UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

IMPLEMENTAÇÃO DE IMPORTAÇÃO E VISUALIZAÇÃO DE MODELOS DE PERSONAGEM NÃO JOGADOR (PNJ) NA MOBILE 3D GAME ENGINE (M3GE)

CLAUDIO JOSÉ ESTÁCIO

BLUMENAU
2006

2006/2-04

CLAUDIO JOSÉ ESTÁCIO

IMPLEMENTAÇÃO DE IMPORTAÇÃO E VISUALIZAÇÃO DE MODELOS DE PERSONAGEM NÃO JOGADOR (PNJ) NA MOBILE 3D GAME ENGINE (M3GE)

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. Paulo Cesar Rodacki Gomes, Dr. - Orientador

BLUMENAU
2006

2006/2-04

IMPLEMENTAÇÃO DE IMPORTAÇÃO E VISUALIZAÇÃO DE MODELOS DE PERSONAGEM NÃO JOGADOR (PNJ) NA MOBILE 3D GAME ENGINE (M3GE)

Por

CLAUDIO JOSÉ ESTÁCIO

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

Presidente:	Prof. Paulo Cesar Rodacki Gomes, Dr. – Orientador, FURB
Membro:	Prof. Maurício Capobianco Lopes – FURB
Membro:	Prof. Dalton Solano dos Reis – FURB

Blumenau, 06 de dezembro de 2006

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente e indiretamente na realização deste.

AGRADECIMENTOS

À Deus, pelo seu imenso amor e graça.

À minha família, que sempre me apoiou.

A minha eterna companheira, Graziela Schappo, pela compreensão e força dada durante todo período da graduação.

Aos meus amigos, pelos empurrões e cobranças.

Ao meu orientador, Paulo Cesar Rodacki Gomes, por ter acreditado na conclusão deste trabalho.

Quando se busca o cume da montanha, não se dá importância às pedras do caminho.

Autor Desconhecido

RESUMO

Este trabalho implementa importação e visualização à Personagem Não Jogador (PNJ) ao motor de jogos Mobile 3D Game Engine (M3GE). É agregada ao motor de jogos a possibilidade de importar um modelo num formato de arquivo muito utilizado para a modelagem de personagens para jogos, o *Quake 2's Models* (MD2). Estes arquivos trabalham com animação quadro a quadro, mas pelo fato de estar trabalhando com pouca memória disponível foi possível importar apenas alguns quadros para demonstração. É utilizado o exemplo de jogo presente na M3GE para a inserção de três personagens MD2 diferentes, porém sem animação, estando somente um quadro de cada um deles presente na visualização. O PNJ para funcionar por completo necessita ter Inteligência Artificial (IA), porém este trabalho não implementa esta funcionalidade. São tratados apenas questões sobre a visualização.

Palavras-chave: Jogos. Computação gráfica. Animação. Personagem Não Jogador (PNJ). Motor de jogos. M3GE. M3G. Dispositivos móveis. J2ME. MD2.

ABSTRACT

This work presents the implementation of Non Player Character (NPC) visualization in the Mobile 3D Game Engine (M3GE). For NPC models, we use Quake 2 Models animated format (MD2). We add to the game engine a module for importing MD2 models, which are used by many game development platforms. The MD2 files store keyframe animation information, however, we were unable to get good results due to the small amounts of available memory in the mobile devices currently in the market. We implement a game prototype in order to demonstrate the results, which show the 3d models with textures and limited keyframe animation. The PNJ to function completely needs to have Artificial Intelligence (AI), however this work does not implement this functionality. Only questions on the visualization are treated.

Key-words: Games. Graphical computation. Animation. Non-Player Character (NPC). Game engine. M3GE. M3G. Mobile devices. J2ME. MD2.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

2D – Duas Dimensões

3D – Tridimensional

API – *Application Program Interface*

BREW – *Binary Runtime Enviroment for Wireless*

DSC – Departamento de Sistemas e Computação

FURB – Universidade Regional de Blumenau

IA – Inteligência Artificial

J2ME – *Java 2 Micro Edition*

M3G – *Mobile 3D Graphics API*

M3GE – *Mobile 3D Game Engine*

MD2 – *quake 2's MoDels*

NPC – *Non-Player Character*

PNJ – Personagem Não Jogador

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Os jogos eletrônicos são uma opção de entretenimento que sempre despertou interesse nas pessoas, principalmente os mais jovens. Os jogos estão se modernizando constantemente e, entre as mais importantes inovações está a utilização de animação tridimensional (3D).

Em poucos anos a evolução dos jogos em consoles e computadores modificaram sua simplicidade dos gráficos em duas dimensões (2D) e sons reproduzidos com algumas notas de bips para um mundo mais realista com gráficos 3D repletos de ação e sons com capacidade de envolver o jogador. Visto esse sucesso, pode-se afirmar que o mesmo acontecerá nos jogos para celulares. Atualmente a maioria dos jogos nesta plataforma são em 2D, mas já existe iniciativa para desenvolvimento em 3D.

Para facilitar o desenvolvimento dos jogos, utilizam-se os motores, que são bibliotecas de software capazes de tratar a maioria das necessidades existentes na sua implementação. A *Mobile 3D Game Engine* (M3GE) é um motor de jogos para telefones celulares desenvolvido no Departamento de Sistemas e Computação (DSC) da Universidade Regional de Blumenau (FURB) (GOMES; PAMPLONA, 2005). Ela é baseada na Java Mobile 3D Graphics API (M3G) (NOKIA, 2004) e já possui algumas funcionalidades implementadas, como, por exemplo, carregar e desenhar um ambiente 3D, criação de câmeras, tratamento de eventos, movimentação de personagens no cenário 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.

Este projeto detalha uma melhoria feita à M3GE que foi a integração de Personagem Não Jogador (PNJ). 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 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 diversos inimigos, alguns parecidos mais com humanos, outros com figuras mitológicas, mutantes, robôs, entre outros. Estes personagens são criados com relação ao enredo do jogo.

Para isso, foram agregadas ao motor funções capazes de importar e visualizar modelos animados de personagens no formato de arquivo *Quake 2's Models* (MD2) (HENRY, 2004). MD2 é um dos formatos mais populares para inclusão de PNJs em jogos 3D. Além de ser bem difundido, tem uma estrutura simples que facilita sua implementação e geralmente é um

arquivo pequeno, o que é uma grande vantagem no caso de jogos em celulares, devido às limitações de processamento e memória deste tipo de dispositivo atualmente. Este formato tem por característica armazenar a animação do personagem quadro a quadro dentro do próprio arquivo.

Para validar a implementação, foram feitos testes num emulador de celulares que imitam um aparelho com tecnologia M3G. Com isso pôde-se avaliar se a aplicação está preparada para os aparelhos móveis do mercado atual.

Normalmente, os PNJs interagem com o jogador como se tivessem vida própria, e para isso são utilizados recursos de Inteligência Artificial (IA), mas pelo fato do presente trabalho estar voltado à importação e visualização do personagem, assuntos relacionados a IA não são tratados.

Este trabalho demonstra-se de grande utilidade, pois visa o aumento na venda de dispositivos móveis que faz com que a exigência na qualidade das aplicações existentes neles também aumente. Assim acontece com os jogos, que começam a exigir maior realismo. Tendo em vista o fato da M3GE ser uma das iniciativas pioneiras a implementar um motor de jogos 3D para celulares utilizando a linguagem Java, a inserção de modelos de personagens animados 3D será de grande utilidade para desenvolvedores que a utilizarem.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é agregar PNJ na M3GE, implementando suporte a importação e visualização de modelos animados 3D no formato de arquivo MD2.

Os objetivos específicos do trabalho são:

a) importar arquivos de modelos 3D no formato MD2;
b) integrar o modelo no grafo de cena da M3GE;
c) visualizar o modelo junto à cena;
d) criar um protótipo de jogo utilizando mais de uma instância de PNJ;
e) testar o protótipo do jogo num emulador de telefone celular.

1.2 ESTRUTURA DO TRABALHO

Este trabalho divide-se em teoria, prática e conclusões. No item fundamentação teórica

procura-se passar ao leitor todo embasamento necessário para o entendimento da pesquisa realizada. Através dele pode-se entender como funciona um motor de jogos, para que servem os PNJs, também é apresentada as características do formato MD2 escolhido para a importação dos modelos. No item desenvolvimento do trabalho é apresentada a implementação do trabalho. Ela começa a formalização da implementação usando diagramas da UML, logo após é apresentada as principais funcionalidades desenvolvidas, e por últimos os testes realizados. Ainda ao final deste trabalho é apresentada uma conclusão onde estão detalhados os comentários junto aos resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são apresentados os principais assuntos que têm relação com o trabalho proposto, os quais são: jogos em dispositivos móveis, motores de jogos 3D, M3GE, Personagem Não Jogador (PNJ) e modelo animado MD2. Ainda ao final são apresentados dois trabalhos correlatos, o wGEN e o mOGE.

2.1 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. Hoje os aparelhos celulares rodam aplicativos, tiram fotos, enviam e recebem correio eletrônico, funcionam como tocadores de música digital e ainda apresentam jogos para entretenimento do usuário. Existem ainda mais funcionalidades disponíveis e outras que surgem a cada evolução das tecnologias (PALUDO, 2003).

Em função dessas características, as empresas vêm investindo muito dinheiro em soluções para o desenvolvimento de jogos para esses aparelhos. Estão entre as soluções mais utilizadas pela comunidade de desenvolvedores 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 ter a *Kilobyte Virtual Machine* (KVM) que é uma máquina virtual (MÜLLER; FRANTZ; SCHREIBER, 2004, p. 2).

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. O BREW apresenta diversos perfis para desenvolvimento, sendo usados para jogos: *Multimedia* e *Enchanced*. 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 pela 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; FRANTZ; SCHREIBER, 2004, p. 4).

O padrão Mophun foi desenvolvido pela sueca Synergenix. É o 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).

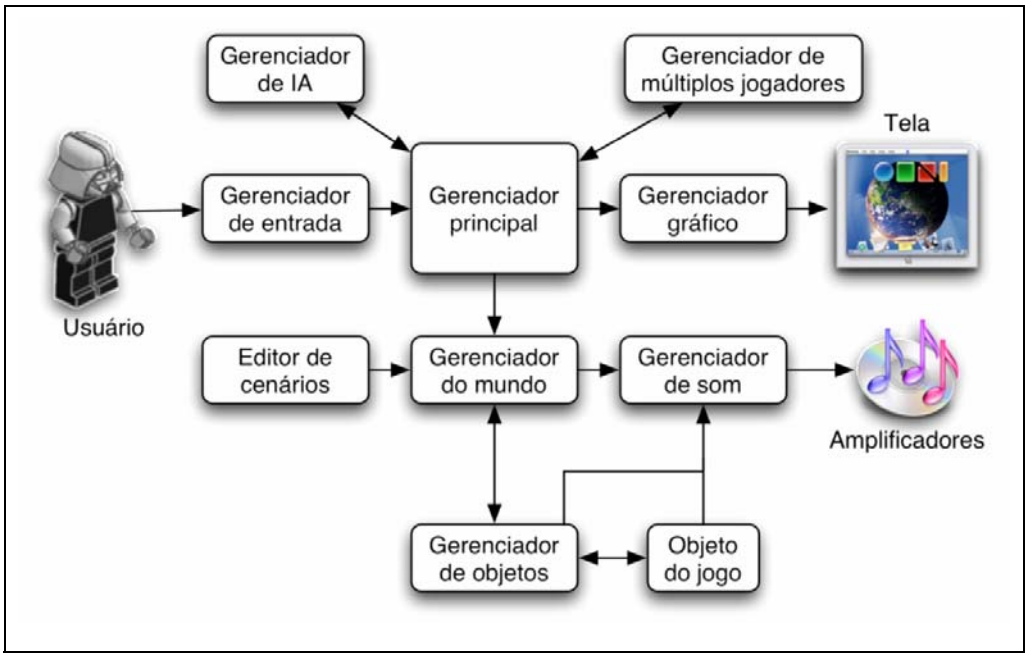
2.2 MOTORES DE JOGOS 3D

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. Os tipos de jogos mais populares são: tiro em primeira pessoa, estratégia em tempo real e simulação de veículos (HARRISON, 2003, p. 1).

