Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Visualizador de Imagens e Vídeos para Nuvem de Pontos

Natanael Dias da Silva Júnior

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientador
Prof. Dr. Bruno Luiggi Macchiavello Espinoza

Brasília
2016

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Pedro Antônio Dourado Rezende

Banca examinadora composta por:

Prof. Dr. Bruno Luiggi Macchiavello Espinoza (Orientador) — CIC/UnB
Prof. Dr. Ricardo L. de Queiroz — CIC/UnB
Prof. Dr. Flávio de Barros Vidal — CIC/UnB

CIP — Catalogação Internacional na Publicação

Silva Júnior, Natanael Dias da.

Visualizador de Imagens e Vídeos para Nuvem de Pontos / Natanael Dias da Silva Júnior. Brasília : UnB, 2016.
113 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2016.

1. Nuvem de Pontos, 2. Visualização Volumétrica, 3. Renderização Baseada em Pontos

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Visualizador de Imagens e Vídeos para Nuvem de Pontos

Natanael Dias da Silva Júnior

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof. Dr. Bruno Luiggi Macchiavello Espinoza (Orientador)
CIC/UnB

Prof. Dr. Ricardo L. de Queiroz
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
CIC/UnB

Prof. Dr. Pedro Antônio Dourado Rezende
Coordenador do Curso de Computação — Licenciatura

Brasília, 7 de dezembro de 2016

Dedicatória

Dedico este trabalho, bem como todas as minhas demais conquistas, a Deus e a minha família.

Agradecimentos

Em primeiro lugar e acima de tudo, agradeço a Deus por estar presente em todos os momentos da minha vida.

Agradeço à Universidade de Brasília e ao Departamento de Ciência da Computação por prestarem um serviço meu desenvolvimento como profissional e como pessoal.

Agradeço ao meu orientador e mentor Dr. Bruno Luiggi Macchiavello Espinoza por nortear minha busca por conhecimento durante todo processo desta obra, por todo tempo e disposição investido em mim e por toda paciência e apoio. Agradeço também ao Dr. Ricardo L. de Queiroz e ao Dr. Flávio de Barros Vidal por toda ajuda e cooperação repassados.

Agradeço a minha família por todo apoio incondicional nos momentos difíceis, pelo incentivo e pela compreensão em todas as minhas escolhas, pela dedicação para com minha pessoa, e acima de tudo, por todo amor e felicidade que me proporcionam.

A todos aqueles que de alguma forma estiveram e estão próximos de mim, fazendo esta vida valer cada vez mais a pena.

Resumo

Com o avanço tecnológico das câmeras de laser scanner 3D a obtenção de nuvem de pontos via escaneamento de modelos 3D tem se tornado cada vez mais acessível, tal tecnologia tem se mostrado importante na visualização e pós-processamento de dados geométricos. Capazes de guardar informações como cores reais e posicionamento de precisão para uma grande quantidade de detalhes escaneados, as nuvens de pontos trazem um nível alto de detalhamento no modelo representados por tipo de dados. Este trabalho se propõe a construir um visualizador de imagens e vídeos em nuvem de pontos com o uso de OpenGL, além da captura e exibição nuvens em tempo real com o uso câmeras estereoscópicas. Inicialmente o trabalho era voltado para um tipo específico de nuvem, porém os requisitos foram aumentados para abordar nuvens em formato PLY, bem como salvar nuvens capturadas por câmeras estereoscópicas neste formato. Entre os objetivos esperados para o visualizador de nuvem de pontos está o desenvolvimento de uma interface gráfica capaz de prover ao usuário toda facilitada na realização de operações de manipulação de visualização. Os resultados conquistados são apresentados ao final deste trabalho.

Palavras-chave: Nuvem de Pontos, Visualização Volumétrica, Renderização Baseada em Pontos

Abstract

With the technological advancement of the 3D scanner laser cameras the obtaining of points clouds by scanning 3D models has become increasingly accessible, such technology has been shown to be important in visualization and post-processing of Geometric data. Capable of storing information such as actual colors and precision positioning for a large amount of scanned detail, the point clouds bring a high level of detail in the model represented by this data type. This work aims to build a viewer of images and videos in points clouds with the use of OpenGL, in addition to capture and display clouds in real time using stereoscopic cameras. Initially the work was geared toward a specific type of cloud, however requirements were raised to address clouds in PLY format as well as save clouds captured by stereoscopic cameras in this format. Among the expected objectives for the point cloud viewer is the development of a graphical interface capable of providing the user with ease in performing visualization manipulation operations. The results are presented at the end of this work.

Keywords: Point Cloud, Volume Visualization, Point-based Rendering

Sumário

1 Introdução **1**
1.1 Objetivos 2
1.2 Organização deste documento 3

2 Fundamentos Teóricos **4**
2.1 Imagem e Vídeo Digital 4
2.1.1 Aquisição 4
2.1.2 Digitalização 5
2.1.3 O Pixel 5
2.1.4 Relacionamento entre pixels 5
2.1.5 Cor 6
2.2 Sinais estereoscópicos 7
2.2.1 2D + Profundidade 9
2.3 Representação de sinais 3D 10
2.3.1 Malha de Polígonos 10
2.3.2 Superfícies Paramétricas 11
2.3.3 Geometria Sólida Construtiva - CSG 13
2.3.4 Técnicas de Subdivisão Espacial 13
2.3.5 Representação por nuvem de pontos 14
2.4 Reconstrução de superfícies 15

3 Trabalhos Correlatos **17**
3.1 MeshLab 17
3.2 OpenFlipper 18
3.3 Pointshop3D 18

4 Metodologia **20**
4.1 Nuvens de Pontos 20
4.1.1 Nuvens de Pontos DB-1 21
4.1.2 Câmeras Trinoculares 21
4.1.3 Nuvens em PLY 22
4.2 Requisitos do Sistema 22
4.3 Materiais e Ferramentas Utilizadas 24
4.3.1 OpenGL 24
4.4 Implementação 28
4.4.1 Arquivos 30
4.4.2 Câmeras Estereoscópicas 33

4.4.3 Visualização de Nuvem de Pontos 34

5 Resultados **39**

6 Conclusões **43**

Referências **44**

Lista de Figuras

1.1 Representação por pontos e por malhas de polígonos [2]. 1

2.1 Representação das vizinhanças de um *pixel* P. 6
2.2 Exemplo de *pixel* conexos e não conexos. 6
2.3 Efeito de Perspectiva. 8
2.4 (a) Efeito de Sombra (b) Efeito de Luz. 8
2.5 Efeito de Oclusão. 9
2.6 Exemplo de mapa de disparidade apresentado por Boesten e Vandewalle [10]. 9
2.7 Representação por Malha de polígonos. Figura adaptada de Watt [12] . . . 10
2.8 Exemplo de um objeto representado por uma malha de polígonos [1]. . . . 11
2.9 Parâmetros u, v e geração de superfícies por quatro pontos limites [13]. . . 12
2.10 Mapeamento para geração de uma superfície por interpolação bi-linear, com fronteiras representadas por retas [13]. 12
2.11 Exemplo de um objeto contruido por meio da tecnica de CSG. Figura retirada de [12]. 14

3.1 MeshLab. 17
3.2 OpenFlipper. 18
3.3 Pointshop3D. 19

4.1 (a) *"Man"*, umas das nuvens de pontos DB-1 capturadas por Queiroz e Chou [3], 178.386 pontos e (b) Coelho de Stanford [23], 362.272 pontos. . . 20
4.2 Trecho transcrito para ASCII de um nuvem de pontos DB-1. 21
4.3 *Bumblebee*® *XB3* [26]. 22
4.4 Nuvem de pontos no formato PLY. 23
4.5 Transformações de sólidos do OpenGL, (a) Translação, (b) Rotação e (c) Escala. 26
4.6 Distorção escalar causada pelo uso de parâmetros não-uniformes. 27
4.7 Arquitetura do sistema. 28
4.8 Diagrama de classes do módulo Modelo. 29
4.9 Diagrama de classes do módulo Interface. 30
4.10 Fotografia estereográfica da estátua em frente ao Instituto de Arte de Chicago. 33
4.11 (a) Projeção perspectiva e (b) Projeção paralela ortogonal. 35
4.12 *Pipeline* da aplicação. 38

5.1 Visualização de nuvens no formato DB-1, (a) *“Man”* e (b) *“Ricardo”*. 39
5.2 Planos de projeções com dimensões erradas. 40
5.3 Planos de projeções com dimensões erradas. 41

5.4 Coelho de Stanford (a) 35.947 pontos e (b) 8.171 pontos, e Cabeça do Manequim (c) com 12.772 pontos 41
5.5 Nuvens de pontos coloridas (a) *"Body"*, (b) *"Model"*, e (c) *"Skull"*. 42

Lista de Tabelas

4.1 Tipos de Dados OpenGL 25
4.2 Diferenças entre nuvens de pontos. 31

Lista de Símbolos

(x, y) Coordenada espacial

f Função de intensidade de brilho ou nível de cinza no ponto

$i(x, y)$ Iluminância

$r(x, y)$ Reflectância

v_i Representação de um ponto no formato DB-1

2D Bidimensional

3D Tridimensional

CAD Computer-Aided Design

DB-1 Data Base 1

LISA Laboratório de Imagens, Sinais e Acústica

PLY Polygon File Format

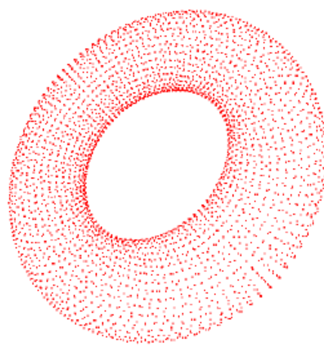
RGB Red, Green and Blue, em português Vermelho Verde Azul

Capítulo 1

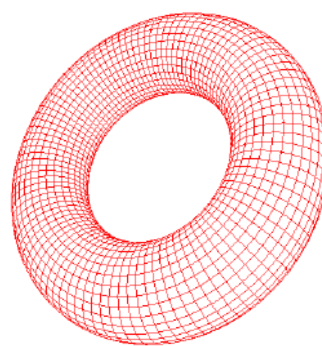
Introdução

A Computação Gráfica está presente em quase todas as áreas do conhecimento humano, da engenharia que utiliza as tradicionais ferramentas CAD (*Computer-Aided Design*), até a medicina que trabalha com modernas técnicas de visualização para auxiliar o diagnóstico por imagens [1]. Modelos tridimensionais são usados para representar entidades do mundo físico no computador, tendo o processo de descrição desses modelos chamado de modelagem. Um dos tópicos da modelagem é a representação dos objetos ou cenas. Existem várias técnicas de representações de objetos 3D e o uso de malhas de polígonos mostra-se como a forma mais comum.

Atualmente, os sistemas de escaneamento 3D tem adotado uma forma simples no que se refere a obtenção de dados sobre um objeto do mundo real, e como consequência, o problema de gerar visualizações fidedignas aos objetos escaneados está recebendo mais atenção [2]. Uma malha de polígonos é basicamente um conjunto de vértices (pontos), arestas (conexões entre os vértices) e faces (conjunto fechado de arestas). Dentre outras formas de representação existente está a nuvem de pontos, baseado na utilização de primitivas mais simples, como o ponto.



(a) Representação de toroide por pontos



(b) Representação de toroide através de malha poligonal

Figura 1.1: Representação por pontos e por malhas de polígonos [2].

Com o avanço tecnológico das câmeras de *laser scanner 3D* a obtenção de nuvem de pontos via escaneamento de modelos 3D tem se tornado cada vez mais acessível. Com uso importante na área da robótica, arqueologia, da ciência topográfica e até no mercado civil, tal tecnologia tem se mostrado importante na visualização e pós-processamento de dados geométricos. As nuvens de pontos são capazes de guardar informações como cores reais e posicionamento de precisão para uma grande quantidade de características, o que garante aos modelos escaneados um nível alto de detalhamento no modelo escaneado. Tipicamente é formado por um conjunto de pontos denso, não estruturado e sem relação de conectividade. O uso de nuvem de pontos trás limitações que vêm sendo vencidas, como por exemplo no que se refere a hardware, a grande quantidade de dados a ser guardado e processado nos milhões ou até mesmo bilhões de pontos em um modelo, vem sendo vencido com o avanço de poder computacional dos computadores, tal avanço também acontece no âmbito de software, onde, ao longo dos anos os algoritmos de processamento geométrico se tornam mais eficientes e confiáveis.

Este trabalho se propõe a construir um visualizador de imagens e vídeos em nuvem de pontos, além da captura e exibição nuvens em tempo real com o uso câmeras estereoscópicas. Inicialmente este trabalho estava focado na exibição das nuvens de pontos capturadas no trabalho de Queiroz e Chou [3], outro ponto a ser alcançado era a leitura e interpretação de arquivos no formato PLY (*Polygon File Format*) usualmente utilizados para armazenar nuvem de pontos. Com a aquisição das câmeras estereoscópicas pelo LISA - Laboratório de Imagens, Sinais e Acústica [4] - este trabalho adicionou a sua abordagem a utilização destas câmeras com a finalidade de capturar e exibir nuvem de pontos.

1.1 Objetivos

É objetivo geral deste trabalho apresentar o desenvolvimento de uma ferramenta computacional de visualização de imagens e vídeos em três dimensões representados por nuvens de pontos.

São objetivos específicos deste trabalho:

- realizar um estudo sobre a representação de superfícies em nuvem de pontos, observando conceitos como, fundamentos teóricos de imagens digitais, sinais estereoscópicos e representação de sinais 3D;
- projetar a ferramenta com as seguintes características:
  (a) arquitetar o sistema de forma modular e de baixo acoplamento;
  (b) desenvolver uma ferramenta de visualização com fácil utilização, com uma interface gráfica amigável e integrado ao sistema operacional;
  (c) prover ao usuário as funcionalidades manipulação de visualização como, rotação, escala, translação;
  (d) modelar um algoritmo de visualização de pontos coeso visando portabilidade entre bibliotecas gráficas.
- esboçar uma comparação entre os formatos de nuvens de pontos a serem suportadas pelo sistema;

- compreender o processo de aquisição nuvem de pontos através de câmeras estereoscópicas especificadas neste trabalho;
- realizar um estudo comparativo entre ferramentas de visualização de malhas geométricas que suportam nuvem de pontos;

1.2 Organização deste documento

Capítulo 2 apresenta uma fundamentação teórica sobre imagem e vídeo digital, sinais estereoscópicos e como representar esses sinais e finaliza com uma breve abordagem sobre reconstrução de superfícies.

Capítulo 3 tem como objetivo apresentar trabalhos relacionados que serviram como para criação dos objetivos e requisitos para esse trabalho.

Capítulo 4 apresenta a metodologia utilizada no desenvolvimento da aplicação proposta por este trabalho, descreve os formatos de nuvens de pontos suportados pela ferramenta, explicita os requisitos do sistema, bem como, explicita os materiais e ferramentas utilizados no desenvolver da metodologia e finaliza o capítulo explicando a implementação da aplicação de visualização de nuvem de pontos.

Capítulo 5 exibe um resumo dos resultados obtidos enquanto o Capítulo 6 finaliza esse trabalho justificando as limitações observadas e com descrição de propostas para trabalhos futuros.

Capítulo 2

Fundamentos Teóricos

O objetivo deste capítulo é introduzir uma base teórica e identificar técnicas que possam ser utilizadas na solução do problema proposto. É apresentado uma fundamentação teórica sobre imagem e vídeo digital, representação de sinais e da reconstrução de superfícies.

2.1 Imagem e Vídeo Digital

Imagem digital é uma projeção discreta 2D de uma cena real 3D. Matematicamente uma imagem digital pode ser representada por uma função discreta. Uma imagem monocromática pode ser descrita como uma função bidimensional $f(x, y)$, onde (x, y) representa uma coordenada espacial e f é a intensidade de brilho ou nível de cinza no ponto (x, y). No caso de imagens coloridas existem mais de um sinal sendo uma função $f(x, y)$ para cada, como por exemplo imagens que usam o padrão RGB, formados pelas cores primárias aditivas vermelho, verde e azul, em inglês *Red*, *Green* e *Blue* [5], podem ser descritas como

$$f(x, y) = f_r(x, y) + f_g(x, y) + f_b(x, y). \tag{2.1}$$

Já um vídeo digital é um conjunto de imagens digitais, aqui chamados de quadros, repetidos sequencialmente em um curto espaço de tempo, provocando a sensação de movimentação. Isto se deve a amostragem do sinal de vídeo no tempo.

