

Universidade Federal de Santa Catarina
Centro de Blumenau
Departamento de Engenharia de
Controle e Automação e Computação



Juliano Emir Nunes Masson

Desenvolvimento de um algoritmo para a reconstrução 3D a
partir de imagens RGB

Blumenau
2018

Juliano Emir Nunes Masson

**Desenvolvimento de um algoritmo para a
reconstrução 3D a partir de imagens RGB**

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como parte dos requisitos necessários para a obtenção do Título de Engenheiro de Controle e Automação.
Orientador: Prof. Dr. Marcelo Roberto Petry

Universidade Federal de Santa Catarina
Centro de Blumenau
Departamento de Engenharia de
Controle e Automação e Computação

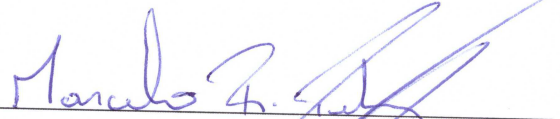
Blumenau
2018

Juliano Emir Nunes Masson

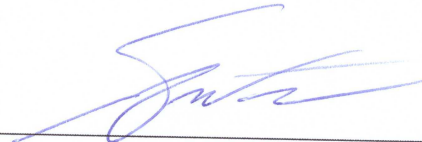
Desenvolvimento de um algoritmo para a reconstrução 3D a partir de imagens RGB

Trabalho de Conclusão de Curso apresentado à Universidade Federal de Santa Catarina como requisito parcial para a obtenção do título de Engenheiro de Controle e Automação.

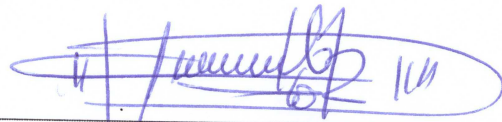
Comissão Examinadora



Prof. Dr. Marcelo Roberto Petry
Universidade Federal de Santa Catarina
Orientador



Prof. Dr. Marcos Vinicius Matsuo
Universidade Federal de Santa Catarina



Prof. Dr. Leonardo Mejia Rincon
Universidade Federal de Santa Catarina

Blumenau, 31 de janeiro de 2019

Dedico este trabalho a todos aqueles que, de alguma forma,
auxiliaram para a concretização desta etapa.

Agradecimentos

Agradeço a minha família pelo apoio durante a graduação. Aos amigos que fizeram destes 5 anos uma jornada de aprendizados e risadas. Aos professores que apesar das dificuldades de um campus em formação não mediram esforços para transmitir seus conhecimentos. Ao professor Dr. Marcelo Roberto Petry, pelo apoio e conselhos durante os anos que trabalhamos juntos.

"Train yourself to let go of everything you fear to lose."
(Yoda)

Resumo

A fotogrametria, reconstrução 3D a partir de imagens 2D, surgiu no século 19 com Laussedat, que criou um mapa de Paris baseado em informações geométricas de fotografias capturadas do topo dos telhados. Apesar de já ter seus princípios definidos a muito tempo, a fotogrametria como é conhecida atualmente não existia. Apenas após o desenvolvimento dos computadores e das câmeras digitais que essa técnica pôde ser utilizada em mais aplicações. Atualmente ela é utilizada em diversas áreas, como no entretenimento, construção civil, inspeção em áreas de risco, etc.

Ela começou a se tornar um tópico mais comum ao público menos especializado com a popularização dos drones, que facilitou a aquisição de fotografias aéreas e permitiu que essa técnica pudesse ser utilizada para a inspeção de grandes áreas, com aplicações principalmente na área da engenharia civil. A popularização das impressoras 3D também aumentou o número de usuários dessa tecnologia, pois permite que objetos de difícil modelagem possam ser digitalizados através de fotos e posteriormente impressos.

Esse trabalho traz um comparativo entre *softwares* e bibliotecas que implementam partes do processo de fotogrametria, além de desenvolver algoritmos que implementam alguns dos métodos descritos na bibliografia. Os resultados obtidos demonstram que com a correta combinação de algoritmos pode-se chegar a reconstruções 3D muito parecidas com as cenas capturadas pelas imagens.

Os resultados foram divididos em 3 categorias: a categoria A com os algoritmos desenvolvidos neste trabalho; a categoria B com os *softwares*/bibliotecas livres; e a categoria C com os *softwares* pagos. A principal diferença entre os resultados das categorias foi o tempo e o consumo de recursos de *hardware*. Em uma comparação qualitativa, os algoritmos desenvolvidos neste trabalho conseguiram um resultado próximo aos do *software* pago, apenas apresentando mais ruídos. O melhor resultado entre todas as categorias foi da combinação do VisualSFM + MeshRecon + TexRecon, que apresentou o maior nível de detalhes e o menor tempo.

Palavras-Chave: 1. Reconstrução 3D. 2. Fotogrametria. 3. *Structure from motion*. 4.

Nuvem de pontos 3D 5. Superfícies 3D

Abstract

The photogrammetry, 3D reconstruction from 2D images, emerged in the 19th century with Laussedat, he constructed the map of Paris based on geometric information extracted from his photographs taken from rooftops in the city. Although its principles have already been defined for a long time, the photogrammetry as it knows nowadays did not exist, just after the development of computers and digital cameras this technique started to be used in more applications. Currently it is used in several areas, such as entertainment, construction, inspection in risk areas, etc.

It began to become a more common topic to the less specialized audience with the popularization of drones, which facilitated the acquisition of aerial photographs and allowed this technique to be used for inspection of large areas with applications mainly in the area of civil engineering. The popularization of 3D printers also increased the number of users of this technology as it allows objects of difficult modeling to be scanned through photos and later printed.

This work brings a comparison between softwares and libraries that implement parts of the photogrammetry process, in addition to developing algorithms that implement some of the methods described in the bibliography. The results obtained show that with the right combination of algorithms, 3D reconstructions can be very similar to the scenes captured by the images.

The results were divided into 3 categories: category A with the algorithms developed in this work; category B with free softwares/libraries; and category C with paid softwares. The main difference between the results of the categories was the time and the consumption of hardware resources. In a qualitative comparison, the algorithms developed in this work achieved a result close to the paid software, only presenting more noise. The best result among all categories was the combination of VisualSFM + MeshRecon + TexRecon, which presented the highest level of details and the shortest time.

Keywords: 1. 3D Reconstruction. 2. Photogrammetry. 3. Structure from motion. 4. Point cloud. 5. 3D Surfaces

Lista de figuras

Figura 1 – Etapas da reconstrução 3D a partir de imagens 2D.	18
Figura 2 – Modelo de projeção central.	22
Figura 3 – Modelo de projeção central mostrando o plano da imagem com os pixels discretos.	24
Figura 4 – Exemplos do uso da homografia [1]. (a) Realidade aumentada. (b) Construção de panorama.	25
Figura 5 – Exemplos da geometria epipolar (Adaptado de [2]).(a) Plano epipolar π . (b) Linha epipolar l'	26
Figura 6 – Exemplo do uso da geometria epipolar para encontrar correspondências entre imagens [2].	27
Figura 7 – Exemplos da retificação estéreo [3].(a) Par estéreo original. (b) Par estéreo retificado.	28
Figura 8 – Procedimento do <i>template matching</i> (Adaptado de [4]).	29
Figura 9 – Encontrando correspondências com a limitação da linha epipolar [5].	32
Figura 10 – Caso geral da visão estéreo [5].	33
Figura 11 – Exemplo de mapa de disparidade [5].	33
Figura 12 – Exemplo de triangulação ativa por luz estruturada [6].	34
Figura 13 – Exemplo da imagem capturada por um sensor que utiliza luz estruturada [6]. (a) Luz estruturada. (b) Mapa de disparidade.	35
Figura 14 – Falha de correspondência com detectores de quinas e mudança de escala.	37
Figura 15 – Primeira etapa do algoritmo SIFT (Adaptado de [7]).	38
Figura 16 – Segunda etapa do algoritmo SIFT (Adaptado de [7]).	38
Figura 17 – Exemplo do algoritmo SIFT [5]. (a) Definição de orientação. (b) Algumas <i>features</i> do SIFT com sua direção e escala.	39
Figura 18 – Da esquerda para a direita, a derivada parcial de segunda ordem da Gaussiana na direção y (L_{yy}) e na direção xy (L_{xy}), a aproximação da derivada parcial de segunda ordem da Gaussiana na direção y (D_{yy}) e na direção xy (D_{xy}) [8].	40
Figura 19 – Definição de orientação através de uma janela deslizante de tamanho $\frac{\pi}{3}$ [8].	41
Figura 20 – Construção do descritor do algoritmo SURF [8].	42
Figura 21 – Exemplo de testes binários com os descritores LDB e M-LDB [9]. (a) LDB. (b) M-LDB.	44

Figura 22 – Curva ROC (Adaptado de [3]). (a) A curva ROC, relacionando as taxas de verdadeiros positivos e falsos positivos. (b) A distribuição das <i>matches</i> e não- <i>matches</i> em função da distância d entre as <i>features</i> e o limite θ	45
Figura 23 – Exemplo dos métodos de <i>matching</i> [3].	46
Figura 24 – (a) Exemplo de uma <i>k-d tree</i> [3]. (b) O método BBF para a <i>k-d tree</i> [3].	47
Figura 25 – Exemplo do erro de re-projeção.	48
Figura 26 – Exemplo de uma nuvem de pontos 3D.	50
Figura 27 – Exemplo da reconstrução <i>Poisson</i> em 2D [10].	50
Figura 28 – Exemplo de variação de profundidade da <i>octree</i> no método <i>Poisson</i> . (a) Profundidade 6. (b) Profundidade 8. (c) Profundidade 10.	51
Figura 29 – Exemplo de um teste de visibilidade do <i>Greedy Projection Triangulation</i> para o ponto R [11].	52
Figura 30 – Exemplo <i>Grid Projection</i>	53
Figura 31 – Exemplo de reconstruções utilizando diferentes métodos. (a) <i>Poisson</i> . (b) <i>Greedy Projection Triangulation</i> . (c) <i>Grid Projection</i>	55
Figura 32 – Exemplo do processo de texturização. (a) Modelo 3D e as câmeras na posição da captura das imagens. (b) Modelo 3D texturizado.	56
Figura 33 – Exemplo de como as imagens devem ser capturadas. O círculo no centro representa o objeto, círculos menores com as setas representam as câmeras e suas direções, e as setas vermelhas indicam o sentido horário/anti-horário.	60
Figura 34 – Fluxograma simplificado da etapa de estimação da postura das câmeras.	62
Figura 35 – Exemplo do processo de criação de pares na etapa de estimação da postura das câmeras. As setas verdes representam pares que tiveram um número de <i>matches</i> maior que 100, já as vermelhas pares com menos que 100 <i>matches</i> . Os números nas setas servem para demonstrar a ordem com que as combinações aconteceram. No primeiro exemplo a imagem 0 fez par com as imagens 1, 2 e 6, já no segundo exemplo a imagem 1 fez par com as imagens 2, 5 e 6.	63
Figura 36 – Fluxograma simplificado da etapa de geração da nuvem densa.	65
Figura 37 – Exemplo do processo de criação de pares na etapa de geração da nuvem densa. As setas verdes representam pares que tiveram o ângulo entre os eixos óticos das câmeras no intervalo de 5° a 30°, já as vermelhas pares com ângulo fora desse intervalo.	65
Figura 38 – Correspondências encontradas utilizando o detector de <i>features</i> SIFT (apenas algumas linhas epipolares foram desenhadas para facilitar a visualização).	66

Figura 39 – Exemplo de uma <i>Seed</i> (seus pontos estão em vermelho). (a) Ponto 1 da <i>Seed</i> . (b) Ponto 2 da <i>Seed</i>	66
Figura 40 – Exemplo da janela de procura ao redor dos pontos da <i>Seed</i> . (a) Ponto 1 da <i>Seed</i> . (b) Ponto 2 da <i>Seed</i>	67
Figura 41 – Exemplo da janela de procura ao redor dos pixels da janela definida anteriormente. (a) Ponto 1 da <i>Seed</i> . (b) Ponto 2 da <i>Seed</i>	67
Figura 42 – Exemplo da máscara utilizada para controlar os pixels que já foram utilizados em correspondências. Os pontos pretos são os pixels já consumidos por alguma correspondência, os pixels brancos não possuem correspondências e estão livres.	68
Figura 43 – Exemplo do filtro para remover falsos positivos. (a) Nuvem sem filtro. (b) Nuvem filtrada.	69
Figura 44 – Fluxograma dos testes, os blocos na cor azul são os algoritmos desenvolvidos neste trabalho.	71
Figura 45 – Exemplos do dataset do trilho.	72
Figura 46 – Exemplos do dataset da gárgula.	72
Figura 47 – Exemplos do dataset da fonte.	73
Figura 48 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do Metashape. (a) Vista lateral. (b) Vista superior.	74
Figura 49 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do Meshroom. (a) Vista lateral. (b) Vista superior.	75
Figura 50 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do 3Dflow. (a) Vista lateral. (b) Vista superior.	75
Figura 51 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do VisualSFM (as cores dos pontos foram retiradas para facilitar a visualização). (a) Vista lateral. (b) Vista superior.	76
Figura 52 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do SFM-AKAZE. (a) Vista lateral. (b) Vista superior.	76
Figura 53 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do SFM-SIFT. (a) Vista lateral. (b) Vista superior.	77
Figura 54 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do SFM-SIFTGPU. (a) Vista lateral. (b) Vista superior.	77
Figura 55 – Resultado da nuvem densa do Metashape. (a) Vista lateral. (b) Vista superior.	78
Figura 56 – Resultado da nuvem densa do 3Dflow. (a) Vista lateral. (b) Vista superior.	79
Figura 57 – Resultado da nuvem densa do SFM-A + DEN. (a) Vista lateral. (b) Vista superior.	79

Figura 58 – Resultado da nuvem densa do SFM-S + DEN. (a) Vista lateral. (b) Vista superior.	80
Figura 59 – Resultado da nuvem densa do VSFM + DEN. (a) Vista lateral. (b) Vista superior.	80
Figura 60 – Resultado da nuvem densa do VSFM + P/C. (a) Vista lateral. (b) Vista superior.	80
Figura 61 – Resultado da superfície do Metashape. (a) Vista geral. (b) Vista detalhe.	81
Figura 62 – Resultado da superfície do Meshroom. (a) Vista geral. (b) Vista detalhe.	82
Figura 63 – Resultado da superfície do 3Dflow. (a) Vista geral. (b) Vista detalhe. .	82
Figura 64 – Resultado da superfície do VSFM + MR. (a) Vista geral. (b) Vista detalhe.	83
Figura 65 – Resultado da superfície do SFM-A + DEN + PR. (a) Vista geral. (b) Vista detalhe.	83
Figura 66 – Resultado da superfície do SFM-S + DEN + PR. (a) Vista geral. (b) Vista detalhe.	84
Figura 67 – Resultado da superfície do VSFM + DEN + PR. (a) Vista geral. (b) Vista detalhe.	84
Figura 68 – Resultado da superfície do VSFM + P/C + PR. (a) Vista geral. (b) Vista detalhe.	84
Figura 69 – Resultado da superfície do SFM-A + DEN + PCL-GY. (a) Vista geral. (b) Vista detalhe.	85
Figura 70 – Resultado da superfície do SFM-A + DEN + PCL-GD. (a) Vista geral. (b) Vista detalhe.	85
Figura 71 – Resultado da superfície do SFM-S + DEN +PCL-GY. (a) Vista geral. (b) Vista detalhe.	86
Figura 72 – Resultado da superfície do SFM-S + DEN + PCL-GD. (a) Vista geral. (b) Vista detalhe.	86
Figura 73 – Resultado da superfície do VSFM + DEN + PCL-GY. (a) Vista geral. (b) Vista detalhe.	87
Figura 74 – Resultado da superfície do VSFM + DEN + PCL-GD. (a) Vista geral. (b) Vista detalhe.	87
Figura 75 – Resultado da texturização do Metashape. (a) Vista detalhada. (b) Vista geral.	87
Figura 76 – Resultado da texturização do Meshroom. (a) Vista detalhada. (b) Vista geral.	88
Figura 77 – Resultado da texturização do 3Dflow. (a) Vista detalhada. (b) Vista geral.	89
Figura 78 – Resultado da texturização do VSFM + MR + TR. (a) Vista detalhada. (b) Vista geral.	89

Figura 79 – Resultado da texturização do VSFM + CMP-MVS. (a) Vista detalhada. (b) Vista geral.	89
Figura 80 – Resultado da texturização do SFM-A + DEN + PR + TR. (a) Vista detalhada. (b) Vista geral.	90
Figura 81 – Resultado da texturização do SFM-S + DEN + PR + TR. (a) Vista detalhada. (b) Vista geral.	90
Figura 82 – Resultado da texturização do VSFM + DEN + PR + TR. (a) Vista detalhada. (b) Vista geral.	91
Figura 83 – Resultado da texturização do VSFM + P/C + PR + TR. (a) Vista detalhada. (b) Vista geral.	91
Figura 84 – Resultado da texturização do SFM-A + DEN + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.	91
Figura 85 – Resultado da texturização do SFM-S + DEN + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.	92
Figura 86 – Resultado da texturização do VSFM + DEN + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.	92
Figura 87 – Resultado da texturização do VSFM + P/C + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.	93
Figura 88 – Resultado da reconstrução 3D da gárgula com a combinação SFM-S + DEN + PR + TR. (a) Vista frontal com textura. (b) Vista frontal sem textura. (c) Vista posterior com textura. (d) Vista posterior sem textura.	94
Figura 89 – Resultado da reconstrução 3D da gárgula com a combinação VSFM + MR + TR. (a) Vista frontal com textura. (b) Vista frontal sem textura. (c) Vista posterior com textura. (d) Vista posterior sem textura.	95
Figura 90 – Resultado da reconstrução 3D da gárgula com o <i>software</i> 3Dflow. (a) Vista frontal com textura. (b) Vista frontal sem textura. (c) Vista posterior com textura. (d) Vista posterior sem textura.	95
Figura 91 – Resultado da reconstrução 3D da fonte com a combinação SFM-S + DEN + PR + TR. (a) Vista lateral esquerda com textura. (b) Vista lateral direita com textura. (c) Vista lateral esquerda sem textura. (d) Vista lateral direita sem textura.	96
Figura 92 – Resultado da reconstrução 3D da fonte com a combinação VSFM + MR + TR. (a) Vista lateral esquerda com textura. (b) Vista lateral direita com textura. (c) Vista lateral esquerda sem textura. (d) Vista lateral direita sem textura.	97

Figura 93 – Resultado da reconstrução 3D da fonte com o *software* 3Dflow. (a) Vista lateral esquerda com textura. (b) Vista lateral direita com textura. (c) Vista lateral esquerda sem textura. (d) Vista lateral direita sem textura. 98

Lista de tabelas

Tabela 1 – Etapas do processo de reconstrução 3D executadas pelos trabalhos relacionados	58
Tabela 2 – Resultados quantitativos para a etapa de estimação da postura das câmeras para o dataset do trilho.	74
Tabela 3 – Resultados quantitativos para a etapa de geração da nuvem densa para o dataset do trilho.	78
Tabela 4 – Resultados quantitativos para a etapa da geração da superfície para o dataset do trilho.	81
Tabela 5 – Resultados quantitativos para a etapa da texturização para o dataset do trilho.	88
Tabela 6 – Resultados quantitativos para a reconstrução 3D do dataset do trilho. .	93
Tabela 7 – Resultados quantitativos para a reconstrução 3D do dataset da gárgula.	96
Tabela 8 – Resultados quantitativos para a reconstrução 3D do dataset da fonte. .	99
Tabela 9 – Exemplo dos tempos de algumas etapas para a geração da nuvem de pontos de um par na etapa de geração da nuvem densa.	104

Sumário

1	INTRODUÇÃO	17
1.1	Problemática da reconstrução 3D	18
1.2	Motivação e contexto	20
1.3	Objetivos	20
1.4	Estrutura do documento	20
2	REVISÃO BIBLIOGRÁFICA	21
2.1	Conceito chave	21
2.1.1	Coordenadas homogêneas	21
2.1.2	Formação da imagem	21
2.1.3	Homografia	24
2.1.4	Geometria epipolar	25
2.1.5	Retificação estéreo	27
2.1.6	<i>Template matching</i>	28
2.1.7	RANSAC	30
2.2	Visão 3D	31
2.2.1	Estereoscopia	31
2.2.2	Triangulação ativa por luz estruturada	34
2.3	Deteção de <i>features</i>	35
2.3.1	SIFT	37
2.3.2	SURF	39
2.3.3	AKAZE	41
2.4	<i>Matching</i>	44
2.5	<i>Bundle adjustment</i>	47
2.6	Geração de superfícies	49
2.6.1	<i>Poisson</i>	49
2.6.2	<i>Greedy Projection Triangulation</i>	51
2.6.3	<i>Grid Projection</i>	53
2.6.4	Comparação dos métodos	54
2.7	Texturização do modelo 3D	55
2.8	Trabalhos relacionados	57
3	ABORDAGEM PROPOSTA	59
3.1	Estimação da postura das câmeras	59
3.2	Geração da nuvem densa	62

4	EXPERIMENTOS E RESULTADOS	70
4.1	Datasets	71
4.2	Estimação da postura das câmeras	73
4.3	Geração da nuvem densa	77
4.4	Geração da superfície	81
4.5	Texturização	86
4.6	Datasets adicionais	93
4.6.1	Gárgula	94
4.6.2	Fonte	96
5	CONCLUSÕES	100
5.1	Considerações finais	100
5.2	Principais contribuições	102
5.3	Trabalhos futuros	103
	REFERÊNCIAS BIBLIOGRÁFICAS	105

1 Introdução

A captura de cenas através de fotografias se mostrou uma grande revolução no século XIX e desde então a quantidade e qualidade das fotografias só vem aumentando. Capturar fotos e vídeos se tornou algo corriqueiro, especialmente pela praticidade dos dispositivos móveis, que hoje possuem câmeras de alta qualidade. Assim como a fotografia outros campos da tecnologia avançaram, de maneira que hoje é possível capturar cenas não apenas em 2D como em fotografias, mas em 3D, mantendo a forma e textura de cenas completas. Existem várias maneiras de se capturar uma cena em 3D, triangulação ativa por luz estruturada, estereoscopia, LIDAR, etc. Em especial a que será abordada nesse trabalho é a utilização de imagens RGB de câmeras monoculares.

A fotogrametria surgiu com Laussedat em 1850, quando ele desenvolveu métodos para construir um mapa da cidade de Paris baseado na informação geométrica extraída de fotografias tiradas do alto dos telhados da cidade. Ele utilizou a distância entre as câmeras (*baseline*) e técnicas de intersecção para analisar seu par estereoscópico ponto por ponto. Laussedat é considerado o pai da fotogrametria, mas essa técnica só pode mostrar todo seu potencial após o desenvolvimento dos computadores e das câmeras digitais [12]. A fotogrametria, ou reconstrução 3D a partir de imagens 2D, é útil na área de entretenimento, com a criação de cenários para jogos eletrônicos, para vegetações, pedras e outros artefatos que demandariam muito tempo no processo manual de modelagem e texturização para ficarem realistas. Além da criação de cenários, muitos filmes e jogos eletrônicos fazem a reconstrução 3D de pessoas para a captura de expressões faciais. Dentro do âmbito da reconstrução de pessoas, outro segmento são as “selfies” em 3D, que são modelos 3D de pessoas. Essa técnica teve um maior destaque nos últimos anos através da popularização de drones, que baratearam significativamente o custo de fotografias aéreas, algo que antes só era possível através de aeronaves. Após essa popularização, serviços de inspeção através de drones começaram a surgir. Através dessas fotografias aéreas é possível reconstruir a área de interesse e realizar análises como a contabilização do volume de material em obras e mineradoras, a inspeção de telhados para a instalação de painéis solares, o cálculo de tensão em paredes de barragens, inspeção de transformadores de alta tensão, etc.

Muitas campanhas publicitárias tomam proveito dessa técnica, principalmente propagandas que envolvem alimentos, pois eles conseguem modificar a aparência do produto para que tenha a melhor apresentação ao consumidor final. Outra aplicação seria a preservação de obras históricas, este foi o intuito do projeto Mosul¹, que reconstruiu várias obras do museu Mosul após a depredação causada por terroristas, utilizando imagens tiradas por pessoas que visitaram o museu. Além de preservar as obras, a reconstrução

¹ <<https://projectmosul.org>>

3D permite que outras pessoas tenham acesso à arte sem a necessidade de visitar o local, ou até pessoas com deficiência visual poderiam apreciar as esculturas através de uma impressão 3D, já que tocar a obra muitas vezes não é uma opção viável.

1.1 Problemática da reconstrução 3D

Apesar de existirem diversos métodos para realizar a reconstrução 3D a partir de imagens 2D, em geral, esse processo é bem definido e pode ser dividido em algumas etapas distintas, essas etapas podem ser visualizadas na Figura 1. A primeira etapa tem como objetivo estimar a postura das câmeras, para isso, o procedimento básico é detectar as *features* (Seção 2.3) em todas as imagens e realizar o *matching* (Seção 2.4) das *features* de uma imagem com as outras. Após o *matching* das *features*, tenta-se estimar a postura relativa entre as imagens de cada par. Com as posturas dos pares calculadas, um par inicial é selecionado e a postura dos outros pares é atualizada para ter como referência esse par inicial. Como a postura das câmeras é estimada através desses *matchings*, ela não vai estar totalmente correta. No processo de detecção de *features* o centro da *feature* pode estar deslocado por alguns pixels. No processo de correspondência das *features* podem ocorrer correspondências erradas, além de outros erros causados pela mudança de escala e perspectiva da imagem. Por esses motivos utiliza-se o *bundle adjustment* (Seção 2.5) durante a estimação da postura das câmeras.



Figura 1 – Etapas da reconstrução 3D a partir de imagens 2D.

A segunda etapa é a geração da nuvem densa. Existem diversas maneiras de realizar o cálculo da nuvem densa, as principais são o cálculo de mapas de profundidade e a propagação de *matches*. Em geral esses métodos utilizam as restrições definidas pela geometria epipolar (Seção 2.1.4) e procuram por pontos que tenham um determinado grau de similaridade calculado através de algum método de *template matching* (Seção 2.1.6). Uma etapa necessária após o cálculo da nuvem densa é o cálculo da normal, pois alguns métodos de geração de superfície necessitam de normais corretamente orientadas, como é o caso do *Poisson*.

A terceira etapa é a geração da superfície (Seção 2.6), essa etapa varia muito dependendo do método utilizado, mas o objetivo é tentar aproximar uma superfície através dos pontos da nuvem densa. Isso pode ser feito de diversas maneiras, um exemplo seria

através da aproximação de uma função, como no método *Poisson* ou de maneira iterativa, como no *Greedy Projection Triangulation*.

A quarta etapa é a texturização (Seção 2.7), que é responsável por texturizar as faces da superfície 3D. Essa etapa pode não ser necessária, pois durante a geração da nuvem densa pode-se colorir a nuvem de pontos 3D utilizando uma média da cor dos pixels que originaram cada ponto. Esse método de colorir a nuvem de pontos é indicado quando a nuvem é muito densa, ou quando se utiliza uma câmera que capta comprimentos de onda diferentes dos que caracterizam a luz visível, como uma câmera térmica.

Pode-se ainda adicionar uma etapa de filtragem, que seria a utilização de filtros para remover possíveis ruídos, filtros de simplificação, etc. Os filtros de simplificação são úteis para diminuir o processamento necessário para a visualização das reconstruções 3D, algo que se mostra necessário com a visualização através de óculos de realidade virtual, onde travamentos podem impossibilitar seu uso, além de causarem enjoos aos usuários.

Infelizmente não são todos os objetos que podem ser reconstruídos utilizando essa técnica de reconstrução 3D. Ela possui algumas limitações, sendo a maioria das limitações relacionadas ao processo de detecção de *features*. Pois se não é possível encontrar características únicas, distinguíveis e constantes entre as imagens, não pode-se retirar as informações relevantes para o processo de correspondência dessas características. Esses problemas podem ser percebidos em objetos brilhosos, reflexivos, transparentes e sem textura. Pois objetos brilhosos e reflexivos vão criar vários pontos brilhantes que não irão manter sua posição nas imagens subsequentes. Objetos transparentes e sem textura não vão ter características distinguíveis para serem relacionadas entre uma imagem e outra. Objetos que estão em movimento também devem ser evitados, já que uma das premissas do algoritmo é que os objetos em cena estão estáticos. Muitas vezes pode-se ter um objeto que esteja dentro dos parâmetros necessários, mas o ambiente o qual ele está inserido traz muitos ruídos, isso pode acabar acrescentando erros no processo, por isso recomenda-se o use de fundos que não possuam textura, ou caso possuam, que sejam estáticos. Outra limitação desse processo é que, por utilizar imagens, a presença de luz é essencial, pois com pouca iluminação não é possível detectar as características das imagens. Mudanças de iluminação entre as imagens também prejudica o resultado, pois características encontradas em uma imagem podem não ser encontrada nas outras, ou caso seja encontrada, não consiga ser relacionada pela diferença de iluminação.

Como a base do processo é a correspondência de características de uma imagem com características de outra imagem, deve-se possuir uma sobreposição de 60% a 90% entre as imagens. Caso exista pouca sobreposição, não serão encontradas muitas correspondências, o que pode diminuir muito a confiança na postura relativa entre as câmeras.

1.2 Motivação e contexto

A reconstrução 3D é uma técnica antiga, com aplicações em diversas áreas, como no entretenimento, construção civil, inspeção em áreas de risco, etc. Apesar de ser antiga o surgimento de novas tecnologias, como as impressoras 3D, faz com que essa técnica ganhe novas aplicações e desafios. O estudo e o esclarecimento das etapas da reconstrução 3D é o primeiro passo para propor possíveis soluções aos desafios encontrados. Isso motiva o desenvolvimento do presente trabalho, de forma a documentar e desenvolver algumas das etapas da reconstrução 3D.

1.3 Objetivos

O objetivo geral deste trabalho é estudar, avaliar e desenvolver algoritmos envolvidos no processo de reconstrução 3D através de imagens 2D adquiridas por câmeras monoculares. O trabalho apresenta os seguintes objetivos específicos:

- Aprofundar os conhecimentos nas etapas da reconstrução 3D.
- Implementar um algoritmo para a estimação da postura das câmeras
- Implementar um algoritmo para a geração da nuvem densa.
- Implementar os algoritmos utilizando bibliotecas de código aberto.
- Utilizar a GPU para realizar os cálculos quando possível.
- Utilizar o processamento *multithread* quando possível.

1.4 Estrutura do documento

No capítulo 2 são abordados os principais tópicos relacionados a reconstrução 3D a partir de imagens 2D, onde são descritos os métodos utilizados no desenvolvimento deste trabalho, além de apresentar alguns trabalhos relacionados. No capítulo 3 é descrito o desenvolvimento dos algoritmos de algumas das etapas da reconstrução 3D. No capítulo 4 são apresentados os experimentos, como eles foram conduzidos e seus resultados, com uma breve discussão. No capítulo 5 são descritas as conclusões obtidas com os experimentos e sugestões para trabalhos futuros.

2 Revisão Bibliográfica

2.1 Conceito chave

2.1.1 Coordenadas homogêneas

Coordenadas homogêneas, ou coordenadas no espaço projetivo \mathbb{P}^n , são muito utilizadas na reconstrução 3D pois permitem a manipulação de um vetor de N dimensões em um espaço de dimensão $N + 1$. A conversão entre coordenadas euclidianas e coordenadas homogêneas de um ponto 2D pode ser vista na Equação 2.1. Uma das vantagens de se utilizar essas coordenadas é a possibilidade de representar linhas e pontos no infinito utilizando apenas números reais. Outra propriedade importante é à invariabilidade a escala, um ponto em coordenadas homogêneas \tilde{x} e $\tilde{x} = \alpha\tilde{x}$ representam o mesmo ponto em coordenadas euclidianas para todo $\alpha \neq 0$ [4].

$$(x, y) = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \begin{bmatrix} x \\ y \\ w \end{bmatrix} = (x/w, y/w) \quad (2.1)$$

2.1.2 Formação da imagem

O processo de formação da imagem, em um olho ou uma câmera, envolve a projeção de um mundo tridimensional em uma superfície bidimensional [4]. Um exemplo simples seria a câmera *pinhole*, que consiste em um pequeno orifício na parede de algum objeto fechado. A luz entra pelo orifício e é projetada de forma invertida na parede oposta ao orifício, formando a imagem. Existem diversas maneiras de se modelar uma câmera, mas o modelo utilizado nesse trabalho será o modelo de projeção central [4], que pode ser observado na Figura 2.

Com esse modelo pode-se encontrar uma relação entre os pontos no plano da imagem 2D com os pontos no espaço 3D. Diferentemente do modelo de câmera *pinhole*, este modelo considera uma câmera com lente. A presença de uma lente faz com que os raios de luz sejam concentrados em um ponto, no caso, a origem da câmera $\{C\}$. O uso de lentes faz com que não seja mais necessária uma grande quantidade de luz para a formação da imagem, mas cria-se a necessidade de ajustar o foco para uma distância específica, algo que não acontece na câmera *pinhole*, que tem toda a cena em foco. Com isso, tem-se que o plano da imagem é formado em $z = f$, onde f é a distância focal. Utilizando a similaridade de triângulos pode-se mostrar que o ponto P no espaço 3D é projetado no

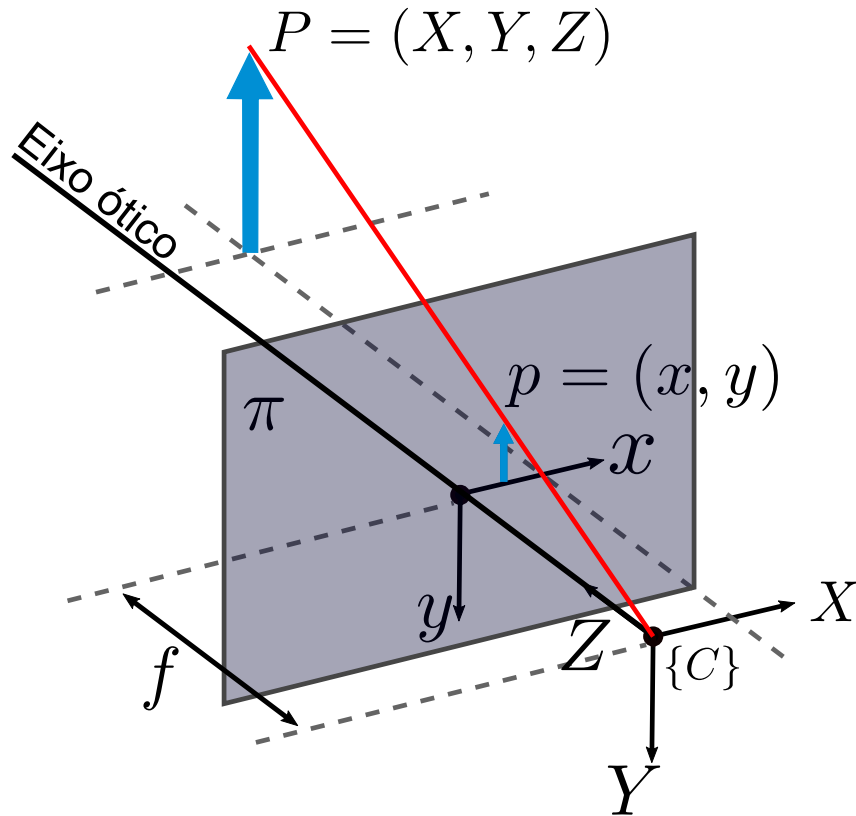


Figura 2 – Modelo de projeção central.

ponto p da imagem através da equação [4]:

$$x = f \frac{X}{Z}, y = f \frac{Y}{Z} \quad (2.2)$$

Essa equação é uma transformação projetiva, o que lhe confere algumas propriedades [4]:

- Linhas retas no mundo são projetadas como linhas retas no plano da imagem.
- Linhas paralelas no mundo se interceptam no horizonte no plano da imagem, com exceção de linhas paralelas ao eixo x do plano da imagem.
- Cônicas são projetadas para cônicas no plano da imagem, mas não necessariamente se mantêm iguais, um círculo pode ser projetado como um círculo ou como uma elipse, por exemplo.
- O tamanho e o formato não são preservados.
- O mapeamento não é único, um ponto $p = (x, y)$ não consegue definir unicamente um ponto $P = (X, Y, Z)$. Sabe-se apenas que o ponto P pertence à linha \overline{pC} (linha vermelha na Figura 2).

Pode-se escrever as coordenadas do ponto no plano da imagem em sua forma homogênea $\tilde{p} = (\tilde{x}, \tilde{y}, \tilde{z})$, de forma que o mapeamento é dado por:

$$\tilde{p} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (2.3)$$

Escrevendo o ponto do espaço 3D na sua forma homogênea $\tilde{P} = (X, Y, Z, 1)^T$, tem-se:

$$\tilde{p} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (2.4)$$

ou

$$\tilde{p} = C\tilde{P} \quad (2.5)$$

Onde a matriz C é chamada de matriz da câmera. Apesar dessa equação relacionar um ponto no plano da imagem com um ponto no espaço 3D, deve-se perceber que as imagens serão manipuladas em um computador, sendo assim, o plano da imagem deve ser discretizado, sendo dividido em pixels. Essa discretização pode ser observada na Figura 3, ela é um *grid* de $L \times A$ sendo L a largura da imagem e A sua altura. A origem do *grid* fica no canto superior esquerdo e ele não possui valores negativos. Suas coordenadas são definidas pelo par (u, v) [4].

