UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

RECONSTRUTOR DE MODELOS 3D UTILIZANDO TÉCNICA DE NÍVEL DE DETALHAMENTO NO IOS

FELIPE AUGUSTO IMIANOWSKY

BLUMENAU
2013

2013/1-14

FELIPE AUGUSTO IMIANOWSKY

RECONSTRUTOR DE MODELOS 3D UTILIZANDO TÉCNICA DE NÍVEL DE DETALHAMENTO NO IOS

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 Ciência da Computação — Bacharelado.

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

BLUMENAU
2013

2013/1-14

RECONSTRUTOR DE MODELOS 3D UTILIZANDO TÉCNICA DE NÍVEL DE DETALHAMENTO NO IOS

Por

FELIPE AUGUSTO IMIANOWSKY

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

Presidente:	Prof. Dalton Solano dos Reis, Mestre – Orientador, FURB
Membro:	Prof. Antonio Carlos Tavares, Mestre – FURB
Membro:	Prof. Aurélio Faustino Hoppe, Mestre – FURB

Blumenau, 09 de julho de 2013

Dedico este trabalho a todos os curiosos de conhecimento.

AGRADECIMENTOS

Agradeço minha família. Em especial os meus pais, Jorge Imianowsky e Karin Starke Imianowsky, e o meu irmão, Guilherme Nattan Imianowsky, pelo apoio, educação e carinho.

Agradeço meus amigos pelas conversas, discussões e experiências de vida.

Agradeço ao professor Dalton Solano dos Reis por me iniciar na computação gráfica e me auxiliar neste trabalho.

E agradeço a todos os desenvolvedores dos jogos que me inspiraram a estudar e desenvolver este projeto.

O homem certo no lugar errado pode fazer toda a diferença no mundo.

G-Man

RESUMO

Este trabalho apresenta a implementação de uma biblioteca capaz de reconstruir modelos 3D utilizando técnica de nível de detalhamento em tempo real na plataforma iOS. A biblioteca importa modelos 3D e permite que o mesmo modelo seja visualizado com diferentes níveis de detalhamento através de parametrizações. O nível de detalhamento aplicado é do tipo contínuo e segue as etapas de geração, seleção e escolha, realizando os cálculos em tempo real no dispositivo móvel. Para os testes foi utilizado o OpenGL ES 2.0 disponível no iOS. O trabalho finaliza apresentando os resultados obtidos em dispositivos móveis. Esses resultados são satisfatórios em aparelhos com configurações mínimas para modelos com até 4000 vértices, apresentando problemas de memória para objetos com estruturas maiores, além de ocorrer situações em que os objetos ficam desconexos em certos níveis de detalhamento.

Palavras-chave: Nível de detalhamento. Computação gráfica. Ios.

ABSTRACT

This works describes the implementation of a library capable to reconstruct 3D models using level of detail techniques in real time on the iOS platform. The library imports 3D models and allows the same model to be viewed at different levels of detail through parameterization. The level of detail applied is continuous LOD and follows the steps of generation, selection and choice, performing calculations in real time on mobile device. For testing we used the OpenGL ES 2.0 available on iOS. The paper concludes by presenting the results on mobile devices. These results are satisfactory for devices with minimal configurations for models with up to 4000 vertices, with memory problems for objects with bigger structures. In addition, in some situations, objects become disconnected at certain levels of detail.

Key-words: Level of detail. Graphics computing. Ios.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

2D – Duas Dimensões

3D – Três Dimensões

API – *Application Programming Interface*

ASCII - *American Standard Code for Information Interchange*

CPU – *Central Processing Unit*

FPS – *Frames Per Second*

GPU – *Graphics Processing Unit*

LOD – *Level Of Detail*

OpenGL – *Open Graphics Library*

OpenGL ES – *Open Graphics Library for Embedded Systems*

PDA – *Personal Digital Assistant*

RF – Requisito Funcional

RNF – Requisito Não Funcional

UML – *Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Os dispositivos móveis, como *smartphones* e *tablets*, estão cada vez mais presentes no dia-a-dia das pessoas (HERNANDEZ et al., 2012, p. 12). Estes aparelhos estão trazendo novas formas para solucionar problemas cotidianos, bem como novos meios de entretenimento e estilos de vida para seus usuários. Dispositivos móveis que podem ser citados neste contexto são o iPhone e o iPad, ambos produtos da empresa Apple e que ganharam notoriedade após seus lançamentos em 2007 e 2010 respectivamente. O iPhone e o iPad, dentre outras características, destacam-se pelo potencial gráfico que apresentam e, por este motivo, são bastante empregados para o desenvolvimento de ferramentas gráficas e jogos que utilizam desde desenhos 2D até ambientes e modelos 3D.

Segundo Piske (2008, p. 13), 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.

A complexidade dos modelos 3D – mensurados geralmente pelo número de polígonos – parece crescer mais rápido que a habilidade dos *hardwares* de processá-los. Não importa o quão potente é a plataforma, o número de polígonos desejável parece sempre exceder o número de polígonos que se pode suportar (LUEBKE et al., 2003, p. 03).

A simplificação de polígonos em *Level Of Detail*[1] (LOD) tem por objetivo remover as primitivas a partir de uma malha de polígonos a fim de produzir modelos mais simples que mantenham as características visuais mais importantes do objeto original (KRUS et al., 1997).

Diante do problema exposto, desenvolveu-se uma biblioteca capaz de importar, visualizar e manipular modelos 3D no formato Wavefront OBJ para a plataforma iOS, permitindo a aplicação de técnica conhecida na computação gráfica como LOD sugerida por Clark em 1976 (KRUS et al., 1997).

[1] Acrônimo utilizado para representar nível de detalhamento

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma biblioteca para a plataforma iOS capaz de reconstruir modelos 3D utilizando técnicas de LOD em tempo real.

Os objetivos específicos do trabalho são:

a) construir uma biblioteca capaz de visualizar modelos 3D no formato Wavefront OBJ para a plataforma iOS;
b) disponibilizar ao usuário funcionalidades para aplicar LOD em modelos 3D em tempo real;
c) dispor de visualização das informações de performance durante a execução.

1.2 ESTRUTURA DO TRABALHO

O trabalho está desenvolvido em quatro capítulos. O segundo capítulo deste trabalho refere-se à fundamentação teórica necessária para o seu entendimento.

O terceiro capítulo descreve o desenvolvimento da biblioteca para a plataforma iOS, os casos de uso envolvidos, os diagramas de classe, diagramas de sequência e as especificações que definem a biblioteca. Ainda no terceiro capítulo, são apresentadas as partes principais da implementação e os resultados obtidos no desenvolvimento do trabalho.

Por fim, o quarto capítulo apresenta as conclusões do presente trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 trata do modelo Wavefront OBJ. Na seção 2.2 é apresentada a biblioteca gráfica OpenGL ES e suas características. A seção 2.3 aborda o conceito de nível de detalhamento. Na última seção são descritos alguns trabalhos correlatos.

2.1 MODELO WAVEFRONT OBJ

O modelo Wavefront OBJ é um formato de arquivo simples, bem documentado e aberto, o que o torna perfeito para exportar informações básicas de modelos 3D. Ferramentas gráficas como o ZBrush, Autodesk Maya, Autodesk 3ds Max e o Blender utilizam-se deste formato para exportar modelos 3D entre si (MARUCCHI-FOINO, 2012, p. 29-30).

O formato de arquivo, utilizando ASCII, foi inicialmente desenvolvido para ser utilizado no software Advanced Visualizer da Wavefront Technologies (MAYA, 1990), mas se tornou amplamente utilizado em outras ferramentas gráficas por sua estrutura simples e de fácil importação e exportação.

Este modelo consiste em 2 arquivos distintos: `.OBJ` e `.MTL`. O arquivo `.OBJ` contém todas as posições de vértices, as normais dos vértices e coordenadas UV de um ou vários objetos 3D. O arquivo `.MTL` contém os dados de todos os materiais que são usados por um ou múltiplos objetos dentro do arquivo `.OBJ`. Possui também informações como cor ambiente e difusa, nome do arquivo de textura e outros parâmetros específicos para os materiais (MARUCCHI-FOINO, 2012, p. 30).

Para identificar diferentes tipos de informações de um modelo 3D os arquivos utilizam palavras-chaves. Desta forma os interpretadores conseguem extrair as informações sabendo ao que estes se referem. Por exemplo, a representação de um vértice inicia com a palavra-chave `v` seguida de 3 valores que representam as coordenadas do vértice.

2.2 BIBLIOTECA GRÁFICA OPENGL EMBEDDED SYSTEMS (OPENGL ES)

OpenGL ES é um conjunto de *Application Programming Interfaces* (APIs) criado pela Khronos Group, destinado a gráficos 3D avançados para dispositivos móveis e embarcados como celulares, *Personal Digital Assistants* (PDAs), consoles, eletrodomésticos e veículos (MUNSHI; GINSBURG; SHREINER, 2009, p. 01).

OpenGL ES é uma versão simplificada do OpenGL que elimina funcionalidades redundantes para oferecer uma biblioteca que é fácil de aprender e simples de implementar para dispositivos móveis (APPLE INC, 2010).

Há 3 especificações do OpenGL ES que foram lançados pela Khronos: a OpenGL ES 1.0, OpenGL ES 1.1 e a OpenGL ES 2.0. As especificações OpenGL ES 1.0 e OpenGL ES 1.1 possuem as fases de execução fixas, enquanto na especificação OpenGL 2.0 as fases são programáveis (MUNSHI; GINSBURG; SHREINER, 2009, p. 02-03).

Por fases de execução fixas e fases de execução programáveis entende-se que, na primeira, para cada iteração da API, as etapas são executadas independente da utilização de atributos como transformação, luz e textura, enquanto que na segunda, estas fases foram substituídas por duas novas etapas que aceitam implementações do usuário da API através da OpenGL ES *Shading Language*.

2.3 NÍVEL DE DETALHAMENTO

O conceito fundamental de nível de detalhamento ou LOD é explicado por Luebke et al. (2003, p. 05-06), onde, no processamento de um modelo 3D, utiliza-se uma representação menos detalhada para as porções pequenas, distantes ou sem importância da cena. Esta representação menos detalhada normalmente consiste de uma seleção de várias versões do objeto na cena, cada versão menos detalhada e mais rápida para processar do que a anterior.

Algoritmos LOD consistem em três partes principais: geração, seleção e escolha. A geração é a parte onde diferentes representações do modelo são geradas com diferentes níveis de detalhes. O mecanismo de seleção escolhe o nível de detalhamento para o modelo baseado em algum critério. Finalmente, a etapa de escolha consiste no processo de troca de um nível de detalhamento para o outro (AKENINE-MÖLLER; HAINES; HOFFMAN, 2008, p. 681).