A conversão de uma cena real para uma imagem ou vídeo digital requer duas etapas indispensáveis: aquisição e digitalização.

2.1.1 Aquisição

Aquisição é o processo de captura e conversão de uma cena real tridimensional em sinais eletrônicos por meio de diversos mecanismos de captura como Scanner e/ou Câmera Digital.

O primeiro processo da aquisição de imagens é a redução de dimensionalidade, o mecanismo de captura deve converter uma cena real tridimensional em um representação bidimensional. Quando a imagem é formada a partir de um processo físico, os seus valores são proporcionais energia irradiada por uma fonte. A função $f(x, y)$ então é representada pela relação de dois componentes, a iluminância $i(x, y)$ e reflectância $r(x, y)$.

A iluminância é o resultado de um ou mais fontes de luz sobre um objeto e a reflectância é o resultado da fração de luz que incide em um ou mais objetos e é transmitido ao ponto (x, y). A função $f(x, y)$ então toma a forma do produto dessas duas componentes.

$$f(x, y) = i(x, y)r(x, y) \tag{2.2}$$

onde

$$0 < i(x, y) < \infty \tag{2.3}$$

e

$$0 < r(x, y) < 1 \tag{2.4}$$

Câmeras Digitais geralmente usam um dispositivo chamado CCD (Charge Coupled Device - Dispositivo de Carga Acoplada) que atua como um conversor de energia luminosa incidente capturada pelas lentes em sinais elétricos analógicos.

2.1.2 Digitalização

O sinal elétrico analógico obtido na fase de aquisição deve passar por uma discretização espacial e em amplitude para obter um formato desejável para o processamento computacional[5], a etapa de digitalização converte esse sinal em uma matriz de M por N pontos.

$$f(x, y) = \begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0, N-1) \\ f(1,0) & f(1,1) & \cdots & f(1, N-1) \\ \vdots & \vdots & \vdots & \vdots \\ f(M-1, 0) & f(M-1, 1) & \cdots & f(M-1, N-1) \end{bmatrix}$$

2.1.3 O Pixel

A imagem digital é formada por um conjunto finito de elementos que possuem um local (x, y) e um valor, que representa a intensidade de luminância nas imagens monocromáticas ou uma cor nas imagens coloridas. Cada elemento desse conjunto é o resultado da amostragem espacial, sendo chamado de *pixel*(aglutinação de *Picture Element*).

2.1.4 Relacionamento entre pixels

Cada *pixel* é normalmente representado por um *byte*, podendo assim assumir 256 valores diferentes. Eles possuem relacionamentos básicas utilizadas em diversas técnicas de processamento de imagem, nesta seção citarei alguns desses relacionamentos.

Vizinhança de um pixel

Um *pixel* P localizado na posição (x, y) possui quatro vizinhos posicionados ao seu redor. Como Gonzales e Woods [6] explicam os vizinhos localizados na horizontal e vertical de P, cujas coordenadas são: (x+1, y), (x-1, y), (x, y+1), (x, y-1) são chamados de vizinhança-4, contendo os *pixels* que estão a 1 unidade de distância de (x, y). Nessa vizinhança não se aborda os vizinhos diagonais do *pixel* P.

Os vizinhos diagonais do *pixel* P possuem as coordenadas

$$(x+1, y-1), (x+1, y+1), (x-1, y+1)e(x-1, y-1) \tag{2.5}$$

e juntos com os *pixels* da vizinhança-4 formam a vizinhança-8, que é o conjunto de todos os vizinhos de *pixel* P, como visto na Figura 2.1.

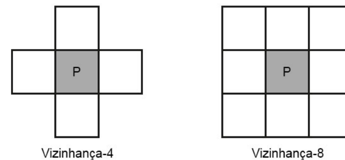


Figura 2.1: Representação das vizinhanças de um *pixel* P.

É importante observar que alguns dos *pixels* vizinhos não estarão presentes na imagem quando P estiver localizado na borda da imagem.

Conectividade

A conectividade entre *pixels* é um conceito importante usado para estabelecer limites, ou bordas, de objetos e componentes de regiões em uma imagem. Segundo Pedrini e Schwartz [7], para verificar se dois *pixels* são conexos é necessário verificar se eles são adjacentes, ou seja, se eles são vizinhos, e se satisfazem determinados critérios de similaridade, tais como a intensidade de cinza. Para verificar a adjacência podemos considerar a vizinhança-4 ou a vizinhança-8, resultando respectivamente na adjacência-4 e adjacência-8. Por exemplo, a Figura 2.2 ilustra um exemplo de *pixels* conexos, ela mostra em uma imagem binária com valores de 0 ou 1, considerando a vizinhança-4 os *pixels* P5 e P4 são conexos pois possuem o mesmo valor, já os *pixels* P2, P6 e P7 não possuem adjacência com P5 já que possuem valores diferentes.

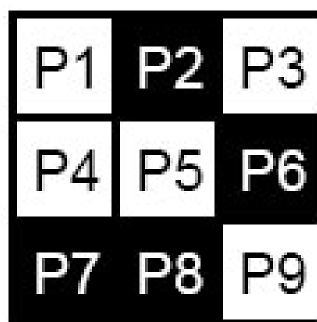


Figura 2.2: Exemplo de *pixel* conexos e não conexos.

2.1.5 Cor

Em cada *pixel* além de uma posição (x, y) é associado um valor para codificar a luminância ou a cor de um ponto. Quando queremos representar a luminância usa-se

somente um número, como por exemplo 1 byte que permite $2^8 = 256$ variações, pode-se assumir valores de 0 a 255. Já para representar uma cor podemos assumir 3 componentes e usar 2^{24} o que dá mais de 16 milhões de possibilidades.

Um modelo de cor (ou também chamado espaço de cor) é uma especificação matemática capaz de descrever cada cor por meio de uma tupla.

Os modelos de cores mais comumente usados são *CMYK* (*Cyan*, *magenta*, *yellow*, *blak*) modelo subtrativo usado preferencialmente em impressões, e para a exibição em monitores os modelos *RGB* (*Red*, *Green*, *Blue*), *HSL* ou *HSV*, baseados em matiz, saturação, brilho e *YUV* definido por duas componentes de cor, luminancia (Y) e crominancia (U e V). Nas seções seguintes serão detalhados os modelos *RGB* e *YUV*, ambos empregados neste trabalho.

RGB

O padrão *RGB* define três cores primárias aditivas e baseia-se no mecanismo de funcionamento do olho humano. Uma cor nesse modelo é formado por tupla tridimensional contendo a quantidade de vermelho, verde e azul, por exemplo no computador, usando um sistema de 24 *bits* (8 *bits* para cada elemento da tupla) temos 16,7 milhões de cores e cada componente da tupla podendo receber um valor de 0 a 255 (usamos $2^8 = 256$ níveis de cor para cada canal, logo $2^8_{vermelho}$ x 2^8_{verde} x $2^8_{azul} = 2^{24} = 16.777.216$ cores).

YUV

O Modelo *YUV* usa dois componente, luminância (Y) e crominância (U e V). A luminância é a representação do brilho (em níveis de cinza), enquanto a crominância é a representação das cores. Esse modelo proporciona a separação do brilho e cor, e foi criado para permitir a transmissão de imagens coloridas e monocromáticas em dispositivos que não tem suporte a cores.

Neste trabalho usaremos a formula para realizar a conversão de *YUV* para *RGB*, onde $K_r + K_g + K_b = 1$.

$$R = Y + \frac{1 - K_r}{0.5} V \tag{2.6}$$

$$G = Y - \frac{2K_b(1 - K_b)}{1 - K_b - K_r} U - \frac{2K_r(1 - K_r)}{1 - K_b - K_r} V \tag{2.7}$$

$$B = Y + \frac{1 - K_b}{0.5} V \tag{2.8}$$

2.2 Sinais estereoscópicos

A Estereoscopia é o fenômeno de vermos uma imagem de dois pontos de vistas diferentes, remete a natureza óptica da nossa fisiologia, afinal possuímos dois olhos apontados para uma vista porém separados por, em média 65 milímetros [8]. Cada olho recebe uma imagem distinta, e é por essa leva diferença de ângulos que possuímos uma visão binocular que nos permite a percepção de profundidade.

Na seção anterior, discorremos sobre imagens digitais como uma representação da visão monocular, criando uma imagem plana de duas dimensões. A estereoscopia pode ser gerada por processos artificiais com a finalidade de conseguir a noção de profundidade e a representação de cenas tridimensionais. A seguir mostraremos algumas técnicas utilizadas para exprimir tal sensação.

Perspectiva - permite a noção de aproximação de um objeto [8], ou seja, determinar se um objeto se encontra mais perto ou mais longe do observador. Quando mais longe o objeto se encontra da câmera, menor será seu tamanho na imagem, o contrário também é valido, objetos possuem maior tamanho na representação quando se encontram mais perto da câmera. Na Figura 2.3 podemos ver dois retângulos com o mesmo tamanho, porém o efeito da perspectiva gerado pelas linhas que convergem para um ponto no horizonte e definem um ponto de fuga, podemos distinguir que o retângulo mais ao centro tem tamanho real maior.

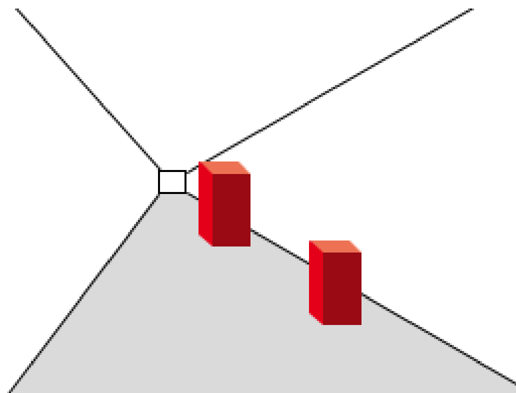


Figura 2.3: Efeito de Perspectiva.

Distribuição de Luz e Sombra - este oferece uma maior sensação de relevo e volume a imagem. A sombra também traz uma visão espacial, podemos, por exemplo, perceber a distância de um objeto a um plano de apoio ao olharmos para sua sombra, na Figura 4.1 (a) é perceptível que a esfera não está fixada ao plano de apoio. Luz e sombra são técnicas comumente usadas para dar a sensação de 3D em projeções 2D [8], como podemos ver na Figura 4.1 (b).

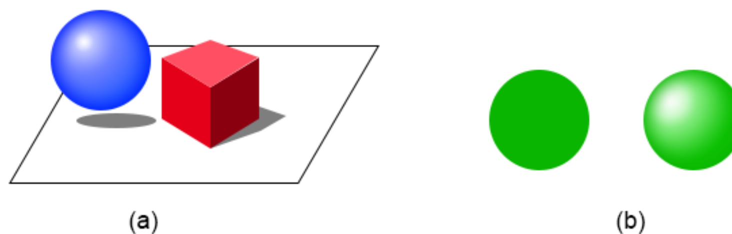


Figura 2.4: (a) Efeito de Sombra (b) Efeito de Luz.

Superposição de Imagens - ou oclusão, é responsável por trazer a sensação de proximidade de objetos por meio da sobreposição a outros objetos. Objetos mais próximos do observador escondem parcialmente ou integralmente objetos mais distantes. A Figura

2.5 ilustra esse efeito, podemos facilmente perceber que forma está a frente da outra em relação ao observador.

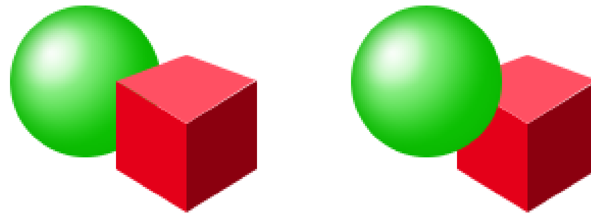


Figura 2.5: Efeito de Oclusão.

2.2.1 2D + Profundidade

É uma alternativa para imagem e vídeo estéreo convencional, no qual as imagens estéreos são geradas por interpolação de imagens e de mapas de profundidade [9], tal abordagem apresenta bons resultados e uma ótima relação custo benefício. O mapa de profundidades pode ser definido como uma matriz contendo os valores de depressão ou altura dos pontos que compõem uma determinada superfície, no caso de captura de modelos 3D a partir de fotografias, esta matriz contém os valores referentes a distâncias dos pontos à câmera, geralmente são representados em uma escala de cinza, então a intensidade de cada *pixel* representa a profundidade como pode ser observado na Figura 2.6.

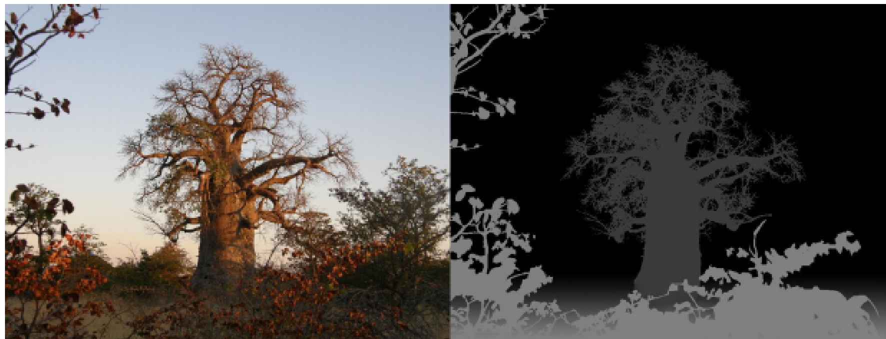


Figura 2.6: Exemplo de mapa de disparidade apresentado por Boesten e Vandewalle [10].

O processo de geração de imagens estereoscópicas com essa técnica é realizado através das seguintes etapas individuais:

- Captura das imagens
- Análise das imagens
- Estimação do mapa de profundidade

A captura das imagens é feita geralmente com duas câmeras alinhadas horizontalmente, entre as câmeras, pode se usar a distância 65 milímetros, que como citado anteriormente corresponde a distância média entre os olhos do ser humano [8], dando ao sistema de câmeras um embasamento inspirado no sistema de visão dos seres humanos, porém isso não regra estrita. Após captura das imagens deve-se encontrar pontos em uma das imagem e ver o quanto ela mudou em outra imagem, com a finalidade de descobrir o deslocamento horizontal - no caso de câmeras alinhadas horizontalmente - desse ponto e depois calcular a distância desse ponto para às câmeras, ou seja, sua profundidade.

2.3 Representação de sinais 3D

Nas seção inicial deste capítulo estudamos a representação de cenas do mundo real em um plano bidimensional (2D), partiremos agora para um representação sólidos tridimensionais (3D). Partiremos da definição de um sólido como um subconjunto fechado e limitado do espaço Euclidiano tridimensional E^3, dada por Martti Mäntylä [11], explicitando assim que, a ideia de sólido está associada a ideia de algo essencialmente tridimensional.

A representação tridimensional de objetos sólidos pode ser dividida em duas grandes categorias: Representação por bordas (B-rep) e representação por Volume.

Existem várias técnicas de representação 3D, Watt [12] cita, de acordo com a importância e frequência de utilização: Malha de Polígonos, Superfícies Paramétricas, Geometria Sólida Construtiva (CSG), Técnicas de Subdivisão Espacial e por fim a Representação Implícita. As representações de malha de polígonos e de subdivisão espacial consistem numa aproximação da forma do objeto que está sendo modelado, representando apenas a superfície do objeto. Já as superfícies paramétricas e a geometria sólida construtiva, por sua vez, são representações exatas, ou seja, representando o volume inteiro do sólido.

2.3.1 Malha de Polígonos

Esta é a forma mais comum de se representar um modelo 3D, onde o objeto é aproximado por uma malha de faces poligonais. Uma maneira simples de se representar uma malha de polígonos é definindo uma lista de vértices (coordenadas x, y e z) e como tais vértices devem ser ligados para que formem um conjunto de polígonos fechados (chamados de faces), geralmente triangulares ou quadráticos. De um modo geral, os objetos possuem superfícies curvas e as faces do polígonos são aproximações a tal superfície.