A relação entre as coordenadas do plano da imagem, dada em metros, para as coordenadas do *grid*, em pixels, é dada por:

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \begin{bmatrix} \frac{1}{\rho_u} & 0 & u_0 \\ 0 & \frac{1}{\rho_v} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \end{bmatrix} \quad (2.6)$$

Onde (u_0, v_0) são as coordenadas do ponto principal em pixels (o centro da imagem em uma câmera ideal), ρ_u é a dimensão horizontal dos pixels em metros e ρ_v é a dimensão vertical dos pixels em metros. Combinando essas relações e adicionando a posição da câmera, pode-se chegar a relação que define o modelo completo da câmera [4]:

$$\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{1}{\rho_u} & 0 & u_0 \\ 0 & \frac{1}{\rho_v} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_K \underbrace{\begin{bmatrix} R & t \\ 0_{1 \times 3} & 1 \end{bmatrix}^{-1}}_{\zeta_C^{-1}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \quad (2.7)$$

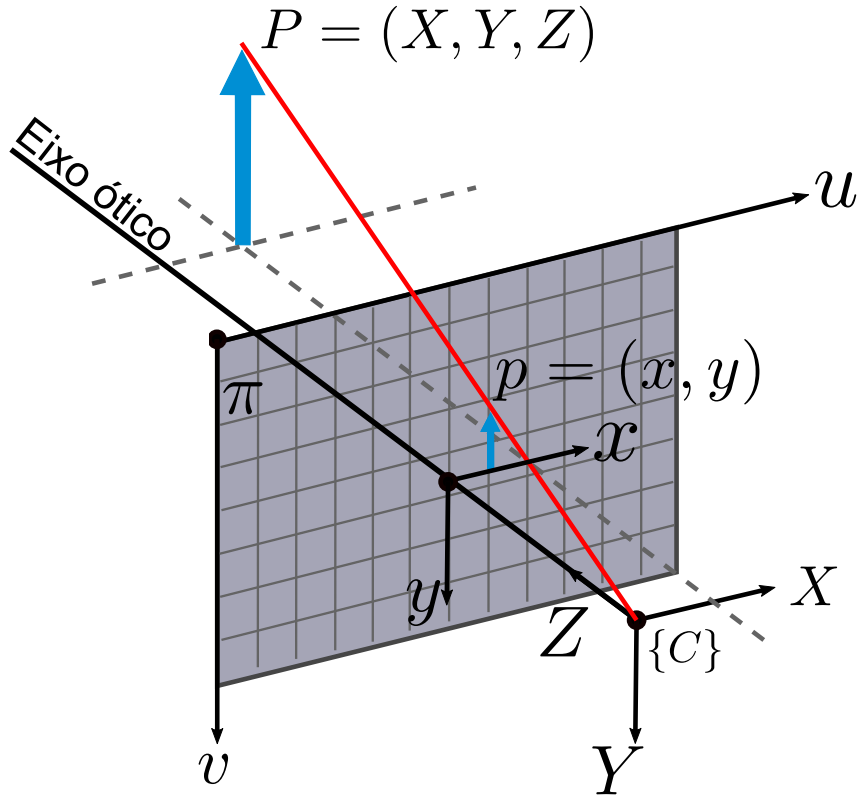


Figura 3 – Modelo de projeção central mostrando o plano da imagem com os pixels discretos.

Onde C é a matriz da câmera, K é a matriz de parâmetros intrínsecos da câmera e ζ_C é a matriz de parâmetros extrínsecos da câmera, R é uma matriz de rotação 3×3 e t é um vetor de translação 3×1 . A matriz de parâmetros extrínsecos descreve a posição da câmera no espaço 3D. Como tem-se cinco parâmetros intrínsecos e seis parâmetros extrínsecos, a matriz da câmera tem apenas onze graus de liberdade, sendo assim, pode-se fixar o elemento C_{34} (fator de escala). O fator de escala pode ser escolhido de forma arbitrária, mas geralmente utiliza-se $C_{34} = 1$ [4].

2.1.3 Homografia

A homografia, ou transformação projetiva, é um tipo de transformação 2D definida por uma matriz 3×3 com oito graus de liberdade. Essa transformação trabalha com coordenadas homogêneas e é caracterizada por manter as linhas retas das imagens, mas não preservar orientação, ângulos, comprimentos e paralelismo. Ela pode ser estimada a partir de quatro pontos em um plano e seus correspondentes em outro plano [2]. A relação entre esses pontos pode ser vista na equação a seguir:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2.8)$$

A homografia pode ser utilizada para a correção de perspectiva de fotos, adição de efeitos visuais (como a ordem de chegada de atletas da nataç o), realidade aumentada, constru o de panoramas e a estimac o da posic o da c mera. Para a estimac o da posic o da c mera   necess rio algum tipo de padr o conhecido na imagem (como o AprilTags [13]), e que a c mera esteja calibrada. O algoritmo de estimac o de postura ir  tentar encontrar o padr o e calcular a matriz de homografia que faz o padr o ser projetado da maneira que est  sendo observado pela c mera, com isso pode-se determinar a posic o da c mera relativa ao padr o [2]. Na Figura 4 pode-se observar alguns exemplos do uso da homografia.

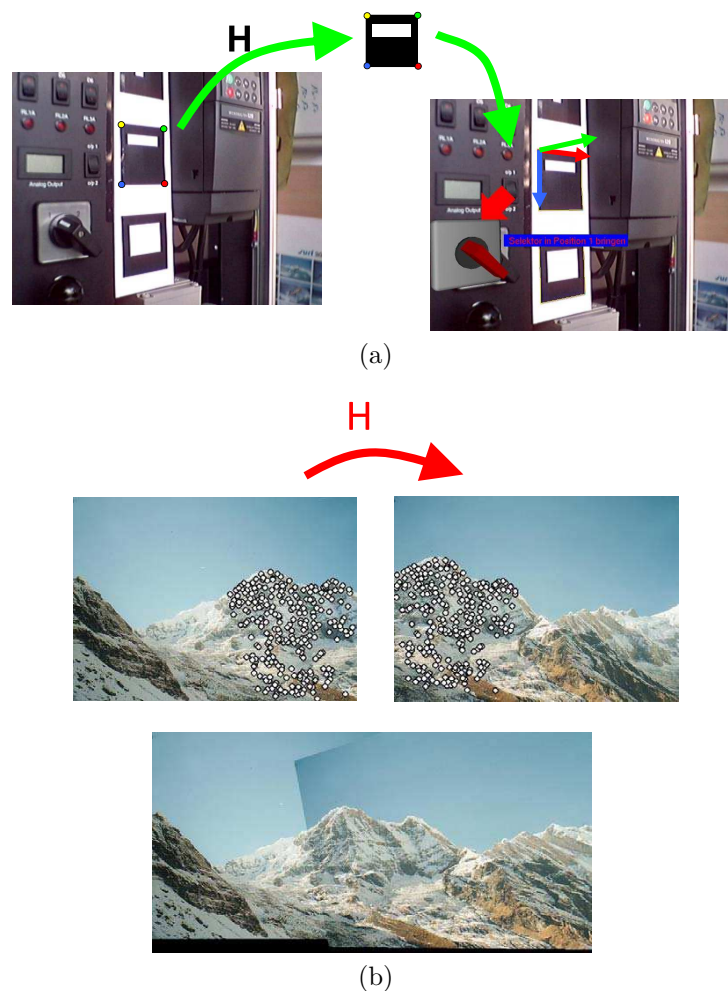


Figura 4 – Exemplos do uso da homografia [1]. (a) Realidade aumentada. (b) Constru o de panorama.

2.1.4 Geometria epipolar

A geometria epipolar   a geometria projetiva intr nseca entre duas vistas, ela   dependente apenas dos par metros intr nsecos das c meras e de suas posturas relativas. Apesar disso, ela pode ser calculada a partir de correspond ncias entre as duas imagens.

A geometria epipolar é essencialmente a geometria de intersecção dos planos das imagens com um conjunto de planos que tem como *baseline* o centro das câmeras. Dado um ponto X no espaço 3D, nomeado como x na primeira vista e x' na segunda vista, pode-se criar um plano entre o centro das câmeras, os pontos x e x' nos planos das imagens e o ponto X no espaço 3D. Esse plano é chamado de plano epipolar (π) e pode ser observado na Figura 5a [2].

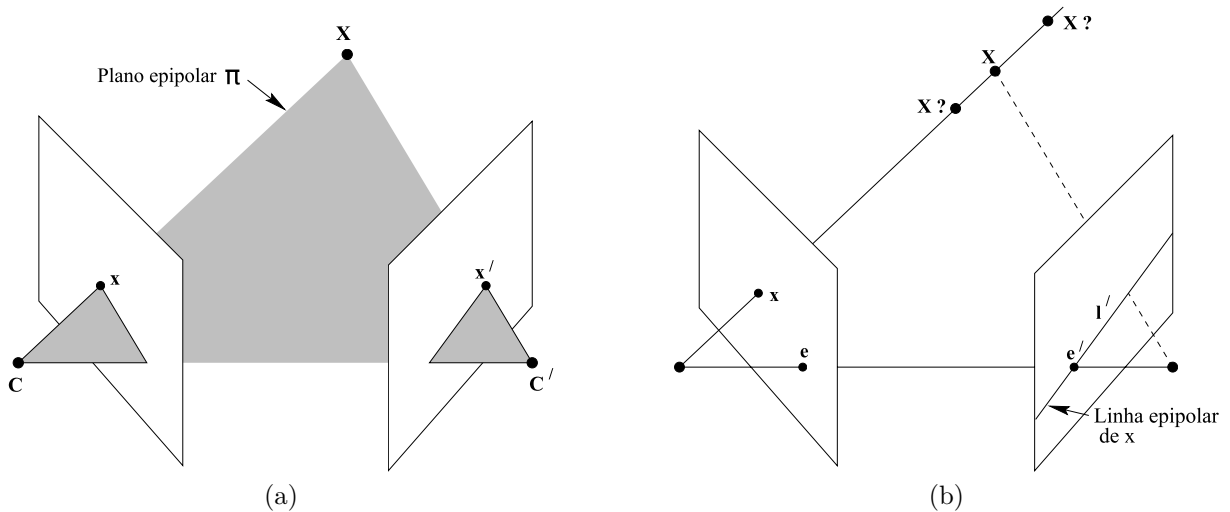


Figura 5 – Exemplos da geometria epipolar (Adaptado de [2]).(a) Plano epipolar π . (b) Linha epipolar l' .

Essa geometria é utilizada para a procura de pontos correspondentes entre câmeras. Supondo que se conheça apenas o ponto x e as propriedades das câmeras, com isso já é possível definir o plano epipolar. Sabe-se que o ponto x' está no plano π e no plano da imagem. Como a linha l' representa a intersecção do plano π com o plano da imagem, necessariamente o ponto x' está sobre a linha l' . Essa linha é chamada de linha epipolar. Essa restrição facilita a procura de pontos correspondentes entre as câmeras, pois não é mais necessário realizar a busca em toda a imagem. A linha epipolar apenas indica os possíveis pontos correspondentes, pois não são todos os pontos que encontram uma correspondência, já que a segunda câmera pode ter perdido visão do ponto X . Na Figura 5b pode-se observar a linha epipolar e os epípolos e e e' , o epípolo é o ponto de intersecção do plano das imagens com a linha que conecta os centros das câmeras [2]. Um exemplo do uso dessa técnica para a procura de correspondências em imagens pode ser observado na Figura 6, os pontos brancos representam as *features* encontradas em cada imagem e as linhas brancas são as linhas epipolares, analisando-se o terceiro ponto de cima para baixo, fica claro que as linhas epipolares estão corretas, pois passam pelo mesmo ponto na pintura do vaso.

A geometria epipolar pode ser representada algebricamente através da matriz fundamental F , que é uma matriz 3×3 de *rank* 2, ou seja, ela tem duas linhas que são linearmente

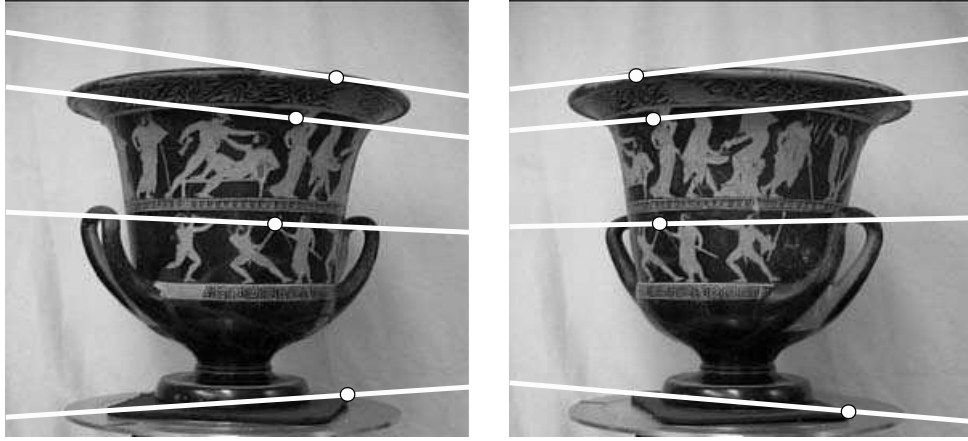


Figura 6 – Exemplo do uso da geometria epipolar para encontrar correspondências entre imagens [2].

independentes. Essa matriz é usada para relacionar as duas vistas. Ela mapeia os pontos de uma imagem para suas correspondentes linhas epipolares na outra imagem. A condição que os pontos devem satisfazer para serem considerados correspondentes é a seguinte:

$$x'^T F x = 0 \quad (2.9)$$

Essa afirmação é verdadeira, pois, se x e x' são correspondentes, então x' está na linha epipolar $l' = Fx$ de x , de forma que $0 = x'^T l' = x'^T Fx$. Essa relação de correspondência dos pontos é importante, pois consegue descrever a matriz fundamental sem a necessidade de conhecer as propriedades das câmeras, permitindo que a matriz fundamental seja calculada apenas através dos pontos correspondentes na imagem.

2.1.5 Retificação estéreo

Como pode-se observar na Figura 6, a geometria epipolar é utilizada para encontrar a correspondência entre imagens, mas essa procura pode-se tornar muito custosa computacionalmente caso tenha-se que calcular cada novo ponto da linha epipolar utilizando a equação da reta. Apesar de não ser uma tarefa complicada, é um cálculo que será realizado várias vezes e que pode ser evitado. Uma forma de lidar com este problema é a retificação estéreo. A retificação estéreo consiste na aplicação de uma transformação na imagem de forma que ambas as imagens sejam projetadas em um mesmo plano, assim as linhas epipolares não variam seus valores em y . Desse modo, a procura pelas correspondências é facilitada, pois a coordenada y da linha e da *feature* que se deseja procurar serão iguais. Para determinar a coordenada x serão feitos incrementos de um valor inicial até um valor final, não são verificados todos os valores de x pois se assume que existe uma pequena variação de uma câmera para outra [3].

Essa abordagem da retificação estéreo é utilizada em várias câmeras estéreo, e permite determinar uma relação simples entre a profundidade Z e a disparidade d . Sendo a

disparidade a diferença entre a coordenada x de uma *feature* em uma imagem com a coordenada x dessa mesma *feature* na outra imagem. Na Equação 2.10 pode-se observar essa relação [3]:

$$d = f \frac{B}{Z} \quad (2.10)$$

Onde f é a distância focal e B é a *baseline* (distância entre o centro das câmeras). Um exemplo da retificação estéreo pode ser visto na Figura 7. Na Figura 7a pode-se observar que as linhas epipolares não estão paralelas nem alinhadas, o que dificulta a procura pela *feature* correspondente. Na Figura 7b as linhas das imagens estão alinhadas, de forma que a procura da *feature* correspondente vai acontecer apenas no eixo x .

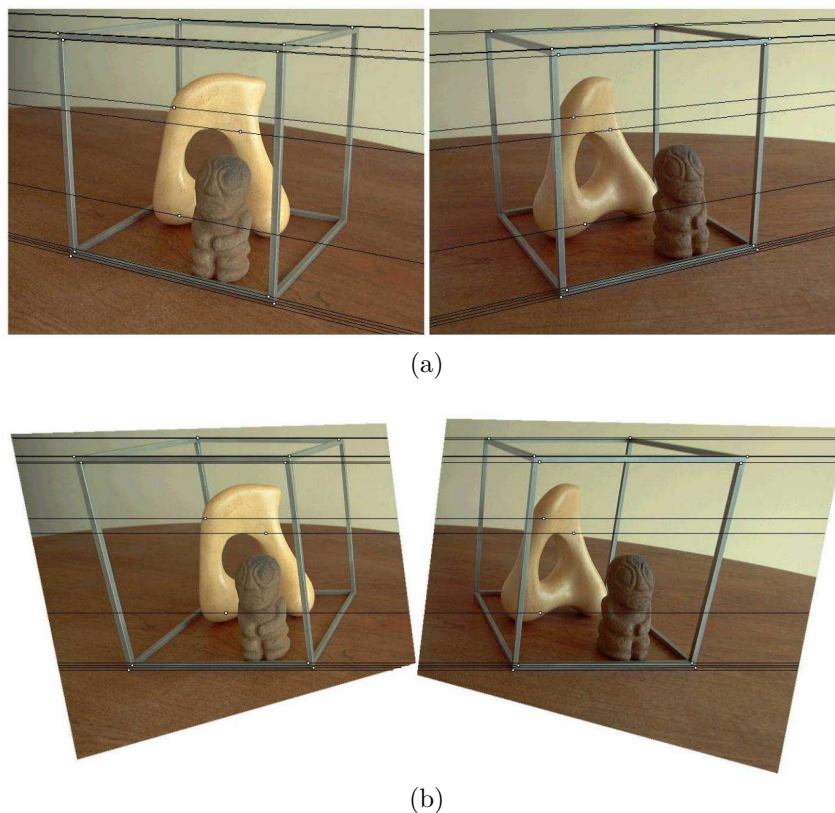


Figura 7 – Exemplos da retificação estéreo [3].(a) Par estéreo original. (b) Par estéreo retificado.

2.1.6 *Template matching*

O *template matching* é uma técnica que visa definir qual parte de uma imagem é mais parecida com um determinado modelo (*template*). O procedimento do *template matching* pode ser observado na Figura 8. Nela tem-se a imagem de entrada I , a imagem de saída O , o *template* T , a função de similaridade s e a janela de procura W . A janela W irá

percorrer todos os pixels da imagem I à procura do *template* T . A cada mudança de posição da janela W o seguinte cálculo é realizado [4]:

$$O[u, v] = s(T, W), \forall (u, v) \in I \quad (2.11)$$

Através deste cálculo todos os pixels da imagem de saída O são calculados. O tamanho do *template* T e da janela W é igual e deve ser ímpar, de forma que $w = 2h + 1$.

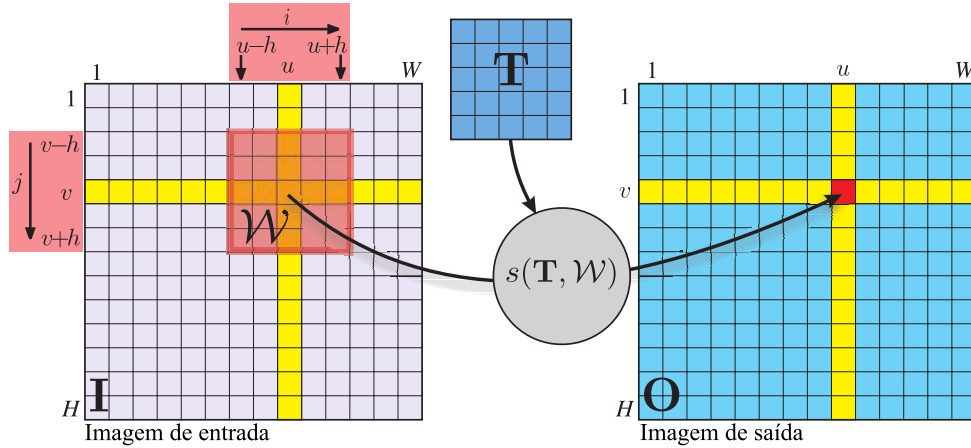


Figura 8 – Procedimento do *template matching* (Adaptado de [4]).

As funções de similaridade mais intuitivas são as que calculam diferença entre os pixels de T e W , tais como a soma das diferenças absolutas (*sum of the absolute differences* SAD) ou a soma das diferenças quadráticas (*sum of the squared differences* SSD). O problema desses métodos é que não é possível determinar o que é ou não uma boa correspondência, apenas qual a melhor entre todas as alternativas testadas. Uma alternativa para contornar este problema consiste na utilização da correlação cruzada normalizada (*normalized cross-correlation* NCC), que faz com que os resultados tenham um valor que varia de -1 a +1, sendo +1 uma região idêntica. Sua única desvantagem é que ela tem um maior custo computacional, mas isso é compensado pelo fato dela ser invariante a mudanças de intensidade, o que é útil em casos de mudanças de iluminação entre uma imagem e outra. Na lista abaixo apresenta-se as funções de similaridade mais comumente utilizadas. As funções que iniciam com o prefixo Z são invariantes a offsets de intensidade, para isso elas subtraem do valor de intensidade o valor médio de intensidade \bar{I}_x da região da imagem.

- *Sum of the absolute differences* (SAD)

$$s = \sum_{(u,v) \in I_1} |I_1[u, v] - I_2[u, v]| \quad (2.12)$$

- *Zero-mean sum of the absolute differences* (ZSAD)

$$s = \sum_{(u,v) \in I_1} |(I_1[u, v] - \bar{I}_1) - (I_2[u, v] - \bar{I}_2)| \quad (2.13)$$

- *Sum of the squared differences* (SSD)

$$s = \sum_{(u,v) \in I_1} (I_1[u, v] - I_2[u, v])^2 \quad (2.14)$$

- *Zero-mean sum of the squared differences* (ZSSD)

$$s = \sum_{(u,v) \in I_1} ((I_1[u, v] - \bar{I}_1) - (I_2[u, v] - \bar{I}_2))^2 \quad (2.15)$$

- *Normalized cross-correlation* (NCC)

$$s = \frac{\sum_{(u,v) \in I_1} I_1[u, v] I_2[u, v]}{\sqrt{\sum_{(u,v) \in I_1} I_1^2[u, v] \sum_{(u,v) \in I_1} I_2^2[u, v]}} \quad (2.16)$$

- *Zero-mean normalized cross-correlation* (ZNCC)

$$s = \frac{\sum_{(u,v) \in I_1} (I_1[u, v] - \bar{I}_1)(I_2[u, v] - \bar{I}_2)}{\sqrt{\sum_{(u,v) \in I_1} (I_1[u, v] - \bar{I}_1)^2 \sum_{(u,v) \in I_1} (I_2[u, v] - \bar{I}_2)^2}} \quad (2.17)$$

2.1.7 RANSAC

*RAN*dome *SAM*ple *CON*sensus (RANSAC) é um método iterativo para aproximar um modelo matemático a partir de dados experimentais. Esse método é capaz de suavizar dados com uma quantidade significativa de *outliers*, e é ideal para as aplicações que os dados são gerados a partir de detectores de *features* (Seção 2.3) [14]. Para realizar a aproximação do modelo pode-se definir duas tarefas diferentes. Primeiro deve-se encontrar o melhor conjunto de pontos que satisfaz um determinado modelo (o problema da classificação), e após isso deve-se calcular os melhores valores para os parâmetros livres do modelo selecionado (o problema da estimação de parâmetros). Técnicas clássicas de estimação de parâmetros, como os mínimos quadrados, otimizam um modelo para se ajustar a todos os dados do conjunto, o problema disso é a incapacidade de rejeitar erros. O RANSAC é útil quando se trabalha com detectores de *features* pois os detectores criam dois tipos de erros, erros de classificação e erros de medição. Erros de classificação ocorrem quando o detector indica incorretamente uma área da imagem como sendo uma *feature*, sendo esse um erro grosseiro que deve ser eliminado na primeira etapa de seleção dos pontos. Erros de medição ocorrem quando o detector detecta a *feature*, mas erra alguma de suas propriedades, como sua orientação, sendo esse erro corrigido durante a estimação dos parâmetros [14].

Ao contrário das técnicas clássicas que tentam usar o máximo de dados possíveis para aproximar um modelo, o RANSAC inicia utilizando o mínimo de dados possíveis, e vai adicionando dados consistentes quando possível. A lógica de funcionamento do

RANSAC pode ser definida como: dado um modelo que requer um mínimo de n pontos para instanciar seus parâmetros livres e o conjunto de dados P (com P tendo n ou mais pontos), randomicamente selecione um subconjunto S_1 de n pontos de P e instancie o modelo. Use o modelo M_1 instanciado para determinar o subconjunto S_1^* de pontos em P que estão a uma certa tolerância de M_1 . O Conjunto S_1^* é chamado de conjunto consenso de S_1 . Se o número de elementos de S_1^* é maior que um limite t , que é função do número estimado de erros em P , use S_1^* para calcular um novo modelo M_1^* . Se o número de elementos de S_1^* é menor que um limite t , randomicamente selecione um subconjunto S_2 e repita o processo. Caso após um determinado número de tentativas não se chegar a um subconjunto que tenha mais elementos que t , use o o subconjunto com o maior número de elementos, ou acabe o processo como falha [14].

O RANSAC contém três parâmetros não especificados: o erro usado para determinar se um ponto é ou não compatível com o modelo; o número de tentativas de criação de subconjuntos; e o limite t , que é o número de pontos necessários para um subconjunto ser considerado correto [14].

- Caso o modelo seja função dos pontos, pode-se definir uma tolerância de forma analítica. Como esse não é o caso mais comum, pode-se definir o erro de forma experimental: amostras de desvio podem ser criadas adicionando ruídos nos dados, calculando o modelo e medindo os erros. O erro pode ser definido como um ou dois desvios padrões além dos erros medidos.
- A decisão de parada de seleção de novos subconjuntos de P pode ser baseada no número k de tentativas esperadas para encontrar um subconjunto com n *inliers*. Sendo w a probabilidade de um ponto estar dentro da tolerância do modelo e z a probabilidade de que pelo menos um dos subconjuntos tenha n pontos sem erros, tem-se:

$$k = \log(1 - z) / \log(1 - w^n) \quad (2.18)$$

- O limite t deve ser grande o suficiente para satisfazer dois propósitos: que o modelo correto foi encontrado, e que uma quantidade suficiente de pontos mutualmente consentintes foram encontrados.

2.2 Visão 3D

2.2.1 Estereoscopia

Estereoscopia é o processo de estimação de profundidade através de duas imagens de uma mesma cena capturadas de ângulos diferentes. Este é o mesmo processo realizado em nossos cérebros para ter a sensação de profundidade em uma cena. A estereoscopia

pode ser dividida em dois problemas, o problema da correspondência e o da reconstrução 3D. O primeiro problema consiste em encontrar pontos nas duas imagens que estejam relacionados com o mesmo ponto na cena, pode-se chamar esses pontos de correspondências. As correspondências podem ser encontradas assumindo que existe uma pequena variação entre a posição das duas câmeras, mas somente essa limitação ainda faz com que tenha-se muitas falsas correspondências. Para restringir ainda mais a procura das correspondências, introduz-se a limitação da linha epipolar, esta limitação diz que um ponto na imagem 1 só pode estar relacionado com um ponto que esteja sobre a linha epipolar da imagem 2 [5]. Um exemplo disso pode ser visto na Figura 9. Na imagem da esquerda foram encontradas cinco *features* representadas pelos quadrados. Seguindo os princípios da geometria epipolar, sabe-se que a correspondência dessas *features* na imagem da direita vai estar sobre alguma linha na imagem da direita. A primeira linha de cima para baixo está relacionada com o primeiro quadrado de cima para baixo e assim sucessivamente.



Figura 9 – Encontrando correspondências com a limitação da linha epipolar [5].

Conhecendo essas correspondências entre as duas imagens, a posição relativa entre as duas câmeras, e os parâmetros intrínsecos das câmeras (distância focal, dimensão horizontal e vertical dos pixels e o ponto principal da câmera) é possível reconstruir a cena. Para adquirir a posição relativa entre as duas câmeras é necessário algum tipo de calibração, esta calibração é usualmente feita com um padrão conhecido que é apresentado para a câmera, desta forma é possível extrair a posição relativa entre as duas câmeras [5].

Pode-se demonstrar o problema com um caso simplificado, que é ilustrado na Figura 10. Considera-se duas câmeras com a mesma orientação, com eixos óticos paralelos e uma separação b (*baseline*). Tem-se um ponto do objeto nas coordenadas (X, Y, Z) . As coordenadas das imagens da esquerda e da direita são respectivamente, (u_l, v_l) e (u_r, v_r) . Analisando a Figura 10 e utilizando a relação entre os triângulos, pode-se escrever [5]:

$$\frac{f}{Z} = \frac{u_l}{X} \quad (2.19)$$

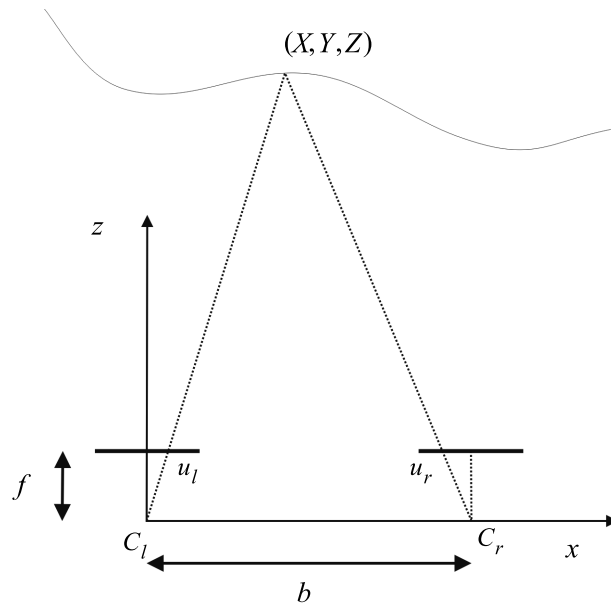


Figura 10 – Caso geral da visão estéreo [5].

$$\frac{f}{Z} = \frac{-u_r}{b - X} \quad (2.20)$$

$$Z = b \frac{f}{u_l - u_r} \quad (2.21)$$

A diferença das coordenadas da imagem $u_l - u_r$ é chamada de disparidade. Esta é uma informação importante, pois é ela que nos possibilita adquirir a profundidade. Um mapa de disparidade pode ser observado na Figura 11, quanto mais escuro maior é a profundidade do ponto na imagem.



Figura 11 – Exemplo de mapa de disparidade [5].

2.2.2 Triangulação ativa por luz estruturada

A triangulação ativa por luz estruturada tem um funcionamento parecido com a estereoscopia, mas ao invés de duas câmeras, tem-se um projetor e uma câmera. O projetor, LCD ou infravermelho, é responsável por projetar um padrão conhecido na cena. A câmera deverá ser capaz de identificar esse padrão na cena, ou seja, a luz emitida pelo projetor deve refletir em algum objeto e retornar para a câmera. A deformação causada durante a reflexão desse padrão conhecido que irá possibilitar a extração da profundidade. Para isso ser possível é necessária uma etapa de calibração, que consiste em projetar o padrão em um plano a uma distância conhecida, isso permitirá o prévio conhecimento das relações geométricas entre a câmera e os pontos no padrão projetado [15] [6].

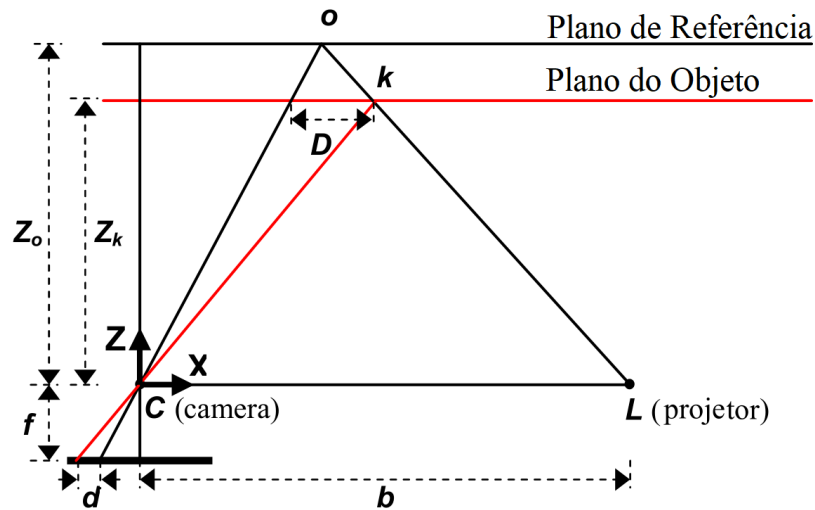


Figura 12 – Exemplo de triangulação ativa por luz estruturada [6].

A Figura 12 ilustra a relação entre a distância de um ponto k e o sensor. Assumindo que um objeto o está no plano de referência a uma distância Z_o e um ponto do padrão incide sobre o objeto. Se esse ponto do objeto (ponto k) estiver fora do plano de referência, haverá uma diferença entre o padrão calibrado e o atual, que será manifestado através de uma mudança da coordenada X do ponto do padrão. A partir da similaridade dos triângulos tem-se:

$$\frac{D}{b} = \frac{Z_o - Z_k}{Z_o} \quad (2.22)$$

$$\frac{d}{f} = \frac{D}{Z_k} \quad (2.23)$$

Onde Z_k é a profundidade do objeto, b é a *baseline*, f é a distância focal, D é o deslocamento em X do ponto do padrão em relação a sua posição no plano de referência

e d é a disparidade. Substituindo 2.23 em 2.22, tem-se que a profundidade é:

$$Z_k = \frac{Z_o}{1 + d \frac{Z_o}{f_b}} \quad (2.24)$$

Na Figura 13 pode-se observar um exemplo de uma imagem capturada por um sensor que utiliza luz estruturada, e o mapa de disparidade resultante. Uma das vantagens deste método em relação ao anterior é que na estereoscopia é estritamente necessário que a cena possua características suficientes para ser reconstruída. Isso é um empecilho para para reconstruir alguns objetos, como uma mesa lisa. O método que utiliza a luz estruturada não sofre com isso, pois ele cria as características através da luz projetada.

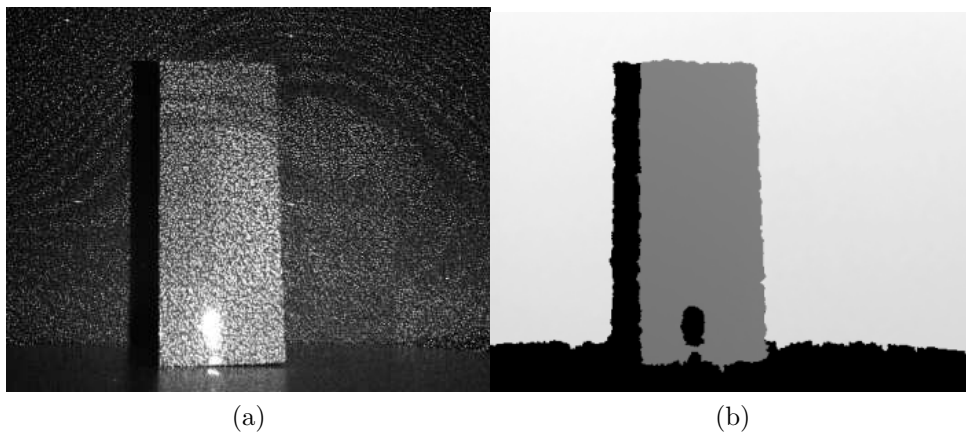


Figura 13 – Exemplo da imagem capturada por um sensor que utiliza luz estruturada [6].
(a) Luz estruturada. (b) Mapa de disparidade.

2.3 Detecção de *features*

Features, *keypoints* ou características, são padrões na imagem que se diferem de seus vizinhos locais em alguma propriedade, como intensidade, cor, textura, etc [16]. Essas *features* podem ser pontos, quinas ou *blobs* (segmento de imagem). As *features* podem ser divididas em três categorias: as que possuem uma interpretação semântica, como *blobs* em um sistema de inspeção de malha que podem representar furos; as que não tem interpretação semântica, sua interpretação não é importante, mas sim a sua estabilidade e precisão para ser encontrada em imagens subsequentes; e as que possuem interpretação semântica se agrupadas (para o reconhecimento de cenas e objetos). A reconstrução 3D é um caso onde a interpretação semântica das *features* não é importante, entretanto, o algoritmo detector de *features* deve ser robusto o suficiente para encontrar a mesma *feature* em imagens diferentes. Segundo [5] e [16] um bom detector de *features* deve apresentar as seguintes propriedades:

- Repetibilidade: dada duas imagens da mesma cena, capturadas de em ângulos diferentes, o detector deve ser capaz de encontrar as mesmas *features* que estão presentes em ambas as imagens.
- Distintividade: as informações utilizadas para descrever as *features* devem variar significativamente para que elas possam ser distinguidas e para que a correspondência entre a mesma *feature* em múltiplas imagens seja confiável.
- Localidade: as *features* devem ser locais, para reduzir a possibilidade de oclusão e para simplificar o modelo de aproximação das deformações geométricas entre imagens tiradas de ângulos diferentes.
- Precisão na localização: as *features* devem ter sua posição e escala bem definidas, uma vez que erros nessa localização são propagados para o posicionamento das câmeras.
- Quantidade de *features*: para o posicionamento das câmeras é importante ter uma grande quantidade de *features*, pois isso permite a escolha de mais pares para a minimização do erro utilizando algoritmos como o RANSAC.
- Eficiência computacional: muitas vezes não se tem a sequência correta das câmeras, sendo necessário fazer o *matching* de cada imagem com todas as outras do dataset. Isso pode se tornar um problema se o detector não conseguir descrever de forma eficiente as *features*.

A repetibilidade é provavelmente a propriedade mais importante de um detector de features, e pode ser atingida de duas formas [16]:

- Invariância: quando grandes deformações são esperadas, o detector deve ser desenvolvido de forma a se tornar invariante a essas deformações.
- Robustez: para casos onde as deformações não são significativas, é preferível que o detector seja robusto a ruídos, efeitos da discretização, compressão, borrões, etc.

No caso da reconstrução 3D, como as imagens a serem analisadas vão vir de diferentes ângulos de visão, é esperado que as imagens apresentem transformações como rotações, translações e mudanças de escala. Por isso, deve-se escolher um detector de *features* do tipo *blobs*, pois apesar deles serem menos precisos em localização se comparados a um detector de quinas, são mais robustos a mudanças de escala e de formato [16]. Na Figura 14 pode-se observar o problema de detecção de quinas em imagens com diferentes escalas. Na esquerda tem-se a imagem de uma quina com uma *feature* representada pelo quadrado vermelho. Na direita tem-se uma segunda imagem da mesma quina com uma mudança de escala. Pode-se perceber que o mesmo detector de *features* foi utilizado, mas

por não ser invariante a escala ele não consegue encontrar a mesma *feature* da primeira imagem. Abaixo serão apresentados alguns métodos de detecção de *features* do tipo *blobs*.

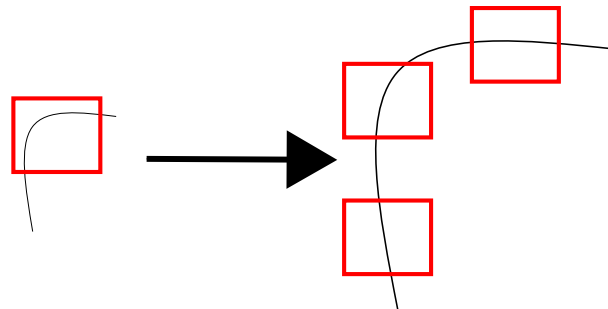


Figura 14 – Falha de correspondência com detectores de quinas e mudança de escala.