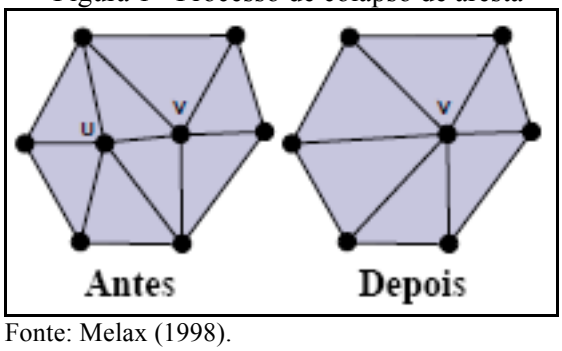
Existem basicamente 3 tipos de LOD: discreto, contínuo e o dependente de visão. No discreto são geradas várias versões de nível de detalhamento de um objeto em um pré-processamento e, durante a execução do programa, apenas escolhe-se qual destas versões será utilizada em cena. Já no LOD contínuo, o sistema de simplificação cria uma estrutura de dados codificada e o nível de detalhamento é escolhido em tempo real. Por fim, o LOD dependente de visão estende as propriedades do tipo contínuo acrescentando o campo de visão como um fator para dinamicamente aplicar o nível de detalhamento nos modelos, portanto um único objeto pode ter vários níveis de detalhamento ao mesmo tempo, dependendo do campo de visão atual (LUEBKE et al., 2003, p. 09-10).

2.3.1 Redução poligonal

Melax (1998) apresenta um algoritmo de redução poligonal capaz de diminuir o número de polígonos de um modelo com bastante detalhamento para uma versão similar ao original, porém com número reduzido de polígonos.

O algoritmo trabalha com o conceito de colapso de arestas proposto por Hoppe (1996). Nesta operação, dois vértices `u` e `v` (a aresta `uv`) são selecionados e um deles (`u`) é movido ou sofre colapso para o outro (neste caso, `v`). Esta etapa é repetida até que o número de polígonos desejado seja satisfeito (MELAX, 1998, p. 45). Geralmente cada iteração deste processo remove um vértice, duas faces e três arestas. A Figura 1 ilustra o processo de colapso de aresta.

Figura 1 - Processo de colapso de aresta



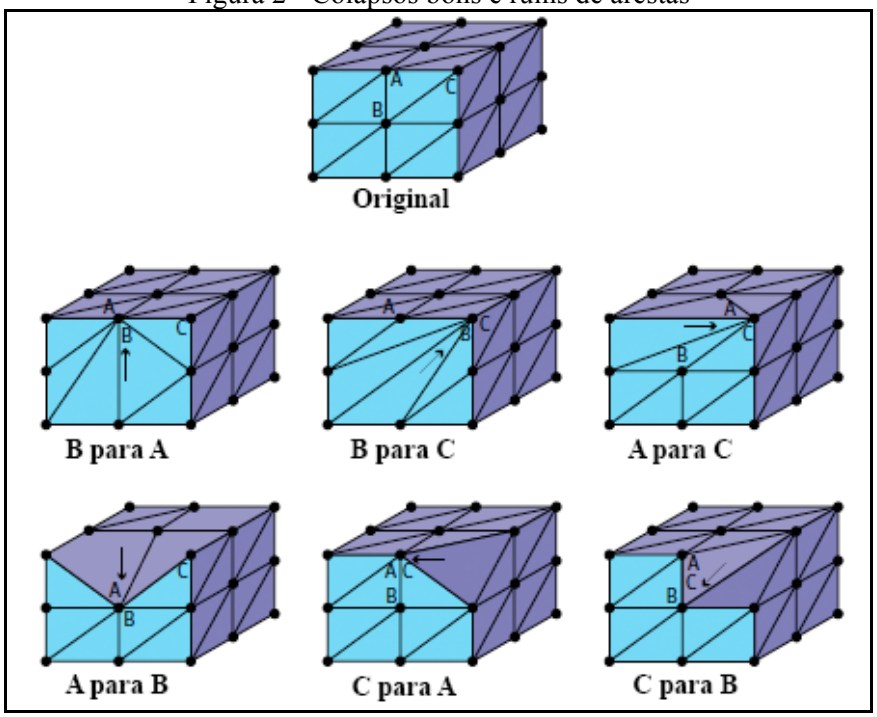
Fonte: Melax (1998).

Segundo Melax (1998) para se produzir bons modelos com polígonos reduzidos similares ao modelo original, é necessário selecionar a aresta que, quando sofrer colapso, causará a menor mudança visual ao modelo.

Na Figura 2 são apresentados exemplos bons e ruins de colapso de arestas. É possível

verificar que o ponto B, por se encontrar no centro da mesma face do ponto A e do ponto C, poderia realizar colapso para um desses dois vértices sem causar um grande impacto visual. Da mesma forma realizar colapso de A para C não afetaria a forma do modelo. Em contrapartida, executar colapso de A para B, C para A ou C para B compromete a similaridade do modelo reduzido com o modelo original.

Figura 2 - Colapsos bons e ruins de arestas



Fonte: Melax (1998).

Para a heurística de seleção da aresta a sofrer colapso com menor impacto visual, Melax (1998) definiu que o custo de colapso para uma aresta corresponde ao tamanho desta multiplicada pelo termo de curvatura. O termo de curvatura, por sua vez, é definido comparando o produto escalar das faces normais dos triângulos, determinando a coplanaridade entre os triângulos que compõe o modelo. A equação matemática do custo de colapso de uma aresta é apresentada no Quadro 1.

Quadro 1 - Equação de custo de colapso de uma aresta

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

Fonte: Melax (1998).

Na equação do Quadro 1, $\|u-v\|$ corresponde ao tamanho da aresta. Este valor é multiplicado por `f.normal•n.normal`, onde `f` representa o conjunto de triângulos que contém o vértice `u` ($f \in Tu$) e n corresponde ao conjunto de triângulos que contém `u` e `v` ($n \in Tuv$). Desta forma a simplificação do modelo é feita priorizando as áreas planares com grande

concentração de vértices.

2.4 TRABALHOS CORRELATOS

A seguir são apresentadas ferramentas que se utilizam das técnicas de nível de detalhamento. Do trabalho de Piske (2008), da AI Aardvark e do Unity 3D serão destacadas as principais características.

2.4.1 Reconstrutor de modelos 3D utilizando técnicas de nível de detalhamento

Piske (2008) desenvolveu uma ferramenta capaz de aplicar técnicas de LOD em modelos 3D no formato Quake 2's Models ou `MD2` para o motor de jogos para celulares Mobile 3D Game Engine (M3GE). Para isso utilizou técnicas de simplificação por meio de colapsos (ver seção 2.3.1) das arestas de custo mínimo, com base no tamanho da aresta e na coplanaridade entre triângulos da malha de polígonos. Através da ferramenta o usuário tem a opção de selecionar o nível de detalhamento a ser aplicado sobre um modelo 3D e, em seguida, salvar este modelo simplificado para ser utilizado na M3GE ou em outra ferramenta que suporte importação de arquivo `MD2`.

A Figura 3 apresenta um exemplo de um modelo 3D com diferentes níveis de detalhamento aplicados utilizando a ferramenta.

Figura 3 - Modelo 3D com 100%, 80%, 60%, 40% e 20% de detalhamento, respectivamente

Fonte: Piske (2008).

Segundo Piske (2008, p. 61), 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.

2.4.2 Simulador de Aviões

A AI Aardvark (2006) utiliza técnicas de LOD para simulação de vôos de avião. A técnica é utilizada nos aviões e também nos cenários de tal forma que os objetos próximos do observador utilizem modelos com alto nível de detalhamento e os que estão mais afastados utilizam modelos com baixo nível de detalhamento, como pode ser visto na Figura 4.

Figura 4 - Utilização de LOD pela AI Aardvark no simulador de vôos

Fonte: AI Aardvark (2006).

O processamento do nível de detalhamento é realizado antes da execução e não em tempo real, de tal forma que a ferramenta alterna entre as diferentes versões da malha de polígonos de um objeto dependendo da sua distância em relação à câmera.

O principal objetivo da utilização de LOD neste caso é o ganho na taxa de *frames* por segundo, permitindo suavização na simulação do vôo, tornando-a mais parecida com a realidade.

Isso é importante, por exemplo, para a simulação de aeroportos muito movimentados onde haverá muitos modelos em cena. Se houver entre 20 e 30 aviões em uma mesma cena e nenhum deles estiver utilizando modelos com nível de detalhamento, a taxa de *frames* por segundo será muito prejudicada (AI AARDVARK, 2006).

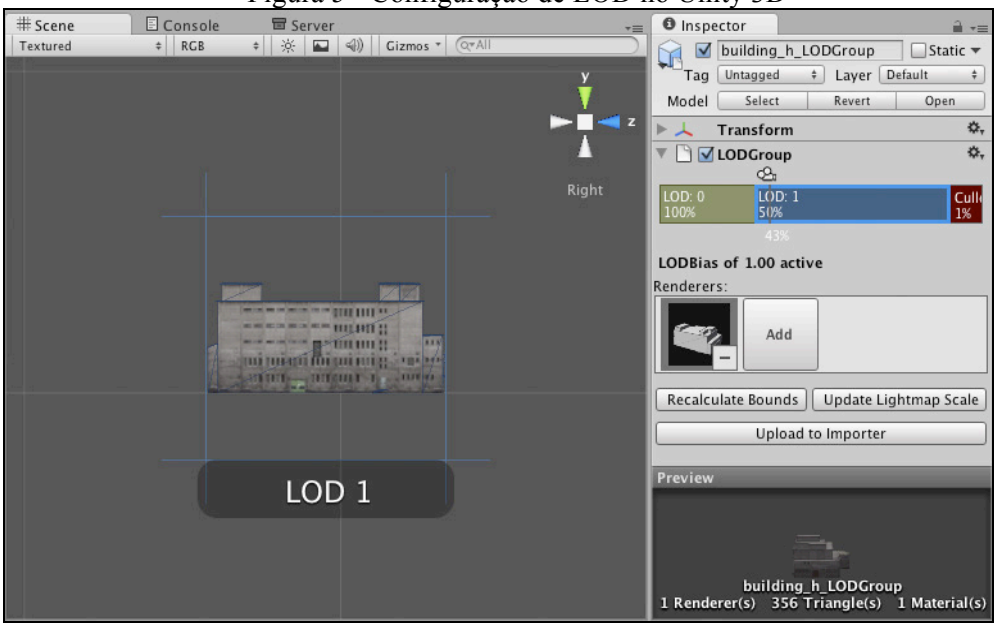
2.4.3 Unity 3D

Unity3D (UNITY, 2012a) é uma ferramenta desenvolvida pela Unity Technologies. É

composta por um motor de jogos 3D e um editor de cenas. Esta ferramenta possui suporte para diversas plataformas como Windows, Mac OS X, iPhone, iPad, Android, navegadores, entre outros.

Conforme as cenas vão ficando maiores, a performance é prejudicada. Uma das maneiras fornecidas pelo Unity3D para gerenciar este aspecto é disponibilizar aos usuários uma funcionalidade com o objetivo de atribuir diferentes malhas de polígonos para o mesmo modelo 3D, dependendo do quão afastada está a câmera em relação ao objeto (UNITY, 2012b), como pode ser visto na Figura 5.