Conforme definido em Macêdo Júnior (2005, p. 15), um motor de gráfico 3D é um sistema que produz uma visualização em tempo real respondendo aos eventos da aplicação. Para o funcionamento desse processo deve-se conhecer 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.

Para Finney (2004, p. 16-17) o motor disponibiliza as principais características para o desenvolvimento de jogos, como por exemplo: renderização de cenário 3D, recursos de comunicação em rede, gráficos, *scripting*, entre outros. Para motores 3D seu principal foco está na apresentação visual do jogo. Assim, por eles serem tridimensionais, cálculos matemáticos são usados para renderizar cada quadro da visualização, juntando-se os vértices com as texturas. Esses recursos devem dar mais realismo aos personagens do jogo quando estes executam ações, como por exemplo, correr, pular e lutar. Alguns exemplos de motores de jogos 3D são: Crystal Space (CRYSTAL SPACE, 2004), Genesis3D (ECLIPSE ENTERTAINMENT, 2004), 3D Game Studio (CONITEC CORPORATION, 2004) e Fly3D (PARALELO COMPUTAÇÃO, 2004).

A Figura 1 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, isto é, os mapas. Também podem ser exemplificado no editor de cenário a criação dos personagens 3D. (GOMES; PAMPLONA, 2005).



Fonte: adaptado de Gomes e Pamplona (2005, p. 2).

Figura 1 – Arquitetura de um motor de jogos

2.3 M3GE

A *Mobile 3D Game Engine* é um protótipo de motor de jogos para dispositivos móveis com suporte a M3G. Foi desenvolvida no Departamento de Sistemas e Computação da FURB

(GOMES; PAMPLONA, 2005).

A M3GE já apresenta diversas funcionalidades implementadas, os quais são: carregar e desenhar um ambiente 3D, criação de câmeras, tratamento de eventos, movimentação de personagens no cenário e tratamento de colisão. É apresentado um exemplo de um jogo com as funcionalidades implementadas, onde um cenário 3D com objetos estáticos pode ser visualizado e um jogador que é representado por um simples cubo. Este jogador é controlado pelos comandos do teclado do celular e também estão inseridas nele funções de tratamento de colisão e capacidade de dar tiros nos objetos da cena. A implementação da M3GE é feita sobre a M3G que por sua vez está implementada através da J2ME, ambas apresentadas a seguir.

Conforme comenta Pinheiro (2003), os primeiros dispositivos móveis continham programas simples e limitados. Com o aumento na exigência dos usuários as aplicações começaram a ficar mais robustas e com maior qualidade, e para isso não ficar limitado ao desenvolvimento somente pelas empresas fabricantes dos aparelhos surgiu a *Java 2 Micro Edition* (J2ME) que é uma linguagem de programação Java para dispositivos móveis. Assim, basta o fabricante disponibilizar em seus aparelhos uma máquina virtual para a execução dos programas, mantendo-se desta forma o sigilo das soluções proprietárias.

Segundo Höfele (2005), M3G é uma API Gráfica 3D para se utilizar 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.

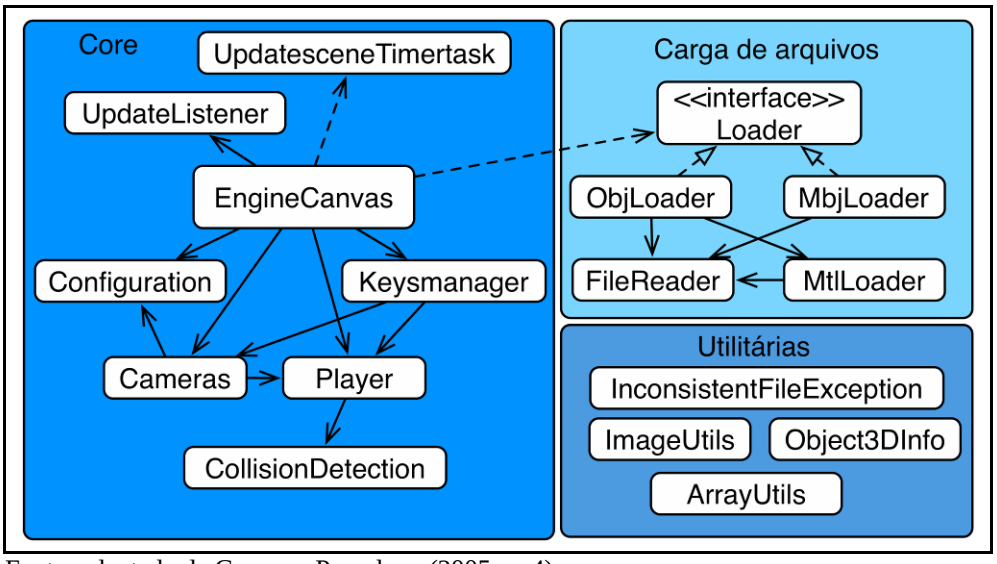
O motor M3GE foi projetado anexo à M3G, significando que se pode acessar diretamente a M3G na aplicação, se necessário. Isto dá maior flexibilidade na implementação dos jogos. O motor foi dividido em dois grandes componentes: o responsável pela leitura dos arquivos de entrada e o *core* conforme descrito em seguida. 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. São elas que 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 do grafo de cena, uma animação, uma textura, entre outros. Este grafo de cena na verdade é uma árvore que armazena toda a estrutura dos objetos do jogo. Em cada nodo da árvore estão presente um ou mais grupos, que por sua vez tem interligados os objetos 3D da classe `object3D`. Eles podem ser um modelo gráfico 3D, uma câmera, uma luz, uma imagem 2D num mundo 3D, entre outros.

A Figura 2 apresenta a arquitetura geral da M3GE, mostrando as classes do core, de

carga de arquivos e utilitárias. O *core* 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 primeira é utilizada para a carga de um arquivo *Wavefront*, que é utilizado para criação de cenário e configurações do ambiente. A segunda é utilizada para a carga do arquivo *Mobile Object File*, que foi criado na M3GE para simplificar o arquivo anterior e facilitar a leitura dos dados. A classe `MtlLoader` é utilizada para a carga de um arquivo de extensão *mtl*, que descreve cores, propriedade de materiais e texturas do ambiente ou de um grupo de objetos 3D. O arquivo *mtl* é utilizado juntamente com o arquivo *Wavefront*. A classe `FileReader` é uma classe especial para facilitar a leitura dos arquivos.

Ainda foram criadas classes para auxiliarem as demais. A classe `InconsistentFileException` que lança uma exceção quando ocorre algum erro na leitura do arquivo. A classe `ImageUtils` é utilizado 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.



Fonte: adaptado de Gomes e Pamplona (2005, p. 4).

Figura 2 – As classes da M3GE

Durante o desenvolvimento da M3GE teve-se que criar um novo formato de arquivo para facilitar a importação de modelos 3D. Isto ocorreu, pois o arquivo escolhido para a importação do cenário, o *Wavefront (obj)*, necessitava ser lido três vezes, e como alguns celulares não tem *cache* de armazenamento essa leitura ficava muito lenta. Mesmo criando um arquivo novo, o *Mobile Object File (mbj)*, para facilitar a leitura, isto reduziu a carga do arquivo em apenas dois segundos. Isto é, nos testes feitos, o arquivo *obj* levou 17 segundos para ser carregado e visualizado, enquanto que o arquivo *mbj* levou 15 segundos.

As conclusões relatadas em Gomes e Pamplona (2005) definiram que Java é promissora para a implementação de jogos para dispositivos móveis, porém sempre com algoritmos velozes. Apesar da M3GE ser apenas um protótipo, ele já vem sendo utilizada para desenvolvimento de jogos comercialmente. Além disso, outra funcionalidade acaba de ser integrada ao motor de jogos, a possibilidade de comunicação entre os celulares através de *bluetooth*.

2.4 PERSONAGEM NÃO JOGADOR (PNJ)

O termo 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 jogos são formas de participar de uma aventura fictícia. Nessa aventura podem existir tanto amigos quanto inimigos. Aí entram os personagens, que são responsáveis por fazer os jogadores sentirem emoções quando jogam. Eles ainda podem ser elementos decorativos das cenas ou interagir diretamente com as ações do jogador (SÁNCHEZ; DALMAN, 2004, p. 1). 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 diversos inimigos, alguns parecidos mais com humanos, outros com figuras mitológicas, mutantes, robôs, entre outros. Estes personagens são criados com relação ao enredo do jogo.

Segundo Byl (2004, p. 5-6), 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.

2.5 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 2 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. Outra característica é a sua estrutura de dados poder usar para desenho do modelo primitivas gráficas do OpenGL (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 divido em duas partes: cabeçalho e dados, comentadas a seguir.

2.5.1 Cabeçalho

Conforme descrito em 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. Por isso primeiro deve-se fazer a leitura do cabeçalho, onde cada informação está armazenada em quatro *bytes* representados por um tipo de dado inteiro.

Antes de apresentar o conteúdo do cabeçalho precisam-se descrever alguns conceitos:

a) quadro chave: a animação do modelo é representada por uma seqüência de quadros, e cada quadro chave representa uma posição do personagem. Quando estes são exibidos em seqüência dá-se a visualização dos movimentos. 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;

b) coordenadas de textura: as coordenadas de textura são utilizadas para mapear a textura ao modelo 3D. Isto é feito através das coordenadas *s* e *t*. Estas coordenadas são organizadas entre dois eixos conhecidos como *s*, para o eixo *x* da textura, e *t*, para o eixo *y*. Cada vértice do modelo 3D recebe uma coordenada de textura *s* e *t*;

c) *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 ao modelo através das coordenadas de textura. No cabeçalho encontram-se o tamanho e a quantidade de *skins* que podem ser utilizados com o modelo;

d) comando OpenGl: os comandos OpenGl presentes no arquivo servem para ser 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_STRIP caso seja positivo ou renderizado com a primitiva GL_TRIANGLE_FAN caso seja negativo. Os seguintes números vêm na seqüência divididos de três em 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;

e) triângulos: são os que definem as faces do modelo. Eles são uma lista de índices aos vértices. Cada conjunto de três índices representa um triângulo;

f) vértices: são os pontos no mundo cartesiano que desenham o modelo 3D. São representados pelos eixos: *x*, *y* e *z*.

g) vetor normal: é um vetor que esta contido na estrutura dos vértices utilizado para calcular a iluminação do modelo. É utilizado para indicar qual a direção está o vértice.

A estrutura completa do cabeçalho pode ser vista no Quadro 1. Esta estrutura representa a mesma seqüência presente dentro do arquivo MD2.

Conteúdo	Descrição
magic	Número Mágico (deve ser igual a 844121161)
version	Versão do arquivo (para Quake2 precisa ser igual a 8)
skinWidth	Largura do skin do modelo (em pixels)
skinHeight	Altura do skin (em pixels)
frameSize	Tamanho (em bytes das informações dos frames)
numSkins	Número de skins avaliados
numVertices	Número de vértices
numTexCoords	Número de coordenadas da textura (vetores bidimensionais ST)
numTriangles	Número de triângulos
numGlCommands	Número de comandos GL
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 GL
offsetEnd	Posição para o final do modelo

Fonte: adaptado de Henry (2004).