2.3.1 SIFT

Scale Invariant Feature Transform (SIFT), é um detector de *blobs* desenvolvido por Lowe em 1999 [17]. Ele é invariante a rotação e escala, e parcialmente invariante à mudanças em iluminação e no ponto de vista da câmera. Sua grande repetibilidade e alta taxa de *matching* fez dele um dos melhores detectores de *features* desenvolvidos. De forma didática, o algoritmo pode ser dividido em 4 passos principais [5] [7]:

- Detecção de extremos no espaço de escala: O primeiro estágio consiste na procura dos pontos de interesse por toda a imagem em todas as escalas. Isso é feito através da função da diferença gaussiana.
- Localização dos *Keypoints*: Para cada candidato de ponto de interesse, um modelo é aproximado para definir a localização e a escala. Os *keypoints* são definidos com base em sua estabilidade.
- Definição de orientação: Uma ou mais orientações são definidas para cada *keypoint* baseado em seu gradiente local. As futuras operações serão feitas com os dados já transformados, fazendo elas invariantes a orientação, escala e localização de cada *feature*.
- Descritor do *keypoint*: Os gradientes locais são medidos a uma determinada escala ao redor de cada *keypoint*. Isso cria uma representação que permite mudanças significativas em formato e iluminação.

O primeiro passo do algoritmo é a geração das imagens de diferença gaussiana (*Difference of Gaussian* - DoG). Para cada oitava do espaço de escala, a imagem inicial é repetidamente convoluída com uma gaussiana. As imagens gaussianas adjacentes são subtraídas para produzir o conjunto de imagens de diferença gaussiana. Após cada oitava,

a imagem original é reduzida por um fator de 2 e o processo se repete, esse processo pode ser observado na Figura 15. A próxima etapa é a seleção dos *keypoints*, isso é feito através da identificação dos mínimos e máximos locais das imagens de diferença gaussiana através das diferentes escalas. Cada pixel das imagens de diferença gaussiana é comparado com seus oito vizinhos na mesma escala, e com os nove vizinhos em escalas adjacentes. Se o pixel for um mínimo ou um máximo local ele é um candidato a *keypoint*, esse processo pode ser observado na Figura 16 [7] [5].

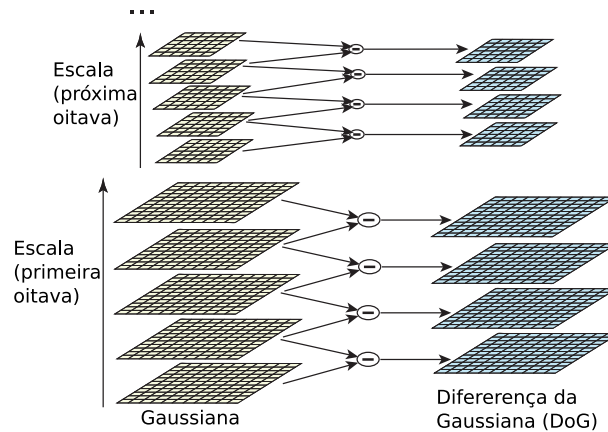


Figura 15 – Primeira etapa do algoritmo SIFT (Adaptado de [7]).

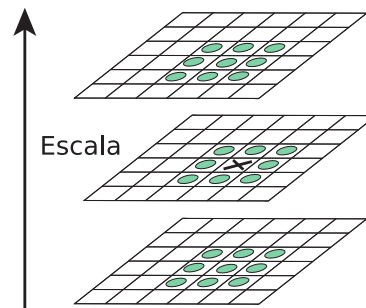


Figura 16 – Segunda etapa do algoritmo SIFT (Adaptado de [7]).

O próximo passo consiste em refinar a localização, escala e relação de curvatura principal dos *keypoints* através da interpolação de dados próximos. Isso permite que *keypoints* com baixo contraste ou em bordas sejam removidos por sua baixa distinção. Após esse refinamento deve-se definir uma orientação para cada *keypoint* baseada em propriedades da imagem, de modo a fazer com que o *keypoint* seja invariante a rotação. Para cada pixel na vizinhança dos *keypoints* é calculado o gradiente de intensidade (magnitude e orientação). Após isso um histograma de orientações é formado com o gradiente de orientações dos pixels ao redor do *keypoint*, esse histograma é construído de forma que a contribuição de cada pixel seja proporcional a magnitude do gradiente. Picos no histograma correspondem a direções dominantes dos gradientes locais, quando o maior pico é localizado, os picos que estiverem com 80% desse valor serão utilizados para criar outros *keypoints* com

mesma posição, mas com orientações diferentes. Esses *keypoints* com mesma localização e diferentes orientações contribuem para a estabilidade do *matching*. O histograma pode ser observado na Figura 17a [7] [5].

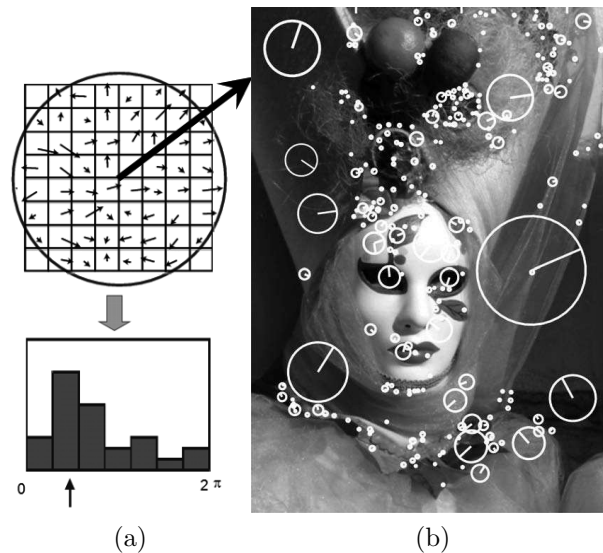


Figura 17 – Exemplo do algoritmo SIFT [5]. (a) Definição de orientação. (b) Algumas *features* do SIFT com sua direção e escala.

O próximo passo é criar o descritor do *keypoint*, que seja distinguível e invariante à iluminação e à mudanças do ponto de vista da câmera. O descritor é baseado em histogramas do gradiente de orientação. Para o descritor ser invariante a rotação, o gradiente das orientações deve ser rotacionado relativamente a orientação do *keypoint*. A vizinhança do *keypoint* é então dividida em sub-regiões de tamanho 4x4, e o histograma do gradiente com oito subdivisões de orientação é calculado para cada uma dessas regiões. Por fim, o descritor é criado agrupando todas as entradas dos histogramas de orientação. O tamanho final do descritor é $4 \times 4 \times 8 = 128$ elementos. Para adquirir a invariância parcial à iluminação o vetor do descritor é normalizado. Um exemplo do algoritmo pode ser visualizado na Figura 17b [7] [5].

2.3.2 SURF

Speeded-Up Robust Features (SURF), é um detector de *blobs* desenvolvido por Bay, H., Tuytelaars, T. e Van Gool, L. em 2006 [18]. Este detector de *features* invariante a escala é fortemente inspirado no SIFT, mas é muito mais rápido. Ele utiliza uma aproximação da matriz Hessiana através de um conjunto de filtros quadrados e utiliza imagens integrais para as convoluções, o que faz o processo ficar muito mais rápido que o SIFT com uma pequena redução na robustez. Imagens integrais são úteis para o cálculo da soma de intensidades de áreas retangulares, principalmente porque o tempo de cálculo é independente do tamanho da área. A entrada de uma imagem integral $I_{\Sigma}(\mathbf{x})$ em

$\mathbf{x} = (x, y)$ representa a soma de todos os pixels de I da origem até o ponto \mathbf{x} , isso pode ser visto na Equação 2.25 [5] [16].

$$I(\mathbf{x}) = \sum_{i=0}^{x} \sum_{j=0}^{y} I(i, j) \quad (2.25)$$

A identificação das *features* é feita através da identificação de pontos máximos no determinante de uma aproximação da matriz Hessiana. Dado um ponto $\mathbf{x} = (x, y)$ em uma imagem I , a matriz Hessiana $Hess(\mathbf{x}, \sigma)$ em \mathbf{x} a uma escala σ é dada pela equação 2.26 [8].

$$Hess(\mathbf{x}, \sigma) = \begin{bmatrix} L_{xx}(\mathbf{x}, \sigma) & L_{xy}(\mathbf{x}, \sigma) \\ L_{xy}(\mathbf{x}, \sigma) & L_{yy}(\mathbf{x}, \sigma) \end{bmatrix} \quad (2.26)$$

Onde $L_{xx}(\mathbf{x}, \sigma)$ é uma convolução da derivada de segunda ordem da Gaussiana $\frac{\partial^2}{\partial x^2} g(\sigma)$ com a imagem I no ponto \mathbf{x} . Apesar das Gaussianas serem ótimas para a análise no espaço escala, na prática elas devem ser discretizadas, o que diminui sua eficiência em múltiplos ímpares de $\frac{\pi}{4}$. Como essa já é uma aproximação, o algoritmo SURF vai além e simplifica essa discretização da derivada de segunda ordem da Gaussiana para uma aproximação com filtros quadrados, que são mais eficientes de serem calculados, e como são usadas imagens integrais o tempo de cálculo independe do tamanho do filtro. Na Figura 18 pode-se observar a derivada parcial de segunda ordem da Gaussiana e os filtros 9x9 utilizados para aproximar a Gaussiana com $\sigma = 1.2$. Esses filtros quadrados podem ser denotados como D_{xx} , D_{yy} e D_{xy} . O determinante da Hessiana aproximada, dado pela Equação 2.27, representa a resposta *blob* da imagem na posição \mathbf{x} . Essas respostas são guardadas em um mapa com várias escalas para a detecção dos pontos máximos [8].

$$\det(Hess_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2 \quad (2.27)$$

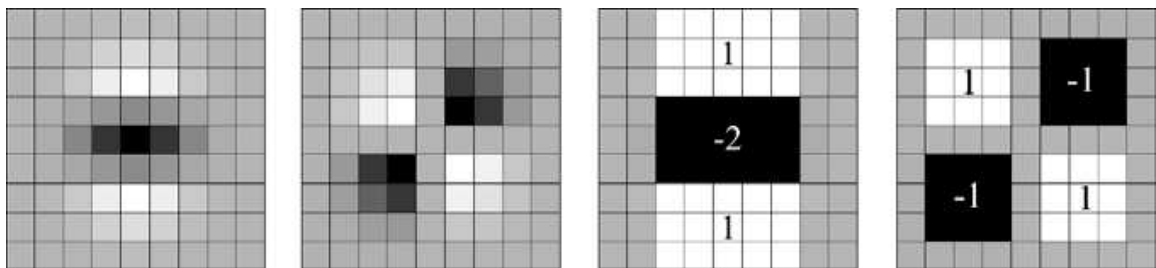


Figura 18 – Da esquerda para a direita, a derivada parcial de segunda ordem da Gaussiana na direção y (L_{yy}) e na direção xy (L_{xy}), a aproximação da derivada parcial de segunda ordem da Gaussiana na direção y (D_{yy}) e na direção xy (D_{xy}) [8].

Como o SURF é um detector invariante na escala, ele deve analisar diferentes escalas para encontrar as *features* novamente, ao contrário do SIFT, que diminui a imagem por um fator de 2 a cada oitava, o SURF aumenta o tamanho do filtro. A principal vantagem dessa abordagem é a eficiência computacional, pois várias escalas podem ser analisadas

em paralelo, já que utilizam a mesma imagem de entrada. Por não existir o redimensionamento da imagem, esse algoritmo não sofre com *aliasing*. O SURF cria o descritor da *feature* de forma similar ao SIFT. O primeiro passo é a definição da orientação, isso é feito através do cálculo da transformada de Haar nas direções x e y em uma vizinhança circular, ao redor do ponto de interesse, com raio de $6s$, sendo s a escala onde o ponto de interesse foi detectado. Assim que as transformadas são calculadas e ajustadas com uma Gaussiana de $\sigma = 2$. As respostas são representadas como pontos no espaço sendo a resposta horizontal representada no eixo das abscissas e a resposta vertical no eixo das ordenadas. A orientação dominante é calculada através da soma de uma janela deslizante de tamanho $\frac{\pi}{3}$, o maior vetor resultante da soma define a orientação do ponto de interesse, como pode ser visto na Figura 19 [8].

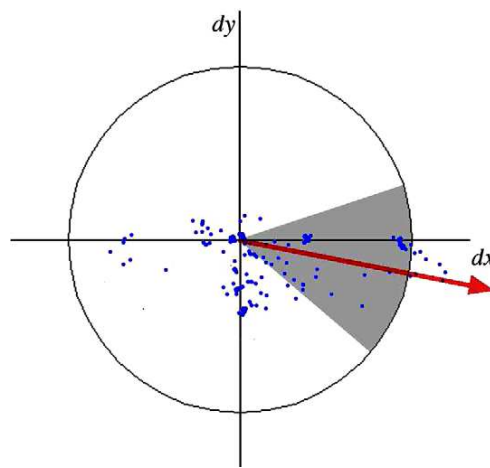


Figura 19 – Definição de orientação através de uma janela deslizante de tamanho $\frac{\pi}{3}$ [8].

Para extrair o descritor, o primeiro passo é construir uma região quadrada, de tamanho $20s$, com centro no ponto de interesse e orientada na direção encontrada na etapa anterior. Essa região é dividida em 4 sub-regiões da mesma forma que no algoritmo SIFT, para cada sub-região as transformadas de Haar horizontal (d_x) e vertical (d_y) são calculadas, para aumentar a robustez do algoritmo a deformações geométricas e erros de localização, uma Gaussiana de $\sigma = 3.3s$ é aplicada. Após isso a resposta das transformadas d_x e d_y são somadas para cada sub-região, a soma dos valores absolutos $|d_x|$ e $|d_y|$ também são considerados para carregar a informação sobre mudanças na polaridade das intensidades. Dessa forma, cada sub-região possui um vetor descritor 4D $\mathbf{v} = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$, concatenando os vetores de todas as sub-regiões tem-se um vetor descritor com 64 elementos, esse processo pode ser visualizado na Figura 20 [8].

2.3.3 AKAZE

Accelerated-KAZE (AKAZE), é um detector de *blobs* desenvolvido por Alcantarilla, P., Nuevo, J. e Bartoli, A. em 2013 [9]. Este detector de *features* invariante a escala

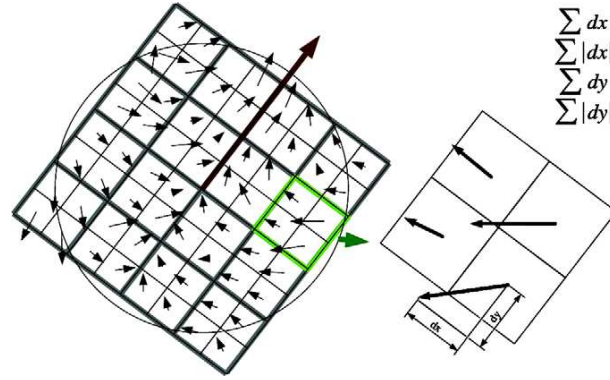


Figura 20 – Construção do descritor do algoritmo SURF [8].

tem com o objetivo superar as desvantagens de outros detectores que utilizam o espaço de escala Gaussiano, construído através de uma forma piramidal como no caso do SIFT ou através da aproximação das derivadas Gaussianas através de filtros quadrados como no caso do SURF. O problema do espaço de escala Gaussiano é que ele não preserva as bordas dos objetos e suaviza na mesma proporção detalhes e ruídos, o que prejudica a distintividade e a precisão na localização das *features*. O AKAZE tenta superar esse problema detectando e descrevendo as *features* em um espaço de escala não-linear, isso faz com que a suavização seja adaptada localmente, suavizando pequenos detalhes mas preservando as bordas dos objetos [9].

O termo *Accelerated* que diferencia o AKAZE do detector KAZE [19], está ligado ao fato do KAZE ter um maior custo computacional, pois como não existem meios analíticos de se resolver equações não-lineares de difusão e é necessário utilizar métodos numéricos. O KAZE utiliza o método *Additive Operator Splitting* (AOS), que apesar de ser estável para qualquer incremento de tempo, requer o cálculo de um grande sistema de equações lineares para obter uma solução. O AKAZE tenta resolver o problema da lentidão do KAZE com a introdução do algoritmo *Fast Explicit Diffusion* (FED), que tem um menor custo computacional e traz resultados mais precisos que o AOS [9].

Métodos FED são construídos a partir da decomposição de filtros quadrados em termos de métodos explícitos. Filtros quadrados iterados (*Iterated box filters*) aproximam Gaussianas com uma boa qualidade e são fáceis de serem implementados. A ideia principal é realizar M ciclos de n incrementos de difusão explícitas, com incrementos de tamanhos τ_j variados originados da fatoração do filtro quadrado [9]:

$$\tau_j = \frac{\tau_{\max}}{2 \cos^2\left(\pi \frac{2j+1}{4n+2}\right)} \quad (2.28)$$

Onde τ_{\max} é o incremento máximo que não viola a condição de estabilidade do método explícito. Alguns incrementos calculados através da Equação 2.28 podem violar a condição de estabilidade, mas pela similaridade entre o FED e os filtros quadrados, sempre se obtém um resultado estável no fim de cada ciclo [9].

O primeiro passo para poder realizar a detecção das *features* é a construção do espaço de escala não linear, que irá gerar as imagens filtradas L^i de onde serão extraídas os pontos de interesse. O espaço de escala é discretizado em uma série de O oitavas e S sub-níveis, o conjunto de oitavas e sub-níveis são identificados pelos índices o e s , respectivamente. Os índices das oitavas e dos sub-níveis são mapeados para sua correspondente escala σ através da seguinte fórmula [9]:

$$\sigma_i(o, s) = 2^{o+s/S}, o \in [0 \dots O - 1], s \in [0 \dots S - 1], i \in [0 \dots M] \quad (2.29)$$

Onde M é o número total de imagens filtradas. Quando se chega no último sub-nível de cada oitava a imagem é diminuída por um fator de 2 utilizando uma máscara de suavização, e essa nova imagem é utilizada como entrada para o processo da próxima oitava. Com o espaço de escala não linear construído, pode-se seguir para a detecção das *features*, deve-se calcular o determinante da matriz Hessiana para todas as imagens L^i no espaço de escala não linear. O conjunto de operadores diferenciais multi-escala são normalizados em escala, utilizando um fator de escala normalizado que leva em consideração a oitava de cada imagem L^i no espaço de escala não linear, $\sigma_{i,norm} = \sigma_i/2^{o^i}$, de forma que [9]:

$$L_{Hessiana}^i = \sigma_{i,norm}^2 (L_{xx}^i L_{yy}^i - L_{xy}^i L_{xy}^i) \quad (2.30)$$

Para encontrar os pontos de interesse, a cada nível i é checada se a resposta do detector é maior que um limite pré-definido e se é maior dentro de uma janela de 3x3 pixels. Para cada máximo em potencial, é checado se ele ainda é máximo em comparação a outros máximos em potencial dos níveis $i + 1$ e $i - 1$, em uma janela de $\sigma_i x \sigma_i$ pixels. Por fim, a posição 2D do *keypoint* é estimada com precisão sub-pixel encontrando o ponto máximo da aproximação de uma função 2D quadrática na resposta do determinante da Hessiana em uma vizinhança de 3x3 [9].

Para a descrição da *feature* é utilizado o *Modified-Local Difference Binary* (M-LDB), que utiliza informações do gradiente e da intensidade do espaço de escala não linear. O LDB [20] é um descritor do tipo binário que divide a imagem em *grids* de tamanho $n \times n$, e faz testes binários entre um par de células desse *grid*. Os testes feitos são: a média de intensidade, o gradiente de primeira ordem em x e o gradiente de primeira ordem em y. A invariância a rotação é obtida encontrando a orientação do *keypoint* e rotacionando o *grid* do LDB. Ao invés de calcular a média de intensidade dos pixels dentro de cada célula do *grid*, o M-LDB faz amostras do *grid* em incrementos que são uma função da escala σ_i da *feature*. Essa amostragem que varia com a escala faz o descritor robusto a mudanças de escala. A diferença entre o LDB e o M-LDB pode ser observada na Figura 21 [9].

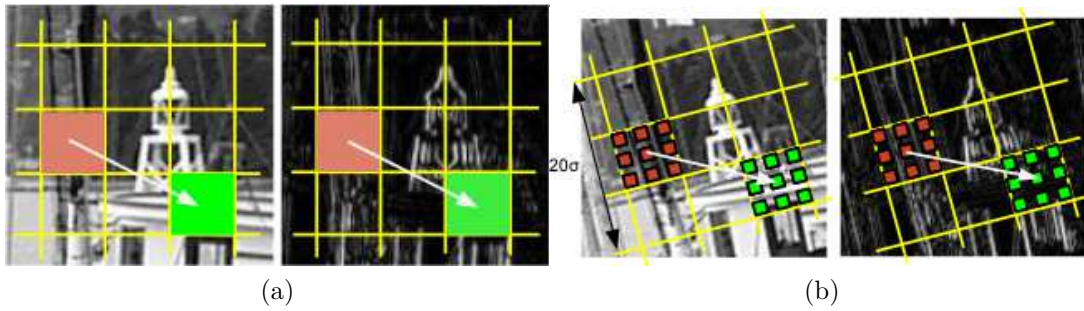


Figura 21 – Exemplo de testes binários com os descritores LDB e M-LDB [9]. (a) LDB. (b) M-LDB.

2.4 Matching

Após a descrição das *features* em duas ou mais imagens, é necessário que seja feita a *matching* (ou correspondência) entre elas. Nesse processo é assumido que o descritor de *features* foi desenvolvido de maneira que a distância euclidiana entre as *features* pode ser utilizada diretamente para fazer a classificação de potenciais *matchings*. O método mais básico de *matching* utilizando a distância euclidiana é definindo um limite e retornando todos os *matches* de outras imagens que estejam dentro desse limite. O problema dessa abordagem é que definindo o limite muito alto, irá causar um grande número de falsos positivos, e definindo um limite muito baixo, um grande número de falsos negativos. Segundo [3] e [21], qualidade de um algoritmo de *matching* pode ser avaliada utilizando o seguinte conjunto de definições:

- Verdadeiro positivo (VP): número de *matches* corretas.
- Falso negativo (FN): *matches* corretas que não foram detectadas.
- Falso positivo (FP): *matches* identificadas que são incorretas.
- Verdadeiro negativo (VN): não-*matches* que foram corretamente rejeitadas.

A partir desses definições pode-se definir os seguintes indicadores de qualidade [3] [21]:

- Taxa de verdadeiros positivos (TVP)

$$TCP = \frac{VP}{VP + FN} = \frac{VP}{P} \quad (2.31)$$

- Taxa de falsos positivos (TFP)

$$TFP = \frac{FP}{FP + VN} = \frac{FP}{N} \quad (2.32)$$

- Valor de positivos predito (VPP)

$$VPP = \frac{VP}{VP + FP} = \frac{VP}{P'} \quad (2.33)$$

- Precisão (PREC)

$$PREC = \frac{VP + VN}{P + N} \quad (2.34)$$

Onde $P = VP + FN$, $P' = VP + FP$ e $N = FP + VN$. A qualidade de um algoritmo de *matching* pode ser avaliada pelas taxas TVP e TFP. Um algoritmo de *matching* ideal teria a TVP igual a 1 e a TFP igual a 0. Variando o limite de distância definido, é possível gerar uma curva chamada de *receiver operating characteristic* (ROC), Figura 22a. O quanto mais próximo essa curva está do canto esquerdo superior, melhor é o desempenho do *matching*. Na Figura 22b pode ser observado como pode-se relacionar o número de *matches* e não-*matches* em função da distância d entre as *features* [3].

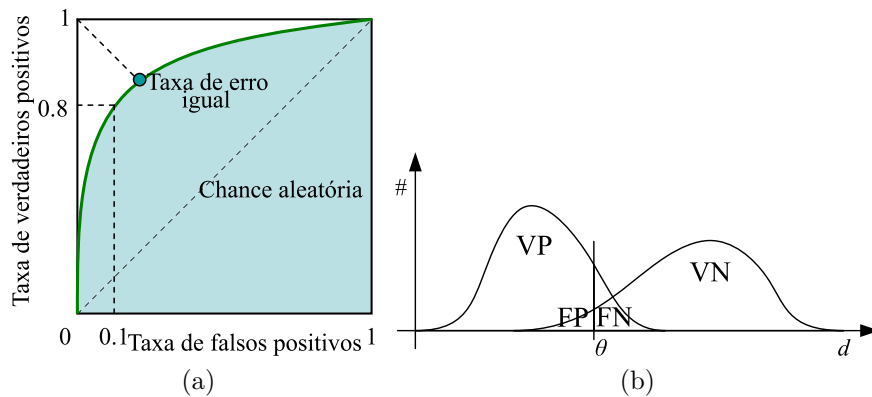


Figura 22 – Curva ROC (Adaptado de [3]). (a) A curva ROC, relacionando as taxas de verdadeiros positivos e falsos positivos. (b) A distribuição das *matches* e não-*matches* em função da distância d entre as *features* e o limite θ .

O problema de utilizar este método de limite de distância é que o limite que seria correto para uma região não necessariamente é correto para outra, variando muito entre as *features*. Uma estratégia mais adequada seria utilizar o método do vizinho mais próximo, como algumas *features* podem não ter nenhum *match*, um limite ainda é aplicado para diminuir a quantidade de falsos positivos. Idealmente esse método iria se adaptar a diferentes regiões no espaço das *features*, mas ele deve ter uma quantidade suficiente de dados para ser treinado, algo que não está disponível na aplicação de reconstrução 3D. Desta maneira, uma melhor forma de abordar o problema é levar em consideração a distância ao segundo vizinho mais próximo. Pode-se definir a relação entre a distância do vizinho mais próximo (*nearest neighbor distance ratio*) através da equação [3]:

$$NNDR = \frac{d_1}{d_2} = \frac{|D_A - D_B|}{|D_A - D_C|} \quad (2.35)$$

Onde d_1 e d_2 são o vizinho mais próximo e o segundo vizinho mais próximo, respectivamente, D_A é o descritor que está sendo analisado e D_B e D_C os descritores dos seus vizinhos mais próximos. Na Figura 23 pode-se observar os três casos de *matching*. A linha pontilhada representa o caso do *matching* pelo limite de distância, pode-se observar

que o método falha em relacionar os pontos D_A e D_B , e relaciona incorretamente D_D com D_C e D_E . No método do vizinho mais próximo, D_A é corretamente relacionado com D_B , mas D_D é incorretamente relacionado com D_C . Usando o NNDR, D_A continua a ser relacionado com D_B , mas pelo grande valor de NNDR não há nenhuma correspondência para D_D [3].

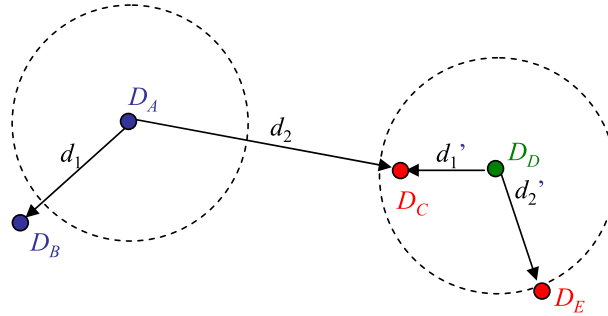


Figura 23 – Exemplo dos métodos de *matching* [3].

Após a decisão do método de *matching*, ainda deve-se decidir qual método utilizar para a busca dos possíveis candidatos para cada *feature*. O método mais simples seria comparar cada *feature* com todas as outras *features*, mas esse é o método mais ineficiente e impraticável para algumas aplicações. Uma abordagem mais adequada para esse problema seria utilizar algum tipo de estrutura indexada, como uma árvore multi-dimensional ou uma tabela *hash*. Essas estruturas podem ser construídas para cada imagem, no caso de algoritmos de procura de objetos, ou podem ser construídas para todas as imagens, o que acelera o processo, pois eliminaria a necessidade de iterar sobre cada imagem [3].

As árvores de busca multi-dimensionais são uma das estruturas indexadas mais utilizadas para a busca dos possíveis candidatos para o processo de *matching*. As mais conhecidas são as *k-d trees*, as quais dividem o espaço de *features* multi-dimensional ao longo de alternados hiperplanos alinhados aos eixos, escolhendo o limite ao longo de cada eixo para maximizar algum critério. Na Figura 24a pode-se observar um exemplo de uma *k-d tree* de duas dimensões, oito diferentes pontos A-H são representados como pequenos diamantes. A *k-d tree* divide recursivamente esse plano utilizando cortes horizontais e verticais (linhas pontilhadas). Cada divisão pode ser nomeada usando a dimensão que o corte foi feito e o número da divisão (Figura 24b). As divisões são arranjadas para balancear a árvore (minimizar a sua profundidade). Uma *k-d tree* clássica primeiramente encontra o ponto em análise (*query point*) no seu apropriado *bin*, e após isso ela procura as folhas próximas, até que ela possa garantir que o vizinho mais próximo foi encontrado. O método de procura *best bin first* (BBF) procura *bins* na ordem de sua proximidade espacial ao *query point*, sendo esse método usualmente o mais eficiente [3].

Após encontrar os possíveis *matches*, e caso já se tenha a posição das câmeras, pode-se utilizar do erro de reprojeção para verificar quais os pontos que estão corretos ou não. Esse processo de selecionar uma certa quantidade de pontos do conjunto para verificar

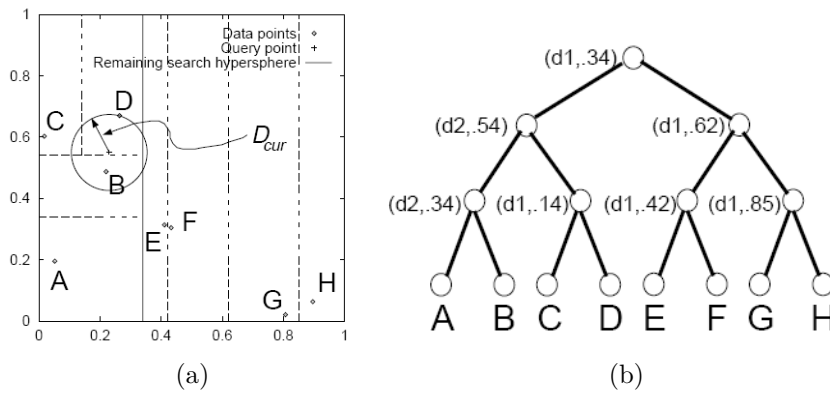


Figura 24 – (a) Exemplo de uma k - d tree [3]. (b) O método BBF para a k - d tree [3].

um conjunto maior, é chamado de amostragem randômica, ou RANSAC. Após verificar quais pontos estão corretamente alinhados, pode-se utilizar de técnicas como a geometria epipolar para encontrar mais pontos [3].

2.5 Bundle adjustment

Bundle adjustment é o problema de refinar uma reconstrução 3D otimizando de forma conjunta os pontos 3D e os parâmetros intrínsecos (distância focal, ponto principal, etc.) e extrínsecos (postura) das câmeras. O termo otimização vem do fato desses parâmetros estarem sendo estimados através da minimização de uma função de custo que quantifica o erro da reconstrução 3D. O *bundle adjustment* e problemas similares de ajustes são formulados como problemas de mínimos quadrados não lineares, nos quais a função de custo que quantifica o erro da reconstrução 3D é o erro de re-projeção. O erro de re-projeção é a distância euclidiana entre a posição em que o ponto 3D foi projetado na imagem e a posição da *feature* que originou esse ponto, esse erro pode ser visto na Figura 25. Onde P representa o ponto no espaço 3D, C e C' são as origens das câmeras, x e x' são as *features* que deram origem ao ponto P , x_p e x'_p são as projeções do ponto P no plano da imagem das câmeras e as linhas vermelhas pontilhadas representam o erro de re-projeção [22].

Considere x sendo um vetor de parâmetros e $f(x) = [f_1(x), \dots, f_k(x)]$ sendo um vetor de erros de re-projeção de uma reconstrução 3D. Então, o problema de otimização que se deseja solucionar é o problema de mínimos quadrados não lineares [23]:

$$x^* = \arg \min_x \sum_{i=1}^k \|f_i(x)\|^2 \quad (2.36)$$

O algoritmo mais utilizado para resolver este tipo de problema é o algoritmo *Levenberg-Marquardt* (LM), ele funciona solucionando uma série de aproximações lineares regularizadas do problema original. Seja $J(x)$ a matriz jacobiana de $f(x)$, então cada iteração do

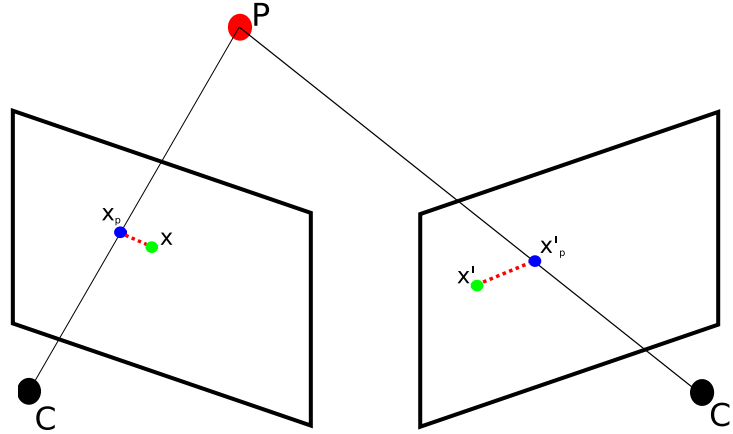


Figura 25 – Exemplo do erro de re-projeção.

LM irá resolver um problema de mínimos quadrados linear da forma [23] [24]:

$$\delta^* = \arg \min_{\delta} \|J(x)\delta + f(x)\|^2 + \lambda \|D(x)\delta\|^2 \quad (2.37)$$

Cada iteração irá atualizar o valor de x da forma $x \leftarrow x + \delta^*$ se $\|f(x + \delta^*)\| < \|f(x)\|$. Onde $D(x)$ é uma matriz diagonal não-negativa, tipicamente a raiz quadrada da diagonal da matriz $J(x)^\top J(x)$, e λ é um parâmetro não-negativo que controla a força da regularização. A regularização é necessária para garantir a convergência do algoritmo, o LM atualiza o valor de λ a cada iteração baseado em quão bem a jacobiana $J(x)$ aproxima $f(x)$. Resolver a Equação 2.37 é equivalente a resolver as equações normais [23]:

$$(J(x)^T J(x) + \lambda D(x)^T D(x))\delta = -J(x)^T f(x) \quad (2.38)$$

A matriz $H_\lambda = J(x)^T J(x) + \lambda D(x)^T D(x)$ é conhecida como a matriz Hessiana aumentada. Em problemas de *bundle adjustment* o vetor de parâmetros é tipicamente organizado como $x = [x_c; x_p]$, onde x_c é o vetor de parâmetros da câmera e x_p o vetor de parâmetros do ponto. Similarmente, D , δ e J com o subscrito c e p representam a parte da câmera e a parte do ponto respectivamente. Seja $U = J_c^T J_c$, $V = J_p^T J_p$, $U_\lambda = U + \lambda D_c^T D_c$, $V_\lambda = V + \lambda D_p^T D_p$ e $W = J_c^T J_p$, então, a Equação 2.38 pode ser reescrita da forma [23]:

$$\begin{bmatrix} U_\lambda & W \\ W^T & V_\lambda \end{bmatrix} \begin{bmatrix} \delta_c \\ \delta_p \end{bmatrix} = - \begin{bmatrix} J_c^T f \\ J_p^T f \end{bmatrix} \quad (2.39)$$

Onde U_λ e V_λ são matrizes diagonais em blocos, esta observação é útil pois permite utilizar o complemento de *Schur* para resolver este sistema linear eficientemente, onde, aplicando a eliminação Gaussiana nos parâmetros dos pontos obtém-se um sistema linear formado apenas pelos parâmetros das câmeras [23]:

$$(U_\lambda - W V_\lambda^{-1} W^T)\delta_c = -J_c^T f + W V_\lambda^{-1} J_p^T f \quad (2.40)$$

A matriz $S = U_\lambda - WV_\lambda^{-1}W^T$ é o complemento de *Schur*. Com a solução da Equação 2.40, o vetor de parâmetros dos pontos δ_p pode ser obtido da seguinte forma [23]:

$$\delta_p = -V_\lambda^{-1}(J_p^T f + W^T \delta_c) \quad (2.41)$$

Como S é simétrica e definida positivamente, a fatoração de *Cholesky* é o método escolhido para resolver a Equação 2.40, mas dependendo da estrutura da matriz tem-se duas opções. A primeira é a fatoração direta, onde se armazena e fatora S como uma matriz densa. Este método tem $O(p^2)$ complexidade de espaço e $O(p^3)$ complexidade de tempo, e é utilizável apenas para algumas centenas de câmeras. Mas como S é uma matriz esparsa, já que a maioria das câmeras vê uma pequena fração da cena, pode-se utilizar métodos esparsos diretos. Esses métodos armazenam S como uma matriz esparsa, usando a reordenação de colunas e linhas para maximizar a esparsidade da decomposição de *Cholesky* e focar o processamento nas partes não-nulas da fatoração [24].

O *bundle adjustment* encontra as posturas relativas das câmeras e as posições relativas dos pontos 3D, e não as posturas e posições absolutas. Isso porque ele é baseado no erro de re-projeção, sendo assim, se todas as posturas e pontos fossem trasladados em 1 m para a direita, o erro de re-projeção não seria alterado. Para evitar esse problema, pode-se fixar a posição de algumas câmeras e alguns pontos 3D, isso é útil quando se está em um processo muito grande de otimização, com várias câmeras. Normalmente após algumas iterações as câmeras iniciais não precisam mais ser modificadas, e devem ser mantidas fixas para acrescentar robustez as próximas iterações.

2.6 Geração de superfícies

A geração de superfície é uma etapa importante para o processo de reconstrução 3D, pois ela será responsável por definir como serão ligados os pontos da nuvem de pontos 3D adquirida na etapa anterior. A seguir serão apresentados 3 métodos para a reconstrução de nuvens de pontos 3D. Para uma simples comparação entre os métodos será utilizada a nuvem de pontos 3D presente na Figura 26, a qual possui 362,271 pontos e tem as normais corretamente orientadas. A comparação será feita na Seção 2.6.4 Comparação de métodos.

