Neural Mesh Reconstruction

by

Zhiqin Chen

M.Sc., Simon Fraser University, Canada, 2019
B.Sc., Shanghai Jiao Tong University, China, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

Declaration of Committee

Name:	**Zhiqin Chen**
Degree:	**Doctor of Philosophy**
Thesis title:	**Neural Mesh Reconstruction**
Committee:	**Chair:** Ali Mahdavi-Amiri Assistant Professor, Computing Science
	Hao Zhang Supervisor Professor, Computing Science
	Yasutaka Furukawa Committee Member Associate Professor, Computing Science
	Andrea Tagliasacchi Examiner Associate Professor, Computing Science
	Matthias Nießner External Examiner Professor, Department of Computer Science Technical University of Munich

Abstract

Deep learning has revolutionized the field of 3D shape reconstruction, unlocking new possibilities and achieving superior performance compared to traditional methods. However, despite being the dominant 3D shape representation in real-world applications, polygon meshes have been severely underutilized as a representation for output shapes in neural 3D reconstruction methods. One key reason is that triangle tessellations are irregular, which poses challenges for generating them using neural networks. Therefore, it is imperative to develop algorithms that leverage the power of deep learning while generating output shapes in polygon mesh formats for seamless integration into real-world applications.

In this thesis, we propose several data-driven approaches to reconstruct explicit meshes from diverse types of input data, aiming to address this challenge. Drawing inspiration from classical data structures and algorithms in computer graphics, we develop representations to effectively represent meshes within neural networks.

First, we introduce BSP-Net. Inspired by a classical data structure Binary Space Partitioning (BSP), we represent a 3D shape as a union of convex primitives, where each convex primitive is obtained by intersecting half-spaces. This 3-layer BSP-tree representation allows a shape to be stored in a 3-layer multilayer perceptron (MLP) as a neural implicit, while an exact polygon mesh can be extracted from the MLP weights by parsing the underlying BSP-tree. BSP-Net is the first deep neural network that is able to produce compact and watertight polygon meshes natively, and the generated meshes are capable of representing sharp geometric features. We demonstrate its effectiveness in the task of single-view 3D reconstruction.

Next, we introduce a series of works that reconstruct explicit meshes by storing meshes in regular grid structures. We present Neural Marching Cubes (NMC), a data-driven algorithm for reconstructing meshes from discretized implicit fields. NMC is built upon Marching Cubes (MC), but it learns the vertex positions and local mesh topologies from example training meshes, thereby avoiding topological errors and achieving better reconstruction of geometric features, especially sharp features such as edges and corners, compared to MC and its variants. In our subsequent work, Neural Dual Contouring (NDC), we replace the MC meshing algorithm with Dual Contouring (DC) with slight modifications, so that our algorithm can reconstruct meshes from both signed inputs, such as signed distance fields or binary voxels, and unsigned inputs, such as unsigned distance fields or point clouds, with high accuracy and fast inference speed in a unified framework. Furthermore, inspired by the volume rendering algorithm in Neural Radiance Fields (NeRF), we introduce differentiable rendering to NDC to arrive at MobileNeRF, a NeRF-based method for reconstructing objects and scenes as triangle meshes with view-dependent textures from multi-view images. MobileNeRF is the first NeRF-based method that is able to run on mobile phones and AR/VR platforms thanks to the explicit mesh representation, demonstrating its efficiency and compatibility on common devices.

Keywords: 3D shape, mesh, and representation; 3D reconstruction from voxels, point clouds, and images; machine learning

Acknowledgements

I would like to express my graditude to my supervisor, Hao (Richard) Zhang, who accepted me as his student when other professors turned me down due to my poor GPA and unimpressive research record. Over the past 6 years, he has given me complete autonomy to pursue my research interests while providing invaluable support in various ways to bring my ideas to fruition. His mentorship has been instrumental in my academic success. Thanks to him, I had a productive, and mostly importantly, happy, graduate experience.

I am also grateful to Zili Yi for involving me in his research project, which was also the first project I participated in the lab at SFU. It was through this experience that I learned how real research was done. It served as a stepping stone for me to enter this field and has a profound impact on my research career.

My sincere appreciation also goes to Kangxue Yin, with whom I have collaborated multiple times, both as fellow students at SFU and during my internship at NVIDIA. I have benefited greatly from his help and learned valuable skills from our collaborations.

I would like to thank Chenyang Zhu, Renjiao Yi, Rui Ma, Changqing Zou, Wallace Pinto Lira, Akshay Gadi Patil, Luwei Yang, Xiang Xu, Manyi Li, Zeshi Yang, Fenggen Yu, Hang Zhou, Qimin Chen, and my other friends and colleagues in the lab. They have made my time at SFU truly wonderful.

I would also like to thank my collaborators, and supervisors and colleagues during my internships: Daniel Cohen-Or and Hui Huang for collaborating on project LOGAN; Siddhartha Chaudhuri, Vladimir Kim, Matthew Fisher, and Noam Aigerman from Adobe; Sanja Fidler and Jun Gao from NVIDIA; Andrea Tagliasacchi, Thomas Funkhouser, and Peter Hedman from Google. I enjoyed the discussions with them on research, presentation, work, life, etc, and I have learned immensely from them.

I want to thank Yasutaka Furukawa for being my supervisory committee member, Andrea Tagliasacchi for serving as my examiner, Matthias Nießner for serving as my external examiner, and Ali Mahdavi-Amiri for chairing my PhD defense. I appreciate their time, effort, and insightful comments.

Finally, I am deeply appreciative of my parents for their love and support throughout my academic journey.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Deep learning has been a game changer in computer science. During the past decade, deep learning methods have evolved from experimental tools in research labs into real life applications such as those in autonomous driving, recommendation systems, object and action recognition, machine translation, speech recognition, generative AI for content creation, and many others. These deep learning methods have profound impact in our lives. One recent example is AI-generated art achieved by large text-to-image models, such as Stable Diffusion [206], which have made creating art accessible to general public and have changed the workflow of many artists. Another prominent example is AI chatbots, such as ChatGPT [183], which are powerful productivity tools that can help people with various tasks ranging from summarizing and drafting documents to coding and debugging.

The success of these deep learning models can be attributed, in part, to their well-established data representations. For instance, images are represented as 2D arrays of pixels, where convolutional neural networks (CNN) have been developed to process and generate them; texts are represented as sequences of words, where sequence models such as recurrent neural networks (RNN) and Transformers [246] can be applied. These representations have been widely accepted in both academia and industry. Therefore, both the software (neural network architectures and algorithms) and the hardware (AI chips and Tensor Processing Units) have been designed and optimized to handle these types of data, accelerating the adoption of deep learning methods in real life.

However, there is no standard data representation for 3D shapes in deep learning models. Various representations have been proposed for neural networks to generate 3D outputs, such as point clouds, voxels, and neural implicit. Point clouds have been a popular representation as the input to neural networks [195, 196, 256], but they may not be a desirable representation as the output, since sparse point clouds cannot be used in realistic rendering, and dense point clouds have high space and computational complexity to be generated by neural networks. Voxels also have the issue of high complexity, even when adaptive spatial structures such as octrees are used [43, 92, 236]. Neural implicit representation [38, 163, 185] was designed to provide a compact way to represent 3D shapes as continuous implicit fields stored in neural networks. It guarantees to produce watertight shapes with manifold surfaces, which is ideal for rendering. And unlike point clouds and voxels where the neural networks have to generate the whole output shape in a single forwarding pass, models with the neural implicit representation can naturally break a shape into small mini-batches for training and inference, making the representation highly scalable. The various advantages of neural implicit make it quicky become the most adopted representation in 3D reconstruction and generative models.

Unfortunately, these popular representations are not suitable for most real-world applications, as polygon meshes are the dominant 3D representation in the industry. Their dominance is reflected in two aspects. First, polygon meshes are the standard representation used by artists to create 3D models. The entire 3D content creation pipeline revolves around polygon meshes, from modeling to texturing (UV mapping) to animation (rigging). Second, polygon meshes are the only 3D representation that GPUs (Graphics Processing Units) accept and render. Modern GPUs have been specifically designed and have gone through heavy optimizations to perform a single type of rendering: on polygons. Thus, even in the case when 3D models are initially created using other methods, such as 3D sculpting or particle simulation, they still need to be converted into meshes for rendering purposes.

Despite the dominance of polygon meshes in real-world applications, they are rarely used as a representation for output shapes in neural networks. The primary reason is that polygon meshes are essentially a special kind of graphs, with arbitrary numbers of vertices and faces that need to be connected in specific ways to form manifold surfaces. In contrast, neural networks are typically constrained to generate fixed-length outputs in a single forwarding pass. As a result, treating meshes as graphs requires the networks to be recurrent and to generate vertices and faces one-by-one. This is neither efficient nor practical, as networks tend to overfit and/or underfit when no inductive bias is present.

Indeed, neural networks do not have to produce meshes natively; they can simply convert their output representations into polygon meshes whenever necessary. However, this raises two issues. First, there needs to be algorithms that reconstruct polygon meshes from other representations. Classic model-driven mesh reconstruction algorithms were built on heuristic assumptions that may not always hold in real-world scenarios, resulting in a range of issues and reduced reconstruction quality. It is expected that data-driven mesh reconstruction algorithms that can learn from shapes in the real world should outperform their traditional counterparts and exhibit improved generalizability, but this ironically leads us back to the question of how to generate meshes with neural networks. Second, mesh reconstruction is a post-processing step applied to the outputs of the neural networks, which is detached from the training phase of these deep learning models. As a result, the networks have no control over the quality and the compactness of the final reconstructed mesh, leading to quantization errors and excessive vertices and triangles in the output mesh. In contrast, producing meshes natively allows the neural networks to be aware of the errors in the final output mesh so that they can be compensated during training, thereby achieving superior quality with fewer mesh elements.

Therefore, deep learning models that produce polygon meshes are not only desirable models in real applications, but also powerful tools for converting other 3D representations into usable formats. Today, creating 3D content still requires significant professional knowledge, and only individuals who have undergone extensive training and practice are capable of modeling 3D shapes and scenes in specialized software. We are in urgent need for deep learning-powered mesh creation tools that can enhance the robustness and user-friendliness of 3D modeling, making it accessible to a broader audience.

In this thesis, we tackle neural mesh reconstruction specifically, where 3D polygon meshes are reconstructed from given inputs by utilizing neural networks that learn priors from training shapes. We propose several representations that draw inspiration from classical data structures and algorithms in computer graphics, to effectively represent meshes within neural networks. And by developing a series of deep learning methods that reconstruct meshes from voxels, point clouds, single images, and multi-view images, we aim to advance the research in mesh-based representation learning, improve the robustness and performance of mesh reconstruction methods, and ultimately, make 3D content creation accessible to everyone.

1.2 Challenges

The major challenge and the focus of this thesis is how explicit meshes can be represented in neural networks. An accompanying challenge is that the original tessellations in the training meshes are usually not the desirable ground truth for the output meshes and cannot be used for direct supervision. Consequently, networks have to be trained to learn mesh tessellations using either pseudo ground truth or no ground truth.

1.2.1 Representing explicit meshes in neural networks

It goes without saying that representing explicit meshes in neural networks is a challenging task. There were attempts [50, 174] that treat a mesh as a graph and generate vertices and faces sequentially, which led to overcomplex network design, poor output quality, and overfitting. A considerable amount of works adopt deformation-based approaches that deform a sphere [249, 115, 152, 31, 299, 171, 292], a set of cuboids [74, 73], or a set of 2D square patches [86, 260, 8] into the target shape. But they are either limited to a fixed topology or having noticeable artifacts such as seams between patches.

Some works aim at shape abstraction, e.g., representing a shape with a sparse set of boxes [239] or superquadrics [187]. Their generated meshes are usually of poor quality due to their simple primitive types. Using convex primitives to construct shapes has the potential to significantly improve shape quality and compactness. Yet, representing general and compact convex primitives with neural networks is highly non-trivial. To tackle this, we draw inspiration from a classical spatial data structure in computer graphics, Binary Space Partitioning (BSP), to devise BSP-Net [36], a network that learns to represent a 3D shape using a set of convex primitives obtained from a BSP-tree built on a set of planes.

Some other works represent meshes in regular grid structures [139, 71], so that convolutional operations can be applied in their networks for efficient learning. However, their algorithms are unable to generate surface details due to the limited representation ability of their mesh tessellations, which are naïvely adopted from classic methods such as Marching Cubes [157]. Therefore, we not only enhance the mesh tessellations in classic methods, allowing Marching Cubes [157] to reconstruct sharp geometric features and Dual Contouring [110] to reconstruct thin sheets and non-orientable surfaces, but also parameterize the tessellations in both algorithms, enabling them to be coupled with deep neural networks for accurate mesh reconstruction from various input types, resulting in Neural Marching Cubes (NMC) [39] and Neural Dual Contouring (NDC) [35]. Moreover, in MobileNeRF [33], we show that textures with transparencies can be used for representing geometric details on coarse meshes, which is especially beneficial in scene reconstruction tasks with differentiable rendering.

1.2.2 Learning mesh tessellations with pseudo or no ground truth

Mesh tessellations can be highly ambiguous, in the sense that the exact same geometry can have limitless possible tessellations. As a consequence, the original tessellations in the ground truth meshes may not be optimal or even representative. On the other hand, the ground truth tessellations used for training must be compatible with neural networks. In other words, the tessellations are only useful when the networks can generate them.

To learn mesh tessellations without ground truth, grid-based mesh reconstruction methods [139] typically sample points on the predicted mesh and the ground truth mesh, and minimize points-to-points or points-to-mesh distances, thereby learning the optimal tessellations by minimizing the mesh reconstruction error. However, this training strategy is not suitable for reconstructing highly detailed shapes, as it requires sampling dense point clouds to capture the details on those shapes, and computing distances between dense point clouds is prohibitively expensive. Therefore, in NMC and NDC where our focus is to reconstruct geometric details, we develop pseudo ground truth tessellations by performing a re-meshing on training meshes, to convert the meshes into the formats that our networks can generate, so that the training can be done efficiently in a fully supervised manner. In MobileNeRF where no ground truth is available, we simply adopt a fixed grid topology. We rely on differentiable rendering to adjust the mesh vertices to approximate the shape, and rely on the opacity in mesh textures to change the mesh topology.

To generate more compact meshes, regular grid structures cannot be relied on since they intrinsically generate dense and uniformly-distributed mesh elements. The networks will need to learn to tessellate shapes adaptively, with more polygons on detailed regions and fewer on featureless regions. Thus, in BSP-Net, we represent an explicit Constructive Solid Geometry (CSG) tree with a neural implicit representation. By training the neural network to learn implicit functions, no explicit mesh tessellation is required during training, and the network learns to allocate primitives adaptively and automatically. When an explicit mesh is needed, we parse the underlying CSG-tree to produce mesh tessellations.

1.3 Contributions

In this thesis, we introduce several deep learning algorithms that reconstruct meshes from various input sources, including voxels, point clouds, single images, and multi-view images. Specifically, We propose the first deep generative network that directly outputs compact and watertight polygon meshes with arbitrary topology, the first data-driven iso-surfacing algorithm that is able to recover sharp geometric features from discretized implicit fields, the first unified and generalizable mesh reconstruction framework that can accommodate multiple input types, and the first NeRF-based novel-view synthesis method that leverages meshes as its representation for fast rendering on mobile devices.

1.3.1 Generating compact meshes with neural networks

Following the introduction of the neural implicit representation by DeepSDF [185], OccNet [163], and our IM-Net [38], there has been a surge in adopting neural implicit in various applications. However, despite being a compact representation, neural implicit does not produce compact meshes, as it requires an iso-surfacing step to extract the meshes from the fields. Coincidentally, during the development of BAE-Net [37] where our original goal was to apply neural implicit in unsupervised shape co-segmentation, we observed that a shallow MultiLayer Perceptron (MLP) exhibited a certain tree structure akin to a classical BSP-tree, suggesting a potential way to represent polygon meshes in MLPs.

Therefore, with the aim of generating compact meshes, we present BSP-Net, a shape decoder neural network based on a neural BSP-tree to decode a latent code into a 3D shape and output polygon meshes natively. BSP-Net is essentially a neural implicit representation, with an MLP to classify whether a given query 3D point is inside or outside the output shape. However, its MLP architecture is reformulated so that the first layer of the neural network explicitly represents multiple planes that fit the surfaces of a 3D shape. Each plane implies a Binary Space Partition, and the output 3D shape is composed by combining the half-spaces from the binary space partitions. This composition is performed explicitly in the second and third layers of the MLP via binarized network weights, where the second layer groups planes into convex parts, and the third layer groups convex parts into the output

3D shape. Therefore, at inference time, the BSP-tree can be explicitly constructed from the network weights, and an explicit polygon mesh can be extracted by parsing the BSP-tree using classic CSG.

We adopt an encoder-decoder setting to first encode the input into a global shape latent code, and then use BSP-Net to decode it into a polygon mesh. In principle, this framework can reconstruct meshes from all types of inputs as long as they can be encoded into a shape latent code, although we only explore voxel and single image inputs in our experiments. We also demonstrate the strengths of BSP-Net through extensive testing on shape segmentation and part correspondence. Our contributions are summarized as follows.

- BSP-Net is the first deep generative network to directly output compact and watertight polygon meshes with arbitrary topology and structure variety.
- BSP-Net is also the first deep generative network that can reconstruct and recover sharp geometric features.
- The learned BSP-tree enables us to infer both shape segmentation and part correspondence in an unsupervised manner.

1.3.2 Data-driven iso-surfacing algorithm

We then seek to extend the capabilities of another classic algorithm in computer graphics, Marching Cubes (MC), through deep learning techniques. Due to their model-driven design, classical MC and its variants are unable to reconstruct geometric features that reveal coherence or dependencies between nearby cubes, such as sharp edges. To boost the performance of MC algorithms, we introduce Neural Marching Cubes (NMC), a data-driven approach for extracting a triangle mesh from a discretized implicit field. In NMC, we re-cast MC from a deep learning perspective, by designing tessellation templates more apt at preserving geometric features, and learning the vertex positions and mesh topologies from training meshes, to account for contextual information from nearby cubes. We develop a compact per-cube parameterization to represent the output triangle mesh, while being compatible with neural processing, so that a simple 3D convolutional network can be employed for the training. In addition, our network learns local features with limited receptive fields, hence it generalizes well to new shapes and new datasets.

We evaluate our NMC approach by extensive comparisons to all well-known MC variants. In particular, we demonstrate the ability of our network to recover sharp features such as edges and corners, and also reconstruct local mesh topologies more accurately than previous approaches. Our contributions are summarized as follows.

- NMC is the first data-driven iso-surfacing algorithm and it is also the first iso-surfacing algorithm able to recover sharp features without requiring additional inputs other than a uniform grid of implicit field values.

- NMC can more faithfully reconstruct local mesh topologies near thin shape structures and closeby surface sheets.
- NMC can be trained to reconstruct clean meshes from noisy inputs by adjusting the training data, thus offering a useful tool for extracting 3D shapes from those shape representations designed for neural networks.

1.3.3 Unified mesh reconstruction framework

Following NMC, we introduce Neural Dual Contouring (NDC), a unified data-driven approach that learns to reconstruct meshes from a variety of inputs, including signed or unsigned distance fields, binary voxels, non-oriented point clouds, and noisy raw scans. NDC generalizes to a broad range of shape types, including CAD models with sharp edges, organic shapes, open surfaces for cloths, scans of indoor scenes, and even non-orientable surfaces.

NDC is based on Dual Contouring (DC). Like traditional DC, it produces exactly one vertex per grid cell and one quad for each grid edge intersection, a natural and efficient structure for reproducing sharp features. However, rather than computing vertex locations and edge crossings with hand-crafted functions that depend directly on difficult-to-obtain surface gradients, NDC uses a neural network to predict them. As a result, NDC can be trained to produce meshes from all kinds of input types as long as they can be converted into a grid structure, and it can produce open surfaces in cases where the input represents a sheet or partial surface.

We show in the experiments that NDC not only has strong generalizability to new shapes and new datasets, but also provides better surface reconstruction accuracy, feature preservation, output complexity, triangle quality, and inference time, compared to previous learned and traditional methods. Our contributions are summarized as follows.

- NDC is the first data-driven approach to mesh reconstruction based on Dual Contouring. It eliminates the need for gradients in the input as in classical DC, and it accounts for local contextual information from nearby cubes.
- NDC is a unified learning model that is applicable to a larger variety of inputs than previous meshing methods. The allowed inputs include signed/unsigned distance fields, binary voxels, and un-oriented point clouds.
- Compared to the previous data-driven iso-surfacing algorithm NMC, NDC has a significant reduction in representational complexity, which translates to across-the-board gains, in terms of simplicity of the network architecture, reduction in network capacity, training and inference times, and more.
- UNDC (unsigned NDC), the sign-agnostic version of NDC, can produce open, even non-orientable, output surfaces. It can recover thin structures thinner than one voxel in the input grid, which no other iso-surfacing algorithms can reconstruct.

1.3.4 Real-time NeRF based on textured polygon meshes

Neural Radiance Fields (NeRF) [167] have demonstrated amazing ability to synthesize images of 3D scenes from novel views when trained on input multi-view images. However, they are typically slow and incompatible with common devices due to their specialized volumetric rendering algorithms. To build fast and compatible NeRF models, we introduce MobileNeRF, a NeRF that can run on a variety of common mobile devices in real time. In MobileNeRF, the NeRF is represented by a set of textured polygons, where the polygons roughly follow the surface of the scene, and the texture atlas stores opacity and feature vectors. To render an image, we utilize the classic polygon rasterization pipeline with Z-buffering to produce a feature vector for each pixel and pass it to a lightweight MLP running in a GLSL fragment shader to produce the view-dependent output color. This rendering pipeline takes full advantage of the parallelism provided by z-buffers and fragment shaders in modern graphics hardware, and thus is much faster than the prior state-of-the-art. Moreover, it requires only a standard polygon rendering pipeline, which is implemented and accelerated on virtually every computing platform, and thus it runs on mobile phones and other devices previously unable to support NeRF visualization at interactive rates.

Our contributions are summarized as follows.

- MobileNeRF is 10$\times$ faster than the prior state-of-the-art (SNeRG [97]), with the same output quality.
- MobileNeRF consumes less GPU memory by storing surface textures instead of volumetric textures, enabling it to run on integrated GPUs with limited memory and power.
- MobileNeRF runs on a web browser and is compatible with all devices we have tested. It is also the first NeRF-based method that is able to run on mobile phones and AR/VR platforms.
- MobileNeRF allows real-time manipulation of the reconstructed objects/scenes, as they are simple triangle meshes.

