizeRequestIDMessage.REQUEST_ID) {
        short id = serverNextId();
        request.setRequestType(SynchronizeRequestIDMessage.RESPONSE_ID);
        request.setSyncObjectId(id);
        request.getMessageClient().sendMessage(request);
    }
}
else if (m instanceof TankDead) {
    TankDead td = (TankDead)m;
    OnlinePlayer op = currentBattle.getOnlinePlayerById(m.getPlayerId());
    PlayerTank tank = op.getTankByName(td.getTankName());
    tank.setAlive(false);
    PlayerTeam teamEnum = tank.getTeam().getTeamEnum();

    int qtyTanksAliveInTeam = 0;
    Collection<OnlinePlayer> players = currentBattle.getAllOnlinePlayers();
    for (OnlinePlayer onlinePlayer : players) {
        for (PlayerTank pTank : onlinePlayer.getTanks()) {
            if (pTank.getTeam().getTeamEnum() == teamEnum && pTank.isAlive()) {
                qtyTanksAliveInTeam++;
            }
        }
    }

    if (qtyTanksAliveInTeam == 0) {
        TeamLeftGame tlg = new TeamLeftGame();
        tlg.setTeam(teamEnum);
        gameServer.sendToAll(tlg);
    }
}
```

Quadro 25 – Método `messageReceived` referente ao estado “*InGame*”

De acordo com a especificação já apresentada na seção 3.2.1.2.2, a mensagem `PlayerInBattle` é recebida quando cada instância de aplicativo cliente termina de carregar o cenário 3D e todos os seus elementos. Cada vez que essa mensagem é recebida, o método `checkIfAllPlayersAreInBattleState` se encarrega de verificar se todos os jogadores já se encontram no estado “na batalha” (quer dizer que a instância de aplicativo cliente já terminou de migrar para o *game state* “InGame”. Caso aconteça isso, é enviada a todos os jogadores a mensagem `AllPlayersAreInGameState`.

A mensagem `SynchronizeRequestIDMessage` é enviada pelo aplicativo cliente quando um tanque de guerra é registrado no `SynchronizationManager`. Ao efetuar o registro, essa classe precisa associar um identificador único, cedido pelo servidor, ao objeto `SyncWrapper` correspondente ao tanque.

Por último, a mensagem `TankDead` é recebida quando um tanque, de fato, “morre”. O

tratamento dessa mensagem envolve a verificação da existência de algum tanque de guerra “vivo” no time que acaba de perder o integrante. Caso não exista nenhum, é enviada a todos os jogadores a mensagem `TeamLeftGame`, notificando a derrota do time e conseqüente vitória do outro.

#### 3.3.2.2 Implementação do aplicativo cliente

O outro componente da arquitetura híbrida abordado na especificação do simulador é o aplicativo cliente. A apresentação da implementação dele segue a mesma subdivisão criada na seção 3.2.1.2, onde foi especificado detalhadamente. As seções seguintes apresentam os aspectos técnicos a respeito do que foi implementado utilizando as ferramentas e técnicas já comentadas.

##### 3.3.2.2.1 Implementação da representação gráfica do AVD

O início do aplicativo cliente ocorre através do método `main` da classe `TankCoders`. A única tarefa desse método é invocar o construtor da classe, que inicializa os recursos necessários para exibição do menu principal do simulador. O Quadro 26 exibe um trecho de código da classe `TankCoders`, evidenciando os passos iniciais do aplicativo cliente.

```
public class TankCoders extends StandardGame {
    private static TankCoders gameInstance;
    private static boolean debug;
    private static boolean serverLocal;
    private static Logger logger;
    private String[] agents;
    private GameMenuState mainMenuState;
    private InGameState inGameState;

    private TankCoders(String[] args) {
        super("Tank Coders - Smart agents that matters");
        gameInstance = this;
        checkArgs(args);
        prepareLogger();
        setWindowSettings();
        initGameComponents();
        start();
    }
    public static void main(String[] args) {
        new TankCoders(args);
    }
   ...
```

Quadro 26 – Trecho de código da classe `TankCoders`

Nota-se que a classe `TankCoders` deriva de `StardardGame`. Dessa forma, essa classe torna-se responsável pela inicialização da aplicação gráfica. No construtor, a primeira tarefa relevante é a checagem dos argumentos recebidos no método `main`, que podem ser “-debug”, “-serverLocal” e “-mas comma-separeted-list”. O primeiro deles é responsável por executar o aplicativo em modo de depuração, fazendo com que diversas mensagens sejam impressas no *console* durante a execução. A segunda opção é utilizada quando os aplicativos cliente e servidor executam no mesmo computador e este não possui nenhuma conexão de rede ativa (opção vista em mais detalhes na seção seguinte). A última opção é utilizada para executar o aplicativo como um jogador do tipo MAS, ou seja, a partir de um projeto Jason (opção detalhada na seção que comenta a integração com o interpretador Jason).

No construtor é invocado também o método `setWindowSettings`, utilizado para definir as configurações da janela. No Quadro 27 é exibido o código desse método.

```
private void setWindowSettings() {
    // everytime clear the last settings.
    try {
        this.getSettings().clear();
    } catch (Exception e) {
        e.printStackTrace();
    }

    // retrieve the game persistent properties.
    GamePersistentProperties prop = GamePersistentProperties.getInstance();

    // windows settings
    this.getSettings().setRenderer(prop.getRenderer());
    this.getSettings().setFrequency(prop.getDisplayFrequency());
    this.getSettings().setWidth(prop.getDisplayWidth());
    this.getSettings().setHeight(prop.getDisplayHeight());
    this.getSettings().setFullscreen(prop.isFullscreen());
    this.getSettings().setDepth(prop.getDepth());
}
```

Quadro 27 – Método `setWindowSettings`

O penúltimo método invocado no construtor é o `initGameComponents`. Esse método é responsável criar o gerenciador de *game states* da *engine* JME e invocar a criação do menu do simulador (através do método `createMainMenu`). Depois de feito isso, o método `start` da classe ancestral `StandardGame` é invocado, exibindo na tela a interface gráfica.

Como comentado na especificação, o *game state* inicial do aplicativo cliente é o `GameMenuState`, criado dentro do método `createMainMenu`. O trecho de código referente à criação desse *game state* é exibido no Quadro 28.

```
mainMenuState = new GameMenuState();
mainMenuState.setActive(true);
...
GameStateManager.getInstance().attachChild(mainMenuState);
...
```

Quadro 28 – Trecho de código da criação do *game state* `GameMenuState`

Na instanciação da classe `GameMenuState` é invocada a criação dos componentes do menu principal do simulador através da classe `GameMenuGUI`. As classes `JoinGameMenuGUI` e `PrepareBattleMenuGUI` consistem na implementação das páginas de escolha do servidor e preparação da batalha, respectivamente.

Na página representada pela classe `JoinGameMenuGUI` é listado o primeiro servidor do simulador TankCoders encontrado na rede local. A implementação dessa tarefa é vista em detalhes na próxima seção, onde é abordada a implementação da interação com o aplicativo servidor.

A última página do menu, representada pela classe `PrepareBattleMenuGUI`, exibe o menu para preparação da batalha, onde são configurados os modelos e times de cada tanque de guerra pertencente ao jogador. Mais detalhes sobre a implementação dessa classe também são vistos na seção seguinte.

Na classe `TankCoders`, o método responsável pela transição do *game state* "Menu" para o *game state* "*InGame*" é o `changeToInGameState`, cujo código é visto no Quadro 29.

```
public void changeToInGameState(StartGameArguments gameArgs) {
    mainMenuState.setActive(false);
    GameStateManager.getInstance().detachChild(mainMenuState);

    if (gameArgs.getLocalPlayer().getType() == PlayerType.AVATAR) {
        inGameState = new AvatarPlayerInGameState(gameArgs);
    }
    else {
        inGameState = new MasPlayerInGameState(gameArgs);
    }

    inGameState.setActive(true);
    GameStateManager.getInstance().attachChild(inGameState);
}
```

Quadro 29 – Método `changeToInGameState`

Nota-se que para instanciar a classe do *game state* "*InGame*" é verificado qual o tipo do jogador da instancia local do simulador. Lembrando que ambas as classes de *game state* instanciadas nesse método derivam da classe `InGameState`, que encapsula basicamente todo o processo de criação do cenário 3D, pois isso praticamente independe do tipo de jogador. Justamente as etapas de criação do cenário que diferem entre os tipos de jogadores são implementadas nas subclasses concretas.

A classe abstrata `InGameState`, que deriva de `StatisticsGameState` da *engine* JME, representa o ponto inicial do *game state* que gerencia o cenário 3D da batalha. É nessa classe que o grafo de cena (apresentado na Figura 20) é atualizado e *renderizado* a cada iteração do *loop* principal da aplicação. O Quadro 30 exibe o trecho inicial do código-fonte dessa classe, sendo que o foco é mostrar quais os passos executados no construtor da classe.

```
public abstract class InGameState extends StatisticsGameState {

    private Renderer renderer;
    protected PhysicsGameState physicsGameState;
    protected InputHandler input;
    protected Camera cam;
    protected Panel hud;
    protected Sky sky;
    protected ITerrain terrain;
    protected Map<String, ITank> tanks;
    protected List<ITank> tanksList;
    protected ITank cameraTargetTank;
    protected StartGameArguments gameArgs;
    private Object killTankSyncObj = new Object();

    public InGameState(StartGameArguments gameArgs) {
        super();
        this.gameArgs = gameArgs;
        this.rootNode = new Node("RootNode");

        initRenderer();
        setupZBuffer();

        initLight();
        setupCamera();
        setupPhysicsGameState();
        setupOptimizations();

        makeSky();
        makeTerrain(gameArgs);
        makeHUD(gameArgs);
        setupEffectManager();
        makeTanks(gameArgs);
        setupChaseCameraOnTarget();
        setupWeaponManager();
        setupGravity();
        setupGameClient(gameArgs);
        startStopwatch();
    }

    ...
```

Quadro 30 – Trecho inicial de código da classe `InGameState`

Nesse construtor, o primeiro passo é inicializar o atributo `renderer` a partir do método `getRenderer` da classe `DisplaySystem` da *engine* JME. Em seguida é criado e configurado um estado Z-Buffer, que é essencial *renderizar* uma cena 3D. Ele permite à placa de vídeo (GPU) determinar quais elementos da cena estão visíveis e quais estão encobertos por outros, fazendo com que não sejam incluídos na cena.

O método `initLight` é responsável por criar e inicializar a iluminação do cenário, definindo um ponto de luz em uma coordenada 3D. Essa coordenada, representada pela classe `Vector3f`, possui os valores dos eixos X, Y e Z, respectivamente, zero, dez mil e zero. Esses valores consistem no centro do cenário, adicionando um valor de dez mil unidades ao eixo Y, fazendo com que o ponto de luz fique consideravelmente acima do terreno.

O método `setupCamera` inicializa e configura a câmera para o cenário. No Quadro 31 é apresentado o código-fonte desse método que basicamente define o *frustum* de visão da

câmera e em seguida define a localização inicial da mesma.

```
private void setupCamera() {
    float aspect = (float) DisplaySystem.getDisplaySystem().getWidth() /
                   (float) DisplaySystem.getDisplaySystem().getHeight();

    cam = DisplaySystem.getDisplaySystem().getRenderer().getCamera();
    cam.setFrustumPerspective(45.0f, aspect, 1, 10000);
    cam.setParallelProjection( false ); // enter to normal perspective mode
    cam.update();

    Vector3f loc = new Vector3f(0.0f, 0.0f, 25.0f);
    Vector3f left = new Vector3f(-1.0f, 0.0f, 0.0f);
    Vector3f up = new Vector3f(0.0f, 1.0f, 0.0f);
    Vector3f dir = new Vector3f(0.0f, 0f, -1.0f);

    cam.setFrame(loc, left, up, dir);
    cam.update();
}
```

Quadro 31 – Método `setupCamera`

Na invocação do método `setFrustumPerspective` da câmera, são informados os parâmetros:

a) `fovY`: ângulo de abertura (em graus) em relação ao eixo Y;

b) `aspect`: aspecto da janela, valor obtido pela razão da largura e altura;

c) `near`: distância mínima a ser considerada pela câmera;

d) `far`: distância máxima a ser considerada pela câmera.

O próximo passo a ser executado no construtor da classe `InGameState` consiste na criação do *game state* que trata as questões de física da aplicação. A classe que cede esse recurso é a `PhysicsGameState` provido pelo *framework* JME Physics. Um objeto dessa classe é anexado ao gerenciador de *game states* `GameStateManager`, para que ele também seja processado a cada iteração do *loop* principal da aplicação.

O método `setupOptimizations` cria e configura o estado `CullState`, responsável por prover a otimização denominada *culling*, já comentada na seção 2.3. O Quadro 32 apresenta como é criado o estado para otimização na *renderização* do grafo de cena, desde o nó raiz (`RootNode`).

```
private void setupOptimizations() {
    CullState cs = DisplaySystem.getDisplaySystem().getRenderer().
                   createCullState();
    cs.setCullMode(CullState.CS_BACK);
    cs.setEnabled(true);
    getRootNode().setRenderState(cs);
}
```

Quadro 32 – Método `setupOptimizations`

Os próximos métodos prefixados pela palavra "*make*" contemplam a criação dos elementos que compõem o cenário 3D. Os elementos, já apresentados no grafo de cena da Figura 20, são: *Sky*, *Terrain*, *Hud* e *Tanks*. A implementação dos elementos *Sky* e *Hud*, por

não apresentar uma complexidade considerável, não é mostrada. Porém os elementos *Terrain* e *Tank*, por serem considerados os principais elementos do cenário, são apresentados detalhadamente.

A criação do terreno inicia pela invocação do método estático `makeTerrain` da fábrica de objetos representada pela classe `TerrainFactory`, como mostrado no Quadro 33. Porém, a única tarefa desse método é invocar o construtor da classe concreta `Terrain` (classe que implementa a interface `ITerrain`), que dá inicio montagem dos elementos do terreno.

```
private void makeTerrain(StartGameArguments gameArgs) {
    terrain = TerrainFactory.makeTerrain(getPhysicsSpace(),
                                         gameArgs.getTerrainHeightMapImage());
    getRootNode().attachChild((Node)terrain);
}
```

Quadro 33 – Método `makeTerrain` da classe `InGameState`

Como apresentado na especificação, a montagem do terreno consiste na criação da malha de triângulos que compõe a topologia (`makeTopology`), na definição da representação física (`makePhysicsRepresentation`), na criação da *bounding box* (`makeModelBound`) e por último, na criação das bandeiras dos quartéis generais (`makeFlags`).

O Quadro 34 exibe o fonte do método `makeTopology`, que cria a malha de triângulos a partir de uma imagem em escala de cinza. A classe `ImageBasedHeightMap` da JME considera que cores mais próximas de branco representam alturas maiores e cores mais próximas de preto, alturas menores. Isso porque uma imagem em escala de cinza só possui um canal de cor, cujo valor varia de 0 (preto) a 255 (branco). O caminho (relativo à pasta onde a aplicação está contida) da imagem que representa a topologia do terreno é escolhida antes de criar-se uma nova batalha no aplicativo servidor. Essa informação chega até o aplicativo cliente no momento que a batalha inicia, sendo que os detalhes sobre esse processo são explicados na próxima seção. Após a criação e posicionamento da malha de triângulos, é aplicada a textura sobre ela, tornando visível a topologia do terreno.

```
private void makeTopology(String heightMapImageFilePath) {
   URL grayScale = Terrain.class.getClassLoader().
                                 getResource(heightMapImageFilePath);
   heightMap = new ImageBasedHeightMap(new ImageIcon(grayScale).getImage());

   Vector3f terrainScale = new Vector3f(200, 10, 200);
   Vector3f position = new Vector3f(-200, -45, -200);
   terrainBlock = new TerrainPage("terrainPage", 33, heightMap.getSize()+1,
                                  terrainScale, heightMap.getHeightMap(), true);
   terrainBlock.setLocalTranslation(position);

   applyTexture(heightMap);

   this.attachChild(terrainBlock);
}
```

Quadro 34 – Método `makeTopology`

Para a representação física do terreno, é executado o código mostrado no Quadro 35. Nele é criado um nó estático (controlado pela física) onde é anexada a malha de triângulos gerada. Nesse nó também é definido um material tipo madeira (`Material.WOOD`), que é o material a ser usado para que o *framework* de física possa prover o aspecto de atrito entre o terreno e os demais objetos sobre ele (no caso, as rodas dos tanques).

