ALEXANDRE VRUBEL

# PIPELINE PARA RECONSTRUÇÃO DIGITAL DE OBJETOS COM SCANNERS 3D DE TRIANGULAÇÃO A LASER: APLICAÇÃO NA PRESERVAÇÃO DIGITAL DE ACERVOS NATURAIS E CULTURAIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Olga Regina Pereira Bellon
Orientador: Prof. Dr. Luciano Silva

CURITIBA

2008

ALEXANDRE VRUBEL

# PIPELINE PARA RECONSTRUÇÃO DIGITAL DE OBJETOS COM SCANNERS 3D DE TRIANGULAÇÃO A LASER: APLICAÇÃO NA PRESERVAÇÃO DIGITAL DE ACERVOS NATURAIS E CULTURAIS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientadora: Profa. Dra. Olga Regina Pereira Bellon
Orientador: Prof. Dr. Luciano Silva

CURITIBA

2008

# AGRADECIMENTOS

Aos professores Olga Regina Pereira Bellon e Luciano Silva, pela inestimável orientação, amizade e grande oportunidade de crescimento intelectual.

Ao CNPq, FINEP e CAPES pelo apoio à pesquisa no Brasil.

Ao pessoal da Fundação Cultural de Curitiba (FCC), especialmente Rodrigo Marques e Jayne Sunyé pelo apoio, e à equipe do Museu Metropolitano de Arte de Curitiba (MUMA), pelo acesso a obras de artes reais para o desenvolvimento e teste de nossa pesquisa.

Aos professores Euclides Fontoura da Silva Junior e Fernando Antônio Sedor, do Museu de História Natural da UFPR, que permitiram a aquisição de dados 3D de fósseis raros.

À administração superior da UFPR, em especial à professora Vera Belo, pelo acesso a obras de arte presentes no gabinete do Reitor.

Aos professores Lindsley Daibert e Luiz Souza, pela oportunidade de avaliar e capturar dados de uma réplica de 30 anos do profeta Habacuc, obra do Mestre Aleijadinho, do acervo do Museu de Belas Artes da UFMG.

À professora Silvia Helena Soares Schwab pelo apoio no contato com pesquisadores do Museu de La Plata, na Argentina, e pelo uso da réplica do Alamito.

À professora Luciane Marinoni pelo acesso a insetos de uma coleção do Setor de Ciências Biológicas.

E às inúmeras pessoas que contribuíram com idéias, discussões e sugestões para a elaboração deste trabalho.

# CONTEÚDO

LISTA DE FIGURAS v

LISTA DE TABELAS ix

LISTA DE ALGORITMOS x

RESUMO xi

ABSTRACT xii

1 INTRODUÇÃO 1

2 VISÃO GERAL DAS ETAPAS 3

3 AQUISIÇÃO DE DADOS E PRÉ-PROCESSAMENTO 5

3.1 Princípio de Funcionamento . . . . . 5
3.2 Processo de Aquisição . . . . . 8
3.3 Análise Qualitativa dos Dados Capturados . . . . . 10
3.3.1 Ruído . . . . . 11
3.3.2 Dados Falsos . . . . . 12
3.3.3 Superfícies Deformadas . . . . . 13
3.3.4 Superfícies Falsas na Silhueta . . . . . 14
3.3.5 Dados Impossíveis de Serem Capturados . . . . . 16
3.4 Pré-Processamento . . . . . 18
3.4.1 Remoção dos Dados de Fundo . . . . . 19
3.5 Conclusões e Trabalhos Futuros . . . . . 25

4 ALINHAMENTO DE VISTAS 27

4.1 Revisão Bibliográfica . . . . . 28
4.1.1 Alinhamento por Pares de Vistas . . . . . 28
4.1.1.1 Pré-Alinhamento . . . . . 28
4.1.1.2 Alinhamento Propriamente Dito . . . . . 30
4.1.1.3 Alinhamento Global . . . . . 31
4.1.2 Alinhamento Simultâneo de Todas as Vistas . . . . . 32
4.2 Solução Adotada . . . . . 32
4.2.1 Algoritmo ICP . . . . . 33
4.2.1.1 Inicialização . . . . . 33
4.2.1.2 Seleção de Pontos e Seus Correspondentes . . . . . 34

4.2.1.3 Rejeição de Pares . . . . . . . . . . . . . . . . . . . . . . . 35
4.2.1.4 Métrica de Erro . . . . . . . . . . . . . . . . . . . . . . . . 36
4.2.1.5 Minimização do Erro e Atualização da Transformação . . 37
4.2.2 Melhoria da Precisão do ICP . . . . . . . . . . . . . . . . . . . . . 40
4.3 Detalhes de Implementação . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3.1 Arquitetura da Ferramenta IMAGO *ModelTool* . . . . . . . . . . . 42
4.3.2 Detalhes da Etapa de Alinhamento por Pares . . . . . . . . . . . . 43
4.4 Resultados Obtidos e Trabalhos Futuros . . . . . . . . . . . . . . . . . 45

**5 ALINHAMENTO GLOBAL 48**
5.1 Revisão Bibliográfica . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2 Solução Adotada . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.1 Algoritmo de Pulli . . . . . . . . . . . . . . . . . . . . . . . . . . 50
5.2.2 Alinhamento com os Eixos Principais . . . . . . . . . . . . . . . . . 52
5.3 Conclusões e Trabalhos Futuros . . . . . . . . . . . . . . . . . . . . . 56

**6 INTEGRAÇÃO DAS VISTAS 57**
6.1 Revisão Bibliográfica . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.1.1 Métodos de Delaunay . . . . . . . . . . . . . . . . . . . . . . . . . 57
6.1.2 Processamento de Superfícies . . . . . . . . . . . . . . . . . . . . . 58
6.1.3 Superfícies Paramétricas . . . . . . . . . . . . . . . . . . . . . . . 59
6.1.4 Métodos Volumétricos . . . . . . . . . . . . . . . . . . . . . . . . 60
6.2 Solução Adotada . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
6.2.1 Algoritmo de Curless e Levoy (VRIP) . . . . . . . . . . . . . . . . 63
6.2.1.1 Detalhes de Implementação . . . . . . . . . . . . . . . . . 69
6.2.2 Algoritmo de Wheeler . . . . . . . . . . . . . . . . . . . . . . . . 72
6.2.2.1 Detalhes de Implementação . . . . . . . . . . . . . . . . . 83
6.2.3 Algoritmo Híbrido . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.2.3.1 Detalhes de Implementação . . . . . . . . . . . . . . . . . 93
6.3 Resultados Obtidos e Trabalhos Futuros . . . . . . . . . . . . . . . . . 94

**7 PREENCHIMENTO DE BURACOS 97**
7.1 Revisão Bibliográfica . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
7.2 Solução Adotada . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
7.3 Detalhes de Implementação . . . . . . . . . . . . . . . . . . . . . . . . 102
7.4 Resultados Obtidos e Trabalhos Futuros . . . . . . . . . . . . . . . . . 103

**8 GERAÇÃO DA MALHA 3D 108**
8.1 Revisão Bibliográfica . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
8.2 Detalhes de Implementação . . . . . . . . . . . . . . . . . . . . . . . . 109

8.3 Resultados Obtidos e Trabalhos Futuros . . . . . 111

**9 PARAMETRIZAÇÃO DE COORDENADAS DE TEXTURAS 112**

9.1 Revisão Bibliográfica . . . . . 113
9.2 Solução Adotada . . . . . 116
9.2.1 Mapeamento Cilíndrico . . . . . 117
9.2.2 Mapeamento Esférico . . . . . 118
9.2.3 Mapeamento em Forma de Cápsula . . . . . 119
9.2.4 Mapeamento por Atlas de Textura . . . . . 121
9.2.4.1 Segmentação em Regiões . . . . . 121
9.2.4.2 Planificação das Regiões . . . . . 123
9.2.4.3 Montagem do Atlas de Regiões . . . . . 123
9.3 Detalhes de Implementação . . . . . 125
9.4 Resultados Obtidos e Trabalhos Futuros . . . . . 127

**10 CÁLCULO DAS PROPRIEDADES DA SUPERFÍCIE 130**

10.1 Revisão Bibliográfica . . . . . 130
10.1.1 Cálculo da BRDF da Superfície . . . . . 130
10.1.2 Captura da Reflectância . . . . . 132
10.1.3 Correção das Imagens Obtidas . . . . . 133
10.2 Solução Adotada . . . . . 134
10.3 Resultados Obtidos e Trabalhos Futuros . . . . . 135

**11 GERAÇÃO DE TEXTURAS 137**

11.1 Revisão Bibliográfica . . . . . 137
11.2 Solução Adotada . . . . . 139
11.3 Detalhes de Implementação . . . . . 139
11.4 Resultados Obtidos e Trabalhos Futuros . . . . . 141

**12 GERAÇÃO DE NÍVEIS DE DETALHE 144**

12.1 Revisão Bibliográfica . . . . . 144
12.2 Solução Sugerida . . . . . 146

**13 CONCLUSÃO 148**

13.1 Resultados Obtidos . . . . . 148
13.2 Principais Contribuições . . . . . 153
13.3 Trabalhos Futuros . . . . . 154

**BIBLIOGRAFIA 170**

# LISTA DE FIGURAS


3.1 Esquema de funcionamento de *scanner* 3D de triangulação *laser*. . . . . . . 5
3.2 *Hardware* e *software* de aquisição . . . . . . . . . . . . . . . . . . . . . . . 6
3.3 Organização dos dados retornados pelos *scanners* de triangulação *laser* . . 7
3.4 Tipos de distorção apresentados por lentes em fotografia . . . . . . . . . . 7
3.5 Erros de cor nas imagens do *scanner* 3D *Vivid 910* . . . . . . . . . . . . . 8
3.6 Captura com *Dynamic Range Expansion* do *Vivid 910* da *Minolta* . . . . . 9
3.7 Como o foco do *laser* interfere na captura de dados . . . . . . . . . . . . . 10
3.8 Ruído presente em vistas de objetos . . . . . . . . . . . . . . . . . . . . . . 11
3.9 Dados falsos retornados pelo *scanner* 3D . . . . . . . . . . . . . . . . . . . 12
3.10 *Space Carving* utilizado para a eliminação de dados falsos . . . . . . . . . . 13
3.11 Superfícies deformadas em vários modelos . . . . . . . . . . . . . . . . . . 14
3.12 Exemplo de conversão de nuvem de pontos 3D para malha 3D . . . . . . . 15
3.13 Eliminação de faces falsas na silhueta . . . . . . . . . . . . . . . . . . . . . 15
3.14 Superfícies incorretas na silhueta de vistas . . . . . . . . . . . . . . . . . . 16
3.15 "Buracos" devidos a regiões inacessíveis ao *scanner* . . . . . . . . . . . . . 17
3.16 Plano de apoio com dificuldades de detecção. . . . . . . . . . . . . . . . . . 20
3.17 Distribuição de normais no espaço . . . . . . . . . . . . . . . . . . . . . . . 21
3.18 Distribuição da nuvem de pontos 3D projetados sobre a normal candidata . 21
3.19 Fluxograma do algoritmo de remoção de planos de apoio. . . . . . . . . . . 23
3.20 Resultados do algoritmo detector de planos de apoio . . . . . . . . . . . . . 24

4.1 Sistema de coordenadas local de cada imagem de profundidade. . . . . . . 27
4.2 *Turntable* utilizada para o pré-alinhamento de vistas . . . . . . . . . . . . 29
4.3 Descarte de pares com pontos na borda de vistas . . . . . . . . . . . . . . 36
4.4 Tipos de métrica de erro . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.5 Ferramenta *ModelTool* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.6 Processo de alinhamento . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

5.1 Relacionamento entre pares de vistas, representado como um grafo. . . . . 48
5.2 Exemplos de falta de alinhamento global . . . . . . . . . . . . . . . . . . . 49
5.3 Pares virtuais conforme Pulli [113] . . . . . . . . . . . . . . . . . . . . . . . 51
5.4 Alinhamento com os eixos principais . . . . . . . . . . . . . . . . . . . . . 55

6.1 Soma de SDFs nas representações volumétricas de vistas . . . . . . . . . . 63
6.2 Fusão da função de distância e de peso em duas dimensões . . . . . . . . . 64
6.3 Defeito gerado próximo a cantos pelo algoritmo de Curless . . . . . . . . . 66
6.4 Representação volumétrica do defeito do algoritmo de Curless . . . . . . . 67

6.5 Efeito dos *outliers* na integração de Curless . . . . . . . . . . . . . . . . . 68
6.6 Perfis de aplicação de peso conforme a distância no algoritmo de Curless . 68
6.7 Forma de cálculo da distância conforme Curless . . . . . . . . . . . . . . . 70
6.8 Estrutura de dados para marcação de *voxels* . . . . . . . . . . . . . . . . . 71
6.9 Comparação entre os algoritmos de Curless e Wheeler . . . . . . . . . . . 75
6.10 Magnitude e sinais de distâncias para bordas de vistas no algoritmo de Wheeler . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.11 Objeto reconstruído pelo algoritmo de Wheeler com vários limiares de consenso . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.12 Cortes da representação volumétrica do algoritmo de Wheeler com vários limiares de consenso . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
6.13 Objeto reconstruído pelo algoritmo de Wheeler com vários limiares de consenso, descartando medidas para bordas de vistas . . . . . . . . . . . . . 78
6.14 Cortes da representação volumétrica do algoritmo de Wheeler com vários limiares de consenso, descartando medidas para bordas de vistas . . . . . . 79
6.15 Objeto reconstruído pelo algoritmo de Wheeler, descartando medidas para bordas de vistas, e utilizando medidas sem consenso . . . . . . . . . . . . 80
6.16 Falha no critério de consenso . . . . . . . . . . . . . . . . . . . . . . . . . 80
6.17 Objeto reconstruído pelo algoritmo de Wheeler, descartando medidas para bordas de vistas, utilizando medidas sem consenso, e com a distância de candidatos limitada . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.18 Resultado do algoritmo de integração de Sagawa [123] . . . . . . . . . . . 82
6.19 Perfil de aplicação de peso conforme a distância no algoritmo Híbrido . . . 89
6.20 Objeto reconstruído pelos 3 algoritmos de integração . . . . . . . . . . . 91
6.21 Detalhes do algoritmo Híbrido . . . . . . . . . . . . . . . . . . . . . . . . 92
6.22 Detalhes dos 3 algoritmos de integração . . . . . . . . . . . . . . . . . . . 94
6.23 Outros detalhes dos 3 algoritmos de integração . . . . . . . . . . . . . . . 95

7.1 Exemplo de buraco do artigo de Davis [38] . . . . . . . . . . . . . . . . . 97
7.2 Exemplos de difusão em 2D para vários tipos de buraco . . . . . . . . . . 99
7.3 Exemplos da difusão em progresso . . . . . . . . . . . . . . . . . . . . . . 100
7.4 Comparação entre o algoritmo de preenchimento de buracos de Sagawa e o de Davis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
7.5 Algoritmo de Davis, executado sobre um modelo integrado por Curless . . 104
7.6 Algoritmo de Davis, executado sobre um modelo integrado pelo nosso algoritmo híbrido . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
7.7 Problema de dados oblíquos em relação às bordas de buracos . . . . . . . . 106
7.8 Buracos preenchidos de um fóssil . . . . . . . . . . . . . . . . . . . . . . . 107

8.1 Os 15 casos do algoritmo *Marching Cubes*, conforme descritos por Lorensen [93]. . . . 108
8.2 Os casos do algoritmo *Marching Cubes*, conforme descritos por Chernyaev [28] 110
8.3 Triângulos muito estreitos (*slivers*) gerados pelo *Marching Cubes* . . . 111

9.1 Exemplo de modelo 3D e sua textura . . . 112
9.2 Formas triviais de parametrização de textura . . . 113
9.3 Convenção do sistema de coordenadas de parametrização UV . . . 117
9.4 Duplicação de vértices na emenda do cilindro . . . 118
9.5 Projeção esférica de um vértice . . . 119
9.6 Segmentação de um modelo em regiões . . . 122
9.7 Exemplo de atlas de textura . . . 125
9.8 Implementação da heurística de montagem do atlas de textura . . . 126
9.9 Modelo de um fóssil, texturizado com diversos métodos de mapeamento . . 128
9.10 Texturas utilizadas na figura 9.9 . . . 129

10.1 Processo de colorização de vértices . . . 136

11.1 Exemplo de textura gerada pelo método de Matsumoto [99] . . . 138
11.2 Exemplos de mapas de textura gerados utilizando parametrização cápsula . 140
11.3 Defeitos de *mip-mapping* em texturas geradas com parametrização por atlas 142

12.1 Operação de *edge collapse* do algoritmo de Hoppe [68] . . . 146
12.2 *Progressive Mesh* aliado a *normal mapping* [129] . . . 147
12.3 *Normal mapping* aplicado a um modelo de baixa resolução [15] . . . 147

13.1 São Francisco reconstruído utilizando o *software* que acompanha o *scanner* Vivid 910 da Konica Minolta . . . 150
13.2 Galo reconstruído utilizando o *software* que acompanha o *scanner* Vivid 910 da Konica Minolta . . . 151
13.3 Pequena estátua em madeira de uma coruja . . . 156
13.4 Réplica da cabeça de uma estátua grega . . . 157
13.5 Reconstrução 3D de um besouro . . . 158
13.6 Estátua em madeira de São Francisco . . . 159
13.7 Réplica do objeto Alamito do Museu de La Plata, na Argentina . . . 160
13.8 Objeto Buddha, do repositório de modelos do laboratório SAMPL da Universidade de Ohio . . . 161
13.9 Estátua em latão de um galo . . . 162
13.10 Estátua criada pelo escultor paranaense Erbo Stenzel . . . 163
13.11 Fóssil da mandíbula de um cavalo nativo do continente americano . . . 164
13.12 Estátua em bronze criada pelo artista Carybé . . . 165

13.13Pequeno vaso . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
13.14Fóssil do crânio de um *protocyon* (ancestral do lobo guará) . . . . . . . . . 167
13.15Estátua em mármore, presente no acervo do gabinete do Reitor da UFPR . 168
13.16Réplica parcial em gesso do profeta Habacuc de Aleijadinho . . . . . . . . 169

# LISTA DE TABELAS

3.1 Parâmetros do algoritmo detector de planos de apoio. . . . . . . . . . . . . 25

4.1 Parâmetros do algoritmo ICP especificado no arquivo *script* de entrada do programa *ModelTool*. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

6.1 Parâmetros do algoritmo de Curless especificado no arquivo *script* de entrada do programa *ModelTool*. . . . . . . . . . . . . . . . . . . . . . . . . 71

6.2 Parâmetros do algoritmo de Wheeler especificado no arquivo *script* de entrada do programa *ModelTool*. . . . . . . . . . . . . . . . . . . . . . . . . 84

6.3 Parâmetros do algoritmo Híbrido de integração de vistas, especificado no arquivo *script* de entrada do programa *ModelTool*. . . . . . . . . . . . . . . 93

7.1 Parâmetros do algoritmo de Davis especificado no arquivo *script* de entrada do programa *ModelTool*. . . . . . . . . . . . . . . . . . . . . . . . . . . 102

13.1 Objetos modelados com o *pipeline* de reconstrução 3D proposto. . . . . . . 149

13.2 Tempos de execução das etapas do *pipeline* de reconstrução 3D proposto. . 152

# LISTA DE ALGORITMOS


4.1 Pseudo-código do algoritmo ICP. . . . . . . . . . . . . . . . . . . . . . . . . 33
5.1 Pseudo-código para o algoritmo de Alinhamento Global de Pulli [113]. . . . 51
6.1 Pseudo-código para o algoritmo de Integração de Vistas de Curless [37]. . . 64
6.2 Pseudo-código para descobrir *voxels* próximos à superfície de uma vista. . . 70
6.3 Pseudo-código para o algoritmo de Integração de Vistas de Wheeler [155]. . 73
6.4 Pseudo-código para a 1ª fase do algoritmo Híbrido de Integração. . . . . . 86
6.5 Pseudo-código para a 2ª fase do algoritmo Híbrido de Integração. . . . . . 87
7.1 Pseudo-código para o algoritmo de Difusão Volumétrica de Davis [38]. . . . 100
9.1 Pseudo-código para o algoritmo de segmentação do modelo em regiões. . . 121
9.2 Pseudo-código para o crescimento de uma região. . . . . . . . . . . . . . . . 122
10.1 Pseudo-código para o algoritmo de colorização dos vértices do modelo. . . . 134

# RESUMO

A reconstrução digital de objetos reais é um problema fascinante, que já ofereceu um grande número de contribuições a várias áreas de aplicação, mas que em algumas situações ainda está longe de ter soluções satisfatórias. Isto é verdade mesmo quando equipamentos sofisticados, tais como scanners de triangulação a laser, são utilizados. Neste trabalho, abordaremos todas as etapas que compõem o complexo pipeline de geração de modelos de objetos reais, desde o processo de aquisição de dados com scanners 3D até a geração do modelo 3D final.

Modelos de objetos reais possuem diversas aplicações práticas, entre elas: preservação e restauração de patrimônios culturais e naturais; arqueologia digital; engenharia reversa de peças e equipamentos; economia de mão-de-obra na geração de conteúdo digital 3D; ensino e entretenimento.

Os diversos algoritmos apresentados são analisados criticamente, de forma que seus méritos e deficiências sejam entendidos no contexto mais amplo do problema como um todo. Além disso, várias melhorias nos algoritmos são apresentadas, bem como considerações de implementação.

Ressaltam-se como contribuições principais deste trabalho a implementação completa do *pipeline* de reconstrução digital; o levantamento dos tipos de defeitos encontrados na prática em imagens de profundidade; a criação de um algoritmo robusto para a detecção e eliminação dos planos de apoio dos objetos capturados; a criação um método mais prático de pré-alinhamento manual de pares de vistas; uma modificação do algoritmo ICP, dividindo-o em duas fases para alcançar maior convergência e precisão; uma análise crítica dos algoritmos de integração volumétrica de Curless e Levoy, e de Wheeler *et al.*, amplamente utilizados na prática; a criação de um novo algoritmo híbrido de integração de vistas, que resolve a maior parte das deficiências dos dois algoritmos anteriores, além de permitir melhores resultados na etapa de preenchimento de buracos; a criação de um algoritmo simples e prático para a criação de atlas de texturas; e um método de gerar as imagens das texturas utilizando placas aceleradoras gráficas.

O método proposto foi aplicado na preservação digital de diversos objetos, em especial algumas obras de arte do Museu Metropolitano de Arte de Curitiba; alguns fósseis do Museu de História Natural da UFPR; exemplares de insetos de uma coleção do setor de Ciências Biológicas da UFPR; algumas obras de arte presentes no gabinete do Reitor da UFPR; e uma réplica parcial de 30 anos da estátua do profeta Habacuc de Aleijadinho, presente no Museu de Belas Artes da UFMG.

# ABSTRACT

The digital reconstruction of real objects is a fascinating problem, which has offered a large number of contributions to several application areas, but that in some situations is still far from satisfactory solutions. This is true even when sophisticated equipment such as laser triangulation 3D scanners are used. In this work, we discuss all the stages of the complex pipeline that generates models of real objects, from the process of acquiring data with 3D scanners to the final generation of a 3D model.

Models of real objects have many practical applications, including: preservation and restoration of cultural and natural heritage; digital archaeology; reverse engineering of parts and equipment; economy of workforce in the generation of 3D digital content; education and entertainment.

The several algorithms used in this work are critically analyzed, so that their merits and weaknesses are understood in the wider context of the problem as a whole. In addition, several improvements to the algorithms are presented, as well as implementation considerations.

The major contributions of this work are: the full implementation of the digital reconstruction pipeline; an analysis of the types of defects found in real range images; the implementation of a robust algorithm for the detection and elimination of the support plane in captured objects; the creation of a more practical method of manual pre-alignment of pairs of views; a modification of the ICP algorithm, dividing it into two phases to achieve greater convergence and accuracy; a critical analysis of volumetric integration algorithms of Curless and Levoy, and Wheeler *et al.*, widely used in practice; the creation of a new hybrid algorithm for view integration, which resolves most of the shortcomings of the two previous methods, besides leading to better results in the hole-filling step; the creation of a simple and practical algorithm to generate a texture atlas; and a method to generate the texture images, using graphics acceleration cards.

The proposed pipeline was applied in the digital reconstruction of several objects, in particular some art works from the Metropolitan Museum of Art in Curitiba; some fossils from the Natural History Museum of UFPR; some insects from a collection from the sector of Biological Sciences of UFPR; some works of art present in the UFPR Headmaster's office; and a 30-years old partial replica of the statue of the prophet Habacuc, made by Aleijadinho, present in the Museum of Fine Arts of UFMG.

# CAPÍTULO 1

# INTRODUÇÃO

A reconstrução digital de objetos reais é um campo da Visão Computacional e da Computação Gráfica cuja pesquisa têm se intensificado nos últimos anos. Vários fatores contribuíram para isto, mas podemos dizer que a evolução tecnológica tanto dos *scanners* 3D quanto dos microcomputadores em geral permitiram novas possibilidades que não eram viáveis há alguns anos atrás.

Do ponto de vista das aplicações, a reconstrução 3D pode ser utilizada em, por exemplo:

- Preservação digital de acervos naturais e culturais [16, 89, 101, 75];
- Arqueologia digital [23, 58];
- Reengenharia de peças;
- Biometria;
- Aplicações médicas, como a geração de próteses personalizadas;
- Geração de conteúdo 3D para aplicações multimídia como filmes ou jogos.

Apesar de toda a pesquisa e das evoluções tecnológicas, as soluções comerciais disponíveis ainda estão muito aquém das expectativas dos usuários. O processo de reconstrução ainda é muito complexo e trabalhoso, e os resultados obtidos em muitos casos não possuem a precisão e qualidade desejadas.

Soluções mais fáceis de usar (como por exemplo, o *Desktop 3D Scanner da NextEngine*[1]) costumam apresentar sérias restrições quanto ao tipo de objetos que conseguem ser capturados, em termos de tamanho, complexidade geométrica e características das superfícies dos objetos.

Por outro lado, soluções mais poderosas (como por exemplo, o *Vivid 910* da *Konica Minolta*[2]) são bem mais caras, e requerem um maior grau de conhecimento e controle por parte do usuário, de forma a superar as dificuldades citadas anteriormente. E mesmo neste caso, algumas vezes a precisão obtida pelo processo de reconstrução acaba não sendo suficiente dependendo do tipo de aplicação desejada.

[1] http://www.nextengine.com
[2] http://www.konicaminolta.com/sensingusa/products/3d/non-contact/vivid910/index.html

Deve-se notar que muitas vezes o problema de precisão insuficiente não é sempre devido ao *hardware* de aquisição; os algoritmos utilizados para o processamento dos dados brutos também inserem problemas e restrições ao processo de reconstrução. Além disso, alguns algoritmos são inerentemente mais suscetíveis a falhas na presença de ruídos nos dados de entrada, ruídos estes que são inevitáveis em qualquer processo físico de aquisição de dados.

Em muitas aplicações a precisão dos resultados é fundamental, como é o caso da preservação de acervos culturais e naturais. Todos estes fatos serviram para motivar o nosso trabalho, que procura abordar o processo completo de reconstrução de objetos reais, através de dados obtidos com *scanners* 3D de triangulação laser. Este é um processo longo e complexo, que envolve diversos algoritmos, estruturas de dados e soluções de compromisso.

A maior parte dos trabalhos na área concentra-se em pontos específicos deste pipeline (sequência de etapas) de geração de modelos, sendo que normalmente as soluções individuais são difíceis de serem integradas, ou utilizam-se de hipóteses simplificadoras que não acontecem na prática. Como resultado temos os problemas e limitações das soluções comerciais, comentados anteriormente.

Neste trabalho, abordaremos os algoritmos mais estabelecidos na literatura, que foram implementados e integrados em nosso pipeline de reconstrução de modelos 3D. Estes trabalhos clássicos (*seminal papers*) também foram utilizados como base para alguns projetos famosos de preservação digital, como o *"The Digital Michelangelo Project"* [89], o *"Pietà Project"* [16] e o *"The Great Buddha Project"* [101, 75].

Utilizamos em nossos experimentos o *scanner* de triangulação laser *Vivid 910* da *Konica Minolta*, para adquirir imagens de profundidade de diversos objetos, bem como dados do repositório do SAMPL (*Signal Analysis and Machine Perception Laboratory*) da Universidade de Ohio.

Abordando o processo completo, poderemos identificar os problemas atuais, de forma que possam ser melhor compreendidos, sugerir algumas melhorias, e direcionar futuros esforços de pesquisa para resolver as lacunas e deficiências encontradas.

# CAPÍTULO 2

# VISÃO GERAL DAS ETAPAS

O processo completo de reconstrução 3D dificilmente é abordado em detalhes em artigos científicos, devido à sua complexidade. Uma visão geral do processo pode ser vista no artigo *"The Digital Michelangelo Project"* [89], no artigo de Bernardini e Rushmeier [15], e no artigo de Ikeuchi *et al.* [75].

De uma forma geral, as etapas da reconstrução 3D são:

- Aquisição de dados e Pré-processamento;
- Alinhamento (registro) de pares de vistas;
- Alinhamento global;
- Integração das vistas alinhadas;
- Preenchimento de buracos;
- Geração da malha 3D;
- Parametrização de coordenadas de texturas;
- Cálculo da cor e outras propriedades da superfície;
- Geração de texturas (cor, *specular maps*, *normal maps*, etc.);
- Geração de vários níveis de detalhe através da redução do número de faces.

Estas etapas podem ser ligeiramente modificadas conforme a abordagem específica sendo utilizada, mas em geral todas estas etapas deveriam estar presentes em um *pipeline* completo de reconstrução digital. A maioria das soluções comerciais (mesmo as mais caras) não costuma tratar das últimas etapas, referentes à texturização e propriedades da superfície, ficando restritos apenas à captura da geometria dos objetos.

Devido à extensão do *pipeline*, nosso trabalho será organizado de uma forma ligeiramente diferente da tradicional. Analisaremos cada etapa individualmente, iniciando com a sua revisão bibliográfica; em seguida, apresentando a solução adotada em detalhes, bem como a justificativa da sua utilização e seu relacionamento com as outras etapas; algumas considerações sobre a implementação, tais como arquitetura e estruturas de dados; e finalmente, resultados obtidos e trabalhos futuros.

Antes de iniciarmos a análise de cada etapa, convém esclarecer nossa filosofia de trabalho e motivação para o mesmo. Nosso principal objetivo foi desenvolver ferramentas e metodologias de trabalho capazes de gerar modelos 3D de objetos reais com a maior fidelidade possível, reduzindo a um mínimo as operações manuais necessárias. Apesar de importante, algumas vezes tivemos que sacrificar desempenho para obter uma maior fidelidade ou uma maior automatização de processo. Outras vezes, soluções de compromisso tiveram que ser adotadas, para que o sistema fosse capaz de rodar dentro de restrições aceitáveis de tempo e espaço.

Nosso desafio inicial foi projetar um sistema que funcionasse. Infelizmente, as soluções comerciais existentes (como a que acompanha o *scanner Vivid 910* da *Konica Minolta*) não atendem esta premissa básica. Em seguida, nos focamos em aprimorar a qualidade e fidelidade dos resultados. Infelizmente, não houve tempo para abordar alguns aspectos, como por exemplo, soluções distribuídas em *clusters* para o *pipeline* proposto. Outras áreas que requerem mais trabalho são as etapas de texturização do *pipeline*, onde apenas implementamos soluções básicas. Estas etapas vêm sendo trabalhadas por outros pesquisadores do grupo IMAGO de pesquisa[1], que têm utilizado nosso trabalho como base.

[1]http://www.imago.ufpr.br

# CAPÍTULO 3

# AQUISIÇÃO DE DADOS E PRÉ-PROCESSAMENTO

A etapa de aquisição de dados consiste em obter várias imagens de profundidade (*range images*) do objeto em questão, de vários pontos de vista diferentes. Além das imagens de profundidade, também são capturadas imagens de intensidade luminosa (fotos digitais), para a posterior extração de texturas.

Em nosso trabalho, apenas consideramos scanners 3D de triangulação laser. Isto se deve ao fato de que eles são considerados um dos meios mais precisos de aquisição de dados 3D. Vários outros métodos de aquisição existem, como sistemas de estéreo-visão ou de medição por tempo de reflexão de laser. O método de triangulação laser possui menor alcance, mas possui precisão na faixa de micrômetros, enquanto os outros métodos possuem precisão na faixa de milímetros [160, 109, 54]. Felizmente, a maior parte de nosso trabalho independe do método de aquisição utilizado.

## 3.1 Princípio de Funcionamento

O funcionamento de um *scanner* de triangulação *laser* (como o *Vivid 910* que utilizamos) pode ser entendido através da figura 3.1.

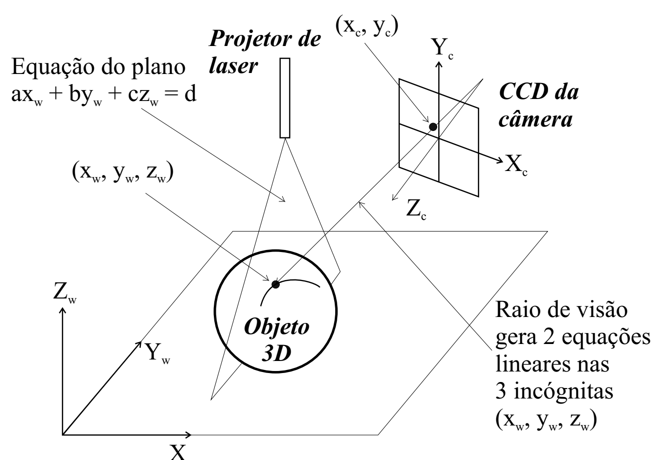
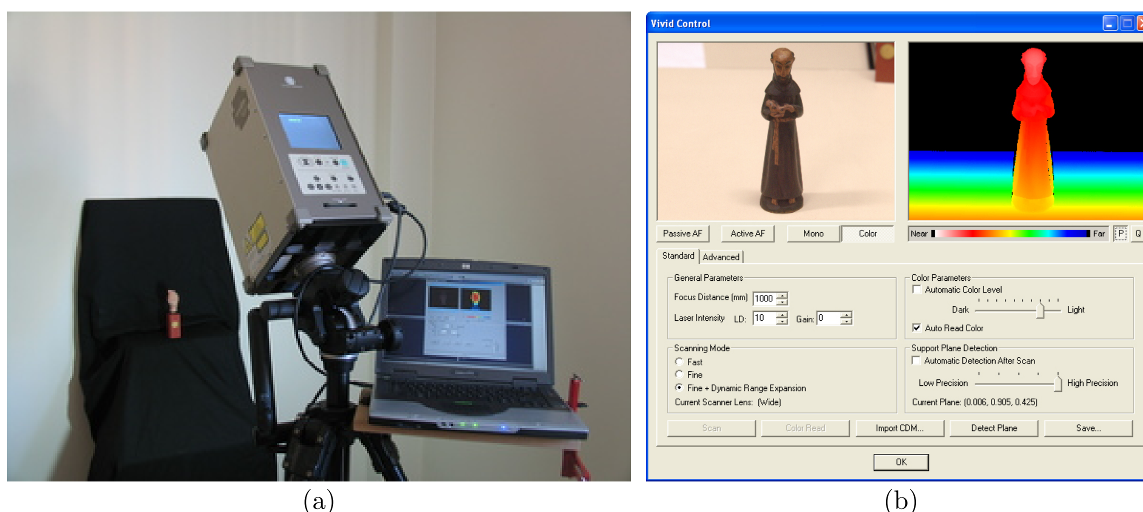



**Figura 3.1:** Esquema de funcionamento de *scanner* 3D de triangulação *laser*.

Uma linha de laser (*laser stripe*) é projetada sobre a superfície do objeto, e a imagem é capturada. Existe um ângulo conhecido entre o emissor de laser e o CCD da câmera, e

através de triangulação, a distância de cada ponto da imagem é calculada. É feita uma varredura com a linha de laser (obtida através da rotação de um espelho), o que acaba capturando uma nuvem de pontos em 3D, que chamamos de imagem de profundidade [54, 36].

Na figura 3.2 podemos ver o *scanner* 3D, e uma tela do programa de aquisição que implementamos para controlá-lo diretamente. Este programa foi batizado de *Vivid Control*. Seu desenvolvimento foi necessário pois o *software* que acompanha o *scanner* era muito pesado, o que tornava o processo de captura muito demorado. Além disso, várias melhorias foram introduzidas, como uma interface simplificada, conversão de formato de dados e a detecção e eliminação do plano de apoio, que será discutida adiante. Para o seu desenvolvimento, utilizou-se o SDK fornecido pelo fabricante [85].




(a) (b)

**Figura 3.2:** *Hardware* e *software* de aquisição. (a) *Scanner* 3D *Vivid 910* da *Konica Minolta*; (b) Tela do programa *VividControl*, desenvolvido para controlar o processo de aquisição do *scanner*.

Podemos observar na figura 3.2(b) tanto a imagem de intensidade luminosa, quanto a imagem de profundidade correspondente. A escala que cores representa a distância, sendo os tons de vermelho os pontos mais próximos, e os tons azulados os mais distantes. Preto significa ausência de dados.

É importante notar que a imagem de profundidade é apenas uma conveniência para facilitar a visualização dos dados capturados. O que o *scanner* realmente retorna é uma matriz bidimensional de pontos 3D, especificados no sistema de coordenadas local da câmera. Tal organização pode ser observada na figura 3.3.

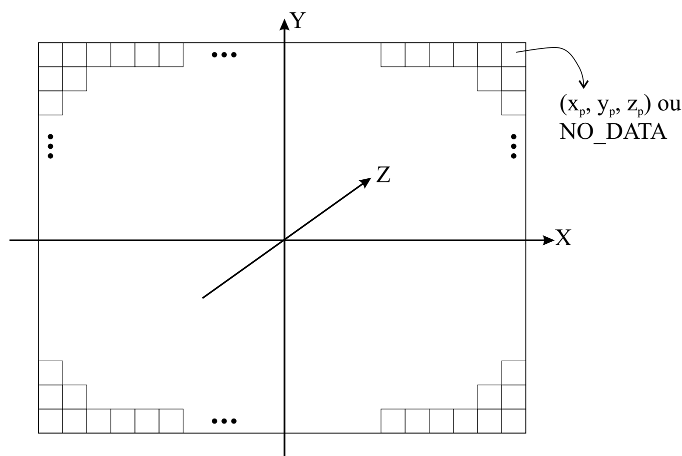



**Figura 3.3:** Organização dos dados retornados pelos *scanners* de triangulação *laser*, também conhecidos como nuvens de pontos. Pontos fora de foco não retornam dados.

É importante perceber que o espaçamento horizontal e vertical dos pontos retornados não é constante, isto é, a nuvem de pontos não é formada simplesmente pela interseção de um *grid* uniforme com o objeto. Isto ocorre porque o *scanner* elimina as distorções radiais da lente da câmera, corrigindo os dados 3D retornados. Em outras palavras, tanto a imagem colorida quanto a imagem de profundidade são "distorcidas", e apenas os dados 3D retornados são precisos. Este efeito de distorção das imagens é conhecido em fotografia como "efeito barril", e pode ser melhor entendido através da figura 3.4.

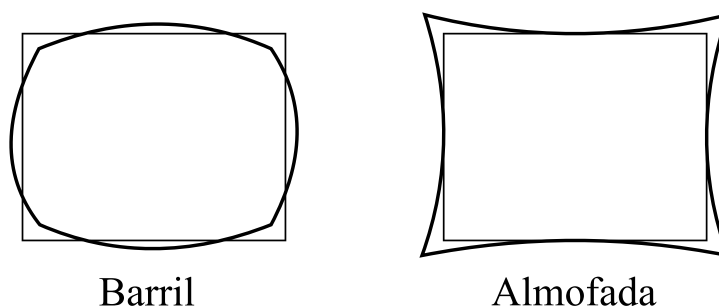



**Figura 3.4:** Tipos de distorção apresentados por lentes em fotografia. No caso do *scanner Vivid 910* da *Minolta*, a distorção apresentada é a tipo barril.

Tendo em vista este fato das distorções presentes em imagens de profundidade, podemos claramente perceber que quaisquer algoritmos que trabalhem diretamente nas imagens 2D de profundidade estão sujeitos a erros, pois estão ignorando a distorção presente nestas imagens, e tratando-as como distribuições uniformes de pontos.

Além da nuvem de pontos 3D, obtemos do *scanner* parâmetros como a posição da câmera (*viewpoint*), o foco da lente, resoluções das imagens e as direções dos eixos do sistema de coordenadas local da câmera. Estes parâmetros nos permitem reconstruir as linhas de visão do *scanner* a cada ponto 3D capturado [85].

Opcionalmente, a imagem de intensidade luminosa correspondente aos pontos capturados também pode ser obtida. Esta imagem pode ser utilizada posteriormente para a reconstrução de texturas do modelo 3D, e a grande vantagem é que ela é obtida do mesmo ponto de vista que a imagem de profundidade; isto implica que nenhum processo de calibração de câmera é necessário para a utilização desta imagem.

Infelizmente, as imagens de intensidade capturadas pelos scanners 3D costumam possuir resolução muito baixa (no caso do *Vivid 910*, as imagens capturadas têm uma resolução de 640x480 *pixels*). Além da baixa resolução, as imagens costumam apresentar cores incorretas. Um exemplo disso pode ser visto na figura 3.5, onde três vistas do mesmo objeto sob as mesmas condições de iluminação geraram cores totalmente diferentes (e erradas):

**Figura 3.5:** Erros de cor nas imagens do *scanner* 3D *Vivid 910*. A imagem central é a que possui as cores mais próximas das reais.

Este erro de coloração normalmente acontece quando se utiliza o método automático de controle de nível de cor do *scanner*, que é a opção padrão; quando este controle é feito manualmente, os erros acontecem, mas são bem menos perceptíveis.

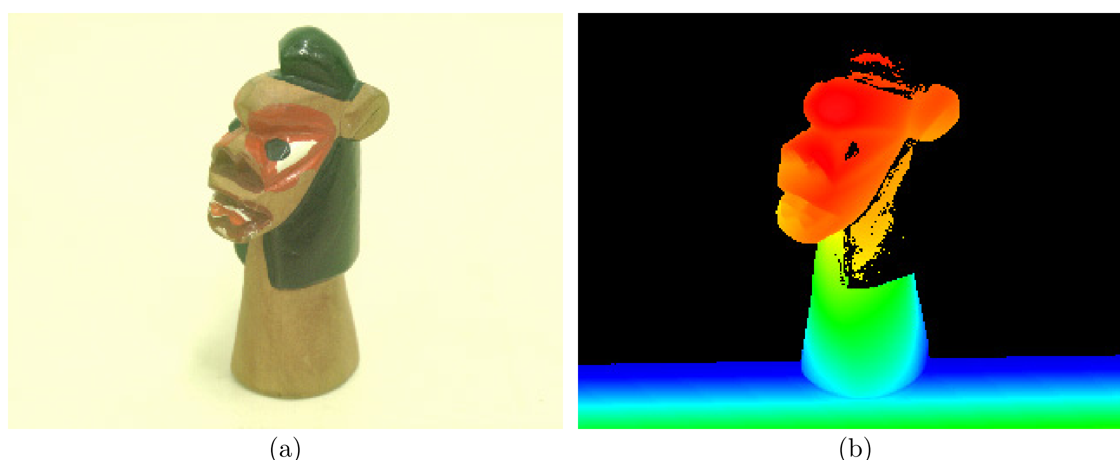
## 3.2 Processo de Aquisição

Conforme o que foi apresentado na seção anterior, começa a ficar mais claro o porquê de serem necessárias várias vistas para reconstruir um objeto em 3D. A informação de um único ponto de vista não contém todos os dados 3D do objeto; na figura 3.2(b), capturamos apenas a parte da frente da estátua. Para capturar as laterais, a parte de trás e a parte inferior, temos que movimentar o objeto, e efetuar capturas adicionais. Além disso, podem ocorrer oclusões, ou seja, uma parte do objeto esconder outra. Isto é muito comum em concavidades, e acaba requerendo vistas adicionais [89, 15, 75].

Outro fator que complica o processo de aquisição é o ajuste dos parâmetros do *scanner*. Um parâmetro que deve ser controlado é a potência do *laser*: quanto mais escuro o objeto, maior deve ser a potência para que ele consiga ser capturado. Por outro lado, quanto maior

a potência, maior o grau de ruído nos dados resultantes. Desta forma, deve-se buscar a menor potência possível capaz de capturar todo o objeto. Isto é dificultado quando o objeto possui diversas cores.

O *scanner Vivid 910* da *Minolta* possui um método de captura onde 3 varreduras são feitas, cada uma delas com uma potência de *laser* diferente, e os dados são combinados e retornados pelo *scanner* [85]. Apesar de melhorar muito a captura de dados de objetos multicoloridos, ainda assim algumas vezes este procedimento não consegue capturar toda a superfície do objeto. Isto pode ser observado na figura 3.6, onde as partes pretas do objeto não foram capturadas.

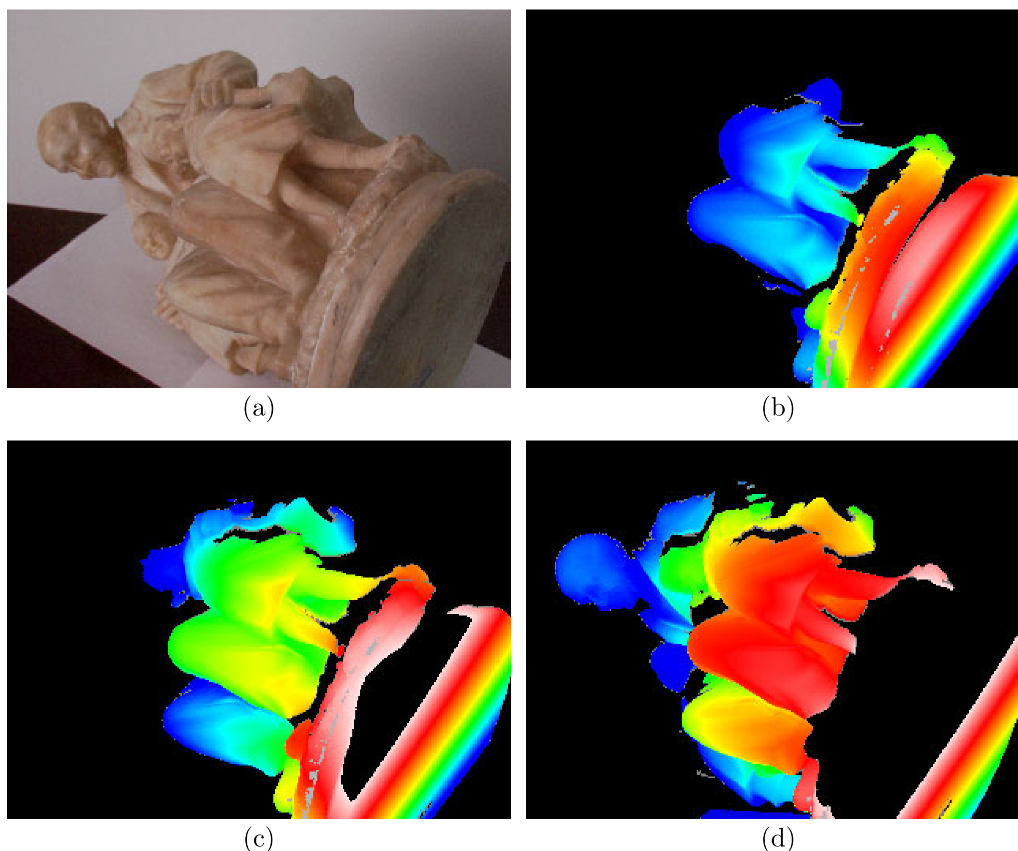


(a) (b)

**Figura 3.6:** Captura com *Dynamic Range Expansion* do *Vivid 910* da *Minolta*. (a) Imagem de intensidade luminosa; (b) Imagem de profundidade. Mesmo com 3 varreduras de *laser* em intensidades diferentes, as regiões pretas não conseguiram ser capturadas.

Outro parâmetro que precisa ser ajustado é o foco da lente. Dados somente são capturados nos trechos da imagem que estejam em foco. Isto acaba fazendo com que o foco defina uma distância mínima e uma distância máxima que limitam a captura em uma única tomada. Para objetos grandes, isto implica em modificar o foco para cada vista, de forma que o objeto esteja sempre dentro dos limites de captura. E quando isto não é possível, várias capturas devem ser feitas, cada uma com um foco diferente, mesmo estando o *scanner* e o objeto imóveis um em relação ao outro. Podemos observar isto ocorrendo em um caso real na figura 3.7.

Resumindo, o processo de aquisição consiste na captura de várias vistas do objeto, sendo que para cada vista, devemos: enquadrar o objeto (através da movimentação do mesmo ou do *scanner*); ajustar os parâmetros de potência e foco do *laser* (algumas vezes num processo iterativo, até que valores aceitáveis de potência e foco sejam descobertos); capturar as imagens de profundidade e de intensidade luminosa; salvar a vista capturada em disco.



(a) (b) (c) (d)

**Figura 3.7:** Como o foco do *laser* interfere na captura de dados. (a) Imagem de intensidade luminosa; (b) Foco próximo; (c) Foco intermediário; (d) Foco afastado. Nota-se que nenhum valor de foco é capaz de capturar todos os dados vista ao mesmo tempo.

Devemos lembrar que cada vista precisa ter regiões em comum com outras vistas, de forma que seja possível alinhá-las [17, 26, 122]. Além disso, esta região de interseção precisa possuir detalhes geométricos capazes de alinhar as vistas. Finalmente, regiões em oclusão (que podem ser observadas, por exemplo, na figura 3.7) precisam ser capturadas de outro ponto de vista, de forma que toda a superfície do objeto seja adquirida.

## 3.3 Análise Qualitativa dos Dados Capturados

A qualidade dos dados retornados pelo *scanner* 3D acaba afetando várias etapas seguintes do *pipeline* de reconstrução de modelos. Logo, é fundamental conhecer os diversos tipos de defeitos que são encontrados nos dados brutos, de forma que estes defeitos não impeçam reconstruções 3D de alta qualidade.

A primeira noção que temos de ter bem clara é que os dados retornados são **ruins**. Por melhor que seja o *scanner*, os dados retornados dificilmente são satisfatórios. A maior parte de nossos testes foi feita com o *scanner Vivid 910* da *Konica-Minolta*, que

é considerado um dos *scanners* 3D mais precisos do mercado [109]; e é com base nestas experiências que afirmamos que a qualidade é baixa. Nas seções a seguir vamos apresentar exemplos dos vários tipos de defeitos que encontramos na prática.

Curless e Levoy [36] estudaram o problema de triangulação via *laser*, levantando vários problemas e sugerindo melhorias através de uma análise de espaço-tempo. Segundo os documentos que acompanham o *Vivid 910* da *Minolta*, ele utiliza esta técnica. Infelizmente, percebemos ainda muitos problemas nos dados capturados.

Para tornar possíveis reconstruções de alta qualidade, precisamos utilizar plenamente a **redundância** das informações. Esta redundância sempre existe, pois para alinhar as várias vistas individuais, estas vistas requerem regiões de sobreposição. E nas regiões de sobreposição, temos várias amostras da mesma superfície. E é através da análise destas amostras que temos que distinguir dados válidos de inválidos, além de atribuir graus de confiabilidade às várias medidas. Em suma, a filosofia requerida é **"nunca confiar em vistas isoladas"**; apenas a análise global de todos os dados é que pode levar a reconstruções automáticas de alta qualidade. Devemos ter em mente esta filosofia quando analisarmos os diversos algoritmos das etapas seguintes do *pipeline* de reconstrução de modelos.

### 3.3.1 Ruído

O defeito mais comum em imagens de profundidade é o ruído na superfície do objeto. Por mais liso e suave que seja um objeto, a sua imagem de profundidade normalmente apresenta uma superfície rugosa. Podemos observar isto na figura 3.8.

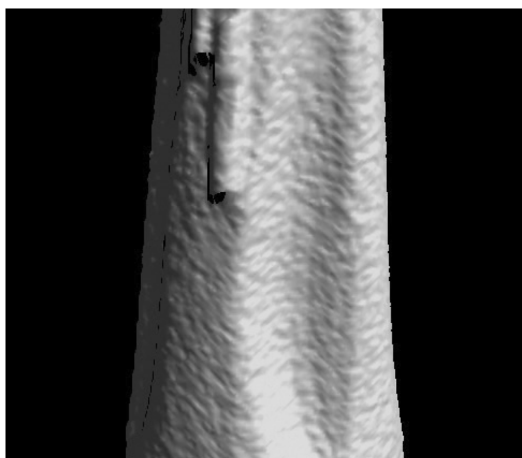


**Figura 3.8:** Ruído presente em vistas de objetos. O objeto original possui uma superfície lisa, no entanto os dados capturados apresentam ruídos.

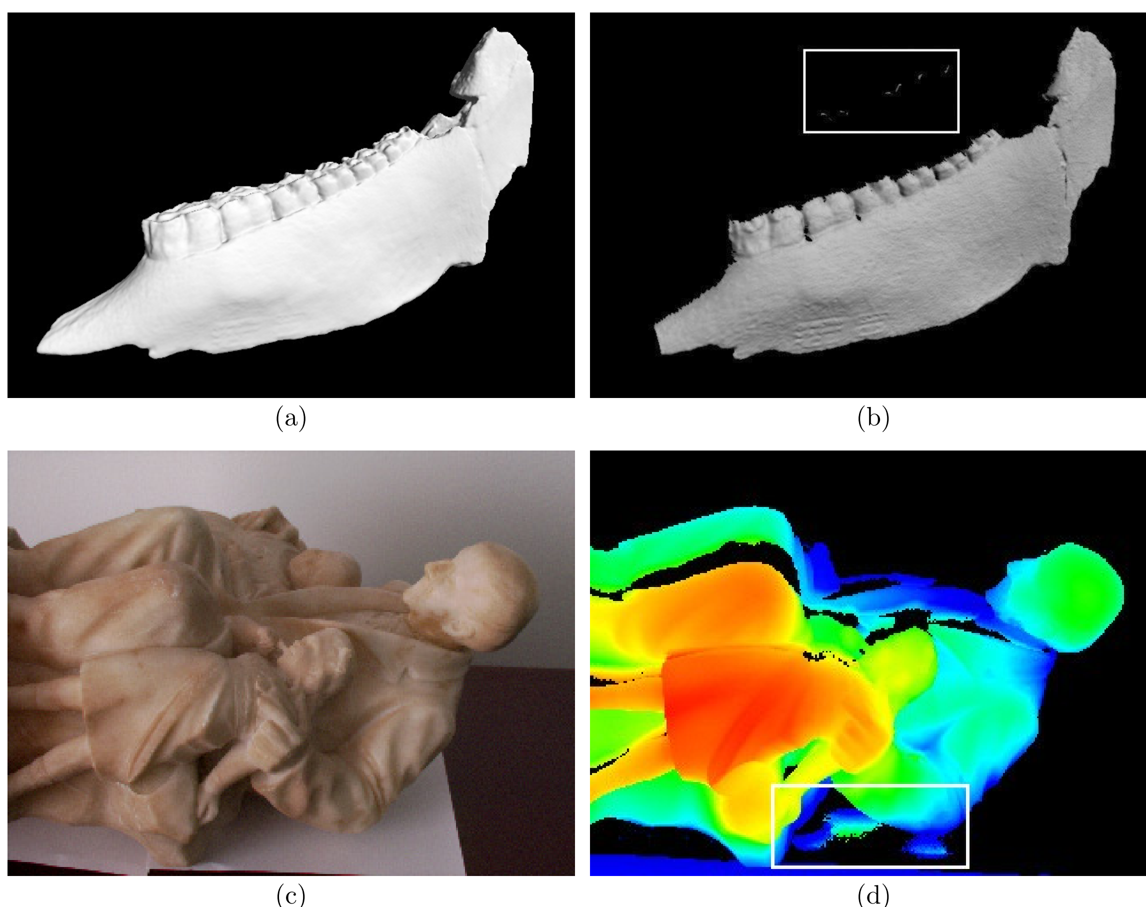
Quanto maior a potência do *laser* utilizada, mais rugosa costuma ser a superfície

capturada. Isto é um problema, pois superfícies mais escuras requerem maior potência de *laser* para serem adquiridas; isto gera como conseqüência que objetos mais escuros costumam ficar mais rugosos do que objetos mais claros.

Uma das formas de atenuar este ruído é através da média ponderada das várias amostras da mesma superfície em vistas diferentes.

### 3.3.2 Dados Falsos

Outro defeito consiste em dados retornados pelo *scanner*, mas que não existem no objeto. Normalmente são pequenos grupos de triângulos que aparecem em regiões que deveriam ser vazias. Costumam ser causados por inter-reflexões do *laser* entre superfícies, ou quando o *laser* pega "de raspão" no objeto, o que é comum na silhueta da peça. Podemos observar este tipo de defeito na figura 3.9.



(a) (b) (c) (d)

**Figura 3.9:** Dados falsos retornados pelo *scanner* 3D. (a) Modelo reconstruído de um fóssil; (b) Uma das vistas do fóssil, com dados falsos; (c) Imagem de intensidade luminosa de uma estátua; (d) Imagem de profundidade correspondente a (c), com dados falsos gerados por reflexão do laser entre a mesa e a estátua, na região do cotovelo.

Uma das formas de se eliminar este tipo de defeito é através da segmentação da vista em regiões conexas, e o descarte de regiões muito pequenas. Outro método consiste em perceber que determinada região do espaço é vazia, levando em conta informações de outras vistas. Como para cada vista sabemos a posição da lente da câmera, torna-se possível definir um volume entre este ponto e os dados retornados, que é sabidamente vazio. Através da composição de informações das várias vistas, temos como eliminar dados de regiões supostamente vazias.

Esta idéia foi originalmente sugerida por Curless e Levoy [37], sendo chamada de *space carving*. No entanto, eles a utilizavam no processo de preenchimento de buracos, e não para a eliminação de defeitos nos dados, como aqui estamos sugerindo. Ela pode ser melhor entendida através da figura 3.10.

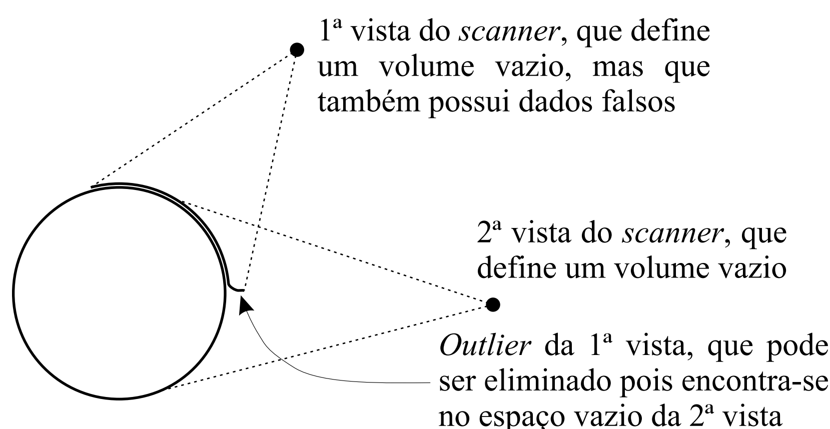



**Figura 3.10:** *Space Carving* utilizado para a eliminação de dados falsos. As regiões vazias de todas as vistas (definidas da posição do *scanner* até os dados capturados) são unidas, e dados contidos nestas regiões são descartados. Neste exemplo, os dados errados da primeira vista podem ser eliminados pois se encontram dentro do volume vazio da segunda vista.

### 3.3.3 Superfícies Deformadas

Este já é um tipo de defeito mais sério, e de detecção mais difícil. Superfícies grandes, e normalmente conexas a dados válidos, são retornadas, mas estas superfícies estão totalmente incorretas em relação ao objeto original. Podemos observar alguns exemplos na figura 3.11.

Este tipo de defeito também é normalmente causado por inter-reflexões do *laser* entre superfícies do objeto, ou entre o objeto e seu plano de apoio. Quanto mais polido (reflexivo) o material do objeto, maior a incidência deste tipo de defeito. Objetos claros são naturalmente mais reflexivos que objetos escuros. Além disso, quanto maior a potência do *laser*, maior a chance de reflexões ocorrerem. Infelizmente, este é um defeito relativamente comum.

A detecção e eliminação deste tipo de defeito de forma automática é particularmente difícil. Nossa solução será apresentada mais adiante, no capítulo 6, que discorre sobre o processo de Integração de Vistas.

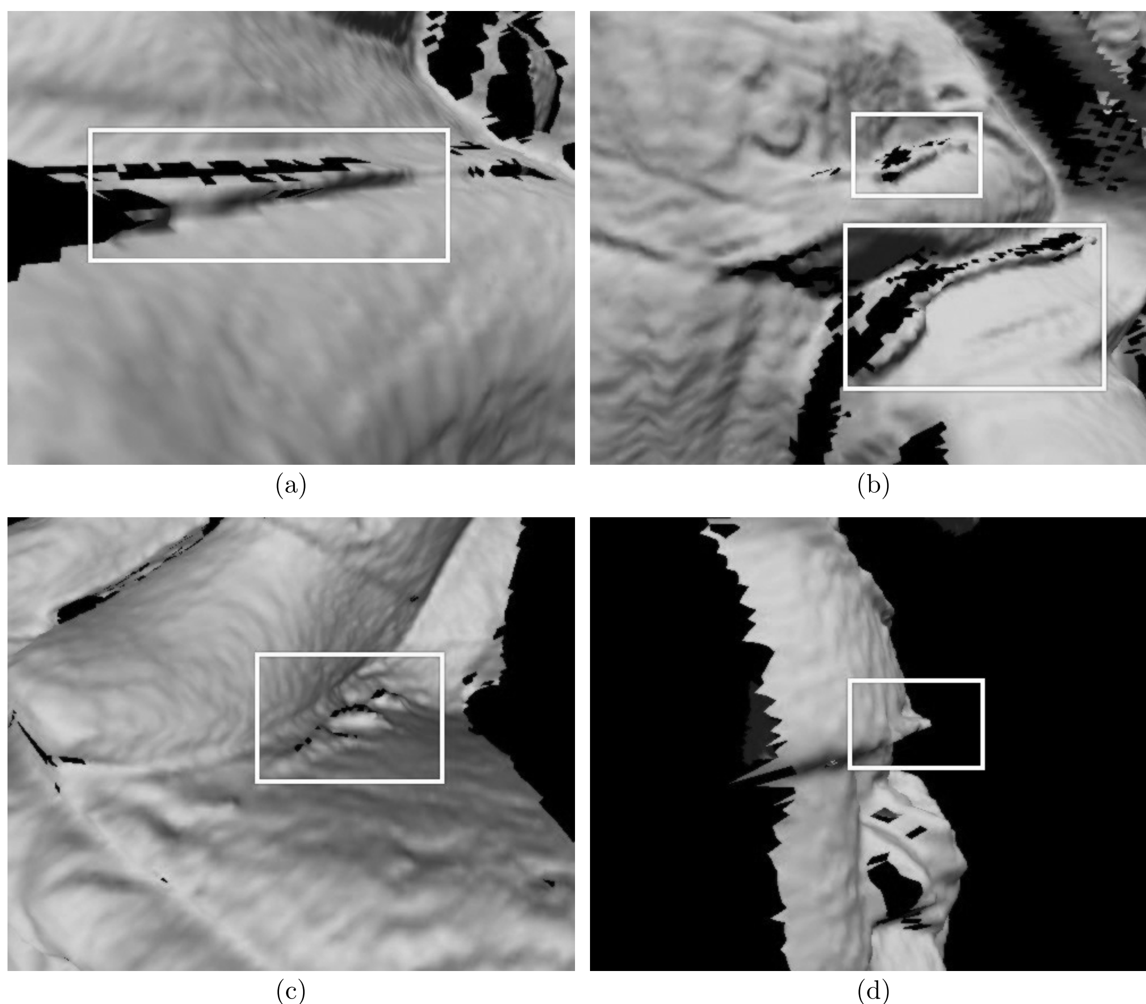



**Figura 3.11:** Superfícies deformadas em vários modelos. (a) e (b) Cristas inexistentes no objeto, mas presentes nas vistas; (c) Ressaltos em uma região que deveria ser plana; (d) Ressalto apontando para fora do objeto, em uma região que deveria formar um canto simples como o da região vizinha.

### 3.3.4 Superfícies Falsas na Silhueta

Outro problema encontrado surge do processo de triangulação dos dados retornados pelo *scanner*. O *scanner* nos fornece apenas uma nuvem de pontos, e estes pontos devem ser interligados para a formação de uma superfície. Como espacialmente os pontos são retornados como um *grid* retangular, fica fácil interligar os pontos com seus vizinhos, conforme pode ser visto na figura 3.12.

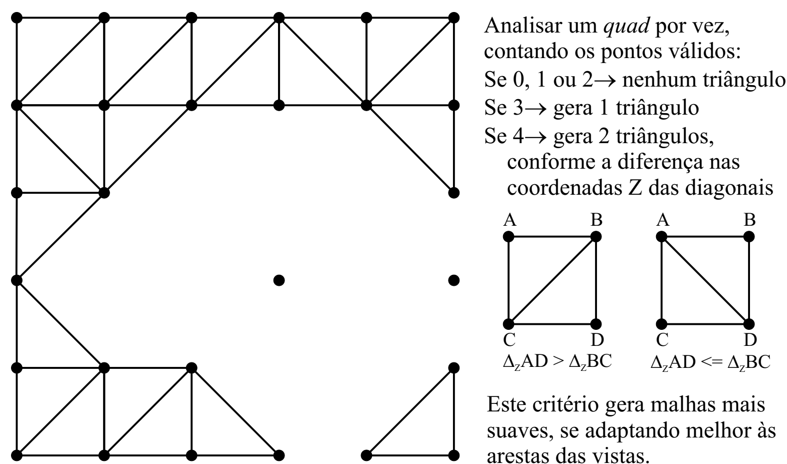
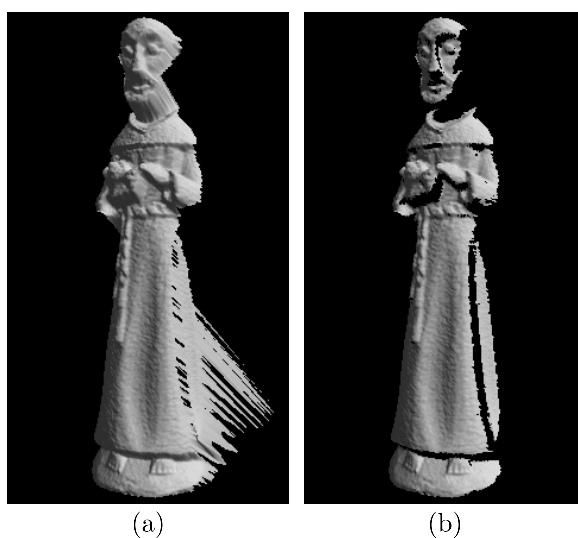



**Figura 3.12:** Exemplo de conversão de nuvem de pontos 3D para malha 3D. Note-se que alguns pontos são descartados, porque não possuem vizinhos adequados.

O problema consiste em saber quando os pontos não são realmente vizinhos, e se referem a partes diferentes do objeto. Isto ocorre nas silhuetas, tanto internas quanto externas do objeto. O nosso critério consiste em verificar o ângulo formado entre a normal do triângulo candidato e a linha de visão do *scanner* ao centro do triângulo; se este ângulo for maior do que um determinado limiar, o triângulo é descartado, pois assume-se que ele foi formado em uma silhueta. Em termos matemáticos, calcula-se o produto interno entre a normal da face e a linha de visão normalizada, que parte do centro da lente para o centro da face em questão. As faces cujo produto interno calculado for menor que o cosseno do limiar, isto é, faces cujas normais acabam formando ângulos maiores que o ângulo do limiar, são descartadas. Tal método pode ser observado na figura 3.13.



(a) (b)

**Figura 3.13:** Eliminação de faces falsas na silhueta. (a) Dados brutos retornados pelo scanner; (b) Dados descartando faces aproximadamente paralelas à linha de visão. Neste exemplo foi usado um limiar de 75°.

Aqui esbarramos numa solução de compromisso: Se o ângulo de tolerância for muito baixo, superfícies reais são descartadas; se ele for muito alto, superfícies inexistentes acabam permanecendo, e infelizmente estas superfícies estão conexas a superfícies corretas, se comportando como o defeito anterior de superfícies deformadas. Em nossos testes, ângulos entre 70° e 80° fornecem bons resultados.

Além disso, existe outra forma deste defeito ocorrer: Os dados retornados pelo *scanner* próximos a silhuetas são pouco confiáveis, isto é, costumam apresentar erros maiores. Assim sendo, algumas vezes estas bordas próximas a silhuetas são realmente superfícies deformadas como no caso anterior.

