Implementação de Suporte a Modelos de Personagem Não Jogador em Dispositivos Móveis na Mobile 3D Game Engine

Paulo César Rodacki Gomes
Cláudio José Estácio
FURB - Universidade Regional de Blumenau
DSC - Departamento de Sistemas e Computação

Vitor Fernando Pamplona
UFRGS - Universidade Federal do Rio Grande do Sul
PPGC - Programa de Pós-Graduação em Computação
Instituto de Informática

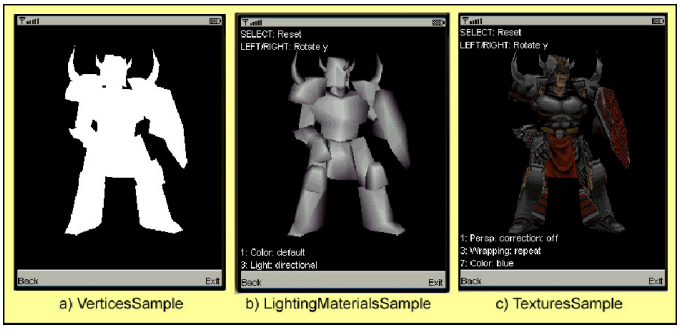

Figura 1: *Personagem Knight em 3 diferentes modos de visualização*

Resumo

Nos últimos anos, jogos 3D para celulares têm recebido significativa atenção dos pesquisadores e incentivos fiscais do estado brasileiro. Ainda longe da qualidade gráfica dos jogos para computadores e consoles, os jogos para celulares, que sofrem pela severa limitação de memória e processamento dos aparelhos modernos, caminham a passos lentos. Recentes investidas contra estas difuldades criaram motores de jogos [Gomes and Pamplona 2005], [de Albuquerque Macêdo Jr. 2005] com a finalidade de facilitar o desenvolvimento e otimizar os jogos. Este trabalho apresenta a implementação do módulo de importação e visualização de Personagens Não Jogador (PNJs) em um dos motores de jogos citados anteriormente, o *Mobile 3D Game Engine* (M3GE). Desta forma, foi adicionado o suporte a um formato de modelos chamado *Quake II's Models* (MD2), muito utilizado para a modelagem de personagens para jogos, pois incorpora recursos de animação quadro a quadro. O trabalho não discute questões relativas à inteligência dos PNJs. São apresentados detalhes da implementação de importação e visualização dos modelos, bem como a demonstração de exemplos e testes de desempenho de forma a validar a solução proposta.

Keywords:: Personagem não Jogador (PNJ), Motores de Jogos, M3G, Dispositivos Móveis, MD2

Author's Contact:

rodacki@inf.furb.br
{claudioestacio, vitorpamplona}@gmail.com

1 Introdução

Os jogos eletrônicos para dispositivos móveis são uma opção de entretenimento que vem despertando cada vez mais interesse tanto por parte dos jogadores quanto pela indústria de software brasileira. Tal categoria de jogos vem se modernizando constantemente, seguindo a evolução do hardware, notoriamente dos telefones celulares, onde dentre os recursos mais importantes, estão as recentes implementações e melhorias nas rotinas de animação 3D.

Sob uma ótica retrospectiva, em poucos anos os jogos para consoles e computadores evoluíram na simplicidade dos gráficos em duas dimensões (2D) e sons reproduzidos com algumas notas de bips, para um mundo mais realista, com visualização de ambientes 3D, com mais ação e sons com recursos interativos e 3D. Visto esse processo evolutivo dos jogos para computadores, pode-se esperar que o mesmo deva acontecer nos jogos para celulares. Atualmente a maioria dos jogos nesta plataforma são em 2D, mas já existem diversas iniciativas para desenvolvimento em 3D. Para facilitar o desenvolvimento dos jogos, utilizam-se os motores, que são bibliotecas de software responsáveis pelos ciclos de interação existentes em um jogo, as quais são capazes de tratar a maioria das necessidades existentes na sua implementação. A Mobile 3D Game Engine (M3GE) [Gomes and Pamplona 2005] é um motor de jogos para telefones celulares baseado na Java Mobile 3D Graphics API (M3G) [Nokia 2004] e já possui diversas funcionalidades implementadas, tais como, importação e renderização de ambientes 3D, criação de câmeras, tratamento de eventos, movimentação de personagens e câmeras pelo cenario e tratamento de colisão. Cabe ressaltar que a M3GE é a primeira implementação livre de motor de jogos 3D em Java que se tem notícia.

À medida que os jogos em dispositivos móveis vão se tornando mais complexos e sofisticados, o nível de realismo gráfico deverá levar os jogadores a acreditar que os jogos ocorrem em mundos muito mais realistas. Um componente importante desse crescente nível de realismo são os Personagens Não Jogador (PNJs). Os PNJs são os personagens presentes no jogo e que não são controlados pelo jogador, mas que se relacionam de alguma forma com ele. Por exemplo, num jogo de primeira pessoa, os PNJs são os personagens inimigos do jogador, que devem ser acertados por seus disparos.

Este artigo apresenta detalhes da implementação de uma extensão da M3GE que consiste na integração de Personagem Não Jogador (PNJ). Tal integração é feita com recursos de importação de modelos sofisticados, incorporação dos mesmos ao grafo de cena disponibilizado pela M3G, e, naturalmente, a renderização destes modelos na cena. A implementação considera a importação de modelos de personagens animados no formato de arquivo *Quake II's Models* (MD2) [Henry 2004]. O MD2 é um dos formatos mais populares para inclusão de PNJs em jogos 3D. O presente trabalho não aborda questões de suporte a inteligência artificial para os PNJs.

O artigo está estruturado da seguinte forma: primeiramente, a seção de introdução contextualiza o problema, e as duas próximas seções apresentam uma breve revisão das principais tecnologias para desenvolvimento de jogos para telefones celulares, incluindo informações gerais a respeito de motores de jogos. A seguir, são comentados alguns trabalhos correlatos, detalhando-se as principais características do motor de jogos *Mobile 3D Game Engine* (M3GE). Após, é apresentado o conceito de Personagem Não Jogador e detalhes dos modelos MD2. Finalmente, é discutida a implementação de suporte à importação de modelos MD2 na M3GE, sendo posteriormente apresentados os resultados e conclusões do artigo.

2 Jogos em Dispositivos Móveis

O avanço da tecnologia dos celulares nos últimos anos fez com que esses aparelhos deixassem de ser apenas usados para transmissão e recepção de voz e passassem a ter diversas utilidades. O desenvolvimento de jogos para tais dispositivos vem ganhando cada vez mais importância, e as soluções mais utilizadas pela comunidade de desenvolvedores são a Sun J2ME, a Qualcomm BREW e a Mophun [Paiva 2003].

A Sun Java 2 Micro Edition (J2ME) é uma versão da linguagem Java para dispositivos móveis, sendo a primeira iniciativa nesse segmento em 2000. Para sua aplicação rodar em um celular, basta que o celular possua uma máquina virtual Java, como a Kilobyte Virtual Machine (KVM) [Müller et al. 2005].