```
private void makePhysicsRepresentation(PhysicsSpace pSpace) {
    StaticPhysicsNode staticNode = pSpace.createStaticNode();
    staticNode.attachChild(terrainBlock);
    staticNode.generatePhysicsGeometry(true);
    staticNode.setMaterial(Material.WOOD);

    this.attachChild(staticNode);
}
```

Quadro 35 – Método `makePhysicsRepresentation`

O último passo da montagem do terreno é a criação e posicionamento das bandeiras dos quartéis generais, que é feito através do método `makeFlags`. Nele, quatro bandeiras (representadas pela classe `HeadquarterFlag`) são criadas e anexadas ao nó raiz do terreno. A classe da bandeira `HeadquarterFlag` é composta por dois pilares (implementados com a classe `Box` da JME) e um pano da cor do time que ela representa (implementada com a classe `CollidingClothPatch`). A classe `HeadquarterFlag` utiliza também um mecanismo de gravidade, onde é aplicada uma força que representa o movimento do ar, simulando o movimento da bandeira gerada pela ocorrência de ventos. O trecho de código responsável pela criação desse mecanismo é mostrado no Quadro 36.

```
...
cloth = new CollidingClothPatch("cloth", 50, 50, 1f, 10);

wind = ClothUtils.createBasicWind(windStrength, windDirection, true);
cloth.addForce(wind);

gravity = ClothUtils.createBasicGravity();
cloth.addForce(gravity);

drag = ClothUtils.createBasicDrag(20f);
cloth.addForce(drag);

cloth.setLocalScale(new Vector3f(.12f, .14f, .12f));
cloth.setLocalTranslation(new Vector3f(3f, 1.3f, 0));

...
```

Quadro 36 – Utilização da classe `CollidingClothPatch` e aplicação de forças

Após a criação do terreno, o construtor da classe `InGameState` (Quadro 30) chama o método `makeTanks`, que é responsável por criar todos os tanques da batalha, inclusive os tanques dos jogadores remotos. A criação dos tanques de guerra consiste no processo mais demorado que ocorre durante a criação do cenário, pois é nesse momento que são carregados os modelos 3D referentes às estruturas do tanque, que são chassi, trilhos e arma principal.

Esses modelos consistem em arquivos no formato Wavefront, exportados pela ferramenta Blender 3D para serem anexados ao grafo de cena como objetos nativos da JME. O código-fonte mostrado no Quadro 37 refere-se ao método `makeTanks`.

```
protected void makeTanks(StartGameArguments gameArgs) {
    tanks = new HashMap<String, ITank>();

    for (Iterator<String> i = gameArgs.getPlayersNamesIterator(); i.hasNext(); ) {
        String playerName = i.next();

        Player p = gameArgs.getPlayer(playerName);
        boolean isRemote = (p.getId() != gameArgs.getLocalPlayer().getId());
        boolean isMASPlayer = (p.getType() == PlayerType.MAS);

        for (PlayerTank tank : p.getTanks()) {
            makeTank(p.getId(), tank, isRemote, isMASPlayer);
        }
    }

    tanksList = new ArrayList<ITank>();
    tanksList.addAll(tanks.values());
}

private void makeTank(short playerIdOwner, PlayerTank pTank, boolean remoteTank,
                      boolean masPlayer) {
    Vector3f location =
                terrain.getTanksInitialLocation()[pTank.getInitialSlotLocation()];

    ITank tank = TankFactory.makeTank(playerIdOwner, pTank, getPhysicsSpace(),
                                      location, remoteTank, masPlayer);
    tank.setPlayerIsTheOwner(!remoteTank);

    ((Node)terrain).attachChild((Node)tank);
    getPhysicsSpace().addToUpdateCallbacks(new TankUpdater(tank));

    tanks.put(pTank.getTankName(), tank);
}
```

Quadro 37 – Criação do tanque na classe `InGameState`

Nota-se que para cada tanque de guerra de cada jogador é invocado o método `makeTank`. Nele, inicialmente é obtido a localização inicial do tanque, em seguida é invocado o método `makeTank` da fábrica de objetos tanque, denominada `TankFactory`. Esse método é explicado em detalhes na seção que aborda a integração com o interpretador Jason. No momento, basta saber que uma classe concreta derivada de `AbstractTank` é instanciada para em seguida ser anexada ao terreno. Em seguida, no método mostrado no Quadro 37, a instância do tanque é adicionada na lista das chamadas *update callbacks* do *game state* de física, através da classe `TankUpdater`. A *update callback* criada para a instância do tanque em questão é utilizada para que o método `update` da classe do tanque seja invocado depois de cada atualização do *game state* de física. A classe `TankUpdater`, para se tornar uma *update callback*, implementa a interface `PhysicsUpdateCallback`, tendo que sobrescrever os métodos `afterStep` e `beforeStep`. Logo, a invocação do método `update` do tanque é feita no método `beforeStep`.

Como apresentado no diagrama de classes da Figura 22 contida na especificação, a classe abstrata `AbstractTank` possui uma subclasse concreta para cada modelo de tanque, que são `M1AbramsTank` e `JadgePantherTank`. A maior parte da implementação de ambos os modelos de tanques está contida na classe ancestral, sendo que pequenas peculiaridades são implementadas nas especializações. As subclasses também são responsáveis por implementar os métodos abstratos da classe ancestral que requisitam informações para criação do tanque e seus elementos. No Quadro 38 é mostrado o construtor da classe `AbstractTank`.

```
public AbstractTank(short playerIdOwner, String tankName, TankTeam team, final
                    PhysicsSpace pSpace, final Vector3f position,
                    final boolean remoteTank, final boolean agentControls) {
    super(tankName);
    this.physicsSpace = pSpace;
    this.playerIdOwner = playerIdOwner;
    this.tankName = tankName;
    this.team = team;
    this.health = INITIAL_HEALTH;
    this.mainGunQtyBullets = getInitialQtyBullets();
    this.remote = remoteTank;
    this.agentControls = agentControls && (!remote);
    this.fireMainGunBullets = new ConcurrentLinkedQueue<Short>();
    this.fireMachineGunBullets = new ConcurrentLinkedQueue<Short>();
    this.notifyTankInfoRate = GamePersistentProperties.getInstance().
                                                    getNotifyTankInfoRate();
    // make ready for immediate update
    this.lastUpdateTime = System.nanoTime() - notifyTankInfoRate;

    if (!remoteTank) {
        makeDynamicChassi(pSpace, position);
        makeSuspensions(pSpace);
        makeTankIdentificationLabel(chassi);
        makeModelBound(chassi);
    }
    else {
        makeStaticChassi(pSpace, position);
        makeTankIdentificationLabel(staticChassi);
        makeModelBound(staticChassi);
    }
}
```

Quadro 38 – Construtor da classe `AbstractTank`.

Dentre os atributos dessa classe, destacam-se:

a) `chassi`: nó dinâmico que representa o chassi de um tanque local;

b) `staticChassi`: nó estático que representa o chassi de um tanque remoto;

c) `rearSuspension` e `frontSuspension`: suspensão traseira e dianteira de um tanque local. Para tanques remotos essa estrutura não é adicionada;

d) `health`: saúde do tanque;

e) `mainGunQtyBullets`: quantidade de balas correntemente disponível para a arma principal;

f) `remote`: determina se é um tanque de um jogador remoto;

g) `agentControls`: determina se é um agente que controla diretamente o tanque.

Nota-se que para esse atributo poder ser verdadeiro, o atributo `remote` precisa ser falso, ou seja, para um agente controlar diretamente o tanque, ele precisa ser do jogador local;

h) `fireMainGunBullets`: fila de balas da arma principal a serem disparadas na próxima chamada do método `update`. Explicado adiante na seção que trata da integração com o Jason;

i) `fireMachineGunBullets`: mesmo que o atributo `fireMainGunBullets`, porém para as balas da metralhadora;

j) `notifyTankInfoRate`: tempo, em nano-segundos, decorrente entre cada notificação de informações ao agente que pode controlar o tanque;

k) `lastUpdateTime`: tempo, em nano-segundos, da última notificação de informações ao agente.

Após a inicialização dos atributos é verificado se o tanque pertence ao jogador local ou a um jogador remoto. Essa verificação é necessária, pois para um jogador remoto não existe a necessidade de utilizar nós dinâmicos, que por sua vez são mais complexos de serem tratados pela *engine* (considera tratamento de colisão, gravidade, e outras questões físicas). Mais detalhes sobre a necessidade de tratar tanques locais e remotos de forma distinta são vistos na próxima seção.

A criação do chassi e da suspensão do tanque acontece no código apresentado no Quadro 39.

```
protected void makeDynamicChassi(final PhysicsSpace pSpace, final Vector3f
                                 position) {
    chassi = pSpace.createDynamicNode();
    chassi.setName("tankChassi");
    chassi.setLocalTranslation(position);

    Node model = ModelUtils.getNodeByObj(getChassiModelPath());
    chassi.attachChild(model);

    chassi.generatePhysicsGeometry(true);
    chassi.setMaterial(Material.WOOD);
    chassi.setMass(getChassiMass());

    this.attachChild(chassi);
}


private void makeSuspensions(final PhysicsSpace pSpace) {
    rearSuspension = getSuspensionImpl(pSpace, chassi, new Vector3f(-
                        getAxisDistances()/2, getSuspensionHeight(), 0), false);
    this.attachChild(rearSuspension);

    frontSuspension = getSuspensionImpl(pSpace, chassi,
              new Vector3f(getAxisDistances()/2, getSuspensionHeight(), 0), true);
    this.attachChild(frontSuspension);
}
```

Quadro 39 – Criação do chassi e suspensão do tanque

Na criação do chassi dinâmico, nota-se que é obtido o modelo 3D que representa graficamente o chassi do tanque de guerra. Isso é feito através da invocação do método estático `getNodeByObj` da classe `ModelUtil`. Esse método carrega um arquivo no formato Wavefront (com extensão ".obj") e converte para um objeto da classe `Node` que pode ser anexado ao grafo de cena como um objeto nativo criado na JME. Além disso, esse método armazena internamente o conteúdo do arquivo carregado, para que na próxima vez que for invocado, não seja necessário carregá-los outra vez. Ainda na criação do chassi, nota-se também que é definida uma massa para o nó dinâmico (`setMass`). Essa informação é utilizada pelo *framework* JME Physics para controlar as colisões e também afetar o objeto pela gravidade ainda a ser definida.

Para a criação das duas suspensões do tanque de guerra, nota-se que é invocado o método `getSuspensionImpl`. Trata-se de um método abstrato e sobrescrito pelas subclasses de `AbstractTank`. Em cada método sobrescrito é instanciado e retornado um objeto de uma subclasse de `Suspesion`, podendo ser `M1AbramsSuspension` e `PantherSuspension`. Essas classes seguem a mesma forma de implementação da classe do tanque, ou seja, a classe abstrata que é responsável pela criação dos elementos da suspensão e as subclasses somente sobrescrevem os métodos abstratos, provendo as informações necessárias.

```
public Suspension(final PhysicsSpace pSpace, final DynamicPhysicsNode chassi,
                  final Vector3f position, boolean canTurn) {
    super("suspensao");

    // creating two support bases of the suspension.
    leftBase = makeBase(pSpace, chassi, position.add(getBaseRelativePosition()));
    rightBase = makeBase(pSpace, chassi,
                         position.subtract(getBaseRelativePosition()));

    // creating two wheels, one per support base.
    leftWheel = makeWheel(leftBase, getTyreRelativePosition(), canTurn);
    this.attachChild(leftWheel);

    rightWheel = makeWheel(rightBase, getTyreRelativePosition().negate(),
                           canTurn);
    this.attachChild(rightWheel);
}
```

Quadro 40 – Construtor da classe `Suspesion`

A suspensão consiste em duas bases de sustentação independentes, uma para cada lado. Em cada base é anexada uma roda invisível. Juntas, as rodas são responsáveis pela movimentação e direção do tanque. Além disso, cada base de sustentação é ligada ao chassi através de uma junta (*joint*) que representa um amortecedor. Esse componente é útil para suavizar o impacto causado pelo peso do chassi sobre o terreno. A Figura 26 ilustra o funcionamento da suspensão do tanque, evidenciando os elementos onde as forças atuam.

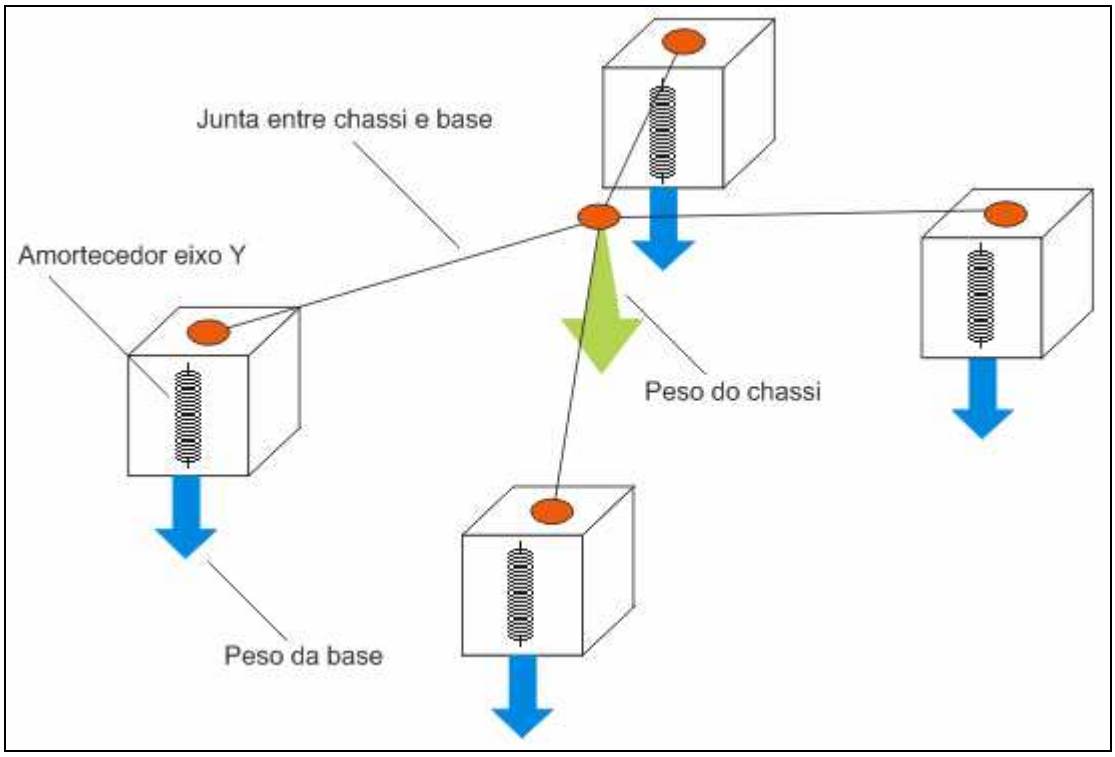



Figura 26 – Esquema de suspensão do tanque de guerra no simulador

A elasticidade da mola que representa o amortecedor é definida pela distância máxima e mínima que a junta de translação no eixo Y pode assumir. Para o amortecedor, também são definidos os valores referentes à velocidade e aceleração da junta. O Quadro 41 ilustra o método `makeBase`, onde o esquema apresentado na figura acima é implementado.

```
private DynamicPhysicsNode makeBase(final PhysicsSpace pSpace, final
                DynamicPhysicsNode chassi, final Vector3f relativePosition) {

    DynamicPhysicsNode suspensionBase = pSpace.createDynamicNode();
    suspensionBase.setName("suspensionBase");

    suspensionBase.setLocalTranslation(chassi.getLocalTranslation()
                                                    .add(relativePosition));
    suspensionBase.createBox("baseBox");
    suspensionBase.setLocalScale(0.1f);
    suspensionBase.setMass(getBaseMass());
    suspensionBase.setMaterial(Material.GHOST); // all the bases are invisible
    this.attachChild(suspensionBase);

    Joint suspensionJoint = pSpace.createJoint();
    suspensionJoint.attach(suspensionBase, chassi);

    // axis for damper of the suspension
    TranslationalJointAxis damper = suspensionJoint.createTranslationalAxis();
    damper.setPositionMaximum(getDamperMaxDisplacement());
    damper.setPositionMinimum(-getDamperMaxDisplacement());
    damper.setAvailableAcceleration(getDamperAcceleration());
    damper.setDesiredVelocity(getDamperSpeed());
    damper.setDirection(DAMPER_AXIS);

    return suspensionBase;
}
```

Quadro 41 – Método `makeBase`

A classe `Suspension` contempla ainda os métodos referentes à aceleração e direção do tanque. Porém essa funcionalidade não é provida diretamente pela suspensão, mas sim pela roda, que é implementada na classe abstrata `Wheel`. A suspensão apenas delega o serviço para as rodas do lado direito e esquerdo da suspensão, sendo assim, ambas as suspensões traseira e dianteira do tanque de guerra possuem tração. Por outro lado, a direção é provida apenas pela suspensão dianteira. É justamente por esse motivo que o construtor da classe `Suspension` recebe o parâmetro chamado `canTurn`.

Para completar a anatomia de um tanque de guerra, o último elemento a ser detalhado é a roda. Esse elemento é representado pela classe abstrata `Wheel` e suas subclasses concretas `M1AbramsWheel` e `PantherWheel`, que provêem as informações necessárias para a criação da roda. O construtor da classe `Wheel` é exibido no Quadro 42.