Uma solução grosseira para estes problemas, algumas vezes sugerida, é a de descartar as bordas da imagem de profundidade. No entanto, esta solução não é boa: primeiro, porque os defeitos costumam ser bem maiores do que apenas um triângulo (ver figura 3.14); e segundo que desta forma dados bons são descartados, o que acaba dificultado outras etapas do *pipeline*. Nossa sugestão é de que estes defeitos sejam resolvidos com o mesmo algoritmo utilizado para a remoção das superfícies deformadas do caso anterior, que será apresentado no capítulo 6 sobre Integração de Vistas.

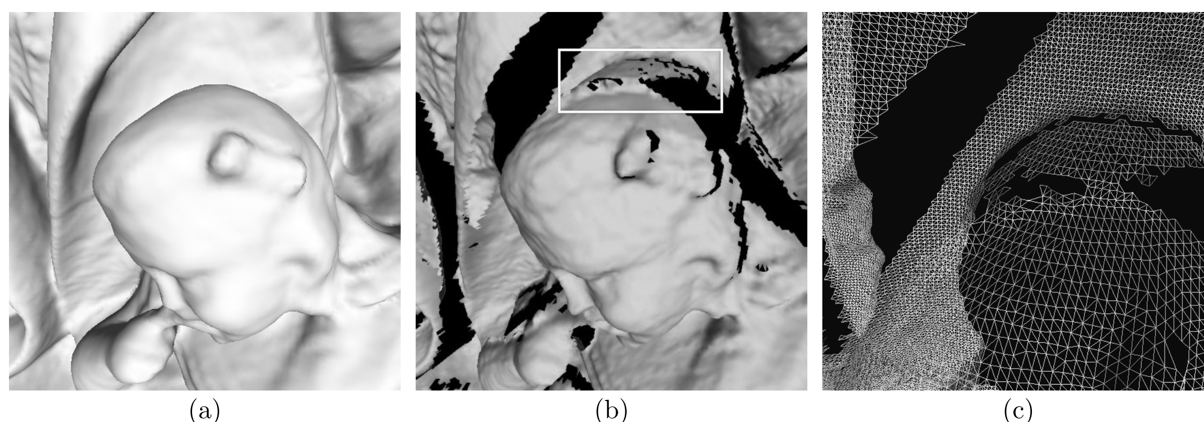



**Figura 3.14:** Superfícies incorretas na silhueta de vistas. (a) Modelo final reconstruído; (b) Uma das vistas, com uma região com superfícies falsas na silhueta indicada por um retângulo; (c) *Zoom* na região do defeito em modo *wireframe*, mostrando que o número de triângulos incorretos é muito alto, ou seja, o descarte simples de triângulos de borda não resolveria o problema.

### 3.3.5 Dados Impossíveis de Serem Capturados

O último tipo de problema se refere à falta de dados. A técnica de aquisição por triangulação *laser* não funciona em reentrâncias profundas, ou em cavidades internas dos objetos; isto ocorre devido ao ângulo de triangulação necessário entre o emissor *laser* e a lente da câmera. Desta forma, existem certas regiões da superfície do objeto que não são

capturadas, que acabam ocasionando buracos no objeto reconstruído. Um exemplo pode ser visto na figura 3.15.

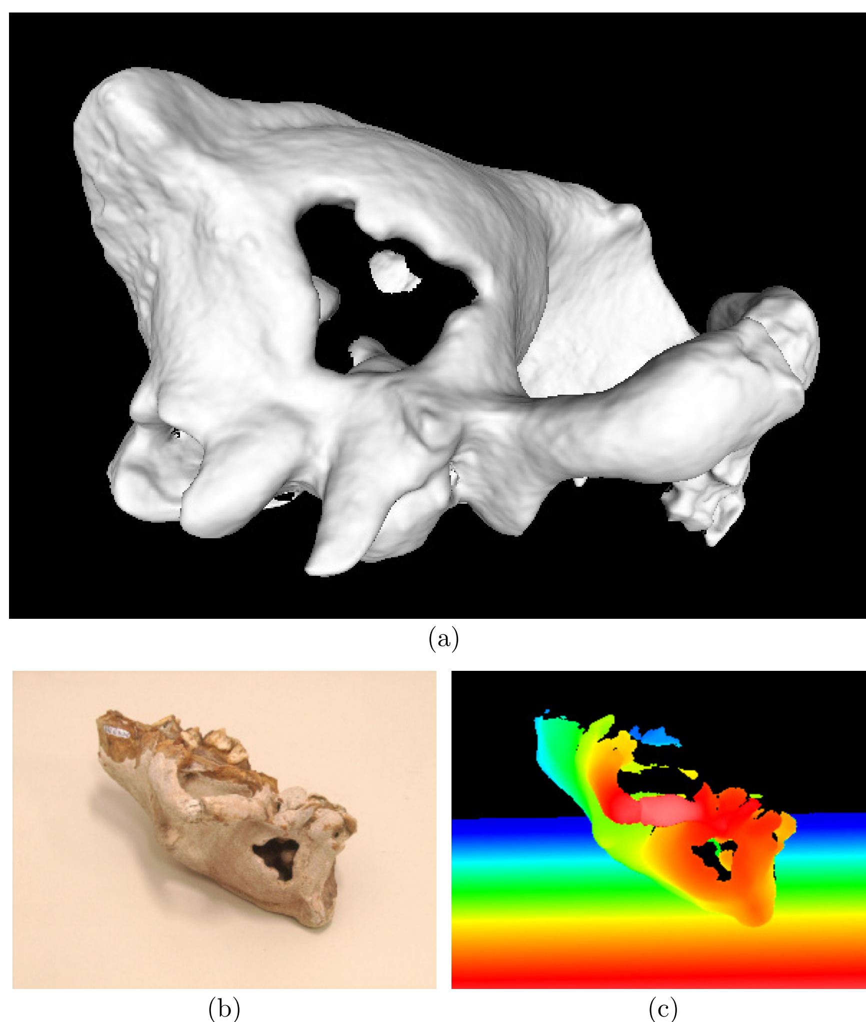


**Figura 3.15:** "Buracos" devidos a regiões inacessíveis ao *scanner*. (a) O interior do crânio deste fóssil não pôde ser capturado através da fratura, pois se trata de uma reentrância muito profunda para o ângulo de triangulação do *scanner*. (b) e (c) Uma das vistas capturadas, e como o interior do crânio sofre oclusão.

Para muitas aplicações, estes buracos são inaceitáveis; tornam-se necessárias técnicas de preenchimento de buracos, de forma a completar os dados. Existem vários algoritmos para preenchimento de buracos, que serão citados no capítulo 7 correspondente mais adiante. O ponto que gostaríamos de ressaltar é que para este processo, quanto mais informações conseguirmos extrair dos dados capturados de forma a guiar o preenchimento de buracos, mais confiável será a superfície criada. Desta forma, nossa solução procura acoplar o processo de Integração de Vistas com o processo de Preenchimento de Buracos, visando garantir uma solução mais correta e plausível para o problema dos dados inexistentes.

## 3.4 Pré-Processamento

Depois de adquirida, a informação de profundidade normalmente precisa ser pré-processada antes de ser utilizada nos estágios seguintes do pipeline de reconstrução 3D. Alguns processamentos realizados são:

- Separar os dados do objeto sendo capturado dos dados do fundo. Como podemos observar nas figuras 3.2(b), 3.6(b) e 3.15(c), além dos pontos do objeto, obtivemos uma série de pontos 3D da mesa onde os objetos estavam apoiados. Nossa solução para este problema será apresentada na seção 3.4.1 a seguir;
- Conversão dos dados de profundidade para um formato próprio de arquivo. Isto é feito para deixar o resto do *pipeline* independente do SDK do scanner [85]. Criamos um arquivo com extensão RNG para armazenar os dados de profundidade, e um arquivo TGA para armazenar as imagens de intensidade luminosa de cada vista. Ambos os formatos são binários, de forma a reduzir o espaço de armazenamento necessário;
- Geração de modelos 3D de cada vista (geração de faces e normais de vértices a partir das nuvens de pontos). Este processo foi explicado anteriormente na seção 3.3.4, e através da figura 3.12. O método que utilizamos para a geração de normais foi ponderar a normal de cada face incidente ao vértice pelo ângulo da face no vértice, somar os vetores ponderados e normalizar o resultado. Este método foi adotado por Thürmer e Wüthrich [147], e segundo o estudo apresentado em [77], apresenta ótima precisão aliada a um ótimo desempenho;
- Geração de informações de adjacência destes modelos (listas de faces e arestas para cada vértice, faces vizinhas de cada face, etc.);
- Criação de *kd-trees* para a busca eficiente de vértices mais próximos de um ponto de busca [10];
- Geração e otimização das malhas 3D das vistas para visualização com OpenGL [19].

Todos estes processamentos e estruturas de dados (com exceção da eliminação dos dados 3D de fundo) são feitos pelo nosso programa IMAGO *ModelTool*, que é responsável por todas as outras etapas do *pipeline* de reconstrução 3D. Para cada vista é gerado um arquivo temporário, para o armazenamento destas informações. Isto acelera a execução da ferramenta, pois estes cálculos são efetuados apenas uma vez, independente do número de reconstruções feitas para o mesmo objeto.

### 3.4.1 Remoção dos Dados de Fundo

Na literatura, não se comenta muito sobre este problema de separar os dados 3D do objeto dos dados de fundo (*background*) em imagens de profundidade. Nos *softwares* comerciais que acompanham os *scanners*, a técnica utilizada é capturar o fundo sem o objeto, e em seguida efetuar nova captura com o objeto. As regiões onde ocorram diferenças correspondem a dados 3D do objeto [85].

Infelizmente, apesar de intuitiva, esta solução é muito problemática na prática. Isto ocorre por causa dos vários defeitos e restrições que já comentamos nas seções anteriores. Normalmente, a cada tomada do objeto, pequenos ajustes de foco e intensidade de *laser* são necessários. Quando uma mudança destas ocorre, os dados de fundo capturados já não são mais os mesmos (vide figura 3.7). Isto nos obriga a recapturar o fundo com os novos parâmetros. Na prática, o que ocorreria seria a captura de duas imagens para cada tomada, retirando-se e recolocando-se o objeto. E esta manipulação excessiva dos objetos, além de trabalhosa, em alguns casos não é desejável (peças sensíveis que devem ser manipuladas com cuidado, como fósseis ou obras de museu, por exemplo).

Outra possibilidade seria a eliminação manual dos dados de fundo. Isto vai contra nosso objetivo de automatizar o processo o máximo possível, além de aumentar em muito o trabalho necessário para a reconstrução dos dados. Devido a todas estas dificuldades, criamos um método que é muito mais prático que os anteriores. Capturamos os objetos apoiados sobre um plano, que pode ser uma mesa, o chão, etc. A única restrição é que o objeto deve estar acima do plano, e o *scanner* não pode capturar nada abaixo do plano. Esta restrição costuma ser facilmente atendida, tanto em ambiente de laboratório quanto em campo.

Em seguida, desenvolvemos um detector automático de planos de apoio. Nos casos representados nas figuras 3.2(b), 3.6(b) e 3.15(c), é relativamente fácil detectar o plano de apoio, pois existe um grande quantidade de informação sobre o plano capturado. Em outros casos, como o da figura 3.16, o processo torna-se bem mais complicado, pois não temos uma superfície bem definida, e sim apenas alguns pontos espalhados que devem ser interpretados como um plano de apoio e eliminados.

Nosso processo de detecção de planos utilizou-se de várias técnicas, como uma variação da transformada de Hough [74], métodos robustos (MSAC) [148] e conhecimentos específicos sobre o problema, como não haver nenhuma informação abaixo do plano de apoio (exceto *outliers*, ou seja, dados incorretos). Este processo foi implementado junto ao nosso programa de aquisição de dados, o *VividControl*.

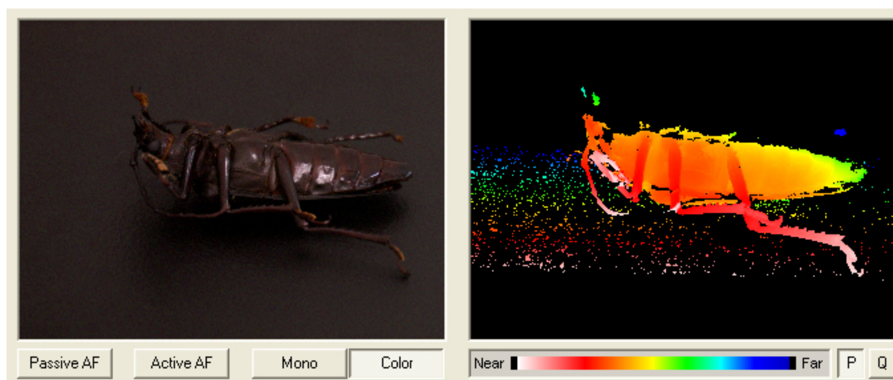



**Figura 3.16:** Plano de apoio com dificuldades de detecção.

Inicialmente, escolhemos alguns parâmetros para a detecção do plano. Aqui esbarramos em uma solução de compromisso, entre detectar o plano com alta precisão (mas que pode acabar não encontrando o plano), ou com menor precisão (mas que pode acabar gerando planos incorretos, isto é, falsos positivos). Em nossa implementação, fizemos todos os parâmetros controlados por um único ajuste, o de precisão. Na prática, tentamos detectar o plano com maior precisão, e quando não o encontramos, vamos interativamente reduzindo a precisão até obter um resultado. O plano detectado é mostrado na imagem de profundidade, de forma que tenhamos um *feedback* durante o processo. Os parâmetros individuais (escolhidos a partir do parâmetro único de precisão) serão detalhados nas explicações do algoritmo a seguir.

Nosso algoritmo de detecção de planos segue os seguintes passos:

1. Estimamos normais para cada ponto 3D da nuvem de pontos, observando seus 4 vizinhos. Em seguida, dividimos o espaço das normais em *buckets*[1] usando uma estrutura de dados similar a um *cube map* [62], acumulando em cada *bucket* quantos pontos possuem normais na direção geral do *bucket* (conforme a figura 3.17). Todo este processo visa criar uma representação da distribuição das normais encontradas na cena capturada;

2. Em seguida, ordenamos os *buckets* por ordem decrescente do número de normais. Cada *bucket* representa uma direção candidata para o plano de apoio. Como os pontos em um mesmo plano tendem a possuir normais semelhantes, a idéia aqui é que grandes regiões planares da vista capturada são representadas por *buckets* com valores altos. Desta forma, examinamos os *buckets* por ordem decrescente de valor para acelerar o processo de descoberta do plano de apoio;

[1] O termo *bucket* é usado como no algoritmo *bucketsort* [35].

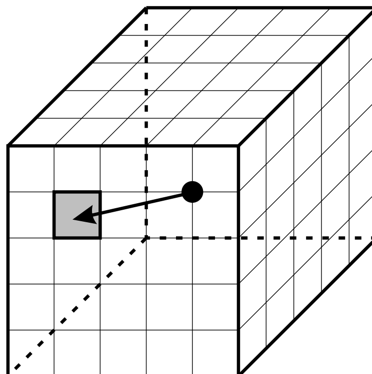


**Figura 3.17:** A normal é imaginada no centro do cubo, e aponta para uma das 6 faces. Dentro da face, um *bucket* é apontado pela normal, e neste *bucket* incrementa-se um contador. Desta forma, os *buckets* acabam armazenando as distribuições das normais no espaço.

3. Para cada *bucket*, geramos uma normal que aponta para o seu centro. Esta é apenas uma direção aproximada, que será refinada posteriormente. Em seguida, projetamos todos os pontos 3D sobre a normal candidata, através da operação de produto interno de vetores. Isto acaba nos gerando uma distribuição 1D, que pode ser representada por um histograma. O que estamos procurando é um histograma como o da figura 3.18, onde há uma grande concentração de valores (pico) no início do histograma, devido ao plano de apoio. Além disso, antes deste pico devem haver muito poucos pontos (apenas alguns *outliers*), e o resto dos pontos (o objeto em si) devem estar espalhados após o pico. Se a normal candidata não apresentar um histograma neste formato, ela é descartada, e o próximo *bucket* é analisado;

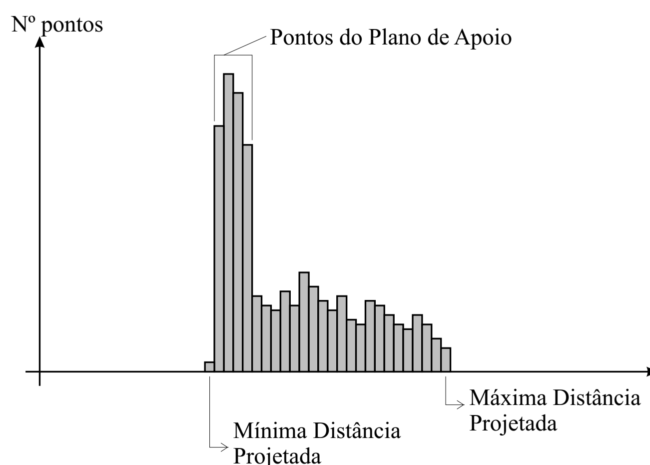



**Figura 3.18:** Distribuição da nuvem de pontos 3D projetados sobre a normal candidata do *bucket*, através da operação de produto interno de vetores. Neste exemplo, a normal candidata é bem próxima da normal do plano de apoio, logo os pontos do plano acabam sendo projetados aproximadamente à mesma distância, gerando um pico no início dos dados do histograma.

4. Para testar a conformidade do histograma, utilizamos dois parâmetros. O primeiro é a porcentagem mínima de pontos no pico do histograma que devem concordar

com a normal candidata ao plano ($minNormal$). Para um ponto concordar com a normal do plano, sua normal e a normal do plano devem formar um ângulo pequeno. Nossos experimentos demonstraram que um ângulo de 30° fornece bons resultados. O segundo parâmetro é a porcentagem mínima de pontos no plano de apoio, isto é, no pico do histograma ($minPeak$). Somamos as distribuições desde o início do histograma, até obtermos a porcentagem $minPeak$ dos pontos totais, e a partir daí expandimos o pico enquanto o parâmetro $minNormal$ for atendido. Ao final do processo, detectamos o início e fim do pico, e se o histograma é aceito ou não. Se não for aceito, voltamos ao passo 3 com o próximo *bucket*;

5. Se o histograma foi aceito, significa que encontramos uma normal candidata promissora. Agora, utilizamos o algoritmo MSAC [148] para refinar a normal candidata, de forma a obter a normal do plano de apoio com maior precisão. Devemos notar que esta etapa é fundamental, pois as normais derivadas dos *buckets* são apenas aproximadas. Para esta etapa, apenas consideramos os pontos presentes no pico do histograma, calculados no passo 4. Resumidamente, o que o algoritmo MSAC faz é sortear aleatoriamente grupos de 3 pontos, definir o plano que passa pelos pontos e medir a compatibilidade do conjunto de todos os pontos do pico em relação a este plano. Este processo é repetido um grande número de vezes, e o melhor plano encontrado durante o processo é o selecionado. Durante o algoritmo MSAC, utilizamos um novo parâmetro, $mmTolerance$. Este parâmetro especifica uma distância em milímetros, e pontos cuja distância ao plano sendo calculado seja menor que $mmTolerance$, são considerados *inliers*, isto é, dados confiáveis, pertencentes ao plano de apoio;

6. Uma vez descoberto o melhor plano pelo MSAC, chegou a hora de mais um refinamento dos dados. O objetivo principal do MSAC é categorizar os pontos como *inliers* ou *outliers*, de forma que apenas os *inliers* sejam utilizados para a definição do plano de apoio. Agora que já sabemos quais pontos são *inliers*, calculamos o melhor plano que aproxime os *inliers* utilizando o método de mínimos quadrados. Em nossa implementação, construímos a matriz de covariância dos *inliers*, e calculamos seus auto-vetores com o método de Jacobi. A normal do plano é o auto-vetor correspondente ao menor auto-valor [112], e a origem do plano corresponde ao centro de massa dos *inliers*;

7. Finalmente, fazemos uma última validação do plano encontrado. Utilizamos mais dois parâmetros, a porcentagem máxima de pontos abaixo do plano de apoio ($maxBelow$), e a porcentagem mínima de pontos acima do plano de apoio ($minAbove$). Dado o plano definido pelo método de mínimos quadrados, categorizamos os pontos em acima, abaixo ou no plano, utilizando o parâmetro $mmTolerance$.

Se o número de pontos abaixo do plano for menor que o definido por $maxBelow$, o número de pontos no plano for maior que o definido por $minPeak$, e o número de pontos acima do plano for maior que o definido por $minAbove$, o plano é aceito, e os dados do objeto correspondem aos pontos acima do plano. Caso contrário, voltamos ao passo 3 para tentar um novo *bucket*.

Podemos observar um fluxograma do algoritmo de remoção de planos na figura 3.19.

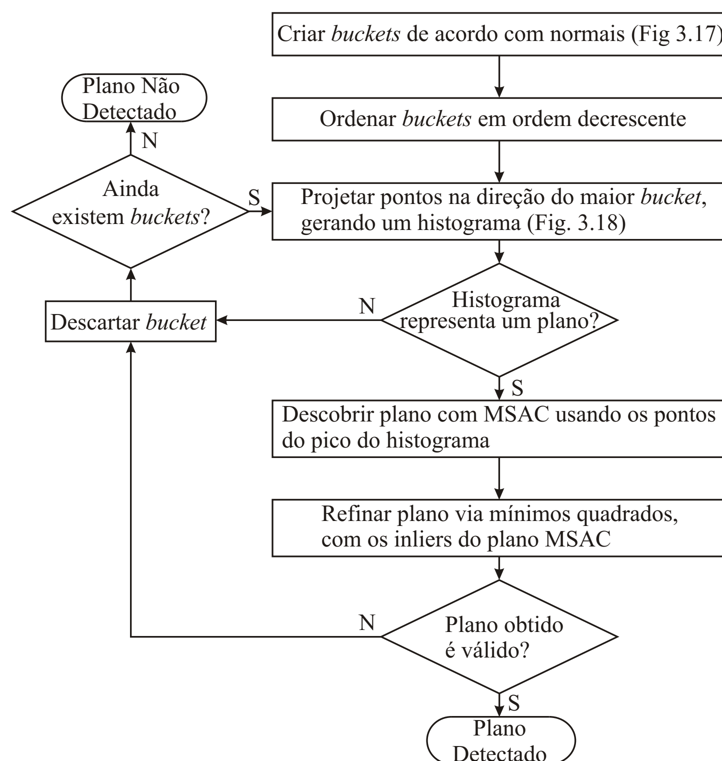



**Figura 3.19:** Fluxograma do algoritmo de remoção de planos de apoio.

O algoritmo foi implementado em nossa ferramenta *VividControl*, e funciona muito bem na prática, conforme pode ser observado na figura 3.20. Ele é capaz de detectar planos de apoio como os exibidos nas figuras 3.16 e 3.20(d), onde apenas 5,6% dos pontos totais pertencem ao plano de apoio. A etapa 4, que testa a conformidade do histograma, pode parecer desnecessária, mas não o é; sem ela, o algoritmo robusto MSAC falharia, pois ele só consegue trabalhar com menos de 50% de *outliers*. Se o MSAC fosse utilizado diretamente sobre os dados da figura 3.20(d), ele não seria capaz de encontrar o plano, pois apenas 5,6% dos pontos totais são *inliers*. O que a etapa 4 faz é aumentar a concentração de *inliers*, para garantir o funcionamento do MSAC. Além disso, ele reduz muito o tempo de detecção. Isto ocorre porque o algoritmo MSAC é muito custoso, e com a etapa 4 apenas normais promissoras levam ao cálculo do MSAC. Em nossos testes, o plano de apoio costuma ser detectado instantaneamente; e quando ele não existe, o algoritmo leva

menos de 5 segundos para chegar a esta conclusão, tempo este medido em um computador PC comum.

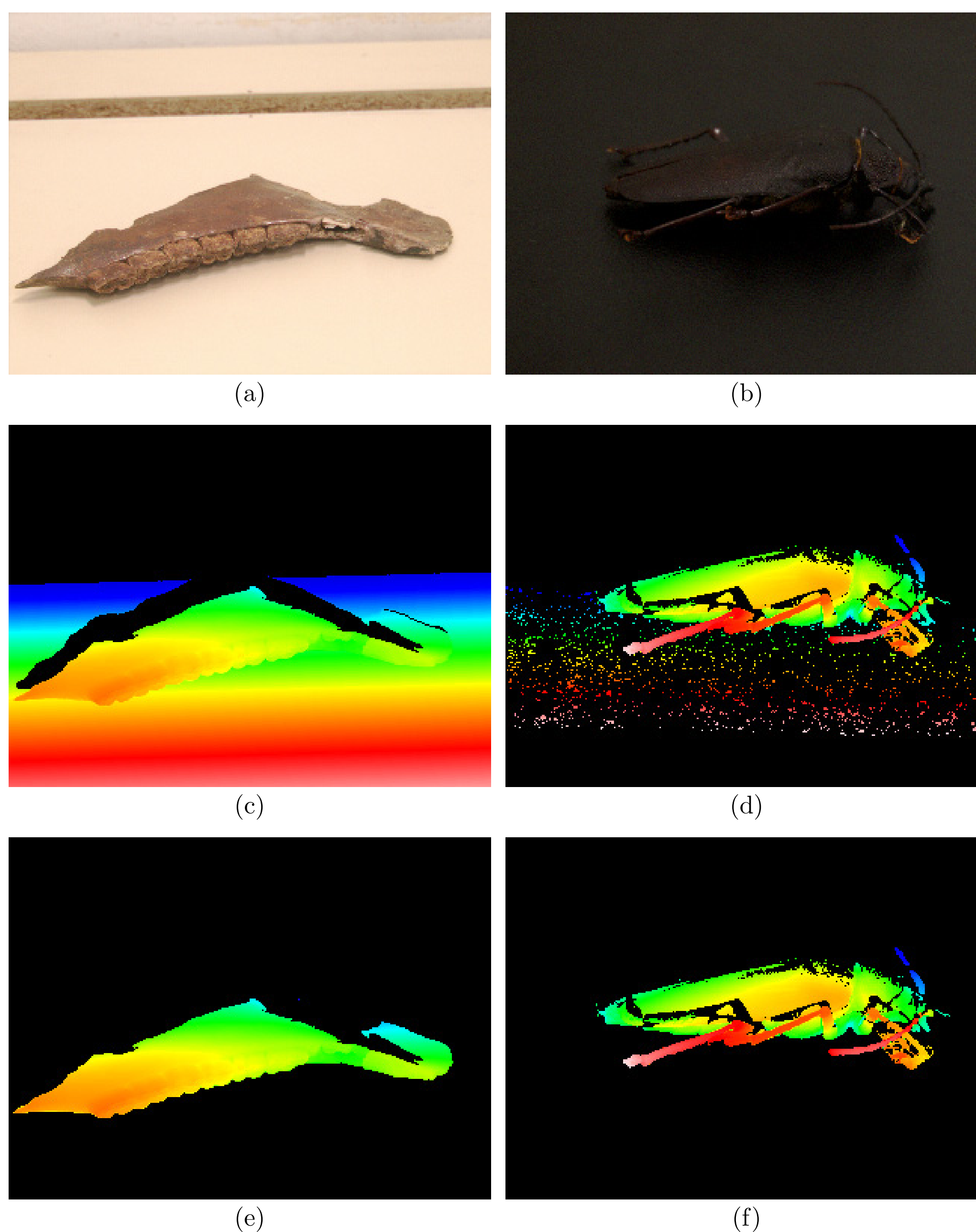


**Figura 3.20:** Resultados do algoritmo detector de planos de apoio. (a)(b) Fotos dos objetos, mostrando onde estão apoiados; (c)(d) Imagens de profundidade capturadas, que contém pontos dos planos de apoio; (e)(f) Imagens de profundidade com os planos de apoio removidos automaticamente. Podemos notar na figura (b) como técnicas de segmentação por cor seriam difíceis de serem aplicadas com sucesso neste caso, devido às cores semelhantes entre o fundo e o objeto.

Podemos observar na tabela 3.1 os diversos parâmetros que controlam o algoritmo detector de planos, e como eles estão associados ao parâmetro de precisão selecionado pelo usuário. Ele foram obtidos através dos nossos experimentos em vários objetos diferentes.

**Tabela 3.1:** Parâmetros do algoritmo detector de planos de apoio. A precisão 0 corresponde à menor precisão, e a precisão 5 à maior. Quanto maior a precisão, maior o risco do plano não ser detectado; quanto menor a precisão, maior o risco de planos incorretos serem retornados.

| Precisão | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| *mmTolerance* (*mm*) | 4.0 | 2.5 | 1.5 | 1.5 | 1.0 | 0.5 |
| *maxBelow* (%) | 3.0 | 2.0 | 1.5 | 1.5 | 1.0 | 0.5 |
| *minPeak* (%) | 1.0 | 1.0 | 1.0 | 5.0 | 10.0 | 20.0 |
| *minAbove* (%) | 7.0 | 8.0 | 8.0 | 8.0 | 9.0 | 10.0 |
| *minNormal* (%) | 77.5 | 80.0 | 80.0 | 80.0 | 82.5 | 85.0 |

Além de separar os dados do objeto dos dados de fundo, o plano de apoio detectado por nosso algoritmo acaba sendo utilizado posteriormente em nosso *pipeline* de reconstrução 3D, pois pontos abaixo dele sabidamente não pertencem ao objeto. Desta forma, conseguimos transformar uma dificuldade em uma aliada, pois cada vista que possui um plano de apoio ajuda a delimitar o *convex hull* do objeto. Podemos entender isto no contexto da figura 3.10, pois os planos de apoio também definem regiões vazias do espaço (todo o volume *abaixo* de cada plano). Isto auxiliará o processo de fusão dos dados 3D, pois ajudará a detectar e descartar dados incorretos, ou seja, dados que se localizem em regiões vazias do espaço.

## 3.5 Conclusões e Trabalhos Futuros

A partir de nossas experiências práticas com o processo de aquisição de dados, podemos citar algumas conclusões:

- Atualmente, o processo de captura requer mão-de-obra especializada. A pessoa responsável pela captura deve ter uma boa noção sobre os parâmetros do *scanner*, de forma a acelerar o processo de descoberta destes parâmetros. Além disso, muita atenção deve ser dada ao posicionamento das diversas vistas, de forma que uma região suficiente de sobreposição exista entre as vistas vizinhas. Finalmente, todo o objeto deve ser capturado, e para isso uma metodologia de escaneamento deve ser seguida. Objetos com geometria complexa complicam muito o processo, pois requerem várias vistas complementares de forma que regiões em oclusão tenham seus dados capturados;

- Os dados capturados consomem muito espaço em disco, o que deve ser levado em consideração. Além disso, um padrão de nomenclatura para os arquivos gerados também é muito importante para facilitar o uso posterior dos dados;

- Mover e rotacionar o objeto costuma ser bem mais rápido do que reposicionar o *scanner* 3D, e para objetos pequenos é a solução recomendada;

- Boas condições de iluminação são necessárias para que as imagens de intensidade luminosa capturadas sejam utilizáveis para a reconstrução da textura do objeto. Ambientes muito claros ou muito escuros costumam prejudicar a captura.

Para facilitar o processo, seria interessante que o *software* de captura ajudasse a identificar regiões ainda não capturadas, bem como estimar a área de sobreposição das vistas. Isto é relativamente trabalhoso, pois exige que as vistas sejam alinhadas durante a sua captura; ou seja, o programa de aquisição deveria ser mais integrado com as próximas etapas do *pipeline*. Outra dificuldade a ser superada é manter todos os dados lidos na memória, pois a quantidade de informações pode ser arbitrariamente grande, podendo em alguns casos chegar a vários gigabytes.

Uma possibilidade seria eliminar dados redundantes nas interseções das vistas, e trabalhar diretamente com nuvens de pontos ao invés de superfícies, para facilitar a manipulação 3D em tempo real. Como as vistas costumam ter uma alta densidade de pontos, renderizar diretamente as nuvens de pontos geraria um resultado praticamente igual à renderização de superfícies, e desta forma nenhum algoritmo de fusão de vistas seria necessário nesta ferramenta de captura de dados. As nuvens de pontos poderiam ser iluminadas normalmente, pois temos como estimar uma normal para cada vértice, além de utilizar sua cor obtida da imagem de intensidade luminosa. A manipulação desta nuvem de pontos permitiria uma visualização rápida das regiões ainda não capturadas do objeto.

# CAPÍTULO 4

# ALINHAMENTO DE VISTAS

O processo de alinhamento (chamado em inglês de *registration*) consiste em fazer com que cada vista (imagem de profundidade) seja posicionada corretamente em relação às outras vistas, permitindo que um modelo final seja integrado a partir das vistas individuais. Este posicionamento é possível porque existem regiões de interseção entre as várias vistas, e fazendo com que estas regiões de interseção coincidam conseguimos que as vistas sejam alinhadas. Computacionalmente, isto equivale a descobrir matrizes homogêneas de transformação $4 \times 4$ para cada vista, que transformam os diversos sistemas de coordenadas individuais para um sistema de coordenadas global do modelo.

Cada vista possui o seu sistema de coordenadas, que é relativo ao *scanner*. Normalmente a origem do sistema se encontra no centro da lente do *scanner*, e com a parte negativa do eixo Z alinhado com o eixo óptico da lente, conforme podemos observar na figura 4.1.

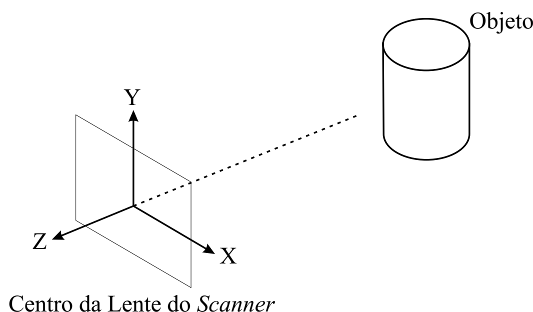



**Figura 4.1:** Sistema de coordenadas local de cada imagem de profundidade.

Como para cada vista ou o *scanner* ou o objeto são movimentados (de forma que outras regiões do objeto pudessem ser capturadas), percebemos que existem vários sistemas locais que devem ser relacionados através de matrizes de transformação de coordenadas.

Convém lembrar que as matrizes possuem apenas componentes de translação e rotação; diferentemente de outros métodos de processamento de imagens, aqui os dados capturados estão todos na mesma escala, pois são capturados em unidades reais (milímetros), e os objetos são rígidos.

## 4.1 Revisão Bibliográfica

O processo de alinhamento de imagens de profundidade já foi extensivamente discutido no meio científico. Várias abordagens foram propostas, e inúmeros algoritmos sugeridos. Os artigos de Rusinkiewicz [122] e Salvi [127] são ótimos *surveys* que comparam os principais métodos utilizados na etapa de alinhamento, os quais rapidamente comentaremos nesta seção.

Existem basicamente 2 abordagens para o processo de alinhamento: alinhamento por pares de vistas (normalmente seguidos por uma etapa de alinhamento global), e alinhamento de todas as vistas simultaneamente.

### 4.1.1 Alinhamento por Pares de Vistas

A primeira abordagem consiste em efetuar vários alinhamentos entre pares de imagens de profundidade vizinhas. Cada par possui uma região de sobreposição, e o objetivo é descobrir a matriz de transformação que converte os pontos da segunda vista para a primeira. Isto é conseguido alinhando as regiões de sobreposição das duas vistas.

Este processo de alinhamento pode ser novamente subdividido em 3 sub-etapas: pré-alinhamento, alinhamento propriamente dito e alinhamento global.

#### 4.1.1.1 Pré-Alinhamento

O pré-alinhamento consiste em obter uma estimativa inicial do transformação que alinha o par de vistas. Isto é necessário pois a maioria dos algoritmos de alinhamento só convergem se esta estimativa inicial for relativamente boa. Técnicas como Algoritmos Genéticos ou *Simulated Annealing* podem eventualmente funcionar sem um pré-alinhamento inicial; mas normalmente o preço pago em termos de número de iterações para convergência é tão alto, que faz com que estes métodos também se utilizem destas técnicas. Várias formas de obter este pré-alinhamento são possíveis:

- **Manualmente**: O usuário pode movimentar as vistas, de forma a deixá-las próximas de sua posição correta. Uma solução comum costuma ser selecionar 3 ou mais pontos em uma vista, e marcar os pontos correspondentes na outra vista. Desta forma, uma estimativa do alinhamento pode ser conseguida [85];

- **Através do controle do processo de aquisição**: Se a captura é feita utilizando uma *turntable* (vide figura 4.2), existe um controle do ângulo de giro da *turntable*, de

forma que uma aproximação da transformação entre as vistas possa ser obtida [85]. Outra possibilidade é posicionar o *scanner* através de um braço robótico ou outro dispositivo mecânico qualquer [89]. Finalmente, marcas (por exemplo, etiquetas coloridas) podem ser fixadas no objeto para estimar o pré-alinhamento, através da sua detecção nas imagens de intensidade luminosa [85].

Aqui convém lembrar que todo dispositivo mecânico possui uma precisão de posicionamento. Logo, estimativas obtidas através destes dispositivos, apesar de normalmente serem muito boas, não podem ser consideradas como o alinhamento final das vistas, apenas como uma estimativa inicial. E o acréscimo de marcas nos objetos acaba impedindo o uso das imagens de intensidade luminosa para a reconstrução de textura, pois elas escondem pequenas regiões dos objetos;

**Figura 4.2:** *Turntable* utilizada para o pré-alinhamento de vistas. Consiste em uma plataforma giratória, de forma que o ângulo de rotação entre cada vista possa ser pré-determinado.[1]

- **Algoritmicamente**: Vários algoritmos foram propostos para esta etapa de pré-alinhamento. A idéia básica é utilizar casamento de padrões presentes nas duas vistas de forma a obter uma estimativa da transformação. Salvi [127] compara os principais algoritmos, a saber:
  - *Point Signature*: Chua [30] sugere um descritor para pontos, de forma que possa ser utilizado para a busca de correspondências;
  - *Spin Image*: É uma imagem 2D que caracteriza um ponto em uma superfície [78]. Através da similaridade das imagens, correspondências entre pontos podem ser obtidas;
  - *Principal Component Analysis*: Usa a direção do eixo principal da nuvem de pontos para alinhar as duas imagens de profundidade [31, 83];

[1]Imagem de http://www.architecture.yale.edu/dmonline/Equipment/3d-laserscanner/turntable.html.

- *RANSAC-based DARCES*: Utiliza o algoritmo robusto RANSAC para encontrar a melhor correspondência de 3 pontos entre as vistas, de forma a estimar a matriz de transformação [25];
- *Modelo Algébrico da Superfície*: Estima a transformações entre superfícies representadas por modelos polinomiais. As nuvens de pontos são convertidas em modelos polinomiais para a aplicação deste método [144];
- *Algoritmo Baseado em Linhas*: Utiliza linhas ou curvas para encontrar pares de correspondências [142, 159];
- *Algoritmos Genéticos*: Utiliza algoritmos genéticos para encontrar as correspondências entre as vistas [21];
- *Principal Curvature*: Utiliza as curvaturas principais de pontos da superfície para estimar a transformação das vistas [48];
- *4-Point Congruent Sets*: Utiliza um método robusto similar ao RANSAC, mas usando 4 pontos coplanares para definir candidatos, reduzindo o número de combinações e aumentando a precisão [1].

Infelizmente, a análise destes algoritmos mostra que muitos deles não são confiáveis, no sentido de sempre fornecerem uma solução aceitável [127]. Em alguns casos, o pré-alinhamento obtido pode ficar muito longe do correto, o que limita severamente o uso destes algoritmos na prática, pois a cada pré-alinhamento uma supervisão manual dos resultados é necessária.

Na maioria dos *softwares* comerciais, o pré-alinhamento acaba sendo feito manualmente, ou deduzido com o auxílio de uma *turntable* (figura 4.2). O problema das *turntables* é que a parte inferior do objeto não é capturada, e em objetos complexos, muitas regiões de oclusão também acabam não sendo capturadas. E se o usuário manualmente reposiciona o objeto (para capturar os dados faltantes), as vistas adquiridas têm que ser alinhadas manualmente, contrariando o propósito inicial do uso de *turntables*.

#### 4.1.1.2 Alinhamento Propriamente Dito

Dada uma estimativa inicial do alinhamento entre duas vistas, esta sub-etapa visa refinar esta estimativa (normalmente de forma iterativa), aumentando a sua precisão. Este refinamento é feito minimizando-se uma função que avalia a distância entre as duas superfícies sendo alinhadas.

O algoritmo mais utilizado é o ICP (*Iterative Closest Point*), através de suas inúmeras variantes. Ele foi originalmente proposto por Besl e McKay [17], e em outro trabalho

independente por Chen e Medioni [26]. Vários pesquisadores contribuíram com melhorias ao algoritmo básico, entre eles podemos citar [18, 55, 98, 139, 103, 113, 150, 59, 63, 80, 134, 164]. Detalhes sobre estas contribuições específicas podem ser encontrados nos *surveys* de Salvi [127] e Rusinkiewicz [122].

A idéia original do ICP (que originou o seu nome) é selecionar pontos na primeira vista, e encontrar pontos correspondentes na segunda vista. Originalmente, o ponto correspondente escolhido era o ponto mais próximo na segunda vista. A distância entre os pontos selecionados era avaliada, e um processo de minimização (normalmente por mínimos quadrados) calculava a transformação necessária para alinhar os pontos correspondentes. Esta transformação era combinada com a estimativa anterior, e todo o processo repetido até a sua convergência. Este algoritmo será detalhado na seção 4.2, pois foi o algoritmo utilizado em nosso *pipeline* de reconstrução 3D.

Uma alternativa ao ICP é o uso do *Simulated Annealing*, como sugerido por Blais [18]. Este é um método de busca estocástico, que possui a vantagem de não necessitar da derivada da função de distância, que pode inclusive ser não-linear. Infelizmente, é um algoritmo mais lento que o ICP, pois costuma requerer um número maior de iterações para sua convergência. Técnicas utilizadas para acelerá-lo normalmente requerem *subsampling* das vistas, o que pode reduzir a precisão do alinhamento.

Outra possibilidade é o uso de algoritmos genéticos, como os sugeridos por Chow [29] e por Silva *et al.* [138]. A idéia é evoluir um cromossomo que codifica os 6 parâmetros da transformação que alinha as vistas (3 ângulos de rotação e 3 medidas de translação). Uma das grandes vantagens destes métodos é a sua robustez à *outliers* e a capacidade de escapar de mínimos locais. Seu grande problema é o tempo necessário para convergência, pois a função de *fitness* requer muitas avaliações da função de distância entre as vistas.

#### 4.1.1.3 Alinhamento Global

Na sub-etapa anterior (4.1.1.2), pares de vistas são alinhados entre si, ou seja, para cada par de vistas que possua uma região de sobreposição suficiente, é calculada uma matriz de transformação que relaciona a segunda vista com a primeira.

Poderíamos imaginar que isto fosse suficiente para alinhar todas as vistas em um sistema comum de coordenadas, mas ocorre um problema: o acúmulo de erros. Imaginemos 20 vistas, tiradas em torno de um objeto. Alinharíamos a 1ª com a 2ª, a 2ª com a 3ª, e assim por diante, até alinharmos a 19ª com a 20ª. No entanto, se assim procedermos, a 1ª e a 20ª vistas estarão desalinhadas, pois pequenos erros de alinhamento entre cada vista são somados 19 vezes [113]. Para distribuir este erro uniformemente, é utilizada a terceira

sub-etapa de alinhamento, que é o alinhamento global. Este processo será apresentado em detalhes no capítulo 5.

### 4.1.2 Alinhamento Simultâneo de Todas as Vistas

Outra abordagem possível é procurar o alinhamento de todas as vistas simultaneamente. Desta forma, o processo de alinhamento como um todo passa a ser global, resolvendo o problema de acúmulo de erros apresentado na seção 4.1.1.3.

Um destes algoritmos foi sugerido por Blais [18]. Em seu trabalho, ele apresenta tanto a solução com alinhamento por pares quanto a solução simultânea, ambas utilizando *Simulated Annealing*. Como era de se esperar, o método simultâneo apresenta melhores resultados, mas um tempo de execução proibitivo, principalmente devido ao excessivo número de dimensões, que inviabilizam o algoritmo *Simulated Annealing*.

Outra abordagem foi sugerida por Masuda [96, 97]. Este método é baseado no *matching* de SDFs (*Signed Distance Fields*), e é robusto em relação a *outliers*. O método requer que as vistas estejam pré-alinhadas, logo os métodos apresentados na seção 4.1.1.1 ainda são necessários neste algoritmo. Uma das vantagens do método é que ele já nos fornece um modelo final integrado, que pode ser extraído diretamente do SDF resultante do algoritmo. Uma desvantagem do método é o gasto excessivo de memória e tempo de processamento, pois SDFs devem ser gerados e armazenados para cada vista durante a execução do algoritmo.

O método utilizando algoritmos genéticos de Silva *et al.* [138] também pode ser aplicado a todas as vistas de forma simultânea. Ele propôs uma nova medida de erro de alinhamento, a SIM (*Surface Interpenetration Measure*) que é robusta, diferentemente da medida RMS normalmente utilizada por outros métodos. Como citado na seção 4.1.1.2, a desvantagem é o elevado custo computacional deste algoritmo, que torna-se ainda maior no caso do alinhamento simultâneo de todas as vistas.

## 4.2 Solução Adotada

Conforme apresentado na seção 4.1, a etapa de alinhamento do *pipeline* de reconstrução 3D possui diversas alternativas para sua implementação. Em nosso trabalho, acabamos optando pelo alinhamento por pares de vistas, utilizando o algoritmo ICP. Como nosso primeiro objetivo era produzir uma solução que funcionasse, e como o ICP já havia sido utilizado com sucesso em outros trabalhos de preservação digital [89, 122], optamos por sua utilização. Outro fator que pesou a seu favor em nossa escolha foi a grande quantidade

de trabalhos já desenvolvidos sobre o algoritmo (vide seção 4.1.1.2), o que mostra que ele já havia atingido uma "maturidade" apropriada aos nossos propósitos. Este alinhamento por pares é refinado pela etapa de alinhamento global, que será descrita no capítulo 5.

### 4.2.1 Algoritmo ICP

O ICP pode ser descrito através do pseudo-código 4.1. Podemos visualizar o algoritmo da seguinte forma: inicialmente, os pontos correspondentes estão errados, mas com a aplicação do algoritmo, as correspondências vão ficando cada vez melhores, de forma que o alinhamento correto vai ficando cada vez mais próximo.

**Algoritmo 4.1** Pseudo-código do algoritmo ICP.

1: Inicialize a matriz de transformação da segunda para a primeira vista com um pré-alinhamento inicial
2: **repita**
3: Selecione pontos em ambas as vistas, encontrando para cada ponto um ponto correspondente na outra vista. Normalmente utiliza-se o ponto mais próximo, que atenda a certos critérios de compatibilidade, como cor, normais, etc.
4: Rejeite (opcionalmente) alguns dos pares de pontos, analisando-os individualmente ou todos em conjunto, de forma a desconsiderar *outliers*
5: Calcule uma medida de erro de alinhamento, a partir dos pares de pontos. Existem duas grandes métricas que podem ser utilizadas, distâncias ponto-a-ponto [17] e distâncias ponto-a-plano [26]
6: Minimize a métrica de erro, calculando para isso uma matriz de transformação que movimenta a segunda vista em relação à primeira
7: Combine a matriz obtida no passo anterior com a matriz atual
8: **até** a convergência da métrica de erro a um valor estável

O artigo de Rusinkiewicz [122] apresenta uma análise bem detalhada de cada estágio deste algoritmo. No entanto, a análise dele é focada para o desempenho do algoritmo, enquanto nós estamos mais interessados na precisão do resultado final, logo algumas escolhas devem ser feitas de maneira diferente. Vamos comentar rapidamente sobre cada estágio, e que resultados obtivemos com nossos testes, que visavam o aumento da precisão.

#### 4.2.1.1 Inicialização

O primeiro passo corresponde ao pré-alinhamento inicial. Muitas vezes, podemos utilizar uma matriz identidade como matriz de transformação inicial, que o algoritmo é capaz de convergir. No entanto, para pontos de vista muito diferentes, torna-se necessário especificar uma matriz inicial. Em nosso *software* a transformação inicial pode ser especificada

diretamente (através de seqüências de rotações e translações), ou conseguida de forma gráfica interativa.

Ao invés do método tradicional de seleção de 3 pontos em cada vista [85], permitimos a manipulação interativa das vistas pelo usuário, que acaba girando e deslocando a 2ª vista de forma a alinhá-la com a 1ª vista. Na prática tal método mostrou-se o mais rápido e prático para a obtenção do pré-alinhamento.

Optamos pelo pré-alinhamento manual de cada par de vistas devido à falta de confiabilidade dos algoritmos de pré-alinhamento existentes (vide seção 4.1.1.1).

O que pode acontecer se o pré-alinhamento não for suficiente é a não-convergência do ICP, que poderia ficar preso em algum mínimo local. Isto é detectado interativamente pelo usuário em nosso *pipeline*, que pode interromper o ICP e melhorar o pré-alinhamento.

#### 4.2.1.2 Seleção de Pontos e Seus Correspondentes

O segundo passo consiste na seleção de pontos e seus correspondentes. Este é o estágio crítico do ICP em termos de desempenho, pois a maior parte do tempo de execução é gasta aqui. Esta etapa é composta de duas partes: a seleção de pontos, e o *matching* destes pontos com pontos correspondentes na outra vista. Conforme apresentado em Rusinkiewicz [122], o melhor método de seleção é o chamado *normal-space sampling*. A idéia é que devemos selecionar pontos com a maior variação de normais possíveis, pois eles representam detalhes da superfície que definem o alinhamento correto. Além disso, é demonstrado que selecionar pontos de ambas as vistas também melhora o processo de convergência. A implementação do *normal-space sampling* é feita separando-se os pontos de cada vista em grupos com normais similares, e escolhendo a mesma quantidade de pontos dentro de cada grupo. Nossos testes mostram que dentro de cada grupo, o melhor critério é o de seleção aleatória, pois isto causa uma certa “vibração” no processo de convergência, que ajuda o algoritmo a não ficar preso em mínimos locais. Devemos notar que apenas uma parte (por exemplo, 15%) dos pontos totais é selecionada, para acelerar este passo.

Gelfand [57] sugeriu um método diferente de seleção de pontos, que em teoria seria mais eficaz que o *normal-space sampling*. A idéia é simular a matriz de covariância usada para a minimização de erro (que será apresentada na seção 4.2.1.5), e selecionar pontos de forma a diminuir a variação dos autovalores da matriz de covariância. Na prática, constatamos que este algoritmo tem a desagradável tendência de dar preferência aos *outliers* da vista, o que faz com que o alinhamento não consiga convergir ou que a convergência se dê para mínimos locais. Assim sendo, o método de *normal-space sampling* é mais simples, mais

rápido e dá melhores resultados que o algoritmo de Gelfand.

Quanto ao *matching* dos pontos selecionados com pontos da outra vista, Rusinkiewicz [122] mostra que em termos de robustez de convergência o melhor algoritmo é o *closest-compatible point*, apesar dele ser um pouco mais lento em termos de tempo de execução. Como nosso principal objetivo é a qualidade do alinhamento, este foi o método que selecionamos. Ele funciona procurando-se na outra vista o ponto mais próximo do ponto selecionado na primeira vista, desde que ele atenda critérios de compatibilidade. Nosso critério de compatibilidade é o ângulo entre normais, que deve estar abaixo de um limiar (normalmente 30° oferece bons resultados). Isto ajuda na seleção de pontos que sejam verdadeiramente correspondentes.

Outro detalhe a ser lembrado é que para acelerar o processo de busca do ponto mais próximo, são utilizadas *kd-trees* [10]. É justamente esta busca de pontos mais próximos que fazem desta etapa o gargalo em termos de desempenho, pois mesmo utilizando-se *kd-trees* este processo é demorado. Uma possibilidade considerada foi dividir o espaço 3D com diagramas de Voronoi [67]. No entanto, o gasto excessivo de memória nos levou a descartar esta opção.

Devemos lembrar que pontos verdadeiramente correspondentes só existem na região de interseção entre as duas vistas. O problema é que normalmente não se sabe *a priori* qual é esta região de interseção. Este problema é tratado pelo próximo passo do algoritmo.

#### 4.2.1.3 Rejeição de Pares

O terceiro passo do algoritmo ICP consiste em rejeitar alguns dos pares, na esperança de se eliminar *outliers*, neste caso, correspondências incorretas. Nossos testes demonstraram que esta rejeição é fundamental em alguns casos, para se garantir a convergência do algoritmo. Chegamos ao cúmulo de descobrir casos onde a partir de uma posição em que as vistas estavam alinhadas, elas acabavam se afastando, ou seja, divergindo. Utilizamos o método de eliminação dos n% pares mais distantes, sugerido por Pulli [113]. Normalmente eliminamos 10% dos pares. Devemos perceber que este processo de rejeição tende a deixar a convergência um pouco mais lenta.

Esta etapa fornece robustez ao algoritmo ICP, de forma a superar os diversos defeitos apresentados nas imagens de profundidade, conforme apresentado na seção 3.3.

Convém notar que *matchings* com pontos que se localizem na borda de quaisquer das duas vistas acabam sempre descartando o par. Conforme explicado em Rusinkiewicz [122], isto ajuda a selecionar pares apenas na região de interseção das duas vistas. Isto pode ser melhor compreendido através da figura 4.3.

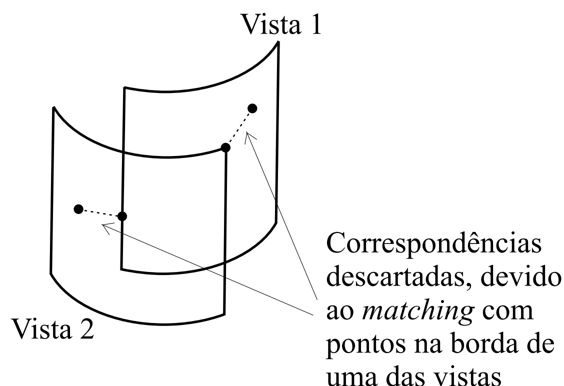



**Figura 4.3:** Descarte de pares com pontos na borda de vistas. Quando o ponto mais próximo se localiza na borda da outra vista, isto quase sempre implica que o primeiro ponto não pertence à região de interseção das vistas, logo o par não deve ser considerado.

#### 4.2.1.4 Métrica de Erro

O quarto passo consiste em calcular uma métrica de erro de alinhamento. Existem duas variantes usuais: somatória do quadrado das distâncias ponto-a-ponto [17], e somatória do quadrado das distâncias ponto-a-plano [26]. Podemos visualizar estas métricas na figura 4.4.

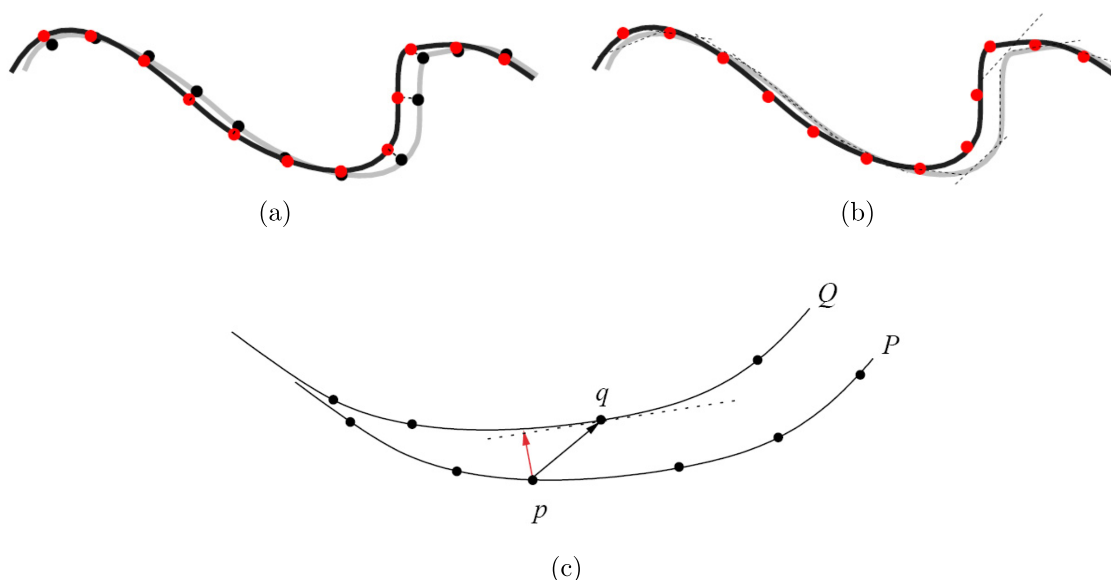



**Figura 4.4:** Tipos de métrica de erro. (a) Métrica ponto-a-ponto; (b) Métrica ponto-a-plano; (c) Detalhe de como a métrica ponto-a-plano é calculada. Dado um ponto $p$ pertencente à vista $P$, calcula-se o ponto mais próximo $q$ pertencente à vista $Q$. A métrica é a distância de $p$ ao plano tangente a $q$ (seta vermelha), plano que é obtido através da normal de $q$.

Na figura 4.4(a), vemos que cada ponto possui o seu correspondente, e a métrica soma o quadrado das distâncias ponto-a-ponto. Na figura 4.4(b), para cada ponto na

segunda vista (pretos), é criado um plano tangente ao ponto, e a métrica consiste na soma dos quadrados das distâncias de cada ponto vermelho ao plano de seu ponto preto correspondente. O plano tangente é obtido utilizando-se as normais dos pontos pretos. Isto foi detalhado na figura 4.4(c).

Rusinkiewicz [122] demonstra que esta métrica de ponto-a-plano, proposta por Chen e Medioni [26], possui uma convergência muito mais rápida e robusta. Isto é fácil de ser compreendido, pois o método permite que as superfícies "deslizem" sem penalidade dentro do plano tangente. Isto acrescenta um grau de liberdade ao processo de convergência. Além disso, existe um outro fato que recomenda esta métrica: como as vistas são tiradas de pontos de vista diferentes, na realidade não temos pontos exatamente coincidentes. Logo, a métrica ponto-a-ponto acaba ficando presa a pontos que na realidade não são verdadeiramente correspondentes. Já a métrica ponto-a-plano tenta aproximar a superfície na redondeza do ponto correspondente com planos, o que é uma aproximação mais aceitável, dada a grande densidade de pontos nas vistas. Isto facilita inclusive o alinhamento de vistas onde o espaçamento entre *samples* seja muito diferente.

#### 4.2.1.5 Minimização do Erro e Atualização da Transformação

O quinto passo consiste em descobrir a matriz de transformação que minimiza a métrica de erro. Normalmente são utilizados métodos de mínimos quadrados para descobrir esta matriz. Na métrica ponto-a-ponto, existe uma solução analítica para a minimização, que é apresentada no artigo de Horn [72]. Já para a métrica ponto-a-plano não existe uma solução fechada, pois se trata de um problema de minimização não-linear. Uma solução é "linearizar" o problema, assumindo que as rotações sejam pequenas. A matemática por trás desta linearização foi introduzida por Chen e Medioni [26], e explicada com maior nível de detalhes no artigo de Gelfand [57]. Apresentaremos a seguir este método.

Inicialmente, vamos apresentar a notação utilizada nas fórmulas. As duas vistas são chamadas de $P$ e $Q$. Para cada vista, temos uma matriz de transformação atual que transforma os pontos do sistema de coordenadas local para o sistema de coordenadas global. São estas matrizes que estamos tentando refinar com o processo de alinhamento. Estas matrizes homogêneas $4 \times 4$ serão representadas por $M_{P \rightarrow World}$ e $M_{Q \rightarrow World}$. No ICP, será a vista $P$ que se moverá para se alinhar com a vista $Q$. Desta forma, nosso objetivo é calcular uma nova matriz $M_{P \rightarrow World}$. Estas matrizes são compostas por rotações nos eixos X, Y e Z e translações nos eixos X, Y e Z. Logo, temos 6 incógnitas a serem calculadas, a partir dos pares de pontos selecionados. Convém ressaltar que em nossa notação utilizamos vetores linha e não coluna, logo para transformar um ponto $A$ por uma matriz $M$, devemos fazer $A' = A * M$.

O que queremos fazer é transformar os pontos de $P$ para o sistema de coordenadas de $Q$, e minimizar a distância destes pontos de $P$ para os planos que passam pelos pontos correspondentes em $Q$. Estes planos são perpendiculares às normais dos pontos em $Q$. Depois que calculamos a transformação que minimiza estas distâncias, esta transformação deve ser aplicada na matriz atual $M_{P \to World}$, atualizando-a. Desta forma, a matriz $M_{Q \to World}$ acaba não sendo modificada pelo processo.

Para converter os pontos do sistema de $P$ para o sistema de $Q$, utilizamos a matriz:

$$M_{P \to Q} = M_{P \to World} * (M_{Q \to World})^{-1} \tag{4.1}$$

Para facilitar o cálculo das rotações, os pontos de $P$ já transformados para o espaço de $Q$ são transladados para a origem. Para isso, calculamos o centro de massa destes pontos transformados, com a fórmula 4.2:

$$CM = \frac{\sum_{i=1}^{n} (P_i * M_{P \to Q})}{n} \tag{4.2}$$

onde $n$ representa o número de pares de pontos e $P_i$ o ponto de $P$ no par $i$.

A matriz 4×4 $TM$ que representa a transformação que alinha $P$ com $Q$ consiste, então, em transladar os pontos para a origem (subtraindo o centro de massa), rotacioná-los nos eixos X, Y e Z, transladá-los nos eixos X, Y e Z, e em seguida compensar a translação para a origem (somando o centro de massa), conforme a equação 4.3:

$$\begin{aligned} TM = {} & \text{Translação}(-CM) * \text{RotaçãoZ}(R_Z) * \text{RotaçãoY}(R_Y) * \\ & \text{RotaçãoX}(R_X) * \text{Translação}(T_X, T_Y, T_Z) * \text{Translação}(CM) \end{aligned} \tag{4.3}$$

Nesta fórmula, Translação() cria uma matriz $4 \times 4$ de translação, RotaçãoX() uma matriz de rotação em torno do eixo X, e assim por diante. Devemos notar que esta matriz $TM$ opera no sistema de coordenadas de $Q$ logo, para atualizar $M_{P \to World}$, usamos a fórmula 4.4:

$$M_{P \to World}' = M_{P \to World} * (M_{Q \to World})^{-1} * TM * M_{Q \to World} \tag{4.4}$$

Agora que já sabemos como a matriz de $P$ é atualizada, precisamos apresentar o algoritmo para calcular as 6 incógnitas (escalares), que são $R_X$, $R_Y$, $R_Z$, $T_X$, $T_Y$ ,$T_Z$.

Estes valores são obtidos resolvendo-se um sistema $6 \times 6$, que calcula a solução por mínimos quadrados. O sistema possui a forma $AX = B$, que pode ser resolvido pelo algoritmo de Choleski [112], uma vez que a matriz de covariância $A$ é simétrica e positiva definida. O formato da matriz $A$ e dos vetores coluna $X$ e $B$, bem como o de algumas variáveis auxiliares é o seguinte:

$$dist_i = (Q_i - (P_i * M_{P \to Q})) \cdot NQ_i \tag{4.5}$$

onde $dist_i$ é um escalar que corresponde à distância do ponto $P_i$ ao plano do ponto $Q_i$. Note-se que para isto foi necessária a normal $NQ_i$ do ponto $Q_i$, e a operação de produto interno entre vetores.

$$TR_i = ((P_i * M_{P \to Q}) - CM) \times NQ_i \tag{4.6}$$

onde $TR_i$ é um vetor que representa o "torque" (força de rotação) para o par $i$. Para calculá-lo utilizou-se a operação de produto vetorial.

$$V_i = \begin{bmatrix} TR_{i_x} \\ TR_{i_y} \\ TR_{i_z} \\ NQ_{i_x} \\ NQ_{i_y} \\ NQ_{i_z} \end{bmatrix} \tag{4.7}$$

onde $V_i$ é um vetor $6 \times 1$ composto pelas três componentes do torque $TR_i$ e pelas três componentes da normal $NQ_i$, e é utilizado para criar a matriz $6 \times 6$ de covariância $A$ e o vetor $6 \times 1$ de residuais $B$.

$$A = \sum_{i=1}^{n} \begin{bmatrix} V_{i_0} * V_{i_0} & V_{i_0} * V_{i_1} & V_{i_0} * V_{i_2} & V_{i_0} * V_{i_3} & V_{i_0} * V_{i_4} & V_{i_0} * V_{i_5} \\ V_{i_1} * V_{i_0} & V_{i_1} * V_{i_1} & V_{i_1} * V_{i_2} & V_{i_1} * V_{i_3} & V_{i_1} * V_{i_4} & V_{i_1} * V_{i_5} \\ V_{i_2} * V_{i_0} & V_{i_2} * V_{i_1} & V_{i_2} * V_{i_2} & V_{i_2} * V_{i_3} & V_{i_2} * V_{i_4} & V_{i_2} * V_{i_5} \\ V_{i_3} * V_{i_0} & V_{i_3} * V_{i_1} & V_{i_3} * V_{i_2} & V_{i_3} * V_{i_3} & V_{i_3} * V_{i_4} & V_{i_3} * V_{i_5} \\ V_{i_4} * V_{i_0} & V_{i_4} * V_{i_1} & V_{i_4} * V_{i_2} & V_{i_4} * V_{i_3} & V_{i_4} * V_{i_4} & V_{i_4} * V_{i_5} \\ V_{i_5} * V_{i_0} & V_{i_5} * V_{i_1} & V_{i_5} * V_{i_2} & V_{i_5} * V_{i_3} & V_{i_5} * V_{i_4} & V_{i_5} * V_{i_5} \end{bmatrix} \tag{4.8}$$

onde $A$ é a matriz $6 \times 6$ de covariância. Note-se que a matriz $A$ é composta por multiplicações das componentes do vetor $V_i$, numeradas de 0 a 5.

$$B = \sum_{i=1}^{n} dist_i * V_i \tag{4.9}$$

onde $B$ é o vetor $6 \times 1$ de residuais.

$$X = \begin{bmatrix} R_x \\ R_y \\ R_z \\ T_x \\ T_y \\ T_z \end{bmatrix} \tag{4.10}$$

onde $X$ é o vetor de incógnitas sendo procurado, obtido resolvendo-se o sistema $AX = B$ pelo algoritmo de Choleski.

Resumindo, devemos construir a matriz $A$ (Eq. 4.8) e o vetor $B$ (Eq. 4.9), calculando o vetor de incógnitas $X$. Em seguida, construímos a transformação $TM$ (Eq. 4.3), e finalmente atualizamos a matriz $M_{P \to World}$ (Eq. 4.4).

Como este método é apenas uma aproximação linear, necessitamos de várias iterações do algoritmo ICP para a métrica de erro convergir. Isto não ocasiona nenhum problema, pois o ICP já necessitava de diversas iterações de forma que as correspondências entre os pontos fosse cada vez mais correta.

### 4.2.2 Melhoria da Precisão do ICP

Nossa implementação das etapas apresentadas anteriormente (seção 4.2.1) funciona de maneira satisfatória. No entanto, com a nossa busca da máxima precisão possível, descobrimos que o ideal seria ter duas fases no ICP: a primeira para se chegar do pré-alinhamento para um alinhamento quase final; e a segunda para aperfeiçoar o alinhamento da etapa anterior, tentando ignorar os *outliers* de forma mais estrita.

A primeira fase funciona utilizando-se os critérios explicados anteriormente, destacando-se principalmente a seleção de apenas uma parte dos pontos totais com o critério de *normal-space sampling*. Para cada iteração, calculamos uma porcentagem de *matching* entre as vistas. Esta porcentagem é calculada verificando quantos pares de pontos possuem distâncias menores que um limiar especificado, em relação ao total de pares. Este limiar é dependente da precisão do *scanner*. Verificamos que no início do alinhamento, enquanto as vistas ainda estão desalinhadas, o percentual de *matching* é muito baixo. Quando o algoritmo converge para o alinhamento quase final, o percentual

de *matching* sobe, normalmente atingindo taxas acima de 80%. Verificamos que a partir deste ponto, a segunda vista fica "vibrando" em relação à primeira, mas sem fugir do alinhamento obtido. Isto ocorre por causa da seleção aleatória de pontos dentro de cada grupo de normais. Como os pontos selecionados são diferentes, as soluções de iterações sucessivas são ligeiramente diferentes, o que causa a vibração. Além disso, percebemos que *outliers* não descartados colaboram com esta vibração.

Esta vibração acaba sendo muito útil nas primeiras iterações do algoritmo, quando ele fica relativamente preso a mínimos locais. Após algumas iterações "vibrando", normalmente o algoritmo consegue se libertar do mínimo local e tender ao mínimo global. Logo, apesar de parecer um paradoxo, no início do algoritmo a existência de alguns "outliers" tende a ser benéfica, para garantir a convergência através da vibração.