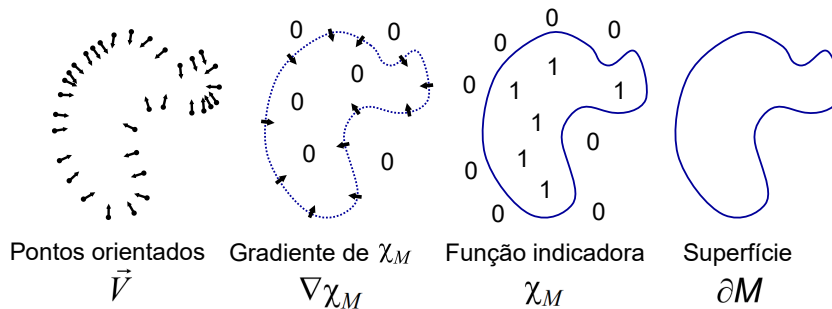
2.6.1 *Poisson*

O método de *Poisson* [10] resolve o problema da reconstrução da superfície através de uma função implícita. É calculada uma função indicadora χ , que assume valor 0 para os pontos fora do modelo e 1 para os pontos dentro do modelo, e com isso gera a superfície extraindo uma *isosurface* apropriada. A principal observação para esse método é que existe uma relação entre pontos orientados da superfície de um modelo e a função



Figura 26 – Exemplo de uma nuvem de pontos 3D.

indicadora do modelo. Essa relação é que o gradiente da função indicadora é um campo vetorial que é zero em todos os lugares exceto nas proximidades da superfície, onde ele assume os valores das normais que apontam para o interior da superfície. Deste modo, pode-se assumir que os pontos orientados são amostras do gradiente da função indicadora do modelo, como pode ser visto na Figura 27. Essa relação reduz o problema de calcular a função χ , para o problema de inverter o operador gradiente da função χ , ou seja, encontrar a função escalar χ cujo gradiente melhor aproxima um campo vetorial \vec{V} definido pelas amostras de pontos orientados ($\min_{\chi} \|\nabla_{\chi} - \vec{V}\|$). Aplicando o operador da divergência esse problema se transforma em um problema padrão de *Poisson*: calcular a função escalar χ cujo laplaciano (divergência do gradiente) se iguala à divergência do campo vetorial \vec{V} :

Figura 27 – Exemplo da reconstrução *Poisson* em 2D [10].

$$\Delta\chi \equiv \nabla \cdot \nabla\chi = \nabla \cdot \vec{V} \quad (2.42)$$

É necessário discretizar a função χ , a maneira mais simples seria utilizar um *grid* 3D convencional para tal tarefa, mas essa estrutura se torna muito custosa para gerar uma superfície rica em detalhes. Já que o *grid* teria muitas divisões para adquirir os detalhes, e essas divisões seriam um gasto de processamento desnecessário em regiões vazias. Para resolver este problema o método utiliza uma *octree* adaptativa, fazendo com que exista uma melhor detalhamento apenas perto dos pontos da superfície. O uso dessa *octree* adiciona um parâmetro de profundidade ao método, que permite o usuário definir o nível

de detalhe que será extraído da nuvem de pontos 3D. O aumento da profundidade vem com um aumento no tempo de execução, na quantidade de memória RAM consumida e na quantidade de triângulos do modelo final. Infelizmente este aumento não é proporcional, um incremento unitário na profundidade da *octree* faz o tempo, o consumo de memória e os triângulos do modelo final aumentarem em um fator de 4 [10]. Um exemplo dessa diferença na profundidade pode ser observado na Figura 28.



Figura 28 – Exemplo de variação de profundidade da *octree* no método *Poisson*. (a) Profundidade 6. (b) Profundidade 8. (c) Profundidade 10.

Uma vantagem deste método é o fato dele ser robusto a ruídos, amostragem não uniforme e *outliers* [25]. Uma desvantagem é a necessidade das normais devidamente orientadas, caso contrário o resultado pode ficar muito longe do esperado.

2.6.2 Greedy Projection Triangulation

Este método é baseado no princípio do crescimento incremental da superfície, que é a ideia de criar uma interpolação, ou aproximação, da superfície através de propriedades de superfície da nuvem de pontos 3D, que no caso do método citado seria o fato dele distinguir duas camadas de pontos através de um critério de distância. A superfície é iniciada criando um triângulo de início e o método continua a adicionar triângulos a superfície até que todos os pontos da nuvem de pontos 3D forem considerados, ou caso não existam mais triângulos válidos a serem adicionados. Se a segunda opção ocorrer o método reinicia a busca utilizando outro ponto que ainda não foi analisado. Para manter um histórico dos pontos que já foram analisados todos os pontos tem uma classificação durante a execução do algoritmo, podendo ser livre, margem, borda e completo. Inicialmente todos os pontos são livres, ou seja, eles não tem conexão com nenhum triângulo, quando todos os triângulos do ponto forem determinados ele é completo. Os pontos que estão na borda da superfície são pontos de margem ou de borda, a diferença é que pontos classificados como borda já foram analisados mas não podem ter mais triângulos por critérios de limite, já os pontos de margem ainda não foram analisados [26] [27] [11]. O algoritmo pode ser resumido nos seguintes passos [28]:

1. Procura dos vizinhos próximos: Para cada ponto p , uma vizinhança k é selecionada através da procura dos k vizinhos mais próximos do ponto dentro de uma esfera com raio $r = \mu \cdot d_0$, que se adapta a densidade local. Onde d_0 é a distância de p até seu vizinho mais próximo e μ é uma constante definida pelo usuário.
2. Projeção da vizinhança utilizando planos tangentes: A vizinhança é projetada em um plano que é aproximadamente tangencial a superfície formada pela vizinhança e o ponto p . Essa etapa é utilizada para a execução do critério de ângulo mínimo e máximo no próximo passo.
3. Seleção de pontos: Os pontos são selecionados por um critério de visibilidade, onde pontos que podem causar a auto-intersecção da superfície são eliminados, isso acontece caso seja conectado a p algum ponto que não é visível para ele. Os pontos não visíveis a p são: todos os pontos entre arestas de borda que estão ligadas ao ponto p (as linhas pontilhadas da Figura 29a), pontos que tem o ponto p fora de sua área de visão também são descartados (ponto V na Figura 29a) e pontos que estão separados do ponto p por um segmento da superfície também são eliminados (ponto branco na Figura 29a). Os pontos que sobram são conectados a p e a pontos consecutivos por arestas, formando triângulos que obedecem um critério de ângulo mínimo e máximo definido pelo usuário. Um exemplo desta seleção pode ser visualizado na Figura 29.

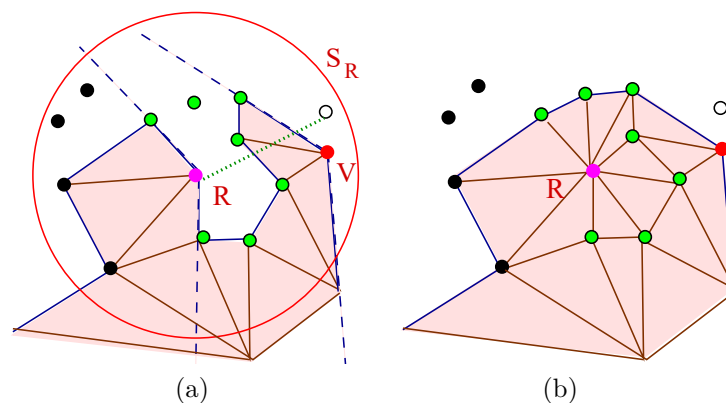


Figura 29 – Exemplo de um teste de visibilidade do *Greedy Projection Triangulation* para o ponto R [11].

Uma vantagem deste método é que ele não faz interpolações e mantém todos os pontos da nuvem de pontos 3D [28]. O método também é muito eficiente para calcular nuvens de pontos 3D planas, mas quando ela não é plana deve-se aumentar o número de vizinhos a serem considerados para gerar os triângulos.

2.6.3 Grid Projection

Ao invés de utilizar *isosurfaces* este método utiliza *Extremal surfaces*, que são superfícies implícitas definidas na extremidade de um campo escalar restringido por um campo direcional. A parte interessante do método utilizar este tipo de superfície implícita é que ela não precisa ser orientada, sendo assim o método funciona sem problemas com uma nuvem de pontos 3D sem normais. *Extremal surfaces* podem ser definidas da forma [29]:

$$S = \{x | x \in \text{arglocalmin}_{y \in l(x, n(x))} s(y)\} \quad (2.43)$$

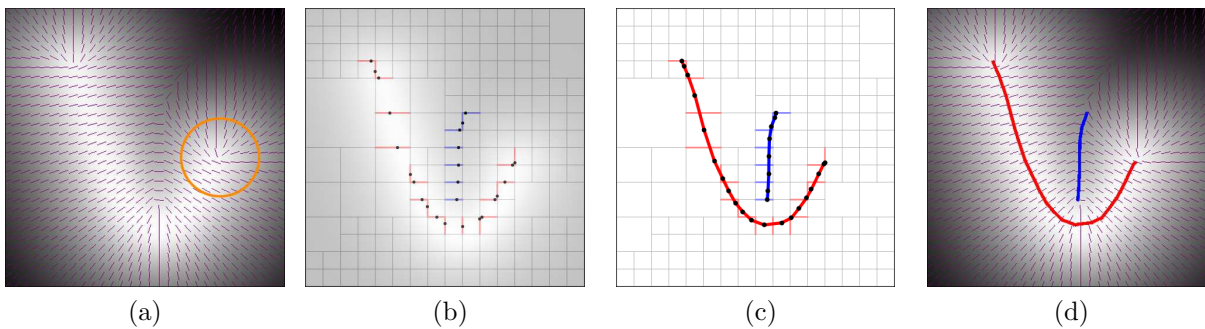


Figura 30 – Exemplo *Grid Projection*.

Onde $s : \mathbb{R}^d \rightarrow \mathbb{R}$ é uma função escalar, $n : \mathbb{R}^d \rightarrow \mathbb{R}\mathbb{P}^{d-1}$ é uma função vetorial não-orientada e $l(x, n(x))$ é a linha através de x com direção $n(x)$. Desta forma, pode-se dizer que S consiste em todos os pontos x em que a função é mínima ao longo da direção x . Um exemplo 2D pode ser visto na Figura 30a, $s(x)$ é mostrado em escala de cinza (valores maiores são mais claros), $n(x)$ são as linhas. Considere a superfície crítica feita de todos os pontos críticos de s (onde a derivada é zero) restringida a linha $l(x, n(x))$. Esses pontos críticos podem ser mínimos locais (o caso em que eles estão sob a *extremal surface*), máximos locais ou pontos de inflexão de s ao longo da linha $l(x, n(x))$. Desta forma, a *extremal surface* é um sub-conjunto da superfície crítica. A superfície crítica pode ser definida como o conjunto em que a função escalar abaixo é zero [29]:

$$g(x) = \vec{n}(x) \cdot \nabla s(x) \quad (2.44)$$

Uma visão geral do funcionamento do algoritmo pode ser vista da forma [29]:

1. O algoritmo recebe como entrada uma função escalar ($s(x)$) e uma função vetorial não orientada ($n(x)$), que podem ser vistas na Figura 30a.
2. É montado um *quadtrees grid* para a separação dos dados, um *quadtrees* é o análogo 2D da *octree*. Ele pode ser visto na Figura 30b.

3. São identificadas as arestas do *grid* que são cruzadas pela superfície crítica, essas arestas são chamadas de arestas críticas, que podem ser observadas em vermelho e azul na Figura 30b.
4. O sub-conjunto das arestas críticas que são intersectados pela *extremal surface* são chamados de *extremal edges*, que são as arestas coloridas em vermelho.
5. Por fim é criada uma superfície que cruza as arestas críticas, esta triangularização é feita utilizando o método *Dual Contouring* [30], ele adiciona vértices em todas as células do *grid* que contenham uma *extremal edge*, e cria um polígono para cada *extremal edge* conectando os vértices dentro das células que compartilham esta *extremal edge*. Essa etapa pode ser observada na Figura 30c e na Figura 30d tem-se a sobreposição da superfície com as funções de entrada.

A forma apresentada até o momento é uma visão genérica para o algoritmo, especificamente para nuvens de pontos 3D são utilizadas como entrada uma nuvem de pontos 3D e normais não-orientadas. Sendo $\{p_i, n_i\}$ o par de ponto e normal, a função vetorial $n(x)$ é definida em qualquer localização x como sendo o autovetor do seguinte tensor com o maior autovalor [29] [31]:

$$G(x) = \sum_{p_i} \theta(\|x - p_i\|) \langle n_i, n_i^T \rangle \quad (2.45)$$

Para a construção de s , é utilizada a soma de pesos Gaussianos [32]:

$$s(x) = - \sum_{p_i} \theta(\|x - p_i\|) \quad (2.46)$$

Esta função alcança o mínimo local perto dos pontos da nuvem de pontos 3D e aumenta de forma suave enquanto se afasta deles. Para o *grid* é utilizada uma *octree*, para se adaptar a densidade da nuvem de pontos 3D. O *grid* é subdividido recursivamente nos casos em que a célula possua mais que um ponto da nuvem, se dois vetores $n(x)$ nos cantos da célula diferem muito de direção, ou se a segunda derivada de s ao longo da linha $l(x, n(x))$ nos cantos da célula apresentem sinais diferentes. Estes critérios servem para garantir que o *grid* tenha resolução suficiente para considerar todos os pontos e para evitar variações da *extremal surface* dentro de uma mesma célula do *grid* [29].

2.6.4 Comparação dos métodos

Pode ser feita uma comparação qualitativa dos modelos resultantes na Figura 31. É notável a superioridade do método *Poisson*, ele consegue gerar uma superfície suave e com um ótimo nível de detalhes. O *Greedy Projection Triangulation* pela sua grande quantidade de parâmetros acaba dificultando seu ajuste, e não foi capaz de gerar um

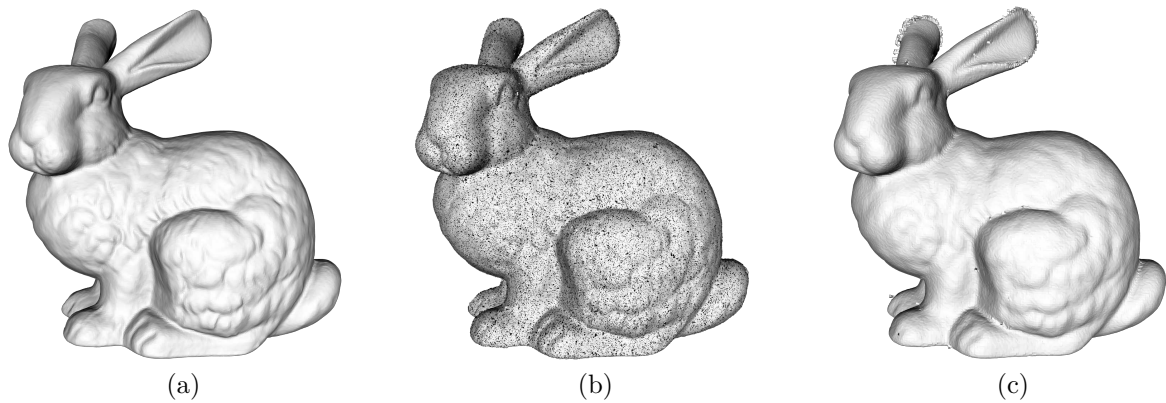


Figura 31 – Exemplo de reconstruções utilizando diferentes métodos. (a) *Poisson*. (b) *Greedy Projection Triangulation*. (c) *Grid Projection*.

modelo fechado da nuvem utilizada, além de ter criado várias faces invertidas (faces na cor preta, indicando que a normal está para o lado errado). O *Grid Projection* teve um bom resultado na maior parte da nuvem, mas acabou apresentando estruturas inexistentes na nuvem original, como os erros que apareceram na orelha do coelho. Uma comparação quantitativa entre os métodos é realizada em Masson e Petry [33].

2.7 Texturização do modelo 3D

Com o modelo já reconstruído é necessário texturizá-lo, para que ele se aproxime do objeto real. Apesar de ser possível adquirir a nuvem de pontos 3D colorida, a sua qualidade fica inferior à texturização já que as faces da superfície seriam coloridas utilizando uma média das cores de seus vértices, de forma que a qualidade do resultado dependa muito da densidade da nuvem de pontos, além da perda de pequenos detalhes. Por isso é preferível utilizar as imagens da câmera RGB. Para realizar o processo de texturização é necessário ter o modelo 3D, as imagens que foram tiradas do objeto e a posição e orientação da câmera em relação ao modelo 3D no momento em que as imagens foram capturadas. Como a postura das câmeras no momento da captura das imagens já é conhecida, o problema é simplificado, basta fazer a relação das faces da superfície 3D com os pixels das imagens. Um exemplo de um modelo 3D com as câmeras já posicionadas pode ser visto na Figura 32a.

A etapa inicial do processo é encontrar a posição na imagem de cada vértice da superfície 3D. Cada posição X, Y, Z pode ser projetada no plano da imagem utilizando a matriz P de projeção da câmera[34]. Tem-se que a matriz P é definida de tal forma [2]:

$$[R|t] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix}, K = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.47)$$

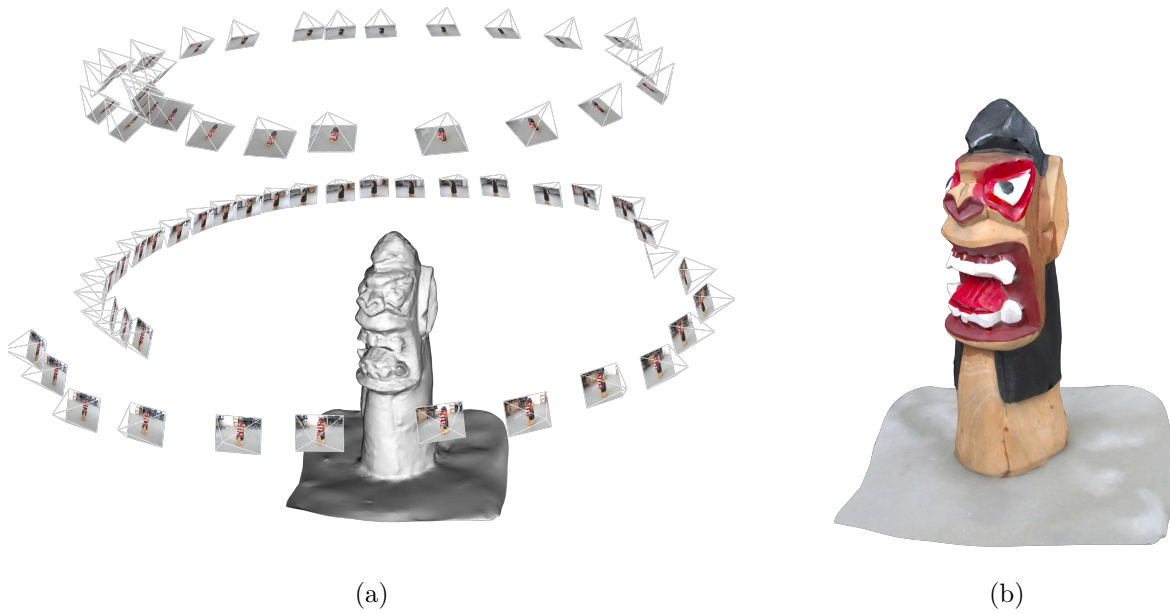


Figura 32 – Exemplo do processo de texturização. (a) Modelo 3D e as câmeras na posição da captura das imagens. (b) Modelo 3D texturizado.

$$P = K[R|t] \quad (2.48)$$

Sendo K a matriz de parâmetros intrínsecos da câmera contendo a distância focal f , R a matriz de rotação, e t o vetor de translação da câmera. Pode-se relacionar a coordenada de um ponto no espaço 3D com uma coordenada na imagem [2]:

$$C_c = \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, C_r = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}, C_c = PC_r \quad (2.49)$$

O vetor C_r representa a coordenada do ponto no espaço 3D e o vetor C_c a coordenada do ponto na imagem[2]. Para cada imagem o método testa quais faces da superfície estão visíveis para a câmera a qual pertence a imagem, então ele relaciona a textura com as faces que estão visíveis [35]. Após esse relacionamento, ainda é necessário fazer um tratamento da textura, pois não necessariamente faces vizinhas vão ser texturizadas a partir de uma mesma imagem. Um dos tratamentos que pode ser feito é o ajuste de cor local, isto é feito em [36] utilizando *Poisson Editing*. Esse método compara as texturas de faces vizinhas, e caso as texturas tenham uma diferença em iluminação ele cria uma transição suave entre as duas texturas. A texturização do modelo mostrado na Figura 32a pode ser vista na Figura 32b.

2.8 Trabalhos relacionados

Como foi visto na Seção 1.1, o processo de reconstrução 3D pode ser dividido em várias etapas. Desta forma, os trabalhos relacionados serão divididos em *softwares* que fazem todo o processo e *softwares*/bibliotecas que são focados em partes específicas do processo. O primeiro trabalho relacionado está ligado com o processo de estimação da postura das câmeras. O VisualSFM [37] é um *software* desenvolvido para a reconstrução 3D utilizando *structure from motion* (SFM), seu foco é em grandes conjuntos de imagens capturadas por diferentes câmeras. Uma de suas vantagens é a utilização da GPU nesse processo, o que acelera seu processamento. Seu detector de *features* e seu *bundle adjustment* tem seu código disponível na internet.

Partindo para a etapa de geração da nuvem densa tem-se o PMVS/CMVS [38], ele entra na categoria de geradores de nuvem densa através da propagação de correspondências. Ele funciona seguindo basicamente três passos: (1) Um processo de *matching* é realizado entre as várias imagens, o resultado é um conjunto de correspondências iniciais; (2) É realizada a expansão das correspondências iniciais, essa expansão é feita utilizando as restrições definidas pela geometria epipolar e algum tipo de *template matching*; (3) As correspondências incorretas são filtradas através de restrições de visibilidade. Uma vantagem desse método é a escalabilidade, pois ele divide grandes nuvens de pontos 3D em segmentos, para que exista memória RAM suficiente para seu processamento.

Para a etapa de geração da superfície tem-se o Screened Poisson Surface Reconstruction (PoissonRecon)[39]. O PoissonRecon é um *software* que gera uma reconstrução *Poisson* através de uma nuvem de pontos 3D com as normais calculadas, seu diferencial é que ela se beneficia do processamento *multithread*, o que acelera significativamente o cálculo da superfície.

Por fim, tem-se a etapa de texturização, para ela tem-se a biblioteca TexRecon [36]. O TexRecon é desenvolvido em C++ e tem como foco a texturização de modelos 3D de larga escala. Como ela é focada em modelos gerados por inúmeras imagens, tem diversos filtros para suavizar mudanças de iluminação locais e globais, borrões, diferentes escalas e permite a remoção de ruídos (como pedestres).

Outros *softwares*/bibliotecas realizam mais do que uma etapa do processo de reconstrução 3D. O MeshRecon [40], faz o processo de geração de nuvem densa e de geração de superfície, para isso ela calcula o mapa de profundidade para as imagens e utiliza uma integração volumétrica para fundir os mapas. Por ser totalmente implementado em GPU ele tem um baixo tempo de processamento, mas a resolução e quantidade das imagens é limitada pela quantidade de VRAM disponível. A *Point Cloud Library* (PCL) [41] é uma biblioteca desenvolvida em C++ especialmente para a manipulação de nuvens de pontos 3D, ela possui módulos de filtragem, segmentação, geração de superfícies, texturização, etc. Essa biblioteca foi escolhida pois possui implementações para os três métodos

<i>Software/ Biblioteca</i>	Estimação da postura das câmeras	Geração da nuvem densa	Geração da superfície	Texturização
VisualSFM	X			
PMVS/CMVS		X		
PCL			X	X
PoissonRecon			X	
TexRecon				X
MeshRecon		X	X	
CMP-MVS		X	X	X
Metashape	X	X	X	X
Meshroom	X	X	X	X
3Dflow	X	X	X	X

Tabela 1 – Etapas do processo de reconstrução 3D executadas pelos trabalhos relacionados

descritos na Seção 2.6 Geração de superfícies, *Poisson*, *Greedy Projection Triangulation* e *Grid Projection*, além de possuir um método de texturização. Apesar de possuir uma implementação do método de *Poisson*, esta foi descartada dos testes já que ela não é *multithread* como o PoissonRecon.

O CMP-MVS [42] é outro algoritmo que realiza mais que uma função, realizando os processos de geração da nuvem densa, geração da superfície e texturização. Um diferencial desse algoritmo é que ele tem como objetivo manter superfícies que são fracamente suportadas. Um exemplo seria uma garrafa transparente, que por ser transparente e ter muitos reflexos é difícil de ser distinguida de ruídos. O Meshroom [43] é outro *software* completo para a reconstrução 3D. Seu diferencial é que a biblioteca utilizada para implementar os métodos das etapas de reconstrução é de código aberto, permitindo os usuários modificarem os algoritmos.

Por fim, tem-se o Metashape [44] e o 3Dflow [45], ambos são *softwares* de código fechado que não explicitam os métodos utilizados. Na Tabela 1 pode-se ter uma visão geral de todos os trabalhos relacionados e quais etapas do processo de reconstrução 3D eles executam.

3 Abordagem proposta

Nesta seção serão apresentados os algoritmos desenvolvidos para a etapa de estimação da postura das câmeras e para a geração da nuvem densa. Essas etapas foram escolhidas para o desenvolvimento pois são as que mais se aproximaram do conteúdo exposto nas aulas de visão computacional, onde foram apresentados os conceitos de detecção de *features*, *matching*, *template matching*, geometria epipolar, etc. O desenvolvimento feito para as outras etapas consistiu apenas em algoritmos básicos que conectam os métodos das bibliotecas utilizadas, eles foram criados com base na documentação das bibliotecas, por isso eles não serão descritos aqui. A linguagem de programação escolhida para o desenvolvimento dos algoritmos foi a C++, principalmente pelo fato da maioria das bibliotecas utilizadas serem implementadas nesta linguagem. Acrescenta-se a isso o fato da documentação em C++ ser em geral mais extensa, e a familiaridade do autor deste trabalho. Como ambiente de desenvolvimento foi escolhido o Visual Studio Community 2015¹ por ser um ambiente de desenvolvimento robusto, que a maioria das bibliotecas disponibiliza binários pré-compilados e pela familiaridade com o ambiente.

Algumas limitações foram impostas para simplificar algumas etapas do desenvolvimento do processo de estimação da postura das câmeras. Definiu-se que as imagens de entrada estariam na sequência em que foram tiradas, e essa sequência seguiria um padrão circular ao redor do objeto, como pode ser visto na Figura 33. O círculo maior no centro da figura representa o objeto, enquanto os círculos menores com as setas representam as câmeras e suas direções, e as setas vermelhas indicam o sentido horário/anti-horário. A escolha da câmera inicial não tem importância, desde que o processo siga no sentido horário/anti-horário de forma sequencial, sem pular nenhuma câmera. O número de câmeras na figura é apenas representativo, a quantidade de imagens necessárias varia com o tamanho do objeto e o nível de detalhamento desejado. Como foi citado na introdução, as imagens devem ter uma sobreposição de 60% a 90%, desta maneira, a câmera não pode sofrer uma modificação de postura muito brusca entre as capturas.

3.1 Estimação da postura das câmeras

Para o desenvolvimento da etapa de estimação da postura das câmeras se faz necessária a escolha de um método para a detecção de *features*, *matching* de *features* e um método para o *bundle adjustment*. Para a detecção de *features* foram utilizadas duas abordagens. Uma abordagem em CPU utilizando o OpenCV² com os detectores AKAZE e SIFT, e

¹ <<https://visualstudio.microsoft.com/pt-br/vs/older-downloads>>

² <<https://opencv.org>>

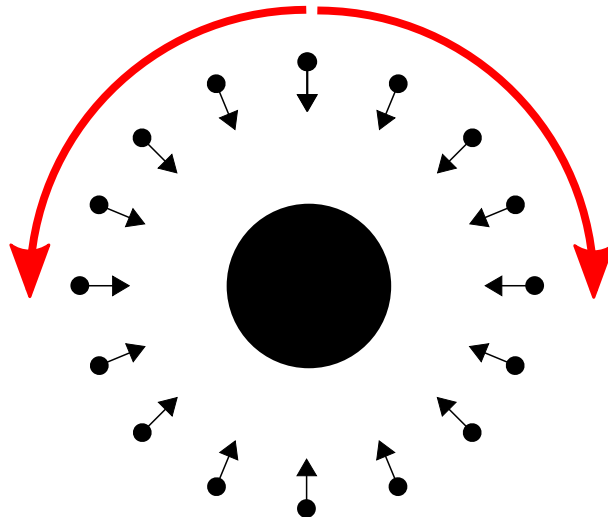


Figura 33 – Exemplo de como as imagens devem ser capturadas. O círculo no centro representa o objeto, círculos menores com as setas representam as câmeras e suas direções, e as setas vermelhas indicam o sentido horário/anti-horário.

o *matcher* baseado no *Fast Approximate Nearest Neighbor Search Library* (FLANN) [46] para o detector SIFT e o *matcher* de força bruta para o detector AKAZE. A outra abordagem em GPU utilizando o SiftGPU³ que implementa o detector SIFT e seu próprio *matcher*. Essas escolhas foram feitas para permitir uma comparação entre uma implementação em CPU e outra em GPU. Para o *bundle adjustment* foi escolhido o *Multicore Bundle Adjustment* (PBA) [23], por ser uma biblioteca bem documentada e de simples execução, além de ser focada no processamento *multithreads*, o que acelera bastante seu tempo de execução.

O algoritmo desenvolvido para a etapa de estimação da postura das câmeras tem as seguintes classes:

- *Camera*: descreve uma câmera. Contém uma imagem, a matriz intrínseca e extrínseca, os *keypoints* detectados e seus descritores.
- *Track*: descreve um ponto no espaço 3D. Contém uma lista de câmeras que têm visão desse ponto, assim como os *keypoints* dessas câmeras que correspondem ao ponto que a *Track* representa.
- *Pair*: descreve um par de câmeras. Contém as duas câmeras que formam o par e uma lista de *Tracks* formadas pelos *matches* entre os *keypoints* das duas câmeras.
- *Graph*: descreve uma cena. Contém a lista de *Tracks* que formam a cena e uma lista de câmeras.

O fluxograma simplificado da etapa de estimação da postura das câmeras pode ser visto na Figura 34. O primeiro passo é carregar as imagens e utilizar o detector de *features* para

³ <<https://github.com/pitzer/SiftGPU>>

extrair seus *keypoints* e descritores. Como as câmeras não estão calibradas, a matriz de parâmetros intrínsecos é desconhecida. Desta maneira, é feita uma inicialização padrão que será otimizada durante o processo do *bundle adjustment*, sua distância focal é 1.2 vezes a altura ou largura da imagem, o que for maior, e o ponto central é tido como o centro da imagem. Após isso, são criados os pares de câmeras, o ideal é que todas as imagens sejam comparadas a todas as outras imagens, mas esse processo é muito custoso. Como definiu-se a limitação que as fotos seriam feitas em sequência, a primeira imagem é comparada com a subsequente até que o número de *matches* seja menor que 100. Quando isso ocorre a imagem começa a ser comparada com as últimas imagens de forma decrescente até que o número de *matches* seja menor que 100, Figura 35. Isso acelera o processo e evita que pares com poucas *matches* sejam considerados. O número mínimo de *matches* foi definido em 100, pois nos testes que o par possui menos de 100 *matches* a postura estimada tem um erro muito grande. Após o *matching* são criadas as *Tracks*, basta instanciar um objeto *Track* para cada *match* que relaciona as imagens do par. Com as *Tracks* criadas é calculada a postura de uma câmera em relação a outra. Isso é feito através de uma função que encontra a matriz essencial que relaciona as duas câmeras, a função encontra a matriz através das *matches* entre as *features* detectadas nas duas câmeras. Como nem todas as *matches* são corretas, a função utiliza o RANSAC para escolher o melhor conjunto de *matches* que relaciona as câmeras, as *matches* que estão fora desse conjunto são descartadas.

Com os pares criados, inicia-se o processo de criação da cena (*Graph*), o primeiro par é adicionado a cena, desta maneira as *Tracks* que formavam o par são copiadas para essa cena. Agora deve-se continuar a agregar novos pares à cena, para isso existem duas listas de pares, os pares criados por câmeras subsequentes (câmeras 0-1, câmeras 1-2, câmeras 2-3, etc) e os pares criados sem ordem (câmeras 0-2, câmeras 0-20, câmeras 3-10, etc). A cada iteração é adicionado um par da lista de câmeras subsequentes. Quando ele é adicionado as suas *Tracks* são comparadas com as *Tracks* da cena. Caso elas estejam relacionadas ao mesmo ponto 3D, seus dados são fundidos. Caso a *Track* do par ainda não exista na cena ela é inserida na lista da *Tracks* da cena. Após o par subsequente ser adicionado, são adicionados os pares criados sem ordem (pares adicionais). Para isso se percorre toda a lista de pares adicionais, e caso as câmeras que formem o par adicional já estejam inseridas na cena, as *Tracks* desse par adicional são inseridas na cena. Nessa adição as *Tracks* do par adicional não são comparadas com as *Tracks* da cena (isso adiciona robustez ao processo). Após a adição dos novos pares é feito o *bundle adjustment* parcial, modificando-se apenas a última câmera adicionada. Caso a quantidade de pontos 3D tenha aumentado mais que 5% é realizado o *bundle adjustment* completo, onde todas as câmeras podem ser modificadas. Após o *bundle adjustment* tem-se uma etapa de filtragem dos pontos 3D para remover os pontos 3D que tenham um erro de re-projeção superior a 100 pixels. Após todos os pares subsequentes serem adicionados é realizado

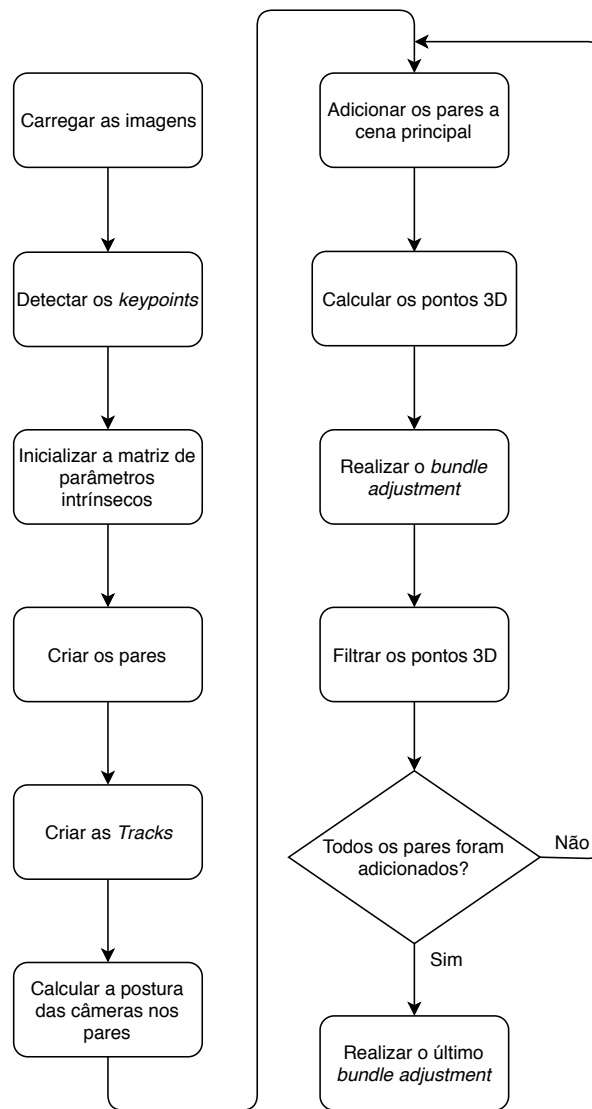


Figura 34 – Fluxograma simplificado da etapa de estimação da postura das câmeras.

um *bundle adjustment* completo e o resultado é salvo no formato ".sfm". Esse arquivo contém a quantidade de câmeras na cena e uma lista de câmeras. Essa lista contém para cada câmera: o caminho até o arquivo da imagem, a matriz de parâmetros extrínsecos e a distância focal x e y da câmera. Apesar de serem salvos dois valores de distância focal, eles são iguais, a repetição acontece apenas para a compatibilidade com o *software* utilizado para visualizar os resultados.

3.2 Geração da nuvem densa

O método implementado aqui segue a ideia de propagação de correspondências. Como entrada para essa etapa tem-se apenas a imagem, a matriz extrínseca e a distância focal de cada câmera. O fluxograma simplificado da etapa de geração da nuvem densa pode ser visto na Figura 36.

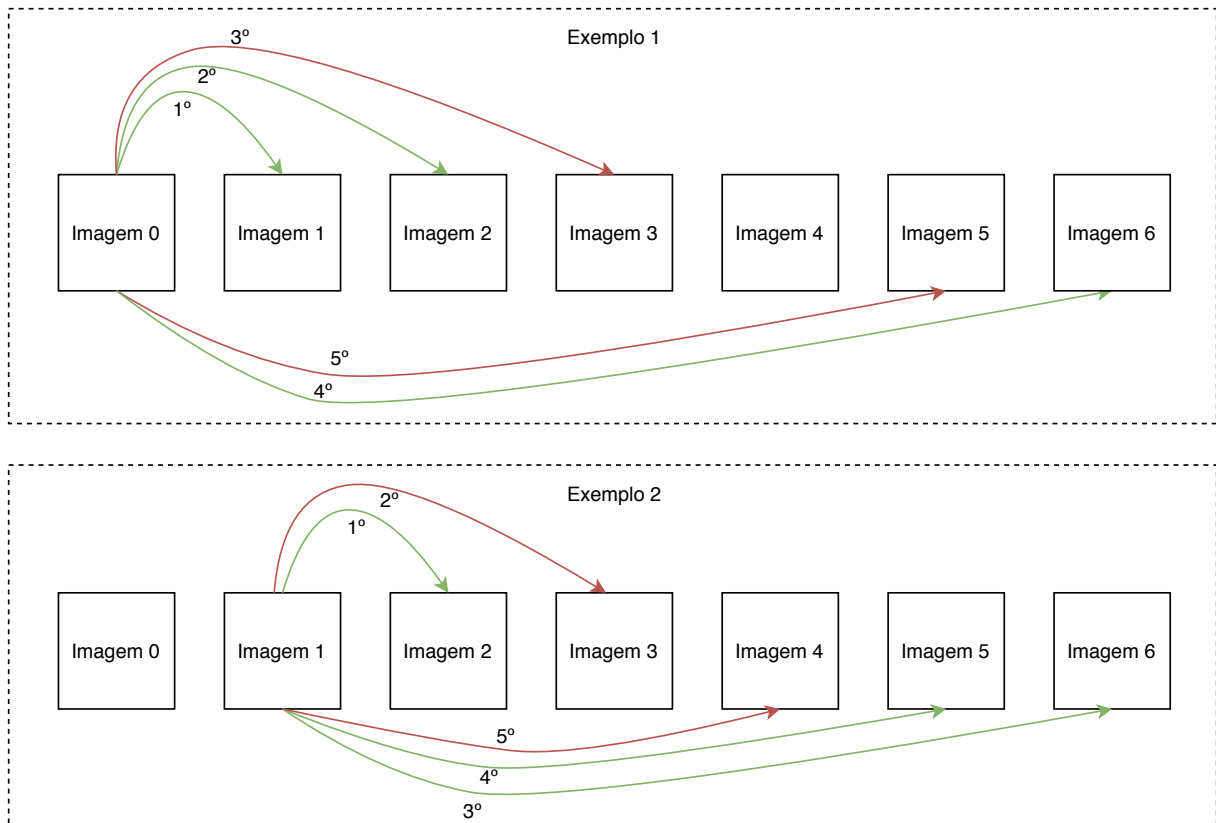


Figura 35 – Exemplo do processo de criação de pares na etapa de estimação da postura das câmeras. As setas verdes representam pares que tiveram um número de *matches* maior que 100, já as vermelhas pares com menos que 100 *matches*. Os números nas setas servem para demonstrar a ordem com que as combinações aconteceram. No primeiro exemplo a imagem 0 fez par com as imagens 1, 2 e 6, já no segundo exemplo a imagem 1 fez par com as imagens 2, 5 e 6.

