

Lucas Pontes de Albuquerque

DQ-Mesh: Fotografia 3D com imagem única em dispositivos *mobile*.



Universidade Federal de Pernambuco

Recife
2024

Lucas Pontes de Albuquerque

DQ-Mesh: Fotografia 3D com imagem única em dispositivos *mobile*.

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: *Inteligência Computational, Computação Gráfica, Visão Computacional*
Orientador: *Tsang Ing Ren*

Recife
2024

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Albuquerque, Lucas Pontes de.
DQ-Mesh: fotografia 3D com imagem única em dispositivos mobile / Lucas Pontes de Albuquerque. - Recife, 2024.
99 f.: il.

Dissertação (Mestrado) - Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-Graduação em Ciência da Computação, 2024.
Orientação: Tsang Ing Ren.
Inclui referências.

1. Visão computacional; 2. Aprendizagem profunda; 3. Computação gráfica. I. Ren, Tsang Ing. II. Título.

UFPE-Biblioteca Central

Dedico este trabalho a toda minha família e a minha companheira Natália.

AGRADECIMENTOS

Agradeço primeiramente a toda minha família pela suporte emocional e por nunca deixarem de acreditar em meu potencial. Especialmente a minha companheira, Natália, que me ajudou a me dedicar a conclusão deste trabalho.

Agradeço ao professor Tsang e a todo mundo do projeto Motorola que me ajudaram com as implementações e na oportunidade de desenvolver uma solução para um problema desafiador. Também agradeço a meus amigos Lucas Augusto, Miguel e Gabriel que me ajudaram com críticas ao trabalho descrito nessa dissertação e que também sempre acreditaram no meu potencial.

ABSTRACT

Despite the increasing advancement and development of deep learning techniques in building 3D models using photographs, it is still a very challenging problem. Even with constant improvements in depth estimation and color filling techniques, current works has focused on adapting new architectures to the construction of 3D structures for rendering. Therefore, these methods can have a high computational cost, making implementation on *mobile* devices difficult. Furthermore, methods with the NeRF (*Neural Radiance Fields*) approach, which are the current state of the art in 3D photography, need a large number of images and camera positions to obtain accurate modeling. In this work we propose DQ-Mesh (*Depth-based Quadtree Mesh*), which consists of a method for building a *mesh* structure using only a depth estimator and a color inpainting model. Unlike state-of-the-art methods, this method does not require training to build the 3D structure, using only the depth map as a reference. Thus, the focus of optimizations for devices with lower computational capacity falls on depth estimation and color inpainting algorithms. Experiments on real datasets show that the performance of our proposed method is comparable in performance to current state-of-the-art models.

Keywords: Computacional Photography. Deep Learning. Computer Graphics.

RESUMO

Apesar do crescente avanço e desenvolvimento de técnicas de aprendizagem profunda na construção de modelos 3D utilizando fotografias, ainda é um problema bastante desafiador. Mesmo com as constantes melhorias em técnicas de estimação de profundidade e de preenchimento de cor, os trabalhos atuais tem focados em adaptar novas arquiteturas à construção das estruturas 3D para renderização. Logo, esses métodos podem ter um elevado custo computacional dificultando a implementação em dispositivos *mobile*. Além disso, os métodos com abordagem NeRF (*Neural Radiance Fields*), que são o atual estado da arte em fotografia 3D, precisam de um grande número de imagens e posições de câmera para obter uma modelagem acurada. Neste trabalho propomos o DQ-Mesh (*Depth-based Quadtree Mesh*), que consiste em um método de construção de uma estrutura *mesh* utilizando apenas um estimador de profundidade e um modelo de preenchimento de cor. Diferente dos métodos no estado da arte, esse método não precisa de treinamento para a construção da estrutura 3D, utilizando apenas o mapa de profundidade como referencia. Assim, o foco das otimizações para embarcar em dispositivos com menor capacidade computacional recai sobre os algoritmos de estimação de profundidade e preenchimento de cor. Os experimentos em conjuntos de dados reais mostram que o desempenho do nosso método proposto tem desempenho comparável aos atuais modelos no estado da arte.

Palavras-chave: Fotografia Computacional. Aprendizagem Profunda. Computação Gráfica.

LISTA DE FIGURAS

Figura 1 – Exemplo de *pipeline* tradicional para NVS utilizando uma imagem de entrada. Fonte: Horry *et al.* (1997) 17

Figura 2 – *Pipeline* tradicional para reconstrução 3D. Fonte:Pollefeys & Gool (2002) 22

Figura 3 – Exemplo de uma representação de nuvens de pontos. Fonte: Tutorial da ferramenta Open3D. 24

Figura 4 – Exemplo de *voxels*. Fonte: Samavati & Soryani (2023). 25

Figura 5 – Exemplo de malhas 3D. Fonte: Tutorial da ferramenta Open3D.. 26

Figura 6 – Na imagem à esquerda vemos um exemplo de como a estrutura de um *quadtree* se apresenta e no lado direito as respectivas regiões de uma imagem delimitadas pelas folhas. Fonte: Berg *et al.* (2008). 27

Figura 7 – Em (a) temos a imagem I_{Ex} de entrada. Em (b) temos o primeiro passo do algoritmo, quando são delimitados os limites das regiões representadas pelo nó, em azul temos as bordas delimitadas pelos pontos extremos (pontos vermelhos). Após verificar a existência de pontos onde $I(i,j) = 1$ (pontos brancos) na região delimitada pelo nó τ na profundidade $d = 0$ em (c) vemos o inicio da recursão no filho $\tau.\tau_{NW}$. Após verificar a existência de pontos brancos na região delimitada pelo nó $\tau.\tau_{NW}$, as recursões nos nós $\tau.\tau_{NW}.\tau_{NW}$, $\tau.\tau_{NW}.\tau_{NE}$, $\tau.\tau_{NW}.\tau_{SW}$ retornam com seus nós com valor $\emptyset$, sendo classificados como folhas. Logo em (d) vemos a chamada da recursão para o nó $\tau.\tau_{NW}.\tau_{SE}$. Com isso chegamos na folha $\tau.\tau_{NW}.\tau_{SE}.\tau_{SE}$ na profundidade limite em (e). Repetindo o processo para os outros nós ficamos com o *quadtree* τ delimitado pelas regiões mostradas em (f). 29

Figura 8 – Exemplo da construção utilizando o *quadtree* resultante da Figura 7. Em (a) vemos a configuração inicial apresentando em azul os pontos utilizados para formar o *quadtree*, com as regiões armazenadas nele representadas pelas linhas pretas. Em (b) são apresentados em vermelho os pontos que representam as extremidades das regiões armazenadas nos nós de τ. No final do algoritmo é retornada uma malha equivalente a apresentada em (c), representada pelas arestas e os pontos nelas contidos armazenadas em *edges*. Nas figuras (d) e (e) são *zooms* aplicados na figura (c) em regiões que representam os casos em que $\|P_{neigh}\| = 4$ (figura (d)) e $\|P_{neigh}\| > 4$ (figura (e)). 31

Figura 9 – Exemplos de mapeamento de textura. Fonte: Retirado do site *link*. . . . 32

Figura 10 – Fonte: Pérez-Enciso & Zingaretti (2019) 34

Figura 11 – Demonstração gráfica de uma rede MLP. 35
Figura 12 – Fonte: Lecun *et al.* (1998) 36
Figura 13 – Fonte: Goodfellow *et al.* (2016) 37

Figura 14 – Fonte: Mildenhall *et al.* (2021) 41
Figura 15 – Fonte: Zhou *et al.* (2018b) 43
Figura 16 – Fonte: Shade *et al.* (1998) 45
Figura 17 – Fonte: Niklaus *et al.* (2019) 46
Figura 18 – Fonte: Shih *et al.* (2020) 46
Figura 19 – Fonte: Kopf *et al.* (2020) 47
Figura 20 – Fonte: Jampani *et al.* (2021) 48

Figura 21 – Diagrama do fluxo do método para a utilização do DQ-Mesh. Os quadrados verdes representam os modelos pré-treinados utilizados. Em azul estão blocos onde estão presentes os algoritmos propostos. Em roxo é o renderizador para gerar imagens a partir de uma posição de câmera. Imagens de exemplo da saída de cada componente do *pipeline* são apontadas pelas setas laranja. 50
Figura 22 – Fonte: Ranftl *et al.* (2021) 51
Figura 23 – Fonte: Xian *et al.* (2018) 52
Figura 24 – Fonte: Suvorov *et al.* (2022) 53
Figura 25 – Diagrama do fluxo do método para a utilização do DQ-Mesh. Em azul estão blocos onde estão presentes os algoritmos desenvolvidos que possuem contribuições. Em verde está o pipeline para extração das imagens de borda utilizadas. Em vermelho está a camada onde serão concatenados os vértices, as faces e as coordenadas de textura das regiões de *foreground*, *background* e de *merge*. Imagens de exemplo da saída de cada componente do algoritmo são apontadas pelas setas laranja. 55
Figura 26 – Imagens de entrada e saída de um exemplo da execução do Algoritmo 5. Na Figura 26a vemos uma entrada I_D com dimensões $H = 16$ e $W = 16$. Em 26b vemos a imagem com os contornos I_{Canny} de entrada. Em 26c vemos uma sobreposição das entradas I_D e I_{Canny} que ao final do algoritmo resulta na imagem I'_D presente em 26d. 57
Figura 27 – Imagens utilizadas e geradas pelo algoritmo 6. Em 27a e 27b vemos as imagens de entrada I_{RGB} e I_D respectivamente. Em 27c é mostrada a máscara de segmentação I_M resultante da inferencia do modelo Mobile-SAM pré-treinado. Na Figura 27d vemos a profundidade interpolada em cada bloco I_{DM}, que após a multiplicação por I_M temos a imagem de saída I'_D mostrada na Figura 27e. 58

Figura 28 – Região das imagem de profundidade mostradas nas Figura 26a e 26d. Em (a) vemos a região na imagem de entrada I_D e seu melhoramento resultante I'_D em (b). 59
Figura 29 – Exemplo de saída do Algoritmo 8, sendo os *pixels* em azul representando as bordas do *foreground* e em verde a borda de *background* de uma região da imagem. 61
Figura 30 – Em (a) temos uma imagem representando os rótulos de uma região da imagem de profundidade I_D. Cada *pixel* colorido representa um vértice $v_b \in V_b$ do *mesh* final, onde os pontos verdes são vértices que o valor de sua componente na dimensão z será igual a profundidade armazenada no *pixel* correspondente. Os azuis representam os vértices de *foreground*, logo recebem na dimensão z o valor médio da profundidade nas bordas de *background* internas dos blocos que o vértice está contido. Em (b) um exemplo de malha 3D de saída da função `back_mesh_generator`. 63
Figura 31 – Em (a) temos uma imagem de exemplo da entrada I_{FE} e em (b) a malha resultante dos V_f, UV_f e F_f da função `fore_mesh_generator`. . . 66
Figura 32 – Imagem de exemplo da malha completa modelada por V, UV e F para a entrada I'_D da Figura 28b. Em azul temos a malha de *foreground* e os triângulos formados em `merge_mesh_generator`. Em verde esta a malha de *background*. 67
Figura 33 – Em cada ponto (x, y) (em verde) da imagem, são calculadas as diferenças de profundidade sobre as linhas horizontais e verticais de comprimento m (em vermelho) para computar os mapas de SD. Fonte: Jampani *et al.* (2021) 69
Figura 34 – Exemplo de funcionamento da função *SD*. Em (a) temos a imagem de entrada I_D. Em (b) vemos a mascara resultante da função *SD* de acordo com os parâmetros ρ, γ e m. 70
Figura 35 – Exemplo mostrando o efeito da combinação entre a máscara gerada por DQ-Mesh e SD. Em (a) vemos os contornos e como em (b) existem regiões da mascara SD que não estão presentes contornos em I_{FE}. Utilizando a imagem em (c) multiplicando com (b) temo o resultado em (d), sem os *pixels* indesejados para processar o *inpaint*. 70
Figura 36 – Exemplo da utilização do modelo treinado da arquitetura LaMaSep. Em (a) e (b) vemos as entradas da rede e em (c) vemos a imagem da inferência. 71

Figura 37 – Em (a) vemos o efeito de esticamento se não for utilizado SFPV como canal alfa. Caso SFPV for utilizado, como os vértices que estão presentes nesse esticamento interpolam a cor com um canal alfa, vemos em (b) que o esticamento é removido. 71
Figura 38 – Em (a) vemos a profundidade para computar a imagem (b) com SFPV. Em (c) vemos a textura que será utilizada pela malha do *foreground* com seus respectivos valores de alfa. 72
Figura 39 – Experimentos com o primeiro subconjunto de imagens do HDR+ utilizando MiDaSv3.0. Na primeira coluna são apresentados os nomes dados às imagens analisadas. A entrada dos métodos para gerar a representação é mostrada na segunda coluna. Na terceira coluna vemos o resultado da renderização através da abordagem SLIDE para construir a malha. Os resultados das malhas utilizando DQ-Mesh e DQ-Mesh mais a melhoria da profundidade são mostrados na quarta e quinta coluna respectivamente. 75
Figura 40 – Experimentos com o segundo subconjunto de imagens do HDR+ utilizando Mi- DaSv3.0. Descrição das colunas semelhante a Figura 39. . 76
Figura 41 – Algumas zonas de interesse das renderizações mostradas nas Figuras 39 e 40. A descrição das colunas é semelhante a descrita em 39, porém ao invés dos resultados nas 3 ultimas colunas estão as regiões de interesse. 78
Figura 42 – Analise do impacto da melhoria de profundidade utilizando segmentação. Na primeira e segunda coluna estão os nomes e as imagens RGB de entrada para os modelos MiDaSv3.0 e MobileSAM, respectivamente. Na terceira coluna vemos as mascaras de segmentação inferidas por MobileSAM. Os resultados da melhoria utilizando as imagens de inferencia do MiDaSv3.0 (quarta coluna) são mostrado na quinta coluna. 80
Figura 43 – Experimentos com o primeiro subconjunto de imagens do HDR+ utilizando MiDaS *mobile*. Descrição das colunas semelhante a Figura 39. 81
Figura 44 – Experimentos com o segundo subconjunto de imagens do HDR+ utilizando MiDaS *mobile*. Descrição das colunas semelhante a Figura 39. 82
Figura 45 – Algumas regiões de interesse de interesse das renderizações mostradas nas Figuras 43 e 44. A descrição das colunas é semelhante a 41. . . . 83
Figura 46 – Mesma descrição que a Figura 42, porém ao invés de mostrar a inferência da arquitetura MiDaSv3.0 são mostrados os resultados com MiDaS *mobile*. 84

Figura 47 – Comparação do tempo de inferência de um subconjunto de 150 imagens do conjunto HDR+Hasinoff *et al.* (2016) rodando por 30 vezes no dispositivo Moto Edge 20+. A primeira barra é a arquitetura LaMa convencional (LaMa Conv) e a segunda é a nossa implementação com operadores FFC em Halide e convoluções separáveis (LaMaSep). Como LaMa Conv é implementado com operadores que possuem suporte para GPUs, diferente da arquitetura LaMa Sep implementada, ambas as arquiteturas foram executadas em CPU. A escala dos valores mostrados é milissegundos 86

LISTA DE TABELAS

Tabela 1 – Tabela com a comparação da quantidade de vértices em cada uma das abordagens. Em SLIDE Jampani *et al.* (2021) os *mesh* são criados utilizando todos os *pixels* do mapa de profundidade como vértices conectados com os seus vizinhos. No caso da construção da malha com DQ-Mesh, depende-se dos contornos computados em I_{FE}, sendo assim, os valores obtidos foram calculados através de uma média utilizando imagens do banco de dados HDR+ Hasinoff *et al.* (2016) e o modelo MiDaS *mobile* para o mapa de profundidade. Na última coluna é mostrado o ganho $\frac{N_v}{\mathscr{E}_v}$, sendo N_v o número de vertices na coluna do SLIDE e $\mathscr{E}_v$ na coluna do DQ-Mesh. 85

Tabela 2 – Resultado das medições do tempo de execução em 2 dispositivos da Motorola. A inferência do MiDaS foi implementada em Tensorflow Lite executado em GPU. O melhoramento de profundidade e o DQ-Mesh foram implementado utilizando Halide e executados na CPU. A inferência do LaMaSep foi implementada em Tensorflow Lite com os operadores FFC customizados em Halide. Esse operadores são executados em CPU, enquanto os outros operadores do modelo LaMaSep são executados em GPU. 86

LISTA DE ACRÔNIMOS

ANN	*Artifitial Neural Networks*
CCL	*Component Connected Labeling*
CNN	*Convolutional Neural Networks*
DQ-Mesh	*Depth-based Quadtree Mesh*
FFC	*Fast Fourier Convolution*
FFT	*Fast Fourier Transform*
GAN	*Generative Adversarial Networks*
LDI	*Layered Depth Images*
MHSA	*Multi-Headed Self-Attention*
MLP	*Multilayer Perceptron*
MPI	*Multiplane Images*
MSE	*Mean Squared Error*
MVS	*Multi-View Stereo*
NeRF	*Neural Radiance Fields*
NVS	*Novel View Synthesis*
SD	*Soft Disocclusions*
SDF	*Signed Distance Field*
SFM	*Struct From Motion*
SFPV	*Soft Foreground Pixel Visibility*

LISTA DE ALGORITMOS

Algoritmo 1 – `Quadtree` 28
Algoritmo 2 – `QuadtreeToMesh`: O algoritmo recebe como entrada um *quadtree* τ e um conjunto de arestas inicialmente com valores $edges = \{\}$. 30

Algoritmo 3 – FFC 54
Algoritmo 4 – `DQ-Mesh`; Visão geral do *pipeline* para gerar a malha 3D que será utilizada para NVS. 56
Algoritmo 5 – `depth_simple_enhancement` 57
Algoritmo 6 – `depth_segment_enhancement` 59
Algoritmo 7 – `depth_mask_interpolation` 60
Algoritmo 8 – `process_edges` 61
Algoritmo 9 – `back_mesh_generator`; Algoritmo que retorna V_b, UV_b e F_b utilizados para compor a malha de *background*. 62
Algoritmo 10 – `fore_mesh_generator`; Retorna V_f, UV_f e F_f utilizados para compor a malha de *foreground*. Assim como em Algoritmo 9, cada bloco é indexado com (x_b, y_b). 64
Algoritmo 11 – `get_mesh_from_edges` 65
Algoritmo 12 – `merge_mesh_generator`; Retorna V_m, UV_m e F_m utilizados para compor a malha que conecta as malhas de *foreground* e *background*. 67
Algoritmo 13 – `get_merge_faces` 68

SUMÁRIO

1 INTRODUÇÃO 16
1.1 MOTIVAÇÃO 18
1.2 OBJETIVOS 19
1.3 ESTRUTURA DA DISSERTAÇÃO 20

2 CONCEITOS BÁSICOS 21
2.1 FOTOGRAFIA 3D 21
2.1.1 Métodos tradicionais de reconstrução 3D 23
2.1.2 Estruturas de dados para modelagem espacial 24
2.1.2.1 Representações baseadas em nuvens pontos 24
2.1.2.2 Representações volumétricas 25
2.1.2.3 Representações de superfícies 25
2.1.3 Construindo *Meshes* com *Quadtrees* 26
2.1.3.1 Quadtrees para um conjunto de pontos 26
2.1.3.2 De Quadtrees para Meshes triangulares 28
2.1.4 Mapeamento de Textura 32
2.2 APRENDIZAGEM PROFUNDA 32
2.2.1 Redes Neurais e *Multilayer Perceptron* 33
2.2.2 Redes Neurais Convolucionais 36

3 TRABALHOS RELACIONADOS 40
3.1 RENDERIZAÇÃO NEURAL 40
3.2 REPRESENTAÇÕES BASEADAS EM MAPA DE PROFUNDIDADE . . 43
3.2.1 *Multiplane Images* 43
3.2.2 *Layered Depth Images* e nuvens de pontos 45

4 MÉTODO PROPOSTO 49
4.1 ESTIMAÇÃO DE PROFUNDIDADE 51
4.1.1 MiDaS v3.0 51
4.1.2 MiDaS *mobile* 52
4.2 ARQUITETURA PARA *INPAINTING* 53
4.2.1 *Fast Fourier Convolution* 53
4.2.2 LaMa com convoluções separáveis 54
4.3 GERAÇÃO DE MALHAS 3D COM DQ-MESH 54
4.3.1 Pré-processamento simples do mapa de profundidade 56

4.3.2 **Pré-processamento do mapa de profundidade utilizando máscaras de segmentação** . . . 57
4.3.3 **Processamento dos contornos** . . . 60
4.3.4 **Geração do *Mesh* das regiões de *background*** . . . 61
4.3.5 **Geração do *Mesh* das regiões de *foreground*** . . . 64
4.3.6 **Combinando os *Mesh* do *foreground* e do *background*** . . . 66
4.4 RENDERIZAÇÃO DA MALHA COM MAPEAMENTO DE TEXTURAS 67
4.4.1 **Geração das máscaras de *inpainting* e da textura do *background*** . . . 69
4.4.2 **Geração da textura do *foreground* com SFPV** . . . 71
4.4.3 **Renderização** . . . 72

5 **EXPERIMENTOS** . . . 73
5.1 EXPERIMENTOS COM MIDASV3.1 . . . 74
5.2 EXPERIMENTOS COM MIDAS *MOBILE* . . . 78
5.3 ANÁLISE DE DESEMPENHO . . . 85

6 **CONCLUSÃO** . . . 88
6.1 CONTRIBUIÇÕES . . . 89
6.2 TRABALHOS FUTUROS . . . 89

REFERÊNCIAS . . . 91

1

INTRODUÇÃO

O desenvolvimento de sistemas baseados no funcionamento da visão humana é um dos interesses da área da Visão Computacional. Um dos campos relevantes dessa área é a tarefa de recuperar um cenário do mundo tridimensional (3D) utilizando imagens ou vídeos, conhecido como reconstrução 3D. Como muitos problemas em visão computacional, a reconstrução 3D é considerada um problema inverso Szeliski (2011) onde busca-se recuperar informações sobre um objeto ou cena a partir de dados com informações ambíguas.

Um dos sub-problemas da reconstrução 3D é a Fotografia 3D Pollefeys & Gool (2002), que envolve construir representações tridimensionais utilizando como dados fotográficos de um ou mais pontos de referência de um cenário real, trazendo uma experiência imersiva de realidade virtual ou aumentada. As aplicações da Fotografia 3D são várias, como modelagem de arquitetura urbana, inspeção industrial e arqueologia. A síntese realista de imagens a partir de posições de câmera diferentes das utilizadas para capturar as fotos utilizadas na modelagem, é chamado *Novel View Synthesis* (NVS) Rematas *et al.* (2017). Antes dessas imagens serem geradas, é necessário que modelos gráficos estejam disponíveis.

Uma maneira de obter modelos gráficos tridimensionais pode ser inspirado em como o próprio ser humano tem a percepção do espaço 3D. Um dos motivos de percebermos a profundidade se dá ao fato de possuirmos dois olhos. Como cada olho representa pontos de visão diferentes, essa diferença é relacionada à distância dos objetos Pollefeys & Gool (2002). Para que máquinas tenham essa capacidade, trabalhos mais tradicionais (com abordagens heurísticas) procuram extrair geometria de múltiplas imagens para gerar um modelo 3D. Anterior às técnicas de Visão Computacional, na metade do seculo XIX a fotogrametria já permitia o uso de fotografias para criação de mapas e medição de construções urbanas. Um exemplo clássico de método heurístico voltado para fotografia 3D é o proposto em Debevec *et al.* (1996), que propõe a renderização de NVS para arquitetura urbana combinando fotogrametria, computação gráfica e visão computacional.

Vários desses métodos heurísticos faziam o uso de múltiplas imagens de um mesmo cenário para computar uma representação da cena em 3D. Além de Debevec *et al.* (1996), podemos citar Chen (1995) onde as fotografias com sobreposição são capturadas em um ponto fixo mudando apenas o ângulo de visão (i.e. panoramas). Porém, essas técnicas estão limitadas

ao uso de um conjunto de imagens, levando a uma perda de praticidade a depender do uso do modelo 3D. Pensando em facilitar o desenvolvimento de animações com ferramentas mais práticas, Horry *et al.* (1997) introduz um método de que realiza NVS utilizando apenas uma imagem e parâmetros ajustados pelo usuário, como ponto de fuga e máscaras para os objetos presentes na imagem (Figura 1).

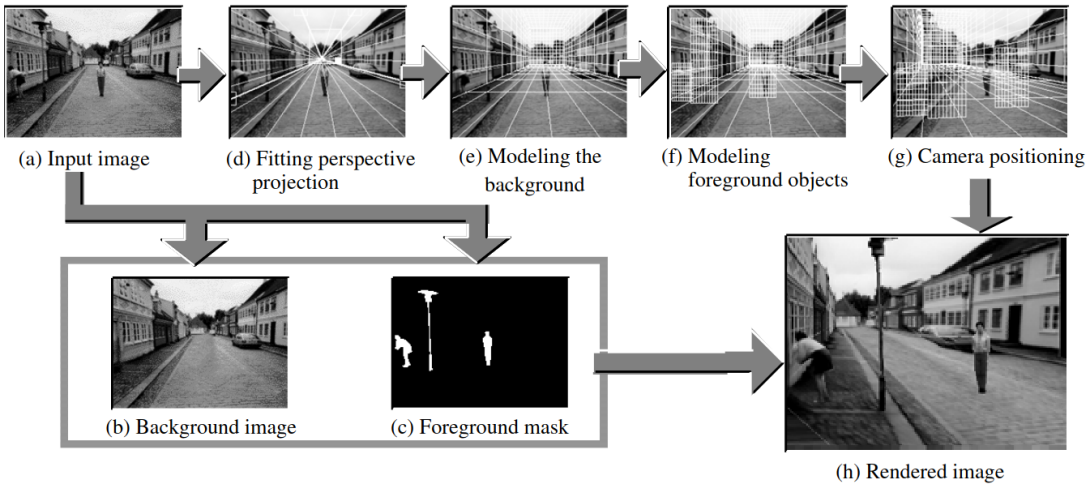


Figura 1: Exemplo de *pipeline* tradicional para NVS utilizando uma imagem de entrada. Fonte: Horry *et al.* (1997)

O problema das abordagens heurísticas consiste na necessidade do usuário definir características relacionadas à geometria das imagens da entrada (e.g. pontos de fuga, traçando as linhas das imagens). Isso leva à dependência de um esforço humano restrito a especialistas que conhecem as ferramentas de edição. Considerando que os métodos utilizando redes neurais LeCun *et al.* (2015) (mais precisamente *Deep Learning*) são o estado da arte em vários problemas de visão computacional (e.g. detecção de objetos He *et al.* (2017), segmentação Ronneberger *et al.* (2015); Kirillov *et al.* (2023), mapas de profundidade Ranftl *et al.* (2021)), pesquisadores em computação gráfica têm desenvolvido técnicas que combinam essas redes com representações 3D tradicionais. Das abordagens existentes na literatura, a de maior impacto recentemente foi a *Neural Radiance Fields* (NeRF) Mildenhall *et al.* (2021). Os métodos baseados em NeRF consistem em representar uma cena como um *Multilayer Perceptron* (MLP), utilizando um *pipeline* de renderização diferenciável para gerar as imagens dada uma nova posição de câmera. Entretanto, apesar desses métodos serem o estado da arte, precisam de uma certa quantidade de imagens de um mesmo cenário estático e as respectivas posições da câmera, o que torna difícil a geração dessas estruturas 3D em cenários mais realistas, como fotos tiradas de um aparelho celular. Além disso, esses métodos precisam de horas ou até dias para serem treinados. Até mesmo trabalhos mais recentes Fridovich-Keil and Yu *et al.* (2022); Yu *et al.* (2021a,b), que procuram otimizar o tempo de treinamento e inferência, ainda precisam de uma grande capacidade de computação e de diversas fotografias da cena que se pretende modelar.

Além de abordagens que utilizam renderização diferenciável (e.g. NeRF), destacam-

se as que, a partir de uma estrutura de dados 3D específica (e.g. *Mesh*, *Multiplane Images* (MPI), *Layered Depth Images* (LDI), etc), utilizam arquiteturas de *Deep Learning* para inferir profundidade e a cor das regiões ocluídas por objetos do cenário Shih *et al.* (2020); Niklaus *et al.* (2019); Zhou *et al.* (2018b). Essas soluções, chamadas de multicamadas de profundidade, utilizam métodos clássicos de renderização, que necessitam de uma menor capacidade de processamento que os métodos de renderização diferenciável. Além disso, esses métodos não necessitam treinar o modelo para um cenário específico, tornando-os mais fáceis de aplicar em cenários mais realistas por serem mais generalistas.

Vários desses trabalhos que utilizam multicamadas, buscam construir representações utilizando uma imagem de entrada. Por exemplo no caso das abordagens baseadas em MPI como Luvizon *et al.* (2021); Tucker & Snavely (2020), são implementadas arquiteturas que a inferência são MPIs, logo a rede preenche o conteúdo de cor das imagens de cada plano de profundidade. Isso pode levar a uma inferência demorada em dispositivos móveis se utilizadas imagens de alta resolução, pois essas redes foram desenvolvidas para rodar em GPUs robustas em aparelhos *desktop*. Além disso, esses tipos de representação pode apresentar *aliasing* a depender da resolução de renderização, levando a um esforço específico para lidar com esse tipo de artefato Srinivasan *et al.* (2019).

Os métodos baseados em LDI utilizando apenas uma imagem de referência, como em Shih *et al.* (2020); Jampani *et al.* (2021); Kopf *et al.* (2020), fazem um processamento heurístico dos contornos da imagem de profundidade para selecionar as regiões que possuem oclusão. Nessas regiões são computadas máscaras que serão utilizadas para inferir cor e profundidade através de arquiteturas CNN. Após isso, a representação com camadas de profundidade é utilizada para renderizar novas imagens dadas diferentes posições de câmera. Semelhante aos baseados em MPI, esses métodos utilizando imagens de alta resolução sem perda de informação (i.e. sem redimensionamento diminuindo a resolução) precisam de mapas de profundidade com a mesma resolução da imagem, levando a uma inferência lenta em dispositivos *mobile*.

1.1 MOTIVAÇÃO

As aplicações possíveis das soluções para NVS em aparelhos móveis têm trazido interesse em tornar essa tecnologia mais acessível. Assim, as soluções que utilizam apenas uma imagem como entrada são mais fáceis e práticas de embarcar nesse contexto. Porém, no caso dos celulares atuais as imagens capturadas pela câmera possuem uma alta resolução, o que pode ser custoso para as abordagens baseadas em MPI Khan *et al.* (2023a) e LDI Kopf *et al.* (2020); Jampani *et al.* (2021) mais eficientes. Boa parte desse custo, além da inferência das arquiteturas, reside no armazenamento. No caso dos MPIs, a depender da solução, podem ser geradas várias camadas com informações redundantes. Para LDIs como Jampani *et al.* (2021); Shih *et al.* (2020), representar cenários através de imagem de alta resolução necessitaria utilizar milhões de vértices.

Uma contribuição relevante de Kopf *et al.* (2020) é a utilização de mapeamento de textura para renderizar imagens de alta resolução independente das dimensões do mapa de profundidade. Apesar de Kopf *et al.* (2020) apresentar uma solução apropriada para *mobiles*, é necessário o desenvolvimento de adaptações na arquitetura de *inpainting* que possuem pouca replicabilidade na literatura. Como é mostrado em Jampani *et al.* (2021), podemos utilizar arquiteturas de *inpainting* e estimação de profundidade conhecidos na literatura para construir representações LDI. Parecido com abordagens clássicas como Horry *et al.* (1997), isso é feito dividindo a imagem em duas camadas uma de *foreground* (onde estão os objetos a frente nas oclusões) e outra de *background* (onde estão os objetos ocluídos). Além disso, as máscaras de *inpainting* são computadas utilizando filtros básicos de processamento de imagens.

Arquiteturas como LaMa Suvorov *et al.* (2022) (para *inpainting*) e MiDaS Ranftl *et al.* (2021, 2022) podem ser facilmente adaptadas para dispositivos *mobile*. O trabalho Suvorov *et al.* (2022) onde é apresentada LaMa, mostra suas vantagens em utilizar camadas convolucionais com transformadas de Fourier ao precisar de um número menor de parâmetros do que os arquiteturas com camadas convolucionais convencionais. No repositório onde é implementado o MiDaS encontra-se modelos voltados para *mobile*, utilizando as camadas apresentadas Tan & Le (2019) para reduzir o custo da inferência.

1.2 OBJETIVOS

Inspirando-se em Jampani *et al.* (2021) e Kopf *et al.* (2020), o objetivo deste trabalho é o desenvolvimento de um método de construção de malhas 3D (*meshes*) utilizando uma imagem de alta resolução em dispositivos *mobile*. Resumidamente, o foco é a redução do uso de memória sem impactar significativamente o tempo para gerar o *mesh* utilizado para NVS. Será utilizado o mapeamento de texturas para evitar *aliasing* e conseguir utilizar a própria imagem de entrada como textura. Com isso, será comparada a qualidade das imagens renderizadas e o tempo de inferência (i.e. construção de toda a estrutura 3D para a renderização) com os resultados dos métodos presentes na literatura.

Com isso, serão os objetivos específicos desta dissertação:

- Encontrar e testar os modelos disponíveis das arquiteturas MiDaS.
- Implementar e treinar uma melhoria da arquitetura LaMa utilizando convoluções separáveis Chollet (2017) para acelerar a inferência e reduzir o uso de memória.
- Implementar um método para refinamento do mapa de profundidade para adequá-lo ao método de geração do *mesh*.
- Implementar um método para gerar uma estrutura de *mesh* 3D eficiente para dispositivos *mobile*.

- Implementar o *pipeline* de renderização utilizando mapeamento de textura para gerar as imagens dada uma posição de câmera.
- Comparar os resultados com as soluções disponíveis na literatura encontradas.