No entanto, para obtermos um alinhamento final preciso, queremos levar em conta apenas *inliers*. Conseguimos isso criando uma segunda fase do ICP. Entramos na segunda fase quando o percentual de *matching* atinge um limiar pré-definido durante um certo número de iterações sucessivas. Este critério provou-se inteligente para detectar o mínimo global. Normalmente utilizamos 3 iterações com matching acima de 80%. A partir deste momento, nosso critério de seleção de pontos é modificado: passamos a utilizar todos os pontos, cujos correspondentes estejam abaixo do limiar de erro do *scanner* (limiar este que é o utilizado para o cálculo do percentual de *matching*). Desta forma, pontos muito distantes são considerados *outliers*, e são descartados. Com este procedimento, conseguimos um comportamento robusto, similar ao MSAC [148]. Por incrível que pareça, apesar de utilizarmos todos os pontos, esta segunda fase é muito mais rápida que a primeira, pois as buscas de pontos mais próximos nas *kd-trees* são limitadas pelo limiar de erro, o que tornam estas buscas muito rápidas. Continuamos na segunda fase até que a métrica de erro se estabilize, o que costuma ocorrer em poucas iterações.

Este algoritmo composto tem se comportado muito bem na prática. Os alinhamentos têm um bom grau de convergência, mesmo para pré-alinhamentos ruins, e a precisão do alinhamento final é muito boa. Os resultados obtidos serão apresentados na seção 4.4.

## 4.3 Detalhes de Implementação

Existem vários passos do algoritmo de alinhamento, cada um dos quais requer algoritmos e estruturas de dados específicas para serem eficientemente implementados. No entanto, antes de analisar os detalhes da etapa de alinhamento, convém entender a arquitetura geral do nosso software de reconstrução digital.

### 4.3.1 Arquitetura da Ferramenta IMAGO *ModelTool*

A nossa implementação do pipeline de reconstrução 3D foi batizada de IMAGO *ModelTool*. O *ModelTool* é uma ferramenta que utiliza OpenGL como API gráfica, e foi desenvolvido com a biblioteca GLUT de forma a ser independente de plataforma. A ferramenta funciona tanto em ambiente Windows quanto Linux. A desvantangem desta abordagem é que a *interface* do programa não é muito amigável, sendo baseada em teclas de controle e uso do *mouse*. Uma tela do programa pode ser vista na figura 4.5.

Além dos arquivos .RNG e .TGA das imagens de profundidade (obtidas através da ferramenta *VividCtrl*, apresentada no capítulo 3), o *ModelTool* requer um arquivo de *script* .SC como parâmetro de entrada. Este é um arquivo texto, no qual as imagens de profundidade são referenciadas, bem como os pares de vistas (podendo incluir algum pré-alinhamento inicial), e finalmente uma lista de todos os parâmetros e limiares utilizados nas diversas etapas do *pipeline* de reconstrução 3D.

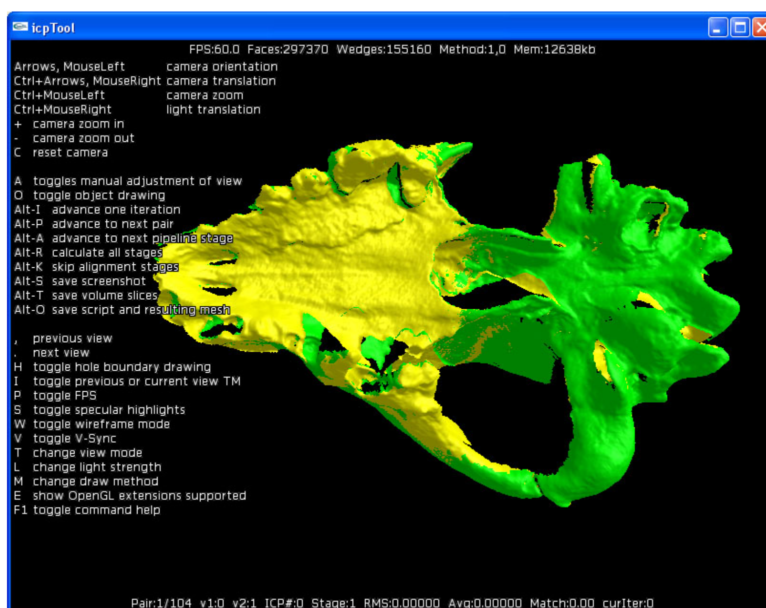



**Figura 4.5:** Ferramenta *ModelTool*. Podemos observar o modo *help* ativado, onde todos os comandos aceitos pela ferramenta estão sendo exibidos na tela.

Quando executado, o *ModelTool* inicia apresentando o primeiro par a ser alinhado. O usuário pode manualmente pré-alinhar a segunda vista em relação à primeira (utilizando o *mouse*), e em seguida executar o ICP para o par. O alinhamento pode ser executado iteração a iteração (Alt-I), ou todo de uma vez para o par (Alt-P). Este processo repete-se para todos os pares especificados no arquivo de *script*.

Um dos grandes problemas que a ferramenta *ModelTool* teve que resolver foi o gasto excessivo de memória das imagens de profundidade e suas estruturas de dados associadas.

Por exemplo, vários de nossos objetos de teste requerem mais de 3 GB para o armazenamento destes dados. Isto impede o armazenamento simultâneo de todos estes dados na RAM, pelo menos em sistemas operacionais de 32 bits, pois o espaço de endereçamento é insuficiente, mesmo com o uso de memória virtual.

Resolvemos este problema criando um mecanismo de *cache*. Quando uma imagem de profundidade é lida pela primeira vez, todas as estruturas de dados auxiliares (que serão apresentadas no decorrer deste trabalho) são criadas e gravadas em um arquivo temporário no disco, junto com as imagens de profundidade originais. O arquivo de *script* estipula uma quantidade de memória RAM que será usada como *cache* destes dados. Desta forma, conforme o programa *ModelTool* vai necessitando de dados das vistas, os dados necessários são carregados para a RAM de forma a serem utilizados. Obviamente, quanto maior o tamanho do *cache*, melhor o desempenho do *ModelTool*.

Uma vez apresentadas estas considerações gerais sobre o *ModelTool*, podemos agora nos concentrar nos detalhes da etapa de alinhamento por pares propriamente dita.

### 4.3.2 Detalhes da Etapa de Alinhamento por Pares

Inicialmente, devemos nos lembrar que o algoritmo ICP trabalha com modelos, e não com pontos isolados. Logo, para cada vista temos um modelo 3D, criado a partir dos pontos 3D retornados pelo *scanner*. Estes modelos serão usados subseqüentemente em outras etapas do *pipeline* de reconstrução. Para construir estes modelos, devemos triangular os pontos 3D, conforme o que já foi apresentado na seção 3.3.4 e na figura 3.12. Além disso, normais devem ser estimadas para cada vértice do modelo, como também já foi discutido na seção 3.4.

Conforme apresentado na seção 4.2.1.2, efetuamos o processo de seleção de pontos com a técnica de *normal-space sampling*. Para implementá-la, temos que tentar dividir o espaço de direções 3D em grupos, e em seguida dividir os vértices entre os diversos grupos. A técnica que usamos foi a mesma empregada na ferramenta *VividCtrl* durante a detecção do plano de apoio (vide seção 3.4.1 e figura 3.17). Esta técnica é inspirada no algoritmo de *cube-mapping* utilizado em texturização [62]. Imaginamos que o vértice está no centro de um cubo, e que a normal acaba apontando para uma das 6 faces. Cada face é dividida em blocos (*buckets*). A partir das componentes $(x, y, z)$ da normal, temos como descobrir a face e o bloco dentro da face que a normal aponta.

Desta forma, temos como separar os vários vértices em grupos, baseados na direção de suas normais. Em nossa implementação, dividimos cada face do cubo em um *grid* $5 \times 5$, gerando um total de $25 * 6 = 150$ grupos de normais. Durante o processo de

seleção, verificamos quantos grupos foram utilizados pela vista, e dividimos o número total de pontos a selecionar pelo número de grupos, para saber quantos vértices sortear em cada grupo. Obviamente, se um grupo possuir poucos vértices, todos eles acabam sendo utilizados.

Para encontrar os pontos correspondentes na outra vista de forma eficiente, são necessárias *kd-trees* [10]. A idéia é que cada vista possui a sua *kd-tree*, que foi construída no sistema de coordenadas local. Isto é necessário para que quando a vista seja reposicionada, a *kd-tree* não tenha que ser recalculada. Logo, antes de acessar a *kd-tree*, o ponto do qual se quer descobrir o vértice mais próximo deve ser transformado para o sistema de coordenadas local da vista, o que é feito através da sua multiplicação pela matriz de transformação atual da vista.

Outro fato que descobrimos é que o ideal é criar uma *kd-tree* tridimensional, baseada nas coordenadas $(x, y, z)$ dos vértices. Experimentamos com *kd-trees* em 6D, por causa do critério de compatibilidade de normais que existe durante o processo de busca. No entanto, a conclusão que chegamos é que em 6D acabamos visitando muito mais nodos, pois normalmente a região de abrangência de normais compatíveis é muito grande.

Para efetuar a rejeição de pares, devemos ordená-los por distância entre os dois pontos de cada par. Esta distância é obtida durante a busca na *kd-tree*, e é salva para este uso posterior. Aqui podemos usar uma variante do *quicksort*, pois não precisamos ordenar os valores na realidade, o que precisamos é separar os $n\%$ pares mais distantes. Logo, na implementação do *quicksort*, se o intervalo a ser ordenado não incluir o ponto de corte, não precisamos efetuar a recursão. Na prática esta otimização não faz muita diferença no tempo de execução, pois o gargalo de desempenho é devido às buscas nas *kd-trees*.

Diversos parâmetros são utilizados para controlar o algoritmo ICP. Estes parâmetros são especificados no arquivo .SC de *script* utilizado como entrada para a ferramenta *ModelTool*. Na tabela 4.1 podemos observar quais são estes parâmetros, uma breve descrição e os valores usuais de cada um deles. Todos os parâmetros foram descobertos e validados empiricamente, através dos extensivos testes realizados.

**Tabela 4.1:** Parâmetros do algoritmo ICP especificado no arquivo *script* de entrada do programa *ModelTool*.

| Parâmetro | Valor Usual | Descrição |
|---|---|---|
| $miscViewCacheSize$ | 500 MB[1] | Tamanho do *cache* de vistas |
| $viewDiscardAngle$ | 75.0° | Ângulo para descarte de faces nas vistas[2] |
| $viewMinGroup$ | 0.1% | Limiar para descarte de grupos de faces nas vistas[3] |
| $icpScannerError$ | 0.70 mm | Distância para definição de *matching*[4] |
| $icpMatchPercent$ | 80.0% | Limiar de *matching* para iniciar 2º estágio do ICP[4] |
| $icpIterToStage2$ | 3 | Iterações com *matching* para iniciar 2º estágio do ICP[4] |
| $icpIterToStage3$ | 100 | Máximo de iterações permitidas no 2º estágio do ICP |
| $icpIterMax$ | 100 | Máximo de iterações permitidas em todo o ICP |
| $icpMaxNormalAngle1$ | 30.0° | Ângulo para compatibilidade de normais no 1º estágio[5] |
| $icpMaxNormalAngle2$ | 15.0° | Ângulo para compatibilidade de normais no 2º estágio |
| $icpSelectPercent$ | 15.0% | Percentagem dos pontos selecionados no 1º estágio[5] |
| $icpRejectPercent$ | 10.0% | Percentagem de pares rejeitados[6] |
| $icpStableToStage3$ | 10 | Iterações para detectar convergência no 2º estágio |
| $icpStableError$ | 0.0001 mm | Variação do erro RMS para detectar convergência[7] |
| $icpMaxMatchDist$ | 10.0 mm | Distância máxima entre pontos correspondentes de um par[8] |

## 4.4 Resultados Obtidos e Trabalhos Futuros

Na figura 4.6 podemos observar algumas imagens do processo de alinhamento. Podemos observar vários aspectos interessantes na figura 4.6. O processo de alinhamento consegue convergir, mesmo para pré-alinhamentos ruins. Além disso, poucas iterações são necessárias para a convergência do algoritmo. Outro detalhe interessante é que a região de sobreposição entre as imagens não é tão grande, principalmente a da primeira vista, representada em verde; e mesmo assim, isso não causou nenhuma dificuldade para o processo de alinhamento. Este comportamento foi apresentado em quase todos os objetos que reconstruímos com nosso *pipeline*.

[1]Valor usual para um sistema com 4GB de RAM.

[2]Vide seção 3.3.4 e figura 3.13.

[3]Percentagem do número total de faces. As faces da vista são agrupadas em regiões conexas, e regiões com menos faces que o limiar são descartadas (vide seção 3.3.2).

[4]Vide seção 4.2.2.

[5]Vide seção 4.2.1.2.

[6]Vide seção 4.2.1.3.

[7]Para detectar a convergência, o erro RMS de alinhamento deve variar menos que $icpStableError$ durante $icpStableToStage3$ iterações seguidas.

[8]Este valor limita as buscas na *kd-tree* pelo ponto correspondente. Desta forma, quanto mais desalinhadas as vistas, maior deve ser este parâmetro para que correspondentes sejam encontrados.

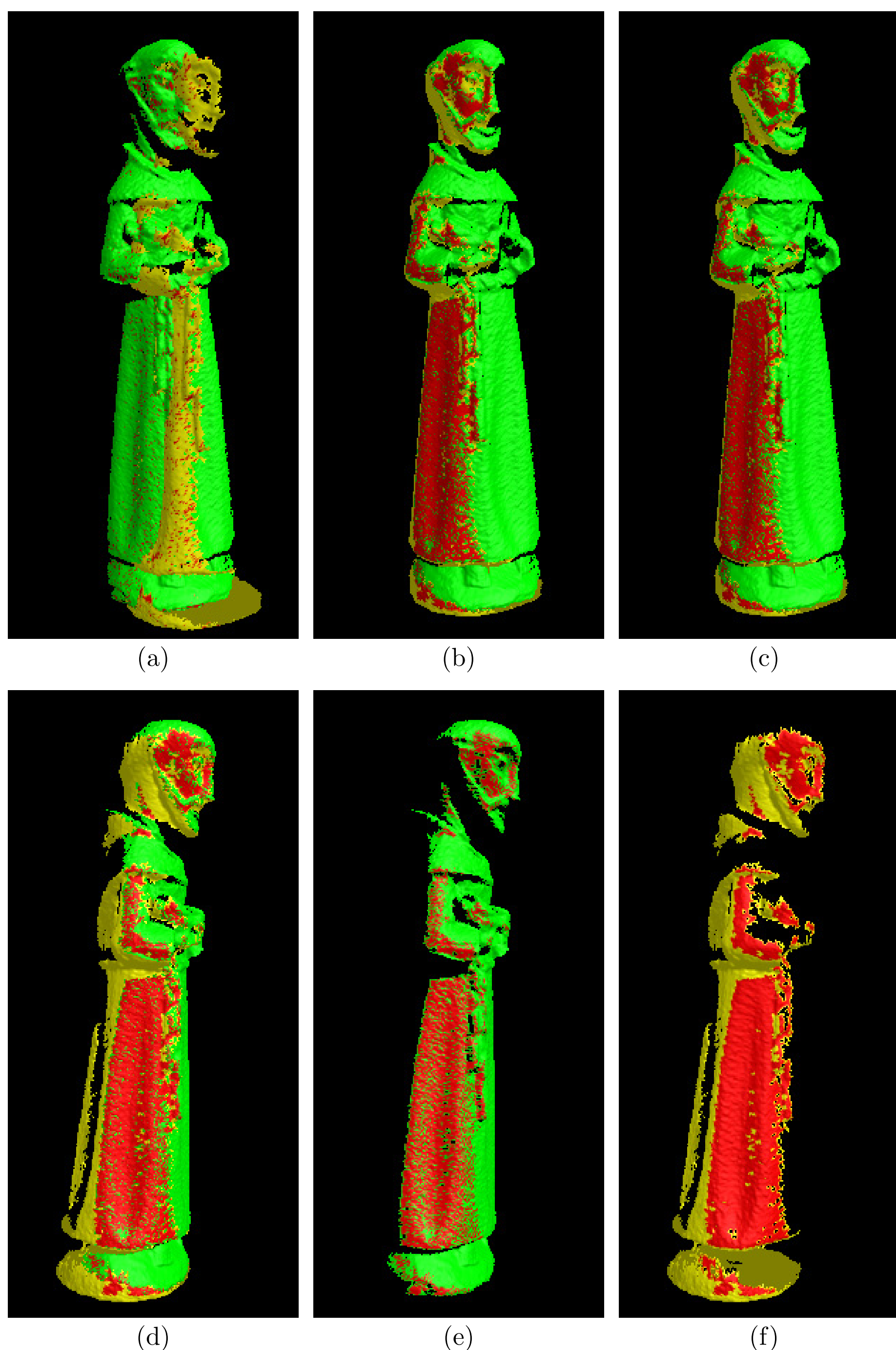



**Figura 4.6:** Processo de alinhamento. (a) mostra o pré-alinhamento inicial, que é relativamente ruim (29% de *matching*). Em (b) vemos a troca para a segunda fase do algoritmo, que ocorre na 6ª iteração, com um *matching* de 97%. Em (c) temos o alinhamento final, que por definição tem 100% de *matching*, e que foi obtido depois de 15 iterações. (d) a (f) mostram o alinhamento de outro ponto de vista, de forma a ressaltar a região de interseção entre as duas vistas. Os pontos em vermelho correspondem aos pares selecionados.

Apesar de já estar funcionando satisfatoriamente, temos algumas idéias que podem melhorar este processo de alinhamento por pares.

Conforme já foi sugerido por Godin [59], poderíamos utilizar as imagens de intensidade luminosa de forma a colorir cada vértice, e utilizar a cor dos vértices como mais um critério de compatibilidade do processo de seleção de pares. Isto ajudaria a convergência em casos onde a geometria do objeto é muito simples para determinar o alinhamento correto.

Outra área que poderia ser estudada é a melhoria do desempenho do algoritmo. O tempo de execução é basicamente dominado pelas buscas nas *kd-trees*, o que implica que melhorias neste processo são indispensáveis para melhorar o desempenho do algoritmo como um todo. Descobrimos que a busca na *kd-tree* possui um tempo muito variável. Quando o vértice mais próximo do ponto de busca é bem próximo (apenas alguns milímetros de distância, nos nossos casos), a busca retorna de maneira muito rápida. Isto ocorre na segunda fase de nosso algoritmo ICP, onde a distância máxima é limitada ao limiar de erro do *scanner*, que normalmente utilizamos como 0,7mm. Apesar de todos os pontos serem considerados, e de necessitar de um número maior de iterações, esta etapa do processo é executada muito mais rapidamente do que a primeira etapa, quando as vistas ainda estão muito desalinhadas. Isto ocorre porque quando o vértice mais próximo é muito distante, muitos nodos da *kd-tree* são visitados, o que reduz drasticamente o desempenho.

Conseguimos aperfeiçoar o desempenho limitando a distância máxima de busca (parâmetro $icpMaxMatchDist$ apresentado na tabela 4.1). A justificativa para isso é que quando um ponto correspondente é muito distante, provavelmente esta correspondência é incorreta de qualquer forma, sendo refinada nas iterações seguintes. Outra possibilidade em que as distâncias são grandes é quando o ponto mais próximo se encontra em uma borda. Como esta correspondência será descartada de qualquer forma, este caso também não gera problemas. Experiências iniciais mostram que o tempo de busca é bastante reduzido (de 2 a 4 vezes), mas em alguns casos a convergência para o mínimo global deixa de ocorrer, principalmente se as vistas iniciarem muito desalinhadas.

Finalmente, outra área a estudar seria a utilização de diferentes métricas de erro para avaliar o alinhamento das vistas, como a SIM (*Surface Interpenetration Measure*) sugerida por Silva *et al.* [138]. Isto seria interessante pois ajudaria a descartar de forma mais precisa os *outliers* durante o processo de alinhamento.

# CAPÍTULO 5

# ALINHAMENTO GLOBAL

Uma vez obtidos os alinhamentos de pares de vistas, a etapa seguinte do *pipeline* de reconstrução 3D consiste em minimizar o acúmulo de erros que acontece quando se utilizam apenas os resultados do alinhamento por pares.

Podemos imaginar o relacionamento entre as vistas como um grafo, onde cada vista corresponde a um vértice do grafo, e um alinhamento entre duas vistas como uma aresta do grafo, como apresentado na figura 5.1.

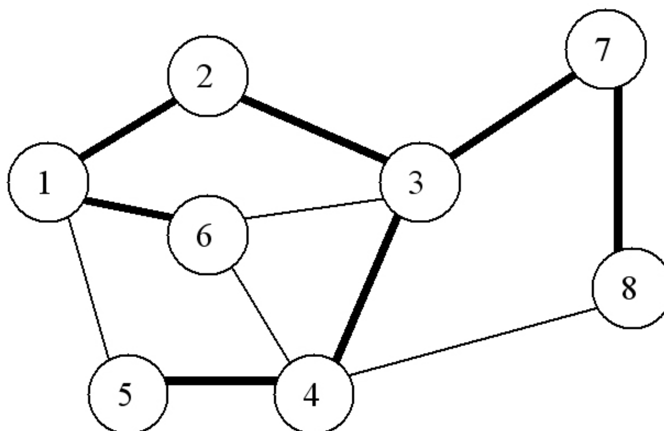
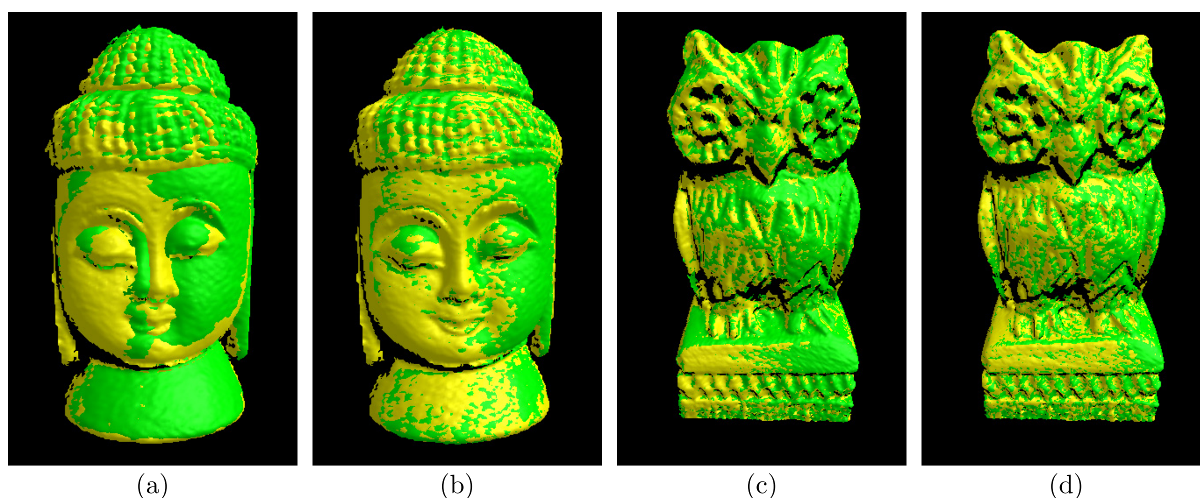



**Figura 5.1:** Relacionamento entre pares de vistas, representado como um grafo.

Como podemos perceber, existem ciclos no grafo. As arestas realçadas representam uma possível solução de alinhamento global, utilizando-se apenas os alinhamentos por pares. O que acontece é que os ciclos acabam sendo eliminados, e uma árvore é formada. Se uma árvore que visite todas as vistas (*spanning tree*) não puder ser formada, é porque faltam alinhamentos de pares capazes de "interligar" todas as vistas. Os alinhamentos de pares representados pelas linhas finas são simplesmente ignorados, pois no fundo o que temos é um sistema sobre-determinado, isto é, com mais equações do que incógnitas. Esta solução simplista é claramente problemática. Por exemplo, o relacionamento entre as vistas 1 e 5 é deduzido através da combinação da vista 1 com a 2, a 2 com a 3, a 3 com a 4 e a 4 com a 5. Como cada alinhamento sempre possui um erro (devido ao ruído), o relacionamento deduzido da vista 1 com a 5 acaba normalmente sendo muito diferente do relacionamento direto (aresta simples) calculado pelo ICP, que é ótimo localmente.

Podemos perceber este problema da falta do alinhamento global na figura 5.2. As imagens de profundidade do Buddha foram adquiridas por um *scanner Minolta Vivid 700*, e as da coruja por um *scanner Minolta Vivid 910*. Como o *Vivid 910* é muito mais preciso que o *Vivid 700*, os erros acumulados na coruja são bem menores que os erros acumulados no Buddha. Este é um ponto importante a ser destacado, ou seja, quanto pior a qualidade dos dados, mais importante se torna a etapa do alinhamento global.



(a) (b) (c) (d)

**Figura 5.2:** Exemplos de falta de alinhamento global. (a) Alinhamento da vista 17 com a vista 0 da estátua de Buddha, deduzida a partir dos alinhamentos de pares ($0 \rightarrow 1$, $1 \rightarrow 2, \ldots,$ $16 \rightarrow 17$). O erro acumulado é absurdamente grande; (b) Alinhamento da vista 17 com a vista 0, após o alinhamento global; (c) Alinhamento da vista 15 com a vista 0 da estátua de uma coruja, deduzida a partir dos alinhamentos de pares; (d) Alinhamento da vista 15 com a vista 0, após o alinhamento global.

## 5.1 Revisão Bibliográfica

Ainda não existe no meio científico um consenso sobre a melhor forma de resolver o problema do alinhamento global. Conforme apresentamos na seção 4.1.2, uma possibilidade é tentar alinhar todas as vistas de forma simultânea, sem uma etapa prévia de alinhamento de pares. O grande problema desta abordagem é a necessidade de estarem carregadas na memória todas as vistas simultaneamente; e isto rapidamente torna a solução impraticável, exceto para objetos simples com poucas vistas capturadas. Outra abordagem consiste em separar o problema local de alinhamento de vistas (resolvido com alinhamento por pares) do problema global de distribuição dos erros acumulados.

Podemos citar alguns trabalhos sobre o alinhamento global: Bergevin *et al.* [11] coloca todas as vistas em um único sistema de referência, e repetidamente seleciona uma vista, alinhando-a com todas as outras. Stoddart e Hilton [143] encontram correspondências en-

tre pares de pontos em todas as vistas, e iterativamente minimizam os erros imaginando "molas" entre os pares de pontos, e simulando o sistema dinâmico obtido até ele ficar estável. Este método de minimização também foi usado por Eggert *et al.* [47], no entanto em seu método, cada ponto participa apenas de um par, o que pode gerar alguns problemas, conforme apresentado no trabalho de Pulli [113]. Benjeema e Schmitt [9] propuseram uma solução não-linear utilizando quaternions, utilizando as várias vistas simultaneamente. Krishnan *et al.* [86] apresentam uma solução global no *manifold* de rotações, sem requerer alinhamentos de pares. Sharp *et al.* [135] tratam especificamente do problema de distribuição de erros, distribuindo os erros de translação e rotação entre os alinhamentos das vistas, representadas como um grafo (como apresentado na figura 5.1). Seu método trabalha apenas sobre as transformações, não levando em conta pontos amostrados nas vistas, o que torna difícil definir bons critérios para governar o processo de distribuição de erros, isto é, quanto "erro" colocar em cada aresta do grafo. Vieira *et al.* [153] propuseram iterações intercaladas de alinhamento e reconstrução, re-alinhando as vistas com o modelo reconstruído, e melhorando o alinhamento global.

Em nosso *pipeline*, utilizamos o algoritmo sugerido por Pulli [113]. Este algoritmo foi utilizado no *"The Digital Michelangelo Project"* [89], provando-se eficaz para manipular grandes quantidades de informações. Seu método será explicado em detalhes a seguir.

## 5.2 Solução Adotada

Nossa etapa de alinhamento global é dividida em duas sub-etapas: o alinhamento global propriamente dito, utilizando o algoritmo de Pulli [113], seguido do alinhamento do modelo com seus eixos principais.

### 5.2.1 Algoritmo de Pulli

O algoritmo de Pulli [113] possui o mesmo princípio de funcionamento do algoritmo de Sharp [135], isto é, tratar apenas do problema global de distribuição dos erros acumulados no grafo de vistas (vide figura 5.1). Desta forma, este método independe de como os diversos pares de vistas foram alinhados individualmente. É interessante notar que quanto mais relacionamentos entre as vistas tivermos (isto é, quanto mais pares) melhor deve ser o resultado global de alinhamento na prática, pois teremos mais informações de como "mover" as vistas para o seu local correto, de maneira a minimizar o erro global.

A idéia básica do algoritmo de Pulli é criar dois conjuntos de vistas, as ativas e as inativas. Inicialmente, uma vista vai para a lista de vistas ativas, e todas as outras para as inativas. Em seguida, vistas inativas vizinhas a vistas ativas são adicionadas uma a uma

à lista de vistas ativas. A cada vista inativa adicionada, o erro é difundido entre todas as vistas ativas até se estabilizar, quando uma nova vista inativa é adicionada. O processo é repetido até que todas as vistas inativas tenham sido ativadas. Podemos observar estes passos em detalhes no algoritmo 5.1.

**Algoritmo 5.1** Pseudo-código para o algoritmo de Alinhamento Global de Pulli [113].

1: $conjuntoInativas \leftarrow$ todas as vistas
2: $conjuntoAtivas \leftarrow \emptyset$
3: $fila \leftarrow \emptyset$
4: $atual \leftarrow$ maisArestas($conjuntoInativas, conjuntoInativas$)
5: $conjuntoAtivas$.adiciona($atual$)
6: $conjuntoInativas$.remove($atual$)
7: **enquanto** $conjuntoInativas \neq \emptyset$ **faça**
8: $atual \leftarrow$ maisArestas($conjuntoInativas, conjuntoAtivas$)
9: $conjuntoAtivas$.adiciona($atual$)
10: $conjuntoInativas$.remove($atual$)
11: $fila$.adiciona($atual$)
12: **enquanto** $fila \neq \emptyset$ **faça**
13: $atual \leftarrow fila$.remove()
14: $vizinhos \leftarrow conjuntoAtivas$.vizinhos($atual$)
15: $mudanca \leftarrow$ alinha($atual, vizinhos$)
16: **se** $mudanca > tolerancia$ **então**
17: $fila$.mescla($vizinhos$)
18: **fim se**
19: **fim enquanto**
20: **fim enquanto**

Para difundir o erro, Pulli move uma vista por vez, em relação às vistas vizinhas dela, tentando minimizar o erro global. Para isso, ele utiliza uma técnica muito interessante, que consiste em pares de pontos virtuais. A idéia é selecionar pontos na região de interseção de cada par de vistas. Ao invés de associar os pontos de uma vista com pontos da outra, o que se criam são pontos correspondentes virtuais, que ficam fixos com a vista que não está sendo movimentada. Isto pode ser melhor entendido na figura 5.3.

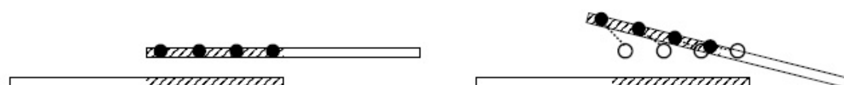


**Figura 5.3:** Pares virtuais conforme Pulli [113]. Os círculos brancos são os correspondentes virtuais dos círculos pretos, e correspondem à posição dos círculos pretos no sistema de coordenadas da vista vizinha. Note-se que estas correspondências são calculadas utilizando-se o alinhamento por pares entre as vistas.

Os pontos marcados com círculos vazados na figura 5.3 são os correspondentes virtuais, que inicialmente estavam na mesma posição dos pontos da vista que está sendo ajustada.

Agora imaginemos que a vista tenha vários vizinhos, logo existem vários correspondentes virtuais. Minimizando a distância entre todos estes pontos (utilizando a métrica ponto-a-ponto do ICP e a solução analítica por mínimos quadrados de Horn [72]), a vista se movimenta de forma a difundir o seu erro de alinhamento para todas as vistas vizinhas.

Uma das vantagens do uso de pares virtuais é que o alinhamento global não é afetado pelos erros individuais do alinhamento por pares. Em outras palavras, se correspondências reais fossem utilizadas, o processo de minimização acabaria priorizando os *piores* alinhamentos, devido a estes erros ao quadrado variarem mais. Já com os correspondentes virtuais, a distância inicial é sempre zero, logo todos os alinhamentos acabam sendo processados de forma igual.

Devemos lembrar um detalhe: O algoritmo de Pulli serve para difundir erros de alinhamento. Logo, ele necessita de um alinhamento inicial para cada vista, sobre o qual ele vai agir. Este alinhamento inicial é derivado do grafo dos alinhamentos por pares (vide figura 5.1), através da descoberta da *spanning tree* que visita todos os nodos do grafo. Uma vez obtida a *spanning tree*, o alinhamento inicial de cada vista corresponde à combinação (multiplicação) das matrizes $4 \times 4$ de cada aresta, partindo da raiz da árvore até a vista específica.

Uma das grandes vantagens do algoritmo de Pulli é que ele não requer que as vistas estejam carregadas simultaneamente na memória. Logo ele é particularmente útil para o registro global de modelos com muitas vistas. Além disso, apenas alguns pontos selecionados para cada par de vistas são necessários, e não as vistas completas. Finalmente, o tempo de execução é bem rápido, muito mais rápido do que a etapa de alinhamento por pares.

### 5.2.2 Alinhamento com os Eixos Principais

Uma vez executado o algoritmo de Pulli, as diversas vistas já estão corretamente alinhadas umas com as outras. No entanto, até este momento, nenhum tratamento foi dado ao sistema de coordenadas final do objeto como um todo. Na prática, se nada for feito, o sistema de coordenadas do objeto final corresponderá ao sistema de coordenadas da vista na raiz da *spanning tree* do grafo de vistas (vide figura 5.1). Infelizmente, este sistema normalmente não é o ideal para o objeto final.

Como várias etapas seguintes do *pipeline* trabalharão com o volume do modelo, convém encontrar um sistema de coordenadas que minimize o volume da *axis-aligned bounding box* do objeto. Isto pode ser conseguido calculando-se os eixos principais da nuvem de pontos, conforme sugerido por Wu [158].

Para efetuar este cálculo dos eixos principais, inicialmente deve-se calcular o centro de massa do objeto conforme a equação 5.1:

$$CM = \frac{\sum_{i=1}^{n} \sum_{j=1}^{v_i} \left( V_{i_j} * M_{V_i \to World} \right)}{\sum_{i=1}^{n} v_i} \tag{5.1}$$

onde $n$ representa o número de vistas, $v_i$ o número de vértices da vista $i$, $V_{i_j}$ o $j$-ésimo vértice da vista $V_i$, e $M_{V_i \to World}$ a matriz $4 \times 4$ que converte do sistema de coordenadas local de $V_i$ para o sistema de coordenadas global do objeto, como calculadas pelo algoritmo de Pulli.

Em seguida, devemos construir uma matriz de covariância dos pontos, usando a fórmula 5.2:

$$Cov = \begin{bmatrix} C_{x,x} & C_{x,y} & C_{x,z} \\ C_{y,x} & C_{y,y} & C_{y,z} \\ C_{z,x} & C_{z,y} & C_{z,z} \end{bmatrix} \tag{5.2}$$

As 9 covariâncias são:

$$\begin{aligned}
C_{x,x} &= \sum_{i=1}^{n} \sum_{j=1}^{v_i} \left( V_{i_j} * M_{V_i \to World} - CM \right)_x * \left( V_{i_j} * M_{V_i \to World} - CM \right)_x \\
C_{x,y} = C_{y,x} &= \sum_{i=1}^{n} \sum_{j=1}^{v_i} \left( V_{i_j} * M_{V_i \to World} - CM \right)_x * \left( V_{i_j} * M_{V_i \to World} - CM \right)_y \\
C_{x,z} = C_{z,x} &= \sum_{i=1}^{n} \sum_{j=1}^{v_i} \left( V_{i_j} * M_{V_i \to World} - CM \right)_x * \left( V_{i_j} * M_{V_i \to World} - CM \right)_z \\
C_{y,y} &= \sum_{i=1}^{n} \sum_{j=1}^{v_i} \left( V_{i_j} * M_{V_i \to World} - CM \right)_y * \left( V_{i_j} * M_{V_i \to World} - CM \right)_y \\
C_{y,z} = C_{z,y} &= \sum_{i=1}^{n} \sum_{j=1}^{v_i} \left( V_{i_j} * M_{V_i \to World} - CM \right)_y * \left( V_{i_j} * M_{V_i \to World} - CM \right)_z \\
C_{z,z} &= \sum_{i=1}^{n} \sum_{j=1}^{v_i} \left( V_{i_j} * M_{V_i \to World} - CM \right)_z * \left( V_{i_j} * M_{V_i \to World} - CM \right)_z \qquad (5.3)
\end{aligned}$$

onde estas covariâncias são obtidas a partir da multiplicação das componentes $(x, y, z)$ da posição de todos os vértices em relação ao centro de massa calculado em (5.1).

Uma vez obtida a matriz de covariância, os 3 eixos principais são obtidos a partir dos 3 autovetores da matriz $Cov$. Como a matriz é simétrica, podemos utilizar o método de

Jacobi para o cálculo dos autovetores [112].

Os 3 autovetores são utilizados para construir uma matriz homogênea $4 \times 4$ de rotação, conforme a equação 5.4:

$$Rot = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

onde $u$, $v$ e $w$ são os 3 autovetores normalizados, que corresponderão respectivamente aos novos eixos $x$, $y$ e $z$ do sistema global de coordenadas do modelo.

Agora, podemos calcular a matriz $TM$ que será usada para atualizar as matrizes de transformação $4 \times 4$ de cada vista (5.5):

$$TM = \text{Translação}(-CM) * Rot \tag{5.5}$$

onde Translação() cria uma matriz $4 \times 4$ de translação.

Finalmente, a matriz $M_{V_i \to World}$ de cada vista $V_i$ deve ser multiplicada pela matriz $TM$ para convertê-la para o novo sistema de coordenadas global (5.6):

$$M_{V_i \to World}' = M_{V_i \to World} * TM \tag{5.6}$$

Na figura 5.4 podemos observar os resultados deste alinhamento com os eixos principais. O novo sistema de coordenadas das figuras 5.4(b) e 5.4(e) poderia ser melhor, como aquele mostrado nas figuras 5.4(c) e 5.4(f).

O que ocorre é que quando utilizamos os vértices de todas as vistas para cálculo dos eixos principais, acabamos dando mais peso para regiões que possuem mais vistas, pois regiões de interseção são consideradas em duplicidade. No caso da estátua da figura 5.4, temos muito mais vistas frontais e da menina do que de outras regiões, o que fez com que o sistema de coordenadas obtido "pendesse" para a frente e para a direita, conforme pode ser observado nas figuras 5.4(b) e 5.4(e).

Para resolver este problema, devemos encontrar uma forma de amostrar uniformemente os pontos do objeto. Nossa solução consiste em criar uma representação volumétrica simplificada do objeto. Criamos um *array* tridimensional booleano ($32 \times 32 \times 32$ em nossa implementação), e inicializamos todos os *voxels* como **falsos**. A seguir, calculamos a *axis-aligned bounding box* do objeto, obtida através das coordenadas mínimas e máximas dos vértices nos eixos $(x, y, z)$. Dividimos esta *bounding box* como o nosso *array* tridimen-

sional (em $32 \times 32 \times 32$ elementos). Agora, para todos os vértices de todas as vistas, descobrimos em qual elemento da *bounding box* ele está contido, e marcamos como **verdadeiro** o elemento correspondente em nosso *array* tridimensional. Desta maneira, o nosso *array* booleano final corresponde à uma representação uniforme da superfície do nosso objeto, onde as redundâncias das vistas foram eliminadas.

Em seguida, efetuamos o alinhamento com os eixos principais conforme apresentado anteriormente. Só que ao invés de utilizar os vértices das vistas, utilizamos as coordenadas 3D do centro de cada *voxel* marcado como **verdadeiro** em nosso *array*. Os resultados podem ser observados nas figuras 5.4(c) e 5.4(f), e são claramente melhores do que a solução calculada diretamente a partir dos vértices das vistas (figuras 5.4(b) e 5.4(e)).

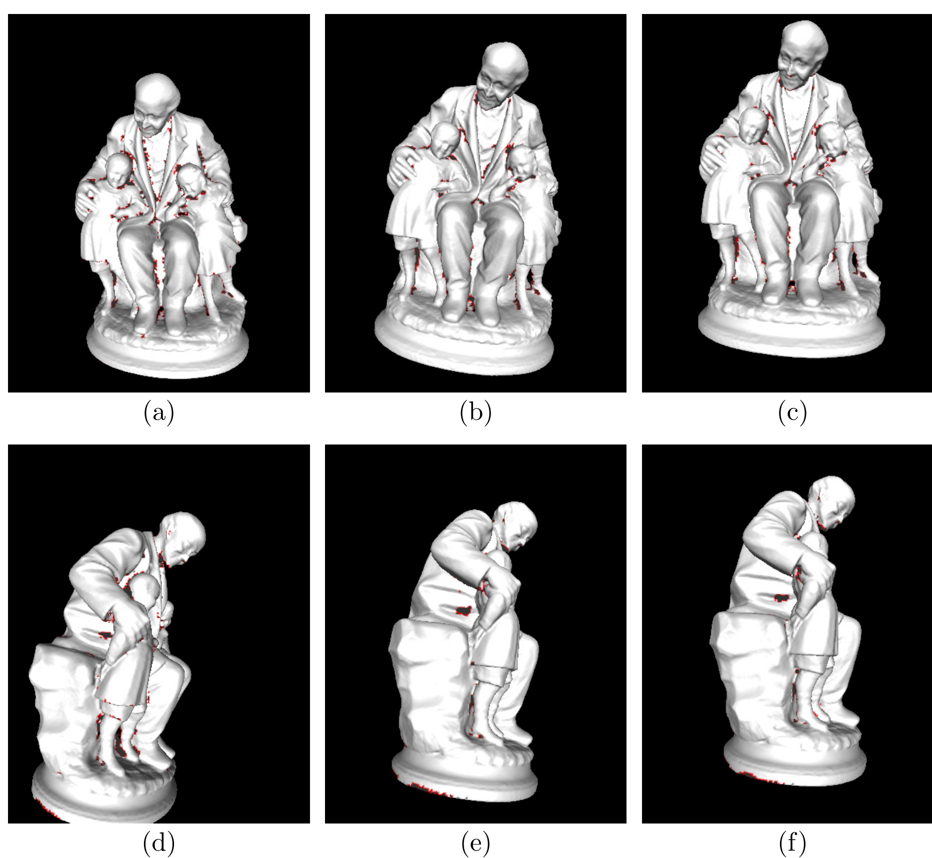



**Figura 5.4:** Alinhamento com os eixos principais. (a) e (d) Alinhamento global com o sistema de coordenadas da 1ª vista, ou seja, o resultado do algoritmo de Pulli; (b) e (e) Alinhamento com os eixos principais, calculados a partir dos vértices de todas as vistas; (c) e (f) Alinhamento com os eixos principais, calculados a partir de uma representação volumétrica simplificada do objeto. A câmera está alinhada com eixos do sistema de coordenadas global em todas as figuras.

## 5.3 Conclusões e Trabalhos Futuros

Durante este capítulo, pudemos observar os resultados da etapa de alinhamento global nas figuras 5.2(b) e 5.2(d), e do alinhamento com os eixos principais nas figuras 5.4(c) e 5.4(f). A melhoria no alinhamento baseado apenas em pares é obvia. O alinhamento com os eixos principais, por hora, gera apenas uma melhoria estética (a estátua fica em uma posição mais "natural"), mas seu verdadeiro valor será apreciado nas etapas seguintes do *pipeline*.

Nossa avaliação do algoritmo de Pulli [113] é muito favorável. Ele apresenta ótimo desempenho e não requer grande gasto de memória, uma vez que todas as vistas não precisam estar carregadas na RAM simultaneamente.

Uma das formas de melhorar ainda mais a qualidade do alinhamento global é através de mais alinhamentos individuais de pares. Conforme apresentamos na seção 5.2.1, quanto mais pares, mais informações estão disponíveis para movimentar as vistas para os seus locais corretos. Atualmente, os pares de vistas para o ICP são especificados manualmente no arquivo de *script* de entrada para o *ModelTool*. Uma idéia seria tentar encontrar automaticamente todos os pares de vistas onde haja um grau de sobreposição suficiente para o ICP.

Isto pode ser conseguido de várias maneiras, sendo que uma delas seria a utilização da nossa representação volumétrica simplificada (*array* booleano $32 \times 32 \times 32$). Poderíamos criar esta representação para cada vista, e comparar os *arrays* de todos os pares possíveis. Uma operação booleana AND entre os *arrays* resultaria no número de *voxels* em comum entre as vistas, e se este número fosse suficientemente alto, um novo par de alinhamento entre as duas vistas candidatas poderia ser criado. Inclusive o pré-alinhamento para este novo par poderia ser calculado a partir do grafo de relacionamento entre vistas. Uma nova execução do *ModelTool* refinaria o alinhamento, levando em conta todos os novos pares adicionados automaticamente.

# CAPÍTULO 6

# INTEGRAÇÃO DAS VISTAS

Uma vez alinhadas, precisamos combinar todas as vistas em um único modelo 3D. Esta é uma etapa crítica do *pipeline* de reconstrução de modelos, pois é aqui que os diversos defeitos oriundos da etapa de aquisição devem ser processados e eliminados (vide seção 3.3).

Muita pesquisa já foi e continua sendo feita para esta etapa do *pipeline*, conforme apresentaremos na seção 6.1. Infelizmente, nossos experimentos práticos revelaram graves deficiências nos principais algoritmos, que apresentaremos em detalhes na seção 6.2. Isto foi surpreendente, pois uma revisão rápida do estado da arte nos leva a crer que este problema já possui boas soluções; a verdade é que até o momento não existe uma única solução sequer que resolva o problema com alta qualidade. Despendemos enormes esforços tentando criar um algoritmo que produzisse resultados aceitáveis, e ainda não estamos plenamente satisfeitos com os resultados obtidos.

Isto nos mostra que a Integração de Vistas ainda requer muitos esforços de pesquisa, de forma que *pipelines* de reconstrução 3D de modelos de alta qualidade possam se tornar uma realidade prática.

## 6.1 Revisão Bibliográfica

Um artigo que apresenta vários métodos de integração de vistas é o de Bernardini e Rushmeier [15]. Classificaremos os diversos algoritmos conforme as técnicas básicas que eles utilizam.

### 6.1.1 Métodos de Delaunay

Estes métodos se utilizam do complexo Delaunay $D(S)$ associado a um conjunto de pontos $S$ em $\mathbb{R}^3$. Este complexo impõe uma estrutura de conectividade aos pontos, e estes métodos acabam extraindo um sub-complexo de $D(S)$ para representar a superfície integrada. Pode-se perceber que esta classe de algoritmos trabalha diretamente com nuvens de pontos. O trabalho de Edelsbrunner [45] apresenta uma revisão destes métodos.

Alguns métodos que podemos destacar são os que utilizam *alpha-shapes* [46, 12, 6]; os

que utilizam o método de *power crusts* [3, 4]; os métodos que utilizam *cocones* [2, 40]; e os métodos que utilizam *eigencrusts* [84].

O grande problema destes algoritmos é que eles são muito sensíveis a ruídos e *outliers*, pois interpolam os pontos de dados. Isto requer uma etapa de pré-processamento capaz de "limpar" os dados de entrada (*e.g.* [163]). Além disso, são algoritmos muito custosos em termos de desempenho, o que limita o tamanho do conjunto de dados de entrada. Por estes motivos, acabamos não experimentando com esta categoria de algoritmos em nosso *pipeline*.

### 6.1.2 Processamento de Superfícies

Outra categoria de algoritmos de integração de vistas se baseia na criação ou manipulação de superfícies diretamente. Alguns destes métodos se utilizam das superfícies dos modelos 3D das vistas individuais.

O algoritmo de Turk e Levoy [152] cria os chamados *zippered meshes*. Modelos 3D das vistas são criados, e regiões na interseção das vistas (onde existem triângulos sobrepostos) são erodidas. Em seguida, estas regiões erodidas são retrianguladas e unidas de forma a gerar o modelo final.

Soucy e Laurendeau [141] descrevem um método que se utiliza de diagramas de Venn para identificar regiões de sobreposição, em seguida reparametrizando e unindo estas regiões.

Bernardini [14] propôs o algoritmo *ball-pivoting*, que trabalha sobre nuvens de pontos, mas sem necessitar de triangulações de Delaunay. A idéia é o crescimento da superfície a partir de um triângulo inicial, gerando faces através da "rotação" de uma esfera em torno das arestas do perímetro da superfície sendo criada, procurando por vértices dentro deste volume de rotação da esfera, de forma a criar novas faces.

Gopi [61] calcula triangulações Delaunay 2D, projetando os pontos sobre um plano, e depois "eleva" os pontos em 3D para a sua posição correta.

Estes algoritmos podem falhar catastroficamente em áreas de grande curvatura (como demonstrado no artigo de Curless [37], na tentativa de reconstrução de uma broca de furadeira), e podem apresentar soluções topologicamente incorretas, principalmente devido à *outliers* nas vistas de entrada. Além disso, alguns algoritmos só funcionam quando a taxa de amostragem de pontos é relativamente constante entre as diversas vistas.

Além destes problemas, estes métodos em geral aproveitam trechos dos modelos das vistas. Sem nenhum pós-processamento, isto acaba gerando superfícies muito ruidosas,

pois cada vista individual possui ruído (vide seção 3.3.1), e simplesmente "juntar" trechos de cada vista preserva o ruído presente nas mesmas. Isto é evidentemente um desperdício de informações, pois para um mesmo ponto na superfície do objeto existem várias amostras, cada uma proveniente de uma vista diferente. Ou seja, alguns algoritmos acabam descartando informações que poderiam ser utilizadas para reduzir o ruído. Finalmente, outro problema ocorre nas emendas das vistas, que acabam deixando "cicatrizes" que são visíveis no objeto final.

### 6.1.3 Superfícies Paramétricas

Estes métodos se baseiam na idéia de deformar uma aproximação inicial do objeto, devido à forças externas e reações e limitações internas. Podemos imaginar isso como um "balão" que vai se inflando ou murchando para assumir a forma do objeto. Outras abordagens usam uma ou mais superfícies geradas analiticamente para representar o modelo integrado.

Terzopoulos [145] usa um modelo deformável onde forças intrínsecas induzem uma preferência por formas simétricas.

Pentland e Sclaroff [110] adotaram um método baseado em elementos-finitos e superfícies paramétricas, e conectam "molas" entre pontos da superfície com pontos no objeto, e o modelo final é obtido quando o sistema se estabiliza.

Sharf *et al.* [132] utilizam um modelo deformável em uma abordagem *coarse-to-fine*, analisando eventos topológicos para modificar dinâmicamente o *genus* do objeto.

Carr *et al.* [24] utilizam *Radial Basis Functions* para gerar a superfície integrada a partir de nuvens de pontos com normais.

Ohtake *et al.* [106] apresenta um método interessante, onde a superfície final é formada pelo *blending* de várias superfícies paramétricas, calculadas e ponderadas localmente. Como a superfície é descrita através de funções implícitas, operações como *morphing*, deformações e CSG são simples de implementar. Fleishman *et al.* [50] aperfeiçoa esta idéia através do uso de estatística robusta.

Kazhdan *et al.* [82] propuseram o método de *Poisson Surface Reconstruction*. Eles calculam funções de base localmente suportadas através da solução de sistemas lineares esparsos, que representam o problema de Poisson.

Sharf *et al.* [133] usam um método de elementos finitos, aliado a restrições fornecidas interativamente pelo usuário para garantir uma reconstrução topologicamente correta.

Outros métodos foram sugeridos, que utilizam curvas de nível (*level sets*) [156, 60, 162]. Estes métodos combinam estatística robusta para eliminar a influência de ruído e *outliers*.

Muitos destes métodos acabam sendo híbridos entre métodos de superfícies paramétricas e métodos de representação volumétrica, que serão apresentados a seguir (na seção 6.1.4).

Uma das desvantagens deste tipo de abordagem é a dificuldade de representar "cantos vivos", já que a maioria dos métodos assumem que uma função continuamente diferenciável representa a superfície. Outro problema costuma ser encontrar o equilíbrio entre uma suavização excessiva e a não-eliminação de ruídos. Finalmente, o preenchimento de buracos intrínseco destes métodos pode gerar resultados incorretos.

### 6.1.4 Métodos Volumétricos

A idéia básica destes métodos é criar uma representação volumétrica implícita do objeto final, onde cada *voxel* possui a medida da distância com sinal para a superfície integrada. O sinal da distância indica se o *voxel* é interior ou exterior ao objeto. Desta forma, a superfície do objeto consiste na isosuperfície que possua a distância 0. Esta isosuperfície pode ser obtida de várias formas, sendo a mais utilizada o algoritmo *Marching Cubes* [93]. Tal representação é conhecida na literatura como SDF (*Signed Distance Field*) e é amplamente utilizada em diversos campos da computação gráfica, principalmente para objetos orgânicos, deformáveis ou criados a partir de equações matemáticas. Ao contrário da abordagem por superfícies paramétricas, os métodos volumétricos não tentam calcular a função de distância analiticamente; ela é definida somente através da interpolação das amostras em cada voxel.

Os vários algoritmos se diferenciam pela forma como a função de distância é estimada a partir dos dados disponíveis. Curless e Levoy [37], em seu algoritmo VRIP (*Volumetric Range Image Processing*), calculam as distâncias conforme a linha de visão do *scanner*, e utiliza como peso para as medidas o ângulo formado entre a linha de visão e a superfície, pois conforme explicamos anteriormente (seção 3.3), medidas tomadas "de raspão" acabam sendo menos confiáveis. O método processa as vistas uma a uma, integrando os resultados na representação volumétrica. Apenas medidas próximas à superfície são processadas e armazenadas, para reduzir o gasto computacional e de memória.

Hilton *et al.* [66] também combina medidas de várias vistas, mas utiliza heurísticas para saber quais medidas considerar na estimativa da função de distância. Isto é usado para descartar vistas próximas, mas com orientações contrárias, entre outros casos. O maior problema deste método é que as regras de seleção e descarte de vistas não garante que todos os pontos fazem parte da mesma porção da superfície do objeto, mesmo no caso ideal sem ruído. Isto acaba resultando em erros na representação volumétrica do objeto, conforme apontado em [155].

Wheeler [155] sugeriu um método onde os *outliers* seriam descartados. Para tanto, ele calcula o que chama de superfície de consenso (*consensus surface*). Para que uma medida seja considerada em consenso, deve haver um quorum mínimo $q$ de medidas similares, ou seja, dentro de uma tolerância. Para cada *voxel*, utiliza-se a medida de consenso mais próxima de 0, ou caso não haja consenso, a medida sem consenso de maior quorum. Este processo é repetido para todos os *voxels*. Para acelerar o algoritmo, utilizam-se *octrees* [100], onde os nodos da *octree* são divididos quando se aproximam da superfície de distância 0, o que reduz significativamente o gasto de memória e o tempo de processamento. Melhorias ao algoritmo de Wheeler foram sugeridas por Sagawa *et al.* [125, 123, 126, 124].

Masuda [97] usa um SDF para representar cada vista, e efetua o alinhamento e integração de todas as vistas simultâneamente. Os passos de alinhamento e integração são alternados até a convergência. Desta forma, consegue-se detectar *outliers* e melhorar a qualidade do processo como um todo. O problema é que o método acaba requerendo muito processamento e memória.

Rusinkiewicz *et al.* [121] usa um método de integração volumétrica sobre nuvens de pontos, para permitir uma visualização prévia do modelo durante a aquisição. Para gerar o modelo definitivo de melhor qualidade após a captura dos dados, utiliza-se o VRIP [37].

Rocchini *et al.* [116] sugere o algoritmo *Marching Intersections* para integrar várias imagens de profundidade. Segundo seu artigo, seu algoritmo apresenta um custo computacional reduzido em relação a outros métodos volumétricos e fornece resultados de melhor qualidade.

Duguet *et al.* [42] sugere uma representação volumétrica que além de integrar as vistas, permite a obtenção de vários níveis de detalhe do objeto final. O método se utiliza de *Gradient Vector Flow* e da equação de Poisson para calcular a função de distância. No entanto, não fica claro como é o comportamento do algoritmo na presença de ruídos e *outliers*.

Hornung e Kobbelt [73] utilizam distâncias sem sinal e um algoritmo de corte de grafos para extrair a isosuperfície. O uso de distâncias sem sinal elimina a necessidade de normais para os pontos de entrada.

Yemez *et al.* [161] propõem a integração volumétrica de dados obtidos através de triangulação ótica e pela técnica de *shape from silhouette*. A vantagem é utilizar os dados de silhueta para completar a reconstrução em regiões de oclusão das imagens de profundidade. Uma dificuldade consiste na detecção das silhuetas sem utilizar iluminação especial ou controle do fundo da imagem.

Os métodos volumétricos possuem como vantagens o fato de usar todas as informações disponíveis (atenuando ruído dos dados de entrada), e garantir a geração de topologias *manifold* [37] (esta garantia depende de uma implementação sem ambiguidades do algoritmo Marching Cubes, conforme apresentaremos no capítulo 8). Sua grande desvantagem é o gasto de memória e processamento, pois para garantir uma boa fidelidade o tamanho dos *voxels* deve ser aproximadamente igual à distância entre amostras do *scanner*, que costuma ser bem pequena (na ordem de $0,33\ mm$). *Voxels* tão pequenos acabam gerando volumes consideráveis, e vários esforços são necessários para viabilizar o processamento de tantas informações com limites práticos de tempo e espaço.

## 6.2 Solução Adotada

Depois de analisar as diversas técnicas de integração de vistas, pudemos perceber problemas em quase todas as abordagens. Os métodos de Delaunay (seção 6.1.1) são muito custosos, e requerem outros algoritmos para eliminar dados incorretos; os métodos de processamento de superfícies (seção 6.1.2) podem apresentar problemas em topologias complexas, e também são sensíveis a dados ruidosos; a maior parte dos métodos de superfícies paramétricas (seção 6.1.3) não conseguem tratar "cantos vivos" e o ajuste das superfícies pode ser problemático devido aos *outliers*; e finalmente os métodos volumétricos (seção 6.1.4) são muito custosos em termos de tempo e espaço. Kazhdan *et al.* [82] comparou vários algoritmos recentes, e todos eles foram incapazes de garantir reconstruções de alta fidelidade, principalmente em regiões de detalhes dos objetos.

Esta revisão bibliográfica nos levou a escolher os métodos volumétricos, porque são os que menos impõem restrições aos objetos reconstruídos; oferecem uma forma fácil de mudar a precisão do resultado (através da variação do tamanho do *voxel*); podem suportar trivialmente a técnica de *space carving*; e conseguem funcionar mesmo com dados incorretos. Além disso, métodos volumétricos já tinham sido empregados em grandes projetos de preservação digital, como o *"The Digital Michelangelo Project"* [89] e o *"The Great Buddha Project"* [101, 75]. Finalmente, métodos volumétricos apresentam uma boa sinergia com a solução adotada para a etapa de preenchimento de buracos, que será analisada em detalhes no capítulo 7.

Implementamos, testamos e modificamos exaustivamente 3 algoritmos: o algoritmo VRIP de Curless e Levoy [37], usado no *"The Digital Michelangelo Project"*; o algoritmo *Consensus Surfaces* de Wheeler [155], usado no *"The Great Buddha Project"*; e uma versão híbrida dos dois algoritmos, desenvolvida por nós para tentar solucionar os problemas dos dois métodos anteriores. A seguir analisaremos em detalhes cada uma destas soluções.

### 6.2.1 Algoritmo de Curless e Levoy (VRIP)

Como todo algoritmo volumétrico, o VRIP procura calcular para cada *voxel* do volume a distância com sinal do centro do *voxel* para a superfície, ou seja, criar um SDF (*Signed Distance Field*) do objeto integrado. A essência do algoritmo é baseada na soma ponderada dos SDFs referentes a cada vista. O peso da ponderação refere-se ao grau de confiabilidade de cada ponto. A soma das diversas medidas acaba gerando uma superfície média, que representa a integração das várias vistas. Pode-se observar tal funcionamento na figura 6.1.

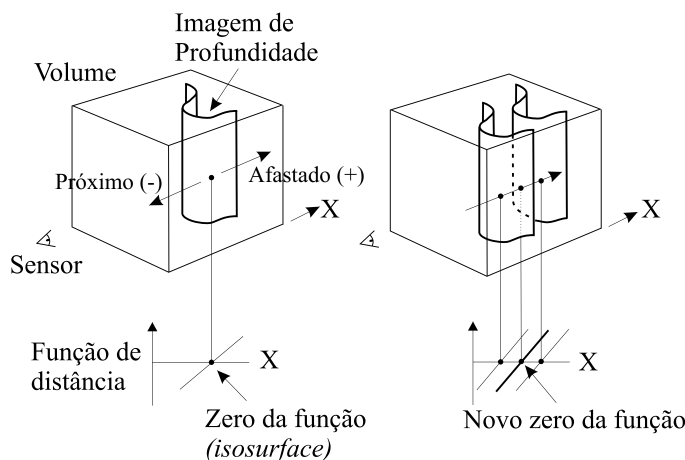



**Figura 6.1:** Soma de SDFs nas representações volumétricas de vistas. Neste exemplo, a soma da função de distância das duas vistas gera uma nova função de distância, onde o zero da função resultante acaba ficando em uma posição intermediária entre os zeros das vistas sendo integradas.

Outro ponto importante a ser comentado é que as distâncias são calculadas apenas na proximidade das superfícies. Desta forma, as medidas estão restritas entre uma distância $-D_{min}$ e uma distância $+D_{max}$. Para isso, o peso das medidas tende a zero quando a distância se aproxima das distâncias-limite. Finalmente, estas distâncias são calculadas na direção da linha-de-visão de cada imagem de profundidade, conforme a figura 6.2 (retirada do artigo de Curless [37]).

Nas figuras 6.2(d), 6.2(e) e 6.2(f) também podemos visualizar os pesos dados a cada *voxel*. Eles estão diretamente relacionados às normais da superfície. Quanto mais a linha-de-visão for perpendicular à superfície, maior o peso. Isto é feito porque na prática constata-se que as medidas onde o *laser* pega "de raspão" no objeto são bem menos confiáveis do que as outras medidas (vide seção 3.3).

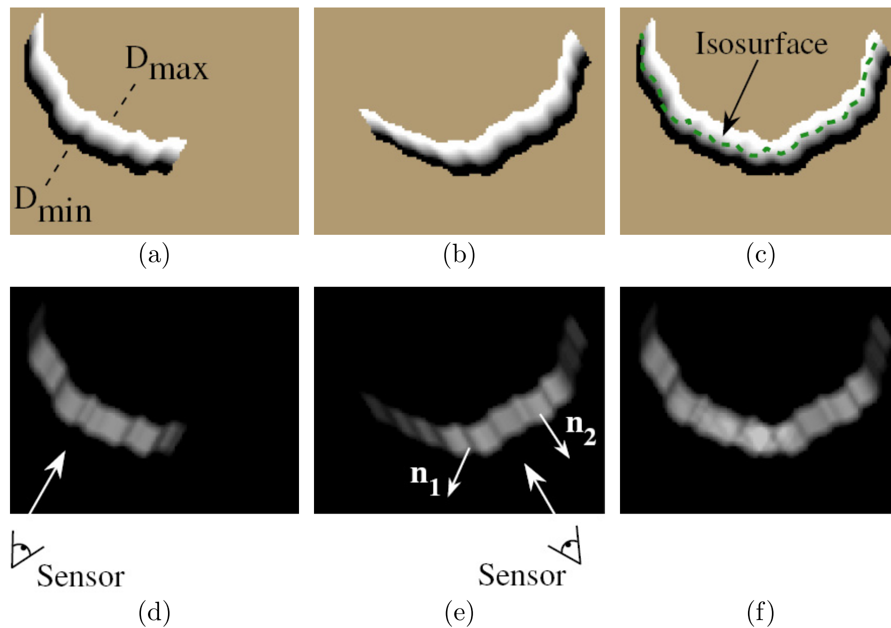



**Figura 6.2:** Fusão da função de distância e de peso em duas dimensões. (a) e (d) são a distância e o peso, respectivamente, gerados para uma vista observada a partir da posição do sensor conforme mostrado em (d). A função de distância varia entre $D_{min}$ e $D_{max}$, como exibido em (a). O peso das medidas diminui conforme o ângulo formado entre o sensor e a superfície, conforme pode ser percebido pelas regiões mais escuras em (e). As normais $n_1$ e $n_2$, mostradas em (e), estão orientadas obliquamente e de frente para o sensor, respectivamente. Nota-se que o peso é menor (mais escuro) para a normal oblíqua. (b) e (e) são a distância e o peso para outra vista do mesmo objeto. (c) é a função de distância $D(x)$ resultante da combinação ponderada de cada voxel de (a) e (b). (f) corresponde a soma $W(x)$ dos pesos de cada voxel. A linha pontilhada verde em (c) corresponde à isosuperfície de valor 0, ou seja, é a estimativa da superfície integrada do objeto.

Podemos formalizar o algoritmo através do pseudo-código 6.1. O método processa as vistas uma a uma, integrando os resultados na representação volumétrica. Apenas medidas próximas à superfície são processadas e armazenadas, para reduzir o gasto computacional e de memória.

**Algoritmo 6.1** Pseudo-código para o algoritmo de Integração de Vistas de Curless [37].

1: Inicialize o volume com distância $D_{max}$ e peso 0 //*voxels* não observados
2: **para** cada vista $i$ **faça**
3:     **para** cada voxel $x$ próximo da superfície da vista $i$ **faça**
4:         Calcule a distância $d_i(x)$ do *voxel* até a superfície da vista $i$ ($-D_{min} \leq d_i(x) \leq +D_{max}$), e o peso $w_i(x)$ desta medida ($0 \leq w_i(x) \leq 1$)
5:         Atualize a distância $D(x)$ e o peso $W(x)$ do volume através das fórmulas $D(x) \leftarrow \frac{W(x)D(x)+w_i(x)d_i(x)}{W(x)+w_i(x)}$ e $W(x) \leftarrow W(x) + w_i(x)$
6:     **fim para**
7: **fim para**

Uma modificação interessante do algoritmo é o chamado "*space-carving*". Todos os *voxels* que estejam entre a superfície da vista e o sensor do *scanner*, são sabidamente "vazios". No algoritmo 6.1, os *voxels* "vazios" são indicados com o valor $-D_{min}$ e o peso 0. Estes *voxels* vazios podem ser utilizados posteriormente para auxiliar a etapa de preenchimento de buracos.

Trabalhamos bastante com este algoritmo, e podemos caracterizar suas principais vantagens e desvantagens:

- Vantagens:
  - Rápido (tempo linear no número de vistas);
  - Permite adição incremental de vistas (útil para refinamentos sucessivos, através da adição de vistas extras);
  - Suaviza o ruído das vistas individuais, gerando uma superfície integrada mais uniforme, devido à média ponderada das várias vistas;
  - Utiliza a informação da posição do sensor do *scanner*, de forma a definir *voxels* sabidamente "vazios". Esta informação é muito útil para os algoritmos de preenchimento de buracos;
  - Não requer que todas as vistas estejam carregadas na memória simultaneamente, pois as vistas são processadas uma a uma.
- Desvantagens:
  - Não descarta *outliers*. O máximo que pode ser feito é a redução do peso deles, mas eles são difíceis de serem detectados, pois cada vista é tratada individualmente e para definir "*outlier*" precisamos relacioná-lo com as medidas das outras vistas;
  - Integra métricas de pontos de vista diferentes, de forma que os valores do SDF acabam não sendo uniformes. Isto pode gerar superfícies ligeiramente deslocadas [155], e pode também atrapalhar algoritmos de preenchimento de buracos;
  - Superfícies com pouca espessura e cantos geram artefatos. Este é um problema crítico, pois impede o algoritmo de ser usado para a criação de réplicas com alta fidelidade. Isto advém do fato que distâncias "atrás" da superfície não são diretamente observadas, e no caso de superfícies com pouca espessura e cantos as distâncias estimadas são completamente erradas. Esta é uma falha teórica no método, já apontada pelos próprios autores do algoritmo, e que não possui solução simples.

Esta última desvantagem merece uma explicação detalhada, pois é o verdadeiro "calcanhar-de-aquiles" do algoritmo. Na figura 6.3 vemos dois modelos, e duas vistas da base do modelo 6.3(a), com a luz em posições diferentes. A base em ambas as estátuas é plana, mas nos modelos acabamos tendo "bordas" ressaltadas, devido à falha teórica do algoritmo.