Em janeiro de 2001 a Qualcomm lançou o Binary Runtime Enviroment for Wireless (BREW). Para este ambiente as aplicações são desenvolvidas em C/C++, e depois de compiladas rodam em um chip separado do processador principal do dispositivo. Esta arquitetura apresenta características semelhantes a J2ME por apresentar suporte a APIs OpenGL ES [Khronos Group 2004]. Ainda existe a Application Program Interface (API) desenvolvida pela própria empresa, a QX Engine, que tem recursos que facilitam o desenvolvimento de jogos. Uma característica interessante apresentada pelo BREW é que para uma aplicação ser comercializável, ela deve estar cadastrada na empresa e passar por rigorosos testes, para evitar que haja erros em tempo de execução. Isto viabiliza uma certificação de qualidade e originalidade à aplicação [Müller et al. 2005].

O padrão Mophun foi desenvolvido pela sueca Synergenix. É menos usado em relação ao Java e ao BREW, mas tem como característica deixar os jogos menores, os quais ficam entre metade e um terço do tamanho das outras duas soluções [Paiva 2003].

3 Motores de Jogos 3D

Um motor de jogo é o coração de qualquer jogo eletrônico. Inicialmente os motores eram desenvolvidos para fazer parte do jogo e serem usados somente para este fim específico. Visto que diferentes jogos têm similaridades em seu processo de desenvolvimento, os motores passaram a implementar funcionalidades comuns a determinados tipos de jogos. Assim seu código fonte é reutilizado, isto é, funções que foram usadas num determinado jogo, podem ser usadas em outros que apresentam características semelhantes [Harrison 2003].

Conforme definido por Macêdo Júnior [de Albuquerque Macêdo Jr. 2005], um motor de jogos 3D é um sistema que produz uma visualização em tempo real respondendo aos eventos da aplicação. Para o funcionamento desse processo deve-se utilizar várias técnicas, algoritmos, estruturas de dados e conhecimentos matemáticos. Estes aspectos ficam transparentes quando se desenvolve um jogo utilizando motor de jogos 3D pois o desenvolvedor fica abstraído dessas funcionalidades, preocupando-se apenas com a lógica e o enredo do jogo.

O motor disponibiliza as principais características para o desenvolvimento de jogos, como por exemplo: renderização de cenários 3D, recursos de comunicação em rede, tratamento de eventos, mapeamento de texturas, *scripting*, inteligência artificial, entre outros [Finney 2004]. O principal foco dos motores 3D está na qualidade e no desempenho computacional da apresentação visual (*rendering*) do jogo. Como os ambientes são tridimensionais, diversos cálculos são usados para renderizar cada quadro da visualização, juntando-se os vértices com as texturas e aplicando efeitos. Alguns exemplos de motores de jogos 3D são: Crystal Space [Crystal Space 2004], Ogre 3D [Ogre3D 2005], Genesis3D [Eclipse Entertainment 2004], 3D Game Studio [Conitec Corporation 2004] e Fly3D [Paralelo Computação 2004].

A figura 2 apresenta a arquitetura de um motor de jogos 3D. O gerenciador principal é o centro dos demais níveis, sendo o responsável por interligar cada funcionalidade do motor de jogos. O gerenciador de entrada organiza e repassa as informações vindas do usuário. O gerenciador gráfico apresenta a cena atual na saída do dispositivo, neste caso a tela. O gerenciador de inteligência artificial gerencia os personagens não jogadores, definindo seus movimentos. O gerenciador de múltiplos jogadores gerencia a comunicação com os demais jogadores. O gerenciador de mundo armazena o grafo de cena do jogo onde estão interligados todos os objetos. O gerenciador de objetos tem a finalidade de inserir, alterar e excluir os objetos do jogo. O objeto do jogo armazena as informações de cada entidade do jogo. O gerenciador de som controla os efeitos sonoros provenientes dos objetos, enviando para a saída, que neste caso são os alto-falantes do dispositivo. E por último o editor de cenário é um nível a parte do restante que tem por finalidade criar a estrutura inicial da visualização dos jogos, composta pelos chamados "mapas". Também pode ser realizada no editor de cenário a criação dos personagens 3D [Gomes and Pamplona 2005].

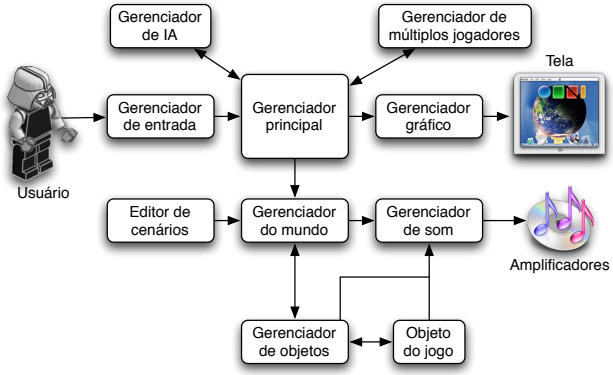

Figura 2: *Arquitetura de um motor de jogos*

4 Trabalhos Correlatos

O desenvolvimento de motores de jogos para dispositivos móveis é uma área ainda recente, mas podemos citar três projetos acadêmicos relacionados ao presente trabalho: o wGEN, o mOGE e a M3GE.

O wGEN é um *framework* de desenvolvimento de jogos 2D para dispositivos móveis [Pessoa 2001]. Ele é o pioneiro na utilização de J2ME num motor de jogos, porém não tem recursos 3D. O wGEN apresenta como componentes e funcionalidades: representação do objeto do jogo, representação do mapa do jogo, gerenciamento de objetos, gerenciamento de entrada, gerenciamento gráfico, gerenciamento do jogo, gerenciamento de rede, gerenciamento de aplicação e integração com o editor de cenários.

Outro motor de jogos com características semelhantes com este trabalho é o Mobile Graphics Engine (mOGE) [de Albuquerque Macêdo Jr. 2005]. Este projeto teve como objetivo prover diversas técnicas de computação gráfica 3D. Para a renderização, o mOGE utiliza o *pipeline* gráfico, que é a decomposição do processo de transformação de objetos 3D em imagens, composto pelos estágios de aplicação, de geometria e de rasterização. Para a representação de objetos foi utilizada malha de triângulos. Para a estrutura dos objetos presentes no jogo é utilizado um grafo de cena. Da mesma forma que na M3GE, a utilização de um grafo de cena facilitou o processo de animação dos objetos. Este motor de jogos apresenta ainda projeção de câmeras, detecção de colisão e efeitos especiais baseados em imagens [de Albuquerque Macêdo Jr. 2005].

No âmbito de jogos comerciais, o motor 3D mais conhecido é o Edgelib [Elements Interactive 2007]. Desenvolvido em C++ para sistema operacional Symbian pela empresa Elements Interactive, este motor já é bastante usado por uma grande quantidade de desenvolvedores. Possui recursos para gráficos 2D e 3D, com suporte para OpenGL ES, além de recursos para desenvolvimento de jogos em rede, áudio, gerenciamento de memória e de arquivos, entre outros.

Recentemente foi anunciado pela empresa Fishlabs o motor de jogos Abyss 2.0 [FISHLABS Entertainment GmbH 2007]. Compatível com celulares que rodem a JSR-184 [Nokia 2003], este motor apresenta recursos de renderização 3D, juntamente com um motor de Inteligência Artificial e Física de corpos Rígidos.