Quadro 1 – Descrição do cabeçalho do arquivo MD2

2.5.2 Dados

As informações de cada vértice do modelo são armazenadas em alguns 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 o tamanho da visualização do jogo apresentada em um visor desses aparelhos é bem menor do que o 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:

a) vetor: composto por três coordenadas do tipo float;

b) informação textura: lista de nomes de texturas associadas ao modelo;

c) coordenadas da textura: são armazenadas na estrutura como short integers;
d) triângulos: cada triângulo possui uma lista com os índices dos vértices e uma lista com os índices das coordenadas da textura;
e) vértices: são compostos de coordenadas 3D, onde são armazenados em um byte cada coordenada e um índice do vetor normal;
f) quadros: têm informações específicas da lista de vértice do quadro;
g) comandos OpenGL: são armazenados em uma lista de integers.

2.5.3 Texturas

O modelo MD2 tem sua textura presente em um arquivo separado e utiliza somente uma textura de cada vez. Todos os vértices do modelo devem ser mapeados para essa mesma textura. Por este motivo a textura deve ter diferentes cores para representar cada parte do personagem, desde os pés até a cabeça, chamados assim de *skins* (MISFIT..., 2005). Na Figura 3 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.

Fonte: adaptado de Java is Doomed (2006).

Figura 3 – Visualização da textura no personagem (*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,

como, PNG, BMP, GIF, entre outros.

2.6 TRABALHOS CORRELATOS

Podem-se relacionar ao projeto desenvolvido dois motores de jogos para dispositivos móveis: o wGEN e o mOGE.

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 parecidas com este trabalho é o Mobile Graphics Engine (mOGE) (MACÊDO JÚNIOR, 2005). Este projeto teve como objetivo prover diversas técnicas avançadas de computação gráfica 3D. Para a geração das imagens da visualização, o mOGE utiliza o *pipeline* gráfico, que é a decomposição deste processo em partes: estágio de aplicação, o de geometria e o de rasterização. Para a representação de objetos foi utilizada malha de triângulos. Para a estrutura dos objetos presentes no jogo é utilizada a árvore do 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 (MACÊDO JÚNIOR, 2005).

Ambos não tem um sessão especifica que trata a importação de PNJs, porém o mOGE apresentam o grafo de cena, que é onde estes personagens devem ser inseridos, para que o gerenciador gráfico do motor possa desenhá-los na tela.

3 DESENVOLVIMENTO DO TRABALHO

Esta sessão apresenta o desenvolvimento do trabalho desde a especificação, passando pela implementação, execução dos testes, e por fim são apresentados os resultados.

Este trabalho teve como base um código fonte escrito na linguagem de programação C disponibilizado por Henry (2004). Através dele pôde-se ter um entendimento do que é necessário para a leitura de um arquivo MD2 e da visualização de seu modelo. Porém este exemplo ainda não implementa visualização das texturas.

Os testes de visualização tiveram base no artigo do Höfele (2005), onde ele descreve e dá exemplos de como utilizar a M3G para a visualização de objetos 3D num aparelho de celular. Ele trata os quesitos da leitura dos vértices, da utilização de câmeras, de luzes, da adição de cores, das transformações e de texturas.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Esta extensão da M3GE possui os seguintes requisitos:

a) importar modelos 3D: deverá ser capaz de ler um arquivo MD2 criando uma classe Java para sua posterior manipulação (Requisito Funcional - RF);
b) inserir personagem ao grafo de cena: o modelo animado terá que estar presente no grafo de cena da M3GE para ser visualizado na cena quando ele for desenhado (RF);
c) visualizar personagem na cena: o personagem, após incluído ao grafo de cena, terá que ser capaz de apresentar seus movimentos extraídos do arquivo MD2 (RF);
d) utilizar a especificação M3G: usar a mesma especificação M3G utilizada na criação da M3GE (Requisito Não Funcional - RNF).

3.2 ESPECIFICAÇÃO

Este trabalho segue o paradigma de orientação a objetos. Além disso, para sua especificação foram utilizados diagramas da UML, sendo eles, diagrama de caso de uso, diagrama de atividades, diagrama de classes e diagrama de seqüência. O diagrama de classes

é muito importante pelo fato de se usar orientação a objetos no desenvolvimento do trabalho. Para a confecção dos diagramas foi utilizada a ferramenta Enterprise Architect.

3.2.1 Diagrama de caso de uso

A Figura 4 demonstra um diagrama de caso de uso que representa as funcionalidades que o jogo pode fazer a partir da M3GE utilizando a importação de modelos MD2. O ator representado pelo jogo tem dois casos de uso. O primeiro tem como função fazer a leitura do arquivo MD2. Após os dados lidos o segundo caso de uso demonstra que o personagem 3D deve ser exibido no tela do jogo.

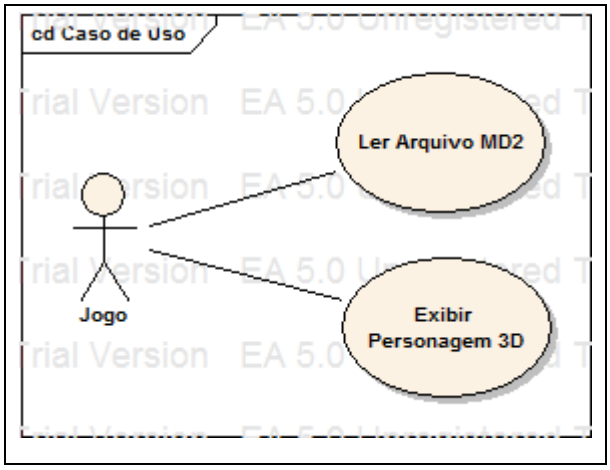


Figura 4 – Diagrama de caso de uso do jogo

3.2.2 Diagrama de atividades

Para se inserir o modelo 3D no grafo de cena para que ele possa ser desenhado pelo motor de jogos ele necessita ter uma estrutura própria. Esta estrutura é apresentada com sendo o nodo *mesh*. Este nodo representa um objeto 3D que armazena diversas características para que seja possível sua manipulação através da API gráfica M3G. Ele armazena posição dos vértices, índices dos triângulos, textura, etc. A Figura 5 demonstra um diagrama de atividades que representa a criação de um nodo *mesh*.

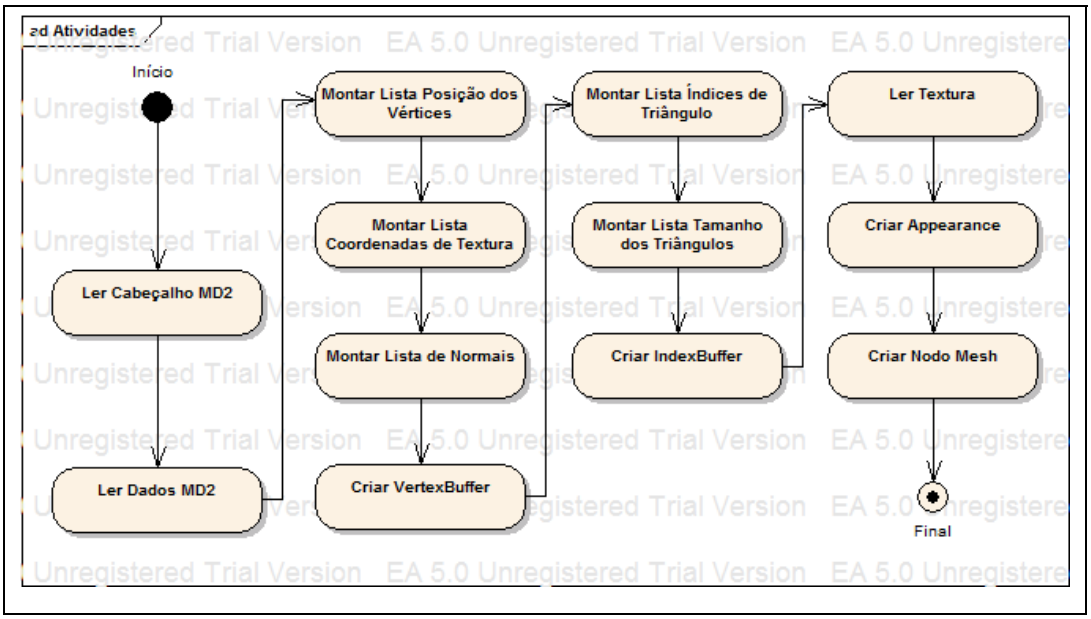


Figura 5 – Diagrama de atividades da criação do nodo *mesh*

3.2.3 Diagramas de classes

Como a M3GE está inserida dentro de M3G e tem-se acesso aos dois módulos, pode-se a princípio criar o carregador de arquivo MD2 em um pacote de classes separado dessas arquiteturas. Em seguida, pode-se usar a M3G para desenvolver a parte de visualização do modelo. E por último pode-se agregar isto ao motor de jogos M3GE.

Para melhor distribuição das classes foram criados dois pacotes, o *md2loader* e dentro deste um pacote chamado *datatypes*.

O pacote *md2loader* apresenta as classes conforme a Figura 6 e descritas em seguida.

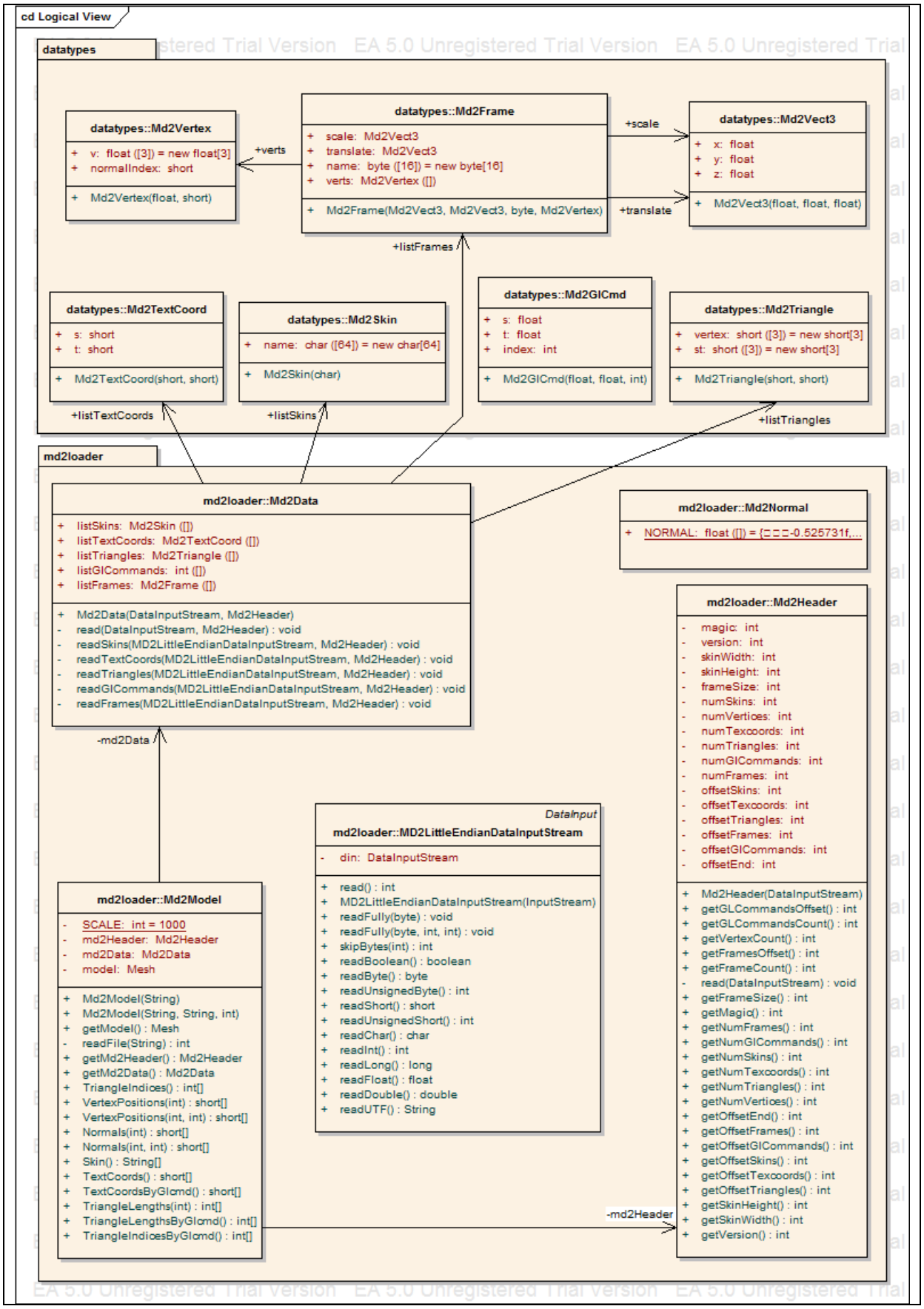


Figura 6 – Diagrama de classes

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 divido em duas partes, cabeçalho e 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 já 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.