Figura 5 - Configuração de LOD no Unity 3D



Fonte: Unity (2012b).

Nota-se que o Unity3D utiliza a abordagem de LOD discreto, pois o processamento das diferentes versões de malha de polígonos é gerado antes da sua execução no motor de jogos, através de outras ferramentas de modelagem onde a manipulação da malha de polígonos seja possível, como no Autodesk 3ds Max e no Autodesk Maya. Durante a execução o Unity3D escolhe a malha de polígonos a ser apresentada com base na configuração de distância da câmera realizada pelo usuário.

2.4.4 Características dos trabalhos correlatos

Esta seção expõe as principais características dos trabalhos correlatos apresentados, como pode ser visto no Quadro 2.

Quadro 2 - Características dos trabalhos correlatos

Trabalhos Correlatos / Características	Reconstrutor de modelos 3D utilizando técnicas de nível de detalhamento	Simulador de Aviões	Unity3D
Importação de modelos 3D	Sim	Sim	Sim
LOD Discreto	Não	Sim	Sim
LOD Contínuo	Sim	?	Não
Permite configurar quantidade de vértices no LOD	Sim	Não	Não
Suporte a dispositivos móveis	Sim	Não	Sim
LOD calculado em tempo real no dispositivo móvel	Não	Não	Não

Fontes: AI Aardvark (2006), Piske (2008) e Unity (2012b).

É possível verificar que o trabalho de Piske (2008) destaca-se principalmente por possibilitar a configuração da quantidade de vértices para a aplicação de LOD. Além disso, o mesmo trabalho, é o único entre as ferramentas citadas que possui registros de utilizar LOD contínuo. O suporte a dispositivos móveis é uma característica tanto do trabalho de Piske (2008) quanto do Unity (2012b).

Todos os trabalhos possuem algum tipo de importação de modelos 3D. Em contrapartida, nenhum deles possui uma forma de executar o LOD em tempo real no dispositivo móvel, sendo que este processo é sempre feito através de um pré-processamento para todos os trabalhos.

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas de desenvolvimento da biblioteca de reconstrução 3D utilizando técnica de LOD, assim como a importação dos arquivos `.OBJ` e `.MTL`. São abordados os principais requisitos, a especificação, a implementação e finalizando com os resultados e discussão.

O presente trabalho agrega conhecimento dos autores referenciados, entretanto, o trabalho de Melax (1998) tem participação fundamental por disponibilizar o código fonte do algoritmo de redução poligonal escrito na linguagem de programação C++.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A biblioteca deverá:

a) permitir a importação de modelo 3D no formato Wavefront OBJ (Requisito Funcional - RF);
b) manipular a visualização do objeto importado em cena através de *zoom* e rotação (RF);
c) permitir aplicar técnicas de LOD no objeto 3D importado em tempo real (RF);
d) permitir ao usuário informar o valor do nível de detalhamento desejado ao aplicar técnicas de LOD sobre o modelo 3D (RF);
e) disponibilizar informações sobre a cena atual como número de vértices, número de pontos e *frames* por segundo (RF);
f) disponibilizar diferentes tipos de visualização do objeto em cena: nuvem de pontos, *wireframe*, sólido e texturizado (RF);
g) ser implementada na linguagem Objective-C (Requisito Não Funcional – RNF);
h) utilizar biblioteca gráfica OpenGL ES (RNF);
i) utilizar o ambiente de desenvolvimento Xcode 4 (RNF);
j) utilizar o simulador do iPhone e do iPad no Xcode 4, bem como o aparelho físico (RNF).

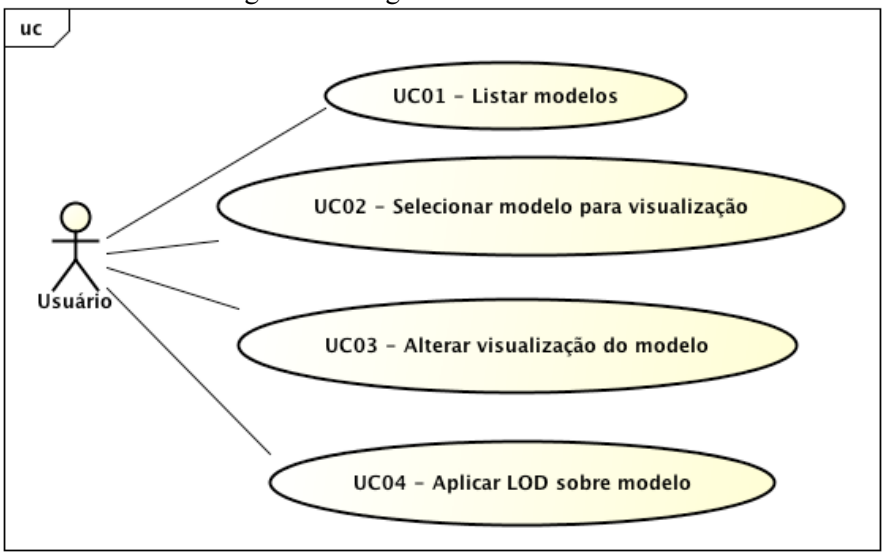
3.2 ESPECIFICAÇÃO

A especificação deste trabalho foi desenvolvida utilizando a ferramenta Astah Community, utilizando os conceitos do paradigma de orientação a objetos e por meio de diagramas da *Unified Modeling Language* (UML). Nas próximas seções são apresentados os diagramas de caso de uso, diagramas de classe e diagramas de sequência.

3.2.1 Casos de uso

Nesta seção são descritos os casos de uso de todas as funcionalidades da biblioteca. Foi identificado um ator, denominado `Usuário`, que utilizará a biblioteca para visualizar modelos `OBJ` e aplicar os conceitos de LOD sobre os mesmos. A Figura 6 ilustra estes casos de uso.

Figura 6 - Diagrama de casos de uso



A seção 3.2.1.1 descreve o caso de uso de listagem dos modelos. A seção 3.2.1.2 apresenta o caso de uso de seleção de um modelo para visualização. Já a seção 3.2.1.3 demonstra as características do caso de uso onde o `Usuário` altera a visualização do modelo. Por fim, a seção 3.2.1.4 descreve o caso de uso onde o LOD é aplicado sobre o modelo.

3.2.1.1 Listar modelos

Este caso de uso descreve como o `Usuário` irá buscar os modelos disponíveis para visualização na biblioteca. Detalhes deste caso de uso são apresentados no Quadro 3.

Quadro 3 - Caso de uso UC01

UC01 – Listar modelos	
Descrição	Para aplicar os conceitos de LOD da biblioteca é necessário que seja escolhido um modelo `OBJ`. Através desta listagem será possível selecionar este modelo.
Cenário Principal	1. O aplicativo inicia e automaticamente apresenta a lista de modelos disponíveis.
Fluxo Alternativo	2. No passo 1 do Cenário Principal, ao invés do aplicativo inicializar automaticamente a listagem de modelos, existe também a possibilidade de o usuário solicitar esta listagem manualmente através da interface.

3.2.1.2 Selecionar modelo para visualização

Este caso de uso descreve como o `Usuário` selecionará o modelo `OBJ` que deseja visualizar para aplicar a técnica de LOD disponível. O Quadro 4 detalha o caso de uso onde a biblioteca interpreta um modelo para visualização pelo `Usuário`.

Quadro 4 - Caso de uso UC02

UC02 – Selecionar modelo para visualização	
Descrição	Ao selecionar o modelo, a biblioteca pode interpretá-lo e prepara-lo para que possa ser visualizado e manipulado.
Pré-condição	É necessário que tenha sido solicitada a listagem dos modelos.
Cenário Principal	1. O usuário seleciona o modelo desejado. 2. A biblioteca interpreta o modelo selecionado e o apresenta ao usuário.

3.2.1.3 Alterar visualização do modelo

Este caso de uso descreve a alteração no modo de visualização de um modelo realizado pelo `Usuário`. A biblioteca possui modos de visualização pré-definidos que poderão ser

utilizados em qualquer modelo. O Quadro 5 mostra mais detalhes deste caso de uso.

Quadro 5 - Caso de uso UC03

UC03 – Alterar visualização do modelo	
Descrição	Para auxiliar na visualização do modelo a biblioteca disponibiliza diferentes formas de apresentação deste.
Pré-condição	É necessário que um modelo tenha sido selecionado e interpretado corretamente.
Cenário Principal	1. O usuário seleciona a nova forma de visualização, podendo ser: - Modelo com textura; - Modelo sem textura; - Apenas arestas (*wireframe*); - Apenas vértices.

3.2.1.4 Aplicar LOD sobre modelo

Este caso de uso descreve como deve ser realizada a ação para aplicação das técnica de LOD disponível na biblioteca sobre um modelo previamente selecionado. O Quadro 6 detalha este caso de uso.

Quadro 6 - Caso de uso UC04

UC03 – Aplicar LOD sobre modelo	
Descrição	Com um modelo selecionado e interpretado, a biblioteca pode aplicar os conceitos de LOD sobre este modelo e simplificar o seu detalhamento utilizando informações como percentual de vértices desejado, além de realizar *cache*[2] para melhorar o desempenho do algoritmo.
Pré-condição	É necessário que um modelo tenha sido selecionado.
Cenário Principal	1. O usuário ativa a opção de LOD através da interface. 2. O usuário ajusta o percentual de vértices que deseja que o modelo apresente para simplificação.
Fluxo Alternativo	3. A qualquer momento, após o passo 1 do Cenário Principal, o usuário pode ativar ou desativar o *cache* de LOD. Uma vez ativado o algoritmo gera um mapeamento de todos os colapsos que serão realizados, otimizando o processamento do algoritmo.

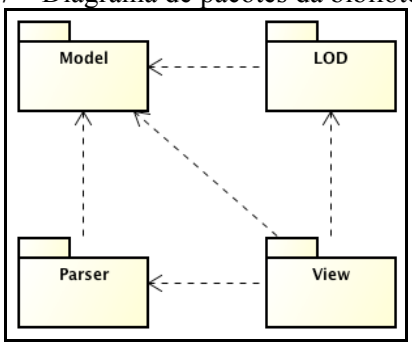
3.2.2 Diagramas de classe

Nesta seção são apresentadas as classes envolvidas na biblioteca de LOD e

[2] *Cache*: termo em inglês que na área de ciências da computação refere-se à estrutura de dados que guarda informações para servir de maneira mais rápida às requisições futuras de uma operação.

interpretação de modelos `OBJ` e `MTL`, bem como seus relacionamentos e estruturas. Na Figura 7 são demonstradas as dependências entre os pacotes definidos para a biblioteca.