Nenhum dos motores citados possuem funcionalidades específicas

para a importação de PNJs, porém o grafo de cena do mOGE é um dos recursos úteis para uma eventual implementação de inserção de PNJs nos jogos, de forma que o motor possa desenhá-los na cena.

5 Mobile 3D Game Engine

A Mobile 3D Game Engine (M3GE) é um protótipo de motor de jogos para dispositivos móveis com suporte a M3G. Ela apresenta diversas funcionalidades implementadas, tais como: importação e renderização de ambientes 3D, criação de câmeras, tratamento de eventos, movimentação de personagens no cenário e tratamento de colisão. Na página do projeto (`https://m3ge.dev.java.net/`), é disponibilizado um exemplo de um jogo com as funcionalidades implementadas, onde um cenário 3D com objetos estáticos pode ser visualizado. A figura 3 demonstra duas telas desta aplicação exemplo. O jogador, representado por um cubo, é controlado pelos comandos do teclado do celular. A movimentação do jogador está sujeita ao tratamento de colisões e possui capacidade de disparar tiros nos demais objetos da cena. A implementação da M3GE é feita sobre a M3G que por sua vez é implementada através da J2ME. A M3G é uma API Gráfica 3D para ser utilizada em dispositivos móveis com suporte a programação Java. Esta foi criada em função da maioria dos jogos para esse tipo de dispositivo ser em 2D e haver uma demanda latente por jogos 3D em celulares [Höfele 2005].

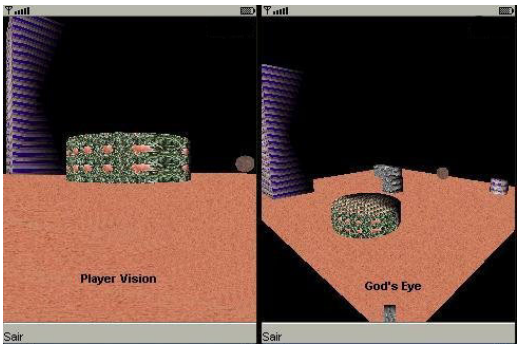

Figura 3: *Aplicação exemplo da M3GE*

O motor M3GE foi projetado anexo à M3G, significando que se pode acessar diretamente a M3G na aplicação, se necessário. Isto dá maior flexibilidade na implementação dos jogos. O motor foi dividido em dois grandes componentes: o responsável pela leitura dos arquivos de entrada e o *core*, além de um terceiro componente de classes utilitárias. A figura 4 ilustra a arquitetura da M3GE, seus principais componentes e classes. Para se montar o cenário do jogo são lidos três tipos de arquivos: um com as configurações gerais do ambiente, um com as malhas de polígonos e outro com uma série de texturas. As classes ObjLoader e MbjLoader são responsáveis por gerenciar os objetos, elas criam os objetos da classe Object3D e os nós do grafo de cena. A classe Object3D é a classe base de todos os objetos 3D visíveis da M3G. Ela inclui o nodo pai ou qualquer outro nodo no grafo de cena, uma animação, uma textura, entre outros. Em cada nodo do grafo estão presentes um ou mais grupos, que por sua vez são interligados aos objetos 3D da classe Object3D. Estes objetos podem ser modelos gráficos 3D, câmeras, luzes, imagens 2D, entre outros.

O componente *core*, na figura4 é o motor de jogos propriamente dito, que tem como gerenciador de entrada a classe KeysManager. A classe Player implementa o personagem principal do jogo. A classe EngineCanvas é o gerenciador principal. Para criar as câmeras existe a classe Cameras. Para fazer o tratamento de colisão trabalham juntas as classes Configuration e CollisionDetection. A classe UpdateListener deve ser implementada pelo desenvolvedor do jogo para tratar funcionalidades na renderização, ou da lógica através da adição de eventos. E a classe UpdatesceneTimertask implementa rotinas para a chamada de renderização num determinado intervalo de tempo.

No pacote de carga de arquivos, a classe Loader é a interface para duas classes, a ObjLoader e a MbjLoader. A M3G utiliza arqui-

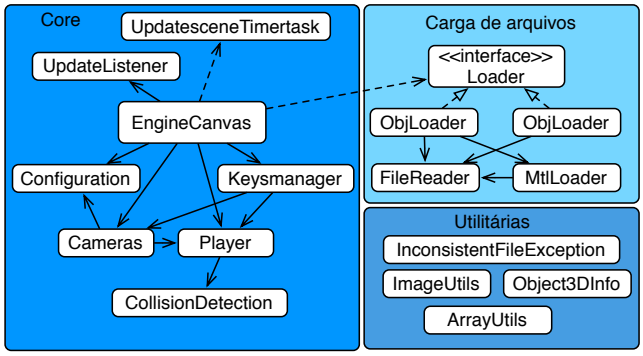

Figura 4: *Arquitetura da M3GE*

vos no formato Wavefront (obj) para importação de modelos 3D, que são carregados pela classe ObjLoader. Experiências anteriores constataram que o processo de importação de modelos em telefones celulares é demasiadamente lento, pois cada arquivo (de geometria e de materiais) precisa ser lido até três vezes. Por isso, a M3GE utiliza um formato próprio, o Mobile Object File (mbj), no qual uma série de informações sobre o modelo já estão pré-processadas, agilizando a sua importação. A classe MtlLoader é utilizada para a carga deste tipo de arquivos, que descrevem cores, propriedades de materiais e texturas do ambiente ou de grupos de objetos 3D. Os arquivos mtl podem ser utilizados juntamente com arquivos Wavefront, permitindo que modelos em ambos os formatos sejam carregados em um mesmo jogo. A classe FileReader é uma classe especial para facilitar a leitura dos arquivos.

A M3GE ainda conta com um conjunto de classes utilitárias. A classe InconsistentFileException lança uma exceção quando ocorre algum erro na leitura de arquivos. A classe ImageUtils é utilizada para fazer a leitura de um arquivo de imagem utilizado como textura. A classe Object3DInfo é utilizada para armazenar o ponto central de um objeto 3D e seu respectivo nome. Por último, a classe ArrayUtils é utilizada para chamar métodos para tratamento de listas.

As conclusões relatadas por Gomes e Pamplona [Gomes and Pamplona 2005] afirmaram que Java é promissora para a implementação de jogos para dispositivos móveis. Apesar da M3GE ser apenas um protótipo, ela já vem sendo utilizada para desenvolvimento de jogos comercialmente [Guiliano 2005]. Além disso, recentemente foi implementado um módulo para jogos multiusuário com comunicação via *bluetooth* [Carandina and Martins 2006].

6 Personagem Não Jogador

O termo Personagem Não Jogador (PNJ) vem do inglês *Non-Player Character* (NPC), e define-se por um personagem presente no jogo, que não é controlado pelo jogador, mas que se relaciona de alguma forma com ele. Os PNJs podem ser elementos decorativos das cenas ou interagir diretamente com as ações do jogador [Sánchez and Dalmau 2004]. Para jogos onde o usuário joga contra o computador, usam-se os PNJs para representar oponentes. Por exemplo, num jogo de primeira pessoa existem na cena personagens que são inimigos do jogador e tem-se como objetivo acertar tiros neles. Estes personagens podem ter diversas aparências, isto depende dos modelos que foram desenhados e também de suas texturas. Deste modo, tem-se no jogo a possibilidade de interação com diversos inimigos, alguns mais parecidos com humanos, outros com figuras mitológicas, mutantes, robôs, entre outros. Estes personagens são criados com relação ao enredo do jogo.