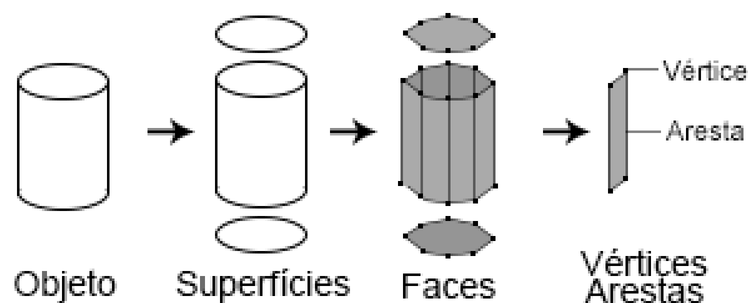
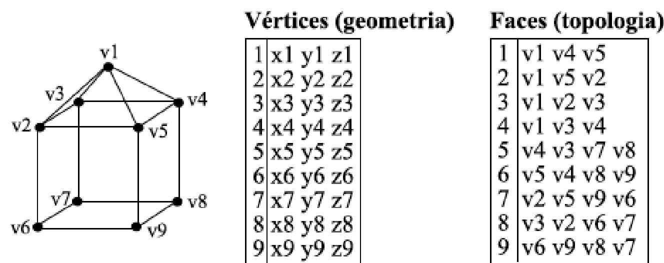


Figura 2.7: Representação por Malha de polígonos. Figura adaptada de Watt [12]

Uma das maneiras de representar um objeto usando malha de polígonos é da forma explícita, onde cada polígono é representado por uma lista de coordenadas dos vértices que o constituem, uma aresta é definida por dois vértices consecutivos e entre o último e primeiro da lista de maneira circular. Vale notar que um vértice pode pertencer a vários polígonos e à existência de uma aresta em dois polígonos indica a adjacência deles, ou seja, apesar de tal abordagem ser de simples entendimento, o fato de ocupar muito espaço e não trazer um representação direta da partilha de vértices ou arestas, são dificuldades a serem consideradas ao se utilizar essa maneira de representação.



Vértices (geometria)

1	x1 y1 z1
2	x2 y2 z2
3	x3 y3 z3
4	x4 y4 z4
5	x5 y5 z5
6	x6 y6 z6
7	x7 y7 z7
8	x8 y8 z8
9	x9 y9 z9

Faces (topologia)

1	v1 v4 v5
2	v1 v5 v2
3	v1 v2 v3
4	v1 v3 v4
5	v4 v3 v7 v8
6	v5 v4 v8 v9
7	v2 v5 v9 v6
8	v3 v2 v6 v7
9	v6 v9 v8 v7

Figura 2.8: Exemplo de um objeto representado por uma malha de polígonos [1].

Outra forma é a representação por ponteiros para Lista de Vértices, onde cada polígono é representado por uma lista de índices (ou ponteiros) para uma lista de vértices.

2.3.2 Superfícies Paramétricas

As superfícies paramétricas possuem grande flexibilidade de manipulação de curvas, por isso são mais usadas em objetos que possuem essa característica.

Superfícies Paramétricas Bilineares

Uma Superfícies Paramétricas Bilineares (também chamado de patch bilinear) é um quadrilátero no espaço tridimensional, definido por quatro pontos (vértices) e arestas retilíneas, não necessariamente coplanares. O nome bilinear vem do fato da superfície ser descrita através da interpolação bilinear dos quatro pontos, ou seja, para se obter uma superfície é necessário associar seus pontos aos limites e gerar o interior usando interpolações lineares sucessivas, primeiro em uma direção e posteriormente em outra direção. Segundo o livro Computação Gráfica: teórica e prática [13], a forma mais simples para definir a parametrização da superfície é considerar o espaço dos parâmetros representados por uma área unitária limitada pelos pontos (0,0),(0,1),(1,0) e (1,1), como exemplificado na Figura 2.9.

Para gerar uma superfície a partir dos pontos A,B,C e D, deve se associar estes pontos aos limites do espaço paramétrico de (0,0),(0,1),(1,0) e (1,1), o interior então é gerado através de duas interpolações lineares. Na primeira interpolação, serão geradas as retas AD e BC, que correspondem aos limites com parâmetro $u = 0$ e $u = 1$, assim qualquer ponto E sobre a reta AD é obtido na forma de

$$E = (1 - v)A + vD \tag{2.9}$$

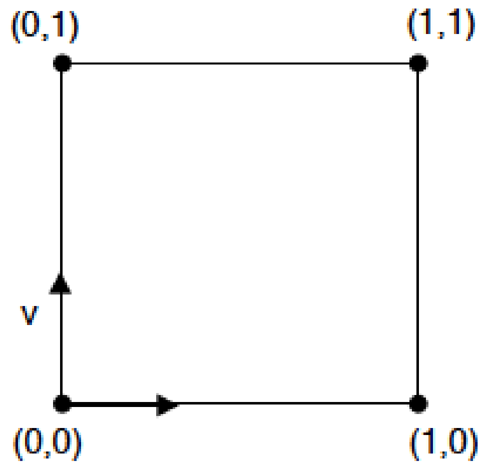


Figura 2.9: Parâmetros u, v e geração de superfícies por quatro pontos limites [13].

sendo que, quando $v = 0$ E é o ponto A, e se $v = 1$ E representa o ponto D. Na primeira interpolação também pode obter qualquer ponto F sobre a reta BC na forma de

$$F = (1 - v)B + vC, \tag{2.10}$$

agora com $v = 0$ correspondendo ao ponto B e $v = 1$ ao ponto C. A Figura 2.10 descreve o mapeamento da geração de superfície.

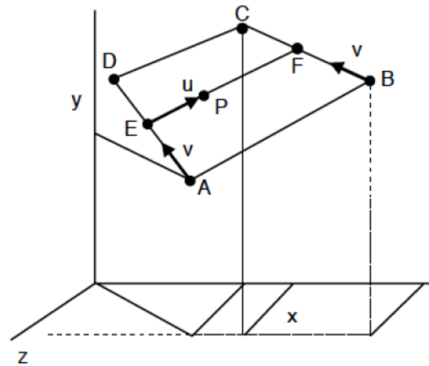


Figura 2.10: Mapeamento para geração de uma superfície por interpolação bi-linear, com fronteiras representadas por retas [13].

Com os pontos E e F é possível utilizar outra interpolação linear para gerar um ponto na superfície, considerando o parâmetro u com a fórmula

$$P(u, v) = (1 - u)E + uF. \tag{2.11}$$

As duas interpolações lineares podem ser reunidas em um uma única fórmula representando uma interpolação bilinear

$$P(u, v) = (1 - u)(1 - v)A + (1 - u)vD + u(1 - v)B + uvC. \tag{2.12}$$

Superfícies Paramétricas Bicúbicas

Também chamado de patch Bicúbico, uma superfície Bicúbica são curvas quadrilaterais. Cada pedaço da malha é definido por uma fórmula matemática semelhante à descrita na seção anterior que, indica sua posição e forma no espaço tridimensional. A superfície é descrita através de um conjunto de pontos de controle dispostos em uma matriz 4x4, chamada *control hull*. Essa forma de representação, além de mais flexível, permite obter uma infinidade de formas alterando somente as especificações matemáticas ou os pontos do *control hull*.

As superfícies paramétricas bicúbicas são descritas como:

$$P(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} P_{ij}B_i(u)B_j(v) \tag{2.13}$$

Como vantagem há a facilidade de cálculo de diversas propriedades, como massa, volume e área, devido a ser descrito por formulações matemáticas. Porém, ao alterar uma forma de um determinado pedaço, existe uma dificuldade em manter a suavidade em relação aos pedaços vizinhos. Outro problema apresentado por essa forma de representação é seu alto custo de tempo para geração de uma visualização realística (render) e memória para representação de objetos complexos.

Um patch bicúbico é derivado de uma curva definida matematicamente por um polinômio de grau três. As principais superfícies bicúbicas são as superfícies Hermite, Bézier e Spline, nas quais as curvas de contorno são definidas por essas curvas.

2.3.3 Geometria Sólida Construtiva - CSG

Geometria Sólida Construtiva (CSG - *Constructive Solid Geometry*) é a técnica de modelagem de sólidos que define os objetos através de operações booleanas e combinações de sólidos simples (primitivas) e/ou objetos já modelados [13]. O objeto é representado através de uma árvore binária, onde os nós são primitivas, objetos já modelados, operadores booleanos ou a representação de translação, rotação, escala, etc.

A Figura 2.11 apresenta uma árvore binária que representa a modelagem de um objeto. Segundo Cassal [14], a CSG está baseada em olhar para um objeto como se ele estivesse dividido em partes e com a combinação dessas partes construir o objeto como um todo.

2.3.4 Técnicas de Subdivisão Espacial

Esta técnica de representação divide e decompõe o espaço em uma grade tridimensional formada por cubos chamados *voxels*, estes podem ser vazios ou conter uma parte de um objeto. Esta representação é análoga a representação bidimensional de uma imagem, onde os *pixel* ((Pic)ture (El)ement) se torna o *voxel* ((Vol)ume (El)ement).

Azevedo e Aura [13] descrevem algumas características importantes desta forma de representação:

- Para determinar se um dado ponto pertence ou não ao sólido, basta verificar se o ponto pertence a algum dos *voxels*;
- É fácil determinar se dois objetos se interferem;

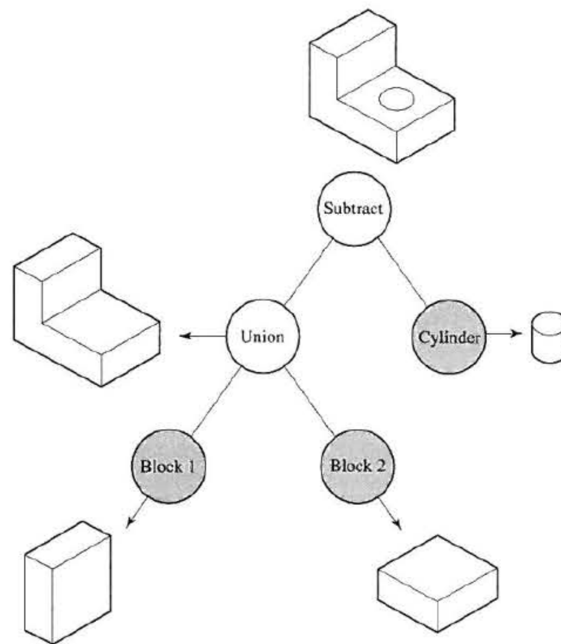


Figura 2.11: Exemplo de um objeto contruido por meio da tecnica de CSG. Figura retirada de [12].

- É fácil a realização de operações *booleanas*, como união, diferença e intersecção entre sólidos;
- É fácil a obtenção da propriedade de massa e volume do objeto, bastando saber o volume de uma das partes e multiplicar pelo número total de divisões ocupadas.

Esta representação apresenta desvantagem ao se trabalhar com objetos complexos e detalhados devido ao grande custo de armazenagem necessária.

2.3.5 Representação por nuvem de pontos

Com o avanço das tecnologias de escaneamento 3D vem proporcionando a obtenção de modelos com complexidade geométrica mais avançada e um maior número dados. As nuvens de pontos são um tipo de representação de superfície conhecidas como não estruturadas [15], que é um tipo de representação explícita e não possui um modelo de dados associado, tal característica oferece a este tipo de representação uma simplicidade estrutural, que junto a possibilidade do processamento direto dos dados capturados pelos scanners 3D, dá suporte à uma reamostragem eficiente para os modelos 3D que usam esta representação.

A nuvem de pontos é um conjunto de pontos, tipicamente, denso, não ordenado, não estruturado, e sem informação de conectividade. Esta representação foi inicialmente proposta por Levoy e Whitted [16] em 1985. Tem como sua forma mais simples somente com as coordenadas espaciais dos pontos:

$$P = \left\{ \mathbf{p}_i \in \mathbb{R}^3, i \in \{1, 2, ..., n\} \right\} \tag{2.14}$$

O objetivo da renderização baseada em pontos (*Point-based Rendering*) consiste na visualização de uma amostragem de pontos como uma superficie contınua, além de diversas técnicas de representação, processamento, e edição de superficies baseadas em pontos, desenvolvidas com a finalidade cobrir um *pipeline* completo para a renderização, que vai desde a aquisição (escaneamento) até a visualização.

A representação por pontos traz algumas desvantagens como a falta de conectividade, topologia ou quantidade diferenciais além da teoria matemática para manipular tal representação. Outro problema é observado ao se trabalhar com *scanners*, esta representação apresenta ruído com maior concentração próximo as feições afiadas, sendo necessário algoritmos de suavização para a redução de ruídos. Algoritmos específicos para suavização de nuvens, bem como para, reconstrução de superfície, estimação de normais, dizimação e amostragem têm sido desenvolvidos como podemos observar no livro de Gross e Pfister [17].

2.4 Reconstrução de superfícies

A reconstrução de superfícies baseia se na obtenção de modelos geométricos complexos através de um conjunto de informações. A representação de objetos através de modelos geométricos vem se tornando uma área de grande importância em muitos campos de pesquisa, tais como, computação gráfica, visão computacional, CAD (*computer aided design* - desenho assistido por computador), além de aplicações na medicina, cartografia e na indústria. Inúmeros grupos de pesquisa vêm desenvolvendo diversos métodos para reconstrução de superfícies, estes podem ser divididos basicamente em duas classes.

Os algoritmos de reconstrução a partir de Seções Planares modelam objetos geométricos a partir de um conjunto de curvas bidimensionais, situadas em planos paralelos. Essas curvas são as bordas ou superfícies que delimitam um objeto sólido. Dado um conjunto de bordas Ci, $i = 1, 2, \ldots, n$, onde cada elemento Ci está situado no plano $z = zi$, a superfície S é a interpolação dessas bordas no plano $z = zi$. Este método tem suas principais aplicações na areá de Imagens Médicas através de tecnologias como Tomografia Computadorizada e Ressonância Magnética.

A outra classe de métodos é a de reconstrução de superfície a partir de Nuvem de Pontos. Denominamos nuvem de pontos, um conjunto de pontos espalhados espaço, inicialmente não relacionados e pertencentes a uma mesma superfície (MULLER, 1997)[18]. Esta classe possui métodos que podem ser subdivididos ou agrupados várias formas, como por exemplo, a classificação segundo a abordagem utilizada no processo, da seguinte maneira:

- Métodos baseados em esculpimento;
- Métodos baseados em funções implícitas;
- Métodos incrementais;
- Métodos baseados em modelos deformáveis.

Métodos baseados em esculpimento ou decomposição espacial, são aqueles que geram uma malha de triângulos e a suavizam até que aproxime do objeto real. Os métodos

baseados em funções implícitas, consistem em usar a amostra para calcular uma função de distância f, a superfície é estimada a partir de f. Os métodos incrementais constroem o objeto de maneira incremental a partir de um elemento inicial, por exemplo, através da conexão dos dois pontos mais próximos da amostra, obtém-se de maneira iterativa as demais arestas da superfície. E por fim, os métodos em modelos deformáveis são aqueles que geram o objeto através da deformação de uma superfície até que se alcance a nuvem de pontos.

Capítulo 3

Trabalhos Correlatos

O desenvolvimento de aplicações de visualização de malhas geométricas 3D, tem sido abordado de varias perspectivas distinta. Este Capitulo fornece uma revisão de varias ferramentas de processamento de malhas 3D, destacando suas principais características. As ferramentas foram escolhidas por suas semelhanças ao requisitos postos a esse trabalho e que possuíssem uma relevância literária, sendo citado por outros trabalhos ou possuindo trabalhos sobre eles.

3.1 MeshLab

MeshLab [19] é um sistema de processamento de malhas 3D, extensível e *open-source*, desenvolvido pela *Visual Computing Lab* do Instituto de Ciência e Tecnologia da Informação "A. Faedo"(ISTI-CNR) [20] com uso de renderização OpenGL baseada em *Shader*. Como resultado, o MeshLab mostra-se como um aplicativo de visualizador de malha, onde um objeto 3D, armazenado em uma variedade de formatos, pode ser carregado e inspecionado de forma interativa e fácil, simplesmente clicando e arrastando o mouse sobre a própria malha do objeto. O usuário, assim que carregado um modelo 3D pode trabalhar com um conjunto de filtros paramétricos ou com diversas ferramentas interativas disponibilizadas pelo *software*.