1.4 Thesis Organization

The thesis is organized as follows. In Chapter 2, we will discuss various 3D representations used in neural networks and review deep-learning mesh reconstruction methods. In Chapter 3, we will introduce BSP-Net for generating compact meshes with neural networks via binary space partitioning. In Chapter 4, we will introduce Neural Marching Cubes (NMC) for reconstructing meshes from implicit fields by learning from example training meshes. In Chapter 5, we will introduce Neural Dual Contouring (NDC), a unified data-driven approach that can reconstruct meshes from all common input types. In Chapter 6, we will introduce MobileNeRF for efficient neural field rendering on mobile architectures by representing NeRF in textured polygon meshes. In Chapter 7, we will provide conclusions and discuss future works.

Related publications This thesis includes previously published material. The following is a list of the papers and their corresponding chapters:

- Chapter 2 appeared in the paper “A Review of Deep Learning-Powered Mesh Reconstruction Methods”. Zhiqin Chen. ArXiv preprint arXiv:2303.02879, 2023.
- Chapter 3 appeared in the paper “BSP-Net: Generating compact meshes via binary space partitioning”. Zhiqin Chen, Andrea Tagliasacchi, and Hao Zhang. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 45–54, 2020.
- Chapter 4 appeared in the paper “Neural Marching Cubes”. Zhiqin Chen and Hao Zhang. ACM Transactions on Graphics (Special Issue of SIGGRAPH Asia), 40(6), 2021.
- Chapter 5 appeared in the paper “Neural Dual Contouring”. Zhiqin Chen, Andrea Tagliasacchi, Thomas Funkhouser, and Hao Zhang. ACM Transactions on Graphics (Special Issue of SIGGRAPH), 41(4), 2022.
- Chapter 6 appeared in the paper “MobileNeRF: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures”. Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023.

Chapter 2

Background

In this chapter, we provide background and works related to this thesis. First, we describe various 3D representations used in neural 3D reconstruction and generative models in Section 2.1, to give an overview of how neural networks can output 3D meshes. Then we provide a comprehensive review of deep-learning mesh reconstruction methods categorized by their input, specifically, mesh reconstruction from voxels in Section 2.2, from point clouds in Section 2.3, from single images in Section 2.4, and from multi-view images in Section 2.5.

2.1 3D Representations for Neural Networks

The foundation for any algorithm is the data representation. Unfortunately, there is no such unified representation for 3D models. In fact, researchers have proposed a wide range of representations for 3D generative tasks.

In this section, we focus on representations that are essentially triangle meshes, therefore **point clouds**, although being a very popular representation, will not be discussed. We also consider representations that can be easily converted into triangle meshes, such as CSG (Constructive Solid Geometry) trees, parametric surfaces, voxels, and neural implicit.

2.1.1 Deformation-based

Deforming a single template mesh to create shapes of different poses has been widely used for face [58] and human body [124, 18, 190] shape reconstruction. However, due to the availability of a single template mesh, this representation is not suitable for reconstructing general 3D objects.

Extending the idea of deforming a single shape, one can first retrieve a most suitable template for the target object, and then deform the template, to achieve optimal performance. The methods usually process the inputs with neural networks to either classify which template is most suitable, such as BCNet [105] and Multi-Garment Net (MGN) [17]; or embed the input into a deformation-aware latent space to retrieve the nearest-neighbor template, such as 3D Deformation Network (3DN) [255], Deformation-Aware 3D Model Embedding and Retrieval [242], and ShapeFlow [106]. Then, a deformation network is applied to deform the template into the target shape.

The above representations all assume high-quality shape templates are given, and they are most used for highly specialized reconstruction tasks. For general shape reconstruction tasks, especially when reconstructing 3D shapes with only 2D image supervision, a primitive shape could be used as the initial shape, and it can be deformed to approximate the target shape. The most commonly used primitive is a simple sphere. Representative works that are trained with ground truth 3D supervision include AtlasNet (sphere version) [86], Pixel2Mesh [249], Pixel2Mesh++ [258], Neural mesh flow [90], and [184]. It may not be the most popular representation for models trained with 3D supervision, but it indeed has been the dominant representation in models trained with only 2D image supervision, such as N3MR (Neural 3d Mesh Renderer) [115], Soft Rasterizer [152], DIB-R [31], DIB-R++ [32], Image GANs meet Differentiable Rendering [299], UNICORN [171], NeRS [292], and [98, 191, 138]. There are multiple ways to perform deformation on the sphere: it can be an MLP (MultiLayer perceptron) that takes a 3D point on the sphere surface and outputs its deformed coordinates [86, 184, 171, 292]; or using graph convolutional networks on the sphere meshes to predict vertex positions [249, 258]; or using a CNN decoder to predict a displacement map on the sphere [191]; the majority of the methods simply use an MLP to predict the offsets of all vertices in the template mesh, outputting a $V \times 3$ vector where V is the number of vertices in the template. Some works on 3D reconstruction with 2D supervision adopt a better initialization than a simple sphere. CMR [111] initializes the template shape as the convex hull of the mean keypoint locations obtained after running SFM (structure from motion) on the annotated keypoints. [113] initializes the template shape using the visual hull of the annotated object masks on the training images. [81] initializes the template shape using a very simple but manually designed mesh. Note that for these works, the template meshes are optimizable during training.

One can also consider deforming a uniform 2D grid as deforming one primitive, with the primitive being a square. Therefore, many works that predict depth images from a single image can be put into this representation category, e.g., “Unsupervised learning of probably symmetric deformable 3d objects from images in the wild” [269]. Similarly, there are models that predict geometry images [87], such as SurfNet [226].

Deforming a single primitive shape limits the topology and the representation ability of the reconstructed shapes, therefore a natural solution is to fit multiple primitive shapes. However, this representation is mostly used when direct 3D supervision is available. Representative works include AtlasNet (patch version) [86], where an output shape is represented as a collection of square meshes, each deformed via an independent MLP. Photometric Mesh Optimization [141] adopts the 25-patch version of AtlasNet to reconstruct an object mesh from multi-view images. Follow-up works fit square meshes into local patches of a shape to perform mesh reconstruction from point clouds, such as Deep geometric prior [260] and

Meshlet priors [8]. The square patches in AtlasNet lead to many issues, including overlapping patches, self-intersections, and conspicuous artifacts on the surface. SDM-NET [74] has output shapes made of deformed unit cube meshes; it addresses the above issues via strong part-wise 3D supervision. Another work [228] applies a deformable parametric template composed of Coons patches [47] to generate manifold and piecewise-smooth shapes made of parametric surfaces.

2.1.2 Set of primitives

A shape can also be approximated by a set of primitive meshes, where the network only needs to predict the parameters for each primitive, in contrast to utilizing deformation networks as those deformation-based representations. A commonly used primitive type is bounding boxes or boxes that approximate the shape, such as those in Im2struct [180] and VP [239]. Boxes are easy to define: the network only needs to predict the size, the translation, the rotation, and the existence probability of each primitive box. An explicit mesh can also be easily obtained by a union of the box meshes.

Follow-up works such as SQ [187] and Hierarchical SQ [186] use superquadrics instead of boxes for the primitives. Superquadrics [10] are a parametric family of surfaces that can be used to describe cubes, cylinders, spheres, octahedra, ellipsoids, and other simple primitives, with just two shape parameters. Therefore, using superquadrics can improve the representation ability to better approximate the target shape. To generate superquadrics, the network needs to predict the shape parameters, the size, the translation, the rotation, and the existence probability of each superquadric primitive. Meshes also can be easily obtained from superquadrics.

Convex primitives proposed in CvxNet [57] and our work BSP-Net [36] are more compact than superquadrics, and also more flexible as their primitive can model any convex shape. Unlike boxes or superquadrics, convex primitives cannot be described by a few parameters. Therefore, the methods require intricate neural network design: the network first predicts a set of planes or half-spaces, and then the network intersects selected half-spaces to form convex primitives, and finally the convex primitives are united to form the output shape. Polygonal meshes can be directly extracted from the tree structure.

To make the primitives more flexible, Neural Star Domain [116] represents each primitive as a star domain. One can consider a star domain as a "height" function defined on the surface of a unit sphere, therefore a star domain can be approximately represented as the coefficients of spherical harmonics. So to generate those primitives, the network needs to predict a few spherical harmonics coefficients for each primitive, plus the translation vector. Meshes can be extracted by deforming a unit-sphere template mesh with respect to the "height" function.

2.1.3 Constructive solid geometry

Constructive Solid Geometry (CSG) is a common representation for CAD (computer-aided design) models, where primitive shapes such as polyhedrons, ellipsoids, and cylinders are merged via boolean operators such as union, intersection, and difference. CSG representation is a compact representation that is able to preserve sharp and smooth geometric features.

When ground truth CSG-trees are available, a network can be trained with fully-supervised learning to produce a sequence of operations and their operands, where the sequence is equivalent to a CSG-tree. A representative work is CSGNet [218].

Yet when ground truth is unavailable, learning to represent a shape as a CSG-tree with neural networks is a very challenging task, because CSG trees include selecting suitable primitives and performing CSG operations. Both steps are discrete and non-differentiable. Our work BSP-Net [36] from the previous section could also be considered as using CSG representation, since it intersects half-spaces to form convex shapes and then unions convex shapes to form the output shape. BSP-Net first learns a continuous approximation of the CSG-tree, and then discretizes it to obtain a real CSG-tree. However, BSP-Net only adopts planes (half-spaces) as primitives, therefore it only approximates piece-wise planar shapes. A follow-up work CAPRI-Net [289] relaxed the primitive types to axis-aligned quadratic surfaces, therefore it is able to represent curved smooth surfaces with a single primitive. In addition, CAPRI-Net also introduced a differentiable difference operator. The predicted quadratic surface primitives are first selected and intersected to form convex shapes; then the convex shapes are unioned to form two potentially concave shapes; finally the difference between the two concave shapes forms the output shape.

While CAPRI-Net uses quadratic surfaces as primitives, other works use primitive shapes as primitives. UCSG-Net [112] predicts parameters of boxes and spheres and uses them as primitives. Similarly CSG-Stump [202] predicts boxes, spheres, cylinders and cones. While UCSG-Net has several CSG layers where each layer supports multiple types of CSG operations; CSG-Stump only has three layers with fixed ordering (complement-intersection-union), similar to CAPRI-Net (intersection-union-difference).

2.1.4 Sketch and extrude

"Sketch and extrude" is also a CSG representation. However, different from the representation described in the previous section where primitive 3D shapes are used for CSG operations, this representation takes an approach more similar to the workflow of creating CAD models, by repeatedly sketching a 2D profile and then extruding that profile into a 3D body. DeepCAD [267] is a pioneer in generating CAD models with such a representation, by applying a Transformer network [246] to predict the command sequences for creating the output shape. Follow-up work [126] reconstructs CAD shapes from rounded voxel models; it represents profiles as 2D occupancy images. Point2Cyl [241] reconstructs CAD shapes from point clouds; it represents profiles as 2D neural implicit. Point2Cyl is also considered a representative work in “primitive detection”, since it first segments the point cloud into patches, and then reconstructs extrusion cylinders from those patches.

The works mentioned above all apply supervised training with ground truth command sequences. ExtrudeNet [203] can learn such sequences unsupervisedly. It uses closed cubic Bezier curves as profiles and proposed a differentiable sketch-to-SDF module and a differentiable extrusion module to construct 3D parts. Those 3D parts are combined using CSG-Stump [202] to form the output shape.

2.1.5 Primitive detection

In previous primitive-based representations, the output shapes are usually generated by a shape decoder from a global shape latent code, therefore they are leaning toward generative models. The representation in this section does not have a shape latent space. It is more like an object detection model in computer vision, where the primitive types and parameters are directly predicted from global and local shape features by the neural networks. Therefore, this representation is usually for reconstructing CAD models of mechanical parts. Specifically, SPFN [135], CPFN [129], ParSeNet [219], HPNet [279], and Point2Cyl [241] all reconstruct parametric surfaces from point clouds. They first use a neural network to extract per-point features from the input point cloud, and then apply a clustering module to segment the point cloud into patches belonging to different primitives, and finally classify the primitive type and regress the primitive parameters for each patch. SPFN [135] and CPFN [129] can reconstruct planes, spheres, cylinders, and cones; ParSeNet [219] and HPNet [279] in addition can reconstruct open and closed B-spline patches. ComplexGen [88] adopts a more structured representation, boundary representation (B-Rep), to recover corners, curves (line, circle, B-spline, ellipse) and patches (plane, cylinder, torus, B-spline, cone, sphere) simultaneously along with their mutual topology constraints. Point2Cyl [241] utilizes a “sketch and extrude” representation.

2.1.6 Grid mesh

Inspired by classic iso-surfacing algorithms such as Marching Cubes (MC) [157], Dual Contouring (DC) [110], and Marching Tetrahedra (MT) [60], which operate on a regular grid structure, several methods have been proposed to also generate a regular grid of parameters, so that surfaces can be extracted cell by cell. Compared to other representations, the advantage of this representation is that a 3D convolutional neural network (CNN) can be applied to produce the grid outputs, and CNNs have been thoroughly studied and heavily optimized to be very efficient and effective, compared to the network architectures (usually MLPs or graph convolutions) used in other representations.

One earliest work on this representation is Deep Marching Cubes (DMC) [139]. DMC has an encoder-decoder structure, where the encoder encodes the inputs into shape latent codes, and the decoder is a 3D CNN to generate a grid of inside-outside signs and a grid of vertex positions. An explicit triangle mesh can be extracted by applying MC algorithm on the predicted sign grid to determine the topology, i.e., mesh tessellation, in each cell, and the positions of the mesh vertices are given by the predicted grid of vertex positions. Our work, Neural Marching Cubes (NMC) [39], has a similar idea in spirit, but it targets iso-surfacing tasks such as mesh reconstruction from grids of signed distances or occupancies. NMC has "local" backbone networks (fully convolutional without bottleneck layer) with small receptive fields for better generalizability, and it has enriched the MC cube tessellation cases to better reconstruct detailed geometry such as sharp features.

Our follow-up work, Neural Dual Contouring (NDC) [35], adopts DC, a much simpler algorithm compared to MC, as the meshing algorithm. NDC has very simple methods: a "local" network generates a grid of signs or intersection flags, another "local" network generates a grid of interior vertex positions, and finally DC is applied to connect the generated vertices to form mesh surfaces. NDC can take any input that can be converted into a grid structure, such as grids of signed or unsigned distances, binary occupancies, and point clouds.

DEFTET [71] also predicts a grid structure - a tetrahedron grid. The method predicts the occupancy for each tetrahedron, and the offset for each vertex relative to their initial positions in the regular tetrahedron grid. The output of DEFTET is a tetrahedral mesh, which is one of the dominant representations for volumetric solids in graphics, and it can be directly used in simulation. Note that DEFTET predicts the occupancy for each tetrahedron, unlike DMC, NMC, and NDC which predict signs on grid vertices. DMTET [222] also uses a tetrahedron grid, but it predicts the signed distance on each grid vertex, so it requires a differentiable iso-surfacing step (differentiable marching tetrahedra). Follow-up work Nvdiffrec [173] combines DMTET and differentiable rendering to reconstruct meshes from multi-view images.

Adaptive O-CNN [254] produces a polygon soup, where the polygons do not connect to form a surface. It uses an octree-like network structure that subdivides nodes according to the expected complexity of surface details. Each leaf node predicts the parameters of a plane, which generates a polygon in that cell.

2.1.7 Voxels

Similar to "Grid mesh" where grid structures are used to store local mesh properties, it is in fact easier for the neural networks to directly predict a grid of voxels carrying implicit field values, such as signed distances or occupancies, and then apply meshing or iso-surfacing algorithms to extract the mesh. Occupancy or signed distance grids have been very popular representations, and the reason is similar to that of "Grid mesh", that a mature technique, convolutional neural networks (CNN), can be applied to produce the grid outputs. Due to the popularity and wide usage of voxels, there are thousands of publications that adopt this representation. Therefore, this section will only cover a few representative works to explain the various use cases.

The simplest way to generate voxels is to use a 3D CNN network to predict a 3D grid of occupancies. A representative work is 3D-R2N2 [43], which has a simple network structure: an encoder-decoder structure where a 2D CNN encoder encodes the input image into a latent code, and a 3D CNN decoder decodes the latent code into a voxel grid. To take multiple images from different views as input, 3D-R2N2 utilizes Long Short-Term Memory (LSTM) [99] to aggregate the sequence of latent codes before feeding them to the shape decoder.

Note that 3D-R2N2 has a bottleneck in the network architecture to produce a global latent code that is not capable of representing detailed shape geometries and is unlikely to generalize to inputs dissimilar to those in the training set. Some works that adopt voxels use a local network or U-Net [207] to take into account local features, such as DECOR-GAN [34] and GenRe [298].

However, voxel grids are of $O(N^3)$ space complexity. It is hard to generate a voxel grid of sufficiently high resolution due to hardware memory constraints. Therefore, octree representation which adaptively subdivides voxels has been applied in many works, such as HSP (Hierarchical Surface Prediction) [92], 3D-CFCN [25], OctNetFusion [205], OGN [236], and Dual OCNN[253].

Finally, predicting grids of signed distances rather than occupancies can better model smooth surfaces, as signed distances contain more information about the surface than binary occupancies. Example works include 3D-EPN [51] and Deep level sets [165].

2.1.8 Neural implicit

Neural implicit representation is an extremely popular representation nowadays. It was proposed in CVPR 2019 by three concurrent works IM-Net [38], OccNet [163], and DeepSDF [185]. Since then there has been an explosion of neural implicit papers. This section will only list a few representative works. We refer the readers to the survey "Neural Fields in Visual Computing and Beyond" [275] for more related works.

Neural implicit representation is essentially an MLP (MultiLayer perceptron) that takes a point's coordinates as input and outputs the inside-outside sign [38, 163] or signed distance [185] of that input point. The MLP itself represents the implicit function of a 3D shape. To generate different output shapes based on the input, the MLP can be conditioned via concatenating a shape latent code with the input point coordinates before feeding the point to the MLP [38, 185], or modulating the MLP network weights using the latent code [163, 227]. The main difference between neural implicit and voxels is that voxels need 3D CNNs to directly predict a grid of implicit field values, therefore it outputs the entire shape with one network forward pass, whereas neural implicit processes each single input point individually with the same MLP. To output the entire shape, a grid of points need to be sampled in space and the MLP needs to run on each of the points to produce a grid of implicit field values. Marching Cubes (MC) [157] is usually applied to the grid to extract an explicit mesh. Theoretically, neural implicit can represent shapes with infinitely fine resolution without increasing the model (MLP) size, in contrast to voxels which have an $O(N^3)$ space complexity, therefore neural implicit has been considered as a compact representation.

Structured Implicit Functions [78] propose to use scaled axis-aligned anisotropic 3D Gaussians (i.e., a set of Gaussian balls) to represent the implicit field, instead of an MLP. It can speed up training and inference of neural implicit and has been adopted in follow-ups that have global shape latent codes.

However, global shape latent codes cannot represent detailed shape geometries and they often lead to overfitting on a specific dataset or category of shapes, therefore methods have been proposed to include local features to condition the MLP so that the network can have better accuracy on local details and generalize better. PIFu [210], PIFuHD [211], DISN [276], and D^{2}IM-Net [136] employ local features from 2D image encoders for single-view 3D reconstruction tasks. LIG [107], ConvONet (Convolutional occupancy networks) [193], SA-ConvONet [235], POCO [21], and IF-Nets [41] employ local features extracted from point clouds or voxels to perform object or scene reconstructions.

Differentiable rendering has also been developed for neural implicit. [153], DVR [179], IDR [283], and DIST [150] can reconstruct objects from multi-view images; they assume the object mask is given for each input image, and each ray intersects the surface at most once (only one intersection point per ray for the gradient to propagate). NeuS [250], HF-NeuS [63], UNISURF [182], VolSDF [282], and [7] also reconstruct objects or scenes from multi-view images. However, they are based on the ray marching volume rendering formula of NeRF [167]. They do not need object masks, and they sample numerous points along each ray to perform volume rendering.

2.1.9 Others

Connect given vertices. This representation can only be used for reconstructing a mesh from a point cloud, because it only connects given vertices from the point cloud. The methods can be classified by whether it infers the inside-outside regions of the shape and thus generates a closed mesh. Most methods do not generate a closed mesh. [149] generates a collection of triangles by proposing candidate triangles, classifying the candidate triangles with a neural network to determine which triangles should exist in the output mesh, and repeating this process. PointTriNet [220] has a classification network to classify whether a given candidate triangle should exist in the output mesh, and a proposal network to suggest likely neighbor triangles for a given existing triangle. [197] first estimates local neighborhoods around each point, and then perform a 2D projection of these neighborhoods so that a 2D Delaunay triangulation is computed to provide candidate triangles, and finally those candidate triangles are aggregated to maximize the manifoldness of the reconstructed mesh. Delaunay triangulation based surface reconstruction methods can guarantee to generate a closed mesh because they first perform 3D Delaunay triangulation on the input points to obtain a tessellation of the 3D space with tetrahedrons, and then classify which tetrahedrons are inside the shape and which are outside. After the inside-outside labels are assigned, a surface can be reconstructed by extracting triangle faces between tetrahedrons of different labels. DeepDT [158] uses a graph neural network on the dual graph of the Delaunay triangulation to predict the label of each tetrahedron.

Generate and connect vertices. This representation first generates a set of mesh vertices with a neural network, and then selectively connects those vertices to form mesh faces with another neural network. The representation can directly generate a 3D mesh as an indexed face set, however, it is rarely used due to its extremely high complexity. Scan2Mesh [50] uses a point cloud generator (MLP) to generate a set of points. Then it constructs a fully connected graph on these points, and uses a graph neural network to predict which mesh edges should exist in the output mesh. Finally, it considers all possible triangle faces that can be formed from the predicted edges, constructs a dual graph on the faces, and uses a graph neural network to predict which mesh faces should exist in the output mesh. Due to the construction of a complete graph on the predicted points and the subsequent graph neural networks, this method can only predict a limited number of vertices (100 in the experiments). PolyGen [174] first generates mesh vertices sequentially from lowest to highest on the vertical axis. The continuous vertex positions are quantized to form discrete bins for likelihood calculation. The next vertex is generated by a vertex Transformer, which takes the current sequence of vertex positions as input, and outputs a distribution over discretized vertex positions. Then it generates polygon faces, also sequentially from lowest to highest on vertex indices. The next face is generated by a face Transformer, which takes the generated vertices and the current sequence of face indices as input, and outputs a distribution over vertex indices.