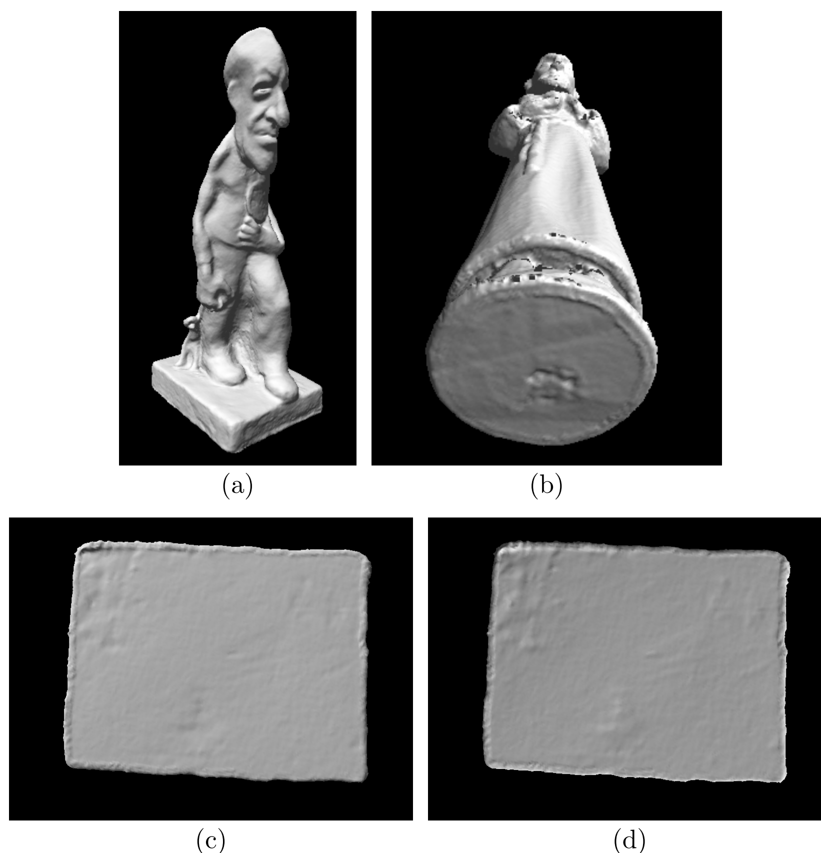
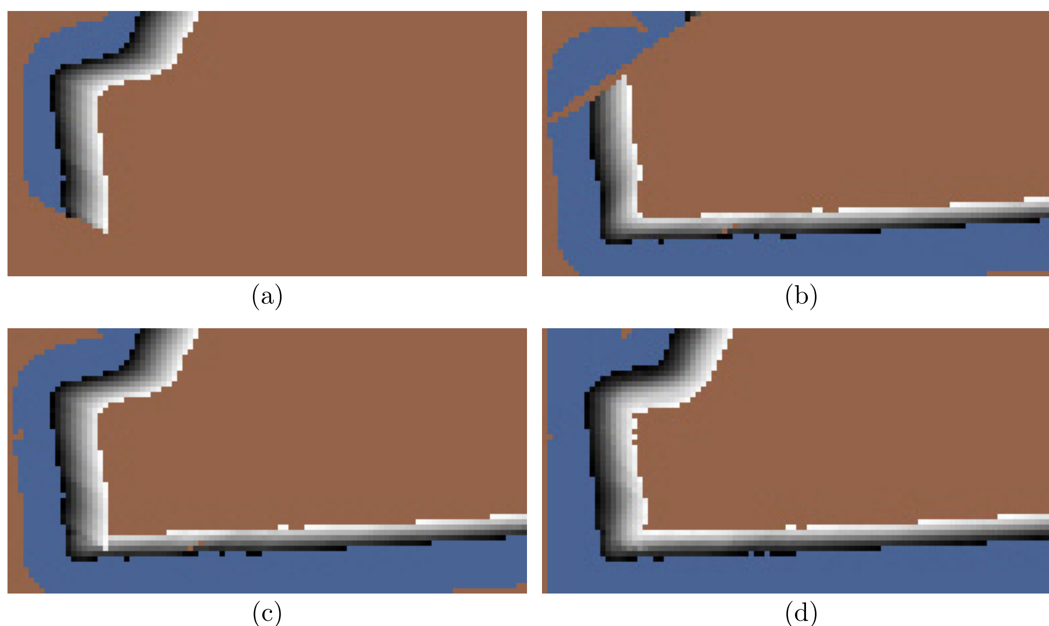



**Figura 6.3:** Defeito gerado próximo a cantos pelo algoritmo de Curless. (a) Modelo reconstruído de uma estátua do artista Erbo Stenzel; (b) Modelo reconstruído de outro objeto, que também apresenta um "ressalto" incorreto na base; (c) e (d) Visões da base da estátua de Stenzel, com a luz em posições diferentes, de forma a facilitar a identificação do "ressalto" gerado numa superfície que deveria ser plana.

Para entender como o "ressalto" é criado, devemos observar cortes da representação volumétrica. Estes cortes podem ser observados na figura 6.4, onde analisamos a borda gerada na base da figura 6.3(a). Para entender como a borda é gerada, observemos a figura 6.4(c). Nela podemos perceber que existe um canto "branco" (valores positivos) invadindo uma região "cinza escura" (valores negativos). Com a execução do algoritmo, os valores acabam sendo combinados, criando o ressalto nas bordas da base da estátua.

O mesmo problema ocorreria no caso de superfícies estreitas, onde os dois lados das superfícies acabariam interferindo um com o outro, aumentando a espessura. Como o

funcionamento do algoritmo requer a combinação dos valores positivos e negativos (para encontrar a superfície média), torna-se difícil eliminar estes artefatos. Os artefatos podem ser reduzidos diminuindo-se os valores $-D_{min}$ e $+D_{max}$, mas isto pode fazer com que vistas ligeiramente desalinhadas não sejam integradas corretamente. Logo, tal medida é apenas paliativa, não resolvendo realmente o problema.



(a) (b) (c) (d)

**Figura 6.4:** Representação volumétrica do defeito do algoritmo de Curless. (a) Representação volumétrica de uma vista tomada de cima para baixo; (b) Representação volumétrica de outra vista, tomada agora de baixo para cima; (c) A vista de cima para baixo sobreposta na de baixo para cima; (d) Representação volumétrica final, levando em consideração todas as vistas. Nestas imagens, azul corresponde à distância $-D_{min}$ (*voxels* vazios), marrom à distância $+D_{max}$ (*voxels* não observados), e os tons de cinza a distâncias entre $-D_{min}$ a $+D_{max}$ (preto a branco). Desta forma, a superfície (distância 0) se encontra no valor cinza 50%.

Quanto aos *outliers*, podemos perceber como eles afetam a qualidade dos modelos gerados através da figura 6.5. Convém notar que o modelo não teve os buracos preenchidos, de forma a não mascarar os artefatos do algoritmo de integração. Como podemos perceber na região ampliada (figura 6.5(b)), vários defeitos são perceptíveis:

- A lateral do pé esquerdo do adulto possui vários ressaltos, gerados por *outliers* de borda de vistas;
- Na altura do calcanhar direito da menina, existem faces totalmente incorretas que estão "penduradas" no ar;
- A lateral da perna esquerda da menina é totalmente irregular, novamente devido a *outliers* de borda.

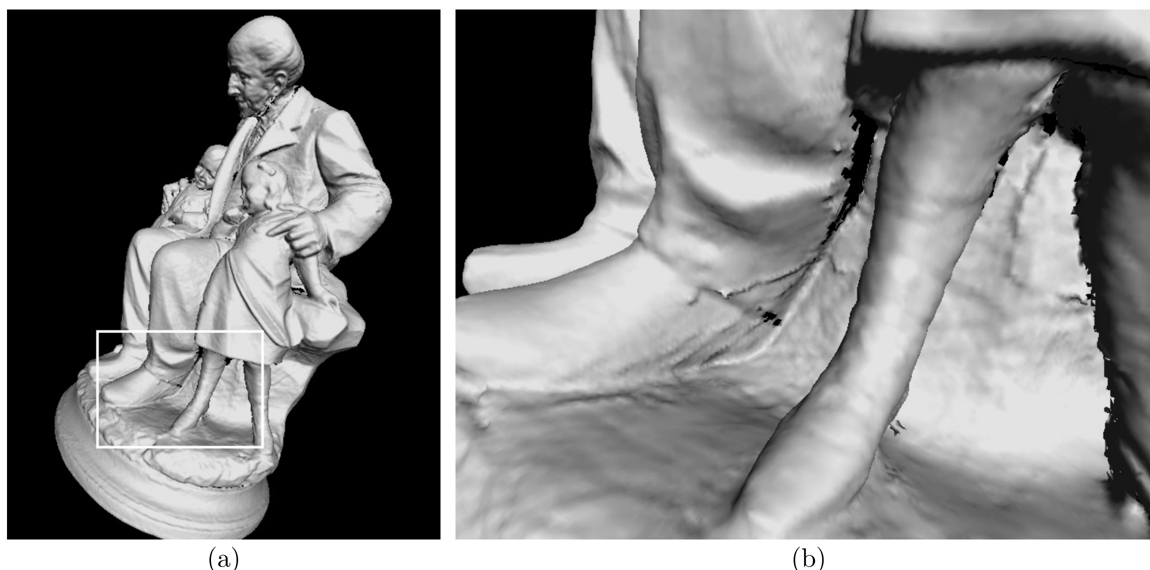



**Figura 6.5:** Efeito dos *outliers* na integração de Curless. (a) Visão geral do modelo integrado; (b) Detalhe ampliado na região das pernas, onde diversas falhas devidas aos *outliers* são observáveis.

Para diminuir a influência dos *outliers*, experimentamos variar a curva de atenuação de pesos, conforme a distância. Ao invés de utilizar uma redução linear a partir da metade da faixa, conforme sugerido por Curless (vide figura 6.6(a)), experimentamos vários perfis de curvas. O que ofereceu o melhor resultado foi aumentar o peso dos *voxels* fora do objeto (negativos), e reduzir o peso dos *voxels* internos (positivos), conforme a figura 6.6(b). No entanto, devemos ressaltar que apesar de reduzir a influência de *outliers* próximos à superfície, este "truque" acaba modificando o posicionamento das superfícies integradas. Em outras palavras, trocamos um tipo de defeito por outro, menos perceptível. Outra vantagem da utilização desta curva é a diminuição dos "ressaltos" em cantos, apresentados na figura 6.3.

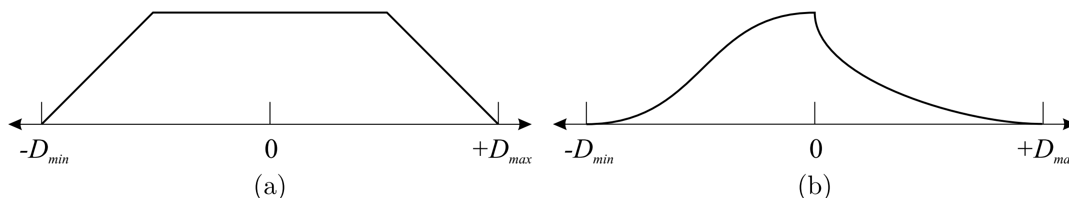



**Figura 6.6:** Perfis de aplicação de peso conforme a distância no algoritmo de Curless. (a) Curva original sugerida no artigo de Curless [37]; (b) Curva modificada, que reduz o efeito indesejável dos *outliers* na superfície integrada. Estes perfis definem um fator entre 0.0 e 1.0 que é multiplicado pelos outros fatores que determinam o peso de cada medida de distância, como por exemplo o ângulo entre a normal e o sensor do *scanner*.

Além da curva de peso conforme a distância (figura 6.6) e do ângulo entre a linha

de visão do *scanner* e a normal da superfície (figura 6.2), outro fator que afeta o peso $w_i(x)$ da medida é a proximidade de bordas da vista. Como mostramos nas seções 3.3.3 e 3.3.4, vários defeitos aparecem próximos a bordas de vistas. Se medidas próximas a bordas tiverem menos peso que medidas do interior da vista, estes defeitos acabam sendo atenuados. Esta regra foi sugerida por Turk e Levoy [152], e adotada por Curless [37].

#### 6.2.1.1 Detalhes de Implementação

O artigo original sugere que a imagem de profundidade seja reamostrada, de forma que os *pixels* e *voxels* possam ser percorridos em ordem. Apesar de num primeiro momento parecer que basta uma transformação 2D na imagem de profundidade, na prática isto não pode ser feito, pois os pontos 3D retornados pelo *scanner* não formam um *grid* regular, devido à compensação da distorção da lente do *scanner*. Isto foi apresentado na figura 3.4.

Logo, para conseguir efetuar a reamostragem, é necessário renderizar a malha 3D da vista, a partir do ponto de vista do *scanner* mas com o plano de imagem rotacionado de forma a se alinhar com os eixos dos *voxels*, isto tudo com uma projeção ortogonal. Em seguida, o *depth buffer* deveria ser capturado de forma a obter uma imagem de profundidade apropriada. Em seguida, para cada *scanline* ocorre uma nova amostragem, para que se obtenham as profundidades correspondentes a cada *voxel* da *scanline*.

Devido à complexidade de implementação deste processo, e como inicialmente estávamos querendo avaliar os resultados qualitativos do algoritmo, ignoramos a reamostragem e implementamos uma função que dadas as coordenadas de um *voxel*, projetamos um raio partindo do centro da lente do *scanner* (cujas coordenadas são conhecidas), passando pelo *voxel* e testando sua interseção com os triângulos que formam a vista (vide figura 6.7). Além disso, com este processo conseguimos estimar o peso, através da interpolação dos pesos dos vértices.

Para acelerar este teste de interseção, acabamos criando uma estrutura de dados auxiliar, onde para cada *pixel* da imagem de profundidade existe uma lista de triângulos que interceptam aquele *pixel*. Quando recebemos as coordenadas do *voxel*, calculamos a sua coordenada 2D na imagem de profundidade. Se esta coordenada estiver fora da imagem, sabemos que não existe interseção. Em seguida, testamos a interseção do raio com os triângulos da lista do *pixel* 2D previamente calculado. A primeira interseção que for encontrada acaba definindo a distância do *voxel* à superfície e o seu peso correspondente. Através desta estrutura de dados, a interseção é extremamente rápida, podendo ser considerada uma operação de tempo $O(1)$, uma vez que as listas de cada *pixel* possuem poucos triângulos (normalmente 6 ou menos), e isto não varia conforme a resolução da imagem de profundidade.

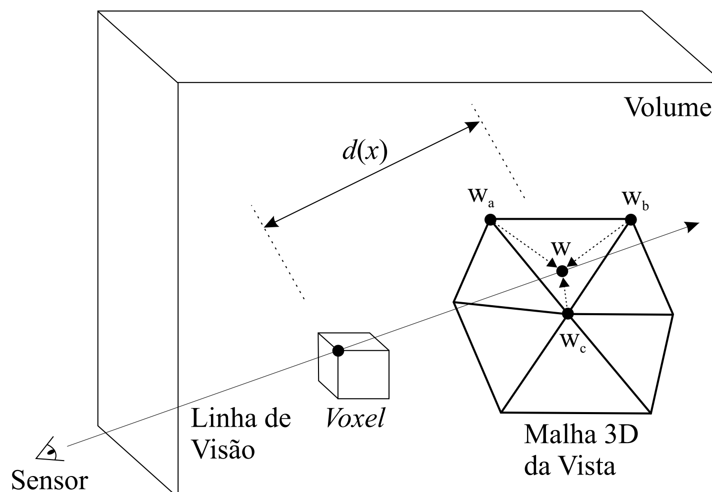



**Figura 6.7:** Forma de cálculo da distância conforme Curless. A partir da posição do sensor e da posição do *voxel* de interesse, calcula-se a interseção desta linha de visão com o modelo 3D da vista. Se não houver interseção, a distância não pode ser calculada; se houver, a distância é medida entre a posição do *voxel* e o ponto de interseção com a malha 3D, sendo positiva quando o *voxel* está atrás da malha 3D. Além disso, o peso é obtido através da interpolação dos pesos calculados para os 3 vértices da face interceptada.

Outro problema resolvido é que precisamos calcular apenas os *voxels* próximos à superfície de cada vista, uma vez que as distâncias estão limitadas à faixa de $-D_{min}$ até $+D_{max}$. Para descobrir estes *voxels* de uma maneira rápida, utilizamos o algoritmo 6.2. Este algoritmo garante que todos os *voxels* próximos à superfície da vista são marcados, e poucos *voxels* marcados acabam sendo fora da faixa; em outras palavras, o volume marcado é bem "justo", reduzindo consideravelmente o tempo de execução do algoritmo de integração. Outras técnicas, como a AABB local de cada vista, acabam marcando muito mais voxels do que a nossa técnica.

**Algoritmo 6.2** Pseudo-código para descobrir *voxels* próximos à superfície de uma vista.

**Requer:** $|-D_{min}| = |+D_{max}|$ //usa-se o mesmo limite para medidas positivas e negativas
1: Descubra o comprimento da maior aresta do modelo 3D da vista em questão
2: $raio \leftarrow \frac{maiorAresta}{2} + D_{max} + tamanhoVoxel$
3: **para** cada vértice $i$ da vista **faça**
4: Marque os *voxels* dentro da esfera centrada no vértice $i$ e com o raio calculado
5: **fim para**
6: Calcule a distância e peso para todos os *voxels* marcados

A estrutura de dados utilizada pelo algoritmo 6.2 para marcar os *voxels* consiste em uma matriz no plano XY de ponteiros para listas de *spans* de *voxels* na coordenada Z, conforme pode ser observado na figura 6.8. A marcação dos *voxels* consiste na manutenção dos *spans*. Para facilitar este processo, a esfera é pré-calculada e é representada como um array 2D de *spans*, que são "mesclados" com os *spans* já existentes de vértices anteriores.

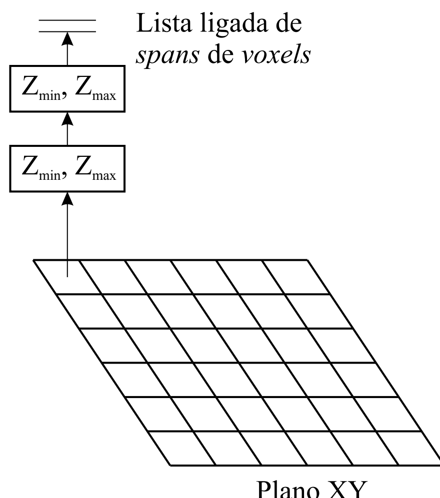



**Figura 6.8:** Estrutura de dados para marcação de *voxels*. Para cada "coluna" de *voxels*, indexada por sua coordenada $(X, Y)$, criamos uma lista ligada de *spans* de *voxels* marcados. Os *spans* estão ordenados na lista, e para cada *span* sabemos o *voxel* inicial de coordenada $Z_{min}$ e o *voxel* final de coordenada $Z_{max}$.

Alguns parâmetros são utilizados para controlar o algoritmo de integração de Curless. Estes parâmetros são especificados no arquivo .SC de *script* utilizado como entrada para a ferramenta *ModelTool*. Na tabela 6.1 podemos observar quais são estes parâmetros, uma breve descrição e os valores usuais (obtidos empiricamente) de cada um deles.

**Tabela 6.1:** Parâmetros do algoritmo de Curless especificado no arquivo *script* de entrada do programa *ModelTool*.

| Parâmetro | Valor Usual | Descrição |
|---|---|---|
| $volVoxelSize$ | 0.5 mm | Tamanho do *voxel* da representação volumétrica[1] |
| $volMaxDist$ | 3.0 mm | Valor máximo de distância para a superfície integrada[2] |
| $volBorderWeight$ | 0.0 | Peso para medidas na borda de vistas |
| $volBorderLen$ | 10 | Tamanho da borda[3] |
| $volBorderDiscard$ | 0.15 | Limiar para descarte de medidas para borda[4] |
| $volZMin$ | 0 | Coordenada Z do início do volume de interesse[5] |
| $volZMax$ | 0x7FFFFFFF | Coordenada Z do final do volume de interesse[5] |

[1]Este valor deveria ser aproximadamente igual à resolução do *scanner*, de forma a não permitir perda de detalhes.

[2]Este valor especifica $-D_{min}$ e $+D_{max}$, que em nossa implementação são iguais (vide algoritmo 6.2).

[3]O tamanho da borda é especificado em "arestas". Para cada vértice, é calculada a distância do vértice até a borda mais próxima, contando o número de arestas que devem ser percorridas para se chegar até a borda da vista.

[4]Para ponto qualquer da vista, calcula-se o seu peso de borda. O peso varia de 1.0 para pontos cuja distância para a borda é maior ou igual a $volBorderLen$, até $volBorderWeight$ na borda. Quaisquer pontos cujo peso seja inferior a $volBorderDiscard$ são descartados. Note-se que desta forma o descarte pode ser de apenas parte de uma face.

[5]Opcionalmente, apenas uma "fatia" do volume pode ser calculada. Esta fatia é paralela ao plano XY, e é especificada pelas duas coordenadas $volZMin$ e $volZMax$.

### 6.2.2 Algoritmo de Wheeler

A grande desvantagem do algoritmo de Curless (seção 6.2.1) é que ele não possui nenhum processo específico para eliminar os *outliers*. Conforme apresentamos, alguns "truques" podem ser utilizados para tentar reduzir o peso destes *outliers*, mas mesmo com peso reduzido, eles ainda acabam interferindo no modelo final. Como o objetivo do nosso trabalho era obter a máxima fidelidade possível, este comportamento do algoritmo de Curless é inaceitável. Isto nos levou a experimentar o algoritmo de Wheeler [155]. Este algoritmo foi utilizado no *"The Great Buddha Project"*, e algumas melhorias ao algoritmo foram sugeridas por Sagawa *et al.* [125, 123, 126, 124].

Da mesma forma que o método de Curless, este algoritmo se utiliza da representação volumétrica com SDF, e extração da malha resultante com o algoritmo *Marching Cubes*. Ele utiliza uma representação via *octrees*, de forma a reduzir o custo computacional e de memória, mas em sua formulação original todos os *voxels* próximo à superfície estão presentes, de forma que o algoritmo *Marching Cubes* pode ser utilizado sem nenhuma modificação. Os artigos de Sagawa [125, 123, 126, 124] aperfeiçoam este algoritmo, eliminando esta restrição de que a *octree* seja dividida até o nível máximo próximo à superfície, mas requer uma adaptação do algoritmo *Marching Cubes*.

O objetivo é tentar eliminar os *outliers* das medições, levando em conta apenas medições consistentes umas em relação às outras. Em seguida, as medidas em consenso são mescladas, de forma a gerar a distância à superfície para o *voxel* em questão. Outro ponto que diferencia este algoritmo daquele de Curless é que aqui as medidas de distância são euclidianas em relação à superfície, ou seja, não levam em conta a posição da lente do scanner, nem o seu eixo de visão.

Para compreender o algoritmo, apresentamos o pseudo-código 6.3. O algoritmo, apesar de parecer extenso, é fácil de ser compreendido. Basicamente, para cada *voxel* de interesse $v$, tentamos encontrar um ponto candidato integrado onde haja um consenso das medidas das várias vistas. Se este ponto em consenso não existir, utilizamos o ponto candidato sem consenso de maior peso (ou confiabilidade) $w$. Os candidatos são integrados levando-se em conta o peso $w'$, que é o mesmo de Curless [37], ou seja, o produto interno entre a normal $n'$ no ponto e o vetor até o ponto de vista do *scanner*. Desta forma, pontos perpendiculares ao *scanner* possuem maior confiabilidade que pontos observados "de raspão".

A idéia do algoritmo é gerar um ponto candidato para cada vista (linhas 3 a 24 do algoritmo 6.3), classificando-o como em consenso ou não (linhas 18 a 23). Após descobertos todos os candidatos, um deles é selecionado (linhas 25 a 29), e chamado de $x$. Finalmente, a distância com sinal do *voxel* é calculada (linhas 30 a 34). O limiar de consenso $\theta$ é testado contra o peso acumulado $w_{acc}$, ou seja, a quantidade de medidas similares em

todas as vistas para o candidato em questão. Note-se que esta similaridade leva em conta a distância e as normais das medidas (linha 11).

**Algoritmo 6.3** Pseudo-código para o algoritmo de Integração de Vistas de Wheeler [155].

1: **para** cada *voxel* de interesse $v$ **faça**
2: Crie 2 conjuntos vazios $C$ (pontos em consenso) e $O$ (pontos sem consenso)
3: **para** cada vista $i$ **faça**
4: $p \leftarrow$ ponto mais próximo de $v$ na superfície da vista $i$
5: $n \leftarrow$ normal de $p$
6: $p_{acc} \leftarrow (0,0,0)$, $n_{acc} \leftarrow (0,0,0)$, e $w_{acc} \leftarrow 0$
7: **para** cada vista $j$ **faça**
8: $p' \leftarrow$ ponto mais próximo de $p$ na superfície da vista $j$
9: $n' \leftarrow$ normal de $p'$
10: $w' \leftarrow$ peso de $p'$ na vista $j$ //peso igual ao de Curless
11: **se** $p'$ estiver suficientemente próximo de $p$ **e** $n'$ for similar à normal $n$ **então**
12: $p_{acc} \leftarrow p_{acc} + w'p'$
13: $n_{acc} \leftarrow n_{acc} + w'n'$
14: $w_{acc} \leftarrow w_{acc} + w'$
15: **fim se**
16: **fim para** //cada vista $j$
17: Calcule o candidato $p \leftarrow \left(\frac{1}{w_{acc}}\right) p_{acc}$, $n \leftarrow \frac{n_{acc}}{\|n_{acc}\|}$ e $w \leftarrow w_{acc}$
18: // $\theta$ é o limiar de consenso
19: **se** $w_{acc} > \theta$ **então**
20: Acrescente $< p, n, w >$ ao conjunto de consenso $C$
21: **senão**
22: Acrescente $< p, n, w >$ ao conjunto de não-consenso $O$
23: **fim se**
24: **fim para** //cada vista $i$
25: **se** $C \neq \emptyset$ **então**
26: $x \leftarrow$ ponto mais próximo de $v$ dentre os candidatos do conjunto $C$
27: **senão**
28: $x \leftarrow$ candidato de $O$ com maior peso $w$
29: **fim se**
30: $d \leftarrow \|v - x\|$ //valor absoluto da distância para o *voxel* $v$
31: **se** $n \cdot (v - x) > 0$ **então**
32: $d \leftarrow -d$ //*voxel* está "fora" do objeto, logo distância é negativa
33: **fim se**
34: $d$ é a distância com sinal do *voxel* $v$
35: **fim para** //cada *voxel* $v$

Também foram efetuados diversos testes com este método, e podemos listar suas vantagens e desvantagens:

- Vantagens:
  - Em regiões com consenso de várias vistas, os *outliers* são eliminados e a superfície integrada é suave;
  - O algoritmo é de fácil implementação.
- Desvantagens:
  - Ele é muito mais pesado do que o algoritmo de Curless [37], pois possui tempo quadrático no número de vistas, enquanto que o algoritmo de Curless possui tempo linear no número de vistas;
  - Ele requer que todas as vistas estejam carregadas na memória simultaneamente, o que é impraticável para objetos maiores, que requerem muitas vistas para que sua geometria seja capturada completamente;
  - Não leva em consideração o espaço vazio entre o sensor do *scanner* e a superfície do objeto, informação esta que pode ser útil no preenchimento de buracos e detecção de *outliers*;
  - Em trechos do objeto abaixo do limiar $\theta$ de consenso, a superfície resultante é muito ruim, sendo muito danificada pelos *outliers* e por pequenos erros de alinhamento. Estas regiões são muito comuns na vizinhança de buracos no objeto integrado;
  - A magnitude e o sinal da distância são calculados incorretamente em torno das bordas das vistas;
  - O algoritmo acaba tentando preencher buracos automaticamente, com resultados que variam de regulares a péssimos.

Infelizmente, apesar de promissor na teoria, na prática o algoritmo de Wheeler se mostrou pior que o de Curless. Podemos comparar os resultados dos dois algoritmos na figura 6.9. Para entender o porquê deste comportamento ruim do algoritmo de Wheeler, vamos analisar com mais detalhes as desvantagens listadas anteriormente.

Toda vista de um objeto é incompleta por definição. Logo, sempre existirão bordas nas vistas. E o problema é que a distância até a superfície é mal-definida para bordas, pois as bordas representam a falta de informação. Podemos entender melhor isto através da figura 6.10. Este problema consiste no fato do algoritmo se utilizar de distâncias euclidianas até o modelo 3D da vista. Quanto o ponto mais próximo do *voxel* se encontra em uma borda da vista, o que ocorre é que a magnitude da distância acaba sendo maior do que deveria ser (deveria ser calculada até a linha tracejada, que é o prolongamento da superfície). Além disso, acabamos tendo *voxels* vizinhos, com sinais diferentes, e com

uma grande diferença de magnitude, que acaba fazendo com que a superfície reconstruída seja criada em uma posição arbitrária pelo algoritmo *Marching Cubes*.

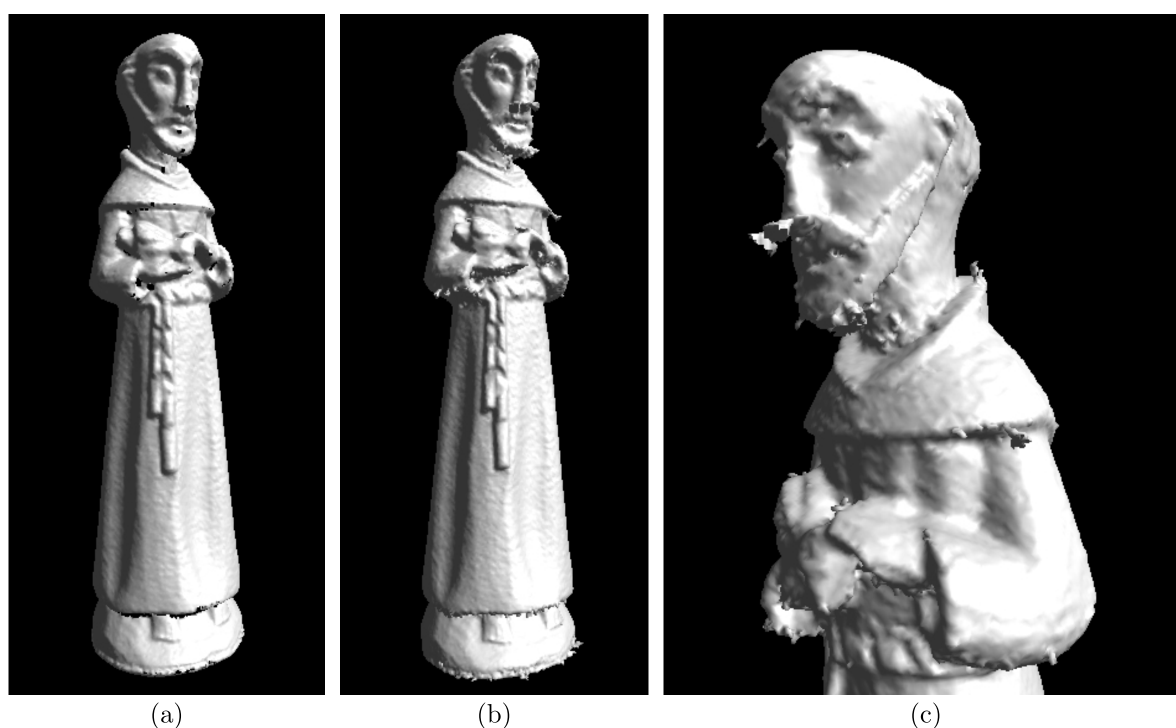



**Figura 6.9:** Comparação entre os algoritmos de Curless (a) e Wheeler (b). Ambos os algoritmos foram executados com o preenchimento de buracos desativado. Nota-se que os maiores problemas do algoritmo de Wheeler ocorreram nas regiões próximas a buracos. (c) Detalhe do modelo integrado com o algoritmo de Wheeler.

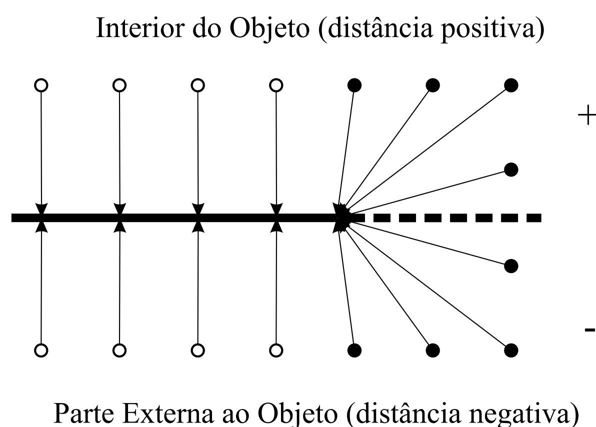



**Figura 6.10:** Magnitude e sinais de distâncias para bordas de vistas no algoritmo de Wheeler. A superfície da vista é representada pela linha contínua, e seu prolongamento (apenas encontrado em outras vistas) pela linha tracejada. Os *voxels* são representados por círculos, e as setas representam as medidas de distância dos *voxels* até a superfície da vista. Nota-se que os *voxels* representados com círculos preenchidos possuem distâncias com magnitude incorreta (maior que a real), pois a superfície continuaria na linha tracejada. Mas o maior problema é a transição abrupta de sinais: Os dois círculos mais próximos da linha tracejada possuem uma magnitude grande e sinais opostos, apesar de se localizarem, na realidade, próximos à superfície real.

Este problema faz com que, se tivermos sorte, estas medidas sejam consideradas sem consenso e desconsideradas. No entanto, se não houver outras vistas da mesma região, ou pior, se as vistas não atingirem o limiar de consenso $\theta$, a superfície resultante é de péssima qualidade, pois tenta preencher os buracos prolongando as superfícies, com magnitudes incorretas, ou misturando medidas de várias vistas, conforme a vista que possuir a maior confiabilidade $w$ em cada ponto. Isto pôde ser observado na figura 6.9.

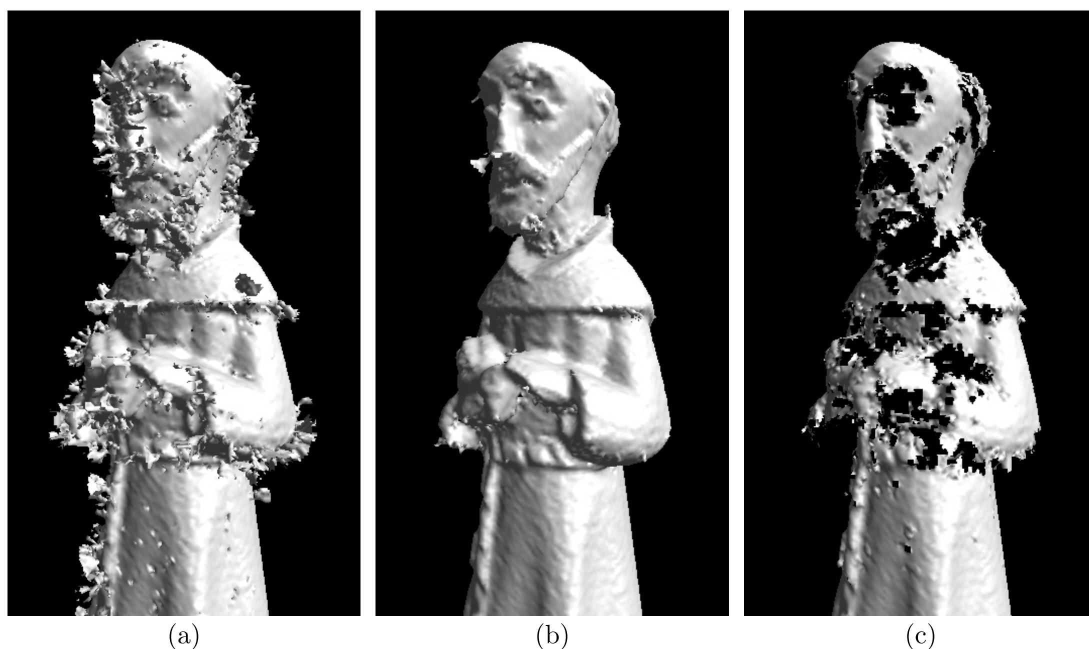
Como podemos perceber na figura 6.9(b), nas regiões do meio do objeto (região do cinto e túnica), como existem várias vistas que capturaram a mesma superfície, os *outliers* foram efetivamente eliminados, e a superfície é relativamente uniforme. No entanto, na figura 6.9(c), percebemos que em regiões de difícil captura por *scanners* de triangulação a laser (região do queixo, por exemplo), onde houve poucas ou nenhuma vista, a superfície deixa de ser suave, e artefatos advindos do incorreto preenchimento de buracos inerente ao algoritmo geram um resultado extremamente insatisfatório.

Mais preocupante que os "brotamentos" e saliências, que poderiam ser identificados e eliminados, é a superfície incorretamente gerada, como aparece na região do pescoço da figura 6.9(c). Ela deveria ser lisa, mas apareceram irregularidades, devido ao fato de *outliers* não terem sido corretamente desprezados, ou pior, *outliers* terem sido "introduzidos" pelo preenchimento de buracos inerente ao algoritmo.

A primeira hipótese para tentar explicar estes defeitos é o uso de superfícies sem consenso junto com superfícies com consenso, o que o algoritmo permite. No entanto, após executado o algoritmo ignorando as superfícies sem consenso, o resultado obtido foi o mesmo da figura 6.9(b). O que percebemos em todos os nossos testes é que raramente o algoritmo utiliza superfícies sem consenso; logo, deve existir outra explicação para os defeitos apresentados, pois eles resultam de "consensos" das várias vistas.

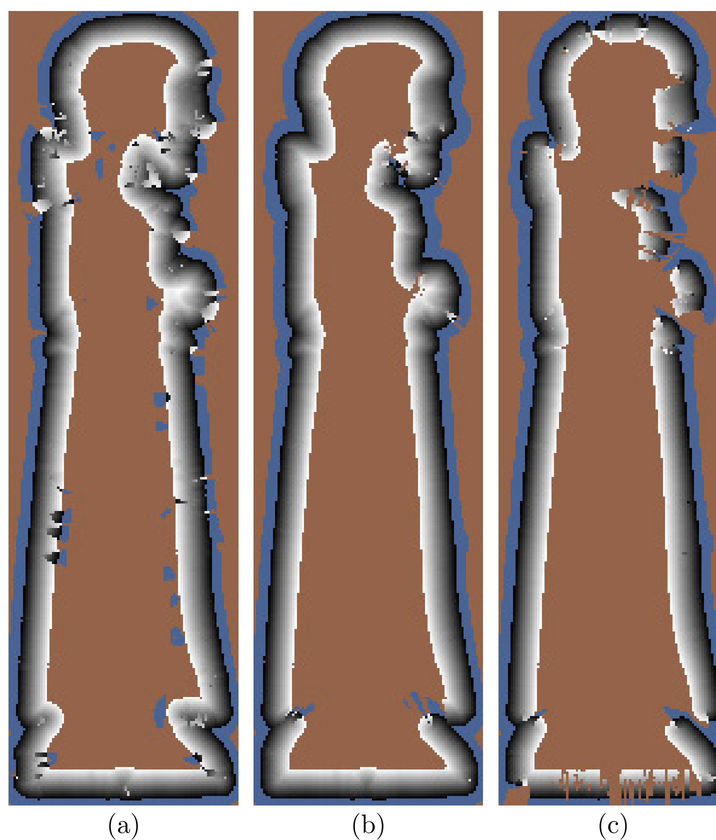
A tentativa seguinte foi aumentar o limiar $\theta$ de consenso. Na figura 6.11 vemos o objeto reconstruído com vários limiares diferentes, em todos os casos utilizamos apenas os pontos em consenso na reconstrução.

Incrivelmente, mesmo para o maior limiar, que deveria deixar apenas superfícies mais confiáveis, o objeto resultante também é "defeituoso". Isto nos fez perder muito tempo procurando por erros de implementação, que não existiam. Até que finalmente percebemos que o problema advinha das distâncias calculadas até as bordas das vistas (vide figura 6.10). Isto pode ser melhor observado em cortes da representação volumétrica, apresentados na figura 6.12.



(a) (b) (c)

**Figura 6.11:** Objeto reconstruído pelo algoritmo de Wheeler com vários limiares de consenso. (a) $\theta = 0.0$; (b) $\theta = 2.0$; (c) $\theta = 5.0$. Apenas medidas em consenso foram utilizadas nas reconstruções.

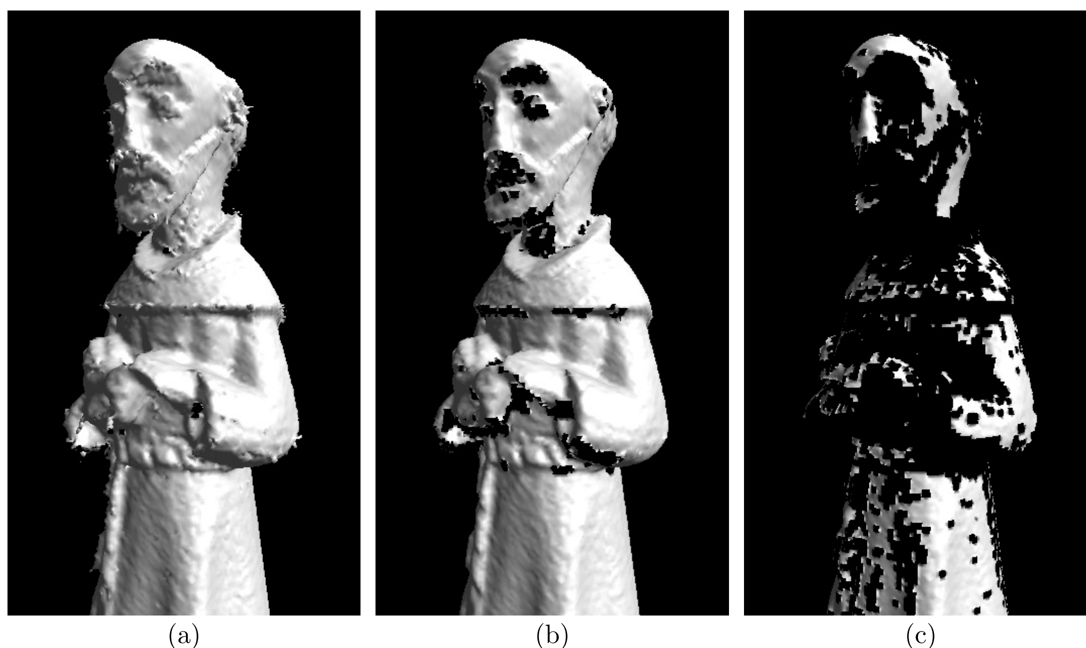


(a) (b) (c)

**Figura 6.12:** Cortes da representação volumétrica do algoritmo de Wheeler com vários limiares de consenso. (a) $\theta = 0.0$; (b) $\theta = 2.0$; (c) $\theta = 5.0$. Apenas medidas em consenso foram utilizadas nas reconstruções, e estes cortes correspondem aos modelos da figura 6.11.

Com um limiar maior, acabam existindo muito mais “bordas” no objeto reconstruído. Podemos ainda observar que próximo às bordas, a função de distância fica muito instável, com magnitudes incorretas e trocas abruptas de sinal. Isto faz com que a superfície resultante seja muito defeituosa. Inclusive, na versão com $\theta = 0.0$, como na prática o algoritmo considera que não existem *outliers*, os *outliers* (que existem) geram bordas que estragam totalmente a representação volumétrica. No artigo de Sagawa [123], é sugerido um processo de troca de sinais, de forma que *voxels* vizinhos fiquem mais consistentes entre si. No entanto, podemos perceber que esta solução é apenas paliativa, pois como podemos ver nos cortes da figura 6.12 o preenchimento de buracos intrínseco do algoritmo gera resultados muito incorretos, e não basta conciliar sinais para gerar superfícies de boa qualidade; um algoritmo mais inteligente de preenchimento de buracos é necessário.

Para comprovar estas observações, nas figuras 6.13 e 6.14 vemos o mesmo objeto com os mesmos limiares, agora desconsiderando distâncias até bordas, e também utilizando apenas medidas em consenso. Podemos ver que agora os resultados são mais coerentes: Quanto maior o limiar, mais confiáveis as superfícies resultantes. Obviamente, quanto maior o limiar, mais buracos acabam surgindo na superfície reconstruída, mas pelo menos as superfícies geradas são “corretas”. E quando o limiar é muito baixo ($\theta = 0.0$), *outliers* não são descartados e estragam a reconstrução, mas mesmo assim, tanto a representação volumétrica quanto o modelo gerado são muito mais corretos do que na versão original do algoritmo.



(a) (b) (c)

**Figura 6.13:** Objeto reconstruído pelo algoritmo de Wheeler com vários limiares de consenso, descartando medidas para bordas de vistas. (a) $\theta = 0.0$; (b) $\theta = 2.0$; (c) $\theta = 5.0$. Apenas medidas em consenso foram utilizadas nas reconstruções.

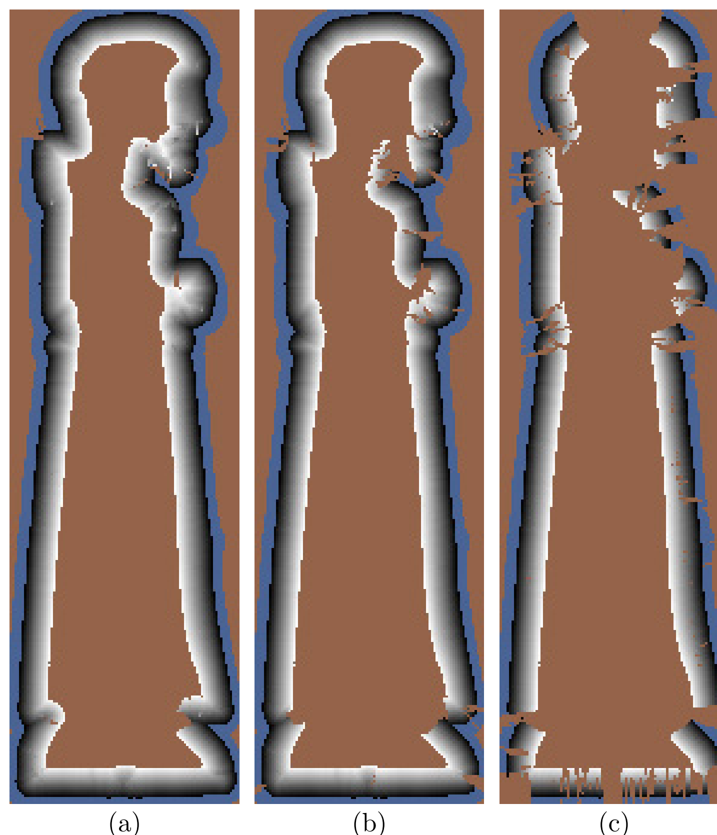
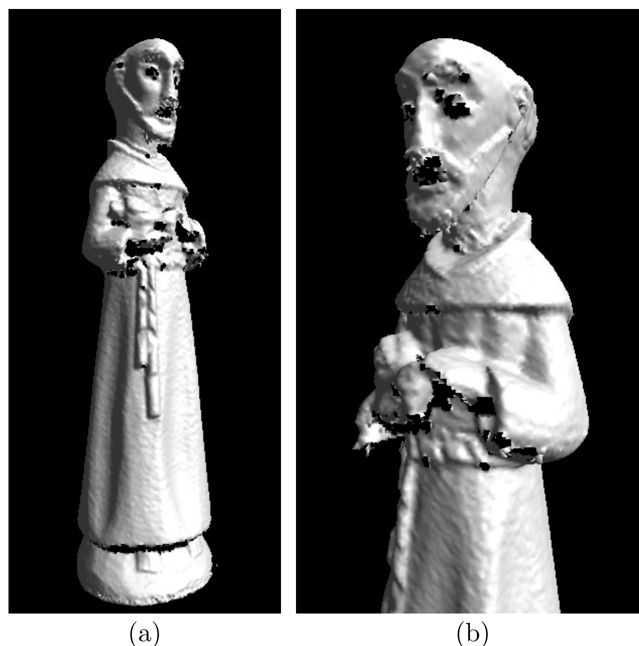



**Figura 6.14:** Cortes da representação volumétrica do algoritmo de Wheeler com vários limiares de consenso, descartando medidas para bordas de vistas. (a) $\theta = 0.0$; (b) $\theta = 2.0$; (c) $\theta = 5.0$. Apenas medidas em consenso foram utilizadas nas reconstruções, e estes cortes correspondem aos modelos da figura 6.13.

No entanto, ainda estamos longe de uma solução satisfatória. Ainda ocorrem *voxels* com sinal "trocado", uma vez que a estimativa de normais do algoritmo não garante resultados 100% corretos em todos os casos. Além disso, o grande problema é que acabamos descartando informações úteis. Regiões abaixo do limiar de menor consenso acabam gerando buracos no modelo, e seria interessante que as informações capturadas fossem utilizadas nestas regiões. Imaginamos que este era o objetivo do algoritmo original, que também utilizava regiões sem consenso. Logo, voltamos a habilitar o uso de superfícies sem consenso (mas continuando a descartar medidas até bordas de vistas), e novamente uma surpresa: O resultado continuou praticamente o mesmo, ou seja, buracos não foram preenchidos, e acabaram surgindo *outliers* indesejáveis, como podemos ver na figura 6.15.

Após analisar estes resultados, chegamos ao âmago do problema: o critério de "consenso" não funciona como o esperado. Se existe um consenso "distante", ele acaba tendo prioridade sobre "não-consensos" mais próximos, e que representam a superfície real do objeto. Podemos entender melhor isto na figura 6.16. Este problema ocorre nos olhos do objeto da figura 6.15(b). O algoritmo acaba escolhendo um consenso na região da nuca,

e ignora o "não-consenso" dos olhos, que seria o mais correto; e isto acaba gerando um buraco nos olhos. Infelizmente, existe o problema inverso, isto é, os não-consensos podem incluir *outliers*, que acabam estragando a reconstrução; isto pode ser observado em outras regiões da figura 6.15.



(a) (b)

**Figura 6.15:** Objeto reconstruído pelo algoritmo de Wheeler, descartando medidas para bordas de vistas, e utilizando medidas sem consenso. (a) Visão geral do objeto, que pode ser comparada com a figura 6.9(b); (b) Detalhe do objeto, que pode ser comparado com as figuras 6.11(b) e 6.13(b). Utilizamos $\theta = 2.0$.

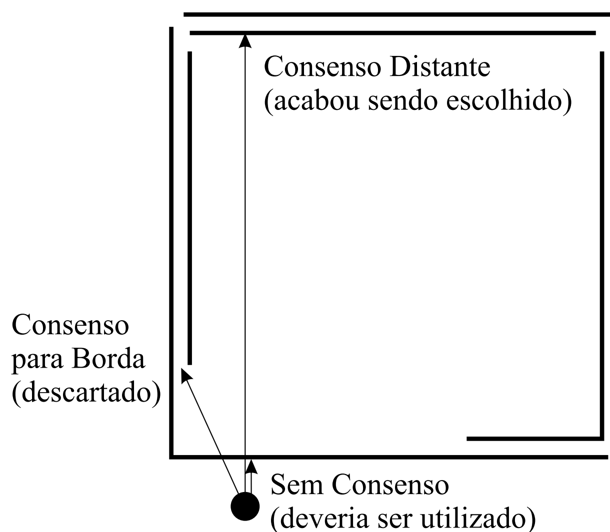



**Figura 6.16:** Falha no critério de consenso. Ao invés de utilizar a medida sem consenso, que seria a mais correta, o algoritmo prefere escolher uma medida com consenso, que pode estar localizada arbitrariamente longe do *voxel* em questão. Mesmo que não fosse utilizado o critério de descartar consensos para bordas, a medida retornada continuaria sendo incorreta.

Poderia ser argumentado que para resolver este problema, bastaria reduzir o limiar de consenso $\theta$, porque daí a região dos olhos estaria em "consenso". No entanto, quanto menor o limiar $\theta$, mais *outliers* acabam deixando de serem eliminados, o que novamente compromete a qualidade da reconstrução. Em outras palavras, limiares grandes não funcionam, e limiares pequenos também não. Um sistema de limiares adaptativos poderia ser imaginado, mas de qualquer forma a fidelidade da reconstrução é afetada.

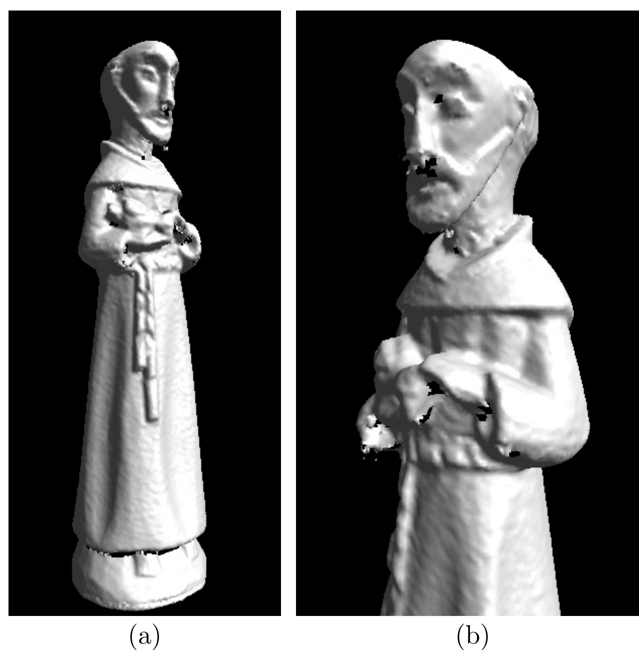
As figuras 6.10 e 6.16 apresentam os defeitos intrínsecos do algoritmo de Wheeler: primeiro, distâncias para superfícies são mal-definidas quando calculadas para bordas de vistas; segundo, mesmo descartando estas medidas mal-definidas, o critério de consenso acaba falhando em lugares onde não há número suficiente de vistas. A conclusão que chegamos é que o algoritmo apenas poderia ser utilizado em objetos onde **todas** as superfícies fossem capturadas diversas vezes, de forma que **todas** elas ficassem acima do limiar de consenso $\theta$. E na prática, isto é muito difícil de ser conseguido, principalmente próximo a regiões em oclusão.

Chegamos a tentar restringir a distância máxima para candidatos de cada *voxel*, numa metodologia similar ao algoritmo de Curless, que limitava a função de distância entre $-D_{min}$ e $+D_{max}$. Nossa esperança era que consensos distantes, como os apresentados na figura 6.16, fossem desconsiderados. Infelizmente, o resultado obtido foi apenas ligeiramente melhor do que o da figura 6.15, ou seja, **longe de ser aceitável**. Esta tentativa pode ser observada na figura 6.17. Convém notar que este foi o melhor resultado que conseguimos obter com o algoritmo de Wheeler, depois de muitas modificações. Convém compará-la com a versão original do algoritmo, mostrada nas figuras 6.9(b) e 6.9(c).

Finalmente, devemos considerar a melhoria do algoritmo de Wheeler proposta por Sagawa [124]. Ele sugere em seu artigo que os sinais dos SDFs sejam iterativamente modificados, de forma a se estabilizarem em um consenso. Realmente, podemos perceber que uma boa parte dos defeitos surgem devido aos sinais incorretos do SDF. No entanto, somos céticos em relação à solução de Sagawa, pois ela ignora a magnitude do SDF, se concentrando apenas em "inverter" sinais, e ela também assume que o método de consenso elimina eficazmente os *outliers*. Conforme demonstramos na figura 6.16, usar apenas dados em consenso podem gerar valores totalmente incorretos de SDFs, tanto em termos de magnitude quanto em termos de sinais. Desta forma, mesmo que os sinais do SDF sejam reparados, seus valores continuam errados, e não temos como garantir que a reconstrução tenha alta precisão.

Isto pode ser observado no próprio artigo de Sagawa, onde um exemplo de modelo reconstruído (vide figura 6.18) possui bordas totalmente irregulares, que não correspondem ao objeto original. No mesmo artigo também podemos observar uma reconstrução do Buddha de Kamakura, que também apresenta vários problemas. Logo nossa avaliação

do método de Sagawa é que, apesar de conseguir gerar modelos mais "aceitáveis", este algoritmo não consegue garantir a precisão dos objetos reconstruídos.



(a) (b)

**Figura 6.17:** Objeto reconstruído pelo algoritmo de Wheeler, descartando medidas para bordas de vistas, utilizando medidas sem consenso, e com a distância de candidatos limitada à faixa $-D_{min}$ a $+D_{max}$. (a) Visão geral do objeto; (b) Detalhe do objeto reconstruído. Utilizamos $\theta = 2.0$, como na figura 6.15. *Outliers* continuam presentes no objeto final.

**Figura 6.18:** Resultado do algoritmo de integração de Sagawa [124]. As bordas reconstruídas são de baixa qualidade, e a superfície não é suave.

Considerando todos estes problemas, chegamos à conclusão que os algoritmos de Wheeler [155] e Sagawa [124], apesar de promissores em teoria, são inferiores ao de Curless [37]

na prática. E como o algoritmo de Curless também possui falhas críticas, torna-se necessário um algoritmo diferente para que possamos integrar e reconstruir modelos 3D com alta qualidade. Um dos objetivos do nosso trabalho foi tentar combinar os pontos fortes de ambos os algoritmos, gerando um novo método que fosse mais satisfatório, tanto em termos de desempenho quanto de precisão nos resultados. O resultado que obtivemos será apresentado na seção 6.2.3.

#### 6.2.2.1 Detalhes de Implementação

Nossa implementação do algoritmo de Wheeler possui algumas características particulares. Como queríamos compará-lo com o algoritmo de Curless, utilizamos as mesmas estruturas de dados para armazenar os *voxels*. Desta maneira, não utilizamos *octrees* como sugerido por Wheeler; no entanto, esta alteração não deveria alterar o resultado obtido, pois a *octree* de Wheeler era subdividida até o nível máximo próximo à superfície do objeto, e nossa representação também possui uma subdivisão uniforme.

O mesmo não pode ser dito para a variação sugerida por Sagawa [123], que intencionalmente evitava a subdivisão da *octree* em regiões relativamente homogêneas. Ele faz isso para reduzir a quantidade de triângulos do modelo final. Em nossa metodologia, preferimos efetuar esta redução poligonal na última etapa do *pipeline*, de forma que tenhamos uma densidade uniforme de vértices para as etapas de texturização do modelo reconstruído.

Na prática, esta nossa abordagem acentua os defeitos do algoritmo de Wheeler e Sagawa, pois as regiões próximas a buracos acabam possuindo uma boa resolução, demonstrando o efeito destrutivo dos *outliers* e dos defeitos apresentados na seção 6.2.2.

Um dos pontos-chave do algoritmo de Wheeler consiste no cálculo da distância entre um ponto (que pode ser um *voxel* ou um ponto candidato) e o modelo 3D de uma vista. Para efetuar este cálculo de maneira relativamente eficiente, utilizamos *kd-trees* das vistas (já disponíveis devido ao processo de alinhamento). A idéia é localizar o vértice da vista mais próximo do ponto em questão, utilizando a *kd-tree*. Em seguida, é feito o cálculo da distância entre o ponto e todas as faces de qual o vértice mais próximo participa. A menor das distâncias encontradas corresponde à distância entre o ponto especificado e o modelo 3D da vista. O algoritmo mais eficiente para calcular a distância de um ponto a um triângulo pode ser encontrado no livro de Eberly [43].

Além de encontrar a distância, este método estima a normal no modelo 3D da vista, interpolando as normais dos vértices. Outra informação obtida é se o ponto mais próximo na vista está localizado em uma borda, para eventualmente descartar esta medida.

Um grande problema do algoritmo é o seu desempenho. Como ele precisa comparar candidatos de cada vista com todas as outras vistas, isto nos obriga a ter todas elas na memória simultaneamente. Infelizmente, isto só é possível para objetos pequenos, que necessitam de poucas vistas. Para objetos maiores, onde todas as vistas não cabem na memória, o *cache* de vistas acaba sofrendo *thrashing*, o que piora consideravelmente o desempenho.

Vários truques foram usados na implementação para reduzir este custo, como por exemplo: descobrir inicialmente quais vistas são próximas de cada *voxel*, para limitar o número de vistas a serem consideradas, utilizando o algoritmo 6.2; construir um *cache* de candidatos, de forma a acumular vários candidatos ao mesmo tempo, reduzindo o número de trocas de vistas no *cache* de vistas. Infelizmente, mesmo com todos os esforços, o algoritmo continua muito mais lento que o de Curless, pois sua complexidade continua sendo quadrática no número de vistas. Em seu artigo [123], Sagawa utiliza *clusters* de computadores para executar o algoritmo, devido ao seu baixo desempenho.

Alguns parâmetros são utilizados para controlar o algoritmo de Wheeler. Estes parâmetros são especificados no arquivo .SC de *script* utilizado como entrada para a ferramenta *ModelTool*. Na tabela 6.2 podemos observar quais são estes parâmetros, uma breve descrição e os valores usuais (obtidos empiricamente) de cada um deles.

**Tabela 6.2:** Parâmetros do algoritmo de Wheeler especificado no arquivo *script* de entrada do programa *ModelTool*.

| Parâmetro | Valor Usual | Descrição |
|---|---|---|
| $miscVolCacheSize$ | 500 MB[1] | Tamanho do *cache* de candidatos |
| $volConsensusDist$ | 0.5 mm | Limiar de distância para teste de compatibilidade[2] |
| $volConsensusAngle$ | 30.0° | Limiar de ângulo entre normais para teste de compatibilidade[2] |
| $volConsensusWeight$ | 2.0 | Limiar $\theta$ de consenso[3] |

## 6.2.3 Algoritmo Híbrido

Pela análise dos algoritmos de Curless [37] e Wheeler [155], podemos ver que ambos possuem falhas teóricas, o que se reflete em problemas nos modelos integrados. Além disso, nenhum dos dois algoritmos consegue eliminar de forma efetiva os diversos tipos de defeitos encontrados em imagens de profundidade reais. Assim sendo, nossos esforços se concentraram em tentar encontrar uma solução que "funcionasse".

[1]Valor usual para um sistema com 4GB de RAM.
[2]Vide linha 11 do algoritmo 6.3.
[3]Vide linha 19 do algoritmo 6.3.

A maior dificuldade é encontrar uma maneira de detectar e desconsiderar os diversos tipos de *outliers*. Conseguimos desenvolver um algoritmo que, apesar de ainda não solucionar completamente o problema, consegue detectar de forma automática a maior parte dos defeitos, e eliminá-los.

A idéia básica do nosso algoritmo é fazer a fusão em duas fases. A primeira fase cria uma representação volumétrica do modelo integrado com o algoritmo de Curless, com algumas modificações; na segunda, efetuamos a integração definitiva, utilizando a representação da primeira fase para detectar e desconsiderar *outliers* durante a construção da representação volumétrica definitiva.

Durante a primeira fase, nós efetuamos a operação de *space-carving*, comentada na seção 3.3.2 e mostrada na figura 3.10. A idéia é aproveitar ao máximo as informações retornadas pelo *scanner* de triangulação laser. Como sabemos a posição da lente do *scanner*, além da superfície capturada do objeto, também sabemos que o volume entre esta superfície e a posição da lente é vazio. Assim sendo, quaisquer *outliers* das outras vistas encontrados nestas regiões vazias podem ser descartados. Além disso, como comentamos na seção 3.4.1, o plano de apoio também pode ser utilizado para definir volumes vazios - todo o volume abaixo do plano sabidamente não pertence ao objeto, e pode ser combinado com os outros volumes vazios extraídos das vistas. O pseudo-código da primeira fase é apresentado no algoritmo 6.4.

O objetivo do algoritmo 6.4 é construir uma representação volumétrica aproximada, que será salva em $volCopy$, através do algoritmo melhorado de Curless que apresentamos na seção 6.2.1. Além disso, um volume binário *empty* é criado, e ele representa o *space-carving* do algoritmo. Para cada *voxel*, existe um bit correspondente no volume *empty*; se o bit estiver valendo 1, é porque aquele *voxel* é considerado vazio por alguma vista ou plano de apoio.

As linhas 1-17 do algoritmo 6.4 correspondem ao algoritmo modificado de Curless com *space-carving*. Como a função de distância calculada na linha 7 é sobre a linha de visão do *scanner*, quaisquer distâncias negativas são consideradas "vazias" e o bit correspondente de *empty* é setado.

Depois de percorridas todas as vistas, o volume binário *empty* sofre uma operação de morfologia matemática de erosão na linha 18. A máscara utilizada corresponde à uma esfera com raio $+D_{max}$. Esta operação é feita por dois motivos: primeiro, para evitar que alguma medida incorreta "cave buracos" no objeto. Apesar de incomum, já observamos este tipo de erro em algumas imagens de profundidade. O outro motivo é o fato de *empty* ser obtido pela união dos espaços vazios de todas as vistas. Desta forma, ele tende a representar a menor medida encontrada para cada ponto da superfície, e

não uma medida média. Reduzindo o volume vazio por uma distância de $+D_{max}$, nós reservamos um espaço próximo à superfície para integrar as medidas das várias vistas, e ao mesmo tempo mantemos uma representação do volume vazio para a eliminação de *outliers* afastados da superfície. Muitos dos defeitos apresentados na seção 3.3 conseguem ser eliminados por este volume binário erodido *empty*, o que é feito entre as linhas 19-23.

**Algoritmo 6.4** Pseudo-código para a 1ª fase do algoritmo Híbrido de Integração.

**Requer:** $|-D_{min}| = |+D_{max}|$ //usa-se o mesmo limite para medidas positivas e negativas
1: Inicialize um volume binário *empty* com todos os bits zerados
2: **para** cada vista $i$ **faça**
3:   **se** a vista $i$ tiver um plano de apoio **então**
4:     Marque todos os bits de *empty* abaixo do plano com 1
5:   **fim se**
6:   **para** cada *voxel* $v$ **faça**
7:     Calcule a distância com sinal $d$ e o peso $w$ do *voxel* $v$ para a vista $i$ pelo método de Curless
8:     **se** $d < 0$ **então**
9:       Marque o bit correspondente a $v$ de *empty* como 1
10:     **fim se**
11:     **se** $d < -D_{min}$ **então**
12:       $d \leftarrow -D_{min}$
13:     **fim se**
14:     $w \leftarrow w$ multiplicado pela curva de peso da figura 6.6(b)
15:     Atualize a representação volumétrica com $d$ e $w$ e a fórmula de Curless mostrada no algoritmo 6.1
16:   **fim para** //cada *voxel* $v$
17: **fim para** //cada vista $i$
18: Faça uma erosão do volume binário *empty* com uma esfera de raio $+D_{max}$
19: **para** cada *voxel* $v$ **faça**
20:   **se** bit correspondente a $v$ de *empty* valer 1 **então**
21:     Marque $v$ como "vazio" (distância $\leftarrow -D_{min}$ e peso $\leftarrow 0$) //apaga *outliers*
22:   **fim se**
23: **fim para**
24: Faça um "*blur*" no volume, misturando as medidas de cada *voxel* com seus 26 vizinhos
25: Salve a representação volumétrica em *volCopy*
26: **para** cada *voxel* $v$ **faça**
27:   **se** bit correspondente a $v$ de *empty* valer 1 **então**
28:     Marque $v$ como "vazio" (distância $\leftarrow -D_{min}$ e peso $\leftarrow 0$)
29:   **senão**
30:     Marque $v$ como "desconhecido" (distância $\leftarrow +D_{max}$ e peso $\leftarrow 0$)
31:   **fim se**
32: **fim para**

Na linha 24 constatamos uma outra operação "estranha", que consiste em suavizar a representação volumétrica. Um filtro de $3\times3\times3$ *voxels* é utilizado, sendo que quanto mais

próximos do *voxel* central, maior o peso dos elementos na máscara. Esta filtragem serve para completar *voxels* individuais sem medida com valores plausíveis, e para atenuar ligeiramente o ruído e *outliers* da superfície. Esta atenuação é importante, pois esta representação volumétrica será utilizada como estimativa das normais do objeto integrado final na segunda fase do algoritmo. Nas linhas 25-32 a representação volumétrica é salva em $volCopy$ e reinicializada para a 2ª fase do algoritmo, cujo pseudo-código é apresentado no algoritmo 6.5.