```
public Wheel(final DynamicPhysicsNode suspensionBase,
             final Vector3f relativePosition, boolean canTurn) {
    super("wheelNode");
    tyre = suspensionBase.getSpace().createDynamicNode();
    tyre.setName("wheel");

    tyre.setLocalTranslation(suspensionBase.getLocalTranslation().
                                            add(relativePosition));
    tyre.createSphere("tyre");
    tyre.setLocalScale(getWheelScale());
    tyre.generatePhysicsGeometry();
    tyre.setMaterial(Material.RUBBER);
    tyre.setMass(getWheelMass());

    Joint wheelBox = suspensionBase.getSpace().createJoint();
    wheelBox.attach(suspensionBase, tyre);
    wheelBox.setAnchor(tyre.getLocalTranslation().subtract(suspensionBase.
                                                    getLocalTranslation()));

    if (canTurn) {
        directionAxis = wheelBox.createRotationalAxis();
        directionAxis.setDirection(DIRECTION_AXIS);
        directionAxis.setAvailableAcceleration(getTurnAcceleration());
    }

    tractionAxis = wheelBox.createRotationalAxis();
    tractionAxis.setDirection(TRACTION_AXIS);

    if (canTurn) {
        tractionAxis.setRelativeToSecondObject(true);
        stopTurningWheel();
    }

    this.attachChild(tyre);
}
```

Quadro 42 – Construtor da classe `Wheel`

Para a criação da roda, primeiramente é criado um nó dinâmico que a representa fisicamente. Nesse nó é criada uma esfera, para funcionar como um pneu. Também é definido o material borracha (`Material.RUBBER`) para a roda, fazendo com que tenha atrito com o material do terreno, como já dito que é do tipo madeira (`Material.WOOD`). Para que a roda

permaneça ligada à respectiva base de suspensão, uma junta é adicionada entre a base de suspensão e o nó dinâmico da roda. Caso a roda pertença à suspensão dianteira, ela precisa estar ligada também a um eixo de rotação em relação a Y (atributo `directionAxis`). Por último é definido o eixo de tração, o qual é baseado em Z. Com esses dois eixos é possível fazer a movimentação e direção do tanque, como é mostrado no Quadro 43.

```
public void accelerate(final int direction) {
    tractionAxis.setAvailableAcceleration(getTractionAcceleration());
    tractionAxis.setDesiredVelocity(direction*getTractionSpeed());
}

public void dropAccelerator() {
    tractionAxis.setAvailableAcceleration(400);
    tractionAxis.setDesiredVelocity(0);
}

public void turnWheel(int direction) {
    directionAxis.setDesiredVelocity(direction*getTurnSpeed());
    directionAxis.setPositionMaximum(getMaxTurn());
    directionAxis.setPositionMinimum(-getMaxTurn());
}

public void stopTurningWheel() {
    directionAxis.setDesiredVelocity(0);
    directionAxis.setPositionMaximum(0);
    directionAxis.setPositionMinimum(0);
}
```

Quadro 43 – Métodos de movimentação e direção da roda

É importante lembrar que a roda implementada nessa classe é utilizada somente para realizar a movimentação e direção do tanque de guerra, sendo que não é visível para o jogador. Os trilhos e rodas visíveis são meramente modelos 3D carregados da mesma forma que o chassi, ou seja, só por efeitos de visualização. A tarefa de anexar esses modelos 3D ao tanque de guerra é realizada nas classes concretas `M1AbramsTank` e `JadgePantherTank`. Essa tarefa não é feita na classe ancestral `AbstractTank`, pois no caso do modelo “M1 Abrams” existe um modelo 3D para cada lado do tanque. Para o modelo “Jadge Panther”, os trilhos e rodas consistem em um único modelo 3D.

No Apêndice A podem ser vistas ilustrações dos modelos 3D utilizados nos tanques de guerra desenvolvidos.

Além de se movimentar no ao longo do terreno, os tanques de guerra podem ainda disparar projéteis (balas) através da chamada arma principal (*main gun*) e da metralhadora (*machine gun*). Na classe `AbstractTank`, os métodos responsáveis por realizar o disparo dos projéteis são chamados de `fireMainGunBullet` e `fireMachineGunBullet`. O Quadro 44 apresenta a implementação desses dois métodos, evidenciando como a tarefa é delegada à classe `WeaponManager`.

```
public void fireMainGunBullet() {
    if (mainGunQtyBullets > 0) {
        Bullet bullet = getMainGunBulletImpl(this);
        WeaponManager.fireBullet(bullet);

        NVEHandler.sendTankActionShotMainGun(tankName);

        decMainGunQtyBullets();
    }
}

public void fireMachineGunBullet() {
    Bullet bullet = getMachineGunBulletImpl(this);
    WeaponManager.fireBullet(bullet);
}
```

Quadro 44 – Métodos para disparo de projéteis

Para o disparo de um projétil pela arma principal, um objeto de uma subclasse concreta de `Bullet` é obtido através do método abstrato `getMainGunBulletImpl`. Esse método retorna uma instância baseado no modelo de tanque, logo o objeto pode ser criado a partir da classe `M1AbramsMainGunBullet` ou `PantherMainGunBullet`. A instância da bala é passada por parâmetro para o método estático `fireBullet` da classe `WeaponManager`, que é responsável de fato pelo disparo do projétil, como mostrado no Quadro 45.

```
public static void fireBullet(Bullet bullet) {
    // add the controller that handle collisions.
    bullet.addController(new ProjectileCollisionController(bullet, terrain, tanks,
                                                            activeProjectiles));

    // attach the bullet in the scene node (terrain)
    scene.attachChild(bullet);
    bullet.updateRenderState();

    // add a force based on tank (or maingun) heading
    bullet.getBulletDyn().addForce(bullet.getVector().
                                                    mult(bullet.getMultForce()));

    // add the bullet in array for collision detection.
    activeProjectiles.add(bullet);
}
```

Quadro 45 – Método `fireBullet` da classe `WeaponManager`

Antes de o projétil ser disparado, é criado um objeto responsável por acompanhar cada atualização de posição da bala durante o seu trajeto. Esse objeto é chamado de controlador da bala e é representado pela classe `ProjectileCollisionController`. Tendo feito isso, o projétil é adicionado ao grafo de cena no nó do terreno. Nesse momento o projétil já fica posicionado à frente da arma, precisando somente ser disparado. Para isso é adicionada uma força ao nó dinâmico do projétil através do método `addForce`, que recebe por parâmetro um vetor indicando a direção e força. Esse vetor é obtido através da instância do projétil no método `getVector`. Esse vetor obtido indica somente a direção da bala, por isso na seqüência é invocado o método `mult` da classe `Vector3f` para multiplicar o vetor resultante pela força da bala (obtida pelo método `getMultForce`). O Quadro 46 apresenta o código-fonte do

método `getVector`.

```
public Vector3f getVector() {
    // rotation around Y
    float mainGunHeading = owner.getCurrentMainGunHeading();
    float newX = FastMath.cos(mainGunHeading);
    float newZ = FastMath.sin(mainGunHeading);
    float newY = .02f; // little inclination

    // rotation around Z
    float mainGunZrotAngle = owner.getCurrentMainGunZRotAngle();
    float cosZrot = FastMath.cos(mainGunZrotAngle);
    float sinZrot = FastMath.sin(mainGunZrotAngle);
    newX = (newX * cosZrot) - (newY * sinZrot);
    newY = (newX * sinZrot) + (newY * cosZrot);

    return new Vector3f(newX, newY, newZ);
}
```

Quadro 46 – Método `getVector` da classe `Bullet`

Esse método efetua o cálculo de duas rotações. A primeira, em torno do eixo Y, serve para determinar a direção da bala horizontalmente. A segunda, em torno do eixo Z, serve para determinar a inclinação da bala. Para a rotação em torno do eixo Y, primeiramente é obtido o ângulo de rotação (em relação ao mesmo eixo) da arma principal. Lembrando que a metralhadora consta acoplada à arma principal, logo em ambas as armas o ângulo é o mesmo. Já para a rotação em torno do eixo Z, é obtido o ângulo de inclinação da arma principal. O cálculo feito em seguida (das duas rotações), consta em determinar novos valores para X, Y e Z baseado no novo ângulo de rotação obtido. Para realizá-lo, tomaram-se em conta os valores iniciais para os eixos X, Y e Z, respectivamente, 1, 0.02 e 0. O código-fonte do cálculo foi elaborado baseando-se na fórmula apresentada no Quadro 47.

```
ROTAÇÃO EM TORNO DE Y
z' = z*cos q - x*sin q
x' = z*sin q + x*cos q
y' = y

ROTAÇÃO EM TORNO DE Z
x' = x*cos q - y*sin q
y' = x*sin q + y*cos q
z' = z
```

Fonte: adaptado de Owen (1998).

Quadro 47 – Formulas de rotação em torno de Y e de Z

Tendo o vetor de direção, o método estático `fireBullet` da classe `WeaponManager` pode aplicar a força da bala, multiplicando o vetor encontrado pelo valor retornado do método `getMultForce`, que é implementado em cada subclasse concreta de `Bullet`.

A respeito do disparo de um projétil a partir de uma arma, a última classe a ser

comentada é a `ProjectileCollisionController`. Ela é responsável por controlar a colisão do projétil disparado com outros elementos do cenário, podendo ser outros projéteis ou tanques de guerra. Essa classe controla também o tempo de vida do projétil disparado, pois passado um determinado número de *frames*, ele tende a ficar parado sobre o terreno sem nenhuma utilidade, logo pode ser retirado do cenário para agilizar o processo de *rendering*. O código fonte dessa classe é mostrado na seção 3.3.2.2.3, onde é apresentada a integração com o interpretador Jason. Nesse momento, basta saber que quando o projétil disparado atinge um determinado tanque, o método `hitByBullet` da classe `AbstractTank` é invocado, diminuindo a corrente saúde do tanque, que pode inclusive chegar a zero, onde o método `kill` é executado para remover o tanque do cenário.

Após a criação e inserção dos tanques no cenário, o construtor da classe `InGameState` precisa ainda configurar o gerenciador de armas `WeaponManager` (para ser utilizado em todo o processo descrito acima) e definir a gravidade do mundo. O código responsável por essas duas tarefas é mostrado no Quadro 48.

```
private void setupWeaponManager() {
    WeaponManager.setup(getRootNode(), terrain, tanks);
}


private void setupGravity() {
    getPhysicsSpace().setDirectionalGravity(new Vector3f(0, -9.81f, 0));
}
```

Quadro 48 – Configuração do gerenciador de armas e da gravidade do mundo

A última tarefa relevante do construtor da classe `InGameState` é configurar a interação com o aplicativo servidor. Essa tarefa, assim como todas as demais etapas da interação, é apresentada na seção seguinte.

#### 3.3.2.2.2 Implementação da interação com o aplicativo servidor

O primeiro papel a ser desempenhado pelo módulo de interação com o aplicativo servidor é listar o primeiro servidor do simulador TankCoders encontrado na rede local. O servidor é listado para que o jogador possa conectar-se a ele a fim de iniciar o processo de preparação da batalha, onde são definidos os modelos e times de cada um de seus tanques. A classe que contempla esse primeiro papel é a `JoinGameMenuGUI`. Nela, especificamente o método `updateServerList` é executado cada vez que o usuário solicita a atualização da lista de servidores, cujo trecho de código é mostrado no Quadro 49.

```
DatagramSocket socket = new DatagramSocket();
try {
    String ipAddress = "255.255.255.255"; // broadcast
    if (TankCoders.isServerLocal()) {
        ipAddress = "127.0.0.1";
    }
    InetAddress addr = InetAddress.getByName(ipAddress);

    byte[] buffer = "ServerStatus".getBytes();
    DatagramPacket dgp = new DatagramPacket(buffer, buffer.length, addr, 2001);
    socket.send(dgp);

    DatagramPacket answer = new DatagramPacket(new byte[110], 110);
    socket.setSoTimeout(2000);
    try {
        socket.receive(answer);
    }
    catch (SocketTimeoutException e) {
        return;
    }
    String answerStr = new String(answer.getData());

    // remove the first char of the string, because it's a slash.
    String serverAddress = answer.getAddress().toString().substring(1);
    String[] serverInfos = answerStr.split("@@");

    HashMap<String, Object> server = new HashMap<String, Object>();
    server.put(SERVERNAME_ENTRY, serverInfos[0]);
    server.put(ADDRESS_ENTRY, serverAddress);
    server.put(STATUS_ENTRY, ServerStatus.valueOf(serverInfos[1]));
    server.put(TEAM1NAME_ENTRY, serverInfos[2]);
    server.put(TEAM2NAME_ENTRY, serverInfos[3]);

    serverList.add(server);
}
finally {
    socket.close();
}

...
```

Quadro 49 – Trecho de código do método `updateServerList`

Esse código é responsável por enviar uma mensagem UDP *broadcast* na rede local especificamente para a porta 2001, que consiste na porta onde o servidor mantém um *socket* a ser utilizado somente para a finalidade de localização de servidores. Após enviar a mensagem, o método fica parado na invocação do método `receive` por no máximo dois segundos. Caso um servidor do simulador responda essa mensagem, ela é processada, inserindo o servidor na lista.

A segunda responsabilidade desse módulo, ainda no menu do simulador, é ceder os recursos necessários para a preparação da batalha. Nessa classe são enviadas mensagens para efetivar a troca dos modelos de tanque e a troca de times no servidor e nos demais clientes. Dentre todas as mensagens enviadas e recebidas nessa classe, a mais relevante é a que determina que o jogador está pronto para batalhar (`UserReady`). Essa mensagem é enviada no momento que o jogador pressiona o botão "*Ready*", representado pelo atributo `ready`. O Quadro 50 mostra a implementação da classe interna anônima responsável por tratar o evento

de botão pressionado.

```
ready.addButtonPressedListener(new IButtonPressedListener() {
    public void buttonPressed(ButtonPressedEvent e) {
        if (ready.isPressed()) {
            labelPlayerStatus.setText(READY_MSG);
            changeVisibleComboBoxComponents(false);

            UserReady ur = new UserReady();
            ur.setPlayerId(localPlayer.getId());
            gameClient.sendToServer(ur);
        }
        else {
            labelPlayerStatus.setText(UNREADY_MSG);
            changeVisibleComboBoxComponents(true);

            UserUnready uur = new UserUnready();
            uur.setPlayerId(localPlayer.getId());
            gameClient.sendToServer(uur);
        }
    }
}
```

Quadro 50 – Tratador de eventos do botão "*Ready*"

Como comentado na especificação, após todos os jogadores enviarem a mensagem `UserReady`, o servidor envia a mensagem `StartBattle` à todos os clientes, que é tratada pelo método `processStartBattleMessage`, cujo trecho de código é mostrado no Quadro 51.

```
...
StartGameArguments gameArgs = new StartGameArguments();
String imgHeightMap = ((StartBattle)message).getTerrainHeightmapImage();
if (imgHeightMap != null) {
    gameArgs.setTerrainHeightMapImage(imgHeightMap);
}

gameArgs.setLocalPlayer(localPlayer);
for (Player player : players.values()) {
    gameArgs.addPlayer(player);
}

TankCoders.getGameInstance().changeToInGameState(gameArgs);
...
```

Quadro 51 – Trecho de código referente ao tratamento da mensagem `StartBattle`

Um objeto da classe `StartGameArguments` é criado e inicializado. Nota-se também que a instância do jogador local e dos demais jogadores inserida no objeto instanciado. Após feito isso, o método `changeToInGameState` é invocado, como descrito na seção anterior e detalhado no Quadro 29.

Já no *game state* "*InGame*", esse módulo de interação com o servidor, é responsável pela sincronização das ações dos tanques de guerra. A classe central que engloba todos os tratamentos necessários é a `NVEHandler`, que fica em constante contato com a classe `GameClient`, responsável por interagir com o *middleware* JGN.

Como comentado na seção anterior, uma das ultimas tarefas do construtor da classe InGameState é configurar a classe NVEHandler. Essa configuração é feita no método setup, como mostrado no Quadro 52.