Sequence of edits. "Modeling 3D Shapes by Reinforcement Learning" [142] models 3D shapes with a sequence of editing operations. The method contains two neural networks, a Prim-Agent that approximates the shape using primitives, and a Mesh-Agent that edits the mesh to create detailed geometry. Given a depth image as shape reference and a set of pre-defined primitives, the Prim-Agent predicts a sequence of actions on primitives (drag primitive corner points or delete primitive) to approximate the target shape. Then the edge loops are added to the output primitives to subdivide each primitive into segments for finer editing control. Finally, the Mesh-Agent takes as input the shape reference and the primitive-based representation, and predicts actions on edge loops (drag edge loop corner points) to create detailed geometry.

2.2 Reconstruction from Voxels

In this section, we review works that reconstruct shapes from a grid of occupancies or signed distances. Based on the motivations, we divide the collection of works into two categories: shape super-resolution and shape parsing, where shape super-resolution reconstructs a more detailed and visually pleasing shape from the input voxels, and shape parsing decomposes the input voxels into primitives and CSG sequences for reverse engineering a CAD shape.

2.2.1 Shape super-resolution

Shape super-resolution strives to recover and even enhance geometric features from voxel inputs. Methods such as OccNet [163] and IM-Net [38] can convert input voxels into neural implicit representation, which can be sampled at arbitrarily high resolution. Although one can argue that the outputs of those methods indeed have higher resolution compared to the input voxels, they do not properly recover shape details; in fact, the reconstructed outputs often lose details presented in the input voxels, due to the usage of a global shape latent code. Therefore, works that perform voxel super-resolution mostly adopt local neural networks which take into account both local and global shape features.

Our works, Neural Marching Cubes (NMC) [39] and Neural Dual Contouring (NDC) [35], are data-driven iso-surfacing algorithms. They reconstruct meshes from input voxels and they are able to recover geometric features such as sharp corners and edges. They adopt and modify the mesh tessellations in classic iso-surfacing algorithms Marching Cubes (MC) [157] and Dual Contouring (DC) [110], and use neural networks to predict the tessellation case and the vertex positions in each voxel to directly output a polygonal mesh. Their neural network backbones have limited receptive fields, meaning that they can only infer geometric features from local regions and they do not have access to global shape information. Thus, the methods focus more on the mesh reconstruction side than the shape super-resolution side.

Similarly, methods such as Convolutional occupancy networks (ConvONet) [193] and "Implicit Functions in feature space" (IF-Nets) [41] can reconstruct shapes as neural implicit from input voxels. ConvONet adopts convolutional encoders to process the input voxels, and predict either a 3D grid of deep features (grid setting) or three 2D grids of deep features on three orthogonal planes (tri-plane setting). Then for each query point, the deep features are retrieved from the grid or the planes via trilinear or bilinear interpolation, and the deep features are concatenated with the query point coordinates to be fed into an MLP to predict the inside/outside status of the query point. IF-Nets also adopts convolutional encoders to compute multi-scale 3D grids encoding global and local features. Note that the backbone networks in these works are sufficiently large to produce global shape features, yet they also utilize local features. They are adept at recovering geometric features from reasonably dense input voxels. However, they do not perform well on coarse inputs, since they only recover details and do not create more details on the coarse shape.

Therefore, methods have been proposed to generate new details on the coarse shape, mostly with the help of a local patch discriminator[82] (PatchGAN[104]). DECOR-GAN [34] performs shape detailization and is able to refine a coarse shape into a variety of detailed shapes with different styles. DECOR-GAN utilizes a 3D CNN generator for upsampling coarse voxels and a 3D PatchGAN discriminator to enforce local patches of the generated shape to be similar to those in the training detailed shapes. DMTET [222] also applies a 3D PatchGAN discriminator on the signed distance field computed from the predicted mesh to improve the local details.

Another approach to generating new details on the coarse shape is to retrieve high-resolution local patches from a database and combine them into a new shape with respect to the structure and context of the input coarse voxels. RetrievalFuse [224] creates a shared embedding space between coarse voxel chunks and a database of high-quality voxel chunks from indoor scene data. For a given coarse voxel input, multiple approximate reconstructions are created with retrieved chunks from the database, and the reconstructed scenes are then fused together with an attention-based blending to produce the final reconstruction.

2.2.2 Shape parsing

Parsing an input voxel grid into primitives and a sequence of operations requires specific output representations, namely, “set of primitives” (Section 2.1.2), “constructive solid geometry” (Section 2.1.3), and “sketch and extrude” (Section 2.1.4). One can refer to those sections for related works. Note that most methods in those sections use global shape latent codes, meaning that the methods can convert any input (point clouds, voxels, images, etc) into parsed shapes, as long as the input can be encoded into global latent codes.

2.3 Reconstruction from Point Clouds

In this section, we review works that reconstruct objects and scenes from point clouds, with or without point normals. We divide the methods into two categories: one based on explicit representations, and the other on implicit representations. Methods with explicit representations can directly output a mesh, but it usually does not guarantee the surface quality, for example, they may not be watertight and may contain no-manifoldness and self-intersections. Methods with implicit representations do guarantee to produce a watertight, manifold mesh without self-intersections, but they require an iso-surfacing algorithm to extract the mesh from the implicit field. Methods with state-of-the-art reconstruction accuracy are mostly using implicit representations.

Note that a significant amount of works in 3D deep learning use a global shape latent code to encode the shape, such as ShapeFlow [106], AtlasNet [86], OccNet [163], DeepSDF[185], Structured Implicit Functions [78], OctField [233], CSG-Stump [202], and DeepCAD [267]. As mentioned in previous sections, global shape latent codes cannot represent detailed shape geometries and they often lead to overfitting on a specific dataset or category of shapes. Therefore, in this section, we will only discuss works that take local features into account.

2.3.1 Explicit representation

Given a clean and mostly uniform point cloud, a straightforward solution to reconstruct a mesh is to create triangles from the given points. Example classic algorithms include Delaunay triangulation based surface reconstruction methods and ball-pivoting [15]. Various methods employ deep learning to improve the performance of such approaches. They are detailed in Section 2.1.9.

If the shape is a single object and has simple topology, it is possible to deform a coarse initial mesh to fit the point cloud in order to reconstruct a mesh. Point2Mesh [94] obtains the initial mesh as the convex hull of the input point cloud, and then deforms the initial mesh to shrink-wrap the point cloud.

For more complex shapes, one could split the input point cloud into overlapping patches, and fit each patch with a deformable 2D square surface. Deep Geometric Prior [260] and Meshlet Priors [8] took inspiration from AtlasNet [86], to overfit each local patch from the input point cloud with a neural network (MLP) deforming a square patch. The resulting set of overfitted local patches can be further sampled and reconstructed to produce a manifold mesh.

The unsigned version (UNDC) in our work Neural Dual Contouring [35] discretizes the space into a 3D grid, and from the input point cloud it predicts for each grid edge whether the edge will be intersected by the output mesh or not. It also predicts an interior vertex for each grid cell, so that if an edge is predicted to have intersected by the mesh, a quad face will be created to connect the four vertices of the four adjacent cells of that edge. Therefore, UNDC can directly reconstruct a quad mesh from input points without the need for point normals.

Methods introduced in Section 2.1.5 “primitive detection” can produce a more structured output: a collection of primitives represented as parametric surfaces. Those methods first use a neural network to extract per-point features from the input point clouds, and then apply a clustering module to segment the point cloud into patches belonging to different primitives, and finally classify the primitive type and regress the primitive parameters for each patch.

Some methods in Section 2.1.3 “Constructive Solid Geometry” and Section 2.1.4 “Sketch and extrude” can reconstruct structured representations (CSG-trees) from point clouds, such as CAPRI-Net [289] and Point2Cyl [241].

2.3.2 Implicit representation

The majority of the deep learning methods adopt implicit representations for shape reconstruction from point clouds. However, the boundary between the two representations "voxels" and "neural implicit" has become blurry since a great number of works use 3D CNNs to predict a grid structure and then use interpolation techniques to obtain the features of continuous query points, which will be used to regress the implicit field values of the query points.

Overfit a single shape. "Sign Agnostic Learning" (SAL) [5], "Implicit Geometric Regularization" (IGR) [85], and "SAL with Derivatives" (SALD) [6] all overfit an MLP representing neural implicit from a point cloud without normals. SAL [5] only assumes the input point cloud is sufficiently dense to produce a good unsigned distance field. It proposed an unsigned similarity function and a geometric network initialization to learn a neural signed distance field from the unsigned distance field. IGR [85] proposed different training objectives and regularization terms to supervise the learned implicit field. SALD [6] improved SAL [5] by including regularizations on the derivatives of the predicted signed distance field. SIREN [227] improves the representation capability of MLPs by using periodic activation functions in MLPs. It can quickly overfit a neural implicit from an oriented point cloud.

Divide space into local cube patches. "Local Implicit Grid" (LIG) [107] and "Deep Local Shapes" (DeepLS) [26] both pointed out that global shape latent codes in early neural implicit methods are not generalizable, therefore they divide the space into a grid of overlapping cube patches, and fit an MLP for each patch. The MLP is pre-trained on a collection of shape patches to learn a low dimensional latent space of plausible (or "real") shape patches. During inference, the MLP is fixed, but a latent code is optimized independently for each patch of the input point cloud to match the MLP's isosurface to the input points in that patch. They both require point normals to help this optimization. Finally, the reconstructed patches are stitched together to give the reconstructed shape or scene. SAIL-S3 [300] has the same idea but it can reconstruct a shape from point clouds without normals, by adopting Sign Agnostic Learning [5].

3D CNN then local neural implicit. ConvONet [193] and IF-Nets [41] are also covered in Section 2.2.1 for voxel super-resolution. They can also reconstruct shapes from point clouds without normals. ConvONet [193] first uses a point cloud encoder to process the input points, and the per-point features are pooled into grids to be processed by CNNs for further feature extraction. IF-Nets [41] discretizes input points into a 3D grid and then adopts CNN encoders to compute multi-scale 3D feature grids. SA-ConvONet [235] is a follow-up of SAL [5] and ConvONet [193] to apply sign agnostic learning proposed in SAL as a post-processing step to improve the reconstruction quality of ConvONet. GIFS [284] does not predict the inside/outside status of each query point, but rather predicts whether two query points are separated by any surface. The concept is similar to Unsigned Neural Dual Contouring (UNDC) [35], but UNDC's meshing is based on Dual Contouring [110], and this work on a modified version of Marching Cubes [157]. It can represent general shapes including non-watertight shapes and shapes with multi-layer surfaces.

Point cloud encoder then local neural implicit. Points2Surf [62] is purely based on point cloud encoders without any CNN. For a query point in space, it adopts one PointNet [195] to encode points sampled at the neighborhood of the query point into a local feature code, and another PointNet to encode the points sampled at the entire input point cloud into a global feature code. The decoder takes both features to predict the signed distance of the query point. POCO [21] proposed to use point cloud convolutions and compute latent vectors at each input point. Then for each query point, it performs a learning-based interpolation on nearest neighbors in input points to retrieve a weighted-averaged feature vector, and the feature vector is processed by an MLP to predict the occupancy of the query point.

Implicit field defined by points. These methods use points (either input points or predicted points) and their properties (such as normals) to directly compute the implicit field value of any query point, similar to classic Radial Basis Function (RBF) surface reconstruction methods. The inference from these points to an implicit field does not involve any deep learning or neural networks. Therefore, the backbone networks that produce these points are the trainable parts in those methods, and they possess learned priors from the training datasets. Neural Splines [261] is a kernel method for surface reconstruction based on kernels arising from infinitely-wide shallow ReLU networks. It can reconstruct an implicit field from a set of points and their normals and it does not involve any neural networks. A follow-up work "Neural Kernel Fields" (NKF) [259] proposed to replace the fixed point properties (normals) with a learned feature vector, so as to have data-dependent kernels. "Shape As Points" (SAP) [192] proposed a differentiable point-to-mesh layer using a differentiable formulation of Poisson Surface Reconstruction (PSR) [117, 118], so that a shape can be represented as a set of points with normals. "Deep Implicit Moving Least-Squares" (Deep IMLS) [151] takes a sparse and un-oriented point cloud as input, and uses a U-Net-like O-CNN autoencoder [251] to predict an octree structure where each octree node contains a fixed number of predicted points with normals. Those predicted points with normals are then used to construct an implicit field by implicit moving least-squares (IMLS) surface formulation [123].

Octrees. The $O(N^3)$ space complexity of regular 3D grids makes methods based on regular grids hard to scale up. Therefore, some methods have been proposed to include adaptive spatial structures such as octrees in the neural networks to improve both efficiency and quality. AdaConv [240] proposed multiscale convolutional kernels that can be applied to adaptive grids as generated with octrees. Dual OCNN [253] designed graph convolutions over the dual graph of octree nodes. Both methods build the octree from the input point cloud and process the input using the proposed convolutional kernels.

2.4 Reconstruction from Single Images

Methods that reconstruct a shape from a single image can be divided into two categories based on the supervision they receive during training. One category is trained with ground truth 3D shapes as supervision. The methods in this category are typically trained on ShapeNet [28]. Another category is trained with only single-view images as supervision. Single-view images mean that there is only one image for each object for training, in contrast to multi-view images where each object has multiple images from different viewpoints. The methods in this category typically train on image datasets of birds, cars, horses, and faces, with shapes of sphere or disk topology.

2.4.1 With 3D supervision

Similar to Section 2.3, a significant amount of works in 3D deep learning use a global shape latent code to encode the shape, such as SurfNet [226], 3D-R2N2 [43], OGN [236], HSP [92], ShapeHD [265], AtlasNet [86], Im2Struct [180], Matryoshka Networks [204], Skeleton-Net [234], IM-Net [38], OccNet [163], Deep Level Sets [165], Deep Meta Functionals [146], topology-modifying AtlasNet [184], Pix2Vox [274], PQ-NET [268], BSP-Net [36], Cvxnet [57], LDIF [77], Neural Template [102], AutoSDF [168]. Global shape latent codes cannot represent detailed shape geometries, and they often lead to learning shape recognition rather than shape reconstruction, as pointed out by "What do single-view 3d reconstruction networks learn?" [237] in 2019. That is, an encoder-decoder structured neural network with a global shape latent code is likely to simply memorize the shapes in the training set during training, and "retrieve" a shape from the memory bank as output during testing. Therefore, in the following, we will only discuss works that take local features into account.

Pixel2Mesh [249] progressively deforms and subdivides a sphere mesh via graph convolutional networks. It extracts image features with a CNN, and then pools image features into the vertices of the mesh to enrich the vertex features, so that the graph convolutional networks can learn local-feature-aware deformations. Geometric Granularity Aware Pixel2Mesh [223] is a follow-up that can edit the topology of the mesh by utilizing an error estimator network to identify faces to prone or repair.

GenRe [298] uses a 2D CNN to predict a depth map from the input image, projects the depth map into a partial spherical map, inpaints the spherical map, projects the complete spherical map and the previous depth map into voxels, and finally processes the voxels by a voxel refinement network to produce the final output. The method shows strong generalizability that it can reconstruct objects from categories not seen during training.

Front2Back [281] first predicts depth, normal, and silhouette maps from the input image. It then detects global reflective symmetries from these maps, and reflects the front depth and normal maps to create partial back depth and normal maps. The partial back depth and normal maps are fed into a network to predict the complete back depth and normal maps. Finally, a 3D mesh can be reconstructed from the Front&back normal and depth maps using Screened Poisson [118].

DISN [276] and PIFu [210] are the two pioneers to first incorporate local features from the input image for single-view 3D reconstruction with neural implicit representation. They learn image feature maps with a CNN, and use the projected location for each 3D query point on the 2D image to extract local features from the image feature maps. The local features are used by MLPs to produce the signed distance or occupancy of the query point. PIFuHD [211] is a follow-up of PIFu that consists of a coarse PIFu network and a fine PIFu network, focusing on global geometry and local details, respectively. Ladybird [278] exploits shape symmetric to predict the signed distance of a query 3D point by using both the query point and its mirrored point with respect to the symmetry plane to extract local features from the image feature maps. D^{2}IM-Net [136] trains the network to learn a detail disentangled reconstruction consisting of a 3D implicit field representing the coarse 3D shape, and two 2D displacement maps capturing the front and back details of the object.

2.4.2 With 2D supervision

Methods often employ a global shape latent code for this task since the task is very difficult and the methods must learn category-specific priors. Most methods adopt a spherical mesh as the template and deform it into the target shape, often with textures. This representation and some related works are covered in Section 2.1.1. Methods that use neural implicit representation for this task often require ground-truth camera pose for each image and multi-view images of the same object. Table 1 in "Share With Thy Neighbors: Single-View Reconstruction by Cross-Instance Consistency" (UNICORN) [171] and Table 1 in "2D GANs Meet Unsupervised Single-view 3D Reconstruction" (GANSVR) [147] are great summaries of recent works on this topic. We recommend interested readers to take a look.

2.5 Reconstruction from Multi-View Images

Only a few works follow the path of single image reconstruction methods in Section 2.4 to learn priors from a collection of training shapes. They can aggregate the global shape latent codes from multiple input images using recurrent neural networks as in 3D-R2N2 [43], aggregate spatial features decoded from global shape latent codes as in Pix2Vox [274], aggregate image features from multiple input images as in Pixel2Mesh++ [258], or aggregate image features from multiple input images and 3D embeddings of spatial locations using a Transformer as in EVolT [248].

Most methods are overfitting a single shape or scene with respect to the multiple input images using methods based on the differentiable rendering algorithms on meshes or neural implicit, or based on the ray marching volume rendering formula of NeRF [167].

2.5.1 Differentiable rendering on explicit representation

Only a few methods adopt explicit mesh representations. NeRS [292], Differentiable Stereopsis (DS) [80], and Neural Deferred Shading (NDS) [263] deform a sphere mesh into the target shape. DEFTET [71], Nvdiffrec [173], and our work MobileNeRF [33] use the "grid mesh" representation. Following differentiable mesh rendering algorithms such as Soft Rasterizer [152], for each pixel, the methods record all or the first k intersected points between the mesh and the ray from the pixel, and aggregate the colors via alpha-compositing during training. DS and DEFTET assume the texture colors are diffuse-only, while NDS and MobileNeRF use MLP neural shader to capture view-dependent effects, and NeRS and Nvdiffrec recover spatially-varying materials and environment map lighting from the input images.

2.5.2 Surface rendering on implicit representation

Most methods in this section have a differentiable rendering formula that assumes the object segmentation mask is given for each input image, and each ray intersects the surface at most once (only one intersection point per ray for the gradient to propagate).

SDFDiff [108], DVR [179], and IDR [283] are pioneers that propose different formulations of differentiable rendering on implicit surfaces, while SDFDiff [108] uses regular grid SDF and others use neural implicit. To model colors, SDFDiff assumes the target shape does not have textures and it does not predict textures for the reconstructed shape; DVR adopts Texture fields [181], using an MLP to predict the RGB color of each surface point, thus it cannot model view-dependent effects; IDR uses an MLP to approximate the bidirectional reflectance distribution function (BRDF) of each surface point.

Neural Lumigraph Rendering (NLR) [119] shows that the extracted mesh can be combined with unstructured lumigraph rendering [23] to achieve real-time rendering. MVSDF [294] leverages stereo matching and feature consistency to optimize the implicit surface representation. RegSDF [293] uses reconstructed point clouds from the input images to supervise and regularize the learning of the neural field. Reparameterization SDF renderer [9] presents a method to compute correct gradients with respect to network parameters in neural SDF renderers.

2.5.3 Volume rendering on implicit representation

The methods in this section adopt a NeRF-style ray marching volume rendering algorithm. For each pixel, the camera shoots a ray crossing it. A number of points are sampled along the ray. Each sampled point carries density ("opacity") and radiance (view-dependent RGB color), predicted by an MLP. The final pixel color is the accumulated radiance of all the sampled points with respect to their density, similar to alpha-compositing. Those methods usually do not need object segmentation masks, and they somehow represent the point density with well-defined neural implicit fields, so that the surface of the shape can be extracted via iso-surfacing.

UNISURF [182], NeuS [250], and VolSDF [282] are pioneers that propose different formulations of volume rendering on implicit surfaces.

"Neural RGB-D Surface Reconstruction" [7] proposes to incorporate depth measurements into the optimization of a NeRF model. NeuralWarp [52] proposes to add a direct photo-consistency term across the different views during optimization to ensure the correctness of the implicit geometry. ManhattanSDF [89] incorporates planer constraints to regularize the geometry in floor and wall regions. Geo-Neus [67] explicitly performs multi-view geometry optimization by leveraging the sparse geometry from structure from motion (SFM) and photometric consistency in multi-view stereo. "Neural 3D Reconstruction in the Wild" [231] follows NeuS [250] and "NeRF in the Wild" (NeRF-W) [161] to reconstruct scenes from Internet photo collections in the presence of varying illumination. SNeS [103] targets 3D reconstruction of partly-symmetric objects, by applying a soft symmetry constraint to the 3D geometry and material properties. SparseNeuS [155] targets 3D reconstruction from sparse images by learning generalizable priors across scenes by introducing geometry encoding volumes for generic surface prediction. MonoSDF [291] utilizes the depth and normal maps predicted by pretrained general-purpose monocular estimator networks for 2D images to improve reconstruction quality and optimization time. HF-NeuS [63] proposes to decompose the SDF into a base function and a displacement function with a coarse-to-fine strategy to gradually increase the high-frequency details.

Chapter 3

BSP-Net: Generating Compact Meshes via Binary Space Partitioning

3.1 Introduction

Recently, there has been an increasing interest in representation learning and generative modeling for 3D shapes. Up to now, deep neural networks for shape analysis and synthesis have been developed mainly for voxel grids [79, 92, 264, 272], point clouds [1, 195, 196, 286, 287], and implicit functions [38, 78, 109, 163, 276]. As the dominant 3D shape representation for modeling, display, and animation, polygonal meshes have not figured prominently amid these developments. One of the main reasons is that the non-uniformity and irregularity of triangle tessellations do not naturally support conventional convolution and pooling operations [93]. However, compared to voxels and point clouds, meshes can provide a more seamless and coherent surface representation; they are more controllable, easier to manipulate, and are more *compact*, attaining higher visual quality using fewer primitives; see Figure 3.1.

For visualization purposes, the generated voxels, point clouds, and implicits are typically converted into meshes in post-processing, e.g., via iso-surface extraction by Marching Cubes [157]. Few deep networks can generate polygonal meshes directly, and such methods are limited to genus-zero meshes [91, 159, 249], piece-wise genus-zero [74] meshes, meshes sharing the same connectivity [72, 232], or meshes with very low number of vertices [50]. Patch-based approaches can generate results which cover a 3D shape with planar polygons [254] or curved [86] mesh patches, but their visual quality is often tampered by visible seams, incoherent patch connections, and rough surface appearance. It is difficult to texture or manipulate such mesh outputs.