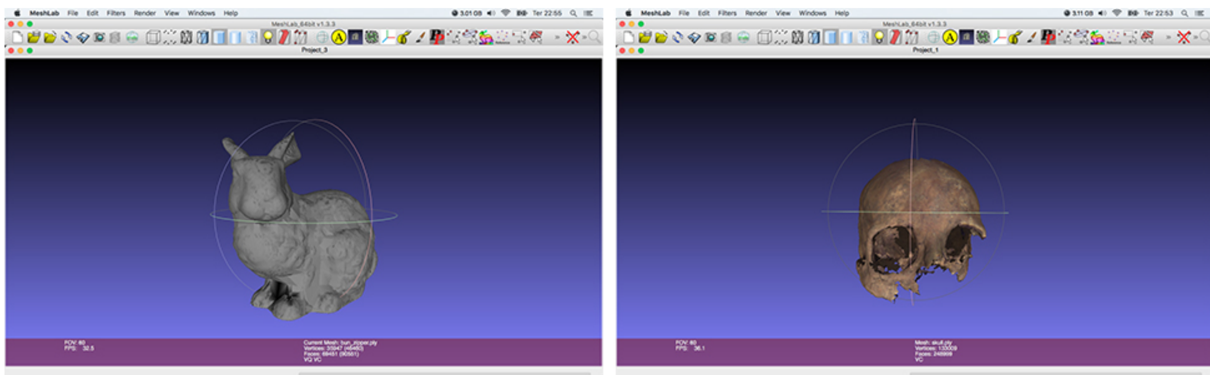


Figura 3.1: MeshLab.

O MeshLab tem suporte de importação e exportação para vários tipos de formatos como PLY, STL, OFF, OBJ, 3DS, COLLADA, PTX, X3D, VRML, porém o mais relevante para análise é o suporte a nuvem de pontos através do formato PLY.

3.2 OpenFlipper

OpenFlipper [21] é um *framework* de processamento, modelagem e renderização de geometria de código aberto extensível e tem objetivo de fornecer uma plataforma comum de desenvolvimento de software para a maioria da comunidade de processamento de geometria, para isso o sistema possui uma interface de script que permite a meta-implementação flexível de funções de controle que se baseiam na funcionalidade disponível nos módulos individuais implementados como plugins em C ++. O *framework* utiliza OpenMesh [22] para a representação de superfície, o que lhes permite uma variedade de diferentes formatos de arquivo como OFF, OBJ, PLY, entre outros.

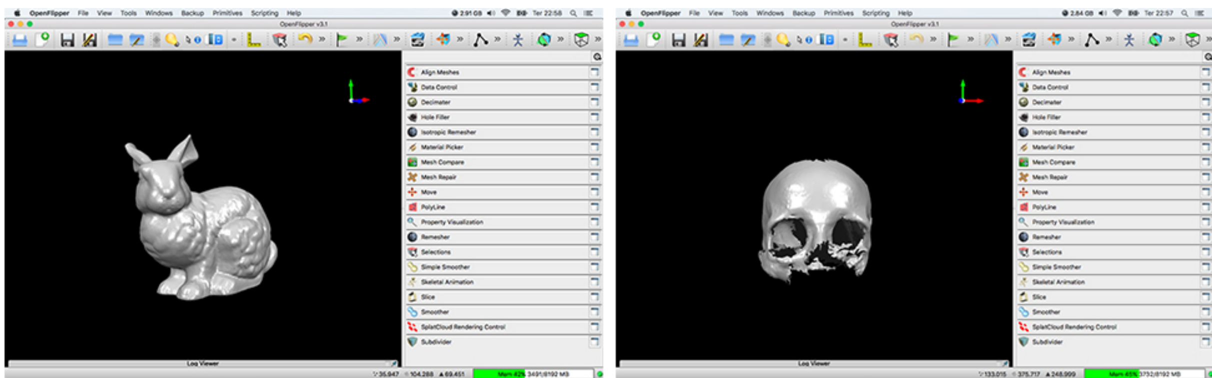


Figura 3.2: OpenFlipper.

Em comparação com outras estruturas de software de processamento de geometria como o Meshlab, por exemplo, existem duas principais características que as diferem, o módulo de script que permite aos usuários executar o OpenFlipper no modo batch e facilita a definição de interfaces customizadas e a estrutura *scenegraph* que pode manipular objetos múltiplos simultaneamente, mesmo com representações diferentes.

3.3 Pointshop3D

Pointshop3D é um sistema interativo de edição de aparência da geometria 3D baseado em pontos, que tem como característica marcante a generalização editores de pixel 2D convencionais, além de suportar uma grande variedade de diferentes técnicas de modelagem de geometrias 3D como, limpeza, texturização, escultura, filtragem e reamostragem.

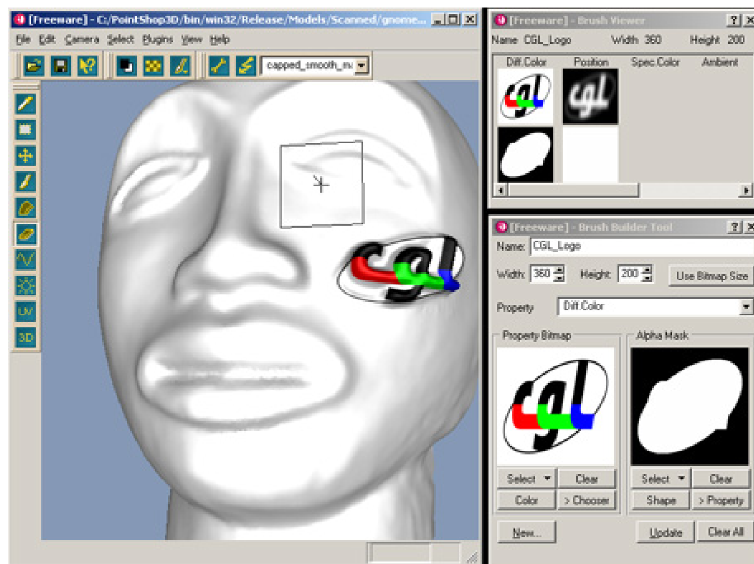


Figura 3.3: Pointshop3D.

Capítulo 4

Metodologia

Este capítulo consiste na descrição da metodologia de implementação de um software de visualização de nuvens de pontos. A Seção 4.1 apresenta os formatos de nuvens que o sistema aceitará, a Seção 4.2 discursa os requisitos do sistema, já a Seção 4.3 apresenta os materiais e ferramentas utilizados no desenvolvimento da aplicação e por fim a Seção 4.4 descreve a implementação do sistema.

4.1 Nuvens de Pontos

Para desenvolvimento deste trabalho foram utilizadas nuvens de pontos de três fontes diferentes, inicialmente nuvens de pontos do banco de dados usadas no trabalho de Queiroz e Chou [3], as quais foram a motivação inicial para realização deste trabalho e a partir de agora serão referidas como "nuvens de pontos DB-1" (database 1). Também foram feitas capturas de nuvens utilizando as câmeras estereoscópicas no Laboratório de Imagens, Sinais e Acústica (LISA) [4], do departamento de Ciências da Computação da Universidade de Brasília. Por fim, também foram utilizados nuvens conhecidas como a do Coelho de Stanford [23] e a Cabeça do Manequim [24].

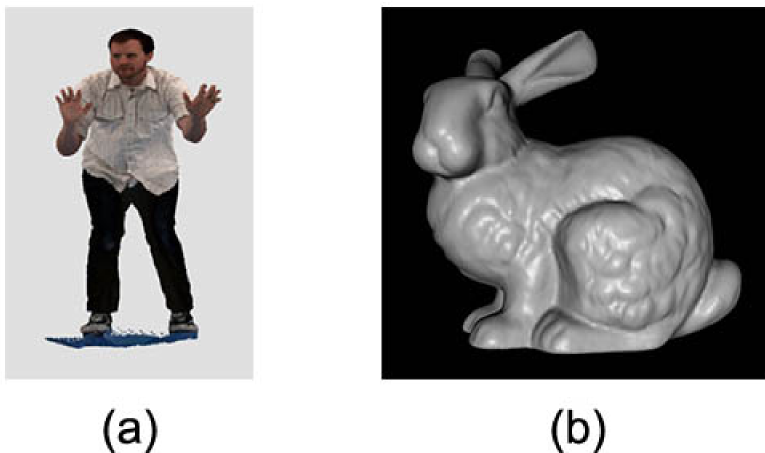


Figura 4.1: (a) *"Man"*, umas das nuvens de pontos DB-1 capturadas por Queiroz e Chou [3], 178.386 pontos e (b) Coelho de Stanford [23], 362.272 pontos.

4.1.1 Nuvens de Pontos DB-1

As nuvens DB-1 [3] possuem um modelo simplificado visando a sua utilização em aplicações em tempo real, cada *voxel* não transparente está associado a informações espaciais e uma cor (RGB ou YUV) na forma

$$v_i = [x_i, y_i, z_i, Y_i, U_i, V_i] \,.$$

Assim a nuvem é representado por uma lista de *voxels* $\{v_i\}$. As nuvens foram disponibilizadas em arquivo .mat do MATLAB [25], foi desenvolvido um pequeno algoritmo para transcrever as nuvens para arquivos de texto ASCII, a Figura 4.2 exemplifica um trecho de uma nuvem após transcrição. O formato dos arquivos transcritos seguiu o princípio de um modelo simplificado, os dados foram dispostos sequencialmente (x, y, z, Y, U e V) e separados por espaço, sendo que cada linha representa um ponto.

```
123 102 159 132 112 156
123 103 159 125 113 156
124 102 159 132 113 156
124 103 159 125 113 156
110 115 165 110 115 153
. . .
```

Figura 4.2: Trecho transcrito para ASCII de um nuvem de pontos DB-1.

4.1.2 Câmeras Trinoculares

Também foram utilizadas nuvens capturadas trinoculares no Laboratório de Imagens, Sinais e Acústica (LISA) [4], através das câmeras de visão estereoscópicas *Bumblebee*® *XB3* (Figura 4.3) da *Point Grey Research* [26] em conjunto com as bibliotecas de desenvolvimento *FlyCapture*^{TM} *SDK* e *Triclops*^{TM} *SDK* responsáveis pela aquisição de dados 3D e imagens coloridas retificadas.

O *Bumblebee*® *XB3* é um sistema composto por três câmeras digitais Sony ICX445 CCD com a distância de 12 centímetros entre elas, possui uma interface FireWire para comunicação em alta velocidade, segundo fabricante pode capturar imagens com uma resolução de 1024x768 *pixels* a uma taxa de 20 frames por segundo. O *FlyCapture*^{TM} *SDK* é um pacote de desenvolvimento de software projetado para controle das câmeras da *Point Grey Research* e o *Triclops*^{TM} *SDK* é uma biblioteca de desenvolvimento capaz de produzir em tempo real mapas de disparidades, através das capturas realizadas pelas três câmeras, o software analisa as imagens e estabelece a correspondência entre seus *pixels*.

Figura 4.3: *Bumblebee®* *XB3* [26].

4.1.3 Nuvens em PLY

Algumas nuvens usadas para testes durante o desenvolvimento deste trabalha foram disponibilizadas no formato PLY (*Polygon File Format*), desenvolvido pela Stanford University para o armazenamento de coleções de polígonos, tem dois sub formatos ASCII e binário, o qual é usado para diminuir o tamanho dos arquivos e aumentar a velocidade de processamento destes. O formato PLY descreve uma lista de elementos, usualmente de vértices e faces, porém podendo conter outros elementos. Os elementos são descritos através de propriedades, como sua localização espacial (x, y, z), cor, transparência, dentre outras propriedades que possam ser especificadas. Todo organização do arquivo, isto é, seus elementos e propriedades, comentários e outras informações pertinentes ao objeto armazenado no arquivo é descrito no cabeçalho do formato. A Figura 4.4 apresenta um nuvem de pontos no formato PLY. Foi utilizada a biblioteca RPly [27] para leitura e escrita dos arquivos deste formato.

4.2 Requisitos do Sistema

O desenvolvimento do sistema buscou basicamente atender a necessidade da visualização de nuvens de pontos no formato DB-1 descritas na Seção 4.1.1. Além disso foi especificado a necessidade de visualizar nuvens de pontos em um formato usualmente definido e nuvens capturadas em tempo real por câmeras estereoscópicas. Para isso, foram definidos requisitos com intuito de especificar exatamente quais os objetivos do sistema, foram eles:

- **Visualização de nuvem de pontos:** o sistema deve ser capaz de exibir os pontos em um plano tridimensional.
- **Leitura de diferentes formatos de nuvem de pontos :** na Seção 4.1 foram especificados quais formatos de nuvens serão lidos pelo sistema.
- **Interatividade com a visualização:** a interface gráfica deve prover a interação do usuário para funções de manipulação do sistema.

ply	
format ascii 1.0	ascii/binário
comment made by Greg Turk	
comment this file is a cube	
element vertice 8	define o elemento "vertice"e a quantidade de 8 no arquivo
property float x	vertex contém coordenada "x"do tipo float
property float y	coordenada y também é uma propriedade de vertice
property float z	coordenada z também pertence a vertice
element face 6	existem 6 elementos "face"no arquivo
property list uchar int vertex_index	"vertex_index"é uma lista de inteiros
end_header	fim do cabeçalho
0 0 0	início da lista de vértices
0 0 1	
0 1 1	
0 1 0	
1 0 0	
1 0 1	
1 1 1	
1 1 0	
4 0 1 2 3	início da lista de faces
4 7 6 5 4	
4 0 4 5 1	
4 1 5 6 2	
4 2 6 7 3	
4 3 7 4 0	

Figura 4.4: Nuvem de pontos no formato PLY.

- **Conexão com as câmeras estereoscópicas:** o sistema deve ser capaz de se conectar com as câmeras do Laboratório de Imagens, Sinais e Acústica do CiC, bem com ser capaz de realizar captura e exibição de nuvens de pontos em tempo real.
- **Aplicativo desktop:** o software deve ser desenvolvido um aplicativo desktop.
- **Exportar nuvem de pontos no formato PLY:** o sistema deve ser capaz de exportar as nuvens capturadas em tempo real no formato PLY.

O sistema deve ser capaz de exibir as nuvens de pontos tridimensionalmente em um plano cartesiano, para isso, é necessário que o sistema consiga ler os arquivos contendo as nuvens, fazer transformações das coordenadas, visto que cada tipo de nuvem citada na seção anterior tem seus próprios sistemas de coordenadas, fazer os tratamentos de coloração, como por exemplo a transformação do sistema de cores YUV para RGB, e por fim exibir o objeto sólido representado pela nuvem.

O sistema também deve exibir as nuvens captadas em tempo real pelas câmeras de visão estereoscópica, para isso capturar imagens sequencialmente, submetê-las ao processo de geração de mapa de disparidade para que possa ser calculada coordenadas (x, y, z) dos pontos.

Para facilitar a visualização de nuvens é essencial o desenvolvimento de funções especiais de interação em resposta às ações do usuário. O usuário poderá manipular alguns parâmetros de visualização do objeto, ou seja, o sistema deverá trazer as funções de rotação, translação e zoom. Tais funcionalidades poderão ser acionadas através do mouse ou do teclado.

Outro requisito referente ao sistema é a interface gráfica. O sistema deve possuir uma interface amigável ao usuário e que permita o acionamento das funcionalidades citadas anteriormente.

4.3 Materiais e Ferramentas Utilizadas

Inicialmente foi escolhido a linguagem de programação Java com intuito de facilitar a implementação dos requisitos referentes a Interface Gráfica, porém com a adição da captura das câmeras em tempo real nos objetivos, alteramos o desenvolvimento do sistema para a utilização da linguagem C++ [28] junto com Ambiente Integrado de Desenvolvimento Qt Creator [29]. O Software teve seu funcionamento atestado sobre os sistemas operacionais OS X El Capitan e Linux Ubuntu 14.01.

Para o desenvolvimento das rotinas gráficas foi utilizado a biblioteca OpenGL [30], padrão mundial no desenvolvimento de aplicações gráficas interativas 2D e 3D, tal biblioteca foi escolhida principalmente devida sua alta portabilidade, esta vantagem foi de vital importância na alteração da linguagem de desenvolvimento do sistema, além disso OpenGL possui outras vantagens como estabilidade, aceleração de hardware e funções de efeitos 3D em real-time. A próxima Seção explicita o uso das rotinas gráficas utilizados no desenvolvimento deste trabalho.