**Algoritmo 6.5** Pseudo-código para a 2ª fase do algoritmo Híbrido de Integração.

**Requer:** $|-D_{min}| = |+D_{max}|$ //usa-se o mesmo limite para medidas positivas e negativas
**Requer:** volume binário *empty* e representação volumétrica $volCopy$ da 1ª fase
1: **para** cada vista $i$ **faça**
2: Descubra os *voxels* próximos da vista $i$ com o algoritmo 6.2
3: **para** cada *voxel* próximo $v$ **faça**
4: **se** bit correspondente a $v$ de *empty* valer 0 **então**
5: $n \leftarrow$normal estimada para o *voxel* $v$ em $volCopy$
6: **se** $discardNoNormal$ for verdadeiro **e** $n$ não pode ser estimada **então**
7: **continue** no próximo *voxel* $v$
8: **fim se**
9: $p' \leftarrow$ ponto mais próximo de $v$ na superfície da vista $i$ //como em Wheeler
10: $n' \leftarrow$ normal de $p'$
11: $w \leftarrow$ peso de $p'$ na vista $i$ //peso igual ao de Curless e Wheeler
12: $d \leftarrow \|v - p'\|$ //valor absoluto da distância para o *voxel* $v$
13: **se** $n' \cdot (v - p') > 0$ **então**
14: $d \leftarrow -d$ //*voxel* está "fora" do objeto, logo distância é negativa
15: **fim se**
16: **se** ($p'$ estiver em uma borda) **ou** ($d > +D_{max}$) **ou** (existe $n$ **e** o ângulo entre $n$ e $n' > volConsensusAngle$) **então**
17: **continue** no próximo *voxel* $v$
18: **fim se**
19: $w \leftarrow w$ * peso de borda * peso do ângulo * peso da distância
20: **se** $d < -D_{min}$ **então**
21: $d \leftarrow -D_{min}$
22: **fim se**
23: Atualize a representação volumétrica com $d$ e $w$ e a fórmula de Curless mostrada no algoritmo 6.1
24: **fim se**
25: **fim para** //cada *voxel* próximo $v$
26: **fim para** //cada vista $i$
27: Elimine *voxels* com medidas de distância incompatíveis com seus vizinhos

A segunda fase do algoritmo efetua a integração definitiva, descartando *outliers*. Esta etapa possui alguns elementos do algoritmo de Wheeler, como o cálculo da distância, que agora é euclidiana. Utilizamos esta métrica para evitar as superfícies ligeiramente deslocadas que são obtidas quando se utiliza a distância de Curless. Além disso, a distância eucli-

diana melhora o algoritmo de preenchimento de buracos, que será discutido no capítulo 7.

Apenas os *voxels* próximos das vistas (descobertos pelo algoritmo 6.2), e que não são considerados vazios pelo volume binário *empty* são calculados. O algoritmo é linear no número de vistas, ao contrário do algoritmo de Wheeler, que é quadrático. Obviamente, a complexidade de todos os algoritmos também depende do número de voxels calculados.

Na linha 5 do algoritmo 6.5 uma normal estimada $n$ é calculada para o *voxel*, a partir da representação volumétrica $volCopy$ obtida na primeira fase. Esta normal será utilizada para validar os dados de cada vista, descartando medidas incorretas e *outliers*. A normal $n$ é calculada através dos gradientes $(x, y, z)$ do SDF $volCopy$. Se a magnitude do gradiente for menor que $\frac{voxelSize}{3}$, ou se o *voxel* não tiver vizinhos válidos, a normal $n$ não pode ser estimada. Existem duas possibilidades neste caso: se o parâmetro booleano $discardNoNormal$ for verdadeiro, o *voxel* é pulado, e continuará como “desconhecido”, sendo processado pela etapa de preenchimento de buracos do *pipeline*; se $discardNoNormal$ for falso, este *voxel* específico não será validado pela normal $n$, o que pode eventualmente levar à aceitação de *outliers* neste *voxel*. Um dos raros casos que necessita do parâmetro $discardNoNormal$ como falso é para objetos com partes de pouca espessura, e com tamanhos de *voxel* muito grandes. Nestes casos, se $discardNoNormal$ fosse verdadeiro, muitos buracos seriam criados nas superfícies pouco espessas, o que piorava o resultado global. Este parâmetro é considerado entre as linhas 6-8.

As linhas 9-15 calculam o ponto mais próximo $p'$ do *voxel* $v$ em questão (para a vista $i$), bem como a normal $n'$ no ponto $p'$ da vista, o peso $w$ (confiabilidade) desta medida, e a distância com sinal $d$. Estes valores são calculados como no algoritmo de Wheeler, apresentado na seção 6.2.2. Para isso utiliza-se a *kd-tree* da vista $i$, e o método do cálculo de distância de um ponto a um triângulo do livro de Eberly [43]. Outra informação obtida é se o ponto $p'$ pertence à uma borda da vista $i$.

A parte mais importante do algoritmo se localiza na linha 16. É neste momento que é considerada a validade da medida para a vista $i$. Se $p'$ estiver localizado em uma borda, a medida é descartada. Isto é feito para desconsiderar distâncias incorretas, conforme explicamos na seção 6.2.2 e na figura 6.10. Medidas maiores que o limite máximo $+D_{max}$ também são desconsideradas. E finalmente, se conseguimos calcular a normal $n$ na linha 5, calculamos o ângulo entre $n$ e a normal $n'$ no ponto $p'$. Se o ângulo for maior que a tolerância $volConsensusAngle$, a medida também é desconsiderada. Isto acaba resolvendo o grave problema do algoritmo de Curless, que é o dos ressaltos próximos a cantos, demonstrados nas figuras 6.3 e 6.4. Além disso, *outliers* também são eliminados por este teste.

Outro ponto fundamental do algoritmo se localiza na linha 19. Nele, a confiabilidade

básica $w$ da medida (que depende do ângulo entre o *scanner* e a normal $n'$) é alterada por outros 3 fatores.

O primeiro fator é o peso de borda, onde medidas próximas à borda da vista possuem peso menor do que medidas no interior da vista. Devemos notar que o peso da borda é analisado para o descarte da linha 16, conforme apresentado na tabela 6.1.

O segundo fator é o peso do ângulo entre as normais $n$ e $n'$. A fórmula deste fator de peso, que varia de 0.0 a 1.0, pode ser visto na equação 6.1.

$$w_{angle} = \frac{(n \cdot n') - \cos(volConsensusAngle)}{1.0 - \cos(volConsensusAngle)} \tag{6.1}$$

onde $n$ e $n'$ são vetores normalizados, logo $n \cdot n'$ vale o cosseno do ângulo entre $n$ e $n'$.

O valor $n \cdot n'$ é utilizado na linha 16 para o descarte da medida, logo $1.0 \geq n \cdot n' \geq \cos(volConsensusAngle)$.

Finalmente, o terceiro fator modificador de $w$ varia conforme a distância com sinal $d$, similarmente às curvas de peso do algoritmo de Curless, mostradas na figura 6.6. No algoritmo híbrido, utilizamos uma nova curva de peso, representada na figura 6.19. Esta curva garante uma integração suave e homogênea da função de distância.

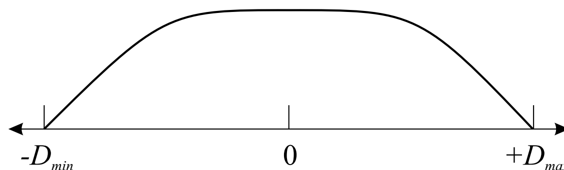



**Figura 6.19:** Perfil de aplicação de peso conforme a distância no algoritmo Híbrido. Este perfil define um fator entre 0.0 e 1.0 que é multiplicado pelos outros fatores que determinam o peso de cada medida de distância.

Na linha 23, o algoritmo 6.5 efetua a integração propriamente dita, ponderando as medidas $d$ de cada vista pelo seu peso modificado $w$. A fórmula de integração é a mesma de Curless, a saber:

$$D(v) = \frac{W(v)D(v) + w_i(v)d_i(v)}{W(v) + w_i(v)} \tag{6.2}$$

$$W(v) = W(v) + w_i(v) \tag{6.3}$$

onde $D(v)$ é a distância acumulada para o *voxel* $v$, $W(v)$ é o peso acumulado para o *voxel* $v$, $d_i(v)$ e $w_i(v)$ são as medidas de distância e peso do *voxel* $v$ na vista $i$, calculados pelo algoritmo 6.5.

Na linha 27, após a integração ser concluída, ainda temos um último tratamento para eliminação de erros. Os *voxels* vizinhos devem possuir medidas de distância similares, uma vez que utilizamos distâncias euclidianas no algoritmo. Assim sendo, dois *voxels* adjacentes (usando vizinhança-6 em 3D) deveriam possuir uma variação de distância de no máximo $voxelSize$. Na prática, não podemos ser tão restritivos, pois a fusão de vistas costuma violar ligeiramente esta condição. No entanto, *voxels* vizinhos com valores muito diferentes devem ser descartados, para evitar a geração de "sujeiras" no modelo reconstruído. Desta forma, utilizamos um limiar $volCompatibleFactor$, que multiplicado pelo $voxelSize$ nos dá a diferença máxima aceitável da distância entre *voxels* vizinhos. *Voxels* que são vizinhos segundo diagonais possuem este limiar aumentado, conforme a geometria dos *voxels* em questão.

Outro detalhe importante, é que esta eliminação não pode ser feita em uma única passada, pois um *voxel* "errado" acabaria "estragando" os seus vizinhos, pois a diferença entre eles seria grande. Assim sendo, primeiro descobrimos todos os *voxels* "suspeitos", e os ordenamos (usando *bucketsort*) pelo número de vizinhanças erradas (que pode ser no máximo 26). Em seguida, eliminamos os mais "errados" um a um, atualizando a lista de suspeitos para desconsiderar vizinhanças com *voxels* já eliminados. Desta forma, a eliminação é mais seletiva, descobrindo corretamente apenas os *voxels* "infratores". Estes *voxels* são eliminados marcando-os como desconhecidos (valor de distância $+D_{max}$).

O algoritmo Híbrido trata de ruídos e *outliers* de várias formas. Com o *space-carving*, representado pelo volume binário *empty*, *outliers* afastados da superfície são eliminados. Com a eliminação da linha 16 do algoritmo 6.5, *outliers* próximos da superfície e medidas fora de consenso com o resultado da primeira fase são eliminados também. Na linha 19, medidas têm sua confiabilidade calculada levando-se em conta vários parâmetros, o que serve para reduzir a influência de quaisquer *outliers* que ainda não tenham sido eliminados. E finalmente o procedimento da linha 27 elimina dados do volume que não sejam coerentes, oriundos da integração de medidas muito diferentes entre si.

Podemos analisar a qualidade do algoritmo híbrido através das figuras 6.20 e 6.21. Na figura 6.20, observamos o mesmo modelo reconstruído pelos 3 algoritmos. O algoritmo híbrido foi o que apresentou melhor resultado, em termos de confiabilidade do modelo retornado. Tanto o algoritmo de Curless quanto o de Wheeler retornaram *outliers* grosseiros no modelo final, mesmo efetuando um pós-processamento, que consistia em eliminar regiões desconexas. Este pós-processamento não é necessário no algoritmo híbrido, pois ele é muito mais eficaz na eliminação de *outliers*, o que pode ser observado através da figura 6.21. Nela vemos 2 detalhes do modelo, e percebemos como a qualidade das vistas individuais é baixa, confirmando o que foi apresentado na seção 3.3. Mesmo assim, o algoritmo híbrido consegue eliminar ou atenuar a maior parte destes defeitos, gerando um

modelo integrado mais confiável.

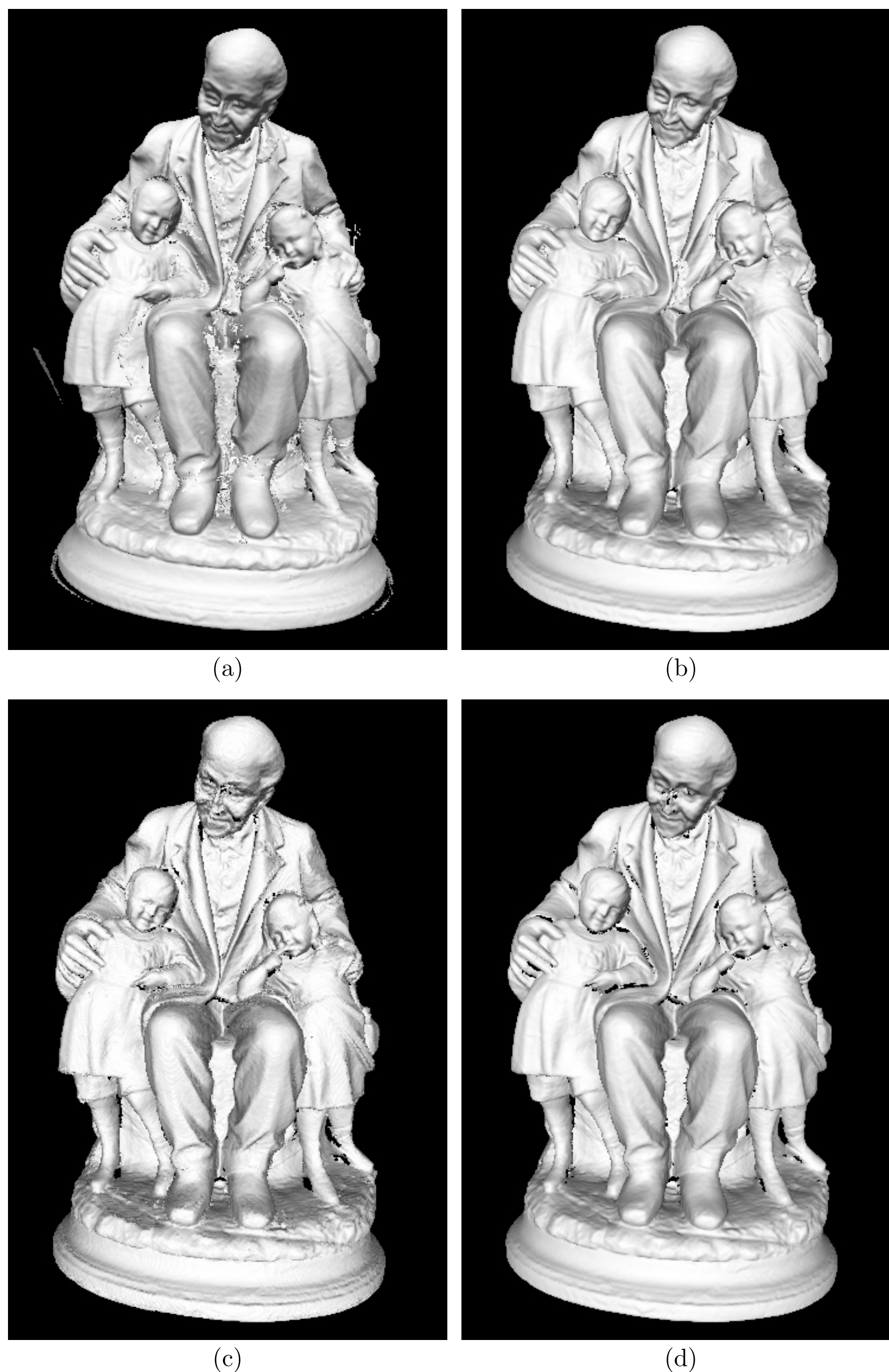


**Figura 6.20:** Objeto reconstruído pelos 3 algoritmos de integração. (a) e (b) Algoritmo de Curless; (c) Algoritmo de Wheeler com nossas melhorias; (d) Algoritmo Híbrido. Na figura (a) temos o resultado do algoritmo de Curless sem nenhum pós-processamento. Em (b) e (c) temos o mesmo objeto, só que eliminamos grupos de faces não conexas. No entanto, *outliers* conectados ao objeto principal continuam presentes. O algoritmo Híbrido (d) não sofreu nenhum pós-processamento, e conseguiu eliminar todos os *outliers* óbvios. O preenchimento de buracos foi desativado para todas as reconstruções.

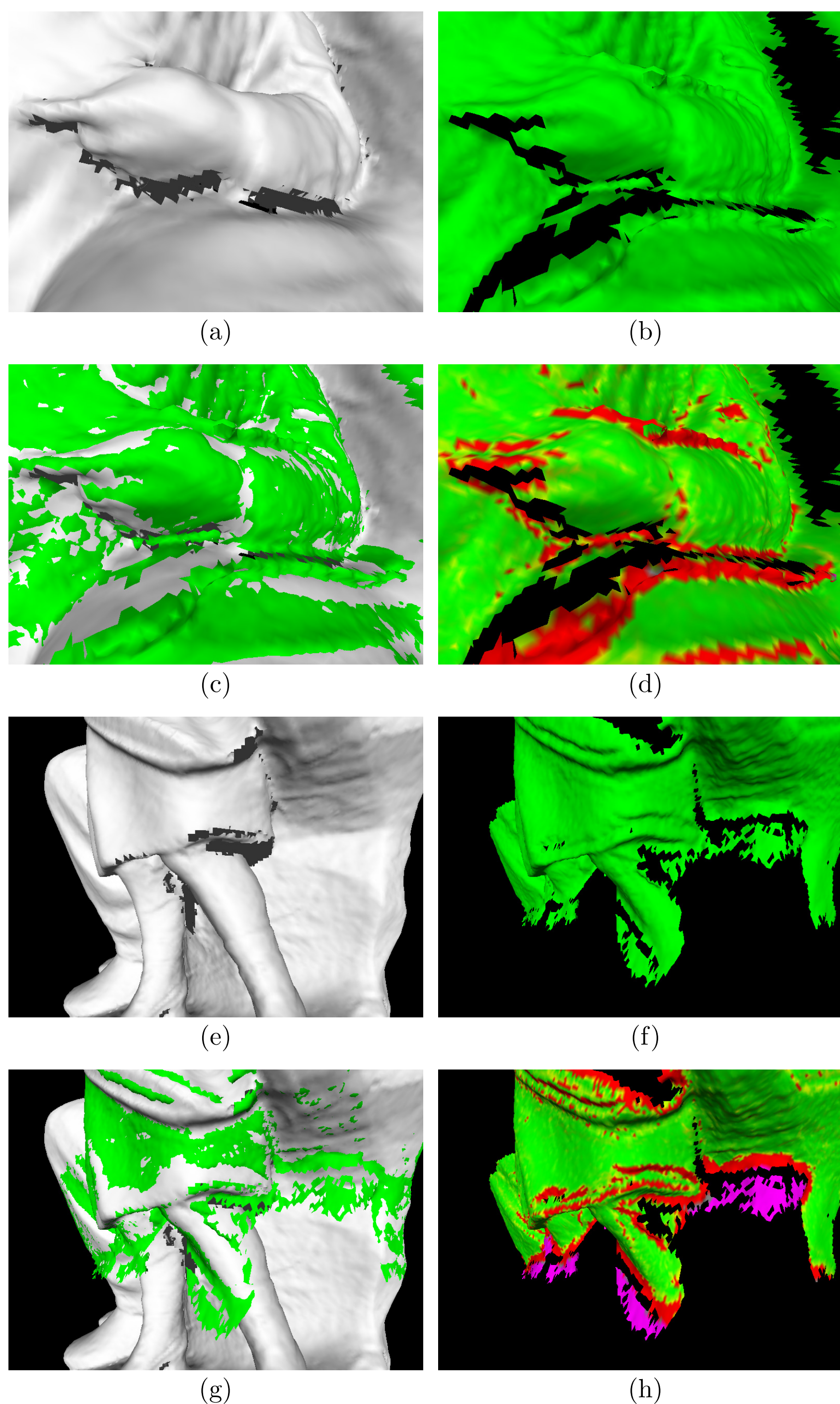


**Figura 6.21:** Detalhes do algoritmo Híbrido. (a) Detalhe do braço do menino no modelo integrado da figura 6.20; (b) Uma das vistas do braço, com superfícies deformadas; (c) Vista sobreposta ao modelo integrado, que ajuda a evidenciar as diferenças; (d) Representação gráfica dos pesos da vista. Vermelho indica superfícies descartadas, e as cores de verde a vermelho representam o peso dado, variando de 1.0 (verde) a 0.0 (vermelho). Cinza indica um *voxel* sem normal, e violeta *voxels* pertencentes ao volume binário *empty*, ambos os casos também são descartados. (e) a (h) apresentam uma outra região, agora com dados falsos, utilizando a mesma convenção. Percebe-se que o algoritmo híbrido detecta com precisão as regiões incorretas das vistas. Os pesos representados são os relativos ao ângulo entre as normais da representação volumétrica *volCopy* e as normais da vista (vide equação 6.1).

#### 6.2.3.1 Detalhes de Implementação

Nossa implementação do algoritmo Híbrido segue quase literalmente o que foi apresentado nos algoritmos 6.4 e 6.5. Muitas das considerações e estruturas de dados apresentadas nas seções 6.2.1.1 e 6.2.2.1 também são reaplicadas no algoritmo Híbrido.

O algoritmo 6.4 é basicamente o algoritmo de Curless, com a nossa curva de peso modificada e com a etapa de *space-carving*, tanto das vistas quanto dos planos de apoio (se houverem). Nele, utilizamos um volume binário *empty* para indicar se os *voxels* são externos ao objeto ou não. Isto é feito para reduzir o gasto de memória. Esta estrutura é implementada como um *array* de inteiros sem sinal, e as coordenadas $(x, y, z)$ do *voxel* são mapeadas para um bit específico. Além disso, foi implementada a operação morfológica de erosão, só que agora em 3D.

O algoritmo 6.5 também é semelhante ao algoritmo de Curless, mas as métricas utilizadas são diferentes, principalmente a distância com sinal, que é calculada como no algoritmo de Wheeler, utilizando-se *kd-trees*.

Esta não é a implementação mais eficiente do algoritmo. O processo de *space-carving*, que é a parte mais lenta, poderia ser efetuado através da rasterização 3D de tetraedros, sendo que cada tetraedro teria uma base triangular em uma face da malha 3D da vista, e o outro vértice no centro da lente do *scanner*. Desta forma, seria desnecessário o cálculo de distância pelo método de Curless para todos os *voxels*. Infelizmente não tivemos tempo de implementar esta melhoria.

Na tabela 6.3 podemos observar os parâmetros que são necessários para controlar o algoritmo Híbrido. Muitos dos parâmetros são os mesmos dos algoritmos de Curless e Wheeler, já apresentados nas tabelas 6.1 e 6.2.

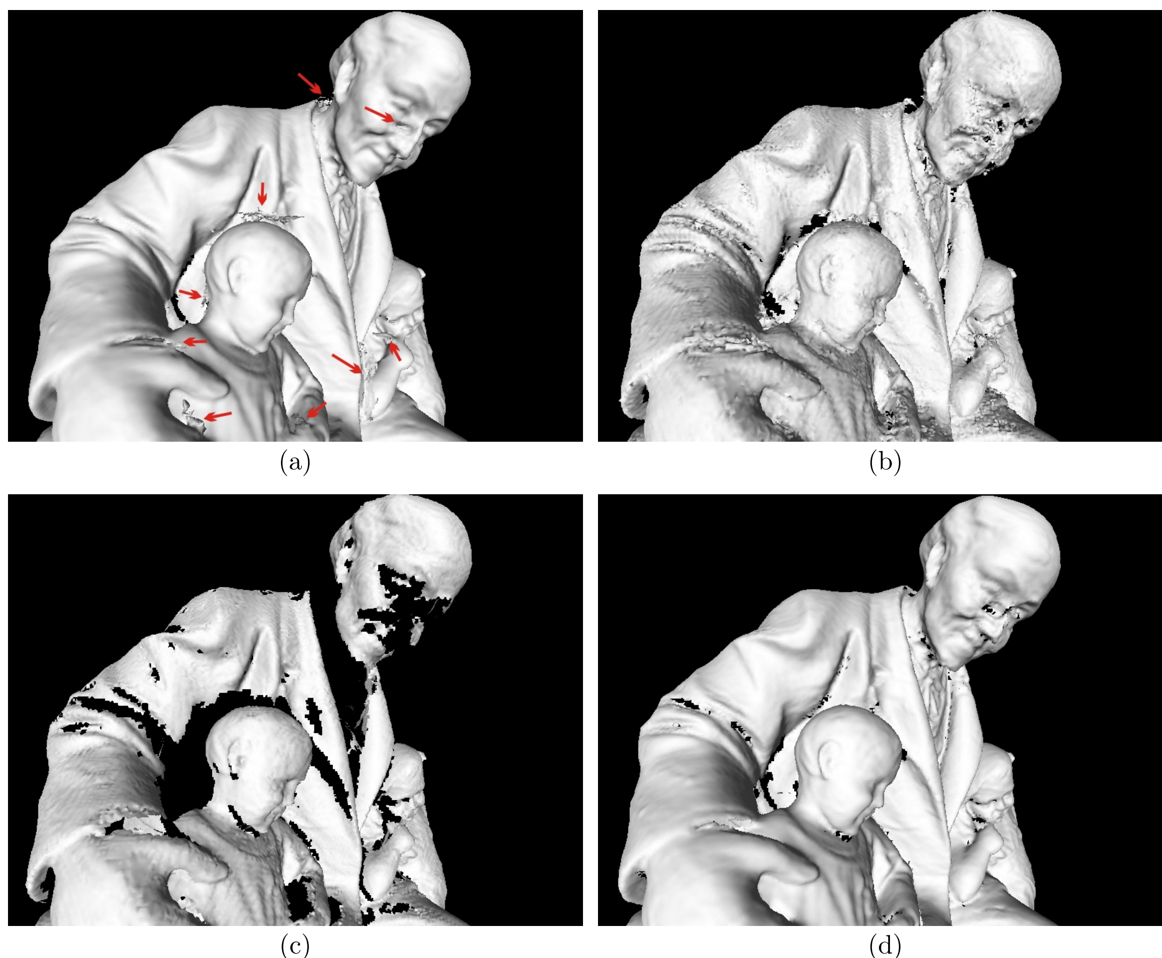
**Tabela 6.3:** Parâmetros do algoritmo Híbrido de integração de vistas, especificado no arquivo *script* de entrada do programa *ModelTool*.

| Parâmetro | Valor Usual | Descrição |
|---|---|---|
| $volVoxelSize$ | 0.5 mm | Tamanho do *voxel* da representação volumétrica[1] |
| $volMaxDist$ | 3.0 mm | Valor máximo de distância para a superfície integrada[1] |
| $volBorderWeight$ | 0.0 | Peso para medidas na borda de vistas[1] |
| $volBorderLen$ | 10 | Tamanho da borda[1] |
| $volBorderDiscard$ | 0.15 | Limiar para descarte de medidas para borda[1] |
| $volConsensusAngle$ | 30.0° | Limiar de ângulo entre normais para teste de validade[2] |
| $volCompatibleFactor$ | 1.5 | Limiar de diferença para eliminação de *voxels* incompatíveis[3] |

[1]Parâmetros idênticos ao algoritmo de Curless, vide tabela 6.1.
[2]Vide linha 16 do algoritmo 6.5.
[3]Vide linha 27 do algoritmo 6.5, e explicação na seção 6.2.3.

## 6.3 Resultados Obtidos e Trabalhos Futuros

A figura 6.20 não consegue demonstrar claramente a diferença dos 3 algoritmos de integração, pois o modelo está afastado. Os problemas se tornam óbvios quando observamos algumas regiões em detalhes, como pode ser visto nas figuras 6.22 e 6.23.



(a) (b) (c) (d)

**Figura 6.22:** Detalhes dos 3 algoritmos de integração. (a) Algoritmo de Curless; (b) Algoritmo de Wheeler, com $\theta = 2.0$ e com não-consensos; (c) Algoritmo de Wheeler, com $\theta = 4.0$ e apenas consensos; (d) Algoritmo Híbrido. Em (a), podemos observar vários defeitos (indicados por setas vermelhas), mas no geral a reconstrução de Curless é boa. A reconstrução (b) é inadmissível, com muitos *outliers* e superfície totalmente irregular. Em (c), o algoritmo de Wheeler gera uma superfície apenas ligeiramente mais suave, mas com muitos buracos, pois o limiar de consenso é mais exigente. Em (d), o algoritmo híbrido conseguiu eliminar quase todos os dados incorretos, manteve a suavidade da superfície, mas acabou gerando um pouco mais de buracos do que o algoritmo de Curless.

O algoritmo de Wheeler é claramente ineficaz para a solução do problema de integração de vistas, comparado com os 2 outros métodos. Curless apresenta bons resultados visuais, mas *outliers* ainda permanecem no modelo, e o modelo é deformado pelos *outliers* que não são removidos, como indicado pela seta vermelha do nariz na figura 6.22(a). Além disso,

os defeitos teóricos do método de Curless, apresentados na seção 6.2.1, que se referem a cantos e superfícies estreitas, também continuam presentes. O algoritmo híbrido consegue eliminar estes defeitos teóricos do método de Curless, e é mais eficaz na eliminação de *outliers*.

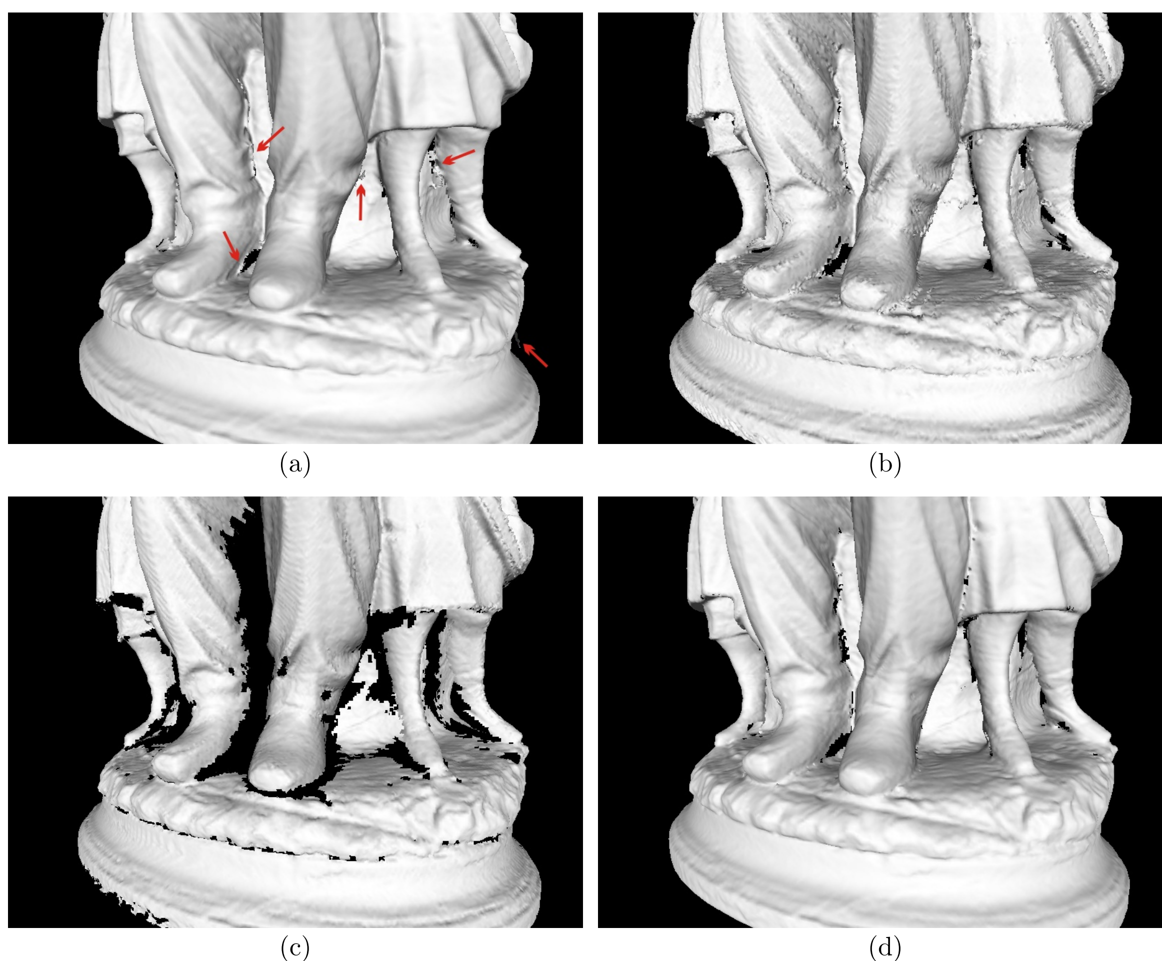


**Figura 6.23:** Outros detalhes dos 3 algoritmos de integração. (a) Algoritmo de Curless; (b) Algoritmo de Wheeler, com $\theta = 2.0$ e com não-consensos; (c) Algoritmo de Wheeler, com $\theta = 4.0$ e apenas consensos; (d) Algoritmo Híbrido. Os resultados são similares aos da figura 6.22.

O preço que se paga pela qualidade dos resultados do algoritmo híbrido é a presença de buracos mais acentuada. O que ocorre é que ao invés de deixar dados duvidosos, como Curless, o algoritmo híbrido descarta estes dados, o que origina os buracos. No capítulo 7, veremos que o algoritmo de preenchimento de buracos utilizado em nosso *pipeline* funciona muito bem, desde que não existam dados incorretos nas bordas dos buracos. Este inclusive foi um dos motivadores do desenvolvimento do algoritmo híbrido: eliminar os dados espúrios próximos a buracos, de forma a melhorar a qualidade do modelo com os buracos preenchidos. Além disso, veremos que o volume binário *empty* calculado pelo algoritmo híbrido acaba contribuindo para a melhor qualidade do processo de preenchimento de buracos.

Infelizmente, o algoritmo híbrido ainda não é perfeito. Alguns *outliers* escapam do processo de eliminação, e os buracos criados poderiam ser um pouco menores. Chegamos a experimentar variantes do algoritmo, mas nenhuma conseguiu apresentar resultados melhores. Uma variante digna de ser mencionada consistia em estimar as normais integradas diretamente a partir do volume binário *empty*. Infelizmente, a representação binária dos *voxels* não permitia uma boa precisão das normais estimadas; além disso, o volume binário *empty* também podia possuir erros, e as normais deduzidas acabavam direcionando incorretamente o processo de eliminação de *outliers*.

Sob o ponto de vista de desempenho, o algoritmo mais rápido é o de Curless. Por exemplo, a integração do modelo das figuras 6.20 a 6.23 levou 7 minutos e 38 segundos para o algoritmo de Curless, 60 minutos e 30 segundos para o algoritmo híbrido e 21 horas e 55 minutos para o algoritmo de Wheeler. Esta diferença se atribui à complexidade dos algoritmos, que é $O(V)$ para os algoritmos Curless e híbrido, e $O(V^2)$ para o algoritmo de Wheeler. Neste exemplo, o modelo foi gerado a partir de 93 vistas, possuindo cerca de 650.000 vértices e 1.300.000 faces. A máquina utilizada foi um PC Core2 Duo de 1,86GHz, com 4GB de RAM. Devemos lembrar que otimizações ainda são possíveis nos algoritmos, principalmente no algoritmo híbrido, conforme sugestões apresentadas na seção 6.2.3.1.

Apesar da grande quantidade de trabalhos recentes, mais pesquisa na área de integração de vistas ainda é necessária, para que a reconstrução automática de objetos 3D com alta fidelidade possa se tornar uma realidade. Isto pode ser confirmado através do artigo de Kazhdan *et al.* [82], onde técnicas recentes, como RBF [24], MPU [106] e Poisson [82] ainda apresentam problemas, mesmo em casos relativamente simples.

# CAPÍTULO 7

# PREENCHIMENTO DE BURACOS

Infelizmente, o processo de aquisição de dados não é completo. Oclusões e concavidades profundas nos objetos impedem a total captura de informações de profundidade, conforme apresentamos na seção 3.3.5. Desta forma, surge a necessidade de inferir as informações que estão faltando, para que um modelo fechado (*watertight*) possa ser gerado. Várias aplicações requerem modelos fechados, como a construção de réplicas e a própria visualização pelos usuários.

Este processo de inferência deve utilizar ao máximo as informações que temos sobre o modelo, de forma a gerar superfícies o mais plausíveis possível. Além disso, a geometria gerada deve ser de boa qualidade, isto é, com uma triangulação homogênea e sem faces que se interpenetrem. Muitos algoritmos não conseguem garantir esta qualidade, pois os buracos a serem preenchidos muitas vezes são topologicamente complexos, como apresentado no artigo de Davis [38] (vide figura 7.1).

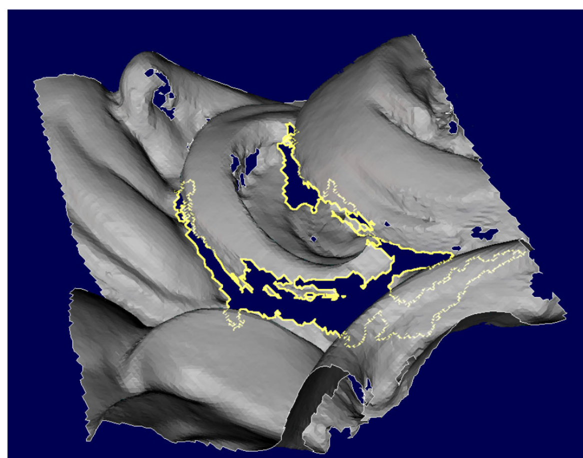


**Figura 7.1:** Exemplo de buraco do artigo de Davis [38]. Existem várias ilhas no interior do buraco, e seu perímetro é altamente irregular.

## 7.1 Revisão Bibliográfica

Um algoritmo muito simples para o preenchimento de buracos consiste em criar triângulos com os vértices do perímetro do buraco. Esta solução, no entanto, só funciona para pequenos buracos, que sejam aproximadamente planos e que tenham uma topologia de disco. Se estas condições não forem atendidas, a probabilidade de gerar triângulos que

se interpenetrem é muito alta. Pudemos constatar a ineficácia deste tipo de solução no *software* que acompanha o *scanner Vivid 910* da *Konica Minolta* [85].

Outro algoritmo consiste em ajustar uma superfície deformável com as bordas do buraco. Os algoritmos de Delingette *et al.* [39] e de Chen e Medioni [27] deformam uma superfície iterativamente de forma a ajustá-la ao modelo. O grande problema destas técnicas é que elas determinam a topologia da superfície deformável *a priori*, o que acaba não funcionando para superfícies muito complexas. Estes métodos são muito ligados aos métodos de integração por superfícies deformáveis, apresentados na seção 6.1.3.

O algoritmo de Curless [37] de integração (vide seção 6.2.1) também apresenta uma forma de preenchimento de buracos, utilizando o *space carving*. No entanto, a solução é muito básica, e superfícies pouco plausíveis são geradas. No entanto, é um dos poucos algoritmos a utilizar as informações sobre os espaços sabidamente vazios, o que será importante para o algoritmo de Davis [38], que adotamos.

Da mesma forma, o algoritmo de Wheeler [155], e a variante de Sagawa [123] também preenchem os buracos, prolongando as superfícies nas bordas dos buracos. Sagawa melhora o preenchimento de buracos através da garantia de consistência dos sinais do SDF, conforme apresentamos na seção 6.2.2. Os resultados não são muito satisfatórios, devido aos vários problemas já apresentados do algoritmo de Wheeler. O grande problema destas abordagens é não levar em conta o espaço sabidamente vazio, e apenas propagar as superfícies. Como as superfícies próximas às bordas dos buracos costumam ser muito instáveis (com muitos defeitos na aquisição de dados), o resultado final é problemático. Isto pode ser observado nas bordas do objeto reconstruído pelo algoritmo de Sagawa, apresentado na figura 6.18.

Vários outros métodos de integração também procuram preencher os buracos automaticamente, principalmente quando trabalham sobre nuvens de pontos (como os métodos de Delaunay). Estes algoritmos foram citados na seção 6.1.1, e os principais são os métodos que usam *alpha shapes* [46, 6], os métodos de *power crusts* [3, 4], *cocones* [2, 40] e *eigencrusts* [84]. Além deles, o método de *ball pivoting* [14] também trabalha diretamente sobre pontos. Como eles trabalham com nuvens de pontos, regiões sem informações simplesmente significam pontos mais afastados, e os algoritmos conseguem tratar estes casos naturalmente. O grande problema destes algoritmos é a sua sensibilidade a ruídos, e a sua dificuldade em garantir resultados topologicamente corretos.

Alguns algoritmos trabalham com representações volumétricas, como os métodos de curvas de nível (*level sets*) [156, 162]. Nestes métodos, buracos são preenchidos durante a própria integração das vistas, através do refinamento dos SDFs. Outra possibilidade são os métodos paramétricos (vide seção 6.1.3), como [24, 41, 106, 132, 82], que acabam

preenchendo buracos automaticamente pela interpolação das superfícies paramétricas.

O método de Davis [38] preenche os buracos na própria representação volumétrica, através da difusão da informação dos *voxels* conhecidos para os *voxels* do buraco. Ele será apresentado em maiores detalhes na seção 7.2.

## 7.2 Solução Adotada

Adotamos a solução de Davis [38] porque ela consegue tratar de casos topologicamente difíceis, gera um resultado garantidamente *2-manifold*, sem interpenetrações e com uma superfície suave, e porque opera sobre a representação volumétrica, que já utilizamos para a etapa de integração do nosso *pipeline*.

A idéia do algoritmo é difundir as informações dos *voxels* conhecidos para os sem informação, de uma forma semelhante a um algoritmo de suavização (*blur*), só que em 3D. Podemos visualizar este princípio na figura 7.2, retirada do artigo de Davis. Nela observamos uma fatia do volume, e a cor marrom significa dados desconhecidos. A isosuperfície corresponde à cor cinza 50%. O interessante é perceber como as superfícies são propagadas de uma forma natural.

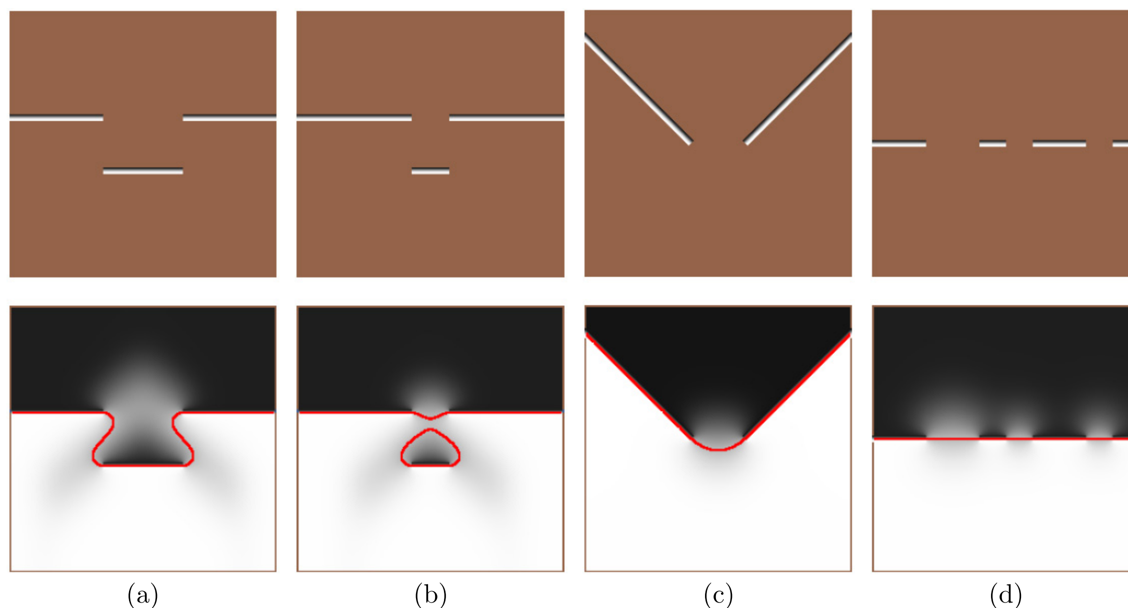
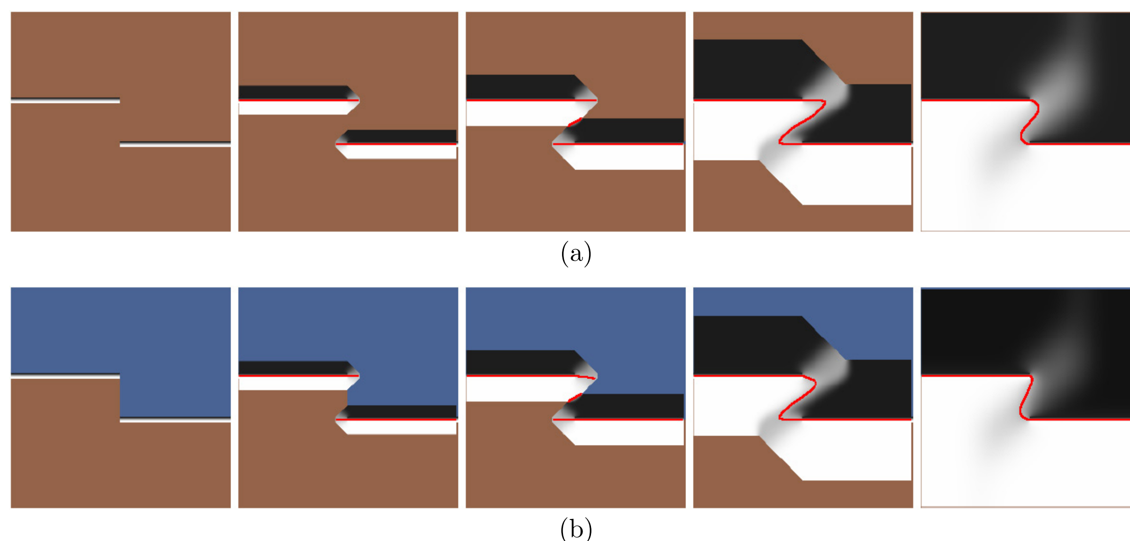



**Figura 7.2:** Exemplos de difusão em 2D para vários tipos de buraco. (a) Um recesso no objeto, que gera dois buracos; (b) Se o recesso for muito estreito, o algoritmo faz uma ponte, unindo os dois lados do recesso. Isto pode ser prevenido com a informação dos espaços garantidamente vazios; (c) Buraco em uma aresta de canto do objeto; (d) Buracos em uma superfície plana. As imagens superiores mostram o volume após a integração, mas antes da difusão volumétrica de Davis. As imagens inferiores são o resultado da difusão, com a linha vermelha marcando a isosuperfície. O algoritmo gera resultados bem razoáveis para os diversos tipos de buracos.

O algoritmo de Davis tem a vantagem de conseguir utilizar as informações de espaços vazios ao redor do objeto, obtidas através do *space carving* (vide seções 3.3.2 e 6.2.3, e a figura 3.10). Na figura 7.3 (também retirada do artigo de Davis [38]) podemos observar como o espaço vazio (representado pela cor azul) influi no processo de difusão volumétrica.



(a)

(b)

**Figura 7.3:** Exemplos da difusão em progresso. (a) Sem a utilização de informações de espaço vazio; (b) Utilizando as informações do *space carving*. Pixels em azul representam espaços vazios, e a linha em vermelho a isosuperfície de valor 0. O algoritmo evita propagar informações nas regiões vazias, mas mantém a suavidade da superfície gerada.

Para entender os detalhes do algoritmo, podemos analisar o pseudo-código 7.1.

**Algoritmo 7.1** Pseudo-código para o algoritmo de Difusão Volumétrica de Davis [38].

1: Encontre as regiões do volume vizinhas a buracos, chame-a de $regionHole$
2: Salve os dados do volume inicial na região $regionHole$ em $originalData$
3: **enquanto** a difusão não convergir **faça**
4:     //difusão
5:     **para** cada *voxel* $v$ pertencente à $regionHole$ **faça**
6:         Aplique um filtro de suavização ao *voxel* $v$
7:     **fim para**
8:     //combinação com os dados originais
9:     **para** cada *voxel* $v$ pertencente à $regionHole$ **faça**
10:         Combine o valor filtrado de $v$ com o valor original em $origData$
11:     **fim para**
12: **fim enquanto**
13: Substitua os *voxels* resultantes de $regionHole$ no volume inicial

Na linha 1 do algoritmo 7.1, encontramos as regiões do volume próximas a buracos. Isto é feito para melhorar o desempenho e reduzir o gasto de memória. Para isto, descobrimos

os *voxels* que estão localizados nas bordas de buracos, e acrescentamos a sua vizinhança à $regionHole$. Um *voxel* está na borda de um buraco quando ele é válido, possui algum *voxel* vizinho inválido e outro *voxel* vizinho válido, mas que possua o valor da distância com sinal contrário. Os dados originais do volume na região $regionHole$ precisam ser salvos para a utilização posterior pelo algoritmo.

O critério de parada do algoritmo na linha 3 pode ser implementado de diversas formas. A mais usual é pré-definir um número de iterações, calculado a partir do tamanho do máximo buraco a ser preenchido e do tamanho do *voxel*. Isto pode ser feito porque sabemos que cada iteração do algoritmo difunde 1 *voxel* para dentro das regiões vazias. Quando estipulamos o tamanho máximo de buraco a ser preenchido, conseguimos estimar o número de iterações necessárias para preenchê-lo, e mais algumas iterações adicionais para estabilizar os valores difundidos.

O corpo do algoritmo consiste em alternar etapas de difusão e de combinação com os dados originais. A difusão é feita nas linhas 4-7, e é obtida através de um filtro de suavização. Como os dados serão suavizados várias vezes, o tipo de filtro acaba não interferindo no resultado. Davis sugere um *box filter* $3 \times 3 \times 3$ ou um *plus filter*. Este *plus filter* acessa os 6 *voxels* vizinhos de $v$ nas direções dos eixos principais $(x, y, z)$. Basicamente, o que o filtro faz é tirar a média entre o *voxel* $v$ e seus vizinhos na máscara do filtro. O único detalhe é que apenas dados já definidos podem entrar nesta operação.

Nas linhas 8-11 o resultado filtrado é combinado com os dados originais. Isto é feito porque não queremos alterar ("borrar") o volume onde temos informação; o que queremos é apenas propagar esta informação para os *voxels* indefinidos. A combinação é feita utilizando-se o peso $w$ do *voxel*, onde $0 \leq w \leq 1$. Normalmente $w$ vale 1, valendo menos que isso apenas em regiões de muito baixa confiabilidade, que são comuns nas bordas de buracos. O valor de $w$ é um dos resultados da etapa de integração. Esta combinação de valores garante uma transição suave entre os dados originais e os dados gerados para tapar os buracos. A fórmula de combinação é $v_{res} = v_{filtered} * (1 - w) + v_{orig} * w$. Note-se que as próximas difusões serão feitas com os valores $v_{res}$ combinados nas etapas anteriores.

Quando se utilizam as informações de *space carving*, os *voxels* vazios possuem distância $-D_{min}$ e um peso $w$ pequeno, que são usados na etapa de combinação. Isto faz com que a superfície seja gradativamente afastada da região vazia.

Uma vez que a difusão tenha encerrado, a região $regionHole$ que foi calculada é mesclada com os dados do volume original na linha 13. Desta forma, o processo de preenchimento de buracos é incorporado naturalmente ao *pipeline* de reconstrução 3D.

## 7.3 Detalhes de Implementação

O algoritmo de Davis utiliza quase todas as informações obtidas na etapa de integração (vide capítulo 6). No entanto, para não ter que operar sobre todo o volume, apenas as regiões próximas a buracos são processadas. Para tanto, seguimos a sugestão de Davis, que consiste em dividir o volume em blocos de $8 \times 8 \times 8$ *voxels*, e apenas processar os blocos que estejam próximos às bordas de buracos. Desta forma, obtemos uma representação esparsa do volume a ser difundido.

Outro detalhe a ser considerado é que o resultado da etapa de integração pode deixar *voxels* com peso $w$ maior que 1. Devemos lembrar de limitar o valor de $w$ a 1 para o correto funcionamento do algoritmo de Davis.

Outra questão é sobre a definição de *voxels* válidos, para efetuar a filtragem. Nossos testes mostraram que vale a pena considerar os *voxels* marcados como vazios pelo *space carving* como válidos, mas com um peso pequeno. Desta forma, eles também são propagados pela operação de filtragem. O artigo de Davis não propaga os espaços vazios. Esta propagação de *voxels* vazios ajuda a gerar superfícies mais plausíveis em regiões de buracos grandes.

Em nossa implementação utilizamos o *plus filter*, descrito na seção 7.2. Fizemos esta escolha por se tratar do filtro de cálculo mais rápido possível.

Alguns parâmetros são utilizados para controlar o algoritmo de preenchimento de buracos de Davis. Estes parâmetros são especificados no arquivo .SC de *script* utilizado como entrada para a ferramenta *ModelTool*. Na tabela 7.1 podemos observar quais são estes parâmetros, uma breve descrição e os valores usuais de cada um deles.

**Tabela 7.1:** Parâmetros do algoritmo de Davis especificado no arquivo *script* de entrada do programa *ModelTool*.

| Parâmetro | Valor Usual | Descrição |
|---|---|---|
| $holeFilling$ | 1 | *Flag* usado para habilitar o preenchimento de buracos |
| $holeSaveSlices$ | 0 | *Flag* usado para gravar fatias do volume durante o algoritmo[1] |
| $holeMaxSize$ | 10.0mm | Tamanho máximo de buraco a ser preenchido[2] |
| $holeEmptyWeight$ | 0.01 | Peso utilizado para *voxels* vazios[3] |

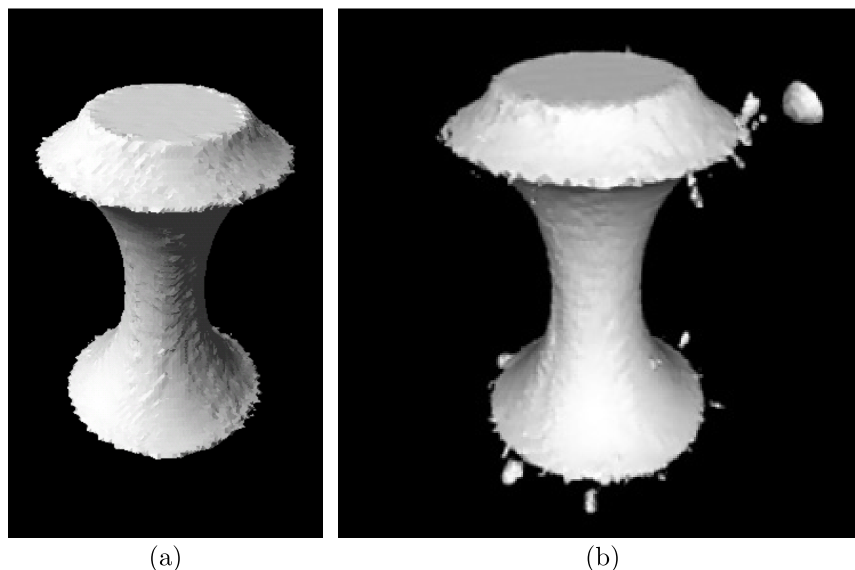
[1] Estas fatias são utilizadas primariamente para depurar o algoritmo.

[2] Quanto maior este valor, maiores os possíveis buracos a serem preenchidos, mas também maior o tempo gasto para preenchê-los.

[3] Este valor é usado como $w$ no algoritmo 7.1 para difundir e combinar os *voxels* vazios, que foram descobertos pelo *space carving*. Isto é necessário pois os *voxels* vazios resultantes do algoritmo de integração têm peso $w = 0$, e este peso precisa ser maior que zero para que os *voxels* vazios interfiram no processo de preenchimento de buracos.

## 7.4 Resultados Obtidos e Trabalhos Futuros

O resultado do algoritmo de Davis depende muito do algoritmo de integração utilizado. Sagawa [123] compara em seu artigo o algoritmo de Davis com o seu próprio, como podemos observar na figura 7.4.



(a) (b)

**Figura 7.4:** Comparação entre o algoritmo de preenchimento de buracos de Sagawa e o de Davis. (a) Resultado de Sagawa; (b) Resultado de Davis. Não consideramos esta comparação representativa do potencial do algoritmo de Davis, pois o algoritmo de integração de Sagawa gera resultados muito ruins nas bordas de buracos; e o algoritmo de Davis é particularmente sensível a *outliers* nas regiões de buracos. Além disso, o espaço vazio ao redor do objeto não foi considerado, o que apenas o algoritmo de Davis conseguiria utilizar. Finalmente, a solução de Sagawa também não é boa, conforme percebemos pela irregularidade da borda gerada.

Sagawa utilizou este exemplo para afirmar que seu método de preenchimento de buracos é superior ao de Davis. Não concordamos com esta afirmação, pois conhecemos o péssimo comportamento do algoritmo de integração de Sagawa, que é praticamente o mesmo de Wheeler, e que foi apresentado na seção 6.2.2. O que podemos confirmar é a sensibilidade do algoritmo de Davis a integrações ruins, isto é, integrações que não conseguem eliminar *outliers* (que é o caso do algoritmo de Sagawa). Como o algoritmo de Davis propaga informações, informações erradas são propagadas, o que gera os resultados implausíveis da figura 7.4(b).

Na figura 7.5, podemos observar o algoritmo de Davis executado sobre a integração de Curless, que foi apresentada na seção 6.2.1. Os resultados são muito melhores que os da figura 7.4(b), e podemos atribuir isto à melhor integração do algoritmo de Curless. Ainda assim, algumas regiões apresentam problemas, devido aos *outliers* não eliminados pelo algoritmo de integração. Estes defeitos realçados são pequenos quando comparados

com o tamanho total do modelo, mas são significativos quando observados de perto.

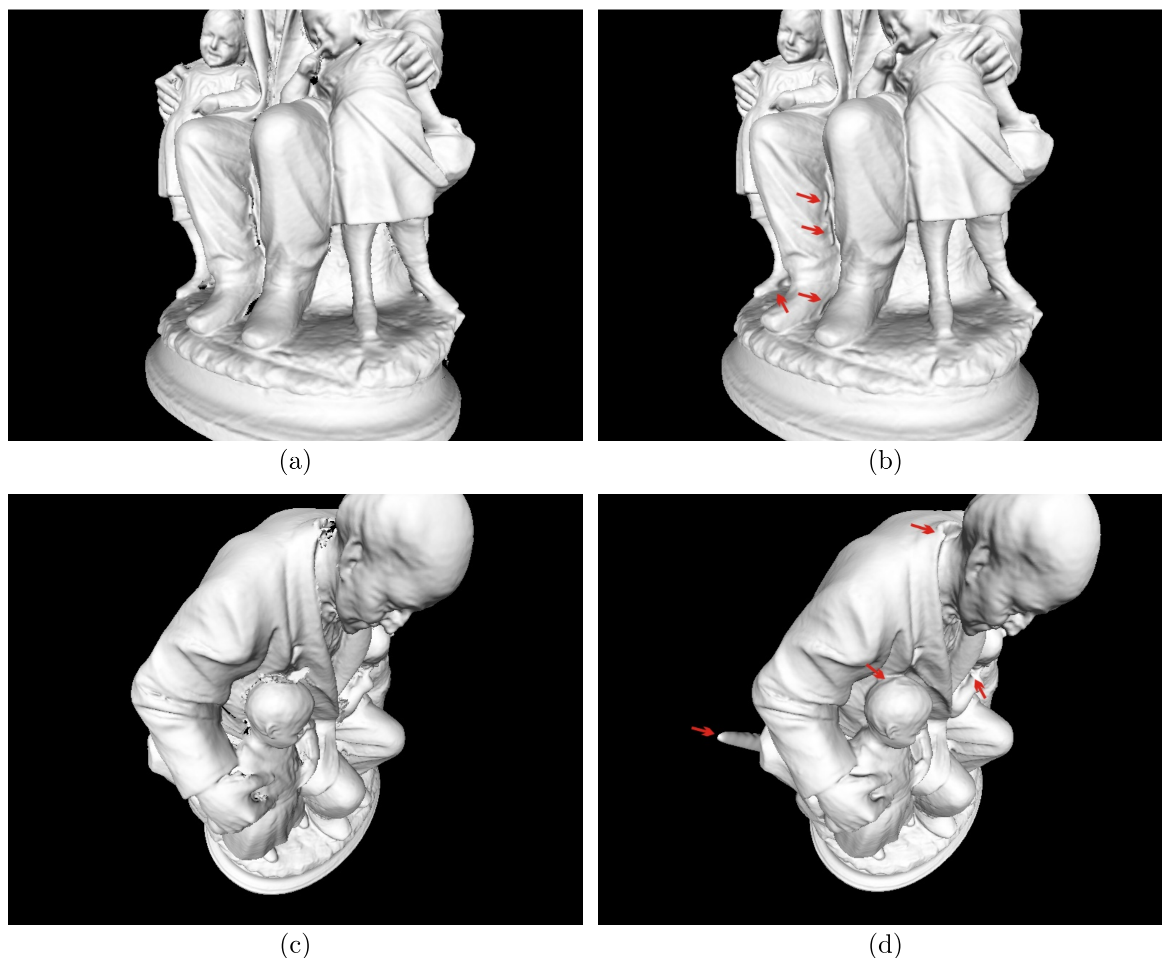
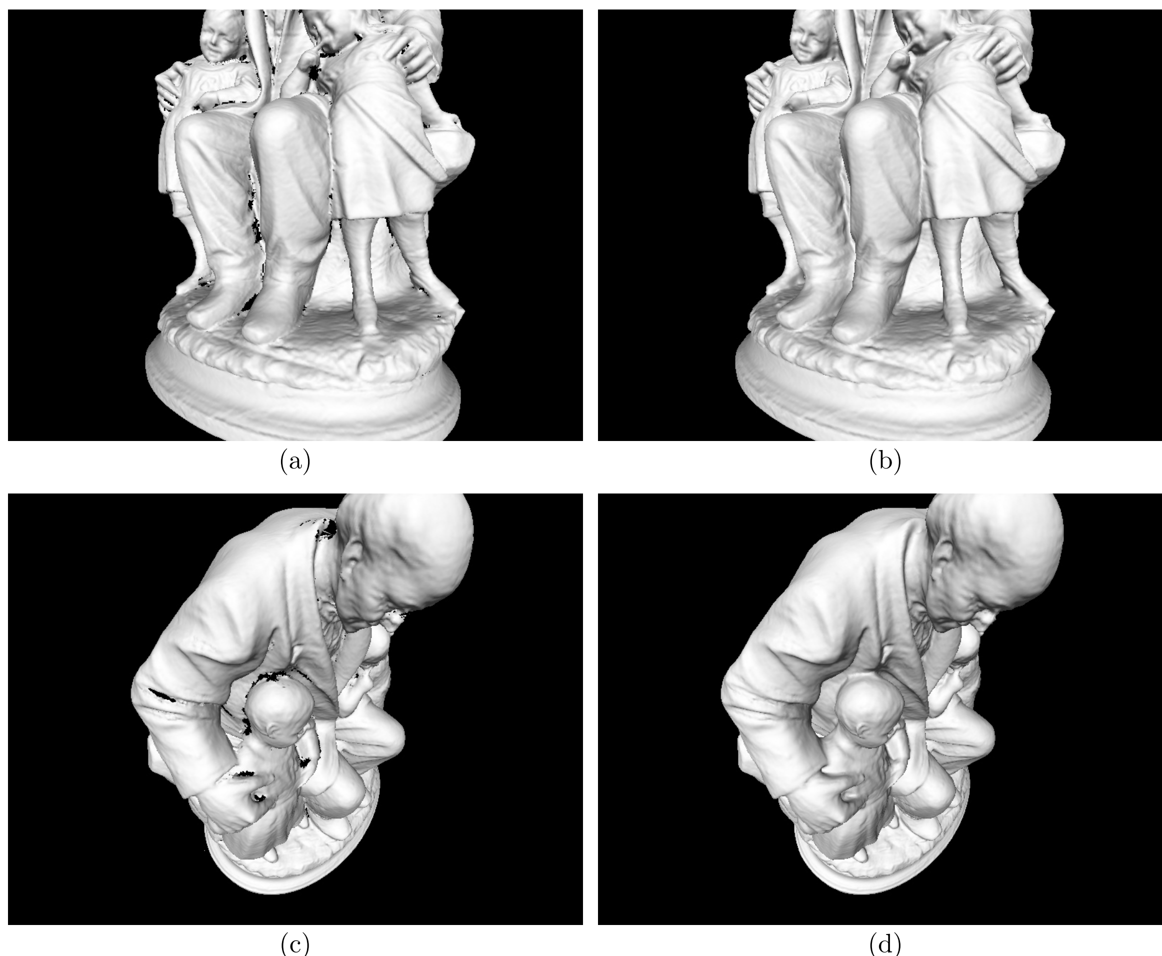



**Figura 7.5:** Algoritmo de Davis, executado sobre um modelo integrado por Curless. (a) Modelo com buracos; (b) Buracos preenchidos pelo algoritmo de Davis, correspondente à vista em (a); (c) Outra vista do modelo, com buracos; (d) Buracos preenchidos pelo algoritmo de Davis, correspondente à vista em (c). As setas vermelhas apontam para defeitos no modelo, oriundos de *outliers* não eliminados pelo algoritmo de Curless. Convém notar que o resultado do preenchimento de buracos é muito bom, pois se trata de um objeto de topologia complexa.

Na figura 7.6, podemos observar o algoritmo de Davis executado sobre a nossa integração híbrida, que foi apresentada na seção 6.2.3. Esta figura apresenta o mesmo objeto da figura 7.5, para que possamos perceber a diferença no resultado de Davis dados algoritmos de integração diferentes. Todos os defeitos marcados com setas vermelhas na figura 7.5 foram eliminados, incluindo os defeitos mais gritantes, como a gola da camisa próximo à nuca. Mesmo possuindo buracos maiores, o resultado do algoritmo híbrido é superior ao de Curless, devido à eliminação dos *outliers*, e da disponibilidade da informação dos espaços vazios, obtidos pelo *space carving*.



(a) (b)

(c) (d)

**Figura 7.6:** Algoritmo de Davis, executado sobre um modelo integrado pelo nosso algoritmo híbrido. (a) Modelo com buracos; (b) Buracos preenchidos pelo algoritmo de Davis, correspondente à vista em (a); (c) Outra vista do modelo, com buracos; (d) Buracos preenchidos pelo algoritmo de Davis, correspondente à vista em (c). Esta figura corresponde ao mesmo modelo e pontos de vista da figura 7.5. Podemos perceber que os buracos, apesar de maiores, foram corretamente preenchidos, e os defeitos indicados por setas vermelhas nas figuras 7.5(b) e 7.5(d) foram eliminados.

Outro fator importante a ser considerado é que o uso de distâncias euclidianas, usadas no algoritmo híbrido, resolvem alguns problemas listados no artigo de Davis [38], a saber, o da propagação incorreta das bordas. Davis mostrou que se os *voxels* válidos não forem perpendiculares às bordas dos buracos, a propagação das informações pode se dar em ângulo, o que gera uma superfície incorreta. Podemos observar isto na figura 7.7, retirada do artigo de Davis.

Este problema ocorre na integração de Curless, pois ele calcula a função de distância seguindo as linhas de visão do *scanner*, que não costumam ser perpendiculares às bordas dos buracos. Já em nosso algoritmo híbrido, a distância calculada acaba sendo sempre perpendicular às bordas dos buracos. Nós garantimos isto através do descarte de distâncias

que sejam calculadas para bordas de vistas. Este descarte, além de resolver os problemas discutidos na seção 6.2.2 e na figura 6.10, acaba resolvendo também este problema do algoritmo de preenchimento de buracos.

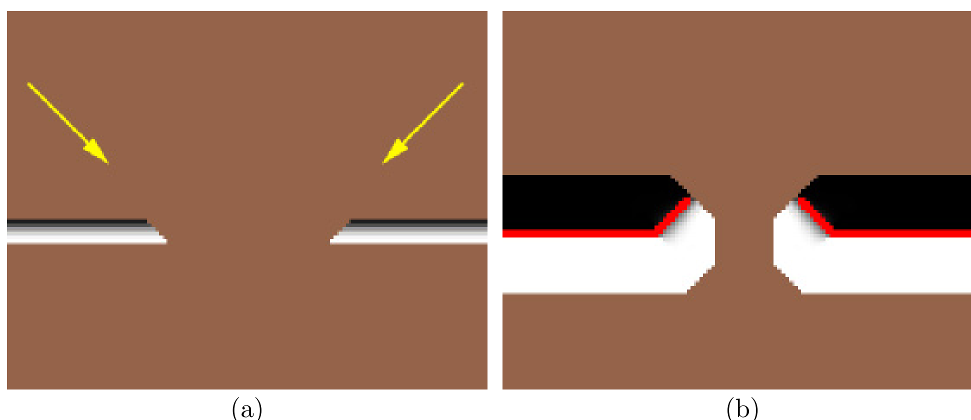



**Figura 7.7:** Problema de dados oblíquos em relação às bordas de buracos. (a) Volume inicial, onde os *voxels* foram calculados pela linha de visão do *scanner*, por exemplo, pelo algoritmo de Curless; (b) Difusão volumétrica acaba prolongando as bordas em ângulo. Isto ocorre independentemente do tipo de filtro utilizado.

Na figura 7.8, apresentamos outro objeto de topologia complexa, e como os buracos foram satisfatoriamente preenchidos. O mais impressionante é o preenchimento no interior do crânio, onde a partir das mínimas informações disponíveis, uma superfície bem plausível foi reconstruída.

Em resumo, podemos afirmar que o algoritmo de preenchimento de buracos de Davis é rápido e gera bons resultados, desde que o algoritmo de integração do *pipeline* garanta a eliminação de *outliers*. Além disso, o uso das informações de espaço vazio, obtidas através de *space carving*, também são importantes para um preenchimento de buracos de boa qualidade.

Como trabalhos futuros, podemos citar os métodos de Pauly *et al.* [108] e de Breckon [20]. Pauly *et al.* sugere completar os objetos a partir de um banco de dados de exemplos; Breckon sugere avaliar as características de percepção humana, como repetições de padrões. Ambos os métodos tentam gerar preenchimentos mais plausíveis a partir de inferências mais complexas, do que simplesmente propagar as informações da redondeza dos buracos. Incorporar estes métodos ao *pipeline* de reconstrução pode melhorar ainda mais a qualidade dos resultados obtidos.