In this paper, we develop a generative neural network which outputs polygonal meshes *natively.* Specifically, parameters or weights that are learned by the network can predict multiple planes which fit the surfaces of a 3D shape, resulting in a *compact* and *watertight*

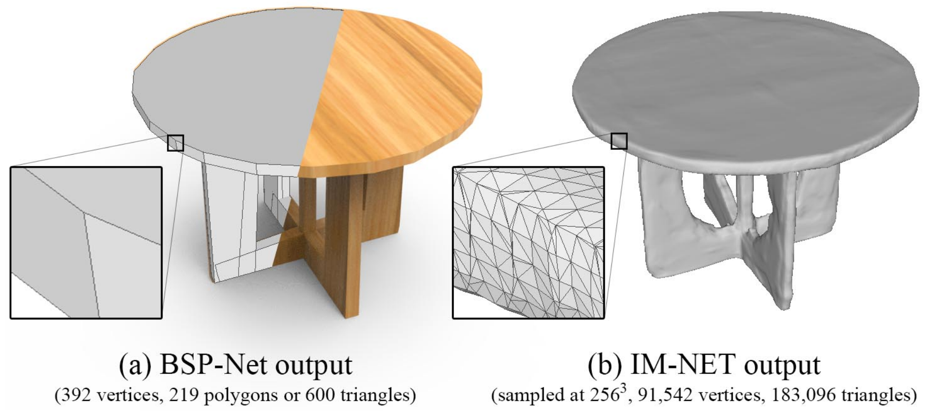

(a) BSP-Net output
(392 vertices, 219 polygons or 600 triangles)
(b) IM-NET output
(sampled at 256^3, 91,542 vertices, 183,096 triangles)

Figure 3.1: (a) 3D shape auto-encoding by BSP-Net quickly reconstructs a *compact*, i.e., low-poly, mesh, which can be easily textured. The mesh edges reproduce *sharp* details in the input (e.g., edges of the legs), yet still approximate smooth geometry (e.g., circular table-top). (b) State-of-the-art methods regress an indicator function, which needs to be iso-surfaced, resulting in over-tessellated meshes which only *approximate* sharp details with smooth surfaces.

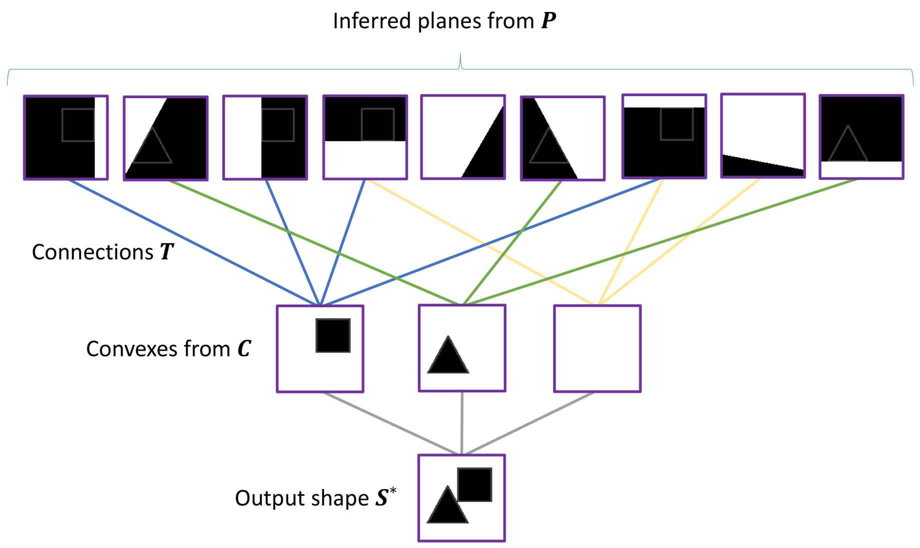

Inferred planes from $\boldsymbol{P}$
Connections $\boldsymbol{T}$
Convexes from $\boldsymbol{C}$
Output shape $\boldsymbol{S}^*$

Figure 3.2: An illustration of “neural” BSP-tree.

polygonal mesh; see Figure 3.1. We name our network *BSP-Net*, since each facet is associated with a *binary space partitioning* (BSP), and the shape is composed by combining these partitions.

BSP-Net learns an *implicit field*: given n point coordinates and a shape feature vector as input, the network outputs values indicating whether the points are inside or outside the shape. The construction of this implicit function is illustrated in Figure 3.2, and consists of three steps: ① a collection of plane equations implies a collection of p binary partitions of space; see Figure 3.2-top; ② an operator $\mathbf{T}_{p \times c}$ groups these partitions to create a collection of c convex shape primitives/parts; ③ finally, the part collection is merged to produce the implicit field of the output shape.

Figure 3.3 shows the network architecture of BSP-Net corresponding to these three steps: ① given the feature code, an MLP produces in layer L_0 a matrix $\mathbf{P}_{p \times 4}$ of canonical parameters that define the implicit equations of p planes: $ax + by + cz + d = 0$; these implicit functions are evaluated on a collection of n point coordinates $\mathbf{x}_{n \times 4}$ in layer L_1; ② the operator $\mathbf{T}_{p \times c}$ is a *binary* matrix that enforces a *selective* neuron feed from L_1 to the next network layer L_2, forming convex parts; ③ finally, layer L_3 assembles the parts into a shape via either sum or min-pooling.

At inference time, we feed the input to the network to obtain components of the BSP-tree, i.e., leaf nodes (planes $\mathbf{P}$) and connections (binary weights $\mathbf{T}$). We then apply classic Constructive Solid Geometry (CSG) to extract the explicit polygonal surfaces of the shapes. The mesh is typically compact, formed by a subset of the p planes directly from the network, leading to a significant speed-up over the previous networks during inference, and without the need for expensive iso-surfacing – current inference time is about 0.5 seconds per generated mesh. Furthermore, meshes generated by the network are guaranteed to be watertight, possibly with *sharp* features, in contrast to smooth shapes produced by previous implicit decoders [38, 109, 163].

BSP-Net is trainable and characterized by *interpretable* network parameters defining the hyper-planes and their formation into the reconstructed surface. Importantly, the network training is *self-supervised* as no ground truth convex shape decompositions are needed. BSP-Net is trained to *reconstruct* all shapes from the training set using the *same* set of convexes constructed in layer L_2 of the network. As a result, our network provides a *natural correspondence* between all the shapes at the level of the convexes. BSP-Net does not yet learn semantic parts. Grouping of the convexes into semantic parts can be obtained manually, or learned otherwise as semantic shape segmentation is a well-studied problem. Such a grouping only need to be done on each convex once to propagate the semantic understanding to all shapes containing the same semantic parts.

Contributions

- BSP-Net is the first deep generative network which directly outputs compact and watertight polygonal meshes with arbitrary topology and structure variety.
- The learned BSP-tree allows us to infer both shape segmentation and part correspondence.
- By adjusting the encoder of our network, BSP-Net can also be adapted for shape auto-encoding and single-view 3D reconstruction (SVR).

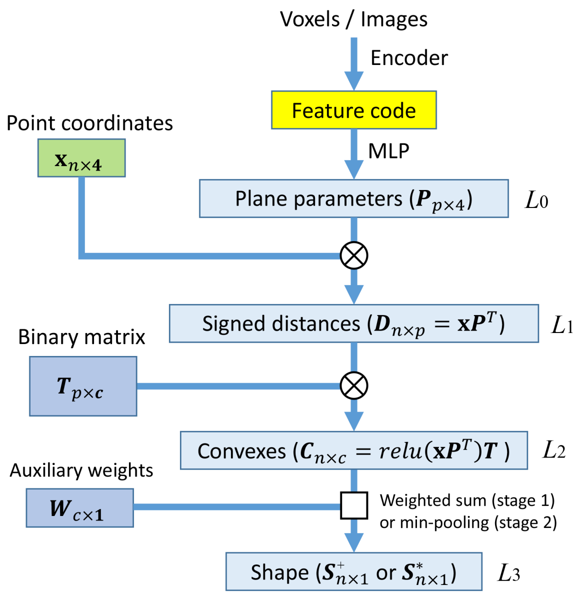

Voxels / Images
Encoder
Feature code
MLP
Point coordinates
$\mathbf{x}_{n\times 4}$
Plane parameters ($\boldsymbol{P}_{p\times 4}$)
L_0
Signed distances ($\boldsymbol{D}_{n\times p} = \mathbf{x}\boldsymbol{P}^T$)
L_1
Binary matrix
$\boldsymbol{T}_{p\times c}$
Convexes ($\boldsymbol{C}_{n\times c} = relu(\mathbf{x}\boldsymbol{P}^T)\boldsymbol{T}$)
L_2
Auxiliary weights
$\boldsymbol{W}_{c\times 1}$
Weighted sum (stage 1)
or min-pooling (stage 2)
Shape ($\boldsymbol{S}^+_{n\times 1}$ or $\boldsymbol{S}^*_{n\times 1}$)
L_3

Figure 3.3: The network corresponding to Figure 3.2.

- To the best of our knowledge, BSP-Net is among the first to achieve *structured* SVR, reconstructing a *segmented* 3D shape from a single unstructured object image.

- Last but not the least, our network is also the first which can reconstruct and recover sharp geometric features.

Through extensive experiments on shape auto-encoding, segmentation, part correspondence, and single-view reconstruction, we demonstrate state-of-the-art performances by BSP-Net. Comparisons are made to leading methods on shape decomposition and 3D reconstruction, using conventional distortion metrics, visual similarity, as well as a new metric assessing the capacity of a model in representing sharp features. In particular, we highlight the favorable fidelity-complexity trade-off exhibited by our network.

3.2 Related work

Large shape collections such as ShapeNet [28] and PartNet [170] have spurred the development of learning techniques for 3D data processing. In this section, we cover representative approaches based on the underlying shape representation learned, with a focus on generative models.

Grid models Early approaches generalized 2D convolutions to 3D [43, 79, 139, 264, 265], and employed volumetric *grids* to represent shapes in terms of coarse occupancy functions, where a voxel evaluates to zero if it is outside and one otherwise. Unfortunately, these methods are typically limited to low resolutions of at most 64^3 due to the cubic growth in memory requirements. To generate finer results, differentiable marching cubes operations have been proposed [157], as well as hierarchical strategies [92, 205, 236, 251, 254] that alleviate the curse of dimensionality affecting dense volumetric grids. Another alternative is to use multi-view images [140, 229] and geometry images [225, 226], which allow standard 2D convolution, but such methods are only suitable on the *encoder* side of a network architecture, while we focus on decoders. Finally, recent methods that perform sparse convolutions [84] on voxel grids are similarly limited to encoders.

Surface models As much of the semantics of 3D models is captured by their *surface*, the boundary between inside/outside space, a variety of methods have been proposed to represent shape surfaces in a differentiable way. Amongst these we find a category of techniques pioneered by PointNet [195] that express surfaces as point clouds [1, 65, 70, 195, 196, 280, 287], and techniques pioneered by AtlasNet [86] that adopt a 2D-to-3D mapping process [260, 226, 249, 280]. An interesting alternative is to consider mesh generation as the process of estimating vertices and their connectivity [50], but these methods do not guarantee watertight results, and hardly scale beyond a hundred vertices.

Implicit models A very recent trend has been the modeling of shapes as a learnable indicator *function* [38, 109, 163], rather than a sampling of it, as in the case of voxel methods. The resulting networks treat reconstruction as a *classification* problem, and are universal approximators [100] whose reconstruction precision is proportional to the network complexity. However, at inference time, generating a 3D model still requires the execution of an expensive iso-surfacing operation whose performance scales cubically in the desired resolution. In contrast, our network directly outputs a *low-poly* approximation of the shape surface.

Shape decomposition BSP-Net generates meshes using a part-based approach, hence techniques that learn shape decompositions are of particular relevance. There are methods that decompose shapes as oriented boxes [239, 180], axis aligned gaussians [78], superquadrics [187], or a union of indicator functions, in BAE-NET [37]. The architecture of our network draws inspiration from BAE-NET, which is designed to segment a shape by reconstructing its parts in different branches of the network. For each shape part, BAE-NET learns an implicit field by means of a binary classifier. In contrast, BSP-Net explicitly learns a tree structure built on plane subdivisions for bottom-up part assembly.

Another similar work is CvxNet [57], which decomposes shapes as a collection of convex primitives. However, BSP-Net differs from CvxNet in several significant ways: ① we target low-poly reconstruction with sharp features, while they target smooth reconstruction; ② their network always outputs K convexes, while the “right” number of primitives is learnt automatically in our method; ③ our optimization routine is completely different from theirs, as their compositional tree structure is *hard-coded*.

Structured models There have been recent works on learning structured 3D models, in particular, linear [305] or hierarchical [134, 304, 169, 180] organization of part bounding boxes. While some methods learn part geometries separately [134, 169], others jointly embed/encode structure and geometry [271, 74]. What is common about all of these methods is that they are *supervised*, and were trained on shape collections with part segmentations and labels. In contrast, BSP-Net is unsupervised. On the other hand, our network is not designed to infer shape semantics; it is trained to learn convex decompositions. To the best of our knowledge, there is only one prior work, Im2Struct [180], which infers part structures from a single-view image. However, this work only produces a box arrangement; it does not reconstruct a structured *shape* like BSP-Net.

Binary and capsule networks The discrete optimization for the tree structures in BSP-Net bears some resemblance to binary [101] and XNOR [198] neural networks. However, only *one* layer of BSP-Net employs binary weights, and our training method differs, as we use a continuous relaxation of the weights in early training. Further, as our network can be thought of as a simplified scene graph, it holds striking similarities to the principles of capsule networks [209], where low-level capsules (hyperplanes) are aggregated in higher (convexes) and higher (shapes) capsule representations. Nonetheless, while [209] addresses discriminative tasks (encoder), we focus on generative tasks (decoder).

3.3 Method

We seek a deep representation of geometry that is simultaneously trainable and interpretable. We achieve this task by devising a network architecture that provides a differentiable Binary Space Partitioning tree (BSP-tree) representation[1] [217, 68]. This representation is easily *trainable* as it encodes geometry via implicit functions, and *interpretable* since its outputs are a collection of convex polytopes. While we generally target 3D geometry, we employ 2D examples to explain the technique without loss of generality.

We achieve our goal via a network containing three main modules, which act on feature vectors extracted by an encoder corresponding to the type of input data (e.g. the features produced by ResNet for images or 3D CNN for voxels). In more details, a first layer that *extracts* hyperplanes conditional on the input data, a second layer that *groups* hyperplanes in the form of half-spaces to create parts (convexes), and a third layer *assembles* parts together to reconstruct the overall object; see Figure 3.3.

[1]While typical BSP-trees are binary, we focus on n-ary trees, with the “B” in BSP referring to binary space partitioning, not the tree structure.

Layer 1: hyperplane extraction Given a feature vector $\mathbf{f}$, we apply a multi-layer perceptron $\mathcal{P}$ to obtain plane parameters $\boldsymbol{P}_{p\times 4}$, where p is the number of planes – i.e. $\boldsymbol{P} = \mathcal{P}_\omega(\mathbf{f})$. For any point $\mathbf{x} = (x, y, z, 1)$, the product $\boldsymbol{D} = \mathbf{x}\boldsymbol{P}^T$ is a vector of *signed* distances to each plane – the ith distance is negative if $\mathbf{x}$ is *inside*, and positive if it is *outside*, the ith plane, with respect to the plane normal.

Layer 2: hyperplane grouping To group hyperplanes into geometric primitives we employ a binary matrix $\boldsymbol{T}_{p\times c}$. Via a max-pooling operation we aggregate input planes to form a set of c *convex* primitives:

$$C^*_j(\mathbf{x}) = \max_i(D_i T_{ij}) \quad \begin{cases} < 0 & \text{inside} \\ > 0 & \text{outside.} \end{cases} \tag{3.1}$$

Note that during training the gradients would flow through only one (max) of the planes. Hence, to ease training, we employ a version that replaces max with summation:

$$C^+_j(\mathbf{x}) = \sum_i \text{relu}(D_i) T_{ij} \quad \begin{cases} = 0 & \text{inside} \\ > 0 & \text{outside.} \end{cases} \tag{3.2}$$

Layer 3: shape assembly This layer groups convexes to create a possibly non-convex output shape via min-pooling:

$$S^*(\mathbf{x}) = \min_j(C^+_j(\mathbf{x})) \quad \begin{cases} = 0 & \text{inside} \\ > 0 & \text{outside.} \end{cases} \tag{3.3}$$

Note that the use of C^+ in the expression above is *intentional.* We avoid using C^* due to the lack of a memory efficient implementation of the operator in TensorFlow 1.

Again, to facilitate learning, we distribute gradients to all convexes by resorting to a (weighted) summation:

$$S^+(\mathbf{x}) = \left[\sum_j W_j \left[1 - C^+_j(\mathbf{x})\right]_{[0,1]}\right]_{[0,1]} \begin{cases} = 1 & \approx \text{in} \\ [0,1) & \approx \text{out,} \end{cases} \tag{3.4}$$

where $\mathbf{W}_{c\times 1}$ is a weight vector, and $[\cdot]_{[0,1]}$ performs clipping. During training we will enforce $\mathbf{W}{\approx}\mathbf{1}$. Note that the inside/outside status here is only *approximate*. For example, when $\mathbf{W}{=}\mathbf{1}$, and all $C^+_j{=}0.5$, one is outside of all convexes, but inside their composition.

Two-stage training Losses evaluated on (3.4) will be approximate, but have better gradient than (3.3). Hence, we develope a two-stage training scheme where: ① in the *continuous* phase, we try to keep all weights continuous and compute an approximate solution via $S^+(\mathbf{x})$

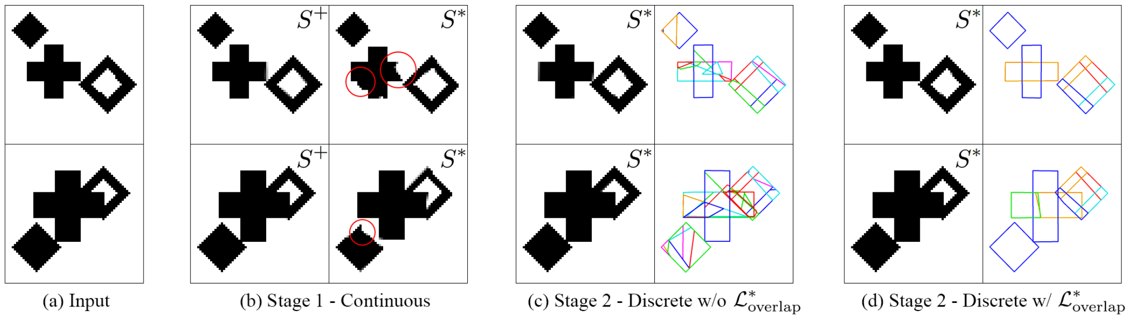

S^+
S^*
(a) Input
(b) Stage 1 - Continuous
(c) Stage 2 - Discrete w/o $\mathcal{L}^*_{\text{overlap}}$
(d) Stage 2 - Discrete w/ $\mathcal{L}^*_{\text{overlap}}$

Figure 3.4: **Evaluation in 2D** – auto-encoder trained on the synthetic 2D dataset. We show auto-encoding results and highlight mistakes made in Stage 1 with red circles, which are resolved in Stage 2. We further show the effect of enabling the (optional) overlap loss. Notice that in the visualization we use different (possibly repeating) colors to indicate different convexes.

– this would generate an approximate result as can be observed in Figure 3.4 (b); ② in the next *discrete* phase, we quantize the weights and use a perfect union to generate accurate results by fine-tuning on $S^*(\mathbf{x})$ – this creates a much finer reconstruction as illustrated in Figure 3.4 (c,d).

Our two-stage training strategy is inspired by classical optimization, where smooth relaxation of integer problems is widely accepted, and mathematically principled.

3.3.1 Training Stage 1 – Continuous

We initialize $\boldsymbol{T}$ and $\boldsymbol{W}$ with random zero-mean Gaussian noise having $\sigma = 0.02$, and optimize the network via:

$$\underset{\omega, \mathbf{T}, \mathbf{W}}{\arg\min} \;\; \mathcal{L}^+_{\text{rec}} + \mathcal{L}^+_{\mathbf{T}} + \mathcal{L}^+_{\mathbf{W}}. \tag{3.5}$$

Given query points $\mathbf{x}$, our network is trained to match $S(\mathbf{x})$ to the ground truth indicator function, denoted by $\mathrm{F}(\mathbf{x}|\mathrm{G})$, in a least-squares sense:

$$\mathcal{L}^+_{\text{rec}} = \mathbb{E}_{\mathbf{x}\sim\mathrm{G}}\left[(S^+(\mathbf{x}) - \mathrm{F}(\mathbf{x}|\mathrm{G}))^2\right], \tag{3.6}$$

where $\mathbf{x}\sim\mathrm{G}$ indicates a sampling that is specific to the training shape G – including random samples in the unit box as well as samples near the boundary $\partial\mathrm{G}$; see [38]. An edge between plane i and convex j is represented by $\mathbf{T}_{ij}$=1, and the entry is zero otherwise. We perform a continuous relaxation of a graph adjacency matrix $\mathbf{T}$, where we require its values to be bounded in the $[0, 1]$ range:

$$\mathcal{L}^+_{\mathrm{T}} = \sum_{t\in\boldsymbol{T}} \max(-t, 0) + \sum_{t\in\boldsymbol{T}} \max(t-1, 0). \tag{3.7}$$

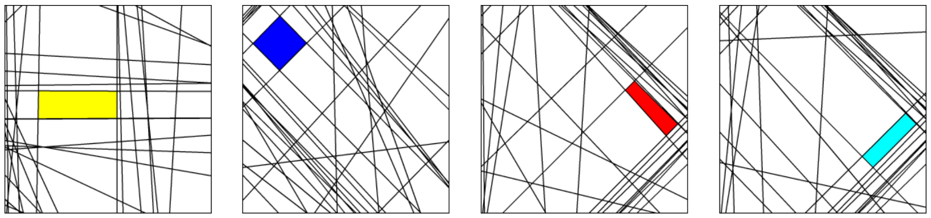


Figure 3.5: **Examples of L_2 output** – a few convexes from the first shape in Figure 3.4, and the planes to construct them. Note how many planes are unused.

Note that this is more effective than using a sigmoid activation, as its gradients do not vanish. Further, we would like $\boldsymbol{W}$ to be close to $\mathbf{1}$ so that the merge operation is a sum:

$$\mathcal{L}_{\mathrm{W}}^{+} = \sum_j |W_j - 1|. \tag{3.8}$$

However, we remind the reader that we initialize with $\boldsymbol{W}{\approx}\mathbf{0}$ to avoid vanishing gradients in early training.