O pacote *datatypes* apresenta as classes conforme mostrado na Figura 6 e descritas em seguida.

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 com as três coordenadas do espaço cartesiano tridimensional que são: 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 triangulo 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 de 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.

3.2.4 Diagrama de seqüência

A Figura 7 representa o diagrama de seqüência que demonstra os passos necessários para obtenção do modelo fazendo-se a leitura do arquivo MD2.

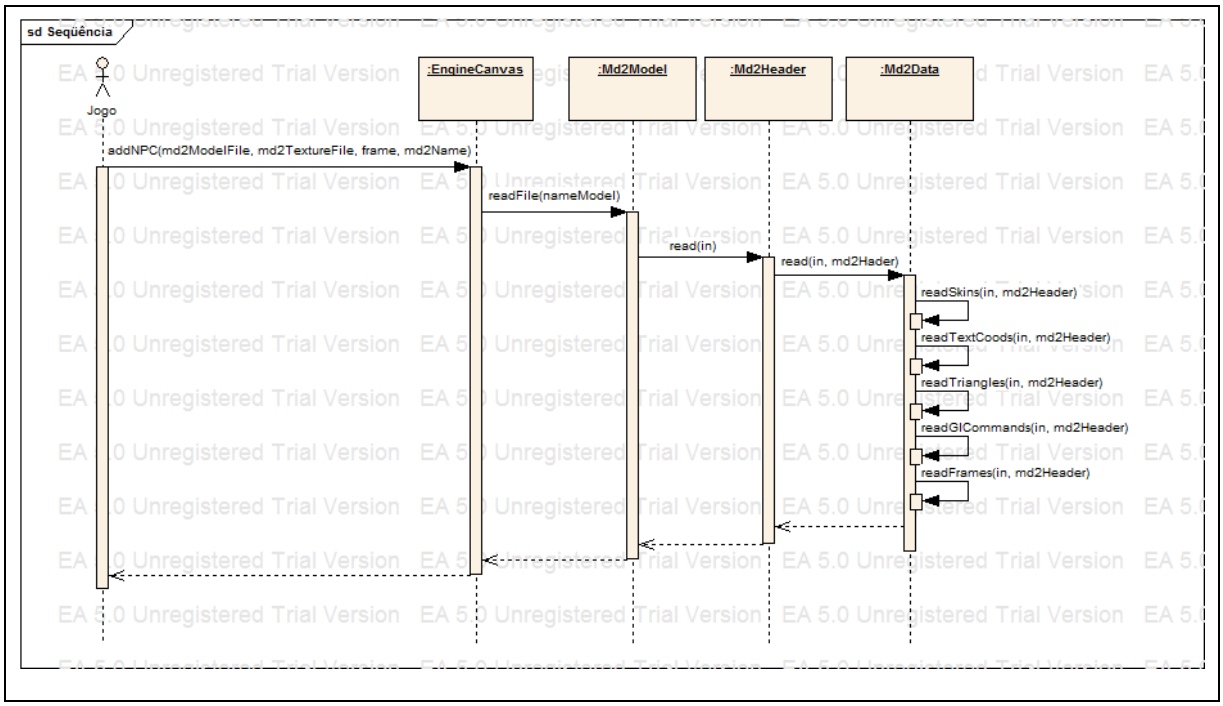


Figura 7 – Diagrama de seqüência do carregamento do arquivo MD2

A importação do modelo começa no jogo que faz uma chamada para a classe `EngineCanvas` através do método *addNPC()* que passa como parâmetros o arquivo MD2, o arquivo de textura, o número do quadro e o nome do personagem. A classe `EngineCanvas` que já estava implementada na M3GE e que é gerenciador principal do motor do jogo faz na seqüência uma chamada a classe `Md2Model`. Esta chamada inicia a leitura do arquivo MD2 passado como parâmetro no método *readFile()*. Posteriormente é feita a leitura do arquivo binário MD2 e seu armazenamento. Primeiramente é feita a leitura do cabeçalho através do método *read()* da classe `Md2Header`. Em seguida são lidos os dados fazendo-se a leitura dos bytes do arquivo utilizando-se o cabeçalho para a localização desses dados através do método *read()*. Por último a classe `Md2Data` vai separando os dados do arquivo em partes através dos métodos: *readSkins()*, *readTextCoords()*, *readTriangles()*, *readGlCommands()* e *readFrames()*. E para o armazenamento destes dados são utilizadas as estruturas do pacote *datatypes*.

3.2.5 Adicionando o carregador de MD2 a M3GE

Devido ao fato do desenvolvimento do trabalho ter sido feito sobre a biblioteca M3G, optou-se por implementar o carregador de MD2 num pacote independente à M3GE conforme mostra a Figura 8. A única alteração feita no antigo código fonte da M3GE foi a adição do

método *addNPC()* à classe `EngineCanvas` conforme apresentado no Quadro 13. Através deste método é possível adicionar um modelo MD2 ao grafo de cena.

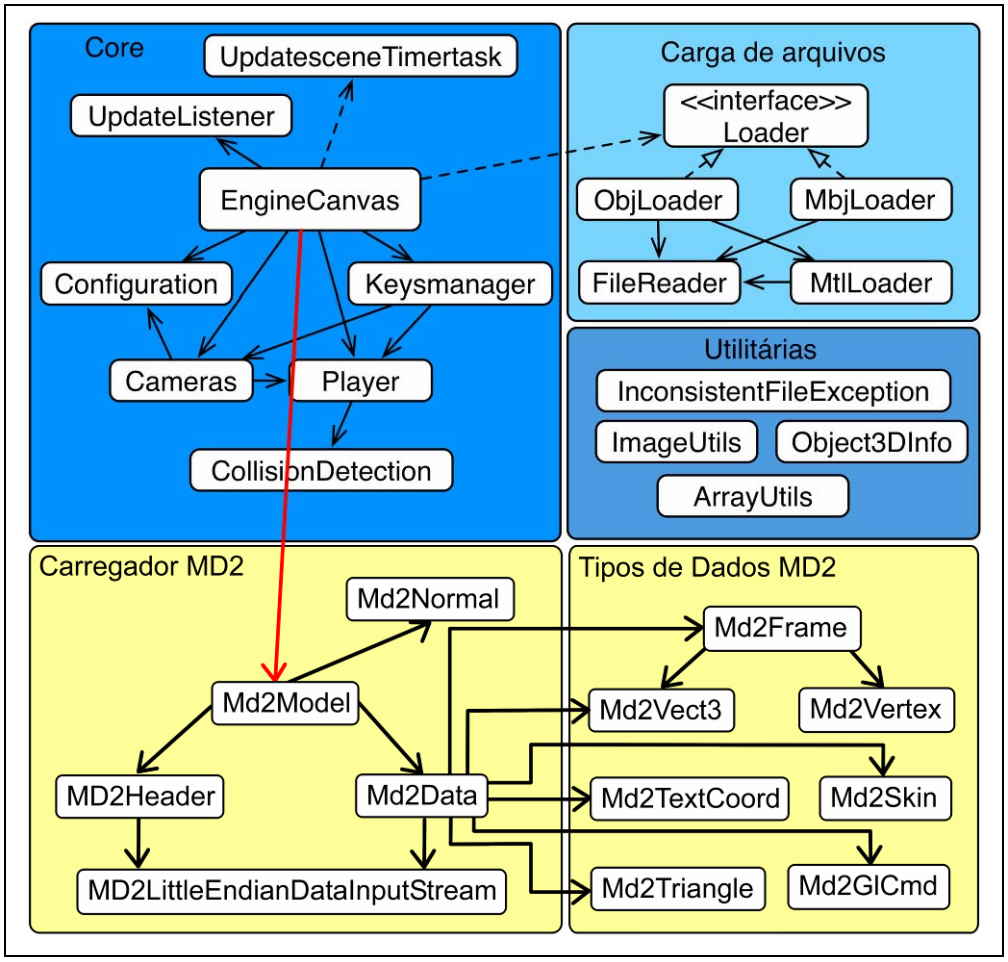


Figura 8 – Carregador MD2 inserido na M3GE

3.3 IMPLEMENTAÇÃO

Esta sessão apresenta o desenvolvimento do código fonte. Em seguida serão apresentadas as ferramentas usadas, as classes criadas e algumas partes de códigos mais relevantes.

3.3.1 Tecnologias utilizadas

Com base no projeto da M3GE foram utilizadas as seguintes tecnologias no desenvolvimento do trabalho:

a) Eclipse: plataforma de desenvolvimento que se utiliza da linguagem de programação Java;
b) J2ME: versão da linguagem de programação Java para dispositivos móveis;
c) Sun Java Wireless Toolkit: pacote de desenvolvimento que utiliza a J2ME e ainda integra um emulador de aparelhos celulares;
d) M3G: API gráfica para dispositivos móveis utilizada para renderização de objetos 3D;
e) EclipseME: plugin utilizado para unir o Sun Java Wirelles Toolkit à plataforma Eclipse. Através dele pode-se ter bastante agilidade na execução dos testes.
f) AC3D: aplicativo gráfico utilizado para criação de modelos 3D. Este apresenta suporte a importação e exportação de arquivos MD2.

3.3.2 Implementação do código fonte

A partir da especificação das classes tem-se início ao desenvolvimento. Para um melhor entendimento do desenvolvimento do projeto este foi dividido nos seguintes subitens: leitura do arquivo binário, armazenamento da estrutura do modelo MD2, vetor normal, visualização do modelo, adicionando o carregador de MD2 a M3GE, testes durante a implementação e a implementação do jogo exemplo.