```
public static void setup(StartGameArguments gameArgs, InGameState inGameState) {
    try {
        players = new HashMap<Short, Player>();
        for (Iterator<Player> iterator = gameArgs.getPlayersIterator();
             iterator.hasNext(); ) {
            Player player = iterator.next();
            players.put(player.getId(), player);
        }

        NVEHandler.localPlayer = gameArgs.getLocalPlayer();
        NVEHandler.tanks = inGameState.getTanks();
        NVEHandler.hud = inGameState.getHud();
        NVEHandler.inGameState = inGameState;
        NVEHandler.gameClient = GameClient.getInstance();
        NVEHandler.syncObjectManager = new SyncTankObjectManager(tanks);

        setupListeners();
        setupSynchronizationManager();

        // notify the server that I change to ingame state.
        gameClient.sendToServer(new PlayerInBattle());
    } catch (Exception e) {
        // ok, no problem with this exception.
    }
}
```

Quadro 52 – Método `setup` da classe `NVEHandler`

As tarefas mais relevantes executadas nesse método são as invocações dos métodos `setupListeners` e `setupSynchronizationManager`. No primeiro deles são registrados os objetos (*listeners*) que tratam eventos de conexão e mensagem. Dentre as mensagens que podem ser recebidas e tratadas no correspondente objeto *listener*, as mais relevantes são: `TankActionShotMainGun`, `TankActionShotMachineGun` e `TankBulletHit`. As duas primeiras, cuja forma de tratá-las é semelhante, consistem na notificação de disparo de um projétil por um tanque de um jogador remoto. A outra mensagem consiste na notificação da colisão de um projétil com um tanque de guerra do jogador local.

A forma como são tratadas as notificações de disparo de projéteis remotos[15] apresenta certa complexidade, pois existe o problema de *threads* concorrentes sendo executadas que precisa ser contornado. Todo e qualquer processamento que faz uso do *framework* de física (JME Physics) precisa ser executado na *thread* principal da *engine* JME, ou seja, a *thread* que contempla o *loop* principal da aplicação. No caso, o disparo de um projétil remoto, ocorre na *thread* criada na classe `GameClient` para a interação com o aplicativo servidor. Portanto, para que o projétil seja disparado corretamente, optou-se por criar uma fila de projéteis na classe

[15] Projétil remoto é referenciado no texto como sendo um projétil disparado por um tanque de um jogador remoto.

`AbstractTank` e quando o disparo precisa ocorrer, é adicionado um elemento na fila para que seja processado na próxima execução do método `update`. Dessa forma, o disparo do projétil é de fato executado dentro da *thread* principal da JME. Existe uma fila de projéteis para cada arma do tanque. O código mostrado no Quadro 53 exibe a implementação dos dois métodos que tratam a notificação dos disparos remotos.

```
private void processTankActionShotMachineGun(Message m) {
    TankActionShotMachineGun tasmg = (TankActionShotMachineGun)m;
    tanks.get(tasmg.getTankName()).addFireMachineGunBullet(Bullet.REMOTE);
}

private void processTankActionShotMainGun(Message m) {
    TankActionShotMainGun tasmg = (TankActionShotMainGun)m;
    tanks.get(tasmg.getTankName()).addFireMainGunBullet(Bullet.REMOTE);
}
```

Quadro 53 – Métodos que tratam a notificação de disparos remotos

Na classe `AbstractTank`, o método update é responsável também por verificar se existem projéteis a serem disparados, caso existe o projétil é disparado localmente. Esse processo é mostrado no trecho de código do Quadro 54.

```
...
// fire main gun bullets task
for (int i = 0; i < fireMainGunBullets.size(); i++) {
    short type = fireMainGunBullets.poll();

    switch (type) {
        case Bullet.LOCAL: fireMainGunBullet(); break;
        case Bullet.REMOTE: fireMainGunBulletRemote(); break;
    }
}

// fire machine gun bullets task
for (int i = 0; i < fireMachineGunBullets.size(); i++) {
    short type = fireMachineGunBullets.poll();

    switch (type) {
        case Bullet.LOCAL: fireMachineGunBullet(); break;
        case Bullet.REMOTE: fireMachineGunBulletRemote(); break;
    }
}
...
```

Quadro 54 – Trecho do método `update` responsável pelo disparo de projéteis enfileirados

Nota-se nesse método que, além de suportar tiros remotos (`Bullet.REMOTE`), é suportado também disparos locais. Essa modalidade de disparos é usada pelo módulo de integração com o interpretador Jason, como é visto na próxima seção. Os métodos `fireMainGunBulletRemote` e `fireMachineGunBulletRemote` são semelhantes aos métodos `fireMainGunBullet` e `fireMachineGunBullet`, a única diferença é que em disparos remotos, no projétil, é definido o atributo `canDamage` igual a falso. Isso faz com que o tiro não afete a saúde de um tanque caso ocorra colisão. Um projétil só afeta a saúde de um tanque caso seja disparado por um tanque do jogador local, por isso que é enviada a mensagem

`TankBulletHit` ao tanque afetado quando ocorre a colisão. O Quadro 55 mostra um trecho do método `update` da classe já comentada `ProjectileCollisionController` que trata a colisão do projétil com os tanques de guerra.

```
...
// check collisions with tanks (except the bullet's owner)
for (ITank tank : tanks.values()) {
    if (tank != tankOwner) {
        if (tank.intersectsWith(bv)) {
            // notify the affected tank if this projectile can damage
            // another tank.
            if (projectile.isCanDamage()) {
                tank.hitByBullet(projectile);

                if (tankOwner.isAgentControls()) {
                    boolean isEnemy = tankOwner.getTeam().getTeamEnum() !=
                                      tank.getTeam().getTeamEnum();
                    AgentRepository.notifyBulletHitTank(tankOwner.
                              getTankName(), tank.getTankName(), isEnemy);
                }
            }

            projectile.removeFromScene();
            return;
        }
    }
}
...
```

Quadro 55 – Trecho do método `update` que trata colisão do projétil com os tanques

Nota-se nesse trecho de código que, após notificar o tanque a respeito da colisão (invocando o método `hitByBullet`) a bala é removida do cenário. Esse método também realiza uma tarefa relacionada à integração com o interpretador Jason, que é vista na próxima seção.

Caso um tanque do jogador local seja afetado por um projétil em outra instância de aplicativo cliente (na instância de simulador de um jogador remoto), é recebida a mensagem `TankBulletHit` e o método `processTankBulletHit` da classe `NVEHandler` executa o método `hitByRemoteBullet` do tanque de guerra correspondente (classe `AbstractTank`). O código fonte dos método `hitByBullet` e `hitByRemoteBullet` são mostrados no Quadro 56. Nele é possível perceber em qual momento que a mensagem `TankBulletHit` é enviada ao jogador do tanque afetado.

Voltando ao método `setup` da classe `NVEHandler`, após criar e configurar os objetos *listeners*, é invocado o método `setupSynchronizationManager`. Ele é responsável por criar o gerenciador de sincronização da movimentação dos tanques de guerra ao longo do cenário. No Quadro 57 é apresentado o fonte desse método.

```
public void hitByBullet(Projectile p) {
    // check if the local player is the tank's owner, if yes then check if this is
    // still alive, if not just report this shot to the player's owner of this
tank
    if (playerIsTheOwner) {
        this.health -= p.getPower();

        if (agentControls) {
            AgentRepository.notifyTankHitByBullet(tankName, p.getPower());
        }

        if (!isAlive()) {
            // before send message, I kill the tank locally (via inGameState)
            NVEHandler.sendTankDead(tankName);
        }
    }
    else {
        NVEHandler.sendTankBulletHit(playerIdOwner, tankName, p.getPower());
    }
}

public void hitByRemoteBullet(int power) {
    this.health -= power;

    if (agentControls) {
        AgentRepository.notifyTankHitByBullet(tankName, power);
    }

    if (!isAlive()) {
        // before send message, I kill the tank locally (via inGameState)
        NVEHandler.sendTankDead(tankName);
    }
}
```

Quadro 56 – Métodos `hitByBullet` e `hitByRemoteBullet`

```
private static void setupSynchronizationManager() {
    TankGraphicalController controller = new TankGraphicalController();

    clientSyncManager = gameClient.createSyncManager(controller);
    clientSyncManager.addSyncObjectManager(NVEHandler.syncObjectManager);
    JGN.createThread(clientSyncManager).start();
}
```

Quadro 57 – Método `setupSynchronizationManager`

A responsabilidade das classes utilizadas nesse método já foi descrita na especificação contida na seção 3.2.1.2.2. Porém a classe `TankGraphicalController`, por ser a principal classe responsável pela efetivação do movimento remoto dos tanques, tem seu fonte mostrado no Quadro 58.

No código do Quadro 57 é criada uma nova *thread* para o objeto gerenciador de sincronização (classe `SynchronizationManager`). O gerenciador, para poder sincronizar a movimentação dos tanques de guerra, precisa que os mesmos estejam registrados nele. O registro dos tanques do jogador local é realizado quando todos os jogadores já estiverem transitado para o *game state* "*InGame*". O cliente sabe quando esse evento acontece ao receber a mensagem `AllPlayersAreInGameState`. O código responsável pelo tratamento dessa mensagem é mostrado no Quadro 59.

```
public class TankGraphicalController implements GraphicalController<Spatial> {

    public void applySynchronizationMessage(SynchronizeMessage message,
                                            Spatial spatial) {
        Synchronize3DMessage m = (Synchronize3DMessage)message;
        Spatial chassi = ((AbstractTank)spatial).getStaticChassi();

        chassi.getLocalTranslation().x = m.getPositionX();
        chassi.getLocalTranslation().y = m.getPositionY();
        chassi.getLocalTranslation().z = m.getPositionZ();
        chassi.getLocalRotation().x = m.getRotationX();
        chassi.getLocalRotation().y = m.getRotationY();
        chassi.getLocalRotation().z = m.getRotationZ();
        chassi.getLocalRotation().w = m.getRotationW();

        // M1Abrams also has a main gun (articulated object)
        if (spatial instanceof M1AbramsTank) {
            Node mainGunNode = ((M1AbramsTank)spatial).getMainGun().getMainNode();
            SynchronizeArticulatedObject3DMessage ma =
                                (SynchronizeArticulatedObject3DMessage)message;

            mainGunNode.setLocalTranslation(chassi.getLocalTranslation().add(
                                        new Vector3f(-2.2f, 2.2f, -0.48f)));

            mainGunNode.getLocalRotation().x = ma.getMainGunRotationX();
            mainGunNode.getLocalRotation().y = ma.getMainGunRotationY();
            mainGunNode.getLocalRotation().z = ma.getMainGunRotationZ();
            mainGunNode.getLocalRotation().w = ma.getMainGunRotationW();
        }
    }

    public SynchronizeMessage createSynchronizationMessage(Spatial spatial) {
        Synchronize3DMessage message = (spatial instanceof JadgPantherTank) ?
                                    new Synchronize3DMessage() :
                                    new SynchronizeArticulatedObject3DMessage();

        Spatial chassi = ((AbstractTank)spatial).getChassi();

        message.setPositionX(chassi.getLocalTranslation().x);
        message.setPositionY(chassi.getLocalTranslation().y);
        message.setPositionZ(chassi.getLocalTranslation().z);
        message.setRotationX(chassi.getLocalRotation().x);
        message.setRotationY(chassi.getLocalRotation().y);
        message.setRotationZ(chassi.getLocalRotation().z);
        message.setRotationW(chassi.getLocalRotation().w);

        // M1Abrams also has a main gun (articulated object)
        if ((spatial instanceof M1AbramsTank)) {
            Node mainGunNode = ((M1AbramsTank)spatial).getMainGun().getMainNode();
            SynchronizeArticulatedObject3DMessage ma =
                                (SynchronizeArticulatedObject3DMessage)message;

            ma.setMainGunRotationX(mainGunNode.getLocalRotation().x);
            ma.setMainGunRotationY(mainGunNode.getLocalRotation().y);
            ma.setMainGunRotationZ(mainGunNode.getLocalRotation().z);
            ma.setMainGunRotationW(mainGunNode.getLocalRotation().w);
        }

        return message;
    }


...
```

Quadro 58 – Classe `TankGraphicalController`

```
private void processAllPlayersAreInGameStateMessage() {
    try {
        // register all the tanks of local player
        for (PlayerTank pTank : localPlayer.getTanks()) {
            ITank tank = tanks.get(pTank.getTankName());
            clientSyncManager.register(tank,
                                       tank.getSynchronizeCreateMessageImpl(),50);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }

    // remove message from the hud
    hud.removeWaitingOtherPlayersMessage();

    // notify the inGameState that all players are in game state.
    inGameState.allPlayersAreInGameState();
}
```

Quadro 59 – Método `processAllPlayersAreInGameStateMessage`

Através do método `register` da classe já instanciada `SynchronizationManager`, um tanque de guerra passa a ser considerado pelo gerenciador de sincronização. Dessa forma, a sincronização da movimentação de cada tanque ocorre a cada cinqüenta milissegundos. Nota-se também que, antes de finalizar a execução do método `update`, é invocado o método `allPlayersAreInGameState` sobrescrito nas subclasses de `InGameState`.

Na classe `AvatarPlayerInGameState` o método `allPlayersAreInGameState` é responsável por criar os controles de teclado para o jogador do tipo *avatar* poder interagir com o tanque de guerra que ele controla. Também são definidas as teclas para controlar a câmera do cenário.

Na classe `MasPlayerInGameState` o método `allPlayersAreInGameState` também é responsável por criar os controles de teclado, porém somente para controlar a câmera do cenário. Esse método possui ainda a responsabilidade de notificar os agentes que controlam os tanques de guerra a respeito do início da batalha, porém isso é visto na seção seguinte que aborda a integração com o interpretador Jason.

#### 3.3.2.2.3 Implementação da integração com o interpretador Jason

O último módulo que completa a implementação do aplicativo cliente do simulador é o que trata a integração com o interpretador Jason. Como já comentado na especificação, a integração é possível devida à possibilidade de personalizar determinados componentes pré-implementados na API do interpretador.

Conforme estudado e apresentado na fundamentação teórica sobre o interpretador

Jason, foi possível notar que a primeira classe (que pode ser personalizada pelo usuário) a ser invocada pelo Jason na execução de um SMA é a classe do ambiente. Com base nisso, foi especificada e implementada a classe `TankCodersEnvironment` para representar o ambiente de qualquer projeto SMA Jason no qual os agentes devem controlar os tanque de guerra do simulador. O código-fonte dessa classe consta no Quadro 60.

```
public class TankCodersEnvironment extends Environment {

    public void init(String[] args) {
        String agsParam = null;
        RunCentralisedMAS runner = RunCentralisedMAS.getRunner();
        if (runner != null) {
            List<AgentParameters> agParameters = runner.getProject().getAgents();

            // prepare a comma-separated list of agentNames
            for (AgentParameters agParameter : agParameters) {
                ClassParameters classPar = agParameter.archClass;
                if (classPar != null) {
                    try {
                        Class c = Class.forName(classPar.className);
                        if (c.equals(TankAgArch.class)) {
                            if (agParameter.qty == 1) {
                                String agName = agParameter.name;
                                agsParam = (agsParam == null) ?
                                     agName : agsParam.concat(",").concat(agName);
                            }
                            else {
                                for (int i=1; i <= agParameter.qty; i++) {
                                    String agName = agParameter.name.
                                                concat(String.valueOf(i));
                                    agsParam = (agsParam == null) ?
                                     agName : agsParam.concat(",").concat(agName);
                                }
                            }
                        }
                    } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                    }
                }
            }

            // invoke the TankCoders simulator with the -mas agents argument
            TankCoders.main(new String[] {"-debug", "-mas", agsParam});
        }
        else {
            throw new NotCentralisedInfraException();
        }
    }
}
```

Quadro 60 – Classe `TankCodersEnvironment`

Através da classe `RunCentralisedMAS` é possível obter a lista de todos os agentes que compõem o SMA (instâncias da classe `AgentParameters`). Em seguida a lista de agentes passa por uma verificação para considerar somente os agentes que possuem a classe `TankAgArch` definida como classe de arquitetura. Após a execução do *loop* de verificação, o método estático `main` classe `TankCoders` é invocado passando os parâmetros de inicialização do aplicativo cliente em modo MAS. Entre os parâmetros passados, o último deles consiste

em uma lista de nomes dos agentes, que durante a batalha, representaram os tanques de guerra do jogador local. Na classe `TankCoders` a lista de agentes é interpretada e armazenada no atributo `agents`. Após isso, a inicialização do aplicativo cliente segue da mesma forma que ocorre com um jogador do tipo *avatar*.

O próximo passo da integração com o interpretador Jason é realizado na criação dos tanques de guerra. Esse processo ocorre na classe `TankFactory`, conforme o Quadro 61.