4.3.1 OpenGL

A OpenGL (*Open Graphical Library*) pode ser definida como uma interface de software (API – *Aplication Program Interface*) portável e rápida para a programação de rotinas gráficas para modelagem 2D ou 3D, com aproximadamente 700 funções para especificação de objetos e operações necessárias para a produção de aplicações gráficas interativas [31], esta Seção discorre sobre as funções mais importantes utilizadas no desenvolvimento do software.

A OpenGL aplica uma convenção para os nomes de suas funções que indica de qual biblioteca a função faz parte e, freqüentemente, quantos e que tipos de argumentos a função tem, na seguinte forma
`<Biblioteca><ComandoRaiz><ContadorArgumentos`$_{opcional}$`><TipoArgumentos`$_{opcional}$`>`.

Como a portabilidade é uma das características desejáveis desse trabalho, optamos por nos atentarmos ao uso da biblioteca *core* do OpenGL, por isso veremos o prefixo `gl` nos comandos utilizados. Para tornar o código mais portável ainda, o OpenGL define tipos de dados próprios, que podem ser mapeados para linguagens de programação como Java ou C, a Tabela 4.1 indica alguns tipos de dados com seus respectivos mapeamentos para a linguagem C e seus sufixos utilizados em algumas funções.

Por exemplo a função `glColor3f`, podemos observar que o prefixo `gl` representa a biblioteca `gl` *core* da OpenGL, e o sufixo `3f` significa que a função possui três valores de ponto flutuante como parâmetro.

Tabela 4.1: Tipos de Dados OpenGL

Tipo de Dado	**Equivalente em C**	**Tipo de Dados OpenGL**	**Sufixo**
8-bit integer	signed char	GLbyte	b
16-bit integer	short	GLshort	s
32-bit integer	int ou long	GLint, GLsizei	i
32-bit floating-point	float	GLfloat, GLclampf	f
64-bit floating-point	double	GLdouble, GLclampd	d
8-bit unsigned integer	unsigned char	GLubyte, GLboolean	ub
16-bit unsigned integer	unsigned short	Glushort	us
32-bit unsigned integer	unsigned long	GLuint,GLenum,GLbitfield	ui

Primitivas

O OpenGL permite que, com apenas algumas primitivas simples, tais como pontos, linhas e polígonos, é possível criar objetos complexos e cenas utilizando simples primitivas gráficas que podem ser combinadas de várias maneiras [32].

Para desenhar pontos, linhas e polígonos é necessário passar uma lista de vértices entre duas chamadas de funções OpenGL
`glBegin(TYPE` *modo*`)`
e
`glEnd()`
onde o argumento *modo* indica qual primitiva iremos desenhar, podendo ser definido como uma linha - ou sequências de linhas - entre os vértices da lista, ou até mesmo triângulos, quadrados ou polígonos conectados pelos vértices passados entre as funções. Para se desenhar pontos basta passar o argumento **GL_POINTS**, indicando assim para o OpenGL que cada vértice da lista é um ponto a ser desenhado.

Para especificar um vértice a ser usado na descrição de um objeto geométrico temos a função
`glVertex[ 2 3 4 ]{ s i f d }(TYPE coords)`
tendo de dois a quatro parâmetros como entrada representando as coordenadas espaciais (x, y, z, w), podendo ter seus valores de entrada como inteiros, ponto flutuantes ou até *bytes*.

Pode-se também definir uma cor para cada vértice usando a função

```
glColor3{ b s i f d ub us ui }(TYPE r, TYPE g, TYPE b)
glColor4{ b s i f d ub us ui }(TYPE r, TYPE g, TYPE b, TYPE a)
```

sendo que para função com tres parametros de entrada temos uma tripla RGB podendo assumir qualquer um dos tipos de dados mapeados pela OpenGL. Já para a função com quatro valores, além da tripla RGB, temos um *alpha* indicando a opacidade da cor, tendo seu valor padrão como 100% opaco e com valor de 0% indicando um ponto configuração de cor transparente.

O OpenGL também nos permite alterar a configuração do tamanho do vértice através da função
`glPointSize(GLdouble size)`
que recebe como parâmetro o tamanho do vértice, por padrão 1.

Pode-se também realizar transformações de modelagem, que são manipulações lineares

nos objetos realizadas por meio de operações matriciais, como rotação, translação ou mudanças na dimensão de escala (Figura 4.5).

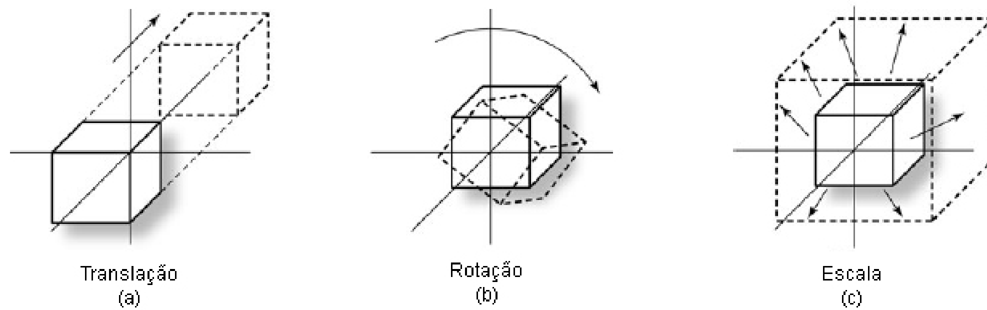


Figura 4.5: Transformações de sólidos do OpenGL, (a) Translação, (b) Rotação e (c) Escala.

Translação

Para transladar um objeto usando OpenGL temos a função
`glTranslate{ f d }(TYPE x, TYPE y, TYPE z)`
que recebe como parâmetro as distâncias de deslocamento em cada eixo na forma de inteiros ou ponto flutuante, constrói uma matriz apropriada e multiplica-a na pilha de matriz atual para gerar o deslocamento desejado. Vale observar que o uso de $(0, 0, 0)$ como argumento para essa função é a operação de identidade, ou seja, não tem efeito sobre um objeto ou seu sistema de coordenadas local.

Escala

Usando OpenGL podemos mudar as dimensões de escala de um objeto, ou seja, multiplicar a matriz atual por uma matriz que encolha ou estique ao longo dos eixos. Para isso temos a função
`glScale{ f d }(TYPE x, TYPE y, TYPE z)`
com parâmetros de entrada os fatores de escala, cada coordenada x, y, e z de cada ponto no objeto é multiplicada pelo argumento correspondente x, y ou z.porém, utilização do fator de escala não-uniforme afetará o objeto visualizado, como pode ser observado na Figura 4.6.

Rotação

A função para realizar uma rotação é
`glRotate{ f d }(TYPE angle, TYPE x, TYPE y, TYPE z)`
e tem como parâmetro o ângulo de rotação e as coordenadas de um vetor que determina o eixo de rotação. Para a rotação ser feita em torno de um dos eixos principais, deve-se definir x, y e z apropriadamente como os vetores unitários nas direções destes eixos. O ângulo de rotação é no sentido anti-horário medido em graus e especificado pelo ângulo do argumento. Nos casos mais simples, a rotação é em torno de apenas um dos sistemas de coordenadas eixos cardinais x, y ou z, podemos então realizar a função para os três

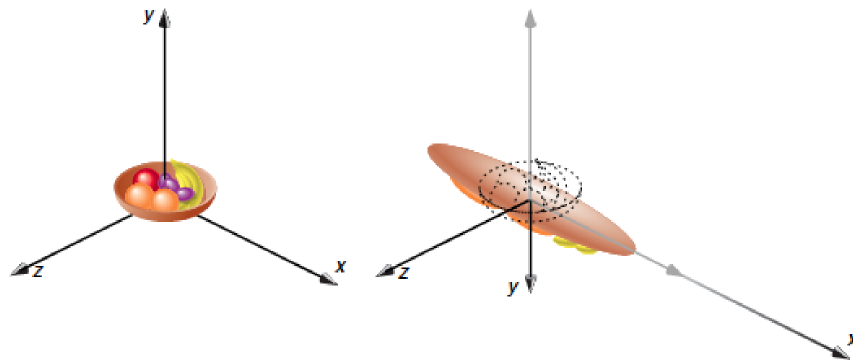


Figura 4.6: Distorção escalar causada pelo uso de parâmetros não-uniformes.

eixos
```
glRotate(angleX, 1, 0, 0)
glRotate(angleY, 0, 1, 0)
glRotate(angleY, 0, 0, 1)
```

Projeção Paralela Ortogonal

Definiremos uma projeção paralela ortogonal por meio da função
`glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar)`
que utiliza seis parâmetros para definir um volume de visão [32], onde *left* e *right* especificam as coordenadas esquerda e direita indicando o plano de corte vertical, *bottom* e *top* as coordenadas inferior e superior e o plano de corte horizontal, e *zNear* e *zFar* indicam as coordenadas de proximidade e distância indicando o eixo de profundidade.

Projeção Perspectiva

Já projeção em perspectiva é selecionada com a função
`glFrustrum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)`
que também utiliza seis valores para definir o volume de visão, que é definido pelos parâmetros (*left*, *bottom*, *-near*) e (*right*, *top*, *-near*) especificam as coordenadas (x, y, z) dos cantos inferior esquerdo e superior direito, respectivamente, do plano de recorte. Os parâmetros *near* e *far* dão as distâncias do ponto de vista aos planos de recorte.

Esta projeção pode ser usada também com a função da biblioteca de utilitários (glu) do OpenGL
`gluPerspective(GLdouble fovy, GLdouble aspect,GLdouble near, GLdouble far)`
onde *fovy* é o ângulo do campo de visão no plano yz com seu valor podendo dentro do intervalo $[0.0, 180.0]$. O parâmetro *aspect* segue como a razão de aspecto que determina a área de visualização na direção x e seu valor é a razão entre o plano x (largura) e o plano y (altura) [13], os parâmetros *near* e *far*.

É importante citar que os parâmetros para ambas funções devem ser positivos.

4.4 Implementação

A arquitetura geral do sistema, conforme apresentado na Figura 4.7 é baseado em uma arquitetura modular, procurando garantir um abstração entre os módulos do sistema.

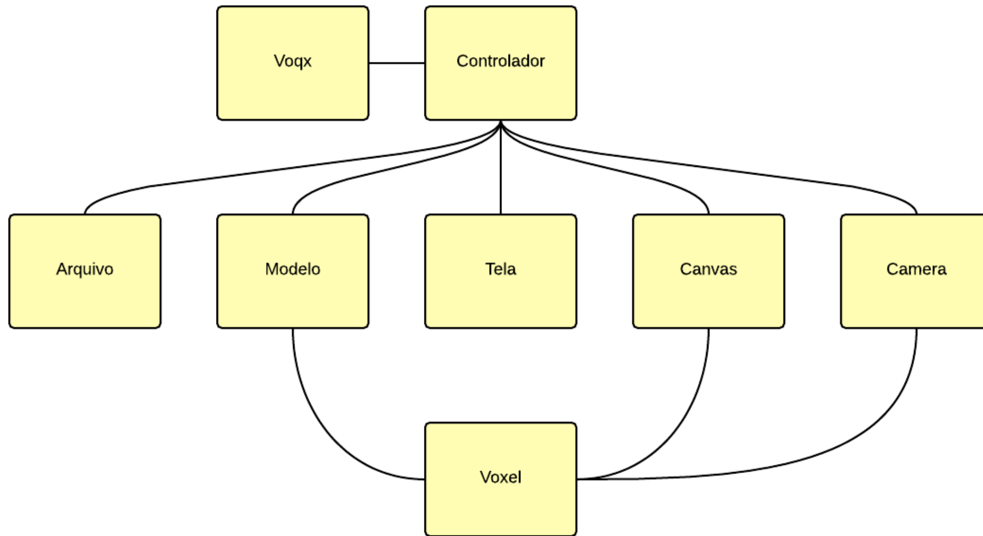


Figura 4.7: Arquitetura do sistema.

A classe **Controlador** serve como um centralizador da aplicação com o objetivo de prover uma abstração entre os módulos do sistema, de maneira que, por exemplo, toda a parte de implementação do módulo de arquivos possa ser alterada sem que atrapalhe a visualização das nuvens. A classe **Controlador** também é responsável pelo controle de todo *pipeline* de execução do sistema.

O módulo **Arquivo** possui somente a classe com o mesmo nome, nela é realizada a leitura e extração de dados dos arquivos, esta classe também é responsável por salvar modelos em PLY e imagens estáticas dos modelos. A classe **Arquivo** é de suma importância para definir os formatos de nuvens aceitos pela aplicação, ela utiliza a biblioteca RPly [27] para interpretar arquivos PLY e descreve as "regras" de leituras para arquivos das nuvens de pontos do formato DB-1. A estrutura modular do sistema é pensada para garantir que ao se trocar a maneira de ler os arquivos ou ao se alterar os formatos aceitos pelo sistema, não seja necessária a alteração modelagem estrutural da nuvens, para isso classe **Controlador** define abstração dos modelos de dados para a classe **Arquivo**.

O módulo **Modelo** é responsável pela modelagem estrutural das nuvens, podendo representar objetos ou cenas estáticas com apenas uma nuvem, ou a representação de tempo real/vídeo com modelagem de várias nuvens de pontos exibidas sequencialmente. A implementação de um objeto ou cena estática é realizada pela classe **Cena**, que, além de guardar uma referência para o arquivo onde a nuvem está armazenada, guarda informações como formato da nuvem, caso haja nome do objeto (caso não haja, este toma como valor o nome do arquivo) e informações retiradas do processamento da nuvem como dimensões do objeto, quantidade de pontos, entre outros. A representação de video é realizada pela classe **Video** que possui alguns atributos a mais que a classe **Cena** e tem o

processamento de dados diferente. As nuvens de pontos também são representadas nesse módulo, sendo descrita através de uma lista de pontos implementa por meio da classe **Voxel** que basicamente tem como atributos as informações espaciais do ponto (x, y, z) no formato de ponto flutuante e uma tripla contendo sua cor no formato RGB onde cada elemento é salva como um *char*. A modelagem de **Cena** possui uma lista de **Voxel**, enquanto a class **Video** possui uma lista frames, onde o frame é uma lista de **Voxel**, ou seja, o vídeo possui uma lista de listas. Inicialmente optou-se por abordar o video dessa maneira para simplificar a execução da visualização sequencial de nuvens de pontos, além disso não havia atributos ou métodos a serem adicionadas que justificassem a modelagem do frame como uma classe, porém esta é uma abordagem inicial e pode facilmente ser alterada caso haja a necessidade de implementar os frames de um vídeo como objeto do sistema.

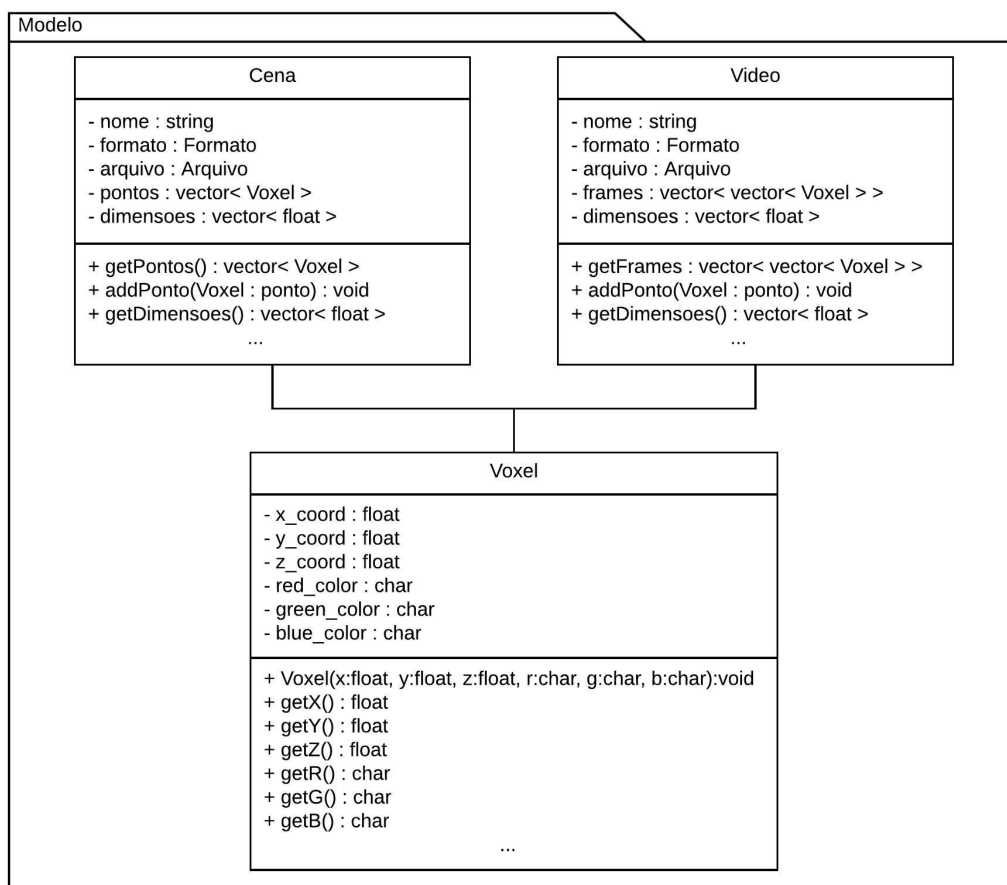


Figura 4.8: Diagrama de classes do módulo Modelo.