3.3.2.1 Leitura do arquivo binário

Para a leitura de um arquivo binário optou-se por utilizar a classe `DataInputStream` que implementa a classe `DataInput` da J2ME pelo fato dela já disponibilizar diversos métodos de retorno nos tipos de dados Java. Porém, quando é feita a leitura de mais de um byte, esta classe acaba invertendo sua seqüência. Para contornar este problema é utilizada a classe `MD2LittleEndianDataInputStream` que sobrescreve os métodos da classe `DataInputStream` fazendo a mudança na seqüência dos *bytes* quando estes são dois ou mais. Um exemplo desta conversão pode ser analisado no código do Quadro 2 onde é feita a leitura de quatro *bytes*, após isso é invertida sua seqüência 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);
}
```

Quadro 2 – Método *readInt()* da classe `MD2LittleEndianDataInputStream`

3.3.2.2 Armazenamento da estrutura do modelo MD2

Para se armazenar a estrutura do modelo MD2 em memória após sua leitura do arquivo são utilizadas as classes do pacote *datatypes*. Estas classes definem a estrutura completa da parte dos dados do arquivo MD2. São elas:

a) `md2Vect3`: armazena três *floats*, *x*, *y* e *z*, que representam as coordenados do espaço cartesiano tridimensional;

b) `md2Skin`: armazena uma lista de 64 *chars*, *name*, que representa o nome da textura do modelo;

c) `md2TextCoord`: armazena dois *floats*, *s* e *t*, que representam as coordenadas de textura;

d) `Md2Triangle`: armazena duas listas de três *shorts* cada, *vertex*, que representa os índices dos vértices do triangulo e, *st*, que representa os índices das coordenadas de textura;

e) `Md2Vertex`: armazena uma lista com três *floats*, *v*, que representa um vértice através das coordenadas do cartesiano, e armazena ainda um *short*, *normalIndex*, que representa um índice para o vetor normal;

f) `Md2Frame`: armazena dois *md2Vect3*, *scale*, que representa o vetor do fator de escala e, *translate*, que representa vetor de translação. Armazena também uma lista de 16 *bytes*, *name*, que representa o nome do quadro, ainda uma lista de *Md2Vertex*, *verts*, que representa uma lista dos vértices;

g) `Md2GlCmd`: armazena dois *floats*, *s* e *t*, que representam as coordenadas de textura, e armazena um *int*, *index*, que representa um índice para o vértice. Esta estrutura é utilizada para a renderização com primitivas OpenGL.

O arquivo MD2 é dividido em duas partes, cabeçalho e dados, desta forma duas classes foram criadas para a manipulação de cada uma dessas partes, a classe `md2Header` e a

`md2Data`.

A classe `Md2Header` é responsável pela leitura do cabeçalho. Esta classe é a primeira a fazer a manipulação do arquivo MD2. Para isso é utilizado o método *read()* conforme ilustrado no Quadro 3. Este método recebe como parâmetro o arquivo já disponível em memória e começa a leitura dos bytes. Antes de qualquer manipulação dos dados é utilizada a classe `MD2LittleEndianDataInputStream` para a manipulação da leitura dos tipos de dados. Esta classe é apresentada na sessão 3.3.2.1. Após isso é feita uma leitura seqüencial de tipos de dados inteiros e armazenadas em cada um dos atributos pertencentes à classe. Para essa leitura é utilizado o método *readInt()* pertencente a classe `MD2LittleEndianDataInputStream`. Cada um desses atributos tem sua importância para o restante do processo de leitura, por isso cada um deles recebe ainda um método que retorne seus valores. Esses métodos são chamados de *geters*, muito comum em orientação a objetos. Por exemplo, o método *getMagic()* retorna o valor do atributo *magic*.

```
private void read(DataInputStream tin) throws IOException {
    MD2LittleEndianDataInputStream in = new MD2LittleEndianDataInputStream(tin);
    magic = in.readInt();
    version = in.readInt();
    skinWidth = in.readInt();
    skinHeight = in.readInt();
    frameSize = in.readInt();
    numSkins = in.readInt();
    numVertices = in.readInt();
    numTexcoords = in.readInt();
    numTriangles = in.readInt();
    numGlCommands = in.readInt();
    numFrames = in.readInt();
    offsetSkins = in.readInt();
    offsetTexcoords = in.readInt();
    offsetTriangles = in.readInt();
    offsetFrames = in.readInt();
    offsetGlCommands = in.readInt();
    offsetEnd = in.readInt();
}
```

Quadro 3 – Método *read()* da classe `md2Header`

A classe `md2Data` é responsável por fazer a leitura de todo o restante dos dados presentes no arquivo MD2. Igualmente à classe `md2Header`, ela possui um método chamado *read()* que traz o arquivo MD2 disponível em memória conforme apresentado no Quadro 4. Porém traz ainda como outro parâmetro os dados lidos anteriormente pela classe `md2Header` que serão utilizados para a leitura do restante dos dados.

```
private void read(DataInputStream tin, Md2Header md2Header) throws IOException {
    MD2LittleEndianDataInputStream in = new MD2LittleEndianDataInputStream(tin);
    readSkins(in, md2Header);
    tin.reset();
    readTextCoords(in, md2Header);
    tin.reset();
    readTriangles(in, md2Header);
    tin.reset();
    readGlCommands(in, md2Header);
    tin.reset();
    readFrames(in, md2Header);
}
```

Quadro 4 - Método *read()* da classe `md2Data`

O método *read()* primeiramente converte o parâmetro de entrada tin para o tipo `MD2LittleEndianDataInputStream` e em seguida começa a leitura dos dados. Para isto este método faz a chamada a mais cinco métodos: *readSkins()*, *readTextCoords()*, *readTriangles()*, *readGlCommands()* e *readFrames()*. Estes métodos estão intercalados por uma função que reinicia a leitura a partir do primeiro byte. Isto acontece, pois o parâmetro *md2Header* é quem informa em qual posição do arquivo está o conteúdo a ser lido. Cada método traz como retorno uma lista do conteúdo lido em um tipo de dados criado para o tratamento de arquivos MD2 que são apresentados na sessão 3.3.2.2. Exceto o método *readGlCommands()*, que traz um lista de tipos inteiros própria da linguagem Java.

O método *readSkins()* apresentado no Quadro 5 é responsável por montar uma lista do tipo `Md2Skin` onde são armazenados os nomes dos *skins* pertencentes ao modelo MD2. Cabe ressaltar que apesar de conter uma lista de *skins*, somente um deles é utilizado de cada vez na visualização dos modelos.

```
private void readSkins(MD2LittleEndianDataInputStream in, Md2Header md2Header) throws
IOException {
    Md2Skin[] listSkins_Temp = new Md2Skin[md2Header.getNumSkins()];
    in.skipBytes(md2Header.getOffsetSkins());
    for (int i=0; i < md2Header.getNumSkins(); i++){
        byte[] buffer = new byte[64];
        in.readFully(buffer);
        char[] name = new char[64];
        for (int j=0; j<64; j++){
            name[j] = (char)buffer[j];
        }
        Md2Skin skin = new Md2Skin(name);
        listSkins_Temp[i]= skin;
    }
    this.listSkins = listSkins_Temp;
}
```

Quadro 5 – Método *readSkins()* da classe `Md2Data`

Antes de iniciar a leitura é necessário apontar no arquivo binário o local onde estão armazenados os *skins*. Para isso, é utilizada a função *skipBytes()* da classe `MD2LittleEndianDataInputStream` que recebe como parâmetro a posição dos *skins* no arquivo através do cabeçalho do MD2. Esta função descarta a quantidade de bytes recebidos como parâmetro e deixa o próximo byte da seqüência pronto para a leitura, sendo este o início da lista de *skins*.

Para a leitura do nome de cada *skin* é feita uma leitura de 64 bytes através da função *readFully()* pertencente a classe `MD2LittleEndianDataInputStream`. Isto é feito dentro de um ciclo de leitura que tem como contador final a quantidade de *skins* extraídos do cabeçalho vindo como parâmetro de entrada da função *readSkins()*.

Ao final da leitura dos *skins* seus dados são armazenados dentro do atributo *listSkins* pertencente a classe `md2Data`.

O método *readTextCoords()* conforme apresentado no Quadro 6 é responsável por montar uma lista do tipo `Md2TextCoord` que armazena as coordenadas das texturas. Conforme o método *readSkins()* também é utilizada a função *skipBytes()* para apontar ao local do arquivo MD2 onde estão armazenadas as coordenadas.

```
private void readTextCoords(MD2LittleEndianDataInputStream in, Md2Header md2Header) throws
IOException {
    Md2TextCoord[] listTextCoords_Temp = new Md2TextCoord[md2Header.getNumTexcoords()];
    in.skipBytes(md2Header.getOffsetTexcoords());
    for (int i=0; i < md2Header.getNumTexcoords(); i++){
        short s = in.readShort();
        short t = in.readShort();
        Md2TextCoord textCoord = new Md2TextCoord(s, t);
        listTextCoords_Temp[i] = textCoord;
    }
    this.listTextCoords = listTextCoords_Temp;
}
```

Quadro 6 – Método *readTextCoords()* da classe `Md2Data`

Após isso, é feito um ciclo de leitura que tem como contador final a quantidade de coordenadas de texturas retiradas do cabeçalho. Dentro deste ciclo são feitas as leituras das coordenadas *s* e *t* através de função *readShort()* pertencente a classe `MD2LittleEndianDataInputStream`. Cada leitura desta função lê dois *bytes*.

Ao final da leitura das coordenadas de textura seus dados são armazenados dentro do método *listTextCoords* da classe `md2Data`.

O método *readTriangles()*, conforme apresentado no Quadro 7, é responsável por montar uma lista do tipo `Md2Triangle` que armazena as coordenadas dos triângulos do modelo MD2. Conforme o método *readSkins()* também é utilizada a função *skipBytes()* para apontar o local do arquivo onde estão armazenadas essas coordenadas.

```
private void readTriangles(MD2LittleEndianDataInputStream in, Md2Header md2Header) throws
IOException {
    Md2Triangle[] listTriangles_Temp = new Md2Triangle[md2Header.getNumTriangles()];
    in.skipBytes(md2Header.getOffsetTriangles());
    for (int i=0; i < md2Header.getNumTriangles(); i++){
        short[] vertex = new short[3];
        for (int j=0; j < 3; j++){
            vertex[j] = in.readShort();
        }
        short[] st = new short[3];;
        for (int j=0; j < 3; j++){
            st[j] = in.readShort();
        }
        Md2Triangle md2Triangle = new Md2Triangle(vertex, st);
        listTriangles_Temp[i] = md2Triangle;
    }
    this.listTriangles = listTriangles_Temp;
}
```

Quadro 7 – Método *readTextCoords()* da classe `Md2Data`

Para a leitura dos triângulos é feita uma estrutura de repetição tendo como contador final a quantidade de triângulos retiradas do cabeçalho. Durante este ciclo são feitas a leitura dos três índices dos vértices do triângulo e dos três índices das coordenadas de texturas através da função *readShort()* da classe `MD2LittleEndianDataInputStream`.

Ao final da leitura das coordenadas dos triângulos seus dados são armazenados no atributo *listTriangles* da classe `Md2Data`.

O método *readGlCommands()* conforme apresentado no Quadro 8 é responsável por montar uma lista do tipo `Md2GlCmd` que armazena comandos OpenGL. Conforme o método *readSkins()* também é utilizada a função *skipBytes()* para apontar o local do arquivo onde estão armazenadas os comandos de OpenGL. Este método foi utilizado durante o processo de desenvolvimento do projeto, porém ao final descobriu-se que é um método que pode ser utilizado opcionalmente. Por isso ele não é usado durante o resto do trabalho, mas é importante sua apresentação para compreensão completa do conteúdo do arquivo MD2.

