**UNIVERSIDADE REGIONAL DE BLUMENAU**

**CENTRO DE CIÊNCIAS EXATAS E NATURAIS**

**CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO**

# RECONSTRUTOR DE MODELOS 3D UTILIZANDO TÉCNICAS DE NÍVEL DE DETALHAMENTO

**TIAGO PISKE**

**BLUMENAU**
**2008**

**2008/1-36**

**TIAGO PISKE**

# RECONSTRUTOR DE MODELOS 3D UTILIZANDO TÉCNICAS DE NÍVEL DE DETALHAMENTO

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 – Orientador

**BLUMENAU**
**2008**

**2008/1-36**

# RECONSTRUTOR DE MODELOS 3D UTILIZANDO TÉCNICAS DE NÍVEL DE DETALHAMENTO

Por

**TIAGO PISKE**

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 – Orientador, FURB |
| Membro: | Prof. Dalton Solano dos Reis – FURB |
| Membro: | Prof. Mauro Marcelo Mattos – FURB |

Blumenau, 10 de julho de 2008

Dedico este trabalho a minha família que nunca deixou de acreditar e meus amigos que me ajudaram diretamente na sua realização.

## AGRADECIMENTOS

Agradeço a Deus por guiar meus passos.

Agradeço à minha família, que me apoiou e incentivou durante toda minha vida.

Agradeço aos meus amigos que me fizeram rir até nas horas mais difíceis.

Agradeço ao meu orientador, o Prof. Paulo Cesar Rodacki Gomes, por ter possibilitado a realização deste trabalho, me orientando da melhor forma.

Agradeço ao Rodrigo Hackbarth por me ajudar no desenvolvimento do *abstract*.

Agradeço a Michelle de Mentzingen Gomes por estar presente e me incentivar no decorrer de todos esses anos de faculdade.

Yuunami niizanae.

Senna

# RESUMO

O presente trabalho detalha a pesquisa e desenvolvimento de uma ferramenta capaz de aplicar técnicas de nível de detalhamento em um modelo 3D no formato de arquivo *Quake 2's Models* (MD2). A finalidade é a redução da malha de polígonos do modelo 3D para aumentar o desempenho desses modelos na Mobile 3D Game Engine (M3GE), motor de jogos 3D para celulares. São discutidas questões como as características do modelo MD2, sua leitura e gravação, assim como textura, vetores normais e malha de polígonos. São utilizadas técnicas de simplificação por meio de "colapso" das arestas de custo mínimo, com base no tamanho da aresta e na coplanaridade entre triângulos da malha de polígonos. A ferramenta disponibilizada permitiu a simplificação de modelos em até 60% o número de vértices, mantendo a geometria e as principais características do modelo em determinados níveis de aproximação do observador.

Palavras-chave: Nível de detalhamento. Modelo MD2. Computação gráfica. Malha de polígonos.

# ABSTRACT

The present work details the research and development of a tool capable of applying level of detail techniques in a 3D model in the Quake 2´s Models (MD2) file format. The purpose is to reduce the polygon mesh of the 3D model to increase this models performance in the Mobile 3D Game Engine (M3GE), 3D games engine to cell phones. Some questions are discussed such as the MD2 model characteristics, its reading and writing, as well as texture, normal vectors and polygon mesh. Techniques of simplifying by collapsing minimum cost edges are used, basing on the size of the edge and on the coplanarity among the polygon mesh triangles. The available tool allowed the models simplification up to 60% of the number of nodes, keeping the models geometry and main characteristics in some levels of the observers approaching.



Key-words: Level of detail. MD2 model. Graphical computing. Polygons mesh.

# LISTA DE ILUSTRAÇÕES

# LISTA DE SIGLAS

2D – Bidimensional

3D – Tridimensional

API – *Application Programming Interface*

BCC – Curso de Ciências da Computação – Bacharelado

BMP – *BitMaP*

DSC – Departamento de Sistemas e Computação

FPS – *Frames Per Second*

FURB – Universidade Regional de Blumenau

GLUI – *openGL User Interface*

GLUT – *openGL Utility Toolkit*

J2ME – *Java 2 Micro Edition*

LOD – *Level Of Detail*

M3GE – *Mobile 3D Game Engine*

MD2 – *quake 2's MoDels*

OpenGL – *Open Graphics Library*

PCX – *PC paintbrush eXchange*

PNG - *Portable Network Graphics*

TGA – *TarGA*

UML – *Unified Modeling Language*

# SUMÁRIO

# 1 INTRODUÇÃO

A sociedade vive em um espaço onde a computação gráfica está presente em diversos aspectos do cotidiano das pessoas, tornando-se assim algo indispensável. É notável a velocidade com que essa área tem evoluído com o passar dos tempos. Hoje com o auxílio deste conhecimento, diversas áreas como arquitetura, astronomia, educação, engenharia, geografia, lazer, marketing, medicina, meteorologia, psicologia e segurança pública são favorecidas prevenindo riscos e reduzindo custos (AZEVEDO; CONCI, 2003, p. 08-09).

Paralelamente à evolução da computação gráfica, os softwares que a utilizam também se especializaram, permitindo desde desenhos simples e práticos em 2D até a modelagem de objetos, terrenos, animais e pessoas em um ambiente 3D. Tais programas são conhecidos como softwares de modelagem 3D. Esses softwares oferecem aos seus usuários a possibilidade de modelar utilizando recursos de textura e malhas de polígonos altamente detalhadas.

Modelos 3D que possuem grande quantidade de detalhes, oferecem uma maior qualidade e percepção de realidade para o observador, contudo trazem consigo um problema que resulta na alta demanda de processamento gráfico, para situações em tempo real ou mesmo quando o hardware não dispõe de grande capacidade de processamento, como celulares. Nesses casos, um nível de detalhe muito elevado é dispensável, devido ao tamanho do visor dos celulares bastante pequenos e suas baixas resoluções, os detalhes não são muito perceptíveis e sua simplificação em celulares favorece assim a aplicação com ganho de desempenho notável.

Com foco na solução do problema apresentado, foi desenvolvida uma ferramenta que utiliza técnicas da computação gráfica conhecidas como *Level Of Detail*[1] (LOD). Akenine-Möller e Haines (2002, p. 389) acrescentam que a idéia básica de LOD é fazer uso de versões simplificadas de um objeto, assim reduzindo a malha de polígonos e textura de um modelo. O conceito de se usar versões com geometrias mais simplificadas para representar objetos menos importantes ou que estejam à grande distância foi sugerida por Clark em 1976 (KRUS et al, 2004).

Esta funcionalidade se fez necessária, proporcionando um ganho de memória e potencializando o desempenho na renderização deste tipo de modelo na *Mobile 3D Game*

[1] Acrônimo usado para representar nível de detalhamento

*Engine* (M3GE) (GOMES; PAMPLONA, 2005), o qual é um projeto de motor de jogos em desenvolvimento no Departamento de Sistemas e Computação (DSC) da Universidade Regional de Blumenau (FURB).

## 1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta que permita reconstruir modelos 3D utilizando técnicas de LOD, simplificando a malha de polígonos mantendo a compatibilidade com a M3GE.

Os objetivos específicos do trabalho são:

a) ler e interpretar o tipo de arquivo quake 2's MoDels (MD2), que armazena o modelo 3D, atualmente utilizado pela M3GE, para que seja feita sua reconstrução simplificada;
b) disponibilizar um algoritmo que utilize técnicas de LOD, reduzindo a malha de polígonos e textura do modelo 3D;
c) permitir ao usuário a escolha do nível de LOD que será aplicado ao modelo, mais ou menos detalhado.

## 1.2 ESTRUTURA DO TRABALHO

Este trabalho divide-se em fundamentação teórica, prática e conclusões. A fundamentação teórica tem como objetivo passar ao leitor todo embasamento necessário para o entendimento da pesquisa realizada. Através dele são apresentadas as características e a relevância do formato de arquivo MD2 para esse trabalho. Também é possível entender a malha de polígonos relacionada aos modelos 3D, assim como a finalidade, importância e comportamento das técnicas de nível de detalhamento.

No capítulo de desenvolvimento do trabalho é apresentado a seqüência da implementação desse trabalho, desde a formalização da implementação usando diagramas da *Unified Modeling Language* (UML), seguida das principais funcionalidades disponibilizadas, e finalizando com uma análise de testes realizados.

Após o capítulo de desenvolvimento do trabalho, são apresentados comentários finais e resultados obtidos.

# 2 FUNDAMENTAÇÃO TEÓRICA

A seguir estão relatados aspectos teóricos que detalham alguns conceitos de diversos autores sobre métodos, técnicas e fórmulas que foram empregadas na realização deste trabalho. Na seção 2.1 comenta-se sobre formato de arquivo MD2, evidenciando suas características e sua relevância. A seção 2.2 tem como foco o item malha de polígonos que está relacionada aos modelos 3D. A seção 2.3 detalha sobre técnica de LOD, sua finalidade, benefício e utilização. A seção 2.4 abrange os principais tópicos da M3GE, também especificando qual a importância da presente ferramenta com relação a M3GE. Por fim, a seção 2.5 apresenta alguns trabalhos correlatos a este tema.

## 2.1 ARQUIVO MD2

A empresa Id Software foi quem desenvolveu o formato de arquivo *quake 2's MoDels*[2] (MD2) para armazenamento de modelos 3D, o qual foi utilizado no jogo Quake 2. O MD2 é um formato de arquivo binário, composto de cabeçalho e dados. O cabeçalho contém todas as informações necessárias para manipulação dos dados.

"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." (ESTÁCIO, 2006, p. 21).

Algumas limitações que o arquivo MD2 possui podem ser citadas como o número máximo de textura, *frames* e vértices (HENRY, 2004). O arquivo MD2 por estar armazenado em formato binário facilita a transmissão por ser leve, embora não permita sua edição em aplicativos comuns de texto.

[2] Formato utilizado na modelagem de personagens do jogo Quake 2.

### 2.1.1 Cabeçalho

O cabeçalho do arquivo MD2 possui sempre o mesmo tamanho, 64 *bytes*, é nele que estão presentes todas as informações para a interpretação dos dados que compõe o arquivo MD2, como, número de vértices, número de texturas, número de coordenadas de textura, número de triângulos, número de comandos *Open Graphics Library* (OpenGL), número de quadros-chave, medidas de altura e largura da textura, tamanho dos quadros-chave em *bytes* e informações de deslocamento para nome e coordenadas de textura, lista de triângulos, frames e comandos OpenGL (SANTEE, 2005, p. 264).

Estácio (2006, p. 21-22) descreve alguns conceitos sobre o conteúdo do cabeçalho:

a) vértices são pontos x, y e z no mundo cartesiano;
b) vetor normal é um vetor associado ao vértice, ele indica para qual direção está apontando a face do vértice, sua utilização dá-se no calculo de iluminação do modelo;
c) coordenadas de textura mapeiam pontos do modelo 3D para textura, esse mapeamento é realizado associando cada vértice as coordenadas de textura *s* e t, Estas coordenadas referenciam o arquivo da textura, *s* representa o eixo *x* e *t* representa o eixo *y*;
d) *skin* é a textura do modelo, ela agrega valor a representação deste modelo para o observador, partes do corpo, roupas, materiais como relógio, entre outros, podem ser identificados através da textura. Seu mapeamento ao modelo 3D é realizado através das coordenadas de textura. Somente uma *skin* de cada vez pode estar associada ao modelo;
e) triângulos representam as faces do modelo 3D, cada triângulo possui três índices para a lista de vértices, sendo assim um triângulo é formado por três vértices;
f) comandos OpenGL estão presentes no arquivo MD2, são utilizados como uma forma de renderização do modelo 3D através das primitivas GL_TRIANGLE_FAN e GL_TRIANGLE_STRIP. Os comandos OpenGL estão organizados da seguinte forma: número de vértices a serem desenhados, primitiva a ser utilizada para renderizar, coordenadas de textura (*s* e *t*) e o índice para o vértice. Os três últimos valores estão representados em uma seqüência de três em três. Estas primitivas têm utilização opcional, pois o arquivo MD2 possui uma lista de triângulos utilizados na modelagem do personagem e também utilizada na renderização do personagem;

g) quadros-chave, o modelo é composto por um conjunto de quadros-chave, cada quadro-chave apresenta uma etapa da animação do modelo e armazena uma lista de vértices, um vetor de escala e translação em relação às coordenadas x, y e z de cada vértice da lista. Para que seja possível visualizar uma animação de qualidade no modelo, é utilizado um cálculo de interpolação entre os quadros-chave.

A estrutura completa do cabeçalho do arquivo MD2 pode ser visualizada no Quadro 1.

| **Deslocamento** | **Número de elementos** | **Descrição** |
|---|---|---|
| 0 | 1 | Número Mágico (deve ser igual a 844121161) |
| 4 | 1 | Versão do Arquivo (para Quake2 deve ser igual a 8) |
| 8 | 1 | Lagura das *skins* |
| 12 | 1 | Altura das *skins* |
| 16 | 1 | Tamanho do frame (em *byte*) |
| 20 | 1 | Número de *skins* (num_skin) |
| 24 | 1 | Número de vértices |
| 28 | 1 | Número de coordenadas de textura (num_st) |
| 32 | 1 | Número de triângulos (num_tris) |
| 36 | 1 | Número de comandos OpenGL (num_glcmds) |
| 40 | 1 | Número de frames (num_frames) |
| 44 | 1 | Deslocamento para os nomes das *skins* (ofs_skins) |
| 48 | 1 | Deslocamento para as coordenadas de textura (ofs_st) |
| 52 | 1 | Deslocamento para a lista de triângulos (ofs_tris) |
| 56 | 1 | Deslocamento para os dados dos frames (ofs_frame) |
| 60 | 1 | Deslocamento para os comandos OpenGL (ofs_glcmds) |
| 64 | 1 | Deslocamento para o final do arquivo (ofs_end) |

Fonte: adaptado de Henry (2002).

Quadro 1 – Formato do arquivo binário MD2

### 2.1.2 Dados

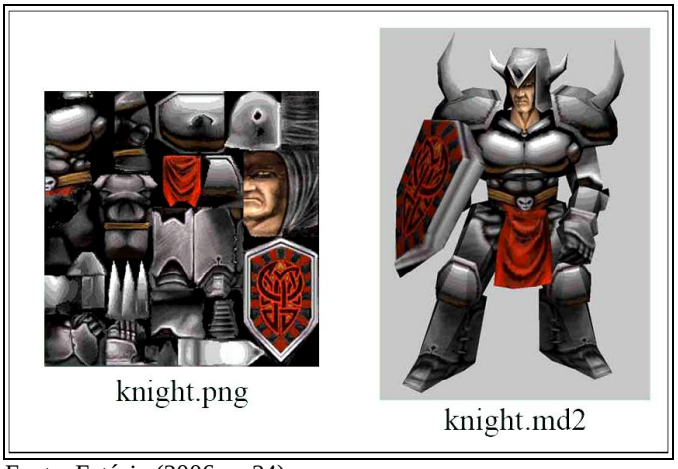
Os dados do arquivo MD2 são de tamanho variável, por ser um arquivo binário, o cabeçalho é indispensável para localização das informações necessárias. Os dados são gravados logo após o cabeçalho e possuem o nome das *skin* associadas ao modelo, cada nome com tamanho de 64 *bytes*, as coordenadas de textura, a lista de triângulos, a lista de frames que contém os vértices do modelo e os comandos OpenGL.