```
public static ITank makeTank(short playerIdOwner, PlayerTank tank, PhysicsSpace
                             pSpace, Vector3f location, final boolean remoteTank,
                             final boolean masPlayer) {
    String tankName = tank.getTankName();
    Node tankNode = null;
    switch (tank.getModel()) {
        case M1_ABRAMS: tankNode = new M1AbramsTank(playerIdOwner, tankName,
                                tank.getTeam(), pSpace, location, remoteTank,
                                masPlayer); break;
        case JADGE_PANTHER: tankNode = new JadgPantherTank(playerIdOwner,
                                tankName, tank.getTeam(), pSpace, location,
                                remoteTank, masPlayer); break;
    }

    ITank tankObj = (ITank)tankNode;

    // check if tank´s owner is local and is a MAS player.
    if (!remoteTank && masPlayer) {
        // extract agArch instance from the centralised MAS.
        TankAgArch agArch = (TankAgArch)RunCentralisedMAS.getRunner().
                                                    getAg(tankName).getUserAgArch();

        // inject instance of the agent in the customized class of AgArch
        agArch.setTankInstance(tankObj);

        // put agent architecture in the repository map.
        AgentRepository.putTank(tankName, agArch);
    }

    return tankObj;
}
```

Quadro 61 – Método `makeTank` da classe `TankFactory`

Após a instanciação da classe concreta do tanque, é feita uma verificação para ver se o tanque já instanciado pertence ao jogador local e se esse é do tipo MAS. Caso a verificação seja verdadeira, primeiramente é obtida a instância da classe de arquitetura do agente. Após isso, a instância do tanque de guerra é definida na instância da classe de arquitetura. Por último, a instância da classe de arquitetura é inserida na estrutura de dados da classe `AgentRepository`, que é explicada adiante.

A instância da classe de arquitetura, tendo a instância do tanque de guerra que o agente controla como um atributo, é possível agir diretamente sobre ela através da invocação de métodos da classe `AbstractTank`.

Para a implementação da classe de arquitetura, inicialmente foi criada a anotação `Action`, cujo código consta no Quadro 62.

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Action {
       String value() default "";
}
```

Quadro 62 – Anotação `Action`

Essa anotação Java é utilizada para determinar quais métodos da classe `TankAgArch` são ações que podem ser executadas pelos agentes programados em AgentSpeak(L). Além da anotação, a classe `AnnotatedAgArch` (que deriva da classe `AgArch` da API do Jason) foi implementada para ser a superclasse de `TankAgArch` e prover maior abstração para ela. O Quadro 63 mostra como o construtor da classe `AnnotatedAgArch` e o método `act` foram implementados.

```
public AnnotatedAgArch() {
    annotatedMethods = new HashMap<String, Method>();

    Method[] allMethods = this.getClass().getDeclaredMethods();
    for (Method method : allMethods) {
        if (method.isAnnotationPresent(Action.class)) {
            String actionName = method.getAnnotation(Action.class).value();
            if (actionName.equals("")) {
                actionName = method.getName();
            }

            annotatedMethods.put(actionName, method);
        }
    }
}

/**
 * This method is called when an agent wants to execute an action.
*/
public void act(ActionExec action, List<ActionExec> feedback) {
    super.act(action, feedback);

    Structure actionTerm = action.getActionTerm();
    String functor = actionTerm.getFunctor();
    Method m = annotatedMethods.get(functor);
    if (m != null) {
        try {
            m.invoke(this, (Object)actionTerm.getTermsArray());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Quadro 63 – Classe `AnnotatedAgArch`

Através da API de reflexão do Java (THE REFLECTION API, 2008), no construtor é possível obter a lista de todos os métodos declarados na classe e nas subclasses. A partir dessa lista, são filtrados apenas os métodos que possuem a anotação `@Action`. Essa lista filtrada é armazenada no atributo `annotatedMethods`, para que quando ocorra a execução do método `act` (pelo interpretador Jason) o método correspondente à ação seja executado. No Quadro 63 percebe-se ainda que para executar o método referente à ação é utilizado o nome da mesma

para obtê-lo.

Podendo contar com os recursos implementados, a classe `TankAgArch`, que é a classe de arquitetura oficial, possui os mecanismos para interagir com a mente do agente, programada através da linguagem AgentSpeak(L). Essa interação com a mente consiste na execução de ações e na inserção de percepções e crenças.

Todos os métodos de ações da classe `TankAgArch` podem ser vistos na Figura 25. Porém o Quadro 64 ilustra como é implementado alguns deles.

```
@Action
public void ahead(Term[] terms) {
    tankInstance.setAhead(getParamAsInt(0, terms));
}


@Action
public void back(Term[] terms) {
    tankInstance.setBack(getParamAsInt(0, terms));
}


@Action
public void fireMainGun(Term[] terms) {
    tankInstance.addFireMainGunBullet(Bullet.LOCAL);
}
```

Quadro 64 – Alguns métodos de ações da classe `TankAgArch`

As percepções são inseridas através do método `addPercept`, onde é passado por parâmetro uma instância da classe `Literal` da API do Jason. Já as crenças são adicionadas através de métodos específicos comentados mais adiante.

Para a adição das percepções e crenças dos agentes, surge a classe `AgentRepository`. Como já comentada na especificação, trata-se de um ponto central para notificar os eventos que ocorrem durante a execução da batalha. Essa classe possui um atributo do tipo `Map` que armazena as instâncias das classes de arquitetura por nome de agente, sendo que ele e todos os métodos dessa classe são estáticos. Dessa forma, não é necessário ter uma instância dessa para acessar os seus recursos.

Dentre todas as notificações descritas na especificação da classe `AgentRepository` realizada na seção 3.2.1.2.3, as principais constam listadas no Quadro 65. O método `notifyTankInfo` é utilizado para notificar periodicamente (tempo configurável) as correntes informações de um tanque específico. Nota-se que as informações englobam a posição do tanque, saúde, quantidade de balas da arma principal, ângulo do chassi e ângulo da arma principal. Baseado nessas informações, os agentes podem executar estratégias previamente implementadas pelo programador do projeto Jason.

```
public static void notifyTankInfo(String tankName, Vector3f p, int health, int
                                  mainGunQtyBullets, float chassiHeading,
                                  float mainGunHeading) {
    synchronized (tankAgArchs) {
        Literal info = new Literal("info");
        info.addTerms(new NumberTermImpl(p.x), new NumberTermImpl(p.y),
                      new NumberTermImpl(p.z), new NumberTermImpl(health),
                      new NumberTermImpl(mainGunQtyBullets),
                      new NumberTermImpl(chassiHeading),
                      new NumberTermImpl(mainGunHeading));

        // add the new position as belief
        tankAgArchs.get(tankName).addInfoBel(info);
    }
}

public static void notifyActionFinished(String tankName) {
    Literal actionFinished = new Literal("actionFinished");
    tankAgArchs.get(tankName).addActionFinishedBel(actionFinished);
}

public static void notifyRadarScanTank(String tankName, String tankNameScanned,
                                       Vector3f p, boolean isEnemy) {
    if (tankAgArchs.containsKey(tankName)) {
        Literal scanTank = new Literal("onScanTank");
        scanTank.addTerms(new StringTermImpl(tankNameScanned),
                          new NumberTermImpl((isEnemy) ? 1 : 0),
                          new NumberTermImpl(p.x), new NumberTermImpl(p.y),
                          new NumberTermImpl(p.z));

        tankAgArchs.get(tankName).addPercept(scanTank);
    }
}

public static void notifyTankHitByBullet(String tankName, int bulletPower) {
    if (tankAgArchs.containsKey(tankName)) {
        Literal hitByBullet = new Literal("onHitByBullet");
        hitByBullet.addTerms(new NumberTermImpl(bulletPower));
        tankAgArchs.get(tankName).addPercept(hitByBullet);
    }
}
```

Quadro 65 – Principais notificações de eventos da batalha

A última tarefa a ser comentada a respeito da integração com o interpretador Jason é a implementação do dispositivo de radar dos tanques, que é realizada no método `update` da classe `MasPlayerInGameState`. Nesse método, periodicamente, para cada tanque de guerra controlado por um agente, é verificado se algum outro tanque está posicionado dentro do raio de abrangência do radar, se estiver, ambos os tanques são notificados do evento ocorrido. A implementação completa desse dispositivo é apresentada no Quadro 66.

Durante o processamento do dispositivo de radar, por estar iterando todos os tanques de guerra controlados por agentes, aproveita-se também para verificar se ocorre colisão entre os tanques. Caso ocorra, ambos os agentes que controlam os tanques envolvidos são notificados através do método `notifyHitTank` da classe `AgentRepository`.

```
public void update(float f) {
    super.update(f);

    if (lastUpdateTime + notifyTankInfoRate < System.nanoTime()) {
        checkTanksRadar();

        lastUpdateTime = System.nanoTime();
    }
}


private void checkTanksRadar() {
    int tanksListSize = tanksList.size();

    for (int i = 0; i < tanksListSize-1; i++) {
        ITank tankI = tanksList.get(i);

        PlayerTeam tankITeam = tankI.getTeam().getTeamEnum();
        Vector3f posTankI = tankI.getWorldPosition();
        boolean tankIAgentControls = tankI.isAgentControls();

        for (int j = i+1; j < tanksListSize; j++) {
            ITank tankJ = tanksList.get(j);
                boolean tankJAgentControls = tankJ.isAgentControls();

                if (tankIAgentControls || tankJAgentControls) {

                PlayerTeam tankJTeam = tankJ.getTeam().getTeamEnum();
                Vector3f posTankJ = tankJ.getWorldPosition();
                boolean isEnemy = (tankITeam != tankJTeam);

                float distance = posTankI.subtract(posTankJ).length() / 2;
                if (distance < GameRulesConstants.RADAR_SCOPE) {
                    if (tankIAgentControls)
                          AgentRepository.notifyRadarScanTank(tankI.getTankName(),
                                        tankJ.getTankName(), tankJ.getMainNode().
                                        getLocalTranslation(), isEnemy);
                    if (tankJAgentControls)
                          AgentRepository.notifyRadarScanTank(tankJ.getTankName(),
                                        tankI.getTankName(), tankI.getMainNode().
                                        getLocalTranslation(), isEnemy);