3.3.2 Training Stage 2 – Discrete

In the second stage, we first quantize $\boldsymbol{T}$ by picking a threshold $\lambda = 0.01$ and assign $t{=}(t{>}\lambda)?1{:}0$. Experimentally, we found the values learnt for $\boldsymbol{T}$ to be small, which led to our choice of a small threshold value. With the quantized $\mathbf{T}$, we fine-tune the network by:

$$\arg\min_{\omega} \ \mathcal{L}_{\text{recon}}^{*} + \mathcal{L}_{\text{overlap}}^{*}, \tag{3.9}$$

where we ensure that the shape is well reconstructed via:

$$\mathcal{L}_{\text{recon}}^{*} = \mathbb{E}_{\mathbf{x}\sim \mathrm{G}}\left[\mathrm{F}(\mathbf{x}|G)\cdot \max(S^{*}(\mathbf{x}),0)\right] \tag{3.10}$$

$$+\mathbb{E}_{\mathbf{x}\sim \mathrm{G}}\left[(1-\mathrm{F}(\mathbf{x}|G))\cdot(1-\min(S^{*}(\mathbf{x}),1))\right]. \tag{3.11}$$

The above loss function pulls $S^{*}(\mathbf{x})$ towards 0 if $\mathbf{x}$ should be inside the shape; it pushes $S^{*}(\mathbf{x})$ beyond 1 otherwise. Optionally, we can also discourage overlaps between the convexes. We first compute a mask M such that $M(\mathbf{x},j){=}1$ if $\mathbf{x}$ is in convex j and $\mathbf{x}$ is contained in *more* than one convex, and then evaluate:

$$\mathcal{L}_{\text{overlap}}^{*} = -\mathbb{E}_{\mathbf{x}\sim \mathrm{G}}\left[\mathbb{E}_j\left[M(\mathbf{x},j)C_j^{+}(\mathbf{x})\right]\right]. \tag{3.12}$$

3.3.3 Algorithmic and training details

In our 2D experiments, we use $p{=}256$ planes and $c{=}64$ convexes. We use a simple 2D convolutional encoder where each layer downsamples the image by half, and doubles the number of feature channels. We use the centers of all pixels as samples. In our 3D experiments, we use p=4,096 planes and c=256 convexes. The encoder for *voxels* is a 3D CNN encoder where each layer downsamples the grid by half, and doubles the number of feature channels. It takes a volume of size 64^3 as input. The encoder for *images* is ResNet-18 without pooling layers that receives images of size 128^2 as input. All encoders produce feature codes $|\mathbf{f}|$=256. The dense network $\mathcal{P}_\omega$ has widths $\{512, 1024, 2048, 4p\}$ where the last layer outputs the plane parameters.

When training the auto-encoder for 3D shapes, we adopt the progressive training from [38], on points sampled from grids that are increasingly denser (16^3, 32^3, 64^3). Note that the hierarchical training is not necessary for convergence, but results in an $\approx 3\times$ speedup in convergence. In Stage 1, we train the network on 16^3 grids for 8 million iterations with batch size 36, then 32^3 for 8 million iterations with batch size 36, then 64^3 for 8 million iterations with batch size 12. In Stage 2, we train the network on 64^3 grids for 8 million iterations with batch size 12.

For single-view reconstruction, we also adopt the training scheme in [38], i.e., train an auto-encoder first, then only train the image encoder of the SVR model to predict *latent* codes instead of directly predicting the output shapes. We train the image encoder for 1,000 epochs with batch size 64. We run our experiments on a workstation with an Nvidia GeForce RTX 2080 Ti GPU. When training the auto-encoder (one model on the 13 ShapeNet categories), Stage 1 takes about ≈3 days and Stage 2 takes ≈2 days; training the image-encoder requires ≈1 day.

3.4 Results and evaluation

We study the behavior of BSP-Net on a synthetic 2D shape dataset (Section 3.4.1), and evaluate our auto-encoder (Section 3.4.2), as well as single view reconstruction (Section 3.4.3) compared to other state-of-the-art methods.

3.4.1 Auto-encoding 2D shapes

To illustrate how our network works, we created a synthetic 2D dataset. We place a diamond, a cross, and a hollow diamond with varying sizes over 64×64 images; see Figure 3.4(a). The order of the three shapes is *sorted* so that the diamond is always on the left and the hollow diamond is always on the right – this is to *mimic* the structure of shape datasets such as ShapeNet [28]. After training Stage 1, our network has already achieved a good approximate S^+ reconstruction, however, by inspecting S^*, the output of our inference, we can see there are several imperfections. After the fine-tuning in Stage 2, our network achieves near perfect reconstructions. Finally, the use of overlap losses significantly improves the compactness of representation, reducing the number of convexes per part; see Figure 3.4(d).

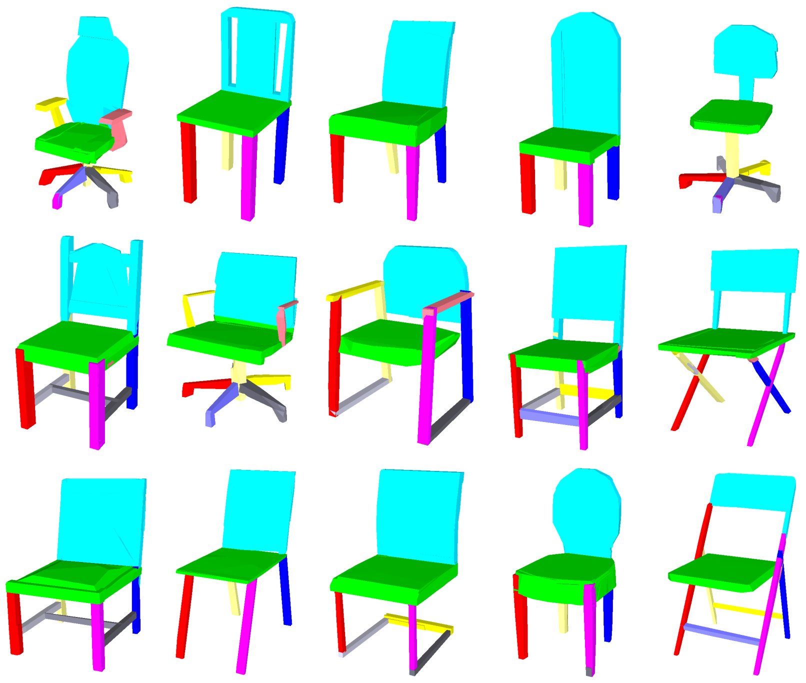


Figure 3.6: **Segmentation and correspondence** – semantics implied from autoencoding by BSP-Net. Colors shown here are the result of a *manual* grouping of learned convexes. The color assignment was performed on a few shapes: once a convex is colored in one shape, we can propagate the color to the other shapes by using the learnt convex id.

	CD	NC	LFD
VP [239]	2.259	0.683	6132.74
SQ [187]	1.656	0.719	5451.44
BAE [37]	1.592	0.777	4587.34
Ours	**0.447**	**0.858**	**2019.26**
Ours + $\mathcal{L}^*_{\text{overlap}}$	0.448	**0.858**	2030.35

Table 3.1: **Surface reconstruction** quality and comparison for 3D shape autoencoding. Best results are marked in bold.

	plane	car	chair	lamp	table	mean
VP [239]	37.6	41.9	64.7	62.2	62.1	56.9
SQ [187]	48.9	49.5	65.6	**68.3**	77.7	66.2
BAE [37]	40.6	46.9	72.3	41.6	68.2	59.8
Ours	74.2	69.5	80.9	52.3	**90.3**	79.3
Ours + $\mathcal{L}^*_{\text{overlap}}$	**74.5**	**69.7**	**82.1**	53.4	**90.3**	**79.8**
BAE* [37]	75.4	73.5	85.2	73.9	86.4	81.8

Table 3.2: **Segmentation**: comparison in per-label IoU.

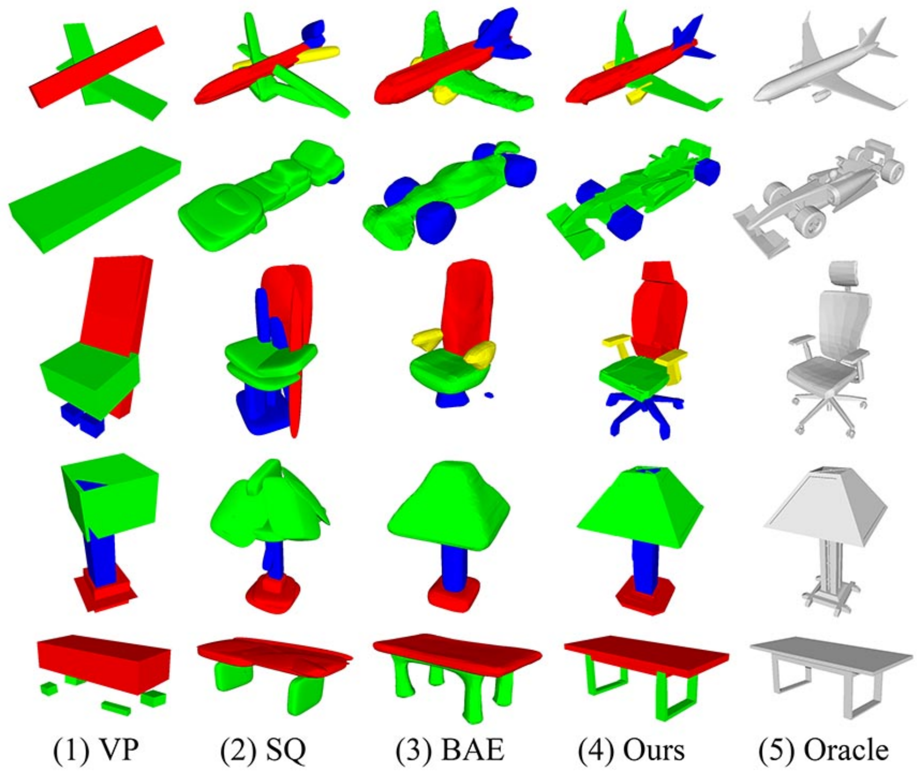


Figure 3.7: **Segmentation and reconstruction / Qualitative**.

Figure 3.5 visualizes the planes used to construct the individual convexes – we visualize planes i in convex j so that $T_{ij}=1$ and $P_{i1}^2 + P_{i2}^2 + P_{i3}^2 > \varepsilon$ for a small threshold ε (to ignore planes with near-zero gradients). Note how BSP-Net creates a natural *semantic* correspondence across inferred convexes. For example, the hollow diamond in Figure 3.4(d) is always made of the same four convexes in the same relative positions – this is mainly due to the static structure in $\boldsymbol{T}$: different shapes need to *share* the same set of convexes and their associated hyper-planes.

3.4.2 Auto-encoding 3D shapes

For 3D shape autoencoding, we compare BSP-Net to a few other shape decomposition networks: Volumetric Primitives (VP) [239], Super Quadrics (SQ) [187], and Branched Auto Encoders (BAE) [37]. Note that for the segmentation task, we also evaluate on BAE*, the version of BAE that uses the values of the predicted implicit function, and not just the classification boundaries – please note that the surface *reconstructed* by BAE and BAE* are *identical*.

Since all these methods target *shape decomposition* tasks, we train *single* class networks, and evaluate segmentation as well as reconstruction performance. We use the ShapeNet (Part) Dataset [285], and focus on five classes: airplane, car, chair, lamp and table. For the car class, since none of the networks separates surfaces (as we perform *volumetric* modeling), we reduce the parts from (wheel, body, hood, roof) $\rightarrow$ (wheel, body); and analogously for lamps (base, pole, lampshade, canopy) $\rightarrow$ (base, pole, lampshade) and tables (top, leg, support) $\rightarrow$ (top, leg).

As quantitative metrics for reconstruction tasks, we report symmetric Chamfer Distance (**CD**, scaled by $\times 1000$) and Normal Consistency (**NC**) computed on $4k$ surface sampled points. We also report the Light Field Distance (**LFD**) [30] – the best-known visual similarity metric from computer graphics. For segmentation tasks, we report the typical mean per-label Intersection Over Union (IOU).

Segmentation Table 3.2 shows the per category segmentation results. As we have ground truth part labels for the point clouds in the dataset, after training each network, we obtain the part label for each primitive/convex by *voting*: for each point we identify the nearest primitive to it, and then the point will cast a vote for that primitive on the corresponding part label. Afterwards, for each primitive, we assign to it the part label that has the highest number of votes. We use 20% of the dataset for assigning part labels, and we use all the shapes for testing. At test time, for each point in the *point cloud*, we find its nearest primitive, and assign the part label of the primitive to the point. In the comparison to BAE, we employ their one-shot training scheme [37, Sec.3.1]. Note that BAE-NET* is specialized to the segmentation task, while our work mostly targets part-based reconstruction; as such, the IoU performance in Table 3.2 is an *upper bound* of segmentation performance.

Figure 3.6 shows *semantic* segmentation and part correspondence *implied* by BSP-Net autoencoding, showing how individual parts (left/right arm/leg, etc.) are matched. In our method, all shapes are corresponded at the primitive (convexes) level. To reveal shape semantics, we *manually* group convexes belonging to the same semantic part and assign them the same color. Note that the color assignment is done on each convex once, and propagated to all the shapes.

Reconstruction comparison BSP-Net achieves significantly better reconstruction quality, while maintaining high segmentation accuracy; see Table 3.1 and Figure 3.7, where we color each *primitive* based on its inferred part label. BAE-NET was designed for segmentation, thus produces poor-quality part-based 3D reconstructions. Note how BSP-Net is able to represent complex parts such as legs of swivel chairs in Figure 3.7, while none of the other methods can.

3.4.3 Single view reconstruction (SVR)

We compare our method with AtlasNet [86], IM-NET [38] and OccNet [163] on the task of single view reconstruction. We report quantitative results in Table 3.3 and Table 3.4, and qualitative results in Figure 3.8. We use the 13 categories in ShapeNet [28] that have more

	Chamfer Distance (CD)					Edge Chamfer Distance (ECD)					Light Field Distance (LFD)				
	Atlas0	Atlas25	OccNet$_{32}$	IM-NET$_{32}$	Ours	Atlas0	Atlas25	OccNet$_{32}$	IM-NET$_{32}$	Ours	Atlas0	Atlas25	OccNet$_{32}$	IM-NET$_{32}$	Ours
airplane	0.587	**0.440**	1.534	2.211	0.759	**0.396**	0.575	1.494	0.815	0.487	5129.36	4680.37	7760.42	7581.13	**4496.91**
bench	1.086	**0.888**	3.220	1.933	1.226	0.658	0.857	2.131	1.400	**0.475**	4387.28	4220.10	4922.89	4281.18	**3380.46**
cabinet	1.231	1.173	**1.099**	1.902	1.188	3.676	2.821	10.804	9.521	**0.435**	1369.90	1558.45	1187.08	1347.97	**989.12**
car	0.799	**0.688**	0.870	1.390	0.841	1.385	1.279	8.428	6.085	**0.702**	1870.42	1754.87	1790.00	1932.78	**1694.81**
chair	1.629	**1.258**	1.484	1.783	1.340	1.440	1.951	4.262	3.545	**0.872**	3993.94	3625.23	3354.00	3473.62	**2961.20**
display	1.516	**1.285**	2.171	2.370	1.856	2.267	2.911	6.059	5.509	**0.697**	2940.36	3004.44	2565.07	3232.06	**2533.86**
lamp	3.858	**3.248**	12.528	6.387	3.480	2.458	2.690	8.510	4.308	**2.144**	7566.25	7162.20	8038.98	6958.52	**6726.92**
speaker	2.328	**1.957**	2.662	3.120	2.616	9.199	5.324	11.271	9.889	**1.075**	2054.18	2075.69	2393.50	1955.40	**1748.26**
rifle	1.001	**0.715**	2.015	2.052	0.888	0.288	0.318	1.463	1.882	**0.231**	6162.03	6124.89	6615.20	6070.86	**4741.70**
couch	1.471	**1.233**	1.246	2.344	1.645	2.253	3.817	10.179	8.531	**0.869**	2387.09	2343.11	1956.26	2184.28	**1880.21**
table	1.996	**1.376**	3.734	2.778	1.643	1.122	1.716	3.900	3.097	**0.515**	3598.59	3286.05	3371.20	3347.12	**2627.82**
phone	1.048	**0.975**	1.183	2.268	1.383	10.459	11.585	16.021	14.684	**1.477**	1817.61	1816.22	1995.98	1964.46	**1555.47**
vessel	1.179	**0.966**	1.691	2.385	1.585	0.782	0.889	12.375	3.253	**0.588**	4551.17	4430.04	5066.99	4494.14	**3931.73**
mean	1.487	**1.170**	2.538	2.361	1.432	1.866	2.069	6.245	4.617	**0.743**	3644.91	3436.14	3795.23	3700.22	**2939.15**

Table 3.3: **Single view reconstruction –** comparison to the state of the art. *Atlas25* denotes AtlasNet with 25 square patches, while *Atlas0* uses a single spherical patch. Subscripts to OccNet and IM-NET show sampling resolution. For fair comparisons, we use resolution 32^3 so that OccNet and IM-NET output meshes with comparable number of vertices and faces.

	CD	ECD	LFD	#V	#F
Atlas0	1.487	1.866	3644.91	7446	14888
Atlas25	**1.170**	2.069	3436.14	2500	4050
OccNet$_{32}$	2.538	6.245	3795.23	1511	3017
OccNet$_{64}$	1.950	6.654	3254.55	6756	13508
OccNet$_{128}$	1.945	6.766	3224.33	27270	54538
IM-NET$_{32}$	2.361	4.617	3700.22	1204	2404
IM-NET$_{64}$	1.467	4.426	2940.56	5007	10009
IM-NET$_{128}$	1.387	1.971	2810.47	20504	41005
IM-NET$_{256}$	1.371	2.273	**2804.77**	82965	165929
Ours	1.432	**0.743**	2939.15	**1191**	**1913**

Table 3.4: **Low-poly analysis –** the dataset-averaged metrics in single view reconstruction. We highlight the number of vertices #V and triangles #F in the predicted meshes.

than 1,000 shapes each, and the rendered views from 3D-R2N2 [43]. We train one model on all categories, using 80% of the shapes for training and 20% for testing, in a similar fashion to AtlasNet [86]. For other methods, we download the *pre-trained* models released by the authors. Since the pre-trained OccNet [163] model has a different train-test split than others, we evaluate it on the intersection of the test splits.

Edge Chamfer Distance (ECD) To measure the capacity of a model to represent *sharp* features, we introduce a new metric. We first compute an "edge sampling" of the surface by generating $16k$ points $\mathbf{S}=\{\mathbf{s}_i\}$ uniformly distributed on the surface of a model, and then compute sharpness as: $\sigma(\mathbf{s}_i) = \min_{j \in \mathcal{N}_\varepsilon(\mathbf{s}_i)} |\mathbf{n}_i \cdot \mathbf{n}_j|$, where $\mathcal{N}_\varepsilon(\mathbf{s})$ extracts the indices of the samples in $\mathbf{S}$ within distance ε from $\mathbf{s}$, and $\mathbf{n}$ is the surface normal of a sample. We set ε=0.01, and generate our edge sampling by retaining points such that $\sigma(\mathbf{s}_i)$<0.1; see Figure 3.8.

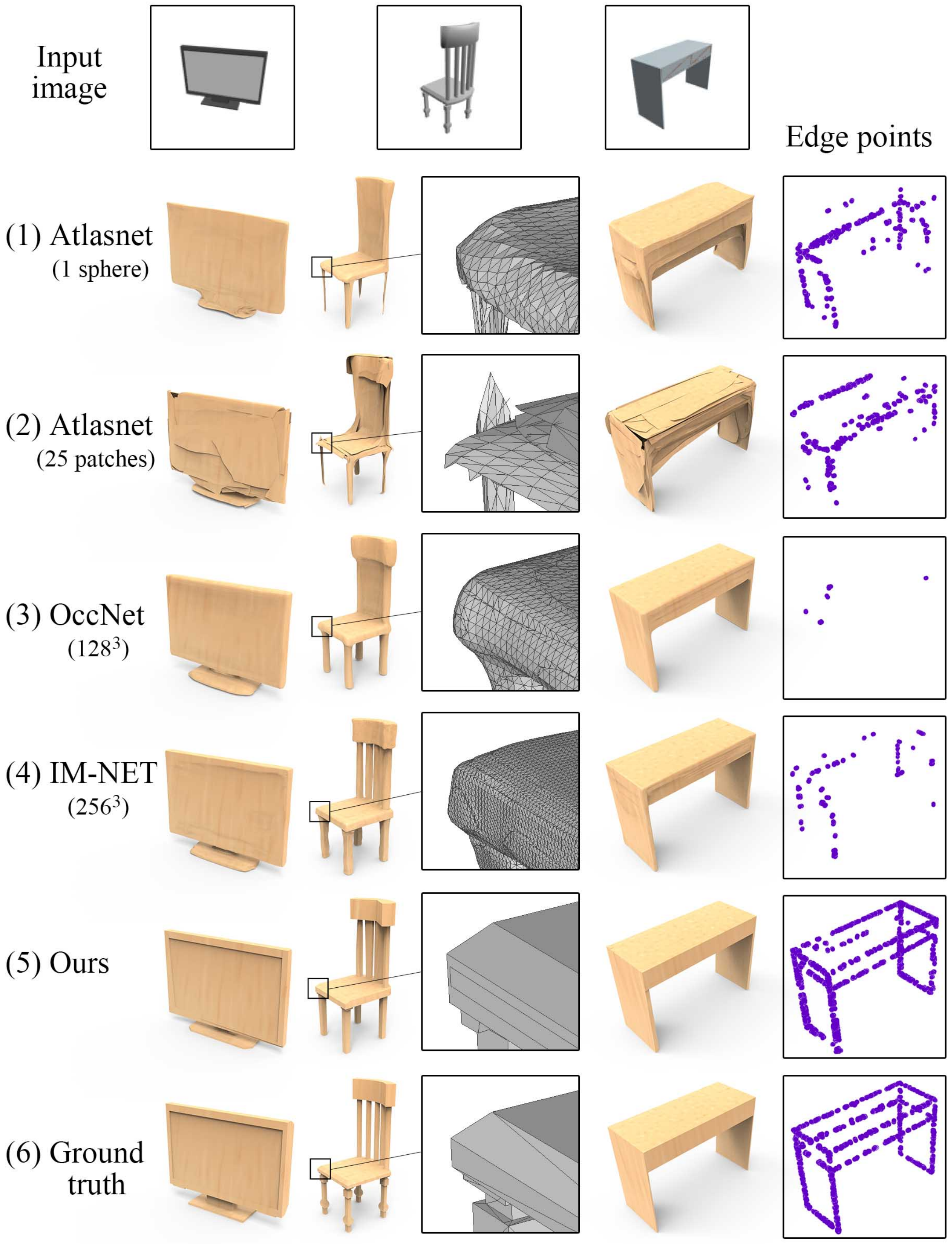


Figure 3.8: **Single-view 3D reconstruction** – comparison to AtlasNet [86], IM-NET [38], and OccNet [163]. Middle column shows mesh tessellations of the reconstruction; last column shows the edge sampling used in the ECD metric.