1.3 ESTRUTURA DA DISSERTAÇÃO

Os capítulos seguintes são organizados da seguinte maneira: no Capítulo 2 são apresentados os fundamentos teóricos básicos para soluções de fotografia 3D e no desenvolvimento de arquiteturas de aprendizagem profunda; no Capítulo 3 são apresentados trabalhos que apresentam soluções de fotografia 3D utilizando redes neurais; no Capítulo 4 são apresentados os algoritmos propostos que formam o *pipeline* para realizar NVS; No Capítulo 5 são apresentados os experimentos realizados e as devidas comparações com os métodos existentes na literatura; No Capítulo 6 é apresentada a conclusão e os trabalhos futuros.

2

CONCEITOS BÁSICOS

Neste capítulo serão apresentados os fundamentos teóricos que formulam o problema de fotografias 3D utilizando aprendizagem profunda. Na Seção 2.1 introduzimos algumas técnicas e os fundamentos básicos que são utilizados para reconstrução 3D. De maneira concreta, através de uma abordagem histórica é apresentada a estrutura geral de como funcionam as estrategias para modelar objetos ou cenas utilizando um conjunto de imagens. Nesse sentido, na Seção 2.1.1 traremos os métodos anteriores ao sucesso das redes CNN e como eles influenciaram no desenvolvimento do atual estado da arte para fotografia 3D. Parte dessa influência pode ser vista através das formas de representação de modelos 3D (descritos em Seção 2.1.2) e como ele são construídos. Uma das formas de computar representações é a construção de malhas utilizando *quadtrees*, que inspirou o DQ-Mesh.

Várias das abordagens tradicionais de reconstrução 3D dependem do uso de características que são geradas manualmente e são projetadas para combinar um conjunto de imagens capturadas de diferentes posições de observação de um mesmo cenário Szeliski (2011); Pollefeys & Gool (2002). Pelo fato desses métodos dependerem de um trabalho manual, torna-se difícil a busca por descritores que geram o modelo ótimo. Devido ao sucesso das redes neurais nos últimos anos tem surgido diferentes abordagens de modelagem 3D baseados em aprendizagem profunda Niklaus *et al.* (2019); Mildenhall *et al.* (2021). Assim, na Seção 2.2 são apresentados os fundamentos teóricos utilizados na construção dos modelos de redes neurais. Também serão apresentados alguns trabalhos com relevância histórica na área de aprendizagem profunda para problemas de processamento de imagens, que influenciaram abordagens atuais voltadas para fotografia 3D.

2.1 FOTOGRAFIA 3D

A fotografia 3D consiste em aplicações que constroem modelos 3D utilizando um conjunto de imagens e então recuperando a aparência da superfície. Esse termo foi introduzido em Pollefeys & Gool (2002), onde são apresentados os passos de *pipeline* de processamento para construir esses modelos. Podemos ver na Figura 2 uma visão geral dos passos de um algoritmo tradicional para aplicações de fotografia 3D descrito em Pollefeys & Gool (2002).

Primeiramente, são capturadas imagens de um cenário ou objeto a ser modelado e computadas as características correspondentes entre as imagens. Essas características podem ser definidas como a descrição de propriedades que otimizam a sua desambiguação ao longo das imagens (e.g. pontos do estão repetidos em diferentes imagens). Utilizando métodos, como o descrito em Harris & Stephens (1988), extraem-se pontos nas diferentes imagens e combinam-se os que tiverem mais similaridade entre as coordenadas. Após a extração desses pontos, o próximo passo é a construção da estrutura de dados espaciais (e.g. uma lista de pontos tridimensionais) e a movimentação da câmera. Então, é computado um mapa de profundidade denso para construir uma estrutura 3D e modelar consistentemente a geometria da cena. Depois, é recuperada a informação de cor (extraindo textura) para ser realizada uma renderização realista.

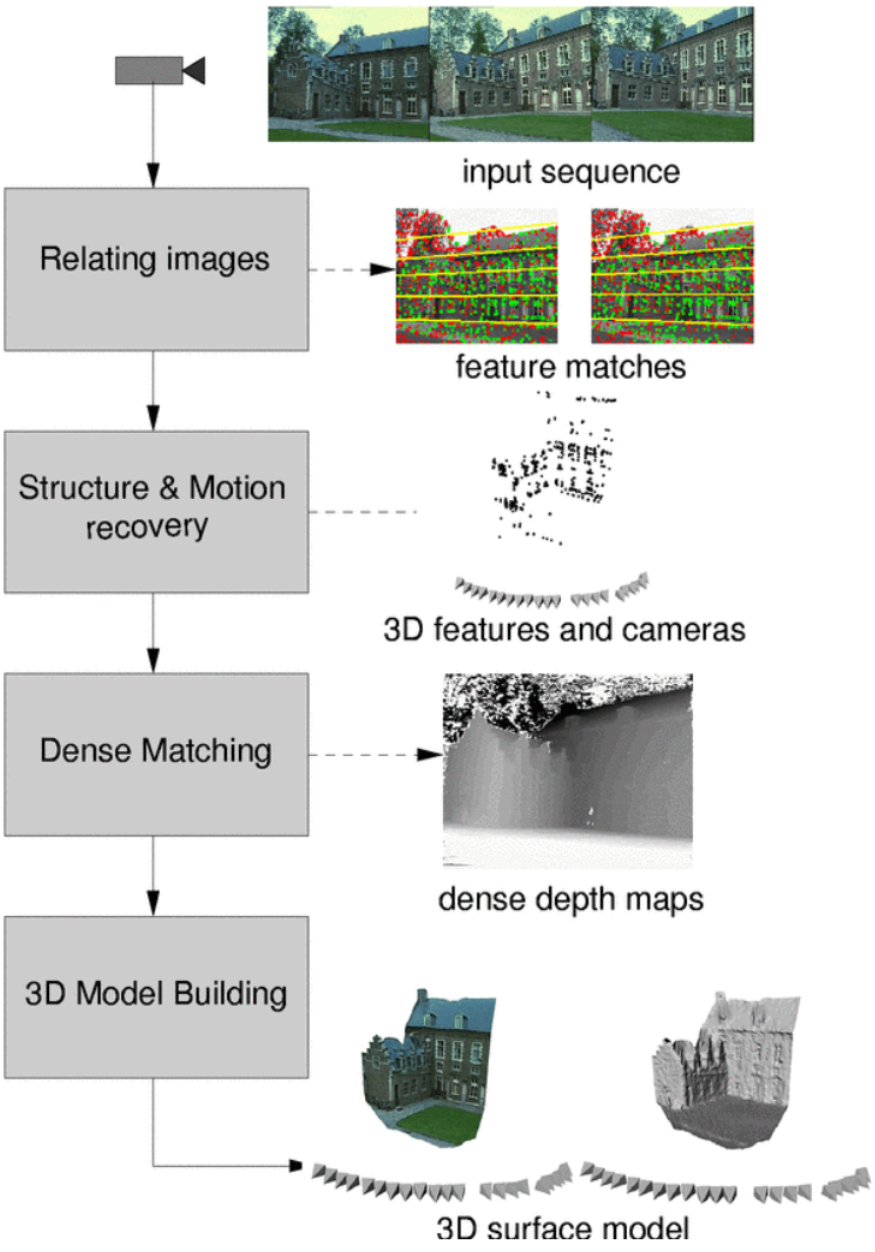


Figura 2: *Pipeline* tradicional para reconstrução 3D. Fonte:Pollefeys & Gool (2002)

Em trabalhos atuais voltados para reconstrução e fotografia 3D, como Mildenhall *et al.* (2021); Shih *et al.* (2020); Pu *et al.* (2023), grande parte desse *pipeline* é substituído por arquiteturas de CNN. Porém, é necessário apresentar as soluções tradicionais e seus problemas para as etapas mostradas na Figura 2 que serviram como base para o desenvolvimento do estado da arte. Assim, na próxima seção apresentaremos abordagens que são importantes para construção, por exemplo, de banco de dados como RealState10K Zhou *et al.* (2018b) e o treinamento de arquiteturas como Tucker & Snavely (2020).

2.1.1 Métodos tradicionais de reconstrução 3D

Os métodos voltados para reconstrução 3D constituem o arcabouço de algoritmos utilizados para construir fotografias 3D. Trabalhos mais clássicos, como Hannah (1974), utilizam triangularização de pontos entre duas câmeras sincronizadas (i.e. imagem estéreo) para computar os dados da estrutura 3D. Utilizando imagens dessincronizadas capturadas através de um sensor (i.e. imagem monocular) é necessário o uso de soluções mais sofisticadas para modelagem 3D. Na literatura podemos encontrar diferentes informações tridimensionais (e.g. textura, foco, movimentação, sombreamento) que podem ser extraídas de sequências de imagens monoculares para construir modelos Szeliski (2011).

Tradicionalmente, imagens estéreo são definidas como a representação de dois diferentes pontos de visão capturados por duas câmeras passivas sincronizadas Szeliski (2011). Como as câmeras são sincronizadas, podemos utilizar triangularização entre as duas posições para computar uma imagem de profundidade ou superfícies 3D. Porém, computar a disparidade utilizando correspondência entre *pixels* não é uma tarefa trivial. No caso mais simples, a busca sobre as correspondências nas imagens é um processo custoso de combinação exaustiva sobre as descritores de características. Além disso, esses descritores comumente utilizados necessitam de regiões da imagem com um certo nível de complexidade de textura para alcançar uma boa acurácia. Caso não tenham informações suficientes, é mais difícil combinar os pontos corretamente, resultando em reconstruções 3D imprecisas com falhas e buracos no modelo Szeliski (2011).

Mesmo que os métodos de combinação de imagens estéreo consigam obter uma boa performance na estimação de profundidade, ainda são insuficientes para computar o formato 3D completos de objetos devido à grande quantidade de regiões ocluídas. Sendo assim, em métodos conhecidos como *Multi-View Stereo* (MVS), são computadas várias imagens estéreo com determinadas posições de câmera Szeliski (2011). Contudo, essas soluções necessitam de uma calibração cuidadosa de múltiplas câmeras, necessitando de ambientes controlados e equipamentos de alto custo.

Retirando a necessidade de câmeras sincronizadas em pontos conhecidos, temos os métodos *Struct From Motion* (SFM) que, dada uma sequência de imagens, estimam a geometria 3D da cena e o movimento da câmera Szeliski (2011). Esses métodos funcionam primeiramente

detectando características em cada uma das imagens em sequência e então essas características são combinadas entre as imagens. Computando as matrizes de câmera utilizando mínimos quadrados não-lineares e com as correspondências entre características, a mesma triangularização empregada em abordagens MVS são aplicadas para recuperar a superfície 3D Szeliski (2011).

2.1.2 Estruturas de dados para modelagem espacial

Após a utilização de métodos para estimar pontos do cenário e a respectiva profundidade dos objetos, precisamos de representações exatas para computar representações 3D realistas dos cenários desejados. Existem diversas maneiras de representar superfícies 3D (e serão introduzidas nas seções seguintes) que podem ser categorizadas como: representação de superfícies explícitas (e.g. malhas (*meshes*) poligonais e *splines*), representações baseadas em pontos (e.g. nuvem de pontos) e representações volumétricas (e.g. *voxels*).

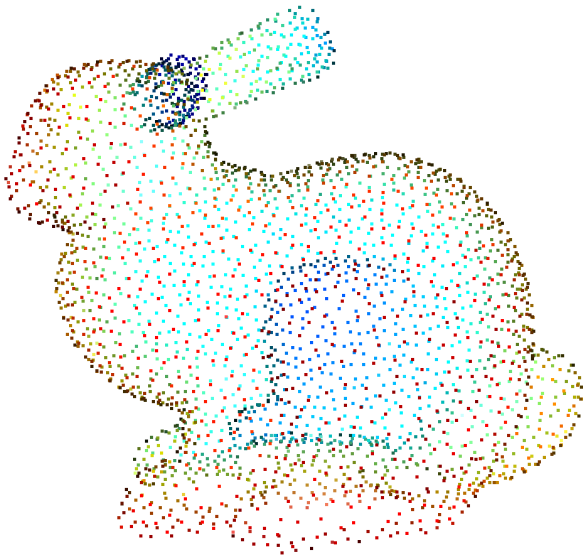


Figura 3: Exemplo de uma representação de nuvens de pontos. Fonte: Tutorial da ferramenta Open3D.

2.1.2.1 Representações baseadas em nuvens pontos

Nessa forma de representação, um conjunto de elementos no espaço Euclidiano e seus respectivos atributos (e.g. coordenadas espaciais, vetor normal, cor) são utilizados para modelar uma superfície, logo não necessitam de uma topologia predeterminada. Sendo assim, ao invés de armazenar os pontos vizinhos acoplados (como *meshes*) utilizam *funções de influência* para juntar os pontos próximos e visualizar a cena ou objeto modelado. Dependendo do tipo de *função de influência* e seus respectivos parâmetros, a topologia do modelo 3D pode ser modificada e interpolar dados 3D parciais com buracos Szeliski (2011). Uma das representações mais

utilizadas em computação gráfica são nuvens de pontos orientados por um vetor normal e um disco delimitado por um raio em uma superfície, conhecidas como elementos de superfície ou *surfels* Pfister *et al.* (2000). Podemos ver um exemplo de visualização de um modelo na Figura 3.

2.1.2.2 *Representações volumétricas*

Uma outra alternativa são as representações volumétricas, que modelam superfícies 3D como uma grade de blocos uniforme (*voxels*)(Figura 4). Estas grades podem ser armazenadas um valor de ocupação ou *Signed Distance Field* (SDF), onde *voxels* representam as distâncias em relação aos limites da superfície Szeliski (2011). Usando interpolação trilinear, os atributos armazenados nos elementos do *voxel* podem ser acessados para calcular informações do modelo em qualquer ponto dentro do *voxel*. Essa interpolação é utilizada em métodos de renderização baseados em amostragem como *ray casting*. Podemos ver um exemplo dessa estrutura na Figura 4.

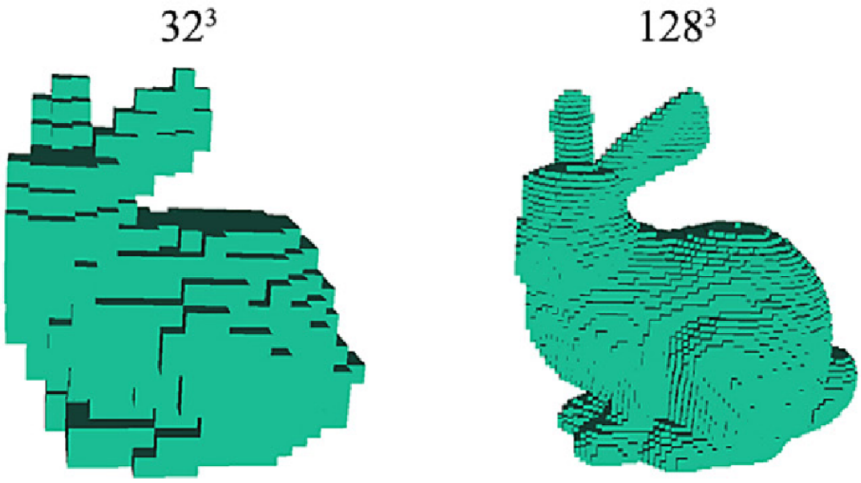


Figura 4: Exemplo de *voxels*. Fonte: Samavati & Soryani (2023).

2.1.2.3 *Representações de superfícies*

Um dos métodos mais comuns de representação são malhas poligonais (ou *meshes*), que armazenam pontos 3D (vértices), arestas (pares de vértices) e faces (conjunto de arestas formando um polígono) formando um grafo 3D. Essas representações permitem a criação de modelos bem detalhados e permitem operações úteis para visualização como interpolação, subdivisão e transformações não rígidas Szeliski (2011). Os *pipelines* de processamento gráficos e aceleradores gráficos (e.g. GPUs) são otimizados para processar e rasterizar bilhões de triângulos por segundo. A maioria das ferramentas de edição gráfica trabalham com malhas de triângulos, fazendo essa representação importante para criação de conteúdo. Na Figura 5 podemos ver um exemplo dessa representação.

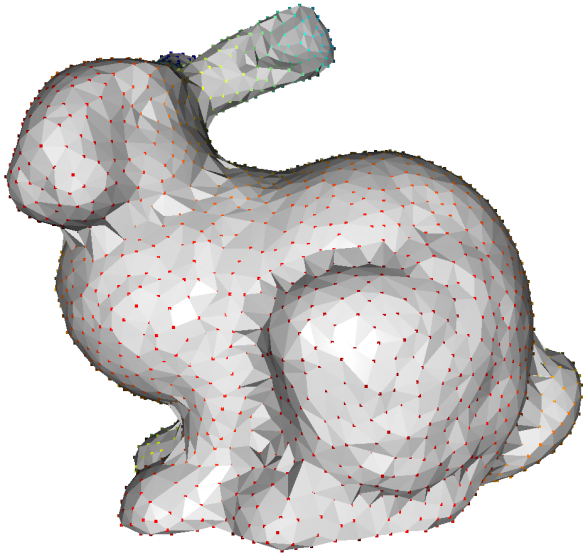


Figura 5: Exemplo de malhas 3D. Fonte: Tutorial da ferramenta Open3D..

2.1.3 Construindo *Meshes* com *Quadtrees*

A forma como as representações são construídas com o objetivo de desenvolver soluções para fotografia 3D pode resultar em diferentes complexidades de reconstrução. Por exemplo, extrair nuvens pontos através de um mapa de profundidade e construir uma malha utilizando técnicas como Kazhdan *et al.* (2006), possui um elevado custo computacional a depender da aplicação e do *hardware* onde será executado. Tradicionalmente, modelos 3D são construídos utilizando técnicas e representações específicas para o objeto desejado, como podemos ver em Szeliski (2011) utilizando como exemplo modelagem de arquitetura urbana, expressões faciais e movimentos do corpo. Uma das formas de simplificar a construção dessas estruturas de dados para modelagem 3D é fazer a busca pelos vértices e arestas utilizando imagens dos contornos dos objetos como guia. Em Berg *et al.* (2008) no Capítulo 14, são apresentados algoritmos de construção de malhas utilizando um conjunto de pontos.

2.1.3.1 Quadtrees para um conjunto de pontos

Um *quadtree* consiste em uma árvore na qual cada nó interno pode ser folha ou ter 4 nós filhos. Cada nó armazena os limites de um quadrado (ou região da imagem). Se um nó b tem filhos, então são armazenados nos filhos os limites dos quadrantes que compõem o quadrado em b. Isso implica que os quadrados armazenados nas folhas constituem subdivisões de um quadrado na raiz da árvore. Na Figura 6 temos um exemplo de um *quadtree* e suas subdivisões rotuladas como NE, NW, SW e SE (i.e. *North-East*, *North-West*, *South-West* e *South-East*) indicando a orientação que cada quadrante se localiza em relação ao seu nó mãe.

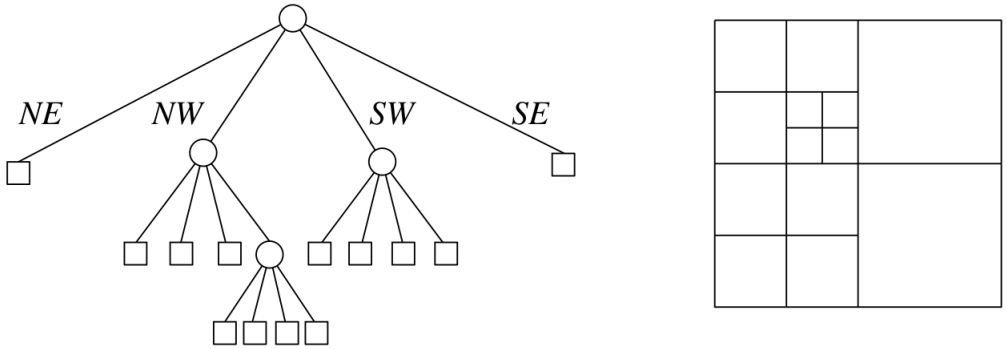


Figura 6: Na imagem à esquerda vemos um exemplo de como a estrutura de um *quadtree* se apresenta e no lado direito as respectivas regiões de uma imagem delimitadas pelas folhas. Fonte: Berg *et al.* (2008).

Como *quadtrees* podem ser utilizados em diferentes tipos de dados, construiremos essas árvores de acordo com os valores dos *pixels* no interior de uma imagem binária I. Assim, a recursão onde será feita a divisão das regiões continuará até que as regiões que o nó está delimitando não possuírem nenhum *pixel* $I(i,j) = 1$. Então, sendo $RI \in [i_0, i_e] \times [j_0, j_e]$ o quadrado (ou região) de uma imagem, definimos um *quadtree* para um conjunto de pontos $P = \{(i,j) | RI(i,j) = 1, i \in [i_0, i_e], j \in [j_0, j_e]\}$ da seguinte maneira:

- Se a cardinalidade $\|P\| \leq 1$ então o *quadtree* é formado por apenas uma folha onde o conjunto P e RI são armazenados.
- Se $\|P\| > 1$, considerando $i_{mid} = (i_0 + i_e)/2$, $j_{mid} = (j_0 + j_e)/2$, os quadrantes de RI e seus respectivos conjuntos P_{NW}, P_{NE}, P_{SW} e P_{SE} são definidos como:

$$
\begin{aligned}
P_{NW} &= \{(i,j) | RI(i,j) = 1, i \in [i_0, i_{mid}], j \in [j_0, j_{mid}]\}, \\
P_{NE} &= \{(i,j) | RI(i,j) = 1, i \in [i_0, i_{mid}], j \in [j_{mid}, j_e]\}, \\
P_{SW} &= \{(i,j) | RI(i,j) = 1, i \in [i_{mid}, i_e], j \in [j_0, j_{mid}]\}, \\
P_{SE} &= \{(i,j) | RI(i,j) = 1, i \in [i_{mid}, i_e], j \in [j_{mid}, j_e]\}.
\end{aligned}
$$

Em Algoritmo 1 podemos ver uma descrição em pseudo-código do algoritmo, que tem como entrada a imagem $I \in \mathbb{R}^{W \times H}$ e os limites da imagem $\{i_0, i_e, j_0, j_e\}$ (inicialmente $i_0 = 0, j_0 = 0$, $i_e = H - 1$ e $j_e = W - 1$) e retorna o *quadtree* τ. A estrutura *quadtree* utilizada no algoritmo tem as variáveis $\tau.i_0, \tau.i_e, \tau.j_0$ e $\tau.j_e$ que determinam a região da imagem que o nó τ delimita e as variáveis que representam os nós filhos $\tau.\tau_{NW}, \tau.\tau_{NE}, \tau.\tau_{SW}$, e $\tau.\tau_{SE}$. Após inicializar as variáveis de τ, é verificado se existe algum *pixel* $I(i,j) = 1$ na região $[i_0, i_e] \times [j_0, j_e]$. Em caso verdadeiro e se $(i_e - i_0) > 1$ e $(j_e - j_0) > 1$, esse processo é repetido em cada um dos 4 quadrantes que compõe a região $[i_0, i_e] \times [j_0, j_e]$ até que a recursão para computar os *quadtrees* dos 4 nós pare e assim retornar τ. No caso contrário, τ é retornado com filhos armazenando $\emptyset$.

Algoritmo 1: `Quadtree`

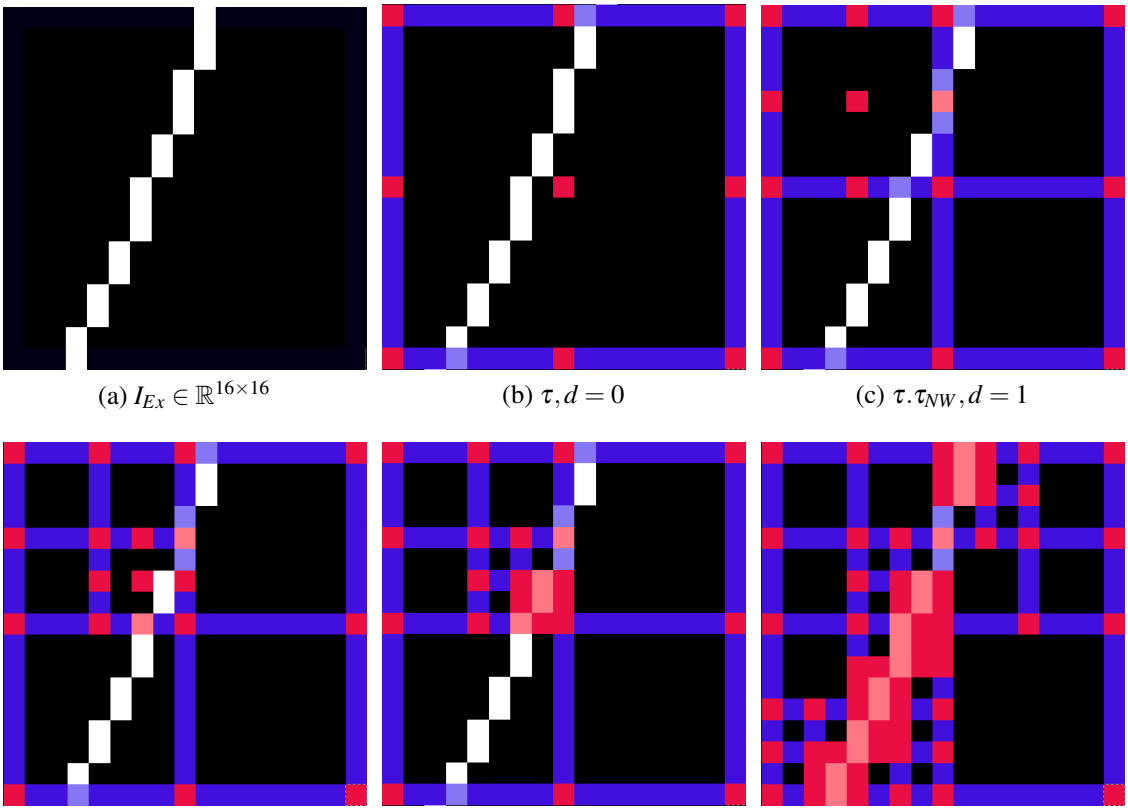
1 **Entrada:** $I \in \mathbb{R}^{H \times W}$, $i_0 \in \mathbb{N}$, $i_e \in \mathbb{N}$, $j_0 \in \mathbb{N}$, $j_e \in \mathbb{N}$
2 **Saída:** Um *quadtree* τ
3 $\tau.i_0, \tau.i_e, \tau.j_0, \tau.j_e \leftarrow i_0, i_e, j_0, j_e$
4 $\tau.\tau_{NW}, \tau.\tau_{NE}, \tau.\tau_{SW}, \tau.\tau_{SE} \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$
5 **if** $1 \in \{I(i,j) | i \in [i_0, i_e], j \in [j_0, j_e]\} \wedge (i_e - i_0) > 1 \wedge (j_e - j_0) > 1$ **then**
6 $\quad i_{mid} \leftarrow \frac{i_0 + i_e}{2}$
7 $\quad j_{mid} \leftarrow \frac{j_0 + j_e}{2}$
8 $\quad \tau.\tau_{NW} \leftarrow$ `Quadtree`$(I, i_0, i_{mid}, j_0, j_{mid})$
9 $\quad \tau.\tau_{NE} \leftarrow$ `Quadtree`$(I, i_0, i_{mid}, j_{mid}, j_e)$
10 $\quad \tau.\tau_{SW} \leftarrow$ `Quadtree`$(I, i_{mid}, i_e, j_0, j_{mid})$
11 $\quad \tau.\tau_{SE} \leftarrow$ `Quadtree`$(I, i_{mid}, i_e, j_{mid}, j_e)$
12 **return** τ

Na Figura 7 vemos exemplos de alguns passos do algoritmo para a imagem $I_{Ex} \in \mathbb{R}^{16 \times 16}$ de entrada (Figura 7(a)). Em Figura 7(b) vemos as regiões delimitadas inicialmente por $i_0 = 0, j_0 = 0$, $i_e = 15$ e $j_e = 15$ com as linhas roxas indicando a região delimitada pelos nós. Os pontos vermelhos representam i_0,j_e, j_0 e j_e dos nós filhos $\tau.\tau_{NW}, \tau.\tau_{NE}, \tau.\tau_{SW}$, e $\tau.\tau_{SE}$. Na Figura 7(c) vemos a primeira chamada da recursão no filho $\tau.\tau_{NW}$, criando mais 4 quadrantes, pois $I_{Ex}(i,j) = 0$ para $i = 0, \ldots, 8$ e $j = 0, \ldots, 8$. Em Figura 7(d) e 7(e) vemos os passos seguintes para os nós filhos de $\tau.\tau_{NW}$. Na Figura 7(f) vemos o resultado final das regiões delimitadas por todos os nós de τ.

2.1.3.2 De Quadtrees para Meshes triangulares

Agora que sabemos como um *quadtree* é construído, precisamos de um algoritmo que construa o *mesh* triangular para ser definida uma representação baseada nos vértices e suas respectivas conexões. Baseando-se em Berg *et al.* (2008), o Algoritmo 2, após o uso do Algoritmo 1, utiliza as extremidades dos quadrantes da imagem $I \in R^{W \times H}$ como vértices e são definidas as conexões entre os vértices para formar os triângulos da malha. Assim, consideraremos sem perda de generalidade uma região contendo 4 *pixels* como a região miníma, considerando-a um critério de parada na busca (linha 5 do Algoritmo 1). Após essas considerações, definimos no Algoritmo 2 a construção do *quadtree* utilizando imagens de contorno como guia (e.g. contornos de Canny). O algoritmo recebe como entrada um *quadtree* τ, o atual nó da busca τ_{node} (inicialmente $\tau = \tau_{node}$) e um conjunto de arestas *edges* (inicialmente vazio $\{\}$)[1]. Toda a computação é feita na busca sobre os filhos $\tau_{node} \in \{\tau_{NW}, \tau_{NE}, \tau_{SW}, \tau_{SE}\}$ de τ para computar as arestas. Caso τ_{node} não seja folha (i.e. $\tau_{node} \neq \emptyset$) é chamado `QuadtreeToMesh` com os 4 filhos de τ_{node} como entrada da função, além de τ e *edges* (linha 4 a 7). Se τ_{node} é folhas, então são computados os pontos $P_{corners} = \{(i_0, j_0), (i_0, j_e), (i_e, j_0), (i_e, j_e)\}$ das extremidades da região em τ_{node}. Esse pontos

[1] Aresta aqui são entendidas como um conjunto contendo dois pontos $edge = \{p_0, p_1\}$ sendo $p_0 = (i_0, j_0)$ e $p_1 = (i_1, j_1)$.



(a) $I_{Ex} \in \mathbb{R}^{16\times16}$ (b) $\tau, d = 0$ (c) $\tau.\tau_{NW}, d = 1$

(d) $\tau.\tau_{NW}.\tau_{SE}, d = 2$ (e) $\tau.\tau_{NW}.\tau_{SE}.\tau_{SE}, d = 3$ (f) $\tau, d = 0$

Figura 7: Em (a) temos a imagem I_{Ex} de entrada. Em (b) temos o primeiro passo do algoritmo, quando são delimitados os limites das regiões representadas pelo nó, em azul temos as bordas delimitadas pelos pontos extremos (pontos vermelhos). Após verificar a existência de pontos onde $I(i, j) = 1$ (pontos brancos) na região delimitada pelo nó τ na profundidade $d = 0$ em (c) vemos o inicio da recursão no filho $\tau.\tau_{NW}$. Após verificar a existência de pontos brancos na região delimitada pelo nó $\tau.\tau_{NW}$, as recursões nos nós $\tau.\tau_{NW}.\tau_{NW}$, $\tau.\tau_{NW}.\tau_{NE}$, $\tau.\tau_{NW}.\tau_{SW}$ retornam com seus nós com valor $\emptyset$, sendo classificados como folhas. Logo em (d) vemos a chamada da recursão para o nó $\tau.\tau_{NW}.\tau_{SE}$. Com isso chegamos na folha $\tau.\tau_{NW}.\tau_{SE}.\tau_{SE}$ na profundidade limite em (e). Repetindo o processo para os outros nós ficamos com o *quadtree* τ delimitado pelas regiões mostradas em (f).

serão entrada de `QuadtreeNeighborsCorners`, que retornará um conjunto de pontos das extremidades dos nós de τ que tocam a região delimitada por $P_{corners}$. Se o conjunto retornado P_{neigh} tiver cardinalidade $\|P\| > 4$, então são formadas arestas conectando os pontos $p \in P_{neigh}$ com seus vizinhos de P_{neigh} mais próximos e todos os pontos p são conectados com o ponto (i_{mid}, j_{mid}). Caso contrário, são conectados apenas os pontos p com seus vizinhos mais próximos. No final, é retornado o conjunto de arestas *edges*. Com um conjunto de arestas armazenados em *edges* conseguimos facilmente converter em malhas 3D pegando os pontos (i, j) e transforma-los de coordenadas de imagem para pontos (x, y) no espaço Euclidiano e recuperando a informação do eixo z através de um mapa de profundidade de mesmas dimensões que I.

Algoritmo 2: `QuadtreeToMesh`: O algoritmo recebe como entrada um *quadtree* τ e um conjunto de arestas inicialmente com valores $edges = \{\}$.