### 2.1.3 Textura

A textura de um modelo MD2 encontra-se em um arquivo separado e cada arquivo MD2 pode somente estar associado a uma textura de cada vez (HENRY, 2004). Worcester

(2007) explica que todos os vértices de um modelo estão mapeados para a mesma textura. O jogo Quake 2 utiliza arquivos de imagens com a extensão *PC paintbrush eXchange* (PCX) para armazenamento das texturas, contudo é permitido utilizar qualquer formato de arquivo de imagem para armazenar a textura.

Uma vantagem é que com a mudança de coloração da textura de um modelo, é possível a criação de diferentes aparências para o mesmo modelo. Está técnica é bastante comum para representação de diferentes personagens em jogos 3D. A figura 1 mostra uma textura com seu respectivo personagem 3D.




Fonte: Estácio (2006, p. 24).

Figura 1 – Textura e personagem 3D

## 2.2 MALHA DE POLÍGONOS

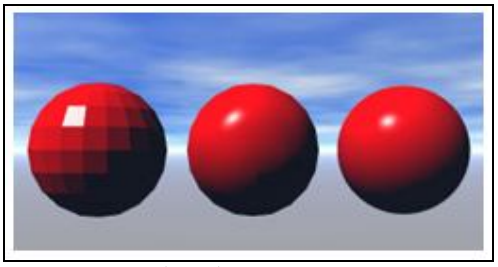
Hill Jr. (2001, p. 288) coloca que a malha de polígonos é simplesmente uma coleção de polígonos ou faces, que juntos formam a pele de um objeto. A malha de polígonos se tornou um caminho padrão de representação de uma ampla classe de formas sólidas em gráficos. Existem vários exemplos, como o cubo, e outros com aproximações de formas mais harmoniosas como a esfera, cilindro e o cone.

A sua proeminência nos gráficos decorre da simplicidade de utilização de polígonos, que são fáceis de representar (por uma seqüência de vértices e de transformações), possuem propriedades simples (iluminação por meio de vetores normais) e são fáceis de desenhar, utilizando uma rotina de preenchimento de polígonos ou mapeando texturas aos polígonos (HILL JR., 2001, p. 288).

A malha de polígonos de um modelo 3D pode ser desenhada utilizando um conjunto de triângulos, quadrados ou ambos, os triângulos são como átomos, pois é a face representada pelo menor número de vértices e arestas, qualquer superfície pode ser muito bem representada pelos mesmos. Os triângulos são a forma mais comum de representação da malha de polígonos de objetos explica Akenine-Möller e Haines (2002, p. 440). O processo de conversão de um polígono complexo em triângulos é chamado de triangulação

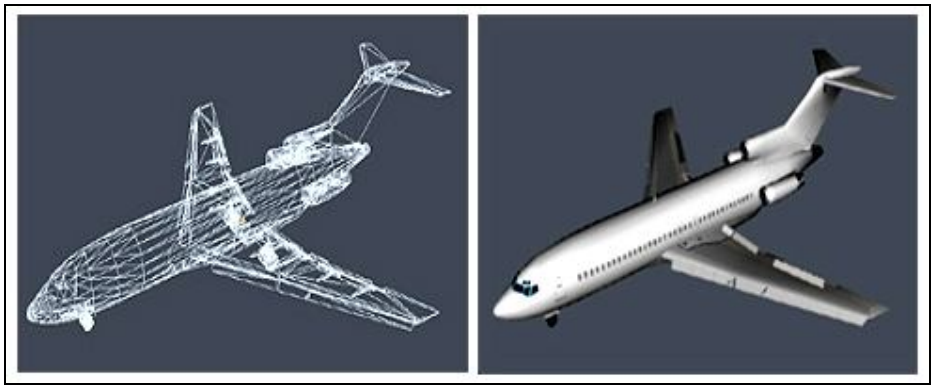
Muitos sistemas de renderização, incluindo OpenGL, baseiam-se em desenhar objetos por meio de uma seqüência de polígonos. Cada face poligonal é enviada através do *graphics pipeline*[3] onde os vértices sofrerão diversas transformações, até que, finalmente, é apresentado o resultado no dispositivo visual (Hill Jr., 2001, p. 288).

De acordo com Hill Jr. (2001, p. 288) alguns objetos podem ser perfeitamente representados por este tipo de malha, outros podem ser somente aproximados. No entanto, existem técnicas que solucionam este problema suavizando o modelo representado. A figura 2 mostra um exemplo de utilização de técnicas de suavização em uma esfera.



Fonte: Watson (2006).
Figura 2 – Utilização de técnica para suavização

O detalhamento de um modelo está diretamente ligado à malha de polígonos do mesmo. A figura 3 exemplifica uma malha de polígonos e o modelo representado pela mesma.



Fonte: AI Aardvark (2006).
Figura 3 – Malha de polígonos de um modelo 3D

[3] Abrange o processo de transformações para exibição do modelo 3D em um dispositivo visual 2D.

## 2.3 NÍVEL DE DETALHAMENTO

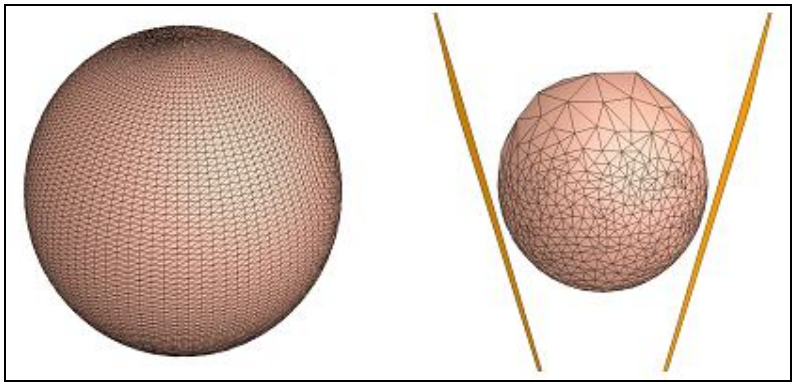
Brito (2007) afirma que é sempre um desafio economizar recursos do computador. Para superar parte desse desafio uma técnica muito utilizada é a LOD. Algoritmos LOD apresentam três atributos principais, denominados de geração, seleção e escolha. A geração pode ser considerada o processo onde diferentes representações de um modelo são geradas com diferentes detalhes. A seleção é o mecanismo de escolha de um nível de detalhe a ser aplicado ao modelo. A escolha consiste na mudança entre diferentes níveis de detalhes (AKENINE-MÖLLER; HAINES, 2002, p. 390).

O alvo da simplificação de polígonos, quando usado para geração de níveis de detalhe, é remover primitivas da malha original, para produzir um modelo simplificado que retêm as características visuais importantes do objeto original (KRUS et al, 2004).

Conforme Krus et al. (2004) a simplificação deve ser conduzida de tal maneira que a forma e as características principais que constituem o objeto sejam preservadas. Assim, os algoritmos têm que procurar algumas características distintas do objeto. Atualmente os algoritmos de LOD são capazes de produzir níveis satisfatórios dos detalhes com respeito às exigências visuais e geométricas, devido a melhorias no hardware, além de próprias melhorias nos algoritmos.

É comum utilizar essa técnica em jogos quando o observador está consideravelmente longe do objeto 3D no ambiente, desse modo o objeto não necessita de um nível de detalhamento muito elevado. O observador não pode enxergar os detalhes minuciosos a uma longa distância. Quanto maior a distância entre o objeto e o observador menor o nível de detalhamento do objeto, isto agiliza o processamento gráfico da cena.

O nível de detalhamento pode ser realizado utilizando dependência ou independência de visão. Na abordagem dependente de visão, a parte que está dentro do campo de visão do observador ou onde ele estiver mais perto ficará com detalhes minuciosos. Este método é exemplificado na figura 4. Na abordagem independente de visão, este campo de visão do observador não é considerado ao gerar a simplificação do modelo, assim todo o modelo é otimizado (DELOURA, 2000, p. 456-457).



Fonte: Hoppe (1997).

Figura 4 – Uso da abordagem de dependência de visão para otimização do modelo

Na figura 4, as linhas que envolvem a esfera da direita representam o campo de visão do observador, que têm uma visão limitada da esfera. Pode-se notar que a região onde o observador visualiza está mais detalhada que as demais. Outra característica é o contorno que mantém a silhueta da esfera permanecer com detalhes minuciosos, assim a forma geométrica do modelo para seu reconhecimento é mantida.

No decorrer do desenvolvimento, análise e pesquisa deste projeto, foram encontrados diversos artigos sobre o tema abordado neste capítulo. A seguir são apresentados resumos de alguns artigos relevantes para a compreensão deste capítulo, detalhando assuntos abordados e seus objetivos.

### 2.3.1 Malhas Progressivas

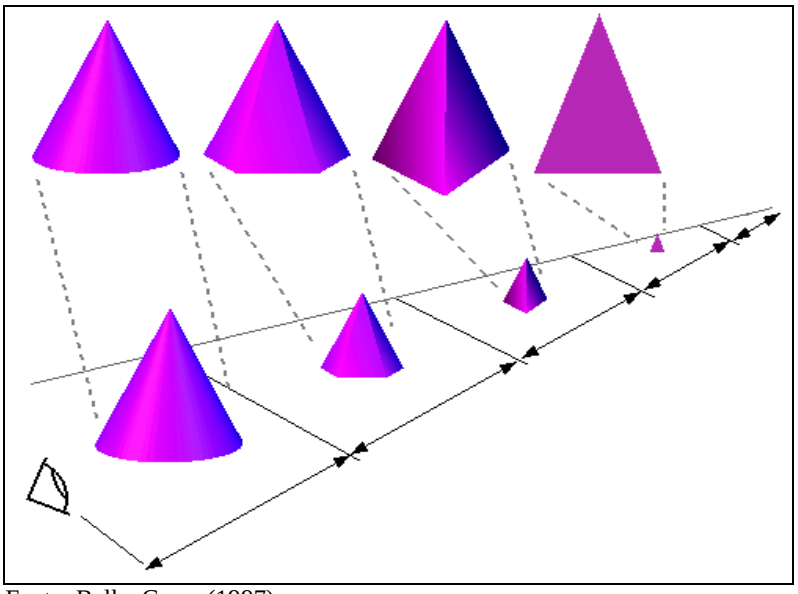
Modelos geométricos altamente detalhados estão cada vez mais se tornando freqüentes na computação gráfica. Estes modelos muitas vezes são representados através de uma malha de triângulos complexa, desafiando o desempenho de renderização e a capacidade de armazenamento. Hoppe (1996) disponibilizou em seu *website* um artigo sobre malhas progressivas, que introduz um novo esquema para armazenamento e transmissão de uma malha de triângulos arbitrária.

Uma malha progressiva representa modelos poligonais com uma seqüência de arestas “colapsadas” (modelos que já sofreram ação de alguma técnica de nível de detalhamento). O artigo aborda diversos problemas práticos nos gráficos, como a transição suave entre diferentes níveis de detalhamento, chamada de nível de detalhamento por aproximação, compressão da malha e o refinamento seletivo.

Além disso, é apresentado um novo procedimento de simplificação de malhas para

construção de uma malha progressiva através de alguma malha arbitrária. A meta deste procedimento de otimização é preservar não apenas a geometria da malha original, mas o mais importante é sua aparência global, tal como definido pela sua aparência discreta e atributos escalares (como identificadores materiais, atributos de cores, vetores normais e coordenadas de textura).

A técnica de nível de detalhamento por aproximação visa melhorar ainda mais o desempenho da renderização, sendo comum se definir várias versões do mesmo modelo com diferentes níveis de detalhe. À medida que o observador se aproxima ou se afasta do modelo, é realizada a troca para uma versão mais ou menos detalhada, como mostra a figura 5. Contudo, a troca pode se tornar perceptível pelo observador, nesse caso o artigo visa suavizar essa transição de uma versão para outra.



Fonte: Bell e Carey (1997).

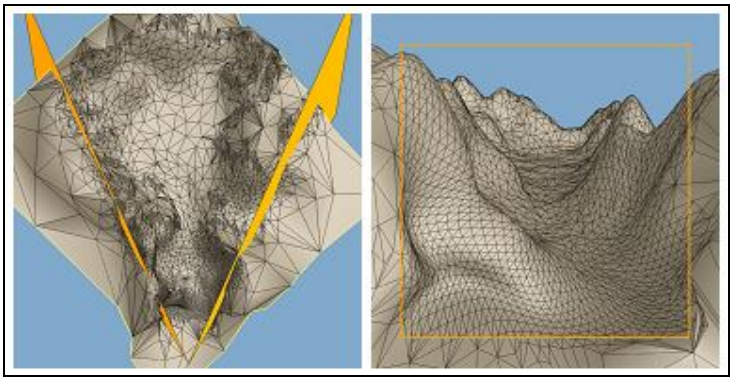
Figura 5 – Variação do nível de detalhamento conforme a distância do modelo do cone

O refinamento seletivo também é comentado neste artigo, como já visto no capítulo de nível de detalhamento, o refinamento seletivo é o uso da abordagem de dependência de visão para otimização do modelo, onde certas regiões do modelo manterão mais detalhes que as demais, considerando o campo de visão do observador. BELL, Gavin; CAREY, Rikk

No artigo está presente a construção da representação de malha progressiva e suas aplicações utilizando diversos modelos práticos. As figuras 6 e 7 demonstram alguns dos resultados obtidos empregando os procedimentos de simplificação desse artigo. A figura 6 apresenta a simplificação por meio do refinamento seletivo levando em consideração a visão do observador delimitada nas imagens pelas linhas amarelas.

A figura 7 ilustra imagens de uma sala antes e depois do processo de simplificação,

reduzindo em até 93% o número de triângulos sem que haja uma percepção nítida de perda de qualidade no modelo.



Fonte: Hoppe (1996).

Figura 6 – Refinamento seletivo da malha de um terreno

Fonte: Hoppe (1996).

Figura 7 – Simplificação de uma sala, à esquerda 150.983 triângulos, à direita 10.000 triângulos

### 2.3.2 Simplificação de superfícies usando métricas de erro quadráticas

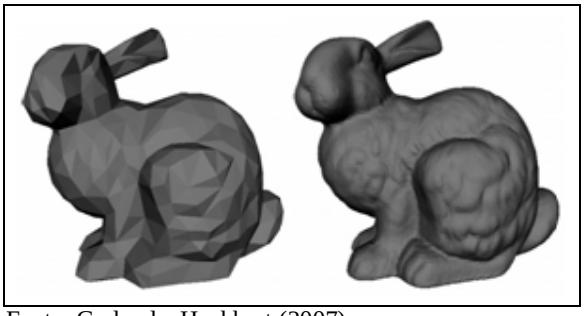
A presente seção é baseada no trabalho de Garland e Heckbert (2007), que descreveu o desenvolvimento de um algoritmo de simplificação de superfícies que pode rapidamente produzir aproximações de alta qualidade de modelos poligonais.

O algoritmo utilizado nesse artigo usa contrações iterativas de pares de vértices para simplificação do modelo e sustenta a aproximação de erro da superfície usando matrizes quadráticas. Contraindo arbitrariamente pares de vértices, o algoritmo é capaz de juntar regiões desconectadas do modelo, evitando que o modelo se quebre em partes menores, facilitando assim uma melhor aproximação geométrica e visual.