O primeiro jogo a incluir um oponente animado por computador foi *Shark Jaws* da empresa Atari em 1974. Nesse jogo, o PNJ é um tubarão e tem como ação perseguir o mergulhador controlado pelo jogador. Para isso, foi usada Inteligência Artificial (IA) de uma forma simplificada, se comparada com o que existe nos jogos atuais [Byl 2004].

7 Modelo Animado MD2

O formato de arquivo de modelo MD2 é um dos formatos mais populares para inclusão de PNJs em jogos 3D [Henry 2004]. Surgiu em novembro de 1997 para ser utilizado no jogo *Quake II* produzido pela ID Software. Suas principais características são: ser um modelo geométrico formado por triângulos, utilizar animação quadro a quadro, isto é, cada quadro representa uma pose do personagem, e quando as poses são exibidas em seqüência dá-se a visualização do movimento. Um outra característica importante é que sua estrutura de dados já foi concebida contendo primitivas gráficas do OpenGL para desenho do modelo, onde os vértices já estão organizados para facilitar a chamada de desenho de primitivas do tipo Gl_TRIANGLE_FAN e Gl_TRIANGLE_STRIP. Estas primitivas têm utilização opcional, pois além da lista de vértices, o arquivo possui uma lista dos triângulos utilizados na modelagem do personagem.

Este formato é uma variação dos arquivos MDL que são usados em jogos como *Half Life* e no *Counter Strike*, que utilizam uma versão derivada do motor *Quake II* [Santee 2005]. A extensão do arquivo do modelo é MD2 e ele está em formato binário dividido em duas partes: cabeçalho e dados, comentadas a seguir.

7.1 Cabeçalho

Conforme descrito por Santee [Santee 2005], é no cabeçalho do arquivo MD2 que estão definidos os números de vértices, faces, quadros chave e tudo mais que está dentro do arquivo. Pelo fato de ser um arquivo binário, o cabeçalho torna-se muito importante para a localização do restante dos dados. Cada informação presente no cabeçalho é armazenada em quatro *bytes* representados por um tipo de dado inteiro.

Antes de apresentar o conteúdo do cabeçalho é preciso rever alguns conceitos:

- Quadro-chave: a animação do modelo é representada por uma seqüência de quadros-chave, e cada um representando uma posição do personagem, devendo ser realizada interpolação entre os quadros-have. Para armazenar um quadro-chave, o modelo guarda uma lista de vértices, o nome do quadro, um vetor de escala e um vetor de translação. Este dois últimos são usados para a descompactação dos vértices, multiplicando cada coordenada do vértice pelo vetor de escala e somando ao vetor de translação. Isto é feito para transformar os vértices de ponto fixo para ponto flutuante;
- Coordenadas de Textura: as coordenadas de textura são utilizadas para mapear a textura no modelo 3D. Isto é feito através das coordenadas s e t. Cada vértice do modelo 3D recebe um par de coordenadas de textura s e t;
- *Skin*: são texturas que dão representação visual externa ao modelo. Através dos *skins* podem-se distinguir partes do corpo, roupa, tecido, materiais anexados, entre outros. Esses *skins* são mapeados para o modelo através das coordenadas de textura. No cabeçalho encontram-se o tamanho e a quantidade de *skins* que podem ser utilizados no modelo;
- Comando OpenGl: os comandos OpenGl presentes no arquivo são usados para a renderização através das primitivas GL_TRIANGLE_FAN e GL_TRIANGLE_STRIP. O arquivo MD2 apresenta uma lista de números inteiros, que estão na seguinte estrutura: o primeiro valor é o número de vértices a ser renderizado com a primitiva GL_TRIANGLE_FAN caso seja positivo ou renderizado com a primitiva GL_TRIANGLE_STRIP caso seja negativo. Os seguintes números vêm na seqüência divididos em grupos de três, onde os dois primeiros representam as coordenadas de textura s e t, e o terceiro o índice para o vértice. Repete-se a estrutura acima até que o número de vértices a desenhar seja zero significando o fim da renderização do modelo;
- Triângulos: são os que definem as faces do modelo. Eles são uma lista de índices da lista de vértices. Cada conjunto de três índices representa um triângulo;
- Vértices: são os pontos no mundo cartesiano que desenham o modelo 3D. São representados pelas coordenadas: x, y e z;
- Vetor Normal: é um vetor que está contido na estrutura dos vértices, utilizado para calcular a iluminação do modelo.

A estrutura completa do cabeçalho pode ser vista na tabela 1. Esta estrutura representa a mesma seqüência presente dentro do arquivo MD2.

Tabela 1: *Cabeçalho dos arquivos MD2*

Conteúdo	Descrição
magic	ID do formato (deve ser igual a 844121161)
version	Versão do arquivo (para Quake2 precisa ser igual a 8)
skinWidth	Largura do skin do modelo (*pixels*)
skinHeight	Altura do skin (em *pixels*)
frameSize	Tamanho das informações dos frames (em *bytes*)
numSkins	Número de skins avaliados
numVertices	Número de vértices
numTexCoords	Número de coord. de textura
numTriangles	Número de triângulos
numGlCommands	Número de comandos OpenGL
numFrames	Número de quadros para animação
offsetSkins	Posição dos skins
offsetTexCoords	Posição das coordenadas de textura no *buffer*
offsetTriangles	Posição dos triângulos no *buffer*
offsetFrames	Posição dos quadros
offsetGlCommands	Posição dos comandos OpenGL
offsetEnd	Posição para o final do modelo

7.2 Dados

As informações de cada vértice do modelo são armazenadas em poucos *bytes*. O resultado disso é um arquivo pequeno, mas com perdas na exatidão da geometria do modelo [MD2 2005]. Dispositivos móveis têm espaço de memória restrita e a resolução dos *displays* nesses dispositivos é bem menor do que em um monitor de um computador do tipo PC ou de uma tela de aparelho de TV. Por isso a perda de qualidade na visualização dos modelos não é tão perceptível quanto seria em um computador *desktop*. Os tipos de dados são apresentados da seguinte forma:

- Vetor: composto por três coordenadas do tipo *float*;
- Informação Textura: lista de nomes de texturas associadas ao modelo;
- Coordenadas da Textura: são armazenadas na estrutura como *short integers*;
- Triângulos: cada triângulo possui uma lista com os índices dos vértices e uma lista com os índices das coordenadas da textura;
- Vértices: são compostos de coordenadas 3D, onde são armazenados em um *byte* cada coordenada e um índice do vetor normal;
- Quadros: têm informações específicas da lista de vértice do quadro;
- Comandos OpenGL: são armazenados em uma lista de *integers*.

7.3 Texturas

As texturas utilizadas nos modelos MD2 são localizadas em arquivos separados. Todos os vértices do modelo devem ser mapeados para essa mesma textura. Por este motivo a textura deve ter diferentes regiões para representar cada parte do personagem, desde os pés até a cabeça, chamados assim de *skins* [Misfit Code 2005]. Na

figura 5 pode-se visualizar um arquivo de textura e ao lado o modelo utilizando-se dessa textura. O mesmo personagem pode utilizar diversos *skins* diferentes para criar novas visualizações. Pode-se comparar a troca de *skin* do personagem à troca de roupa de uma pessoa.