Entrada: τ, τ_{node}, $edges$
Saída: $edges$
if $\tau_{node} \neq \emptyset$ **then**
    $edges \leftarrow$ `QuadtreeToMesh`$(\tau, \tau_{node}.\tau_{NW}, edges)$
    $edges \leftarrow$ `QuadtreeToMesh`$(\tau, \tau_{node}.\tau_{NE}, edges)$
    $edges \leftarrow$ `QuadtreeToMesh`$(\tau, \tau_{node}.\tau_{SW}, edges)$
    $edges \leftarrow$ `QuadtreeToMesh`$(\tau, \tau_{node}.\tau_{SE}, edges)$
else
    $i_0, i_e, j_0, j_e \leftarrow \tau_{node}.i_0, \tau_{node}.i_e, \tau_{node}.j_0, \tau_{node}.j_e$
    $P_{corners} \leftarrow \{(i_0, j_0), (i_0, j_e), (i_e, j_0), (i_e, j_e)\}$
    $P_{neigh} \leftarrow$ `QuadtreeNeighborsCorners`$(\tau, P_{corners})$
    if $\|P_{neigh}\| > 4$ **then**
        $i_{mid} \leftarrow \frac{i_0 + i_e}{2}$
        $j_{mid} \leftarrow \frac{j_0 + j_e}{2}$
        $edges \leftarrow edges \bigcup$ Todas as arestas que conectam os pontos $p \in P_{neigh}$ vizinhos e todos p formam arestas com (i_{mid}, j_{mid})
    else
        $edges \leftarrow edges \bigcup$ Todas as arestas que conectam os pontos $p \in P_{neigh}$ vizinhos
return τ

Na Figura 8 temos um exemplo do funcionamento do algoritmo, utilizando o *quadtree* resultante da Figura 7. Na Figura 8(a), temos as linhas pretas delimitando os intervalos $[\tau_{node}.i_0, \tau_{node}.j_0] \times [\tau_{node}.i_e, \tau_{node}.j_e]$ de cada um dos nós de τ e os pontos azuis representando o centro dos *pixels* da imagem de entrada (Figura 7a). Na Figura 8(b), temos as extremidades $\{(i_0, j_0), (i_0, j_e), (i_e, j_0), (i_e, j_e)\}$ de cada um dos nós de τ representados com a cor vermelha. Em Figura 8(c), vemos o *mesh* final com cada uma das arestas conectadas e os pontos (i_{mid}, j_{mid}) representados pela cor verde. Nas Figuras 8(d) e 8(e) temos exemplos da conexão das arestas de um nó caso $\|P_{node}\| > 4$ e $\|P_{node}\| = 4$ respectivamente.

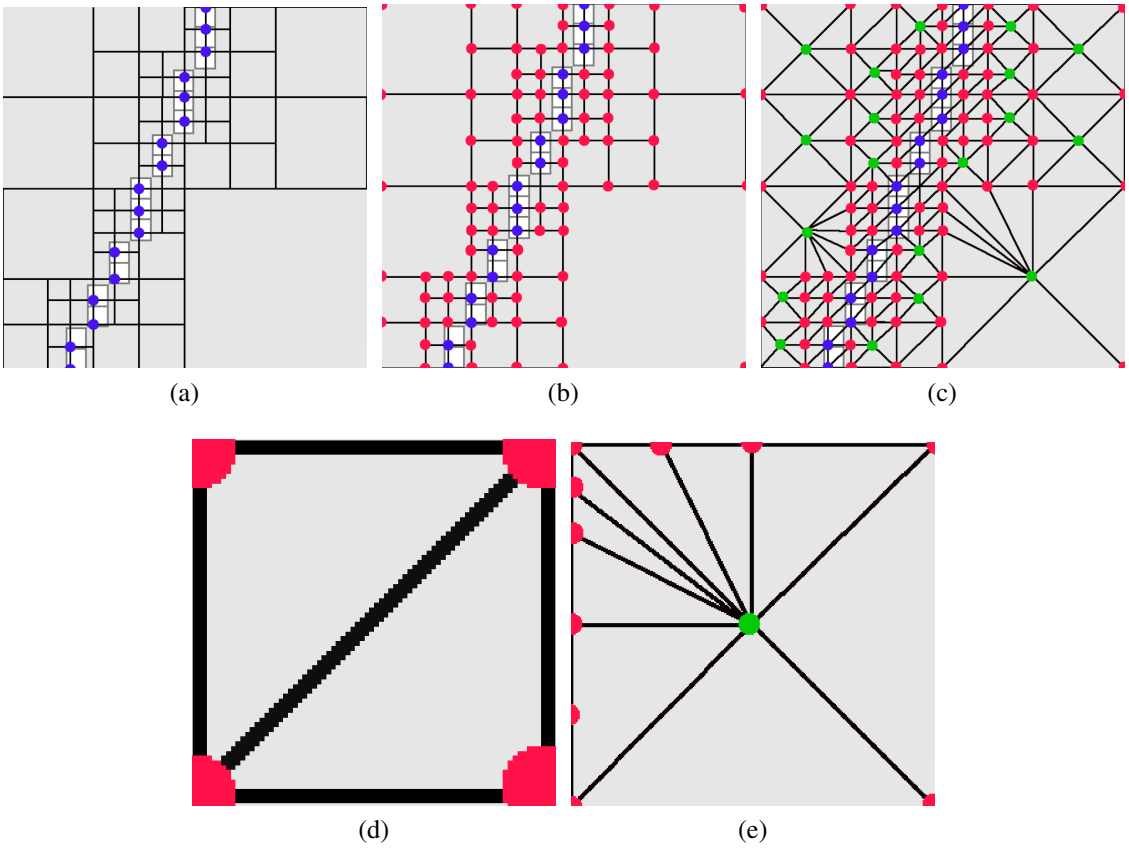


Figura 8: Exemplo da construção utilizando o *quadtree* resultante da Figura 7. Em (a) vemos a configuração inicial apresentando em azul os pontos utilizados para formar o *quadtree*, com as regiões armazenadas nele representadas pelas linhas pretas. Em (b) são apresentados em vermelho os pontos que representam as extremidades das regiões armazenadas nos nós de τ. No final do algoritmo é retornada uma malha equivalente a apresentada em (c), representada pelas arestas e os pontos nelas contidos armazenadas em *edges*. Nas figuras (d) e (e) são *zooms* aplicados na figura (c) em regiões que representam os casos em que $\|P_{neigh}\| = 4$ (figura (d)) e $\|P_{neigh}\| > 4$ (figura (e)).

2.1.4 Mapeamento de Textura

Depois de construída uma estrutura de dados que armazena o modelo 3D, é necessário um mapeamento de textura que descreve a distribuição de cor na superfície do objeto. Primeiramente é definida uma parametrização por coordenadas de textura (u,v) como uma função dos pontos amostrados da superfície. Tradicionalmente, uma forma comum e simples de definir essa função é associá-la a cada triângulo. Ou seja, é computada uma bijeção entre cada (u,v) correspondente a um ponto da imagem e um ponto (x,y,z) no espaço. Uma vez definidos (u,v) para cada vértice do triângulo, podemos utilizar as imagens de referência do objeto como fonte de textura utilizando projeção de mapeamento de textura Szeliski (2011).

Na Figura 9 vemos alguns exemplos de como funciona o mapeamento em textura. Na imagem superior vemos uma esfera com linhas em preto representando a malha do modelo 3D e a imagem equivalente que sera mapeada nas mesmas regiões delimitadas na esfera. Aplicando informação de cor temos o resultado como mostrado no modelo do globo terrestres. No exemplo do cubo vemos que o mapeamento de textura é como colar uma imagem em cada superfície dos polígonos presentes na malha.

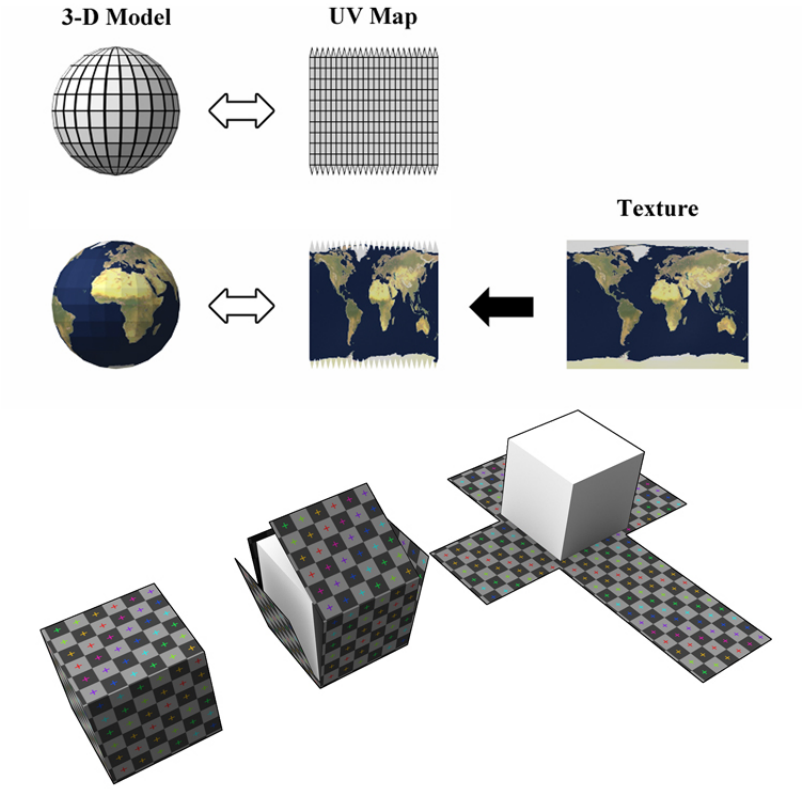


Figura 9: Exemplos de mapeamento de textura. Fonte: Retirado do site *link*.

2.2 APRENDIZAGEM PROFUNDA

Recentemente, os métodos que se baseiam no uso de *Artifitial Neural Networks* (ANN), mostraram-se capaz de inferir padrões estatísticos dos dados além de serem utilizados como

extratores de características mais complexas, sendo aplicadas e tornando-se o estado da arte nas mais diversas áreas Goodfellow *et al.* (2016). ANNs podem ser definidas como funções parametrizadas, nas quais os parâmetros são otimizados utilizando gradiente descendente Goodfellow *et al.* (2016). Apesar de serem mais eficientes em tempo e em generalização, dependem de bancos de dados robustos (i.e. com alta variabilidade estatística) e de um elevada capacidade computacional. Porém, com o recente avanço na eficiência das GPUs e da alta viabilidade de banco de dados públicos para problemas mais básicos (e.g. classificação e segmentação), tem contornado esses problemas.

Particularmente, iremos focar nas abordagens para geração de fotografia 3D utilizando apenas uma imagem. As ANN são consideradas aproximadores universais de funções, podendo ser utilizado para os mais diversos problemas e tem sido foco de pesquisa o desenvolvimento de abordagens com redes neurais para modelagem 3D. Sendo assim, nessa seção introduziremos os fundamentos teóricos básicos para definir arquiteturas de ANN, assim como arquiteturas conhecidas na literatura que inspiraram os trabalhos mais recentes na área de fotografia 3D utilizando aprendizagem profunda.

2.2.1 Redes Neurais e *Multilayer Perceptron*

As arquiteturas de ANN são vistas como aproximadores de funções não lineares baseadas no uso de gradiente descendente para minimizar o erro em relação a uma função alvo. O início da utilização de gradiente descendente para aproximar funções no século XIX Cauchy (1847) inspirou a construção de vários métodos de aprendizagem de máquina, incluindo *perceptron* Rosenblatt (1958). Os fundamentos teóricos mais relevantes das ANN modernas foram desenvolvidos em meados dos anos 1980s, considerando trabalhos notáveis como Werbos (1981); Lecun (1985); Hinton *et al.* (1986); Rumelhart *et al.* (1986). A maior parte das melhorias de performance nas ANN de 1986 até os dias de hoje é devido a dois fatores principais. O primeiro é o desenvolvimento de conjuntos de dados (*datasets*) robustos (i.e. com uma grande quantidade de dados), reduzindo o desafio de obter uma boa generalização com redes neurais. O segundo é devido ao tamanho das redes que também estão maiores devido ao desenvolvimento de computadores com maior poder computacional. Contudo, poucas alterações teóricas ou algorítmicas têm melhorado a performance das ANN notavelmente Goodfellow *et al.* (2016).

Devido a seu impacto histórico, uma das arquiteturas com operações importantes para a aprendizagem profunda são as MLP, formadas por *perceptrons* Rosenblatt (1958). Essas redes também são conhecidas como *feedforward*, pois o fluxo de informação é processado a partir de uma entrada x e é propagada por uma cadeia de funções concatenadas até a saída y. Por exemplo, para um classificador $y = f(x)$, a rede define um mapeamento $y = f^*(x, \theta)$ onde $f^* = f^0 \circ f^1 \circ \cdots \circ f^{L-1}$ onde $L \in \mathbb{N}$ é o número de funções intermediárias (ou camadas como é comumente chamado). Várias aplicações comerciais tem como base as arquiteturas com camadas *feedforward*, por exemplo, detecção de objetos em fotos utilizando camadas de redes

convolucionais são um tipo de redes *feedforward* Goodfellow *et al.* (2016).

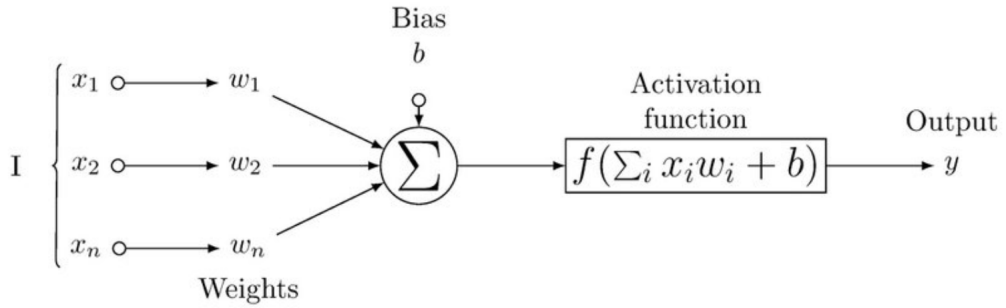


Figura 10: Fonte: Pérez-Enciso & Zingaretti (2019)

Uma arquitetura MLP é geralmente representada como nós conectados (i.e. um grafo ou uma rede) em camadas, ou seja, uma primeira coluna de nós nos quais a saída será a entrada da próxima coluna de nós e assim em diante. Esse arranjo pode ser visualizado na Figura 11, onde a primeira coluna de nós é considerada como entrada da arquitetura, as colunas intermediárias são conhecidas como camadas escondidas (*hidden layers*) e o último nó seria a saída da arquitetura. Considerando a Figura 10 como uma representação de um nó (ou neurônio) de uma rede MLP, podemos definir a função f^l_{node} que representa esse nó como

$$f^l_{node}(\mathbf{x},\theta) = \sum_{i=0}^{D-1} x_i w_i + b = \phi(\mathbf{x}W + b) \tag{2.1}$$

sendo $\mathbf{x} = (x_0,\ldots,x_{D-1}) \in \mathbb{R}^D$ a entrada, $\theta = \{W,b\}$ os parâmetros que serão aprendidos onde $W \in \mathbb{R}^D$ os pesos e $b \in \mathbb{R}$ conhecido como viés (*bias*) e ϕ é conhecida como função de ativação.

Como é mostrado na equação 2.1, podemos representar a multiplicação e soma dos pesos como um produto escalar entre vetores. Sendo assim, definimos a função f^0 como a primeira camada da arquitetura da seguinte forma

$$f^0(\mathbf{x},\Theta^0) = \phi(\mathbf{x}\mathbf{W}^0 + \mathbf{b}^0), \tag{2.2}$$

onde $\Theta^0 = \{\mathbf{W^0},\mathbf{b^0}\}$ são os parâmetros que serão aprendidos, sendo $\mathbf{W}^0 \in \mathbb{R}^{D\times D^0}$ os pesos, $\mathbf{b}^0 \in \mathbb{R}^{D^0}$ os vieses e $D^0 \in \mathbb{N}$ a quantidade de nós na camada. Com isso, como foi dito anteriormente podemos definir uma arquitetura com $L \in \mathbb{N}$ camadas como uma composição de funções $f^0 \circ \cdots \circ f^{L-1}$, sendo as funções f^l de cada camada

$$f^l(\mathbf{x},\Theta^l) = \phi(\mathbf{x}^{l-1}\mathbf{W}^l + \mathbf{b}^l), \tag{2.3}$$

onde $\mathbf{x}^{l-1} \in \mathbb{R}^{D^{l-1}}$ é a saída da camada anterior, $\Theta^l = \{\mathbf{W^l},\mathbf{b^l}\}$ são os parâmetros que serão aprendidos, sendo $\mathbf{W}^l \in \mathbb{R}^{D^{l-1}\times D^l}$ os pesos, $\mathbf{b} \in \mathbb{R}^{D^l}$ os vieses, $D^l \in \mathbb{N}$ a quantidade de nós na camada e $D^{l-1} \in \mathbb{N}$ a quantidade de nós da camada anterior.

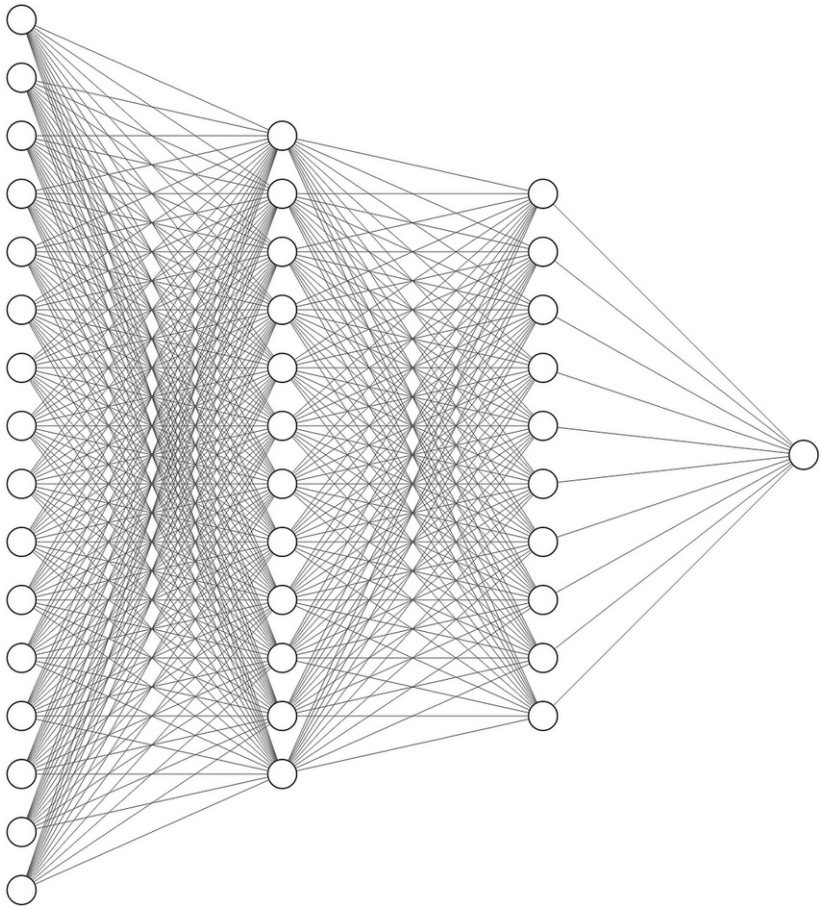


Figura 11: Demonstração gráfica de uma rede MLP.

Sabendo como são definidas as funções que representam as camadas de uma arquitetura de MLP, para encontrar os valores de $\Theta = \{\Theta^0, \ldots, \Theta^{L-1}\}$ que melhor aproximam a função alvo f, o método de aprendizado consiste em otimizar Θ utilizando uma função de perda $\mathscr{L}$. Essa otimização, chamada de treinamento, ocorre de maneira iterativa, utilizando otimizadores baseados em gradiente que buscam convergir a função de perda a uma assíntota horizontal. Considerando o classificador $y = f(x)$, podemos por exemplo definir uma função de perda $\mathscr{L}$ baseada no erro quadrático entre a inferência de f e as classes $\mathscr{Y} = \{y_0, \ldots, y_{N-1}\}$ de um conjuntos de dados $\mathscr{X} = \{\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}\}$ como

$$\mathscr{L}(\Theta) = \sum_{i=0}^{N-1} (f(\mathbf{x}_i, \Theta) - y_i)^2. \tag{2.4}$$

Podemos chamar a função $\mathscr{L}$ na Equação 2.4 de *Mean Squared Error* (MSE) e é uma das funções de perda convencionais nos treinamentos de redes ANN. Após a escolha de uma função de perda $\mathscr{L}$, são computadas as derivadas parciais em relação a cada peso θ^l_{mn} presentes na camada l e na posição m e n na matriz de pesos $\mathbf{W}^l$ e são atualizados utilizando gradiente descendente. Um hiper-parâmetro (não aprendível) importante para o gradiente descendente é a taxa de aprendizagem, que é responsável por ponderar a velocidade de aprendizagem da rede (o

ritmo que os parâmetros são atualizados). É chamada de *back-propagation* a propagação por toda a rede, através da regra da cadeia, da atualização do pesos através da derivada parcial de uma função de perda.

Além do gradiente descendente e da função de perda MSE, existem diversos métodos de otimização e funções de perda na literatura. Exemplos notáveis de otimizadores são o Adam Kingma & Ba (2015) (otimização baseada em gradiente e estimadores adaptativos de momentos de baixa ordem), gradiente descendente estocástico Sutskever *et al.* (2013); Rosenblatt (1958) e Adagrad Duchi *et al.* (2011). Como as funções de perda variam de problema a formas de treinamento, consideramos como exemplos notáveis a função de entropia cruzada e a *softmax* Goodfellow *et al.* (2016).

2.2.2 Redes Neurais Convolucionais

As redes convolucionais (*Convolutional Neural Networks* (CNN)) tem um papel importante na história da aprendizagem profunda, sendo responsáveis pelo desenvolvimento dos primeiros modelos de *deep learning* a performarem bem em aplicações comerciais Goodfellow *et al.* (2016). Esse tipo de arquitetura tem, consequentemente, se destacado na construção de métodos de fotografia 3D. Para entender como é definida uma camada de convolucional, primeiramente precisamos entender o que é uma operação de convolução. Podemos definir essa operação considerando uma imagem $I \in \mathbb{R}^{H \times W}$ e uma janela (ou *kernel*) $K \in \mathbb{R}^{H_K \times W_K}$ como

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.5}$$

sendo i e j os índices nas respectivas dimensões do *pixel* na imagem de saída $S \in \mathbb{R}^{H \times W}$, m e n são os índices dos pesos a serem aplicados na imagem de entrada. Na Figura 13 é ilustrado como podemos visualizar a operação de convolução. A motivação para a utilização desse tipo de operação nas arquiteturas de ANN é pela capacidade de oferecer três vantagens: aplicar iterações espaças (i.e. pelo fato do *kernel* ser menor que a entrada no tamanho das dimensões, os mesmos pesos são aplicados em diferentes regiões da imagem); compartilhamento de parâmetros (i.e. os pesos podem ser aplicados a diferentes funções dentro da arquitetura); e representações equivariantes (i.e. a saída é condicionada às informações da entrada).

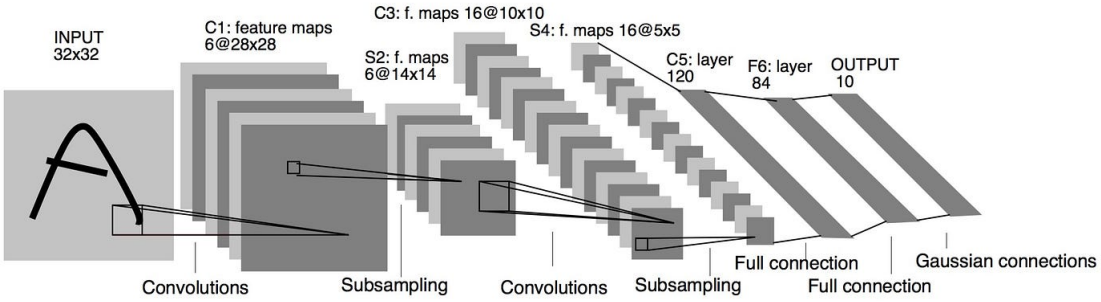


Figura 12: Fonte: Lecun *et al.* (1998)

A primeira arquitetura utilizando operações convolucionais foi proposta por Lecun *et al.* (1998) e a estrutura geral pode ser vista na Figura 12. O problema que essa arquitetura busca modelar é a de classificação de imagens, no caso, classificação de dígitos numa imagem em escala de cinza. As 4 primeiras camadas consistem em camadas convolucionais seguidas de camadas de sub amostragem (*subsampling*), conhecidas como *pooling* que serão descritas mais adiante nesse capítulo. Após isso, são aplicadas camadas totalmente conectadas (*fully connected*) que são basicamente as MLPs discutidas anteriormente. A saída $y \in \mathbb{R}^{10}$ dessa arquitetura representa a probabilidade da imagem de entrada $\mathbf{x} \in \mathbb{R}^{32\times32}$ pertencer a cada uma das 10 classes.

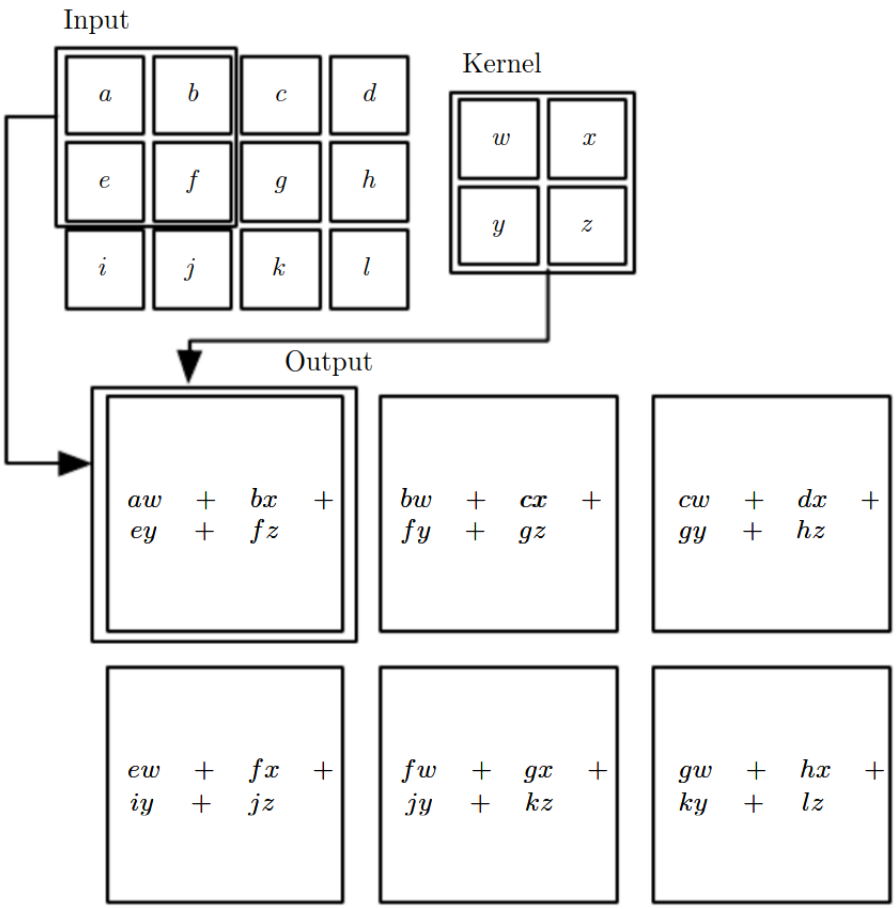


Figura 13: Fonte: Goodfellow *et al.* (2016)

A maior contribuição em Lecun *et al.* (1998) foi a introdução de camadas convolucionais com pesos a serem aprendidos.Utilizando a operação descrita na seção 2.2.2, dada uma entrada $I^{l-1} \in \mathbb{R}^{H^{l-1}\times W^{l-1}\times C^{l-1}}$ (que seria a saída da camada anterior $l-1$ ou a entrada da arquitetura), consideramos a saída $I^l \in \mathbb{R}^{H^l\times W^l\times C^l}$ de uma camada convolucional como

$$I^l(i,j,c) = (I^{l-1} * \mathbf{K}^l)(i,j) = \phi\left(\sum_m \sum_n \sum_p I^{l-1}(i-m, j-n, p)\mathbf{K}^l(p,m,n,c)\right) + \mathbf{b} \qquad (2.6)$$

onde $\mathbf{K}^l \in \mathbb{R}^{C^{l-1}\times H_k\times W_k\times C^l}$ são os pesos a serem aprendidos utilizados na convolução, ϕ a função

de ativação e $\mathbf{b} \in \mathbb{R}^{C^l}$ o viés.

Mesmo com o sucesso da LeNet em aplicações reais como detecção de fraudes bancárias Goodfellow *et al.* (2016), apenas após o sucesso da AlexNet Krizhevsky *et al.* (2012) na competição de classificação de imagens ImageNet Russakovsky *et al.* (2014), as arquiteturas de ANN com camadas convolucionais (ou CNN) começaram a ganhar o interesse comercial que existe atualmente Goodfellow *et al.* (2016). Krizhevsky *et al.* (2012) também introduz o uso de mais GPUs em paralelo para armazenar e executar a inferência de uma arquitetura mais robusta, capaz de generalizar com maior acurácia o banco de dados ImageNet.

Outras arquiteturas de impacto, que influenciam trabalhos até os dias atuais são as UNetRonneberger *et al.* (2015), ResNetHe *et al.* (2016) e VGGSimonyan & Zisserman (2014). Em UNet é introduzida uma arquitetura com codificadores (*encorders*) e decodificadores (*decoders*) para segmentação de imagens. Outra contribuição importante são as *skip-connections* permitindo o compartilhamento de informação entre o codificador e o decodificador. A UNet serve como base para diversas arquiteturas que tem imagens como entrada e saída, como inferência de mapas de disparidade (ou profundidade) Alhashim & Wonka (2018), problemas de computação fotográfica Chen *et al.* (2018); Shi *et al.* (2016) e preenchimento de cor (*inpainting*) Liu *et al.* (2018).

Tanto a ResNet quanto VGG também foram ganhadoras do desafio da ImageNet. Na ResNet He *et al.* (2016) foram introduzidas as camadas residuais para evitar *gradient vanishing*, quando os gradientes da rede no treinamento são baixos devido à alta quantidade de camadas convolucionais, levando à baixa generalização (*underfitting*). Em VGG Simonyan & Zisserman (2014) são apresentadas diferentes arquiteturas com niveis diferentes de profundidade, analisando seu desempenho nos bancos de dados. Tanto a ResNet quanto VGG pré-treinados na ImageNet são utilizados como extratores de características em trabalhos voltados para NVS Niklaus *et al.* (2019); Zhou *et al.* (2018b); Srinivasan *et al.* (2019).

Uma das chaves para o sucesso das redes CNN vem da sua versatilidade na construção de arquitetura, permitindo a adaptação de diferentes formas de treinamento e a criação de novos operadores. Introduzidos em Vaswani *et al.* (2017), os operadores *transformer* tiveram um papel fundamental na construção de arquiteturas para problemas envolvendo dados sequenciais e na composição das famosas arquiteturas GPT 3 e 4 OpenAI *et al.* (2023). Depois adaptados para imagens em Dosovitskiy *et al.* (2021) e utilizados nos atuais estados da arte das redes de difusão Rombach *et al.* (2022); Ho *et al.* (2020) em problemas de texto para imagens (*text-to-image*).

Um dos esquemas de treinamento de maior impacto são as redes *Generative Adversarial Networks* (GAN) Goodfellow *et al.* (2014). Consiste em uma estratégia, inspirada em teoria dos jogos, onde duas redes são utilizadas de maneira adversaria durante o treinamento. Isso significa em resumo que uma rede gera imagens e outra classifica, onde a imagem de entrada é rotulada se faz parte de um conjunto ou não. As duas redes são treinadas ao mesmo tempo. Com crescimento do interesse no desenvolvimento de redes GAN Cheng *et al.* (2020), as pesquisas voltaram-se para diferentes aspectos dos modelos como aplica GAN a novos problemas como

geração de imagem Karras *et al.* (2021), *text-to-image* Zhu *et al.* (2019) e também para geração de objetos 3D Luo *et al.* (2021); Niemeyer & Geiger (2021).

Apesar de CNNs serem menos computacionalmente custosas que MLPs, as arquiteturas no estado da arte tornaram-se cada vez maiores, com mais parâmetro e profundidade. Um exemplo são as atuais redes de difusão Rombach *et al.* (2022); Ho *et al.* (2020), que podem chegar a bilhões de parâmetros Saharia *et al.* (2022). Por esse motivo, existem trabalhos que buscam reduzir a quantidade de parâmetros desenvolvendo métodos para treinar redes menores Gou *et al.* (2021). Focando em reduzir o tempo de inferência, em Chollet (2017) e Howard *et al.* (2017) são apresentadas modificações na operação de convolução para diminuir a complexidade computacional das camadas. Em Li *et al.* (2022) vemos esse esforço para camadas com operações *transformer*.

As representações de objetos e cenas 3D e seus métodos de visualização (renderização), constituem ferramentas poderosas de modelagem. Junto a isso, redes convolucionais e MLP fornecem formas de especialização de redes neurais para trabalharem com dados com topologia tabular. Conhecendo os fundamentos teóricos básicos e trabalhos importantes para formação dos campos da fotografia 3D e aprendizagem profunda é necessária a investigação de abordagens que buscam desenvolver soluções para NVS combinando esses conhecimentos e as possibilidade de uso em dispositivos móveis.

3

TRABALHOS RELACIONADOS

Para sumarizar as abordagens para NVS existentes na literatura que podem ser utilizadas para fotografia 3D, falaremos das principais abordagens para renderização neural (e.g. NeRF) e das baseadas em mapas de profundidade (e.g. nuvens de pontos, MPI e LDI). Cada um desses trabalhos serão analisados para o uso em dispositivos móveis. Inicialmente serão apresentados os métodos que alcançam uma modelagem com maior precisão e que atualmente são o estado da arte nas métricas de qualidade. Sendo assim, as abordagens NeRF serão as primeiras a serem apresentadas. Por necessitarem de um alto poder computacional e utilizarem um conjunto de imagens para a otimização dos parâmetros, também apresentaremos os esforços para diminuir o tempo de convergência do modelo de uma cena e na redução da quantidade de imagem para o treinamento.

Após isso, apresentaremos os métodos que buscam generalizar a construção da estrutura 3D utilizando apenas uma imagem e uma arquitetura treinada. Focaremos especialmente nas técnicas com menor custo computacional, como as MPI. Para comparar com o método proposto, serão levantados trabalhos que constroem *meshes* ou LDI a partir de mapas de profundidade e redes de preenchimento de cor (*inpainting*). Como neste trabalho é apresentada uma nova técnica de construção de malhas 3D, serão apresentados trabalhos voltados para a construção de *meshes* utilizando técnicas de *deep learning*.