O programa desenvolvido para a etapa de geração da nuvem densa tem as seguintes classes:

- *Camera*: descreve uma câmera. Contém uma imagem, a matriz intrínseca e extrínseca, os *keypoints* detectados e seus descritores (essa é a mesma classe *Camera* da etapa de estimação da postura das câmeras).
- *Seed*: descreve um ponto no espaço 3D. Contém duas câmeras, dois pontos 2D, cada um relativo a uma câmera e a pontuação obtida no *template matching*.
- *Pair*: descreve um par de câmeras. Contém as duas câmeras que formam o par e uma lista de *Seeds* formadas pelas *matches* das duas câmeras.

Após o carregamento das câmeras são criados os pares das câmeras, aqui são considerados todos os pares que tiveram o ângulo entre os eixos óticos das câmeras no intervalo de 5° a 30° . Esse intervalo foi definido pois câmeras com um ângulo menor que 5° geralmente são câmeras com uma *baseline* pequena e que apresentam apenas uma mudança de

escala, criando muitos pontos errados. Câmeras com um ângulo superior a 30° começam a ter menos correspondências entre si, além de suas imagens retificadas ficarem muito distorcidas. Um exemplo do processo de criação dos pares pode ser visto na Figura 37. Com os pares criados seleciona-se o primeiro par da lista de pares e calcula-se a retificação estéreo das imagens do par. Com as imagens retificadas é utilizado o detector de *features* SIFT implementado pelo OpenCV para encontrar os *keypoints* e seus descritores, com os descritores calculados é feito o *matching*. Como é utilizada a mesma classe *Camera* da etapa de estimação da postura das câmeras, esse processo poderia ser ignorado, já que esses dados já foram calculados, mas considera-se que os dados da estimação de postura não estão disponíveis. Como o processo de *matching* não é perfeito, os *outliers* são removidos com base no erro até a linha epipolar. Como as imagens estão retificadas, as linhas epipolares estão paralelas e os *matches* devem ter o mesmo valor de y , mas como existem erros nessa detecção aceita-se um erro de 10 pixels até a linha epipolar, Figura 38. As correspondências feitas com o SIFT irão gerar as primeiras *Seeds*.

As *Seeds* são utilizadas para realizar os cálculos dos pontos 3D, seus erros de re-projeção e facilitar a organização dos dados durante o processo de *template matching*. As *Seeds* iniciais são utilizadas para o processo de cálculo das novas *Seeds*, esse processo acontece da seguinte forma para todas as *Seeds* na lista de *Seeds* iniciais:

- Seleciona-se uma *Seed*, um exemplo de *Seed* pode ser visto na Figura 39.
- Define-se uma janela de procura de 13 pixels ao redor do ponto 1 e 2 da *Seed* selecionada, Figura 40.
- Para cada pixel dentro dessa janela é feito o seguinte procedimento:
 - Define-se uma janela de 11 pixels ao redor do pixel na imagem da câmera 1 e no pixel correspondente na imagem da câmera 2, Figura 41.
 - A janela que está na imagem da câmera 2 é movimentada, em incrementos unitários, 5 pixels para a esquerda e 5 para a direita (linha amarela tracejada na Figura 41).
 - A cada vez que a janela é movimentada calcula-se a pontuação ZNCC entre a janela da imagem da câmera 1 e a janela da imagem da câmera 2. Caso seja encontrada uma pontuação do ZNCC maior que 0.5 o resultado é salvo em uma lista temporária de *Seeds* novas. Caso contrário esse pixel não encontra correspondência e é ignorado.

Após todas as *Seeds* da lista de *Seeds* iniciais passarem por esse processo é feita a reordenação da lista temporária de *Seeds* novas, ordenando ela em ordem decrescente pela pontuação do ZNCC. Com a lista reordenada, ela é consumida da seguinte maneira: começando do primeiro elemento (com a maior pontuação do ZNCC), cada elemento é

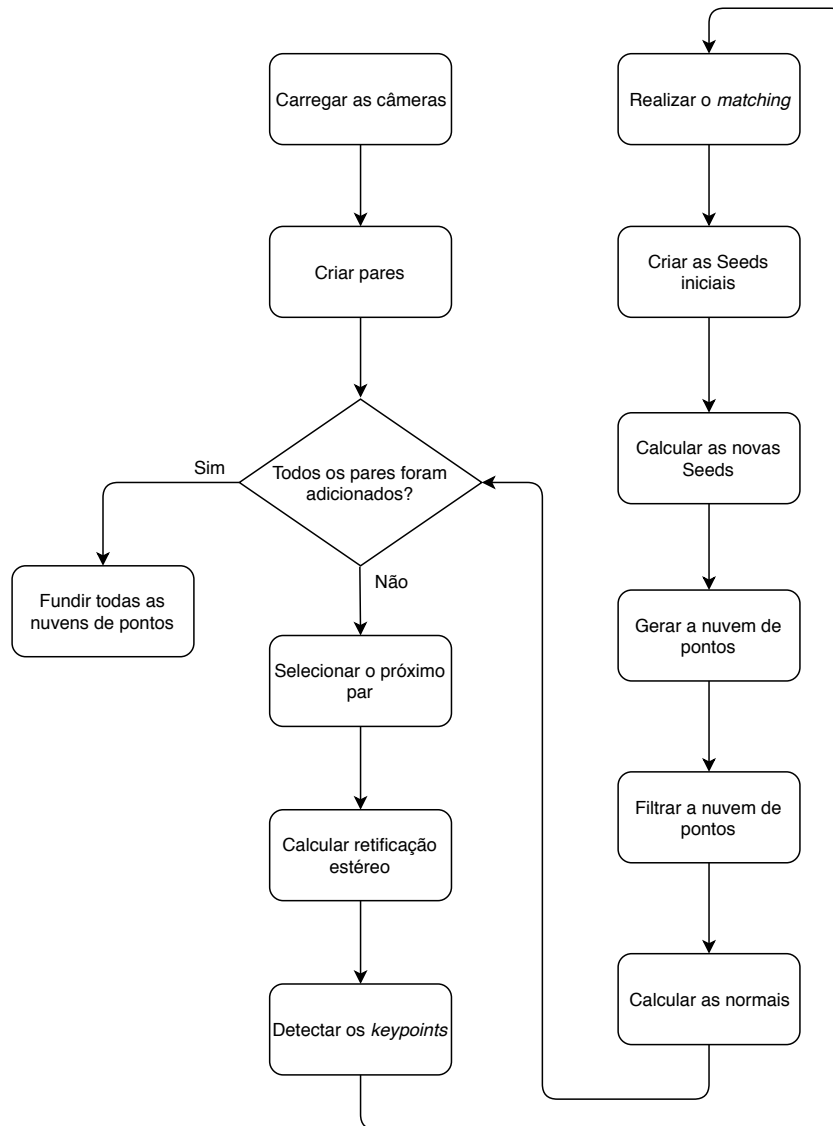


Figura 36 – Fluxograma simplificado da etapa de geração da nuvem densa.

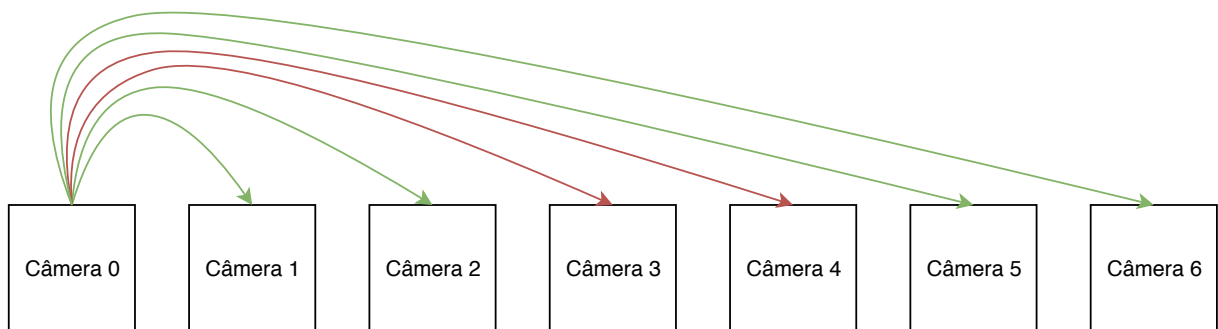


Figura 37 – Exemplo do processo de criação de pares na etapa de geração da nuvem densa. As setas verdes representam pares que tiveram o ângulo entre os eixos óticos das câmeras no intervalo de 5° a 30°, já as vermelhas pares com ângulo fora desse intervalo.

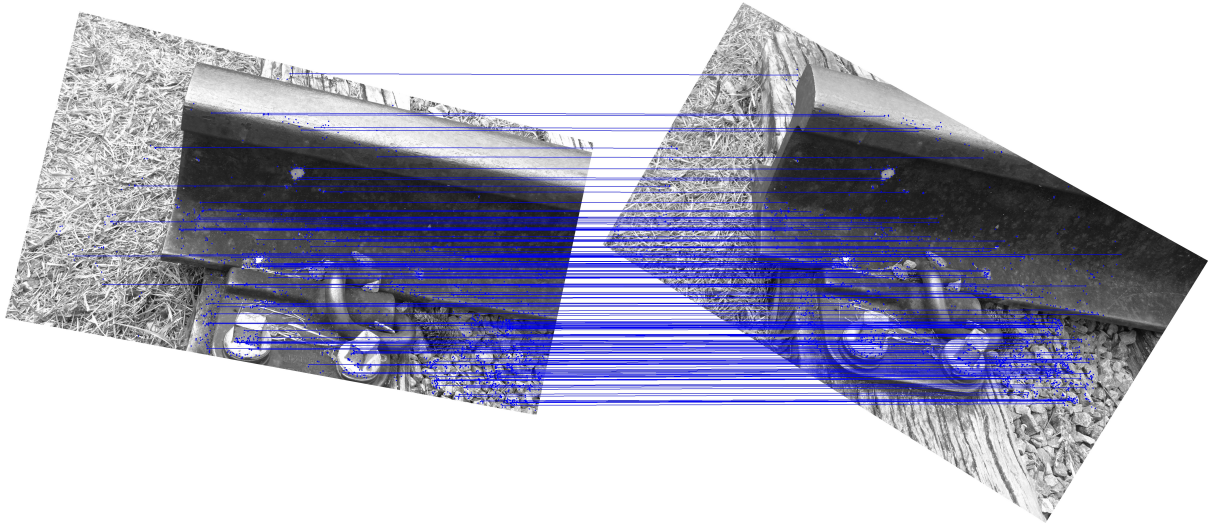


Figura 38 – Correspondências encontradas utilizando o detector de *features* SIFT (apenas algumas linhas epipolares foram desenhadas para facilitar a visualização).

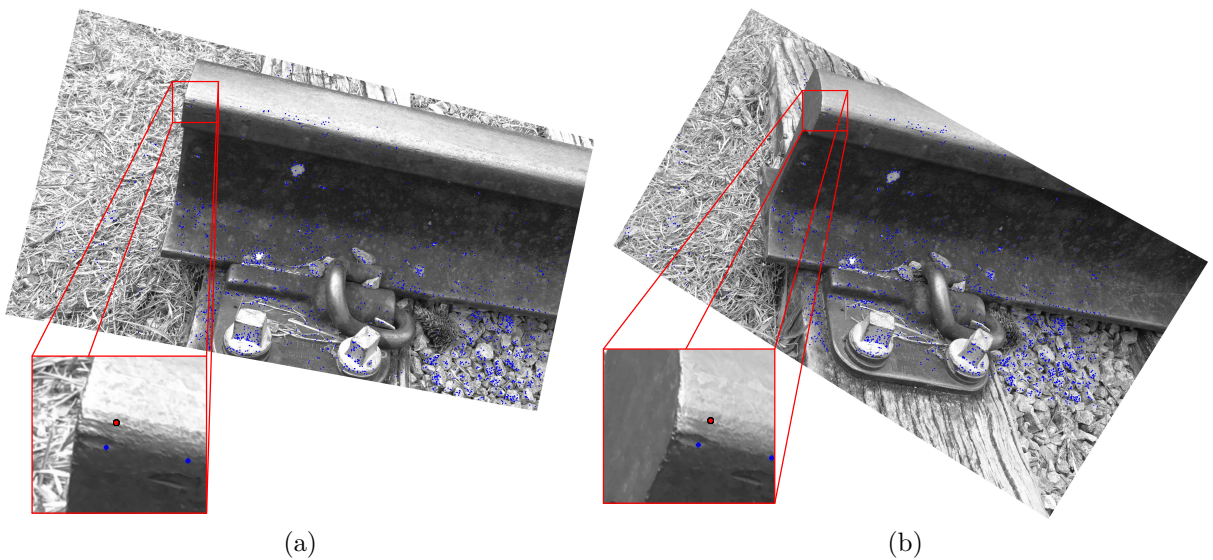


Figura 39 – Exemplo de uma *Seed* (seus pontos estão em vermelho). (a) Ponto 1 da *Seed*. (b) Ponto 2 da *Seed*.

testado, se o ponto 2 da *Seed* está livre na máscara que controla as correspondências, esse pixel é marcado como ocupado na máscara e essa *Seed* é adicionada a uma lista temporária de *Seeds*. Caso o ponto 2 da *Seed* não esteja livre, a *Seed* é descartada. A implementação foi feita dessa maneira para permitir a execução do cálculo do ZNCC em paralelo, evitando problemas de leitura e escrita simultânea na máscara. Além disso, muitas vezes uma correspondência errada pode consumir um espaço que deveria ser consumido por outra correspondência, o que é evitado pela reordenação do vetor de acordo com a pontuação do ZNCC. As *Seeds* da lista de *Seeds* iniciais que já foram utilizadas são transferidas para uma lista de *Seeds* usadas, e as *Seeds* da lista temporária são transferidas para a lista de *Seeds* iniciais. Esse processo de cálculo das novas *Seeds* é repetido 20 vezes. Esse

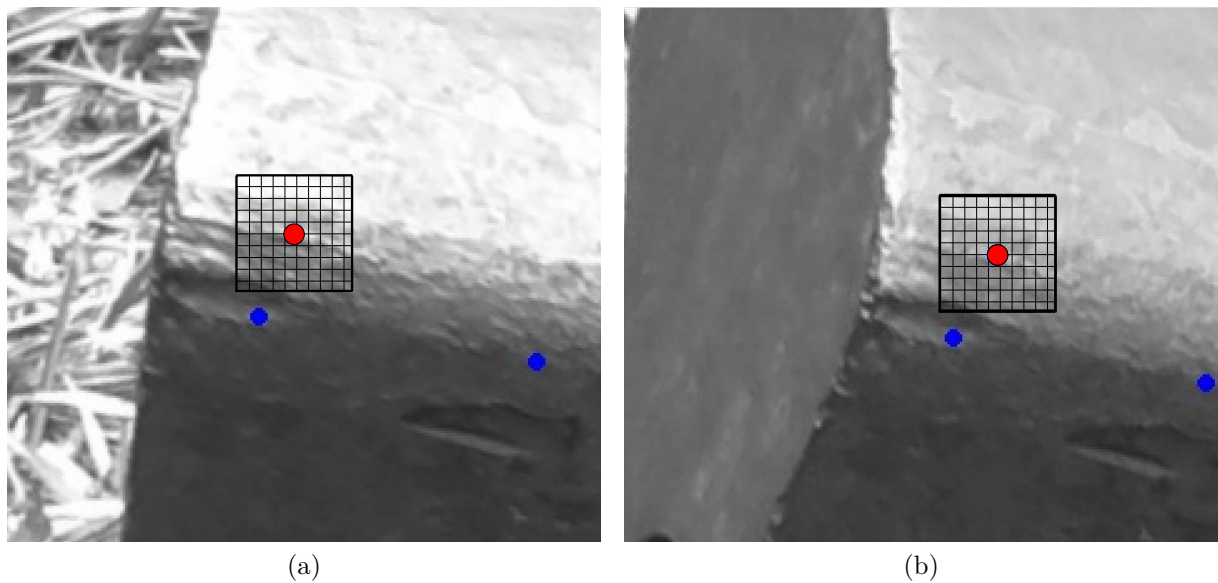


Figura 40 – Exemplo da janela de procura ao redor dos pontos da *Seed*. (a) Ponto 1 da *Seed*. (b) Ponto 2 da *Seed*.

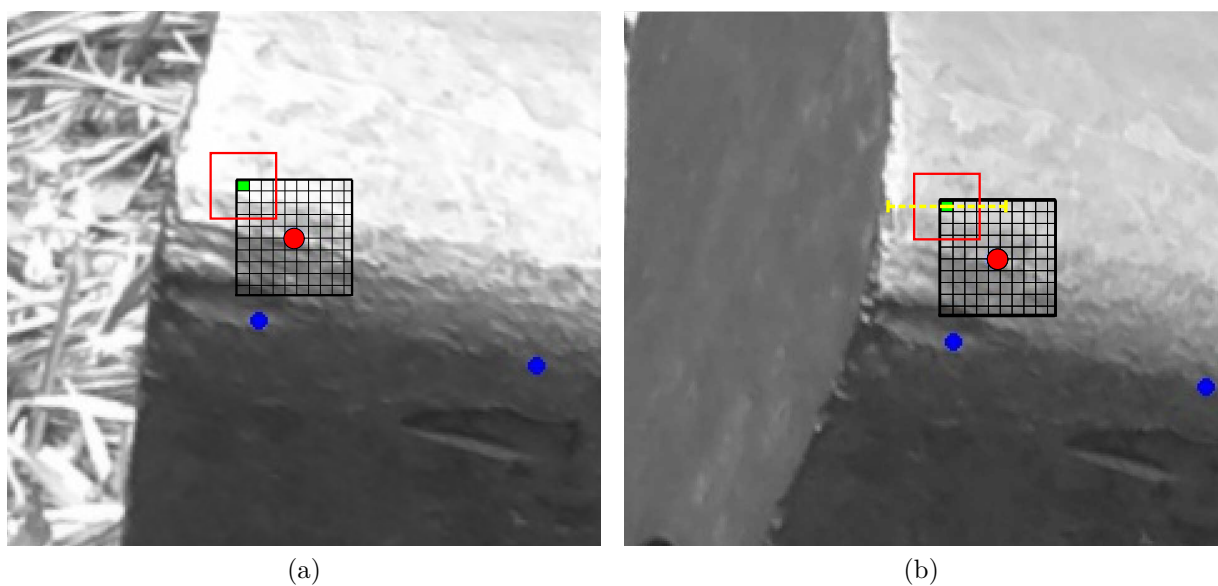


Figura 41 – Exemplo da janela de procura ao redor dos pixels da janela definida anteriormente. (a) Ponto 1 da *Seed*. (b) Ponto 2 da *Seed*.

número de repetições foi definido empiricamente, mas poderia ter sido relacionado com a quantidade de falsos positivos encontrados a cada iteração ou com a quantidade de pixels disponíveis na máscara. Um exemplo da máscara que controla os pixels utilizados para as correspondências pode ser visto na Figura 42.

Com todas as *Seeds* calculadas a nuvem de pontos é gerada através da triangulação dos pontos que tenham o erro de re-projeção inferior a 1 pixel. Após a triangulação dos pontos adicionou-se um processo de filtragem, pois muitos falsos positivos são encontrados através do processo de expansão das *Seeds*. O filtro funciona analisando a densidade da

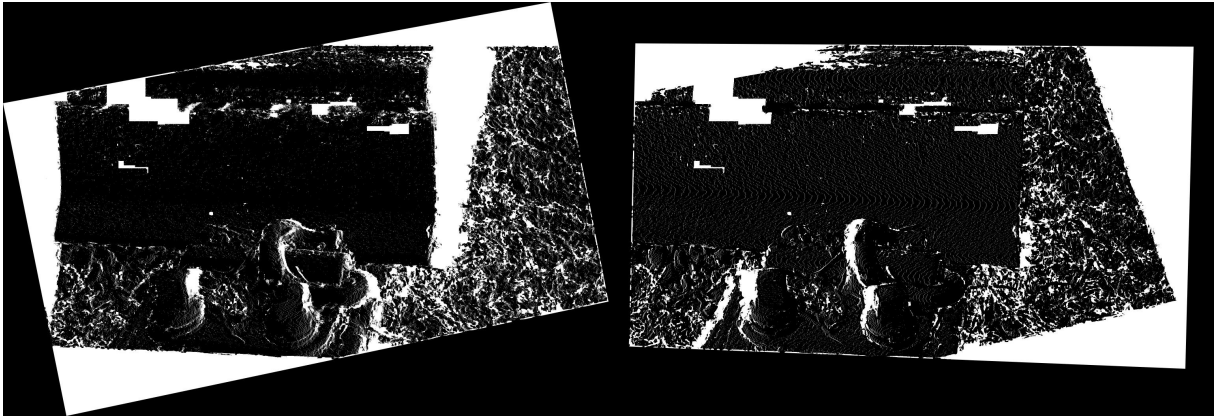
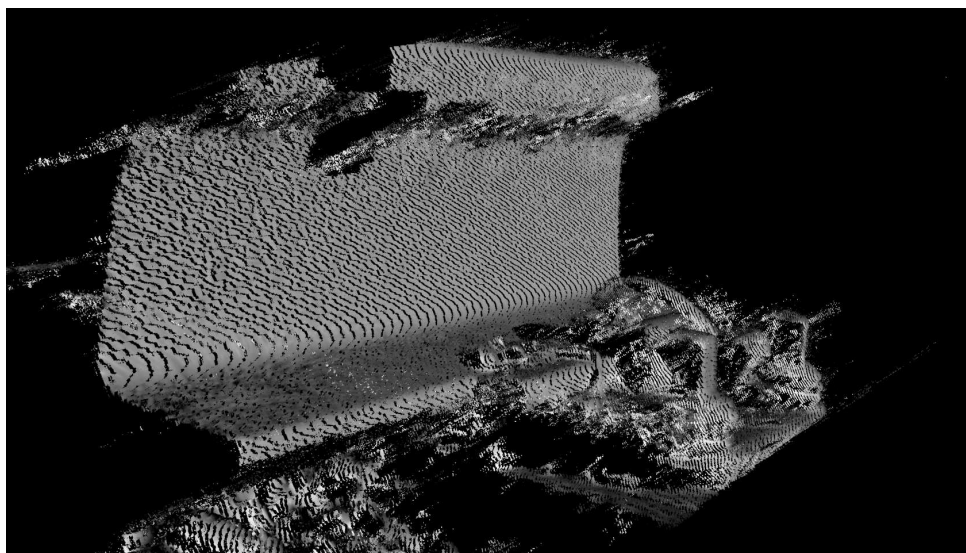
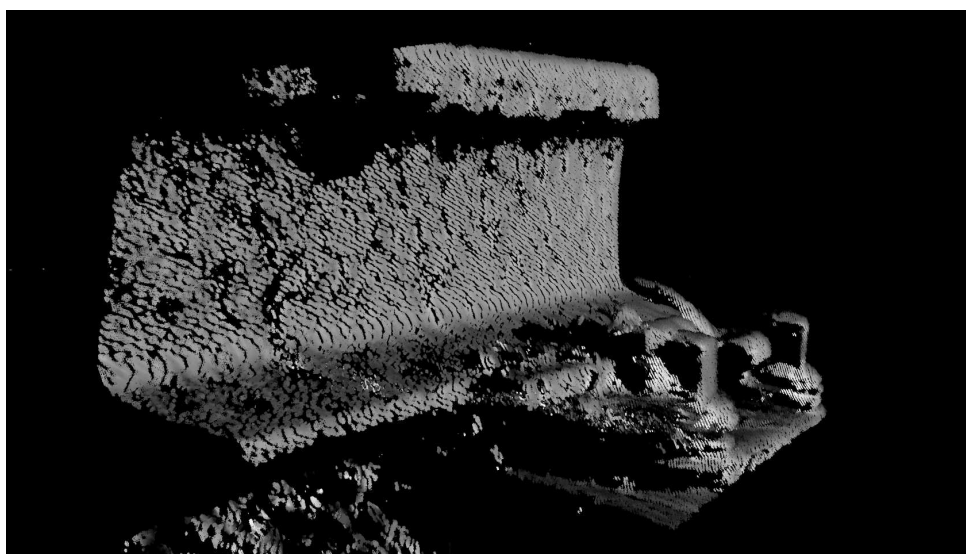


Figura 42 – Exemplo da máscara utilizada para controlar os pixels que já foram utilizados em correspondências. Os pontos pretos são os pixels já consumidos por alguma correspondência, os pixels brancos não possuem correspondências e estão livres.

região ao redor do ponto. Pontos que tenham menos que 50 vizinhos dentro de uma esfera de raio 0.01 (valor definido empiricamente) são descartados, a diferença causada por esse filtro pode ser vista na Figura 43. Com a nuvem devidamente filtrada tem-se o processo de cálculo da normal dos pontos, para isso é utilizado um filtro que aproxima a normal da superfície, que para garantir a correta orientação das normais aponta elas em direção a câmera que gerou os pontos. Com isso tem-se a nuvem de pontos densa. Após esse processo ser feito para todos os pares tem-se o processo que funde todas as nuvens de pontos, esse processo apenas agrupa todos os pontos 3D gerados em uma única nuvem, sem nenhum tipo de preocupação com a repetição do mesmo ponto 3D. Como efeito colateral áreas que aparecem em mais de um par de câmeras são desnecessariamente descritas por vários pontos, criando um problema de super amostragem.



(a)



(b)

Figura 43 – Exemplo do filtro para remover falsos positivos. (a) Nuvem sem filtro. (b) Nuvem filtrada.

4 Experimentos e resultados

Os experimentos foram feitos através da combinação de diversos algoritmos e *softwares*, cujas combinações podem ser observadas na Figura 44. Os blocos em azul representam os algoritmos desenvolvidos neste trabalho. As setas indicam o fluxo do processo, um exemplo de combinação seria: VisualSFM→MeshRecon→TexRecon. O processo inicia com a estimação da posição das câmeras utilizando o VisualSFM, seguido pela geração da nuvem densa e da superfície com o MeshRecon, finalizando com a texturização utilizando o TexRecon. Os testes foram divididos de acordo com as etapas básicas do processo de reconstrução detalhado na Seção 1.1. No caso do resultado de alguma etapa ficar muito abaixo do esperado, ele não é utilizado nas etapas seguintes. Todos os *softwares* e bibliotecas foram executados utilizando sua configuração padrão. Para realizar os experimentos foi utilizado um computador de bancada com as seguintes configurações: CPU: Intel Core i7-2600k @ 3.7GHz, memória RAM: 16GB DDR3 @ 1866MHZ; GPU: NVIDIA GTX 1060 6GB; Sistema operacional: Windows 10 64 bits.

Os testes quantitativos irão avaliar o número de pontos gerados na etapa de estimação da postura das câmeras e na etapa de geração da nuvem densa, as faces geradas na etapa da geração da superfície e as faces texturizadas na etapa de texturização, além do tempo de execução, memória RAM e memória VRAM (memória da GPU) utilizada. O tempo de execução é disponibilizado pelos próprios algoritmos, já a quantidade de memória RAM e VRAM utilizadas foram medidas através de um histórico de utilização de recursos do computador adquirido através do *software* MSI Afterburner¹.

Os resultados serão apresentados em tabelas. Para permitir a compreensão das combinações utilizadas os métodos serão escritos da seguinte forma: <método de estimação da postura das câmeras> + <método de geração da nuvem densa> + <método de geração da superfície> + <método de texturização>. Para os *softwares* será apenas utilizado o nome do *software*, já que não é realizado nenhum tipo de combinação.

Para permitir a legibilidade das tabelas de resultados, os nomes dos métodos serão abreviados. Para os métodos de estimação da postura das câmeras tem-se: VisualSFM (VSFM), SFM-AKAZE (SFM-A), SFM-SIFT (SFM-S) e SFM-SIFTGPU (SFM-SG). Para os métodos de geração de nuvem densa tem-se: PMVS/CMVS (P/C) e DENSE (DEN). Para os métodos de geração de superfície tem-se: PoissonRecon (PR), PCL-Greedy (PCL-GY) e PCL-Grid (PCL-GD). Para os métodos de texturização tem-se: TexRecon (TR) e PCL_TEXT (PCL_T).

¹ <<https://www.msi.com/page/afterburner>>

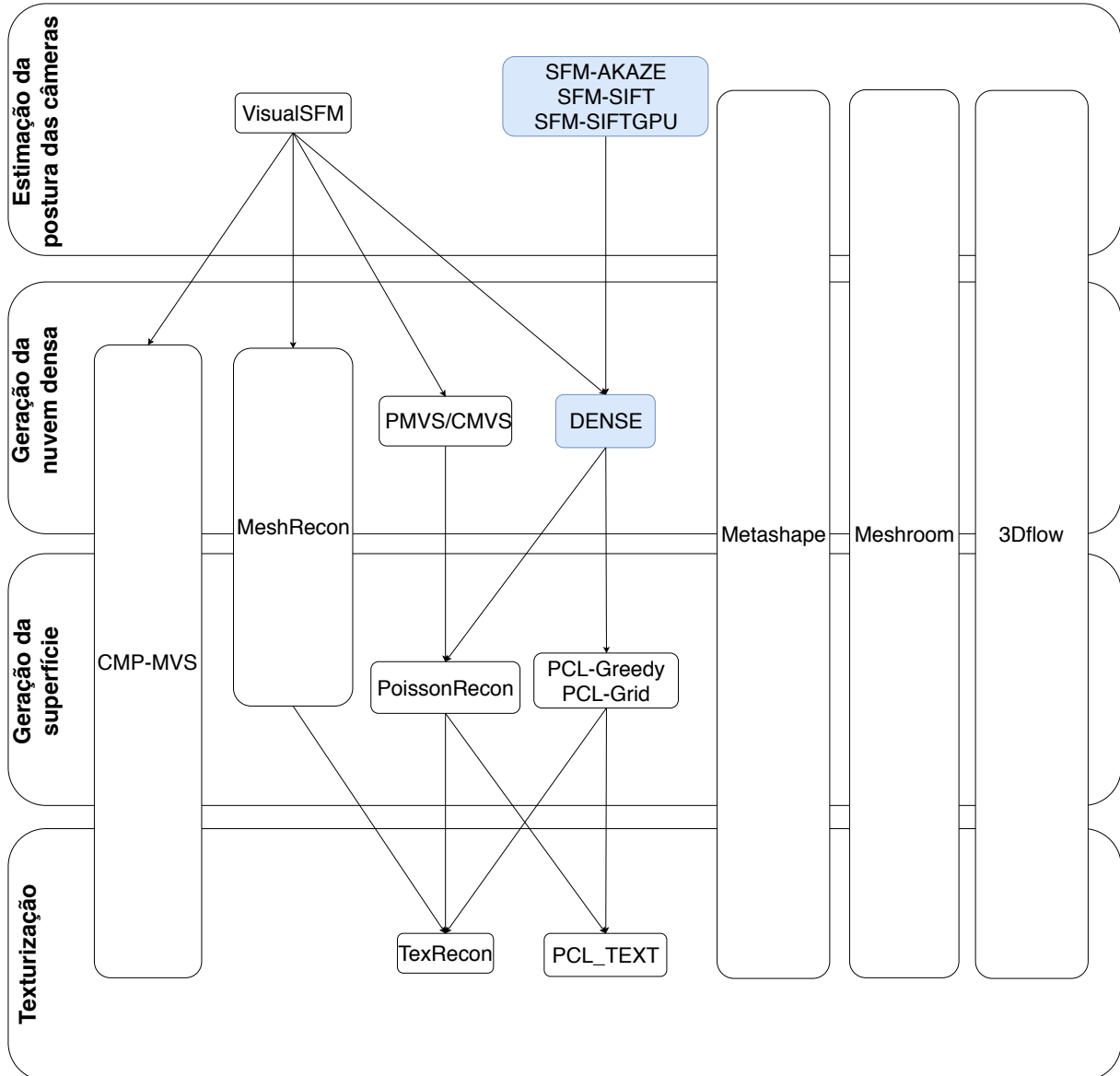


Figura 44 – Fluxograma dos testes, os blocos na cor azul são os algoritmos desenvolvidos neste trabalho.

4.1 Datasets

Para os experimentos iniciais foi utilizado um dataset com 21 imagens de um trilho de trem. As imagens foram capturadas movendo-se a câmera ao redor do trilho e mantendo o mesmo no centro das imagens, Figura 45. Uma característica desse dataset é a variação da iluminação nas imagens, isso pode ser observado comparando-se os parafusos na Figura 45d e na Figura 45c. Essa não é uma característica desejável, mas será utilizada para comparar o comportamento dos algoritmos de texturização.

Para os experimentos adicionais foram utilizados dois datasets, o primeiro dataset para os experimentos adicionais é disponibilizado no site do MeshRecon, ele contém 23 imagens de uma gárgula. Este dataset é interessante pois a gárgula possui vários detalhes

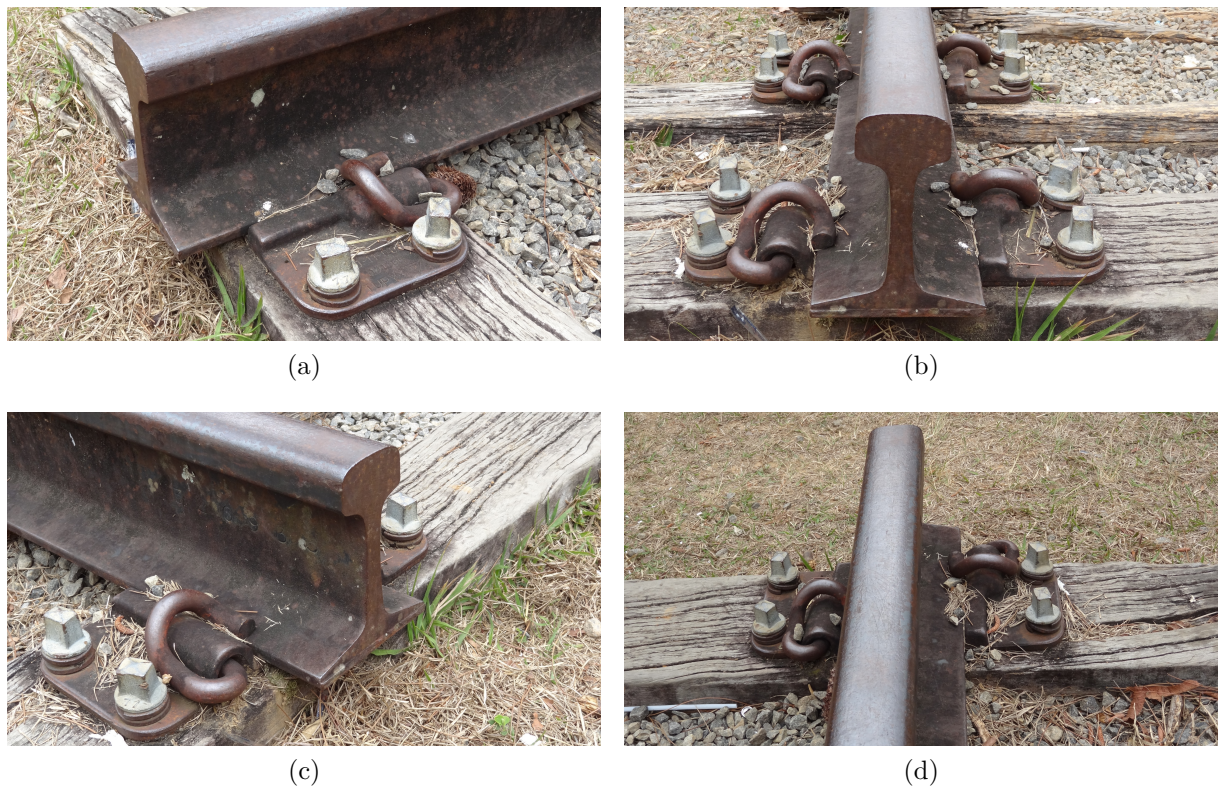


Figura 45 – Exemplos do dataset do trilho.

pequenos, desta forma pode-se analisar qual combinação consegue manter esses detalhes na reconstrução. Um exemplo das imagens do conjunto pode ser visto na Figura 46.



Figura 46 – Exemplos do dataset da gárgula.