Figura 5: *Exemplo de skin*

O nome da textura é armazenado em 64 *bytes* do tipo *string*. Os *skins* são normalmente usados no formato de arquivo PCX, porém pode-se encontrar em outros diversos formatos, tais como PNG, BMP, GIF, entre outros.

8 Implementação de PNJs na M3GE

A especificação do módulo para PNJs foi feita com utilização dos diagramas de caso de uso, de atividades, de classes e de seqüência da UML. A figura 6 ilustra os dois casos de uso implementados nesta primeira versão, representando as funcionalidades que o jogo pode realizar a partir da M3GE utilizando a importação de modelos MD2. O ator representado pelo jogo possui dois casos de uso. O primeiro tem como função fazer a leitura e importação do arquivo MD2. Após os dados serem lidos, o segundo caso de uso demonstra que o personagem 3D deve ser exibido na tela do jogo.

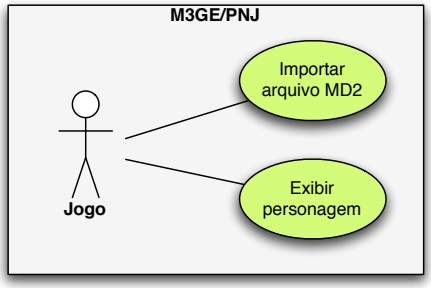

Figura 6: *Diagrama de casos de uso*

Para inserção do modelo 3D no grafo de cena, é necessário o uso de uma estrutura própria. Esta estrutura é apresentada como sendo o nodo *Mesh*, que representa um objeto 3D e armazena diversas características necessárias para sua manipulação através da API gráfica M3G. A estrutura armazena posição dos vértices, índices dos triângulos, textura, etc. A Figura 7 demonstra um diagrama de atividades que representa a criação de um nodo do tipo *Mesh*.

Devido ao fato de a aplicação (jogo) poder acessar tanto a M3GE quanto a M3G, optou-se por implementar o carregador de arquivo MD2 em um pacote de classes separado dessas arquiteturas. A M3G é utilizada para implementação da parte de visualização do modelo, porém o módulo é agregado ao motor de jogos M3GE. Para melhor distribuição das classes foram criados dois pacotes, o `md2loader` e o `datatypes`. O diagrama de classes dos dois pacotes é apresentado na figura 12.

A classe Md2Model é a responsável por montar um nó que será acrescentado ao grafo de cena. Este nó contém todas as especificações necessárias para se desenhar o modelo existente. Pelo fato do arquivo MD2 ser dividido em duas partes, cabeçalho e

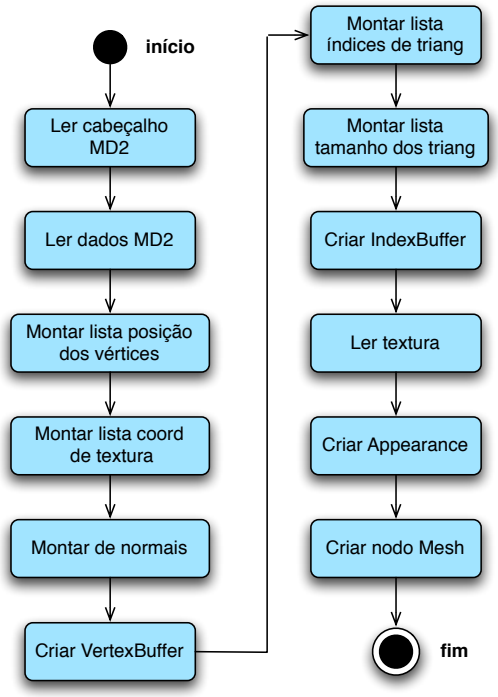

Figura 7: *Diagrama de atividades da criação do nodo Mesh*

dados, duas classes foram criadas, uma para cada parte do arquivo, sendo elas, a classe Md2Header responsável pela leitura e armazenamento do cabeçalho do arquivo, e a md2Data responsável pela leitura e armazenamento dos dados do arquivo. Para os vetores normais do modelo, existe uma lista pré calculada, que está presente na classe Md2Normal. Ainda para a leitura dos dados dos arquivos teve-se que utilizar uma classe para o tratamento dos tipos de dados, chamada MD2LittleEndianDataInputStream. Esta classe faz parte do projeto Xith3D [Lehmann 2004]. Este projeto é um pacote para tratar cenas gráficas e fazer sua renderização. Ele é todo escrito em Java e foca a criação de jogos.

As classes pertencentes ao pacote `datatypes` são utilizadas para armazenar em objetos todo conteúdo proveniente da leitura dos dados. Desta forma pode-se trabalhar com uma estrutura de dados própria para o arquivo MD2. A classe Md2Vect3 é responsável pelo armazenamento de um vetor no espaço cartesiano tridimensional com as três coordenadas: x, y e z. A classe Md2Skin é responsável pelo armazenamento do nome da textura do modelo. A classe Md2TextCoord é responsável pelo armazenamento das coordenadas da textura através dos atributos s e t. A classe Md2Triangle é responsável pelo armazenamento de um triângulo através de três índices dos vértices e três índices das coordenadas de textura. A classe Md2Vertex é responsável pelo armazenamento de um vértice através de três coordenadas e um índice para o vetor normal. A classe Md2Frame é responsável pelo armazenamento das informações dos quadros de animação do modelo, compostos por um vetor com o fator de escala, um vetor de translação, o nome do quadro e uma lista dos vértices. A classe Md2GlCmd é responsável pelo armazenamento de uma estrutura para ser utilizada com primitivas OpenGL quando renderizadas. Esta estrutura guarda apenas as coordenadas de textura s e t, e um índice para o vértice.

A figura 8 mostra a estrutura da M3GE após a agregação do módulo para suporte a PNJs com arquivos MD2. Como a implementação é feita em um pacote separado dos três módulos básicos da M3GE, única alteração feita no antigo código fonte da M3GE foi a adição do método `addNPC()` à classe EngineCanvas. Através deste método é possível adicionar um modelo MD2 ao grafo de cena.

Para a leitura de um arquivo binário optou-se por utilizar a classe DataInputStream que implementa a interface DataInput da J2ME pelo fato dela já disponibilizar diversos métodos de retorno nos tipos de dados Java. Porém, esta classe acaba invertendo a seqüência de *bytes*. Para contornar este problema é utilizada a classe