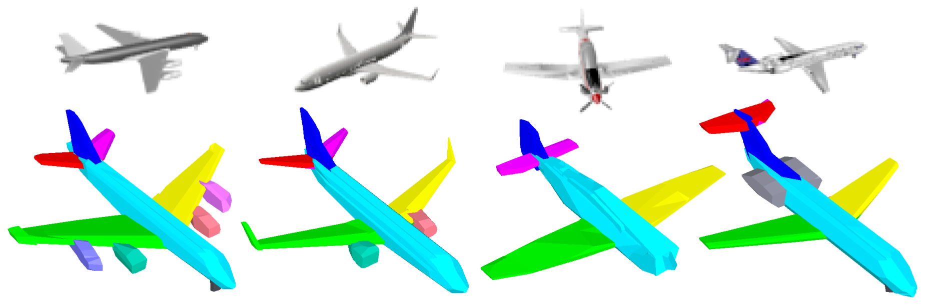


Figure 3.9: **Structured SVR** by BSP-Net reconstructs each shape with corresponding convexes. Convexes belonging to the same semantic parts are manually grouped and assigned the same color, resulting in semantic part correspondence.

Given two shapes, the ECD between them is nothing but the Chamfer Distance between the corresponding edge samplings.

Analysis Our method achieves *comparable* performance to the state-of-the-art in terms of Chamfer Distance. As for visual quality, our method also *outperforms* most other methods, which is reflected by the superior results in terms of Light Field Distance. Similarly to Figure 3.6, we manually color each convex to show part correspondences in Figure 3.9. We visualize the triangulations of the output meshes in Figure 3.8: our method outputs meshes with a smaller number of polygons than state-of-the-art methods. Note that these methods cannot generate low-poly meshes, and their vertices are always distributed quasi-uniformly.

Finally, note that our method is the *only* one amongst those tested capable of representing sharp edges – this can be observed quantitatively in terms of Edge Chamfer Distance, where BSP-Net performs much better. Note that AtlasNet could also generate edges in theory, but the shape is not watertight and the edges are irregular, as it can be seen in the zoom-ins of Figure 3.8. We also analyze these metrics aggregated on the entire testing set in Table 3.4. In this final analysis, we also include OccNet_{128} and IM-NET_{256}, which are the *original* resolutions used by the authors. Note the average number of *polygons* inferred by our method is 655 (recall #polygons $\leq$ #triangles in polygonal meshes).

3.5 Conclusions

We introduce BSP-Net, an unsupervised method which can generate compact and structured polygonal meshes in the form of convex decomposition. Our network learns a BSP-tree built on the same set of planes, and in turn, the same set of convexes, to minimize a reconstruction loss for the training shapes. These planes and convexes are defined by weights learned by the network. Compared to state-of-the-art methods, meshes generated by BSP-

Net exhibit superior visual quality, in particular, sharp geometric details, when *comparable* number of primitives are employed.

The main limitation of BSP-Net is that it can only decompose a shape as a *union* of convexes. Concave shapes, e.g., a teacup or ring, have to be decomposed into many small convex pieces, which is unnatural and leads to wasting of a considerable amount of representation budget (planes and convexes). A better way to represent such shapes is to do a *difference* operation rather than union. How to generalize BSP-Net to express a variety of CSG operations is an interesting direction for future work.

Current training times for BSP-Net are quite significant: 6 days for $4,096$ planes and 256 convexes for the SVR task trained across all categories; inference is fast however. While most shapes only need a small number of planes to represent, we cannot reduce the total number of planes as they are needed to well represent a large *set* of shapes. It would be ideal if the network can adapt the primitive count based on the complexity of the input shapes; this may call for an architectural change to the network.

While its applicability to RGBD data could leverage the auto-decoder ideas explored by [109], the generalization of our method beyond curated datasets [28], and the ability to train from only RGB images are of critical importance.

Chapter 4

Neural Marching Cubes

4.1 Introduction

The Marching Cubes (MC) algorithm [157] is the most prominent method for isosurface extraction, and has been widely adopted in scientific visualization, shape reconstruction, and by the recent emerging approaches for learning neural implicit representations [38, 163, 109]. MC takes as input a uniform grid of values representing a discretized implicit field, and extracts a triangle mesh representing the zero-isosurface of the field. The classical MC determines the local mesh topology and tessellation in each cube of the grid by examining the signs at the eight cube corners and referring to a predefined look-up table indexed by the sign configurations. If the isosurface intersects a cube edge, a new mesh vertex is added to that edge with its position computed via linear interpolation.

With its popularity and wide adoption, MC has seen notable improvements over the years. Marching Cubes 33 [40] is one of the first works to assume that the implicit field in each cube follows *trilinear interpolation* with respect to the cube vertices, and the ensuing meshing algorithm aims for topological correctness under the trilinearity assumption. This increases the number of unique cases of mesh tessellations from 15 (in the original MC [157]) to 33, hence the name. The algorithm itself requires many tests to determine which topological case a cube belongs to, and the process can be error-prone. As a result, many patch-ups and improvements have been made to MC 33 [133, 156, 64, 49]. Still, the trilinearity assumption, which has persisted, can lead to poor estimates of the true implicit field in general, and especially near *sharp features* of a shape, as shown in Figure 4.1.

Indeed, while MC has been employed for decades, its inability to recover sharp features, arguably its most long-standing issue, has not been fully resolved. Earlier tessellation templates [157, 273, 40] were quite coarse and designed for reconstructing soft and smooth objects. Even with a refined tessellation to possibly represent sharp features in later follow-ups, e.g., [156], there is insufficient information in isolated cubes to disambiguate between soft patches and sharp edges, and this issue is worsened when the inputs are binary occupancies instead of signed distances. In general, a shape edge is not a “point-wise” feature,

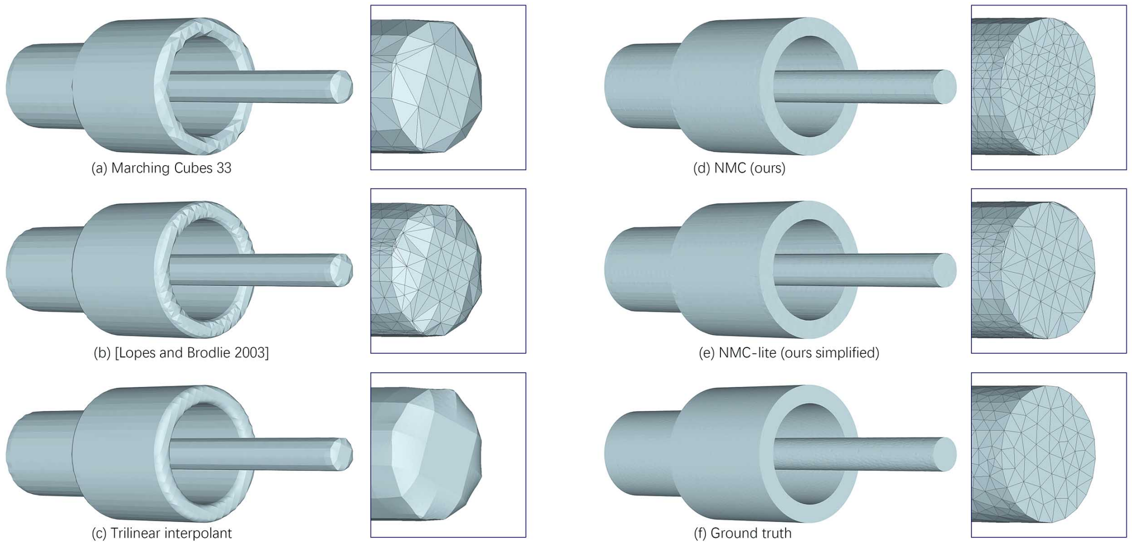

(a) Marching Cubes 33
(d) NMC (ours)
(b) [Lopes and Brodlie 2003]
(e) NMC-lite (ours simplified)
(c) Trilinear interpolant
(f) Ground truth

Figure 4.1: Our Neural Marching Cubes (NMC) is trained to reconstruct the zero-isosurface of an implicit field, while preserving geometric features such as sharp edges and smooth curves. We compare NMC (d), and a simplified version (e), to the best-known MC variants (a-b), as well as a trilinear interpolant (c), none of which could reconstruct the features. The inputs to all methods are the same: a uniform grid of signed distances sampled from the ground truth (f).

but a geometry property that reveals coherence or dependencies over neighboring cubes. Hence, edge prediction should account for that *context*, yet classical MC algorithms have not used such neighbor information.

In this paper, we introduce *Neural Marching Cubes* (NMC), a *data-driven* approach for isosurfacing from a discretized occupancy or signed distance field (SDF). The main premise of our work is that there is sufficient *predictability* in the vertex positions and local mesh topologies of "nice" mesh tessellations under the MC setting, in particular, when they reflect persistent features, such as sharp edges, over neighboring cubes. Hence, a well-designed learning approach would be more effective than handcrafting all the templates and making heuristic decisions such as trilinear interpolation. To this end, we re-design the tessellation templates so that they are more apt at preserving geometric features including sharp edges and corners, and develop a neural network to learn the vertex positions and mesh topologies from a set of training meshes, so as to account for *contextual* information from nearby cubes.

To realize NMC, we must address several immediate challenges. First, we need a per-cube parameterization that is compatible with neural processing, so that the output mesh can be predicted by a network. Such a representation must contain all the information required to perform our modified MC algorithm, including mesh topology and vertex positions in each cell, while minimizing redundancy for efficient network training. Second, our reconfigured mesh tessellation templates must be *refined*, *complete* in the sense of avoiding ambiguities

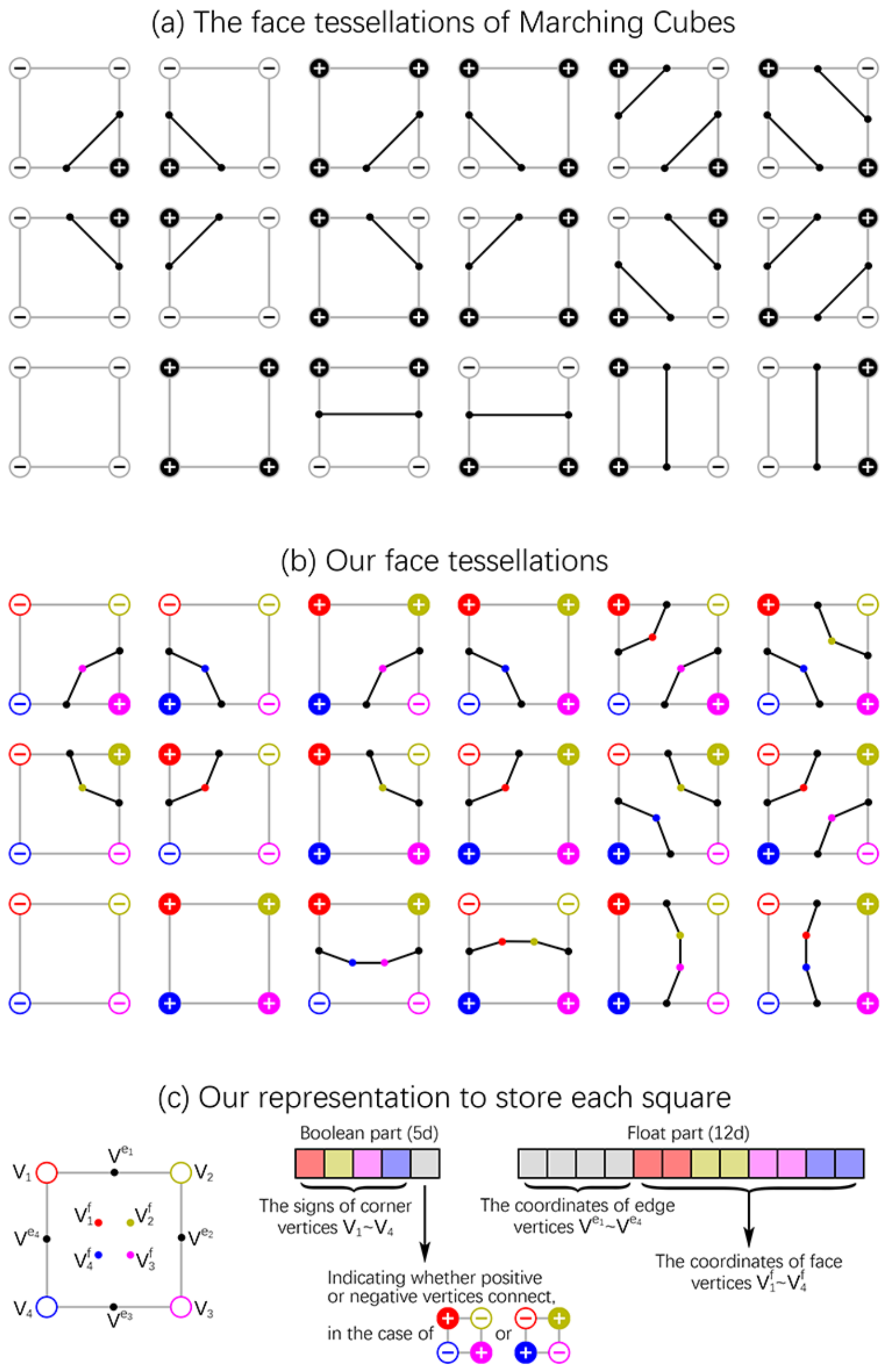

(a) The face tessellations of Marching Cubes
(b) Our face tessellations
(c) Our representation to store each square
V_1
V_2
V_3
V_4
V^{e_1}
V^{e_2}
V^{e_3}
V^{e_4}
V^f_1
V^f_2
V^f_3
V^f_4
Boolean part (5d)
Float part (12d)
The signs of corner vertices V_1~V_4
The coordinates of edge vertices V^{e_1}~V^{e_4}
The coordinates of face vertices V^f_1~V^f_4
Indicating whether positive or negative vertices connect, in the case of
or

Figure 4.2: Tessellation design (b) and parameterization (c) for NMC in 2D, in contrast to classical MC (a). Four new (face) vertices are added inside each square (c), each associated to a corner vertex (solid/hollow circle with +/- sign to indicate outside/inside), with matching color. Meshing information is encoded by a vector with a "boolean part" revealing topology and a "float part" storing all vertex positions; see Section 4.3.1 for more details.

and covering all topological varieties, and be consistent with our defined representation. Last but not least, we must obtain quality training meshes resembling ideal outputs of NMC *automatically* from a collection of 3D shapes. These ground truth meshes should use the designed tessellation templates in each cube, and be stored in the aforementioned representation.

Figure 4.2 illustrates our tessellation design and per-cube representation in the 2D case. In contrast to classical MC, we add new vertices to each cube, leading to more refined tessellations that can better represent geometric features. By carefully designing the representation (see Section 4.3), all topological cases that are applicable to our design, including all cases in MC 33, as well as all vertex positions, can be compactly encoded in a vector form for neural processing. Also, by associating the added vertices to their respective corner vertices, the new tessellations can all be obtained naturally and efficiently by following a few design guidelines.

With the above representation, our network for NMC is simply a 3D variant of ResNet [95] which inputs implicit field values. The network is trained with ground-truth meshes to set up both the topological and geometric losses, which operate respectively on the binary and float parts of the *3D version* (see Figure 4.3) of the per-cube vector representation shown in Figure 4.2(c). By limiting its receptive field, our network learns local features, rather than from the entire shape, so that it generalizes well to new shapes and new datasets. Finally, we devise an optimization-based approach (see Section 4.3.4) to obtain the ground truth output meshes from raw 3D shapes based on our representation and tessellation design.

We evaluate NMC by qualitative and quantitative comparisons, in terms of reconstruction quality and feature preservation, to well-known MC variants, on both signed distance and binary voxel inputs. We show that our method is the first MC-based approach that is able to recover sharp features without requiring additional inputs other than a uniform grid of implicit field values. In addition, our network can more faithfully reconstruct local mesh topologies near thin shape structures and closeby surface sheets. We also provide a simplified version of NMC, which adopts the same mesh tessellation templates as [156], to study the fidelity-complexity trade-off. We finally show that our model can be trained to reconstruct clean meshes from noisy inputs by adjusting the training data, thus offering a useful tool for extracting 3D shapes from those shape representations designed for neural networks.

4.2 Related work

Our work is inspired by and closely related to Marching Cubes [157] and its several variants. For completeness, we also discuss other algorithms for isosurfacing and emphasize the strengths of our NMC approach. Also relevant are recent works from the rapidly advancing field of neural geometry learning.

4.2.1 Marching Cubes (MC) and Variants

The original MC algorithm [157] was proposed to reconstruct 3D structures from medical scan images, while a concurrent work [273] developed similar ideas for reconstructing soft objects. However, these methods did not guarantee surface consistency due to ambiguities of the tessellations in each cube; they may generate holes when tessellations in adjacent cells produce different corner connections on the common face [61]. Asymptotic Decider [178] addressed the issue by assuming the implicit field within each face follows bilinear interpolation with respect to the four face vertices, and proposed a solution to produce topologically correct tessellations under the bilinearity assumption. Several follow-up works [162, 40, 176] further assumed the implicit field within each cube follows trilinear interpolation with respect to the eight cube vertices. Specifically, [40] was the first work to enumerate all possible topological cases with respect to the trilinear interpolant in the cube, and proposed Marching Cubes 33, which contains 33 unique cases under cube rotational and inversion symmetries (inverting all vertex signs), or 31 cases under rotation, mirroring, and inversion; see Figure 4.4(a). In comparison, the original MC algorithm had 15 and 14 cases, respectively.

While correctly enumerating all the topological cases does resolve ambiguities of the tessellations, the many tests required to identify specific topologies present a computational challenge. Lewiner et al. [133] provided an efficient implementation of MC 33 by utilizing an extended look-up table, but left some unresolved issues [64] which were tackled by follow-up work [49]. Van Gelder and Wilhelms [244] pointed out that to avoid non-manifold edges, the triangles in the tessellation templates should not lie in the face of a cube, as in MC 33, and this issue was addressed in later improvements [44, 156]. Specifically, [156] introduced additional vertices on cube faces and interiors, but still followed the trilinearity assumption to place vertices, leading to poor estimates of surface features; see Figure 4.1. To extract sharp features from volume data, prior works typically required additional information, such as the positions and normals of the intersection points between cube edges and the shape, for vertex placement [121]. A key point of NMC is that feature recovery is a *learnable* problem from training meshes. Once trained, our network can accurately predict sharp features and local mesh topologies without any additional input.

4.2.2 Other Isosurfacing Algorithms

Classic isosurfacing algorithms such as dual contouring [110] could preserve sharp features, but it requires the gradients of the intersection points on the edges of the grid cells, which could complicate the input setup. Dual contouring could also produce non-manifold edges, which is an issue addressed later by [213]. In dual marching cubes [214], the implicit function is required as input and queried during reconstruction. There have also been other extensions to dual contouring [59, 216], including works [245, 296] which employ adaptive subdivision for simplification and efficient isosurfacing. Like dual contouring, all these methods require additional inputs such as the gradient information or the function of the underlying implicit field. By learning from training data, NMC can better preserve geometric features given only a uniform grid of sampled scalar values as input.

Marching Tetrahedra (MT) [60] is another variant of MC: it splits each cube into tetrahedra to produce tessellations therein. The tessellation cases for MT are simpler than thoses of MC, but they were also not designed to recover sharp features. Finally, in a recent work called Analytic Marching [131], rather than taking a signed distance field to mesh, the input is implemented as a trained multi-layer perceptron (MLP) with rectified linear units (ReLU). The meshing is then performed by a marching over "analytic cells", which correspond to linear regions resulting from a partitioning by the MLP. Overall, none of the above isosurfacing algorithms learn mesh tessellations from training data.

4.2.3 Neural Geometry Learning

With rapid advances in geometric deep learning, different neural representations have been proposed for 3D shapes. Voxels [43, 264, 92], point clouds [1, 65], and implicit models [38, 163, 109] are among the most popular. But they all require a post processing step to extract a mesh. Deformable meshes/patches [86, 249, 255] can directly output well-tessellated meshes, but they rely heavily on the input mesh templates and are unable to alter their topologies. There are only a handful of works [36, 74, 57, 50] that could output polygonal meshes directly. In contrast, our method takes a discretized implicit field as input and directly outputs a triangle mesh, and therefore could act as the post processing step for many of the above representations.

NMC follows a recent trend in applying machine learning to classical low-level geometry processing tasks including mesh denoising [252], shape transform [286], point cloud upsampling [290], skeletonization [143], and subdivision [148], among many others. In particular, in neural subdivision, Liu et al. [148] proposed a graph neural network to perform geometry-aware subdivision on triangle meshes. It recursively subdivides an input mesh by applying classic loop subdivision, while the positions of the newly added vertices are predicted by a neural network conditioned on local geometry.

In terms of combining machine learning and MC, two notable works, Deep Marching Cubes (DMC) [139] and MeshSDF [201], both aim to make MC *differentiable*. Specifically, DMC learns a differentiable approximation of MC by predicting probabilistic occupancies and vertex displacements, while MeshSDF adopts a continuous model of how signed distance function perturbations locally impact surface geometry to obtain a differentiable surface parameterization. Another work, DefTet[71], shares a similar spirit as DMC, as it reconstructs tetrahedral meshes by predicting occupancies in an initial tetrahedral grid, and deforming the vertices to approximate the output shape.

Our work differs from DMC and MeshSDF in several major ways. First, our goals are different. The goal of DMC and MeshSDF is to directly obtain an explicit mesh representation from discrete raw inputs, e.g., point clouds, voxels, or images, in an end-to-end trainable manner, while NMC builds a framework to make MC learnable from training meshes. The input to NMC is a discrete implicit field of distances or occupancies obtained by any model, neurally or not. Second, the focus of DMC and MeshSDF is the end-to-end differentiability, while our focus is on designing and training a refined neural MC model to better reconstruct geometry and topology, in particular sharp features, unlike any other previous MC variant or differential extension. Case in point, DMC only adopted 8 out of the 15 mesh tessellation templates from the original MC [157] that have singly connected topologies, falling far short of [156] and NMC in terms of topological granularity. Last but not the least, DMC and MeshSDF rely on global features to predict the output shapes, not aiming to generalize to other shape categories not present in the training set. In contrast, our network employs a limited receptive field for each cube, leading to a more robust and general isosurfacing algorithm.