O segundo dataset é conhecido como *fountain-P11* [47]. Contém 11 imagens de uma fonte e a calibração da câmera, possibilitando uma melhor análise do resultado da etapa do posicionamento das câmeras. Um exemplo das imagens do conjunto pode ser visto na

Figura 47. A matriz intrínseca da câmera obtida através da calibração pode ser vista na Equação 4.1.

$$K = \begin{bmatrix} 2759.48 & 0 & 1520.69 \\ 0 & 2764.16 & 1006.81 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.1)$$

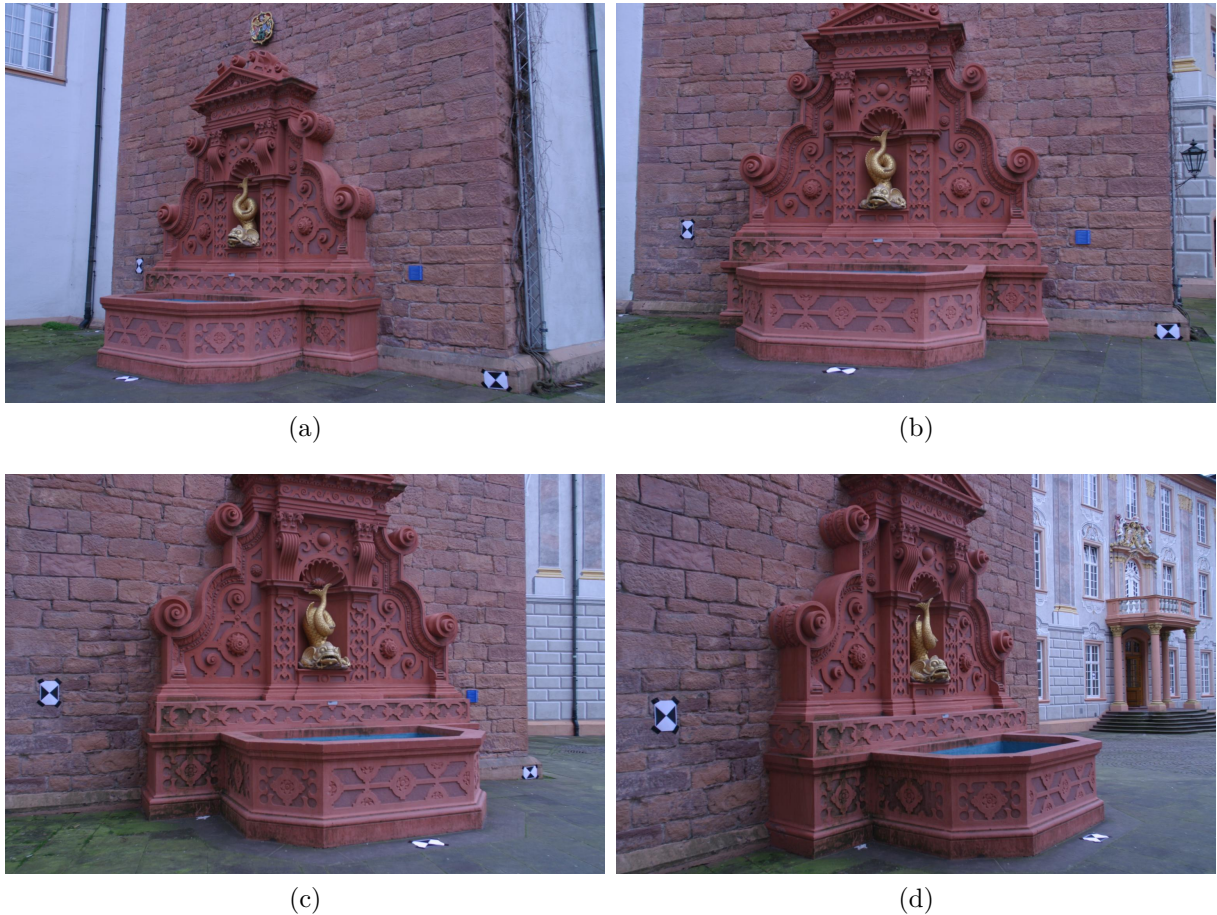


Figura 47 – Exemplos do dataset da fonte.

4.2 Estimação da postura das câmeras

Nesta seção serão apresentados os resultados dos testes feitos para os métodos de estimação de postura das câmeras. Esses métodos tem como entrada apenas as imagens do dataset e a saída deles será uma nuvem de pontos esparsa e a matriz intrínseca e extrínseca das câmeras. Essa nuvem de pontos esparsa representa os pontos que foram encontrados em pelo menos duas imagens e foram utilizados para estimar a postura das câmeras. Em alguns resultados alguns dos pontos da nuvem de pontos esparsa estão visivelmente errados, como pontos abaixo da terra. Isto mostra que apesar de alguns métodos conseguirem estimar com uma precisão boa a postura das câmeras, os pontos errados acabam

Método	Tempo	Número de pontos	Memória RAM	Memória VRAM
Metashape	14s	8.076	60MB	-
Meshroom	1min 16s	17.890	2.8GB	-
3Dflow	49s	4.488	640MB	-
VSFM	58s	18.782	-	709MB
SFM-A	10min 55s	13.849	598MB	-
SFM-S	12min 41s	37.713	1.8GB	-
SFM-SG	13min 39s	2.246	840MB	702MB

Tabela 2 – Resultados quantitativos para a etapa de estimação da postura das câmeras para o dataset do trilho.

adicionando erros na estimação. Apesar das imagens terem sido capturadas pela mesma câmera, os métodos tratam cada câmera como única, ou seja, os parâmetros intrínsecos variam. Isso permite que o algoritmo do *bundle adjustment* tenha mais variáveis para manipular e assim estime a melhor postura das câmeras. Os resultados quantitativos podem ser observados na Tabela 2.

Na Figura 48 tem-se o resultado do *software* Metashape. Ele teve o resultado mais rápido e com um consumo de memória RAM muito inferior aos outros métodos. O fato de sua nuvem de pontos ter uma quantidade baixa de pontos quando comparada a outros resultados mostra a robustez de seu método de estimação de postura, pois com poucos pontos ele já foi capaz de convergir para um resultado bom.

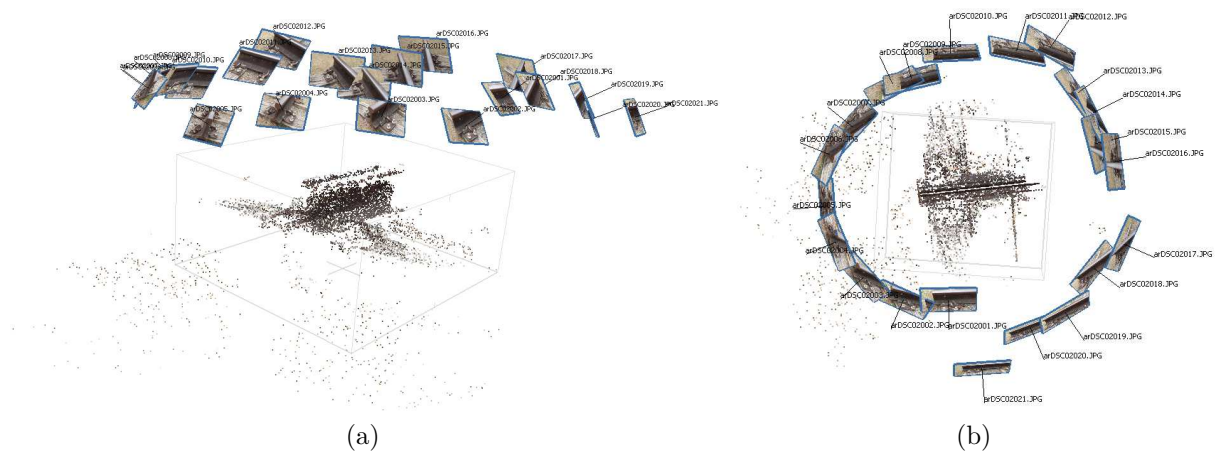


Figura 48 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do Metashape. (a) Vista lateral. (b) Vista superior.

Na Figura 49 tem-se o resultado do *software* Meshroom. Apesar do tempo baixo e a grande quantidade de pontos gerados, foi consumida mais memória RAM que os métodos desenvolvidos neste trabalho (SFM-AKAZE, SFM-SIFT e SFM-SIFTGPU), que são códigos que ainda não passaram por uma etapa de otimização.

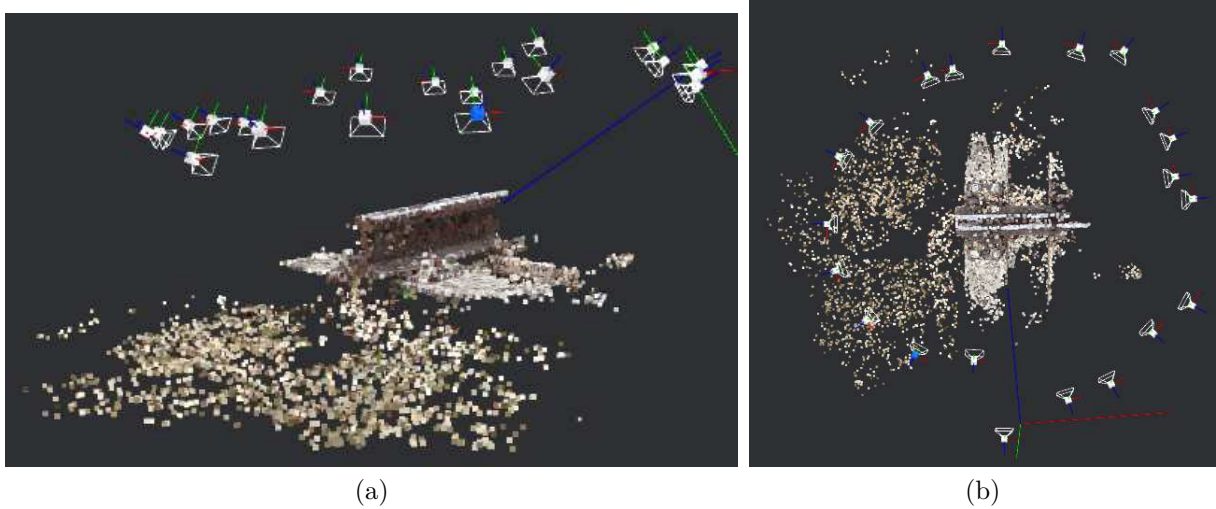


Figura 49 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do Meshroom. (a) Vista lateral. (b) Vista superior.

Na Figura 50 tem-se o resultado do *software* 3Dflow. Esse resultado teve um bom tempo, e conseguiu superar o resultado do Metashape na quantidade de pontos utilizados para conseguir uma estimação da postura das câmeras. Apesar disto, pode-se perceber que existem diversos pontos flutuando acima do trilho, Figura 50a. Além disso, sabe-se pelas fotos que não deveriam existir objetos nessa área, indicando a falha no filtro de falsos positivos do algoritmo.

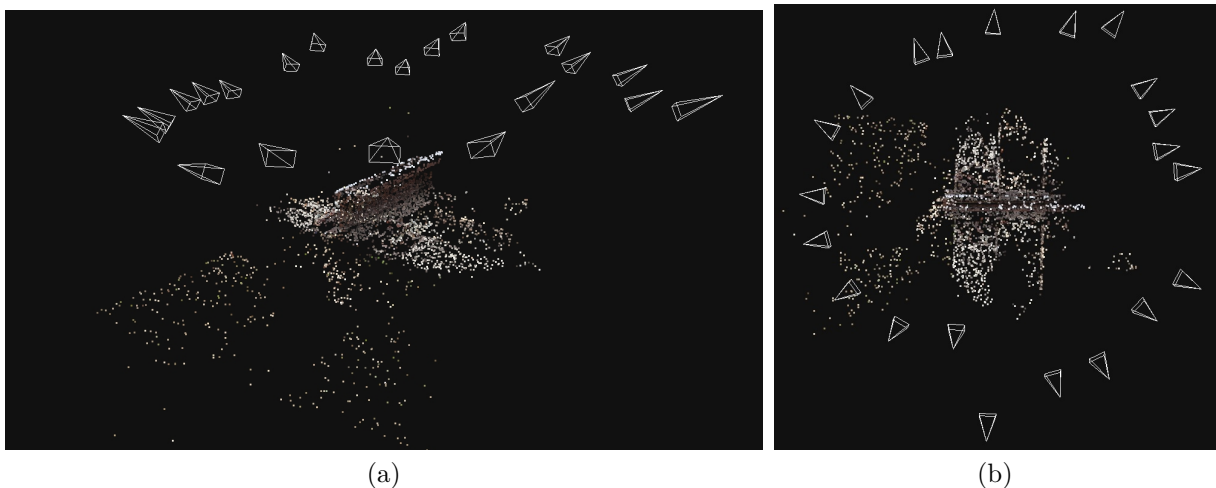


Figura 50 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do 3Dflow. (a) Vista lateral. (b) Vista superior.

Na Figura 51 tem-se o resultado do *software* VisualSFM. Esse resultado teve um bom tempo e uma boa quantidade de pontos, mas por ser o único método que utiliza principalmente a GPU para realizar os cálculos, era esperado um tempo melhor.

Na Figura 52 tem-se o resultado do algoritmo SFM-AKAZE. Esse resultado teve o

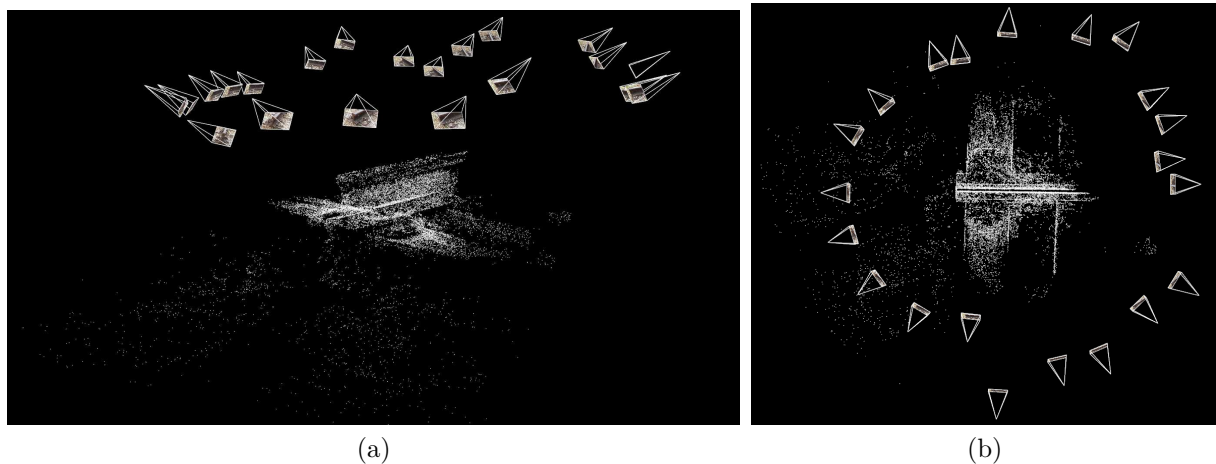


Figura 51 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do VisualSFM (as cores dos pontos foram retiradas para facilitar a visualização). (a) Vista lateral. (b) Vista superior.

melhor tempo entre os algoritmos desenvolvidos neste trabalho, e uma quantidade de pontos parecida com o Meshroom, mas com um consumo de memória inferior. Seu tempo poderia ter sido melhor se fosse possível utilizar o *matcher* baseado em FLANN, já que o *matcher* força bruta é ineficiente.

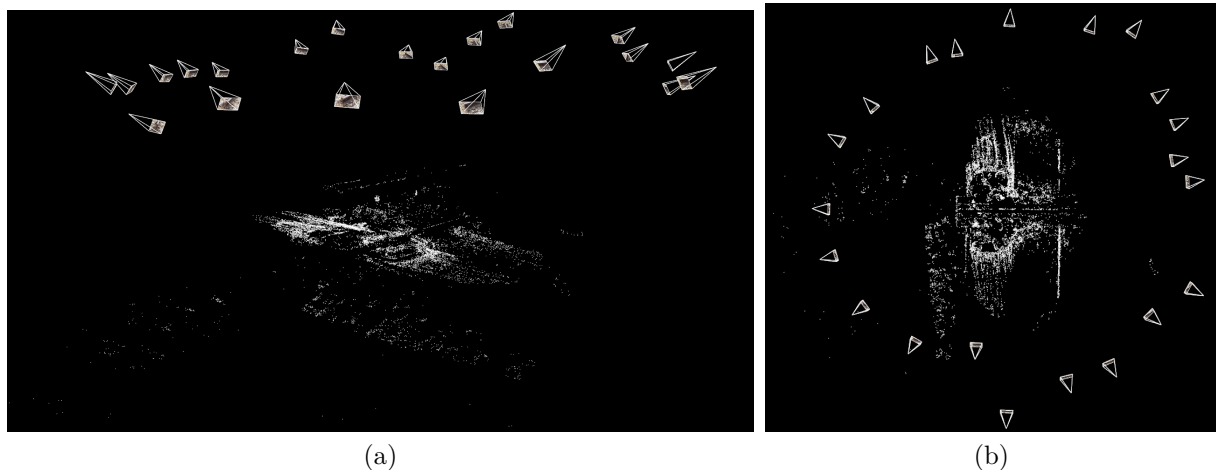


Figura 52 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do SFM-AKAZE. (a) Vista lateral. (b) Vista superior.

Na Figura 53 tem-se o resultado do algoritmo SFM-SIFT. Esse resultado teve um bom tempo e a maior quantidade de pontos gerados, o que acabou influenciando no consumo de memória RAM. Essa grande quantidade de pontos não é necessariamente boa, já que se consome muita memória RAM e tempo, enquanto seria possível chegar ao mesmo resultado utilizando menos pontos e conseqüentemente menos tempo e memória RAM.

Na Figura 54 tem-se o resultado do algoritmo SFM-SIFTGPU. Esse resultado teve o pior tempo e a menor quantidade de pontos gerados, além de ter sido o único método

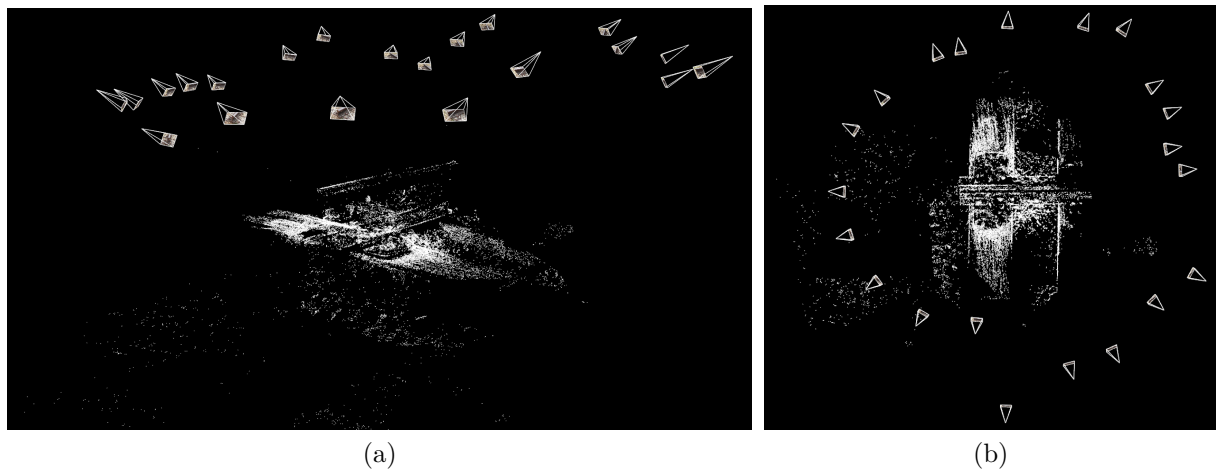


Figura 53 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do SFM-SIFT. (a) Vista lateral. (b) Vista superior.

que falhou em estimar a posição de todas as câmeras. Pode-se perceber na Figura 54b que existe uma fileira inteira de câmeras desalinhadas e por este motivo o método foi descartado para as próximas etapas.

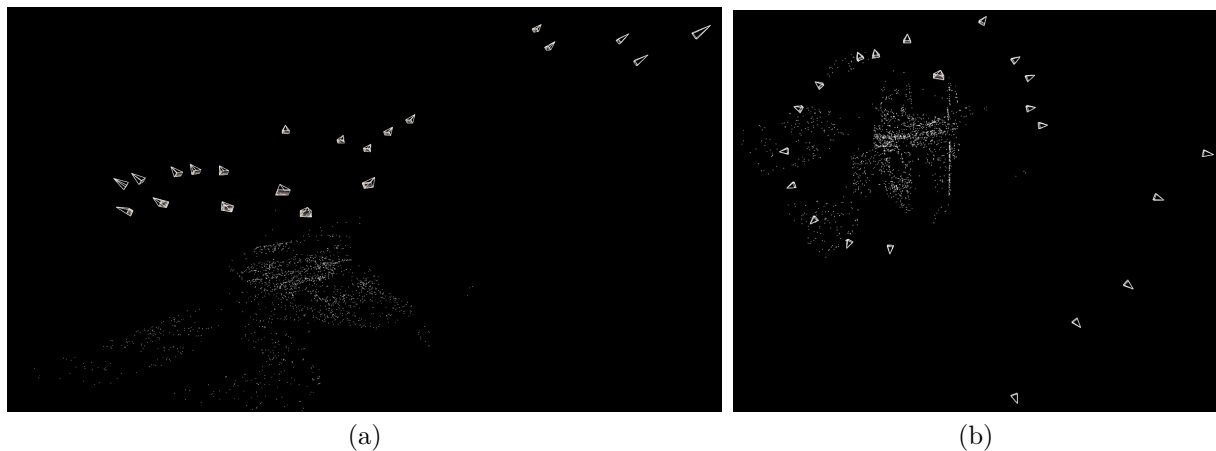


Figura 54 – Resultado da estimação da nuvem esparsa de pontos e postura das câmeras do SFM-SIFTGPU. (a) Vista lateral. (b) Vista superior.

4.3 Geração da nuvem densa

Nesta seção são apresentados os resultados dos testes feitos para os métodos de geração da nuvem densa. Esses métodos tem como entrada as imagens do dataset, e as matrizes intrínsecas e extrínsecas das câmeras, e como saída será uma nuvem de pontos densa com as normais dos pontos calculadas. Os resultados quantitativos podem ser observados na Tabela 3.

Método	Tempo	Número de pontos	Memória RAM	Memória VRAM
Metashape	1min 20s	988.390	100MB	-
3Dflow	1min 19s	957.083	654MB	-
SFM-A + DEN	1hr 28min	18.836.227	8.8GB	-
SFM-S + DEN	1hr 44min	24.363.445	12.8GB	-
VSFM + DEN	3hr 56min	37.009.608	11.1GB	-
VSFM + P/C	3min 34s	536.751	1.8GB	-

Tabela 3 – Resultados quantitativos para a etapa de geração da nuvem densa para o dataset do trilho.



Figura 55 – Resultado da nuvem densa do Metashape. (a) Vista lateral. (b) Vista superior.

Na Figura 55 tem-se o resultado do *software* Metashape. Novamente esse *software* conseguiu entregar um bom resultado, com um dos menores tempos, uma boa quantidade de pontos e um consumo de memória RAM muito inferior aos outros algoritmos. Analisando as imagens pode-se perceber que o *software* limita a área reconstruída, pois algumas partes do cenário que estavam presentes em mais de uma foto não foram adicionadas na reconstrução. O topo do trilho também não ficou correto, nas áreas onde o trilho apresenta mais reflexos o *software* acabou criando buracos que não deveriam existir.

Na Figura 56 tem-se o resultado do *software* 3Dflow. Esse *software* teve os números muito parecidos com o Metashape, mas com um maior consumo de memória RAM. Apesar disso, conseguiu reconstruir de maneira correta o topo do trilho e não restringiu a área de reconstrução, conseguindo reconstruir um bom pedaço do gramado.

Na Figura 57 tem-se o resultado da combinação SFM-AKAZE + DENSE. Apesar da grande quantidade de pontos, pode-se perceber que essa combinação falhou em várias áreas do trilho. Isso se deve principalmente ao resultado do SFM-AKAZE, que não conseguiu estimar tão bem a postura das câmeras.

Na Figura 58 tem-se o resultado da combinação SFM-SIFT + DENSE. Essa combina-

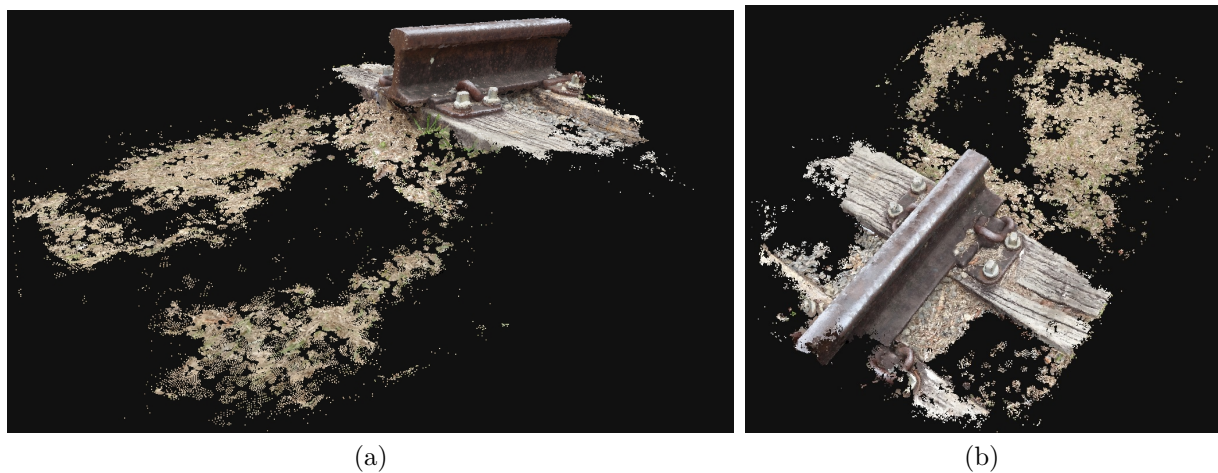


Figura 56 – Resultado da nuvem densa do 3Dflow. (a) Vista lateral. (b) Vista superior.

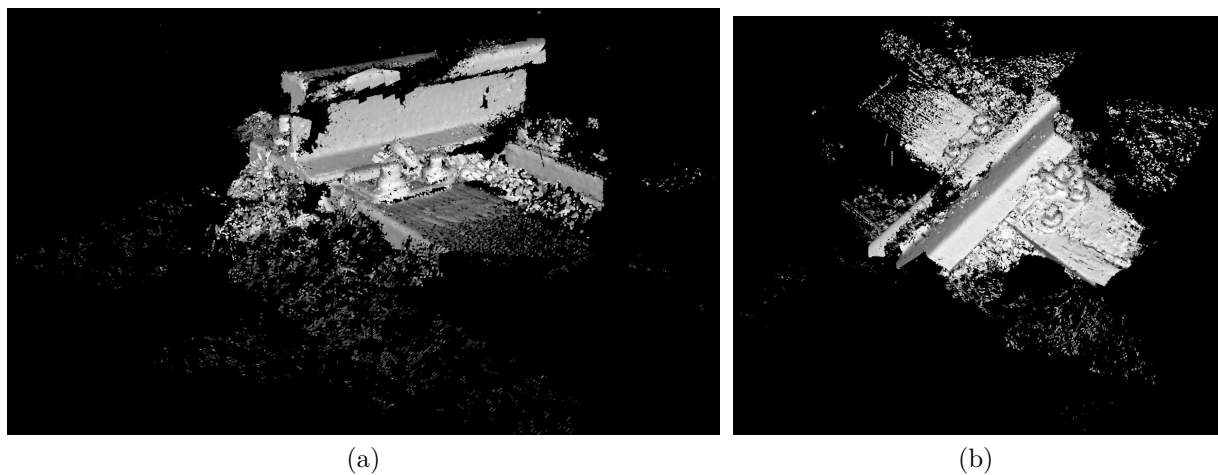


Figura 57 – Resultado da nuvem densa do SFM-A + DEN. (a) Vista lateral. (b) Vista superior.

ção teve um resultado melhor se comparado ao SFM-AKAZE + DENSE, principalmente pelo fato do SFM-SIFT ter estimado melhor a posição das câmeras. Mesmo assim o método não foi capaz de reconstruir a maior parte do gramado.

Na Figura 59 tem-se o resultado da combinação VisualSFM + DENSE. Essa combinação conseguiu reconstruir de maneira mais completa o gramado e o trilho. Mas a sua grande quantidade de pontos também introduz uma grande quantidade de ruído.

Na Figura 60 tem-se o resultado da combinação VisualSFM + PMVS/CMVS. Apesar desta combinação não ter gerado uma grande quantidade de pontos, ela foi capaz de gerar uma nuvem uniforme, reconstruindo praticamente toda a parte visível do trilho, mas falhou na reconstrução de grande parte do gramado e foi a única combinação que falhou na reconstrução da tábua que aparece mais a direita na Figura 60a.

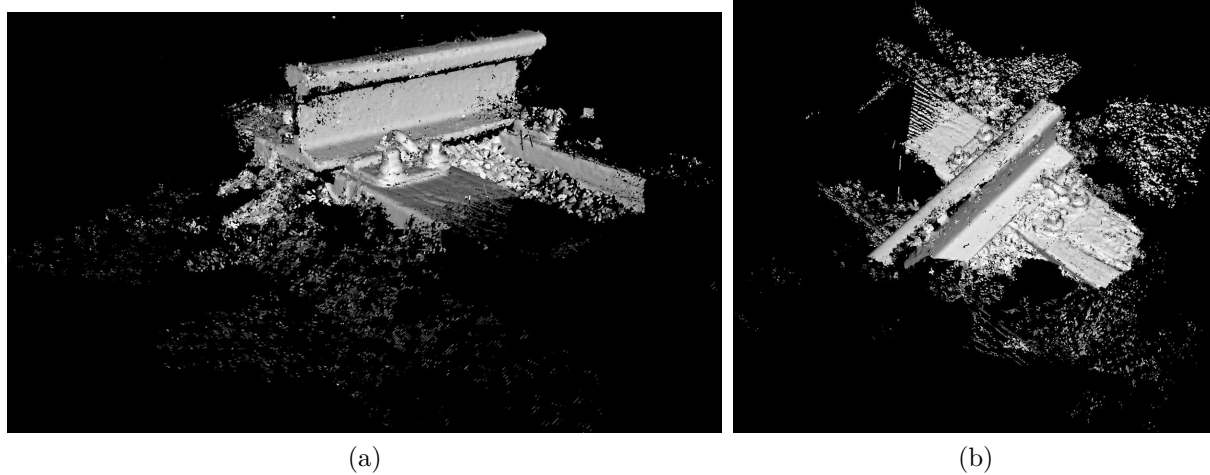


Figura 58 – Resultado da nuvem densa do SFM-S + DEN. (a) Vista lateral. (b) Vista superior.

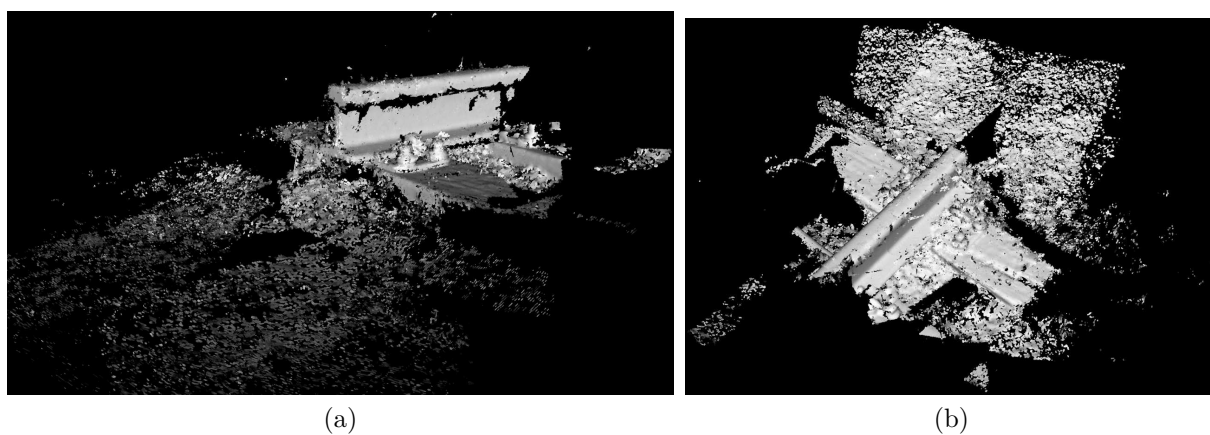


Figura 59 – Resultado da nuvem densa do VSFM + DEN. (a) Vista lateral. (b) Vista superior.

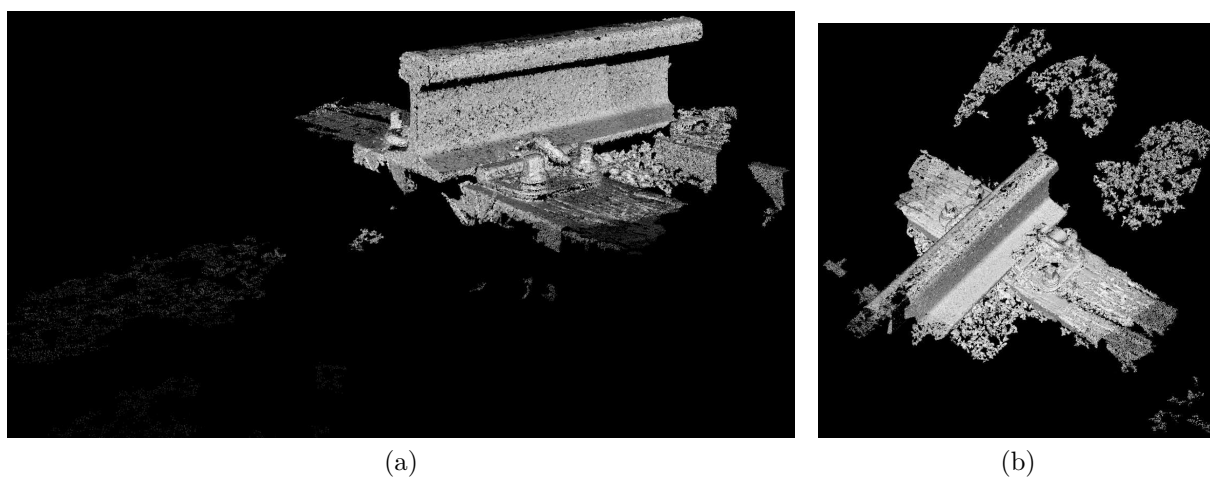


Figura 60 – Resultado da nuvem densa do VSFM + P/C. (a) Vista lateral. (b) Vista superior.

Método	Tempo	Número de faces	Memória RAM	Memória VRAM
Metashape	51s	197.671	589MB	-
Meshroom	20min	627.411	3.8GB	1.6GB
3Dflow	2min 48s	187.784	442MB	-
VSFM + MR	2min 25s	946.788	1.9GB	2.1GB
SFM-A + DEN + PR	30s	578.720	1.4GB	-
SFM-S + DEN + PR	46s	1.428.162	1.9GB	-
VSFM + DEN + PR	54s	1.170.496	2.7GB	-
VSFM + P/C + PR	23s	852.126	200MB	-
SFM-A + DEN + PCL-GY	47min 45s	35.639.858	6.8GB	-
SFM-A + DEN + PCL-GD	1hr 50min	578.882	5.7GB	-
SFM-S + DEN + PCL-GY	1hr 26min	45.647.089	8.3GB	-
SFM-S + DEN + PCL-GD	2hr 28min	1.329.298	7.2GB	-
VSFM + DEN + PCL-GY	3hr 35min	68.281.813	11.7GB	-
VSFM + DEN + PCL-GD	3hr 36min	609.276	8.2GB	-

Tabela 4 – Resultados quantitativos para a etapa da geração da superfície para o dataset do trilho.

4.4 Geração da superfície

Nesta seção serão apresentados os resultados dos testes feitos para os métodos de geração da superfície. Como entrada esses métodos tem a nuvem de pontos densa com normais. Para o caso do MeshRecon (MR), ele terá como entrada os mesmos arquivos da etapa de geração da nuvem densa, já que ele faz os dois processos de forma consecutiva e não disponibiliza a nuvem criada, apenas a superfície. Os métodos PCL-Greedy (PCL-GY) e PCL-Grid (PCL-GD) são as implementações dos métodos *Greedy Projection Triangulation* (Seção 2.6.2) e *Grid Projection* (Seção 2.6.3), respectivamente. Os resultados quantitativos podem ser observados na Tabela 4.

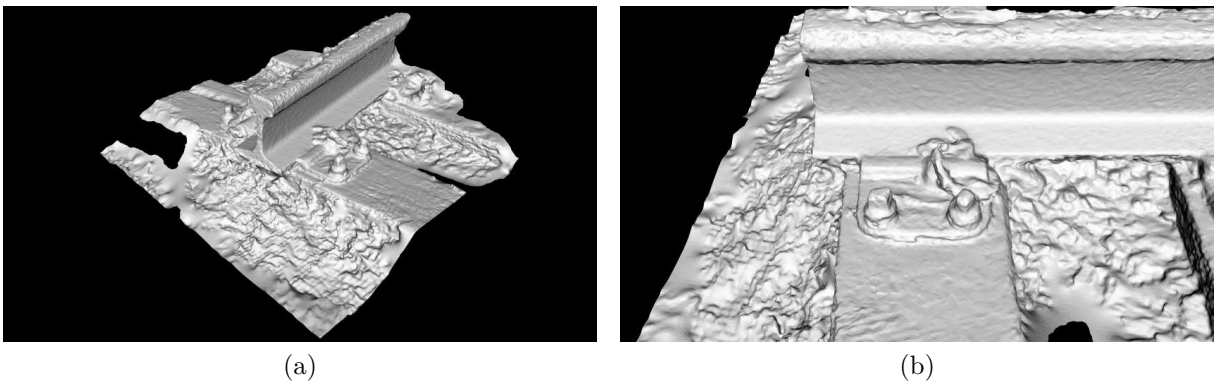


Figura 61 – Resultado da superfície do Metashape. (a) Vista geral. (b) Vista detalhe.

Na Figura 61 tem-se o resultado do *software* Metashape. Esse resultado teve o segundo melhor tempo de execução, perdendo apenas para o PoissonRecon. O número de

faces foi baixo, mas isso se deve pela falta do gramado. O uso de memória RAM foi baixo comparado aos outros testes. Como um dos problemas da nuvem densa do Metashape foram os buracos no topo do trilho, isso ficou mais evidente após a geração da superfície, onde o topo do trilho não ficou uniforme.

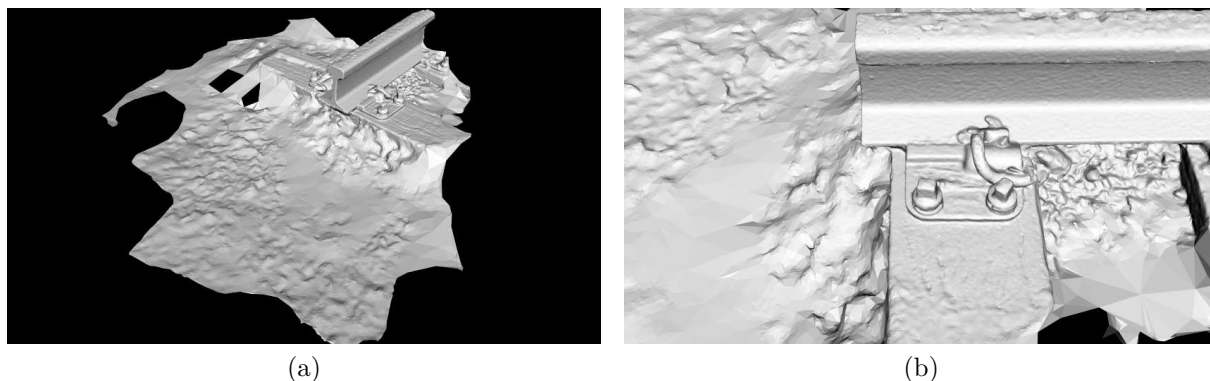


Figura 62 – Resultado da superfície do Meshroom. (a) Vista geral. (b) Vista detalhe.