                    // is really near then check intersects.
                    if (distance < 90) {
                        if (tankI.intersectsWith(tankJ.getWorldBound())) {
                            if (tankIAgentControls)
                                AgentRepository.notifyHitTank(tankI.getTankName(),
                                                tankJ.getTankName(), isEnemy);
                            if (tankJAgentControls)
                                AgentRepository.notifyHitTank(tankJ.getTankName(),
                                                tankI.getTankName(), isEnemy);
                        }
                    }
                }
            }
        }
    }
}
```

Quadro 66 – Implementação do dispositivo de radar no *game state* `MasPlayerInGameState`

Com isso, a apresentação da implementação do aplicativo cliente é finalizada. A seguir é descrito um estudo de caso mostrando a operacionalidade da implementação realizada. Após isso, o capítulo é concluído com a apresentação dos resultados obtidos confrontados com os trabalhos correlatos elencados.

### 3.3.3 Operacionalidade da implementação

A página WEB de documentação do simulador, entre outras utilidades, demonstra os passos para iniciar o aplicativo servidor e também o cliente. Para o aplicativo cliente, duas maneiras de inicialização são possíveis, uma para cada tipo de jogador, que pode ser *avatar* ou MAS. As seções subseqüentes apresentam um estudo de caso, mostrando primeiro a inicialização e criação de uma batalha no aplicativo servidor. Em seguida é mostrado como um jogador tipo *avatar* entra na batalha, prepara seu tanque e fica pronto para batalhar. Por último é descrito como um jogador tipo MAS conecta-se à batalha, também preparando seus tanques e iniciando a batalha.

#### 3.3.3.1 O aplicativo servidor

Quando o aplicativo servidor é executado, a tela apresentada na Figura 27 é exibida. Nela o usuário (não necessariamente um jogador) deve informar o nome da batalha, o nome dos times, a quantidade máxima de tanques em cada time e a imagem do terreno. Depois de informado, o usuário pode iniciar a batalha clicando no botão “Start Server”.


Figura 27 – Tela do aplicativo servidor

### 3.3.3.2 O aplicativo cliente para um jogador tipo *avatar*

Ao executar o aplicativo cliente, é exibida a tela mostrada na Figura 28. Nela o jogador pode escolher a opção “Join Game” para ir à página do menu onde é selecionado o servidor. As demais opções do menu principal não apresentam uma importância significante.


Figura 28 – Tela inicial do aplicativo cliente

Na Figura 29 é mostrada a página do menu onde é selecionado o servidor da batalha de tanques de guerra. O jogador deve informar um nome, selecionar o servidor na lista e por último escolher a opção “Join” para conectar-se ao servidor.

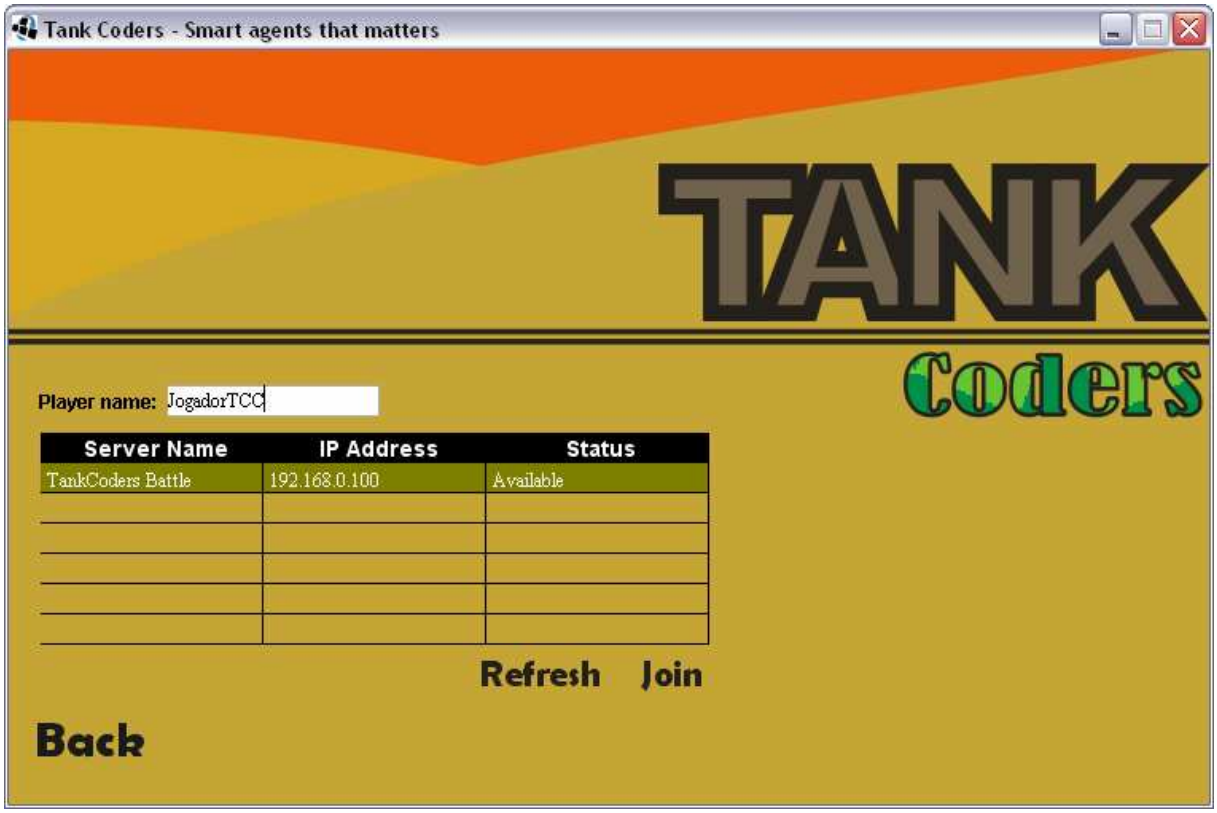



Figura 29 – Tela do menu de seleção do servidor

Após a conexão, pode ser visto na tela do aplicativo servidor que o jogador aparece na lista de usuários conectados, como ilustrado na Figura 30.

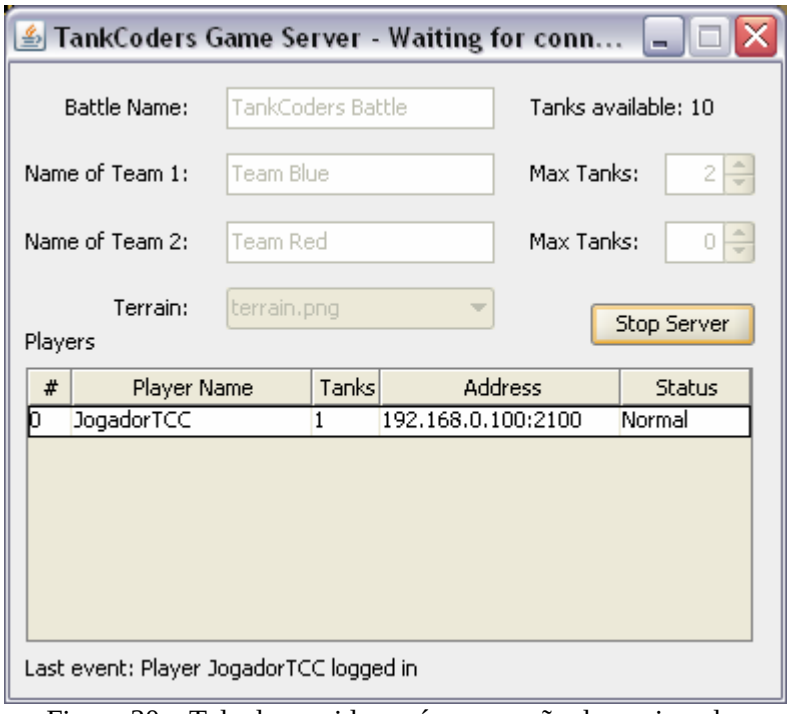



Figura 30 – Tela do servidor após a conexão de um jogador

A próxima página do menu consiste na preparação da batalha. O jogador do tipo *avatar* só tem direito de mudar as informações referentes ao seu único tanque. Logo, ele pode

mudar o modelo do tanque de guerra e também alternar o time no qual ele compõe. A tela desse menu é mostrada na Figura 31.


Figura 31 – Tela do menu de preparação da batalha com um jogador conectado

Se o jogador pressionar o botão “Ready” a batalha já é iniciada. Isso acontece, pois no momento só um jogador consta na lista do servidor e o mesmo apresenta-se como pronto para batalhar, sendo assim o servidor compreende que a batalha deve ser iniciada. Com base nisso, para que o jogador não inicie a batalha sozinho, o mesmo deve aguardar até que outros jogadores se conectem. Na seção seguinte, um jogador MAS também conecta-se ao servidor, portanto quando isso ocorrer, ambos os jogadores podem pressionar o botão “Ready”.

### 3.3.3.3 O aplicativo cliente para um jogador tipo MAS

Antes de executar o aplicativo cliente como um jogador do tipo MAS, existe o processo de desenvolvimento do SMA que contém os agentes responsáveis pelo controle dos tanques de guerra no cenário do simulador. A Figura 32 exibe um projeto Jason criado dentro do *plug-in* da IDE Eclipse, evidenciando o conteúdo do arquivo de configuração do projeto Jason (arquivo com extensão “.mas2j”). A primeira configuração feita é em relação à infra-estrutura, que para o simulador TankCoders, só é permitida a opção “Centralised”. Em

seguida é informada a classe do ambiente, que no caso do simulador, sempre deve ser informada a classe "`br.furb.inf.tcc.tankcoders.jason.TankCodersEnvironment`". Por último, para cada agente do SMA (no caso só existe um agente chamado "*sample*") deve ser informada a classe de arquitetura "`br.furb.inf.tcc.tankcoders.jason.TankAgArch`".

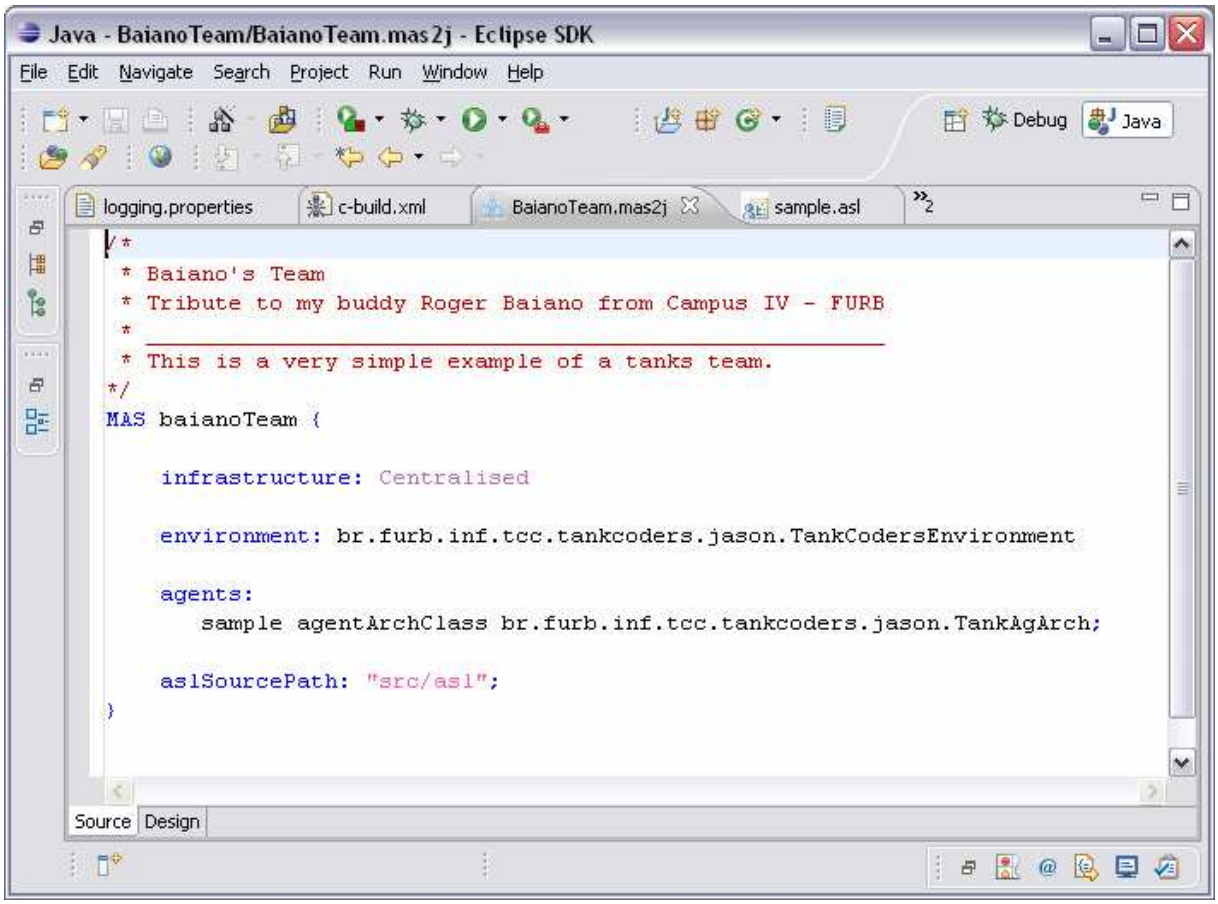



Figura 32 – Projeto de um SMA no *plug-in* do Jason

O código AgentSpeak(L) do agente chamado "sample" é mostrado na Figura 33. Trata-se de um agente bastante simples, utilizado somente para demonstrar a implementação de alguns eventos e planos. Nesse agente, quando ocorre o evento `+startBattle` o agente tem como meta executar o plano `startMyBattlePlan`. Esse plano consiste em imprimir o texto "começou" no console, mover o tanque vinte metros para frente, esperar até que ação de movimentação termine e por último disparar um tiro com a arma principal. No código-fonte desse agente também consta a implementação dos eventos `+onHitByBullet` e `+onHitTank`, que tratam, respectivamente, a colisão de um tiro e a colisão de outro tanque com o tanque controlado pelo agente.

Para que o desenvolvedor do SMA saiba quais eventos (através de percepções e crenças) podem ocorrer durante a batalha e quais ações estão disponíveis para serem executadas, a documentação WEB do projeto serve para auxiliar. Nela é possível consultar

informações detalhadas de cada percepção, crença e ação de um agente. O Apêndice B apresenta de forma completa todos os fontes do projeto Jason desse estudo de caso.

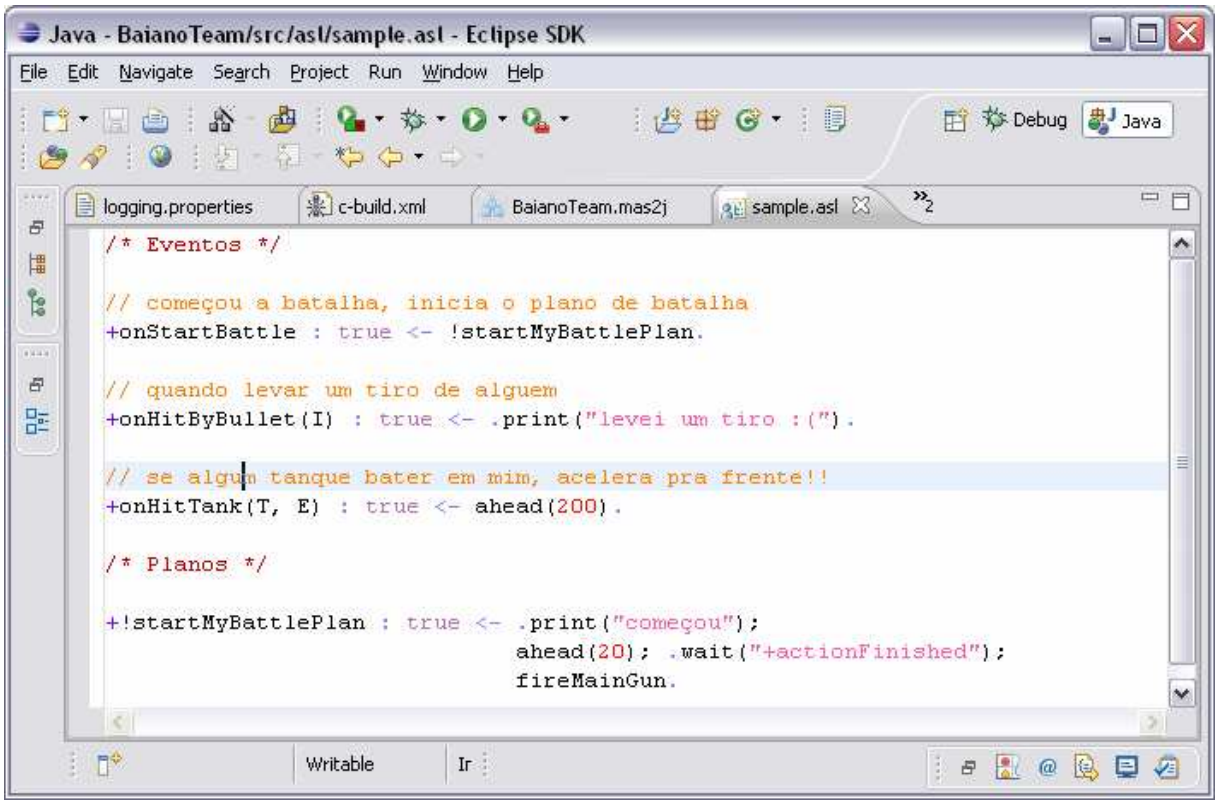



Figura 33 – Código AgentSpeak(L) do agente "*sample*" no *plug-in* do Jason

Tendo o SMA desenvolvido no Jason, o aplicativo cliente do simulador pode ser executado para batalhar com outros jogadores. Para realizar a execução do projeto no *plug-in* do Jason para a IDE Eclipse, o usuário clica com o botão direito sobre o arquivo de configuração do projeto (arquivo ".mas2j") e escolhe a opção "*Run in Jason*". Após isso o aplicativo cliente inicia da mesma forma que ocorre com o jogador tipo *avatar*, passando pelas mesmas páginas de menu.

Na página do menu de preparação da batalha, ambos os jogadores conectados pressionam o botão "*Ready*", fazendo com que a batalha comece a ser carregada, exibindo a tela apresentada na Figura 34.

Ao iniciar a batalha, o tanque chamado "sample" (controlado pelo agente "sample"), executa seu plano inicial (`startMyBattlePlan`), que é se mover para frente (`ahead`) vinte metros. Já o jogador tipo *avatar*, através das teclas A, S, D e W, movimenta seu tanque no cenário. A tecla "barra de espaço" é pressionada para disparar um tiro da arma principal, sendo que a tecla "*control*" é pressionada para disparar um tiro da metralhadora. Caso o tanque seja do modelo "M1 Abrams", pode-se ainda movimentar a base da arma principal para os lados, isso é feito pressionando as teclas direcionais. A Figura 35 mostra o cenário 3D

da batalha logo após o tanque do jogador tipo MAS ser acertado por um tiro disparado pelo tanque do jogador tipo *avatar*. Nota-se que a saúde (*health*) do tanque afetado passa para 80%, sendo que quando chega em 0% o tanque “morre”.


Figura 34 – Tela exibida ao iniciar o carregamento do cenário


Figura 35 – Cenário 3D da batalha

Na Figura 35 pode-se notar ainda que o cenário apresentou uma taxa de 10 FPS, isso ocorreu pois nessa execução foi utilizado um computador com placa gráfica contendo baixo poder de processamento e pouca memória.

A Figura 36 apresenta o *console* do interpretador Jason onde podem ser vistas as mensagens de log do aplicativo cliente. Também podem ser vistas as mensagens impressas através da ação interna `.print` que os agentes executam. É possível observar que, na penúltima linha do *console* é impressa a mensagem "[sample] levei um tiro :(". Mensagem que é impressa quando ocorre o evento `+onHitByBullet`.

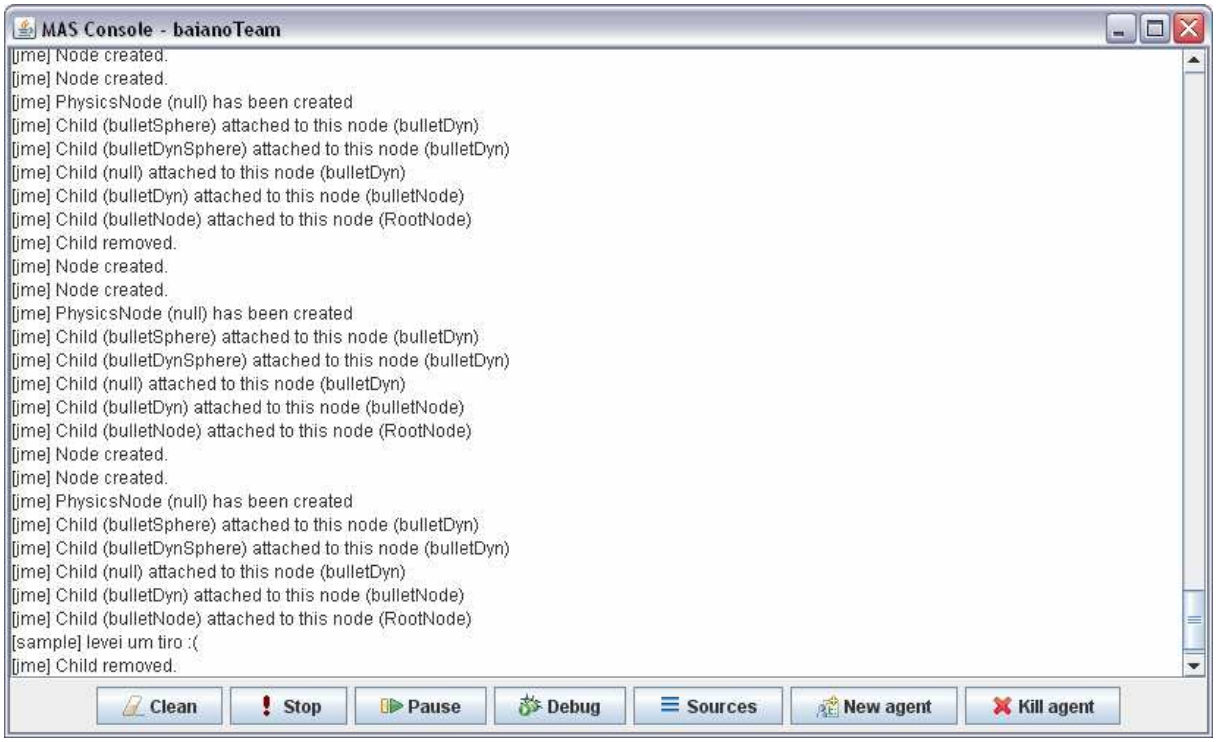



Figura 36 – *Console* do interpretador Jason

Quando um jogador do tipo MAS tiver mais de um agente, é possível alternar o foco da câmera que segue um tanque específico (alvo da câmera). Para realizar isso, a tecla "Tab" é pressionada e outro tanque do jogador passa a ser seguido pela câmera.

Para ambos os tipos de jogadores, após a "morte" de todos os seus tanques, uma câmera "livre" (*free-look*) é configurada para que o jogador possa acompanhar o andamento do resto da batalha. Para movimentar essa câmera o jogador utiliza as teclas A, S, D e W, juntamente com o mouse.

A batalha é executada até que um time fique sem nenhum tanque. No estudo de caso apresentado nessa seção, ambos os tanques pertenciam ao time azul (time 1), isso só foi feito por questões de demonstração. Ao finalizar a batalha, o aplicativo cliente volta para o menu de seleção do servidor, notificando o time vencedor.

## 3.4 RESULTADOS E DISCUSSÃO

Com o desenvolvimento do presente trabalho pôde-se enumerar diversos resultados em relação às três grandes áreas de pesquisa utilizadas. Tratam-se de resultados cujo principal foco é contribuir para o desenvolvimento de novos projetos com características semelhantes.

Os principais resultados obtidos foram:

a) a utilização do *middleware* JGN para a comunicação entre os clientes e o servidor apresentou-se como uma boa solução, pois além de prover um alto nível de abstração ao desenvolvedor, permitindo que mensagens sejam representadas por classes, faz uso de conceitos estudados na fundamentação teórica, tal como *broadcast*, e os protocolos UDP e TCP;
b) a *engine* gráfica JME também proporcionou um alto nível de abstração para o desenvolvimento do cenário 3D. O principal fato que contribui para essa abstração é que a JME trabalha com o conceito de grafo de cena, onde os elementos do cenário puderam ser estruturados hierarquicamente de forma bastante clara de ser compreendida;
c) o cenário 3D do simulador apresentou um bom nível de imersão junto ao jogador. Sendo que diversos comportamentos do mundo real foram inseridos na sua construção, tal como gravidade, massa e material dos objetos, colisão entre objetos dinâmicos, suspensão dos tanques e disparo de projéteis com adição de força;
d) a definição de uma câmera chamada "*Chase Camera*" (tipo de câmera que fica localizada sempre atrás do tanque) contribuiu principalmente para melhorar a visão do jogador para efetuar a mira ao disparar os projéteis;
e) a versão 1.1.1 do interpretador Jason apresentou bom desempenho ao executar as ações dos agentes sobre os tanques de guerra no cenário. Esse fato também contribuiu principalmente para aumentar o nível de realidade que é tão requisitado na construção de ambientes virtuais;
f) as ações, percepções e crenças disponibilizadas (e bem documentadas) pelo simulador, apresentam uma forma abstrata de interagir a mente do agente programada em AgentSpeak(L) com o taque de guerra no cenário. Por exemplo, para mover o tanque para frente, a ação `ahead` é disponibilizada, sendo que a mesma recebe por parâmetro um valor em metros que o tanque deve ser movido;
g) a utilização da ferramenta Ant apresentou-se como uma maneira prática para

realizar tarefas repetitivas, tal como: executar os aplicativos, gerar arquivo JAR e gerar o Javadoc.

A maioria dos resultados descritos acima foram notados (obtidos) durante a realização da fase de testes do simulador. Essa etapa consistiu na execução do simulador em seis computadores, sendo que eles apresentavam as mesmas configurações de hardware. Tratavam-se de microcomputadores com processador Pentium HT de 2.8Ghz, 1Gb de memória RAM e placa gráfica GeForce FX 5200 de 128Mb de memória dedicada.

Fazendo uma relação com os trabalhos correlatos elencados, pode-se perceber que o simulador desenvolvido possui diversas características derivadas deles. A partir do projeto Robocode, o estilo das ações, percepções e crenças disponibilizadas aos agentes do simulador foram baseadas na API que ele disponibiliza para o desenvolvimento dos seus tanques. Dessa forma, um utilizador desse projeto pode facilmente passar a utilizar também o simulador TankCoders, sendo que a única diferença, em nível de criação dos tanques, é o paradigma de programação. Porém, o simulador desenvolvido apresenta um ambiente mais atrativo para aplicação de conceitos de inteligência artificial, sendo que o Robocode apresenta um cenário em duas dimensões e também não permite que seja executado em computadores distintos (jogadores remotos).

A partir do trabalho desenvolvido pelo acadêmico Alisson Rafael Appio, cujo nome é “Sistema Multiagentes Utilizando a Linguagem AgentSpeak(L) para Criar Estratégias de Armadilha e Cooperação em um Jogo Tipo PacMan”, pôde-se retirar idéias a respeito da integração feita entre o jogo denominado PacMan_MAS e o interpretador Jason. A principal vantagem do TankCoders em relação ao jogo citado é que os agentes desenvolvidos para o simulador precisam considerar que estão contidos em um ambiente 3D, fazendo com que mais realismo seja proporcionado, conseqüentemente aumentando o grau de complexidade no desenvolvimento do SMA. Isso faz com que técnicas mais avançadas possam ser aplicadas pelos estudantes da área de inteligência artificial.

Por último, o trabalho do acadêmico Vandeir Eduardo, denominado o “Protótipo de um Ambiente Virtual Distribuído Multiusuário”, contribuiu principalmente para o levantamento das técnicas a serem consideradas na construção do AVD do simulador. No projeto desse trabalho, utilizou-se a *engine* oficial do Java para construção de aplicações gráficas, denominada Java3D. Uma vantagem do TankCoders em relação a esse trabalho é que fez-se uso de uma *engine* gráfica, ainda não utilizada por trabalhos de conclusão de curso na FURB, chamada JME.