```
private void readGlCommands(MD2LittleEndianDataInputStream in, Md2Header md2Header) throws
IOException {
int[] listGlCommands_Temp = new int[md2Header.getNumGlCommands()];
in.skipBytes(md2Header.getOffsetGlCommands());
    for (int i=0; i < md2Header.getNumGlCommands(); i++){
        listGlCommands_Temp[i] = in.readInt();
    }
this.listGlCommands = listGlCommands_Temp;
}
```

Quadro 8 – Método *readGlCommands()* da classe `Md2Data`

Para a leitura dos comandos OpenGL é feito um ciclo de leitura onde tem-se como contador final a quantidade de comandos OpenGL extraídos do cabeçalho. Durante este ciclo é feita a leitura de cada comando através da função readInt() da classe `MD2LittleEndianDataInputStream`. Cada leitura desta função lê quatro *bytes*, ou seja, um tipo inteiro.

Ao final da leitura dos comandos OpenGL seus dados são armazenados no atributo *listGlCommands* da classe `Md2Data`.

O método *readFrames()* conforme apresentado no Quadro 9 é responsável pela leitura dos quadros referentes ao modelo MD2. Conforme o método *readSkins()* também é utilizada a função *skipBytes()* para apontar o local do arquivo onde estão armazenadas os quadros do arquivo MD2.

Para a leitura dos quadros do modelo MD2 é feito um ciclo de leitura que tem como contador final a quantidade de quadros extraídos do cabeçalho do arquivo MD2. Durante esse ciclo é feita uma série de leituras. A primeira é a leitura do vetor de escala através de três funções *readFloat()*, que lê quatro bytes utilizado para armazenar cada coordenada do vetor. Em seguida é lido o vetor de translação através de três funções *readFloat()*, que armazenam cada coordenada do vetor. Logo após é lido o nome do quadro através de um ciclo para a leitura de 16 bytes utilizando a função *readByte()* que lê um byte de cada vez. Ambas as funções de leitura pertencem à classe `MD2LittleEndianDataInputStream`.

Ainda dentro do ciclo anterior é criado um novo ciclo que tem como contador final a quantidade de vértices pertencentes ao quadro que é extraído do cabeçalho. Durante esse ciclo é feita a leitura de cada coordenada do vértice através da função *readUnsignedByte()* que lê um byte sem sinal. Ao mesmo tempo em que é feita a leitura, é descompactada a coordenada do vértice através de sua multiplicação pelo vetor de escala e de sua soma ao vetor de translação lidos anteriormente. Ainda durante o segundo ciclo é feita a leitura do índice do vetor normal através da função read() pertencente à classe `MD2LittleEndianDataInputStream`. Esta função lê dois *bytes*.

```
private void readFrames(MD2LittleEndianDataInputStream in, Md2Header md2Header) throws
IOException {
    Md2Frame[] listFrames_Temp = new Md2Frame[md2Header.getNumFrames()];
    in.skipBytes(md2Header.getOffsetFrames());
    for (int i=0; i < /*md2Header.getNumFrames()*/1; i++){
        Md2Vect3 md2Vect3_Scale = new Md2Vect3(in.readFloat(), in.readFloat(),
in.readFloat());
        Md2Vect3 md2Vect3_translate = new Md2Vect3(in.readFloat(), in.readFloat(),
in.readFloat());
        byte[] name = new byte[16];
        for (int j=0; j < 16; j++){
            name[j] = in.readByte();
        }
        Md2Vertex[] listMd2Vertex = new Md2Vertex[md2Header.getNumVertices()];
        for (int k=0; k < md2Header.getNumVertices(); k++){
            float[] v = new float[3];
            v[0] = (md2Vect3_Scale.x * in.readUnsignedByte()) + md2Vect3_translate.x;
            v[2] = (md2Vect3_Scale.z * in.readUnsignedByte()) + md2Vect3_translate.z;
            short normalIndex;
            normalIndex = (short)in.read();
            Md2Vertex md2Vertex = new Md2Vertex(v, normalIndex);
            listMd2Vertex[k] = md2Vertex;
        }
        Md2Frame md2Frame = new Md2Frame(md2Vect3_Scale, md2Vect3_translate, name,
listMd2Vertex);
    }
    this.listFrames = listFrames_Temp;
}
```

Quadro 9 – Método *readFrames()* da classe `Md2Data`

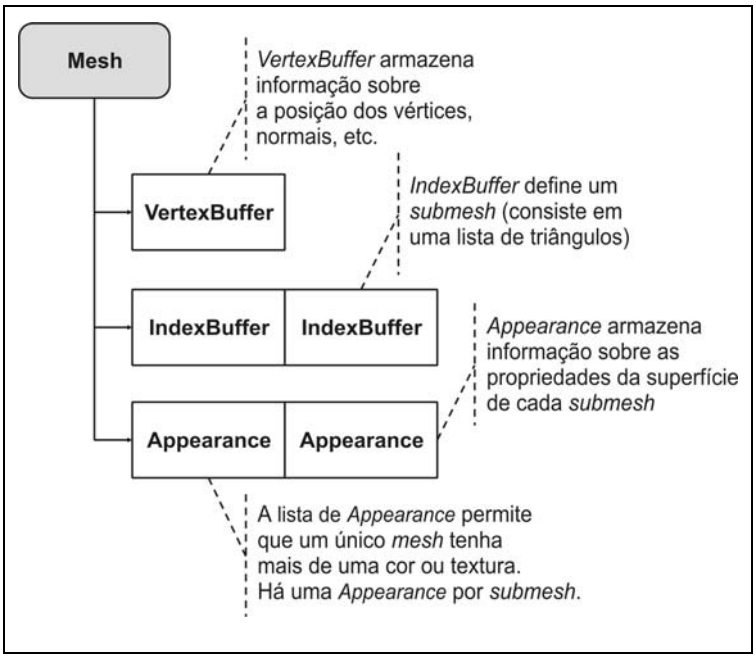
Ao final da leitura de todos os quadros seus dados são armazenados dentro do atributo *listFrames* da classe `Md2Data`. A função *readFrames()* foi modificada para a leitura do primeiro quadro apenas. Isto ocorre pela falta de memória que impossibilita a leitura de todos os quadros do modelo.

3.3.2.3 Vetor normal

Para a utilização do vetor normal nos vértices do modelo MD2 basta buscá-los numa lista de vetores já calculados. Isto ocorre, pois o arquivo MD2 traz apenas uma lista de índices para esses vetores. Desta forma foi criada a classe `Md2Normal` que armazena a lista de vetores normais que tem como função receber o índice do arquivo para a localização deste vetor. Estes vetores estão armazenados na classe através de uma lista estática de *floats*. Cada três componentes da lista representam um vetor.

3.3.2.4 Visualização do modelo

O motor M3GE já tem implementado no gerenciador gráfico funções capazes de desenhar os objetos em cena. Então para o PNJ ser desenhado, basta anexá-lo ao grafo de cena, pois todos os objetos anexados a ele são desenhados pelo motor. Porém, para isto ocorrer o modelo deve ter uma estrutura conforme 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.



Fonte: adaptado de Nokia Corporation (2003).

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, conforme mostrado no trecho presente no Quadro 10.

```
public Md2Model(String nameModel, String nameTexture, int frame) {
    try {
        readFile(nameModel);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        System.out.println("Error loading model " + nameModel);
        e.printStackTrace();
    }
    short[] VERTEX_POSITIONS = VertexPositions(frame);
    int[] TRIANGLE_INDICES = TriangleIndices();
    int[] TRIANGLE_LENGTHS = TriangleLengths(TRIANGLE_INDICES.length);
    short[] VERTEX_TEXTURE_COORDINATES = TextCoords();
    short[] VERTEX_NORMALS = Normals(frame);
                    . . .
}
```

Quadro 10 – Método construtor da classe `Md2Model` parte 1

O método *readFile()* da classe `Md2Model` tem como objetivo ler o arquivo MD2 através das classes `Md2Header` e `Md2Data` e armazenar seu conteúdo em dois atributos, o *md2Header*

e o *md2Data*. Deve sempre primeiro utilizar a classe `md2Header`, pois ela instancia um parâmetro necessário para a classe `Md2Data`. Este método ainda verifica se o arquivo MD2 é válido, analisando o atributo *magic* e *version* da classe `Md2Header`. O *magic* é um número para identificar o tipo do arquivo. Ele deve ser igual a 844121161 ou a string “IDP2”. E o atributo *version* serve para identificar a versão do arquivo que neste caso deve ter valor igual a 8.

Tendo os atributos *md2Header* e *md2Data* conteúdos válidos dá-se seqüência a montagem da estrutura. Para isso cinco métodos são chamados:

a) *VertexPositions()*: retorna uma lista contendo as posições dos vetores;
b) *TriangleIndices()*: retorna uma lista contendo os índices dos vértices que formam cada triângulo. Para a implementação desta função teve-se que inverter a lista completa dos índices, pois estava sendo visualizada a parte interna do modelo;
c) *TriangleLengths()*: retorna uma lista contendo a quantidade de vértices de cada triângulo;
d) *TextCoords()*: retorna um lista com as coordenadas de textura;
e) *Normals()*: retorna uma lista com os vetores normais.

Para um retorno correto com relação aos tipos de dados teve-se que se implementar uma alteração de ponto flutuante recebido do arquivo MD2 para ponto fixo necessário para se desenhar o modelo na M3GE. Para isso criou-se a constate *SCALE* no valor de 1000 que serve como multiplicador. Foi escolhido este valor, pois os vértices estavam próximos da escala um. Por exemplo, dois vértices de valores 1.05 e 1.45, se eles fossem apenas arredondados eles ficariam com o mesmo valor, 1.00, mas se multiplicar por 1000, eles terão valores de 1050 e 1450. Para se fazer a conversão basta multiplicar o valor pelo *SCALE* e depois fazer seu arredondamento. Com isso o tamanho do modelo aumentou em mil vezes, mas isto pode ser contornado reduzindo seu tamanho antes de desenhá-lo através das funções da M3G. Isto ocorre, pois para se montar o nodo *Mesh*, na lista da posição dos vértices existe um parâmetro na função *setPositions* que define um fator de escala do objeto 3D a ser desenhado. E esse valor foi definido de tal forma que o modelo seje reduzido em mil vezes enquanto se monta a lista de vértices.

Ainda dentro do construtor da classe `Md2Model` é criada a estrutura do nodo *Mesh* a partir do conteúdo das listas retornadas dos métodos apresentados anteriormente. Estas listas já contêm a estrutura pronta para ser usada na construção do nodo *Mesh*. Após a criação do nodo *Mesh* ele é 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`.

Conforme apresentado no Quadro 11 para a criação de um *VertexBuffer* primeiramente deve-se instanciá-lo. Em seguida para a criação das listas das posições de vértices, dos vértices normais e das coordenadas de textura são usadas uma lista do tipo *VertexArray*. Esta lista tem em seu construtor três parâmetros, sendo eles: número de vértices neste *VertexArray*, número de componentes por vetor e número de bytes por componente. Para se incluir conteúdo a esta lista de vetores existe a função *set()* que também tem 3 parâmetro sendo eles: índice do primeiro vetor, número de vetores e a lista de valores. Após o *VertexArray* estar preenchido com os valores este pode ser adicionado ao *VertexBuffer*. Para isto existe um método para cada tipo de lista: *setPositions()* para adicionar uma lista de posições de vetores, *setNormals()* para adicionar uma lista com os vetores normais e *setTextCoords()* para adicionar uma lista com as coordenadas de textura.