Na Figura 62 tem-se o resultado do *software* Meshroom. Esse resultado teve um tempo de execução maior que os outros *softwares* pois ele não possui uma etapa separada para o processo de geração de nuvem densa, que é feita em conjunto com a etapa da geração da superfície. O consumo de recursos também foi alto, já que ele também utilizou a placa de vídeo.

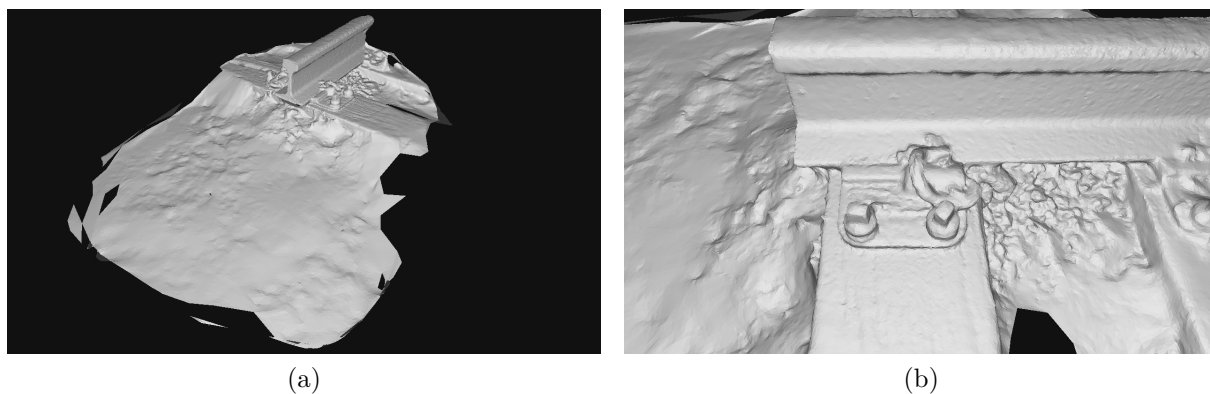


Figura 63 – Resultado da superfície do 3Dflow. (a) Vista geral. (b) Vista detalhe.

Na Figura 63 tem-se o resultado do *software* 3Dflow. Esse resultado teve um bom tempo de execução e uma baixa quantidade de faces e de memória RAM utilizada. A baixa quantidade de faces não foi um problema, pois ele ainda conseguiu manter os detalhes do trilho.

Na Figura 64 tem-se o resultado da combinação VisualSFM + MeshRecon. Assim como o Meshroom, o MeshRecon também não separa as etapas de geração de nuvem densa e geração de superfície, mas ele consegue fazer isso de uma maneira mais eficiente.

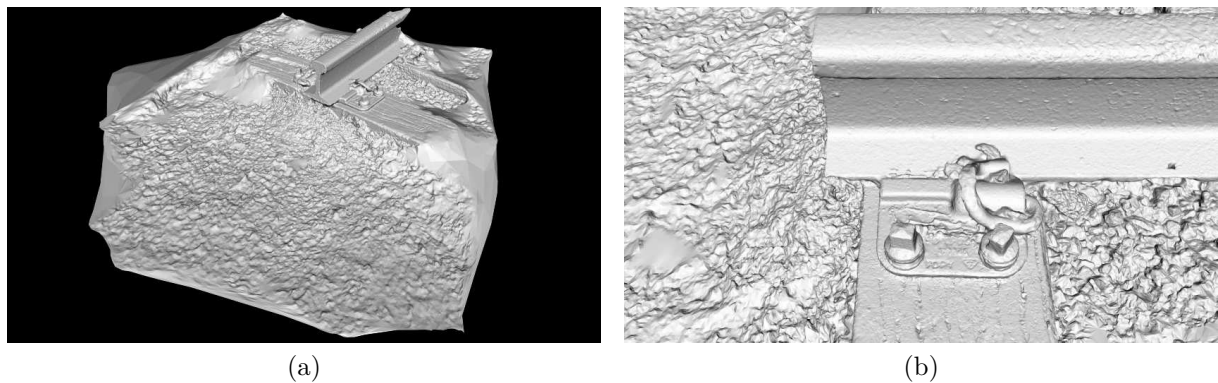


Figura 64 – Resultado da superfície do VSFM + MR. (a) Vista geral. (b) Vista detalhe.

Seu tempo de execução foi muito menor e a quantidade de faces geradas também foi maior. O destaque desta combinação são os detalhes. Na Figura 64b pode-se perceber que o método conseguiu manter as ranhuras da madeira, essa foi a única combinação que conseguiu esse nível de detalhes.

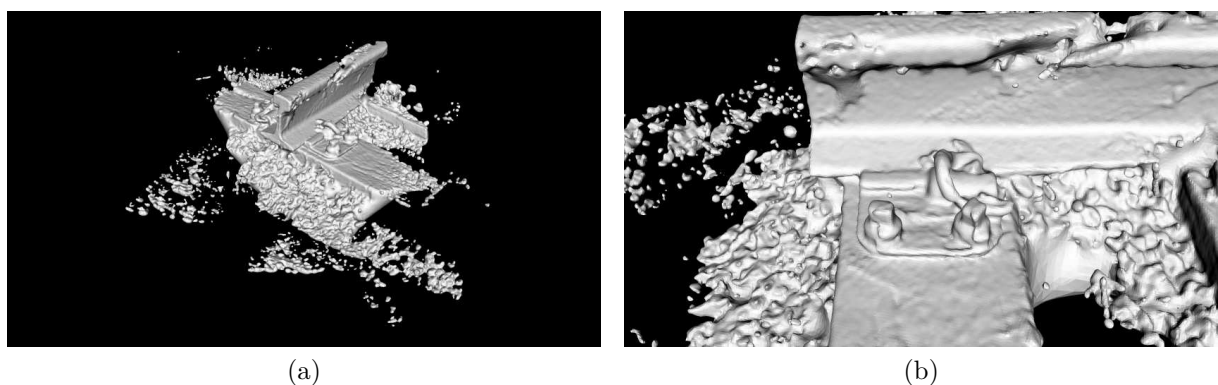


Figura 65 – Resultado da superfície do SFM-A + DEN + PR. (a) Vista geral. (b) Vista detalhe.

Na Figura 65 tem-se o resultado da combinação SFM-AKAZE + DENSE + Poisson-Recon. Novamente é visível que erros na etapa de estimação da postura das câmeras acabam prejudicando muito as próximas etapas. O trilho ficou com uma grande falha, e os parafusos sofreram com um defeito de duplicação, Figura 65b. O erro no parafuso aconteceu pelo erro na postura das câmeras, que fez com que durante a etapa da geração da nuvem densa dois parafusos foram formados onde deveria existir apenas um.

Na Figura 66 tem-se o resultado da combinação SFM-SIFT + DENSE + PoissonRecon. Essa combinação teve um bom resultado, o único ponto negativo foi a pouca quantidade de pontos no gramado, de forma que o *Poisson* não conseguiu manter uma superfície.

Na Figura 67 tem-se o resultado da combinação VisualSFM + DENSE + PoissonRecon. Essa combinação teve um bom resultado, mas a grande quantidade de ruídos que apareceu na nuvem densa acabou atrapalhando a geração da superfície. A parte superior

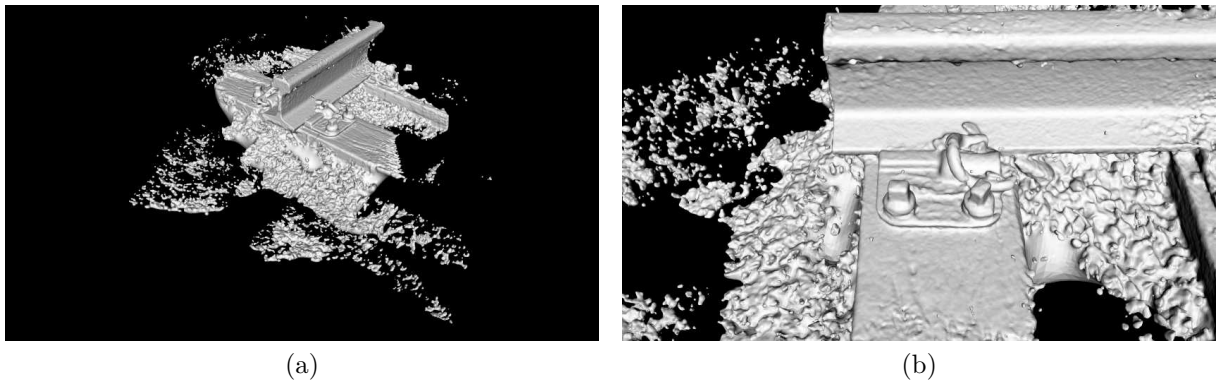


Figura 66 – Resultado da superfície do SFM-S + DEN + PR. (a) Vista geral. (b) Vista detalhe.

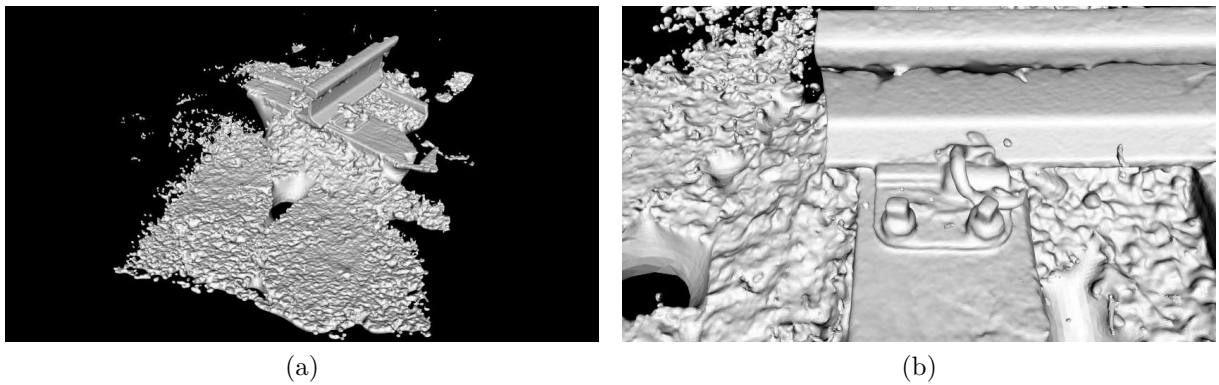


Figura 67 – Resultado da superfície do VSFM + DEN + PR. (a) Vista geral. (b) Vista detalhe.

do trilho tem alguns artefatos saindo de sua estrutura, que foram gerados por esses ruídos. Um ponto positivo foi que uma grande parte do gramado foi reconstruída.

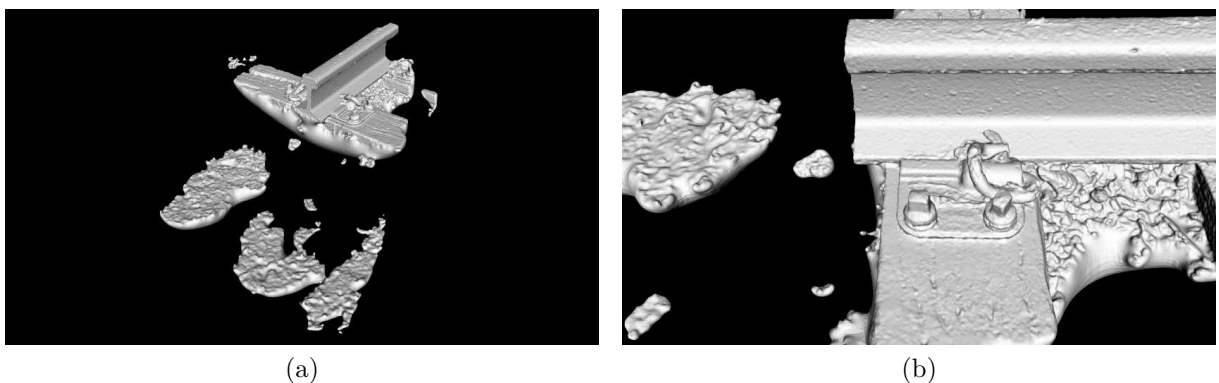


Figura 68 – Resultado da superfície do VSFM + P/C + PR. (a) Vista geral. (b) Vista detalhe.

Na Figura 68 tem-se o resultado da combinação VisualSFM + PMVS/CMVS + PoissonRecon. Essa combinação teve o menor tempo de execução e consumo de memória

RAM. A reconstrução ficou boa, mas como a etapa de geração da nuvem densa não reconstruiu o gramado corretamente, essa parte ficou falha.

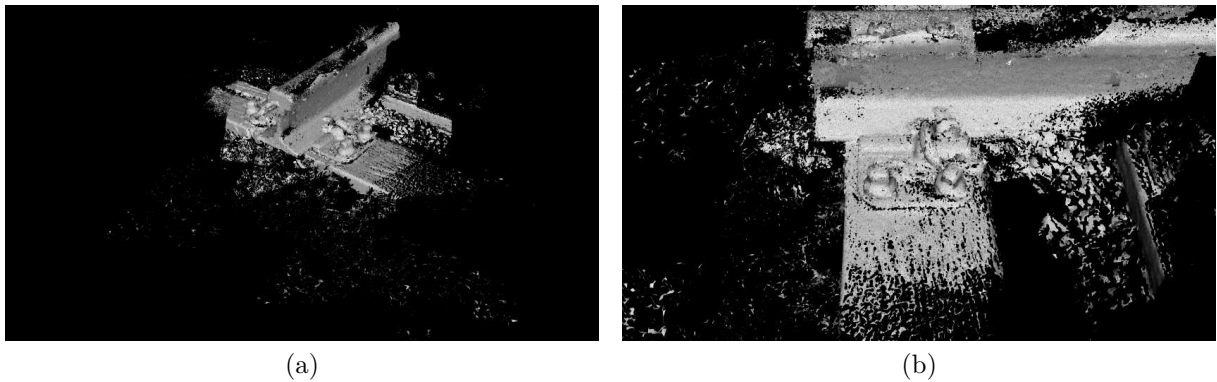


Figura 69 – Resultado da superfície do SFM-A + DEN + PCL-GY. (a) Vista geral. (b) Vista detalhe.

Na Figura 69 tem-se o resultado da combinação SFM-AKAZE + DENSE + PCL-Greedy. Essa combinação teve um péssimo resultado, teve um tempo, número de faces e consumo de memória RAM muito altos, além disso o resultado ficou muito longe do ideal. Isso acontece pois o método *Greedy Projection Triangulation* não é o mais indicado para nuvens com muito ruído, pois ele não consegue suavizar o impacto deles, como o *Poisson* faz. Este mesmo problema é percebido na combinação SFM-SIFT + DENSE + PCL-Greedy, Figura 71, e na combinação VisualSFM + DENSE + PCL-Greedy, Figura 73.

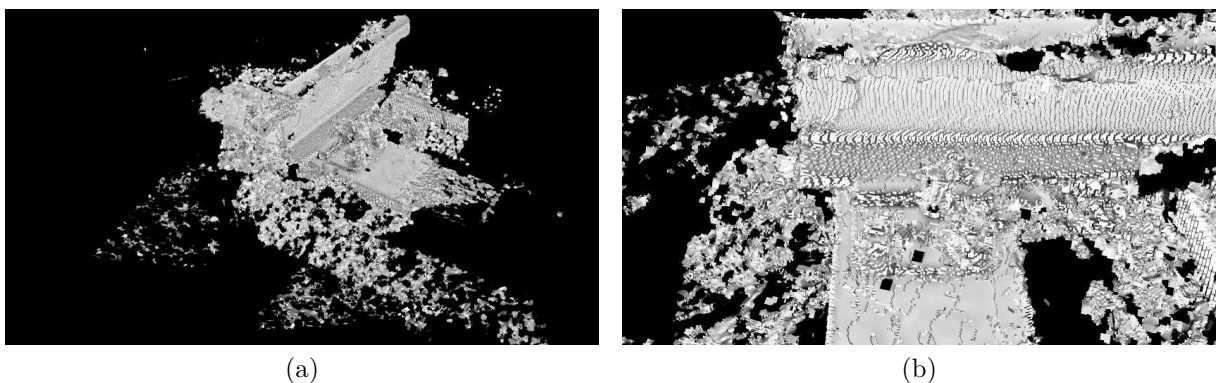


Figura 70 – Resultado da superfície do SFM-A + DEN + PCL-GD. (a) Vista geral. (b) Vista detalhe.

Na Figura 70 tem-se o resultado da combinação SFM-AKAZE + DENSE + PCL-Grid. Essa combinação não teve um bom resultado e o tempo de execução e consumo de memória RAM ficaram altos. Assim como o *Greedy Projection Triangulation*, o *Grid Projection* não é o método mais indicado para nuvens com muito ruído, pois ele não consegue suavizar o impacto deles, como o *Poisson* faz. Um ponto positivo é que ao contrário do *Greedy Projection Triangulation*, o *Grid Projection* não cria as faces conectando

os pontos, isso faz com que o número de faces não seja tão exagerados como no *Greedy Projection Triangulation*. Este mesmo problema é percebido na combinação SFM-SIFT + DENSE + PCL-Grid, Figura 72, e na combinação VisualSFM + DENSE + PCL-Grid, Figura 74.

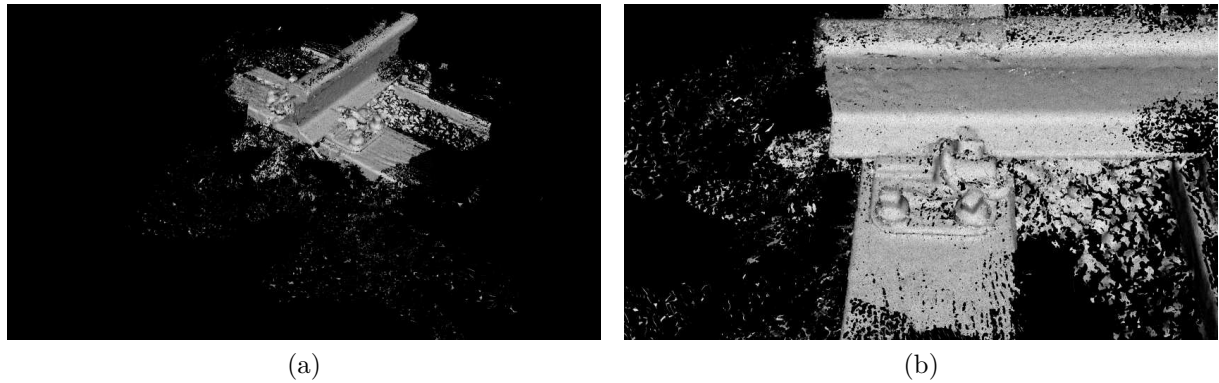


Figura 71 – Resultado da superfície do SFM-S + DEN + PCL-GY. (a) Vista geral. (b) Vista detalhe.

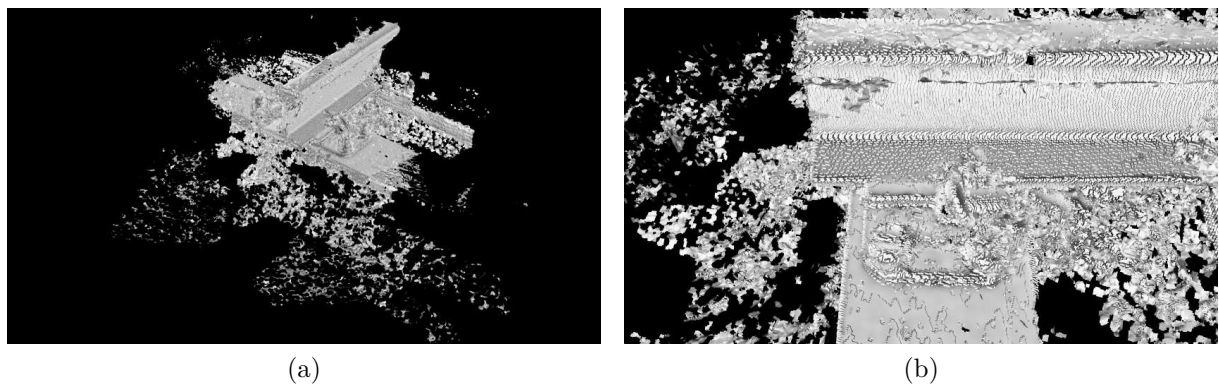


Figura 72 – Resultado da superfície do SFM-S + DEN + PCL-GD. (a) Vista geral. (b) Vista detalhe.

Como os resultados feitos com os métodos PCL-Greedy e PCL-Grid tiveram um resultado muito abaixo do esperado, eles não foram utilizados nos testes da etapa de texturização.

4.5 Texturização

Nesta seção serão apresentados os resultados dos testes feitos para os métodos de texturização. Os métodos de texturização tem como entrada a superfície gerada, as imagens do dataset e as matrizes intrínsecas e extrínsecas das câmeras, e como saída tem-se a superfície texturizada. Nos resultados do método PCL_TEXT (PCL_T), foram excluídas do número de faces texturizadas as faces que foram texturizadas com a imagem padrão.

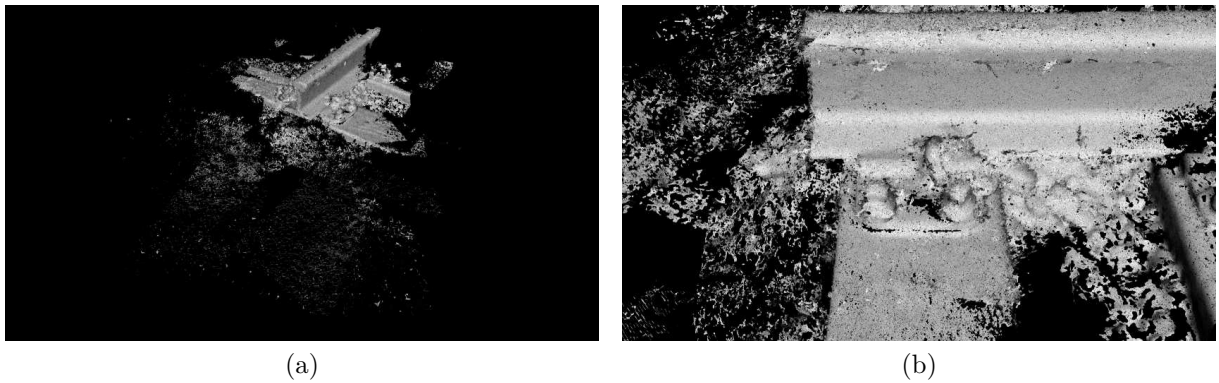


Figura 73 – Resultado da superfície do VSFM + DEN + PCL-GY. (a) Vista geral. (b) Vista detalhe.

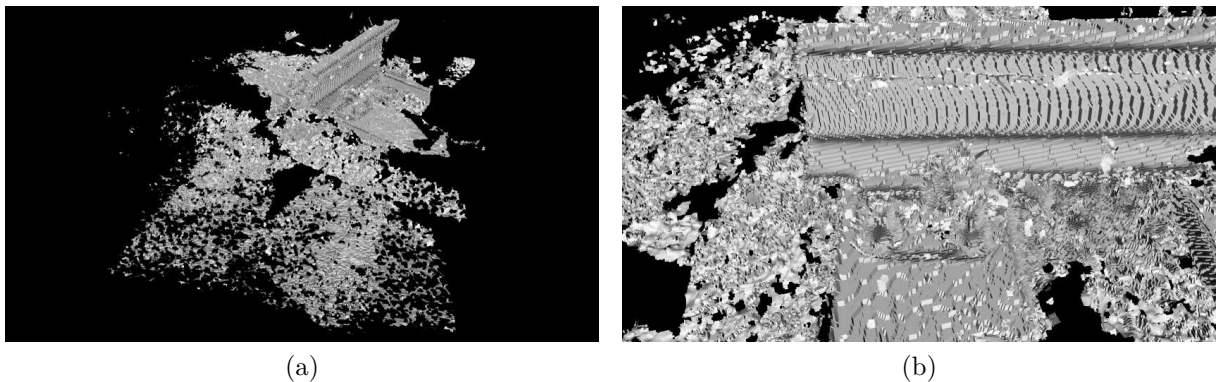


Figura 74 – Resultado da superfície do VSFM + DEN + PCL-GD. (a) Vista geral. (b) Vista detalhe.

Essa imagem é utilizada quando o método não consegue encontrar uma câmera com visão da face, sendo assim ele não consegue texturizar essa face. Os resultados quantitativos podem ser observados na Tabela 5.



Figura 75 – Resultado da texturização do Metashape. (a) Vista detalhada. (b) Vista geral.

Na Figura 75 tem-se o resultado do *software* Metashape. Esse resultado teve uma boa

Método	Tempo	Número de faces texturizadas	Memória RAM	Memória VRAM
Metashape	5min	197.671	14.5GB	-
Meshroom	36s	627.345	3.7GB	-
3Dflow	1min 34s	187.784	3.3GB	-
VSFM + MR + TR	2min 43s	934.362	2.1GB	-
VSFM + CMP-MVS	38min	4.488.603	5.2GB	2.5GB
SFM-A + DEN + PR + TR	1min 28s	478.014	1.7GB	-
SFM-S + DEN + PR + TR	2min 55s	1.169.793	1.7GB	-
VSFM + DEN + PR + TR	2min 15s	876.106	1.7GB	-
VSFM + P/C + PR + TR	1min 55s	791.127	1.5GB	-
SFM-A + DEN + PR + PCL_T	50s	483.688	684MB	-
SFM-S + DEN + PR + PCL_T	2min	1.188.269	1GB	-
VSFM + DEN + PR + PCL_T	1min 34s	912.430	900MB	-
VSFM + P/C + PR + PCL_T	1min 18s	805.619	880MB	-

Tabela 5 – Resultados quantitativos para a etapa da texturização para o dataset do trilho.

qualidade de textura, mas o tempo foi o maior de todos os métodos de texturização, e o uso da memória RAM foi muito alto.



Figura 76 – Resultado da texturização do Meshroom. (a) Vista detalhada. (b) Vista geral.

Na Figura 76 tem-se o resultado do *software* Meshroom. Esse resultado teve uma qualidade mediana, esse método não conseguiu lidar com a diferença de iluminação entre as imagens, além de adicionar um efeito de borrão em algumas partes da reconstrução.

Na Figura 77 tem-se o resultado do *software* 3Dflow. Esse foi o método que conseguiu lidar da melhor forma com as mudanças de iluminação. O topo do trilho seguiu uma iluminação uniforme, sem mudanças bruscas, como pode ser observado em outros testes.

Na Figura 78 tem-se o resultado da combinação VisualSFM + MeshRecon + TexRecon. Esse método de texturização teve um bom resultado, mas o topo do trilho acabou ficando "manchado" por essas diferenças. Apesar disto, a transição entre essas áreas é



Figura 77 – Resultado da texturização do 3Dflow. (a) Vista detalhada. (b) Vista geral.



Figura 78 – Resultado da texturização do VSFM + MR + TR. (a) Vista detalhada. (b) Vista geral.

suave. Outro ponto interessante é que ao contrário de outros métodos de texturização, o TexRecon não distorce as imagens para preencher as faces que ele não tem visão, isso evita que faces que podem ser texturizadas percam qualidade para evitar faces sem textura.



Figura 79 – Resultado da texturização do VSFM + CMP-MVS. (a) Vista detalhada. (b) Vista geral.

Na Figura 79 tem-se o resultado da combinação VisualSFM + CMP-MVS. O tempo desse método acabou sendo o maior de todos na etapa de texturização, isso acontece pois ele não separa as etapas de geração de nuvem densa, geração de superfície e texturização,

apesar disso, ele conseguiu um consumo de recursos menor que o Metashape. O resultado da texturização não ficou muito bom, pois ele adicionou um efeito de borrão em toda a textura, misturou as iluminações no topo do trilho, e texturizou o gramado com uma cor sólida, sem nenhum detalhe.



Figura 80 – Resultado da texturização do SFM-A + DEN + PR + TR. (a) Vista detalhada. (b) Vista geral.

Na Figura 80 tem-se o resultado da combinação SFM-AKAZE + DENSE + Poisson-Recon + TexRecon. O resultado dessa combinação não ficou muito bom, já que o erro ocorrido na etapa de estimação da postura das câmeras acabou se propagando para todas as etapas.



Figura 81 – Resultado da texturização do SFM-S + DEN + PR + TR. (a) Vista detalhada. (b) Vista geral.

Na Figura 81 tem-se o resultado da combinação SFM-SIFT + DENSE + PoissonRecon + TexRecon. Essa combinação teve um resultado bom, mas ruídos da etapa da geração da nuvem densa fizeram com que algumas partes do trilho foram texturizadas com o gramado.

Na Figura 82 tem-se o resultado da combinação VisualSFM + DENSE + PoissonRecon + TexRecon. Essa combinação teve um resultado bom, os ruídos da etapa de geração de nuvem densa não influenciaram negativamente na texturização.



Figura 82 – Resultado da texturização do VSFM + DEN + PR + TR. (a) Vista detalhada. (b) Vista geral.



Figura 83 – Resultado da texturização do VSFM + P/C + PR + TR. (a) Vista detalhada. (b) Vista geral.

Na Figura 83 tem-se o resultado da combinação VisualSFM + PMVS/CMVS + PoissonRecon + TexRecon. Essa combinação teve um bom resultado, mas fica evidente que ocorreram erros na geração da nuvem densa, pois a ponta do trilho foi texturizada com o gramado.



Figura 84 – Resultado da texturização do SFM-A + DEN + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.

Na Figura 84 tem-se o resultado da combinação SFM-AKAZE + DENSE + Poisson-Recon + PCL_TEXT. A texturização com o método do PCL_TEXT não conseguiu um bom resultado, devido aos vários recortes que apareceram no meio da textura, além de borrões e mudanças bruscas de iluminação.



Figura 85 – Resultado da texturização do SFM-S + DEN + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.

Na Figura 85 tem-se o resultado da combinação SFM-SIFT + DENSE + Poisson-Recon + PCL_TEXT. Novamente tem-se o problema dos recortes na textura, além da texturização indevida do trilho com o gramado.

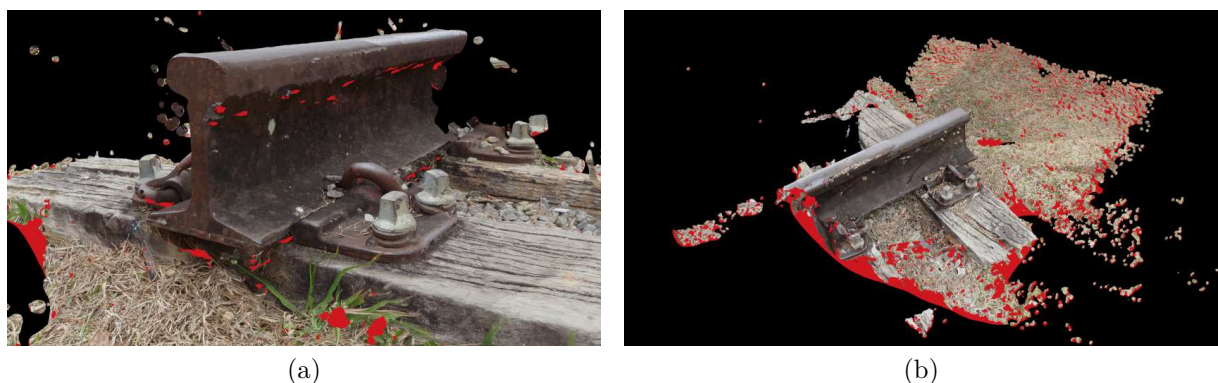


Figura 86 – Resultado da texturização do VSFM + DEN + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.

Na Figura 86 tem-se o resultado da combinação VisualSFM + DENSE + PoissonRecon + PCL_TEXT. Novamente tem-se o problema dos recortes na textura e apesar da superfície aparentar estar correta, o método ainda sim texturizou o trilho com a grama.

Na Figura 87 tem-se o resultado da combinação VisualSFM + PMVS/CMVS + PoissonRecon + PCL_TEXT. Os recortes na textura não ficaram tão visíveis nessa reconstrução, mas o topo do trilho sofreu com mais alterações de iluminação.



Figura 87 – Resultado da texturização do VSFM + P/C + PR + PCL_T. (a) Vista detalhada. (b) Vista geral.

Categoria	Método	Tempo total	Pico de memória RAM	Pico de memória VRAM
A	SFM-A + DEN + PR + TR	1hr 41min	8.8GB	-
A	SFM-S + DEN + PR + TR	2hr 22s	12.8GB	-
B	Meshroom	21min 52s	3.8GB	1.6GB
B	VSFM + MR + TR	6min 29s	2.1GB	2.1GB
B	VSFM + CMP-MVS	38min 58s	5.2GB	2.5GB
B	VSFM + DEN + PR + TR	4hr 7s	11.1GB	709MB
B	VSFM + P/C + PR + TR	6min 50s	1.8GB	709MB
C	3Dflow	6min 30s	3.3GB	-
C	Metashape	7min 25s	14.5GB	-

Tabela 6 – Resultados quantitativos para a reconstrução 3D do dataset do trilho.

4.6 Datasets adicionais

Para aprofundar as comparações foram testados alguns datasets adicionais. Eles foram testados utilizando as melhores combinações de algoritmos do experimento inicial feito com o trilho de trem. As combinações de algoritmos utilizadas no dataset inicial foram classificadas em 3 categorias: a categoria A contém os algoritmos desenvolvidos neste trabalho; a categoria B contém os *softwares*/bibliotecas livres; e a categoria C os *softwares* pagos. Posteriormente foram escolhidas as melhores combinações de cada categoria, utilizando como critério o tempo total, o consumo de memória RAM e VRAM, além da qualidade da reconstrução. Os dados utilizados nas comparações podem ser vistos na Tabela 6.

A escolha do melhor algoritmo da categoria A foi simples, pois apesar da combinação SFM-SIFT + DENE + PoissonRecon + TexRecon ter tido um tempo e consumo de RAM maiores que a combinação SFM-AKAZE + DENE + PoissonRecon + TexRecon, teve um resultado muito melhor, pois no caso do SFM-AKAZE o trilho teve uma grande falha no centro, como pode ser visto na Figura 65. Para a categoria B o primeiro quesito

para a eliminação foi o tempo total, já que essa categoria possui uma grande variação de tempo. Foram eliminadas as combinações VisualSFM + DENSE + PoissonRecon + TexRecon, VisualSFM + CMP-MVS e o Meshroom. Apesar das combinações VisualSFM + MeshRecon + TexRecon e VisualSFM + PMVS/CMVS + PoissonRecon + TexRecon apresentaram consumo de tempo e de memória similares. A combinação VisualSFM + MeshRecon + TexRecon foi escolhida já que seu resultado conseguiu criar uma reconstrução com mais detalhes. Para a categoria C o maior diferencial foi o pico de uso de memória RAM, que foi muito mais alto no Metashape. Além disso, o 3Dflow conseguiu recriar uma boa parte do gramado, enquanto o Metashape ignorou essa área, desta forma para essa categoria o *software* escolhido foi o 3Dflow.

4.6.1 Gárgula

Na Figura 88 tem-se o resultado da combinação SFM-SIFT + DENSE + PoissonRecon + TexRecon para o dataset da gárgula. Pode-se perceber que apesar dos ruídos, a combinação conseguiu uma quantidade de detalhes parecida com os outros resultados. Uma falha dessa combinação foi na parte inferior traseira da base da gárgula, que ficou deformada. A maior parte dos ruídos que aparece entre as asas e a cabeça da gárgula foram causados por falsos positivos de tentativas de *matches* com os objetos que aparecem no fundo da cena. Esse ruídos são menos comuns quando se tem o fundo branco, como na base da gárgula, demonstrando a importância de escolher um fundo que não atrapalhe a reconstrução.

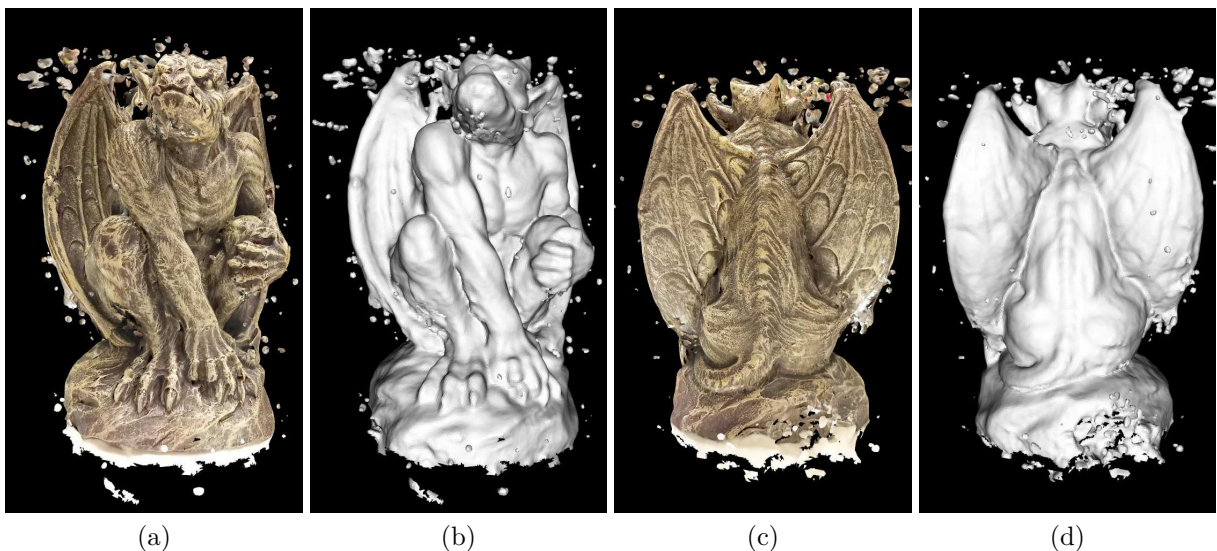


Figura 88 – Resultado da reconstrução 3D da gárgula com a combinação SFM-S + DEN + PR + TR. (a) Vista frontal com textura. (b) Vista frontal sem textura. (c) Vista posterior com textura. (d) Vista posterior sem textura.

Na Figura 89 tem-se o resultado da combinação VisualSFM + MeshRecon + TexRecon

para o dataset da gárgula. Novamente, as reconstruções feitas com o MeshRecon trazem um nível de detalhes muito alto. Isto fica nítido nos pés e na coluna da gárgula, que trouxeram detalhes que foram suprimidos nos outros resultados.



Figura 89 – Resultado da reconstrução 3D da gárgula com a combinação VSFM + MR + TR. (a) Vista frontal com textura. (b) Vista frontal sem textura. (c) Vista posterior com textura. (d) Vista posterior sem textura.

Na Figura 90 tem-se o resultado do *software* 3Dflow para o dataset da gárgula. Esse resultado conseguiu um bom nível de detalhes, mas acabou adicionado um ruído acima da asa da gárgula.