Os módulos **Tela** e **Canvas** são responsáveis pela parte visual da aplicação - módulo **Interface** -, sendo a **Tela** a implementação de toda interface do sistema e **Canvas** a implementação do módulo gráfico com OpenGL responsável pela visualização das nuvens de pontos. Ambos módulos possuem implementações diferentes dependendo do tipo de nuvem a ser visualizado, na visualização de um vídeo por exemplo a classe **CanvasVideo** deve percorrer uma lista de frames e a classe **TelaVideo** deve fornecer ao usuário uma interface capaz de prover a interação com o vídeo. Já no da visualização de nuvens de pontos capturadas pelas câmeras, a classe **CanvasCamera** deve ser capaz de requisitar

ao **Controlador** a leitura do buffer da câmera e retorno de nuvens de modo contínuo. A Figura 4.9 exibe o diagrama de classes - simplificado - desse módulo.

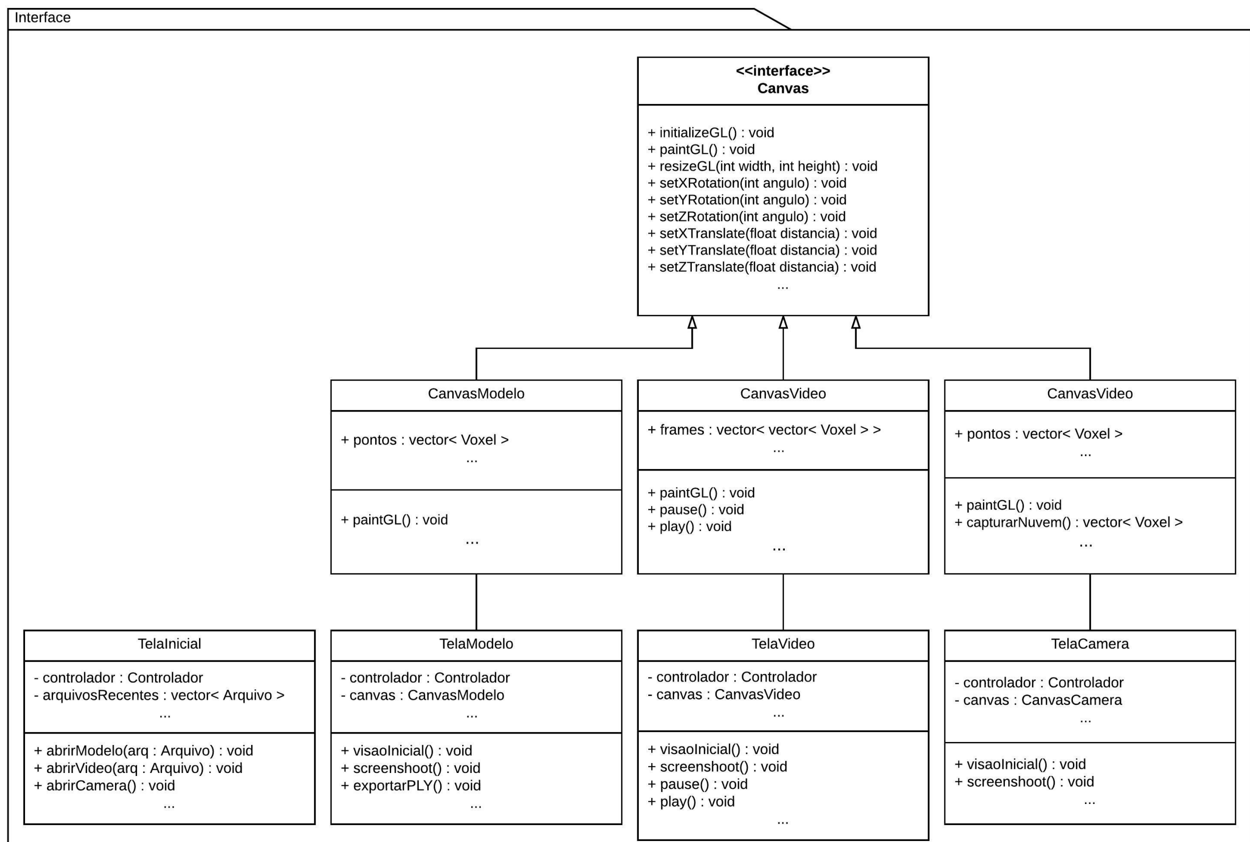


Figura 4.9: Diagrama de classes do módulo Interface.

E por fim, o módulo **Camera** é responsável pela conexão e controle das câmeras estereoscópicas com o sistema, bem como a captura das nuvens de pontos. A classe **Camera** é responsável por capturar imagens e realizar o processo de criação de nuvens, vale ressaltar que assim como na implementação do frame para a classe de **Video** a classe **Camera** não usa outra classe para modelar a estrutura dos seus dados de saída, o que se é retornado diretamente para a classe **CanvasCamera** é uma lista de **Voxel**. Além de não se notar uma estrutura que justificasse a modelagem e implementação de uma nova estrutura de dados, buscou-se simplificar o processo buscando conexão mais rápida da câmera com a exibição das nuvens geradas por ela.

4.4.1 Arquivos

Na Seção 4.1 explicita-se quais formatos de nuvens de pontos seriam suportadas por esse trabalho, tal requisito traz um desafio inicial para o projeto, a leitura e interpretação destes diferentes formatos. Um exemplo desta diferença entre as maneiras de guardar uma nuvem de pontos pode ser percebida ao se focar em como estes formatos expressão as coordenadas espaciais de cada ponto, com uma forma básica todas usam uma tupla (x, y, z) para isso, porém, como pode ser visto na Tabela 4.2, a maneira e variação de valores possíveis para essa tupla não são uniformes entre os formatos, por exemplo, as

nuvens de pontos DB-1 usam valores inteiros de 0 a 511, enquanto as nuvens do formato PLY tem um variação de valores definida por quem cria os arquivos, usualmente são valores números reais e com variações entre -1 e 1. Como as coordenadas espaciais de cada ponto serão armazenada terá forte influência no momento de montar o plano cartesiano para a visualização da nuvem.

Tabela 4.2: Diferenças entre nuvens de pontos.

Man, nuvem DB-1			Coelho de Stanford			Cabeça do Manequim de Hoppe		
x	y	z	x	y	z	x	y	z
123	102	159	-0.0378297	0.12794	0.00447467	6.79831	8.16914	204.439
123	103	159	-0.0447794	0.128887	0.00190497	6.11263	25.7879	28.7497
124	102	159	-0.0680095	0.151244	0.0371953	0.083602	26.4344	26.4514
124	103	159	-0.00228741	0.13015	0.0232201	12.194	27.3909	24.4867
110	115	165	-0.0226054	0.126675	0.00715587	8.06998	26.9384	25.4057
111	115	165	-0.0251078	0.125921	0.00624226	11.5872	8.19899	204.244
110	116	165	-0.0371209	0.127449	0.0017956	15.5428	25.8804	28.3039
110	117	165	0.033213	0.112692	0.0276861	14.6559	8.51352	209.337
111	116	164	0.0380425	0.109755	0.0161689	21.1287	25.5197	28.2786
111	116	165	-0.0255083	0.112568	0.0366767	18.2749	26.8284	24.3368

Algumas nuvens de pontos são armazenadas usando o modelo de cores YUV, porém no sistema trabalha com o modelo RGB, deve-se realizar a conversão desses modelos como citado na Seção 2.1.5 usando as equações 2.6, 2.7 e 2.8.

Desta maneira a nuvem de pontos é representada dentro do sistema por uma lista contendo duas triplas, uma de *floats* armazenando as coordenadas (x, y, z) da maneira como foram lidas do arquivo e uma tripla de inteiros com as informações de cor já transformadas para melhor utilização da biblioteca gráfica OpenGL.

Outra informação armazenada pelo sistema para cada modelo ou frame - no caso de vídeos - são as dimensões do modelo, as medidas do objeto em cada eixo (x, y, z) na forma de

$$dimensao_\alpha = max(nuvem_\alpha) - min(nuvem_\alpha) \tag{4.1}$$

sendo $nuvem_\alpha$ todas as coordenadas da nuvem no eixo α.

Desta maneira, de forma geral e simplificada o algoritmo de aquisição da nuvem de pontos somente percorre o arquivo criando e adicionando pontos ao modelo, da seguinte

maneira

Algoritmo 1: Obtenção de nuvem de pontos

Entrada: nuvem de pontos
para *todos os pontos da nuvem* **faça**
  $x \leftarrow$ coordenada X do ponto;
  $y \leftarrow$ coordenada Y do ponto;
  $z \leftarrow$ coordenada Z do ponto;
  se RGB **então**
    $R \leftarrow$ informação de cor r do ponto;
    $G \leftarrow$ informação de cor g do ponto;
    $B \leftarrow$ informação de cor b do ponto;
  fim
  senão
    $Y \leftarrow$ informação de cor y do ponto;
    $U \leftarrow$ informação de cor u do ponto;
    $V \leftarrow$ informação de cor v do ponto;
    $R \leftarrow$ abs((1.164 * (y - 16)) + (1.596 * (v - 128)));
    $G \leftarrow$ abs((1.164 * (y - 16)) - (0.813 * (v - 128)) - (0.391 * (u - 128)));
    $B \leftarrow$ abs((1.164 * (y - 16)) + (2.018 * (u - 128)));
  fim
  voxel $\leftarrow$ novo voxel(x, y, z, R, G, B);
  adiciona voxel na lista;
fim
Saída: lista de voxel

O algoritmo apresentado acima é uma simplificação e possui uma variação para cada formato de nuvem a ser obtido. A interação do *loop* principal um ponto da nuvem é obtido por uma função, esta é uma abstração para a maneira de obter um ponto da nuvem, os arquivos das nuvens DB-1, por exemplo, estão armazenadas em arquivo texto ASCII com um ponto em cada linha e os atributos separados pelo caracter espaço, esta função então trabalha separando os atributos da linha referente ao ponto em questão e a retornando em forma de vetor. Para o acesso aos atributos do ponto como x, y ou z basta acessar o vetor retornado pela função de obtenção de ponto na posição conhecida de cada atributo, por exemplo sabe-se que o x é o primeiro valor da linha na representação DB-1, então basta acessar o vetor retornado na posição 0, isto vale para todas as outras informações espaciais e de cor do ponto.

Para as nuvens do formato PLY usamos a biblioteca RPly [27], a estrutura do algoritmo permanece a mesma, porém agora é a biblioteca que se responsabiliza pelo trabalho de pegar um ponto dentro da nuvem e retornar seus atributos.

No final do algoritmo é executado uma função para adicionar um voxel na lista, esta recebe um Voxel como entrada e o adiciona no *array* de pontos do modelo e de maneira iterativa também preenche as informações de dimensão do modelo, quando a cada ponto adicionado na lista verifica se a coordenada ponto é valor máximo ou mínimo para os eixos e no caso positivo altera o resultado da dimensão seguindo a fórmula de cálculo de dimensões do modelo apresentado anteriormente (Fórmula 4.1).

O algoritmo de obtenção das nuvens realizados pelas câmeras estereoscópicas segue a mesma estrutura, a próxima Seção se propõe a mostrar os passos para aquisições de

nuvens de pontos bem como explicitar as pequenas diferenças entre os algoritmos.

4.4.2 Câmeras Estereoscópicas

Outro requisito do sistema é a captação de nuvem de pontos com o uso de câmeras estereoscópicas, o módulo de Câmera é responsável por todas as tarefas relacionados desde o gerenciamento das câmeras até a obtenção das nuvens seguindo o mesmo modelo apresentado anteriormente.

Para a criação da nuvem de pontos é realizado a captura de imagens pelo sistema de câmeras *Bumblebee® XB3*, é utilizado uma técnica semelhante a de 2D + Profundidade apresentado anteriormente onde o processamento estéreo realizado em três etapas: estabelecer correspondência entre características das imagens, calcular o deslocamento relativo entre essas características e por fim determinar suas localizações em relação a câmera.

Considere a Figura 4.10 que apresenta um par de imagens com deslocamento horizontal obtidos por uma câmera estereoscópica.

Figura 4.10: Fotografia estereográfica da estátua em frente ao Instituto de Arte de Chicago.

Usando uma régua, se medir a distância horizontal entre a borda esquerda das imagens e o pontos, perceberá que a distância na imagem à esquerda é maior que a distância da imagem direita. Por a bandeira dos Estados Unidos está mais distante da borda esquerda na imagem à direita se comparado a imagem da esquerda.

A disparidade de um ponto então é definida como a diferença entre as coordenadas desse ponto na imagem à esquerda e à direita, obtendo assim a fórmula

$$D(P) = x(P_{esquerda}) - x(P_{direita}) \quad (4.2)$$

onde P é um ponto na imagem. Vale notar que é estamos calculando a distância horizontal entre os pontos, seguindo o deslocamento horizontal entre as câmeras, por isso só usamos o valor de x para cada coordenada.

Quando aplicamos a Equação 4.2 a Figura 4.10 temos $D(A) = x(A_{esquerda}) - x(A_{direita})$ e $D(A) = x(A_{esquerda}) - x(A_{direita})$, se calcularmos $D(A)$ e $D(B)$ podemos notar que

$D(B) > D(A)$, o que indica que o ponto B está mais próximo da câmera em relação ao ponto A.

A biblioteca Triclops estabelece a correspondência entre as imagens utilizando o método de Soma de Diferenças Absolutas, que se propõe a realizar o seguinte algoritmo:

Algoritmo 2: Método de Soma de Diferenças Absolutas

```
for cada pixel na imagem do
    selecione uma vizinhança de um determinado tamanho;
    compare essa vizinhança à uma série de vizinhanças da outra imagem;
    escolha a melhor correspondência;
end
```

A comparação dos vizinhos é feito usando a seguinte fórmula

$$\min_{d=d_{min}}^{d_{max}} \sum_{i=\frac{m}{2}}^{\frac{m}{2}} \sum_{j=\frac{m}{2}}^{\frac{m}{2}} \left| I_{direita}[x+i][x+j] - I_{esquerda}[x+i+d][y+j] \right| \tag{4.3}$$

onde d_{min} e d_{max} da disparidade, m é o tamanho da vizinhança e $I_{direita}$ e $I_{esquerda}$ são as imagens da direita e esquerda.

A saída do algoritmo é um mapa de disparidade, que é uma imagem indicando para cada pixel a diferença de localização deste na imagem direita em relação à imagem esquerda [33]. Com o mapa de disparidade é possível determinar a distância entre as câmeras e os pontos da imagem retificada, gerando um mapa de profundidade. A biblioteca Triclops fornece uma função que converte mapas de profundidade em coordenadas espaciais (x, y, z) para cada pixel, associamos esta coordenada com a cor RGB deste pixel e obtemos uma nuvem de pontos.