As principais vantagens da utilização dessa abordagem são a eficiência, qualidade e

generalidade, sendo capaz de simplificar modelos complexos em um curto espaço de tempo. No entanto, são necessários números de até 10 casas decimais por vértice para uma aproximação satisfatória. A aproximação gerada pelo algoritmo mantém alta fidelidade do modelo original, mantendo as características mais relevantes. A fórmula de custo empregada no algoritmo em questão é similar a fórmula de custo utilizada no artigo malhas progressivas (seção 2.3.1). Uma característica dessa abordagem é não preservar as posições originais dos vértices, sendo assim necessário recalcular propriedades do modelo como, por exemplo, as coordenadas de textura e os vetores normais.

A técnica empregada nesse artigo foi testada em diversos modelos com finalidade divergentes, como terrenos e animais, e em ambos os testes o algoritmo apresentou resultados significativos. Um exemplo pode ser visto na figura 8 representada por um modelo de um coelho.



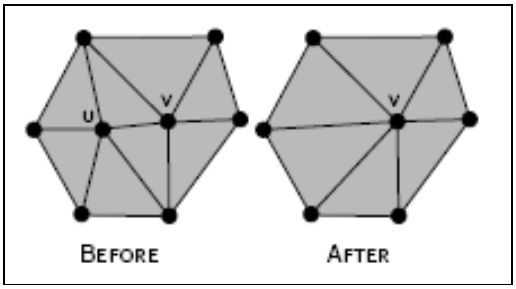
Fonte: Garland e Heckbert (2007).
Figura 8 – Simplificação de um coelho, à esquerda 1.000 faces, à direita 69.451 faces

### 2.3.3 Redução poligonal

Esta seção apresenta um simples, rápido e eficiente algoritmo de redução poligonal focado no trabalho de Melax (1998). Tal algoritmo somente trabalha com triângulos, mas isso não chega a ser uma limitação já que de fato muitas aplicações trabalham exclusivamente com triângulos.

Muitos dos melhores algoritmos atualmente na literatura geralmente são variações do algoritmo de malha progressiva de Hoppe (1996). Neste caso o algoritmo trabalha com colapsos de arestas, realizando essa operação iterativamente, onde uma aresta é selecionada, e um de seus dois vértices é “colidido”, ou “colapsado” contra o outro. Esse processo é repetido até que o objetivo seja atendido. Como conseqüência, a cada vez que for realizado esse processo no modelo, são eliminados um vértice, dois triângulos e três arestas, como mostra a

figura 9, onde o vértice u é colidido com o vértice v. Note que o custo de colidir o vértice u com o vértice v é diferente do custo de colidir o vértice v com o vértice u, pois o conjunto de triângulos ligados ao vértice u utilizados na fórmula de simplificação, é diferente do conjunto de triângulos ligados ao vértice v, assim a malha simplificada resultante de uma operação de colisão é diferente da malha gerada pela outra.




Fonte: Melax (1998).

Figura 9 – Processo de colisão de vértices

Para produzir um bom nível de detalhamento no polígono, utilizando a abordagem de colapso de arestas, é necessário escolher a aresta que quando colapsada cause a menor mudança visual. Quanto melhor o método de escolha, mais difícil é o seu desenvolvimento e mais tempo ele irá levar para definir um custo de colapso para todas as arestas. O algoritmo desse artigo foca uma abordagem rápida para definir o custo de colapso de cada aresta e por esse motivo pode ser utilizado para aplicações de tempo real.

Obviamente faz sentido eliminar pequenos detalhes em primeiro lugar. Note também que são necessários menos polígonos para a representação de superfícies coplanares e mais polígonos são necessários para representar áreas com altas curvaturas. O custo de colapso das arestas é calculado com base nessas heurísticas, a curvatura e o tamanho das faces e arestas são determinantes para a escolha da aresta a ser colapsada.

A heurística da fórmula para simplificação do modelo MD2 define o custo de colapso de uma aresta pelo tamanho da mesma multiplicado ao termo de curvatura. O termo de curvatura é determinado comparando o produto escalar das faces normais dos triângulos, assim determinando a coplanaridade entre triângulos da malha de polígonos. A idéia é simplificar regiões formadas por pequenos triângulos que estejam quase ou totalmente alinhados no mesmo plano. O quadro 2 define a equação matemática para esta operação.

$$\cos t(u,v) = \|u - v\| \times \max_{f \in Tu} \left\{ \min_{n \in Tuv} \left\{ \left(1 - f.normal \bullet n.normal\right) \div 2 \right\} \right\}$$

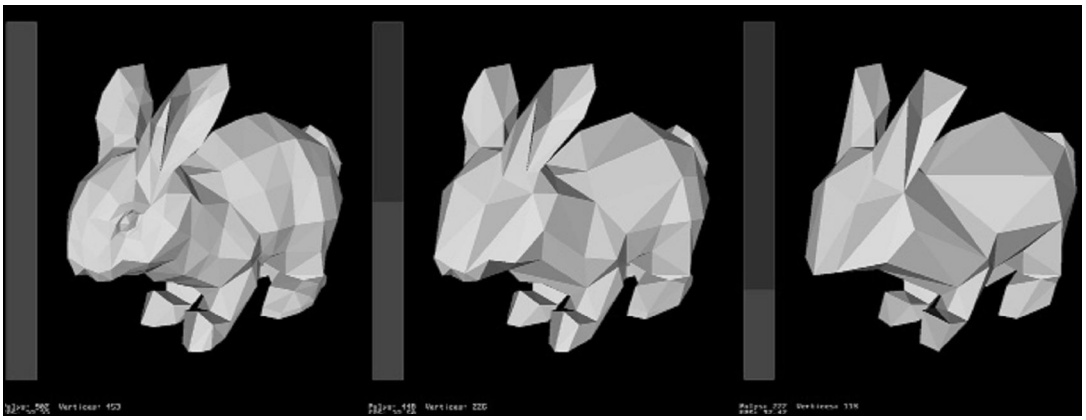
Fonte: Melax (1998).

Quadro 2 – Equação de custo de colapso da aresta

A equação do quadro 2 define o custo de colapso da aresta. Para se obter o custo da mesma são necessários seus dois vértices, representados na equação por u e v. O primeiro

segmento da fórmula, dado por ‖u-v‖ representa o cálculo da magnitude da aresta (o seu tamanho). No segundo segmento da fórmula é utilizado o cálculo f.normal•n.normal que define o produto escalar entre os vetores normais dos triângulos de f e n, caso o resultado dessa operação seja zero (0) os vetores normais são ortogonais (90º). O f representado no cálculo representa o conjunto de triângulos que contém o vértice u (f∈Tu) e n representa o conjunto de triângulos que contém ambos os vértices u e v (n∈Tuv). O objetivo do segundo segmento da fórmula é encontrar triângulos que estejam mais planares entre si, e assim realizar a simplificação nesta área.

Uma das grandes vantagens que garantem a velocidade e eficiência do algoritmo se dá pelo fato de que as posições dos vértices no modelo não é alterada. Isso garante que vetores normais dos vértices, coordenadas de textura e propriedades similares não são afetadas, evitando que seja realizado um recálculo para obtenção dessas propriedades. A figura 10 mostra um exemplo da utilização desse algoritmo em um modelo 3D que representa um coelho.



Fonte: Melax (1998).
Figura 10 – Etapas de simplificação de um modelo (esquerda para direita) 453, 200 e 100 vértices

## 2.4 M3GE

Como citado anteriormente a M3GE é um projeto de motor de jogos em desenvolvimento no DSC da FURB. Esse projeto é uma biblioteca de desenvolvimento responsável pelo gerenciamento de jogos para telefones celulares. A M3GE permite o uso de diversas funções como a criação de um ambiente 3D a partir de um arquivo de configurações, a criação de um número indefinido de câmeras (que podem ser trocadas dinamicamente

durante o jogo) e o tratamento de eventos do usuário, entre outras funcionalidades (GOMES; PAMPLONA, 2005).

A M3GE foi dividida em dois grandes componentes: um responsável pela leitura dos arquivos de entrada e outro responsável pelo motor de jogos propriamente dito.

A construção de um cenário de jogo na M3GE é realizada através de um arquivo de configurações gerais do ambiente, um arquivo com as malhas de polígonos e uma série de arquivos de texturas (GOMES; PAMPLONA, 2005).

Aplicações de jogos 3D requerem memória para potencializar o desempenho da aplicação. Em dispositivos celulares esse recurso é escasso e o desenvolvimento nesses dispositivos significa controlar o uso da memória disponível para a aplicação.

Uma forma de economizar esse recurso em aplicações de jogos para celulares é utilizar modelos 3D com um nível de detalhe reduzido. Devido ao tamanho do visor e a baixa resolução gráfica dos dispositivos celulares a simplificação dos detalhes não é muito perceptível, sendo válida e contribuindo para um ganho de memória e desempenho da aplicação.

## 2.5 TRABALHOS CORRELATOS

Abaixo estão apresentados os trabalhos correlatos ao tema proposto, como a extensão da M3GE, um ambiente para navegação virtual e um simulador de aviões, citando conceitos e objetivos dos mesmos.

### 2.5.1 Extensão da M3GE

Estácio (2006) resume seu projeto como uma extensão da M3GE que implementa a importação e visualização do Personagem Não Jogador (PNJ) a mesma. O arquivo MD2 é utilizado no projeto de Estácio, é usado para armazenar o modelo 3D do PNJ para a importação e a visualização do mesmo na M3GE.

O termo PNJ tem origem do inglês onde é usado como *Non-Player Character* (NPC). Resumidamente, o NPC ou PNJ define-se como um personagem presente no jogo que não é controlado por um jogador e utilizam técnicas de inteligência artificial para se relacionarem

com os jogadores, os PNJ podem representar oponentes, como por exemplo, em jogos de lutas (ESTÁCIO, 2006, p. 20).

Estácio (2006, p. 49-51) explica que seu projeto é responsável pela inserção de PNJs na M3GE, os modelos 3D utilizados como PNJs são lidos de arquivos MD2. Devido à falta de memória nos dispositivos celulares, seu projeto carrega apenas um quadro de cada personagem na cena, o que impossibilita a realização da animação dos PNJs. Os resultados apresentados com a conclusão do projeto demonstram que é possível utilizar o carregador de arquivos MD2 para adicionar personagens aos jogos da M3GE. Contudo, um fato relevante é a demora na leitura e representação do modelo na tela do telefone celular (cerca de um minuto e quarenta e cinco segundos no total).

### 2.5.2 Ambiente para navegação virtual

Corseuil et al. (2004, p. 02) descreve o *ENvironment for VIRtual Objects Navigation* (ENVIRON) como sendo um visualizador de alta performance em tempo real de objetos exportados a partir do aplicativo 3DSMax. O ENVIRON foi projetado para solucionar problemas de baixa performance para modelos de grande complexidade.

O ambiente desenvolvido utiliza um algoritmo de LOD que tem como objetivo a simplificação e otimização das malhas para visualização. O algoritmo LOD tem como finalidade transmitir ao visualizador as informações paramétricas da geração dos objetos. O visualizador, de acordo com a posição do objeto, gera a malha de triângulos necessária para a visualização (CORSEUIL et al, 2004, p. 02).

Conforme Corseuil et al. (2004, p. 03), o uso de objetos paramétricos permite uma economia no consumo de memória. Objetos distantes o suficiente da câmera vão ocupar apenas o espaço necessário para guardar os poucos parâmetros de geração do objeto. Alguns objetos não têm uma representação paramétrica fácil de trabalhar. Nestes casos o algoritmo implementado consiste em gerar diferentes níveis de detalhamento para a malha de cada objeto, cada um com a quantidade certa de triângulos para uma determinada distância da câmera, ou seja, a representação com a menor quantidade de vértices que represente fielmente o objeto.

### 2.5.3 Simulador de aviões

A AI Aardvark (2006) utiliza técnicas de LOD para simulação de vôo de aviões, onde a intenção é o ganho de *frames* por segundo. Aumentando a velocidade de amostragem destes *frames*, haverá uma maior suavização da representação gráfica da cena, tornando os vôos simulados mais próximos da realidade.

Modelos de aviões que são mostrados a longa distância, utilizam técnicas de LOD, diminuindo o processamento desnecessário na cena (AI AARDVARK, 2006). Um exemplo de processamento desnecessário é mostrado na figura 11, onde são utilizados 32,768 polígonos para representação de um avião aterrissando, e a mesma quantidade de polígonos é utilizada para representar um segundo avião que parece um pequeno ponto no horizonte.


Fonte: AI Aardvark (2006).

Figura 11 – Imagem que demonstra a importância da das técnicas de LOD

# 3 DESENVOLVIMENTO

O presente capítulo aborda a análise e especificação de requisitos e da ferramenta, através de modelos e diagramas. Descrevendo técnicas e ferramentas utilizadas na implementação do software; testes executados e resultados obtidos.

O presente trabalho agrega conhecimento dos autores referenciados, entretanto, os trabalhos de Henry (2004) e Melax (1998) têm participação especial no desenvolvimento, por disponibilizarem código fonte escritos na linguagem de programação C, devidamente adaptados as necessidades dessa ferramenta. Henry (2004) e Melax (1998) disponibilizam respectivamente códigos fontes para a leitura do arquivo MD2 e para cálculo e aplicação do nível de detalhamento na malha de polígonos.

Na seção de execução dos testes, é apresentada a utilização da ferramenta desenvolvida por Estácio (2006), que permite a importação e visualização de arquivos MD2 na biblioteca M3GE disponibilizada por Gomes e Pamplona (2006).

## 3.1 ANÁLISE E ESPECIFICAÇÃO DE REQUISITOS

As principais características da presente ferramenta são descritas conforme a especificação dos seguintes requisitos:

a) ler e interpretar arquivos do tipo MD2;
b) permitir que o usuário escolha um nível de detalhamento para a geração do novo modelo 3D;
c) implementar o uso de técnicas de LOD para a redução da malha de polígonos do modelo;
d) geração de um modelo simplificado a partir de um modelo lido;
e) permitir ao usuário salvar arquivos MD2 de modelos simplificados pela ferramenta;
f) o sistema deverá ser portável, executar em plataformas como Windows 2000, XP e MacOS.

## 3.2 ESPECIFICAÇÃO

Este trabalho foi desenvolvido utilizando a linguagem de programação C++, seguindo o paradigma de orientação a objetos. Em sua especificação é utilizado os diagramas de caso de uso, classes e seqüência presentes na UML. Considerando o fato da ferramenta seguir o paradigma de orientação a objetos, o diagrama de classes possui um papel importante na melhor visualização da mesma. Para a confecção dos diagramas citados foi utilizada a ferramenta de apoio Enterprise Architect.

### 3.2.1 Diagrama de caso de uso

No diagrama de caso de uso são representadas as ações que o usuário pode realizar através da ferramenta desenvolvida. A figura 12 ilustra os casos de uso para o usuário da ferramenta.