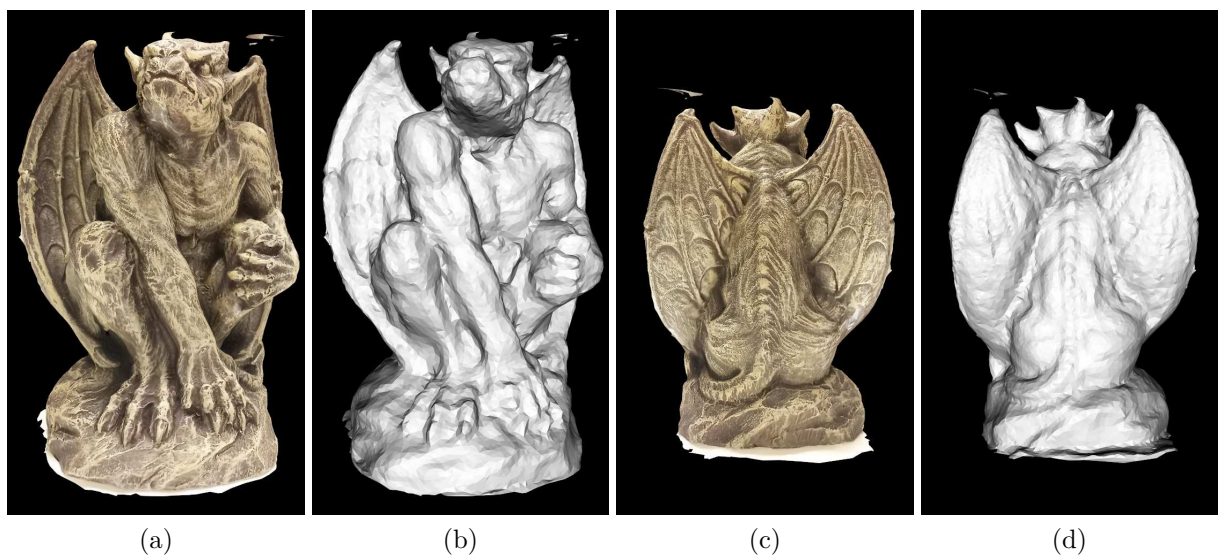


Figura 90 – Resultado da reconstrução 3D da gárgula com o *software* 3Dflow. (a) Vista frontal com textura. (b) Vista frontal sem textura. (c) Vista posterior com textura. (d) Vista posterior sem textura.

Método	Tempo total	Pico de memória RAM	Pico de memória VRAM
3Dflow	2min 45s	1GB	-
VSFM + MR + TR	2min 13s	1GB	1.2GB
SFM-S + DEN + PR + TR	1hr 19min	5.4GB	-

Tabela 7 – Resultados quantitativos para a reconstrução 3D do dataset da gárgula.

Os resultados quantitativos desses testes podem ser vistos na Tabela 7. Analisando-se o resultado e o tempo, a combinação VisualSFM + MeshRecon + TexRecon foi a melhor combinação, principalmente pelo nível de detalhes de suas reconstruções. O 3Dflow vem em segundo lugar pelo maior tempo mas menor consumo de recursos, e a combinação SFM-SIFT + DENSE + PoissonRecon + TexRecon por último pelo tempo muito maior que os outros algoritmos.

4.6.2 Fonte

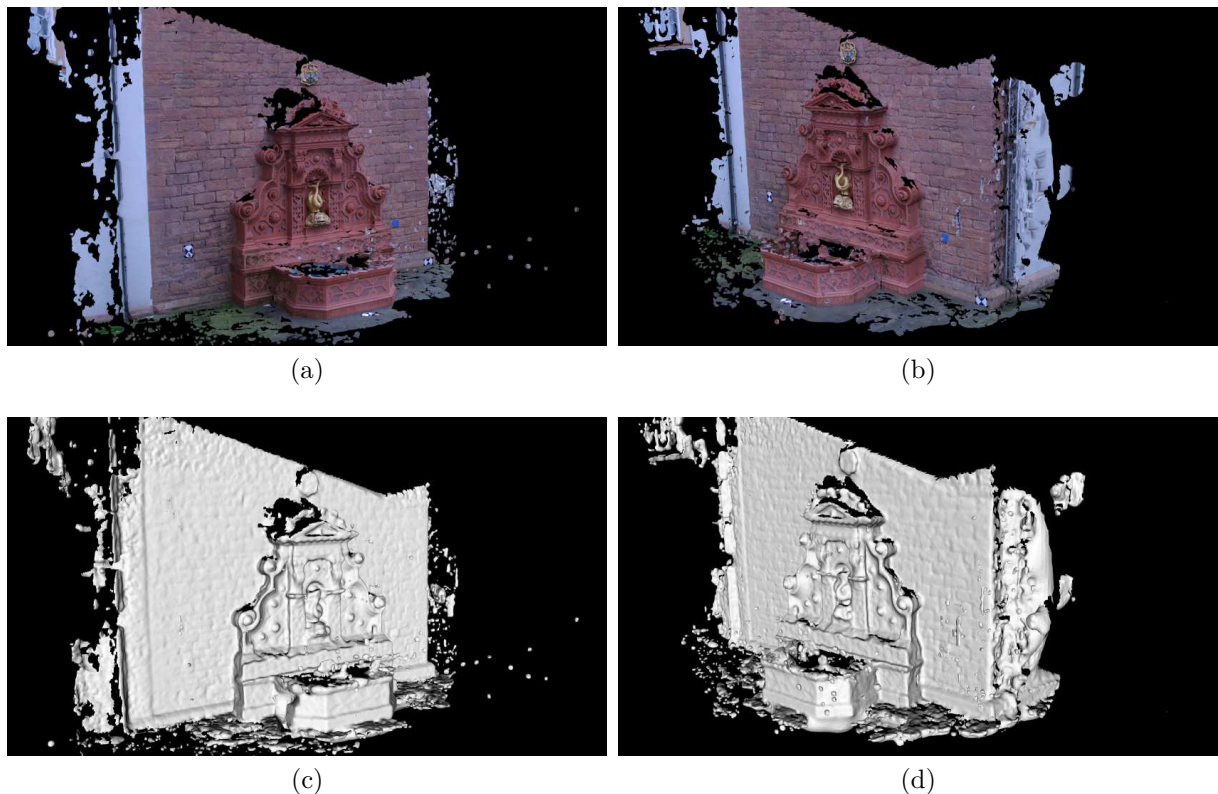


Figura 91 – Resultado da reconstrução 3D da fonte com a combinação SFM-S + DEN + PR + TR. (a) Vista lateral esquerda com textura. (b) Vista lateral direita com textura. (c) Vista lateral esquerda sem textura. (d) Vista lateral direita sem textura.

$$K_{SFM-SIFT} = \begin{bmatrix} 2770.53 & 0 & 1536 \\ 0 & 2770.53 & 1024 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.2)$$

Na Figura 91 tem-se o resultado da combinação SFM-SIFT + DENSE + Poisson-Recon + TexRecon para o dataset da fonte. Pode-se perceber que apesar dos ruídos, a combinação conseguiu uma quantidade de detalhes parecida com os outros resultados, principalmente as divisões das pedras na parede. Uma falha dessa combinação foi o canto esquerdo da fonte, que ficou deformado. O SFM-SIFT trata cada câmera como única, desta forma tem-se mais de uma matriz intrínseca, mas a média das matrizes pode ser vista na Equação 4.2. Esse método não separa a distância focal x e y, além de não estimar o centro da câmera.

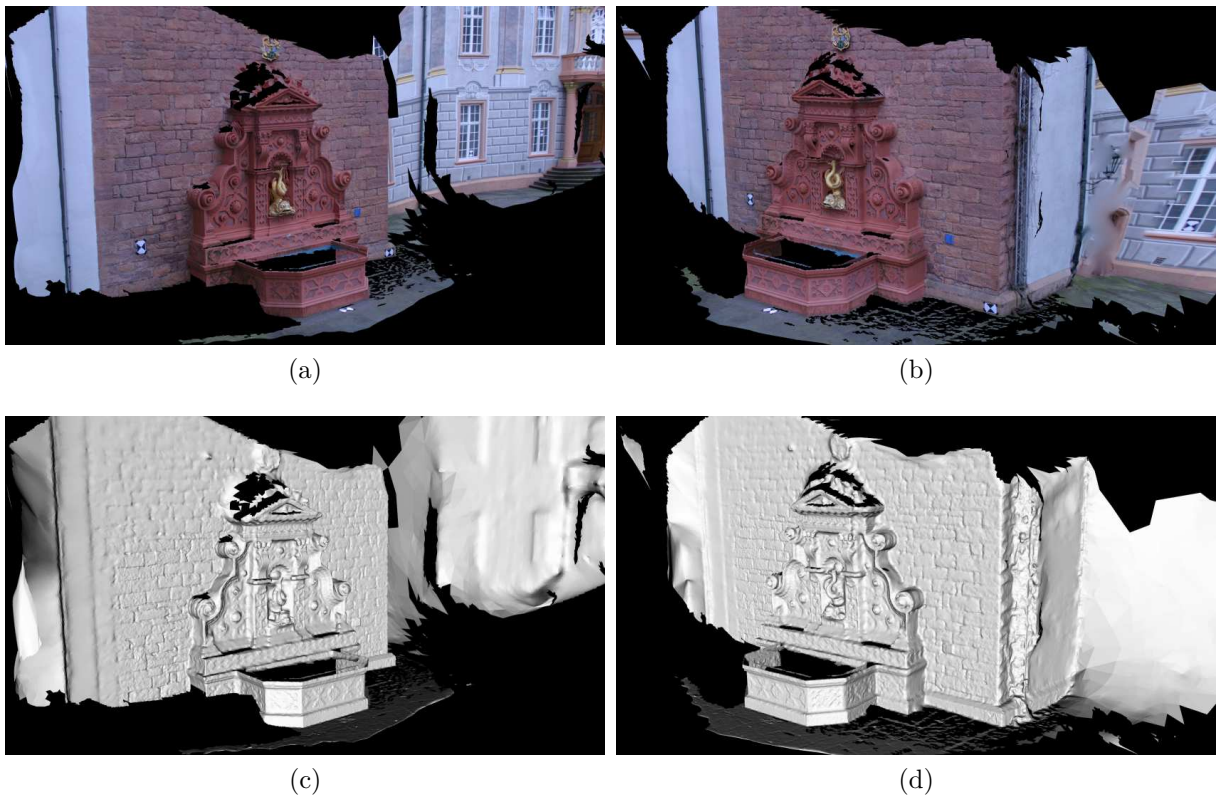


Figura 92 – Resultado da reconstrução 3D da fonte com a combinação VSFM + MR + TR. (a) Vista lateral esquerda com textura. (b) Vista lateral direita com textura. (c) Vista lateral esquerda sem textura. (d) Vista lateral direita sem textura.

$$K_{VisualSFM} = \begin{bmatrix} 2765.51 & 0 & 1536 \\ 0 & 2765.51 & 1024 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.3)$$

Na Figura 92 tem-se o resultado da combinação VisualSFM + MeshRecon + TexRecon para o dataset da fonte. Os reconstruções feitas com o MeshRecon trazem um nível de detalhes muito alto, as divisões das pedras na parede ficam mais destacadas, além das inscrições na base da fonte. Outro ponto positivo é que essa combinação conseguiu reconstruir de forma coerente as janelas do prédio que aparece ao fundo das imagens. Como ponto negativo destaca-se a falta do canto esquerdo da fonte. O VisualSFM trata cada câmera como única, desta forma tem-se mais de uma matriz intrínseca, mas a média das matrizes pode ser vista na Equação 4.3. Esse método não separa a distância focal x e y , além de não estimar o centro da câmera.

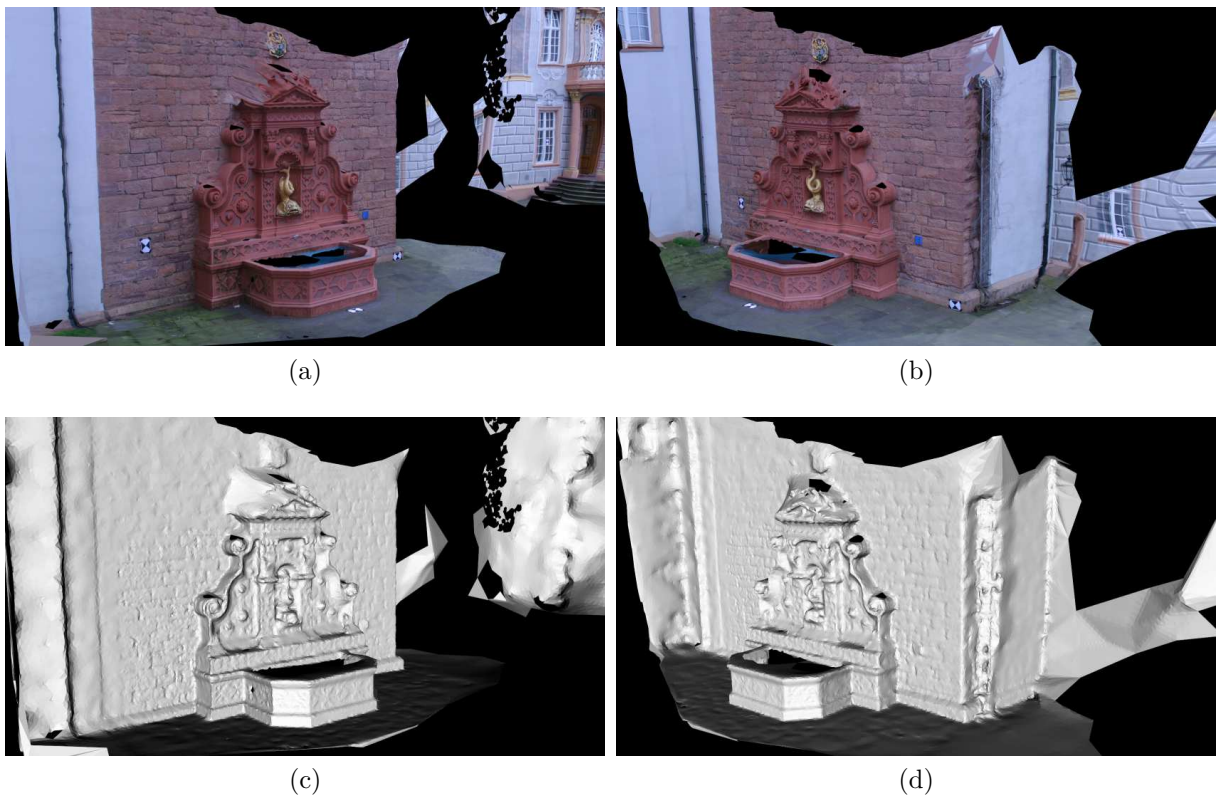


Figura 93 – Resultado da reconstrução 3D da fonte com o *software* 3Dflow. (a) Vista lateral esquerda com textura. (b) Vista lateral direita com textura. (c) Vista lateral esquerda sem textura. (d) Vista lateral direita sem textura.

$$K_{3Dflow} = \begin{bmatrix} 2758.98 & 0 & 1516.25 \\ 0 & 2758.98 & 1003.78 \\ 0 & 0 & 1 \end{bmatrix} \quad (4.4)$$

Na Figura 93 tem-se o resultado do *software* 3Dflow para o dataset da fonte. Esse foi o resultado que conseguiu reconstruir a maior parte da fonte, com uma boa qualidade, apesar de também apresentar uma falha no canto esquerdo da fonte. O prédio que aparece no fundo das imagens também foi reconstruído, mas o resultado não ficou bom. Como ponto negativo tem-se o topo da fonte: na tentativa de evitar buracos na superfície o

Método	Tempo total	Pico de memória RAM	Pico de memória VRAM
3Dflow	5min 30s	3GB	-
VSFM + MR + TR	4min 33s	1.6GB	1.7GB
SFM-S + DEN + PR + TR	5hr 36min	12.6GB	-

Tabela 8 – Resultados quantitativos para a reconstrução 3D do dataset da fonte.

software alongou a textura para preencher o espaço, o que traz um aspecto pior que apenas um buraco. O 3Dflow é o único algoritmo, entre os três escolhidos, que interpreta que todas as imagens foram capturadas por uma única câmera, então tem-se apenas uma matriz intrínseca que pode ser vista na Equação 4.4. Esse método não separa a distância focal x e y , mas estima o centro da câmera e as distorções radiais.

Os resultados quantitativos desses testes podem ser vistos na Tabela 8. Analisando-se o resultado e o tempo, a combinação VisualSFM + MeshRecon + TexRecon foi a melhor combinação, principalmente pelo nível de detalhes de suas reconstruções. O 3Dflow vem em segundo lugar pelo maior tempo mas menor consumo de recursos, e a combinação SFM-SIFT + DENSE + PoissonRecon + TexRecon por último pelo tempo muito maior que os outros algoritmos. Entre a comparação das matrizes intrínsecas, o melhor resultado ficou com o 3Dflow, por ser o único método a estimar o centro da câmera.

5 Conclusões

5.1 Considerações finais

Os resultados da etapa de estimação de postura mostraram uma clara vantagem do Metashape, cujo resultado deu-se em um tempo muito baixo além de consumir pouquíssimos recursos de *hardware*. O fato de sua nuvem de pontos esparsa ter poucos pontos demonstrou não ser um problema, já que essa nuvem não é utilizada nas próximas etapas. O resultado do Meshroom não foi satisfatório, pois apesar de acertar a posição das câmeras utilizou mais memória RAM que os algoritmos desenvolvidos neste trabalho, que ainda não estão totalmente otimizados. O resultado do 3Dflow ficou interessante, apesar de não conseguir ter um tempo tão baixo como o Metashape, conseguiu estimar a posição das câmeras com quase a metade dos pontos utilizados pelo Metashape, o que demonstra a robustez do método de estimação. O resultado do VisualSFM foi bom, pois conseguiu em pouco tempo uma nuvem com uma grande quantidade de pontos, consumindo poucos recursos. A velocidade do VisualSFM está ligada ao uso da GPU para as etapas de detecção e *matching* das *features*. O resultado do SFM-SIFT teve um consumo muito grande de memória RAM e tempo de processamento, que se deve principalmente pela grande quantidade de correspondências encontradas através do SIFT. O resultado do SFM-AKAZE foi satisfatório, pois conseguiu consumir menos recursos de *hardware* e tempo que o SFM-SIFT, e encontrou um resultado similar. O tempo do SFM-AKAZE foi prejudicado por apenas possuir o *matcher* de força bruta, que é mais lento que o *matcher* baseado no FLANN utilizado no SFM-SFIT. O SFM-SIFTGPU foi descartado para as próximas etapas pois ele falhou em estimar a posição das câmeras. Na Figura 54b pode-se perceber que uma fileira inteira de câmeras foi posicionada de forma incorreta. Esse erro aconteceu pela baixa quantidade de correspondências encontradas pelo SIFTGPU.

Para a etapa de geração de nuvem densa o Metashape teve um bom tempo e consumo de recursos, sendo o algoritmo que utilizou menos memória RAM. Seu resultado, além de limitar a área da reconstrução, acabou falhando nas áreas com pontos luminosos no topo do trilho, criando buracos onde deveria existir uma superfície lisa. O resultado do 3dFlow ficou bom. Foi o mais rápido e teve um baixo consumo de memória RAM, além de ter reconstruído uma grande parte do gramado. Os testes realizados com o DENSE tiveram resultados parecidos entre si, com um tempo de execução e consumo de memória RAM altos. A grande quantidade de pontos demonstrou não acrescentar mais informação pois a maioria descrevia a mesma região. Esse efeito de super amostragem cria um problema para os algoritmos de geração da superfície, principalmente para os métodos incrementais como o *Greedy Projection Triangulation*. A diferença da quantidade de pontos gerados entre

as combinações SFM-AKAZE + DENSE, SFM-SIFT + DENSE e VisualSFM + DENSE mostra a qualidade do processo de estimação da postura das câmeras, quanto maior o número de pontos melhor a estimação. Pois com as câmeras alinhadas corretamente os erros na retificação estéreo serão menores, fazendo com que mais pontos sejam gerados. O resultado do PMVS/CMVS ficou interessante, teve um tempo baixo, mas gerou menos pontos e consumiu mais memória RAM que o Metashape e o 3Dflow, além de ser o único algoritmo que falhou na reconstrução de uma das tábuas.

A etapa de geração da superfície foi a que trouxe os mais variados resultados. O Metashape não obteve um bom resultado. Além dos erros do topo do trilho originados na etapa de geração da nuvem densa, ele não conseguiu manter os detalhes das tábuas e das inscrições no ferro. O Meshroom teve um tempo superior pois como já foi comentado, ele não separa as etapas de geração da nuvem densa e a geração da superfície. Seu consumo de recursos também ficou alto, além de apresentar o mesmo problema de buracos no topo do trilho que o Metashape. O 3Dflow teve um bom resultado, conseguiu reconstruir grande parte do gramado e manteve alguns detalhes das tábuas, isso com um baixo tempo e consumo de memória RAM. O MeshRecon teve o melhor resultado. Assim como o Meshroom ele não distingue as etapas de geração da nuvem densa e geração da superfície. Apesar disso teve um baixo tempo e alto número de faces, sua superfície foi a que teve mais detalhes. Os testes realizados com o PoissonRecon tiveram um baixo tempo de execução e um alto número de faces. Pode-se perceber que o consumo de memória RAM está diretamente ligado com o tamanho da nuvem de pontos de entrada, pois o VisualSFM + DENSE consumiu mais memória que a combinação VisualSFM + PMVS/CMVS. Esse método também se mostrou o melhor para lidar com o problema da super amostragem criado pelo algoritmo DENSE. Os testes com o método PCL-Greedy não foram satisfatórios, como mencionado anteriormente, por ele ser incremental necessita analisar ponto a ponto, criando um processo muito custoso para nuvens de alta densidade, além de não ter a capacidade de eliminar ruídos. Os testes com o método PCL-Grid também não foram satisfatórios, nenhum dos resultados criou uma superfície suave, além de ter criado várias áreas desconexas. Por esses motivos os resultados dos métodos PCL-Greedy e PCL-Grid não seguiram para a próxima etapa.

Para a etapa de texturização o Metashape teve um bom resultado, mas seu tempo foi o maior entre os métodos que executam apenas a etapa de texturização, além de ter o maior consumo de memória RAM. O Meshroom teve o melhor tempo entre os algoritmos, mas não conseguiu lidar corretamente com as diferenças em iluminação entre as imagens e adicionou muitos borrões na textura. O 3Dflow teve um ótimo resultado, foi o único que conseguiu evitar o problema da mudança de iluminação no topo do trilho, texturizando-o com apenas um tom, além disso, não adicionou efeitos de suavização em outras áreas. Os testes realizados com o TexRecon tiveram um resultado consistente, o uso de memória RAM não variou tanto com o número de faces, apenas seu tempo foi alterado. O topo do

trilho foi texturizado com imagens contendo diferentes iluminações, mas teve transições suaves entre elas, o que amenizou o problema. Os testes feitos com o PCL_TEXT tiveram os piores resultados, em todos os testes apareceram cortes na textura, além de áreas totalmente borradas e distorcidas, por esse motivo esses resultados foram descartados na análise final.

Após os testes separados de cada etapa com o conjunto de imagens do trilho, foi feito o teste com as melhores combinações dos algoritmos com conjuntos de imagens adicionais, a fonte e a gárgula. Os resultados obtidos com os datasets adicionais mantiveram as observações feitas com o dataset do trilho. A combinação VisualSFM + MeshRecon + TexRecon continua a ser a melhor alternativa para uma reconstrução rápida, com pouco consumo de recursos de *hardware* e rica em detalhes. O 3Dflow é uma boa opção por ter um tempo baixo, um bom resultado e consumir apenas memória RAM. Os algoritmos que consomem a memória VRAM, em especial o MeshRecon, aumentam muito o consumo com imagens de maior resolução e com datasets maiores. Isso é um problema pois a memória VRAM é mais cara e difícil de ser expandida que a memória RAM. Outro ponto interessante do 3Dflow é que sua licença é gratuita para projetos com menos de 50 imagens. Isso faz dele útil para pequenos testes, mas datasets profissionais usualmente ultrapassam 150 imagens, sendo necessário comprar uma licença do *software*. Por fim tem-se a combinação dos algoritmos desenvolvidos neste trabalho. Apesar do tempo e do consumo de memória serem muito maiores, o resultado não ficou tão distante dos outros algoritmos. Os ruídos foram o principal ponto negativo, mas a maioria deles pode ser removida por um filtro que elimina elementos desconectados do maior conjunto de faces, caso o filtro fosse aplicado já na superfície.

5.2 Principais contribuições

A principal contribuição deste trabalho foi a documentação das etapas da reconstrução 3D, fazendo com que cada etapa fosse descrita de maneira mais clara. Com essa descrição foi possível a implementação dos algoritmos da etapa de estimação da postura das câmeras e da geração da nuvem densa. Como esses algoritmos foram implementados utilizando bibliotecas de código aberto, pode-se reproduzir os resultados obtidos neste trabalho seguindo as explicações do Capítulo 3. Os algoritmos também estão disponíveis através do repositório público: <https://github.com/julianomasson/TCC>.

As comparações feitas no Capítulo 4 são úteis para a decisão de qual *software*/combinação de algoritmos utilizar para projetos de reconstrução 3D. Pois muitas vezes o processo de instalação/compilação dos algoritmos não é trivial e pode tomar muito tempo. A análise feita com diferentes datasets em condições diferentes pode auxiliar nesta decisão.

5.3 Trabalhos futuros

Para o algoritmo desenvolvido na etapa da estimação da postura das câmeras foram propostas três abordagens, com os detectores de *features* AKAZE, SIFT e o SIFTGPU. A implementação do AKAZE poderia ter um ganho significativo no seu tempo se o *matcher* utilizado fosse igual o do SIFT, baseado no FLANN, ao invés de utilizar o *matcher* força bruta. A implementação do SIFTGPU poderia ser revista, e se possível também utilizar outro *matcher*, já que correspondências erradas foram o principal fator de falha dessa abordagem. Rever como as *Tracks* relacionadas (encontradas em vários pares de imagens) afetam o processo do *bundle adjustment*, pois se percebeu que relacionar todas as *Tracks* acaba piorando o resultado. Isso que fez com que as *Tracks* da lista de pares adicionais não fossem relacionadas. Outro ponto que iria acelerar o processo da estimação da postura das câmeras seria aumentar a mínima melhora necessária no erro de re-projeção da cena (*Graph*) para a continuação do processo de *bundle adjustment*. Muitas vezes esse processo de otimização continua por vários segundos sem uma melhora significativa no erro de re-projeção. A adição de um novo par de imagens na cena, que adicionaria novas variáveis e restrições para o *bundle adjustment*, faria com que ele convergisse para um erro de re-projeção menor mais rápido. Apesar do processamento *multithread* ter sido utilizado em algumas tarefas, ele poderia ter sido mais utilizado se alguma biblioteca específica para isto fosse utilizada, disponibilizando listas com leitura e escrita seguras nessas condições. Para facilitar o desenvolvimento e a validação dessa etapa, seria interessante utilizar um conjunto de imagens que disponibilize a posição real das câmeras, de modo que seja possível verificar o impacto de cada mudança do algoritmo no resultado final.

Para o algoritmo desenvolvido na etapa de geração da nuvem densa, seu ponto negativo quando comparado a outros algoritmos é o alto tempo de execução. A principal causa desta lentidão é a mesma da super amostragem da nuvem de pontos, cada par de imagens é tratado como uma câmera estereoscópica. Gerando sua própria nuvem de pontos, que precisa ser filtrada e passar pela etapa de cálculo da normal. Um exemplo dos tempos das etapas principais para a geração da nuvem de pontos de um par podem ser vistos na Tabela 9 (as etapas seguem a nomenclatura do fluxograma na Figura 36), pode-se perceber que os processos de filtragem e cálculo das normais tomam quase o mesmo tempo que todo o processo até a geração da nuvem de pontos. Uma maneira de resolver o problema do tempo e da super amostragem seria reconstruir apenas as *features* encontradas em pelo menos três imagens. Isso diminuiria a quantidade de *features* e aumentaria a confiança no cálculo do ponto 3D, já que ele teria que ter um erro de re-projeção baixo em três câmeras.

Outra opção seria substituir esse filtro, que é implementado no PCL e não utiliza o processamento *multithread*, por um filtro que analisa o erro de re-projeção em uma terceira imagem, mesmo ela não tendo visão do ponto. Apesar de não parecer lógico, as restrições

Etapa	Tempo
Detectar os <i>keypoints</i> e Realizar o <i>matching</i>	4s
Calcular as novas <i>Seeds</i>	5min 4s
Gerar a nuvem de pontos	30s
Filtrar a nuvem de pontos	34s
Calcular as normais	4min 29s

Tabela 9 – Exemplo dos tempos de algumas etapas para a geração da nuvem de pontos de um par na etapa de geração da nuvem densa.

da geometria epipolar ainda são válidas. Considere dois pares de imagens, as imagens 1 e 2 (par 1) e as imagens 2 e 3 (par 2). O ponto 3D p foi gerado pelo par 1 e a imagem 3 não tem visão desse ponto. Reprojetoando o ponto p no plano da imagem 3 ele ficaria fora dos limites da imagem. Mas como se tem a relação entre as imagens 2 e 3 e a coordenada do ponto p na imagem 2, pode-se desenhar a linha epipolar do ponto p na imagem 3. Se as imagens estiverem retificadas, a linha epipolar não tem variações em y . Desta forma, apenas a coordenada em y do ponto reprojetoado importa, se ela estiver dentro de um limite de proximidade da linha epipolar, o ponto é válido.

Outra maneira de abordar o processo de cálculo das novas *Seeds* seria avaliar apenas os pixels mais externos da janela de procura (Figura 40). Isso ajudaria a reduzir a super amostragem do algoritmo, pois o objetivo não é criar uma nuvem de pontos tão densa quanto uma superfície fechada (caso ideal em que todos os pontos visíveis pelas duas câmeras encontram correspondência). O objetivo dessa etapa é analisar toda imagem em busca de correspondências, de modo que a densidade da nuvem possa ser suficiente para criar um bom resultado com os métodos de geração de superfície.

Referências Bibliográficas

- 1 G., G. C. e B. 2d projective transformations (homographies). 2012.
- 2 HARTLEY, R.; ZISSERMAN, A. *Multiple View Geometry in Computer Vision*. [S.l.]: Cambridge University Press, 2004.
- 3 SZELISKI, R. *Computer vision: algorithms and applications*. [S.l.]: Springer Science & Business Media, 2010.
- 4 CORKE, P. *Robotics, Vision and Control: Fundamental Algorithms In MATLAB® Second, Completely Revised, Extended And Updated Edition*. [S.l.]: Springer International Publishing, 2017. (Springer Tracts in Advanced Robotics). ISBN 9783319544137.
- 5 SIEGWART, R.; NOURBAKHSI, I. R.; SCARAMUZZA, D. *Introduction to autonomous mobile robots*. [S.l.]: MIT press, 2011.
- 6 KHOSHELHAM, K. Accuracy analysis of kinect depth data. In: *ISPRS workshop laser scanning*. [S.l.: s.n.], 2011. v. 38, n. 5, p. W12.
- 7 LOWE, D. G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, Springer, v. 60, n. 2, p. 91–110, 2004.
- 8 BAY, H. et al. Speeded-up robust features (surf). *Computer vision and image understanding*, Elsevier, v. 110, n. 3, p. 346–359, 2008.
- 9 ALCANTARILLA, P. F.; SOLUTIONS, T. Fast explicit diffusion for accelerated features in nonlinear scale spaces. *IEEE Trans. Patt. Anal. Mach. Intell*, v. 34, n. 7, p. 1281–1298, 2013.
- 10 KAZHDAN, M.; BOLITHO, M.; HOPPE, H. Poisson surface reconstruction. In: *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2006. (SGP '06), p. 61–70. ISBN 3-905673-36-3.
- 11 GOPI, M.; KRISHNAN, S. A fast and efficient projection-based approach for surface reconstruction. In: *IEEE. Computer Graphics and Image Processing, 2002. Proceedings. XV Brazilian Symposium on*. [S.l.], 2002. p. 179–186.
- 12 ATKINSON, K. *Close Range Photogrammetry and Machine Vision*. [S.l.]: Whittles, 1996. ISBN 9781870325462.
- 13 WANG, J.; OLSON, E. AprilTag 2: Efficient and robust fiducial detection. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. [S.l.: s.n.], 2016.
- 14 FISCHLER, M. A.; BOLLES, R. C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, ACM, v. 24, n. 6, p. 381–395, 1981.

- 15 STIVANELLO, M. E. et al. Algoritmos para medição de superfícies em movimento usando visão 3d. 2013.
- 16 TUYTELAARS, T.; MIKOLAJCZYK, K. et al. Local invariant feature detectors: a survey. *Foundations and trends® in computer graphics and vision*, Now Publishers, Inc., v. 3, n. 3, p. 177–280, 2008.
- 17 LOWE, D. G. Object recognition from local scale-invariant features. In: IEEE. *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. [S.l.], 1999. v. 2, p. 1150–1157.
- 18 BAY, H.; TUYTELAARS, T.; GOOL, L. V. Surf: Speeded up robust features. In: SPRINGER. *European conference on computer vision*. [S.l.], 2006. p. 404–417.
- 19 ALCANTARILLA, P. F.; BARTOLI, A.; DAVISON, A. J. Kaze features. In: SPRINGER. *European Conference on Computer Vision*. [S.l.], 2012. p. 214–227.
- 20 YANG, X.; CHENG, K.-T. Ldb: An ultra-fast feature for scalable augmented reality on mobile devices. In: IEEE. *2012 IEEE international symposium on mixed and augmented reality (ISMAR)*. [S.l.], 2012. p. 49–57.
- 21 FAWCETT, T. An introduction to roc analysis. *Pattern recognition letters*, Elsevier, v. 27, n. 8, p. 861–874, 2006.
- 22 TRIGGS, B. et al. Bundle adjustment—a modern synthesis. In: SPRINGER. *International workshop on vision algorithms*. [S.l.], 1999. p. 298–372.
- 23 WU, C. et al. Multicore bundle adjustment. In: IEEE. *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. [S.l.], 2011. p. 3057–3064.
- 24 AGARWAL, S. et al. Bundle adjustment in the large. In: SPRINGER. *European conference on computer vision*. [S.l.], 2010. p. 29–42.
- 25 BERGER, M. et al. State of the art in surface reconstruction from point clouds. In: *EUROGRAPHICS star reports*. [S.l.: s.n.], 2014. v. 1, n. 1, p. 161–185.
- 26 MENCL, R.; MULLER, H. Interpolation and approximation of surfaces from three-dimensional scattered data points. In: IEEE. *Scientific Visualization Conference (dagstuhl'97)*. [S.l.], 1997. p. 223–223.
- 27 MENCL, R.; MULLER, H. Interpolation and approximation of surfaces from three-dimensional scattered data points. In: IEEE. *Scientific Visualization Conference, 1997*. [S.l.], 1997. p. 223–223.
- 28 MARTON, Z. C.; RUSU, R. B.; BEETZ, M. On fast surface reconstruction methods for large and noisy point clouds. In: IEEE. *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*. [S.l.], 2009. p. 3218–3223.
- 29 LI, R. et al. Polygonizing extremal surfaces with manifold guarantees. In: ACM. *Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*. [S.l.], 2010. p. 189–194.
- 30 JU, T. et al. Dual contouring of hermite data. In: ACM. *ACM transactions on graphics (TOG)*. [S.l.], 2002. v. 21, n. 3, p. 339–346.

- 31 AMENTA, N.; KIL, Y. J. Defining point-set surfaces. In: ACM. *ACM Transactions on Graphics (TOG)*. [S.l.], 2004. v. 23, n. 3, p. 264–270.
- 32 SÜSSMUTH, J.; GREINER, G. Ridge based curve and surface reconstruction. In: *ACM International Conference Proceeding Series*. [S.l.: s.n.], 2007. v. 257, p. 243–251.
- 33 MASSON, J. E. N.; PETRY, M. R. Comparison of mesh generation algorithms for railroad reconstruction. In: *2017 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*. [S.l.: s.n.], 2017. p. 266–271.
- 34 GOLDBERG, D. *Seamless Texture Mapping of 3D Point Clouds*. [S.l.], 2014.
- 35 BONDAREV, E. et al. On photo-realistic 3d reconstruction of large-scale and arbitrary-shaped environments. In: IEEE. *2013 IEEE 10th Consumer Communications and Networking Conference (CCNC)*. [S.l.], 2013. p. 621–624.
- 36 WAECHTER, M.; MOEHRLE, N.; GOESELE, M. Let there be color! large-scale texturing of 3d reconstructions. In: SPRINGER. *European Conference on Computer Vision*. [S.l.], 2014. p. 836–850.
- 37 WU, C. Towards linear-time incremental structure from motion. In: IEEE. *3DTV-Conference, 2013 International Conference on*. [S.l.], 2013. p. 127–134.
- 38 FURUKAWA, Y.; PONCE, J. Accurate, dense, and robust multiview stereopsis. *IEEE transactions on pattern analysis and machine intelligence*, IEEE, v. 32, n. 8, p. 1362–1376, 2010.
- 39 KAZHDAN, M.; HOPPE, H. Screened poisson surface reconstruction. *ACM Transactions on Graphics (ToG)*, ACM, v. 32, n. 3, p. 29, 2013.
- 40 KANG, Z.; MEDIONI, G. Fast dense 3d reconstruction using an adaptive multiscale discrete-continuous variational method. In: IEEE. *Applications of Computer Vision (WACV), 2014 IEEE Winter Conference on*. [S.l.], 2014. p. 53–60.
- 41 RUSU, R. B.; COUSINS, S. 3d is here: Point cloud library (pcl). In: IEEE. *Robotics and automation (ICRA), 2011 IEEE International Conference on*. [S.l.], 2011. p. 1–4.
- 42 JANCOSEK, M.; PAJDLA, T. Multi-view reconstruction preserving weakly-supported surfaces. In: IEEE. *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*. [S.l.], 2011. p. 3121–3128.
- 43 ALICEVISION. *Meshroom*. <https://alicevision.github.io>, Acessado em 14 de janeiro de 2019.
- 44 AGISOFT. *Metashape*. <https://www.agisoft.com>, Acessado em 14 de janeiro de 2019.
- 45 3DFLOW. *3Dflow*. <https://www.3dflow.net>, Acessado em 14 de janeiro de 2019.
- 46 MUJA, M.; LOWE, D. G. Fast matching of binary features. In: *Computer and Robot Vision (CRV)*. [S.l.: s.n.], 2012. p. 404–410.
- 47 STRECHA, C. et al. On benchmarking camera calibration and multi-view stereo for high resolution imagery. In: IEEE. *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. [S.l.], 2008. p. 1–8.