# 4 CONCLUSÕES

Atualmente, ao iniciar um estudo do paradigma de POA, uma dificuldade que se tem é imaginar a aplicabilidade de tais técnicas em sistemas mais complexos. Isso acontece, pois para quem está iniciando o estudo, normalmente não possui conhecimento necessário para a elaboração de um sistema mais robusto e por sua vez mais interessante. Visando contornar esta dificuldade, o simulador desenvolvido neste trabalho oferece um ambiente mais atrativo para a prática de técnicas de POA. Com esse simulador, estudantes desta área podem se concentrar apenas na elaboração dos seus agentes, sem se preocupar com a construção do ambiente onde os agentes atuam, que quase sempre é a etapa que mais consome tempo.

Como o objetivo principal do presente trabalho era construir um simulador de um ambiente virtual distribuído em 3D para batalhas de tanques de guerra onde os tanques são controlados por *avatares* ou agentes autônomos, vários aspectos foram estudados e apresentados. Dessa forma, para o sucesso na realização desse objetivo, foi necessário definir e utilizar determinadas ferramentas, *frameworks* e *middlewares* para que uma maior abstração fosse provida no desenvolvimento dos aplicativos do simulador.

O primeiro *middleware* estudado foi o JGN. Apesar de ter-se apresentado como uma boa solução para o desenvolvimento do simulador, principalmente no início foram encontrados muitos problemas para compreender seu funcionamento. Isso se deve ao fato de não possuir uma documentação bem definida. O único suporte obtido foi através de um fórum na WEB criado para esse *middleware*.

Para prover a possibilidade de controlar os tanques através de agentes BDI, a linguagem AgentSpeak(L) do interpretador Jason, por colocar em prática os conceitos da arquitetura BDI, tornou-se uma boa escolha principalmente para a programação de agentes cognitivos, que são aqueles que possuem habilidades sociais para atingirem suas metas. A ferramenta Jason, além de possuir diversas qualidades, como boa documentação, tutoriais e documentação da API, também torna-se uma boa opção justamente por ter como um dos desenvolvedores, o professor Jomi Fred Hübner do Departamento de Sistemas e Computação (DSC) da FURB. Para a implementação do time de agentes utilizado na operacionalidade da implementação, foi utilizado o *plug-in* do Jason para a IDE Eclipse, que foi inclusive desenvolvido pelo mesmo autor do presente trabalho em parceria com os professores Jomi F. Hubner e Rafael H. Bordini da universidade de Durhan na Inglaterra. Constatou-se ainda que a linguagem AgentSpeak(L) é muito poderosa para o desenvolvimento de aplicações onde o

ambiente dos agentes muda constantemente. Por ser uma linguagem declarativa, a programação torna-se mais elegante, proporcionando um alto nível de abstração na especificação das atitudes mentais dos agentes (crenças, desejos e intenções).

Por último, fez-se uso também da *engine* gráfica JME e do *framework* JME Physics. Com esses recursos, pôde-se criar um cenário 3D com elementos bastante convincentes, ou seja, provendo um bom nível de imersão ao jogador. Esse fato tornou-se muito importante para que os objetivos do trabalho pudessem ser alcançados. Tanto a *engine*, como o *framework* de física apresentou uma boa documentação da API e também aplicativos de exemplos, ajudando principalmente na etapa inicial da especificação e desenvolvimento do simulador.

Com base nos testes realizados com o simulador, pôde-se também identificar algumas limitações do simulador, que são:

a) os modelos 3D utilizados para representar os tanques de guerra mostraram-se inadequados para a utilização em cenários como o do simulador desenvolvido, ou seja, cenários em tempo real. Isso fez com que a taxa de *frames* por segundo diminuísse consideravelmente, pois muitos detalhes precisam ser *renderizados*;
b) não foi implementado um mecanismo de nível de detalhamento (*Level Of Detail* – LOD) de acordo com a distância dos demais tanques de guerra. Esse mecanismo faria com que a taxa de *frames* por segundo aumentasse consideravelmente, pois menos detalhes precisariam ser *renderizados* quando os demais tanques estivessem a uma grande distância da câmera;
c) foram aplicadas poucas texturas ao terreno e também nenhum tipo de obstáculo estático foi implantado, tal como construções e vegetação. Esses elementos ajudariam a aumentar o nível de realidade do cenário 3D;
d) não foi implementada uma câmera que seguisse a rotação da arma principal no modelo de tanque “M1 Abrams”. Isso facilitaria a mira dos alvos por parte do jogador tipo *avatar*;
e) não foi implementado um radar visual para mostrar a localização de todos os tanques de guerra no cenário 3D. Isso contribuiria para a elaboração de estratégias por parte dos jogadores tipo *avatar*.

## 4.1 EXTENSÕES

Sugerem-se as seguintes extensões para a continuidade do trabalho:

a) desenvolver novos modelos 3D específicos para os tanques de guerra do simulador. Esses modelos poderiam conter também a animação da esteira do tanque, que ajudaria a elevar o nível de realidade no cenário. Seria ideal que os novos modelos contivessem um número de faces entre 600 e 1200, contribuindo para o aumento da taxa de FPS;

b) implementar um sistema de sonorização 3D para o cenário. Incluindo ruídos do ambiente, barulho do motor dos tanques e som do disparo de projéteis. Para prover isso, a engine JME contém uma integração com a biblioteca OpenAL, que além de ser multiplataforma, é bastante difundida na comunidade;

c) implementar efeitos de explosão quando ocorre a “morte” de um tanque. Incluir também efeitos de fumaça quando a saúde do tanque está baixa. Outra possibilidade é gerar um efeito de fogo ao disparar um projétil através das armas do tanque;

d) criar novos modelos de terrenos para a batalha. Incluindo novos pontos de início para os tanques de guerra. Podem ser inseridos também elementos estáticos, como vegetação, construções e riachos;

e) implementar ações internas genéricas para facilitar o desenvolvimento dos times de agentes, tais como: métodos de busca e métodos que encontram o melhor caminho a ser seguido por um tanque de guerra partindo de um ponto até outro (*pathfinding*). Algoritmos dessa natureza poderiam ser implementados devido à possibilidade dos agentes poderem refletir sobre a topologia estática do terreno. Dessa forma, poderiam ser implementados como ações internas do Jason;

f) disponibilizar uma forma de notificar (através crenças ou percepções) os agentes quando passam perto de obstáculos do terreno, tais como montanhas. Esse recurso ajudaria os agentes a escolher melhores caminhos a serem seguidos;

g) implementar um mecanismo para que o simulador funcione e tenha um bom desempenho também em redes WAN. Para isso, poderiam ser utilizadas as técnicas de *Dead Reckoning* e *Heartbeats* citadas na fundamentação teórica, sendo que a técnica de UDP *broadcasting* usada na implementação precisaria ser substituída.

# REFERÊNCIAS BIBLIOGRÁFICAS

ALVARES, Luiz Otavio; SICHMAN, Jaime Simão. Introdução aos sistemas multiagentes. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 16., 1997, Brasília. **Jornada de atualização em informática (JAI'97)**. Brasília: UnB, 1997. p. 1-38.

ANNOTATIONS. [S.l.], 2008. Disponível em: <http://java.sun.com/docs/books/tutorial/java/javaOO/annotations.html>. Acesso em: 4 jun. 2008.

APPIO, Alisson R. **Sistema multiagentes utilizando a linguagem AgentSpeak(L) para criar estratégias de armadilha e cooperação em um jogo tipo PacMan**. 2004. 49 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

BORDINI, Rafael H.; HÜBNER, Jomi F. et al. **Jason**: a Java-based interpreter for an extended version of AgentSpeak. [S.l.], 2008. Disponível em: <http://jason.sourceforge.net>. Acesso em: 26 maio 2008.

BORDINI, Rafael Heitor; HÜBNER, Jomi Fred; WOOLDRIDGE, Michael J. **Programming multi-agent systems in AgentSpeak using Jason**. Chichester: J. Wiley, 2007.

BORDINI, Rafael H.; VIEIRA, Renata. Linguagens de programação orientadas a agentes: uma introdução baseada em AgentSpeak(L). **Revista de Informática Teórica e Aplicada**, v. 10, p. 7-38, ago. 2003.

CECIN, Fábio; TRINTA, Fernando. **Jogos multiusuário distribuídos**. [S.l.], 2007. Disponível em: <http://www.inf.unisinos.br/~sbgames/anais/tutorials/Tutorial1.pdf>. Acesso em: 30 maio 2008.

EDUARDO, Vandeir. **Protótipo de um ambiente virtual distribuído multiusuário**. 2001. 121 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau.

FERREIRA, Alexandre Guimarães. **Uma arquitetura para a visualização distribuída de ambientes virtuais**. 1999. 83 f. Dissertação de Mestrado (Mestrado em Ciências da Computação) – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

GOSSWEILER et al. **An introductory tutorial for developing multi-user virtual environmentsnts**. [S.l.], 1994. Disponível em: <http://www.cs.virginia.edu/papers/gossweiler94introductory.pdf>. Acesso em: 26 maio 2008.

HELP ROBOCODE. [S.l.], 2005. Disponível em: <http://robocode.sourceforge.net/help>. Acesso em: 26 maio 2008.

JAVA EVERYWHERE. [S.l.], 2008. Disponível em: <http://www.sun.com/java/everywhere/index.jsp>. Acesso em: 26 maio 2008.

JAVA PLATAFORM SE 6. [S.l.], 2007. Disponível em: <http://java.sun.com/javase/6/docs/api/java/net/package-summary.html>. Acesso em: 26 maio 2008.

JENNINGS, N. R. Coordination techniques for distributed artificial. In: (EDS), O'Hare G.M.P Jennings N.R. (Ed.). **Foundations of distributed Artificial Intelligence**. [S.l.]: John Wiley & Sons, Inc, 1996.

JOAL. [S.l.], 2008. Disponível em: <https://joal.dev.java.net>. Acesso em: 30 maio 2008.

KUROSE, James F; ROSS, Keith W. **Redes de computadores e a Internet**: uma abordagem top-down. Tradução Arlete Simille Marques. 3. ed. São Paulo: Pearson Addison Wesley, 2005.

MACEDONIA, Michael R.; ZYDA, Michael J. **A taxonomy for networked virtual environments**. [S.l.], 1997. Disponível em: <http://www.cybertherapy.info/pages/taxonomy.pdf>. Acesso em: 26 maio 2008.

OPENGL OVERVIEW. [S.l.], 2008. Disponível em: <http://www.opengl.org/about/overview/>. Acesso em: 26 maio 2008.

OWEN, G. Scot. **3D rotation**. [S.l.], 1998. Disponível em: <http://www.siggraph.org/education/materials/HyperGraph/modeling/mod_tran/3drota.htm/>. Acesso em: 2 jun. 2008.

PATTON, Greeg. **Introduction to the JMonkeyEngine**. [S.l.], 2004. Disponível em: <http://fortworthjug.org:8090/fwjug/resources/topics/IntroTojME.pdf>. Acesso em: 26 maio 2008.

PINHO, Márcio S. et al. **Um modelo de interface para navegação em mundos virtuais.** Porto Alegre, 1999. Disponível em: <http://grv.inf.pucrs.br/Pagina/Publicacoes/Bike/Portugues/Bike.htm Acesso em: 26 maio 2008.

ROBOCODE: build the best – destroy the rest. [S.l.], 2007. Disponível em: <http://robocode.sourceforge.net>. Acesso em: 26 maio 2008.

ROBOCUP. [S.l.], 2008. Disponível em: < http://www.robocup.org>. Acesso em: 26 maio 2008.

RUSSELL, Stuart; NORVIG, Peter. **Inteligência artificial**. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2004.

SHAW, Chris; GREEN, Mark. **The MR toolkit peers package and experiment**. Edmonton, Alberta, 1993. Disponível em: <http://www.sfu.ca/~shaw/papers/vrais93.ps.gz>. Acesso em: 26 maio 2008.

THE JAVA TUTORIALS. [S.l.], 2008. Disponível em: <http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html>. Acesso em: 28 maio 2008.

THE REFLECTION API. [S.l.], 2008. Disponível em: <http://java.sun.com/docs/books/tutorial/reflect/index.html>. Acesso em: 3 jun. 2008.

SINGHAL, Sandeep; ZYDA, Michael. **Networked virtual environments**: design and implementation. New York: ACM Member Services, 1999.

WOOLDRIDGE, Michael. **An introduction to multiagent systems**. New York: John Wiley & Sons, 2002.

WOOLDRIDGE, Michael. Intelligent agents. In: WEISS, Gerhard (Ed.). **Multiagent systems**: a modern approach to distributed artificial intelligence. Cambridge, MA: MIT Press, 1999. p. 27–77.

## APÊNDICE A – Modelos 3D dos tanques de guerra utilizados

A Figura 37 apresenta o modelo 3D do tanque “M1 Abrams” dentro da ferramenta Blender 3D. Esse modelo foi obtido na internet[16] e poucas modificações foram realizadas para inseri-lo no simulador. A primeira modificação feita foi aplicar as texturas, que consistem em imagens de cor plana, somente para melhorar o aspecto visual dele no cenário. Outra modificação feita foi em relação ao nível de detalhes do modelo. Por não ter sido modelado especificamente para ser inserido em jogos ou simuladores, seu nível de detalhes é muito grande, contendo aproximadamente 49114 faces, tornado sua *renderização* na cena muito demorada. Para diminuir os detalhes, foi aplicado o modificador *Decimate* do Blender, onde um parâmetro numérico que varia de 1 a 10 é informado. Quanto menor o valor desse parâmetro, menos faces são geradas para o objeto correntemente selecionado, conseqüentemente diminuindo o nível de detalhamento. Para esse modelo, o número de faces após a aplicação do modificador foi aproximadamente de 24557.

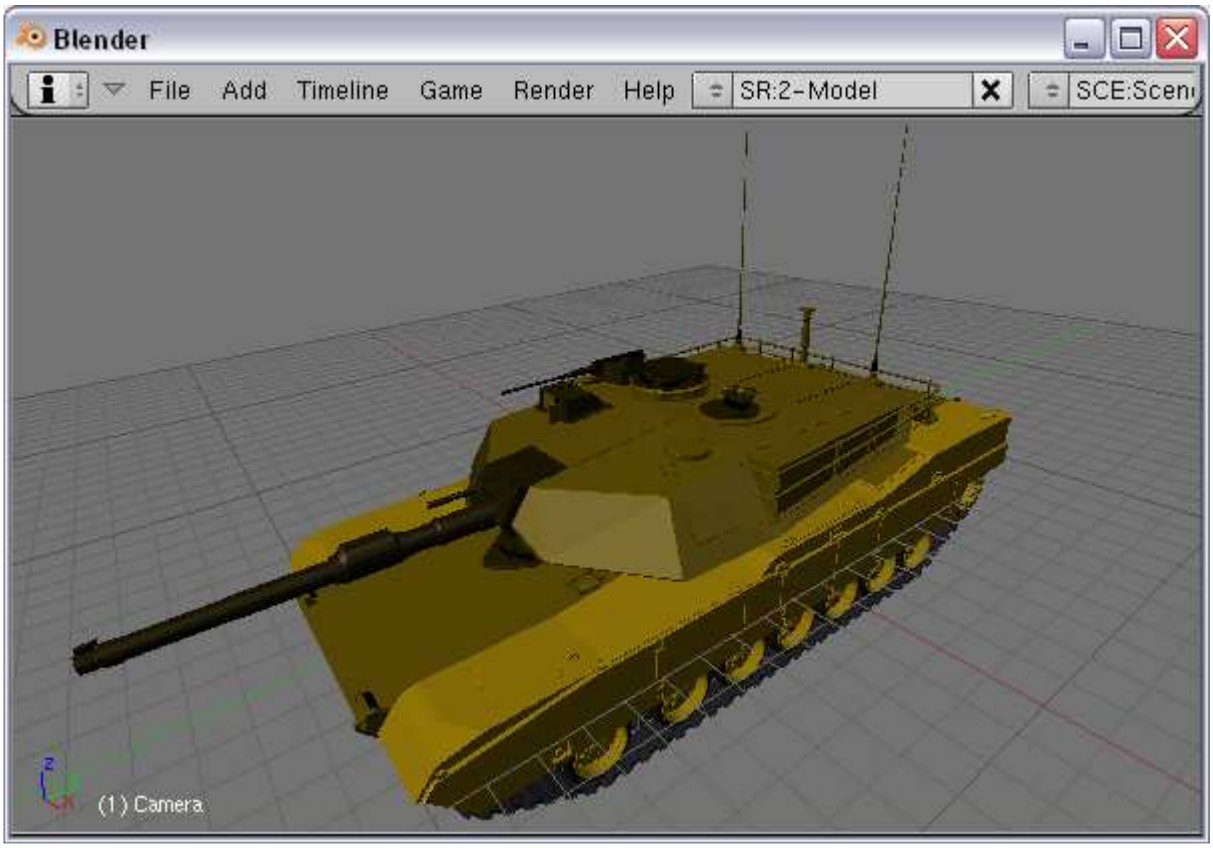



Figura 37 – Modelo 3D para o tanque “M1 Abrams”

A Figura 38 mostra o modelo 3D do tanque de guerra “Jadge Panther” também na

[16] Através do web site http://www.3dxtras.com/.

ferramenta Blender 3D. Esse modelo também foi obtido na internet (no mesmo web site) e as mesmas modificações foram realizadas para tornar o modelo mais leve com o intuito de adicioná-lo ao cenário do simulador. Antes da aplicação do modificador *Decimate*, o modelo apresentava aproximadamente 36622 faces e após o uso do modificador esse número baixou para 16646 faces.

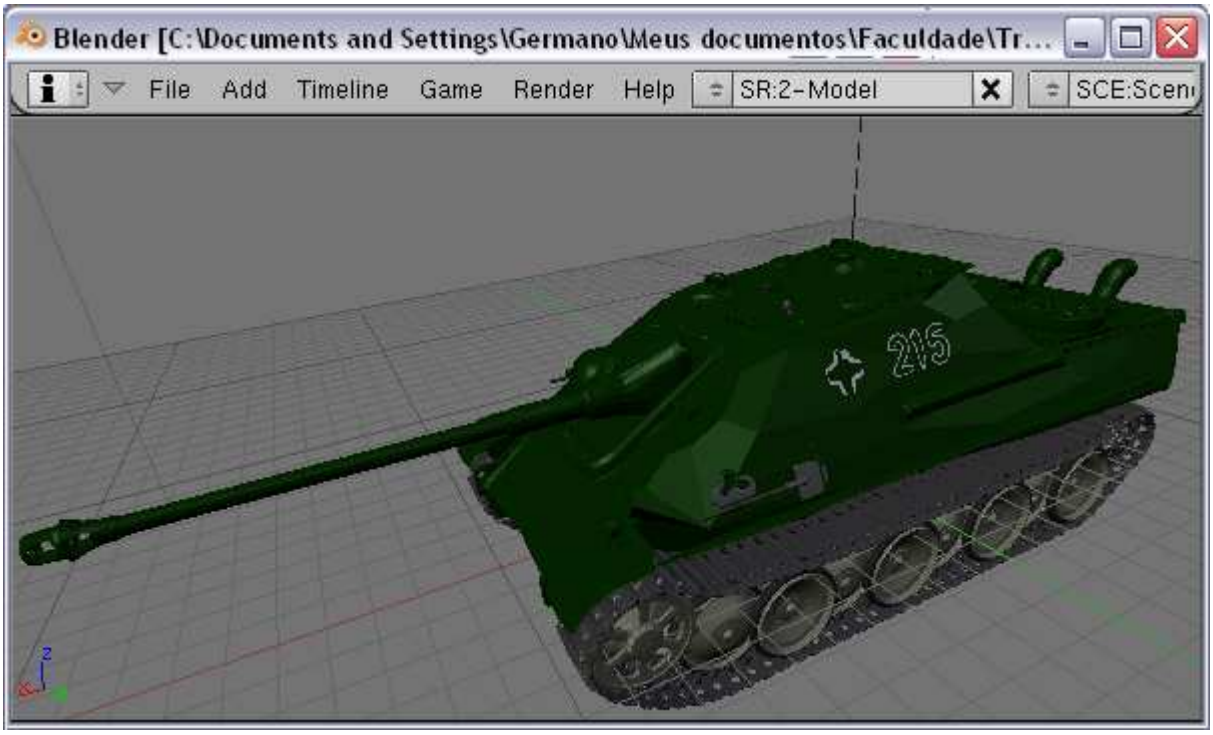



Figura 38 – Modelo 3D para o tanque “Jadge Panther”

# APÊNDICE B – Código-fonte completo do projeto SMA Jason do estudo de caso apresentado na operacionalidade da implementação

No Quadro 67 é apresentado o conteúdo do arquivo de configuração do projeto Jason chamado `baianoTeam`. Esse projeto é composto pelos agentes `boss` e `slave`, cujo código é mostrado nos Quadros 68 e 69, respectivamente.