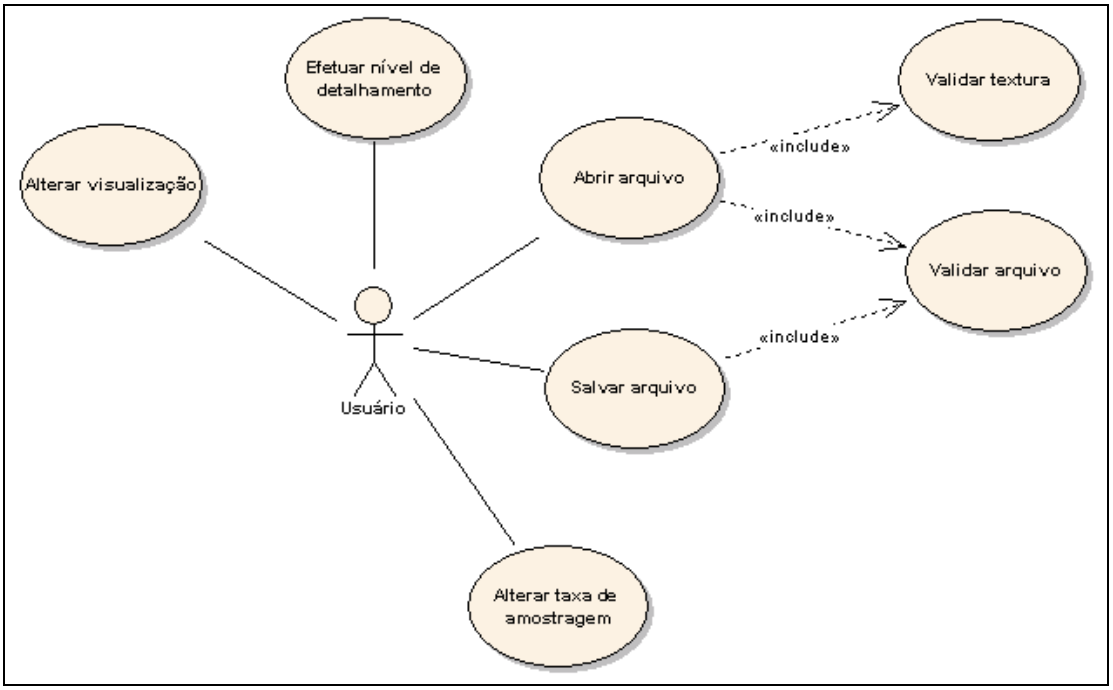



Figura 12 – Diagrama de casos de uso

O usuário tem a opção de abrir um arquivo MD2 através do caso de uso “abrir arquivo”, ao executar essa operação são realizadas duas validações, a primeira verifica se o arquivo é valido e a segunda verifica se a textura é valida. É disponibilizada uma opção de

salvar o arquivo MD2 em disco, através do caso de uso “salvar arquivo”, ao executar essa operação é realizada uma validação sobre o nome do arquivo a ser salvo. As operações de abrir e salvar arquivo incluem os casos de uso “validar textura” e “validar arquivo” responsáveis pela verificação da textura e do arquivo respectivamente, como pode ser visto no diagrama da figura 12.

Já os casos de uso “alterar visualização”, “alterar taxa de amostragem” e “efetuar nível de detalhamento” tem como objetivos, respectivamente, dispor ao usuário diferentes formas de visualização do modelo 3D, aumentar ou diminuir a taxa de *frames* por segundo e gerar a simplificação do modelo lido.

### 3.2.2 Diagrama de classes

A ferramenta é composta por dois módulos principais que permitem a realização de seu objetivo, sendo eles, o módulo de manipulação do arquivo MD2, onde é possível a leitura e interpretação do arquivo e também de sua textura, assim como a animação do modelo 3D e a gravação do modelo em disco. O outro módulo presente na ferramenta é o de simplificação da malha de polígonos a partir de um modelo 3D lido, o qual possui uma estrutura para armazenamento da malha de polígonos para aplicação de cálculos e técnicas de LOD sobre a mesma.

As figuras 13 e 14 a seguir ilustram através dos diagramas de classes, os módulos de manipulação do arquivo MD2 e simplificação da malha de polígonos, respectivamente. Características importantes para o melhor entendimento dos diagramas são abordadas na seqüência.

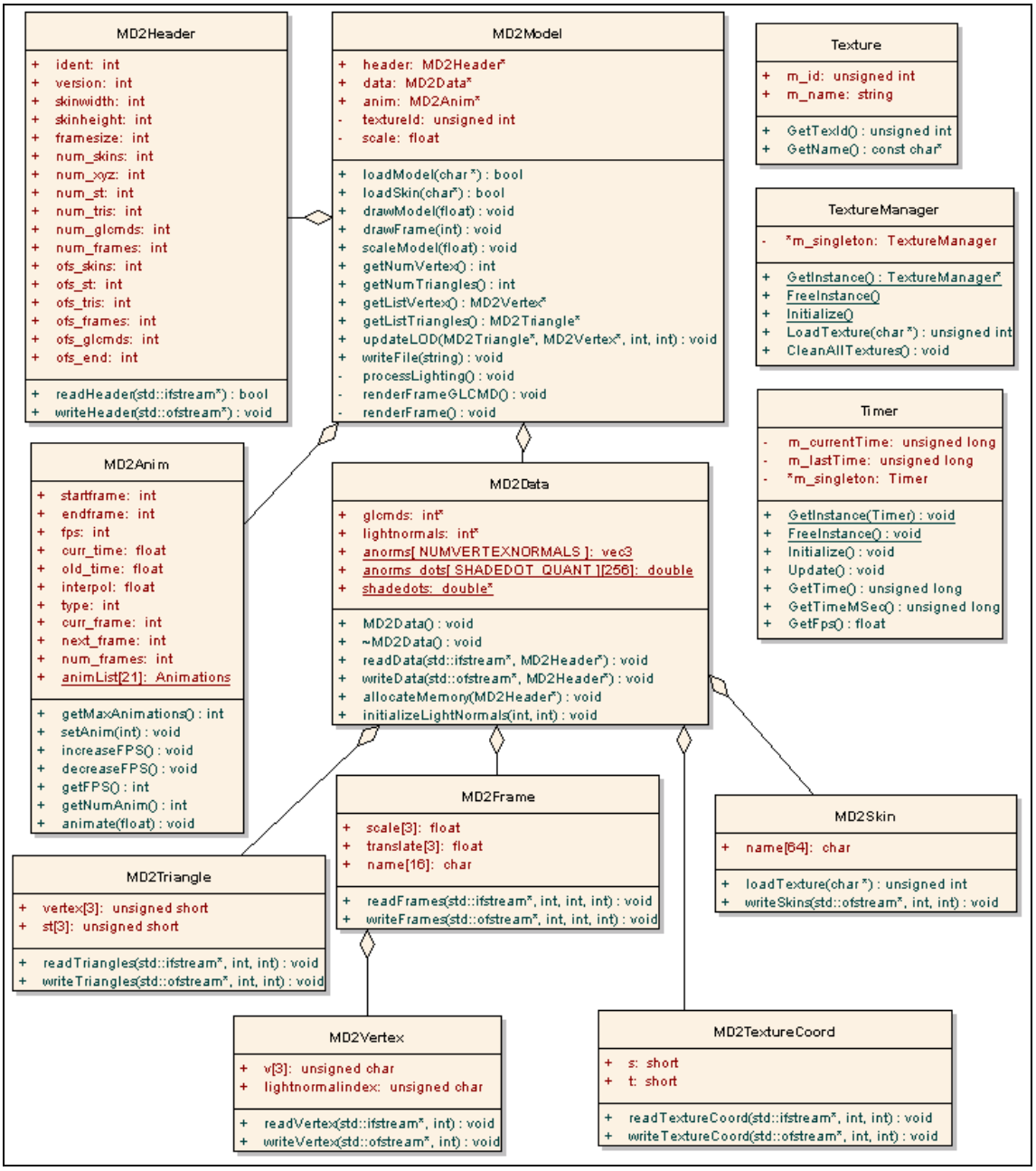



Figura 13 – Diagrama de classe do módulo de manipulação do arquivo MD2

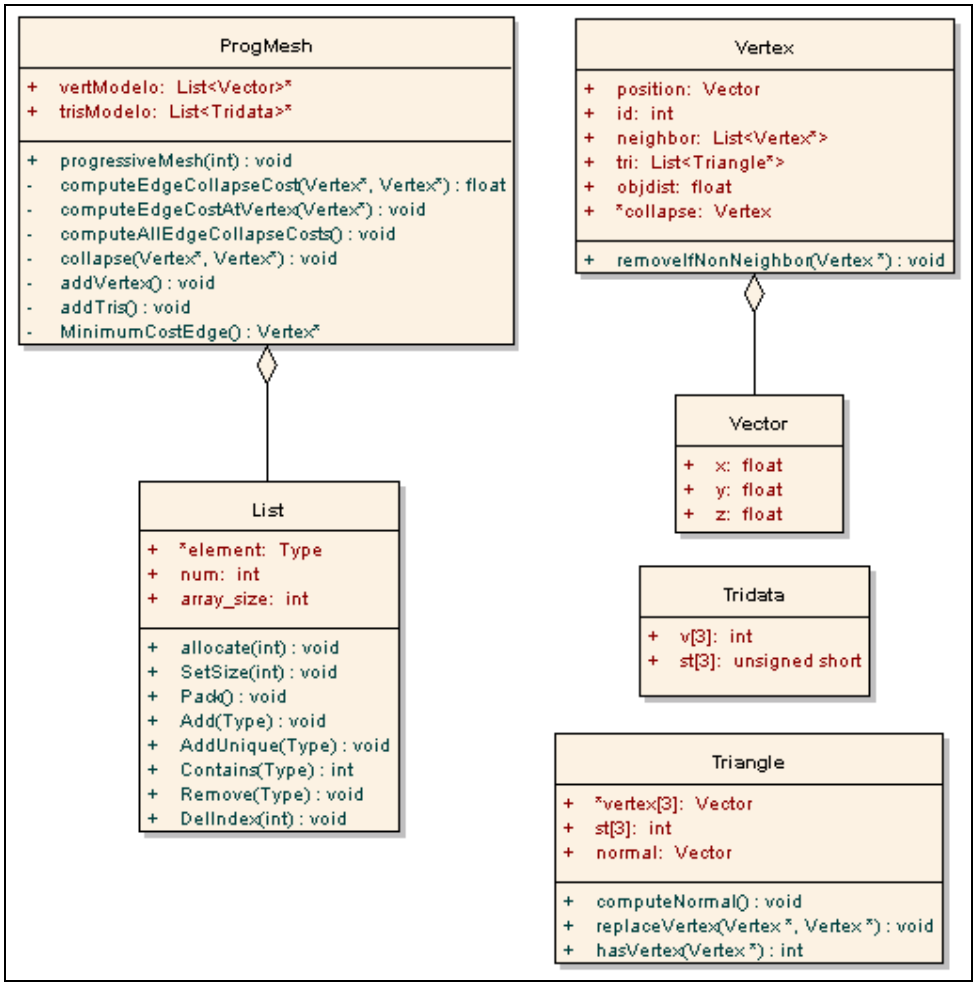



Figura 14 – Diagrama de classe do módulo de simplificação da malha de polígonos

No diagrama de classes de manipulação do modelo MD2 (figura 13) a classe `MD2Model` é responsável pelo desenho do modelo 3D no *canvas*[4] e também pelo gerenciamento das classes que manipulam o cabeçalho, os dados e a animação do modelo. A classe `MD2Header` armazena as informações referentes ao cabeçalho e realiza a leitura e escrita do cabeçalho do arquivo MD2. A classe `MD2Data` gerencia classes de manipulação de dados do arquivo como coordenadas de textura, *frames*, triângulos, entre outros. O arquivo MD2 possui suporte a animação do modelo, essa animação é representada por meio da interpolação entre quadros chaves. O cálculo e gerenciamento de informações para a realização dessa operação é realizado pela classe `MD2Anim`, a qual recebe informações da classe `Timer`, que obtém dados relativos ao tempo, como tempo atual e último tempo registrado.

As classes `Texture` e `TextureManager` são classes de baixo nível disponibilizadas por Henry (2004) que efetuam a leitura da imagem nos formatos PCX, *TarGA* (TGA) e

[4] Área designada na computação para desenho de formas gráficas 2D ou 3D.

*BitMaP* (BMP), utilizada como textura para o modelo 3D. A classe `MD2Skin`, é responsável por ler e gravar o nome das texturas que o arquivo MD2 utiliza.

Os dados utilizados para renderização do arquivo MD2 são representados pelas classes `MD2Triangle`, `MD2TextureCoord`, `MD2Frame` e `MD2Vertex`. Cada uma dessas classes efetua a leitura e gravação de seus dados no arquivo MD2.

`MD2Triangle` são os triângulos da malha de polígonos, compostos de índices para três vértices e três coordenadas de textura. Cada vértice possui três coordenadas, x, y e z, além de um índice para o seu vetor normal, a classe `MD2Vertex` contém essas informações e está agregada a classe `MD2Frame`, que possui dados como escala, translação e nome do *frame*, além de seus vértices representados através da agregação comentada. Cada coordenada de textura é representada pela classe `MD2TextureCoord` que armazena um ponto formado por s e t, onde respectivamente é um ponto x e y da imagem a ser utilizada como textura.

No diagrama de simplificação da malha de polígonos (figura 14) a classe `ProgMesh` constitui núcleo do algoritmo de simplificação de malhas. Através dela são realizados todos os procedimentos para que ocorra a simplificação, desde do armazenamento dos dados referentes a malha de polígonos até a estrutura final com a nova malha simplificada. As classes `Triangle` e `Tridata` representam os triângulos da malha de polígonos, onde a classe `Tridata` é utilizada como ponte entre os triângulos da malha de polígonos do modelo e da malha de polígonos a ser simplificada pelo algoritmo. As classes `Vertex` e `Vector` representam os vértices dessa malha, onde a classe `Vector` é utilizada como ponte entre os vértices do modelo 3D e os vértices a serem simplificados pelo algoritmo. A classe `List` fornece uma opção de agrupamento de dados por lista genérica e controle de índices ao algoritmo.

### 3.2.3 Diagrama de seqüência

Nos diagramas de seqüência estão representadas três funcionalidades consideradas críticas da ferramenta. Duas pertencem ao módulo de manipulação do arquivo MD2 e envolve a funcionalidade de abrir (figura 15) e salvar (figura 16) arquivo MD2, e uma pertence ao módulo de simplificação da malha de polígonos (figura 17), utilizada na funcionalidade de efetuar o nível de detalhamento sobre o modelo lido.