4.3 Neural Marching Cubes

We detail our representation for performing Neural Marching Cubes (NMC). We first introduce in a 2D example how the local topologies and tessellations in a square can be represented with a fixed-length code of binary values and float numbers; see Figure 4.2. Then we expand the representation into the 3D cube for NMC, as shown in Figure 4.3. We show how to design the mesh tessellations with respect to our representation and how our training data could be generated from raw meshes. Finally, we describe the network architecture and objective functions we designed to train NMC.

4.3.1 2D NMC: representation in a 2D square

We follow the common assumption in MC that if the two vertices of an edge in a cube (or a square) have different signs, there will be one and only one intersection point between the edge and the underlying zero-isosurface. As a result, all the situations in a square can be enumerated as in Figure 4.2(a). However, the tessellation templates in the classic MC algorithms are unable to represent geometric features such as sharp edges by design, hence they must be re-designed. Simply adding one vertex on each generated edge (black line) of the original templates could already solve the issue. But since we want a neural network to fully predict the meshing in each square, including the added vertices, we need to design a representation with a fixed format to store all the necessary information, so that the representation could be directly parsed into an output mesh, while being compatible with neural processing and training.

First, we add four face vertices ($\mathrm{v}_1^{\mathrm{f}} \sim \mathrm{v}_4^{\mathrm{f}}$) on the face, each associated with one corner vertex ($\mathrm{v}_1 \sim \mathrm{v}_4$), as shown in Figure 4.2(c) left. With the added vertices, new tessellations that can better preserve geometric features can be easily derived; see Figure 4.2(b).

Next, we design a fixed-length vector to fully encode the output mesh (edges) in each square; see Figure 4.2(c). The vector is split into a *boolean* or *binary part* to describe the topological cases, and a *float part* to store floating point numbers as vertex positions.

When the signs of the four corner vertices are given, there is only one ambiguity case, when both ends of a diagonal line are with the same signs but the ends of an edge are with different signs; see top-right corner of Figure 4.2(b). This ambiguity can be resolved by adding a face sign which is positive if the connected vertices are positive, and vice versa. Hence, the boolean part has 5 values storing the signs: one face sign and four signs for the corner vertices. On the other hand, we need to store all vertex positions in the float part, whether the vertices are being used or not. Since each edge vertex only has one degree of freedom, four edge vertices take 4 floats to store. Adding the 8 numbers for the 2D coordinates of the four face vertices, in total we have 12 numbers in the float part.

However, note that the representation for each square is not the same as the representation we use to predict the entire 2D shape, because of the redundancy: an edge vertex is shared by two adjacent squares, and a corner vertex is shared by four. Therefore, when representing the squares of an entire shape, we only need to store the sign of one corner vertex (v_1) and the face sign in the boolean part, and two edge vertices ($\mathrm{v}^{\mathrm{e}_1}$,$\mathrm{v}^{\mathrm{e}_4}$) and all four face vertices in the float part, leading to 2d and 10d vectors, respectively. Afterwards, a 2D Convolutional Neural network (CNN) could be applied to take as input an $M \times N$ array of implicit field values, and output an $M \times N \times 12$ array that is parsed into a 2D mesh.

4.3.2 3D NMC: representation in a 3D cube

As shown in Figure 4.3, we design the NMC representation for a 3D cube in a similar manner as its 2D counterpart shown in Figure 4.2. In addition to the edge vertices ($\mathrm{v}^{e_1} \sim \mathrm{v}^{e_{12}}$), we keep the four added face vertices for each of the six faces of the cube. We also add eight interior vertices ($\mathrm{v}_1^{\mathrm{c}} \sim \mathrm{v}_8^{\mathrm{c}}$) in the cube, each affiliated with one corner vertex ($\mathrm{v}_1 \sim \mathrm{v}_8$) of the cube. Details on how to properly tessellate the cube with these new vertices will be discussed in Section 4.3.3. In this section, we focus on how to represent the topological cases and the positions of the added vertices, using a boolean and a float part respectively, as shown in Figure 4.3(c).

For the boolean part, we require at least eight signs at the corner vertices of the cube and six face signs to describe or index the local mesh topology in each cube. However, these are not sufficient to resolve all ambiguities. As already observed in MC 33 [40], when there are two connected components with the same sign on the surface of a cube, the two could be connected with a tunnel inside the cube. The real situations could be far more complicated

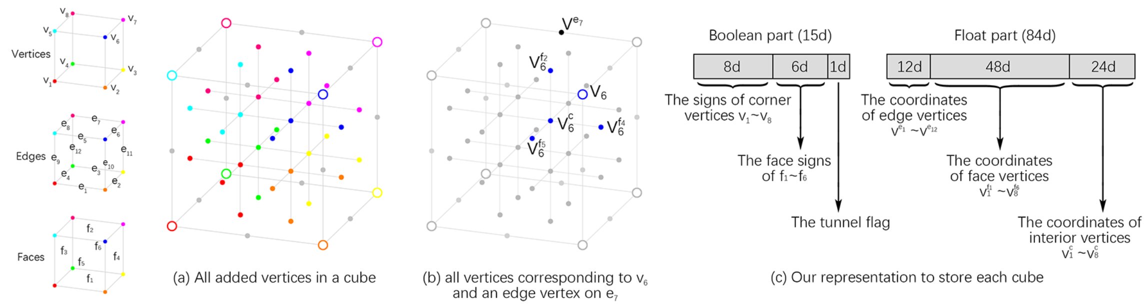

Vertices
Edges
Faces
(a) All added vertices in a cube
(b) all vertices corresponding to v₆ and an edge vertex on e₇
Boolean part (15d)
Float part (84d)
8d
6d
1d
12d
48d
24d
The signs of corner vertices v₁~v₈
The face signs of f₁~f₆
The tunnel flag
The coordinates of edge vertices
The coordinates of face vertices
The coordinates of interior vertices
(c) Our representation to store each cube

Figure 4.3: Per-cube parameterization for our NMC in 3D, with 12 edge vertices (v^{e_i}), $6 \times 4 = 24$ new face vertices ($v_j^{f_k}$), along with 8 new interior vertices (v_j^c), as shown in (a). Vertices with the same color correspond to the same cube vertex, as shown in (b), where the grey lines in (a-b) are for ease of visualization only. The vector representation for local mesh topology (the boolean part) and vertex positions (the float part) is given in (c), where the number of floats needed to represent a vertex depends on the degrees of freedom, e.g., one for an edge vertex, two for a face vertex, and three for an interior vertex.

than that. There could be more than two connected components on the surface of the cube, and there could be more than one tunnel to connect the two components. To simplify the situation, we draw inspiration from the topological cases in MC 33, which are shown in Figure 4.4(a). Note that all the 33 cases have either zero or one tunnel, if there are exactly two connected components with the same sign. Therefore, we assume that in the case of two connected components, there could be one tunnel connecting the components, or none at all. In other cases with just one or more than two connected components, we assume zero tunnel. With such a simplifying assumption, we only need to add one binary value to indicate whether a tunnel exists, leading to a total of 15 binary values in the boolean part.

For the float part, we need to store 44 added vertices: 12 edge vertices, $6 \times 4 = 24$ face vertices, and 8 interior vertices, as shown in Figure 4.3(a-b). Since each edge vertex only has one degree of freedom, 12 edge vertices would only need 12 floats to store. Each face vertex has two degrees of freedom, therefore 24 face vertices take 48 floats to store. Plus the 24 floats for the 3D coordinates of the 8 interior vertices, in total we have 84 numbers in the float part.

However, similar to the 2D cases, the representation for each cube is not the same as the representation we use to predict the entire 3D shape, because of the redundancy: a corner vertex is shared by eight cubes, an edge vertex is shared by four, and a face vertex is shared by two. Therefore, when representing the cubes of an entire shape, we only need to store 5 values in the boolean part (the sign of v_1, f_1, f_3, f_5; and the tunnel flag), and 51 values in the float part (3 edge vertices on edges e_1, e_4, e_9; 12 face vertices on faces f_1, f_3, f_5; and all 8 interior vertices). Afterwards, a 3D CNN can be applied to take as input an $M \times N \times P$ array of implicit field values, and output an $M \times N \times P \times 56$ array that could be parsed into a 3D mesh.

4.3.3 3D NMC: tessellating a 3D cube

In this section, we elaborate how we tessellate the cube with respect to each topological case. To facilitate the tessellation design, we developed a graphical user interface for interactive modelling, and employed the interface to design and render all the cases as shown in Figures 4.4 and 4.5.

Generally, the tessellation design needs to comply with several basic principles. First, the resulting mesh should contain only triangles. Second, the mesh should completely separate vertices with different signs, i.e., any path inside the cube that connects two vertices of different signs must intersect with the mesh. Third, there should be no non-manifold edges or vertices. Specific to our method, there is a fourth principle to follow: we are only allowed to use the vertices present in our cube representation as described in Section 4.3.2.

However, the very first step we need to take before designing the tessellations, is to enumerate how many unique topological cases there are. In our cube representation, we have 15 binary values to describe the cases, therefore we have a total of $2^{15} = 32,768$ cases. Yet, if we consider the presence of rotational symmetries, mirroring symmetries, inversion symmetries (inverting all vertex and face signs in a cube), and remove all invalid cases with respect to the tunnel flag, we have a total of **37** *unique cases.*

We can directly use the face tessellations in Figure 4.2 to tessellate the six faces of a cube, as shown in Figure 4.5(b). Afterwards, we follow several guidelines to create a mesh inside the cube: a) If there is a tunnel, then the tunnel must contain the small center cube made by the eight interior vertices; b) if there is no tunnel, then connect all available face vertices to their corresponding interior vertices; c) avoid long triangles. See Figure 4.6 for several examples.

The completed tessellations of our method can be found in Figure 4.5(a). The tessellation design allows much freedom and does not necessarily have to follow our guidelines. For example, we could simply take the tessellations in [156] into our framework, as show in Figure 4.4(b). Since this tessellation design employs fewer vertices and triangles, we coin our Neural Marching Cubes using this specific tessellation design as *NMC-lite.* Note that in both cases, the training data will be prepared and the network will be trained with respect to their own tessellation designs.

4.3.4 Data preparation

We now introduce data preparation for NMC, i.e., the preprocessing step to convert a raw mesh into a form compatible with our cube representation in Figure 4.3 for neural processing; it is an $M \times N \times P \times 5$ array for the boolean part and an $M \times N \times P \times 51$ array for the float part, when the input grid is $M \times N \times P$. We divide a raw 3D mesh into small cubes to process each separately, as in MC. For each cube, we first determine its topological case. Then we put the corresponding tessellation template inside that cube, and optimize

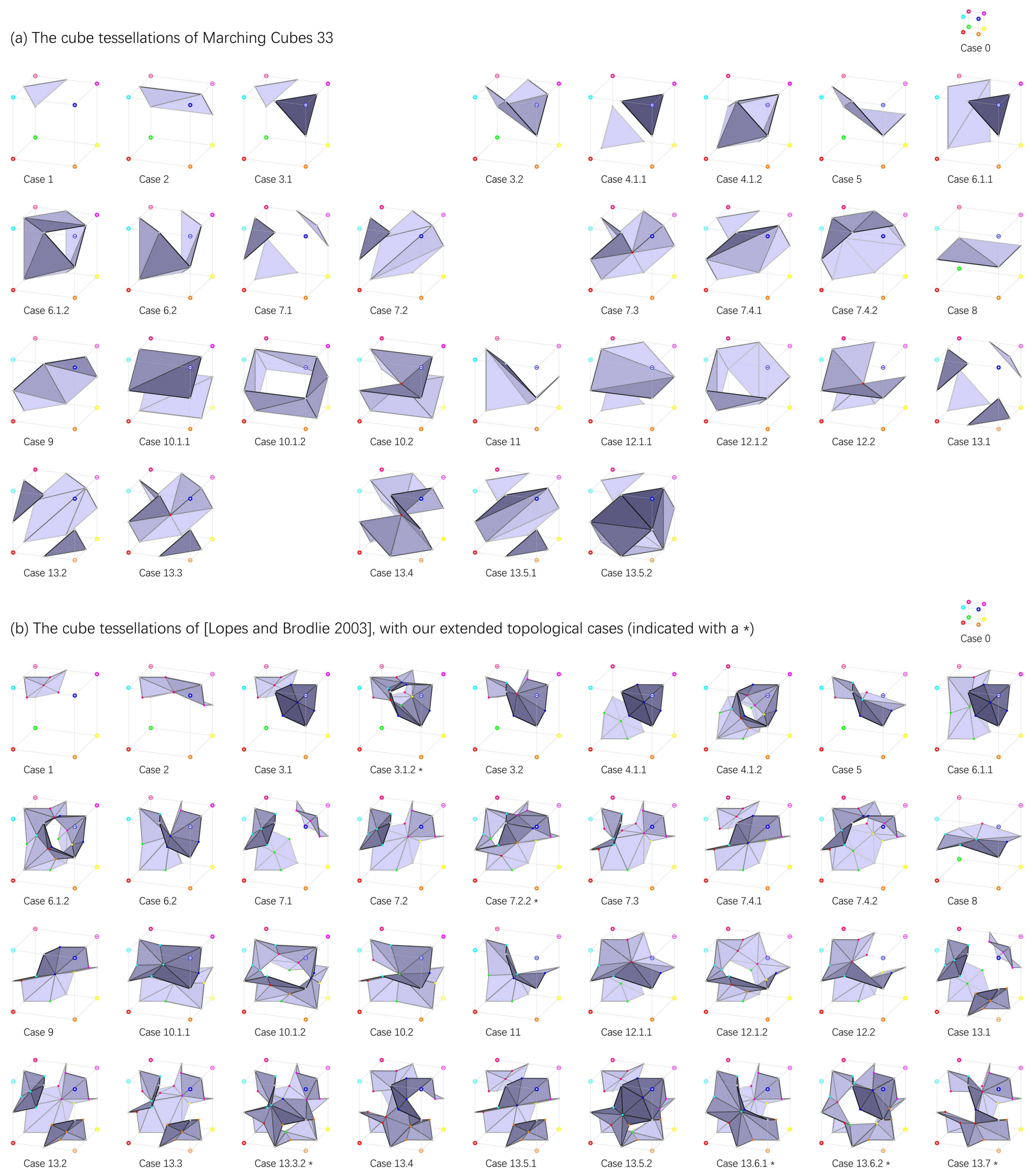
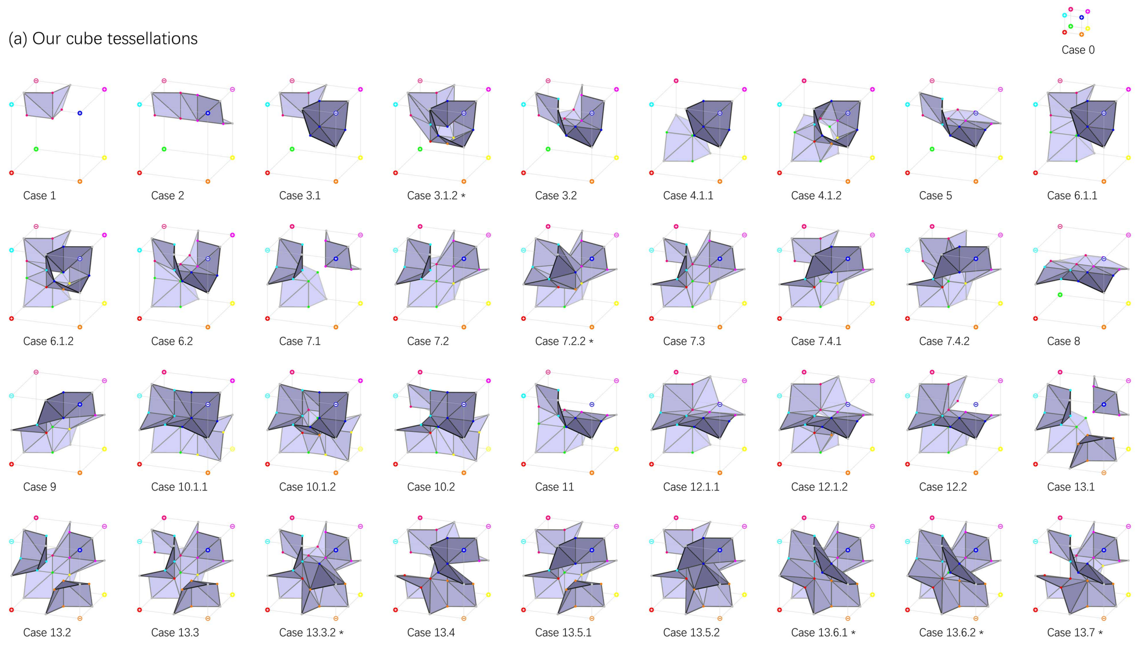


Figure 4.4: The 3D cube tessellations of Marching Cubes 33 [40] and [156]. Note that they both present 31 cases, since Case 12.3 is equivalent to Case 12.2 and Case 14 is equivalent to Case 11, with respect to rotational and mirroring symmetries. In (b), we also add our extended topological cases to [156], indicated with a *, to form a simplified version of our NMC tessellations, denoted as NMC-lite.



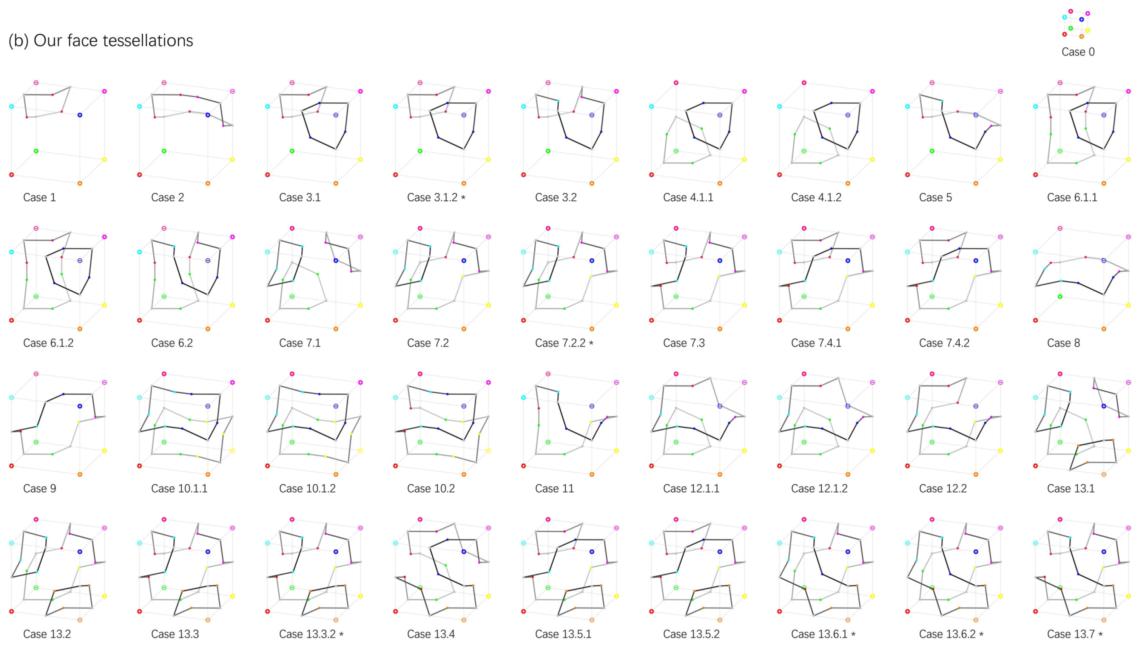


Figure 4.5: Our cube tessellations and face tessellations for all the 37 unique topological cases considered by NMC, where vertices with the same color correspond. Note Case 0 in the top-right corner which indexes the case where all signs on the cube vertices are the same.

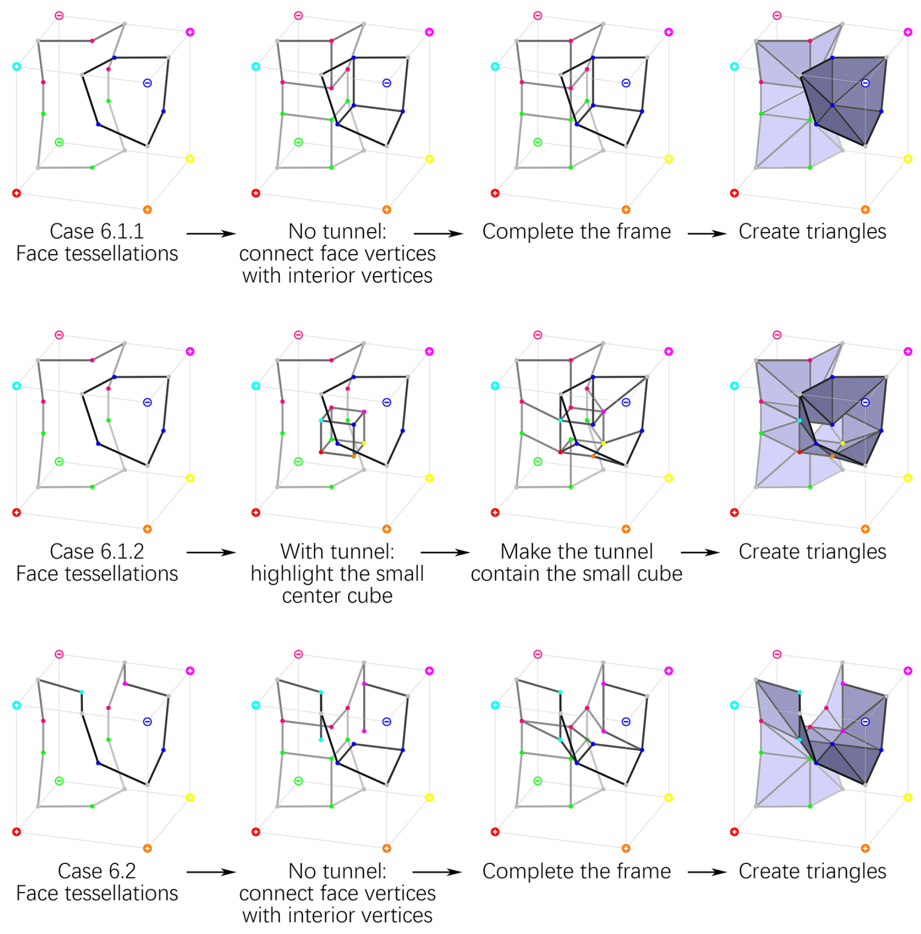


Figure 4.6: Example tessellation steps for our NMC designs. The face tessellations in the first column follow Figure 4.2, therefore they are considered as "given", and we only need to add new structures inside the cube.