Figura 7 – Diagrama de pacotes da biblioteca de LOD

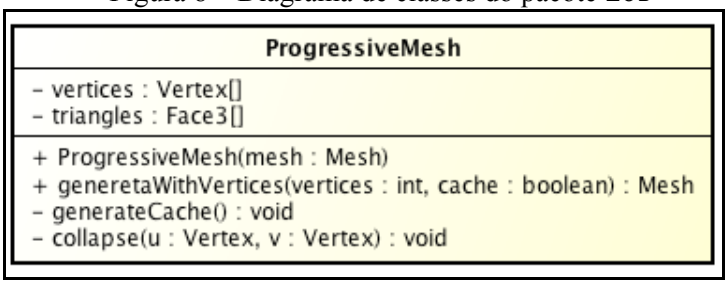


A seção 3.2.2.1 apresenta as classes que fazem parte do pacote `LOD`. A seção 3.2.2.2 descreve o pacote `Model` e as classes que o compõe. Já a seção 3.2.2.3 demonstra as classes do pacote `Parser`. Por fim, a seção 3.2.2.4 apresenta o pacote `View` e as suas classes.

3.2.2.1 Pacote `LOD`

Este pacote é composto pela principal classe da biblioteca. Denominado `LOD`, o pacote é responsável pela manipulação do modelo fornecido pelo `Usuário` para a geração de diferentes representações gráficas utilizando os conceitos de LOD. A Figura 8 apresenta as principais funcionalidades deste pacote.

Figura 8 – Diagrama de classes do pacote `LOD`



A classe `ProgressiveMesh` é responsável por gerenciar a construção de um modelo em diferentes níveis de detalhamento. Para isso, em seu construtor, é fornecido um dado do tipo `Mesh`, que representa a malha de polígonos do modelo original.

Através do método `generetaWithVertices` é possível gerar uma nova malha de polígonos com a quantidade de vértices desejados expressa pelo parâmetro `vertices`, havendo ainda a possibilidade de realizar *cache* da redução poligonal. Para realizar este *cache* a classe utiliza-se do método privado `generateCache`.

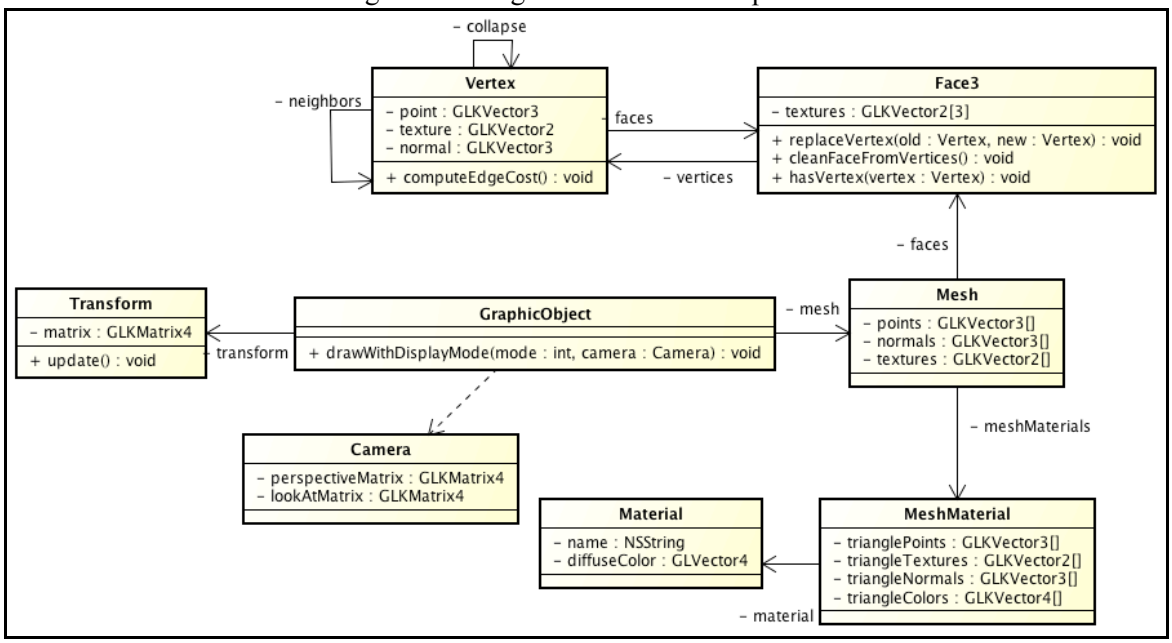
O método `collapse` recebe como parâmetro dois vértices do tipo `Vertex`, `u` e `v`, e é responsável por realizar a ação de colapso de aresta, movendo `u` para `v`. São removidos os triângulos que contém a aresta `uv` e, em seguida, são atualizados os demais triângulos que possuem `u` para utilizarem `v`. Ao fim, o vértice `u` é removido da lista de vértices.

Os atributos privados `vertices` e `triangles` são utilizados para que seja possível a redução poligonal sem que a malha de polígonos original seja comprometida. Desta forma, antes de inicializar a redução poligonal, `vertices` será alimentado com a lista de vértices completa do modelo original e, da mesma forma, `triangles` receberá todas as faces da malha de polígonos original.

3.2.2.2 Pacote `Model`

O pacote `Model`, composto pelas classes ilustradas na Figura 9, representam a estrutura do modelo esperado para ser utilizado pela biblioteca. São classes que encapsulam as propriedades de um objeto gráfico e possuem as informações necessárias para a aplicação do LOD.

Figura 9 – Diagrama de classes do pacote `Model`



A classe `Vertex` se responsabiliza por representar a informação de um vértice. O vértice é composto pelas coordenadas `x`, `y` e `z`, a normal e coordenadas de textura. Algumas informações complementares, como vértices vizinhos e faces as quais o vértice pertence, são dados utilizados para otimização dos algoritmos de LOD. O mesmo se aplica ao método

`computeEdgeCost`, que é utilizado para calcular os custos de colapso do vértice em relação ao seus vértices vizinhos, durante o processo de nível de detalhamento.

Já a classe `Face3` possui as propriedades para representação de um face de um modelo. Para tanto, a face é composta por 3 vértices, a normal e as coordenadas de textura. Métodos como `replaceVertex` auxiliam os algoritmos de LOD na substituição de um vértice por outro.

O `Material` é uma classe com representação direta a um arquivo `MTL`, portanto sua principal função é encapsular os valores referentes as características visuais de uma parte do modelo como textura e cor difusa.

O `Mesh` busca representar a malha de polígonos do modelo e, para isso, possui propriedades como a lista de pontos com as coordenadas `x`, `y` e `z` não repetidas, bem como as normais, as coordenadas de textura e as faces que, neste caso, representam os triângulos do modelo.

Devido a necessidade de se utilizar apenas uma configuração de material por vez para desenhar um modelo, os objetos que possuam mais de uma material, precisam que estas partes sejam desenhadas uma de cada vez. Para isso, a classe `MeshMaterial` auxilia a separar estas partes cada qual com seu material e complementa com outras informações em forma de triângulos, ou seja, sempre com listas com tamanhos múltiplos de 3 para as representações dos pontos, texturas, normais e cores. Esta é a principal classe utilizada para transferir as informações do modelo à biblioteca OpenGL ES.

A classe `Camera` é utilizada para representar a câmera da biblioteca, possuindo matrizes como `perspectiveMatrix`, para representar a matriz de perspectiva e a `lookAtMatrix` para representar a matriz de visualização, ambas utilizadas em cálculos matemáticos para a apresentação de modelos na biblioteca.

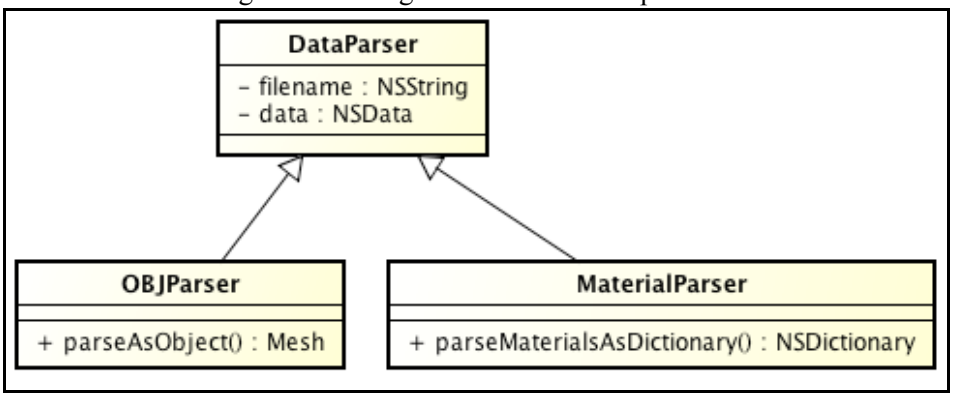
Para a transformação de um modelo, utiliza-se a classe `Transform`. Esta classe foi projetada pra executar as funções de translação, rotação e escala de um modelo durante a execução da biblioteca e, para tanto, utiliza-se do método `update` responsável por calcular e retornar a matriz contendo todas as transformações de um objeto gráfico.

A classe `GraphicObject`, então, engloba toda a representação de um modelo contendo a malha de polígonos no atributo `mesh` e a transformação do modelo no atributo `transform`. Através de seu método `drawWithDisplayMode` é capaz de apresentar o modelo nos diversos modos de visualização suportados pela biblioteca e projetar o modelo transformado utilizando a visualização fornecida pelo parâmetro contendo as informações da câmera.

3.2.2.3 Pacote `Parser`

O pacote denominado `Parser` possui as classes necessárias para interpretar um modelo `OBJ` e `MTL`. A Figura 10 apresenta o modelo de classes deste pacote.

Figura 10 – Diagrama de classes do pacote `Parser`



A classe `DataParser` é uma abstração utilizada pelas demais classes deste pacote para inicializar os dados necessários à leitura de um arquivo.

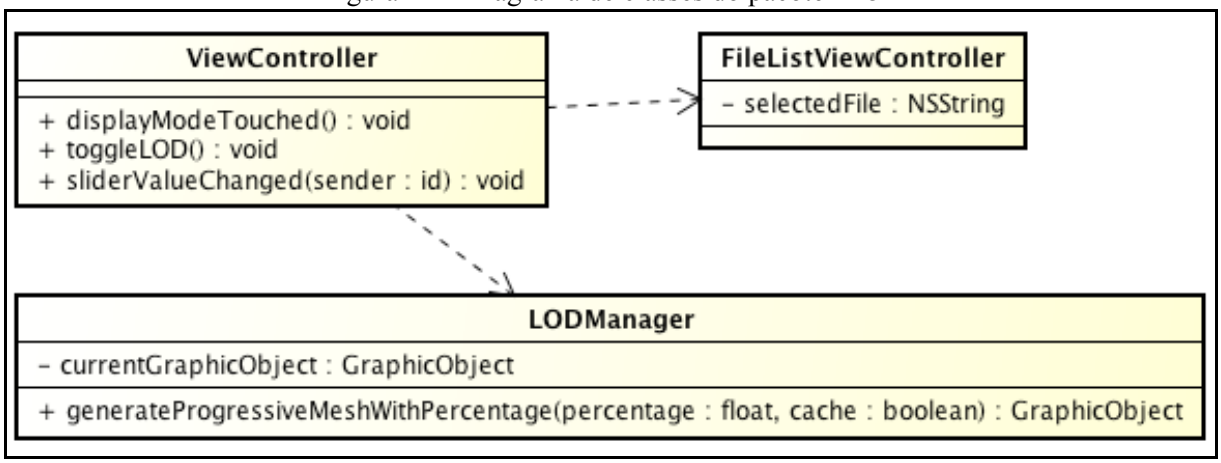
A classe `OBJParser` é uma extensão da classe `DataParser` e é responsável por ler e interpretar o arquivo `OBJ` e retornar o objeto na estrutura esperada pela biblioteca, descrito no pacote `Model`, através do método `parseAsObject`.