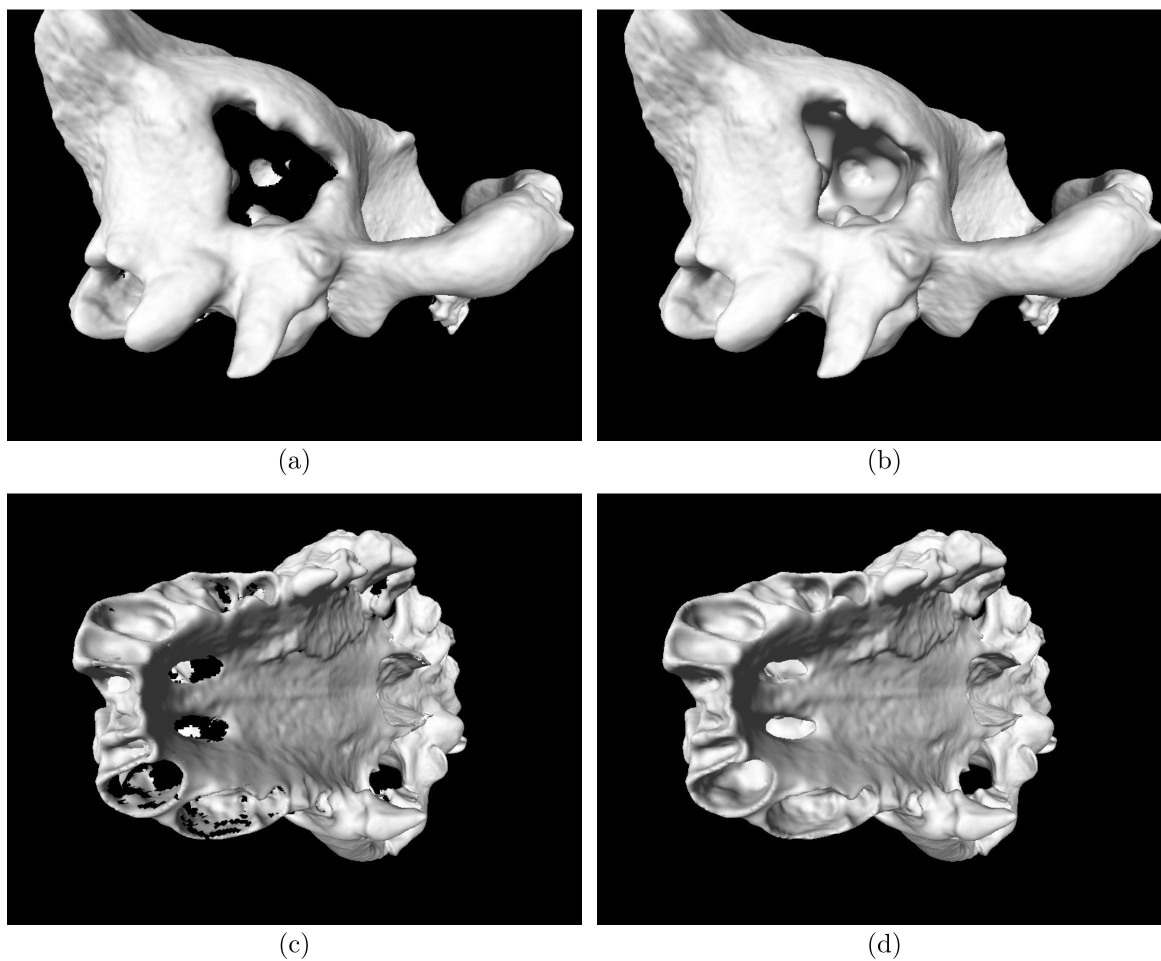



**Figura 7.8:** Buracos preenchidos de um fóssil. (a) e (c) Modelo com buracos, integrado pelo nosso algoritmo híbrido; (b) e (d) Buracos preenchidos pelo algoritmo de Davis. Apesar da topologia complexa e dos buracos muito grandes no interior do crânio, as superfícies criadas são plausíveis.

# CAPÍTULO 8

# GERAÇÃO DA MALHA 3D

De posse da representação volumétrica integrada e com os buracos preenchidos, é relativamente trivial a geração de uma malha poligonal. O método padrão, adotado por quase todos os *pipelines* de reconstrução, é o algoritmo *Marching Cubes*.

## 8.1 Revisão Bibliográfica

O algoritmo *Marching Cubes* foi proposto inicialmente por Lorensen *et al.* [93]. O algoritmo funciona percorrendo-se ordenadamente os *voxels* do volume. Neste contexto, fica mais fácil imaginar os *voxels* como pontos no espaço. Para cada cubo do volume, existem 8 valores de *voxels* nos vértices do cubo, e triângulos são gerados de forma a separar os *voxels* de valores positivos dos de valores negativos no mesmo cubo. Isto acaba extraindo a isosuperfície equivalente ao valor 0, que representa a superfície do objeto. Devido à simetria, existem 15 casos diferentes de configurações, que podem ser vistos na figura 8.1.

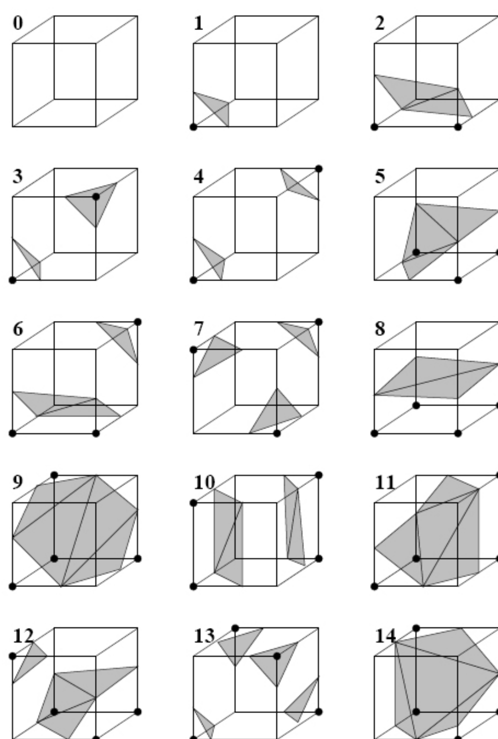



**Figura 8.1:** Os 15 casos do algoritmo *Marching Cubes*, conforme descritos por Lorensen [93].

Convém notar que o posicionamento dos triângulos depende da magnitude dos *voxels* nos vértices de cada cubo, ou seja, é feita uma interpolação linear em cada aresta para descobrir a posição dos vértices correspondentes ao valor 0 do SDF.

O algoritmo original apresentava situações ambíguas em alguns destes casos. Estas ambiguidades correspondem a diferentes topologias possíveis para o mesmo caso do *Marching Cubes*. O efeito prático, se desconsiderarmos estas ambiguidades, é a geração de uma malha com pequenas fendas, ou seja, deixamos de ter um modelo fechado (*watertight*).

Vários autores sugeriram formas de corrigir as ambiguidades. Elas foram resolvidas Nielson e Hamann [105], através da suposição de que os dados variam trilinearmente no interior de cada cubo. Esta solução levou à criação de uma tabela de casos modificada, apresentada no trabalho de Montani *et al.* [22]. Seguindo a mesma linha de raciocínio, foi possível a resolução de ambiguidades internas ao cubo, através do trabalho de Chernyaev [28]. Finalmente, Lewiner *et al.* [90] completou o trabalho de Chernyaev, apresentando uma implementação com todas as tabelas necessárias.

Na figura 8.2, retirada do artigo de Chernyaev [28], podemos examinar todos os casos necessários para a eliminação das ambiguidades.

A grande limitação do algoritmo *Marching Cubes* é que ele requer uma distribuição homogênea de *voxels*, isto é, todos os *voxels* devem ter o mesmo tamanho. Alguns algoritmos de integração, como o de Wheeler [155] e o de Sagawa [123] utilizam *octrees* para armazenar o volume. Desta forma, Sagawa sugeriu uma modificação do algoritmo *Marching Cubes* para tratar de "cubos deformados", que ocorrem na *octree*, quando a isosuperfície passa por nodos de tamanhos diferentes. Em nosso *pipeline*, não precisamos utilizar esta solução, pois não utilizamos *octrees*.

## 8.2 Detalhes de Implementação

O algoritmo *Marching Cubes* é muito rápido, pois apesar da complexidade dos diversos casos, a maioria das decisões ficam codificadas em tabelas (*look-up tables*). Nossa implementação baseou-se no trabalho de Lewiner *et al.* [90], que criou as tabelas para o método de Chernyaev.

Nós otimizamos a implementação de Lewiner *et al.*, de forma que apenas 4 fatias do volume precisassem estar simultaneamente carregadas na memória. Isto foi necessário, pois a implementação original requeria que todo o volume estivesse carregado na RAM, o que é impraticável para muitos objetos que precisamos reconstruir. Nós precisamos de 4 fatias adjacentes, para poder calcular o gradiente em cada *voxel*. Este gradiente é importante para a geração das normais de cada vértice da malha 3D gerada pelo algoritmo.

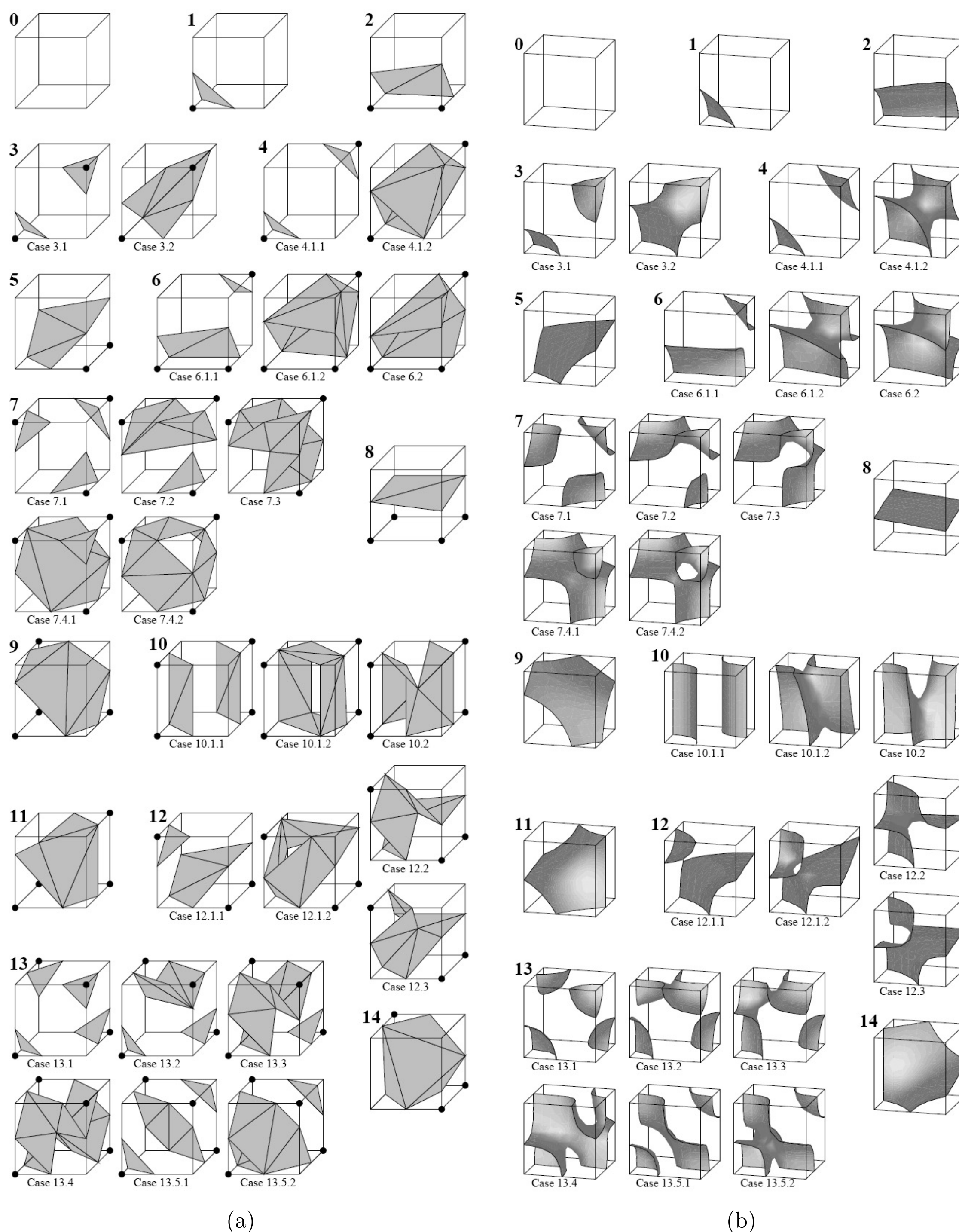



**Figura 8.2:** Os casos do algoritmo *Marching Cubes*, conforme descritos por Chernyaev [28]. (a) Triangulação dos casos; (b) Isosuperfície real dos casos.

Utilizamos um sistema de buffer circular para evitar o recálculo das fatias, quando uma nova fatia é necessária. A fatia mais antiga é sobreposta com os dados da nova fatia, que são obtidos descompactando-se a nossa representação volumétrica, que foi armazenada utilizando-se uma compressão RLE. Como a maioria dos *voxels* é vazia ou desconhecida

(distância vale $-D_{min}$ ou $+D_{max}$, respectivamente), uma grande economia de memória é obtida utilizando-se esta representação compactada. Esta idéia foi originalmente sugerida no artigo de Curless [37].

## 8.3 Resultados Obtidos e Trabalhos Futuros

Todas as figuras de modelos integrados apresentados nesta tese utilizaram o algoritmo *Marching Cubes*. Logo, ele funcionou para todos os casos testados. O único problema que encontramos foi a tendência do algoritmo de gerar triângulos muito estreitos, conhecidos como *slivers*. Na figura 8.3, podemos observar de perto uma triangulação do *Marching Cubes*, que gerou *slivers*.

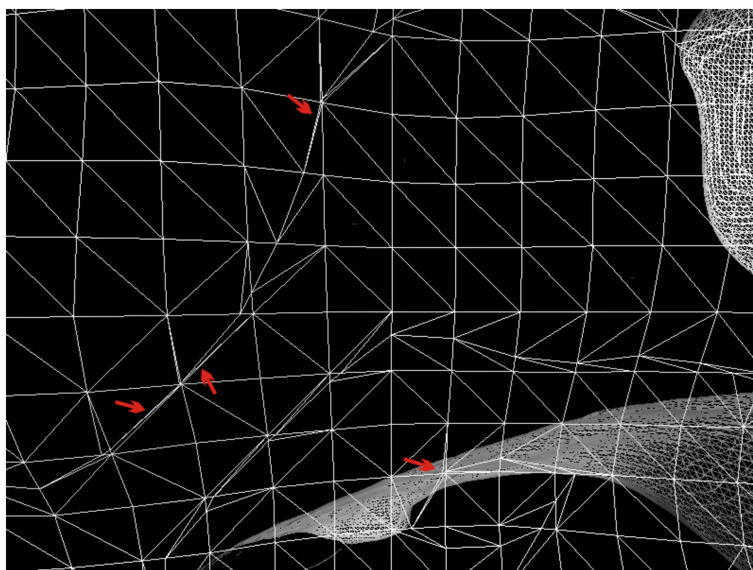


**Figura 8.3:** Triângulos muito estreitos (*slivers*) gerados pelo *Marching Cubes*. Alguns deles estão indicados com setas vermelhas. Em casos extremos, os triângulos são tão estreitos que o cálculo dos ângulos do triângulo e da normal da face se tornam instáveis, ocorrendo *quiet NaNs* durante os cálculos.

Estes *slivers*, em casos extremos, acabam gerando erros de cálculo em ponto-flutuante, que podem comprometer estágios posteriores do *pipeline*. Como uma melhoria futura, sugerimos uma etapa de pós-processamento sobre a malha 3D resultante do *Marching Cubes*, de forma a eliminar os *slivers*. Algoritmos de geração de níveis de detalhes, como os que serão apresentados no capítulo 12, resolvem perfeitamente este problema.

Fora a questão dos *slivers*, não existe muito mais o que ser melhorado nesta etapa, pois trata-se de um algoritmo já bem consolidado e usado amplamente na prática.

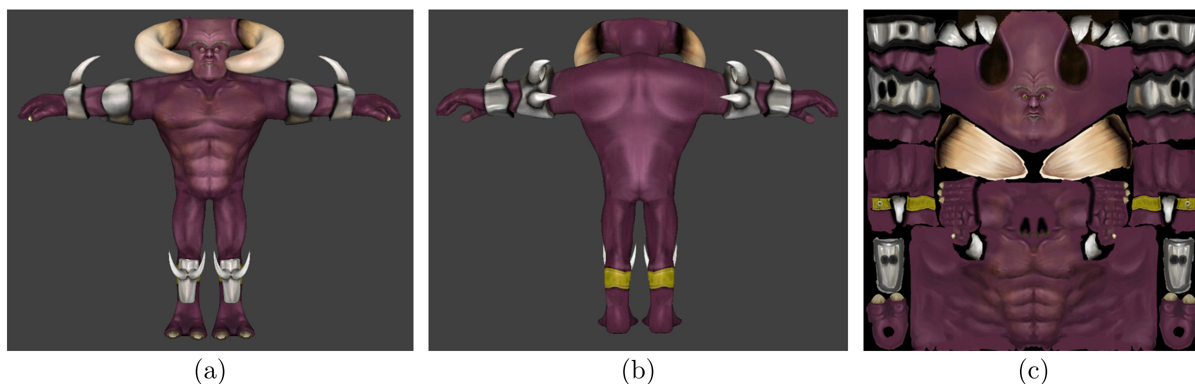
# CAPÍTULO 9

# PARAMETRIZAÇÃO DE COORDENADAS DE TEXTURAS

Após a geração do modelo 3D, podemos dizer que a parte geométrica do problema de reconstrução 3D já foi resolvido. No entanto, ainda é necessário reconstruir as características da superfície do objeto, como cor e especularidade. Em Computação Gráfica, estas características superficiais são representadas por texturas. Logo, antes que possamos reconstruir estas características, precisamos ser capazes de aplicar texturas ao modelo 3D que acabou de ser gerado.

Nesta etapa do *pipeline*, o objetivo é criar um mapeamento entre cada vértice do modelo, que possui coordenadas 3D $(x, y, z)$, para um par de coordenadas 2D $(u, v)$, que são as chamadas coordenadas de textura dos vértices. Estas coordenadas de textura indicam uma posição nas imagens das várias texturas necessárias. Este processo de mapeamento 3D $\rightarrow$ 2D não é trivial; e em objetos de topologia complexa, como os que trabalhamos, o problema torna-se ainda mais complexo.

Para entender melhor o processo de parametrização, podemos observar a figura 9.1.



(a) (b) (c)

**Figura 9.1:** Exemplo de modelo 3D e sua textura. (a) Vista frontal do modelo 3D de um monstro; (b) Vista traseira do modelo; (c) Textura do modelo. Tanto o modelo quanto a textura foram criados manualmente. Podemos observar que a textura consiste em vários "pedaços" do modelo, que foram texturizados separadamente, e finalmente "encaixados" em uma única imagem.

Na figura 9.1 observamos o monstro "Chifrudo", criado no 3D Studio MAX para o jogo Inferno[1], e sua textura, também criada manualmente. Podemos observar que as faces

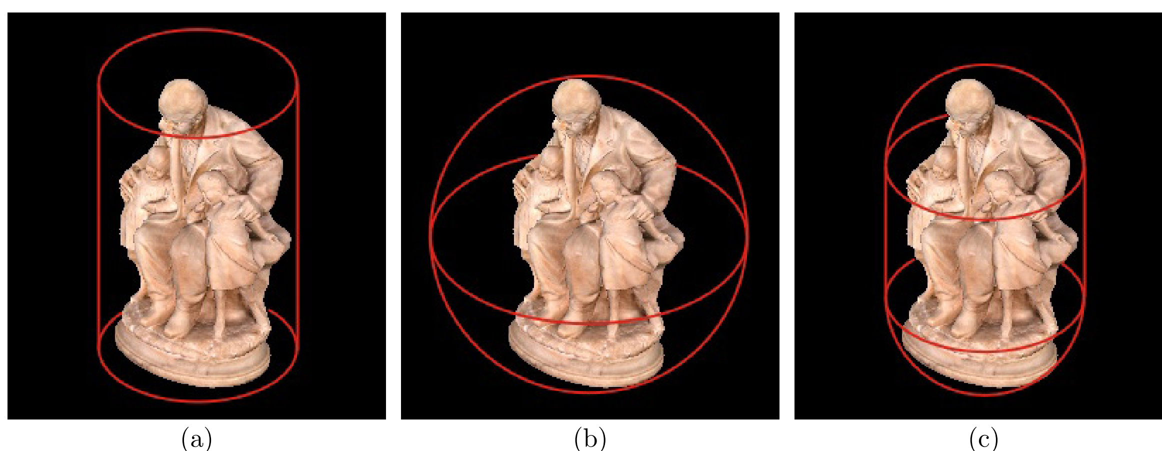
[1]Jogo desenvolvido pela empresa Continuum Entertainment, http://www.continuum.com.br

do modelo foram planificadas, gerando a parametrização da textura. Em alguns casos, foi necessário "recortar" partes do modelo, de maneira a facilitar a planificação, como o que ocorreu na cabeça, braços e pernas. O que precisamos em nosso *pipeline* é uma forma de fazer este processo de forma automatizada, e com bons resultados.

Este exemplo de planificação esbarra em um outro problema, que surge na parametrização: alguns vértices precisam ser duplicados. Isto ocorre em cada "emenda" entre pedaços de textura. Um vértice na linha de corte acaba tendo 2 ou mais pares de coordenadas de textura $(u, v)$ diferentes, um par para cada trecho de textura diferente. Desta forma, podemos perceber que o processo de parametrização acaba aumentando o número de vértices do modelo.

## 9.1 Revisão Bibliográfica

Existem várias formas de mapeamento trivial (cilíndrico, esférico ou em cápsula), normalmente encontradas em *softwares* de modelagem, e que já implementamos em nosso *pipeline*. Podemos imaginar estes métodos como uma simples projeção dos vértices do objeto para a forma geométrica usada na parametrização. Na figura 9.2, podemos observar o modelo, e imaginá-lo dentro de um cilindro, esfera ou cápsula. Em seguida, cada vértice é projetado para a superfície da forma usada para a planificação, e as coordenadas $(u, v)$ são calculadas a partir da posição projetada.



(a) (b) (c)

**Figura 9.2:** Formas triviais de parametrização de textura. (a) Mapeamento cilíndrico; (a) Mapeamento esférico; (a) Mapeamento por cápsula.

No entanto, estes mapeamentos são suficientes apenas para objetos muito simples. Objetos mais complexos exigem um processo mais sofisticado de geração das coordenadas de textura. Isto ocorre porque utilizando os mapeamentos triviais nestes objetos,

faces diferentes são projetadas sobre uma mesma região na imagem, o que não é permitido; cada face do modelo deve ter o seu próprio trecho da textura, para armazenar suas características superficiais.

O problema da parametrização de coordenadas de textura pode ser dividido em 3 partes: corte da malha, parametrização e montagem do atlas de textura. Através da figura 9.1 podemos compreender melhor estas 3 etapas. O corte da malha tem por objetivo gerar um ou mais trechos do modelo que sejam topologicamente equivalentes a um disco. Em seguida, cada um destes trechos é planificado. Finalmente, os vários "pedaços" planificados devem ser encaixados em um único espaço de parametrização, construindo o que chamamos de atlas de texturas. Note-se que foi isto que aconteceu na figura 9.1(c): O modelo foi dividido em "pedaços", cada pedaço foi planificado, e em seguida os pedaços foram encaixados em uma única textura.

A geração automática de boas parametrizações de textura é um campo ativo de pesquisa atualmente, principalmente devido ao fato que as parametrizações precisam utilizar soluções de compromisso entre características desejáveis conflitantes.

No caso da reconstrução 3D, é importante que a parametrização seja a mais uniforme possível, ou seja, apresente poucas distorções. Além disso, deve-se minimizar o número de "cortes" na parametrização, isto é, de parametrizações duplicadas para os mesmos vértices, uma vez que isso prejudica o desempenho da visualização em tempo real e o processo posterior de geração de níveis de detalhe. Além disso, de preferência deve-se utilizar totalmente toda a área retangular da textura, para que a resolução efetiva da textura seja a maior possível.

Bernardini [15], em seu *survey*, discute várias técnicas de texturização adotadas por pesquisadores em reconstrução 3D. No entanto, fica claro que a maioria das soluções acaba sendo *ad hoc*; percebe-se isto porque o problema da parametrização da textura não é abordado. A parametrização, nestes casos, acaba sendo definida pelas imagens coloridas originais capturadas, e as técnicas utilizadas para combiná-las. Não ocorre um tratamento específico da parametrização, o que em nossa opinião é uma falha grave. Afinal, a parametrização não serve apenas para mapear cores ao objeto, e sim para mapear as diversas propriedades da superfície, que serão discutidas no capítulo 10. Logo, a parametrização não deve depender das imagens coloridas, e sim da topologia do objeto.

Gu *et al.* [64] sugeriu um método de planificação que alterna etapas de corte e planificação da malha. Um corte inicial da malha é obtido através de um algoritmo heurístico. Com isto, obtém-se uma malha homeomórfica a um disco. Em seguida, esta malha cortada é planificada com o algoritmo sugerido por Floater [51]. Depois disso, identifica-se o triângulo de maior deformação, e o corte da malha é aumentado, ligando-se um dos

vértices da face mais deformada até a borda da malha. Este processo é repetido até que os novos cortes aumentem a deformação, ao invés de diminuí-la. Um atlas não precisa ser gerado, pois toda a malha foi tratada como um único disco.

Levy *et al.* [95] seleciona arestas em regiões de grande curvatura, para definir linhas de características, que acabam determinando regiões da malha. Nestas regiões, Levy seleciona sementes, que através de um processo de crescimento de regiões acabam dividindo a malha em pedaços homeomórficos a discos. Estas regiões são planificadas utilizando-se um método de conformação por mínimos quadrados, e finalmente um atlas de textura é construído, encaixando-se as regiões planificadas por ordem decrescente de tamanho.

Sheffer [136] sugere uma forma para fazer os cortes em regiões de alta curvatura e baixa visibilidade. A visibilidade dos vértices é calculada por *raytracing*, a partir de vários pontos de vista diferentes. Em seguida, vértices com alta curvatura e pertencentes a regiões de baixa visibilidade são selecionados, funcionando como bases para linhas de corte. A parametrização para as regiões obtidas é baseada na minimização da deformação angular entre as arestas da malha [137].

Wang *et al.* [154] introduziu um método bem interessante para encontrar caminhos mínimos de corte, através de um mapa geodésico de distância de contorno. Este mapa representa a distância de cada vértice até a borda da malha. Uma vez elaborado este mapa, calcula-se a curvatura gaussiana de cada vértice, e são descobertos caminhos de corte que minimizem o estiramento da malha. A planificação em si é obtida através da minimização de energia de um sistema de massa-mola. Isto distribui as deformações mais suavemente por toda a malha.

Liyan *et al.* [92] segmenta uma versão simplificada da malha, que em seguida é projetada sobre a malha original, para obter e refinar as linhas de corte. Sua parametrização parte de um centro, e cria linhas de mesma distância geodésica para o centro, que lembram curvas de nível. Com base nestas distâncias é que as faces são transportadas para o plano.

Sander *et al.* [128] apresentou um método de parametrização que leva em conta a variação da textura. Regiões de alta frequência acabam ganhando maior área no espaço de parametrização, o que acaba reduzindo o erro geral da texturização. Este processo é repetido iterativamente até o resultado ser considerado aceitável. A parametrização inicial é obtida através de uma versão simplificada da malha [129]. Para construir o atlas de texturas, Sander utiliza uma heurística onde cada pedaço da textura é envolvido por um retângulo, e estes retângulos são posteriormente encaixados, formando o atlas. Tewari *et al.* [146] conseguiu aperfeiçoar o trabalho de Sander, levando em conta algumas características do hardware gráfico.

Mullen *et al.* [102] apresentam a *Spectral Conformal Parametrization*, que gera parametrizações de alta qualidade através de métodos de álgebra linear. Gera resultados rapidamente, mas não garante a geração de parametrizações "sem dobras". No entanto, o resultado pode servir de estimativa inicial para outros métodos.

Devemos lembrar aqui uma solução proposta por alguns autores [140, 79], que parece trivial, mas não funciona: mapear cada triângulo isoladamente no espaço de textura. Apesar de simples e de garantir resultados sem sobreposição de faces, um mapeamento feito desta forma gera efetivamente cortes em todas as arestas. Isto significa um aumento na média de 6 vezes no número de vértices, devido às duplicações necessárias. Além disso, um mapa gerado desta forma ocasiona um péssimo desempenho de renderização (devido a *cache misses* dentro da placa gráfica), e torna impraticável o uso de *mip-mapping*, que é fundamental para obter resultados rápidos e de boa qualidade. Assim sendo, esta técnica é completamente inviável para uso prático.

Finalmente, Floater e Hormann [52] publicaram um *survey* sobre os métodos de parametrização, que explica os principais conceitos e algoritmos da área. No SIGGRAPH 2007 foi apresentado um curso sobre parametrização de modelos [71], que apresenta e faz um comparativo das principais técnicas de parametrização existentes.

## 9.2 Solução Adotada

Devido à complexidade dos diversos algoritmos de parametrização, em nosso *pipeline* acabamos implementando um método próprio, mais simples, mas que segue as idéias básicas apresentadas na seção 9.1. Além deste método (baseado em atlas de textura), também implementamos as 3 formas de mapeamento simples, apresentadas na figura 9.2. Convém lembrar que os mapeamentos simples não necessitam de atlas de texturas, pois projetam todo o objeto de uma vez, sem segmentá-lo em regiões. Isto acaba utilizando automaticamente todo o espaço de parametrização disponível.

Nossa convenção para o cálculo das coordenadas $(u, v)$ é o seguinte: Ambas as coordenadas variam de 0.0 a 1.0, sendo a origem $(0.0, 0.0)$ no canto superior esquerdo da imagem, com o eixo U crescendo para a direita, e o eixo V crescendo para baixo. Isto pode ser melhor observado na figura 9.3.

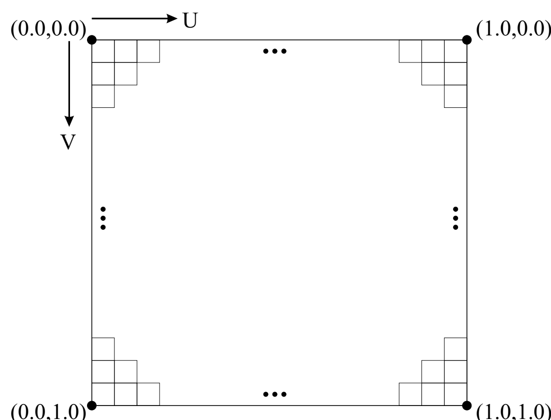



**Figura 9.3:** Convenção do sistema de coordenadas de parametrização UV. Note-se que este sistema independe do tamanho da imagem da textura, pois as coordenadas variam sempre de 0.0 a 1.0.

### 9.2.1 Mapeamento Cilíndrico

Para efetuar o mapeamento cilíndrico, apresentado na figura 9.2(a), primeiro devemos encontrar a AABB (*axis-aligned bounding box*) que envolve todos os vértices do modelo. Em nossa implementação, o eixo Z corresponde à altura do cilindro. Desta forma, para o cálculo da coordenada $v$ de textura de cada vértice, basta comparar a sua coordenada Z em relação aos valores $Z_{min}$ e $Z_{max}$ da *bounding box*, como apresentado na equação 9.1.

$$v = 1.0 - \frac{Z - Z_{min}}{Z_{max} - Z_{min}} \tag{9.1}$$

onde $v$ é a coordenada de textura vertical, $Z$ é a coordenada Z do vértice, e $Z_{min}$ e $Z_{max}$ os limites verticais da *bounding box*.

Para obter a coordenada $u$, precisamos do ponto médio da *bounding box* no plano XY, que possui coordenadas $center_{xy} = (\frac{X_{min}+X_{max}}{2}, \frac{Y_{min}+Y_{max}}{2})$. Em seguida, calculamos o vetor $dir$ que liga este ponto médio até as coordenadas $(x, y)$ do vértice, isto é, $dir = (x, y) - center_{xy}$. Calculamos o ângulo deste vetor em relação ao eixo X, e a coordenada $u$ é derivada a partir deste ângulo, conforme a equação 9.2.

$$u = \frac{\pi + \arctan(dir_y/dir_x)}{2\pi} \tag{9.2}$$

onde $u$ é a coordenada de textura horizontal, $dir$ é o vetor 2D que liga o centro da *bounding box* ao vértice, e arctan retorna valores entre $-\pi$ a $\pi$.

Após calculados as coordenadas $(u, v)$ de cada vértice, devemos proceder à duplicação de vértices, que ocorre na emenda entre $u = 1.0$ e $u = 0.0$. Para tanto, observamos as faces

cujos vértices possuam coordenadas $u$ muito diferentes (por exemplo, uma diferença maior que 0.5). Alguns vértices destas faces devem ser modificados, ficando com sua coordenada $u$ fora da faixa $0.0 \rightarrow 1.0$. Isto pode ser melhor entendido através da figura 9.4.

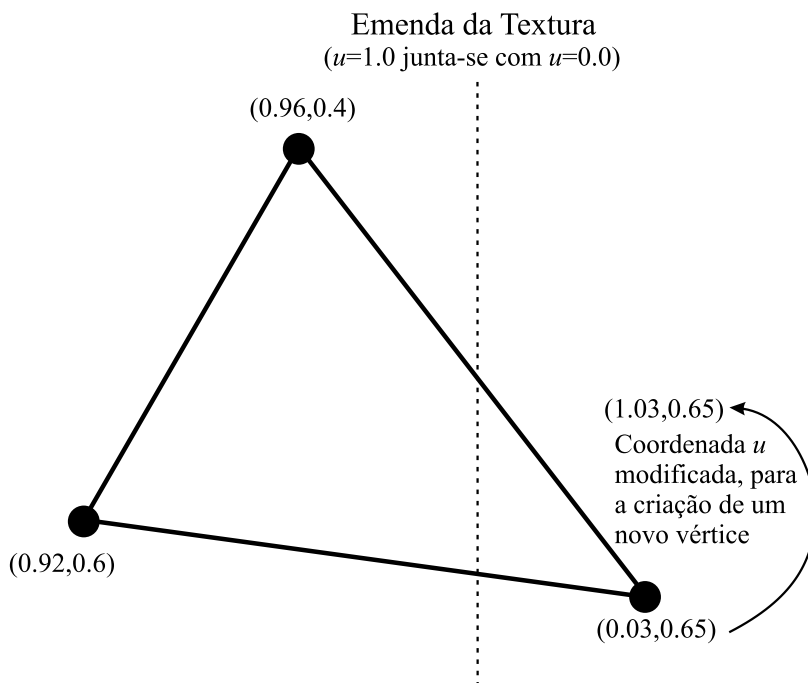



**Figura 9.4:** Duplicação de vértices na emenda do cilindro. Faces que atravessam a emenda precisam que alguns de seus vértices sejam duplicados, com a coordenada $u$ acrescentada de 1.0, senão a textura será mapeada incorretamente.

### 9.2.2 Mapeamento Esférico

Para efetuar o mapeamento esférico, apresentado na figura 9.2(b), primeiro devemos encontrar a AABB (*axis-aligned bounding box*) que envolve todos os vértices do modelo. Em seguida, descobrimos o centro geométrico da *bounding box*, que será utilizado para o centro da esfera de projeção. As coordenadas $(u, v)$ serão obtidas através da longitude e latitude de cada vértice em relação à esfera, conforme pode ser observado na figura 9.5.

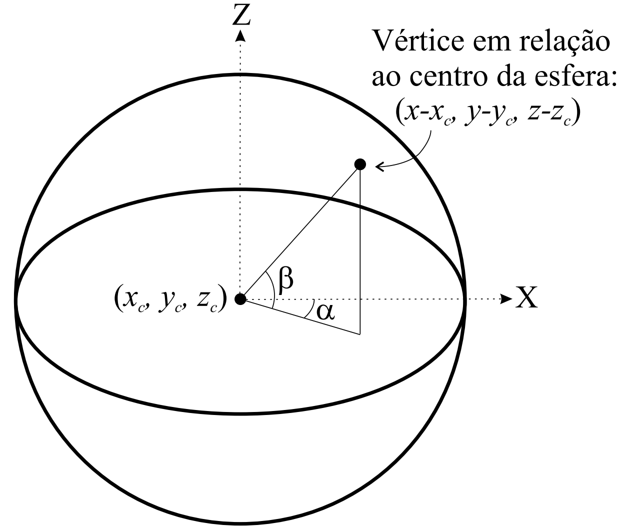



**Figura 9.5:** Projeção esférica de um vértice. O vértice de coordenadas globais $(x, y, z)$ é posicionado em relação ao centro da esfera de projeção, de coordenadas $(x_c, y_c, z_c)$. A partir daí, temos como calcular a longitude $\alpha$ em relação ao eixo X (que varia de $-\pi$ a $\pi$), e a latitude $\beta$ em relação ao plano XY (que varia de $-\frac{\pi}{2}$ a $\frac{\pi}{2}$). A coordenada $u$ será derivada da longitude $\alpha$, e a coordenada $v$ da latitude $\beta$.

As fórmulas para o cálculo das coordenadas $(u, v)$ são as seguintes:

$$\alpha = \arctan\left(\frac{y - y_c}{x - x_c}\right) \tag{9.3}$$

$$\beta = \arctan\left(\frac{z - z_c}{\sqrt{(x - x_c)^2 + (y - y_c)^2}}\right) \tag{9.4}$$

$$u = \frac{\pi + \alpha}{2\pi} \tag{9.5}$$

$$v = \frac{\frac{\pi}{2} - \beta}{\pi} \tag{9.6}$$

onde $\alpha$ e $\beta$ são a longitude e latitude, conforme a figura 9.5; $(x, y, z)$ são as coordenadas originais do vértice; $(x_c, y_c, z_c)$ são as coordenadas do centro da *bounding box*; $(u, v)$ são as coordenadas de textura calculadas, e arctan retorna valores entre $-\pi$ a $\pi$.

Da mesma forma que ocorreu no mapeamento cilíndrico (seção 9.2.1), precisamos duplicar alguns vértices, em faces que cruzem a emenda entre $u = 1.0$ e $u = 0.0$. O procedimento é o mesmo, já apresentado na figura 9.4.

### 9.2.3 Mapeamento em Forma de Cápsula

O mapeamento em forma de cápsula, apresentado na figura 9.2(c), não passa de uma combinação do mapeamento cilíndrico (seção 9.2.1) com o mapeamento esférico (seção 9.2.2). Este mapeamento resolve uma grande deficiência do mapeamento cilíndrico, que é não

considerar o topo e a base na projeção. Completando o cilindro com dois hemisférios, conseguimos que o topo e base do objeto sejam melhor projetados na textura.

Um detalhe a ser considerado é o da orientação da cápsula. Uma vez calculada a AABB do modelo, o eixo da maior dimensão da *bounding box* deve ser usado como o eixo da cápsula.

Para determinar o raio dos dois hemisférios, calculamos a maior distância 2D entre todos os vértices e o centro da cápsula, num plano perpendicular ao seu eixo. Por exemplo, se o eixo da cápsula for o eixo Z, o raio dos hemisférios será: (Equação 9.7)

$$raio = \max\left(\sqrt{(x - x_c)^2 + (y - y_c)^2}\right) \tag{9.7}$$

onde $raio$ é o raio dos hemisférios e do cilindro da cápsula; $(x, y, z)$ são as coordenadas originais do vértice, e $(x_c, y_c, z_c)$ são as coordenadas do centro da *bounding box*.

Uma vez obtido o raio, ao projetar-se cada vértice, devemos observar se o vértice está na região do hemisfério superior, na região do cilindro, ou na região do hemisfério inferior. Seguindo o nosso exemplo, onde o eixo da cápsula é o eixo Z, podemos definir (sendo $z_{min}$ e $z_{max}$ as coordenadas mínimas e máximas da *bounding box*, respectivamente):

- Hemisfério superior: se $z - z_{min} \geq z_{max} - raio$;
- Hemisfério inferior: se $z - z_{min} \leq raio$;
- Corpo do cilindro: quando $z_{max} - raio > z - z_{min} > raio$.

Uma vez descoberta a região, basta aplicar a matemática apresentada nas seções 9.2.1 e 9.2.2. O único detalhe é que as coordenadas $v$ de cada região não variarão mais de 0.0 a 1.0, e sim em faixas conforme a região, ou seja: para o hemisfério superior, a coordenada $v$ variará de 0.0 a $\frac{raio}{z_{max}-z_{min}}$; para o corpo do cilindro, de $\frac{raio}{z_{max}-z_{min}}$ a $\frac{z_{max}-raio}{z_{max}-z_{min}}$; e para o hemisfério inferior, de $\frac{z_{max}-raio}{z_{max}-z_{min}}$ a 1.0.

Em nossos testes, percebemos que "achatar" os hemisférios melhorava a texturização dos objetos. Em nossa implementação, utilizamos os hemisférios escalados em 50% da sua altura. Para conseguir isto, necessitamos apenas reajustar as faixas do mapeamento UV, e compensar este fator de 50% no cálculo da longitude $\beta$ de cada hemisfério.

Como ocorreu nas outras formas de mapeamento simples, precisamos duplicar alguns vértices, em faces que cruzem a emenda entre $u = 1.0$ e $u = 0.0$. O procedimento é o mesmo dos casos anteriores, conforme a figura 9.4.

### 9.2.4 Mapeamento por Atlas de Textura

Como já foi explicado anteriormente, os mapeamentos simples (cilíndrico, esférico e cápsula) não funcionam para objetos complexos, pois faces diferentes acabam sendo mapeadas no mesmo lugar. A única forma de resolver este problema é através de um mecanismo de parametrização que garanta esta condição.

Nós desenvolvemos um algoritmo básico, inspirado nos métodos apresentados na seção 9.1. A idéia é segmentar o modelo em regiões praticamente planares, planificar cada região e em seguida, montar um atlas de textura.

#### 9.2.4.1 Segmentação em Regiões

O algoritmo para a segmentação em regiões pode ser observado no pseudo-código 9.1. Este algoritmo básico segmenta o modelo em diversas regiões, crescendo cada região a partir de uma face inicial. O algoritmo para o crescimento de uma região pode ser visto no pseudo-código 9.2.

**Algoritmo 9.1** Pseudo-código para o algoritmo de segmentação do modelo em regiões.

1: Crie a primeira região a partir de uma face qualquer
2: **enquanto** houverem faces sem região **faça**
3:     **se** há alguma face sem região no perímetro da região anterior **então**
4:         Crie uma nova região a partir desta face do perímetro
5:     **senão**
6:         Crie uma nova região a partir de qualquer face sem região
7:     **fim se**
8: **fim enquanto**

A idéia básica do algoritmo 9.2 é ir crescendo a região, enquanto as faces vizinhas forem compatíveis com a região. O critério de compatibilidade é testado na linha 9. Mantemos uma normal média da região, obtida pela soma das normais dos vértices pertencentes à região.

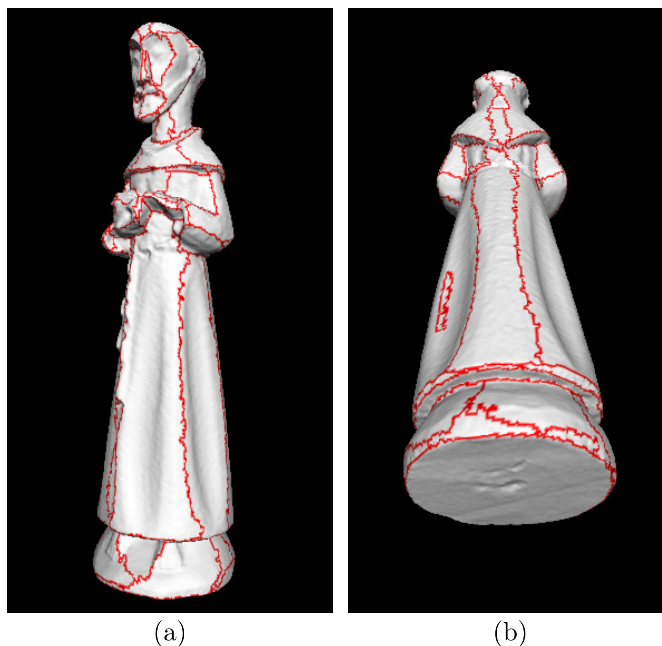
Para uma nova face ser adicionada, as normais de seus vértices devem possuir um ângulo entre elas e a normal da região inferior à uma tolerância. Em nossa implementação, utilizamos 30° como ângulo de tolerância. Evitamos utilizar a normal da face, porque devido aos *slivers* gerados pelo *Marching Cubes* (vide figura 8.3), as normais das faces podem ser instáveis. Como as normais dos vértices foram derivadas a partir do gradiente do SDF, elas não são afetadas pelos *slivers*, e por isso as utilizamos no nosso critério de teste.

**Algoritmo 9.2** Pseudo-código para o crescimento de uma região.

**Requer:** $faceInicial$ sem região
1: Marque $faceInicial$ como pertencente à região
2: $curFace \leftarrow faceInicial$
3: $perimeter \leftarrow \emptyset$
4: **repita**
5: **para** cada uma das faces $v$ vizinhas a $curFace$ **faça**
6: $perimeter \leftarrow v$, se ela ainda não pertencer a nenhuma região
7: **fim para**
8: **para** cada face $f$ pertencente a $perimeter$ **faça**
9: **se** a face $f$ for compatível com a normal média da região **então**
10: $curFace \leftarrow f$
11: Retire $f$ de $perimeter$
12: Marque $f$ como pertencente à região
13: **interrompa-para**
14: **fim se**
15: **fim para**
16: **até** não conseguir adicionar nenhuma face $f$ à região

O critério de fim de crescimento da região da linha 16 ocorre quando a região for cercada por outras regiões (perímetro vazio), ou quando ela crescer o máximo possível dentro do limiar de compatibilidade.

Na figura 9.6, podemos observar um modelo dividido em regiões por nosso algoritmo.



(a) (b)

**Figura 9.6:** Segmentação de um modelo em regiões. Em (a) e (b) vemos as várias regiões geradas, e como trechos planares do objeto acabam ficando na mesma região. Ou seja, a divisão se adapta à topologia do objeto, gerando regiões menores am áreas de grande curvatura.

Devemos lembrar que os vértices nas bordas das regiões deverão ser duplicados, pois possuirão mais de um par de coordenadas $(u, v)$.

#### 9.2.4.2 Planificação das Regiões

A técnica utilizada para planificar cada uma das regiões baseia-se no alinhamento por eixos principais de Wu [158], já utilizado na seção 5.2.2. Calcula-se o centro de massa e os 3 eixos principais, descobertos através dos autovetores da matriz de covariância. Em seguida, verificamos quais dos 3 eixos forma o menor ângulo com a normal média da região, este será o eixo de projeção. Os dois outros eixos acabam definindo o plano de projeção dos vértices, isto é, a direção U e V para a região.

O que acaba acontecendo é que os vértices são "achatados" na direção do eixo de projeção, planificando a região. As coordenadas $(u, v)$ de cada vértice são obtidas através das fórmulas:

$$u = (p - cm) \cdot uDir \tag{9.8}$$

$$v = (p - cm) \cdot vDir \tag{9.9}$$

onde $(u, v)$ correspondem às coordenadas planificadas dos vértices; $p$ é a posição 3D de cada vértice; $cm$ é o centro de massa da região; $uDir$ e $vDir$ são os dois eixos principais que definem o plano de projeção.

Devemos notar que estas coordenadas $(u, v)$ obtidas ainda não são definitivas, pois elas estão definidas com as unidades do modelo (em nosso caso, milímetros). Elas serão ajustadas posteriormente, quando da criação do atlas de textura.

#### 9.2.4.3 Montagem do Atlas de Regiões

Agora que já possuímos todas as regiões planificadas, resta apenas o problema de como encaixar todas estas regiões em uma única textura.

Um detalhe que vale a pena ser mencionado é que as placas gráficas 3D apenas apresentam bom desempenho quando as texturas possuem dimensões que sejam potências de 2, como por exemplo $1024 \times 1024$ ou $2048 \times 1024$. Desta forma, vamos gerar nosso atlas sempre com texturas quadradas, que poderão ser renderizadas com dimensões que sejam potências de 2 sem problemas. Note-se que a dimensão final da textura neste momento não nos interessa; apenas precisamos gerar coordenadas $(u, v)$ que variem de 0.0 a 1.0 (como mostrado na figura 9.3), pois elas se adaptarão automaticamente para qualquer

tamanho de imagem.

Nosso algoritmo de criação de atlas funciona da seguinte forma: cada região será tratada como um retângulo, obtido observando-se as coordenadas $(u, v)$ mínimas e máximas de cada região. Em seguida, ordenamos as regiões por ordem decrescente de área. Vamos inserindo as regiões no atlas nesta ordem, rotacionando-as 90° se isto facilitar o encaixe da região. A idéia é compactar as regiões o máximo possível, isto é, encaixar as "peças" na textura de cima para baixo, na posição mais alta que conseguir, e também levando em conta o aproveitamento de espaço, isto é, inserir a região em um "buraco" o mais justo possível.

Antes de iniciar o processo de criação do atlas, precisamos definir um "espaço" onde as regiões serão posicionadas. Para tanto, calculamos a superfície do modelo (em $mm^2$), multiplicamos for um fator de perda (devido à envolver as regiões em retângulos), e tiramos a raiz quadrada para obter a dimensão da textura, que será quadrada. Isto define o "espaço" onde as regiões tentarão ser encaixadas.

Caso o processo de encaixe não seja bem sucedido, aumentamos o tamanho da textura (em $mm$) e repetimos o processo, até que todas as regiões tenham sido encaixadas. Como resultado, saberemos a posição do canto superior esquerdo de cada região, e se ela deve ser rotacionada 90° ou não.

Um vez encerrado este processo, podemos calcular as coordenadas $(u, v)$ definitivas das regiões. Verificamos as regiões que precisem ser rotacionadas 90° e simplesmente trocamos as coordenadas $u$ com as $v$. Em seguida, ajustamos as coordenadas com as seguintes fórmulas:

$$u' = \frac{u - u_{min} + u_{region}}{texDim} \quad (9.10)$$

$$v' = \frac{v - v_{min} + v_{region}}{texDim} \quad (9.11)$$

onde $(u', v')$ correspondem às coordenadas de textura finais dos vértices; $(u, v)$ são as coordenadas temporárias em $mm$, calculadas pela planificação da região; $(u_{min}, v_{min})$ são as menores coordenadas $(u, v)$ da região, isto é, o canto superior esquerdo da mesma, em $mm$; $(u_{region}, v_{region})$ é a posição da região no atlas, em $mm$; e $texDim$ é a dimensão (altura e largura) da textura, em $mm$.

Na figura 9.7, podemos observar o atlas de texturas que foi calculado para as regiões mostradas na figura 9.6.

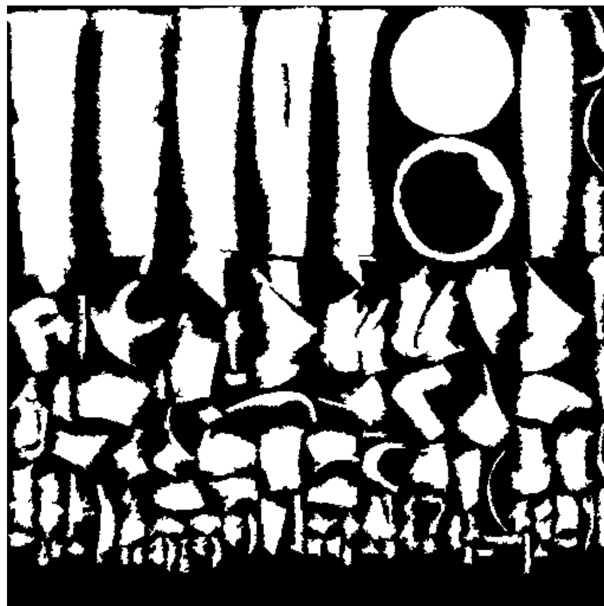


**Figura 9.7:** Exemplo de atlas de textura. Este atlas foi calculado para as regiões apresentadas na figura 9.6. As regiões são representadas em branco, e os espaços não utilizados em preto. Podemos perceber como houve relativamente um bom aproveitamento do espaço da textura.

## 9.3 Detalhes de Implementação

Os 3 métodos simples de parametrização (cilíndrico, esférico e por cápsula) possuem implementação trivial. O único detalhe é a duplicação de alguns vértices, que foi explicada através da figura 9.4. Devemos perceber que os vértices só são duplicados para faces que atravessem a emenda $u = 1.0$ para $u = 0.0$. Nas outras faces, utiliza-se o par $(u, v)$ calculado normalmente.

O mapeamento por atlas de textura, por sua vez, é bem mais complexo. É necessária uma estrutura de dados de adjacências da malha, de forma a descobrir rapidamente as 3 faces vizinhas a uma face qualquer. Isto faz-se necessário no algoritmo 9.2.

Outro detalhe é como armazenar as regiões, de forma eficiente. Como não sabemos *a priori* quantas regiões serão criadas, nem quantas faces pertencerão a cada região, uma implementação trivial utilizaria listas ligadas. Infelizmente, listas ligadas são extremamente ineficientes no *hardware* atual, pois tendem a causar muitos *cache misses*, devido ao fatos dos nodos das listas estarem espalhados pela memória. O uso de vetores tende a ser muito mais eficiente. Desta forma, em nossa implementação dos algoritmos 9.1 e 9.2, não utilizamos listas ligadas. Nós criamos um vetor de rótulos $faceLabel$, com dimensão do número de faces, e conforme as regiões são criadas, armazenamos neste vetor o número da região a que cada face pertence. Ao término do algoritmo 9.1, já sabemos o número total de regiões, e através de uma passada em $faceLabel$ temos como saber quantas faces

cada região possui. Logo, temos como armazenar as informações das regiões em vetores alocados dinamicamente.

Outra necessidade é descobrir os vértices de cada região. Para tanto, alocamos um vetor temporário de *flags*, de dimensão igual ao número de vértices total do modelo. Através de um *loop* nas faces da região, vamos marcando os *flags* dos vértices de cada face. Ao final desse *loop*, saberemos o número de vértices da região, e eles estarão marcados no vetor de *flags*. Desta forma, transferi-los para um vetor alocado dinamicamente para cada região torna-se trivial.

Quanto à montagem do atlas, a implementação da heurística de "encaixe" de cada região é feita mantendo-se uma lista de retângulos vazios na textura. Esta lista é inicializada com um único retângulo, que cobre toda a imagem. Quando uma região é inserida, ela divide o retângulo em dois sub-retângulos, que se sobrepõem (vide figura 9.8). O retângulo anterior é descartado, e os dois novos retângulos adicionados à lista. Se na hora de adicionar à lista, um retângulo estiver totalmente contido dentro de outro, o novo retângulo é descartado. Desta forma, encaixar uma região consiste em encontrar o melhor retângulo da lista, onde a região caiba (lembrando que a região pode ser rotacionada em 90°). Como nota, cada retângulo leva em conta o espaço aproveitado, e o posicionamento do retângulo em relação à imagem; e o retângulo de melhor nota é o escolhido para a inserção da região.

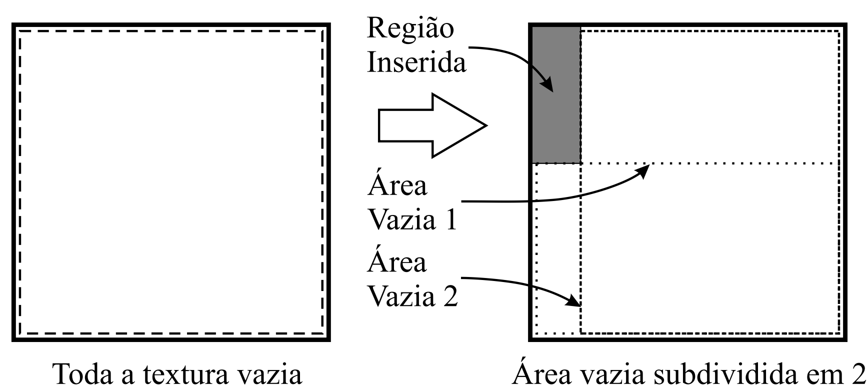



**Figura 9.8:** Implementação da heurística de montagem do atlas de textura. Uma área vazia é subdividida quando uma região é inserida dentro dela. As duas sub-áreas vazias resultantes se sobrepõem, e acabam representando a maior região que pode ser inserida dentro de cada uma delas. Conforme o processo vai se repetindo, vamos tendo uma lista de retângulos vazios cada vez mais numerosa, e com retângulos menores. Um retângulo contido dentro de outro é descartado.

Um problema detectado durante o desenvolvimento do método por atlas foi a geração de regiões muito pequenas. Percebemos que isto ocorria em parte por causa dos *slivers*, que acabavam interrompendo o crescimento das regiões. Para resolver o problema, fizemos pequenas alterações nos algoritmos 9.1 e 9.2. Quando uma região criada é pequena, a

região não é aceita e suas faces são marcadas. Depois que todas as regiões foram criadas, existe um passo adicional que anexa as faces restantes que eram de regiões pequenas nas regiões vizinhas destas faces.

Existe um parâmetro que é utilizado para controlar o algoritmo de parametrização de textura, especificado no arquivo .SC de *script* utilizado como entrada para a ferramenta *ModelTool*. Este parâmetro é o *textureMapType*, que pode ser *cylindrical*, *spherical*, *capsule*, *atlas* ou *none*. Desta forma, as várias formas de parametrização podem ser escolhidas pelo usuário da ferramenta. No modo *none* a etapa de parametrização é ignorada, e não será gerada textura para o modelo.

## 9.4 Resultados Obtidos e Trabalhos Futuros

Nas figuras 9.9 e 9.10, podemos observar os diversos tipos de mapeamentos, aplicados ao mesmo objeto. Além disso, podemos observar os defeitos dos mapeamentos simples, causados pela sobreposição de faces nas mesmas coordenadas. Apenas a parametrização por atlas não sofre destes problemas.

Nossa solução por atlas já torna possível a texturização do objeto, fornecendo uma solução inicial que funciona. Isto já é um avanço, pois o *software* que acompanha o *scanner* Vivid 910 só é capaz dos mapeamentos cilíndrico e esférico, que claramente não resolvem o problema.

Entretanto, a parametrização por atlas ainda requer várias melhorias. Conforme pode ser observado na figura 9.7, as bordas das regiões são muito irregulares. Para a etapa posterior de redução da malha poligonal, isto causará problemas. Estas bordas de regiões deveriam ser o mais “retas” possíveis, para que a malha pudesse ser simplificada seguindo o contorno da região. Sander [129] sugere um método para refinar as bordas das regiões, tentando deixá-las mais lineares, que poderia ser aplicado em nosso *pipeline*.

Nosso método para crescimento de regiões não garante efetivamente que faces não se sobreponham quando a região é planificada. Isto deveria ser considerado durante o crescimento das regiões, no entanto ainda estamos trabalhando em uma maneira eficiente para fazer este teste.

Outra otimização possível seria considerar a geometria real da região planificada quando da montagem do atlas, ao invés de envolver cada região com um retângulo. Isto acabaria melhorando o aproveitamento do espaço da textura.

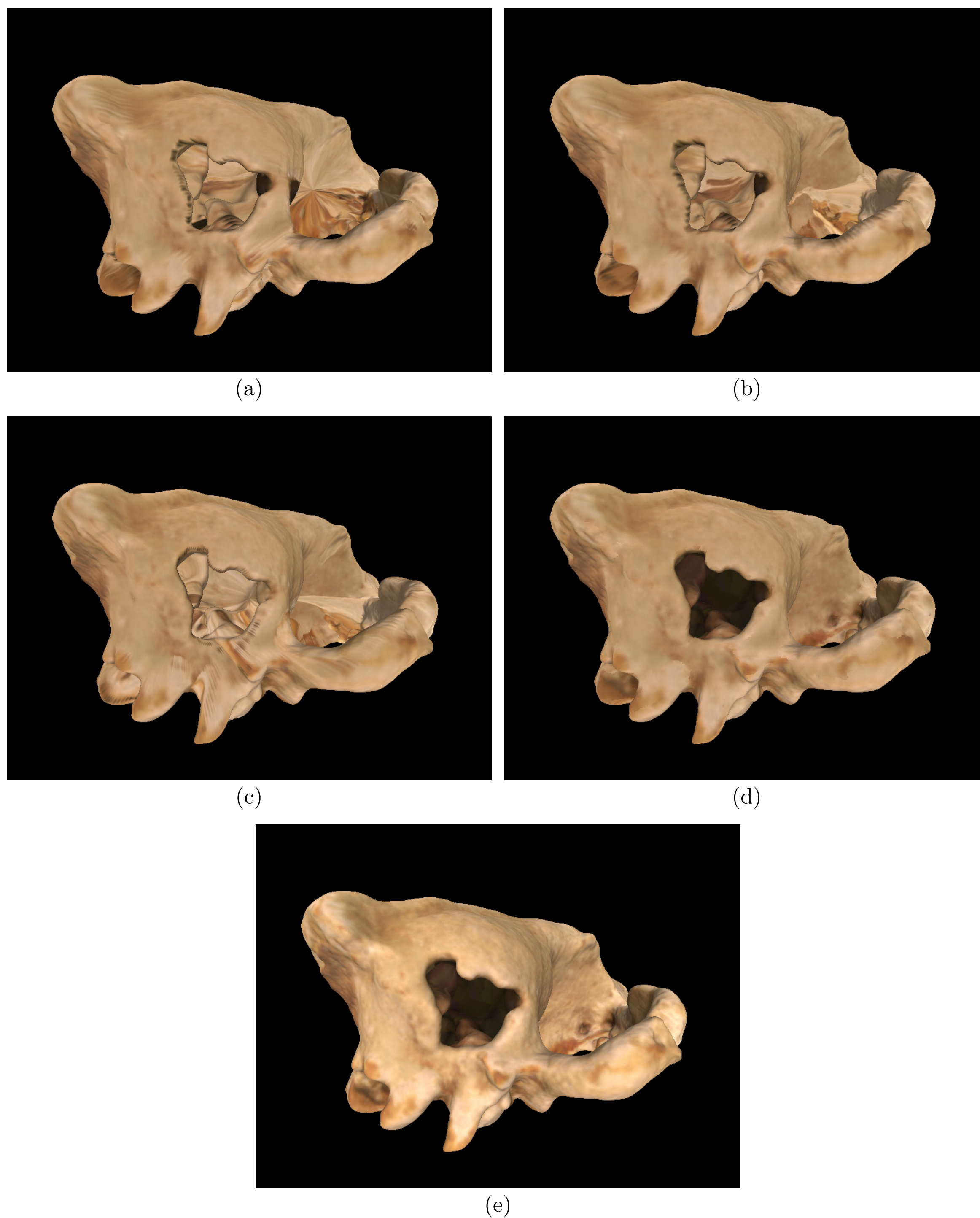


**Figura 9.9:** Modelo de um fóssil, texturizado com diversos métodos de mapeamento. (a) Mapeamento cilíndrico; (b) Mapeamento esférico; (c) Mapeamento por cápsula; (d) Mapeamento por atlas de textura; (e) Modelo original sem textura, com cores nos vértices, usado para gerar as texturas de (a) a (d). Podemos observar que apenas o mapeamento por atlas conseguiu preservar as informações originais de (e), sendo que os 3 métodos simples aplicam cores erradas em regiões internas do objeto. Isto ocorre devido à faces diferentes serem mapeadas na mesma posição da textura, ficando desta forma com a mesma cor.

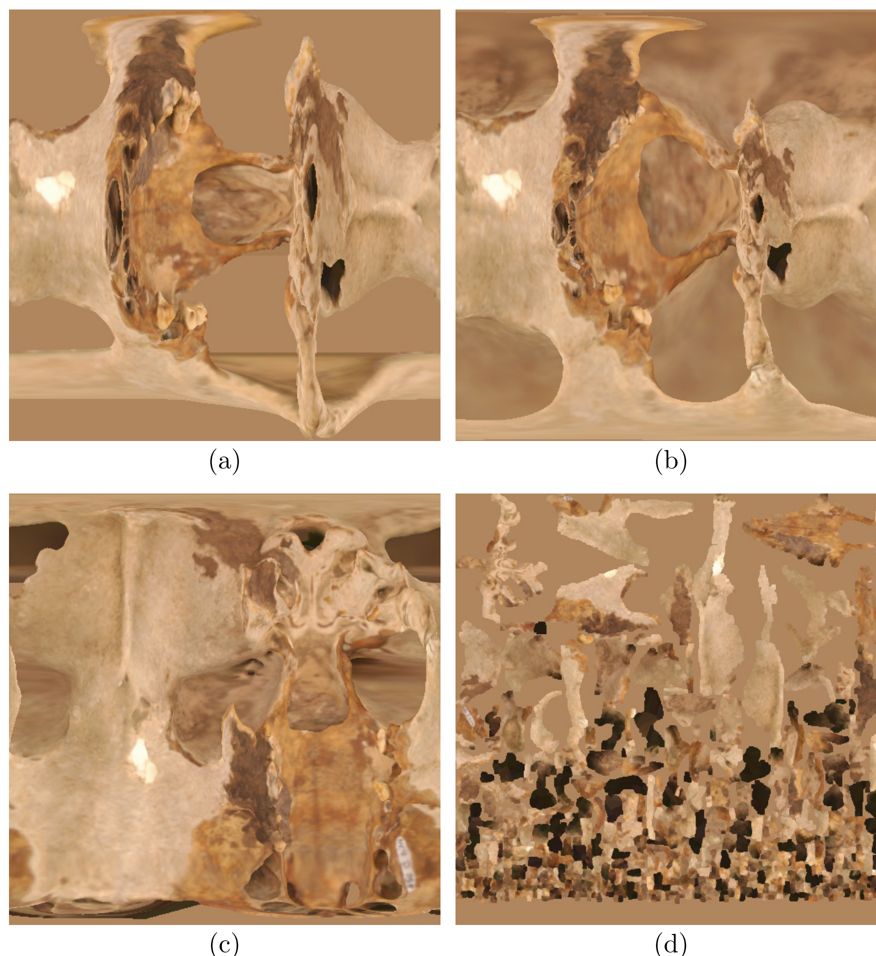


**Figura 9.10:** Texturas utilizadas na figura 9.9. (a) Mapeamento cilíndrico; (b) Mapeamento esférico; (c) Mapeamento por cápsula; (d) Mapeamento por atlas de textura. Podemos perceber como os três mapeamentos simples sofrem de uma grande distorção, bem como regiões do objeto são sobrepostas por outras faces. O mapeamento por atlas elimina estes problemas. O método utilizado para gerar estas imagens será apresentado nos capítulos 10 e 11.

Outras técnicas de parametrização podem ser experimentadas, mas somos relativamente céticos à soluções que cortam a malha, gerando uma única topologia de disco para o objeto como um todo. Isto porque a distorção acaba sendo muito elevada nas bordas da textura, conforme pode ser observado no artigo de Gu [64]. Desta forma, achamos mais promissor experimentar as técnicas que dividem o modelo em regiões e constroem um atlas de textura, similar ao nosso método, por permitir um maior controle sobre o grau de distorção máximo da parametrização de cada face. Várias técnicas de parametrização global [71], na prática, também usam esta nossa filosofia, uma vez que a malha base de parametrização acaba implicitamente dividindo o modelo em regiões, e criando uma textura final composta por um atlas das várias regiões base.

# CAPÍTULO 10

# CÁLCULO DAS PROPRIEDADES DA SUPERFÍCIE

A reflectância ou cor da superfície é a principal característica da superfície dos objetos que buscamos reconstruir. Além dela, seria importante obter outras características, como especularidade, rugosidade, transparência, entre outras. Todas estas características, uma vez conhecidas, permitem que as renderizações posteriores do modelo reconstruído sejam o mais fiéis possíveis.

A obtenção da cor da superfície é bem mais complicada do que aparenta, pois normalmente não temos muito controle sobre a iluminação que estava afetando o objeto na hora da captura das imagens de intensidade. Isto faz com que existam regiões de sombra, que acabam mudando quando o objeto é reposicionado para a obtenção de outras vistas. Além disso, se a superfície não for fosca, os *specular highlights* acabam também modificando as cores que são capturadas.

Outra dificuldade consiste no fato de que os modelos de iluminação utilizados em Computação Gráfica (Gouraud, Phong, Blinn, etc. [53]) não representam corretamente a física do processo de iluminação. Logo, sua utilização para compensar as diferenças de iluminação garantidamente resultará em resultados apenas aproximados, mesmo que as fontes de luz fossem controladas. Métodos mais sofisticados, como radiosidade, são bem mais complexos, e exigem a correta modelagem de todo o ambiente onde houve a captura [5, 53]. Todos estes fatores acabam tornando o processo de descoberta da verdadeira cor da superfície um desafio considerável.