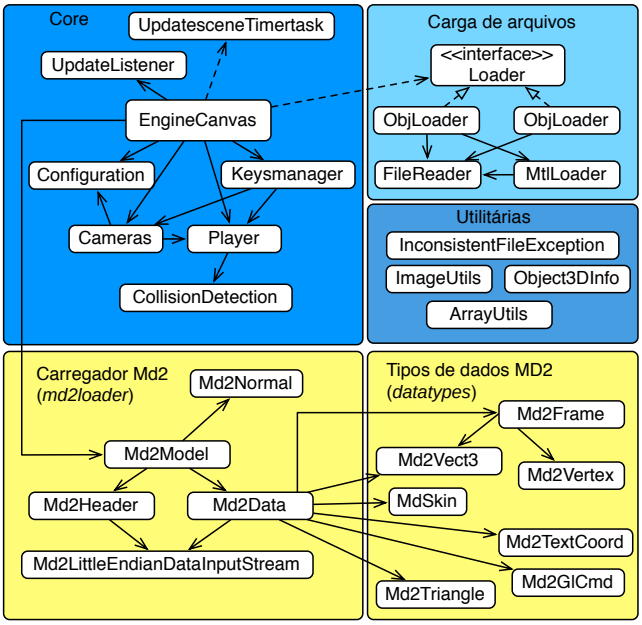

Figura 8: *Carregador MD2 inserido na M3GE*

MD2LittleEndianDataInputStream que sobrescreve os métodos da classe DataInputStream fazendo a mudança na seqüência dos *bytes* quando há dois ou mais *bytes* para serem lidos. Um exemplo desta conversão pode ser analisado no código abaixo, onde é feita a leitura de quatro *bytes*. Após isso, sua seqüência é invertida e retorna-se o valor no tipo de dado primitivo de Java, o int, que representa um número inteiro.

```
public int readInt() throws IOException {
    int[] res=new int[4];
    for(int i=3;i>=0;i--)
        res[i]=din.read();
    return ((res[0] & 0xff) << 24) |
           ((res[1] & 0xff) << 16) |
           ((res[2] & 0xff) << 8) |
           (res[3] & 0xff);
}
```

O gerenciador gráfico da M3GE já implementa funções capazes de desenhar os objetos em cena. Então para o PNJ ser desenhado, basta anexá-lo ao grafo de cena. Porém, para isto ocorrer o modelo deve ter uma estrutura compatível o nodo *Mesh* pertencente à biblioteca M3G. Este nodo define um objeto 3D através de seus triângulos juntamente com suas características conforme demonstrado na figura 9.

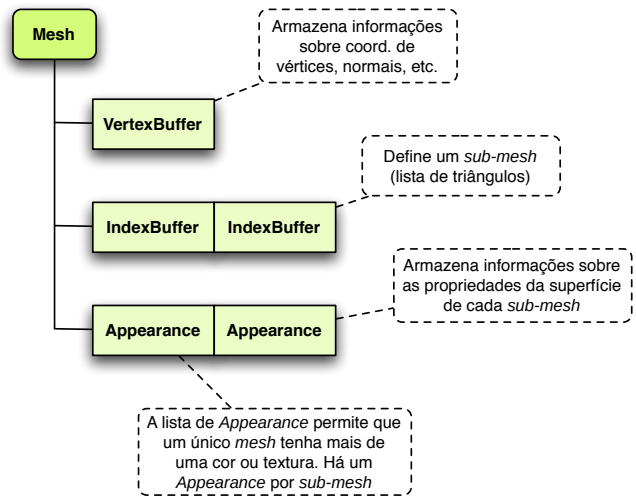

Figura 9: *Estrutura do nodo Mesh*

O nodo *Mesh* tem em sua estrutura o VertexBuffer que armazena informações relativas a posição dos vetores, a lista de normais, a lista de cores e as coordenadas da textura. Ainda possui uma lista de IndexBuffer que define os triângulos. Outra parte presente na estrutura é a lista de Appearance que é composta pelas informações de luzes, pelo modo de desenho do polígono, o modo de composição dos *pixels*, o modo de visualização baseado na distância da câmera e por último uma lista de texturas.

A classe Md2Model tem como função fazer as chamadas de leitura do arquivo MD2 e, a partir dos dados recebidos, montar uma estrutura completa de um nodo Mesh capaz de ser desenhado pela M3GE. Isto ocorre a partir do construtor da própria classe que recebe como parâmetros o local e nome do arquivo MD2, o local e nome do arquivo de textura e o índice do quadro a ser visualizado. Os métodos de leitura de arquivo MD2 desta classe convertem e preparam os dados de acordo com a estrutura do nodo *Mesh*, que é então armazenado no atributo model do objeto instanciado pela classe Md2Model. Em seguida ele é adicionado ao grafo de cena através do método `addNPC()` da classe EngineCanvas. Este método foi projetado para receber quatro parâmetros: local e nome do arquivo MD2, local e nome do arquivo de textura, número do quadro a ser visualizado e o nome do modelo. O primeiro passo deste método é criar um nodo Mesh através da criação de um objeto da classe Md2Model responsável pela leitura e estruturação do modelo. Em seguida é utilizada a classe Object3DInfo, que já estava implementada na M3GE, para inserir um nome ao modelo. Este nome terá utilidade para o dispositivo de tiros implementado na M3GE, podendo assim distinguir qual objeto foi atingido. Após isso, basta inserir o nodo *Mesh* a um grupo e depois incluí-lo ao nodo especial *World* que armazena todo o grafo de cena do jogo. O nodo *World* é um atributo da classe EngineCanvas.

9 Resultados

Para desenvolvimento de todo o projeto, incluindo os exemplos, foi utilizada a IDE Eclipse, com J2ME, Sun Java Wireless Toolkit, as APIs M3GE e M3G, e o Eclipse ME. Durante a implementação, vários testes foram realizados, inclusive avaliando resultados em diversos tipos de visualização. A figura 1, por exemplo, ilustra a visualização apenas com coordenadas de vértices e triângulos, com vetores normais, e com texturas, respectivamente. Nesta terceira visualização pode-se ver melhor os detalhes do personagem através do *skin* escolhido para ser utilizado como textura, cujo arquivo está no formato PNG.

Figura 10: *Jogo exemplo com importação de PNJs*

Para finalizar os testes e verificar a viabilidade do projeto foi implementado um protótipo de jogo, ilustrado na figura 10. Este protótipo é baseado no antigo jogo exemplo da M3GE, porém nesta versão foram adicionados três personagens a cena: o Knight

(arquivos knight.md2 e knight.png), o Centaur (arquivos centaur.md2 e centaur.png) e o Knight Evil (arquivos knight.md2 e knight_evil.png). Os personagens Knight e Knight Evil são carregados a partir do mesmo arquivo MD2 porém são utilizados skins diferentes. Toda funcionalidade que já estava presente no antigo jogo exemplo continuou funcionando durante e após a inclusão dos personagens. Até mesmo o tratamento de colisão que já estava implementado pela M3GE funcionou perfeitamente nos personagens adicionados. Pode-se inclusive utilizar a funcionalidade de tiro presente no jogo, observando as mensagens de texto exibidas quando algum PNJ é atingido.

Uma deficiência inicial constatada foi a impossibilidade de carregamento de mais de um quadro do MD2, devido à pouca quantidade de memória *heap* padrão disponível tanto no emulador quanto no dispositivo real onde foram feitos testes. Embora o suporte à animação esteja implementado e tenham sido feitos testes com até 5 quadros de animação para o modelo Knight, invariavelmente surge exceção de falta de memória. De certa forma isto já era esperado no início do projeto, entretanto seguiu-se a premissa de que brevemente os telefones celulares rodando Java terão memória suficiente para execução de jogos maiores e mais complexos.