```
public Md2Model(String nameModel, String nameTexture, int frame) {
                . . .
    // Create vertex data.
    VertexBuffer modelVertexData = new VertexBuffer();

    VertexArray vertexPositions =
    new VertexArray(VERTEX_POSITIONS.length/3, 3, 2);
    vertexPositions.set(0, VERTEX_POSITIONS.length/3, VERTEX_POSITIONS);
    modelVertexData.setPositions(vertexPositions, 0.00001f, null);

    VertexArray vertexNormals =
    new VertexArray(VERTEX_NORMALS.length/3, 3, 2);
    vertexNormals.set(0, VERTEX_NORMALS.length/3, VERTEX_NORMALS);
    modelVertexData.setNormals(vertexNormals);

    VertexArray vertexTextureCoordinates =
    new VertexArray(VERTEX_TEXTURE_COORDINATES.length/2, 2, 2);
    vertexTextureCoordinates.set(0,
    VERTEX_TEXTURE_COORDINATES.length/2, VERTEX_TEXTURE_COORDINATES);
    modelVertexData.setTexCoords(0, vertexTextureCoordinates, 0.001f, null);

    // Create the triangles that define the model; the indices point to
    // vertices in VERTEX_POSITIONS.
    TriangleStripArray modelTriangles = new TriangleStripArray(TRIANGLE_INDICES,
    TRIANGLE_LENGTHS);
                . . .
}
```

Quadro 11 – Método construtor da classe `Md2Model` parte 2

Ainda com análise ao código fonte do Quadro 11 tem-se que criar os triângulos que definem o modelo. Para isso é utilizada o tipo *TriangleStripArray* que define uma lista de *triangle strip*. No *triangle strip*, os três primeiros índices do vetor definem o primeiro triângulo. Cada índice subseqüente juntamente com seus dois antecessores define um novo triângulo. Por exemplo, a lista S = (2, 0, 1, 4) define dois triângulos: (2, 0, 1) e (0, 1, 4). Desta forma é criado o *TriangleStripArray* passando-se seus dois parâmetros que são: a lista com o índice dos triângulos e a lista com a quantidade de vértices por triângulo. Normalmente os triângulos são representados por três vértices, conforme a própria definição de triângulo diz, porém este número pode ser alterado para a leitura de polígonos com mais de três vértices.

Para finalizar a explicação do método construtor da classe `Md2Model` é apresentado o Quadro 12. Neste quadro esta implementada a criação de uma *Appearance* utilizada para armazenar as texturas. Para isso primeiramente é instanciado um objeto de *Appearance*. Em seguida é feita a leitura da textura e convertida para o tipo de dado *Imagem2D*. Após isso é instanciado um objeto de *Texture2D* que recebe como parâmetro a textura lida anteriormente. Para adicionar a textura lida à *Appearance* é utilizado o método *setTexture()* que tem dois parâmetros: o primeiro recebe um índice referente a textura a ser usada e o outro o objeto de *Texture2D*.

```
public Md2Model(String nameModel, String nameTexture, int frame) {
                . . .
    // Define an appearance object.
    Appearance modelAppearance = new Appearance();

    try
    {
        // Load image for texture and assign it to the appearance. The
        // default values are: WRAP_REPEAT, FILTER_BASE_LEVEL/
        // FILTER_NEAREST, and FUNC_MODULATE.
        Image2D image2D = ImageUtils.loadImage(nameTexture);
        Texture2D modelTexture = new Texture2D(image2D);

        // Index 0 is used because we have only one texture.
        modelAppearance.setTexture(0, modelTexture);
    }
    catch (Exception e)
    {
        System.out.println("Error loading image " + nameTexture);
        e.printStackTrace();
    }

    model = new Mesh(modelVertexData, modelTriangles, modelAppearance);
}
```

Quadro 12 – Método construtor da classe `Md2Model` parte 3

O método construtor da classe `Md2Model` finaliza com a criação e retorno do objeto *Mesh* utilizando-se os três parâmetros para sua criação: *VertexBuffer*, *IndexBuffer* e *Appearance*.

A classe `md2Model` ainda apresenta o método *getModel()* que tem por finalidade retornar o atributo que contém o modelo MD2 na estrutura de um nodo *Mesh*. Também existe um método *Skin()*, que retorna a lista com os nomes de todos os *skins* pertencentes ao modelo.

3.3.2.5 Adicionando o PNJ ao jogo

O método *addNPC()* 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 do instanciamento 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 que a M3GE implementa, 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. O nodo *World* é um atributo da classe `EngineCanvas`.

```
public Node addNPC(String md2ModelFile, String md2TextureFile, int frame, String md2Name){
    Md2Model md2Model = new Md2Model(md2ModelFile, md2TextureFile, frame);
    Mesh model = md2Model.getModel();
    Object3DInfo object3DInfo = new Object3DInfo();
    object3DInfo.name = md2Name;
    model.setUserObject(object3DInfo);
    Group group = new Group();
    group.addChild(model);
    world.addChild(group);
    return model;
}
```

Quadro 13 – Método *addNPC()* da classe `EngineCanvas`

3.3.3 Testes durante a implementação

Durante a implementação foi possível realizar diversos testes para cada evolução do projeto. Isto ficou facilitado pelo fato da plataforma de desenvolvimento Eclipse estar diretamente ligada ao emulador de celular presente na Sun Java Wireless Toolkit. Estes dois se interligam através de um *plugin*, o EclipseME.

Os códigos fontes utilizados para os testes foram baseados no artigo Höfele (2005). Este artigo disponibiliza diversos exemplos de como utilizar a M3G através do desenho de um cubo. O código desse artigo foi sendo alterado de tal forma, que ao invés de desenhar o cubo, ele pudesse desenhar o modelo MD2 lido.

Primeiramente, para desenhar o modelo foram utilizados apenas seus vértices através da seqüência na lista de triângulos, obtendo-se a aparência do personagem conforme demonstrado no modelo (a) da Figura 10. Apesar de se tratar de um modelo 3D esta visualização dá a impressão de ser em 2D.

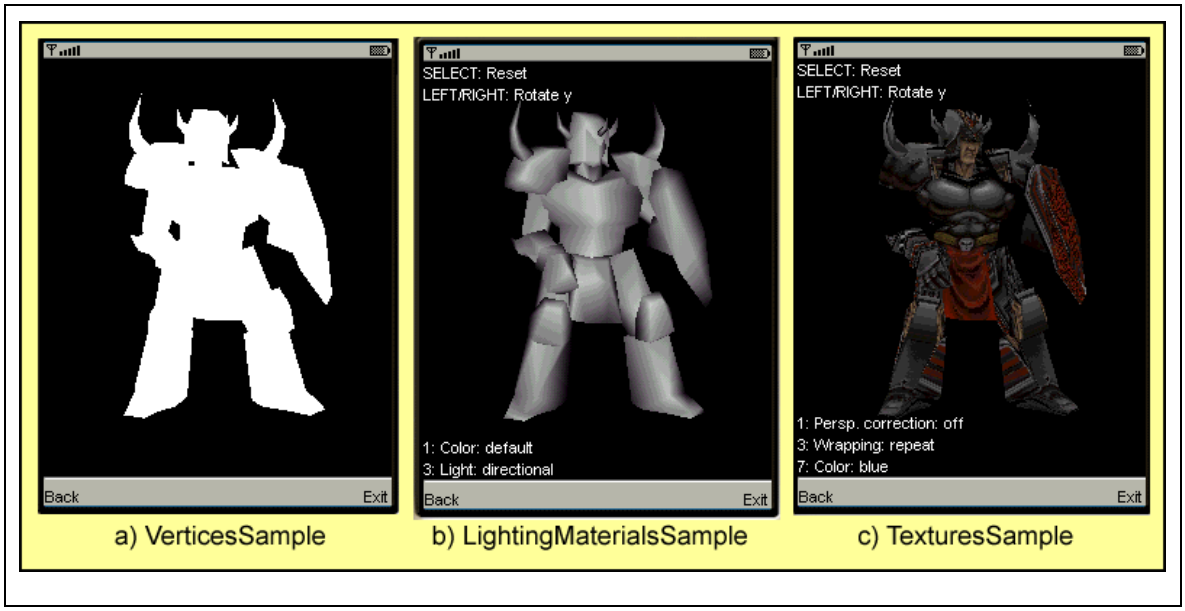


Figura 10 – Personagem Knight em 3 diferentes visualizações

Posteriormente foi adicionada a lista dos vértices normais do modelo. Deste modo pode-se utilizar luzes que eram capazes de refletir no personagem, podendo-se visualizar suas formas 3D conforme modelo (b) da Figura 10.

Por último o personagem (c) representa a adição de textura ao modelo. Nesta visualização pode-se ver todos os detalhes do personagem através do *skin* escolhido para ser utilizado como textura. A textura utilizada no modelo é um arquivo do tipo PNG, apesar de por característica utilizar-se arquivos PCX com modelos MD2. Desta forma pode-se obter agilidade no desenvolvimento pelo fato da M3G trabalhar com o arquivo PNG e não com o PCX.

3.3.4 Implementação do jogo exemplo

Para finalizar os testes e verificar a viabilidade do projeto foi implementado um protótipo de jogo. Este protótipo é baseado no antigo jogo exemplo da M3GE, porém nesta versão foram adicionados três personagens a cena, representados no Quadro 14.

Nome do Personagem	Arquivo MD2	Textura
Knight	knight.md2	knight_FlipVertical.png
Centaur	centaur.md2	centaur_FlipVertical.png
Knight Evil	knight.md2	knight_evil.png

Quadro 14 – Personagens adicionados ao jogo exemplo

Os personagens são adicionados ao jogo na mesma ordem que são apresentados na

tabela 1. Os personagens Knight e Knight Evil são carregados a partir do mesmo arquivo MD2 porém são utilizados *skins* diferentes, isto é, para cada modelo são utilizados texturas diferentes.

Toda funcionalidade que já estava presente no jogo 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 até utilizar-se a funcionalidade de tiro presente no jogo, onde quando se acerta um objeto da cena aparece o nome deste no visor, indicando que o mesmo foi atingido. Uma visualização geral do jogo pode ser vista na Figura 11.

Figura 11 – Jogo exemplo com personagens MD2

Para o desenvolvimento do jogo foi utilizado o código fonte demonstrado no Quadro 15. Este código primeiramente demonstra a necessidade da criação de um *canvas* onde deverá ser visualizado o jogo. Este *canvas* pertence à classe `EngineCanvas` da M3GE. Para a criação do *canvas* é utilizado o construtor dessa classe onde são passados três parâmetros: local e nome do arquivo onde estão armazenados todos os objetos da cena, local e nome do arquivo de propriedades onde estão armazenadas as configurações do jogo e um *flag* que indica o tipo do primeiro arquivo, podendo ser MBJ ou OBJ. Em seguida é instanciado um objeto da classe `GameControl` onde são passados dois parâmetros: a largura e altura do *canvas*. É esta classe

que trata o controle do usuário com o jogo. Após ela ser instanciada é adicionada ao *canvas*. Após isto se tem um cenário, um jogador e câmeras. Ficam faltando apenas os adversários que são inseridos posteriormente através do método *addNPC()* da classe `EngineCanvas`. Neste exemplo são utilizadas três vezes este método com a finalidade de adicionar três personagens. Após a criação do modelo tem-se como retorno o nodo dele, e através deste nodo podem-se fazer as transformações de rotação, translação e escala do personagem conforme a necessidade do jogo. Neste exemplo cada personagem é adicionado em uma posição diferente do cenário conforme apresentado na Figura 11.