4.4.3 Visualização de Nuvem de Pontos

Agora que sabemos como o sistema obtém e armazena as nuvens de pontos podemos partir para o visualização das nuvens. Vale ressaltar que este trabalha não implementa a reconstrução de superfícies a partir de nuvens de pontos, somente sua exibição. Na Seção 4.3.1 explicitamos algumas das funções do OpenGL 3.1 com *pipeline* fixo que mais utilizamos no desenvolvimento deste trabalho, esta seção busca, não só, demonstrar a implementação da visualização de objetos voltada para esta abordagem, mas também explicitar de maneira geral cada passo do desenvolvimento com finalidade de que o mesmo possa ser reproduzido utilizando OpenGL com *shaders* ou até mesmo outra biblioteca gráfica.

Para visualizarmos uma nuvem de pontos com três dimensões em um plano bidimensional como o monitor devemos realizar esta transformação do plano 3D em 2D facilmente obtida através de projeções com o uso da biblioteca gráfica OpenGL. Existem diversas projeções geométricas, neste trabalho usaremos duas, a projeção paralela ortogonal onde as linhas de projeção são paralelas entre si e perpendiculares ao plano de projeção, e a projeção perspectiva que é uma transformação dentro do espaço tridimensional e suas projeções representam a cena vista de um ponto de observação a uma distância finita [13] como pode ser observado na Figura 4.11.

Em ambas projeções devemos definir as coordenadas de corte do plano de projeção (a superfície onde será projetado o objeto), como vimos anteriormente, as nuvens de pontos

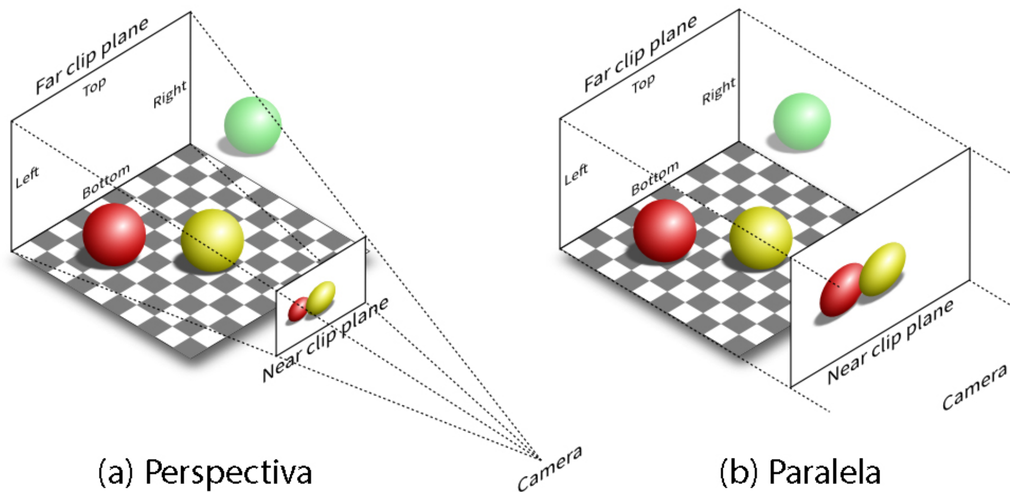


Figura 4.11: (a) Projeção perspectiva e (b) Projeção paralela ortogonal.

não possuem uniformidade na variação dos valores de coordenadas espaciais (x, y, z), ao se usar coordenadas de corte menor que as dimensões do objeto o mesmo será projetado fora da superfície de visualização, este problema também pode variar para um objeto minúsculo em uma superfície muito grande ao se escolher coordenadas muito maiores que o necessário. Para resolvermos este problema guardamos as informações de dimensão do objeto com a finalidade de definir quais parâmetros corretos para a criação plano de projeção, foram realizados testes usando as dimensões do objeto diretamente aplicadas aos parâmetros horizontais e verticais da projeção e também usando razões desses valores, por fim os valores usados como parâmetros para as nuvens usadas para teste foram encontrados através da equação

$$k^{\lceil log_k(max(nuvem_\alpha))\rceil} \tag{4.4}$$

sendo k uma constante qualquer e $nuvem_\alpha$ todas as coordenadas da nuvem no eixo α.

Após criarmos um plano de projeção do tamanho adequado para a visualização do objeto podemos partir para a execução do algoritmo de projeção

Algoritmo 3: Visualização de nuvem de pontos

Entrada: nuvem de pontos
define escala - zoom;
define translação - move;
define rotação - rotate;
define tamanho do ponto - point size;
for *cada ponto na nuvem* **do**
  define cor do ponto;
  define coordenada do ponto;
  exibe o ponto;
end

O algoritmo tem como entrada uma nuvem de pontos de acordo com a modelagem do sistema, em forma de uma lista objetos do tipo *Voxel*. Antes de percorrer essa lista e projetar os pontos deve-se realizar as operações de transformadas de matriz do OpenGL

- escala, translação e rotação - que são responsáveis pela manipulação da visualização do objeto em nossa projeção.

Como definido nos requisitos o sistema deve ser hábil a representar o objeto em várias posições no espaço, o que implica em possibilitar ao usuário do sistema a submissão de manipulações sobre a visualização do objeto através das funções *Zoom*, *Rotate* e *Move*.

A função *Zoom* realiza a mudar as dimensões de escala do objeto multiplicando, para cada um de seus pontos, suas coordenadas por um fator de escala. Para um ponto de coordenada (x, y, z), por exemplo, temos a posição no eixo x como $x' = x \cdot S_x$ com S_x como fator de escala, o mesmo vale para os eixos y e z com os fatores S_y e S_z, temos a equação na forma matricial

$$P' = P + S = [x' \quad y' \quad z'] = [x \quad y \quad z] \cdot \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} = [xS_x \quad yS_y \quad zS_z] \tag{4.5}$$

sendo P' o novo posicionamento do ponto P ao ser multiplicado com o fator de escala S. O comando de alteração escalar do OpenGL tem como parâmetros de entrada os fatores de escala, porém, utilização do fator de escala não-uniforme afetará o objeto visualizado, por isso, usaremos um único fator de escala na forma de $S_x = S_y = S_z$ para evitar esticar ou achatar o objeto em um dos eixos. O valor do *slider* de escala na interface do sistema será usado como entrada para a função de escala e permitirá ao usuário escalonar as dimensões do objeto projetado.

A função *Move* realiza a manipulação de translação na matriz de projeção do objeto, ou seja com essa função é possível mover o objeto sobre o espaço. É possível efetuar a translação de pontos no plano (x, y, z) adicionando quantidades às suas coordenadas [13], por exemplo, cada ponto em (x, y, z) pode ser movido por T_x unidades em relação ao eixo x, seguindo com T_y e T_z para os eixos y e z temos a nova posição do ponto como (x', y', z'), que pode ser escrito como:

$$x' = x + T_x \tag{4.6}$$

$$y' = y + T_y \tag{4.7}$$

$$z' = z + T_z \tag{4.8}$$

ou utilizando notação matricial, a translação pode ser definida como a soma de dois vetores: o vetor de coordenadas iniciais do ponto e o vetor de deslocamento

$$P' = P + T = [x'y'z'] = [xyz] + [T_xT_yT_z] \tag{4.9}$$

sendo P' o novo posicionamento do ponto P ao ser somado com o deslocamento T. Para transladar uma nuvem de pontos devemos alterar todos os pontos pelo mesmo vetor $[T_xT_yT_z]$, o sistema então é responsável por captar o movimento do mouse ao se realizar a função *Move* e aplicará como parâmetros de entrada na função de translação do OpenGL demonstrada na Seção 4.3.1, como a movimentação do mouse ocorre em dois sentidos (horizontal e vertical - 2D) e a translação pode nos tres eixos (x, y, z), foi implementado uma política para a ação de clicar e arrastar do mouse:

- arrastar o mouse **verticalmente** com o botão **esquerdo** pressionado altera o parâmetro T_y;
- arrastar o mouse **verticalmente** com o botão **direito** pressionado altera o parâmetro T_z;
- arrastar o mouse **horizontalmente** com qualquer um dos botões pressionado altera o parâmetro T_x;

Já a função *Rotate* é responsável pela rotação do objeto em torno de um eixo. É possível efetuar a rotação de pontos no plano (x, y, z) definindo um ângulo de rotação para cada eixo, sendo a rotação de um ponto no espaço tridimensional obtida pela multiplicação dos ângulos de rotação em torno dos eixos ao ponto [13]. Para a rotação ser feita em torno de um dos eixos principais, deve-se definir x, y e z apropriadamente como os vetores unitários nas direções destes eixos. Para nosso caso faremos as rotações sempre em torno dos eixos principais usando um comando para cada eixo, definindo os somente os ângulos para o plano x, y e z, estes são obtidos através do clicar e arrastar do mouse ou da alteração dos slides da função *Rotate*. A política de rotação em 3 dimensões segue o definido anteriormente para a movimentação:

- arrastar o mouse **verticalmente** com o botão **esquerdo** pressionado altera o ângulo de rotação do plano y;
- arrastar o mouse **verticalmente** com o botão **direito** pressionado altera o ângulo de rotação do plano z;
- arrastar o mouse **horizontalmente** com qualquer um dos botões pressionado altera o ângulo de rotação do plano x;

Depois de manipular a visualização do objeto podemos passar para a exibição do objeto em si, devemos listar percorrer a lista de pontos e exibir os pontos em suas coordenadas espaciais com as suas determinadas cores. Usando a primitiva `GL_POINTS` dizemos ao OpenGL que a desenharemos pontos em cada coordenada passada, como foi explicado na Seção 4.3.1, basta então percorrer a lista de pontos usando algum laço de repetição e a cada ponto definir um vértice passando as coordenadas espaciais do ponto e definir sua respectiva cor.

Podemos representar mais de 16 milhões de cores com 3 *bytes*, por isso, nos comandos responsáveis por definir a cor do ponto, os parâmetros de entrada são uma tripa de *bytes* (RGB) definindo as cores do ponto. O OpenGL nos permite usar inteiros, *float* ou *double* como entrada para função de cor, porém, foi escolhido o uso *bytes* para diminuir o consumo de memória, visto que, em c++ por exemplo, o um inteiro é representado por 4 *bytes*, um *float* também é representado por 4 *bytes* e um *double*, este valor se torna significativo ao se notar que são necessários representações para de três valores (R,G,B) para cada ponto de uma nuvem.

Já no comando para setar os vértices - pontos - usamos as coordenadas espaciais como *floats*, visto que a maioria das nuvens são representadas coordenadas em pontos flutuantes, também porque não há diferença - em aspecto de memória - com inteiros. A visualização

de pontos permite a configuração do tamanho do ponto através função *Point Size* que recebe como parâmetro o tamanho do ponto informado através do *slider* da função.

O algoritmo 3 de visualização da nuvem de ponto roda toda vez que os parâmetros de manipulação são alterados, ou seja, toda vez que o usuário chamar alguma função com tais finalidades.

Tendo explicitado o desenvolvimento da visualização de nuvem de pontos temos um *pipeline* básico da aplicação apresentado na Figura 4.12.

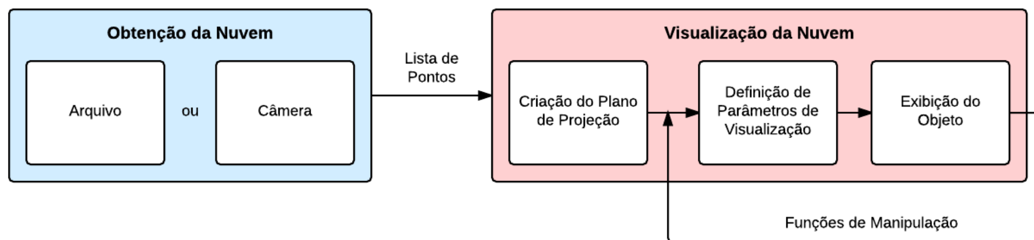


Figura 4.12: *Pipeline* da aplicação.

Capítulo 5

Resultados

De maneira geral, o trabalho alcançou seu objetivo principal, propiciar uma ferramenta de visualização de nuvens de pontos com uma interface gráfica capaz de fornecer ao usuário a interação sobre a exibição. Este Capítulo apresenta algumas visualizações de nuvens de pontos com o uso do sistema desenvolvido neste trabalho. São apresentadas visualizações de nuvens de pontos de diferentes fontes e com as mais variadas superfícies.

Este trabalho teve como motivação inicial a necessidade de uma ferramenta para visualização de nuvens de pontos no formato DB-1, a Figura 5.1 apresenta os resultados obtidos com duas nuvens desse formato. Estas nuvens foram captadas no trabalho realizado por Queiroz e Chou e tem **(a)** denominada como *"Man"* contendo 178.386 de pontos, enquanto a nuvem **(b)** é denominada *"Ricardo"* e contém 207.007 de pontos.

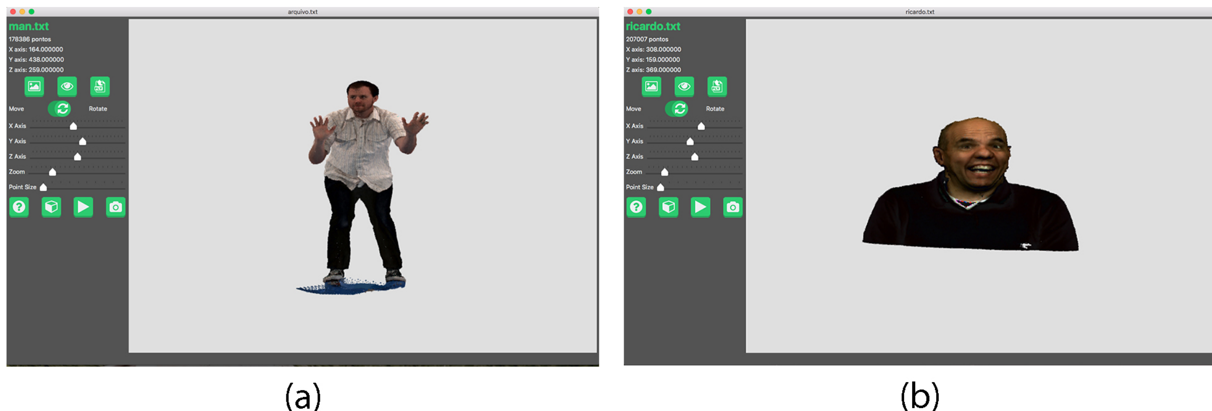


Figura 5.1: Visualização de nuvens no formato DB-1, (a) *"Man"* e (b) *"Ricardo"*.

No Capítulo anterior explanamos sobre o formato das nuvens DB-1, apesar de conter uma estrutura básica as nuvens foram disponibilizadas em arquivos Matlab e com modelo de cores YUV, com *script* simples estas nuvens foram transcritas para um arquivo de texto ASCII contendo a lista pontos, como foi observado anteriormente, algumas ferramentas de processamento de malhas conseguiram abrir o arquivo texto pedindo para que o usuário indica-se a atributos de cada ponto para que fosse definido como a ferramenta deveria ler a estrutura de dados salva no arquivo, porém esta não interpretavam e nem permitiam realizar a conversão do modelo de cores YUV. A ferramenta desenvolvida neste trabalho obteve êxito em realizar a conversão do modelo de cores YUV para RGB.

Um dos requisitos da ferramenta era realizar a exportação de nuvem de pontos no formato PLY, o qual é um dos mais utilizados para guardar nuvens, esta se mostra necessário para salvar nuvens capturadas por meio das câmeras estereoscópicas e também para realizar a transcrição das nuvens do formato DB-1 para um formato mais universal.

Vale ressaltar novamente que a ferramenta desenvolvida neste trabalho não realiza a reconstrução de superfície, porém ao se observar a visualização da nuvens anteriores pode falsamente passar a ideia da reconstrução de superfícies, isto é ocasionado devido a densidade alta da nuvem e da configuração do tamanho da projeção adotado. A Seção 4.4.3 que explica o desenvolvimento da visualização de nuvem de pontos, inicia explicitando a importância de se criar um plano de projeção com tamanho correta para a exibição em OpenGL, foi usado a equação

$$k^{\lceil log_k(max(nuvem_\alpha))\rceil} \tag{5.1}$$

onde $nuvem_\alpha$ todas as coordenadas da nuvem no eixo α e com $k = 9$, para obter a proporção razoável entre o tamanho objeto representado pela nuvem de pontos e o tamanho do plano de projeção. A Figura 5.2 apresenta exemplos de planos de projeções criados com tamanhos equivocados, como por exemplo usando a largura do objeto para definir os planos de corte horizontal do plano de projeção **(a)**, ou usando valores consideravelmente maiores que as dimensões do objeto **(b)**, neste exemplo usou-se a Equação 5.1 com $k = 20$.