Para avaliação do problema de quantidade disponível de memória foi utilizado o monitor de memória do Sun Java Wireless Toolkit. Constatou-se que quando ocorre a exceção "Out of memory", ela é realmente devida ao consumo de memória por parte das classes instanciadas. A configuração padrão do emulador possui cerca de 500 Kb de memória *heap*, compatível com a quantidade existente na maioria dos celulares atualmente disponíveis no mercado. Porém alguns celulares mais novos já possuem *heap* de 1.5 Mb. Percebeu-se também que o problema não está exatamente no armazenamento do modelo e sim na complexidade do procedimento de importação que demanda a utilização de diversas classes até que se tenha o nodo Mesh pronto para ser desenhado. Na verdade, para a leitura dos personagens foram utilizados mais de 500 Kb, porém antes que o *heap* chegasse ao seu limite alguns objetos instanciados que não iriam ser mais utilizados eram descartados pelo *Garbage Collector* do Java.

Ainda destaca-se a demora na leitura e visualização do modelo que leva em torno de trinta segundos para aparecer na tela do celular, acarretando em um tempo total para a carga completa do jogo em torno de um minuto e quarenta e cinco segundos. Isto pode ser considerado um fato negativo, pois alguns jogos podem necessitar que seus PNJs sejam apresentados instantaneamente na cena, e o problema se agrava quando há a necessidade de apresentar mais de um personagem simultaneamente. Nos testes realizados em um aparelho celular Siemens CX 65, a renderização ocorreu em 5 *frames* por segundo (fps).

Tabela 2: *Testes de consumo de memória*

Modelo	Weapon	Axe	Centaur	Knight
Tam MD2 (Kb)	38.0	62.0	240.0	312.0
Tam PNG (Kb)	42.8	7.29	157.0	132.0
Num Text Coord	70	78	469	307
Num Triang	72	126	536	634
Num Frames	173	173	199	198
Mem 1 Frame	84.5	57.0	276.0	263.9
Mem 5 Frames	93.5	72.6	322.2	338.5
Mem all Frames	471.9	732.8	3053.9	3942.4

Alguns testes adicionais foram realizados com o emulador, configurando-se sua memória *heap* no valor máximo permitido de 65536 Kbytes. A tabela 2 apresenta os resultados de consumo de memória em Kb de quatro modelos MD2 diferentes, com exibição e apenas um quadro de animação (*Mem 1 Frame*), cinco quadros (*Mem 5 Frames*) e finalmente todos os quadros (*Mem all Frames*). Os testes consideram o tamanho do arquivo de modelo MD2 (*Tam MD2*) em Kb, o tamanho do arquivo de texturas (*Tam PNG*), as quantidades de coordenadas de textura (*Num Text Coord*), triângulos (*Num Triang*) e quadros de animação (*Num Frames*). Para todos os casos, a taxa de renderização ficou em torno de 15 a 17 fps. Na figura 11 são exibidos *screenshots* dos 4 testes. Os resultados mostram um consumo de memória ainda proibitivo, porém passível de se tornar viável com o aumento de memória disponível nos futuros modelos de telefones celulares.

Atualmente a motor de jogos wGEN possui mais funcionalidades implementadas do que a M3GE (tais como recursos de áudio, por exemplo). Porém a M3GE acaba ganhando a possibilidade de importar arquivos MD2, coisa que não é possível na wGEN, além do fato de a wGEN ser eminentemente um motor para jogos 2D. Devido ao fato da wGEN ter sido implementada utilizando a J2ME, acredita-se que caso futuramente ela venha a ter recursos 3D, poderá com relativa facilidade incluir modelos MD2 utilizando a implementação feita no presente trabalho. Isto será bem simples, pois o presente projeto foi desenvolvido com características de um *plugin* à M3GE.

Figura 11: *Testes de consumo de memória*

O *framework* mOGE é implementado em um nível mais baixo, acessando diretamente a biblioteca OpenGL ES, diferentemente da solução adotada pela M3GE que trabalha com o M3G. No caso da M3GE, a M3G já traz muitas funcionalidades de visualização de objetos 3D em dispositivos móveis, que auxiliou no desenvolvimento do projeto. Por exemplo, o fato de poder incluir um nodo Mesh ao grafo de cena e a própria API M3G ser capaz de desenhar os objetos anexos a este grafo.

10 Conclusões

O presente trabalho apresentou um estudo e a implementação de recursos para inclusão de personagens não-jogadores na M3GE. Isto é feito através da leitura e visualização de modelos no formato M2D.

A visualização dos modelos apresentou boa qualidade gráfica, entretanto, conclui-se que utilizar o arquivo MD2 para adicionar

PNJ aos jogos é atualmente inviável no desenvolvimento de jogos comerciais perante a ainda pequena quantidade de memória disponível e à demora em renderizar os personagens nos atuais aparelhos de telefonia celular disponíveis no mercado brasileiro. Porém, acredita-se que este problema poderá ser contornado de duas formas. A primeira consiste na tendência de melhoria nos recursos de hardware dos futuros dispositivos móveis. A segunda consiste em utilizar técnicas de redução de nível de detalhe das malhas de polígonos dos modelos, diminuindo assim a carga de processamento gráfico exigida para sua renderização. Acredita-se que isto pode trazer bons resultados, e justifica-se devido à baixa resolução gráfica dos visores dos atuais aparelhos celulares.

Para testes e validação dos resultados, foi implementado um protótipo de jogo com mais de um PNJ em cena, entretanto, devido a limitações de memória dos dispositivos reais, não foi possível visualizar de forma satisfatória as animações dos modelos MD2. Foi possível a visualização de todos os quadros de animação nos modelos MD2 testados em emuladores, desde que estes fossem configurados para permitir o uso máximo de memória permitido.

Os resultados obtidos neste trabalho demonstram que é possível utilizar o arquivo MD2 para a manipulação de personagens em motores de jogos para dispositivos móveis, apesar da demora na importação dos modelos. Desta forma, abre-se a possibilidade de desenvolvimento de estudos que possibilitem novas descobertas para viabilizar a utilização de MD2 em telefones celulares, especialmente considerando as animações dos modelos como um requisito altamente desejável para o desenvolvimento de jogos com PNJs. Com esta primeira experiência, os autores sugerem também estudos para a integração de recursos de Inteligência Artificial aos PNJs, de forma a melhorar a qualidade dos jogos 3D para dispositivos móveis.

Referências

BYL, P. B. 2004. *Programming believable characters for computer games*. Charles River Media, Massachusetts.

CARANDINA, R., AND MARTINS, T. 2006. Suporte a jogos multiusuário em engine 3d para dispositivos móveis. Tech. rep., Centro Universitário Senac, São Paulo, Junho.

CONITEC CORPORATION, 2004. 3d gamestudio / a6. `http://conitec.net/a4info.htm`, Acesso em: 17 abr. 2006.

CRYSTAL SPACE, 2004. Crystal space 3d. `http://crystal.sourceforge.net`, Acesso em: 17 abr. 2006.

DE ALBUQUERQUE MACÊDO JR., I. J. 2005. moge – mobile graphics engine: o projeto de um motor gráfico 3d para a criação de jogos em dispositivos móveis. Tech. rep., UFPE.

ECLIPSE ENTERTAINMENT, 2004. Welcome to the home of genesis3d. `http://www.genesis3d.com/`, Acesso em: 17 abr. 2006.

ELEMENTS INTERACTIVE, 2007. Edgelib mobile game engine. `http://www.edgelib.com/`, Acesso em: 25 ago. 2007.