3.1 RENDERIZAÇÃO NEURAL

A geração de imagem e vídeo foto-realistas é um dos objetivos da computação gráfica. A estratégia dos métodos e representações desenvolvidas nos últimos anos é emular o comportamento físico de uma câmera real. Assim, parâmetros físicos precisam ser especificados para o processo de renderização. Esses parâmetros podem ser descrições geométricas e da composição material do objeto (e.g. cor, opacidade, reflectância). Para aproximar a forma como as imagens reais são capturadas, métodos baseados em aproximação matemática e em heurísticas são utilizados para renderizar imagens em tempo real, apesar de menor realismo nas imagens geradas. Devido ao sucesso das redes neurais em tarefas de visão computacional, a renderização neural é uma área emergente que permite o aprendizado de ANNs através de um método de

renderização diferenciável utilizando as observações de um objeto ou cena. A ideia por trás da renderização neural é a combinação de técnicas tradicionais de computação gráfica (baseadas em modelos físicos) e as vantagens oferecidas pela aprendizagem profunda. Nessa revisão, das abordagens de renderização neural existentes, focaremos em talvez na de maior impacto que é a NeRF.

NeRF, inicialmente proposta em Mildenhall *et al.* (2021), é uma das abordagens de NVS mais populares e exploradas nos últimos anos. Diferente de trabalhos anteriores, funciona amostrando pontos do espaço (x, y, z) de um raio r que tem origem em uma posição de câmera $o = (x_o, y_o, z_o)$ e direções $\{\theta, \phi\}$ e utilizando esses dados como entrada para um rede MLP, como é mostrado na Figura 14. Com a predição da cor e da densidade de cada ponto amostrado ao longo do raio $r(t) = o + td$ (onde d é o vetor de direção resultante de (θ, ϕ) e t é a posição no raio) os *pixels* da imagem gerada são calculados através da seguinte fórmula de renderização volumétrica (*volumetric rendering*) dos campos de radiância

$$C(r) = \sum_{i=1}^{N} T_i \left(1 - e^{(-\sigma_i \delta_i)}\right) c_i, \tag{3.1}$$

onde $T_i = e^{-\sum_{j=0}^{i-1} \sigma_j \delta_j}$, N representa o número de pontos amostrados, $\delta_i = t_{i+1} - t_i$ representa a distância entre os pontos vizinhos t_{i+1} e t_i, σ representa a densidade e c a cor preditas para o ponto no raio r. Como é mostrado em Fridovich-Keil and Yu *et al.* (2022) o sucesso do NeRF recai sobre essa forma de renderização volumétrica.

Entretanto, o treinamento do NeRF original para a MLP generalizar uma cena pode durar em torno de 24 horas em 4 GPUs RTX2080 e demorar cerca de 30 segundos para renderizar apenas uma imagem 1280x720 Zhang *et al.* (2020). Buscando contornar essa limitação, diminui-se o numero de chamadas da rede no treinamento em Kurz *et al.* (2022) utilizando duas redes menores. Uma que irá inferir a densidade num conjunto de amostras do raio (ao invés de por ponto), que será utilizada para descartar amostras com menos densidade, e outra para inferir a cor e a densidade das amostras selecionadas. Em Zhang *et al.* (2023b) busca-se acelerar o treinamento predefinindo as subdivisões nas imagens alvo (*ground truth*), utilizando *quadtrees*, para determinar quais regiões do plano projetivo, a partir da posição de câmera, serão computados um número menor de raios na renderização.

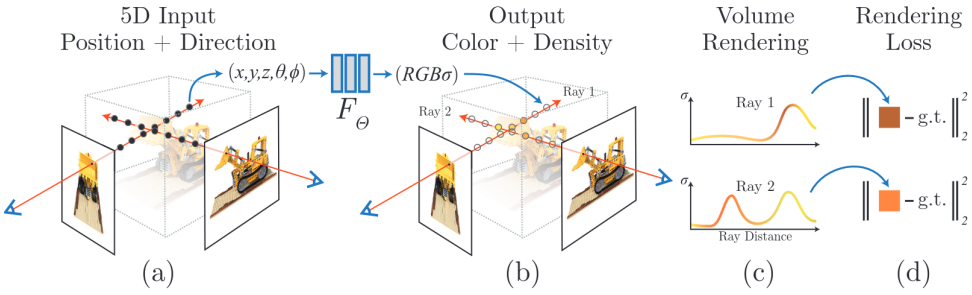


Figura 14: Fonte: Mildenhall *et al.* (2021)

Para acelerar o tempo de renderização, vemos em Reiser *et al.* (2021) a estratégia de colocar pequenas MLPs em um agrupamento de *voxels*, onde cada MLP infere os pontos presentes dentro do *voxel*. Liu *et al.* (2020) também acelera a renderização utilizando *voxels*, porém define um limiar de densidade para selecionar quais *voxels* vão ter pontos amostrados. A estratégia utilizadas por Garbin *et al.* (2021) e Yu *et al.* (2021a) para acelerar a renderização pós treinamento é o armazenamento da informação sobre o espaço Euclidiano modelado pela MLP. Garbin *et al.* (2021) permite o armazenamento propondo a utilização de duas MLPs, uma para características dependentes de posição e outras de direção que serão combinadas para computar cor e densidade. No caso de Yu *et al.* (2021a) vemos uma estratégia de treinamento que faz com que o campo neural de radiância (*neural randiance fields*) fique armazenado numa *octree*.

A abordagem mais notável tanto na aceleração do treinamento quanto na renderização é Fridovich-Keil and Yu *et al.* (2022). Diferente de outras abordagens baseadas em NeRF, em Fridovich-Keil and Yu *et al.* (2022) não são utilizadas redes MLP para inferir o conteúdo de cor e densidade do espaço. Ao invés disso, utiliza *voxels* esparsos onde os vértices são parâmetros aprendíveis e a informação interna é calculada utilizando interpolação trilinear. Isso é possível devido à fórmula de renderização volumétrica, que permite um *pipeline* totalmente diferenciável, fazendo que os parâmetros sejam otimizados assim como os pesos de uma MLP. Isso resulta em um treinamento de 11 minutos em uma GPU Titan RTX (o que demandaria em torno de 1 dia para NeRFs) para cenários limitados (*bounded scenes*) e 27 minutos para cenários ilimitados (*unbounded scenes*).

Todos os trabalhos acima se limitam a modelar objetos estáticos e efeitos de luz constantes. Para cenários mais desafiadores como modelagem de cenas cotidianas ou objetos maiores como construções, essas abordagens não são adequadas. Em Martin-Brualla *et al.* (2021) é definida uma rede para inferir as regiões das imagens utilizadas no treinamento com objetos que obstruem partes da cena alvo a ser modelada. Assim, a arquitetura completa consegue modelar cenários ilimitados (*unbounded*) a partir de imagens em diversas condições de iluminação e objetos não estáticos.

Como a maior parte dos trabalhos baseados em NeRF utilizam duas ou mais imagens para treinar os modelos, em Yu *et al.* (2021b); Xu *et al.* (2022) são propostos modelos que são treinados para generalizar mais de um cenário. Yu *et al.* (2021b) extrai características da imagem do cenário desejado através de uma arquitetura pré-treinada e utiliza como entrada da MLP do NeRF, além das coordenadas do ponto e as direções. Em Xu *et al.* (2022) é apresentado um processo de treinamento semi-supervisionado, onde são gerados pseudo-rótulos geométricos e semânticos para guiar o treinamento.

Para dispositivos *mobile*, em Chen *et al.* (2023) apresenta uma modificação nos algoritmos de treinamento de NeRFs para sejam renderizados utilizando *pipelines* tradicionais de rasterização de polígonos. Sendo assim, a abordagem descrita em Chen *et al.* (2023) converte NeRF em uma representação de polígonos texturizados, onde as texturas são características utilizadas como entradas de pequenas MLPs para cada *pixel*. Apesar de alcançar renderização

em tempo real utilizando malhas 3D texturizadas, ainda necessita de treinamento do NeRF e converter em *mesh* para cada cenário desejado.

3.2 REPRESENTAÇÕES BASEADAS EM MAPA DE PROFUNDIDADE

Através de um mapa de profundidade conseguimos construir representações geométricas que conseguem renderizar em tempo real na maioria dos *hardwares* atuais, como malhas 3D, MPI, LDI e nuvens de pontos. Assim, essas representações podem ser utilizadas para desenvolver abordagens com um custo computacional inferior ao dos voltados para renderização neural. Uma forma popular de realizar NVS em tempo real é o uso de múltiplas camadas com imagens coloridas semi-transparentes. Assim, inicialmente descreveremos as abordagens que utilizam arquiteturas CNN para inferir o conteúdo dos planos de um MPI Szeliski & Golland (1998). Como algumas arquiteturas baseadas em MPI inferem a profundidade através dos planos ou utilizam um mapa de disparidade para guiar essa inferência, também serão apresentadas abordagens que utilizam nuvens de pontos e LDIs Shade *et al.* (1998). Nesses dois casos, os trabalhos apresentados utilizam a estrutura do mapa de profundidade para inferir o conteúdo das regiões onde existem oclusões ao invés de preencher o conteúdo de vários planos. Logo essas técnicas conseguem utilizar arquiteturas voltadas para tarefas específicas (como inferência de profundidade e *inpainting*) para construir LDIs e nuvens de pontos utilizando uma imagem de entrada.

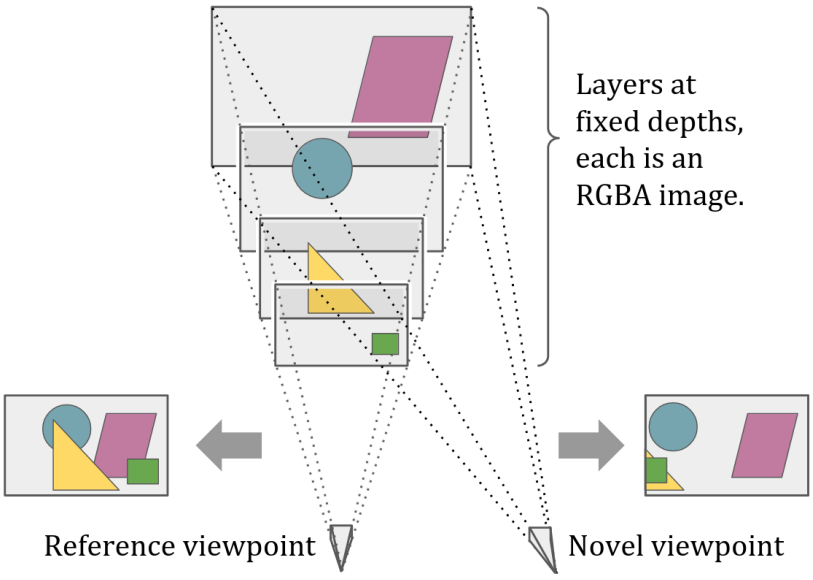


Figura 15: Fonte: Zhou *et al.* (2018b)

3.2.1 *Multiplane Images*

Introduzido em Szeliski & Golland (1998) com o foco em recuperar disparidade (profundidade), cor e transparência a partir de imagens estéreo, representações *multiplane images*

MPI são criadas com intervalos fixos de profundidade e todos os objetos contidos na cena são distribuídos entre os planos mais próximos da sua profundidade estimada. O uso de CNNs para gerar MPIs foi apresentado inicialmente em Zhou *et al.* (2018b), onde o MPI é gerado de um par de imagens estéreo e renderizado através da deformação dos planos devido a projeção dada uma matriz de câmera.

Para melhorar a qualidade da modelagem do MPI, Srinivasan *et al.* (2019) utiliza duas redes CNN com convolução 3D. Uma dessas redes é utilizada para remover o conteúdo que oclui os objetos presentes em cada camada e outra para inferir opacidade das camadas. Focando na construção de MPI utilizando apenas uma imagem e uma arquitetura para inferência, Tucker & Snavely (2020) utiliza nuvens de pontos esparsos extraídas de vídeos de uma cena (pelo uso de SFM) como regularizador da composição da profundidade das camadas.

Utilizando mapas de profundidade para simplificar a construção de MPIs, Luvizon *et al.* (2021); Han *et al.* (2022) propõe esquemas para adaptar o número de camadas de acordo com a complexidade do mapa de profundidade. Em Luvizon *et al.* (2021) é proposto um método heurístico baseado no resultado do filtro de Canny aplicado ao mapa de profundidade para definir o numero de planos. Já em Han *et al.* (2022) é definida uma rede com auto-atenção (*self-attention*) para ajustar os planos de um MPI predefinida baseada no mapa de profundidade e outra para inferir o conteúdo de cor e densidade.

MPIs também são combinadas com NeRF para renderizar imagens com alta qualidade, como é mostrado nos trabalhos Li *et al.* (2021); Wizadwongsa *et al.* (2021); Pu *et al.* (2023). Em todos eles, as redes são treinadas renderizando através da adaptação da fórmula de renderização volumétrica do NeRF para MPI. Porém, em Li *et al.* (2021) é treinada uma arquitetura *enconder* e *decoder* que busca generalizar a construção de um MPI para apenas uma imagem, onde o *decoder* infere o conteúdo de cor e de densidade para cada plano. Já em Wizadwongsa *et al.* (2021) uma MLP é utilizada para inferir coeficientes de harmônicos esféricos dada coordenada do MPI e outra MLP para inferir o conteúdo RGB da imagem renderizada. Apesar de conseguir modelar efeitos dependentes da posição de câmera (e.g. reflexão, espelhamento), é necessário treinar o modelo para cada cena desejada. Mais recentemente, Pu *et al.* (2023), também adapta a fórmula de renderização volumétrica para treinar duas redes de *inpainting*, uma para a profundidade e outra para cor, gerando MPI através do mapa de profundidade.

Apesar do sucesso dessas abordagens, MPIs costumam possuir informações redundantes e necessitar de uma grande quantidade de camadas para conseguir bons resultados. Baseado na observação que a complexidade local do mapa de profundidade é menor que na imagem inteira, em Khan *et al.* (2023b) é treinada uma rede que infere um mapa de confiança e de profundidade das subdivisões da imagem de entrada, cada uma dessas saídas usando diferentes decodificadores. O mapa de confiança de cada pedaço é utilizado para gerar máscaras através de *Weighted K-Means Clustering*, que definirão as regiões que estão presentes os objetos em cada camada. Assim, são separadas as profundidades de cada máscara e utilizadas como entrada de uma rede que irá gerar um MPI para cada pedaço. Com isso, o custo computacional frente às

abordagens anteriores é reduzido.

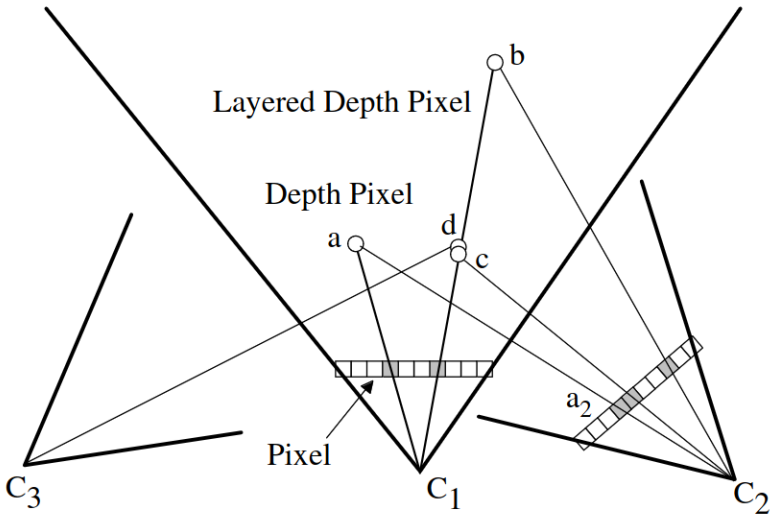


Figura 16: Fonte: Shade *et al.* (1998)

3.2.2 *Layered Depth Images* e nuvens de pontos

Diferente das MPIs, as LDIs utilizam profundidade variáveis em cada camada. Definido em Shade *et al.* (1998), as imagens de profundidade em camadas contém as informações da câmera (e.g. posição, direção, dimensões do plano projetivo) mais uma imagem $H \times W$ onde cada *pixel* contém as informações de cor e um *array* de tamanho L, contendo quais valores de profundidade são válidos (i.e. onde possuem pontos do objeto). Com isso, não é necessário o uso de *z-buffer* na renderização e ordenar as profundidades, renderizando da camada mais profunda para a mais próxima. Diversos trabalhos atuais Shih *et al.* (2020); Jampani *et al.* (2021); Kopf *et al.* (2020), empenharam-se em desenvolver técnicas de geração de LDI para imagem única de entrada utilizando aprendizagem profunda. Em resumo, essas abordagens consistem em treinar separadamente arquiteturas para estimação de profundidade geral e para preenchimento de cor e profundidade das regiões com oclusão. Inicialmente, por exemplo em Tatarchenko *et al.* (2015); Cun *et al.* (2018), a estratégia para gerar representações utilizando mapas de profundidade utiliza parâmetros da câmera de uma nova posição para sintetizar imagens, levando a uma renderização lenta. Pensando em acelerar essa renderização, Niklaus *et al.* (2019) propõe um *pipeline* para construir nuvens de pontos de maneira iterativa, preenchendo a informação de cor e profundidade dadas diferentes posições de câmera.

Na Figura 17 vemos a estrutura geral da abordagem proposta em Niklaus *et al.* (2019). Inicialmente é gerado o mapa de profundidade através de uma série de duas redes propostas: uma para a estimação e outra para refinamento do mapa de profundidade. A rede que estima a profundidade é treinada utilizando duas redes de segmentação pré-treinadas para alcançar resultados geometricamente acurados. A rede de refinamento é treinada utilizando os contornos

da imagem de entrada para garantir bordas mais acuradas enquanto redimensiona o mapa de profundidade para as dimensões originais.

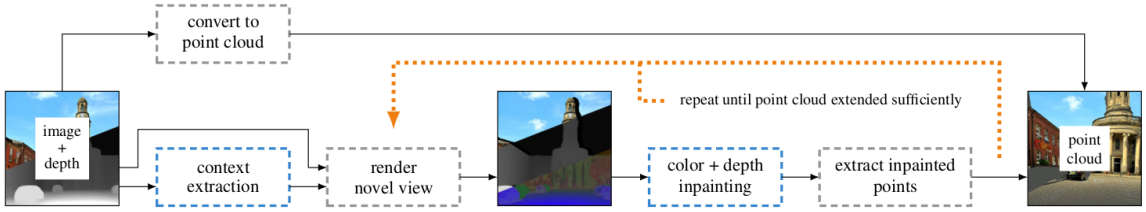


Figura 17: Fonte: Niklaus *et al.* (2019)

Após isso, o mapa de profundidade é utilizado em conjunto com a imagem de entrada para extrair características de contexto (no bloco *context extraction* na Figura 17) através de uma pequena rede que é treinada junto com as arquiteturas de preenchimento. Após essa extração, são renderizadas imagens a partir de várias posições de câmera e o conteúdo faltante é preenchido através de uma rede de preenchimento de cor e profundidade. A entrada e as regiões preenchidas são então convertidas em nuvens de pontos e esse processo é repetido para cada posição de câmera até que a nuvem de pontos esteja suficientemente expandida nas regiões ocluídas.

As nuvens de pontos possuem um custo adicional por necessitarem de funções que combinem o conteúdo presente nos pontos vizinhos ou converter em outras representações como *meshes* para sintetizar imagens realistas. Pensando nisso, dada uma imagem de profundidade em Shih *et al.* (2020) são processados os contornos dos objetos e separados em camadas para construir uma representação de LDI através de componentes conectados. Como todos os pontos vizinhos são inicialmente conectados (apenas uma camada), quando contidos nos contornos são separados formando regiões de síntese e contexto para cada camada. Após dilatar por um número de iterações, são definidas as máscaras que serão utilizadas como *background* (contexto) para preenchimento do conteúdo de cor e profundidade das regiões ocluídas (síntese).

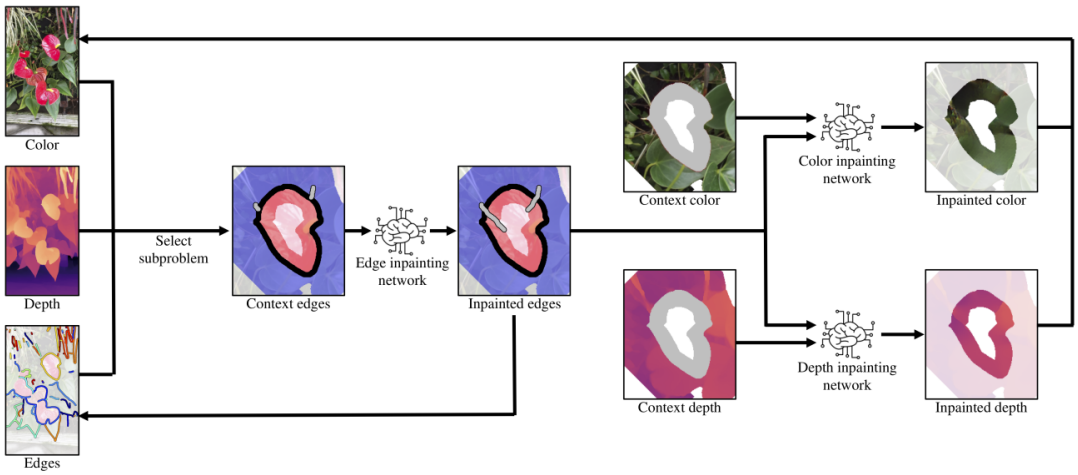


Figura 18: Fonte: Shih *et al.* (2020)

Como é mostrado na Figura 18, são utilizadas 3 arquiteturas de *inpainting* em cada

camada do LDI. A primeira arquitetura é treinada para preencher o conteúdo de cor entre os contornos extraídos da imagem RGB de entrada. A segunda infere o conteúdo de cor nos *pixels* presentes na máscara de contexto utilizando como base as cores presentes na máscara de síntese. A terceira é utilizada para predizer a profundidade de *background* nas regiões da máscara de síntese. Tanto essa abordagem quanto Niklaus *et al.* (2019) possuem um alto custo computacional por precisarem iterativamente predizer o conteúdo das regiões ocluídas, levando minutos para gerar um modelo 3D utilizando uma RTX 2080.

Para contornar esse custo, Kopf *et al.* (2020) define uma estratégia de implementação que altera a estrutura convencional de redes CNNs, propondo uma arquitetura de *inpainting* específica para estruturas LDI. Essa arquitetura utiliza como entrada um *array* com pontos, os vizinhos nos quais estão conectados. Esses pontos possuem rótulos que fazem parte das regiões ocluídas (i.e. tem informação preenchida pela arquitetura) ou de objetos em primeiro plano (i.e mantém a informação de cor de entrada). Assim, as camadas convolucionais são aplicadas nesses pontos, inferindo sobre as camadas do LDI. Outra contribuição presente em Kopf *et al.* (2020) é conversão do LDI em malhas 3D, conectando todos os pontos dos contornos e amostrando pontos internos do objeto para formar polígonos. Necessitando um trabalho de modificação dos próprios *frameworks* existentes para desenvolvimento de redes neurais (e.g. *TensorFlow*), torna difícil a replicabilidade desse método. Talvez, por esse motivo, não foram encontrados trabalhos que o utilizaram como referência de qualidade a ser comparada.

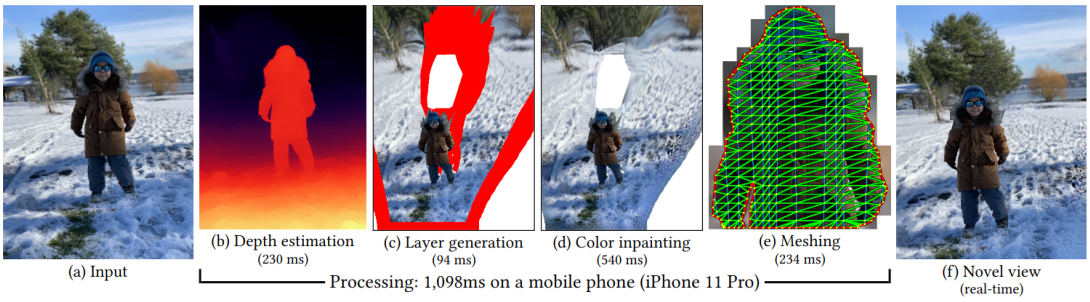


Figura 19: Fonte: Kopf *et al.* (2020)

Mais recentemente Jampani *et al.* (2021) simplifica o problema de gerar LDIs utilizando apenas uma imagem ao propor o método SLIDE, que utiliza apenas duas camadas, uma de *foreground* (regiões ocludentes) e outra de *background* (regiões ocluídas). Nessa abordagem, o mapa de profundidade é utilizado para calcular a visibilidade do *foreground* (chamada de *soft foreground visibility* (SFV)), a máscara de desoclusão (*soft disocclusions* (SD)) e oclusão (*soft occlusions*(SO)). As duas últimas são utilizadas numa rede de *inpainting* para preenchimento de cor e profundidade, assim como Shih *et al.* (2020), porém fazendo apenas uma inferência em toda a imagem. No final o LDI é convertido em *mesh*, utilizando a saída RGBD do *inpainting* como *background* e o RGBD de entrada como *foreground*. Para que os triângulos com vértices que possuem uma maior diferença de profundidade não sejam esticados e cubram a região do *background*, o SFV é utilizado como componente de transparência dos vértices.

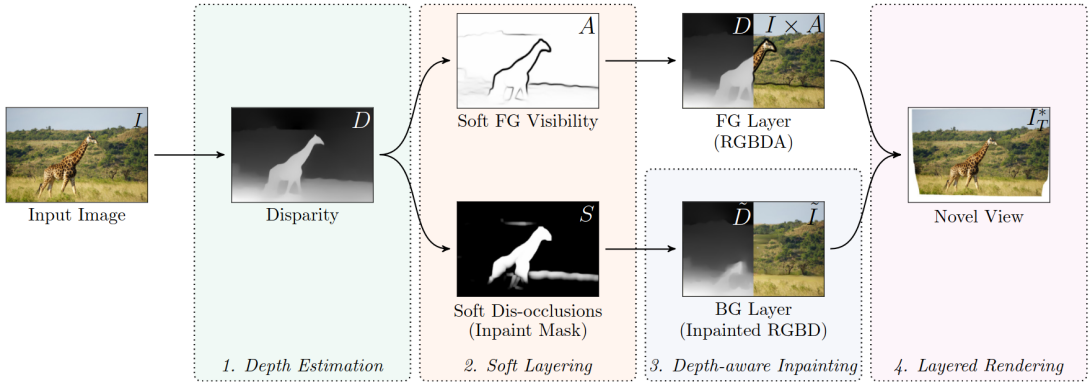


Figura 20: Fonte: Jampani *et al.* (2021)

4

MÉTODO PROPOSTO

Sabendo as técnicas no estado da arte, podemos concluir que vários desses métodos possuem uma alta complexidade computacional, principalmente NeRFs. Apesar dos avanços em *hardwares* para dispositivos móveis, como o uso de TPUs e *chips* de processamento gráficos, renderizar uma fotografia 3D ainda pode ser uma tarefa com a necessidade de um certo esforço de otimização (e.g. escovando *bits*). Dos diferentes tipos de estruturas de dados e métodos de renderização, sabemos que os mais eficientes em tempo e são comumente utilizados para renderizar em tempo real são os MPI, LDI e *meshes* 3D. Pelo fato das melhores abordagens utilizando MPI Khan *et al.* (2023b); Pu *et al.* (2023) precisarem de uma arquitetura específica para inferir o conteúdo nas regiões ocluídas em cada camada de profundidade da imagem, nos baseamos na abordagem descrita em Jampani *et al.* (2021). Diferente de Shih *et al.* (2020), em Jampani *et al.* (2021) o conteúdo de cor e de profundidade são inferidos através de apenas uma inferência na arquitetura de *inpainting*, tornando-se menos custoso. Para construção do modelo 3D da cena em um *mesh*, todos os *pixels* da imagem de entrada são considerados vértices e conectados com seus vizinhos na formação dos triângulos (da mesma forma que em Figura 8d). Por isso, esse método acaba consumindo uma quantidade de memória desnecessária por cobrir regiões com baixa variação de profundidade com um alto número de polígonos. Além disso, por fazer o preenchimento da profundidade utilizando o contexto geral da imagem, certas regiões da superfície de *background* podem acabar ficando à frente da superfície de *foreground*, dependendo da distribuição de profundidade no entorno das regiões.

Pensando nisso, desenvolvemos uma abordagem que, após definidas as arquiteturas para mapas de profundidade e *inpainting*, constrói malhas 3D utilizando *quadrees* em diferentes regiões da imagem de entrada. As arquiteturas utilizadas serão descritas nas seções 4.1 e 4.2. Outra contribuição presente neste trabalho é a melhoria do mapa de profundidade através de uma abordagem heurística utilizando segmentação (i.e. deixando mais nítidos e com profundidade mais uniforme nos objetos delimitados no mapa de segmentação). Depois disso, descrevemos as etapas presentes no método para construção de malhas 3D, o *Depth-based Quadtree Mesh* (DQ-Mesh). O *pipeline* completo para a construção do modelo 3D da abordagem proposta aparece na Figura 21. Por último, semelhante a Jampani *et al.* (2021) utilizaremos *soft foreground pixel visibility* para computar o nível de transparência dos *pixels* da textura utilizada como *foreground*

e renderizar utilizando rasterização com *z-buffer*. A abordagem completa é descrita na Figura 21.

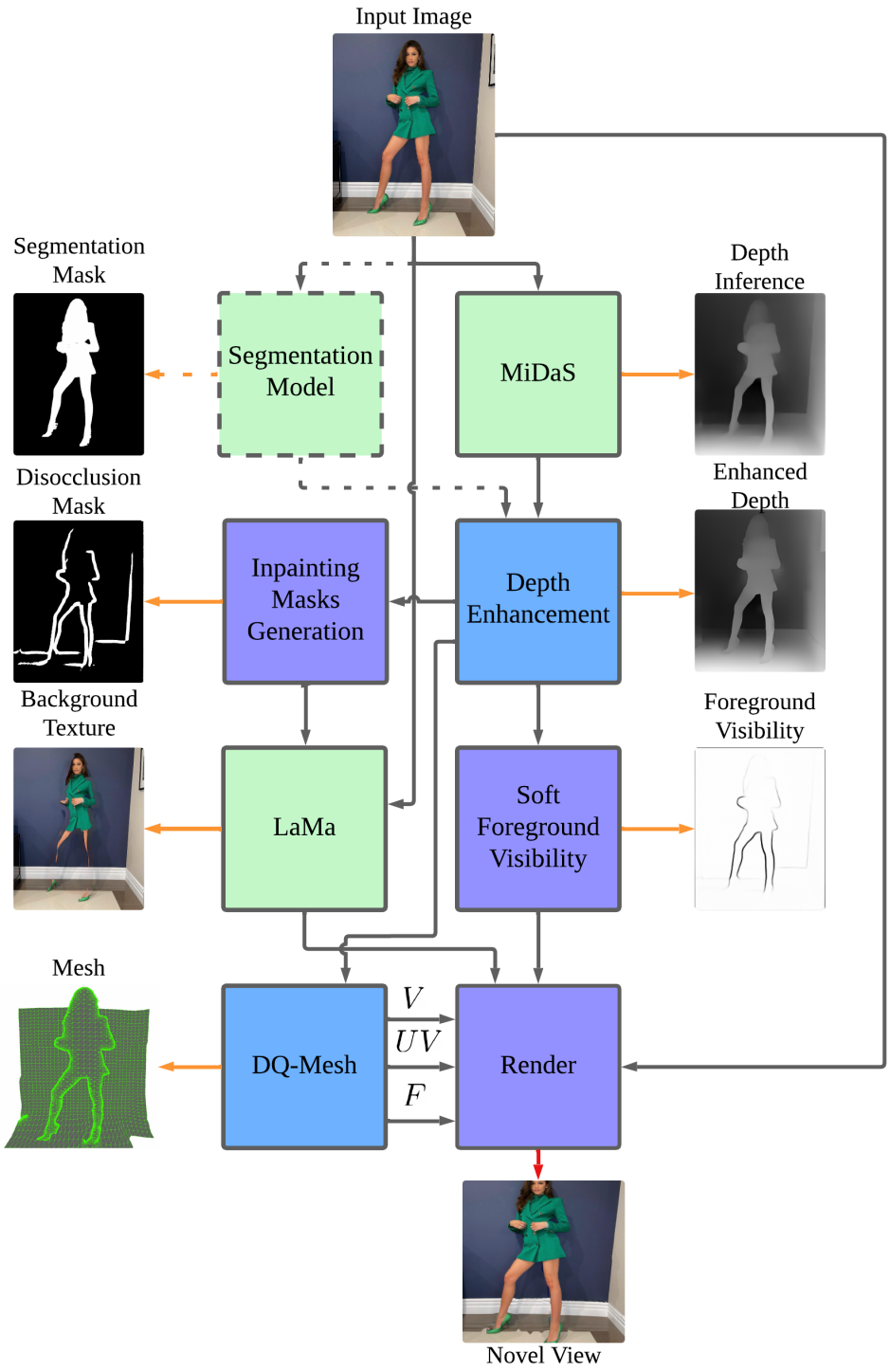


Figura 21: Diagrama do fluxo do método para a utilização do DQ-Mesh. Os quadrados verdes representam os modelos pré-treinados utilizados. Em azul estão blocos onde estão presentes os algoritmos propostos. Em roxo é o renderizador para gerar imagens a partir de uma posição de câmera. Imagens de exemplo da saída de cada componente do *pipeline* são apontadas pelas setas laranja.

4.1 ESTIMAÇÃO DE PROFUNDIDADE

A estimação de profundidade consiste na tarefa de inferir a profundidade de uma cena utilizando imagens de entrada. No caso do uso de apenas uma imagem, chama-se o problema de estimação monocular de profundidade. Vários trabalhos atuais utilizam estimação monocular para reconstrução 3D Niklaus *et al.* (2019); Shih *et al.* (2020); Han *et al.* (2022). As profundidades em cada *pixel* podem ser computadas através do uso de câmeras *duais*, presentes em diversos dispositivos móveis atuais, ou estimados através de uma imagem RGB. Como é desafiador deduzir informação de profundidade de um *pixel* utilizando apenas uma imagem, os atuais progressos nesse problema tem sido endereçados a abordagens de aprendizagem profunda. Sendo assim, optamos por utilizar o estado da arte dos métodos de estimação de profundidade, nesse caso a família de arquiteturas MiDaS Ranftl *et al.* (2021, 2022).