```
MAS baianoTeam {

       // always Centralised
       infrastructure: Centralised

       // always use this environment class.
       // -debug to see important stuff.
       environment:
                br.furb.inf.tcc.tankcoders.jason.TankCodersEnvironment("-debug")

       // always use this agentArchClass
       agents:
           boss agentArchClass br.furb.inf.tcc.tankcoders.jason.TankAgArch;
           slave agentArchClass br.furb.inf.tcc.tankcoders.jason.TankAgArch;

       // to find asl source into the folder: "src/asl"
       aslSourcePath: "src/asl";
}
```

Quadro 67 – Arquivo de configuração do projeto Jason

```
/* Events */
// when battle starts.
+onStartBattle : true <- !startMyBattlePlan.

// if some bullet hits me, take cover!!!
+onHitByBullet(I) : true <- .print("I took a shot"); !takeCover.

// if some tank hits me, move ahead or back 200 mts.
+onHitTank(T, E) : movingAhead(10)
                 <- +movingBack; dropAccelerator; back(200);
                    .wait("+actionFinished"); -movingAhead(10); -movingBack.
+onHitTank(T, E) : not movingAhead(X) <- +movingAhead(0); ahead(200).
+onHitTank(T, E) : not movingBack & movingAhead(X) & X < 10
                 <- ?movingAhead(X); - movingAhead(X); .print(X);
                    +movingAhead(X+1); ahead(200);
                    .wait("+actionFinished"); -movingAhead(X+1).

+onDeath : true <- .print("I'm dead :( looking for a new boss").

/* Plans */
// my initial battle plan
+!startMyBattlePlan : true <- .print("battle has started, let's fight!");
                              ahead(870); .wait("+actionFinished");
                              turnLeft(85, 1); .wait("+actionFinished");
                              ahead(1000).

// find a "safe" place in the battle field.
+!takeCover : true <- .print("Ouchh, I'll find a safe place...");
                      back(50); .wait("+actionFinished");
                      turnLeft(45, -1); .wait("+actionFinished");
                      ahead(200).
```

Quadro 68 – Código-fonte do agente `boss`

```
/* Events */
// when battle starts.
+onStartBattle : true <- !startMyBattlePlan.

// if some bullet hits me, take cover!!!
+onHitByBullet(I) : true <- .print("I took a shot"); !takeCover.

// if some tank hits me, move ahead or back 200 mts.
+onHitTank(T, E) : movingAhead(10)
                 <- +movingBack; dropAccelerator; back(200);
                    .wait("+actionFinished"); -movingAhead(10); -movingBack.
+onHitTank(T, E) : not movingAhead(X) <- +movingAhead(0); ahead(200).
+onHitTank(T, E) : not movingBack & movingAhead(X) & X < 10
                 <- ?movingAhead(X); - movingAhead(X); .print(X);
                    +movingAhead(X+1); ahead(200);
                    .wait("+actionFinished"); -movingAhead(X+1).

+onDeath : true <- .print("I'm dead :( looking for a new boss").

/* Plans */
// my initial battle plan
+!startMyBattlePlan : true <- .print("battle has started, let's fight!");
                              ahead(910); .wait("+actionFinished");
                              turnLeft(70, 1); .wait("+actionFinished");
                              ahead(1300).

// find a "safe" place in the battle field.
+!takeCover : true <- .print("Ouchh, I'll find a safe place...");
                      back(50); .wait("+actionFinished");
                      turnLeft(45, -1); .wait("+actionFinished");
                      ahead(200).
```

Quadro 69 – Código-fonte do agente `slave`

O código de ambos os agentes é extremamente parecido. A única diferença entre eles está na implementação da meta `startMyBattlePlan`. Isso acontece, pois esses agentes não refletem diretamente sobre a topologia do terreno, fazendo com que a distância a ser percorrida pelos tanques até o campo de batalha esteja fixa no código. Outro motivo deve-se ao fato de que o simulador não possui nenhuma forma de notificar os agentes (através de crenças ou percepções) quando estão passando perto de obstáculos, como montanhas, por exemplo.

## ANEXO A – Comparação entre protocolos e formas de comunicação mais utilizadas na construção de AVDs

No Quadro 70, Singhal e Zyda (1999) apresentam uma comparação entre os protocolos e formas de comunicação mais utilizadas em AVDs.

| **PROTOCOLO** | **CARACTERÍSTICAS** | **LIMITAÇÕES** | **CARACTERÍSTICAS AVD** |
|---|---|---|---|
| TCP | Garantia de entrega de pacotes;<br>Entrega de pacotes ordenada;<br>Checagem de pacotes com *CheckSum*;<br>Controle do fluxo de transmição. | Suporta somente conectividade ponto-a-ponto;<br>*Overhead* de largura de banda;<br>Envio de pacotes podem ser adiados para garantir a ordenação. | Usado em AVDs com um número relativamente baixo de usuários;<br>Tipicamente usado em uma arquitetura cliente-servidor. |
| UDP | Transmição de dados baseada em pacotes;<br>Baixo overhead;<br>Entrega imediata. | Suporta somente conectividade ponto-a-ponto;<br>Sem garantia ou confiabilidade;<br>Sem garantia de ordenação;<br>Possível corrupção de pacotes. | Usado em AVDs com grande requerimento de dados;<br>Usado tanto em arquitetura cliente-servidor quanto em peer-to-peer. |
| IP *Broadcasting* | Igual ao UDP, porém esse permitindo entrega à múltiplos *hosts*. | Igual ao UDP, porém esse limitando-se a redes locais. | Usado em AVDs de pequena escala com alto requerimento de dados e sensível à entrega de dados. |
| IP *Multicasting* | Igual ao IP *Broadcasting*, porém esse permitindo o uso em redes WANs. | Igual ao UDP, porém esse só funcionando em redes que contenham equipamentos compatíveis com o serviço de *multicast*. | Usado em AVDs de larga escala com arquitetura peer-to-peer, particularmente sobre a internet. |

Fonte: adaptado de Singhal e Zyda (1999).

Quadro 70 – Comparação entre protocolos e formas de comunicação