FINNEY, K. C. 2004. *3D game programming, all in one*. Thomson Course Technology, Boston.

FISHLABS ENTERTAINMENT GMBH, 2007. Fishlabs 3d mobile java games. `http://www.fishlabs.net/en/engine/index.php`, Acesso em: 25 ago. 2007.

GOMES, P. C. R., AND PAMPLONA, V. F. 2005. M3ge: um motor de jogos 3d para dispositivos móveis com suporte a mobile 3d graphics api. In *SBGames2005 - Simpósio Brasileiro de Jogos para Computador e entretenimento Digital, Anais do WJOGOS 2005*, SBC, 55–65.

GUILIANO, S., 2005. Autonomous productions. `http://www.autonomousproductions.com/`, Acesso em: 14 set. 2005.

HARRISON, L. T. 2003. *Introduction to 3D game engine design using directx 9 and C#*. Springer-Verlag, New York.

HENRY, D., 2004. Md2 file format (quake 2's model). `http://tfc.duke.free.fr/coding/md2-specs-en.html`, Acesso em: 24 mar. 2006.

HÖFELE, C., 2005. 3d graphics for java mobile devices, part 1: M3g's immediate mode. `http://www-128.ibm.com/developerworks/wireless/library/wi-mobile1/`, Acesso em: 17 mar. 2006.

KHRONOS GROUP, 2004. Opengl es: overview. `http://www.opengles.org/opengles/index.html`, Acesso em: 17 abr. 2006.

LEHMANN, J., 2004. Xith3d: contents. `http://xith.org/tutes/GettingStarted/html/contents.html`, Acesso em: 25 out. 2006.

MD2, 2005. Md2 - gpwiki. `http://gpwiki.org/index.php/MD2`, Acesso em: 24 mar. 2006.

MISFIT CODE, 2005. Misfit model 3d. `http://www.misfitcode.com/misfitmodel3d/olh_quakemd2.html`, Acesso em: 24 mar. 2006.

MÜLLER, L. F., FRANTZ, G. J., AND SCHREIBER, J. N. C. 2005. Qualcomm brew x sun j2me: um comparativo entre soluções para desenvolvimento de jogos em dispositivos móveis. In *SULCOMP - Congresso Sul Catarinense de Computação*, vol. 1, Sulcomp, 1–8.

NOKIA. 2003. Jsr-184 mobile 3d api for j2me. Tech. rep., [P.O.Box].

NOKIA, 2004. Ri binary for jsr-184 3d graphics api for j2me. `http://www.forum.nokia.com/main/`, Acesso em 19 abr. 2006.

OGRE3D, 2005. Ogre 3d: open source graphics engine. `http://www.ogre3d.org`, Acesso em: 15 jul. 2007.

PAIVA, F., 2003. O jogo das operadoras. `http://www.teletime.com.br/revista/55/capa.htm`, Acesso em: 05 abr. 2006.

PARALELO COMPUTAÇÃO, 2004. Fly3d.com.br. `http://www.fly3d.com.br`, Acesso em: 17 abr. 2006.

PESSOA, C. A. C. 2001. *wGEM*. Master's thesis, UFPE, Recife.

SÁNCHEZ, D., AND DALMAU, C. 2004. *Core techniques and algorithms in game programming*. New Riders, Indianapolis.

SANTEE, A. 2005. *Programação de jogos com C++ e DirectX*. Novatec Editora, São Paulo.

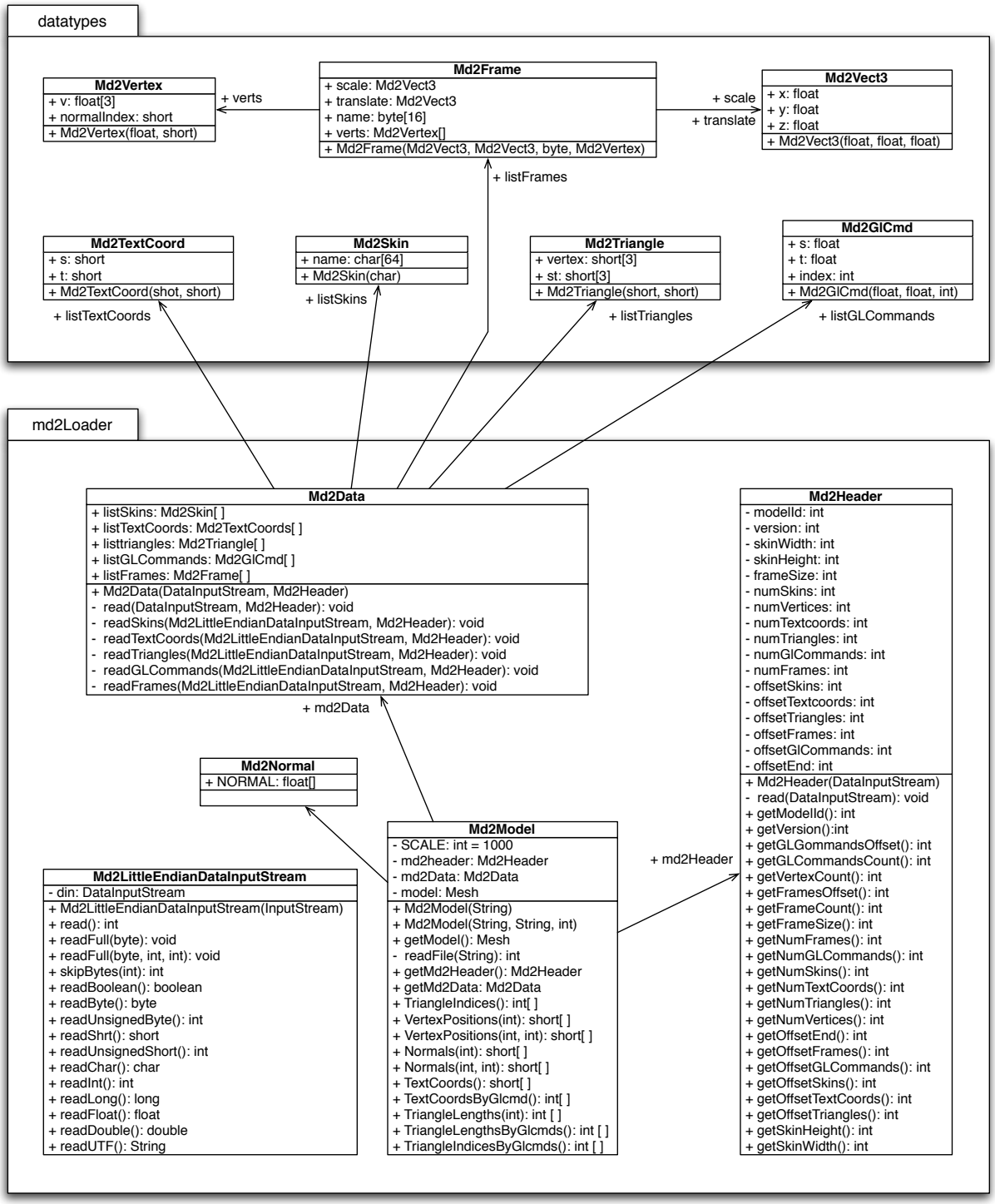

Figura 12: *Diagrama de classes dos pacotes md2loader e datatypes*