4.1.1 MiDaS v3.0

Em Ranftl *et al.* (2021) é apresentada a arquitetura mostrada na Figura 22, que utiliza *Transformer* Dosovitskiy *et al.* (2021) nas camadas de codificação (*encoder*) e camadas convolucionais na decodificação (*decoder*). As camadas *Transformers* utilizam um conjunto de *tokens* como entrada de blocos *Multi-Headed Self-Attention* (MHSA) Vaswani *et al.* (2017), que computa a correlação entre os *tokens* transformando a representação do conjunto. Inspirados na arquitetura Dosovitskiy *et al.* (2021), na arquitetura proposta para MiDaS v3.0 as imagens de entrada são divididas em pedaços não sobrepostos. Esses pedaços são redimensionados como vetores unidimensionais e convertidos em *tokens* utilizando uma projeção linear. Esse processo resulta num conjunto de *tokens* $\mathbf{t}^0 = \bigcup_{i=0}^{N_p} \{t_i^0\}$ onde $t_i^0 \in \mathbb{R}^{T_d}$ são os *tokens*, $N_p = \frac{HW}{p^2}$ é a quantidade de *tokens* para uma imagem com dimensões $H \times W$ dividido em blocos de tamanho p^2 e T_d é a dimensão de cada *token*.

Os *tokens* $\mathbf{t}^0$ são utilizados como entrada de uma sequência de L camadas *transformer*, convertendo novas representações $\mathbf{t}^l$ onde l refere-se à saída da camada l. Em Ranftl *et al.* (2021) são apresentadas 3 variantes da arquitetura proposta, porém iremos focar na variação ViT-Large, utilizada nos experimentos desta dissertação. Nessa variação temos que $L = 24$ e $T_d = 1024$.

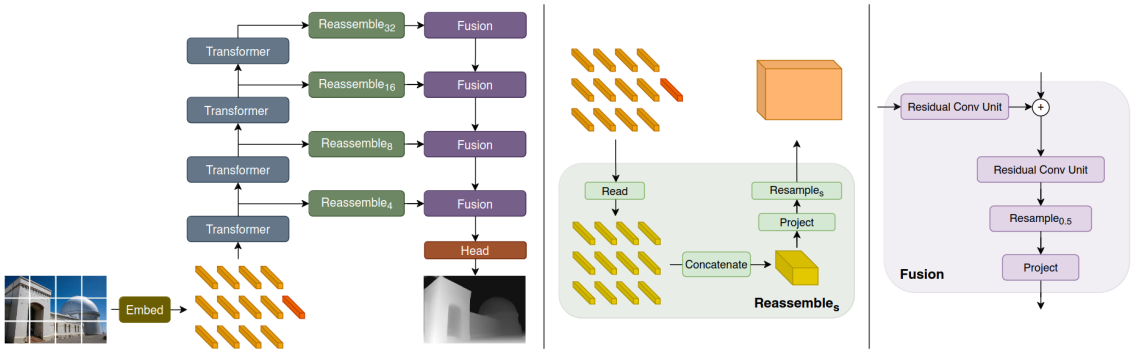


Figura 22: Fonte: Ranftl *et al.* (2021)

Na etapa de decodificação o conjunto de *tokens* de cada camada é reconstruído como imagens de características com dimensões $\frac{H}{p} \times \frac{W}{p} \times T_d$. Com isso, essas representações são combinadas com as saídas das camadas convolucionais (com exceção da camada com menor resolução) através de uma camada residual. A reconstrução é mostrada pelo bloco nomeado como *Reassemble* e combinação com camadas residuais no bloco *Fusion* da Figura 22.

4.1.2 MiDaS *mobile*

Como a arquitetura DPT-Large possui cerca de 343 milhões de parâmetros Ranftl *et al.* (2021) e é implementada utilizando camadas convolucionais e *transformer*, a inferência torna-se custosa para dispositivos com GPUs com menor capacidade computacional. Sendo assim, a arquitetura MiDaS *mobile* pré-treinada é utilizada para inferir o mapa de profundidade em dispositivos móveis. O modelo MiDaS *mobile* é uma versão da arquitetura proposta em Ranftl *et al.* (2022) porém ao invés de utilizar a ResNet50 He *et al.* (2016) como *decoder*, é utilizada uma arquitetura da família EfficientNet (mais especificamente EfficientNet-B0 Tan & Le (2019)). Com essa substituição a inferência do modelo fica 12.8 vezes mais rápida,[1] além de conter cerca de 24 milhões de parâmetros. Em Figura 23 vemos a arquitetura MiDaS em sua versão original inspirada em Xian *et al.* (2018), utilizando ResNet50 no *decoder* (cubos azuis). No caso da arquitetura MiDaS *mobile* esses blocos são substituídos por EfficientNet-B0.

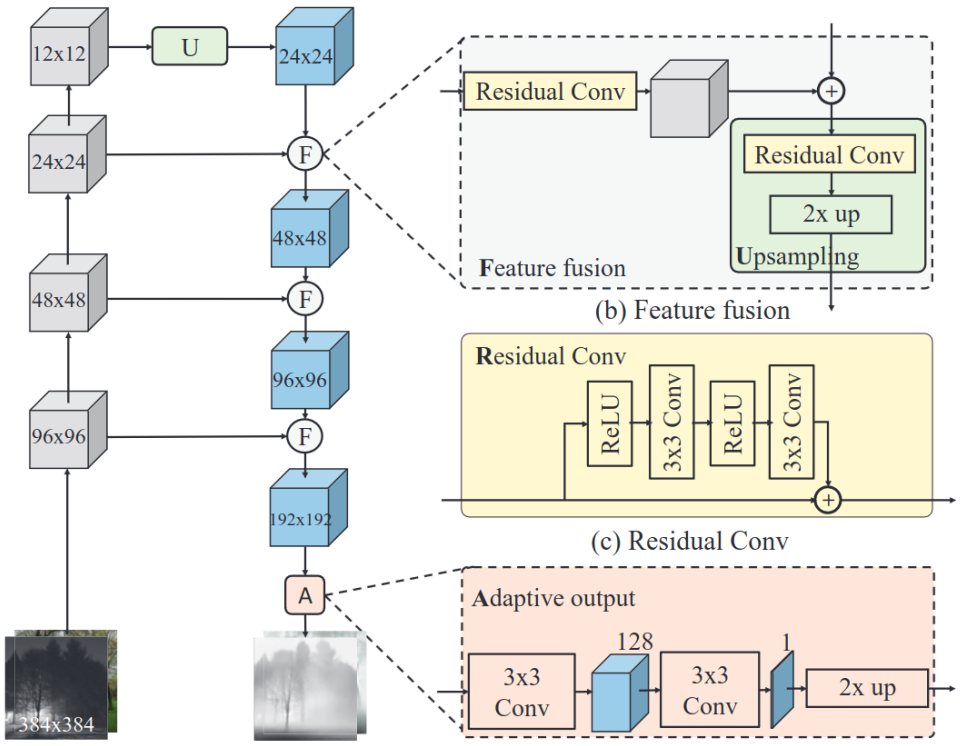


Figura 23: Fonte: Xian *et al.* (2018)

[1] Esses dados estão descritos em `https://github.com/isl-org/MiDaS/tree/master/mobile`

4.2 ARQUITETURA PARA *INPAINTING*

A pintura interna de imagens (ou *image inpainting*) é o problema que consiste em prover uma restauração plausível para regiões faltantes em uma imagem. Métodos heurísticos de *image inpainting* consistem em propagar a informação da fronteira das regiões faltantes Bertalmio *et al.* (2000); Ballester *et al.* (2001) ou, para áreas menores, preencher texturas ou objetos completos se tiver objeto similar disponível em outra região da imagem Efros & Leung (1999); Huang *et al.* (2014). Porém esses métodos são limitados em gerar conteúdos semanticamente plausíveis, especificamente para imagens com grandes regiões faltantes.

Para o *pipeline* de fotografia 3D implementado, foi escolhida a arquitetura LaMa (citada como modelo base em Suvorov *et al.* (2022)), que dos métodos no estado da arte como Zheng *et al.* (2022); Zhao *et al.* (2021) consegue resultados competitivos utilizando uma menor quantidade de parâmetros, facilitando a adaptação para dispositivos *mobile*. Na Figura 24, temos uma visão geral da arquitetura. Inicialmente temos como entrada uma imagem $I_m \in \mathbb{R}^{H \times W \times C}$, onde $C = 4$ é o número de canais, sendo os 3 primeiros de cor (e.g. RGB) e o último uma máscara binária. Inicialmente, a arquitetura tem 3 blocos de *downscale*, onde as dimensões H e W são reduzidas. A saída dessas camadas será a entrada de 9 blocos residuais *Fast Fourier Convolution* (FFC) concatenados (a saída de um é a entrada do outro, com exceção do primeiro e do último). A imagem final da arquitetura é gerada após a saída do último bloco FFC passar por 3 blocos de *upscale* que retorna a imagem para para as dimensões de entrada.

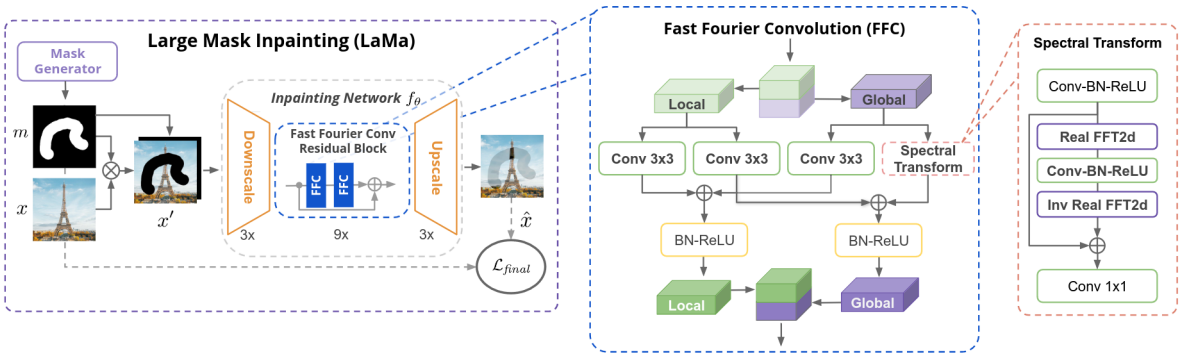


Figura 24: Fonte: Suvorov *et al.* (2022)

4.2.1 *Fast Fourier Convolution*

A contribuição mais relevante de Suvorov *et al.* (2022) é a utilização de camadas FFC, proposta por Chi *et al.* (2020) que permite o uso do contexto global das camadas anteriores. Isso é alcançado através do uso da *Fast Fourier Transform* (FFT) na dimensão dos canais, obtendo campo receptivo (*receptive field*) que cobre a imagem inteira. Numa camada FFC (quadriculado azul em 24) os canais dividem-se em duas partições: uma onde será processada a informação local e outra global. Na partição de processamento de características locais são utilizadas camadas convencionais de convolução. Na outra partição é utilizada a FFT *Real* para

processar o contexto global. A FFT *Real*, por ser aplicada, assim como sua inversa, apenas a valores reais, utiliza apenas metade do espectro comparada a FFT convencional.

No Algoritmo 3 vemos os passos que são aplicados nas camadas FFC, também definida como *Spectral Transform* na Figura 24. Primeiramente, é aplicado a transformada real da FFT denotada pela função $RealFFT2d : \mathbb{R}^{H\times W\times C} \rightarrow \mathbb{C}^{H\times \frac{W}{2}\times C}$ e a parte real e a imaginaria são concatenadas utilizando a função $ComplexToReal : \mathbb{C}^{H\times \frac{W}{2}\times C} \rightarrow \mathbb{R}^{H\times \frac{W}{2}\times 2C}$. Depois, é aplicado um bloco convolucional no domínio da frequência $ReLU \circ BN \circ Conv1X1 : \mathbb{R}^{H\times \frac{W}{2}\times 2C} \rightarrow \mathbb{R}^{H\times \frac{W}{2}\times 2C}$. Para retornar ao domínio espacial, combinaremos as camadas para retornar ao espaço complexo utilizando $ComplexToReal : \mathbb{R}^{H\times \frac{W}{2}\times 2C} \rightarrow \mathbb{C}^{H\times \frac{W}{2}\times C}$ e então a transformada inversa é aplicada utilizando $InverseRealFFT2d : \mathbb{C}^{H\times \frac{W}{2}\times C} \rightarrow \mathbb{R}^{H\times W\times C}$. No final as partições locais e globais são concatenadas novamente, formando a saída do bloco FFC.

Algoritmo 3: FFC

1 **Entrada:** $I_{in} \in \mathbb{R}^{H\times W\times C}$
2 **Saída:** $I_{out} \in \mathbb{R}^{H\times W\times C}$
3 $I_{fft} \leftarrow RealFFT2d(I_{in})$
4 $I_{real} \leftarrow ComplexToReal(I_{fft})$
5 $I'_{real} \leftarrow ReLU \circ BN \circ Conv1\times 1(I_{real})$
6 $I_{complex} \leftarrow RealToComplex(I'_{real})$
7 $I_{out} \leftarrow InverseRealFFT2d(I_{complex})$

4.2.2 LaMa com convoluções separáveis

A arquitetura LaMa é apresentada em Suvorov *et al.* (2022) como eficiente em relação a outras arquiteturas de *inpainting* à época, em relação à quantidade de parâmetros. Nesse caso, a arquitetura necessita de uma quantidade menor de parâmetros para alcançar resultados semelhantes ou melhores que as abordagens comparadas em Suvorov *et al.* (2022). Sendo assim, como são utilizadas camadas convolucionais clássicas nas arquiteturas LaMa, neste trabalho a arquitetura foi reconstruída utilizando operadores de convolução separável Chollet (2017) para diminuir o tempo de inferência. Além disso, para a utilização de operadores FFC melhor otimizados, esses operadores foram implementados utilizando a linguagem Halide Ragan-Kelley *et al.* (2013). Essa nova versão será chamada ao longo deste trabalho de LaMaSep, possuindo 2,4 milhões de parâmetros. Após essa reconstrução da arquitetura, treinamos o modelo utilizando o banco de dados Zhou *et al.* (2018a) com geração de máscaras aleatórias.

4.3 GERAÇÃO DE MALHAS 3D COM DQ-MESH

Após a inferência do mapa de profundidade $I_D \in \mathbb{R}^{H\times W}$, precisamos de um algoritmo para gerar a estrutura 3D e renderizar as imagens dada uma nova posição de câmera. Para isso, foi desenvolvido o algoritmo DQ-Mesh, onde dado um mapa de profundidade são computados

os vértices V, as faces F e as coordenadas do mapeamento de textura UV da malha 3D. A estrutura geral do algoritmo pode ser vista na Figura 25 e é descrita em pseudo-código no Algoritmo 4. Primeiramente, são aplicadas melhorias no mapa de profundidade para que os objetos presentes na imagem tenham uma menor variação de profundidade, para que não gere artefatos nos contornos do objeto nas imagens renderizadas. Assim, foram desenvolvidos dois pré-processamentos, um mais simples (detalhado na seção 4.3.1) e outro mais robusto, utilizando máscaras de segmentação (descrito na seção 4.3.2).

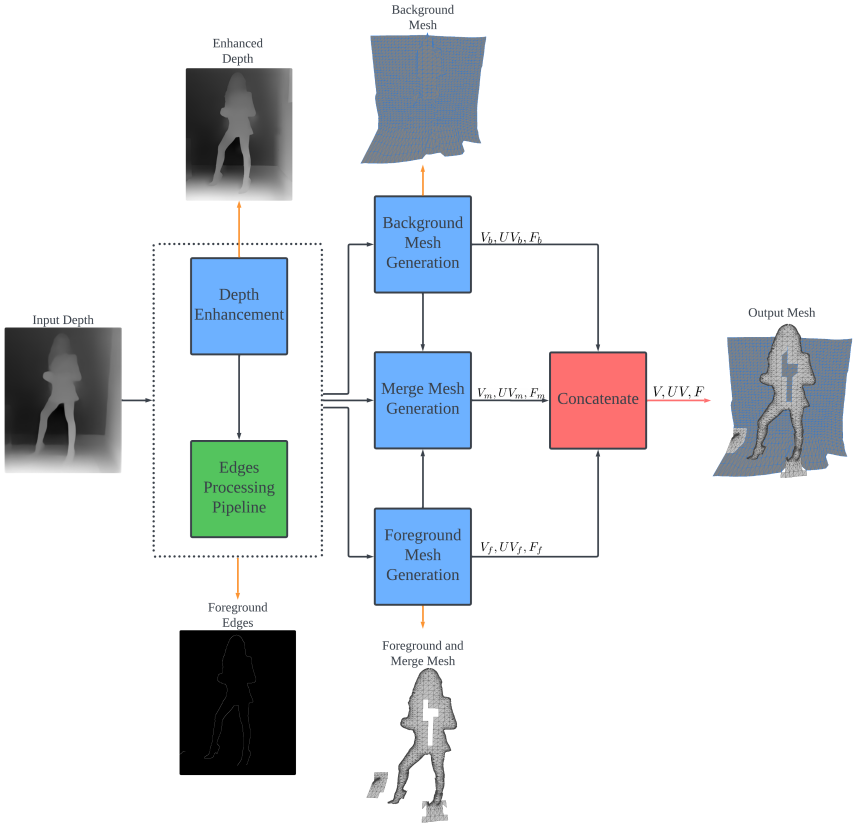


Figura 25: Diagrama do fluxo do método para a utilização do DQ-Mesh. Em azul estão blocos onde estão presentes os algoritmos desenvolvidos que possuem contribuições. Em verde está o pipeline para extração das imagens de borda utilizadas. Em vermelho está a camada onde serão concatenados os vértices, as faces e as coordenadas de textura das regiões de *foreground*, *background* e de *merge*. Imagens de exemplo da saída de cada componente do algoritmo são apontadas pelas setas laranja.

Após as melhorias do mapa de profundidade são computadas as bordas dos objetos na cena com algoritmos de detecção de borda (e.g. filtro de Canny Canny (1986)). A imagem resultado do filtro de borda será considerada como o contorno de *foreground* $I_{FE} \in \mathbb{R}^{H \times W}$ (i.e. os contornos das regiões dos objetos que estão na frente e aparecem na imagem de entrada). Assim, é necessário que também seja calculado o contorno das regiões de *background* $I_{BE} \in \mathbb{R}^{H \times W}$ (i.e. ocluídas pelos objetos com contorno I_{FE}).

Inspirando-se em Fridovich-Keil and Yu *et al.* (2022); Zhang *et al.* (2023b); Khan *et al.* (2023b); Shih *et al.* (2020); Kopf *et al.* (2020) as imagens I_D, I_{FE} e I_{BE} são divididas em blocos

Algoritmo 4: `DQ-Mesh`; Visão geral do *pipeline* para gerar a malha 3D que será utilizada para NVS.

Entrada: $I_D \in \mathbb{R}^{H \times W}$, $B_s \in \mathbb{N}$
Saída: V, UV, F, I_M
$I_{FE}, I_{BE} \leftarrow$ `process_edges`(I_D)
$I'_D \leftarrow$ `depth_simples_enhancement`(I_D, I_{FE})
$V_b, UV_b, F_b, I_{BM}, I_{Bl}, I_{BIdx} \leftarrow$ `back_mesh_generator`$(I'_D, I_{FE}, I_{BE}, B_s)$
$N_{BIdx} \leftarrow \|V_b\|$
$V_f, UV_f, F_f, I_{FM}, I_{Fl}, I_{FIdx} \leftarrow$ `fore_mesh_generator`$(N_{BIdx}, I'_D, I_{FE}, I_{BE}, B_s)$
$N_{FIdx} \leftarrow N_{BIdx} + \|V_f\|$
$V_m, UV_m, F_m \leftarrow$ `merge_mesh_generator`$(N_{FIdx}, I'_D, I_{FIdx}, I_{BIdx}, I_{Bl}, I_{BM}, I_{FM} B_s)$
$I_M \leftarrow I_{BM} + I_{FM}$
$V \leftarrow$ `concatenate`$([V_b, V_f, V_m])$
$UV \leftarrow$ `concatenate`$([UV_b, UV_f, UV_m])$
$F \leftarrow$ `concatenate`$([F_b, F_f, F_m])$
return V, UV, F, I_M

$B_s \times B_s$, onde B_s é o tamanho das dimensões do bloco. Em cada um dos blocos, contendo a profundidade e as bordas de diferentes regiões das imagens, serão computados V, F e UV em quatro etapas. A primeira etapa, descrita na seção 4.3.4, consiste em computar os vértices V_b, as faces F_b e as coordenadas do mapeamento de textura UV_b das regiões de *background*. Na segunda etapa, descrita na seção 4.3.5, são computados os vértices V_f, as faces F_f e as coordenadas do mapeamento de textura UV_f das regiões de *foreground*. Na terceira etapa, descrita na seção 4.3.6, serão calculados os vértices V_m, as faces F_m e as coordenadas do mapeamento de textura UV_m das regiões intermediárias (i.e. entre as regiões de *foreground* e *background*). Na última etapa serão concatenados os vértices, coordenadas de textura e faces das 3 últimas etapas para obter V, UV e F;

4.3.1 Pré-processamento simples do mapa de profundidade

Os mapas de profundidade inferidos de arquiteturas de CNN podem ser borrados nos contornos dos objetos (Figura 26a). Como o algoritmo DQ-Mesh utiliza a profundidade nos contornos dos objetos para gerar uma malha 3D, precisamos de algoritmos que diminuam a variação de profundidade nessas regiões. Assim, implementamos uma abordagem (Algoritmo 5) com algoritmos de dilatação e erosão Gonzalez & Woods (2008) buscando diminuir essa variação. Inicialmente, é armazenado em $I_{dilated}$ o mapa de profundidade dilatado por um algoritmo de dilatação `dilate` por duas iterações. Após isso, em I_{comb}, que inicialmente é uma cópia de I_D, são adicionados os *pixels* computados em $I_{dilated}$ nas posições (i, j) onde $I_{Canny}(i, j) == 1$. Os *pixels* entre I_{com} e I_D que possuem valor inferior, serão preenchidos aplicando mais uma vez a função `dilate` e armazenando em $I'_{dilated}$. Para que o contorno I_{Canny} fique alinhado com a profundidade, aplicamos uma erosão (i.e. `erode`) e o mapa de profundidade I'_D é retornado. Podemos ver um exemplo da execução do algoritmo na Figura 26.

Algoritmo 5: `depth_simple_enhancement`

Entrada: $I_D \in \mathbb{R}^{H \times W}$, $I_{Canny} \in \mathbb{R}^{H \times W}$
Saída: $I'_D \in \mathbb{R}^{H \times W}$
$I_{dilated} \leftarrow \texttt{dilate}(I_D, it \leftarrow 2)$
$I_{comb} \leftarrow I_D$
Para os valores de i e j onde $I_{Canny}(i, j) == 1$, temos que $I_{comb}(i, j) \leftarrow I_{dilated}$
$I'_{dilated} \leftarrow \texttt{dilate}(I_{comb}, it \leftarrow 1)$
$I'_D \leftarrow \texttt{erode}(I'_{dilated}, it \leftarrow 1)$
return I'_D



(a) $I_D \in \mathbb{R}^{16 \times 16}$ (b) $I_{Canny} \in \mathbb{R}^{16 \times 16}$ (c) $I_D + I_{Canny}$ (d) I'_D

Figura 26: Imagens de entrada e saída de um exemplo da execução do Algoritmo 5. Na Figura 26a vemos uma entrada I_D com dimensões $H = 16$ e $W = 16$. Em 26b vemos a imagem com os contornos I_{Canny} de entrada. Em 26c vemos uma sobreposição das entradas I_D e I_{Canny} que ao final do algoritmo resulta na imagem I'_D presente em 26d.

4.3.2 Pré-processamento do mapa de profundidade utilizando máscaras de segmentação

Recentemente, Kim *et al.* (2022) combinam a utilização de máscaras de segmentação com mapas de disparidade para obter resultados mais precisos nos contornos dos objetos. Porém, a estratégia consiste em treinar simultaneamente arquiteturas de segmentação e de estimação de profundidade. O nosso refinamento consiste em uma abordagem heurística de melhoria de um mapa de disparidade dada uma ou mais máscaras. Essas máscaras (e.g. Figura 27c) serão utilizadas como guia para gerar mapas de profundidade mais precisos nas regiões delimitadas por elas com menor variação de profundidade (e.g. Figura 27e). No caso da arquitetura de segmentação utilizamos a MobileSAM Zhang *et al.* (2023a), que é uma versão compacta e mais eficiente em tempo de execução do trabalho Kirillov *et al.* (2023). Além de uma imagem de entrada, essa arquitetura utiliza coordenadas de pontos selecionados na imagem. O resultado é uma geração iterativa de máscaras de segmentação, onde são inferidas as regiões que os pontos de entrada estão contidos. Assim, um objeto presente na imagem pode ser selecionado utilizando YOLO Redmon *et al.* (2015), ou diretamente pelo usuário, para gerar máscaras de segmentação relacionadas com aquele objeto.

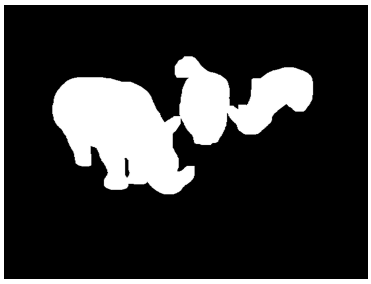
O método proposto é resumido em Algoritmo 6, onde a entrada são as imagens I_D (mapa de profundidade) e I_{RGB} (imagem RGB). Inicialmente, é realizada a inferência na arquitetura



(a) I_{RGB}



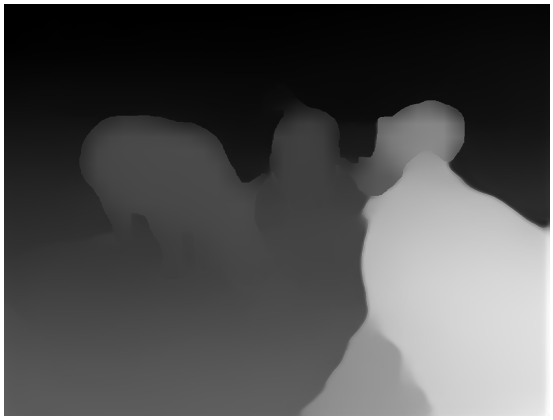
(b) I_D



(c) I_M



(d) I_{DM}



(e) I'_D

Figura 27: Imagens utilizadas e geradas pelo algoritmo 6. Em 27a e 27b vemos as imagens de entrada I_{RGB} e I_D respectivamente. Em 27c é mostrada a máscara de segmentação I_M resultante da inferencia do modelo MobileSAM pré-treinado. Na Figura 27d vemos a profundidade interpolada em cada bloco I_{DM}, que após a multiplicação por I_M temos a imagem de saída I'_D mostrada na Figura 27e.

pré-treinada da MobileSAM, representada pela função `MobileSAM` : $\mathbb{R}^{H\times W\times 3} \to \mathbb{R}^{H\times W\times M}$ onde $M \in \mathbb{N}$ é quantidade de máscaras retornadas. Para que o resultado tenha distribuição de profundidade uniforme dentro das regiões da imagem que serão construídos os triângulos da malha, a entrada também será divida em blocos com dimensões $B_s \times B_s$. O par $\{x_b, y_b\}$ armazena a posição do bloco em relação às dimensões das imagens de entrada. Então o valor de profundidade em cada *pixel* nas regiões da imagem delimitadas pelas extremidades do bloco em cada máscara m é computado por `depth_mask_interpolation` e armazenados em I_{DM}. No final, as máscaras são combinadas acumulando a profundidade I_{DM} computada em cada máscara com I_D. Sendo assim, as posições (i, j) da imagem $I_{DM}(m, i, j) = 0$ para todo $m = 0, \ldots, M-1$ recebem o valor de profundidade da entrada (i.e. $I'_D(i, j) = I_D(i, j)$). As imagens intermediarias do algoritmo são mostradas na Figura 27.

Em Algoritmo 7 é descrita a função `depth_mask_interpolation` que tem como entrada as imagens I_D e I_M, o índice m da máscara onde os *pixels* na região da imagem delimitada por x_b, y_b e B_s. Primeiramente são computadas as posições $\{x_0, y_0\}$ que representam o inicio do bloco e as posições da outra extremidade $\{x_1, y_1\}$ final do bloco (i.e. $x_1 = x_0 + B_s$ e $y_1 = y_0 + B_s$). Depois disso, são computadas as profundidades das 4 extremidades do bloco utilizando a função



(a) I_D (b) I'_D

Figura 28: Região das imagem de profundidade mostradas nas Figura 26a e 26d. Em (a) vemos a região na imagem de entrada I_D e seu melhoramento resultante I'_D em (b).

`max_depth_mask` e armazenado em D_{00}, D_{10}, D_{01} e D_{11}. A computação de cada D utilizando `max_depth_mask` pode ser simplificada através da equação

$$D = \max_{K_i=\{-B_s,B_s\},K_j=\{-B_s,B_s\}} \left(\frac{1}{B_s^2} \sum_{i=x}^{x+K_i} \sum_{j=y}^{y+K_j} I_M(i,j,m) I_D(i,j) \right) \tag{4.1}$$

onde K_i e K_j determinam a janela onde serão computadas as médias de cada quadrante e (m,x,y) são as coordenadas de entrada. Após isso é aplicada uma interpolação bilinear em cada *pixel* em $(i,j) \in [x_0,x_1] \times [y_0,y_1]$ utilizando os valores $\{D_{00},D_{01},D_{10},D_{11}\}$ e armazenando o resultado em $I_{DM}(i,j)$.

Algoritmo 6: `depth_segment_enhancement`

Entrada: $I_{RGB} \in \mathbb{R}^{H\times W\times 3}$, $I_D \in \mathbb{R}^{H\times W}$, $B_s \in \mathbb{N}$
Saída: $I'_D \in \mathbb{R}^{H\times W}$
$I_M \leftarrow$ `MobileSAM`(I_{RGB})
$W_b, H_b \leftarrow \frac{W}{B_s}, \frac{H}{B_s}$
$I'_D \leftarrow I_D$
for $m \leftarrow 0$ ***to*** M **do**
  for $y_b \leftarrow 0$ ***to*** H_b **do**
    for $x_b \leftarrow 0$ ***to*** H_b **do**
      $I_{DM} \leftarrow$ `depth_mask_interpolation`$(I_D, I_M, m, y_b, x_b, B_s)$
      $M_m \leftarrow$ A mascara na posição m de I_M
      $I'_D \leftarrow M_m I'_D + (1 - M_m) I_{DM}$
return I'_D

Algoritmo 7: `depth_mask_interpolation`

Entrada: $I_D \in \mathbb{R}^{H\times W}$, $I_M \in \mathbb{R}^{H\times W\times M}$, $m \in \mathbb{N}$, $y_b \in \mathbb{N}$, $x_b \in \mathbb{N}$, $B_s \in \mathbb{N}$
Saída: $I_{DM} \in \mathbb{R}^{H\times W\times M}$
$x_0, y_0 \leftarrow B_s x_b, B_s y_b$
$x_1, y_1 \leftarrow x_0 + B_s, y_0 + B_s y_b$
$D_{00} \leftarrow$ `max_depth_mask`$(I_D, I_M, x_0, y_0, B_s)$
$D_{10} \leftarrow$ `max_depth_mask`$(I_D, I_M, x_1, y_0, B_s)$
$D_{01} \leftarrow$ `max_depth_mask`$(I_D, I_M, x_0, y_1, B_s)$
$D_{11} \leftarrow$ `max_depth_mask`$(I_D, I_M, x_1, y_1, B_s)$
for $i \leftarrow x_0$ ***to*** x_1 **do**
    for $j \leftarrow y_0$ ***to*** y_1 **do**
        $\psi_0 \leftarrow \frac{x_1 - i}{x_1 - x_0} D_{00} + \frac{i - x_0}{x_1 - x_0} D_{10}$
        $\psi_1 \leftarrow \frac{x_1 - i}{x_1 - x_0} D_{01} + \frac{i - x_0}{x_1 - x_0} D_{11}$
        $I_{DM}(i, j) \leftarrow \frac{y_1 - j}{y_1 - y_0} \psi_0 + \frac{j - y_0}{y_1 - y_0} \psi_1$
return I_{DM}

4.3.3 Processamento dos contornos

Para gerar as imagens utilizadas como entrada do método de geração do *Mesh* 3D, é necessário um *pipeline* para gerar os contornos de *foreground* e *background*, como o descrito em Algoritmo 8. Em $I_{blur} \in \mathbb{R}^{H\times W}$ é armazenado o resultado do borramento do mapa de profundidade, onde $k \in \mathbb{N}$ é o tamanho das dimensões do *kernel* e $\sigma \in \mathbb{R}^+$ é o parâmetro da Gaussiana. Após o borramento, aplicamos um filtro de Canny Canny (1986), onde $I_{canny} \in \mathbb{R}^{H\times W}$ armazena o resultado do filtro, $low_{th} \in \mathbb{R}^+$ é o limiar mínimo da diferença entre os *pixels* e up_{th} o limiar máximo. Como resultado do filtro de Canny temos que os *pixels* de I_{canny} tem valores 1 se forem rotulados como contorno e 0 caso contrario. Como pretendemos gerar a malha 3D para os objetos no *foreground*, após o filtro de Canny, precisamos filtrar pequenos filamentos de descontinuidade que levariam à computação desnecessária de regiões que possuem pouca informação de contexto para o *inpainting*. Para isso, primeiramente aplicamos um algoritmo de *Component Connected Labeling* (CCL), como em Bai *et al.* (2010), onde $I_{Fccl} \in \mathbb{R}^{H\times W}$ armazena a imagem final com cada filamento rotulado de 1 a N_{Fccl}. Depois, os componentes que possuírem uma quantidade de *pixels* inferior a L_{th} serão filtrados utilizando o algoritmo `filter_canny_ccl`. Nesse algoritmo é armazenado em $I_{FE} \in \mathbb{R}^{H\times W}$ a imagem I_{canny} sem os pequenos filamentos.