## 10.1 Revisão Bibliográfica

Existem muitas abordagens para estimar detalhes da superfície dos objetos. Muitos artigos adotam soluções simplificadas, ou requerem *hardware* especial. No *survey* de Bernardini [15] a maioria das técnicas a seguir é abordada em maiores detalhes.

### 10.1.1 Cálculo da BRDF da Superfície

As características superficiais de um objeto são representadas através de BRDFs (*Bidirectional Reflectance Distribution Functions*). Para renderizar um objeto de forma precisa,

é necessário conhecer a BRDF para cada ponto da superfície. A BRDF $f(\lambda, x, y, \omega_i, \omega_r)$ em um ponto $(x, y)$ da superfície representa a razão entre a energia luminosa refletida na direção $\omega_r$ a partir da energia luminosa incidente na direção $\omega_i$, para o comprimento de onda $\lambda$. A BRDF pode variar consideravelmente com a posição, direções e comprimento de onda. BRDFs são explicados com mais detalhes por Foley *et al.* [53].

Estimar com precisão a BRDF da superfície de um objeto é um processo complicado. Muitas vezes, hipóteses simplificadoras costumam ser assumidas, como assumir que a superfície é Lambertiana (isto é, possui comportamento uniforme para todas as direções) [15]. Isto é claramente um problema para superfícies com especularidade.

Ikeuchi e Sato [76], e Baribeau *et al.* [8] desenvolveram métodos para estimar a BRDF, através da captura de uma superfície cuja BRDF não varie espacialmente, ou seja, uma superfície homogênea. Os dados capturados fornecem a BRDF para vários pares de $\omega_i$ e $\omega_r$. Em seguida, os métodos ajustam um modelo paramétrico para a BRDF, a partir dos dados capturados. Este modelo da BRDF foi criado por Torrance e Sparrow [149]. Logo, estes métodos funcionam para objetos que possam ser segmentados em grandes regiões com propriedades homogêneas.

Outros métodos foram desenvolvidos para estimar BRDFs que variem tanto espacialmente quanto pela direção de incidência da luz. Estes métodos se utilizam da técnica *photometric stereo*. Esta idéia, introduzida por Woodham [157], usa $N$ imagens de um mesmo objeto e ponto de vista, com $N$ condições diferentes de iluminação. Originalmente, Woodham usava sua técnica para estimar as normais da superfície. Kay e Caelli [81] combinam a técnica de *photometric stereo* com as imagens de profundidade, expandindo a idéia de Ikeuchi e Sato [76]. Eles também usam o modelo de Torrance e Sparrow. No entanto, inter-reflexões de luz no objeto não são tratadas.

Sato [130] também combinou *photometric stereo* com imagens de profundidade, para calcular a BRDF de todo um objeto. Ele separou o problema da iluminação difusa da especular, para a construção do modelo de Torrance-Sparrow. Os parâmetros especulares são particularmente difíceis de modelar, uma vez que não há como garantir a captura de reflexões especulares para todos os pontos do objeto. Desta forma, estes parâmetros acabam sendo interpolados em áreas maiores do objeto.

Lensch *et al.* [88] usa uma abordagem diferente para calcular a BRDF da superfície. Inicialmente, ele assume que toda a superfície possui a mesma BRDF, que é estimada a partir de diversas imagens em *high dynamic range* do objeto. Em seguida, divide-se os pixels em dois grupos, baseado na distância para o BRDF inicial. Este processo se repete, até que a distância das amostras para a BRDF de seu grupo esteja dentro de uma tolerância.

### 10.1.2 Captura da Reflectância

Muitas das técnicas da seção 10.1.1 dependem da aquisição de dados especializados. Como exemplo, existem técnicas que utilizam a imagem de reflectância do laser projetado para obter as imagens de profundidade [76, 8]. No entanto, estas imagens correspondem a um comprimento de onda específico $\lambda$, associado à cor do laser. Baribeau [8] inclusive precisa de *lasers* em vários comprimentos de onda para a aplicação de sua técnica. Uma vantagem do uso desta reflectância é a inexistência de "sombras", uma vez que é o *laser* que ilumina a superfície. A desvantagem é que nem todos os *scanners* retornam esta informação para o usuário, como o que ocorre com o Vivid 910 da Konica Minolta.

A maioria dos *scanners* 3D captura imagens coloridas do objeto. A grande vantagem é que estas imagens são obtidas do mesmo ponto de vista que as amostras da imagem de profundidade, ou seja, não é necessária uma calibração para utilizar estas imagens. A desvantagem é que normalmente estas imagens não possuem boa resolução e qualidade; isto foi abordado na seção 3.1 e na figura 3.5.

Outra técnica possível é a utilização de câmeras profissionais para capturar fotos do objeto. Estas câmeras podem ter uma posição fixa em relação ao *scanner* 3D, ou as capturas podem ser independentes.

Quando a posição em relação ao *scanner* 3D é fixa, utilizam-se técnicas de calibração para obter o mapeamento entre as fotos coloridas e as imagens de profundidade. Ballard [7] explica detalhadamente várias técnicas de calibração disponíveis. A idéia é obter correspondências entre a imagem colorida, e os pontos 3D da imagem de profundidade. Em seguida, métodos clássicos como o de Tsai [151] são utilizados para obter a matriz de calibração.

Quando não há uma calibração entre os dados 3D e as imagens coloridas capturadas, devemos utilizar um método de alinhamento (similar às técnicas apresentadas no capítulo 4), para descobrir a matriz de transformação entre as fotos e os dados 3D.

Neugebauer [104] descreve um método para refinar o alinhamento entre imagens e um modelo 3D. Os parâmetros de calibração vão sendo refinados, assumindo que as várias imagens foram obtidos com a mesma câmera, ou seja, a distorção da lente, foco e outros parâmetros intrínsecos são constantes.

Lensch [87] apresenta um método para descobrir a posição da câmera em relação ao objeto através da comparação entre imagens binárias. Estas imagens são obtidas através da segmentação das fotos coloridas entre objeto e fundo, e da renderização do modelo utilizando as posições candidatas da câmera. Os parâmetros da câmera são refinados reduzindo-se a diferença entre as imagens binárias.

Ikeuchi [75] também apresentou um método para alinhar imagens coloridas com modelos 3D, utilizando-se da imagem de reflectância do *laser* como um dado auxiliar. Através da detecção de correspondências nas imagens coloridas e de reflectância, o alinhamento das imagens coloridas com o modelo 3D é obtido.

### 10.1.3 Correção das Imagens Obtidas

Muitas técnicas se concentram em determinar apenas a reflectância difusa da superfície. Nestes casos, assume-se que a superfície é Lambertiana, ou seja, a energia luminosa é difundida igualmente em todas as direções. Assim sendo, o BRDF torna-se muito simples, pois não depende mais dos ângulos $\omega_i$ e $\omega_r$. Obviamente, esta aproximação funciona apenas para objetos foscos, isto é, que não possuam nenhuma especularidade.

Usando esta simplificação, o objetivo agora é corrigir as imagens coloridas capturadas (vide seção 10.1.2), de forma a eliminar a influência das fontes de luz. A idéia básica é considerar o ângulo de incidência da luz, e a normal da superfície de forma a calcular a cor real da superfície, como se ela estivesse sendo iluminada com uma fonte de luz perpendicular à cada ponto da superfície. Outro fator a ser considerado é a rampa gama de resposta das imagens coloridas, pois nem sempre os valores de intensidade nas imagens correspondem à uma escala linear.

De um ponto de vista prático, o que se procura fazer é clarear ou descartar pixels em regiões de sombra, e detectar pixels em regiões de brilho (especularidade) para descartá-los. Um exemplo da aplicação desta técnica foi apresentada por Levoy *et al.* [89].

Outro ponto problemático consiste na iluminação ambiente, que também acaba afetando a cor nas imagens. A iluminação ambiente pode ser imaginada como toda a contribuição de iluminação indireta que o objeto recebe, seja através de inter-reflexões da luz dentro do próprio objeto, ou com os objetos ao redor. Um método para tentar se contornar esta dificuldade consiste em capturar duas imagens, uma apenas com a iluminação ambiente, e outra com a iluminação ambiente e mais uma fonte de luz cuja posição é controlada. Em seguida, calcula-se a diferença das duas imagens, e esta diferença representa a iluminação apenas da fonte de luz conhecida [89, 16].

Mesmo com esta técnica, ainda existem dificuldades. A resolução de cor de imagens digitais costuma ser muito baixa (8 bits para cada canal), o que faz com que a imagem de diferença calculada possa apresentar sérios problemas de quantização. Além disso, o próprio balanço de cores feito pelas máquinas fotográficas pode fazer com que a diferença entre as imagens não seja a diferença real. Assim sendo, técnicas de captura de cor com *high dynamic range* tornam-se necessárias.

Rushmeier *et al.* [120, 119] apresenta um método de captura que utiliza *photometric stereo* combinado com um *scanner* 3D. Neste método, o *scanner* obtém dados geométricos de baixa resolução, que são complementados com uma estimativa de maior resolução das normais do objeto, derivadas a partir da geometria e das várias imagens com diferentes iluminações. Desta forma, além de corrigir as cores, uma estimativa mais precisa das normais também é obtida, o que aumenta muito o realismo dos modelos gerados.

Algumas outras técnicas foram desenvolvidas para correção da cor das imagens, no entanto elas se confundem com as técnicas utilizadas para a geração da textura definitiva. Estas técnicas serão abordadas no capítulo 11.

## 10.2 Solução Adotada

No momento, nosso *pipeline* está se utilizando apenas das imagens obtidas pelo *scanner* 3D, pois não houve tempo para trabalhar com câmeras de alta definição. Como estas imagens possuem uma correspondência um-para-um com os pontos 3D capturados, no momento estamos utilizando a técnica de *vertex color*, isto é, cada vértice possui uma cor definida, e as cores dos pixels entre os três vértices de cada triângulo são interpoladas com sombreamento Gouraud [53]. Assim sendo, o objetivo desta etapa de nosso *pipeline* foi combinar as imagens coloridas de todas as vistas, de forma a gerar cores nos vértices do modelo integrado. Este método está descrito no algoritmo 10.1.

**Algoritmo 10.1** Pseudo-código para o algoritmo de colorização dos vértices do modelo.

1: Inicialize a cor de cada vértice com o RGB acumulado $accRGB = (0, 0, 0)$ e com um peso acumulado $accW = 0$
2: **para** cada vista $i$ **faça**
3:     **para** cada vértice $v$ do modelo integrado **faça**
4:         $n \leftarrow$ normal do vértice $v$ no sistema de coordenadas da vista $i$
5:         $dir \leftarrow$ vetor que aponta da posição do vértice $v$ para a posição da câmera do *scanner*, no sistema de coordenadas da vista $i$
6:         **se** $n \cdot dir \leq 0$ **então**
7:             **interrompa-para** // cada vértice $v$
8:         **fim se**
9:         Calcule a distância de $v$ para a vista $i$ seguindo a linha de visão do *scanner*
10:         **se** distância for menor que a tolerância de erro do *scanner* **então**
11:             $accRGB \leftarrow accRGB + (rgb_i * w_i)$
12:             $accW \leftarrow accW + w_i$
13:         **fim se**
14:     **fim para**
15: **fim para**
16: Difunda cores para vértices ainda não coloridos
17: Normalize as cores de $accRGB$, dividindo cada elemento pelo $accW$ correspondente

Nas linhas 4-8 é feito um teste de *culling*, onde vértices cujas normais não apontem em direção à câmera do *scanner* são descartados. Isto é feito através da operação de produto interno da linha 6, que acaba avaliando o ângulo entre a normal e a linha de visão para o vértice.

Se o vértice passou por este teste, calculamos a sua distância até a malha da vista $i$ na linha 9. Este teste é o mesmo utilizado pelo algoritmo de integração de Curless, e além de obtermos a distância, também encontramos o ponto de interseção e sua cor $rgb_i$, bem como um peso $w_i$ para esta medida, que varia conforme o ângulo entre a linha de visão e a malha da vista. Assim sendo, medidas vistas "de frente" possuem maior peso que medidas obtidas "de raspão".

Na linha 10, confirmamos se a distância é próxima a zero (usando uma tolerância), ou seja, que este ponto foi observado pelo *scanner* 3D na vista $i$. O valor da tolerância está relacionado ao nível de ruído dos dados capturados pelo *scanner* 3D. Em nossos testes, utilizamos o valor de $0,5mm$, obtido empiricamente. Se o ponto foi observado, podemos acumular a cor e peso deste vértice em $accRGB$ e $accW$ nas linhas 11-12.

Depois deste processo, na linha 16 alguns vértices podem ainda não ter sido coloridos. Isto ocorre em vértices de buracos que foram preenchidos, por exemplo. Para atribuir cores plausíveis a estes vértices, a informação de cor dos vértices vizinhos é difundida, através das arestas que conectam estes vértices. As cores são combinadas levando-se em conta o peso acumulado $accW$.

Finalmente, na linha 17 precisamos normalizar as cores utilizando o peso acumulado $accW$, para que os valores RGB voltem à faixa de 0 a 255.

## 10.3 Resultados Obtidos e Trabalhos Futuros

Nosso método tem nos fornecido uma estimativa da cor do objeto, conforme pode ser observado na figura 10.1. Apesar de ser um método grosseiro e que deve ser muito aperfeiçoado, já apresenta resultados melhores do que os obtidos com o *software* comercial que acompanha o *scanner* 3D da Konica Minolta.

Devemos notar que nenhuma correção na cor dos vértices foi efetuada. Isto acaba fazendo com que as cores calculadas possuam efeitos de sombra, principalmente em recessos do objeto. Além disso, fica implícita a suposição de que os materiais são Lambertianos. Um coeficiente especular pode ser definido para o objeto como um todo de forma manual, mas isto é uma solução *ad hoc* insatisfatória.

Outra desvantagem da colorização pelos vértices é que a definição da imagem depende

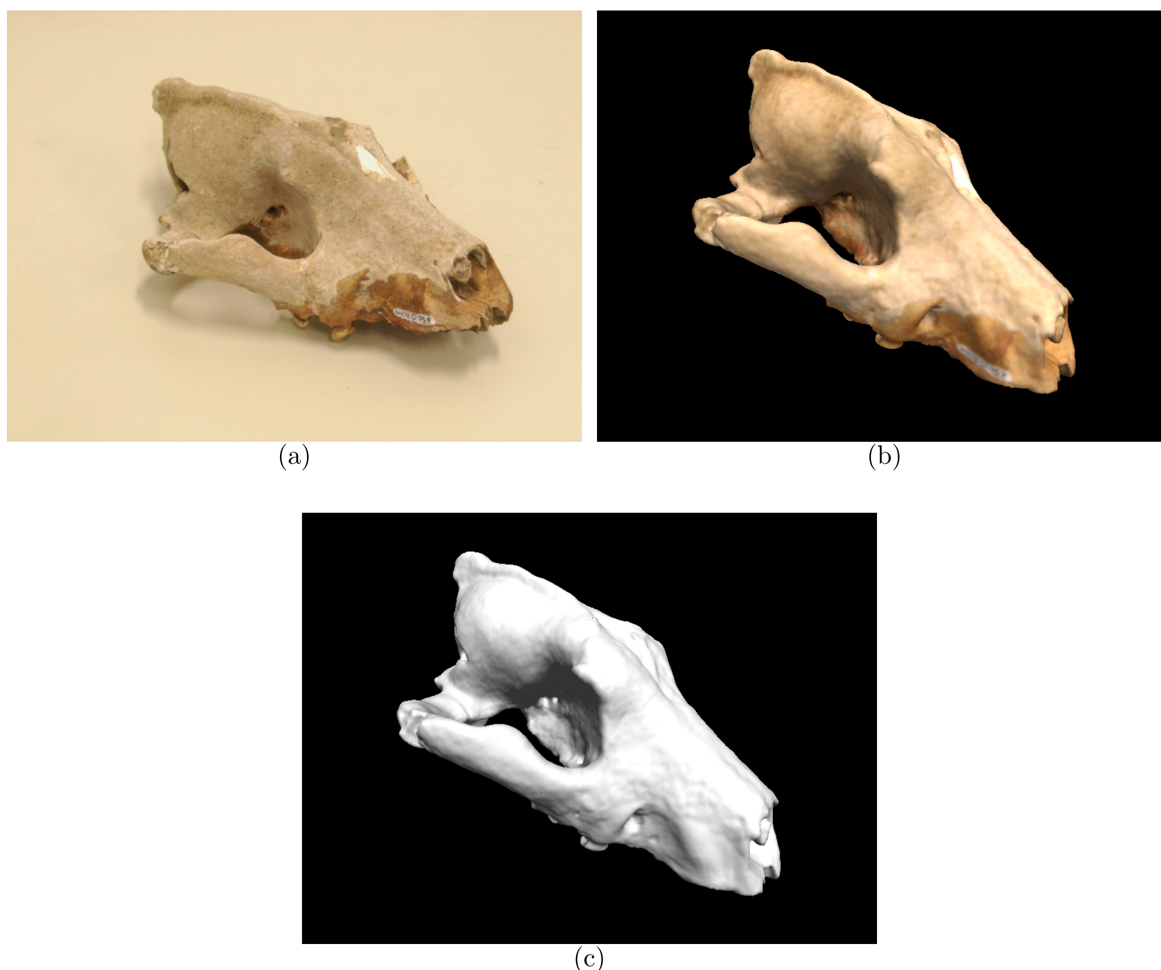


**Figura 10.1:** Processo de colorização de vértices. (a) Foto de uma das vistas capturadas; (b) Modelo reconstruído, com as cores calculadas nos vértices; (c) O mesmo modelo, sem a colorização dos vértices. Convém notar que neste exemplo não se utilizou mapeamento de texturas.

da definição geométrica. Assim, se *voxels* maiores que a distância média entre amostras forem utilizados, perdemos informações de cor das vistas. O efeito prático é um "borramento" das cores.

Outro problema consiste nos erros de alinhamento das vistas. Como em nosso *pipeline* ainda não consideramos as cores no processo de alinhamento, as diferentes imagens de cada vista podem não estar perfeitamente alinhadas. Isto se reflete novamente em um "borramento" das cores, principalmente em regiões de alta frequência, como textos escritos na superfície do objeto.

Em suma, o método atual é apenas uma tentativa inicial, que serve para dar uma idéia das cores do objeto. Os métodos apresentados na seção 10.1 devem ser melhor analisados, e os mais promissores implementados no *pipeline* para melhorar o resultado desta etapa.

# CAPÍTULO 11

# GERAÇÃO DE TEXTURAS

A geração de texturas combina o resultado das duas etapas anteriores: a parametrização da textura (capítulo 9) e as características superficiais do modelo (capítulo 10). Seu objetivo é codificar as propriedades superficiais em uma ou mais imagens (texturas), que serão posteriormente utilizadas para renderizar o modelo reconstruído.

No entanto, existem algumas soluções que não separam claramente as etapas anteriores. Outro problema que pode ser resolvido nesta etapa ao invés da etapa de geração de características da superfície, é como combinar as diversas imagens de intensidade luminosa obtidas.

## 11.1 Revisão Bibliográfica

Existem duas abordagens básicas para esta etapa: gerar a textura a partir de cores calculadas para os vértices do modelo; ou combinar as várias imagens obtidas para cada face (normalmente corrigidas para eliminar as diferenças de iluminação) em uma única imagem e parametrização.

Soucy *et al.* [140] apresenta um método para gerar texturas a partir de cores nos vértices. Ele simplifica a malha, para reduzir o número de triângulos. Ele texturiza cada triângulo individualmente, requerendo que a imagem de cada triângulo seja a metade de um quadrado. Para obter continuidade entre as faces, os vértices precisam ser mapeados para pixels exatos e triângulos vizinhos precisam que suas imagens possuam dimensões que sejam múltiplas inteiras uma da outra. Todas as imagens são combinadas em uma única textura, definindo assim a parametrização final. Este método apresenta vários problemas, como a distorção excessiva das imagens dos triângulos, e a dificuldade de uso de *mip-mapping* [53].

Métodos que combinam imagens para gerar uma textura têm que resolver o problema de oclusões. Antes de um trecho da imagem ser utilizado para texturizar uma face, é necessário confirmar que aquela face era visível do ponto de vista da câmera que a capturou. Normalmente uma renderização do modelo utilizando *z-buffer* (a partir da posição da câmera) é necessária para determinar esta informação.

Matsumoto *et al.* [99] seleciona o trecho de uma das imagens para cada triângulo do

objeto reconstruído. Para descobrir qual imagem utilizar, ele dá pesos para a imagem que apresente maior área projetada para a face, e para as imagens que sejam as mesmas das faces vizinhas, com a solução obtida resolvendo-se um problema de minimização de energia. Em seguida, os vários trechos de imagem são empacotados em uma única imagem, definindo a parametrização da textura. Um exemplo de textura gerada desta forma pode ser visto na figura 11.1.

**Figura 11.1:** Exemplo de textura gerada pelo método de Matsumoto [99]. Podemos perceber a redundância de informações na textura calculada, o gasto de espaço com a imagem de fundo, bem como uma fragmentação excessiva em alguns casos.

Rocchini *et al.* [117] divide o modelo em regiões, de forma que cada região possa ser texturizada a partir de uma única imagem. Ele usa um processo de crescimento de regiões, a partir da lista de imagens que observam cada vértice. Finalmente, ocorre um processo de alinhamento de características para que as fronteiras entre regiões não apresentem descontinuidades.

Ao invés de utilizar uma única imagem por face, alguns métodos combinam todas as imagens que contém a face. Para evitar "blurring" e outros problemas, eles requerem métodos de alinhamento que levem em consideração as imagens coloridas capturadas. Johnson e Kang [79] usam todas as imagens de cada face, ponderando-as com o ângulo formado entre a normal da face e a direção para a câmera, ou seja, dando maior peso para imagens frontais.

Pulli *et al.* [114] usa o método de *view dependent textures*, para combinar as 3 imagens capturadas mais próximas do ponto do qual se deseja observar o objeto, ponderando-as pela diferença entre o ponto de vista desejado e o ponto de vista que cada imagem foi adquirida, bem como pelo ângulo entre a superfície e a linha de visão (como em Johnson e Kang [79]), e finalmente com uma transição que dá menos peso para bordas de vistas. O problema deste método é que esta textura combinada precisa ser gerada dinâmicamente para o ponto de vista desejado. Em placas gráficas modernas, esta combinação já pode

ser feita em hardware; no entanto, isto inevitavelmente causa uma perda de desempenho, quando comparado aos métodos de texturização simples.

Neugebauer e Klein [104] também combinam várias imagens, ponderando pela ângulo entre a normal e a linha de visão, e pela distância até a borda da região que usará cada imagem. Como eles utilizam imagens que podem possuir variações de iluminação (como *specular highlights*), um terceiro peso é usado para eliminar *outliers*.

Bernardini *et al.* [13] também descreve um método onde várias imagens são combinadas. Eles utilizam o sistema apresentado em [120, 119]. Pontos aleatórios são amostrados sobre o modelo, de forma a balancear a cor das várias imagens, através de um método de mínimos quadrados. Finalmente, os mapas são combinados levando-se em conta pesos devidos ao ângulo entre normais e linha de visão, distância até a borda, e distância da câmera até a face. Este método também gera *normal maps* [49, 111] de alta definição para o modelo, obtidas com o uso de *photometric stereo*.

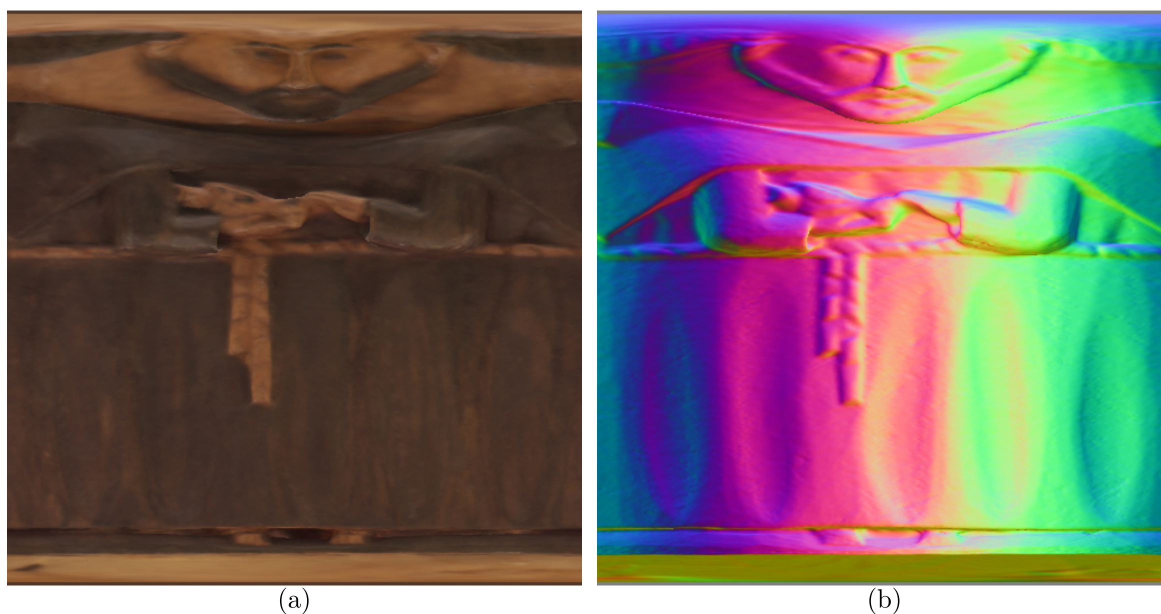
## 11.2 Solução Adotada

Como nosso pipeline calcula explicitamente a parametrização da textura e as cores nos vértices, a geração de textura é um procedimento relativamente trivial. Para fazer isso, renderizamos o modelo, utilizando as coordenadas de textura $(u, v, 0)$ como as coordenadas $(x, y, z)$ dos vértices, utilizando as cores definidas para cada vértice, que acabam sendo interpoladas pela placa gráfica 3D pela superfície de cada triângulo utilizando o sombreamento Gouraud. Esta renderização é feita utilizando-se uma matriz de projeção ortogonal, e utilizando um *render target* do tamanho da textura desejada.

Na figura 11.2, vemos duas texturas geradas com esta técnica. A figura 11.2(a) utiliza as cores dos vértices do objeto, e acaba gerando a textura difusa do objeto. A figura 11.2(b) representa o chamado *normal map* [49, 111], utilizado para obter o efeito de *bump mapping* em modelos de menor número de polígonos. Para gerar este *normal map*, ao invés da cor de cada vértice codificamos a direção da normal $(n_x, n_y, n_z)$ de cada vértice nos campos de cor $(r, g, b)$. Texturas de outras características superficiais, como o coeficiente especular de cada vértice (*specular maps*), poderiam ser geradas da mesma forma. Outros exemplos de texturas geradas podem ser observados na figura 9.10.

## 11.3 Detalhes de Implementação

Existem vários detalhes que devem ser considerados para a geração de texturas. As texturas precisam ter dimensões que sejam potências de 2, para que tenham alto desempenho



(a) (b)

**Figura 11.2:** Exemplos de mapas de textura gerados utilizando parametrização cápsula. (a) Mapa de cor difusa; (b) *Normal map*, utilizado para o efeito de *bump mapping* em modelos de menor poligonagem.

e compatibilidade com todas as placas gráficas disponíveis. Em nosso *pipeline*, utilizamos parametrizações baseadas em texturas quadradas, o que já permite atender facilmente este critério de dimensão da textura.

Para o cálculo desta dimensão, devemos levar em conta o tamanho do *voxel* utilizado na reconstrução. Como cada *voxel* possui a sua cor, devemos possuir uma textura onde *voxels* vizinhos sejam mapeados para *pixels* diferentes. Um exemplo pode ajudar a explicar este ponto. Imaginemos um objeto, que possui área de $90.000mm^2$. Ele foi parametrizado utilizando atlas de textura, e a dimensão calculada para o atlas foi $400mm \times 400mm$. Consideremos que cada voxel possui $0.5mm$ de tamanho. Neste caso, podemos dizer que o tamanho mínimo da textura deveria ser $800 \times 800$ pixels. Como 800 não é uma potência de 2, o tamanho recomendado para a textura neste case deveria ser $1024 \times 1024$ pixels.

Como algumas de nossas parametrizações não garantem que faces não sejam mapeadas no mesmo lugar (cilíndrico, esférico e cápsula), ao invés de utilizar $(u, v, 0)$ para renderizar a textura, utilizamos $(u, v, z_{max} - z_{center})$, onde $z_{center}$ é a distância de cada vértice para o centro da forma de parametrização, e $z_{max} = \max(z_{center})$ de todos os vértices. Assim, faces mais distantes do centro são mapeadas mais próximas da câmera ortogonal utilizada, e o *z-buffer* garante que elas serão renderizadas sobre as outras. Este procedimento foi utilizado para gerar os mapas das figuras 9.10(a) a 9.10(c), bem como os da figura 11.2.

Para gerar *normal maps* (como a figura 11.2(b)), devemos codificar as normais em cores $(r, g, b)$. Para tanto, usamos as fórmulas:

$$r = (n_x + 1.0) * 127.5 \quad (11.1)$$
$$g = (n_y + 1.0) * 127.5 \quad (11.2)$$
$$b = (n_z + 1.0) * 127.5 \quad (11.3)$$

O resultado renderizado para *normal maps* normalmente requer que os *pixels* sejam renormalizados. Isto ocorre porque as normais devem ter comprimento 1, e a interpolação Gouraud das normais codificadas nos vértices podem gerar cores que não correspondam a vetores unitários. Logo, devemos examinar cada *pixel*, convertê-lo para um vetor $(n_x, n_y, n_z)$, normalizar o vetor e recodificá-lo em $(r, g, b)$.

Outro problema prático que pode ocorrer é a limitação de tamanho de textura máximo da placa gráfica. As placas costumam ter um limite de $4096 \times 4096$ pixels para a máxima textura, e podem ocorrer casos em que desejemos criar uma textura maior que esse limite. Assim somos obrigados a gerar a textura por partes, combinando as partes em um único arquivo no final.

Outro problema ocorre na utilização destas texturas geradas, principalmente com parametrização por atlas. Os cortes no modelo são visíveis, e este defeito se acentua nos *mip-maps*. Isto ocorre por causa da filtragem bilinear que a placa faz ao acessar a textura, fazendo com que *pixels* fora da região mapeada sejam acessados e modifiquem a cor utilizada na renderização. Para atenuar este problema, depois de gerada a textura, difundimos os *pixels* calculados para os não calculados, usando um filtro de suavização (em nosso caso, um *box filter* $3 \times 3$ dos valores válidos). Repetimos este processo até todos os pixels possuírem uma cor definida. Na figura 11.3 podemos observar este problema, e como ele é disfarçado com o uso de nossa técnica de difusão.

## 11.4 Resultados Obtidos e Trabalhos Futuros

Na prática, apenas texturas geradas com mapeamento por atlas podem ser utilizadas, pois os erros causados pelos outros mapeamentos são muito grandes, conforme já apresentamos na figura 9.9.

Infelizmente, no momento a qualidade dos resultados é limitada pela resolução geométrica do modelo, isto é, o tamanho do *voxel* utilizado na reconstrução. Além disso, as cores nos vértices geradas na etapa anterior (vide capítulo 10) também são muito aproximadas, pois não compensaram os efeitos de sombras e *specular highlights*. Conforme as etapas anteriores forem aperfeiçoadas, as texturas geradas nesta etapa serão melhores.

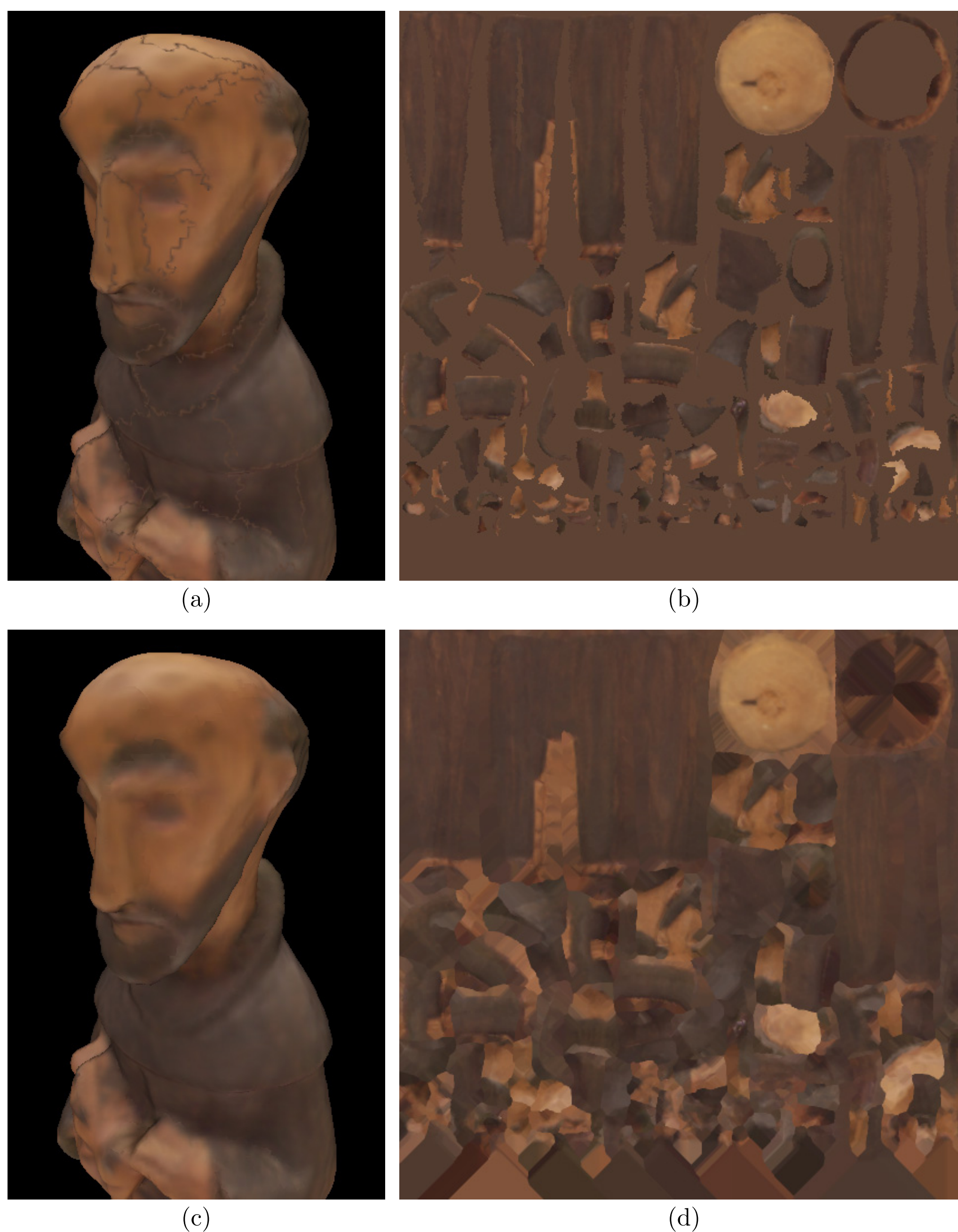


**Figura 11.3:** Defeitos de *mip-mapping* em texturas geradas com parametrização por atlas. (a) Modelo com bordas de regiões visíveis devido ao *mip-mapping* e filtragem de textura; (b) Textura utilizada na figura (a); (c) Modelo com os defeitos atenuados; (d) Textura utilizada na figura (c). Podemos perceber que *pixels* indefinidos na imagem (b) foram preenchidos com cores dos *pixels* vizinhos em (d).

No caso de utilizarmos imagens de maior resolução ao invés das imagens de intensidade adquiridas com o *scanner* 3D, a mesma metodologia poderia ser utilizada, com pequenas modificações. Ao invés de definirmos cores para os vértices, atribuiríamos coordenadas de texturas para cada foto, obtidas através de calibração. Um grau de transparência, correspondente ao peso da imagem em cada vértice, também pode ser atribuído. Em se-

guida, utilizando várias passadas de texturização (na pior das hipóteses, uma passada por imagem de intensidade adquirida), e combinando as passadas com *blending*, obteríamos uma textura final que combina as fotos de alta resolução, utilizando a parametrização escolhida. Além disso, poderíamos utilizar um *render target* de ponto flutuante para conseguir maior fidelidade de cor, já que este é um recurso amplamente disponível nas placas 3D modernas [49, 111]. Um detalhe que deve ser considerado neste processo é descobrir quais faces são observadas por cada imagem, para tratar corretamente dos casos de oclusão. Outra dificuldade pode ser a necessidade de um ajuste das matrizes de calibração, para garantir um bom alinhamento dos detalhes capturados nas diversas imagens. Sem este ajuste, é possível que a imagem resultante seja menos nítida (devido à um efeito de "borramento") do que as imagens originais.

# CAPÍTULO 12

# GERAÇÃO DE NÍVEIS DE DETALHE

Após completadas todas as etapas anteriores, já possuímos um modelo 3D de alta resolução do objeto em questão. Para fins científicos e de preservação digital, é o resultado que desejávamos. No entanto, existem outras aplicações onde este resultado ainda não é satisfatório. Por exemplo, para visualizar o modelo em tempo real, o alto número de triângulos (na casa de milhões) acaba tornando a manipulação do modelo muito desajeitada, pois a visualização é feita com uma taxa de poucos quadros por segundo. Assim sendo, seria desejável reduzir a densidade poligonal para permitir uma interação mais rápida com o objeto.

Outra vantagem de uma malha reduzida é na transferência de dados. Um modelo com milhões de triângulos ocupa muita memória, e transferir modelos deste tamanho pela Internet é impraticável.

Utilizando o *normal map* gerado na etapa anterior, podemos aliar boa qualidade com modelos compactos. Isto é possível pois a iluminação do modelo é calculada em cada *pixel*, ao invés de cada vértice, como é o padrão. Estas técnicas já são amplamente utilizadas na indústria de jogos eletrônicos [49, 111].

## 12.1 Revisão Bibliográfica

Existem muitas soluções para a simplificação da geometria de modelos 3D. Luebke [94] apresentou um *survey* dos diversos métodos, ajudando a estabelecer critérios de escolha, baseados no objetivo da simplificação. Cignoni *et al.* [32] apresenta comparativos de desempenho. Heckbert e Garland [65], e Puppo e Scopigno [115] apresentam um sumário dos vários métodos existentes.

Schroeder *et al.* [131] propôs o método de "*decimation*", para resolver o problema de simplificar malhas geradas pelo algoritmo *Marching Cubes* a partir de representações volumétricas. O método elimina vértices da malha, retriangulando o buraco gerado. Apesar de extremamente rápido, não consegue gerar resultados com muita fidelidade ao modelo original, principalmente em regiões de maior curvatura.

Rossignac e Borrel [118] usam o método de *Vertex Clustering*, onde um grupo de vértices é agrupado em um único vértice, para simplificar modelos. Este método, apesar

de rápido, não preserva a topologia dos modelos, e pode gerar topologias inválidas.

Eck *et al.* [44] usa *Multiresolution Analysis* (MRA), que é baseada numa representação de *wavelets*, para gerar níveis de detalhe de um modelo, através da adição ou eliminação de coeficientes *wavelet*. Infelizmente, o método se comporta como um filtro de suavização geométrico, que não funciona bem para objetos com arestas muito definidas.

Cohen *et al.* [34] introduz *Simplification Envelopes* para garantir limites de fidelidade à simplificação. Estes envelopes nada mais são do que superfícies deslocadas a partir do modelo original, para fora e para dentro do objeto, pelo valor de uma tolerância. O método garante que a simplificação esteja contida no espaço entre os envelopes. Infelizmente, o método possui uma implementação muito complexa. Este método foi expandido por Cohen, Olano e Manocha [33] para restringir a simplificação de cores e normais, além da posição dos vértices. Isto é conseguido através do uso de texturas e *normal maps*.

O algoritmo *Quadric Error Metrics* de Garland e Heckbert [56] é um dos mais utilizados, por conseguir boa fidelidade e ser muito rápido. Ele agrupa pares de vértices, e usa como métrica para guiar o processo a soma do quadrado da distância dos vértices para os planos das faces do modelo. Cada vértice possui uma matriz simétrica $4 \times 4$ que armazena os coeficientes desta equação quadrática de representação de erro. Hoppe [70] aperfeiçoou esta métrica, para levar em conta outros atributos dos vértices, como normais, cores ou coordenadas de textura.

Lindstrom e Turk [91] guiam o seu processo de simplificação pelo resultado visual da mesma. A idéia é eliminar arestas, juntando os seus vértices em um novo (*edge collapse*). Para avaliar o impacto da simplificação, imagens são renderizadas de vários pontos de vista, e as imagens simplificadas são comparadas com as originais, e a métrica de erro é calculada. A grande vantagem deste método é que ele avalia a qualidade como os usuários, ou seja, através das imagens resultantes. A desvantagem é o grande poder de processamento necessário, mesmo utilizando placas gráficas avançadas.

Hoppe [68, 69, 129] sugeriu um método dinâmico para a geração de níveis de detalhe de um modelo, conhecido como *Progressive Meshes*. A idéia básica do método é reduzir a quantidade de faces e vértices de um modelo, através de *edge collapses*. Nesta operação, uma aresta é eliminada, e os dois vértices da aresta são unidos em uma nova posição. Desta forma, cada *edge collapse* reduz um vértice do modelo. Para a escolha da melhor aresta a ser eliminada, existe uma medida de erro, que é avaliada para cada potencial aresta a ser eliminada, e a cada iteração do algoritmo se escolhe a aresta que quando eliminada acrescenta o menor erro ao modelo 3D. A operação de *edge collapse* é utilizada por vários algoritmos de simplificação, e pode ser melhor entendida através da figura 12.1. Armazenando a sequência de *edge collapses*, podemos criar uma faixa contínua de níveis

de detalhe para o objeto.

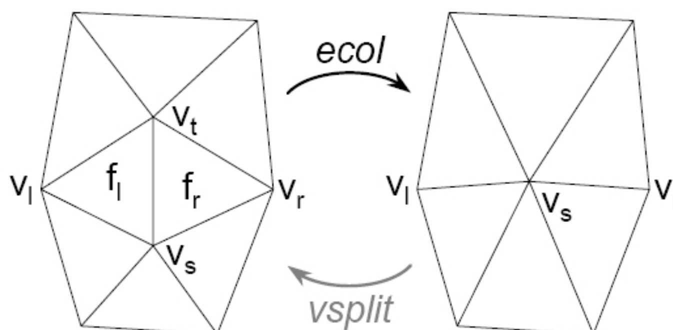
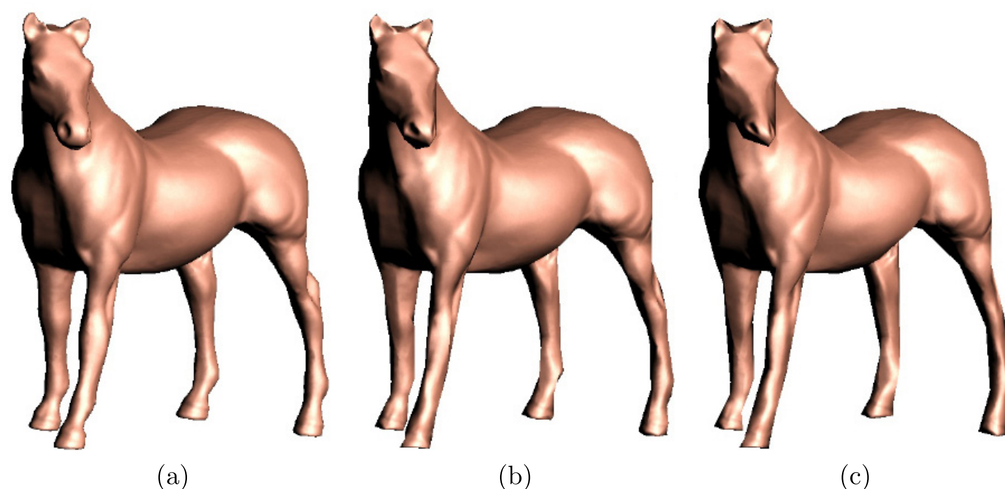



**Figura 12.1:** Operação de *edge collapse* do algoritmo de Hoppe [68]. Uma aresta é eliminada, e um novo vértice é criado em seu lugar. A operação inversa, que expande o vértice para a aresta original, é chamada de *vertex split*.

## 12.2 Solução Sugerida

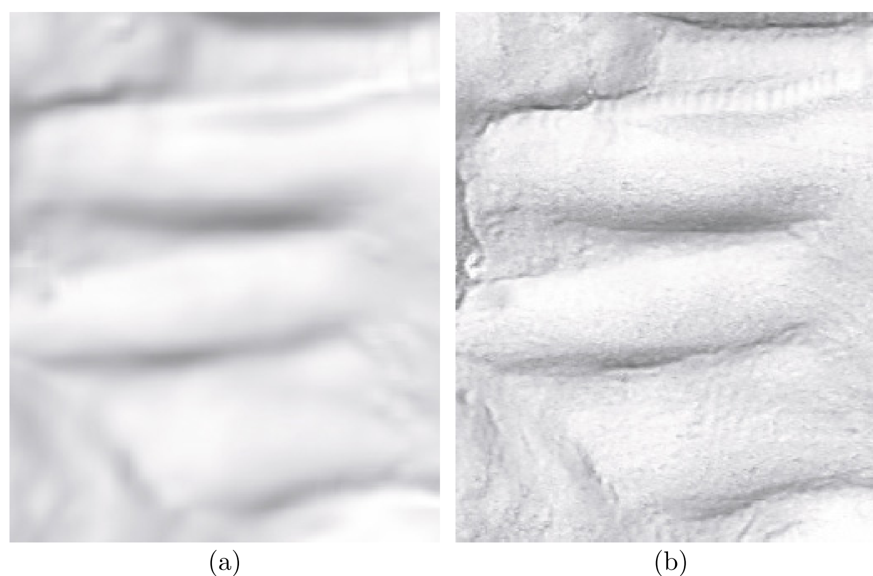
Infelizmente, ainda não incorporamos nenhum algoritmo de simplificação de modelos em nosso *pipeline*. A análise dos diversos algoritmos existentes, nos leva a sugerir o uso de *Progressive Meshes*. No entanto, para o cálculo dos mesmos, sugerimos o método de *Quadric Error Metric* aperfeiçoado por Hoppe [70], pois possui qualidade e desempenho superiores à técnica original apresentada em [68]. O uso de *progressive meshes* apresenta uma série de vantagens:

- Eles geram um espectro contínuo de níveis de detalhe do modelo em questão. Em outras palavras, podemos escolher um número arbitrário de vértices (menor ou igual ao número de vértices do modelo inicial), e obter o modelo com este número de vértices. Assim sendo, podemos adaptar a complexidade do modelo à capacidade de processamento gráfico de cada usuário;
- Esta técnica permite a visualização progressiva de um modelo, mesmo antes de ele ter sido completamente carregado. Isto pode ser muito útil em visualizadores que recebam os modelos via Internet;
- Utilizando os *progressive meshes* em conjunto com a técnica de *normal mapping*, a perda de qualidade visual é praticamente imperceptível [129]. Apenas a silhueta do objeto fica comprometida, mas o interior do modelo retém a alta qualidade do modelo de alta poligonagem. Esta técnica vem sendo amplamente utilizada em jogos eletrônicos para conciliar restrições de *hardware* e alta qualidade [49, 111]. Podemos observar um exemplo desta técnica nas figuras 12.2 e 12.3.



(a) (b) (c)

**Figura 12.2:** *Progressive Mesh* aliado a normal mapping [129]. (a) Modelo original, com 10.000 faces; (b) 1200 faces; (c) 700 faces. Podemos perceber que mesmo no modelo com extremamente baixa poligonagem (c), o interior do modelo apresenta os detalhes do modelo original (a).



(a) (b)

**Figura 12.3:** *Normal mapping* aplicado a um modelo de baixa resolução [15]. (a) Imagem renderizada da geometria do modelo, com o método usual; (a) Geometria iluminada utilizando-se o *normal map*. O acréscimo de detalhes é notável.

Um ponto a ser lembrado é que este processo de geração de níveis de detalhe pode ser demorado, uma vez que ele somente será feito uma única vez. Assim, métodos de alta qualidade podem ser utilizados.

# CAPÍTULO 13

# CONCLUSÃO

O principal objetivo de nosso trabalho foi avaliar os algoritmos existentes de reconstrução digital de modelos 3D, utilizando *scanners* de triangulação *laser*. E a única forma de corretamente avaliar os algoritmos de cada etapa, e os relacionamentos entre as etapas, foi através da implementação do *pipeline* completo de reconstrução 3D.

Foi somente através destas experiências empíricas que pudemos validar ou refutar as diversas afirmações encontradas nos artigos avaliados, conclusões estas que foram apresentadas durante os capítulos anteriores. Outras consequências benéficas deste esforço de implementação foram:

- Criação de ferramentas que permitem o uso do *scanner* 3D adquirido (*Vivid 910* da Konica Minolta) em projetos de preservação digital. Isto foi necessário pois o *software* que acompanha o *scanner* só é capaz de reconstruir objetos muito simples, que requerem poucas vistas capturadas;
- Permitir a continuação de pesquisas na área, através da identificação dos pontos problemáticos que precisam de melhorias;
- Facilitar a implementação e comparação de resultados de novas técnicas que venham a ser desenvolvidas.

## 13.1 Resultados Obtidos

Rapidamente tornou-se clara a necessidade prática de se implementar o *pipeline*, uma vez que o *software* que acompanha o *scanner* 3D é extremamente ineficaz. Alguns problemas encontrados no *software* da Konica Minolta foram:

- O plano de apoio e outros *outliers* deviam ser manualmente eliminados, o que era uma atividade tediosa e suscetível a erros;
- O tempo perdido no pré-alinhamento de vistas era muito grande, pela necessidade de se marcar pontos correspondentes em cada par de vistas, e devido às restrições de movimentação da câmera;

- Quando se tentou a reconstrução de objetos maiores, o programa simplesmente era finalizado por falta de memória, mesmo sendo executado em um sistema com 4GB de RAM;
- O preenchimento de buracos é inutilizável. Cada buraco deve ser preenchido individualmente, com o usuário testando manualmente 3 algoritmos, que só funcionam para topologias simples. Normalmente o programa é finalizado quando se tenta preencher um buraco mais complexo;
- As cores calculadas para o modelo integrado são completamente inconsistentes;
- As únicas formas de parametrização de textura existentes são a cilíndrica e esférica, que não servem para objetos complexos, como foi apresentado no capítulo 9.

Nas figuras 13.1 e 13.2, podemos observar modelos reconstruídos com o *software* que acompanha o *scanner*. Em ambos os casos, não foi possível preencher os buracos, porque o programa era finalizado. As cores geradas são inaceitáveis em ambos os modelos. A geometria da estátua de São Francisco ficou aceitável, por se tratar de um objeto simples. Já no caso do galo, muitos *outliers* ocorreram nas regiões de bordas, que possuem muitos buracos que não foram preenchidos. O resultado obtido utilizando-se o nosso *pipeline* pode ser observado nas figuras 13.6 e 13.9, respectivamente.

Reconstruímos vários objetos para avaliar as diversas técnicas implementadas em nosso *pipeline*. Na tabela 13.1 podemos observar algumas informações sobre os objetos reconstruídos. A tabela foi ordenada pelo número de faces resultantes dos modelos 3D.

**Tabela 13.1:** Objetos modelados com o *pipeline* de reconstrução 3D proposto.

| Objeto | Nº de Vistas | Nº de Pares | Tam. Dados | Tam. *Voxel* | Dimensões do Volume | Nº de Vértices | Nº de Faces | Tempo |
|---|---|---|---|---|---|---|---|---|
| Coruja | 22 | 33 | 266MB | 0,3mm | 106×118×206 | 86.128 | 172.252 | 152s |
| Cabeça | 30 | 40 | 411MB | 0,4mm | 125×104×399 | 129.960 | 259.916 | 269s |
| Besouro | 43 | 69 | 979MB | 0,3mm | 100×669×708 | 136.706 | 273.424 | 322s |
| São Francisco | 24 | 44 | 358MB | 0,4mm | 113×117×405 | 153.360 | 306.716 | 182s |
| Alamito | 24 | 41 | 369MB | 0,5mm | 164×161×278 | 171.262 | 342.520 | 239s |
| Buddha | 18 | 18 | 94MB | 0,7mm | 163×173×317 | 222.940 | 445.306 | 195s |
| Galo | 32 | 36 | 671MB | 0,5mm | 446×384×243 | 296.786 | 593.584 | 963s |
| Stenzel | 22 | 36 | 330MB | 0,8mm | 252×213×525 | 333.452 | 666.900 | 565s |
| Cavalo | 34 | 53 | 602MB | 0,5mm | 363×677×266 | 337.810 | 675.632 | 1.357s |
| Carybe | 19 | 26 | 319MB | 0,5mm | 210×285×492 | 357.348 | 714.696 | 589s |
| Vaso | 36 | 40 | 621MB | 0,2mm | 241×242×419 | 396.495 | 792.809 | 1.373s |
| Protocyon | 56 | 104 | 1,22GB | 0,5mm | 190×455×296 | 436.863 | 873.746 | 1.348s |
| Vitor | 93 | 212 | 3,36GB | 1,0mm | 266×301×549 | 605.315 | 1.210.630 | 3.214s |
| Habacuc | 123 | 197 | 6,09GB | 1,5mm | 278×497×584 | 707.063 | 1.409.939 | 5.271s |

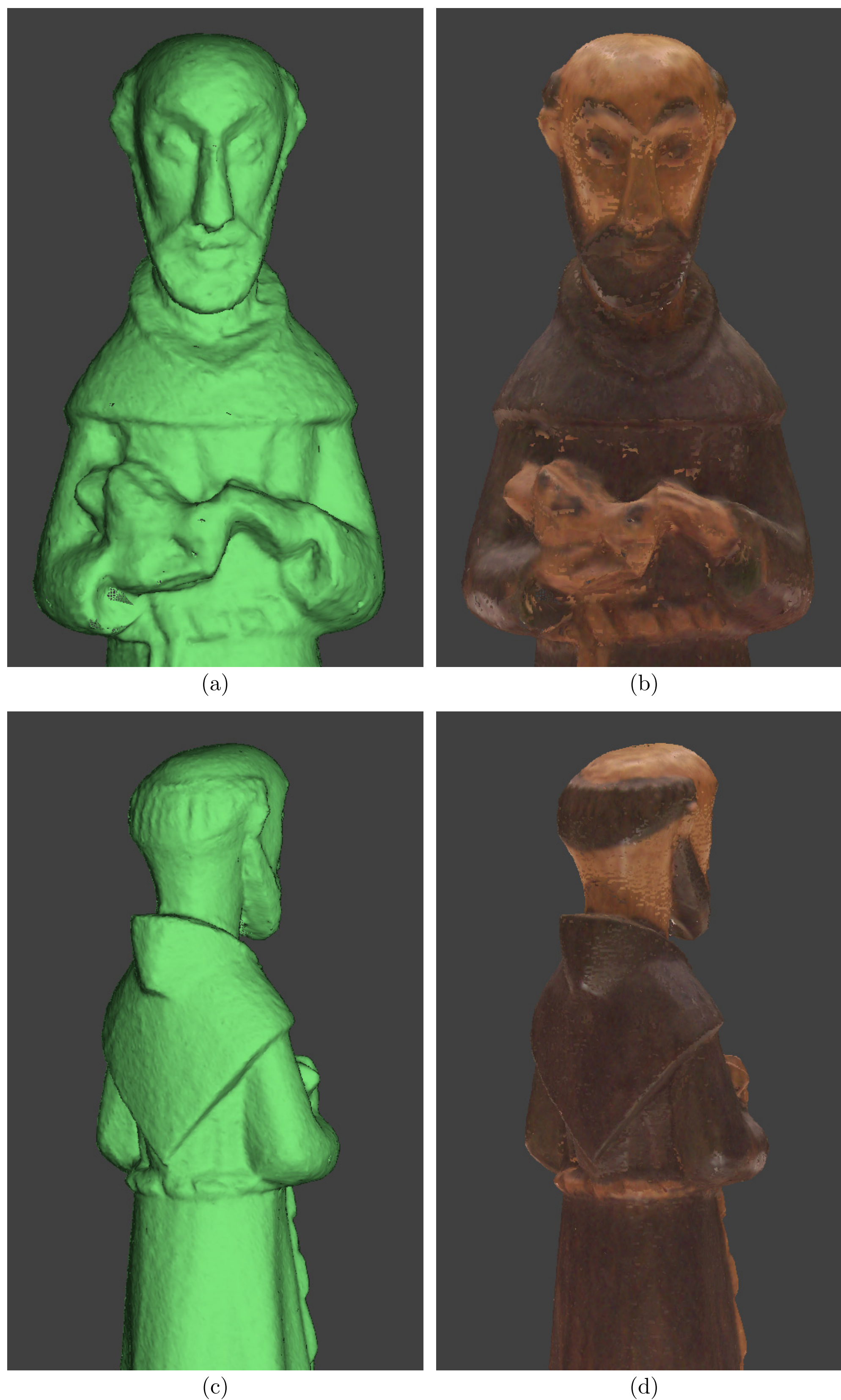
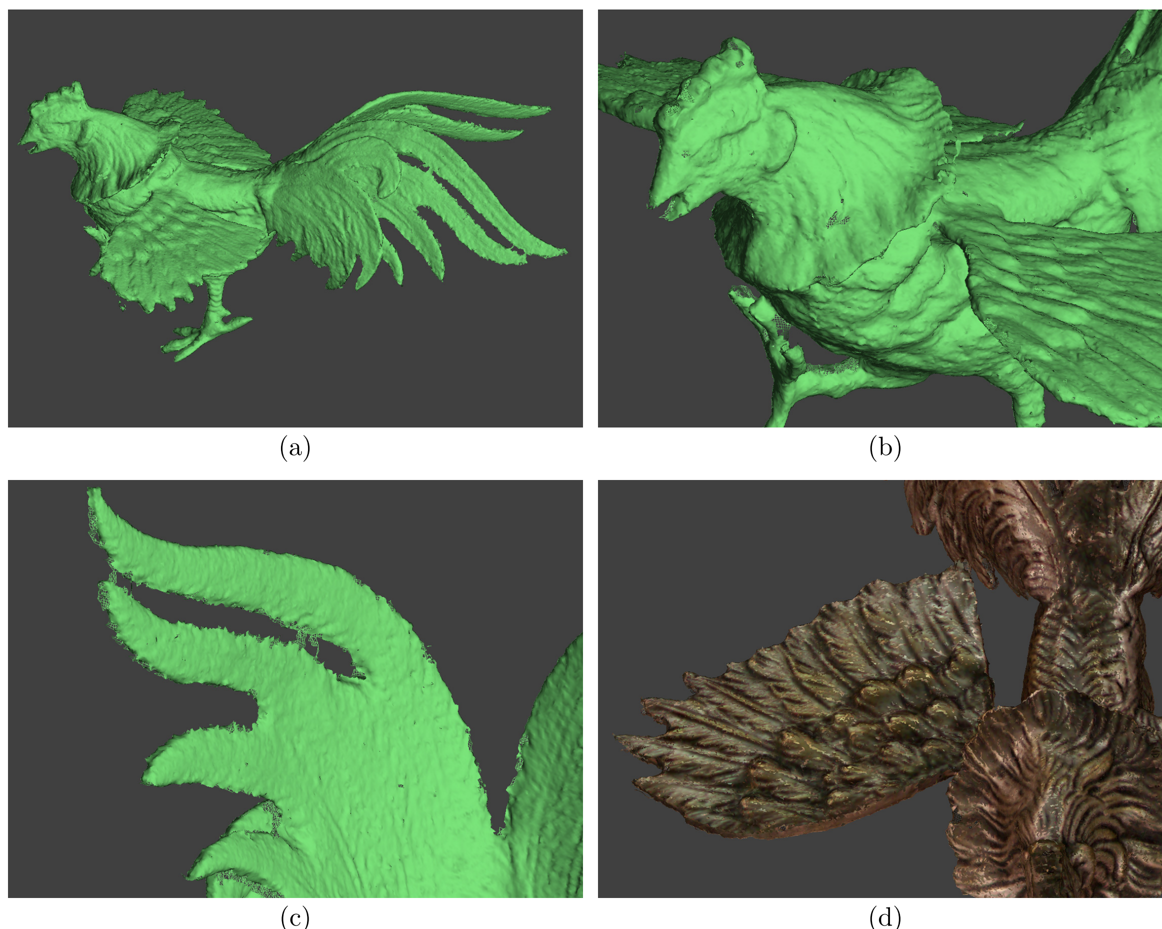


**Figura 13.1:** São Francisco reconstruído utilizando o *software* que acompanha o *scanner* Vivid 910 da Konica Minolta. O modelo gerado possui 189.972 faces, e algumas faces se interceptam. O preenchimento de buracos não foi possível, pois fazia o programa ser encerrado. Nota-se como as cores calculadas são incorretas, principalmente na região da cabeça.



(a) (b) (c) (d)

**Figura 13.2:** Galo reconstruído utilizando o *software* que acompanha o *scanner* Vivid 910 da Konica Minolta. O modelo gerado possui 441.749 faces, e algumas faces se interceptam. O preenchimento de buracos não foi possível, pois fazia o programa ser encerrado. Em (d) nota-se como as cores calculadas são incorretas, variando abruptamente de uma face vizinha para a outra.

A coluna "Nº de Vistas" da tabela 13.1 nos informa o número de capturas efetuadas de cada objeto. Estas vistas são relacionadas em pares para o seu alinhamento via ICP, sendo o número de pares informado na coluna "Nº de Pares".

A coluna "Tam. Dados" se refere aos arquivos temporários criados, que possuem todas as estruturas de dados auxiliares de cada vista. Através dela, podemos ter uma idéia mais precisa da quantidade de informações que devem ser processadas para reconstruir cada objeto. Para economizar memória, criamos um mecanismo de *cache* que se utiliza destes arquivos temporários, de forma a viabilizar a reconstrução dos objetos mais complexos.

O tamanho do *voxel* escolhido aparece na coluna "Tam. *Voxel*". Este tamanho está relacionado à resolução das vistas capturadas pelo *scanner*. No caso dos objetos maiores (os últimos da tabela 13.1), utilizamos *voxels* maiores do que a resolução do *scanner* para acelerar um pouco o processo de reconstrução, pois a etapa de integração volumétrica

ainda não foi otimizada. A coluna “Dimensões do Volume” está especificada em *voxels*.

A geometria do modelo 3D é indicada pelas colunas “Nº de Vértices” e “Nº de Faces”. Note-se que os modelos gerados possuem muitos triângulos, para capturar com precisão os detalhes da superfície. Estes detalhes são preservados no *normal map* gerado.

O tempo total de execução pode ser observado na coluna “Tempo” da tabela 13.1. No entanto, uma avaliação mais precisa da complexidade e do desempenho de cada etapa pode ser obtida através da tabela 13.2, que lista o tempo gasto em cada etapa do *pipeline*.

**Tabela 13.2:** Tempos de execução das etapas do *pipeline* de reconstrução 3D proposto.

| Objeto | ICP[1] | Alin. Global | Integr. Volum. | Prench. Buracos | *Marching Cubes* | Cor dos Vértices | Geração Textura | **Tempo Total** |
|---|---|---|---|---|---|---|---|---|
| Coruja | 28,9s | 0,5s | 109,2s | 7,2s | 2,4s | 1,2s | 2,3s | 151,8s |
| Cabeça | 43,8s | 0,8s | 207,8s | 8,5s | 2,7s | 2,3s | 3,2s | 269,2s |
| Besouro[2] | 105,2s | 3,2s | 115,3s | 19,6s | 18,0s | 12,0s | 49,0s | 322,3s |
| São Francisco | 41,9s | 0,8s | 126,4s | 3,6s | 3,2s | 2,2s | 3,5s | 181,7s |
| Alamito | 46,2s | 0,6s | 174,0s | 6,5s | 4,0s | 2,3s | 5,0s | 238,7s |
| Buddha | 11,6s | 0,4s | 142,1s | 26,3s | 4,8s | 2,1s | 7,9s | 195,2s |
| Galo | 70,9s | 2,2s | 780,7s | 82,5s | 8,8s | 6,2s | 11,4s | 962,8s |
| Stenzel | 38,6s | 1,1s | 501,6s | 1,8s | 6,8s | 4,3s | 10,8s | 564,9s |
| Cavalo | 77,3s | 3,2s | 1.199,7s | 33,6s | 13,3s | 7,4s | 22,3s | 1.356,9s |
| Carybe | 33,5s | 0,6s | 504,5s | 24,1s | 8,0s | 3,8s | 14,8s | 589,5s |
| Vaso | 68,7s | 2,1s | 1.248,3s | 9,5s | 8,4s | 7,5s | 28,3s | 1.372,9s |
| Protocyon | 131,9s | 13,5s | 1.046,7s | 113,4s | 11,5s | 14,5s | 16,3s | 1.347,8s |
| Vitor | 538,8s | 49,0s | 2.441,6s | 21,6s | 12,0s | 125,6s | 25,5s | 3.214,2s |
| Habacuc | 685,7s | 191,9s | 4.169,9s | 14,4s | 16,7s | 163,9s | 28,9s | 5.271,4s |

Podemos observar na tabela 13.2 que as duas etapas mais custosas são o alinhamento por pares e a integração volumétrica. A etapa de integração volumétrica ainda não foi otimizada, pois ainda não estamos satisfeitos com os resultados obtidos pelo algoritmo híbrido. Utilizando-se *3D scan conversion*, a integração volumétrica pode ser acelerada significativamente. Convém notar que mais experimentos são necessários para a etapa de integração, principalmente o teste de outros algoritmos, para aperfeiçoar os resultados da etapa; logo a otimização, no momento, não é prioritária.

Os objetos listados nas tabelas 13.1 e 13.2 podem ser observados nas figuras 13.3 a 13.16. Foram capturados e reconstruídos diversos tipos de objetos: obras

[1]Este tempo não inclui o pré-alinhamento manual, mas inclui o tempo de pré-processamento para a geração do arquivo temporário para cada vista.

[2]O besouro foi integrado utilizando o algoritmo de Curless modificado, pois o algoritmo híbrido acabava deixando buracos muito grandes, que não eram preenchidos corretamente. Além disso, o modelo foi reconstruído com o plano de apoio, também para ajudar no preenchimento de buracos. Isto explica as grandes dimensões do volume na tabela 13.1.

de arte do Museu de Arte Metropolitana de Curitiba (MUMA), obras de arte da coleção da Reitoria da UFPR, uma réplica do profeta Habacuc de Aleijadinho, presente no Museu de Belas Artes da UFMG, fósseis do museu de História Natural da UFPR, insetos para o projeto Taxonline coordenado pela UFPR, além de vários outros objetos de teste. Procuramos casos reais que realmente testassem os limites do *pipeline* implementado, de forma que os problemas dos diversos algoritmos pudessem ser diagnosticados.

Uma grande dificuldade é avaliar quantitativamente a precisão dos modelos 3D reconstruídos. Os casos interessantes surgem em objetos complexos (como o *protocyon*, vide figura 13.14), e não temos nenhuma forma precisa para extrair medidas deste tipo de objeto. Processos tradicionais de medição, como o uso de micrômetros, fornecem apenas distâncias entre dois pontos, e há a imprecisão na definição de exatamente quais pontos foram usados como referência. Assim sendo, estes métodos não conseguem estimar a verdadeira superfície de um objeto.

Outra abordagem seria construir fisicamente um objeto a partir de um modelo 3D, e em seguida capturar e reconstruir o modelo. Isto permitiria a comparação entre o modelo original e o modelo reconstruído. Infelizmente, esta abordagem também não é perfeita: todo processo de fabricação está sujeito à tolerâncias, e o objeto físico criado não passa de uma aproximação do modelo 3D que o originou. Desta forma, a comparação entre o modelo reconstruído e o modelo original não estaria apenas medindo os erros do algoritmo de reconstrução, mas também os erros do processo de fabricação.

Devido à estas incertezas, preferimos efetuar apenas comparações qualitativas entre os algoritmos, que foram apresentadas no decorrer do trabalho.

Sob o ponto de vista prático, nosso *pipeline* de reconstrução 3D foi implementado em C++ no programa *ModelTool*. Ele é multi-plataforma, podendo ser executado tanto em Linux quanto em Windows, e se utiliza de OpenGL como API de renderização. Além dele, foi implementado o programa *VividControl*, também em C++, para controlar o processo de aquisição de dados do *scanner* Vivid 910 da Konica Minolta. No entanto, o *VividControl* funciona apenas em plataforma Windows, pois é a única plataforma suportada pelo SDK do fabricante [85]. O *ModelTool* possui 28.220 linhas de código, e o *VividControl*, 9.048.