Da mesma forma, a classe `MaterialParser`, estende as funcionalidades da classe `DataParser` e auxilia na leitura e interpretração do arquivo `MTL` durante a construção do objeto esperado, através do método `parseMaterialsAsDictionary`.

3.2.2.4 Pacote `View`

O pacote `View` é responsável por gerenciar a interface com o usuário. A Figura 11 ilustra o seu modelo de classes.

Figura 11 – Diagrama de classes do pacote `View`



A classe `LODManager` é a principal classe deste pacote, pois é através dela que ocorre a comunicação entre a interface e a biblioteca de LOD. O método `generateProgressiveMeshWithPercentage` recebe os parâmetros da interface e, com o auxílio do pacote `LOD`, executa o processo de redução poligonal.

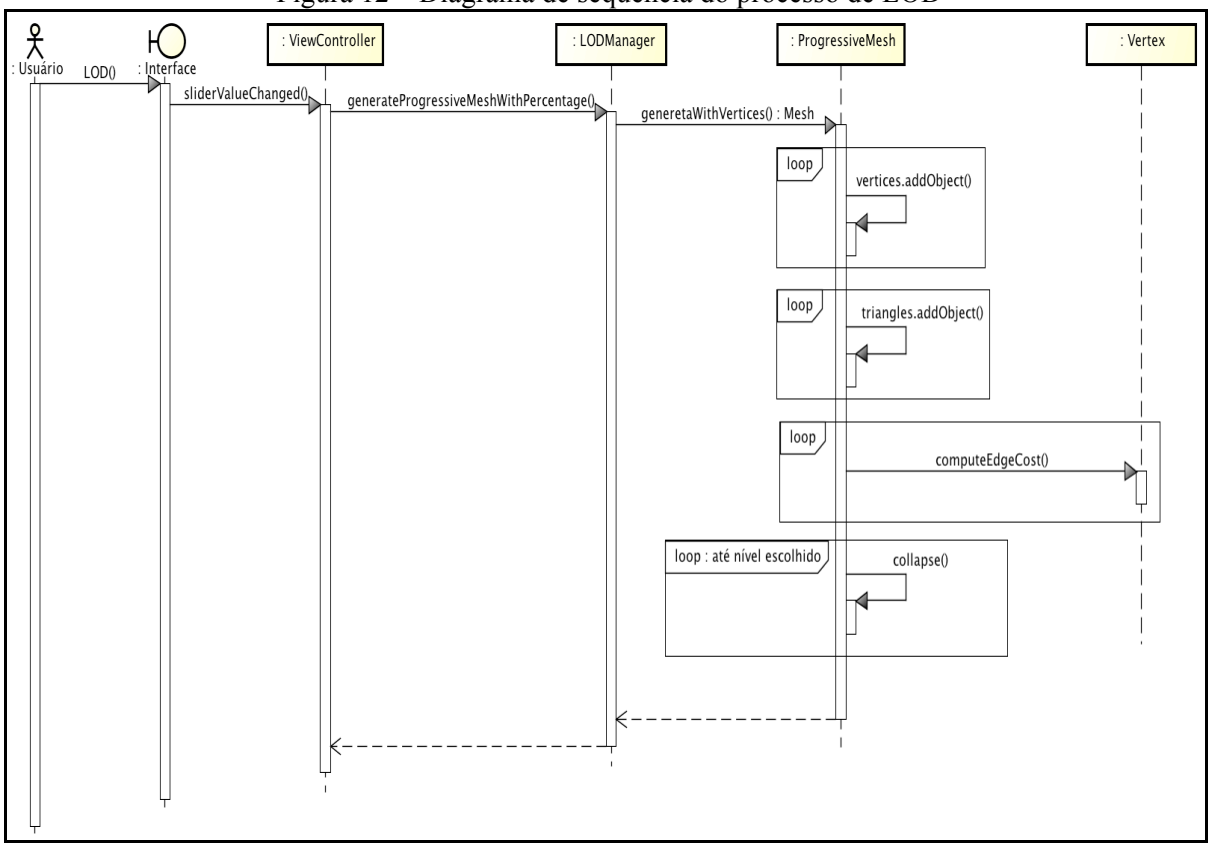
A classe `ViewController` é responsável por gerenciar a tela de visualização e manipulação dos modelos e realizar a execução dos algoritmos de LOD nestes através da classe `LODManager`. Possui o método `sliderValueChanged` responsável por obter o número de vértices informado pelo usuário na interface e repassá-lo ao pacote `LOD`.

Já a classe `FileListViewController` tem por responsabilidade administrar a tela de listagem de arquivos disponíveis no aplicativo para serem utilizados pela biblioteca.

3.2.3 Diagramas de sequência

O diagrama de sequência ilustrado pela Figura 12 apresenta a aplicação de LOD sobre um modelo utilizando a biblioteca.

Figura 12 – Diagrama de sequência do processo de LOD



No diagrama apresentado, o `Usuário` solicita um novo nível de detalhamento para o modelo previamente carregado. Desta forma, a biblioteca executa a geração de um nova malha de polígonos, representado por `Mesh`. Adicionam-se todos os vértices e triângulos da malha de polígonos original em novas listas e, com estas, são feitos os colapsos dos vértices utilizando o algoritmo de Melax (1998) até o nível de detalhamento solicitado. Ao fim, é retornada a nova malha de polígonos reduzida para ser utilizada na apresentação do modelo com o novo nível de detalhamento.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

O desenvolvimento da biblioteca de reconstrução 3D utilizando técnica de LOD foi feito com o uso da linguagem de programação Objective-C. O iOS 6.0 é a plataforma suportada pela biblioteca. Para o ambiente de desenvolvimento foi utilizado o Xcode 4.6.1 em conjunto com o *framework* `GLKit` para as representações matemáticas de vetores e matrizes e os cálculos das mesmas, além de métodos auxiliares como, por exemplo, o carregamento de texturas. A biblioteca OpenGL ES 2.0 foi empregada para as construções gráficas da biblioteca de reconstrução 3D.

A biblioteca também faz uso de arquivos `OBJ` que estão pré-carregados e podem ser utilizados para a visualização dos conceitos de LOD empregados.

Os testes foram realizados através do simulador fornecido pelo ambiente de desenvolvimento Xcode além do uso dos dispositivos iPhone 3GS e iPad 4.

3.3.2 Implementação da biblioteca

Nesta seção são apresentadas as principais implementações da biblioteca como leitura dos arquivos `OBJ` e `MTL` e aplicação dos conceitos de LOD sobre os modelos importados utilizando a biblioteca.

3.3.2.1 Leitura de arquivos `OBJ` e `MTL`

Como apontado no capítulo de fundamentação teórica (ver seção 2.1) os arquivos `OBJ` são utilizados para representar informações básicas de um modelo 3D através de palavras-chaves. Para extrair as informações destes arquivos utilizou-se a classe `NSScanner` presente na linguagem Objective-C, que auxilia na extração e conversão de informações contidas numa variável ou constante do tipo `NSString`.

O Quadro 7 apresenta o método que realiza a interpretação de uma linha do arquivo `OBJ` chamado `parseLine:toMesh:materials:currentMaterial` presente na classe `OBJParser`.

Quadro 7 - Método `parseLine` de arquivo `OBJ`

```
1.  - (void)parseLine:(NSString *)line toMesh:(Mesh *)mesh materials:(NSMutableDictionary **)materials
    currentMaterial:(Material **)currentMaterial
2.  {
3.      NSScanner *scanner = [NSScanner scannerWithString:line];
4.      NSString *word = nil;

5.      if ([scanner scanWord:&word]) {

6.          if ([word isEqualToString:@"v"]) {
7.              [mesh addPoint:[self parsePointWithScanner:scanner]];

8.          } else if ([word isEqualToString:@"vn"]) {
9.              [mesh addNormal:[self parseNormalWithScanner:scanner]];

10.         } else if ([word isEqualToString:@"f"]) {
11.             NSArray *faces = [self parseFacesWithScanner:scanner toMesh:mesh withMaterial:*currentMaterial];

12.             for (Face3 *face in faces) {
13.                 [mesh addFace:face];
14.             }

15.         } else if ([word isEqualToString:@"vt"]) {
16.             [mesh addTexture:[self parseTextureCoordinateWithScanner:scanner]];

17.         } else if ([word isEqualToString:@"mtllib"]) {
18.             NSDictionary *newMaterials = [self parseMaterialsWithScanner:scanner];

19.             if (newMaterials && *materials) {
20.                 [*materials addEntriesFromDictionary:newMaterials];
21.             }

22.         } else if ([word isEqualToString:@"usemtl"]) {
23.             NSString *name = [self parseUseMaterialWithScanner:scanner];

24.             if (materials) {
25.                 *currentMaterial = [*materials objectForKey:name];

26. #ifdef DEBUG
27.                 if (!*currentMaterial) {
28.                     NSLog(@"Undefined material '%@'", name);
29.                 }
30. #endif
31.             }}}
32. }
```

Nota-se que a biblioteca primeiramente tenta identificar o tipo de dado contido na linha através da palavra-chave (linhas 6, 8, 10, 15, 17 e 22) para em seguida realizar a extração dos valores, diferenciado para cada tipo de informação. Após a interpretação, o resultado é adicionado nas propriedades correspondentes do objeto `Mesh` (linhas 7, 9, 13 e 16) para que seja utilizado pela biblioteca como representação da malha de polígonos de um objeto gráfico.

O processo de interpretação de dados é semelhante no arquivo `MTL` realizado pelo método `parseLine:materials:currentMaterial` da classe `MaterialParser`, como pode ser visto no Quadro 8, com a principal diferença que, neste caso, os valores são mapeados para a classe `Material` (linhas 23, 26, 29, 33, 35, 37 e 39).