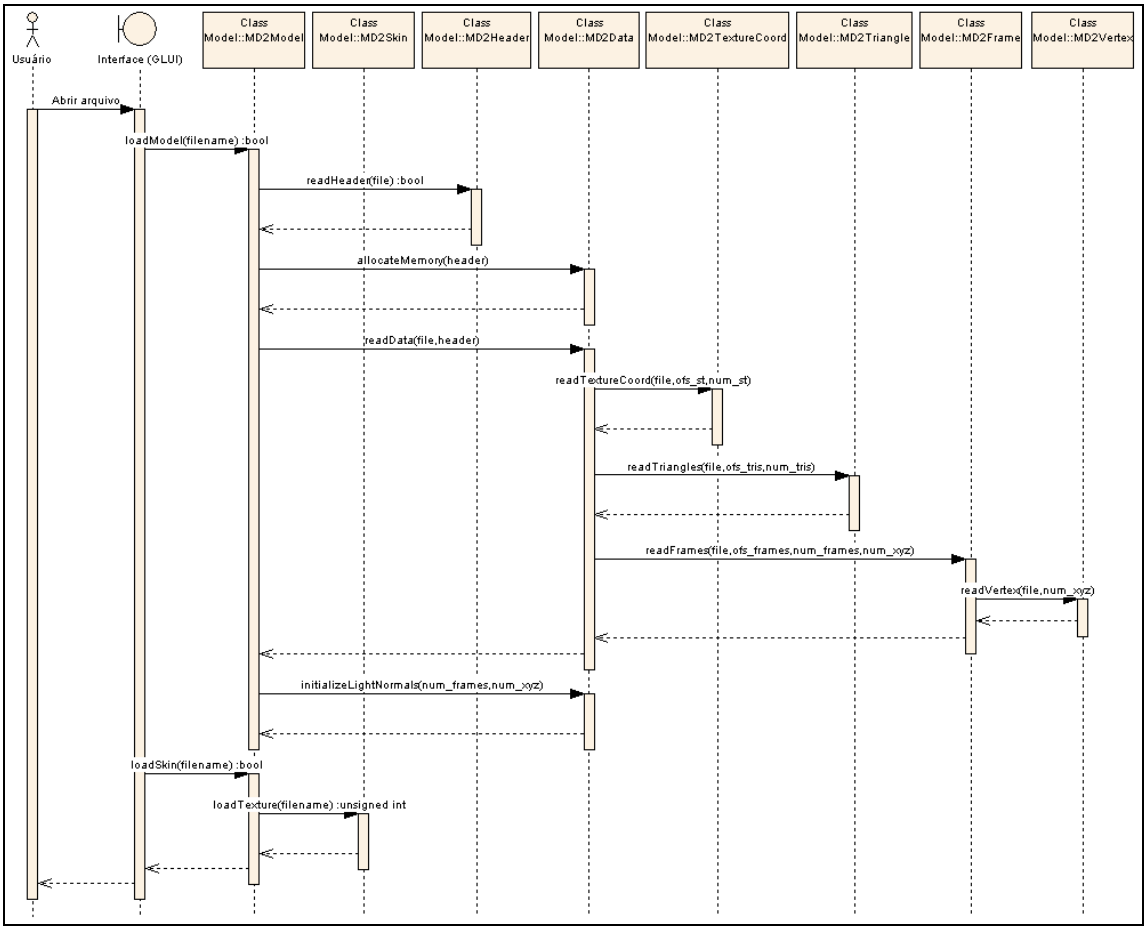



Figura 15 – Diagrama de seqüência da leitura do arquivo MD2

O carregamento do modelo MD2 tem início com a ação de abrir arquivo executada pelo usuário da ferramenta. Neste momento a interface do sistema executa uma chamada a classe `MD2Model` (através do método `loadModel(filename))` passando via parâmetro o caminho concatenado ao nome do arquivo a ser aberto (ou somente o nome do arquivo caso esse esteja no mesmo local da ferramenta). A classe `MD2Model` verifica a existência do arquivo e na seqüência executa uma chamada para a classe `MD2Header` solicitando que a classe execute o método de leitura do cabeçalho. Na seqüência é realizada a chamada do método `allocateMemory(header)` da classe `MD2Data` enviando como parâmetro o cabeçalho lido no passo anterior. O método executa a alocação de memória dos dados utilizando informações do cabeçalho (já que os dados possuem tamanho dinâmico). Após a alocação de memória para os dados é executada a chamada de leitura dos dados através do método `readData(file, header)` com a referência para o arquivo lido e o cabeçalho como parâmetros na classe `MD2Data`. O método então solicita para as classes `MD2TextureCoord`, `MD2Triangle` e `MD2Frame` que realizem leitura dos dados de coordenadas de textura, triângulos e *frames* respectivamente. Os *frames* são responsáveis pela leitura de seus vértices. Após a leitura dos

dados do arquivo MD2 é realizada a chamada do método `initializeLightNormals(num_frames, num_xyz)` na classe `MD2Data` para inicializar uma lista com os índices dos vetores normais utilizados no modelo. Por fim, a interface chama o método `loadSkin(filename)` da classe `MD2Model`, passando via parâmetro o caminho concatenado ao o nome do arquivo de imagem a ser aberto (ou somente o nome do arquivo de imagem caso esse esteja no mesmo local da ferramenta). Na seqüência a classe `MD2Model` efetua a chamada do método `loadTexture()` presente na classe `MD2Skin`, que inicia a leitura do arquivo de imagem para utilizar como textura do modelo 3D.

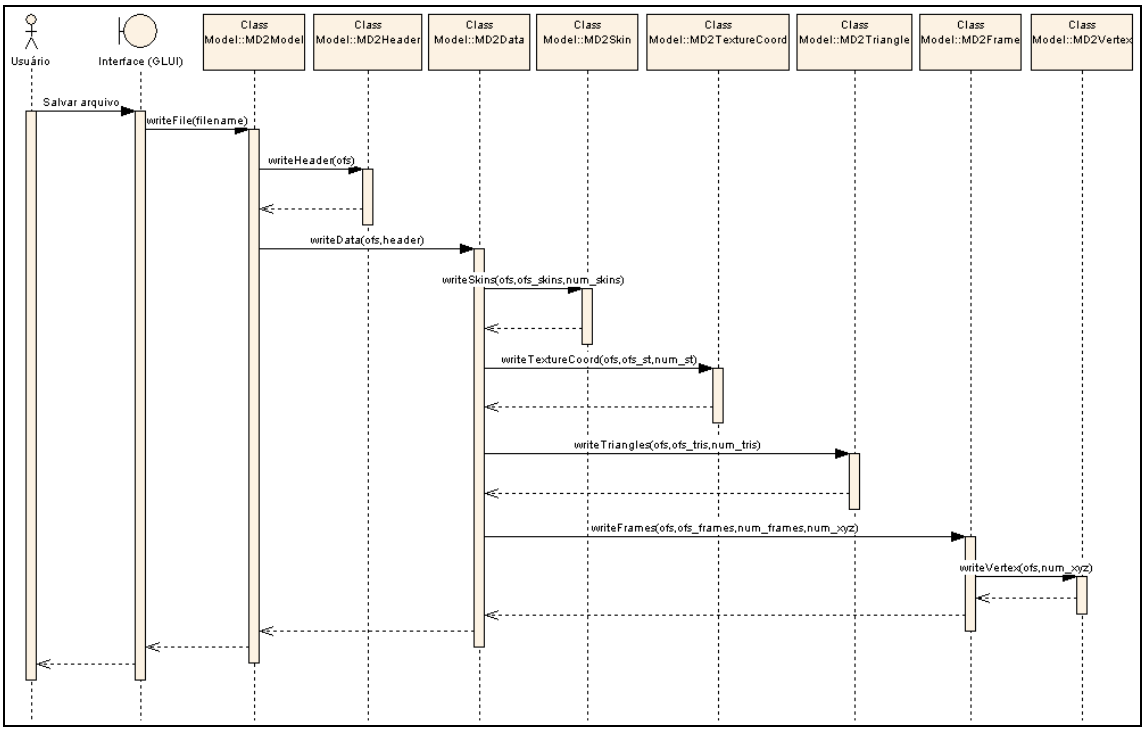



Figura 16 – Diagrama de seqüência da escrita do arquivo MD2

A figura 16 representa o diagrama de seqüência que demonstra os passos necessários para o salvamento do arquivo MD2. O salvamento tem início com a ação do usuário de salvar o arquivo, através da interface da ferramenta, que procede requisitando o método de escrita de arquivo `writeFile(filename)` da classe `MD2Model`. Este método realiza a escrita do cabeçalho do arquivo com o método `writeHeader(file)` da classe `MD2Header` e a escrita dos dados do arquivo com o método `writeData(file, header)` da classe `MD2Data`. Assim a classe `MD2Data` dá seqüência a um conjunto de chamadas de escrita de dados, das classes `MD2Skin`, que realiza a escrita dos dados de textura; `MD2TextureCoord`, grava as informação de coordenadas de textura no arquivo. E `MD2Triangle`, que salva no arquivo as informações referentes aos triângulos do modelo. Por fim da `MD2Frame` que realiza a gravação dos dados do *frame*, incluindo os vértices do modelo.

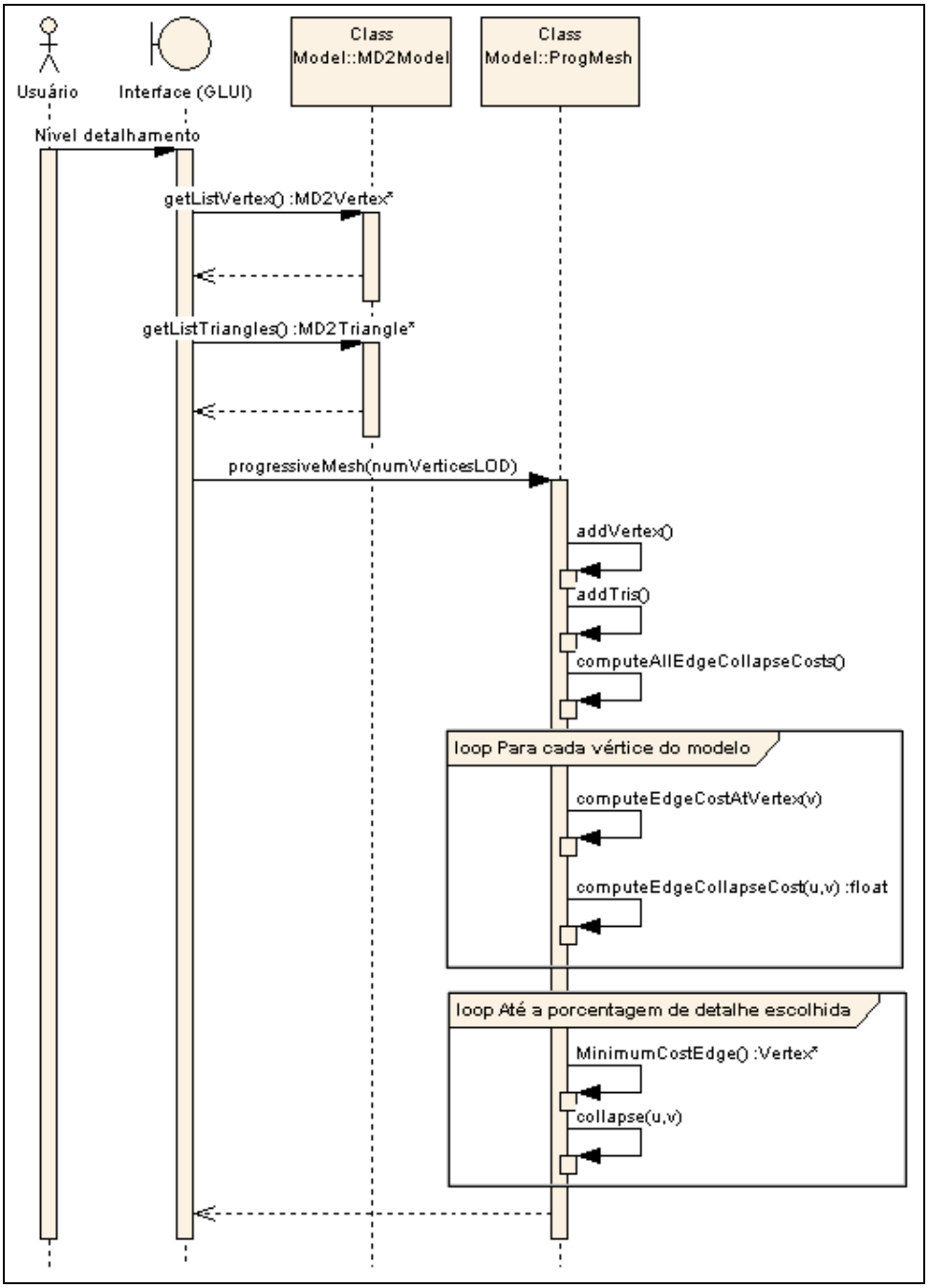



Figura 17 – Diagrama de seqüência da simplificação da malha de polígonos

O diagrama de seqüencia representado na figura 17 mostra o conjunto de chamadas realizadas para que ocorra a simplificação da malha de polígonos. Como nos diagramas de seqüência anteriores. Os procedimentos têm início por meio de uma ação executada pelo usuário da ferramenta que seleciona o nível de detalhamento desejado e confirma a execução da simplificação. Após a confirmação do usuário, a interface do sistema solicita ao modelo lido a sua lista de vértices e triângulos através dos métodos `getListVertex()`. E `getListTriangles()` da classe `MD2Model`. Logo que a interface recebe a lista de vértices e

triângulos do modelo, ela prossegue utilizando a classe do algoritmo de simplificação, `ProgMesh`; realizando a chamada do método `progressiveMesh(numVerticesLOD)`. Esse método efetua as chamadas dos método `addVetex()` e `addTris()` que realizam a adição dos vértices e triângulos ao controle interno do algoritmo de simplificação, e na seqüência o método `computeAllEdgeCollapseCosts()`, com a finalidade de computar previamente o custo de todas as arestas do modelo. Assim para cada vértice do modelo é chamada a função `computeEdgeCostsAtVertex(v)`. Esta função computa o custo do vértice atual com todos os seus vértices vizinhos, encontrando assim o custo de cada aresta. Para computar o custo de cada aresta encontrada é chamado o método `computeEdgeCollapseCost(u, v)`, que realiza o cálculo do custo. Assim que definido um custo para todas as arestas do modelo, o algoritmo realiza a simplificação até que a condição de nível de detalhe definida pelo usuário seja satisfeita. A simplificação ocorre através dos métodos `minimumCostEdge()` que retorna a aresta de menor custo e `collapse(u, v)` que realiza o colapso da aresta retornada pelo primeiro método, é também recalculado o custo de todas as arestas vizinhas a aresta que sofreu o colapso.

## 3.3 IMPLEMENTAÇÃO

A seção de implementação comenta como foi realizado o desenvolvimento da ferramenta com base na especificação, apresentando trechos de código fonte, fórmula de custo da simplificação, ferramentas e bibliotecas utilizadas, operacionalidade do software e testes realizados.

### 3.3.1 Tecnologias utilizadas

Foram utilizadas as seguintes ferramentas na etapa de implementação e testes do presente trabalho:

a) microsoft Visual Studio 2005: plataforma de desenvolvimento que oferece suporte a linguagem C++;
b) linguagem C++: linguagem de programação conhecida e difundida pelo seu alto

desempenho em aplicações gráficas;

c) OpenGL: *Application Programming Interface* (API) para hardware gráfico utilizada para geração de aplicações gráficas bidimensionais ou tridimensionais;
d) *openGL Utility Toolkit* (GLUT): conjunto de funções previamente definidas para facilitar o uso no desenvolvimento de aplicações OpenGL;
e) *openGL User Interface* (GLUI): biblioteca destinada a desenvolvimento de interface com usuário, trabalha em conjunto com GLUT;
f) arquivos MD2: especificação de formato de arquivo binário para armazenamento de um modelo 3D;
g) eclipse: plataforma de desenvolvimento que oferece suporte a linguagem de programação Java, empregado na realização dos testes na M3GE;
h) *Java 2 Micro Edition* (J2ME): linguagem de programação java para dispositivos móveis (ESTÁCIO, 2006, p. 33), utilizada para realização dos testes na M3GE;
i) M3GE: motor de jogos em desenvolvimento no DSC da FURB (GOMES; PAMPLONA, 2005), utilizado para realização dos testes;
*j)* *Sun Java Wireless Toolkit*: pacote de desenvolvimento que integra um emulador de aparelhos celulares e utiliza a J2ME (ESTÁCIO, 2006, p. 33), utilizado nos testes;
k) eclipseME *plugin*: utilizado para união do *Sun Java Wireless Toolkit* à plataforma Eclipse (ESTÁCIO, 2006, p. 33), também utilizado na confecção dos testes.

### 3.3.2 Implementação do código fonte

Nesta seção são apresentados trechos de códigos fonte explicando o funcionamento da ferramenta. São abordados tópicos como leitura, escrita e armazenamento do modelo MD2, assim como sua iluminação e sua renderização na interface. Também são abordadas funcionalidades de visualização do modelo na tela e criação da interface com o usuário a partir da GLUT e GLUI.