## 13.2 Principais Contribuições

Além de apresentar uma visão geral do processo de reconstrução digital, e de um levantamento da bibliografia sobre o tema, várias contribuições foram apresentadas neste trabalho:

- Apresentamos um levantamento dos vários tipos de defeitos comuns nos dados capturados com *scanners* de triangulação laser. Simplesmente adicionar ruído gaussiano a modelos perfeitos não é suficiente para validar técnicas de reconstrução digital, devido à severidade dos problemas encontrados em casos reais;
- Apresentamos um algoritmo robusto para a detecção e eliminação dos planos de apoio de objetos capturados. Este é um problema prático que não é abordado pela bibliografia. Nossa solução é muito mais confiável e prática do que as técnicas de separação entre objeto e fundo utilizadas em processamento de imagens;
- Implementamos um mecanismo de pré-alinhamento manual que é muito mais prático do que o método clássico de selecionar pontos correspondentes nos pares de vistas. Nosso método consiste em posicionar uma vista sobre a outra através do *mouse*, de forma aproximada. A convergência da nossa variante do ICP garante a eficácia do nosso método;
- Propusemos uma variante do ICP dividida em duas fases: a primeira maximiza a convergência do algoritmo, e a segunda consegue maior precisão através de uma seleção robusta dos pontos nas vistas;
- Implementamos e avaliamos criticamente os algoritmos de Curless e de Wheeler de integração volumétrica, demonstrando os defeitos e falhas teóricas dos métodos. Além disso, sugerimos várias melhorias em ambos os algoritmos;
- Apresentamos um novo algoritmo híbrido de integração volumétrica, que consegue efetivamente detectar e eliminar a maioria dos *outliers* das vistas, gerando um modelo integrado de melhor qualidade, e mais adequado para a etapa de preenchimento de buracos;
- Demonstramos como o algoritmo de preenchimento de buracos de Davis é muito sensível a *outliers*, e que para bons algoritmos de integração ele apresenta resultados muito satisfatórios;
- Apresentamos um algoritmo simples e prático para a criação de atlas de textura;
- Apresentamos uma forma de gerar texturas utilizando as placas gráficas.

## 13.3 Trabalhos Futuros

Como quase sempre ocorre no meio científico, este trabalho trouxe algumas soluções, mas abriu diversas novas avenidas para futuras pesquisas. Fica claro que ainda não existem técnicas que garantam reconstruções precisas. Melhorias podem ser introduzidas em todo o *pipeline*, desde mudanças no mecanismo de aquisição [36, 107] até o processo de integração, como demonstrado através de nosso algoritmo híbrido. Como utilizamos

um *scanner* 3D comercial, não tínhamos como alterar o processo de aquisição; desta forma, concentramos nossos esforços em métodos de eliminação de defeitos nas etapas subsequentes do *pipeline*.

Algumas problemas que encontramos, e que podem ser alvos de trabalhos futuros são:

- Facilitar o planejamento das capturas das vistas. Atualmente, é necessário uma pessoa experiente para que todas as vistas necessárias para uma reconstrução completa sejam capturadas. Desenvolver ferramentas e técnicas que auxiliem nesta etapa pode contribuir significativamente com a eficiência e qualidade do processo como um todo;
- Fazer experimentos com as técnicas de pré-alinhamento, de forma a eliminar uma etapa que continua manual em nosso *pipeline*;
- Aprimorar o alinhamento das vistas, através de novas métricas e com técnicas eficientes de eliminação de *outliers*. Uma possibilidade é usar realimentação de estágios posteriores (que identificaram *outliers*), de forma a aprimorar o alinhamento;
- Desenvolvimento de técnicas mais precisas de integração de vistas. Os métodos avaliados ainda precisam de muitas melhorias, antes de serem capazes de garantir boas reconstruções;
- Aprimorar o processo de geração de texturas para os objetos reconstruídos. Nosso *pipeline* ainda fornece resultados muito aproximados para estas etapas finais da reconstrução. Convém notar que melhorias nestas etapas devem ser acompanhadas por melhorias nos mecanismos de renderização dos modelos gerados, de forma a permitir renderizações realísticas;
- Incorporar técnicas de geração de níveis de detalhes, como as apresentadas no capítulo 12.

Esperamos que nosso trabalho possa servir de base para os futuros esforços de pesquisa na área de preservação digital. O escopo do trabalho foi intencionalmente escolhido para ser bem abrangente, de forma que cada pesquisador futuro pudesse ter uma visão mais completa do processo de reconstrução, e de como as diversas pesquisas em áreas mais específicas deveriam relacionar-se, para que um processo completo, integrado e confiável de reconstrução digital possa ser obtido.

**Figura 13.3:** Pequena estátua em madeira de uma coruja, do acervo do grupo IMAGO. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 172.252 faces, e foi reconstruído com *voxels* de 0,3mm.

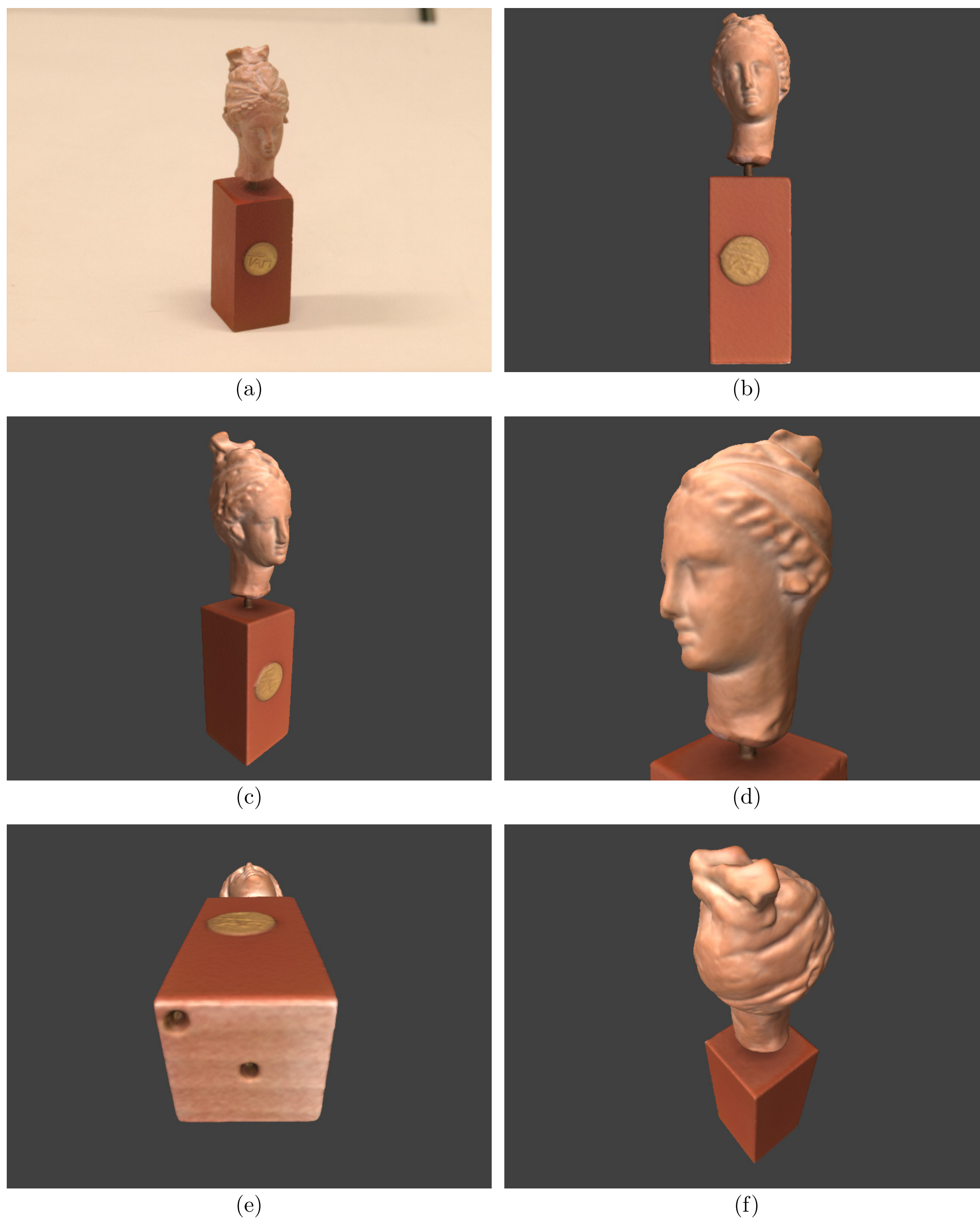


**Figura 13.4:** Réplica da cabeça de uma estátua grega, do acervo do grupo IMAGO, obtida no congresso IEEE ICIP 2002 em Tessaloniki, Grécia. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 259.916 faces, e foi reconstruído com *voxels* de 0,4mm.

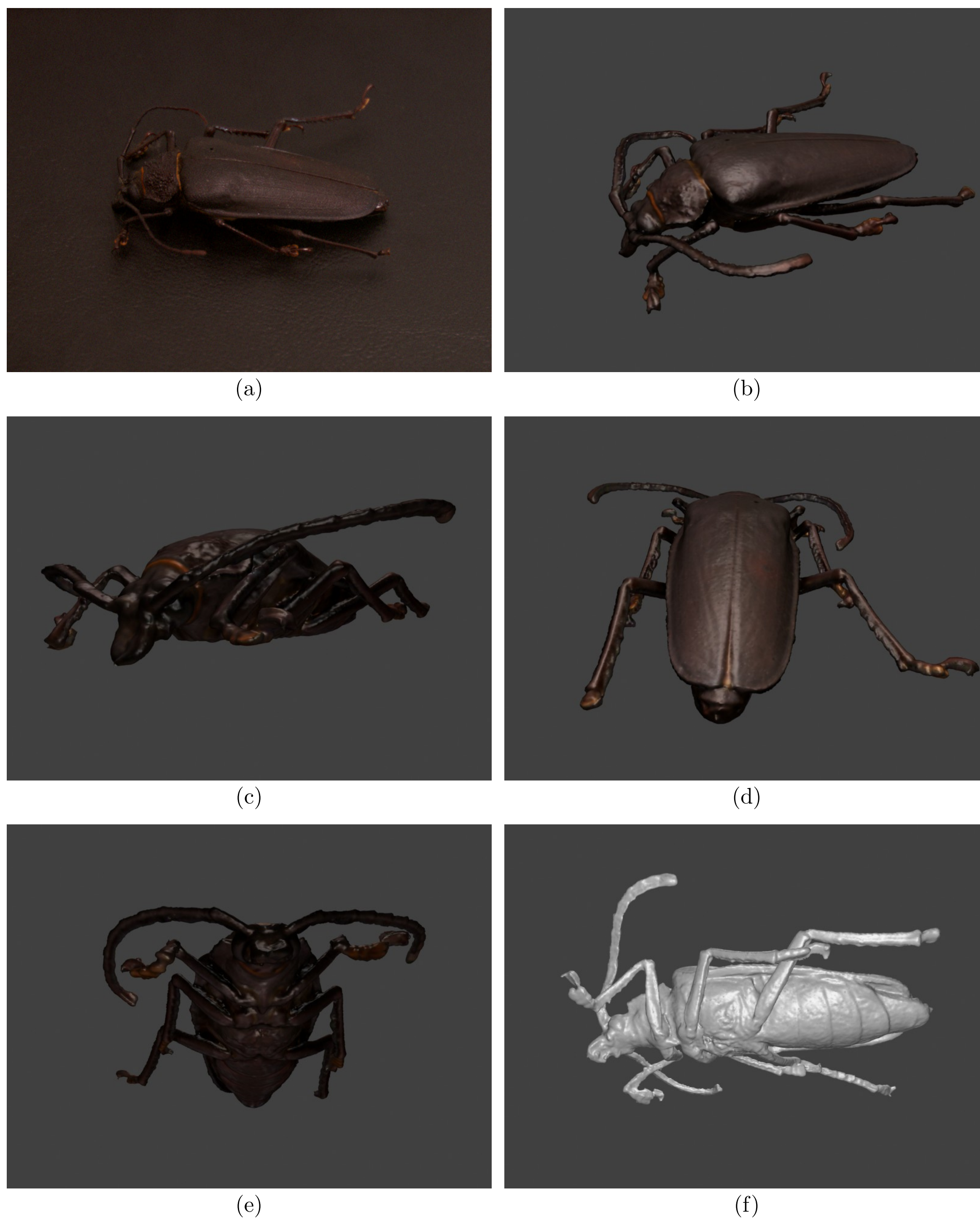



**Figura 13.5:** Reconstrução 3D de um besouro, de propriedade das Coleções Biológicas da UFPR. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 273.424 faces, e foi reconstruído com *voxels* de 0,3mm. Na figura (f) observamos o modelo sem cor, de forma a realçar a geometria reconstruída. Apesar do resultado relativamente bom, utilizar *scanners* de triangulação *laser* para capturar insetos não é recomendado, pois detalhes como pêlos não são capturados. Além disso, a precisão obtida é questionável, pois os dados capturados não são de boa qualidade.

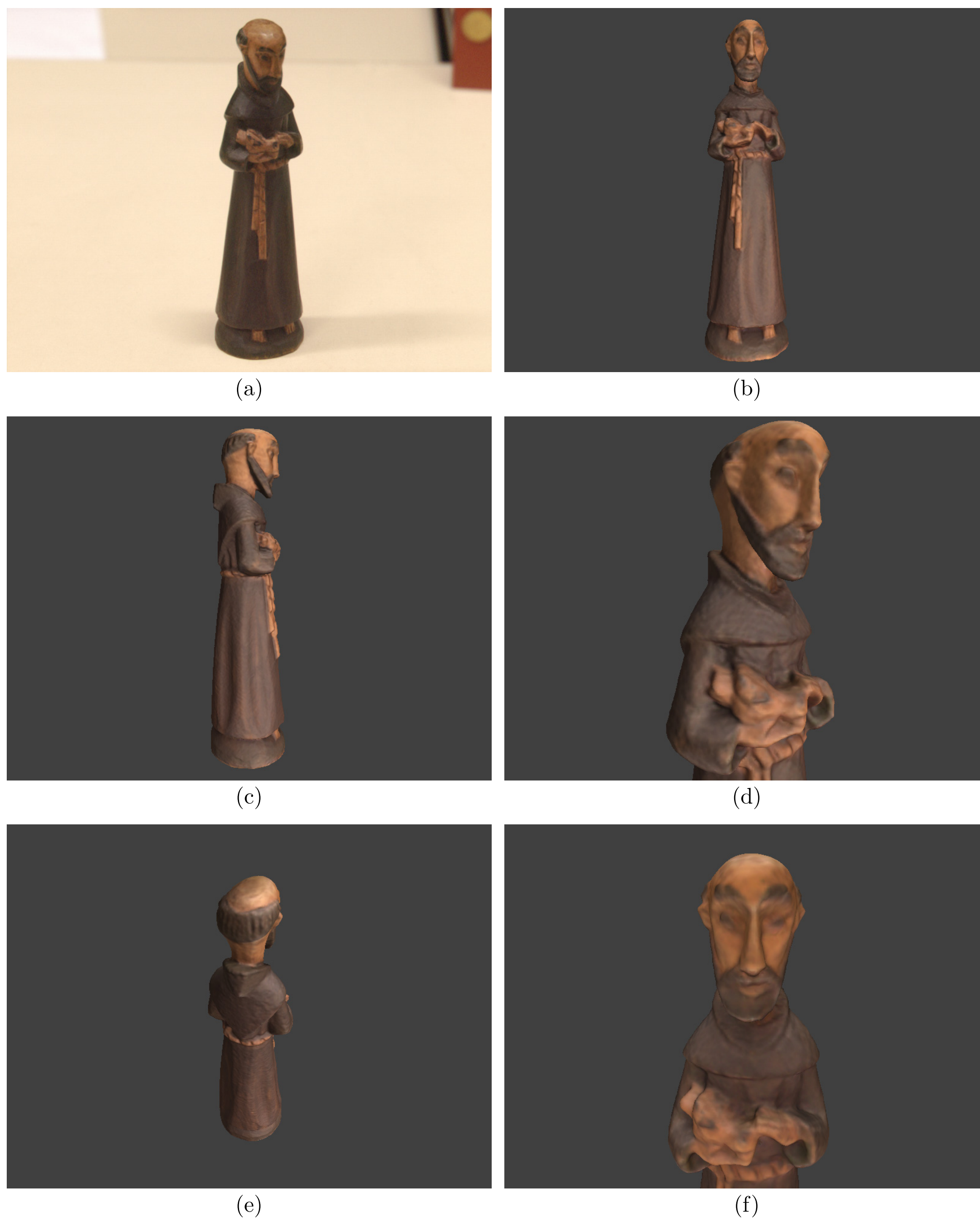
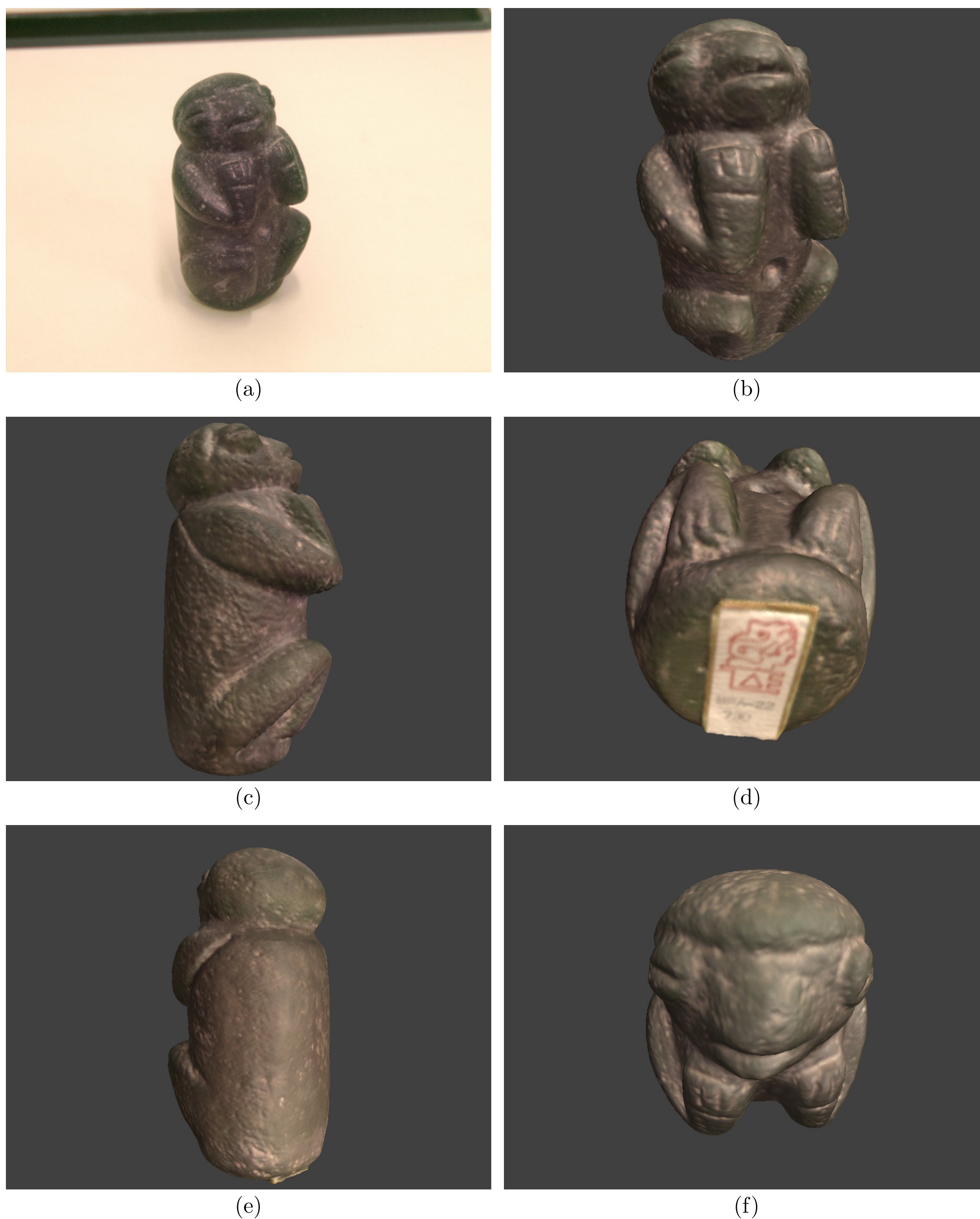


**Figura 13.6:** Estátua em madeira de São Francisco, do acervo do grupo IMAGO. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 306.716 faces, e foi reconstruído com *voxels* de 0,4mm.



(a) (b) (c) (d) (e) (f)

**Figura 13.7:** Réplica do objeto Alamito do Museu de La Plata, na Argentina, pertencente ao acervo da Professora Sílvia Schwab. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 342.520 faces, e foi reconstruído com *voxels* de 0,5mm.

**Figura 13.8:** Objeto Buddha, do repositório de modelos do laboratório SAMPL da Universidade de Ohio. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 445.306 faces, e foi reconstruído com *voxels* de 0,7mm. A qualidade dos dados deste objeto é muito inferior à qualidade dos outros objetos que foram reconstruídos, pois utilizou-se do *scanner* Vivid 700 da Konica Minolta. Todos os outros objetos que reconstruímos utilizaram-se do *scanner* Vivid 910, também da Konica Minolta. Além de possuírem muito mais ruído, as imagens do Vivid 700 possuem uma resolução de apenas 200 × 200 *pixels* (o Vivid 910 possui 640 × 480 *pixels*). Este objeto foi capturado utilizando-se apenas uma *turntable*, logo é impossível reconstruir corretamente o topo e a base da estátua.

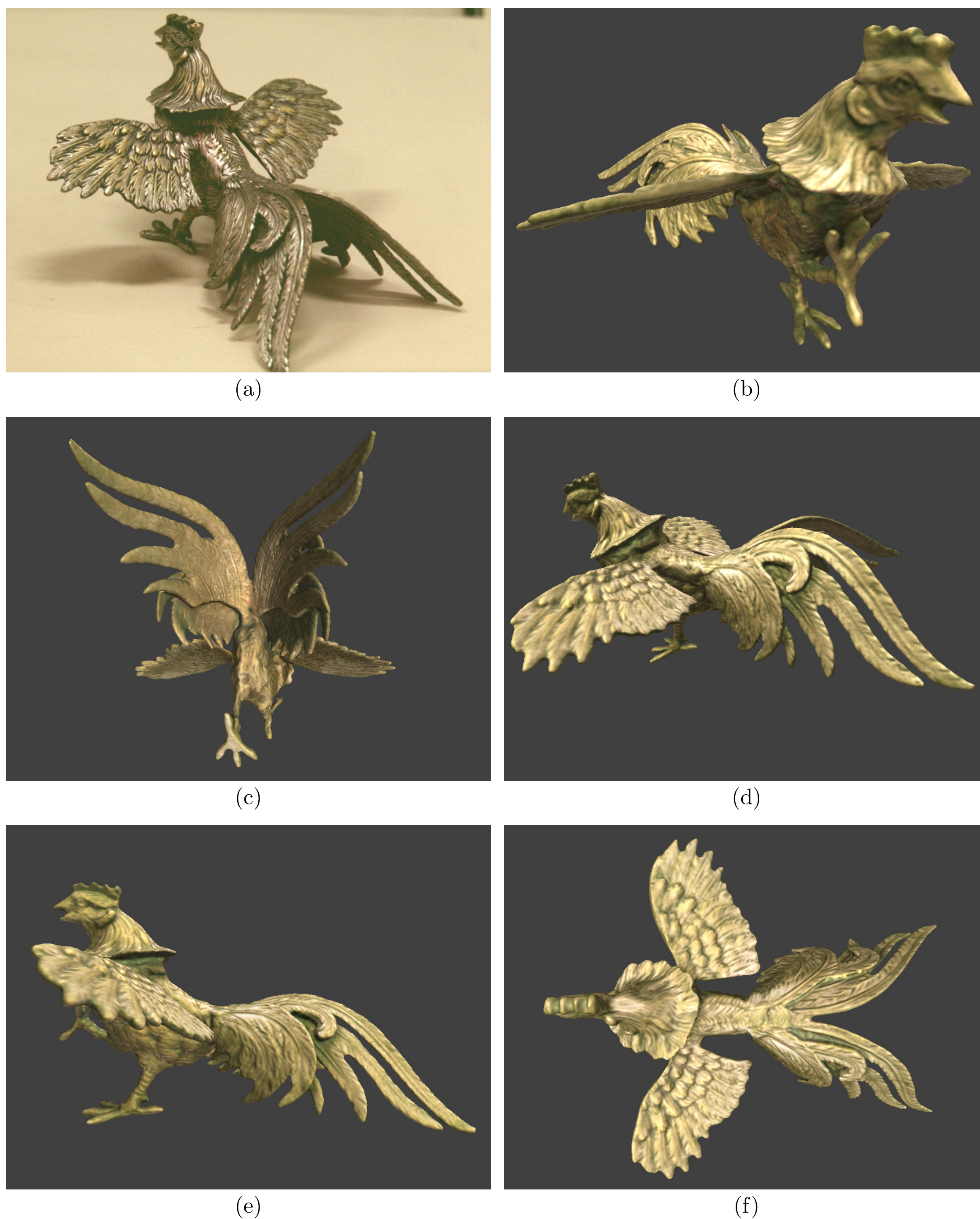


**Figura 13.9:** Estátua em latão de um galo, do acervo pessoal de Alexandre Vrubel. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 593.584 faces, e foi reconstruído com *voxels* de 0,5mm. Apesar de ser um material não cooperativo opticamente (devido ao alto grau de especularidade), a reconstrução geométrica não apresentou problemas. As cores das imagens individuais foram manualmente corrigidas, pois estavam muito erradas (vide figura 3.5).

**Figura 13.10:** Estátua criada pelo escultor paranaense Erbo Stenzel, do acervo do Museu Metropolitano de Arte de Curitiba (MUMA). (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 666.900 faces, e foi reconstruído com *voxels* de 0,8mm.

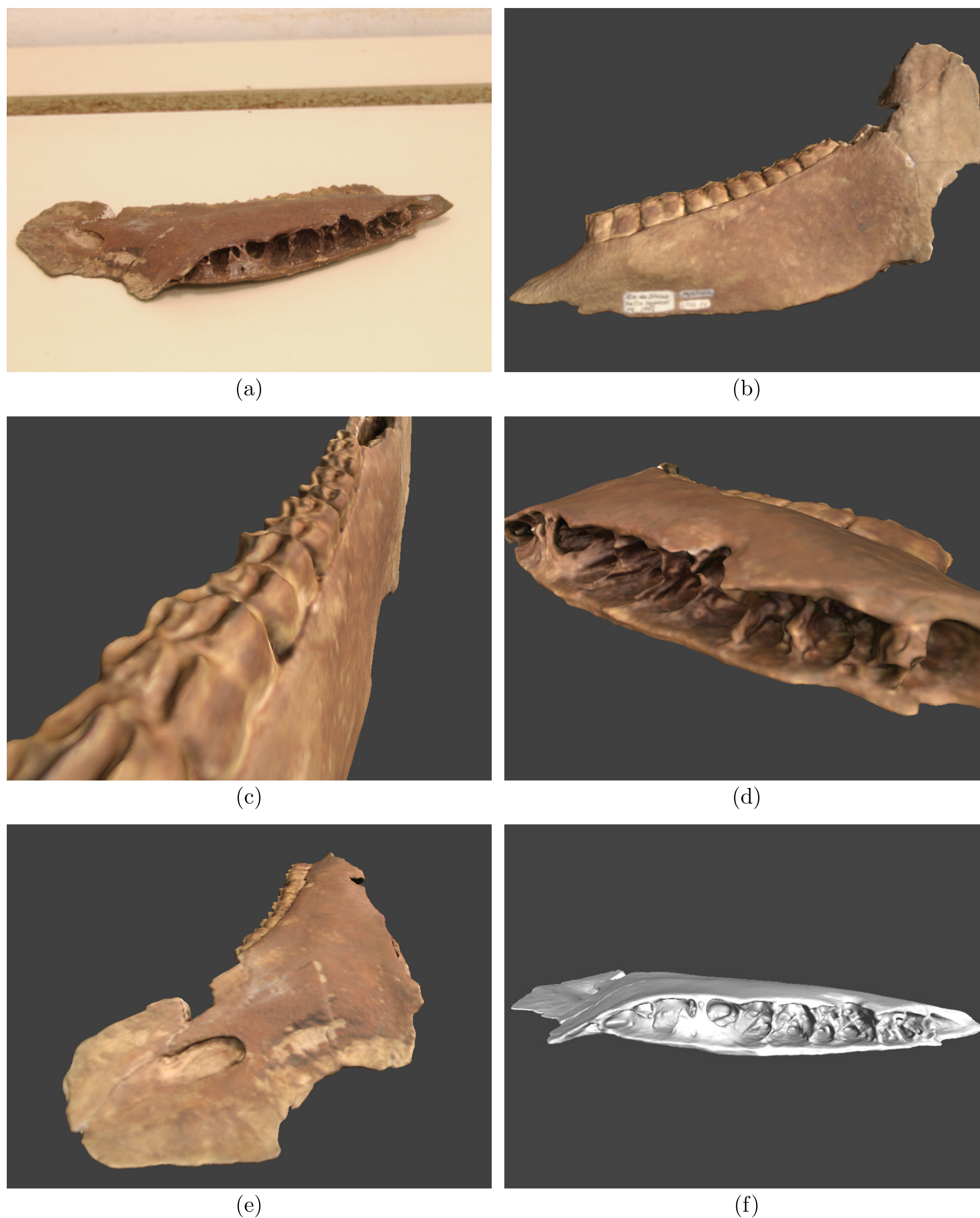


**Figura 13.11:** Fóssil da mandíbula de um cavalo nativo do continente americano, do acervo do Museu de História Natural da UFPR. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 675.632 faces, e foi reconstruído com *voxels* de 0,5mm. Na figura (f) renderizamos o modelo sem cores para mostrar a complexa geometria que foi reconstruída.

**Figura 13.12:** Estátua em bronze criada pelo artista Carybé, do acervo do Museu Metropolitano de Arte de Curitiba (MUMA). (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 714.696 faces, e foi reconstruído com *voxels* de 0,5mm. Na figura (f) renderizamos o modelo sem cores para mostrar como detalhes da superfície da estátua foram corretamente capturados e reconstruídos.

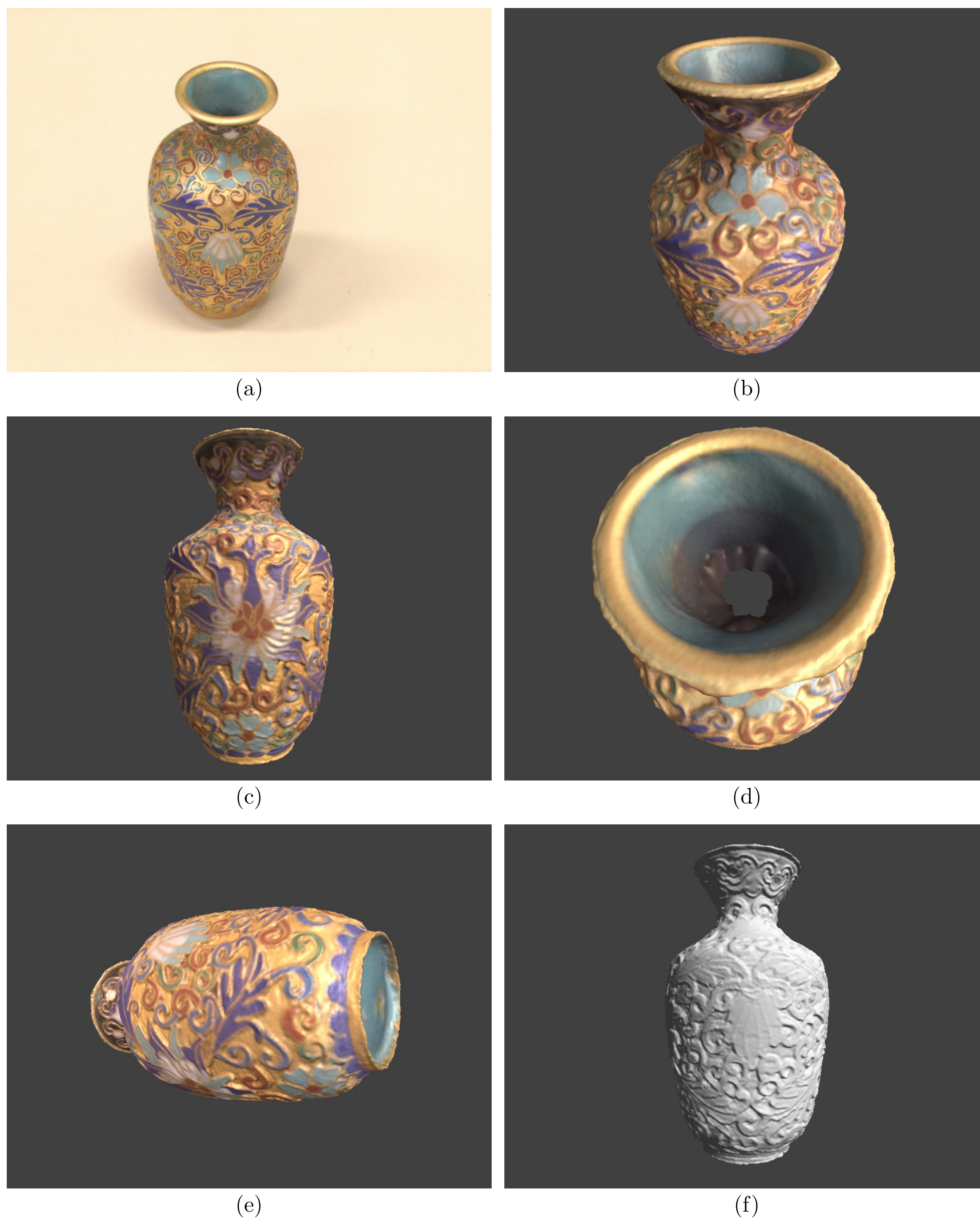
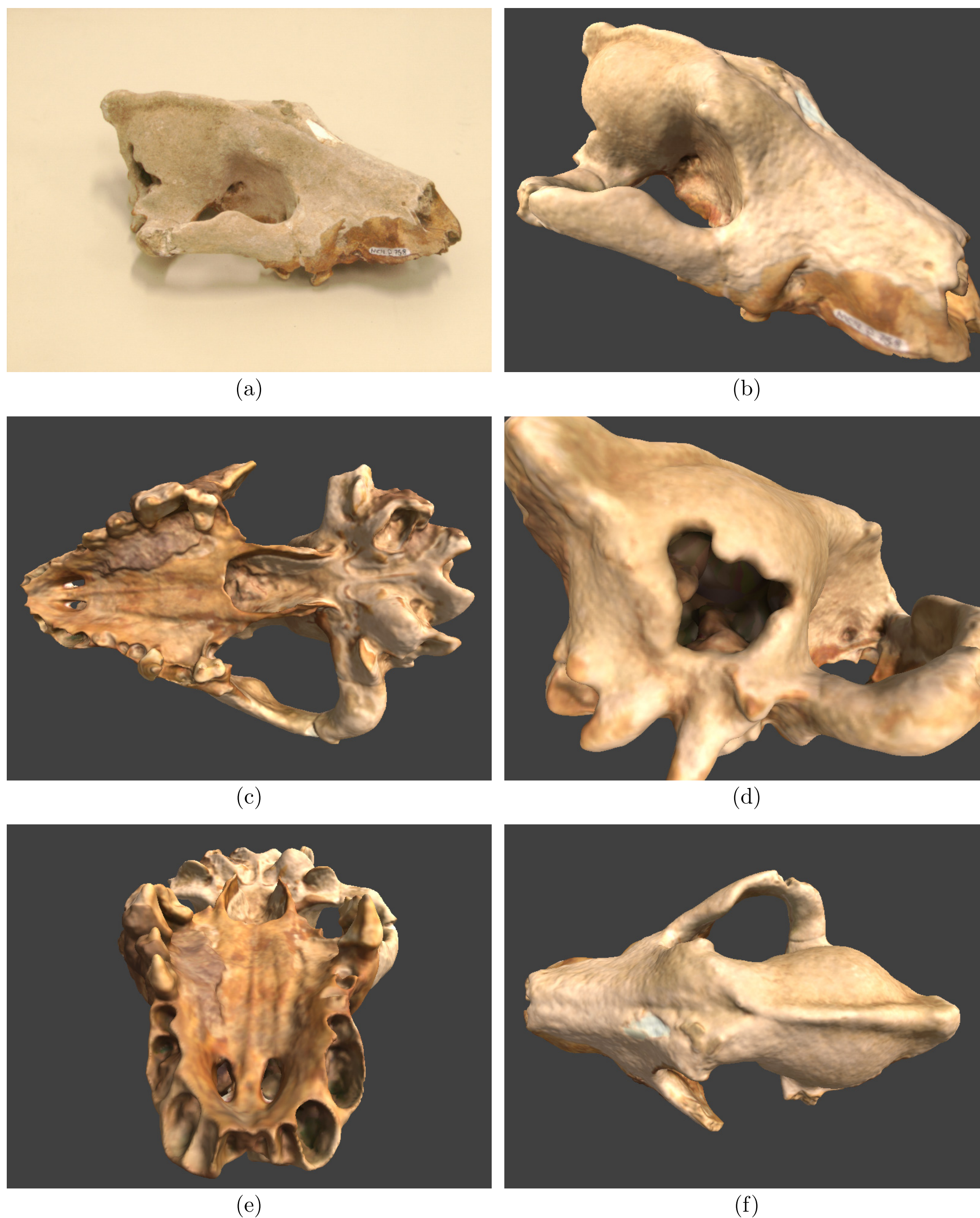


**Figura 13.13:** Pequeno vaso, do acervo do grupo IMAGO. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 792.809 faces, e foi reconstruído com *voxels* de 0,2mm. Na figura (d) podemos observar como o preenchimento de buracos falha, pois é impossível capturar dados do interior do vaso via *scanners* de triangulação *laser*. Na figura (f) renderizamos o modelo sem cores para mostrar como detalhes da superfície foram corretamente capturados e reconstruídos. Este objeto é particularmente difícil, pois possui alta especularidade (o que retornou dados ruins), e apenas os pequenos detalhes da superfície permitem o alinhamento das vistas.



(a) (b) (c) (d) (e) (f)

**Figura 13.14:** Fóssil do crânio de um *protocyon* (ancestral do lobo guará), do acervo do Museu de História Natural da UFPR. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 873.746 faces, e foi reconstruído com *voxels* de 0,5mm. Foi o objeto com mais oclusões e de topologia mais complexa que reconstruímos.

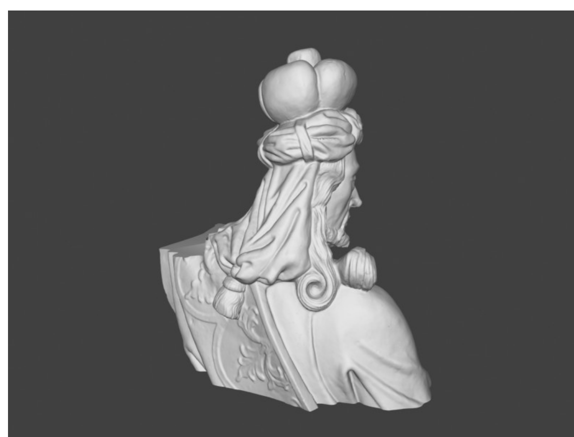
(a) (b) (c) (d) (e) (f)

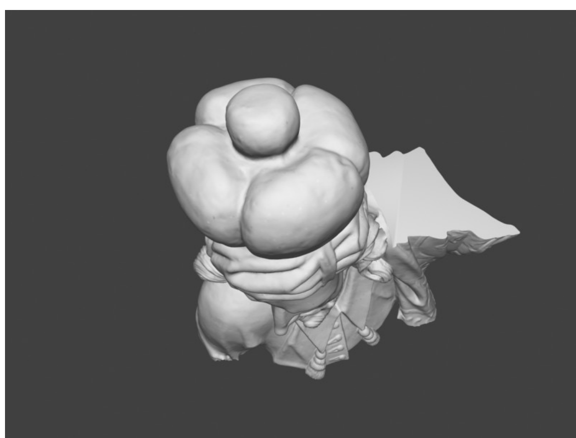
**Figura 13.15:** Estátua em mármore, presente no acervo do gabinete do Reitor da UFPR. (a) Imagem colorida original de uma das vistas capturadas; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 1.210.630 faces, e foi reconstruído com *voxels* de 1,0mm. Os dados adquiridos são extremamente ruins, devido à especularidade e translucência do mármore. Como nosso *pipeline* ainda não corrige a iluminação das imagens coloridas, as cores calculadas nem sempre são corretas. O objeto reconstruído não é perfeito, mas os defeitos apresentados são pequenos e em regiões localizadas, normalmente onde havia muita inter-reflexão do *laser*.

(a)

(b)

(c)


(d)


(e)

(f)

**Figura 13.16:** Réplica parcial em gesso de 30 anos do profeta Habacuc de Aleijadinho. (a) Imagem colorida original do objeto; (b) a (f) Renderizações do modelo 3D reconstruído. O modelo possui 1.409.939 faces, e foi reconstruído com *voxels* de 1,5mm. Como se trata de uma estátua em gesso, neste caso não renderizamos as vistas com as cores capturadas, que estavam muito incorretas. Esta réplica encontra-se no Museu de Artes Plásticas da UFMG.

# BIBLIOGRAFIA


[1] D. Aiger, N. J. Mitra, e D. Cohen-Or. 4-points congruent sets for robust pairwise surface registration. *ACM Trans. Graphics*, 27(3):1–10, 2008.

[2] N. Amenta, S. Choi, T. K. Dey, e N. Leekha. A simple algorithm for homeomorphic surface reconstruction. *Proceedings of the 16th Annual Symposium on Computational Geometry*, páginas 213–222, 2000.

[3] N. Amenta, S. Choi, e R. K. Kolluri. The power crust. *Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, páginas 249–266, 2001.

[4] N. Amenta, S. Choi, e R. K. Kolluri. The power crust, unions of balls, and the medial axis transform. *Computational Geometry: Theory and Applications*, 19(2-3):127–153, 2001.

[5] I. Ashdown. *Radiosity: A Programmer's Perspective*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[6] C. L. Bajaj, F. Bernardini, e G. Xu. Automatic reconstruction of surfaces and scalar fields from 3D scans. *Proceedings SIGGRAPH*, páginas 109–118, 1995.

[7] D. H. Ballard e C. M. Brown. *Computer Vision*. Prentice Hall, 1982.

[8] R. Baribeau, M. Rioux, e G. Godin. Color reflectance modeling using a polychromatic laser range sensor. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):263–269, 1992.

[9] R. Benjemaa e F. Schmitt. A solution for the registration of multiple 3D point sets using unit quaternions. *Proceedings of the European Conference on Computer Vision-Volume II*, páginas 34–50, 1998.

[10] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of ACM*, 18(9):509–517, 1975.

[11] R. Bergevin, M. Soucy, H. Gagnon, e D. Laurendeau. Towards a general multi-view registration technique. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(5):540–547, 1996.

[12] F. Bernardini, C. L. Bajaj, J. Chen, e D. Schikore. Automatic reconstruction of 3D CAD models from digital scans. *International Journal of Computational Geometry and Applications*, 9(4/5):327–369, 1999.

[13] F. Bernardini, I. Martin, e H. Rushmeier. High quality texture reconstruction. *IEEE Transactions on Visnalization and Computer Graphics*, 7:318–332, 2001.

[14] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, e G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, 1999.

[15] F. Bernardini e H. E. Rushmeier. The 3D model acquisition pipeline. *Computer Graphics Forum*, 21(2):149–172, 2002.

[16] F. Bernardini, H. E. Rushmeier, I. M. Martin, J. Mittleman, e G. Taubin. Building a digital model of Michelangelo's Florentine Pietà. *IEEE Computer Graphics and Applications*, 22(1):59–67, 2002.

[17] P. J. Besl e N. D. McKay. A method for registration of 3D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[18] G. Blais e M. D. Levine. Registering multiview range data to create 3D computer objects. Relatório Técnico TR-CIM-93-16, Centre for Intelligent Machines, McGill University, Montreal, Québec, Canada, 1993.

[19] OpenGL Architeture Review Board, D. Shreiner, M. Woo, J. Neider, e T. Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2.1 (6th Edition)*. Addison Wesley Professional, 2007.

[20] T. P. Breckon e R. B. Fisher. Amodal volume completion: 3D visual completion. *Computer Vision and Image Understanding*, 99(3):499–526, 2005.

[21] K. Brunnström e A. Stoddart. Genetic algorithms for free-form surface matching. *Proceedings of the International Conference on Pattern Recognition*, páginas 689–693, 1996.

[22] R. Scopigno C. Montani, R. Scateni. A modified look-up table for implicit disambiguation of marching cubes. *Visual Computer*, 10(6):353–355, 1994.

[23] K. Cain e P. Martinez. Multiple realities: Video projection in the tomb of Ramsses II. Proceedings of the 2nd Italy-United States Workshop, Rome, Italy, November 3-5, 2003.

[24] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, e T. R. Evans. Reconstruction and representation of 3D objects with radial basis functions. *Proceedings SIGGRAPH*, páginas 67–76, 2001.

[25] C. Chen, Y. Hung, e J. Cheng. Ransac-based darces: A new approach to fast automatic registration of partially overlapping range images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(11):1229–1234, 1999.

[26] Y. Chen e G. Medioni. Object modelling by registration of multiple range images. *Image and Vision Computing*, 10(3):145–155, 1992.

[27] Y. Chen e G. Medioni. Description of complex objects from multiple range images using an inflating balloon model. *Computer Vision and Image Understanding*, 61(3):325–334, 1995.

[28] E. V. Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Relatório Técnico CERN CN 95-17, CERN, 1995.

[29] C. Chow, H. Tsui, e T. Lee. Surface registration using a dynamic genetic algorithm. *Pattern Recognition*, 37:105–117, 2004.

[30] C. S. Chua e R. Jarvis. Point signatures: A new representation for 3D object recognition. *International Journal of Computer Vision*, 25(1):63–85, 1997.

[31] D. Chung, I. Yun, e S. Lee. Registration of multiple range views using the reverse calibration technique. *Pattern Recognition*, 31(4):457–464, 1998.

[32] P. Cignoni, C. Montani, e R. Scopigno. A comparison of mesh simplification algorithms. *Computers and Graphics*, 22(1):37–54, 1998.

[33] J. Cohen, M. Olano, e D. Manocha. Appearance preserving simplification. *Proceedings ACM SIGGRAPH*, volume 32, páginas 115–122, 1998.

[34] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, e W. Wright. Simplification envelopes. *Proceedings ACM SIGGRAPH*, páginas 119–128, 1996.

[35] E. Corwin e A. Logar. Sorting in linear time - variations on the bucket sort. *Journal of Computing Sciences in Colleges*, 20(1):197–202, 2004.

[36] B. Curless e M. Levoy. Better optical triangulation through spacetime analysis. *Proceedings ICCV*, páginas 987–994, 1995.

[37] B. Curless e M. Levoy. A volumetric method for building complex models from range images. *Proceedings SIGGRAPH*, páginas 303–312, 1996.

[38] J. Davis, S. R. Marschner, M. Garr, e M. Levoy. Filling holes in complex surfaces using volumetric diffusion. *Proceedings of the First International Symposium on 3D Data Processing, Visualization, and Transmission*, páginas 428, 2002.

[39] H. Delingette, M. Hebert, e K. Ikeuchi. Shape representation and image segmentation using deformable surfaces. *Image and Vision Computing*, 10(3):132–144, 1992.

[40] T. K. Dey, J. Giesen, e J. Hudson. Delaunay based shape reconstruction from large data. *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, páginas 19–27, 2001.

[41] H. Q. Dinh, G. Turk, e G. Slabaugh. Reconstructing surfaces using anisotropic basis functions. *Proceedings ICCV*, 2:606–613, 2001.

[42] F. Duguet, C. H. Esteban, G. Drettakis, e F. Schmitt. Level of detail continuum for huge geometric data. Relatório Técnico 5552, INRIA - Institut National de Recherche en Informatique et en Automatique, 2005.

[43] D. H. Eberly. *3D Game Engine Design: A Practical Approach to Real-time Computer Graphics*. Morgan Kaufmann Publishers Inc., 2000.

[44] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, e W. Stuetzle. Multiresolution analysis of arbitrary meshes. *Proceedings ACM SIGGRAPH*, páginas 173–182, 1995.

[45] H. Edelsbrunner. Shape reconstruction with delaunay complex. *Proceedings of the Third Latin American Symposium on Theoretical Informatics*, páginas 119–132, 1998.

[46] H. Edelsbrunner e E. P. Mücke. Three-dimensional alpha shapes. *ACM Transactions on Graphics*, 13(1):43–72, 1994.

[47] D. W. Eggert, A. W. Fitzgibbon, e R. B. Fisher. Simultaneous registration of multiple range views for use in reverse engineering. Relatório Técnico 804, Dept. of Artificial Intelligence, University of Edinburgh, 1996.

[48] J. Feldmar e N. Ayache. Rigid, affine and locally affine registration of free-form surfaces. *International Journal of Computer Vision*, 18(2):99–119, 1996.

[49] R. Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[50] S. Fleishman, D. Cohen-Or, e C. T. Silva. Robust moving least-squares fitting with sharp features. *ACM Trans. on Graphics*, 24(3):544–552, 2005.

[51] M. S. Floater. Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14(3):231–250, 1997.

[52] M. S. Floater e K. Hormann. Surface parameterization: A tutorial and survey. *Advances in Multiresolution For Geometric Modelling*, páginas 157–186. Springer Verlag, 2005.

[53] J. D. Foley, A. van Dam, S. K. Feiner, e J. F. Hughes. *Computer Graphics (2nd ed. in C): Principles and Practice.* Addison-Wesley Publishing, Boston, MA, USA, 1996.

[54] R. Fontana, M. Greco, M. Materazzi, E. Pampaloni, L. Pezzati, C. Rocchini, e R. Scopigno. Three-dimensional modelling of statues: The Minerva of Arezzo. *Journal of Cultural Heritage*, 3(4):325–331, 2002.

[55] H. Gagnon, M. Soucy, R. Bergevin, e D. Laurendeau. Registration of multiple range views for automatic 3-D model building. *Computer Vision and Pattern Recognition*, páginas 581–586, 1994.

[56] M. Garland e P. Heckbert. Simplification using quadric error metrics. *Proceedings ACM SIGGRAPH*, volume 31, páginas 209–216, 1997.

[57] N. Gelfand, S. Rusinkiewicz, L. Ikemoto, e M. Levoy. Geometrically stable sampling for the icp algorithm. *Proceedings 3DIM*, páginas 260–267. IEEE Computer Society, 2003.

[58] G. Godin, F. Blais, L. Cournoyer, J. A. Beraldin, J. Domey, J. Taylor, M. Rioux, e S. El-Hakim. Laser range imaging in archaeology: Issues and results. *Proceedings Workshop on Aplications of Computer Vision to Archaeology*, 01:11–18, 2003.

[59] G. Godin, D. Laurendeau, e R. Bergevin. A method for the registration of attributed range images. *Third International Conference on 3-D Digital Imaging and Modeling*, 0:179–186, 2001.

[60] J. Gomes e O. D. Faugeras. Level sets and distance functions. *Proceedings of the European Conference on Computer Vision*, páginas 588–602, 2000.

[61] M. Gopi, S. Krishnan, e C. T. Silva. Surface reconstruction based on lower dimensional localized delaunay triangulation. *Computer Graphics Forum (Eurographics 2000)*, volume 19(3), páginas 467–478, 2000.

[62] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[63] M. Greenspan e G. Godin. A nearest neighbor method for efficient ICP. *Third International Conference on 3-D Digital Imaging and Modeling*, 0:161–168, 2001.

[64] X. Gu, S. J. Gortler, e H. Hoppe. Geometry images. *Proceedings SIGGRAPH*, páginas 355–361. ACM Press, 2002.

[65] P. Heckbert e M. Garland. Survey of polygonal surface simplification algorithms. *SIGGRAPH 97 Course Notes*, (25), 1997.

[66] A. Hilton, A. J. Stoddart, J. Illingworth, e T. Windeatt. Reliable surface reconstructiuon from multiple range images. *Proceedings of the European Conference on Computer Vision*, páginas 117–126, 1996.

[67] K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, e T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. *Proceedings SIGGRAPH*, páginas 277–286, 1999.

[68] H. Hoppe. Progressive meshes. *Proceedings SIGGRAPH*, páginas 99–108, New York, NY, USA, 1996. ACM.

[69] H. Hoppe. Efficient implementation of progressive meshes. *Computers and Graphics*, 22(1):27–36, 1998.

[70] H. Hoppe. New quadric metric for simplifying meshes with appearance attributes. *Proceedins IEEE Visualization*, páginas 59–66, 1999.

[71] K. Hormann, B. Levy, e A. Sheffer. Mesh parameterization: Theory and practice. *ACM SIGGRAPH Course Notes*, 2007.

[72] B. K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of Optical Society of America A*, 4(4):1127–1135, 1987.

[73] A. Hornung e L. Kobbelt. Robust reconstruction of watertight 3D models from non-uniformly sampled point clouds without normal information. *Proc. Eurographics SGP*, páginas 41–50, 2006.

[74] P. V. C. Hough. Method and means for recognizing complex patterns. *US Patent*, 1962.

[75] K. Ikeuchi, T. Oishi, J. Takamatsu, R. Sagawa, A. Nakazawa, R. Kurazume, K. Nishino, M. Kamakura, e Y. Okamoto. The Great Buddha project: Digitally archiving, restoring, and analyzing cultural heritage objects. *International Journal of Computer Vision*, 75(1):189–208, 2007.

[76] K. Ikeuchi e K. Sato. Determining reflectance properties of an object using range and brightness images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(11):1139–1153, 1991.

[77] T. Jirka e V. Skala. Gradient vector estimation and vertex normal computation. Relatório Técnico DCSE/TR-2002-08, University of West Bohemia in Pilsen, 2002.

[78] A. Johnson. *Spin-Images: A Representation for 3D Surface Matching.* Tese de Doutorado, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[79] A. Johnson e S. Kang. Registration and integration of textured 3D data. *Proceedings of the International Conference on Recent Advances in 3D Digital Imaging and Modeling*, páginas 234–241, 1997.

[80] T. Jost e H. Hügli. A multi-resolution scheme ICP algorithm for fast shape registration. *First International Symposium on 3D Data Processing Visualization and Transmission*, 0:540–543, 2002.

[81] G. Kay e T. Caelli. Inverting an illumination model from range and intensity maps. *CVGIP: Image Understanding*, 59(2):183–201, 1994.

[82] M. Kazhdan, M. Bolitho, e H. Hoppe. Poisson surface reconstruction. *Proc. Eurographics SGP*, páginas 61–70, 2006.

[83] S. Kim, C. Jho, e H. Hong. Automatic registration of 3D data sets from unknown viewpoints. *Workshop on Frontiers of Computer Vision*, páginas 155–159, 2003.

[84] R. Kolluri, J. R. Shewchuk, e J. F. O'Brien. Spectral surface reconstruction from noisy point clouds. *Proc. Eurographics SGP*, páginas 11–21, 2004.

[85] Konica-Minolta. *Konica Minolta Vivid 910 SDK Documentation.*

[86] S. Krishnan, P. Y. Lee, J. B. Moore, e S. Venkatasubramanian. Global registration of multiple 3D point sets via optimization-on-a-manifold. *Proceedings of the Symposium on Geometry Processing*, páginas 187, 2005.

[87] H. P. A. Lensch, W. Heidrich, e H.-P. Seidel. Automated texture registration and stitching for real world models. *Proceedings of the IEEE 8th Pacific Conference on Computer Graphics and Applications*, páginas 317–326, 2000.

[88] H. P. A. Lensch, J. Kautz, M. Goesele, W. Heidrich, e H.-P. Seidel. Image-based reconstruction of spatially varying materials. *Proceedings of the 12th Eurographics Rendering Workshop*, páginas 103–114, 2001.

[89] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, e D. Fulk. The Digital Michelangelo project: 3D scanning of large statues. *Proceedings SIGGRAPH*, páginas 131–144, 2000.

[90] T. Lewiner, H. Lopes, A. W. Vieira, e G. Tavares. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of Graphics Tools*, 8(2):1–15, 2003.

[91] P. Lindstrom e G. Turk. Image-driven simplification. *ACM Transactions on Graphics*, 19(3):204–241, 2000.

[92] Z. Liyan, L. Shenglan, W. Xi, e Z. Laishui. Segmentation and parametrization of arbitrary polygon meshes. *Proceedings of the Geometric Modeling and Processing Conference*, páginas 143, 2004.

[93] W. E. Lorensen e H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Proceedings SIGGRAPH*, páginas 163–169, New York, NY, USA, 1987. ACM.

[94] D. P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Computer Graphics and Applications*, 21(3):24–35, 2001.

[95] B. Lévy, S. Petitjean, N. Ray, e J. Maillot. Least squares conformal maps for automatic texture atlas generation. *Proceedings SIGGRAPH*, 2002.

[96] T. Masuda. Generation of geometric model by registration and integration of multiple range images. *International Conference on 3-D Digital Imaging and Modeling*, 0:254–261, 2001.

[97] T. Masuda. Object shape modelling from multiple range images by matching signed distance fields. *International Symposium on 3D Data Processing Visualization and Transmission*, 0:439–448, 2002.

[98] T. Masuda, K. Sakaue, e N. Yokoya. Registration and integration of multiple range images for 3-D model construction. *Proceedings of the International Conference on Pattern Recognition*, volume 1, páginas 879–883, 1996.

[99] Y. Matsumoto, H. Terasaki, K. Sugimoto, e T. Arakawa. A portable three-dimensional digitizer. *Proceedings of the International Conference on Recent Advances in 3D Digital Imaging and Modeling*, páginas 197–204, 1997.

[100] D. J. R. Meagher. *The Octree Encoding Method for Efficient Solid Modeling.* Tese de Doutorado, Rensselaer Polytechnic Institute, 1980.

[101] D. Miyazaki, T. Oishi, T. Nishikawa, R. Sagawa, K. Nishino, T. Tomomatsu, Y. Takase, e K. Ikeuchi. The Great Buddha project: Modelling cultural heritage through observation. *Proceedings Sixth International Conference on Virtual Systems and MultiMedia*, páginas 138–145, 2000.

[102] P. Mullen, Y. Tong, P. Alliez, e M. Desbrun. Spectral conformal parametrization. *Proc. Eurographics SGP*, volume 27, 2008.

[103] P. J. Neugebauer. Geometrical cloning of 3D objects via simultaneous registration of multiple range images. *Proceedings of the International Conference on Shape Modeling and Applications*, páginas 130–139, 1997.

[104] P. J. Neugebauer e K. Klein. Texturing 3D models of real world objects from multiple unregistered photographics views. *Computer Graphics Forum*, volume 18, páginas 245–256, 1999.

[105] G. M. Nielson e B. Hamann. The asymptotic decider: Resolving the ambiguity in marching cubes. *Proceedings of the IEEE 2nd conference on Visualization*, páginas 83–91, 1991.

[106] Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, e H. P. Seidel. Multi-level partition of unity implicits. *ACM Transactions on Graphics*, 22(3):463–470, 2003.

[107] J. Park e A. C. Kak. 3D modeling of optically challenging objects. *IEEE Transactions on Visualization and Computer Graphics*, 14(2):246–262, 2008.

[108] M. Pauly, N. J. Mitra, J. Giesen, M. Gross, e L. J. Guibas. Example-based 3D scan completion. *Proceedings of the third Eurographics Symposium on Geometry Processing*, páginas 23–32, 2005.

[109] G. Pavlidis, A. Koutsoudis, F. Arnaoutoglou, V. Tsioukas, e C. Chamzas. Methods for 3D digitization of cultural heritage. *Journal of Cultural Heritage*, 8(1):93–98, 2007.

[110] A. Pentland e S. Sclaroff. Closed-form solutions for physically based shape modeling and recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(7):715–729, 1991.

[111] M. Pharr e R. Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[112] W. H. Press, B. P. Flannery, S. A. Teukolsky, e W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1988.

[113] K. Pulli. Multiview registration for large data sets. *International Conference on 3D Digital Imaging and Modeling*, páginas 160–168, 1999.

[114] K. Pulli, M. Cohen, T. Duchamp, H. Hoppe, L. Shapiro, e W. Stuetzle. View-based rendering: Visualizing real objects from scanned range and color data. *Proceedings of the 8th Eurographics Workshop on Rendering*, páginas 23–34, 1997.

[115] E. Puppo e R. Scopigno. Simplification, LOD, and multiresolution - principles and applications. Relatório Técnico C97-12, National Research Council of Italy, Pisa, 1997.

[116] C. Rocchini, P. Cignoni, F. Ganovelli, C. Montani, P. Pingi, e R. Scopigno. The marching intersections algorithm for merging range images. *Visual Computer: International Journal of Computer Graphics*, 20(2-3):149–164, 2004.

[117] C. Rocchini, P. Cignoni, C. Montani, e R. Scopigno. Multiple textures stitching and blending on 3D objects. *Proceedings of the 10th Eurographics Workshop on Rendering*, páginas 127–138, 1999.

[118] J. Rossignac e P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. *Geometric Modeling in Computer Graphics*, páginas 455–465, 1993.

[119] H. E. Rushmeier e F. Bernardini. Computing consistent normals and colors from photometric data. *3D Digital Imaging and Modeling*, 0:99–108, 1999.

[120] H. E. Rushmeier, F. Bernardini, J. Mittleman, e G. Taubin. Acquiring input for rendering at appropriate level of detail: Digitizing a pietà. *Proceedings of the 9th Eurographics Workshop on Rendering*, páginas 81–92, 1998.

[121] S. Rusinkiewicz, O. Hall-Holt, e M. Levoy. Real-time 3D model acquisition. *ACM Trans. on Graphics*, 21(3):438–446, 2002.

[122] S. Rusinkiewicz e M. Levoy. Efficient variants of the ICP algorithm. *Proceedings 3DIM*, páginas 145–152, 2001.

[123] R. Sagawa e K. Ikeuchi. Taking consensus of signed distance field for complementing unobservable surface. *Proceedings 3DIM*, páginas 410–417, 2003.

[124] R. Sagawa e K. Ikeuchi. Hole filling of a 3D model by flipping signs of a signed distance field in adaptive resolution. *IEEE TPAMI*, 30(4):686–699, 2008.

[125] R. Sagawa, K. Nishino, e K. Ikeuchi. Robust and adaptive integration of multiple range images with photometric attributes. *Proceedings CVPR*, páginas II:172–179, 2001.

[126] R. Sagawa, K. Nishino, e K. Ikeuchi. Adaptively merging large-scale range data with reflectance properties. *IEEE TPAMI*, 27(3):392–405, 2005.

[127] J. Salvi, C. Matabosch, D. Fofi, e J. Forest. A review of recent range image registration methods with accuracy evaluation. *Image and Vision Computing*, 25(5):578–596, 2007.

[128] P. V. Sander, S. J. Gortler, J. Snyder, e H. Hoppe. Signal-specialized parametrization. *In Eurographics Workshop on Rendering*, 2002.

[129] P. V. Sander, J. Snyder, S. J. Gortler, e H. Hoppe. Texture mapping progressive meshes. *Proceedings SIGGRAPH*, páginas 409–416, 2001.

[130] Y. Sato, M. Wheeler, e K. Ikeuchi. Object shape and reflectance modeling from observation. *Proceedings SIGGRAPH*, páginas 379–387, 1997.

[131] W. J. Schroeder, J. A. Zarge, e W. E. Lorensen. Decimation of triangle meshes. *SIGGRAPH Computer Graphics*, 26(2):65–70, 1992.

[132] A. Sharf, T. Lewiner, A. Shamir, L. Kobbelt, e D. Cohen-Or. Competing fronts for coarse-to-fine surface reconstruction. *Computer Graphics Forum*, 25(3):389–398, 2006.

[133] A. Sharf, T. Lewiner, G. Shklarski, S. Toledo, e D. Cohen-Or. Interactive topology-aware surface reconstruction. *ACM Trans. on Graphics*, 26(3):43, 2007.

[134] G. C. Sharp, S. W. Lee, e D. K. Wehe. ICP registration using invariant features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(1):90–102, 2002.

[135] G. C. Sharp, S. W. Lee, e D. K. Wehe. Multiview registration of 3D scenes by minimizing error between coordinate frames. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(8):1037–1050, 2004.

[136] A. Sheffer e J. Hart. Seamster: Inconspicuous low-distortion texture seam layout. *Proceedings IEEE Visualization*, 2(4), 2002.

[137] A. Sheffer e E. Sturler. Parameterization of faceted surfaces for meshing using angle based flattening. *Engineering with Computers*, 17(3):326–337, 2001.

[138] L. Silva, O. R. P. Bellon, e K. L. Boyer. *Robust Range Image Registration.* World Scientific Series on Machine Perception and Artificial Intelligence, World Scientific Publishing, 2005.

[139] D. Simon. *Fast and Accurate Shape-Based Registration.* Tese de Doutorado, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

[140] M. Soucy, G. Godin, e M. Rioux. A texturemapping approach for the compression of colored 3D triangulations. *The Visual Computer*, 12(10):503–513, 1996.

[141] M. Soucy e D. Laurendeau. A general surface approach to the integration of a set of range views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(4):344–358, 1995.

[142] I. Stamos e M. Leordeanu. Automated feature-based range registration of urban scenes of large scale. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 02:555–561, 2003.

[143] A. J. Stoddart e A. Hilton. Registration of multiple point sets. *Proceedings of the 13th IAPR International Conference on Pattern Recognition*, páginas 40–44, 1996.

[144] J. Tarel, H. Civi, e D. B. Cooper. Pose estimation of free-form 3D objects without point matching using algebraic surface models. *Proceedings of IEEE Workshop on Model-Based 3D Image Analysis*, páginas 13–21, 1998.

[145] D. Terzopoulos, A. Witkin, e M. Kass. Constraints on deformable models: Recovering 3D shape and non-rigid motion. *Artificial Intelligence*, 36(1):91–123, 1988.

[146] G. Tewari, J. Snyder, P. V. Sander, S. J. Gortler, e H. Hoppe. Signal-specialized parameterization for piecewise linear reconstruction. *Proceedings of the ACM 2004 Eurographics*, páginas 55–64, 2004.

[147] G. Thürmer e C. A. Wüthrich. Computing vertex normals from polygonal facets. *Journal of Graphics Tools*, 3(1):43–46, 1998.

[148] P. Torr e A. Zisserman. MLESAC: A new robust estimator with application to estimating image geometry. *Computer Vision and Image Understanding*, 78(1):138–156, 2000.

[149] K. Torrance e E. Sparrow. Theory for off-specular reflection for roughened surface. *Journal of the Optical Society of America*, 57:1105–1114, 1967.

[150] E. Trucco, A. Fusiello, e V. Roberto. Robust motion and correspondence of noisy 3-D point sets with missing data. *Pattern Recognition Letters*, 20(9):889–898, 1999.

[151] R. Y. Tsai. A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf tv cameras and lenses. *International Journal of Robotics and Automation*, RA-3:323–344, 1987.

[152] G. Turk e M. Levoy. Zippered polygon meshes from range images. *Proceedings SIGGRAPH*, páginas 311–318, 1994.

[153] T. Vieira, A. Peixoto, L. Velho, e T. Lewiner. An iterative framework for registration with reconstruction. *Proceedings VMV*, páginas 101–108, 2007.

[154] C. C. L. Wang, Y. Wang, Tang K., e M. M. F. Yuen. Reduce the stretch in surface flattening by finding cutting paths to the surface boundary. *Computer-Aided Design*, 36:665–677, 2004.

[155] M. D. Wheeler, Y. Sato, e K. Ikeuchi. Consensus surfaces for modeling 3D objects from multiple range images. *International Conference on Computer Vision*, páginas 917–924, 1998.

[156] R. T. Whitaker. A level-set approach to 3D reconstruction from range data. *International Journal of Computer Vision*, 29(3):203–231, 1998.

[157] R. J. Woodham. Photometric method for determining surface orientation from multiple images. *Optical Engineering*, 19:139–144, 1980.

[158] X. Wu. A linear-time simple bounding volume algorithm. *Graphics Gems III*, páginas 301–306, 1992.

[159] J. V. Wyngaerd e L. V. Gool. Automatic crude patch registration: Toward automatic 3D model building. *Computer Vision and Image Understanding*, 87(1-3):8–26, 2002.

[160] N. Yastikli. Documentation of cultural heritage using digital photogrammetry and laser scanning. *Journal of Cultural Heritage*, 8(4):423–427, 2007.

[161] Y. Yemez e C. J. Wetherilt. A volumetric fusion technique for surface reconstruction from silhouettes and range data. *CVIU*, 105(1):30–41, 2007.

[162] H. K. Zhao, S. Osher, e R. Fedkiw. Fast surface reconstruction using the level set method. *Proceedings of the IEEE Workshop on Variational and Level Set Methods*, páginas 194–201, 2001.

[163] H. Zhou e Y. Liu. Accurate integration of multi-view range images using k-means clustering. *Pattern Recognition*, 41(1):152–175, 2008.

[164] T. Zinsser, H. Schnidt, e J. Niermann. A refined ICP algorithm for robust 3-D correspondences estimation. *International Conference on Image Processing*, páginas 695–698, 2003.