Após computar os contornos das regiões de *foreground* I_{FE}, precisamos gerar os contornos das regiões de *background*. Para isso, aplicamos a função `back_canny` que rotula com 1 os *pixels* em I_{BCanny} vizinhos das bordas em I_{FE} que possuem diferença maior que b_{th} e os demais com 0. Após isso, assim como nas bordas de *foreground*, aplicamos um método de CCL (obtendo I_{Bccl} e N_{Bccl}) e filtramos utilizando `filter_canny_ccl`, gerando assim a imagem dos contornos de *background* I_{BE}. Na Figura 29 vemos um exemplo da geração desses

Algoritmo 8: `process_edges`

Entrada: $I_D \in \mathbb{R}^{H \times W}$
Saída: $I_{FE} \in \mathbb{R}^{H \times W}$, $I_{BE} \in \mathbb{R}^{H \times W}$
$I_{blur} \leftarrow$ `gaussian_blur`$(I_D, k \leftarrow 3, \sigma \leftarrow 0)$
$I_{FCanny} \leftarrow$ `canny`$(I_D, low_{th} \leftarrow 30, up_{th} \leftarrow 50)$
$I_{Fccl}, N_{Fccl} \leftarrow$ `ccl`(I_{FCanny})
$I_{FE} \leftarrow$ `filter_canny_ccl`$(I_{FCanny}, I_{ccl}, N_{ccl}, L_{th})$
$I_{BCanny} \leftarrow$ `back_canny`(I_D, I_{FE}, b_th)
$I_{Bccl}, N_{Bccl} \leftarrow$ `ccl`(I_{BCanny})
$I_{BE} \leftarrow$ `filter_canny_ccl`$(I_{BCanny}, I_{Bccl}, N_{Bccl})$

contornos, sendo os *pixels* em azul representando as bordas do *foreground* e em verde as bordas de *background*.

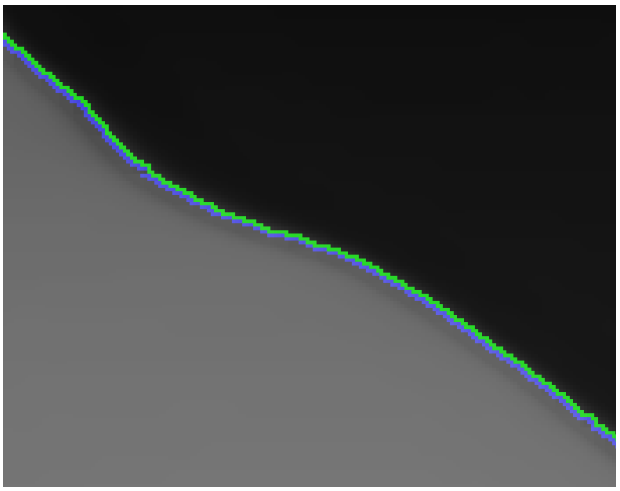


Figura 29: Exemplo de saída do Algoritmo 8, sendo os *pixels* em azul representando as bordas do *foreground* e em verde a borda de *background* de uma região da imagem.

4.3.4 Geração do *Mesh* das regiões de *background*

Para computar o *mesh* das regiões de *background* utilizaremos as imagens I_D, I_{FE}, I_{BE} como entrada para a função descrita em Algoritmo 9. Esse método funciona dividindo a imagem em blocos de tamanho $B_s \times B_s$. Nesses blocos são alocados os vértices V_b, as coordenadas de textura UV e as faces dos triângulos F_b de acordo com as extremidades dos blocos (i.e. cada bloco formará uma malha semelhante à mostrada em Figura 8d). Para posteriormente conectar o *mesh* do *background* com o do *foreground* precisamos rotular quais vértices fazem parte de cada região. Para isso comparamos a profundidade em cada vértice com a média da profundidade em cada região e recebe um rótulo de acordo com a que mais se aproxima. Após isso, se o vértice for rotulado como da região de *background* receberá a profundidade de acordo com sua posição em I_D. Se o vértice estiver na região de *foreground* receberá a profundidade de acordo com a profundidade média da região de *background* para que o vértice fique por trás da malha de

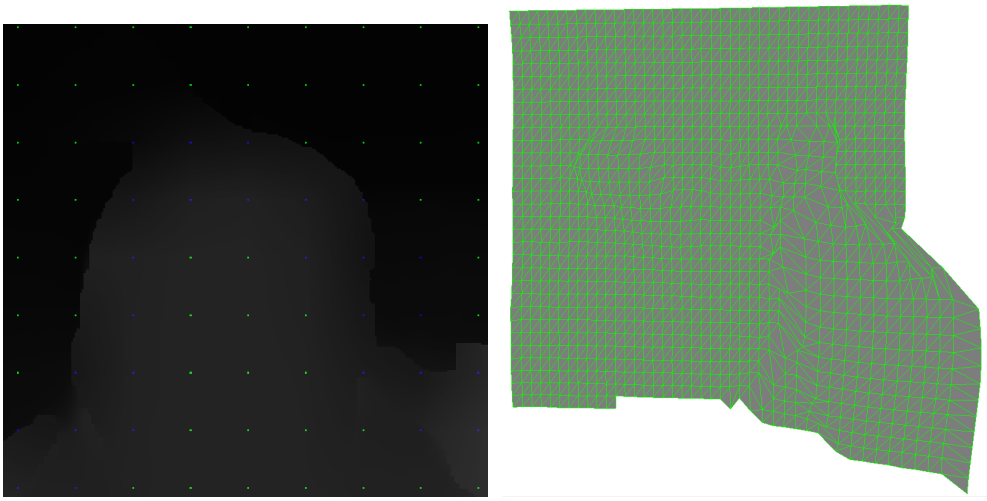
foreground.

Algoritmo 9: `back_mesh_generator`; Algoritmo que retorna V_b, UV_b e F_b utilizados para compor a malha de *background*.

Entrada: $I_D \in \mathbb{R}^{H\times W}$, $I_{FE} \in \mathbb{R}^{H\times W}$, $I_{BE} \in \mathbb{R}^{H\times W}$, $B_s \in \mathbb{N}$
Saída: V_b, UV_b, F_b, $I_{BM} \in \mathbb{R}^{H\times W}$, $I_{Bl} \in \mathbb{R}^{H\times W}$, $I_{BIdx} \in \mathbb{R}^{H\times W}$
$W_b, H_b \leftarrow \frac{W}{B_s}, \frac{H}{B_s}$
$F_b \leftarrow \{\}$
$V_b \leftarrow \{\}$
$UV_b \leftarrow \{\}$
$v_{idx} \leftarrow 0$
for $x_b \leftarrow 0$ ***to*** W_b **do**
    for $y_b \leftarrow 0$ ***to*** H_b **do**
        $x, y \leftarrow B_s x_b, B_s y_b$
        $I_{BIdx}(x,y) \leftarrow v_{idx}$
        $v_{idx} \leftarrow v_{idx} + 1$
        $D_{Fm} \leftarrow \frac{\sum_{i=x}^{x+B_s}\sum_{j=y}^{y+B_s} I_D(i,j) I_{FE}(i,j)}{\sum_{i=x}^{x+B_s}\sum_{j=y}^{y+B_s} I_{FE}(i,j)}$
        $D_{Bm} \leftarrow \min_{i=x,j=y}^{x+B_s,y+B_s}\left(I_D(i,j)\right)$
        $x_v, y_v \leftarrow \frac{2x}{W} - 1, \frac{2y}{H} - 1$
        $x_{uv}, y_{uv} \leftarrow \frac{x}{W}, \frac{y}{H}$
        if $|I_D(x,y) - D_{Fm}| < |I_D(x,y) - D_{Bm}| \wedge \exists 1 \in \{I_{FE}(i,j) | (i,j) \in [x, x+B_s] \times [y, y+B_s]\}$ **then**
            $I_{Bl}(x,y) \leftarrow 2$
            $z_n \leftarrow D_{Bm}$
        else
            $I_{Bl}(x,y) \leftarrow 1$
            $z_n \leftarrow I_D(x,y)$
        $V_b \leftarrow V_b \cup \{(x_v, y_v, z_n)\}$
        $UV_b \leftarrow UV_b \cup \{(x_{uv}, y_{uv})\}$
for $x_b \leftarrow 0$ ***to*** W_b **do**
    for $y_b \leftarrow 0$ ***to*** H_b **do**
        $i_0, i_0, i_e, j_e \leftarrow B_s x_b, B_s y_b, B_s(x_b+1), B_s(y_b+1)$
        $v^0_{idx}, v^1_{idx}, v^2_{idx}, v^3_{idx} \leftarrow I_{BIdx}(i_0, j_0), I_{BIdx}(i_e, j_0), I_{BIdx}(i_0, j_e), I_{BIdx}(i_e, j_e)$
        $I_{BM}(i,j) \leftarrow 1, \forall (i,j) \in [x, x+B_s] \times [y, y+B_s] \wedge \exists I_{Bl}(i,j) = 2 \wedge \nexists I_{FE}(i,j) = 1$
        $face_0, face_1 \leftarrow (v^0_{idx}, v^1_{idx}, v^2_{idx}), (v^1_{idx}, v^2_{idx}, v^3_{idx})$
        $F_b \leftarrow F_b \cup \{face_0, face_1\}$
return V_b, UV_b, F_b

Descrevendo em mais detalhes o Algoritmo 9, primeiramente são inicializadas as variáveis F_b, V_b e UV_b que armazenam os dados que compõe a malha do *background*. Cada deles serão interpretados como *array* vazios, aqui apresentados com a notação de conjuntos $\{\}$. As coordenadas armazenadas em V_b e UV_b, são calculadas com base na posição das extremidades dos blocos (ponto coloridos da Figura 30a). Cada iteração sobre o bloco localizado em (x_b, y_b)

computa a posição no espaço da extremidade localizada em (x,y) da imagem, sendo $x = B_s x_b$ e $y = B_s y_b$ (linha 10). O cálculo das coordenadas do espaço pode ser visto na linha 15 e da textura na linha 16. Para o valor da coordenada do eixo z (profundidade), se dentro do bloco existir algum contorno (i.e. $\exists 1 \in \{I_{FE}(i,j) | (i,j) \in [x, x+B_s] \times [y, y+B_s]\}$) é comparada a profundidade $I_D(x,y)$ com as médias de profundidade do *foreground* D_{Fm} e *background* D_{Bm}. Se a diferença for maior em relação a D_{Fm} o *pixel* é rotulado como *background* (i.e. $I_{Bl}(x,y) \leftarrow 1$, pontos verdes da Figura 30a) e a coordenada z do vértice recebe a profundidade diretamente de $I_D(x.y)$ (linha 22). Em caso contrario, o *pixel* é rotulado como *foreground* (i.e. $I_{Bl} \leftarrow 2$, pontos azuis na Figura 30a) e a coordenada z recebe o valor de D_{Bm}. Isso é feito para garantir que a malha do *background* tenha triângulos nas regiões da imagem onde $I_{FE}(x,y) = 1$ que estejam por trás dos triângulos do *mesh* de *foreground*. Após calcular o valor de todas as coordenadas do vértice, adicionamos essas coordenadas em V_b e UV_b. Como são interpretados como *array*, a operação $\cup$ funciona como um concatenação, logo, seria como adicionar (x_v, y_v, z_n) e (x_{uv}, y_{uv}) na última posição de V e UV, respectivamente (como uma fila). A variável v_{idx} representa a posição de cada vértice em V_b e UV_b, iterando seu valor de acordo com a quantidade de vertices em cada bloco.



(a) (b)

Figura 30: Em (a) temos uma imagem representando os rótulos de uma região da imagem de profundidade I_D. Cada *pixel* colorido representa um vértice $v_b \in V_b$ do *mesh* final, onde os pontos verdes são vértices que o valor de sua componente na dimensão z será igual a profundidade armazenada no *pixel* correspondente. Os azuis representam os vértices de *foreground*, logo recebem na dimensão z o valor médio da profundidade nas bordas de *background* internas dos blocos que o vértice está contido. Em (b) um exemplo de malha 3D de saída da função `back_mesh_generator`.

Com as coordenadas espaciais V_b e textura de cada vértice UV_b da malha de *background* armazenadas, itera-se novamente sobre os blocos para computar a conexão entre os vértices (triângulos) e a máscara do *background* utilizada no *inpainting*. Os triângulos da malha serão representados por tuplas contendo os índices de cada vértice. Logo, recuperamos de I_{BIdx} o valor

do índice de cada extremidade do bloco (linha 28). Como cada bloco possui 4 extremidades, formaremos duas faces de triângulo (linha 30) e adicionamos em F_b. Na Figura 31b vemos um exemplo de construção da malha para a profundidade mostrada em 28b.

Algoritmo 10: `fore_mesh_generator`; Retorna V_f, UV_f e F_f utilizados para compor a malha de *foreground*. Assim como em Algoritmo 9, cada bloco é indexado com (x_b, y_b).

1 **Entrada:** $N_{Bidx} \in \mathbb{N}$, $I_D \in \mathbb{R}^{H\times W}$, $I_{FE} \in \mathbb{R}^{H\times W}$, $I_{BE} \in \mathbb{R}^{H\times W}$, $B_s \in \mathbb{N}$
2 **Saída:** V_f, UV_f, F_f,$I_{FM} \in \mathbb{R}^{H\times W}$, $I_{Fl} \in \mathbb{R}^{H\times W}$, $I_{FIdx} \in \mathbb{R}^{H\times W}$
3 $W_b, H_b \leftarrow \frac{W}{B_s}, \frac{H}{B_s}$
4 $v_{idx} \leftarrow N_{Bidx}$
5 $F_f \leftarrow \{\}$
6 $V_f \leftarrow \{\}$
7 $UV_f \leftarrow \{\}$
8 **for** $x_b \leftarrow 0$ ***to*** W_b **do**
9 **for** $y_b \leftarrow 0$ ***to*** H_b **do**
10 $x, y \leftarrow B_s x_b, B_s y_b$
11 $edges \leftarrow \{\}$
12 **if** $\exists 1 \in \{I_{FE}(i,j) | i \in [x, x+B_s], j \in [y, y+B_s]\}$ **then**
13 $\tau \leftarrow$ `Quadtree`$(I_{FE}, 0, x, x+B_s, y+B_s, y, y+B_s)$
14 $edges \leftarrow$ `QuadtreeToMesh`$(\tau, edges)$
15 $I_{FM}(i,j) \leftarrow 1, \forall (i,j) \in [x, x+B_s] \times [y, y+B_s]$
16 $I_{Fl}(i,j) \leftarrow 2, \forall (i,j) \in edges$
17 **for** $\forall (i,j) \in edges$ **do**
18 $I_{FIdx}(i,j) \leftarrow v_{idx}$
19 $v_{idx} \leftarrow v_{idx} + 1$
20 $V_{list}, UV_{list}, F_{list} \leftarrow$ `get_mesh_from_edges`$(I_{FIdx}, I_D, edges)$
21 $V_f, UV_f, F_f \leftarrow V_f \cup V_{list}, UV_f \cup UV_{list}, F_f \cup F_{list}$
22 **return** V_f, UV_f, F_f, I_{FM}, I_{Fl}, I_{FIdx}

4.3.5 Geração do *Mesh* das regiões de *foreground*

Após computar o V_b e F_b, precisamos calcular o *mesh* dos objetos na região de *foreground*. Para isso, assim como no Algoritmo 9, utilizaremos as imagens I_D, I_{FE} e o número de vértices de *background* N_{Bidx} como entrada para a função descrita em Algoritmo 10. Nesse algoritmo a imagem também será dividida em blocos onde serão construídos *meshes* se algum *pixel* de I_{FE} da região da imagem que o bloco representa conter um valor maior que 0. Para construir essa malha 3D utilizamos o Algoritmo 2 descrito no capítulo 2 em cada um dos blocos. Sendo assim, cada bloco será utilizado como imagem de entrada para Algoritmo 2 e cada aresta retornada será utilizada para computar as faces F_f, os vértices V_f e as coordenadas de textura UV_f.

Descrevendo melhor o Algoritmo 10, faremos uma busca nas regiões delimitadas por $i \in [x, x+B_s]$ e $j \in [y, y+B_s]$ onde $I_{FE}(i,j) = 1$. Inicialmente é computada a posição inicial do

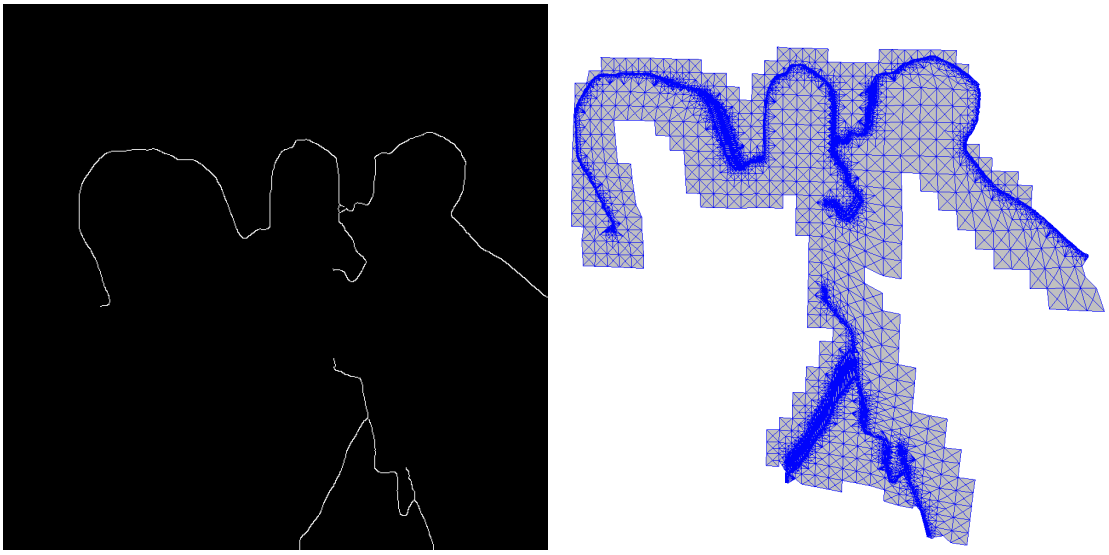
bloco (x,y) e inicializada a variável que armazena as arestas *edges*. Após isso, é verificado se existe algum *pixel* rotulado como *foreground* em I_{FE} (linha 12), em caso verdadeiro é feita a busca chamando a função `Quadtree` com as entradas sendo $I = I_{FE}$, $(i_0, j_0) = (x,y)$ e $(i_e, j_e) = (x+B_s, y+B_s)$. O *quadtree* retornado é utilizado como entrada da função *QuadtreeToMesh* e assim armazenamos em *edges* as arestas retornadas. Após isso, toda região $[x, x+B_s] \times [y, y+B_s]$ é rotulada como máscara de *foreground* em $I_{FM} = 1$. Para posteriormente conectar com a malha do *background*, rotulamos como *foreground* os vertices $(i,j) \in edges$ fazendo $I_{FI}(i,j) = 2$. Após computar os índices de cada um dos vértices presentes em *edges* e armazenando em I_{FIdx}, utilizamos I_{FIdx} e *edges* como entrada do algoritmo `get_mesh_from_edges` para computar as coordenadas V_{list}, UV_{list} e as faces F_{list} presentes no interior do bloco. E assim esses dados vão sendo somados com V_f,UV_f e F_f dos blocos anteriores. Na Figura 31a vemos um exemplo de contorno de *foreground* I_{FE} e a malha resultante 31b utilizando as profundidades de I'_D da Figura 28b.

Algoritmo 11: `get_mesh_from_edges`

```
1  Entrada: edges, I_Idx ∈ ℝ^{H×W}, I_D ∈ ℝ^{H×W}
2  Saída: V_list, UV_list, F_list
3  F_list ← {}
4  V_list ← {}
5  UV_list ← {}
6  for (i, j) ∈ edges do
7      v_idx ← I_Idx(i, j)
8      x_v, y_v ← 2i/W − 1, 2j/H − 1
9      x_uv, y_uv ← i/W, j/H
10     z_n ← I_D(i, j)
11     V_list ← V_list ∪ {(x_v, y_v, z_v)}
12     UV_list ← UV_list ∪ {(x_uv, y_uv)}
13     neighbors_0 ← { Todos os vizinhos conectados com (i, j)}
14     for (i_0^n, j_0^n) ∈ neighbors_0 do
15         neighbors_1 ← Os vizinhos de (i_0^n, j_0^n) que estão conectados com (i, j)
16         for (i_1^n, j_1^n) ∈ neighbors_1 do
17             face ← {I_Idx(i, j), I_Idx(i_0^n, j_0^n), I_Idx(i_1^n, j_1^n)}
18             if face ∉ F then
19                 F_list ← F_list ∪ {face}
20 return V_list, UV_list, F_list
```

Em Algoritmo 11, detalhamos os passos da função `get_mesh_from_edges` para computar V_{list}, UV_{list} e F_{list}. Para cada ponto $(i,j) \in edges$ são computadas as coordenadas espaciais (x_v, y_v, z_v) e de textura (x_{uv}, y_{uv}) e são armazenadas em V_{list} e UV_{list}, respectivamente (linha 11 e 12). Após isso, $neighbors_0$ recebe todos os pontos que formam uma aresta com (i,j) e forma-se a face do triângulo conectando os vizinhos em comum entre (i,j) e $neighbors_0$ (i.e. os pontos contidos em $neighbors_1$, linha 17). Em F_{list} são armazenados a tupla

$(I_{Idx}(i,j), I_{Idx}(i_0^n, j_0^n), I_{Idx}(i_1^n, j_1^n))$. Depois de repetir esse processo para todo $(i,j) \in edges$, são retornadas os conjuntos V_{list}, UV_{list} e F_{list}.



(a) (b)

Figura 31: Em (a) temos uma imagem de exemplo da entrada I_{FE} e em (b) a malha resultante dos V_f, UV_f e F_f da função `fore_mesh_generator`.

4.3.6 Combinando os *Mesh* do *foreground* e do *background*

Após computar as malhas de *foreground* e *background* precisamos conectá-las para formar um *mesh* sem buracos. Para isso, utilizando o método descrito em Algoritmo 12, iremos novamente iterar sobre as regiões das imagens delimitadas por $\{[x, x+B_s], [y, y+B_s]\}$. Se a região possuir algum *pixel* no qual $I_{BM}(x,y) > 0$, a função `get_merge_faces` é chamada para computar as faces F_f contidas no bloco, os vértices V_f e as coordenadas UV_f equivalentes ao centro do bloco. Na Figura 32 podemos o ver a malha resultante depois de concatenar os *arrays* de cada uma das etapas (linhas 11, 12 e 13 de Algoritmo 4).

Na função `get_merge_faces` descrita em Algoritmo 13, armazenamos em v_{list} as posições (x_b, y_b) presentes na borda do bloco nas quais $I_{FIdx}(x_b, y_b) \geq 0$ ou $I_{BIdx}(x_b, y_b) \geq 0$ (i.e. possuem índices válidos em I_{FIdx} e I_{BIdx} dentro do bloco). Se no final da interação a cardinalidade $\|v_{list}\| > 2$, então computamos o vértice no centro do bloco localizado em (i_{mid}, j_{mid}) da imagem com as coordenadas espaciais $v_{mid} = (x_v, y_v, z_n)$ e coordenadas de textura (x_{uv}, y_{uv}), sendo armazenado na posição v_{idx} de V_m e em UV_m. Por último são computados os índices dos vértices em V que estão em v_{list} para conecta-los com o vértice no ponto (i_{mid}, j_{mid}) e formar as faces dos triângulos. Para isso, varremos v_{list} como uma fila e conectamos em cada iteração o ponto (i,j) (com índice v_{idx}^0) e da iteração anterior ij_{step}(com índice v_{idx}^1) e armazenamos em $face$ a tupla que representa a face do triângulo $(v_{idx}, v_{idx}^0, v_{idx}^1)$. Se $face \notin F$, então adicionamos o triângulo em F_{list}.

Algoritmo 12: `merge_mesh_generator`; Retorna V_m, UV_m e F_m utilizados para compor a malha que conecta as malhas de *foreground* e *background*.

Entrada: $N_{Fidx} \in \mathbb{N}$, $I_D \in \mathbb{R}^{H\times W}$, $I_{FIdx} \in \mathbb{R}^{H\times W}$, $I_{BIdx} \in \mathbb{R}^{H\times W}$, $I_{Bl} \in \mathbb{R}^{H\times W}$, $I_{BM} \in \mathbb{R}^{H\times W}$, $B_s \in \mathbb{N}$
Saída: V_m, UV_m, F_m, $I_M \in \mathbb{R}^{H\times W}$
$W_b, H_b \leftarrow \frac{W}{B_s}, \frac{H}{B_s}$
$v_{idx} \leftarrow N_{Fidx}$
$F_m \leftarrow \{\}$
$V_m \leftarrow \{\}$
$UV_m \leftarrow \{\}$
$I_M \leftarrow \{0\}^{H\times W}$
for $x_b \leftarrow 0$ ***to*** W_b **do**
    for $y_b \leftarrow 0$ ***to*** H_b **do**
        $x, y \leftarrow B_s x_b, B_s y_b$
        if $I_{BM}(x,y) > 0$ **then**
            $v_{idx}, V_{list}, UV_{list}, F_{list}, I_{Mb} \leftarrow$ `get_merge_faces`$(v_{idx}, I_D, I_{FIdx}, I_{BIdx}, I_{Bl}, x, y, B_s)$
            $I_M \leftarrow I_M + I_{Mb}$
            $V_m, UV_m, F_m \leftarrow V_m \cup V_{list}, UV_m \cup UV_{list}, F_m \cup F_{list}$
return V_m, UV_m, F_m

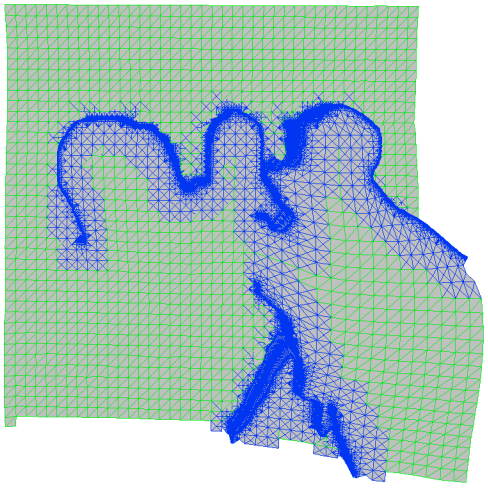


Figura 32: Imagem de exemplo da malha completa modelada por V, UV e F para a entrada I'_D da Figura 28b. Em azul temos a malha de *foreground* e os triângulos formados em `merge_mesh_generator`. Em verde esta a malha de *background*.

4.4 RENDERIZAÇÃO DA MALHA COM MAPEAMENTO DE TEXTURAS

Após a construção das malhas obtendo as coordenadas espaciais, de textura e as conexões que formam os triângulos, precisamos definir como será aplicada a informação de cor nas malhas. Para essa finalidade, utilizaremos mapeamento de textura Szeliski (2011). Pelo fato de nas

Algoritmo 13: `get_merge_faces`

Entrada: $v_{idx} \in \mathbb{N}, I_D \in \mathbb{R}^{H \times W}, I_{FIdx} \in \mathbb{R}^{H \times W}, I_{BIdx} \in \mathbb{R}, I_{Bl} \in \mathbb{R}, I_{Fl} \in \mathbb{R}, x \in \mathbb{N}, y \in \mathbb{N}, B_s \in \mathbb{N}$
Saída: $v_{idx} \in \mathbb{N}, V_{list}, UV_{list}, F_{list}$
$v_{list} \leftarrow \{\}$
$F_{list} \leftarrow \{\}$
$V_{list} \leftarrow \{\}$
$UV_{list} \leftarrow \{\}$
for $x_b \leftarrow x$ ***to*** $x + B_s$ **do**
    for $y_b \leftarrow y$ ***to*** $y + B_s$ **do**
        if $(x_b = x) \vee (y_b = y) \vee (x_b = x + B_s) \vee (y_b = y + B_s)$ **then**
            if $(I_{FIdx}(x_b, y_b) \geq 0) \vee (I_{BIdx}(x_b, y_b) \geq 0)$ **then**
                $v_{list} \leftarrow v_{list} \cup \{(x_b, y_b)\}$
if $\|v_{list}\| > 2$ **then**
    $i_0, i_e, j_0, j_e \leftarrow x, x + B_s, y, y + B_s$
    $i_{mid}, j_{mid} \leftarrow \frac{i_0 + i_e}{2}, \frac{j_0 + j_e}{2}$
    $x_v, y_v \leftarrow \frac{2i_{mid}}{W} - 1, \frac{2j_{mid}}{H} - 1$
    $x_{uv}, y_{uv} \leftarrow \frac{i_{mid}}{W}, \frac{j_{mid}}{H}$
    $z_n \leftarrow I_D(i_{mid}, j_{mid})$
    $v_{mid}, uv_{mid} \leftarrow (x_v, y_v, z_n), (x_{uv}, y_{uv})$
    $v_{idx} \leftarrow v_{idx} + 1$
    $V_{list}, UV_{list} \leftarrow V_m \cup \{v_{mid}\}, UV_m \cup \{uv_{mid}\}$
    $ij_{step} \leftarrow$ Primeiro par adicionado em v_{list}
    for $(i, j) \in v_{list} \setminus ij_{step}$ **do**
        if $I_{Bl}(i, j) > 0$ **then**
            $v^0_{idx}, v^1_{idx} \leftarrow I_{BIdx}(i, j), I_{BIdx}(ij_{step})$
        else
            $v^0_{idx}, v^1_{idx} \leftarrow I_{FIdx}(i, j), I_{FIdx}(ij_{step})$
        $face \leftarrow \{v_{idx}, v^0_{idx}, v^1_{idx})\}$
        if $face \notin F$ **then**
            $F_{list} \leftarrow F_{list} \cup \{face\}$
        $ij_{step} \leftarrow (i, j)$
return $v_{idx}, V_{list}, UV_{list}, F_{list}, I_{FM}$

regiões ocluídas possuírem contexto diferente do objeto à sua frente, definiremos uma imagem textura para a malha de *background* e outra para o *foreground*. Sendo assim, utilizaremos a imagem de entrada como a textura do *foreground* e para o *background* será o resultado da inferência em LaMaSep, preenchendo o conteúdo de cor das regiões ocluídas.

Para a criação das máscaras de *inpainting*, utilizamos o método descrito em Jampani *et al.* (2021) chamado de *Soft Disocclusions* (SD) com a máscara criada na execução do DQ-Mesh. A saída da inferência no modelo LaMa, é utilizado como textura pelo mapeamento definido pelas coordenadas armazenadas UV_b e a imagem original é utilizada com as coordenadas UV_f e

UV_m. Para evitar artefatos devido a triângulos formados na malha de *foreground* que possuem vértices com uma grande diferença na coordenada z (Figura 37a), ou seja com maior diferença de profundidade de acordo com I_D, utilizamos a função *Soft Foreground Pixel Visibility* (SFPV) para computar uma imagem alfa, fazendo que os *pixels* presentes nas regiões com maior gradiente em I_D fiquem transparentes.

4.4.1 Geração das máscaras de *inpainting* e da textura do *background*

Na construção da máscara para o *inpainting*, é necessario computar quais *pixels* tem o potencial de serem desocluídos quando a câmera é movimentada. O *pixel* do *background* provavelmente será desocluído se seu valor no mapa de profundidade for maior comparado aos *pixels* ao seu redor. Essa ideia é formulada em Jampani *et al.* (2021), definindo a função *SD* como

$$SD(x,y) = \text{ReLU}\left(\tanh\left(\gamma \max_{(x_i,y_j)}\left(I_D(x,y) - I_D(x_i,y_j) - \rho K_{(x_i,y_j)}\right)\right)\right) \tag{4.2}$$

onde $K_{(x_i,y_j)} = \sqrt{(x_i-x)^2+(y_j-y)^2}$ (i.e. a distância entre (x,y) e (x_i,y_j)), $\rho \in \mathbb{R}$ é o parâmetro que controla a influência de $K_{(x_i,y_j)}$ e $\gamma \in \mathbb{R}$ controla a inclinação da função tanh. Para considerar apenas a parte positiva de tanh, é utilizada a função $\text{ReLU}(x) = \max(0,x)$ presente em diversas arquiteturas de *deep learning* como ativação Agarap (2018). Como seria custoso computar a diferença de disparidade entre todos os *pixels* da imagem, a comparação é feita apenas sobre uma quantidade fixa m dos vizinhos na horizontal e na vertical, como é mostrado em vermelho Figura 33. Na Figura 34 vemos exemplos do uso da função para o mapa de profundidade mostrado em 34a.