Na seqüência, é mostrado o algoritmo de nível de detalhamento utilizado para gerar a malha progressiva do modelo, apresentando etapas como o preenchimento das estruturas internas do algoritmo com dados da malha do modelo original. É explicado também o cálculo do custo das arestas para a escolha da aresta com o menor custo e o colapso da aresta com menor custo.

#### 3.3.2.1 Leitura do arquivo MD2

Como visto no capítulo da fundamentação teórica o arquivo MD2 é um arquivo binário, composto de cabeçalho e dados. Para efetuar a leitura do arquivo, optou-se por utilizar as funções `open()` que abre o arquivo, `seekg()` que posiciona o cursor para leitura em determinada posição do arquivo; `read()` que efetua a leitura de um bloco de dados em *bytes* e `close()` que fecha o arquivo implementadas na classe `fstream` nativa do C++.

O quadro 3 exemplifica o uso de algumas funções comentadas para a leitura dos *frames* do modelo 3D, onde a variável *file* representa um ponteiro para o arquivo aberto, as variáveis `ofs_frames` o deslocamento dentro do arquivo para os *frames* e `num_frames` que é o número de frames armazenados no arquivo.

A segunda linha do quadro 3 posiciona o cursor de leitura a partir do inicio do arquivo na posição onde estão gravados os dados dos *frames*. A linha 3 representa um *loop* para cada *frame*, onde é lido a escala, translação e nome do *frame*, respectivamente nas linhas 4, 5 e 6. A linha 7 efetua a chamada do método `readVertex(file, num_xyz)` para a leitura dos vértices passando via parâmetro o ponteiro do arquivo aberto e o número de vértices.

```
1.  void MD2Frame::readFrames( std::ifstream* file, int ofs_frames,
                               int num_frames, int num_xyz ) {
2.      file->seekg( ofs_frames, std::ios::beg );
3.      for (int i = 0; i < num_frames; ++i) {
4.          file->read (reinterpret_cast<char *>(this[i].scale),
                        sizeof (vec3));
5.          file->read (reinterpret_cast<char *>(this[i].translate),
                        sizeof (vec3));
6.          file->read (reinterpret_cast<char *>(this[i].name),
                        sizeof (char) * 16);
7.        this[i].verts->readVertex(file, num_xyz);
8.      }
9.  }
```

Quadro 3 – Método de leitura dos *frames* do modelo 3D

#### 3.3.2.2 Escrita do arquivo MD2

A escrita do arquivo MD2 é similar a leitura do mesmo. Para efetuar a gravação do arquivo em disco é necessária a classe `fstream` nativa do C++ que permite acesso as funções `open()` que abre o arquivo; `seekp()` que posiciona o cursor para escrita em determinada posição do arquivo; `write()` que efetua a gravação de um bloco de dados em *bytes* e `close()`

que fecha o arquivo.

No quadro 4 é exibido um trecho do código fonte onde é realizada a escrita dos *frames n*o arquivo MD2, onde a variável `ofs` representa um ponteiro para o arquivo aberto, as variáveis `ofs_frames` o deslocamento dentro do arquivo para gravar os *frames* e `num_frames` que é o número de frames a ser gravados no arquivo.

A linha 2 tem como objetivo posicionar o cursor de escrita na posição onde serão gravados os frames, a partir do início do arquivo. Na linha subseqüente é realizada para cada *frame* do modelo o processo de gravação de seus dados como escala, translação e nome, (respectivamente nas linhas 4, 5 e 6). A sétima linha efetua a chamada do método de gravação dos vértices do *frame* através do método `writeVertex(ofs, num_xyz)` passando via parâmetro o ponteiro do arquivo aberto para escrita e o número de vértices a serem gravados.

```
1.  void MD2Frame::writeFrames( std::ofstream* ofs, int ofs_frames,
                                int num_frames, int num_xyz ) {
2.    ofs->seekp(ofs_frames, std::ios::beg);
3.    for (int i = 0; i < num_frames; ++i) {
4.        ofs->write (reinterpret_cast<char *>(this[i].scale),
                      sizeof (float)*3);
5.        ofs->write (reinterpret_cast<char *>(this[i].translate),
                      sizeof (float)*3);
6.        ofs->write (reinterpret_cast<char *>(this[i].name),
                      sizeof (char)*16);
7.        this[i].verts->writeVertex(ofs, num_xyz);
8.    }
9. }
```

Quadro 4 – Método de gravação dos *frames* do modelo 3D

### 3.3.2.3 Armazenamento da estrutura do modelo MD2

Para armazenar a estrutura do modelo MD2 em memória, foram utilizadas classes que espelham a especificação do arquivo MD2, são elas:

a) `MD2Model` é a classe que simboliza o modelo MD2. Ela armazena referências para as classes `MD2Header` e `MD2Data`, além de um atributo denominado *scale* (do tipo *float*) onde é armazenado a escala geral do modelo a ser desenhado e outro atributo denominado `textureId` (do tipo `unsigned int`) onde é armazenado o identificador da textura que está sendo utilizada pelo modelo;

b) `MD2Header` é a classe que representa o cabeçalho do modelo e armazena todos os atributos pertencentes ao cabeçalho do arquivo MD2;

c) `MD2Data` é a classe que representa os dados do modelo, possui referências para as

classes `MD2Frame`, `MD2Skin`, `MD2Triangle` e `MD2TextureCoord`, também dois atributos do tipo *array* de inteiros para os comandos OpenGL e índices dos vetores normais, que são os dados encontrados no arquivo MD2;

d) `MD2Frame` é a classe que representa o frame, nela estão presentes informações como escala e translação do tipo *float*, o nome do *frame* e uma lista de classes `MD2Vertex` que contém os dados dos vértices desse *frame*;

e) `MD2Vertex` é a classe que representa um vértice, armazenando as posições x, y e z e um índice para o seu vetor normal;

f) `MD2Skin` é a classe que representa a textura do modelo, guarda o nome das *skin* do modelo MD2, cada nome com tamanho de 64 *bytes*;

g) `MD2Triangle` é a classe que representa um triangulo da malha de polígonos do modelo, é composta por dois *arrays* de três posições do tipo `unsigned short` um dos *arrays* contém três índices de vértices do modelo que pertencem a este triângulo e o outro *array* contém três índices das coordenadas de textura que irão fixar nesse triângulo;

h) `MD2TextureCoord` é a classe que representa as coordenadas de textura do modelo, armazena dois atributos denominados de *s* e *t* do tipo *short* que são as coordenadas de textura para referenciar o arquivo de imagem como textura do modelo.

A classe `MD2Model` gerencia as classes `MD2Header` e `MD2Data` que representam o cabeçalho e dados do arquivo MD2. Para o carregamento do modelo MD2 é utilizado o método `loadModel(const char *filename)` da classe `MD2Model`, que recebe como parâmetro o caminho do arquivo a ser aberto. No quadro 5 está representado esse método.

```
1.  bool MD2Model::loadModel(const char *filename) {
2.    std::ifstream file;
3.    file.open( filename, std::ios::in | std::ios::binary );
4.    if( file.fail() )
5.      return false;
6.    if (!header->readHeader(&file)) {
7.      file.close();
8.      return false;
9.    }
10.   data->allocateMemory(header);
11.   data->readData(&file, header);
12.   data->initializeLightNormals(header->num_frames, header->num_xyz);
13.   anim->num_frames = header->num_frames;
14.   file.close();
15.   return true;
16. }
```

Quadro 5 – Método de que inicia o carregamento do modelo MD2

No quadro 5 a terceira linha abre o arquivo para a leitura, enquanto a quarta linha

verifica se a operação de abrir o arquivo foi executada com sucesso. Na linha 6 se inicia a leitura do cabeçalho através do método `readHeader(&file)` da classe `MD2Header`, enviando por parâmetro a referência para o arquivo. Nesta linha também é realizada uma conferência, pois na leitura do cabeçalho é verificado se o arquivo lido é um arquivo MD2. A linha 10 mostra a chamada para alocação de memória para os dados, o cabeçalho é enviado como parâmetro nesta chamada. Através do cabeçalho é possível saber a quantidade de *bytes* necessários para alocação. Na seqüência o método `readData(&file, header)` da classe `MD2Data` é chamado para a leitura dos dados. Por parâmetro é enviado a referência para o arquivo e o cabeçalho. Como o arquivo é binário o cabeçalho tem o objetivo de auxiliar na busca pelos dados. A linha 12 mostra uma chamada de método para inicialização de um *array* com todos os vetores normais do modelo.

Para um melhor entendimento do funcionamento do método de carregamento do arquivo MD2, trechos de códigos fonte das chamadas de métodos realizados no carregamento do arquivo MD2 citados anteriormente são apresentados em quadros na seqüência com explicações e maiores detalhes.

O quadro 6 exemplifica o método de leitura e validação do cabeçalho através do método `readHeader(&file)` da classe `MD2Header`. Como é mostrado no quadro 6 a linha 2 executa a leitura do cabeçalho do arquivo enquanto a linha 3 faz a validação, verificando se o arquivo lido é um arquivo MD2.

```
1.  bool MD2Header::readHeader( std::ifstream* file) {
2.      file->read( (char *)this, sizeof( MD2Header ) );
3.      if( (ident != MD2_IDENT) && (version != MD2_VERSION) ) {
4.          return false;
5.      }
6.      return true;
7.  }
```

Quadro 6 – Método leitura e validação do cabeçalho de arquivo MD2

No quadro 7 é mostrado a alocação de memória para o dados, a partir dos dados lidos no cabeçalho do arquivo MD2, por meio do método `allocateMemory(MD2Header* header)` que recebe por parâmetro o cabeçalho do arquivo e aloca memória informações de textura, coordenadas de textura, triângulos, vetores normais, frames e vértices.

```
1.  void MD2Data::allocateMemory( MD2Header* header ) {
2.       skins = new MD2Skin[ header->num_skins ];
3.       textureCoord   = new MD2TextureCoord[ header->num_st ];
4.       triangles = new MD2Triangle[ header->num_tris ];
5.       lightnormals   = new int[ header->num_xyz * header->num_frames
    ];
6.       frames   = new MD2Frame[ header->num_frames ];
7.       for (int i = 0; i < header->num_frames; ++i) {
8.            frames[i].verts = new MD2Vertex[ header->num_xyz ];
9.       }
10. }
```

Quadro 7 – Método para alocação de memória para os dados

A leitura dos dados é realizada com a chamada do método `readData(&file, header)` da classe `MD2Data` onde é enviado via parâmetro a referência para o arquivo lido e o cabeçalho do arquivo que auxilia a leitura dos dados. Como representado no quadro 8, onde é realizada a leitura das coordenadas de textura (linha 2), dos triângulos da malha de polígonos (linha 3) e dos *frames* com seus respectivos vértices (linha 4).

```
1.  void MD2Data::readData( std::ifstream* file, MD2Header* header ) {
2.      textureCoord->readTextureCoord(file, header->ofs_st,
                                       header->num_st);
3.      triangles->readTriangles(file, header->ofs_tris,
                                 header->num_tris);
4.      frames->readFrames(file, header->ofs_frames, header->num_frames,
                           header->num_xyz);
5.  }
```

Quadro 8 – Método leitura dos dados do arquivo MD2

#### 3.3.2.4 Animação do modelo

O arquivo MD2 tem a capacidade de armazenar quadros (*frames*) para efetuar a animação do modelo 3D. Assim o modelo é composto por um conjunto de quadros-chave, onde cada quadro-chave apresenta uma etapa da animação do modelo. Para que seja possível visualizar uma animação de qualidade no modelo, é utilizado um cálculo de interpolação entre os quadros-chave.

A classe `MD2Anim` tem controle dos eventos de animação, gerando o cálculo de interpolação e adequando a taxa de quadros exibidos por segundo conforme a necessidade do usuário. O quadro 9 mostra o cálculo utilizado para gerar o valor de interpolação. Sua chamada é realizada através do método `animate(float time)` que recebe via parâmetro o tempo atual em segundos.

```
1.  void MD2Anim::animate( float time ) {
2.      curr_time = time;
3.      if( curr_time - old_time > (1.0 / fps) ) {
4.        curr_frame = next_frame;
5.        next_frame++;
6.        if( next_frame > endframe )
7.          next_frame = startframe;
8.        old_time = curr_time;
9.      }
10.     if( curr_frame > (num_frames - 1) )
11.       curr_frame = 0;
12.     if( next_frame > (num_frames - 1) )
13.       next_frame = 0;
14.     interpol = fps * (curr_time - old_time);
15. }
```

Quadro 9 – Método utilizado na animação do modelo 3D

#### 3.3.2.5 Iluminação do modelo

Para criar a iluminação no modelo 3D, é utilizado um vetor normal que está associado aos vértices do modelo e indica para qual direção está apontando a face do vértice. O vetor normal geralmente é calculado antes da renderização do modelo, contudo para os arquivos MD2 existe uma lista de vetores normais pré-calculados. Assim o arquivo apenas armazena em cada vértice o índice que indica qual vetor normal pré-calculado deverá ser utilizado para esse vértice.

#### 3.3.2.6 Renderização do modelo MD2 no *canvas*

A renderização é realizada por meio do OpenGL e pode ser efetuada de duas formas; A primeira através de comandos OpenGL que estão presentes no arquivo MD2, que são utilizados como uma forma de renderização do modelo 3D através das primitivas GL_TRIANGLE_FAN e GL_TRIANGLE_STRIP. Os comandos OpenGL têm como vantagem a velocidade superior a outra técnica na renderização, isso devido ao fato de que para renderizar mais de um triângulo, essas primitivas desenham apenas um vértice onde esses estejam sobrepostos. Para isso é dada um seqüência de desenho dos vértices, contudo caso ocorra alguma modificação na malha de polígonos as primitivas deixarão de funcionar corretamente, em decorrência da mudança dos vértices. Sua utilização é opcional, e não são utilizadas na renderização do modelo MD2 na M3GE.