```
public class Md2LoaderInGame extends MIDlet {
    EngineCanvas canvas;

    public void startApp() {
    Display d = Display.getDisplay(this);

    try {
        canvas = new EngineCanvas("/org/m3ge/examples/md2LoaderInGame/map5.mbj",
"/org/m3ge/examples/md2LoaderInGame/m3ge.properties", true);
        GameControl gameControl = new GameControl(canvas.getHeight(), canvas.getWidth());

        canvas.setUpdateListener(gameControl);

        d.setCurrent(canvas);

        //add md2Model knight
        Node model1 = canvas.addNPC("/knight.md2", "/knight_FlipVertical.png", 0, "Knight");
        Transform transform = new Transform();
        transform.postRotate(-90.0f, 1.0f, 0.0f, 0.0f);
        transform.postTranslate(0.0f, 0.5f, 0.1f);
        transform.postScale(0.5f, 0.5f, 0.5f);
        model1.setTransform(transform);

        //add md2Model centaur
        Node model2 = canvas.addNPC("/centaur.md2", "/centaur_FlipVertical.png", 0,
"Centaur");
        Transform transform2 = new Transform();
        transform2.postRotate(-90.0f, 1.0f, 0.0f, 0.0f);
        transform2.postTranslate(0.0f, -0.2f, 0.08f);
        transform2.postScale(0.4f, 0.4f, 0.4f);
        model2.setTransform(transform2);

        //add md2Model knight evil
        Node model3 = canvas.addNPC("/knight.md2", "/knight_evil.png", 0, "Knight Evil");
        Transform transform3 = new Transform();
        transform3.postRotate(-90.0f, 1.0f, 0.0f, 0.0f);
        transform3.postTranslate(0.0f, 0.3f, 0.1f);
        transform3.postScale(0.5f, 0.5f, 0.5f);
        model3.setTransform(transform3);

        ExitCommand.createExitCommand("Sair", this, canvas);
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Quadro 15 – Implementação do jogo exemplo

3.4 RESULTADOS E DISCUSSÃO

Após a conclusão do projeto, foi possível utilizar o carregador de MD2 para adicionar personagens aos jogos da M3GE. Isto teve grande utilidade, pois a M3GE não possuia capacidade de inserir novos personagens após a criação do *canvas*. Era possível apenas a inserção ou alteração do jogador.

Uma deficiência constatada foi a impossibilidade de carregamento de mais um quadro do MD2, pelo fato da falta de memória que ocorria na execução no emulador. Desta forma não foi possível a implementação de animação dos personagens. Até mesmo carregando um arquivo menos complexo fez com que surgisse uma exceção de falta de memória.

Para visualizar o problema da falta de memória foi utilizado o monitor de memória que existe no Sun Java Wireless Toolkit. Constatou-se que quando ocorre a exceção "*Out of memory*" significa que as classes instanciadas ultrapassavam a memória de execução do programa. Este emulador tem 500000 *bytes* de memória *heap*. E isto é o que normalmente existe nos celulares atuais no mercado. Porém alguns celulares mais novos já estão vindo com um *heap* de 1500000 *bytes*. Desta forma percebeu-se que o problema não está no armazenamento do modelo e sim na sua complexa leitura que demanda a utilização de diversas classes até que se tenha o nodo *Mesh* pronto para ser desenhado. Na verdade para as leituras dos personagens foram utilizados mais de 500000 bytes, porém antes que o *heap* chegasse ao seu limite alguns objetos instanciados que não iriam ser mais utilizados eram descartados pelo *Gabage Colector* do Java, que é o responsável por eliminar os objetos da memória.

Ainda destaca-se a demora na leitura e visualização do modelo que leva em torno de trinta segundos para aparecer na tela do celular. Tendo um tempo total para a carga completa do jogo 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. Ainda mais que há a necessidade de apresentar mais de um personagem ao mesmo tempo em cena. A princípio, isto poderia ser considerado uma grave deficiência, mas espera-se que a constante evolução dos recursos do processamento e memória dos celulares deve naturalmente resolver estes problemas. Isto é uma evolução alcançável dos celulares, visto que se têm hoje aparelhos como os *smartfones*, PDA's, entre outros, com grandes recursos de hardware.

Atualmente a motor de jogos wGEN possui mais funcionalidades implementas 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 só trabalhar com objetos 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.

O *framework* mOGE é implementado em um nível mais baixo, acessando diretamente a biblioteca OpenGL ES, diferentemente da solução adotada pelo 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.

4 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 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 renderização dos modelos. Acredita-se que isto pode trazer bons resultados, e justifica-se devido a 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 emulados, não foi possível visualizar animações dos modelos MD2.

As ferramentas e tecnologias utilizadas facilitaram bastante o desenvolvimento do trabalho, principalmente na questão dos testes. Talvez a escolha de Java tenha dificultado a obtenção de um bom desempenho do projeto, pelo fato ser uma linguagem interpretada. Porém, o foco principal em tecnologia móvel é criar algo que além de rápido seja portável visto a grande quantidade de modelos de celulares que existem, e portabilidade é uma das grandes vantagens da utilização da linguagem Java neste contexto.

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 execução. Desta forma, abre-se a possibilidade de desenvolvimento de estudos que possibilitem novas descobertas para viabilizar a utilização de MD2 para dispositivos móveis, especialmente considerando as animações dos modelos como um requisito altamente desejável para o desenvolvimento de jogos com NPCs.

4.1 EXTENSÕES

Como sugestões para os trabalhos futuros podem-se destacar:

a) análise e verificação das técnicas utilizadas procurando reduzir o tempo de leitura do arquivo MD2;
b) utilizar algoritmos que simplifiquem e reduzam o número de vértices do arquivo MD2 criando um modelo reduzido deste arquivo, em particular, sugere-se a investigação do uso de técnicas de nível de detalhe (LOD – *Level of Detail*) em malhas de polígonos 3D;
c) utilizar os quadros do arquivo MD2 para a obtenção de animação;
d) incluir o carregador de MD2 em outro *framework*.

REFERÊNCIAS BIBLIOGRÁFICAS

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

CONITEC CORPORATION. **3D GameStudio / A6**. San Diego, 2004. Disponível em: <http://conitec.net/a4info.htm>. Acesso em: 17 abr. 2006.

CRYSTAL SPACE. **Crystal Space 3D**. [S.l.], 2004. Disponível em: <http://crystal.sourceforge.net/>. Acesso em: 17 abr. 2006.

ECLIPSE ENTERTAINMENT. **Welcome to the home of Genesis3D**. [RedMond], 2004. Disponível em: <http://www.genesis3d.com/>. Acesso em: 17 abr. 2006.

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

GOMES, Paulo C. R.; PAMPLONA, Vitor F. M3GE: um motor de jogos 3D para dispositivos móveis com suporte a Mobile 3D Graphics API. In: WORKSHOP BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL (WJOGOS), 4., 2005, São Paulo. **Anais...** São Paulo: SBC, 2005, p. 55-65.

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

HENRY, David. **MD2 file format (Quake 2's model)**. [S.l.], 2004. Disponível em: http://tfc.duke.free.fr/coding/md2-specs-en.html>. Acesso em: 24 mar. 2006.

HÖFELE, Claus. **3D graphics for Java mobile devices**, part 1: M3G's immediate mode. [S.l.], 2005. Disponível em: <http://www-128.ibm.com/developerworks/wireless/library/wi-mobile1/>. Acesso em: 17 mar. 2006.

JAVA IS DOOMED. [S.l.], [2006]. Disponível em <http://javaisdoomed.sourceforge.net/>.Acesso em: 25 out. 2006.

KHRONOS GROUP. **OpenGL ES**: overview. [San Francisco], 2004. Disponível em: <http://www.opengles.org/opengles/index.html>. Acesso em: 17 abr. 2006.

LEHMANN, Jens. **Xith3D**: contents. [S.l.], 2004. Disponível em: <http://xith.org/tutes/GettingStarted/html/contents.html>. Acesso em: 25 out. 2006.

MACÊDO JÚNIOR, Ives J. A. **MOGE – Mobile Graphics Engine**. O projeto de um motor gráfico 3D para a criação de jogos em dispositivos móveis. 2005. 75 f. Trabalho de Graduação (Bacharel em Ciências da Computação) - Centro de Informática, Universidade Federal de Pernambuco, Recife.

MD2. [S.l.], [2005]. Disponível em: <http://gpwiki.org/index.php/MD2>. Acesso em: 24 mar. 2006.

MISFIT model 3D. [S.l.], [2005]. Disponível em: <http://www.misfitcode.com/misfitmodel3d/olh_quakemd2.html>. Acesso em: 24 mar. 2006.

MÜLLER, Lucas F.; FRANTZ, Guilherme J.; SCHREIBER, Jacques N. C. Qualcomm Brew X Sun J2ME: um comparativo entre soluções para desenvolvimento de jogos em dispositivos móveis. In: SULCOMP - Congresso Sul Catarinense de Computação, 1., 2005, Criciúma. **Anais...** Criciúma: Sulcomp, 2005. p. 1-8.

NOKIA. **RI binary for JSR-184 3D graphics API for J2ME™**. [P.O.Box], 2004. Disponível em: <http://www.forum.nokia.com/main/1,6566,040,00.html?fsrParam=2-3-/main.html&fileID=5194>. Acesso em: 17 abr. 2006.

_____. **Mobile 3D graphics API specification**. Version 1.0. [S.l.], 2003. Disponível em: <www.cs.lth.se/EDA075/assignment1/jsr184-specification-1.0.pdf>. Acesso em 19 abr. 2006.

PAIVA, Fernando. O jogo das operadoras. **Teletime**, Rio de Janeiro, v. 55, não paginado, maio 2003. Disponível em <http://www.teletime.com.br/revista/55/capa.htm>. Acesso em: 05 abr. 2006.

PALUDO, Lauriana. **Um estudo sobre as tecnologias Java de desenvolvimento de aplicações móveis**. 2003. 118 f. Dissertação (Mestrado em Ciência da Computação) - Departamento de Informática e Estatística, Universidade Federal de Santa Catarina, Florianópolis.

PARALELO COMPUTAÇÃO. **Fly3D.com.br**. [Niterói], 2004. Disponível em: <http://www.fly3d.com.br/>. Acesso em: 17 abr. 2006.

PESSOA, Carlos A. C. **wGEM**: um framework de desenvolvimento de jogos para dispositivos móveis. 2001. 110 f. Dissertação (Mestrado em Ciências da Computação) - Centro de Informática, Universidade Federal de Pernambuco, Recife.

PESSOA, C. A. ; RAMALHO, G. ; BATTAIOLA, André Luiz . wGEM: um framework de desenvolvimento de jogos para dispositivos móveis. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE (SEMISH), 29., 2002, Florianópolis. **Anais...** Florianópolis: SBC, 2002. p. 225-235.

PINHEIRO, Christiano. **J2ME**: Java para os portáteis. [S.l.], 2003. Disponível em: <http://www.imasters.com.br/artigo.php?cn=1539&cc=19>. Acesso em: 17 mar. 2006.

SÁNCHEZ, Daniel; DALMAU, Crespo. **Core techniques and algorithms in game programming**. Indianapolis: New Riders, 2004.

SANTEE, André. **Programação de Jogos com C++ e DirectX**. São Paulo: Novatec Editora, 2005.

ANEXO A – Índice da Documentação do Carregador de MD2

No Quadro 16 é apresentado o índice da Documentação do Carregador de MD2 das classes implementadas para se carregar um modelo MD2 no motor de jogos M3GE. Esta documentação foi criada através de um Javadoc, e está presente no código fonte na pasta "doc".

CLASS HIERARCHY

- class java.lang.Object
  - class org.m3ge.md2loader.**Md2Data**
  - class org.m3ge.md2loader.datatypes.**Md2Frame**
  - class org.m3ge.md2loader.datatypes.**Md2GlCmd**
  - class org.m3ge.md2loader.**Md2Header**
  - class org.m3ge.md2loader.**MD2LittleEndianDataInputStream** (implements java.io.DataInput)
  - class org.m3ge.md2loader.**Md2Model**
  - class org.m3ge.md2loader.**Md2Normal**
  - class org.m3ge.md2loader.datatypes.**Md2Skin**
  - class org.m3ge.md2loader.datatypes.**Md2TextCoord**
  - class org.m3ge.md2loader.datatypes.**Md2Triangle**
  - class org.m3ge.md2loader.datatypes.**Md2Vect3**
  - class org.m3ge.md2loader.datatypes.**Md2Vertex**

Quadro 16 – Índice da documentação do carregador de MD2