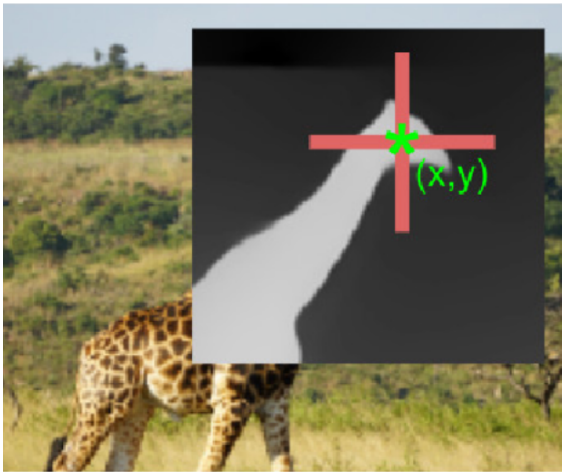
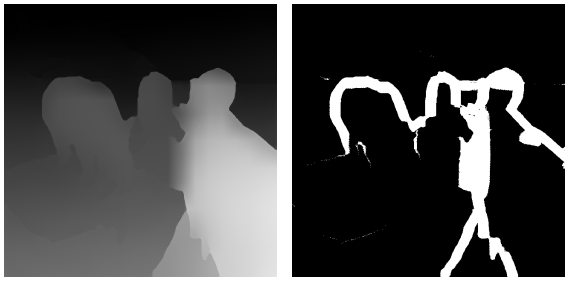


Figura 33: Em cada ponto (x,y) (em verde) da imagem, são calculadas as diferenças de profundidade sobre as linhas horizontais e verticais de comprimento m (em vermelho) para computar os mapas de SD. Fonte: Jampani *et al.* (2021)

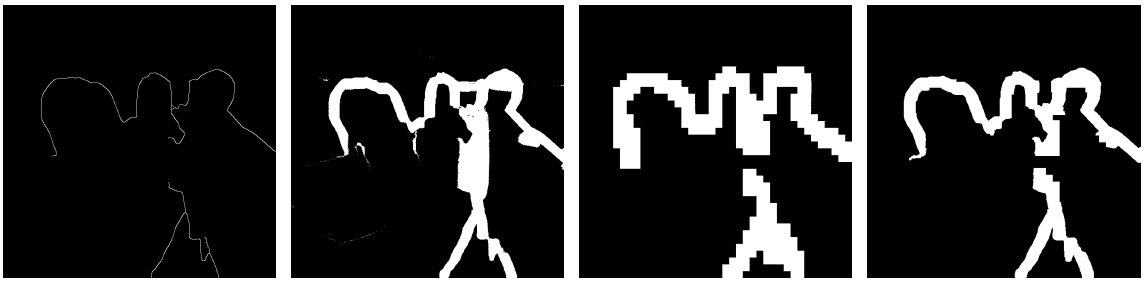


(a) I_D (b) $\rho = 0.7, \gamma = 1, m = 25$

Figura 34: Exemplo de funcionamento da função *SD*. Em (a) temos a imagem de entrada I_D. Em (b) vemos a mascara resultante da função *SD* de acordo com os parâmetros ρ, γ e m.

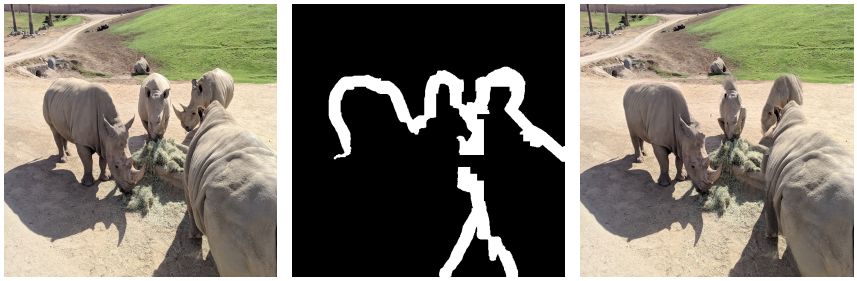
Para que não seja aplicado o *inpaint* em regiões que não estão ocluídas pela malha de *foreground*, computamos a máscara utilizada fazendo $M = I_M SD$. Sendo assim, os blocos que foram rotulados com 0 durante a computação do DQ-Mesh não precisaram ter a cor preenchida com o contexto do *background*. Na Figura 35 vemos exemplos de máscaras. Na Figura 35b vemos que *SD* possuem alguns *pixels* em regiões da imagem que não estão presentes no contorno (Figura 35a) utilizado no DQ-Mesh e em Figura 35d vemos que esse problema é resolvido através da multiplicação entre a máscara do DQ-Mesh e o SD.

Após a computação da máscara é colocado como entrada da arquitetura LaMaSep treinada e o resultado é utilizado como imagem de textura pelas coordenadas UV_b. Em Figura 36a e 36b vemos um exemplo da imagem RGB e a máscara M que serão a entrada do modelo. A saída da inferência I_{BTex} utilizada como textura da malha de *background* é mostrada na Figura 36c.



(a) I_{FE} (b) *SD* (c) I_M (d) $M = I_M SD$

Figura 35: Exemplo mostrando o efeito da combinação entre a máscara gerada por DQ-Mesh e SD. Em (a) vemos os contornos e como em (b) existem regiões da mascara SD que não estão presentes contornos em I_{FE}. Utilizando a imagem em (c) multiplicando com (b) temo o resultado em (d), sem os *pixels* indesejados para processar o *inpaint*.



(a) I_{RGB} (b) M (c) I_{BTex}

Figura 36: Exemplo da utilização do modelo treinado da arquitetura LaMaSep. Em (a) e (b) vemos as entradas da rede e em (c) vemos a imagem da inferência.

4.4.2 Geração da textura do *foreground* com SFPV

Se apenas utilizarmos a imagem RGB de entrada diretamente como textura para as coordenadas UV_f e UV_m os triângulos que possuem vértices com maior diferença na coordenada z irão interpolar pequenas regiões de cor, causando um efeito de esticamento nos contornos dos objetos, (e.g Figura 37a). Para resolver esse problema utilizaremos a abordagem SFPV proposta em Jampani *et al.* (2021) que consiste em aplicar o filtro definido pela equação

$$A = e^{-\beta((\nabla_x I_D)^2+(\nabla_y I_D)^2)} \tag{4.3}$$

sendo ∇_x e ∇_y os operadores Sobel Gonzalez & Woods (2008) na largura e altura de I_D e $\beta \in \mathbb{R}$ um parametro escalar que calcula a suavidade da transparência de acordo o grau do gradiente. O resultado A desse filtro é utilizado como canal alfa junto a imagem RGB de entrada, formando a textura da malha de *foreground* I_{FTex}. Na Figura 38b vemos a saída desse método utilizando Figura 38a como entrada e em Figura 38c vemos a imagem de textura resultante para a região de *foreground*. Em Figura 37b vemos que o problema da aparência de esticamento é diminuído.



(a) M (b) I_{RGB}

Figura 37: Em (a) vemos o efeito de esticamento se não for utilizado SFPV como canal alfa. Caso SFPV for utilizado, como os vértices que estão presentes nesse esticamento interpolam a cor com um canal alfa, vemos em (b) que o esticamento é removido.



(a) I_{RGB} (b) M (c) I_{FTex}

Figura 38: Em (a) vemos a profundidade para computar a imagem (b) com SFPV. Em (c) vemos a textura que será utilizada pela malha do *foreground* com seus respectivos valores de alfa.

4.4.3 Renderização

Para gerar os *frames* dada uma posição de câmera precisamos de um método de renderização. Como é construída uma malha 3D, utilizamos a rasterização Gomes *et al.* (2012), que é uma abordagem com uma baixa complexidade computacional. Sendo assim, cada um dos triângulos que compõe o *mesh* são projetadas num plano dada uma matriz de câmera C. Esse plano é interpretado como a tela onde serão mostradas as imagens renderizadas. As cores dos pontos nessa tela são definidos pelo triângulo mais próximo (i.e. utilizando z-buffer) do plano que foi projetado. Por sua vez as cores são interpoladas de acordo com a região da imagem que as coordenadas em textura UV do triângulo estão mapeadas.

5

EXPERIMENTOS

Para avaliar o desempenho dos métodos propostos, faremos uma análise qualitativa baseada em cenários reais de imagens captadas por aparelhos *mobile*. Foi escolhido aleatoriamente um subconjunto de 12 imagens do banco de dados HDR+ Hasinoff *et al.* (2016), (Figuras 39 e 40), pois são imagens de alta resolução (4208×3120) capturadas por câmeras móveis (mais especificamente o *smartphone* Pixel da Google). Como nosso método propõe uma melhoria ao SLIDE Jampani *et al.* (2021), comparamos imagens renderizadas utilizando a mesma posição de câmera e os *meshes* gerados pela estratégia sugerida em Jampani *et al.* (2021) e utilizando DQ-Mesh. As imagens renderizadas para comparação foram escolhidas entre 20 outras geradas seguindo uma trajetória circular entre os eixos x e y e aproximando no eixo z.

Tentaremos demonstrar com essas comparações nas duas primeiras seções que existe um ganho de qualidade nas imagens geradas quando se utiliza a rasterização com mapeamento de textura ao invés de utilizar todos os *pixels* das imagens RGBD como vértices. Logo, a imagem de entrada terá que ser redimensionada de acordo com o mapa de profundidade para que cada vértice receba a cor de um *pixel*. Ou seja, as dimensões do mapa de profundidade definem a quantidade de vértices e o quanto de detalhes de cor estarão presentes na imagem renderizada. Para comparação, as malhas geradas com o SLIDE Jampani *et al.* (2021) utilizaram mapas de profundidade com dimensões 2014×1560 para diminuir a perda de detalhes das imagens de entrada. No caso do SLIDE utilizando o DQ-Mesh (i.e. o *pipeline* proposto para geração das malhas 3d), geramos as malhas utilizando mapas de profundidade 640×480 (mantendo o *aspect ratio*) e blocos com $B_s = 16$ (i.e. com dimensões 16×16), porém utilizando como textura imagens 2014×1560. Logo, as duas formas de gerar as malhas compartilham o mesmo conteúdo de cor. As imagens renderizadas tem tamanho 1024×1024 e são mostradas nas Figuras 39,40,43 e 44.

Chamaremos durante as seções seguintes de SLIDE a abordagem semelhante à proposta em Jampani *et al.* (2021), porém utilizando os modelos treinados de MiDaS v3.0, MiDaS *mobile* e LaMaSep. Em SLIDE a malha é gerada da mesma forma que em Jampani *et al.* (2021), construindo um LDI considerando todos os *pixels* como vértices conectados como na Figura 8d para formar um *mesh* sem utilizar mapeamento de textura (armazenando a cor do *pixel* que originou o vértice). É chamada de SL+DQ-Mesh a abordagem que utiliza os mesmos modelos

que em SLIDE porém a malha é gerada utilizando DQ-Mesh. E chamaremos de SL+DQ-Mesh+Seg, onde é seguido o mesmo processo de SL+DQ-Mesh porém o mapa de profundidade antes de ser utilizado como entrada em DQ-Mesh é melhorado utilizando segmentação através da implementação do algoritmo `depth_segment_enhancement`.

Como o trabalho propõe uma abordagem para dispositivos *mobile*, na última seção comparamos a economia de memória nas malhas geradas pelo DQ-Mesh e se convertemos todos os *pixel* do mapa de profundidade em vértices conectados como na Figura 8d. Além disso, o *pipeline* proposto foi implementado utilizando C++ e Halide Ragan-Kelley *et al.* (2013). Os modelos foram implementados utilizando a API do Tensorflow Lite para C++. Esses resultados estão nas Tabelas 1 (na análise do uso de memória) e 2 (análise do tempo).

5.1 EXPERIMENTOS COM MIDASV3.1

Na Figura 39 vemos na primeira coluna as imagens de entrada e nas seguintes o resultado das renderizações utilizando as abordagens que serão comparadas. Na Imagem 1 da Figura 39 vemos que SLIDE produz um *mesh* no qual quando renderizamos utilizando certa posição de câmera, remove parte do rosto da pessoa que aparece na imagem, cortando seu nariz e seus olhos. A malha gerada utilizando DQ-Mesh porém consegue ser um pouco mais precisa nos contornos devido a seu método de melhoramento simples da profundidade. O *mesh* resultante de SL+DQ-Mesh+Seg utilizando a melhoria de profundidade com segmentação possui contornos um pouco mais precisos, devido a precisão da máscara de segmentação, fazendo com que fiquem mais bem definidas as pálpebras e os cílios dos olhos.

Na Imagem 2 vemos uma diferença maior entre as abordagens. Como o SLIDE utiliza LDI e a profundidade inferida pelo LaMaSeg no contexto definido pela máscara resultante do algoritmo SD, a profundidade inferida foi mais próxima da câmera que a profundidade utilizada no *foreground*. Logo, os braços da pessoa na imagem acabam sendo cortados e algumas regiões atrás da pessoa acabam distorcidas devido a variações de profundidade. Podemos ver esse mesmo problema na Imagem 3 perto da churrasqueira, onde vemos a cor do *background* inferida por LaMaSep. Nos resultados utilizando DQ-Mesh, como o preenchimento é feito através da média da profundidade dos contornos do *background*, a profundidade na região do braço (Imagem 2) e da churrasqueira (Imagem 3) é melhor distribuída, evitando que esse tipo de artefato aconteça.

Na Imagem 4 vemos um exemplo onde os métodos tiveram praticamente o mesmo resultado, com exceção da região perto da escada atrás do caminhão, que ficou um pouco distorcida na saída do SLIDE. Em Imagem 5 e 6 os resultados ficaram próximos em qualidade, porém na Imagem 5 o método SLIDE distorceu um pouco o olho da pessoa que está atrás do objeto de vidro e em 6 a parte direita dos óculos está faltando nos resultados do SLIDE e do SL+DQ-Mesh, porém esse problema aparece em menor grau após a melhoria da profundidade utilizando segmentação.



Figura 39: Experimentos com o primeiro subconjunto de imagens do HDR+ utilizando MiDaSv3.0. Na primeira coluna são apresentados os nomes dados às imagens analisadas. A entrada dos métodos para gerar a representação é mostrada na segunda coluna. Na terceira coluna vemos o resultado da renderização através da abordagem SLIDE para construir a malha. Os resultados das malhas utilizando DQ-Mesh e DQ-Mesh mais a melhoria da profundidade são mostrados na quarta e quinta coluna respectivamente.



Figura 40: Experimentos com o segundo subconjunto de imagens do HDR+ utilizando MiDaSv3.0. Descrição das colunas semelhante a Figura 39.

Em Figura 40, vemos outras imagens do conjunto HDR+. Na Imagem 7 vemos falhas no rinoceronte mais próximo da câmera que o SLIDE errou novamente no preenchimento da profundidade de *background*, deformando parte do animal. Porém, nas 3 abordagens aparece a região do *background* com partes de cor de contexto, na região do focinho do rinoceronte localizado na parte superior direita. A Imagem 8 também mostra novamente resultados bem próximos entre os métodos, com pequena deformação na mão direita da estátua no resultado do SLIDE, resultante de *background* sobressalente perto dos contornos.

Em Imagem 9 vemos falhas nos resultados do SL+DQ-Mesh e SL+DQ-Mesh+Seg, onde aparece na malha de background parte da textura do *foreground* que não é preenchido pelo *inpainting*. Vemos esses artefatos na parte superior e na esquerda perto do fim da gruta. Porém na Imagem 10 vemos que o resultado do SLIDE perto da árvore acaba esticando parte do *background*, o que não acontece nas saídas dos outros métodos.

Na Imagem 11 vemos que a malha de *background* do SLIDE sobrepõe a de *foreground*, causando esticamento e a remoção de uma das flores pelo conteúdo da textura do *background*. No resultado do SL+DQ-Mesh vemos que esse problema não aparece, porém o caule da flor mais a esquerda está um pouco distorcido. Nenhum desses problemas aparece na imagem gerada pela renderização da malha de SL+DQ-Mesh+Seg, porém, devido a imprecisão na máscara de segmentação, o dedão está com um pedaço faltando.

Na Imagem 12 vemos que os métodos mais uma vez tiveram resultados bem próximos. Nesse o SLIDE foi mais preciso na região entre a rena da estátua e a torre da igreja do que SL+DQ-Mesh e SL+DQ-Mesh+Seg. Porém possui deformidades perto da base direita da estátua, que aparece também em SL+DQ-Mesh, mas é resolvido em SL+DQ-Mesh+Seg.

Para avaliar a melhoria da profundidade utilizando segmentação, na Figura 42 comparamos algumas imagens de saída do modelo MiDaS v3.0 com a respectiva melhoria. Em todas imagens vemos que o efeito desejado dessa melhoria é alcançado, quando as regiões dentro das máscaras possuem profundidade suavizadas, diminuindo grandes variações dentro da máscara. Na Imagem 3 vemos que parte do boné era cortado pela profundidade, e acaba sendo preenchido após a melhoria. Em Imagem 10 vemos que a árvore fica mais nítida e a montanha mais suavizada. Em Imagem 2 vemos que no quadril esquerdo, como a máscara que contorna a pessoa e a região a sua esquerda, no algoritmo é computado como apenas uma região, porém sem trazer grandes artefatos na malha resultante, pois a região foi suavizada. Nas Imagens 1, 7 e 8 vemos melhorias mais evidentes, onde os objetos ficaram mais nítidos com menor variação de profundidade dentro deles.

Em Figura 41 vemos as zonas de interesse citadas anteriormente que possuem alguns artefatos nas imagens renderizadas. Na Imagem 3 vemos mais um artefato causado pela imprecisão do mapa de profundidade, onde vemos que em SLIDE (em menor escala) e em SL+DQ-Mesh parte do braço da pessoa de verde é removida. Na saída do método SL+DQ-Mesh+Seg esse artefato não é encontrado, já que toda a região que é segmentada como uma pessoa tem a profundidade suavizada. Na Imagem 1 vemos de mais de perto o problema citado da remoção dos

olhos e da boca da pessoa. Na ampliação da Imagem 11 vemos bem as deformações presentes nas flores no resultado do SLIDE e na Imagem 2 vemos o problema nos braços.



Figura 41: Algumas zonas de interesse das renderizações mostradas nas Figuras 39 e 40. A descrição das colunas é semelhante a descrita em 39, porém ao invés dos resultados nas 3 ultimas colunas estão as regiões de interesse.

5.2 EXPERIMENTOS COM MIDAS *MOBILE*

São comparados na Figura 43 e 44 os métodos para a construção das malhas 3D utilizando a arquitetura MiDaS *mobile* para inferência e as mesmas imagens de entrada mostradas nas Figuras 39 e 40. Na Imagem 1 vemos a mesma deformação na malha do método SLIDE na Figura 39, retirando os olhos e o nariz da pessoa. Utilizando DQ-Mesh, esse artefato também aparece, devido a maior imprecisão do mapa de profundidade. Vemos que em DQ-Mesh + Segmentação o problema não aparece, mostrando que a melhoria do mapa de profundidade foi eficaz em resolver o problema da imprecisão.

Na Imagem 2, dessa vez o destaque positivo fica com *mesh* resultante do SLIDE, que ficou apenas com pequenas deformações na região do braço direito da mulher. Isso acontece pelo fato do mapa de profundidade não ter grandes descontinuidades nas regiões dos braços, sendo processados como *background*, como podemos ver na Figura 46. Já em DQ-Mesh e em DQ-Mesh + Segmentação, vemos falhas nas extremidades dos braços, causadas pela diferença de profundidade entre o braço e a região da porta, delimitando a porta como *foreground* separando a região do braço que é processada como *background* e da porta como *foreground*.

Na Imagem 3 vemos resultados próximos entre os três métodos. As diferenças mais notáveis são a remoção da aba do boné nos métodos SLIDE e DQ-Mesh e pequenas deformidades no braço da mulher no meio e da mulher na lado direito na saída do SLIDE. Em DQ-Mesh + segmentação vemos que essas deformidades não são apresentadas. Porém, vemos na Imagem 4 que nesse caso o SLIDE e DQ-Mesh acabaram gerando um *mesh* que remove parte da frente do caminhão, devido ao mapa de profundidade que não cobre essa região devidamente. Problema esse que novamente não aparece na malha de saída de DQ-Mesh + Segmentação. Nas Imagens 5 e 6 vemos que os três métodos obtiveram resultados similares, sem grandes deformações.

Focando nas imagens apresentadas na Figura 44, vemos na Imagem 7 que o método DQ-Mesh e DQ-Mesh na imagem de textura do *foreground* cortaram parte do chifre do rinoceronte devida a imprecisão do mapa de profundidade. Na Imagem 8, os resultados dos três métodos praticamente não possuem diferença. Já na Imagem 9, vemos que dessa vez a abordagem SLIDE apresenta o mesmo artefato que em DQ-Mesh, mostrando parte da textura do *background*. Em DQ-Mesh+Segmentação, por sua vez, não aparecem tais artefatos.

Nas Imagens 10 e 12 os três métodos alcançaram resultados similares, com nenhuma diferença notável. Já na Imagem 11 aparece um aspecto de *ghosting* na região inferior esquerda (no objeto preto e nas mãos) nos resultados das abordagens SLIDE e DQ-Mesh. Ja em DQ-Mesh+Segmentação aparece o mesmo artefato do dedão presente em Figura 40, devido ao mesmo motivo.

É comparado na Figura 46 o efeito da melhoria no mapa de profundidade utilizando a inferência do MiDaS *mobile*. Na Imagem 3 temos o mesmo problema que o apresentado em 42 e vemos que também nesse caso a melhoria consegue recuperar a região da profundidade na aba do boné. Na Imagem 10 a imagem é novamente suavizada na montanha e denotando melhor a árvore, porém ainda com a profundidade muito próxima da montanha. Na Imagem 2 vemos que a diferença de profundidade nos braços em relação ao corpo é notável. Sendo assim, a melhoria, mesmo que fazendo aparecer a região do braço, ainda irá calcular com base na maior profundidade na região delimitada pela máscara de segmentação. Na Imagem 1 vemos que a profundidade fica mais ajustada aos contornos da mulher, porém, devido a "nuvens"de profundidade na inferência do MiDaS *mobile*, vemos que perto dos contornos temos profundidade próximas ao *foreground*. Nas Imagens 7 e 8 vemos que os objetos ficam mais nítidos e com menor variação de profundidade dentro da máscara.

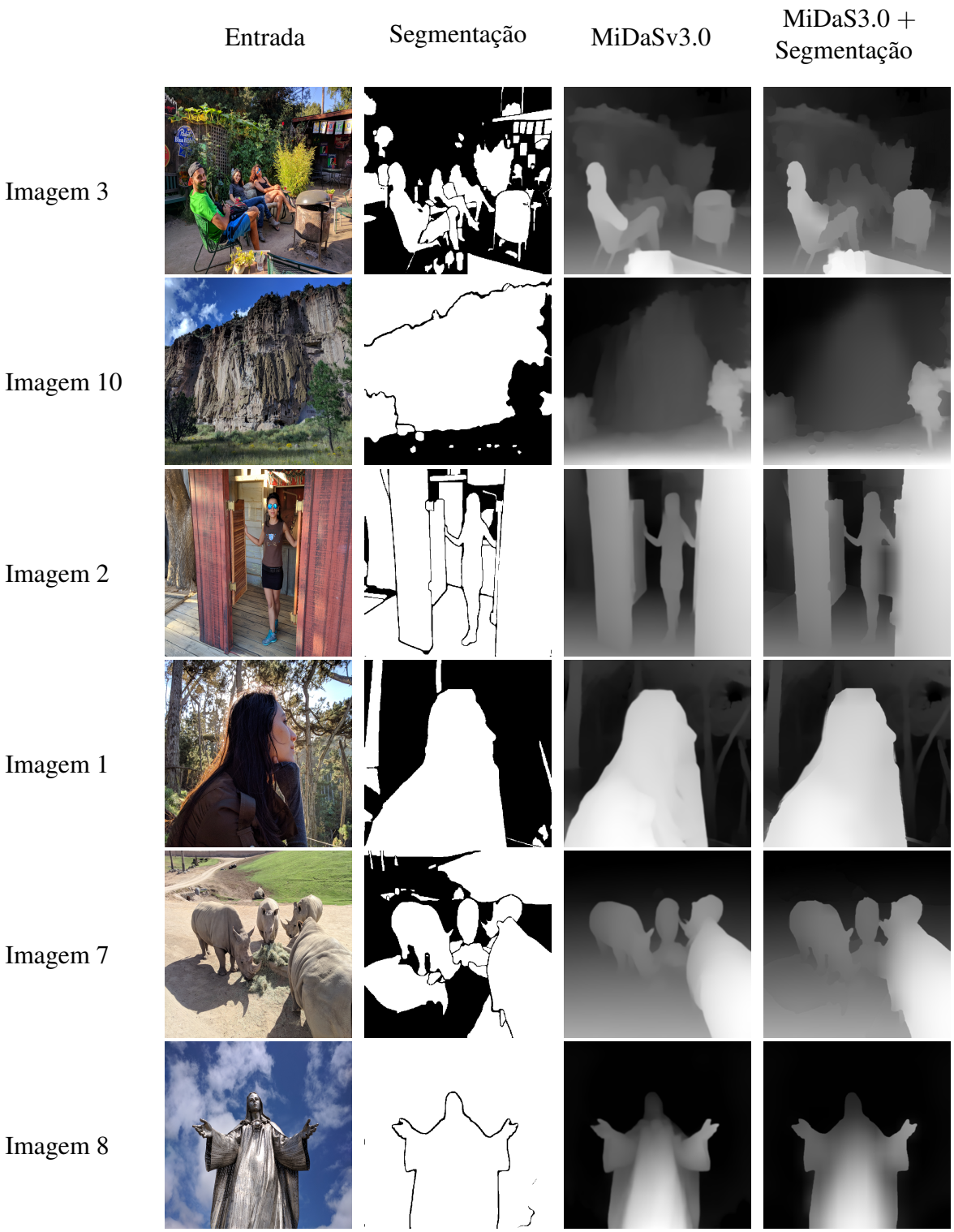


Figura 42: Analise do impacto da melhoria de profundidade utilizando segmentação. Na primeira e segunda coluna estão os nomes e as imagens RGB de entrada para os modelos MiDaSv3.0 e MobileSAM, respectivamente. Na terceira coluna vemos as mascaras de segmentação inferidas por MobileSAM. Os resultados da melhoria utilizando as imagens de inferencia do MiDaSv3.0 (quarta coluna) são mostrado na quinta coluna.



Figura 43: Experimentos com o primeiro subconjunto de imagens do HDR+ utilizando MiDaS *mobile*. Descrição das colunas semelhante a Figura 39.



Figura 44: Experimentos com o segundo subconjunto de imagens do HDR+ utilizando MiDaS *mobile*. Descrição das colunas semelhante a Figura 39.

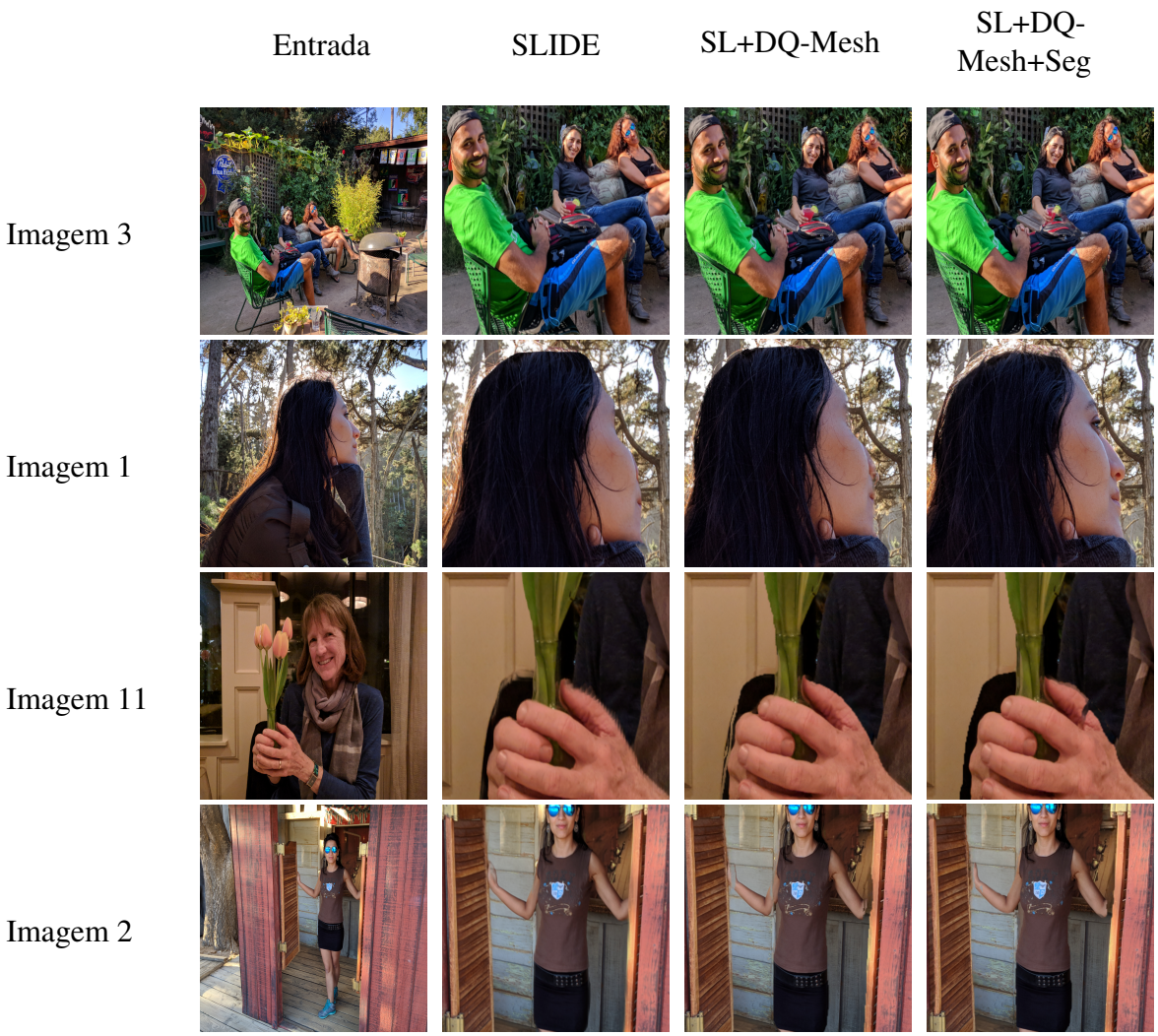


Figura 45: Algumas regiões de interesse de interesse das renderizações mostradas nas Figuras 43 e 44. A descrição das colunas é semelhante a 41.

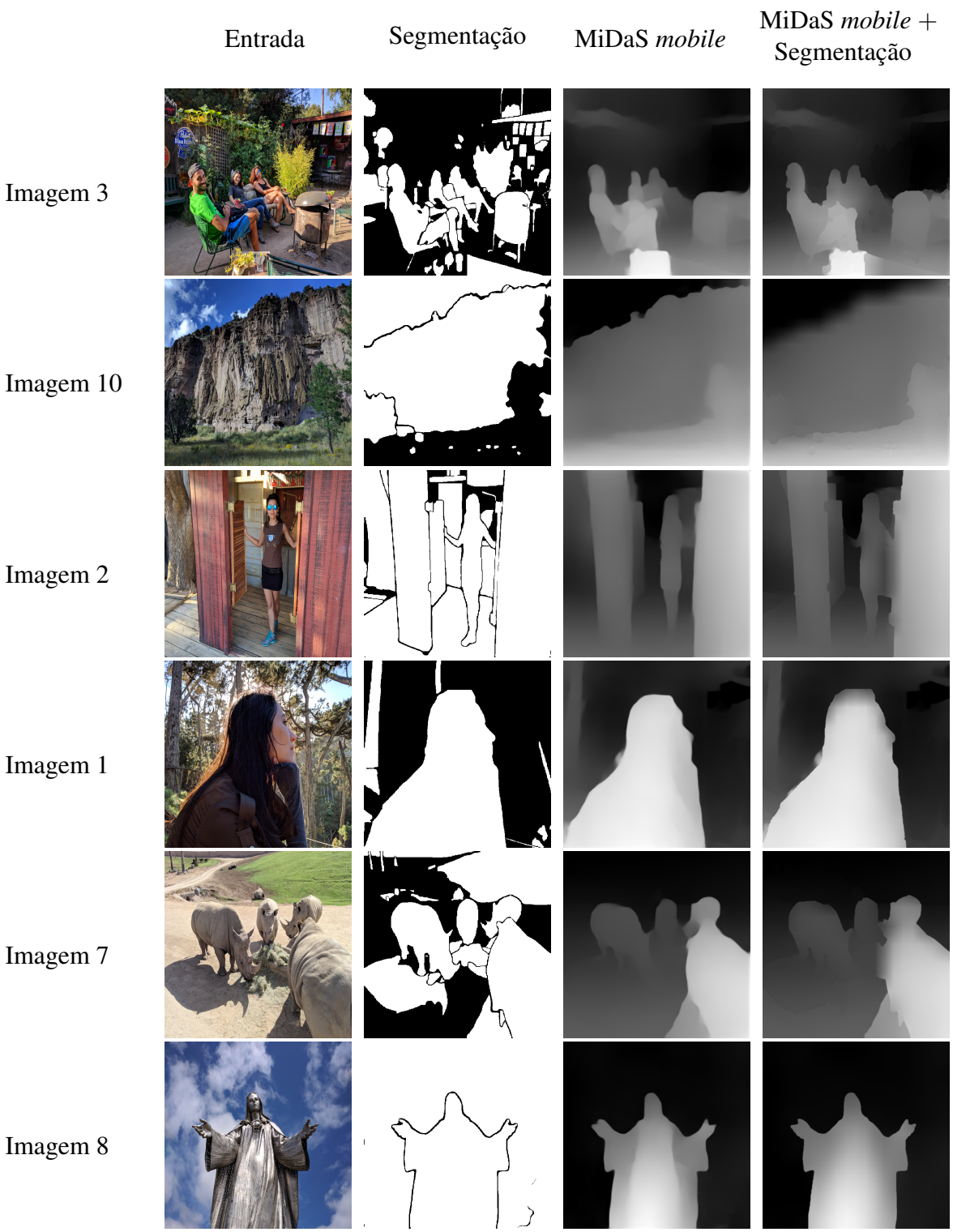


Figura 46: Mesma descrição que a Figura 42, porém ao invés de mostrar a inferência da arquitetura MiDaSv3.0 são mostrados os resultados com MiDaS *mobile*.