A segunda forma de renderização é através primitiva GL_TRIANGLES, uma técnica muito comum que utiliza a lista de triângulos do modelo 3D. Esta técnica é utilizada pela M3GE. Optou-se então pela renderização através da primitiva GL_TRIANGLES. O quadro 10 ilustra trechos do método de renderização `renderFrame()` especificado na classe `MD2Model`.

```
1.  void MD2Model::renderFrame( void ) {
       ...
2.     glBindTexture( GL_TEXTURE_2D, textureId );
3.     glBegin (GL_TRIANGLES);
4.     for (int i = 0; i < header->num_tris; ++i) {
5.       for (int j = 0; j < 3; ++j) {
6.          MD2Frame *pFrameA = &data->frames[anim->curr_frame];
7.          MD2Frame *pFrameB = &data->frames[anim->next_frame];

8.          MD2Vertex *pVertA =
                        &pFrameA->verts[data->triangles[i].vertex[j]];
9.          MD2Vertex *pVertB =
                        &pFrameB->verts[data->triangles[i].vertex[j]];

            ...

10.         MD2TextureCoord pTexCoords =
                       data->textureCoord[ data->triangles[i].st[j] ];
11.         GLfloat s = static_cast<GLfloat>(pTexCoords.s) /
                                                  header->skinwidth;
12.         GLfloat t = static_cast<GLfloat>(pTexCoords.t) /
                                                 header->skinheight;
13.         glTexCoord2f (s, t);

14.         glNormal3fv( data->anorms[
                data->lightnormals[ data->triangles[i].vertex[j] ] ] );

15.         vec3 vecA, vecB, v;

16.         vecA[0] = pFrameA->scale[0] * pVertA->v[0] +
                                                  pFrameA->translate[0];
17.         vecA[1] = pFrameA->scale[1] * pVertA->v[1] +
                                                  pFrameA->translate[1];
18.         vecA[2] = pFrameA->scale[2] * pVertA->v[2] +
                                                  pFrameA->translate[2];
19.         vecB[0] = pFrameB->scale[0] * pVertB->v[0] +
                                                  pFrameB->translate[0];
20.         vecB[1] = pFrameB->scale[1] * pVertB->v[1] +
                                                  pFrameB->translate[1];
21.         vecB[2] = pFrameB->scale[2] * pVertB->v[2] +
                                                  pFrameB->translate[2];

22.         v[0] = (vecA[0] + anim->interpol *
                                        (vecB[0] - vecA[0])) * scale;
23.         v[1] = (vecA[1] + anim->interpol *
                                        (vecB[1] - vecA[1])) * scale;
24.         v[2] = (vecA[2] + anim->interpol *
                                        (vecB[2] - vecA[2])) * scale;

25.         glVertex3fv (v);

26.       }
27.    }
28.    glEnd();
       ...
29. }
```

Quadro 10 – Método de renderização do modelo MD2

Como ilustrado no quadro 10 a renderização inicia referenciando o identificador gerado para o arquivo de imagem lido (linha 2). Na seqüência é selecionado cada vértice atual

e do *frame* seguinte de cada triângulo do modelo (linhas 6 a 9). Utilizando as coordenadas de textura presentes nos triângulos a textura é fixada ao modelo (linhas 10 a 13). Na linha 14 é efetuada a iluminação do modelo passando ao OpenGL o vetor normal do vértice. As linhas 15 a 21 aplicam um conjunto de transformações de escala e translação aos vértices. Procedendo as transformações é realizada a interpolação dos vértices utilizando o valor de interpolação gerado pela classe `MD2Anim`, assim desenhando o vértice interpolado no *canvas* (linhas 22 – 25).

#### 3.3.2.7 Preparação e execução do algoritmo de simplificação da malha de polígonos

A classe `ProgMesh` é responsável por aplicar técnicas de nível de detalhamento em um modelo 3D lido. A intensidade da simplificação a ser aplicada no modelo é informada pelo usuário. A execução do algoritmo possui três etapas principais, são elas as etapas de preparação, execução e finalização. A preparação consiste em fornecer os dados da malha de polígonos ao algoritmo de simplificação, esses dados são o número de vértices, número de triângulos, a lista de vértices e triângulos. A execução abrange todas as operações de simplificação da malha de polígonos, incluindo a definição de custos, colapso de arestas, reajuste das entradas na lista triângulos, entre outras. A finalização disponibiliza a nova malha de polígonos simplificada, também chamada malha progressiva, para renderização do modelo simplificado.

No quadro 11 é mostrado a função `progressiveMesh()` que realiza a chamada do algoritmo de simplificação da malha de polígonos. Na linha 2 se refere ao parâmetro de simplificação, esse parâmetro define a intensidade de simplificação da malha de polígonos, é calculado com base no percentual informado pelo usuário na interface do programa. Na linha 8 é realizada chamada da função de execução do algoritmo enviando o parâmetro de simplificação obtido na linha 2.

```
1.  void progressiveMesh( void ) {
2.     int numVerticesLOD = model->getNumVertex()*
                          (gluiVariables.percentLOD->get_int_val()*0.01);
3.     if (progMesh) {
4.        delete(progMesh);
5.     }
6.     progMesh = new ProgMesh();
7.     getDadosModelo(progMesh->vertModelo, progMesh->trisModelo);
8.     progMesh->progressiveMesh(numVerticesLOD);
9.     infoDisplay.numTriangles = progMesh->trisModelo->num;
10.    setDadosModelo(progMesh->trisModelo);
11. }
```

Quadro 11 – Preparação para execução do algoritmo de LOD

No quadro 12 a função `getDadosModelo()` recebe via parâmetro uma referência para as listas de vértice e triângulo do algoritmo de simplificação. As linhas 2 e 7 solicitam as listas de vértices e triângulos do modelo MD2, os dados de vértices e triângulos são fornecidos ao algoritmo de simplificação nas linhas 5 e 14.

```
1.  void getDadosModelo(List<Vector>*vertModelo,
                        List<Tridata>*trisModelo) {
2.    MD2Vertex *verts = model->getListVertex();
3.    for(int i=0; i < model->getNumVertex(); i++) {
4.        Vector v = Vector(verts[i].v[0],
                            verts[i].v[1],
                            verts[i].v[2]);
5.        vertModelo->Add(v);
6.    }
7.    MD2Triangle *tris = model->getListTriangles();
8.    for(int i=0; i < model->getNumTriangles(); i++) {
9.        Tridata t;
10.       for (int j=0; j < 3; j++) {
11.          t.v[j] = tris[i].vertex[j];
12.          t.st[j] = tris[i].st[j];
13.       }
14.       trisModelo->Add(t);
15.   }
16. }
```

Quadro 12 – Fornecimento dos dados da malha para o algoritmo de LOD

A execução do algoritmo de LOD tem início no método `progressiveMesh(int numVerticesLOD)` da classe `ProgMesh` LOD, onde é pré-computado o custo de todas arestas do modelo e simplificado o modelo até atingir o desejo de simplificação do usuário recebido por parâmetro na variável `numVerticesLOD`.

### 3.3.2.8 Fórmula de custo do algoritmo de LOD

O quadro 13 apresenta a tradução da equação que formula o custo de colapso da aresta para a linguagem C++. Na linha 3 é definido o tamanho da aresta. As linhas 6 a 10 encontram

os triângulos que possuem ambos os vértices `u` e `v` e as linhas 11 a 20 encontram a curvatura da aresta, assim o custo da aresta é definido pelo tamanho multiplicado pela curvatura da mesma.

```
1.  float ProgMesh::computeEdgeCollapseCost(Vertex *u,Vertex *v) {
2.     int i;
3.     float edgelength = magnitude(v->position - u->position);
4.     float curvature=0;
5.     List<Triangle *> sides;
6.     for(i=0;i<u->tri.num;i++) {
7.        if(u->tri[i]->hasVertex(v)){
8.           sides.Add(u->tri[i]);
9.        }
10.    }
11.    for(i=0;i<u->tri.num;i++) {
12.       float mincurv=1;
13.       for(int j=0;j<sides.num;j++) {
14.          float dotprod = u->tri[i]->normal ^ sides[j]->normal;
15.          if ((1-dotprod)/2.0f<mincurv)
16.             mincurv = (1-dotprod)/2.0f;
17.       }
18.       if (mincurv>curvature)
19.          curvature = mincurv;
20.    }
21.    return edgelength * curvature;
22. }
```

Quadro 13 – Tradução da equação de custo da aresta para código fonte

### 3.3.2.9 Colapso de uma aresta

O colapso de uma aresta é realizado “colidindo” um de seus vértices contra o outro, o vértice colidido é removido, assim as arestas que incidiam antes sobre esse vértice passam agora a incidir sobre o vértice restante, o custo para colapso dessas arestas é recalculado. Nesta operação o vértice resultante da colisão não sofre qualquer alteração em seus atributos, permanecendo seus atributos originais como as coordenadas de textura e vetores normais. O quadro 14 exemplifica essa operação utilizada no algoritmo de LOD, através do método de `collapse()`.

```
1.  void ProgMesh::collapse(Vertex *u,Vertex *v) {
2.     if(!v) {
3.       delete u;
4.       return;
5.     }
6.     int i;
7.     List<Vertex *>tmp;
8.     for(i=0;i<u->neighbor.num;i++) {
9.        tmp.Add(u->neighbor[i]);
10.    }
11.    for(i=u->tri.num-1;i>=0;i--) {
12.       if(u->tri[i]->hasVertex(v)) {
13.          delete(u->tri[i]);
14.       }
15.    }
16.    for(i=u->tri.num-1;i>=0;i--) {
17.       u->tri[i]->replaceVertex(u,v);
18.    }
19.    delete u;
20.    for(i=0;i<tmp.num;i++) {
21.       computeEdgeCostAtVertex(tmp[i]);
22.    }
23. }
```

Quadro 14 – Método de colapso de uma aresta

#### 3.3.2.10 Desenvolvimento da interface da ferramenta

Para o desenvolvimento da interface com o usuário da ferramenta, optou-se por utilizar a GLUT que fornece recursos pra criação do *canvas*, onde é renderizado o modelo 3D, além de disponibilizar funções para eventos de teclado e mouse. Também foi utilizada a GLUI, uma biblioteca destinada ao desenvolvimento de interface com usuário, que fornece a opção de criar componentes de interface como botões, campos de texto, entre outros. No quadro 15 é apresentado um trecho de código fonte, onde é criado uma barra de ferramentas para agrupar os componentes da interface (linha 1). Na seqüência são criados dois componentes do tipo *panel* que agrupam elementos como botões, campo de texto, *panels*, entre outros (linha 2 e 4). Na linha 5 é criado uma campo de texto com descrição de “Nome MD2:” e incluído ao *panel* criado na linha 4.

```
1.  GLUI *glui = GLUI_Master.create_glui( "Barra de ferramentas", 0, 400,
    50 );
2.  GLUI_Panel *objPanelFile = new GLUI_Panel( glui, "Arquivo" );
3.  objPanelFile->set_alignment(GLUI_ALIGN_LEFT);
4.  GLUI_Panel *objPanelOpen = new GLUI_Panel( objPanelFile, "Abrir" );
5.  gluiVariables.fileName = glui->add_edittext_to_panel(objPanelOpen,
    "Nome md2:", GLUI_EDITTEXT_TEXT);
```

Quadro 15 – Criação de componentes utilizando a GLUI

### 3.3.3 Operacionalidade da implementação

Ao executar-se a ferramenta são apresentadas as telas ilustradas na figura 18, na barra de ferramentas é necessário que o usuário informe os caminhos concatenados aos nomes do arquivo MD2 e do arquivo de textura, Para textura é permitido o uso de arquivo de imagem PCX, TGA ou BMP. Após informar os caminhos corretos, o usuário deve pressionar o botão Abrir.

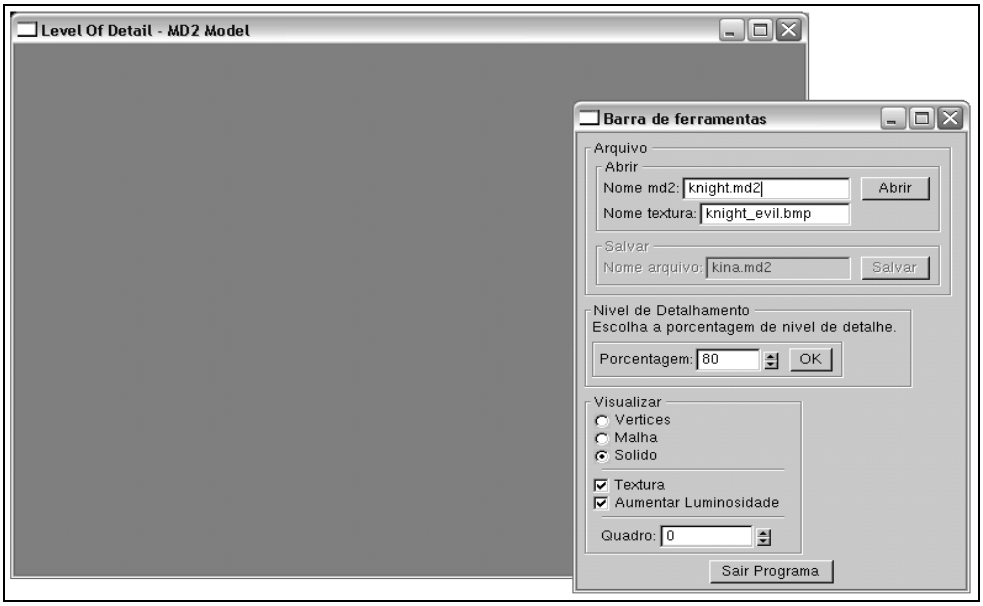



Figura 18 – Telas inicias da ferramenta

Após pressionar a opção para abrir o arquivo MD2 e arquivo de textura, o modelo é renderizado no *canvas*, a janela titulada de Level Of Detail – MD2 Model. Também são exibidas informações sobre a malha de triângulos e a taxa de quadros por segundo. Após o modelo ser renderizado, o usuário da ferramenta tem as seguintes opções:

a) executar a redução de nível de detalhamento: o usuário ajusta a porcentagem que de nível de detalhe e pressiona o botão “OK” para confirmar;
b) visualizar vértices: o usuário seleciona a opção “Vértices” para serem apresentados somente os vértices do modelo;
c) visualizar malha: o usuário seleciona a opção “Malha” para que seja apresentada a malha de polígonos do modelo;
d) visualizar sólido: o usuário seleciona a opção “Sólido” para que a malha de polígonos seja preenchida, tomando forma sólida;
e) visualizar textura: o usuário possui a opção de ligar ou desligar a textura do

modelo;

f) aumentar luminosidade: o usuário possui a opção para tornar a luminosidade mais intensificada no modelo;
g) quadro: o usuário escolhe qual o quadro de animação que deseja visualizar no modelo;
h) rotacionar: na área de desenho do modelo o usuário possui a opção de rotacioná-lo, mantendo pressionado o botão esquerdo do mouse e movendo para a direção que deseja rotacionar o modelo;
i) zoom: na área de desenho do modelo o usuário possui a opção afastar ou aproximar a visão do modelo, mantendo pressionado o botão direito do mouse e movendo para cima para aproximar o modelo e para baixo para afastá-lo;
j) sair programa: o usuário possui a opção de fechar a ferramenta.

A figura 19 ilustra as opções de visualização disponibilizadas pela ferramenta em um modelo de um cavaleiro com escudo. Algumas visualizações disponíveis são a de vértices, malha, sólido e textura, apresentados nesta ordem na imagem.

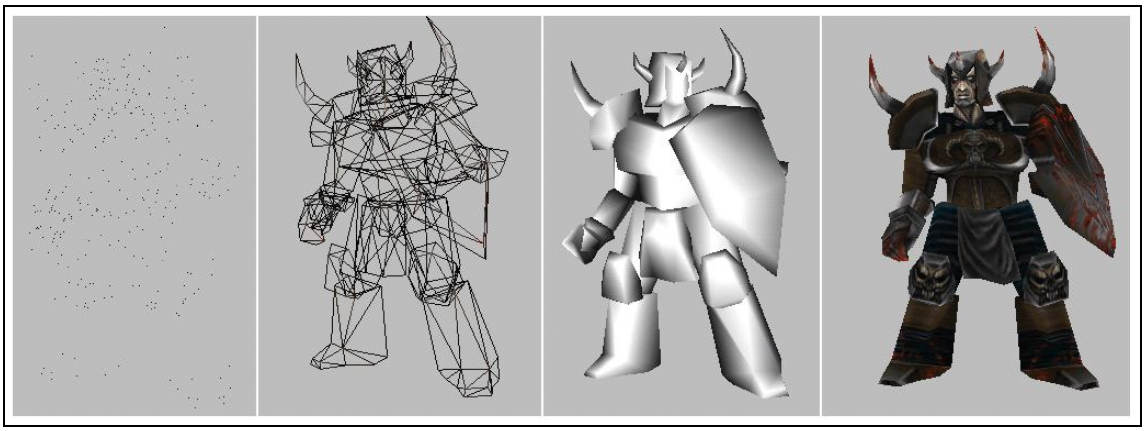


Figura 19 – Opções de visualização disponibilizadas pela ferramenta

Quando o usuário executar o nível de detalhamento no modelo lido, é renderizado o novo modelo simplificado e habilitada a opção para salvar o modelo simplificado em um arquivo MD2. Para salvar o modelo, o usuário deve informar o caminho a ser salvo concatenado ao nome do arquivo escolhido.