Quadro 8 - Método `parseLine` de arquivo `MTL`

```
- (void)parseLine:(NSString *)line materials:(NSMutableDictionary *)materials
  currentMaterial:(Material **)currentMaterial
{
    NSScanner *scanner = [NSScanner scannerWithString:line];
    NSString *word = nil;

    if ([scanner scanWord:&word]) {
        if ([word isEqualToString:@"newmtl"]) {
            Material *material = [self parseNewMaterialWithScanner:scanner materials:materials];
            if (*currentMaterial) {
                [materials setObject:*currentMaterial forKey:(*currentMaterial).name];
            }
#ifdef DEBUG
            if ([materials objectForKey:material.name]) {...}
#endif
            *currentMaterial = material;
        } else {
            if (!*currentMaterial) {
#ifdef DEBUG
                NSLog(@"All directives, except for 'newmtl', must come after a valid 'newmtl' directive");
#endif
            } else {
                if ([word isEqualToString:@"Ka"]) {
                    GLKVector4 ambientColor = [self parseColorWithScanner:scanner];
                    (*currentMaterial).ambientColor = ambientColor;
                } else if ([word isEqualToString:@"Kd"]) {
                    GLKVector4 diffuseColor = [self parseColorWithScanner:scanner];
                    (*currentMaterial).diffuseColor = diffuseColor;
                } else if ([word isEqualToString:@"Ks"]) {
                    GLKVector4 specularColor = [self parseColorWithScanner:scanner];
                    (*currentMaterial).specularColor = specularColor;
                } else if ([word isEqualToString:@"illum"]) {
                    GLKVector4 specularColor = (*currentMaterial).specularColor;
                    GLKVector4 illumination = [self parseIlluminationWithScanner:scanner
                                                                currentSpecularColor:specularColor];
                    (*currentMaterial).specularColor = illumination;
                } else if ([word isEqualToString:@"Tr"] || [word isEqualToString:@"d"]) {
                    (*currentMaterial).transparency = [self parseTransparencyWithScanner:scanner];
                } else if ([word isEqualToString:@"Ns"]) {
                    (*currentMaterial).specularExponent = [self parseSpecularExponentWithScanner:scanner];
                } else if ([word isEqualToString:@"map_Kd"]) {
                    (*currentMaterial).diffuseTextureMap = [self parseDiffuseTextureMapWithScanner:scanner];
                }}}}
}
```

3.3.2.2 Reconstrução de modelos utilizando nível de detalhamento

A principal classe responsável pela execução do nível de detalhamento na biblioteca de LOD é a `ProgressiveMesh`. Através dela o usuário pode visualizar um mesmo modelo com diferentes níveis de detalhamento em tempo real no dispositivo iOS, informando o número desejado de vértices que este modelo deve apresentar.

Para instanciar esta classe é necessário que em seu construtor seja informada a malha de polígonos do objeto original representada pela estrutura `Mesh`.

Uma vez informada uma malha de polígonos válida a biblioteca tem condições de gerar novas malhas de polígonos.

O Quadro 9 apresenta o método `generateMeshWithVertices:cache` que gera um `Mesh` utilizando ou não um *cache* de colapsos, dependendo do valor do parâmetro booleano `cache` informado.

Quadro 9 - Método `generateMeshWithVertices:cache`

```
1. - (Mesh *)generateMeshWithVertices:(NSUInteger)vertices cache:(BOOL)cache
2. {
3.     Mesh *mesh = nil;

4.     if (cache) {
5.         mesh = [self generateMeshWithCacheWithVertices:vertices];

6.     } else {
7.         // limpa cache
8.         self.collapseMap = nil;
9.         self.permutation = nil;
10.        self.cachedTriangles = nil;

11.        mesh = [self generateMeshWithoutCacheWithVertices:vertices];
12.    }

13.    return mesh;
14.}
```

Caso o valor informado por *cache* seja falso, a biblioteca executará o método `generateMeshWithoutCacheWithVertices` (linha 11) que realizará o colapso dos vértices da malha de polígonos original até atingir o número de vértices informado pelo parâmetro `vertices`, como descrito no Quadro 10.

Os vértices e triângulos da malha de polígonos original são duplicados para novas listas (linhas 1 até 9). Em seguida, são computados os valores de colapso e o vértice de colapso de cada vértice (linhas 10 até 13), estes guardados nos respectivos atributos `objdist` e `collapse` de `Vertex`. A biblioteca, então, recupera o vértice com menor custo (linha 16) e realiza o colapso com seu vértice candidato (linha 17). Uma vez eliminado o vértice, seu candidato tem o valor de colapso recalculado. A biblioteca prossegue com este ciclo até atingir o número de vértices desejado (linhas 14 até 18).

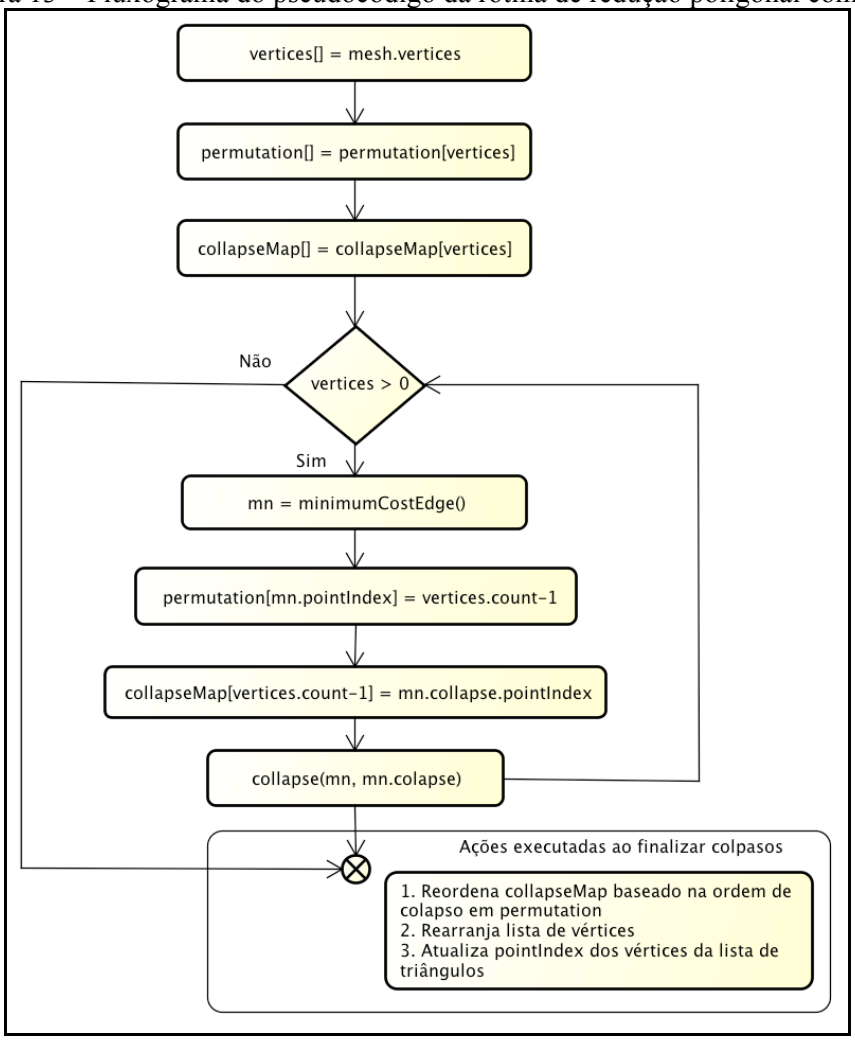
Quadro 10 - Método `generateMeshWithoutCacheWithVertices`

```
1. - (Mesh *)generateMeshWithoutCacheWithVertices:(NSUInteger)vertices
2. {
3.     Mesh *originalMesh = self.originalMesh;

4.     self.vertices = [NSMutableArray arrayWithCapacity:originalMesh.points.count];

5.     // adiciona os pontos da malha original numa nova lista
6.     for (NSUInteger i = 0; i < originalMesh.points.count; i++) {...}

7.     self.triangles = [NSMutableArray arrayWithCapacity:originalMesh.faces.count];

8.     // adiciona os triangulos da malha original numa nova lista
9.     for (Face3 *face in originalMesh.faces) {...}

10.    // calcula o custo de colapso dos vertices e seus candidatos
11.    for (Vertex *vertex in self.vertices) {
12.        [vertex computeEdgeCost];
13.    }

14.    while (self.vertices.count > vertices) {
15.        // recupera o vertice com menor custo de colapso
16.        Vertex *mn = [self minimumCostEdge];
17.        [self collapse:mn v:mn.collapse];
18.    }

19.    Mesh *newMesh = [[Mesh alloc] init];

20.    for (Face3 *face in self.triangles) {
21.        [newMesh addFace:face];
22.    }

23.    return newMesh;
24.}
```

Por outro lado, caso o valor informado para *cache* seja verdadeiro para o método `generateMeshWithVertices:cache` no Quadro 9, será executado o método `generateMeshWithCacheWithVertices` (linha 5).

Para realizar a redução poligonal utilizando *cache*, a biblioteca executa o colapso de arestas do número máximo de vértices do modelo original até não sobrar nenhum vértice. Durante cada colapso o algoritmo salva, com o auxílio de listas auxiliares, a ordem em que estes ocorreram. O algoritmo foi proposto e disponibilizado por Melax (1998). A Figura 13 demonstra este processo através de um fluxograma do pseudocódigo.

Figura 13 – Fluxograma do pseudocódigo da rotina de redução poligonal com *cache*



Primeiramente, a variável `vertices` recebe uma cópia da lista de vértices do objeto. As listas `permutation` e `collapseMap` não possuem valores e são inicializadas com o tamanho da quantidade de vértices. Em seguida, para cada vértice da lista, é identificado o ponto com custo mínimo de colapso. A variável `permutation`, então, guarda no índice do ponto de custo mínimo de colapso, a ordem de colapso. Enquanto que `collapseMap` guarda, no último índice livre, o vértice com o qual o ponto de custo mínimo realizou o colapso. Com

estas informações é possível realizar, enfim, o mapeamento de colapsos. O algoritmo, em seguida, reordena `collapseMap` baseado na ordem de colapso em `permutation`, rearranja a lista de vértices e atualiza o índice dos vértices nos triângulos.

3.3.3 Operacionalidade da implementação

Esta seção apresenta o uso da biblioteca de LOD através do aplicativo a nível de usuário. Serão apresentadas as principais funções do aplicativo para visualização da aplicação dos conceitos de LOD sobre os modelos exemplos.

3.3.3.1 Tela de listagem de modelos

Ao iniciar o aplicativo é aberta a tela de listagem de modelos responsável por apresentar os objetos 3D disponíveis no aplicativo para visualização e manipulação. Nesta tela são carregados apenas os arquivos com extensão `OBJ`. A Figura 14 ilustra a tela descrita.

Figura 14 - Tela de listagem de modelos

3.3.3.2 Tela de visualização de modelos

Ao selecionar um modelo na tela de listagem, o usuário é redirecionado para a tela de visualização de modelos para manipular o objeto escolhido, apresentado na Figura 15.

Figura 15 - Tela de visualização de modelos

Na parte superior esquerda existe o botão responsável por mostrar ou esconder as informações da cena compostas por número de *Frames Per Second* (FPS), número de vértices do modelo e o seu número de faces.

O botão na parte superior direita fica encarregado de abrir a tela de listagem de modelos caso o usuário deseje carregar um novo objeto.

Os modos de visualização do modelo 3D estão disponíveis na parte inferior esquerda do aplicativo e representam, respectivamente, o modelo com texturas, o modelo sem texturas, o *wireframe* do objeto e a nuvem de pontos do modelo.