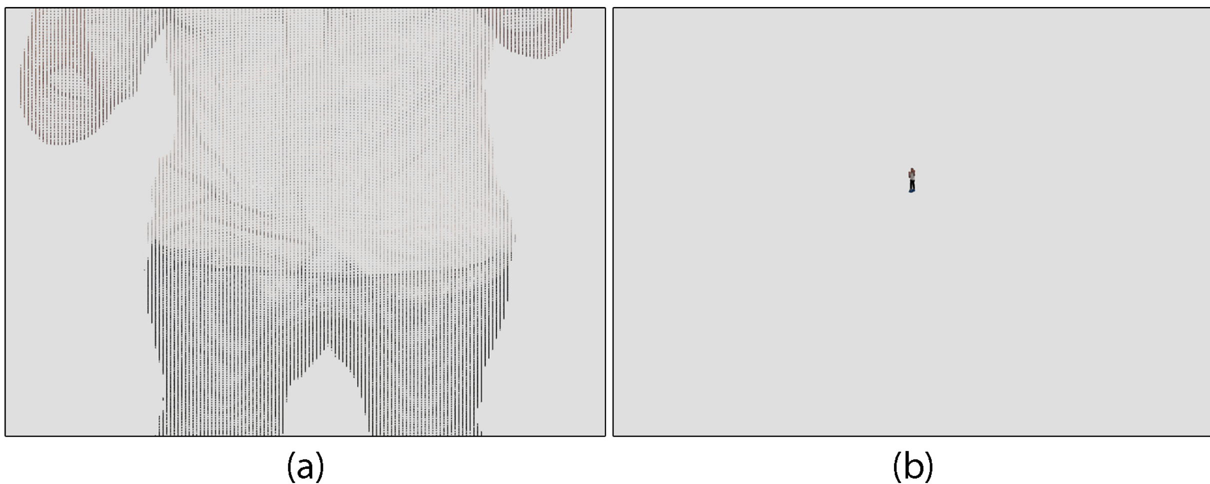


Figura 5.2: Planos de projeções com dimensões erradas.

Ao se usar a função *Zoom*, responsável pela de manipulação visualização alterando a escala de exibição o efeito citado anteriormente se perde, mesmo com nuvens densas a distância entre os pontos se tornam buracos na visualização do objeto, usando outra manipulação na visualização disponível no sistema, a *Point Size*, pode-se obter novamente este efeito, ao se aumentar o tamanho do pontos, os buracos são "ocupados" e a função acaba agindo assim de maneira semelhante a técnica de *splatting*. Outra maneira de criar o efeito de uma superfície reconstruída é usando a técnica de subdivisão espacial e dividindo o plano de projeção em um grid tridimensional e ao invés de exibir pontos em um plano exibir os espaços do grid que possuírem um ponto. A Figura 5.3 apresenta exemplos desses efeitos onde **(a)** mostra a nuvem *"Man"* exibida após o uso da função *Zoom* e **(a)** exibe o resultado de aumentar o tamanho do ponto usando a função *Point*

Size, da mesma maneira, **(c)** exibe a nuvem *"Ricardo"* após o *Zoom* e **(d)** demonstra o uso de subdivisão espacial para visualização de nuvem de pontos.

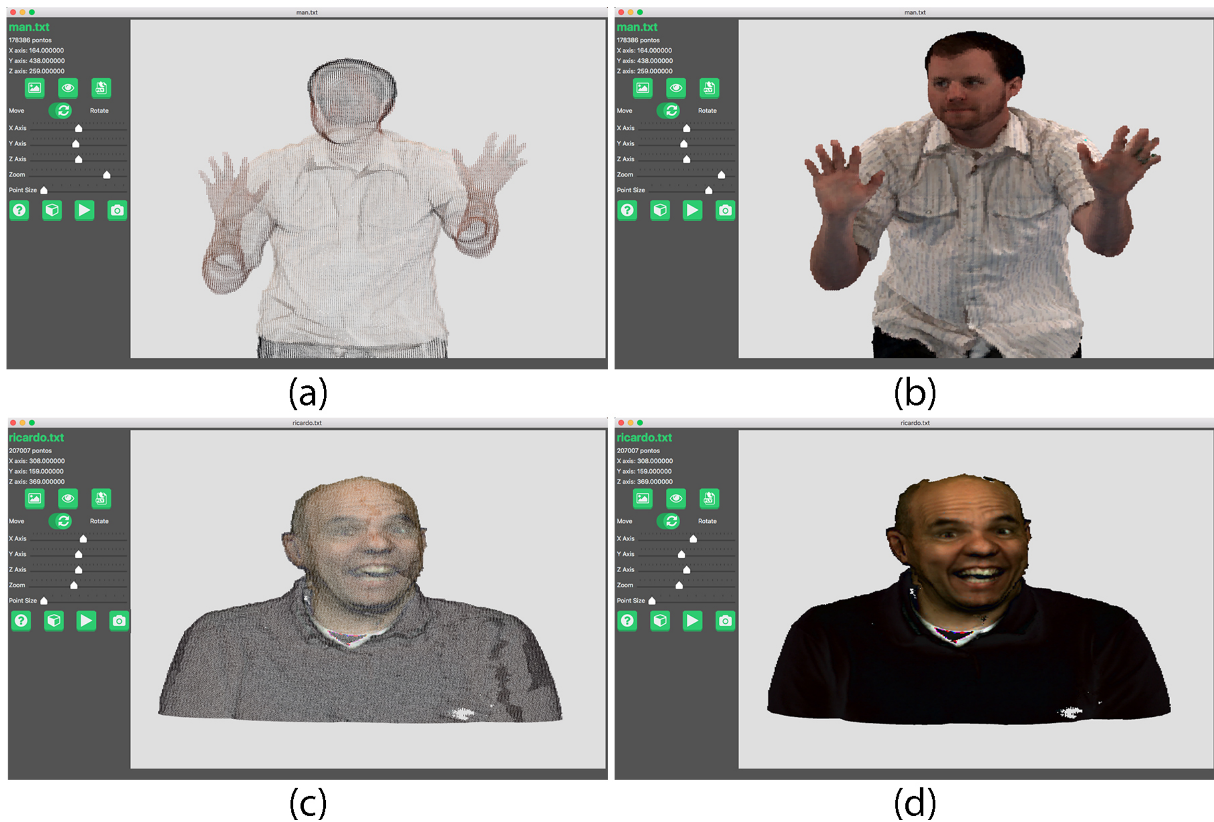


Figura 5.3: Planos de projeções com dimensões erradas.

Também foram utilizados nuvens de pontos vastamente conhecidas como a do Coelho de Stanford [23] apresentado na Figura com duas versões, **(a)** contendo 35.947 pontos e **(b)** com 8.171 pontos, além da Cabeça do Manequim [24] **(c)** com 12.772 pontos, todas disponibilizadas no formato PLY e sem informações de cores.

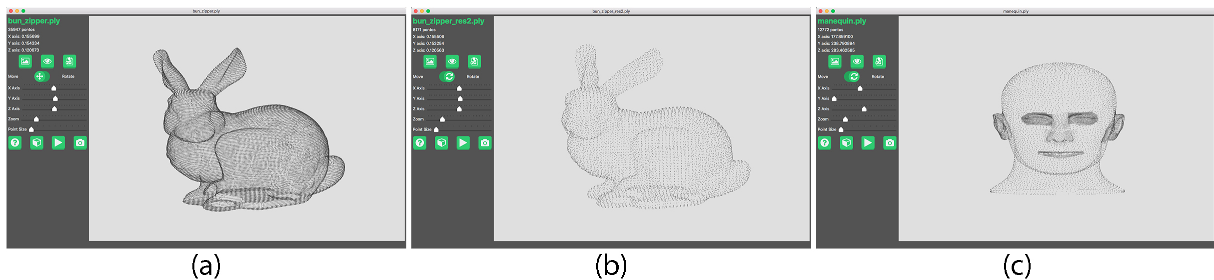


Figura 5.4: Coelho de Stanford (a) 35.947 pontos e (b) 8.171 pontos, e Cabeça do Manequim (c) com 12.772 pontos

Nuvens de pontos coloridas também foram testadas, a Figura 5.5 apresenta alguns modelos encontrados em repositórios onlines: **(a)** *"Body"*[1] contendo 76.270 pontos, **(b)**

[1]http://www.kscan3d.com/gallery/

“Model”[1] com 15.969.639 de pontos e **(c)** *“Skull”*[2] com 133.009 pontos.

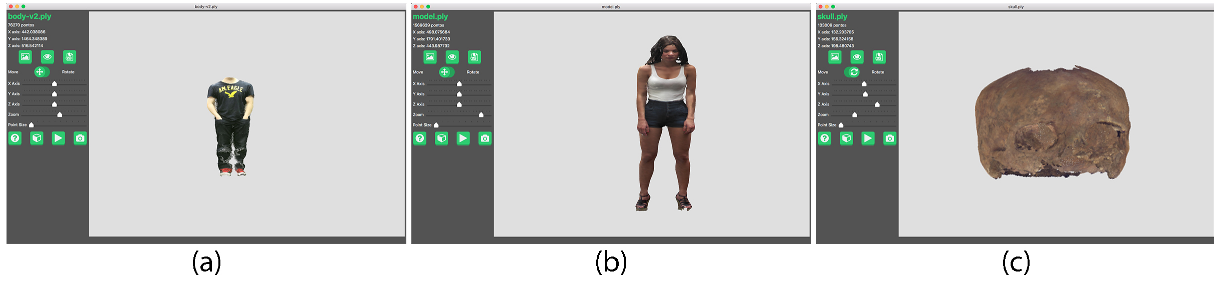


Figura 5.5: Nuvens de pontos coloridas (a) *“Body”*, (b) *“Model”*, e (c) *“Skull”*.

[2]https://people.sc.fsu.edu/ jburkardt/data/ply/ply.html

Capítulo 6

Conclusões

Este Capítulo conclui este trabalho destacando suas principais limitações e listando possíveis trabalhos futuros.

Para cumprir os objetivos firmados a este trabalho, realizou-se uma revisão teórica sobre os fundamentos empregados na renderização baseada em pontos. Considerando os requisitos propostos e a interpretação dos resultados apresentados no Capítulo anterior, pode-se afirmar que em linhas gerais, os objetivos foram alcançados.

Os resultados apresentados no Capítulo anterior são da implementação em C++ em um computador com sistema operacional OS X, porém vale ressaltar que o sistema também foi testado em ambientes Linux e teve sua versão inicial desenvolvida em Java, exibindo a portabilidade exigida nos requisitos do sistema.

Este trabalho não se propôs a implementação da reconstrução de superfície a partir de nuvens de pontos, nem a manipulação da nuvem em si (somente da manipulação da visualização desta). Por isso, sugere-se a trabalhos futuros ou outros desdobramentos, o desenvolvimento funcionalidades como:

- Reconstrução de superfícies a partir de nuvens de pontos;
- Estimação de normais, como observado em [34];
- Manipulações de nuvens como filtragens, suavização, segmentações e mapeamento de textura;

Referências

[1] Isabel Harb Manssour and Marcelo Cohen. Introdução à computação gráfica. *RITA*, 13(2):43–68, 2006. vii, 1, 11

[2] José Luiz Soares Luz. Visualização de nuvens de pontos com aproximações quase planares e blending de textura. 2005. vii, 1

[3] Ricardo L de Queiroz and Philip A Chou. Compression of 3d point clouds using a region-adaptive hierarchical transform. *IEEE transactions on image processing: a publication of the IEEE Signal Processing Society*, 2016. vii, 2, 20, 21

[4] Laboratório de imagens, sinais e acústica (lisa). `http://www.lisa.unb.br/index.php?lang=br`, 2016. Acessado: Outubro de 2016. 2, 20, 21

[5] Ogê Marques Filho and Hugo Vieira Neto. *Processamento digital de imagens*. Brasport, 1999. 4, 5

[6] Rafael C Gonzalez and Richard E Woods. Digital image processing second edition. *Beijing: Publishing House of Electronics Industry*, 2002. 5

[7] Hélio Pedrini and William Robson Schwartz. *Análise de imagens digitais: princípios, algoritmos e aplicações*. Thomson Learning, 2008. 6

[8] Claudio Kirner and Romero Tori. Realidade virtual. *Apostila de graduação. Faculdade de informática. Fundação Eurípides de Marília*, 2004. 7, 8, 10

[9] Maria Laura Chavez Cabrera. Conversão de vídeo 2d para 3d em filmagens panorâmicas de futebol. 2014. 9

[10] Dion Boesten and Patrick Vandewalle. Depth estimation for stereo image pairs. 2009. vii, 9

[11] Martti Mäntylä. An introduction to solid modeling. 1988. 10

[12] Alan H Watt and Alan Watt. *3D computer graphics*, volume 2. Addison-Wesley Reading, 2000. vii, 10, 14

[13] Eduardo Azevedo and Aura Conci. *Computação gráfica: teoria e prática*. Elsevier, 2003. vii, 11, 12, 13, 27, 34, 36, 37

[14] Marcos Luis Cassal. *Geração de sombras em objetos modelados por geometria sólida construtiva*. PhD thesis, Universidade Federal do Rio Grande do Sul, 2001. 13

[15] Lars Linsen. *Point cloud representation*. Univ., Fak. für Informatik, Bibliothek, 2001. 14

[16] Marc Levoy and Turner Whitted. *The use of points as a display primitive*. University of North Carolina, Department of Computer Science, 1985. 14

[17] Markus Gross and Hanspeter Pfister. *Point-based graphics*. Morgan Kaufmann, 2007. 15

[18] Heinrich Müller. Surface reconstruction-an introduction. In *Scientific Visualization Conference, 1997*, pages 239–239. IEEE, 1997. 15

[19] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. Meshlab: an open-source mesh processing tool. In *Eurographics Italian Chapter Conference*, volume 2008, pages 129–136, 2008. 17

[20] Visual computing lab of istituto di scienza e tecnologie dell'informazione "a. faedo". `http://vcg.isti.cnr.it/`, 2016. Acessado: Outubro de 2016. 17

[21] Jan Möbius and Leif Kobbelt. Openflipper: An open source geometry processing and rendering framework. In *International Conference on Curves and Surfaces*, pages 488–500. Springer, 2010. 18

[22] Mario Botsch, Stephan Steinberg, Stephan Bischoff, and Leif Kobbelt. Openmesh-a generic and efficient polygon mesh data structure. 2002. 18

[23] The stanford 3d scanning repository. `http://graphics.stanford.edu/data/3Dscanrep/`, 2015. Acessado: Novembro de 2015. vii, 20, 41

[24] Hugues Hoppe. *Surface Reconstruction from Unorganized Points*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1994. 20, 41

[25] Matlab - mathworks. `http://www.mathworks.com/products/matlab/index.html`, 2016. Acessado: Julho de 2016. 21

[26] Point grey research, inc. `https://www.ptgrey.com/`, 2016. Acessado: Julho de 2016. vii, 21, 22

[27] Rply: Ansi c library for ply file format input and output. `http://w3.impa.br/~diego/software/rply/`, 2016. Acessado: Julho de 2016. 22, 28, 32

[28] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1995. 24

[29] Qt creator 5.7. `http://doc.qt.io/`, 2016. Acessado: Julho de 2016. 24

[30] Opengl. `https://www.opengl.org/`, 2016. Acessado: Outubro de 2016. 24

[31] Dave Shreiner. Opengl programming guide seventh edition the official guide to learning opengl versions 3.0 and 3.1, 2010. 24

[32] Richard S Wright and Michael Sweet. *OpenGL SuperBible with Cdrom*. Sams, 1999. 25, 27

[33] Marcelo Gattass. *Mapas de Disparidade utilizando Cortes de Grafo e Multi-Resolução*. PhD thesis, PUC-Rio, 2010. 34

[34] Niloy J Mitra and An Nguyen. Estimating surface normals in noisy point cloud data. In *Proceedings of the nineteenth annual symposium on Computational geometry*, pages 322–328. ACM, 2003. 43