## 3.4 RESULTADOS E DISCUSSÃO

O presente trabalho apresentou a implementação de nível de detalhamento de modelos

MD2. Constatou-se que é possível interpretar e realizar uma redução na malha de polígonos dos modelos MD2 de forma eficiente pois a maior parte dos métodos de LOD comportam essa operação em tempo real. A simplificação de modelos MD2 é de grande utilidade, pois devido ao tamanho do visor dos celulares bastante pequenos e suas baixas resoluções, os detalhes não são muito perceptíveis e modelos simplificados em celulares favorecem assim a aplicação um ganho de desempenho notável.

Apesar de funcional, o conceito de nível de detalhamento em arquivos MD2 é complexo, pois os modelos MD2 possuem a característica de nativamente serem leves e possuírem uma malha de polígonos considerada pequena em relação a modelos encontrados em diversos artigos, como o de Hoppe (1996).

A técnica de nível de detalhamento utilizada na presente ferramenta se mostra eficiente ao que se refere a desempenho de execução, um fato importante caso esta técnica seja utilizada em aplicações de tempo real. Contudo, uma característica dessa técnica é não agregar valor a convexidade do modelo. Assim quando um modelo sofrer a ação de nível de detalhamento muito drástica, em determinados casos esse pode “quebrar”, tornando-se desconexo, o que desvaloriza seu reconhecimento geométrico, um exemplo disso pode ser visto na figura 20, onde o circulo vermelho marca a região onde houve a quebra do modelo após a simplificação.

Figura 20 – Quebra da malha de polígonos de um modelo 3D

Foram executados quinze testes, em modelos MD2 como cubo (8 vértices, 12 triângulos), centauro (274 vértices, 536 triângulos), cavaleiro (365 vértices, 634 triângulos),

ogro (358 vértices, 670 triângulos) e pessoa (315 vértices, 590 triângulos), utilizando a técnica de nível de detalhamento empregada nesta ferramenta. Parte dos modelos simplificados como cubo, centauro, cavaleiro medieval e a pessoa apresentaram resultados aceitáveis, tendo como base sua execução em celulares. Contudo em no modelo do ogro o resultado da simplificação não é satisfatório. Isso acontece em alguns modelos devido à fórmula de custo procurar simplificar regiões com triângulos menores e coplanares. Porém o arquivo MD2 é limitado em número de vértices e as regiões que concentram os maiores detalhes são, por exemplo, a cabeça, assim o algoritmo em alguns casos tende a simplificar regiões importantes do modelo.

O software desenvolvido por Estácio (2006) é similar ao módulo de importação de modelos MD2 desta ferramenta, entretanto tal software possui algumas particularidades na leitura do arquivo MD2. Por ser utilizado em dispositivos celulares, é utilizada uma classe para inversão da seqüência de bytes lida e para textura é permitido o uso de arquivo de imagem *Portable Network Graphics* (PNG), o qual não é reconhecido por esta ferramenta.

Na seqüência são apresentados alguns modelos simplificados, utilizando esta ferramenta, que mostraram resultados aceitáveis e modelos que não tiveram um comportamento ideal na execução do algoritmo.

A figura 21 apresenta resultados da simplificação aplicada sobre o modelo que representa um cavaleiro com escudo, onde da esquerda para a direita, a primeira imagem representa o modelo original com 634 triângulos, na segunda imagem é exibido com 488 triângulos, na terceira imagem apresenta o modelo com 346 triângulos e a última imagem da exibe o modelo com apenas 210 triângulos. É possível observar que neste nível de detalhamento o modelo sofre “quebras” em determinadas regiões como comentado anteriormente.

Figura 21 – Simplificações do modelo representado por um cavaleiro de escudo

A figura 22 segue o mesmo modelo da figura 21, onde o modelo da figura é apresentado diferentes níveis de aproximações, quando o modelo está muito longe do observador, a maioria dos detalhes podem ser eliminados. A primeira imagem da esquerda para direita possui 361px de altura por 207px de largura, a segunda possui 202px de altura e 125px de largura, a terceira possui 90px de altura e 60px de largura e a ultima imagem possui 32px de altura e 21px de largura.

Figura 22 – Variação da distância dos mesmos modelos simplificados exibidos na figura 21

A figura 23 ilustra uma situação onde o algoritmo não obtém uma boa simplificação, em um modelo que representa um ogro, devido a região da cabeça possuir menor custo de colapso.

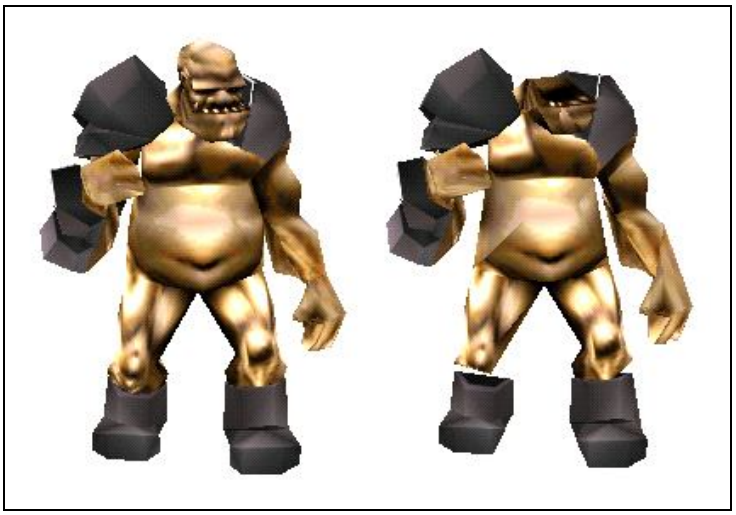


Figura 23 – Simplificação não eficiente de um determinado modelo

A figura 24 mostra a execução da M3GE através de um emulador de celular, apresentando o modelo do cavaleiro com escudo com 50% de simplificação.


Figura 24 – Modelo simplificado carregado na M3GE

Figura 25 – Modelo do cavaleiro em diferentes níveis de detalhe

Houve a execução de cinco testes para análise de comportamento de um dos modelos simplificados na M3GE utilizando um emulador de celulares. O modelo utilizado para realização destes cinco testes foi o cavaleiro em diferentes níveis de detalhamento, sendo eles 100%, 80%, 60%, 40% e 20%, (figura 25). Os testes apresentam resultados obtidos com base nas premissas de número de triângulos do modelo, *Frames Per Second*[5] (FPS), número de *bytes* alocados para o modelo, tamanho do arquivo MD2 e tempo para o carregamento do modelo. A taxa máxima de quadros exibidos por segundo no emulador é de 20FPS. Para os testes que envolvem FPS e tempo gasto para carregar o modelo, foram utilizados vinte modelos idênticos, os quais foram lidos e exibidos pela M3GE. Devido à performance do emulador de celulares ser melhor que o dispositivo celular. Assim permitindo notar a

[5] Os FPS utilizados foram fornecidos pelo emulador de celulares.

diferença entre um nível e outro de detalhe. Os resultados dos testes são apresentados na Tabela 1.

Tabela 1 – Testes realizados na M3GE com modelo cavaleiro com escudo

| **COMPORTAMENTO DE MODELOS COM SIMPLIFICAÇÕES DISTINTAS NA M3GE** | | | | | |
|---|---|---|---|---|---|
| | **Modelo 100% (não simplificado)** | **80%** | **60%** | **40%** | **20%** |
| Triângulos existentes | 634 | 488 | 346 | 210 | 94 |
| FPS (com 20 modelos) | 10 | 13 | 16 | 18 | 20 |
| Número de *bytes* alocados pelo modelo | 36.270 | 33.496 | 30.770 | 28.116 | 25.702 |
| Número de *bytes* do arquivo MD2 | 320.236 | 248.088 | 190.272 | 132.456 | 74.640 |
| Tempo para carregamento (com 20 modelos) | 1’30” | 1’03” | 51” | 30” | 20” |

# 4 CONCLUSÕES

O trabalho apresentou um estudo e a disponibilização de uma ferramenta para aplicação de técnicas de nível de detalhamento sobre modelo MD2 e exportação de arquivos MD2 com modelos 3D simplificados. Isto é realizado através da interpretação e leitura do arquivo MD2 e também o reconhecimento da malha de polígonos do modelo 3D.

Os resultados apresentaram uma boa simplificação e se mostraram favoráveis na utilização na M3GE que se justifica devido a baixa resolução gráfica dos atuais aparelhos celulares e aos visores que possuem um tamanho pequeno. Contudo a fórmula de definição do custo para colapso das arestas pode se mostrar ineficiente em alguns modelos. Entretanto é possível melhorar a técnica para definição do custo e o algoritmo de nível de detalhamento, para que tenha maior liberdade no reposicionamento dos vértices obtendo um melhor resultado. Para a validação dos testes foram geradas diversas simplificações de modelos MD2 em diferentes níveis de aproximação. É possível realizar a simplificação em até 60% do número de vértices de determinados modelos, mantendo suas características principais em determinadas aproximações.

As tecnologias utilizadas no desenvolvimento deste trabalho facilitaram principalmente na execução dos testes e disponibilização da interface da ferramenta. A linguagem C++ escolhida para o desenvolvimento facilitou a disponibilização da ferramenta, em questões de desempenho gráfico e documentação de apoio.

Os resultados obtidos neste trabalho demonstram que é possível efetuar técnicas de nível de detalhamento em modelos MD2, podendo-se obter resultados bastante promissores e colaborando com o trabalho disponibilizado por Estácio (2006). Assim se abre estudos que utilizam técnicas de LOD para o desenvolvimento de jogos ou aplicações gráficas.

## 4.1 EXTENSÕES

Sugestão de extensões para a presente ferramenta, gerando trabalhos futuros são:

a) análise e aprimoramento da fórmula que define o custo de colapso das arestas;
b) verificar a possibilidade de algoritmos que não tenham limitações no posicionamento dos vértices do modelo, podendo gerar modelos mais aproximados

e com menos triângulos que os atuais;

c) analisar o recálculo dos vetores normais e coordenadas de textura;
d) utilização desta técnica de nível de detalhamento em aplicações de tempo real, devido ao seu desempenho;
e) disponibilizar o carregamento do modelo MD2 animado na extensão da M3GE desenvolvida por Estácio (2006);
f) possibilitar na operação de simplificação da malha de polígonos a variação do zoom instantânea no modelo 3D;
g) pré-verificar qualidade da malha de polígonos, procurando por triângulos Delaunay.

# REFERÊNCIAS BIBLIOGRÁFICAS


AI AARDVARK. **LOD**. [S.l.], [2006]. Disponível em: <http://www.ai-aardvark.com/modeling/LOD_101/aia_LOD.html>. Acesso em: 14 set. 2007.

AKENINE-MÖLLER, Tomas; HAINES, Eric. **Real-time rendering**. 2nd ed. Natick: AK Peters, 2002.

AZEVEDO, Eduardo; CONCI, Aura. **Computação gráfica**: teoria e prática. Rio de Janeiro: Campus, 2003.

BELL, Gavin; CAREY, Rikk. **The annotated VRML 97 reference**. [S.l.], 1997. Disponível em: <http://www.cs.vu.nl/~eliens/documents/vrml/reference/BOOK.HTM>. Acesso em: 07 jul. 2008.

BRITO, Allan. **LOD**: level of detail para jogos e animações. [S.l.], 2007. Disponível em: <http://www.allanbrito.com/2007/08/27/lod-level-of-detail-para-jogos-e-animacoes/>. Acesso em: 11 set. 2007.

CORSEUIL, Eduardo T. L. et al. **ENVIRON:** visualization id CAD models in a virtual reality environment. Rio de Janeiro, 2004. Disponível em: <http://66.102.1.104/scholar?hl=pt-BR&lr=&client=firefox-a&q=cache:rA1CSG_sG9AJ:www.tecgraf.puc-rio.br/publications/artigo_2004_environ_cad_models.pdf+author:%22Corseuil%22+intitle:%22ENVIRON%E2%80%93Visualization+of+CAD+Models+In+a+Virtual+...%22+>. Acesso em: 20 set. 2007.

DELOURA, Mark A. **Game programming gems**. Rockland: Charles River Media, 2000.

ESTÁCIO, Claudio J. **Implementação de importação e visualização de modelos de personagem não jogador (PNJ) na Mobile 3D Game Engine (M3GE)**. 2006. 55 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <https://m3ge.dev.java.net/source/browse/*checkout*/m3ge/documentation/portuguese/quake/TCC2006-2-04-VF-ClaudioJEstacio.pdf?rev=1.1> Acesso em: 11 set. 2007.

GARLAND, Michael; HECKBERT, Paul S. **Surface simplification using quadric error metrics**. [S.l.], 2007. Disponível em: <http://graphics.cs.uiuc.edu/~garland/papers/quadrics.pdf>. Acesso em: 10 abr. 2008.

GOMES, Paulo C. R.; PAMPLONA, Vitor F. M3GE: um motor de jogos 3D para dispositivos móveis com suporte a Mobile 3D Graphics API. In: WJOGOS, 6., 2005, São Paulo. **Anais...** São Paulo: USP, 2005, p. 55-65.

HENRY, David. **MD2 file format (Quake 2's models)**. [S.l.], 2004. Disponível em: <http://tfc.duke.free.fr/coding/md2-specs-en.html>. Acesso em: 12 set. 2007.

HENRY, David. **The Quake II's MD2 file format**. [S.l.], 2002. Disponível em: <http://tfc.duke.free.fr/old/models/md2.htm>. Acesso em: 17 abr. 2008.

HILL JR., Francis S. **Computer graphics using OpenGL**. 2nd ed. New Jersey: Prentice Hall, 2001.

HOPPE, Hugues. **Progressive mesh**. Redmond, 1996. Disponível em: <http://research.microsoft.com/~hoppe/pm.pdf>. Acesso em: 23 abr. 2008.

_____, Hugues. **View-dependent refinement of progressive meshes**. Redmond, 1997. Disponível em: <http://research.microsoft.com/~hoppe/vdrpm.pdf>. Acesso em: 23 abr. 2008.

KRUS, Mike et al. **Levels of detail & polygonal simplification**. [S.l.], 2004. Disponível em: <http://www.acm.org/crossroads/xrds3-4/levdet.html>. Acesso em: 15 set. 2007.

MELAX, Stan. **A simple, fast, and effective polygon reduction algorithm**. [S.l.], 1998. Disponível em: <http://www.melax.com/gdmag.pdf>. Acesso em: 04 maio 2008.

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

WATSON, David. **CADTutor**: the best free tutorials on the web. [S.l.], 2006. Disponível em: <http://www.cadtutor.net/dd/bryce/atob/atob.html>. Acesso em: 25 abr. 2008.

WORCESTER, Kevin. **Misfit model 3D**. [S.l.], 2007. Disponível em: <http://www.misfitcode.com/misfitmodel3d/olh_quakemd2.html>. Acesso em: 24 set. 2007.