Ao pressionar o botão inferior direito o aplicativo apresenta as principais funcionalidade de LOD disponíveis na biblioteca, como visto na Figura 16.

Figura 16 - Funcionalidades do LOD

A funcionalidade denominada `Cache`, possibilita ao usuário ativar ou desativar o uso de *cache* durante a execução do algoritmo de LOD. Ao lado, o componente *slider*, indicado pelo valor percentual, possibilita configurar a quantidade de vértices que serão apresentados.

A Figura 17 apresenta um exemplo de um modelo carregado configurado para apresentar 40% dos vértices do modelo original, sem o uso de *cache* para LOD e a apresentação do objeto com textura.

Figura 17 - Tela de exemplo com modelo apresentando 40% dos vértices originais

3.4 RESULTADOS E DISCUSSÃO

O presente trabalho apresentou o desenvolvimento de uma biblioteca capaz de interpretar modelos, no formato Wavefront OBJ, e reconstruí-los utilizando técnica de nível de detalhamento (LOD) em tempo real em dispositivos iOS.

O desenvolvimento de LOD discreto, no qual a malha de polígonos é substituída por uma nova malha a uma certa distância, foi descartada do escopo desta biblioteca por se tratar

de um processo simples de troca de uma malha por outra na estrutura do objeto gráfico.

Descartado o LOD discreto, o desenvolvimento voltou-se para o LOD contínuo onde, para tanto, foi utilizado o algoritmo de redução poligonal de Melax (1998). Este algoritmo prioriza a simplificação do modelo em áreas planares com grande concentração de vértices e se mostrou satisfatório quando utilizado em dispositivos móveis.

Foi adicionado um método de *cache* para o LOD que, primeiramente, reduz o número de vértices do modelo a zero mapeando os colapsos que foram realizados, para que, nas próximas requisições de LOD, utilize-se este *cache*. Esta implementação foi baseada no código de Melax (1998), afim de otimizar e acelerar o processamento de nível de detalhamento, e mostrou bons resultados. Entretanto, para modelos que possuem muitos vértices, a primeira etapa, de geração do *cache*, se mostrou lenta e, em alguns dispositivos como o iPhone 3GS, ao executar sucessivamente esta operação, o aplicativo acaba por fechar por uso excessivo da memória.

Também foram realizadas algumas tentativas de implementação do LOD dependente de visão utilizando como base a biblioteca VDSlib de Luebke (2000). Entretanto, devido à complexidade e o tempo para desenvolvimento do trabalho, este tipo de LOD não foi incluído na versão final da biblioteca.

3.4.1 Desempenho da biblioteca

Esta seção analisa o desempenho obtido pela biblioteca nos testes de memória e processamento do dispositivo utilizando o programa Instruments do pacote de ferramentas do Xcode.

É importante ressaltar que os testes foram realizados utilizando o simulador do iOS, um iPhone 3GS que possui 256MB de memória eDRAM, processador ARM Cortex-A8 de 600 MHz de velocidade e um processador gráfico PowerVR SGX535 e um iPad 4 com 1 GB de memória DDR2 RAM, processador *dual core* Apple Swift de 1.4 GHz e processador gráfico Quad-core PowerVR SGX554MP4. Os testes no simulador não foram incluídos neste trabalho por utilizarem recursos do computador e não representarem o cenário real dos dispositivos.

3.4.1.1 Análise de uso de memória e processamento

Foram utilizados 3 modelos para comparar o uso de memória e processamento no dispositivo. O primeiro modelo possui 98 pontos e é considerado um objeto com quantidade baixa de pontos. Já o segundo modelo, possui 1.232 vértices e possui quantidade média de pontos. Por fim, o terceiro modelo possui 4.210 vértices e, para os testes da biblioteca, é considerado um modelo com muitos vértices. A Figura 18 apresenta os modelos testados.

Figura 18 – Modelos testados com 98, 1.232 e 4.210 vértices, respectivamente

Para realizar os testes foram consideradas as seguintes operações:

a) interpretação do modelo `OBJ`;
b) aplicação de LOD sobre o objeto sem utilizar *cache*;
c) aplicação de LOD sobre o objeto utilizando *cache*.

3.4.1.1.1 Uso de memória na interpretação do modelo `OBJ`

A Tabela 1 apresenta os resultados de memória obtidos pela biblioteca de LOD no processo de interpretação dos modelos citados anteriormente. Foram mantidos apenas os resultados do iPad 4, pois o iPhone 3GS obteve valores semelhantes. Na tabela estão indicados o uso de memória antes, durante e após a finalização dos processos de interpretação.

Tabela 1 - Uso de memória na interpretação de modelos

USO DE MEMÓRIA NA INTERPRETAÇÃO DE MODELOS			
Número de vértices do modelo	Memória ocupada antes da interpretação	Maior quantidade de memória ocupada durante interpretação	Memória ocupada após interpretação
98	1.19MB	2.06MB	1.42MB
1232	1.19MB	6.41MB	3.40MB
4210	1.20MB	22.63MB	9.05MB

Os pontos de maior uso de memória, indicados na terceira coluna, correspondem à interpretação e criação de objetos do tipo `Face3` que são utilizado para representar a estrutura do objeto gráfico pela biblioteca de LOD. Também nesta etapa são alocados muitos objetos `NSConcreteValue` que são tipos de dados nativos do Objective-C utilizados para guardar tipos primitivos de dados em listas.

É possível verificar que, após a interpretação, o uso de memória diminui, em média, metade da maior quantidade de memória ocupada durante a interpretação. Isto ocorre pois os objetos temporários, para análise do arquivo `OBJ` e criação da estrutura gráfica, são removidos, restando apenas o objeto gráfico interpretado.

3.4.1.1.2 Uso de memória na aplicação de LOD

Para verificar o uso de memória na aplicação de LOD da biblioteca adotaram-se as seguintes etapas:

a) carregar o modelo a ser testado;
b) aplicar LOD de 50% a quantidade de vértices do objeto;
c) aplicar LOD de 100% a quantidade de vértices do objeto.

A Tabela 2 expressa os resultados obtidos no processo de aplicação sem utilizar *cache*. Novamente ambos os dispositivos apresentaram resultados semelhantes e, portanto, foram mantidos apenas os valores do iPad 4.

Tabela 2 - Uso de memória na aplicação de LOD sem *cache*

USO DE MEMÓRIA NA APLICAÇÃO DE LOD SEM *CACHE*			
Número de vértices do modelo	Memória ocupada antes da aplicação	Memória após aplicação de LOD 50%	Memória ocupada após aplicação de LOD 100%
98	1.43MB	1.62MB	1.72MB
1232	3.41MB	4.30MB	5.60MB
4210	9.06MB	12.09MB	16.71MB

Assim como na interpretação de modelos, as principais classes em uso na memória são

`NSConcreteValue` e `Face3`, isto porque são utilizadas muitas listas e muitos triângulos para reconstruir a nova malha de polígonos. Para os 2 primeiros modelos o uso de memória é satisfatório tanto para o iPhone 3GS quanto para o iPad, enquanto que no terceiro modelo, exclusivamente no iPhone 3GS, houve casos em que a aplicação alertava sobre o uso excessivo de memória, o que poderia fazer com que a mesma fechasse repentinamente.

Também foram realizados testes de uso de memória durante o processo de aplicação de LOD utilizando *cache*. As etapas adotadas para estes testes são as mesmas citadas no início deste capítulo e foram transcritas na Tabela 3 com os resultados do iPad 4, que representam os resultados dos 2 dispositivos.

Tabela 3 - Uso de memória na aplicação de LOD com *cache*

USO DE MEMÓRIA NA APLICAÇÃO DE LOD COM *CACHE*			
Número de vértices do modelo	Memória ocupada antes da aplicação	Memória após aplicação de LOD 50%	Memória ocupada após aplicação de LOD 100%
98	1.43MB	1.88MB	2.02MB
1232	3.42MB	8.94MB	10.06MB
4210	9.06MB	29.48MB	33.14MB

É possível verificar que, em relação a aplicação de LOD sem *cache*, o uso de memória dobrou, exceto para o modelo com 98 vértices. Isto ocorre devido ao mapeamento de colapsos armazenado, que gera mais listas e duplicação de algumas informações, como por exemplo, novas listas de triângulos do objeto com informações adicionais de mapeamento.

Novamente os 2 primeiros modelos foram executados com resultados satisfatórios em ambos os dispositivos, enquanto que o terceiro, somente no iPhone 3GS, teve vários pontos de alerta de memória emitidos pela aplicação e casos em que o aplicativo fechava. Uma possível solução para este problema seria, no momento que o aplicativo recebesse uma notificação de falta de memória, abortar o processamento do nível de detalhamento atual e executar o menor percentual de detalhamento aplicado ao objeto.

3.4.1.1.3 Processamento na aplicação de LOD

Para os testes de processamento, descritos na Tabela 4 e Tabela 5, representando o iPhone 3GS e o iPad 4, respectivamente, utilizou-se os dados de interpretação dos modelos comparando a primeira aplicação de LOD de 50% com a segunda de 100% e também a utilização e não utilização do *cache*. O processamento leva em conta o tempo de execução do método `generateMeshWithVertices:cache:` descrito na seção de implementação deste

trabalho.

Tabela 4 - Processamento na aplicação de LOD no iPhone 3GS

PROCESSAMENTO NA APLICAÇÃO DE LOD NO IPHONE 3GS				
Tipo / Vértices	SEM *CACHE*		COM *CACHE*	
	Tempo de processamento 1º LOD	Tempo de processamento 2º LOD	Tempo de processamento 1º LOD	Tempo de processamento 2º LOD
98	1.470ms	413ms	2.625ms	284ms
1232	12.667ms	5.031ms	24.389ms	3.385ms
4210	56.345ms	17.762ms	113.801ms	11.870ms

Tabela 5 - Processamento na aplicação de LOD no iPad 4

PROCESSAMENTO NA APLICAÇÃO DE LOD NO IPAD 4				
Tipo / Vértices	SEM *CACHE*		COM *CACHE*	
	Tempo de processamento 1º LOD	Tempo de processamento 2º LOD	Tempo de processamento 1º LOD	Tempo de processamento 2º LOD
98	432ms	148ms	718ms	69ms
1232	3.463ms	1.306ms	6.928ms	901ms
4210	16.546ms	4.515ms	33.441ms	3.285ms

Observa-se que, para todos os casos, o segundo LOD é mais rápido que o primeiro. Isto ocorre, pois na primeira execução ainda não foram criadas as estruturas auxiliares para geração do nível de detalhamento.

O processamento do LOD sem *cache* se mostrou superior na primeira execução, isto porque no processo sem *cache* o algoritmo apenas realiza os colapsos, enquanto que no processo com *cache* o algoritmo reduz o número de vértices do objeto a zero, independente do valor de LOD informado, mapeia os colapsos realizados e, por fim, aplica a redução com o valor de LOD. Já na segunda execução, o LOD com *cache* se sobressaiu nos testes de todos os objetos.