Dimensões da entrada ($H \times W$), Dimensões do bloco ($B_s \times B_s$)	Nº Vertices SLIDE	Nº Vertices* DQ-Mesh	Ganho
640×640, 16×16	819200	$\sim$23885	$\sim$34.3$\times$
1280×1280, 32×32	3276800	$\sim$64992	$\sim$50.4$\times$
2560×2560, 64×64	13107200	$\sim$142131	$\sim$92.2$\times$

Tabela 1: Tabela com a comparação da quantidade de vértices em cada uma das abordagens. Em SLIDE Jampani *et al.* (2021) os *mesh* são criados utilizando todos os *pixels* do mapa de profundidade como vértices conectados com os seus vizinhos. No caso da construção da malha com DQ-Mesh, depende-se dos contornos computados em I_{FE}, sendo assim, os valores obtidos foram calculados através de uma média utilizando imagens do banco de dados HDR+ Hasinoff *et al.* (2016) e o modelo MiDaS *mobile* para o mapa de profundidade. Na última coluna é mostrado o ganho $\frac{N_v}{\mathscr{E}_v}$, sendo N_v o número de vertices na coluna do SLIDE e $\mathscr{E}_v$ na coluna do DQ-Mesh.

5.3 ANÁLISE DE DESEMPENHO

Na Tabela 1 vemos a quantidade de vértices gerados pelas abordagens SLIDE (i.e. todo os *pixels* do mapa de profundidade são vértices) e DQ-Mesh para avaliar a eficiência em uso de memória. Na primeira coluna estão as dimensões dos mapas de profundidade $H \times W$ e o tamanho do bloco B_s onde serão computados os vértices da malha de *foreground* do DQ-Mesh (mantendo o mesmo número de vértices na malha *background* independente de $H \times W$). Na segunda coluna, é apresentado o número de vértices N_v caso todos os *pixels* do mapa de profundidade do *background* e do *foreground* forem conectados com seus vizinhos (como é descrito em SLIDE Jampani *et al.* (2021)), fazendo $N_v = 2HW$. Como o método de DQ-Mesh computa um número de vértices de acordo com os *pixels* em I_{FE}, na segunda coluna é mostrada a média do número de vértices $\mathscr{E}_v$ gerados sobre um subconjunto de 150 imagens do banco de dados HDR+ Hasinoff *et al.* (2016). As imagens de profundidade utilizadas para construir as malhas foram geradas utilizando MiDaS *mobile* e redimensionadas utilizando interpolação bicubica. Na terceira coluna é mostrado o ganho $\frac{N_v}{\mathscr{E}_V}$ em cada uma das 3 dimensões. Analisando esta coluna, vemos que o DQ-Mesh alcança uma economia de $\sim 90.21\times$ em relação a abordagem em SLIDE que pretende armazenar as cores dos vértices utilizando uma imagem com dimensões 2560×2560. Essa comparação é feita apenas caso deseja-se utilizar o DQ-Mesh com mapas de profundidade com as mesmas dimensões que a utilizada para construir as malhas em SLIDE. Esse ganho então pode ser ainda maior, devido ao mapeamento de textura conseguimos utilizar imagens de qualquer resolução independente das dimensões do mapa de profundidade utilizado.

Analisando os tempos de execução em dispositivos móveis, foram realizados dois experimentos. No primeiro foi utilizado o mesmo subconjunto do banco HDR+ de 150 imagens para executar no *pipeline* completo para gerar a representação 3D em 2 dispositivos da Motorola (Edge 20 + e Edge 40 Ultra). Assim, mediu-se tempo da inferência do mapa de profundidade (no caso da arquitetura MiDaS *mobile*), o melhoramento de profundidade simples (descrito no Algoritmo 5), a execução do DQ-Mesh e da arquitetura LaMaSep. O modelo do MiDaS

	Moto Edge 20 +	Moto Edge 40 Ultra
Inferencia MiDaS	82.14 ms	40.29 ms
Melhoramento de Profundidade	78.62 ms	26.63 ms
DQ-Mesh	72.09 ms	30.05 ms
LaMaSep	742.74 ms	528.47 ms
Tempo Total	975.59 ms	625.44 ms

Tabela 2: Resultado das medições do tempo de execução em 2 dispositivos da Motorola. A inferência do MiDaS foi implementada em Tensorflow Lite executado em GPU. O melhoramento de profundidade e o DQ-Mesh foram implementado utilizando Halide e executados na CPU. A inferência do LaMaSep foi implementada em Tensorflow Lite com os operadores FFC customizados em Halide. Esse operadores são executados em CPU, enquanto os outros operadores do modelo LaMaSep são executados em GPU.

mobile pré-treinado é implementado utilizando Tensorflow Lite[1], está disponível no repositório do MiDaS [2] e tem dimensões 224×224 das imagens de entrada e saída. O melhoramento do mapa de profundidade (incluindo o redimensionamento da inferência do MiDaS para 640×640) e o DQ-Mesh foram implementados em Halide Ragan-Kelley *et al.* (2013), para melhor controlar a performance dos algoritmos de acordo com o *hardware* que será executado. O tamanho das dimensões dos blocos B_s utilizado foi $B_s = 16$, onde foram processados os triângulos do *foreground*. A arquitetura LaMaSep, por utilizar algumas camadas implementadas em Halide, tem algumas camadas executadas em GPU e outras em CPU. No caso do modelo do MiDaS *mobile*, todas as camadas são executadas em GPU. Observa-se que o *pipeline* completo consegue gerar malhas em menos de 1 segundo nos dois dispositivos, sendo o maior gargalo de tempo os modelos de mapa de profundidade e de *inpainting*.

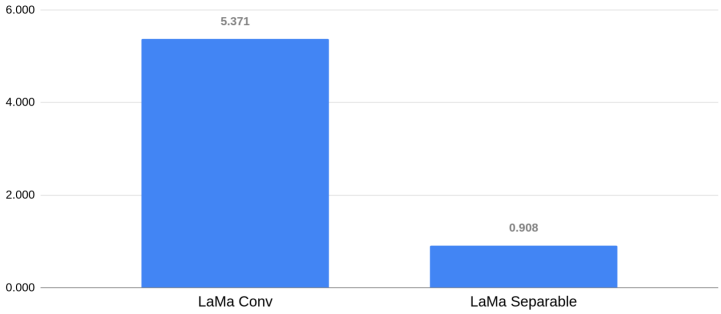


Figura 47: Comparação do tempo de inferência de um subconjunto de 150 imagens do conjunto HDR+Hasinoff *et al.* (2016) rodando por 30 vezes no dispositivo Moto Edge 20+. A primeira barra é a arquitetura LaMa convencional (LaMa Conv) e a segunda é a nossa implementação com operadores FFC em Halide e convoluções separáveis (LaMaSep). Como LaMa Conv é implementado com operadores que possuem suporte para GPUs, diferente da arquitetura LaMa Sep implementada, ambas as arquiteturas foram executadas em CPU. A escala dos valores mostrados é milissegundos

[1] `https://www.tensorflow.org/lite`
[2] `https://github.com/isl-org/MiDaS/tree/master/mobile`

No segundo experimento, foi comparado o ganho de performance da LaMaSep em relação a LaMa tradicional. Assim como os experimentos anteriores, utilizamos as 150 imagens retiradas do HDR+ como entrada. Como as arquiteturas utilizam imagens de tamanho 256×256, é aplicado um redimensionamento (interpolação bicúbica) nas imagens de entrada. O tempo mostrado na Figura 47 não inclui essa interpolação, que seria a mesma para ambas as arquiteturas. A primeira barra na Figura 47 mostra o tempo médio da inferencia em milissegundos calculado da arquitetura LaMa com convoluções tradicionais (rotulado como LaMa Conv). A segunda barra mostra o tempo médio em milissegundos da arquitetura LaMaSep (rotulada como LaMa Separable). Ambos os modelos foram construídos utilizando Tensorflow Lite. Como LaMa Conv possui operações que têm suporte para GPUs, para uma comparação justa, ambas as arquiteturas foram executadas na CPU do dispositivo Edge 20+. Assim, comparando os resultados, o modelo LaMa Conv leva em média 5.371 segundos para realizar a inferência enquanto LaMa Separable demora em média 908 milissegundos, uma diferença de $5.9\times$.

6

CONCLUSÃO

O problema da modelagem 3D de objetos ou cenários através de um conjunto de imagens possui uma diversidade de soluções com diferentes níveis de complexidade computacional. Os métodos tradicionais (anteriores a abordagens com arquiteturas CNN), necessitam de um ambiente controlado com o uso de câmeras sincronizadas e possuem um alto custo computacional. Como este trabalho está voltado para a execução dessas soluções em dispositivos móveis, essas limitações são ainda mais acentuadas devido a uma menor capacidade de processamento desses dispositivos.

O sucesso das redes neurais em visão computacional levou ao desenvolvimento de técnicas de reconstrução 3D utilizando essas redes. Assim, foram apresentados os conceitos que formam o campo da fotografia 3D (e.g. os métodos tradicionais para estimação de profundidade e as formas de representação) que inspiraram as abordagens recentes que utilizam aprendizagem profunda. Depois, são descritos os fundamentos teóricos que fundamentam as redes neurais e alguns trabalhos de importância histórica da área.

Analisando os trabalhos mais recentes voltados para fotografia 3D utilizando aprendizagem profunda, foram mostradas diferentes estratégias para acelerar e reduzir o custo da construção do modelo 3D. NeRFs são capazes de aprender características visuais e modelar objetos e cenas de maneira precisa. Porém, além de precisar de um grande número de imagens, é necessária a otimização dos parâmetros através de um treinamento para cada cena desejada. Isso resulta na inviabilidade desses métodos para uso comercial em dispositivos *mobile*. As abordagens que utilizam MPI conseguem contornar esse problema, porém podem se tornar custosas para uso em imagens com a resolução dos atuais aparelhos móveis e as abordagens com melhor qualidade visual podem precisar de um número considerável de planos, aumentando o custo de armazenamento.

Através da combinação de diferentes arquiteturas CNN para sub-problemas envolvendo o *pipeline* de modelagem (e.g. estimação de mapas de profundidade e preenchimento de cor), as soluções baseadas em LDI oferecem um caminho de implementação com melhor adaptação para dispositivos móveis. Isso se dá devido ao fato que as arquiteturas geralmente utilizadas são treinadas para problemas com validação em computadores com menor poder de processamento (e.g. estimação de mapas de profundidade e preenchimento de cor). Essa validação ocorre

porque existem arquiteturas específicas para esses dispositivos além de um foco na redução de parâmetros e tempo de inferência como o MiDaS *mobile*.

6.1 CONTRIBUIÇÕES

As abordagens utilizando LDI geralmente se baseiam na construção de malhas 3D conectando todos os *pixels* vizinhos do mapa de profundidade. Assim, existe um consumo de mémoria desnecessário ao concetar vértices com profundidade muito próxima. Isso resulta em representações que podem ser custosas se buscam modelar imagens de alta resolução sem perda de qualidade (mantendo as características de alta frequência da imagem de entrada). Pensando nisso, uma das contribuições trazidas no desenvolvimento do algoritmo DQ-Mesh é a construção de malhas através de um mapa de profundidade utilizando *quadtrees*.

DQ-Mesh permite que a malha final utilize a imagem de através do mapeamento de textura. Logo, não é necessário redimensionar o mapa de profundidade a depender da quantidade de informação de cor que pretende ser armazenada. Assim, conseguimos renderizar imagens com uma alta qualidade sem precisar gerar *meshes* com um número desnecessário de vértices. Além disso, utilizando mapeamento de textura podemos utilizar técnicas como *mipmap* para evitar efeito de *aliasing*, o que é mais difícil de mitigar nas abordagens MPI e se os vértices utilizarem diretamente a cor do *pixel* da imagem de entrada.

A arquitetura LaMa utilizada para predizer a cor das regiões ocluídas é originalmente proposta utilizando camadas convolucionais tradicionais. Nos experimentos utilizando LaMaSep verificamos que existe uma redução consistente do tempo de inferência da arquitetura em celulares sem uma perda significativa de qualidade.

Para ajustar os mapas de profundidade utilizados como entrada com o DQ-Mesh foi desenvolvido um método de melhoria dos mapas de profundidade utilizando mapas de segmentação. Os mapas estimados utilizando arquiteturas CNN podem ser imprecisos nos contornos, levando a separação de partes de um mesmo objeto. Nos experimentos observa-se que esse método de melhoria pode preencher essas partes faltantes a depender da precisão dos mapas de segmentação.

6.2 TRABALHOS FUTUROS

Apesar dos resultados mostrados no Capítulo 5, as análises apresentadas são insuficientes para concluir o impacto dos algoritmos propostos na qualidade da imagem renderizada. Ainda falta analisar métricas de qualidade em diferentes bancos de dados comparando com diferentes abordagens. Por exemplo, extrair parâmetros de câmera em vídeos da internet listados no banco de dados RealState10K Zhou *et al.* (2018b) utilizando SFM. Existe também o banco de dados apresentado em Garg *et al.* (2019), onde são retiradas várias imagens de uma cena utilizando câmeras de celulares sincronizados e já possui os parâmetros intrínsecos e extrínsecos de câmera.

As análises sobre o impacto do algoritmo de melhoria de profundidade também são insuficientes para concluir que há uma melhoria de fato. Além das arquiteturas para inferência de mapas de disparidade e segmentação utilizadas (MiDaS v3.0, MiDaS *mobile* e MobileSAM) é necessário comparar a melhoria proposta com mais arquiteturas utilizando os bancos de dados citados acima. Como a melhoria baseia-se em interpolar as profundidades dos pontos nas máscaras, podem ser desenvolvidos treinamentos de *distillation* (i.e. treinar redes com um menor número de parâmetros através de redes pré-treinadas maiores) buscando adequar a inferência do mapa de profundidade aos objetos delimitados pela segmentação. Além disso falta a adaptação para dispositivos móveis as arquiteturas treinadas para segmentação para que não influenciam tanto no *pipeline* de geração das malhas.

Também é necessário uma análise quantitativa sobre a qualidade do preenchimento de cor inferido por LaMaSep. Além disso, ao utilizar máscaras aleatórias no treinamento, o modelo aprende a completar o conteúdo faltante dos objetos onde são aplicadas as máscaras, fazendo com que preencha com informação do contexto de *foreground* na região do *background*. Utilizando a estratégia de geração de máscaras presente em Jampani *et al.* (2021), a arquitetura aprenderá a preencher as máscaras focando nas regiões de contexto do *background*. Apesar da redução de tempo em relação ao modelo LaMa original, nos experimentos a arquitetura LaMaSep ainda é o maior gargalo de tempo no *pipeline*. Para reduzir ainda mais esse tempo, é necessário implementar a portabilidade dos operadores em Halide para execução em GPUs. Essa ideia se aplica também ao DQ-Mesh, que pelo fato de também ser implementado em Halide, também pode ser portado para executar em GPU.

REFERÊNCIAS

Agarap, A. F. (2018). Deep learning using rectified linear units (relu). cite arxiv:1803.08375Comment: 7 pages, 11 figures, 9 tables.

Alhashim, I. & Wonka, P. (2018). High quality monocular depth estimation via transfer learning. *ArXiv*, abs/1812.11941.

Bai, L.-Q., Zhang, Y.-Z., Zhao, H., Song, C.-H., & Jing, W. (2010). A new linear-time component-labeling algorithm. In *2010 2nd International Conference on Software Technology and Engineering*, 1:V1–150–V1–153.

Ballester, C., Bertalmio, M., Caselles, V., Sapiro, G., & Verdera, J. (2001). Filling-in by joint interpolation of vector fields and gray levels. *IEEE Transactions on Image Processing*, 10(8):1200–1211.

Berg, M. d., Cheong, O., Kreveld, M. v., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition.

Bertalmio, M., Sapiro, G., Caselles, V., & Ballester, C. (2000). Image inpainting. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 417–424.

Canny, J. (1986). A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698.

Cauchy, A. (1847). Methode generale pour la resolution des systemes d'equations simultanees. *C.R. Acad. Sci. Paris*, 25:536–538.

Chen, C., Chen, Q., Xu, J., & Koltun, V. (2018). Learning to see in the dark. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Chen, S. E. (1995). Quicktime vr: an image-based approach to virtual environment navigation. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, 29–38.

Chen, Z., Funkhouser, T., Hedman, P., & Tagliasacchi, A. (2023). Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. In *The Conference on Computer Vision and Pattern Recognition (CVPR)*.

Cheng, J., Yang, Y., Tang, X., Xiong, N., Zhang, Y., & and, F. L. (2020). Generative adversarial networks: A literature review. *KSII Transactions on Internet and Information Systems*, 14(12):4625–4647.

Chi, L., Jiang, B., & Mu, Y. (2020). Fast fourier convolution. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., & Lin, H., editors, *Advances in Neural Information Processing Systems*, 33:4479–4488.

Chollet, F. (2017). Xception: Deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1800–1807.

Cun, X., Xu, F., Pun, C.-M., & Gao, H. (2018). Depth assisted full resolution network for single image-based view synthesis. In *ACM SIGGRAPH 2018 Posters*.

Debevec, P. E., Taylor, C. J., & Malik, J. (1996). Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, 11–20.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159.

Efros, A. & Leung, T. (1999). Texture synthesis by non-parametric sampling. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 2:1033–1038 vol.2.

Fridovich-Keil and Yu, Tancik, M., Chen, Q., Recht, B., & Kanazawa, A. (2022). Plenoxels: Radiance fields without neural networks. In *CVPR*.

Garbin, S. J., Kowalski, M., Johnson, M., Shotton, J., & Valentin, J. (2021). Fastnerf: High-fidelity neural rendering at 200fps.

Garg, R., Wadhwa, N., Ansari, S., & Barron, J. T. (2019). Learning single camera depth estimation using dual-pixels. *ICCV*.

Gomes, J., Velho, L., & Costa, M. (2012). *Computer GraphicsL Theory and Practice*. Taylor and Francis.

Gonzalez, R. C. & Woods, R. E. (2008). *Digital image processing*. Prentice Hall, Upper Saddle River, N.J.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, 2672–2680.

Goodfellow, I. J., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA, USA. `http://www.deeplearningbook.org`.

Gou, J., Yu, B., Maybank, S. J., & Tao, D. (2021). Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819.

Han, Y., Wang, R., & Yang, J. (2022). Single-view view synthesis in the wild with learned adaptive multiplane images. In *ACM SIGGRAPH 2022 Conference Proceedings*.

Hannah, M. J. (1974). Computer matching of areas in stereo images.

Harris, C. & Stephens, M. (1988). A combined corner and edge detector. In *Proceedings of the 4th Alvey Vision Conference*, 147–151.

Hasinoff, S. W., Sharlet, D., Geiss, R., Adams, A., Barron, J. T., Kainz, F., Chen, J., & Levoy, M. (2016). Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 35(6).

He, K., Gkioxari, G., Dollár, P., & Girshick, R. B. (2017). Mask r-cnn. In *ICCV*, 2980–2988.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

Hinton, G. E., Mcclelland, J. L., & Rumelhart, D. E. (1986). Distributed representations. In Rumelhart, D. E. & Mcclelland, J. L., editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, 77–109. MIT Press, Cambridge, MA.

Ho, J., Jain, A., & Abbeel, P. (2020). Denoising diffusion probabilistic models. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*.

Horry, Y., Anjyo, K.-I., & Arai, K. (1997). Tour into the picture: using a spidery mesh interface to make animation from a single image. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, 225–232.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861.

Huang, J.-B., Kang, S. B., Ahuja, N., & Kopf, J. (2014). Image completion using planar structure guidance. *ACM Trans. Graph.*, 33(4).

Jampani, V., Chang, H., Sargent, K., Kar, A., Tucker, R., Krainin, M., Kaeser, D., Freeman, W. T., Salesin, D., Curless, B., & Liu, C. (2021). Slide: Single image 3d photography with soft layering and depth-aware inpainting. In *Proceedings of the IEEE International Conference on Computer Vision*.

Karras, T., Laine, S., & Aila, T. (2021). A style-based generator architecture for generative adversarial networks. *IEEE Transactions on Pattern Analysis amp; Machine Intelligence*, 43(12):4217–4228.

Kazhdan, M., Bolitho, M., & Hoppe, H. (2006). Poisson surface reconstruction. In *Proceedings of the Fourth Eurographics Symposium on Geometry Processing*, 61–70.

Khan, N., Xiao, L., & Lanman, D. (2023a). Tiled multiplane images for practical 3d photography. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, 10420–10430.

Khan, N., Xiao, L., & Lanman, D. (2023b). Tiled multiplane images for practical 3d photography. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 10454–10464.

Kim, S. Y., Zhang, J., Niklaus, S., Fan, Y., Lin, Z., & Kim, M. (2022). Layered depth refinement with mask guidance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Kingma, D. & Ba, J. (2015). Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., Dollar, P., & Girshick, R. (2023). Segment anything. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 4015–4026.

Kopf, J., Matzen, K., Alsisan, S., Quigley, O., Ge, F., Chong, Y., Patterson, J., Frahm, J.-M., Wu, S., Yu, M., Zhang, P., He, Z., Vajda, P., Saraf, A., & Cohen, M. (2020). One shot 3d photography. *ACM Trans. Graph.*, 39(4).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., & Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, 1097–1105. Curran Associates, Inc.

Kurz, A., Neff, T., Lv, Z., Zollhöfer, M., & Steinberger, M. (2022). Adanerf: Adaptive sampling for real-time rendering of neural radiance fields.

Lecun, Y. (1985). Une procédure d'apprentissage pour réseau à seuil asymétrique. *Proceedings of Cognitiva 85, Paris*, 599–604.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Li, J., Feng, Z., She, Q., Ding, H., Wang, C., & Lee, G. H. (2021). Mine: Towards continuous depth mpi with nerf for novel view synthesis. In *ICCV*.

Li, Y., Yuan, G., Wen, Y., Hu, J., Evangelidis, G., Tulyakov, S., Wang, Y., & Ren, J. (2022). Efficientformer: Vision transformers at mobilenet speed. In Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., & Oh, A., editors, *Advances in Neural Information Processing Systems*, 35:12934–12949.

Liu, G., Reda, F. A., Shih, K. J., Wang, T.-C., Tao, A., & Catanzaro, B. (2018). Image inpainting for irregular holes using partial convolutions. In *Computer Vision – ECCV 2018: 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XI*, 89–105.

Liu, L., Gu, J., Lin, K. Z., Chua, T.-S., & Theobalt, C. (2020). Neural sparse voxel fields. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*.

Luo, A., Li, T., Zhang, W.-H., & Lee, T. S. (2021). Surfgen: Adversarial 3d shape synthesis with explicit surface discriminators. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 16238–16248.

Luvizon, D. C., Carvalho, G. S. P., dos Santos, A. A., Conceicao, J. S., Flores-Campana, J. L., Decker, L. G. L., Souza, M. R., Pedrini, H., Joia, A., & Penatti, O. A. B. (2021). Adaptive multiplane image generation from a single internet picture. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2556–2565.

Martin-Brualla, R., Radwan, N., Sajjadi, M. S. M., Barron, J. T., Dosovitskiy, A., & Duckworth, D. (2021). NeRF in the Wild: Neural Radiance Fields for Unconstrained Photo Collections. In *CVPR*.

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., & Ng, R. (2021). Nerf: Representing scenes as neural radiance fields for view synthesis. *Commun. ACM*, 65(1):99–106.

Niemeyer, M. & Geiger, A. (2021). Giraffe: Representing scenes as compositional generative neural feature fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 11453–11464.

Niklaus, S., Mai, L., Yang, J., & Liu, F. (2019). 3d ken burns effect from a single image. *ACM Trans. Graph.*, 38(6).

OpenAI, :, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., Avila, R., Babuschkin, I., Balaji, S., Balcom, V., Baltescu, P., Bao, H., Bavarian, M., Belgum, J., Bello, I., Berdine, J., Bernadett-Shapiro, G., Berner, C., Bogdonoff, L., Boiko, O., Boyd, M., Brakman, A.-L., Brockman, G., Brooks, T., Brundage, M., Button, K., Cai, T., Campbell, R., Cann, A., Carey, B., Carlson, C., Carmichael, R., Chan, B., Chang, C., Chantzis, F., Chen, D., Chen, S., Chen, R., Chen, J., Chen, M., Chess, B., Cho, C., Chu, C., Chung, H. W., Cummings, D., Currier, J., Dai, Y., Decareaux, C., Degry, T., Deutsch, N., Deville, D., Dhar, A., Dohan, D., Dowling, S., Dunning, S., Ecoffet, A., Eleti, A., Eloundou, T., Farhi, D., Fedus, L., Felix, N., Fishman, S. P., Forte, J., Fulford, I., Gao, L., Georges, E., Gibson, C., Goel, V., Gogineni, T., Goh, G., Gontijo-Lopes, R., Gordon, J., Grafstein, M., Gray, S., Greene, R., Gross, J., Gu, S. S., Guo, Y., Hallacy, C., Han, J., Harris, J., He, Y., Heaton, M., Heidecke, J., Hesse, C., Hickey, A., Hickey, W., Hoeschele, P., Houghton, B., Hsu, K., Hu, S., Hu, X., Huizinga, J., Jain, S., Jain, S., Jang, J., Jiang, A., Jiang, R., Jin, H., Jin, D., Jomoto, S., Jonn, B., Jun, H., Kaftan, T., Łukasz Kaiser, Kamali, A., Kanitscheider, I., Keskar, N. S., Khan, T., Kilpatrick, L., Kim, J. W., Kim, C., Kim, Y., Kirchner, H., Kiros, J., Knight, M., Kokotajlo, D., Łukasz Kondraciuk, Kondrich, A., Konstantinidis, A., Kosic, K., Krueger, G., Kuo, V., Lampe, M., Lan, I., Lee, T., Leike, J., Leung, J., Levy, D., Li, C. M., Lim, R., Lin, M., Lin, S., Litwin, M., Lopez, T., Lowe, R., Lue, P., Makanju, A., Malfacini, K., Manning, S., Markov, T., Markovski, Y., Martin, B., Mayer, K., Mayne, A., McGrew, B., McKinney, S. M., McLeavey, C., McMillan, P., McNeil, J., Medina, D., Mehta, A., Menick, J., Metz, L., Mishchenko, A., Mishkin, P., Monaco, V., Morikawa, E., Mossing, D., Mu, T., Murati, M., Murk, O., Mély, D., Nair, A., Nakano, R., Nayak, R., Neelakantan, A., Ngo, R., Noh, H., Ouyang, L., O'Keefe, C., Pachocki, J., Paino, A., Palermo, J., Pantuliano, A., Parascandolo, G., Parish, J., Parparita, E., Passos, A., Pavlov, M., Peng, A., Perelman, A., de Avila Belbute Peres, F., Petrov, M., de Oliveira Pinto, H. P., Michael, Pokorny, Pokrass, M., Pong, V., Powell, T., Power, A., Power, B., Proehl, E., Puri, R., Radford, A., Rae, J., Ramesh, A., Raymond, C., Real, F., Rimbach, K., Ross, C., Rotsted, B., Roussez, H., Ryder, N., Saltarelli, M., Sanders, T., Santurkar, S., Sastry, G., Schmidt, H., Schnurr, D., Schulman, J., Selsam, D., Sheppard, K., Sherbakov, T., Shieh, J., Shoker, S., Shyam, P., Sidor, S., Sigler, E., Simens, M., Sitkin, J., Slama, K., Sohl, I., Sokolowsky, B., Song, Y., Staudacher, N., Such, F. P., Summers, N., Sutskever, I., Tang, J., Tezak, N., Thompson, M., Tillet, P., Tootoonchian, A., Tseng, E., Tuggle, P., Turley, N., Tworek, J., Uribe, J. F. C., Vallone, A., Vijayvergiya, A., Voss, C., Wainwright, C., Wang, J. J., Wang, A., Wang, B., Ward, J., Wei, J., Weinmann, C., Welihinda, A., Welinder, P., Weng, J., Weng, L., Wiethoff, M., Willner, D., Winter, C., Wolrich, S., Wong, H., Workman, L., Wu, S., Wu, J., Wu, M., Xiao, K., Xu, T., Yoo, S., Yu, K., Yuan, Q., Zaremba, W., Zellers, R., Zhang, C., Zhang, M., Zhao, S., Zheng, T., Zhuang, J., Zhuk, W., & Zoph, B. (2023). Gpt-4 technical report.

Pfister, H., Zwicker, M., van Baar, J., & Gross, M. (2000). Surfels: surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, 335–342.

Pollefeys, M. & Gool, L. V. (2002). From images to 3d models. *Commun. ACM*, 45(7):50–55.

Pu, G., Wang, P.-S., & Lian, Z. (2023). Sinmpi: Novel view synthesis from a single image with expanded multiplane images.

Pérez-Enciso, M. & Zingaretti, L. M. (2019). A guide on deep learning for complex trait genomic prediction. *Genes*, 10(7).

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530.

Ranftl, R., Bochkovskiy, A., & Koltun, V. (2021). Vision transformers for dense prediction. *ICCV*.

Ranftl, R., Lasinger, K., Hafner, D., Schindler, K., & Koltun, V. (2022). Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(3).

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2015). You only look once: Unified, real-time object detection. cite arxiv:1506.02640.

Reiser, C., Peng, S., Liao, Y., & Geiger, A. (2021). Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. *CoRR*, abs/2103.13744.

Rematas, K., Nguyen, C. H., Ritschel, T., Fritz, M., & Tuytelaars, T. (2017). Novel views of objects from a single image. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(8):1576–1590.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2014). Imagenet large scale visual recognition challenge. cite arxiv:1409.0575Comment: 43 pages, 16 figures. v3 includes additional comparisons with PASCAL VOC (per-category comparisons in Table 3, distribution of localization difficulty in Fig 16), a list of queries used for obtaining object detection images (Appendix C), and some additional references.

Saharia, C., Chan, W., Saxena, S., Li, L., Whang, J., Denton, E., Ghasemipour, S. K. S., Gontijo-Lopes, R., Ayan, B. K., Salimans, T., Ho, J., Fleet, D. J., & Norouzi, M. (2022). Photorealistic text-to-image diffusion models with deep language understanding. In Oh, A. H., Agarwal, A., Belgrave, D., & Cho, K., editors, *Advances in Neural Information Processing Systems*.

Samavati, T. & Soryani, M. (2023). Deep learning-based 3d reconstruction: a survey. *Artif. Intell. Rev.*, 56(9):9175–9219.

Shade, J., Gortler, S., He, L.-w., & Szeliski, R. (1998). Layered depth images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, 231–242.

Shi, W., Caballero, J., Huszár, F., Totz, J., Aitken, A. P., Bishop, R., Rueckert, D., & Wang, Z. (2016). Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1874–1883.

Shih, M., Su, S., Kopf, J., & Huang, J. (2020). 3d photography using context-aware layered depth inpainting. *CoRR*, abs/2004.04727.

Simonyan, K. & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

Srinivasan, P. P., Tucker, R., Barron, J. T., Ramamoorthi, R., Ng, R., & Snavely, N. (2019). Pushing the boundaries of view extrapolation with multiplane images. *CoRR*, abs/1905.00413.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, III–1139–III–1147.

Suvorov, R., Logacheva, E., Mashikhin, A., Remizova, A., Ashukha, A., Silvestrov, A., Kong, N., Goka, H., Park, K., & Lempitsky, V. (2022). Resolution-robust large mask inpainting with fourier convolutions. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, 2149–2159.

Szeliski, R. (2011). *Computer vision algorithms and applications*. Springer, London; New York.

Szeliski, R. & Golland, P. (1998). Stereo matching with transparency and matting. In *Proceedings of the Sixth International Conference on Computer Vision*, 517.

Tan, M. & Le, Q. V. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In Chaudhuri, K. & Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, 97:6105–6114.

Tatarchenko, M., Dosovitskiy, A., & Brox, T. (2015). Single-view to multi-view: Reconstructing unseen views with a convolutional network. *CoRR*, abs/1511.06702.

Tucker, R. & Snavely, N. (2020). Single-view view synthesis with multiplane images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 6000–6010.

Werbos, P. J. (1981). Applications of advances in nonlinear sensitivity analysis. In *Proceedings of the 10th IFIP Conference, 31.8 - 4.9, NYC*, 762–770.

Wizadwongsa, S., Phongthawee, P., Yenphraphai, J., & Suwajanakorn, S. (2021). Nex: Real-time view synthesis with neural basis expansion. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Xian, K., Shen, C., Cao, Z., Lu, H., Xiao, Y., Li, R., & Luo, Z. (2018). Monocular relative depth perception with web stereo data supervision. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 311–320.

Xu, D., Jiang, Y., Wang, P., Fan, Z., Shi, H., & Wang, Z. (2022). Sinnerf: Training neural radiance fields on complex scenes from a single image.

Yu, A., Li, R., Tancik, M., Li, H., Ng, R., & Kanazawa, A. (2021a). PlenOctrees for real-time rendering of neural radiance fields. In *ICCV*.

Yu, A., Ye, V., Tancik, M., & Kanazawa, A. (2021b). pixelNeRF: Neural radiance fields from one or few images. In *CVPR*.

Zhang, C., Han, D., Qiao, Y., Kim, J. U., Bae, S.-H., Lee, S., & Hong, C. S. (2023a). Faster segment anything: Towards lightweight sam for mobile applications. *arXiv preprint arXiv:2306.14289*.

Zhang, K., Riegler, G., Snavely, N., & Koltun, V. (2020). Nerf++: Analyzing and improving neural radiance fields.

Zhang, W., Xing, R., Zeng, Y., Liu, Y.-S., Shi, K., & Han, Z. (2023b). Fast learning radiance fields by shooting much fewer rays. *IEEE Transactions on Image Processing*, 32:2703–2718.

Zhao, S., Cui, J., Sheng, Y., Dong, Y., Liang, X., Chang, E. I., & Xu, Y. (2021). Large scale image completion via co-modulated generative adversarial networks.

Zheng, H., Lin, Z., Lu, J., Cohen, S., Shechtman, E., Barnes, C., Zhang, J., Xu, N., Amirghodsi, S., & Luo, J. (2022). Image inpainting with & nbsp;cascaded modulation gan and & nbsp;object-aware training. In *Computer Vision – ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XVI*, 277–296.

Zhou, B., Lapedriza, A., Khosla, A., Oliva, A., & Torralba, A. (2018a). Places: A 10 million image database for scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(6):1452–1464.

Zhou, T., Tucker, R., Flynn, J., Fyffe, G., & Snavely, N. (2018b). Stereo magnification: learning view synthesis using multiplane images. *ACM Trans. Graph.*, 37(4).

Zhu, M., Pan, P., Chen, W., & Yang, Y. (2019). Dm-gan: Dynamic memory generative adversarial networks for text-to-image synthesis. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 5795–5803.