Nota-se também que, em relação ao iPhone 3GS, o iPad 4 pode executar, em certos casos, as operações 4 vezes mais rápido devido as suas configurações, o que diminui o tempo de espera para o processamento do LOD.

Como o iOS limita o número de FPS em 60, para demonstrar a eficácia da biblioteca, foi utilizada a ferramenta de análise de performance presente no Xcode. Esta ferramenta permite verificar a utilização de recursos de GPU e os tempos de processamento de CPU e GPU do que está sendo apresentado na tela do aparelho no momento. A Figura 19 apresenta o

processamento do modelo com 4.210 vértices sem aplicação de LOD. Já a Figura 20 apresenta o processamento do mesmo modelo com LOD de 50%.

Figura 19 – Performance sem utilizar LOD

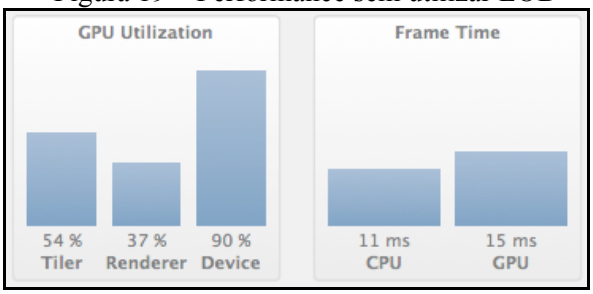
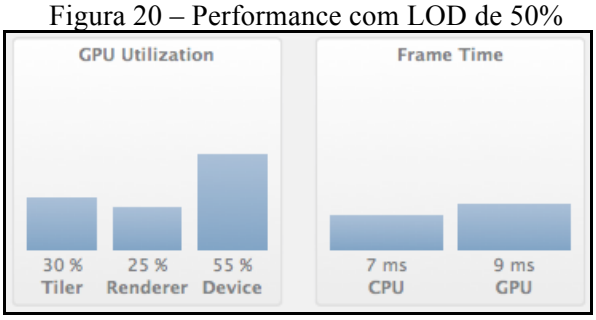


Figura 20 – Performance com LOD de 50%



É possível constatar que, sem LOD, a utilização de recursos da GPU é muito grande, chegando, por exemplo, a 90% a utilização pelo aparelho. Ao aplicar o LOD de 50% a utilização cai significativamente, bem como o tempo de processamento de *frame* pela CPU e GPU. Desta forma o aparelho utiliza menos recursos e processa o modelo de maneira mais rápida.

3.4.2 Análise de geometria de objetos reconstruídos

Como identificado por Piske (2008) e verificado também neste trabalho, o algoritmo de Melax (1998) se mostra eficiente quanto ao seu desempenho e a fidelidade geométrica. Entretanto, ao sofrer redução poligonal muito grande, um modelo pode se tornar desconexo. Isto porque o algoritmo não leva em conta a convexidade do objeto e acaba comprometendo a identificação geométrica em relação ao modelo original. A Figura 21 exemplifica o problema descrito, destacando, com uma elipse vermelha, a parte ausente do objeto que o torna desconexo.

Figura 21 – Modelo desconexo com redução

O percentual de LOD para que o objeto comece a se tornar desconexo varia de acordo com a geometria do modelo. Desta forma, é interessante o estudo de outras técnicas para diminuir o nível de detalhamento de um objeto de tal modo que o mesmo não fique desconexo.

3.4.3 Comparação com trabalhos correlatos

Nesta seção é apresentado um comparativo entre os trabalhos correlatos previamente citados e a biblioteca desenvolvida neste trabalho. No Quadro 11 o presente trabalho intitula-se `iOBJ` e tem as características analisadas junto aos trabalhos correlatos.

Quadro 11 – Comparativo da biblioteca com trabalhos correlatos

Trabalhos / Características	iOBJ	Reconstrutor de modelos 3D utilizando técnicas de nível de detalhamento	Simulador de Aviões	Unity3D
Importação de modelos 3D	Sim	Sim	Sim	Sim
LOD Discreto	Não	Não	Sim	Sim
LOD Contínuo	Sim	Sim	?	Não
Permite configurar quantidade de vértices no LOD	Sim	Sim	Não	Não
Suporte a dispositivos móveis	Sim	Sim	Não	Sim
LOD calculado em tempo real no dispositivo móvel	Sim	Não	Não	Não

Pode-se notar que a biblioteca desenvolvida se sobressai quanto ao aspecto de processar o LOD em tempo real no próprio dispositivo móvel em comparação com os demais trabalhos. E, assim como o trabalho de Piske (2008), permite a configuração da quantidade de pontos a ser apresentada.

A biblioteca `iOBJ` possui importação de modelos 3D, assim como as demais ferramentas, porém, por interpretar arquivos Wavefront OBJ, facilita o desenvolvimento de futuros trabalhos na área de computação gráfica para dispositivos móveis, que desejem utilizar de sua implementação para interpretar modelos neste formato. Os trabalhos de Oliveira (2013) e Cassaniga (2013), ainda não finalizados, já estão utilizando a interpretação de arquivos Wavefront OBJ produzida neste trabalho.

Diferente dos demais trabalhos, que tem o LOD como objetivo para aplicar em abordagens específicas, a biblioteca desenvolvida não se focou em nenhuma aplicação direta além do estudo de LOD em tempo real em dispositivos móveis.

4 CONCLUSÕES

Este trabalho apresentou o desenvolvimento de uma biblioteca que interpreta arquivos Wavefront OBJ e aplica técnica de nível de detalhamento sobre os modelos interpretados em tempo real na plataforma iOS.

Os resultados de simplificação de modelos apresentaram boas reconstruções com fidelidade geométrica e bom uso de recursos e desempenho do dispositivo móvel.

Para alguns modelos com muitos vértices, a memória limitada do dispositivo e o volume de processamento acabam por comprometer a eficácia da biblioteca. Em outro caso, uma redução de detalhamento muito grande, acaba por prejudicar a fidelidade geométrica de um modelo, tornando-o desconexo.

Em relação aos trabalhos correlatos, o trabalho desenvolvido se destaca por apresentar o uso do algoritmo aplicado em tempo real no dispositivo móvel e possibilitar a configuração de percentuais de nível de detalhamento, podendo-se verificar os resultados diretamente no dispositivo.

Também foi desenvolvido um *cache* para o processo de LOD que auxilia no tempo de execução do algoritmo de redução poligonal. Este *cache* apresentou resultados satisfatórios a partir da segunda aplicação de LOD sobre os modelos, otimizando o seu tempo de processamento e reduzindo o tempo de espera para a execução do algoritmo.

4.1 EXTENSÕES

Durante o desenvolvimento do trabalho, foram identificados alguns pontos para melhoramento e novas implementações:

a) análise e aprimoramento da fórmula que define o custo de colapso de aresta;
b) análise e aprimoramento do algoritmo de *cache* para melhora no processamento e uso de memória;
c) realização do carregamento de modelos em outros formatos que possuem outras propriedades, como, por exemplo, animações;
d) aplicação de outros algoritmos de nível de detalhamento contínuo para comparar a eficiência do algoritmo;

e) aplicação de outros algoritmos de nível de detalhamento que resolvam ou diminuam o problema de modelos desconexos;
f) otimização no algoritmo de geração de *cache* do LOD;
g) aplicação de novos algoritmos de geração de *cache* do LOD;
h) aplicação do algoritmo em outras plataformas móveis, como Android e Windows Phone;
i) aplicação de nível de detalhamento baseado em visão.

REFERÊNCIAS BIBLIOGRÁFICAS

AI AARDVARK. **LOD**. [S.l.], [2006]. Disponível em: <http://www.ai-aardvark.com/modeling/LOD_101/aia_LOD.html>. Acesso em: 6 set. 2012.

AKENINE-MÖLLER, Tomas; HAINES, Eric; HOFFMAN, Naty. **Real-time rendering**. 3rd ed. Wellesley: A K Peters, 2008.

APPLE INC. **OpenGL ES programming guide for iOS**. [San Francisco], 2010. Disponível em: <https://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/Introduction/Introduction.html>. Acesso em: 31 ago. 2012.

CASSANIGA, Mateus J. **MJ3D-Portais – biblioteca de algoritmos de portais para a plataforma iOS**. 2013. 91 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HERNANDEZ, Daniel M. et al. **A popularização dos smartphones e tablets**. 2012. 38 f. Pesquisa apresentada como requisito parcial para a obtenção do diploma. (Técnico de Nível Médio em Informática) - ETEC de Hortolândia, Hortolândia. Disponível em: < http://www.etechortolandia.com.br/novo/files/ptcc_smartphones.pdf>. Acesso em: 10 jun. 2013.

HOPPE, Hughes. **Progressive mesh**. Redmond, 1996. Disponível em: <http://research.microsoft.com/en-us/um/people/hoppe/pm.pdf>. Acesso em: 27 mar. 2013.

KRUS, Mike et al. **Levels of detail & polygonal simplification**. [S.l.], 1997. Disponível em: <http://dl.acm.org/citation.cfm?id=270966>. Acesso em: 31 ago. 2012.

LUEBKE, David et al. **Level of detail for 3D graphics**. San Francisco: Morgan Kaufmann Publishers, 2003.

_____. **VDSlib**: a view-dependent simplification package. [Virginia], [2000?]. Disponível em: <http://vdslib.virginia.edu/>. Acesso em: 20 maio 2013.

MARUCCHI-FOINO, Romain. **Game and graphics programming for iOS and Android with OpenGL ES 2.0**. [S.l.]: John Wiley & Sons, 2012. Disponível em: <http://itunes.apple.com/book/game-graphics-programming/id497171933?mt=11>. Acesso em: 18 set. 2012.

MAYA. **B1.** Object files (.obj). [S.l.], [1990?]. Disponível em: <http://www710.univ-lyon1.fr/~jciehl/Public/utiles/maya_obj_spec.pdf >. Acesso em: 16 set. 2012.

MELAX, Stan. **A simple, fast, and effective polygon reduction algorithm**. [S.l.], 1998. Disponível em: <http://www.melax.com/gdmag.pdf>. Acesso em: 27 mar. 2013.

MUNSHI, Aaftab; GINSBURG, Dan; SHREINER, Dave. **OpenGL ES 2.0 programming guide**. San Francisco: Addison-Wesley, 2009.

OLIVEIRA, Marcelo da M. **Visualização volumétrica de imagens DICOM para iOS**. 2013. 59 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

PISKE, Tiago. **Reconstrutor de modelos 3D utilizando técnicas de nível de detalhamento**. 2008. 64 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://www.bc.furb.br/docs/MO/2008/330435_1_1.pdf>. Acesso em: 27 ago. 2012.

UNITY. **UNITY:** game development tool. [S.l.], 2012a. Disponível em: <http://unity3d.com/unity/>. Acesso em: 5 set. 2012.

_____. **Unity – level of detail (pro only)**. [S.l.], 2012b. Disponível em: <http://docs.unity3d.com/Documentation/Manual/LevelOfDetail.html>. Acesso em: 5 set. 2012.