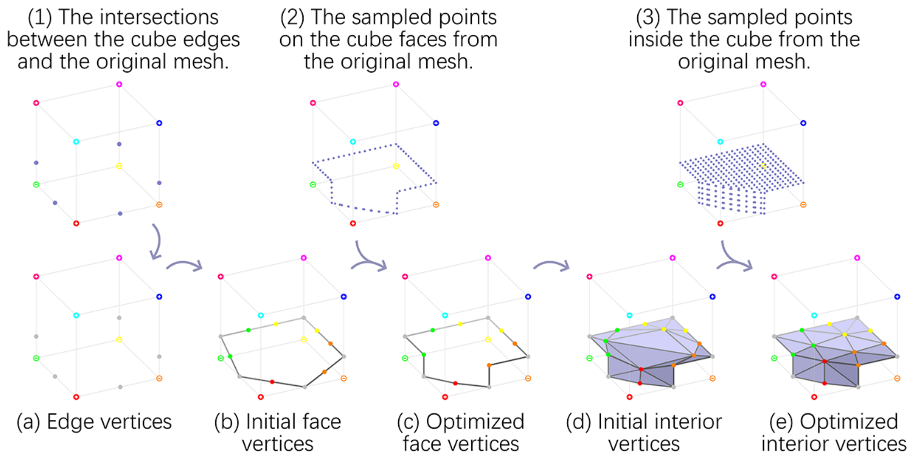


Figure 4.7: Our preprocessing step to prepare the training mesh data for NMC. After determining the topological case for the cube, we optimize the vertex positions to approximate the original mesh. The initial face vertices are mid-points or trisection points, while the initial interior vertices in the cube are averages of connected edge vertices and face vertices.

the vertices of the tessellation template to minimize the Chamfer Distance with respect to the original mesh. An overview is given in Figure 4.7.

To determine the topological case in a cube, we compute a $9 \times 9 \times 9$ grid of signed distances inside the cube, so that each face contains 9×9 signed distances. We then check the connectivities between the vertices through the SDF grid, where adjacent grid points with the same sign are considered connected, to determine the case for each of the six faces. After the face cases are determined, we only need to test whether there is a tunnel to determine the cube case, which can be done by checking the number of connected components inside the cube. Note that in several situations, the cube cannot be represented with our templates, e.g., when there are two or more intersections on a cube edge, or when there are complex structures inside the cube that are unaccounted for.

We perform tests to validate whether an edge/face/cube can be represented using our templates by checking the number of connected components, which are compared against the templates in Figure 4.2 (for faces in the 2D case) or Figure 4.5 (for cubes in 3D). The edges/faces/cubes that do not have matching numbers are deemed to be invalid. For example, if the end vertices (grid points) of an edge are with different signs, then the 9 grid points on the edge should contain exactly two connected components, one positive and one negative. In a 3D cube, say Case 6.1.1, there are three connected components, one positive and two negatives, while in Case 6.1.2, there are two connected components because of the tunnel.

The removal of invalid edges/faces/cubes from the training meshes is critical to NMC and this is accomplished by a *masking* process. Specifically, we generate masks during data preparation to indicate valid values in our representation with **1's** and invalid or irrelevant values with **0's**, where invalidity implies that the edge/face/cube cannot be represented by our designed tessellation templates, and a value stored in our representation is irrelevant if it does not affect the output mesh (e.g., the face sign in an unambiguous face, or the tunnel flag in a cube that cannot form a tunnel). For shape s, we denote the input array as $I_s \in \mathbb{R}^{M \times N \times P}$, the array of the boolean part as $B_s \in \{0,1\}^{M \times N \times P \times 5}$, and the array of the float part as $F_s \in [0,1]^{M \times N \times P \times 51}$. Therefore, the mask of B_s is $M_{B_s} \in \{0,1\}^{M \times N \times P \times 5}$ and the mask of F_s is $M_{F_s} \in \{0,1\}^{M \times N \times P \times 51}$.

After the topological case is settled, we put the corresponding tessellation template inside the cube and optimize its vertices to approximate the original mesh. However, since adjacent cubes share edges and faces, we first determine the positions of all edge vertices, then all face vertices, and finally all interior vertices, to avoid repeated computations. Note that only the edge vertices do not require optimization since we can find them by checking the intersection points between the cube edges and the original mesh. Take the interior vertices for example – while the face vertices can be optimized in a similar way, we densely sample points on the mesh inside the cube to obtain a point cloud P. Denote the vertices, edges, and triangles in the tessellation template as V, E, and T, respectively. Denote the point-triangle Euclidean distance as $D(p,t)$, and the point-point Euclidean distance as $d(v_1,v_2)$, we have the objective function

$$L_{total} = L_{P\to T} + L_{T\to P} + \gamma L_{reg}, \text{with} \tag{4.1}$$

$$L_{P\to T} = \frac{1}{|P|}\sum_{p\in P}\min_{t\in T} D(p,t), \tag{4.2}$$

$$L_{T\to P} = \frac{1}{|T|}\sum_{t\in T}\min_{p\in P} D(p,t), \tag{4.3}$$

$$L_{reg} = \frac{1}{|V|}\sum_{v_1\in V}\min_{\{v_1,v_2\}\in E} d(v_1,v_2), \tag{4.4}$$

where $L_{P\to T}$ is the point-to-triangle distance, $L_{T\to P}$ the triangle-to-point distance, and L_{reg} a regularization term to keep the surface as "tight" as possible by minimizing edge lengths, and γ is set to 0.1.

Note that it is possible to use the above objective function to train the network directly, instead of using a mean squared error loss as we will describe in the next section. However, to ensure the quality of the generated mesh, we usually sample a very dense point cloud to peform the optimization. The computational time and memory cost make it intractable to train the network directly with the optimization objectives.

4.3.5 NMC network and objective functions

The input to our network is an array of implicit field values I_s, and the ground truth outputs contain a boolean array B_s and a float array F_s. The masks M_{B_s} and M_{F_s} indicate which values in B_s and F_s are valid. Since they are all 3D arrays (with feature channels), we could directly apply 3D convolutional neural networks for the task. Specifically, we use a 3D variant of ResNet [95] as our backbone network, with receptive fields of size 7^3.

For the objective functions, we use a binary cross entropy (BCE) loss for the boolean part and a mean squared error (MSE) loss for the float part. Denote the outputs of our network as $D_s = f_B(I_s)$ and $H_s = f_F(I_s)$ for the boolean part and float part, respectively, and denote the entire shape dataset as $\mathcal{S}$. Let all multiplications in the following equations be element-wise products, then we have

$$L_{bool} = \mathbb{E}_{s\sim\mathcal{S}}||-M_{B_s}(B_s\log(D_s) + (1-B_s)\log(1-D_s))||_1, \tag{4.5}$$

$$L_{float} = \mathbb{E}_{s\sim\mathcal{S}}||M_{F_s}(F_s - H_s)||_2^2. \tag{4.6}$$

We could directly sum L_{bool} and L_{float} with a weighting parameter to obtain the final objective function. However, our experiments showed that it is tedious to find an appropriate weight for the two terms. Therefore, we choose to use two distinct networks to predict D_s

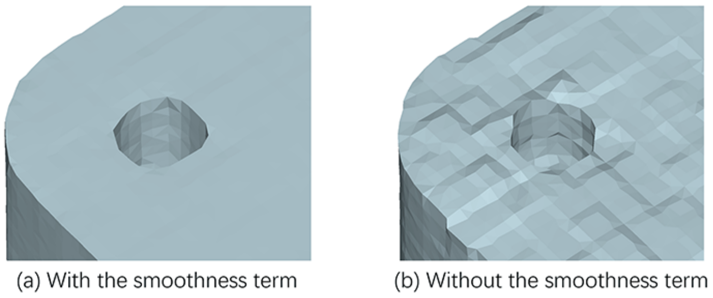


Figure 4.8: Output meshes when our network is trained with vs. without the smoothness term when the inputs are binary voxel/occupancy grids.

and H_s separately, so that one network is trained with L_{bool} and another with L_{float} without any interference.

However, the above settings are not sufficient for binary voxel inputs, due to considerable ambiguities of the possible shapes represented by the input voxels. Therefore, we use the aforementioned settings for SDF grid inputs, and make a few adjustments when the inputs are binary voxels. Specifically, we enlarge the receptive fields of our backbone network from 7^3 to 15^3 to reduce ambiguity, and add a smoothness term to the loss function on the float part,

$$L^*_{float} = L_{float} + \gamma L_{smooth}, \text{with}$$
$$L_{smooth} = \mathbb{E}_{s \sim \mathcal{S}} \sum_{(u,v) \in E_s} ||\mathbb{1}(|F^u_s - F^v_s| < \sigma) \cdot (H^u_s - H^v_s)||^2_2, \tag{4.7}$$

where E_s denotes the set of all edges (u, v) in the ground truth (GT) output mesh for shape s, $F^u_s \in \mathbb{R}^3$ is the coordinates of vertex u in the GT mesh, and $H^u_s \in \mathbb{R}^3$ is the coordinates of u in the *predicted* mesh. Note that the mesh tessellations of the GT mesh and the predicted mesh are the same since the tessellations are determined solely by the boolean part, and we use the GT boolean part when training the float part. In addition, $\mathbb{1}(\cdot)$ in the equation converts true/false into 1/0, respectively, and the parameters $\sigma = 0.0002$ and $\gamma = 10$ are fixed throughout our experiments.

Overall, the smoothness term encourages the output surfaces to align with the *coordinate axes*, with the underlying assumption that the GT surfaces generally share the same characteristic. We show the impact of L_{smooth} in Figure 4.8 and explain this choice by experimenting with different smoothness terms in the experiments.

4.4 Results and evaluation

In this section, we show results and evaluate NMC both qualitatively and quantitatively, on both SDF and binary voxel inputs. We compare NMC to well-known MC variants, and demonstrate its generalizability and the ability to handle noisy input.

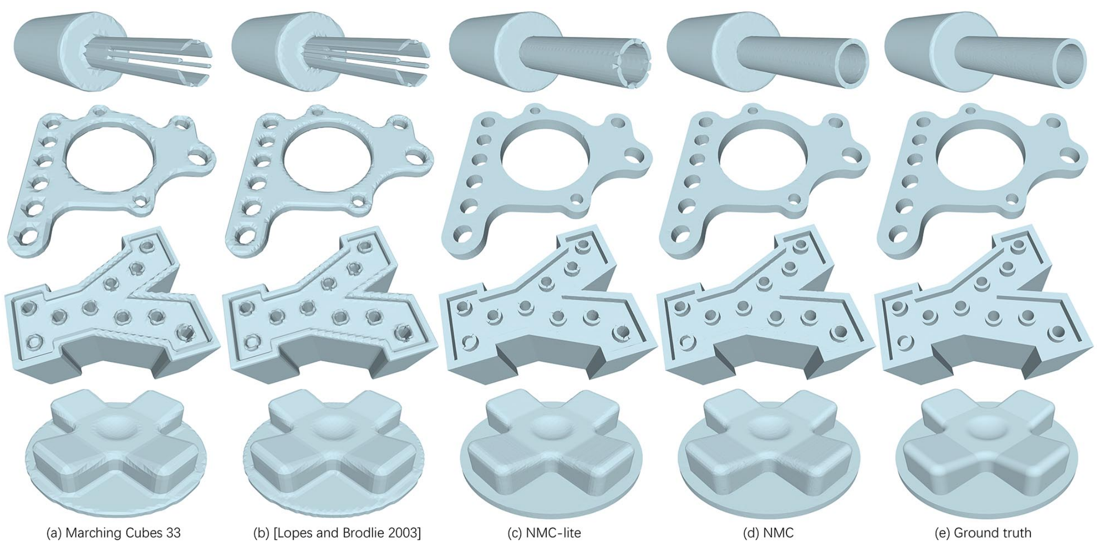


Figure 4.9: Results of reconstructing 3D meshes from SDF grid inputs at 64^3 resolution. The shapes in the first two rows are from the ABC test set, and the last two rows from Thingi10K.

Datasets. For our experiments, we mainly work with the first chunk of the ABC dataset [122], which consists of triangle meshes of CAD shapes. Such CAD shapes are characterized by their rich geometric features (e.g., sharp edges, corners, smooth curves, etc.) and topological varieties. We split the set into 80% training (4,280 shapes) and 20% testing (1,071 shapes). During data preparation, we obtain triangle meshes over 32^3 and 64^3 grids to train our network. Evaluation is conducted on the testing set. Later, to assess the generalizability of NMC, we also test (not train) the same network on the Thingi10K dataset [302], which contains a variety of 3D-printing models.

Evaluation metrics. To perform quantitative evaluation, we sample 100K points $\mathbf{S} = \{\mathbf{s}_i\}$ uniformly distributed over the surface of a shape, and then use Chamfer Distance (CD) and F-score (F1) to evaluate the overall quality of a reconstructed mesh, and Normal Consistency (NC) to evaluate the quality of its surface normals.

Inspired by BSP-Net [36], we employ Edge Chamfer Distance (ECD) and Edge F-score (EF1) to evaluate the preservation of sharp edges. We use the same "sharpness" definition in BSP-Net as $\sigma(\mathbf{s}_i) = \min_{j \in \mathcal{N}_\varepsilon(\mathbf{s}_i)} |\mathbf{n}_i \cdot \mathbf{n}_j|$, where $\mathcal{N}_\varepsilon(\mathbf{s})$ extracts the indices of the points in $\mathbf{S}$ within distance ε from $\mathbf{s}$, and $\mathbf{n}_i$ is the surface normal at point $\mathbf{s}_i$. We compute an "edge sampling" of the surface by retaining points for which $\sigma(\mathbf{s}_i) < 0.2$. Given two shapes, the ECD and EF1 between them are simply the CD and F1 between the corresponding edge samples. We also count the number of generated vertices (#V) and triangles (#T) to reveal the fidelity-complexity trade-off.

128^3 resolution	CD($\times 10^5$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	4.143	0.870	0.972	4.063	0.193	22,048.41	44,107.11
64^3 resolution	CD($\times 10^5$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	4.850	0.788	0.950	5.736	0.105	5,472.51	10,953.67
Lopes2003	4.803	0.798	0.958	6.841	0.100	21,979.95	43,892.05
Trilinear	4.733	0.803	0.960	7.275	0.098	-	-
NMC-lite	4.341	**0.877**	**0.975**	**0.382**	**0.759**	22,710.56	43,876.87
NMC	**4.323**	**0.877**	**0.975**	0.390	0.758	42,766.54	85,543.83
32^3 resolution	CD($\times 10^4$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	5.239	0.570	0.900	5.504	0.048	1,297.38	2,595.47
Lopes2003	5.343	0.577	0.911	6.213	0.047	5,215.12	10,397.68
Trilinear	5.161	0.585	0.915	7.217	0.045	-	-
NMC-lite	3.922	0.823	**0.950**	**0.532**	0.631	5,464.48	10,389.43
NMC	**3.919**	**0.824**	0.949	0.598	**0.634**	9,728.20	19,460.09

Table 4.1: Quantitative comparison results on ABC test set with SDF input.

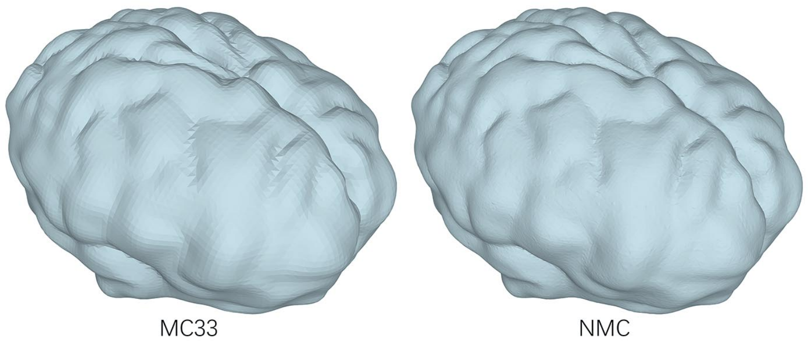


Figure 4.10: Reconstruction results on a brain model (in Thingi10K) with smooth features by MC33 and NMC, from SDF inputs. NMC preserves the surface details (especially around the valley areas) better.

128^3 resolution	CD($\times 10^5$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	4.533	0.985	0.984	0.892	0.383	12,551.21	25,076.50
Lopes2003	4.487	0.985	0.986	0.858	0.409	50,649.41	100,417.26
NMC-lite	**3.696**	**0.992**	**0.987**	**0.559**	**0.628**	50,205.72	100,401.08
NMC	3.706	**0.992**	**0.987**	0.625	**0.628**	83,023.47	166,036.10

Table 4.2: Quantitative comparison on organic FAUST shapes with SDF input.

Mesh extraction from SDF grids. We first compare the two versions of our method, NMC and NMC-lite, with the two best-known MC variants to date, Marching Cubes 33 [133] (MC33) and [156] (Lopes2003), on the task of mesh extraction from grids of SDF values. Quantitative comparison results are reported in Table 4.1, with two choices of input resolutions: 64^3 and 32^3, on the ABC test set. The results show that, with the same SDF inputs, our method outperforms MC33 and Lopes2003 on all the quantitative measures considered.

We also add a row (top row in Table 4.1) for MC33 with the input resolution upscaled to 128^3. As we can see, even with $8\times$ the amount of input information as NMC and NMC-lite, MC33 *underperforms* on all measures except for CD. In terms of edge preservation, our method is superior. This is also verified by the visual results in Figure 4.11, comparing MC33 on 128^3 input and NMC on 64^3 input.

Figure 4.9 shows qualitative comparisons between the various methods, on sample inputs from the ABC test set and Thingi10K. We can observe that NMC, and to a lesser extent, NMC-lite, are the only methods that can recover sharp edges and corners, while the smooth curves are also well preserved. In fact, our method can reconstruct both sharp and soft edges well, as demonstrated in the last row. Furthermore, examples in the first row and the third row (at a smaller scale) exhibit thin structures in a shape, which causes both MC33 and Lobes2003 to produce incorrect local topologies, due to the trilinear interpolant assumption. On the other hand, our method infers the correct topological cases — the ambiguous Case 10.1.1 (see Figure 4.5), resulting in more faithful reconstructions.

In Figure 4.1(c), we show the isosurface of a trilinear interpolant, and in Table 4.1, we report quantitative results associated with trilinear interpolation. The “ground truth” trilinear interpolant could be considered as the upper bound of all methods that follow the trilinearity assumption. Therefore, our method outperforming the trilinear interpolant further proves that NMC is fundamentally superior at feature-preserving isosurface extraction.

Organic shapes. In Figure 4.10, we show that when the ground truth shape has smooth undulations, our method is still able to reconstruct the surface details better than MC33. For a more comprehensive test on organic shapes, we compare the various methods on 100 meshes of human body shapes from the FAUST dataset [19]. The quantitative results in Table 4.2 show that NMC and its variant can learn to predict both smooth and sharp features well, outperforming both MC33 and Lopes2003. Augmenting the training set with more organic shapes should further improve performance on such inputs, since our method is data-driven.

Varying input grid resolutions. In Figure 4.11, we show how MC33 and NMC perform as the input SDF resolutions vary from 8^3 to 128^3, where we recall that our network was trained on meshes obtained at 32^3 and 64^3 resolutions. It is clear that our method can easily

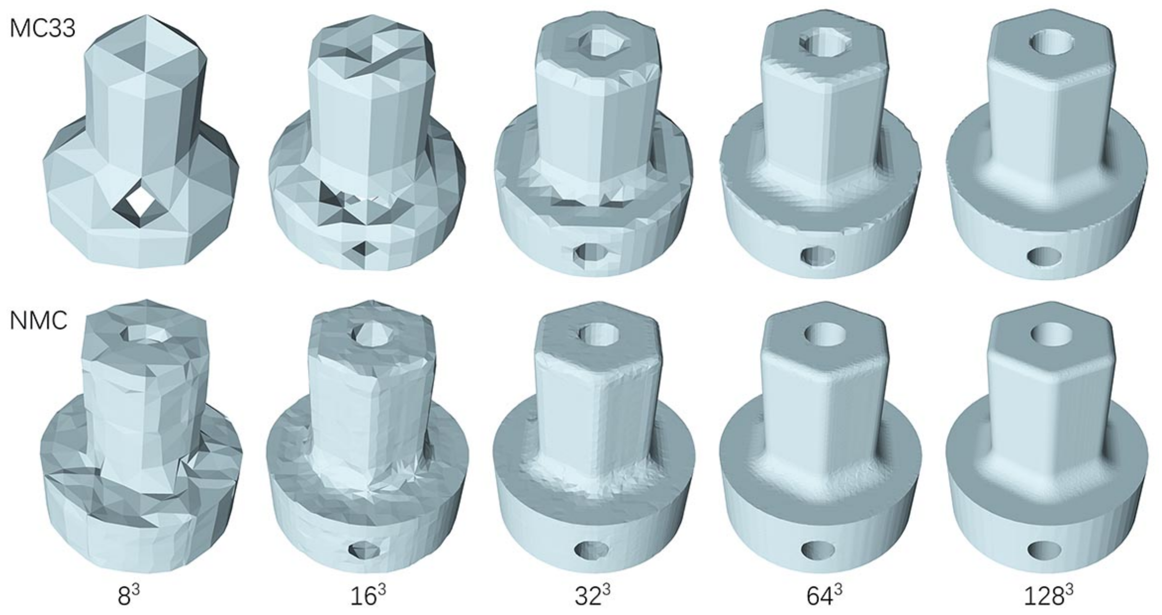


Figure 4.11: Results of reconstructing 3D mesh shapes from SDF inputs as the input grid resolutions vary. The holes in the MC33 results are due to incorrectly predicted topological cases. NMC consistently outforms MC33 at all input resolutions, up to 128^3, but with a “diminishing margin".

adapt to higher-resolution inputs, but degrades in reconstruction quality at the lower end. This is expected since as the cube size grows relative to the shape, the geometric varieties inside the cubes would surpass the set of topological cases covered by NMC. Nevertheless, NMC appears to consistently outperform MC33 at all resolutions, up to at least 128^3. As the resolution continues to grow however, the difference between NMC and MC33 will diminish since the topological cases per cube would be much simplified.

Mesh extraction from binary voxels. When the inputs are binary voxels instead of signed distances, the isosurface extraction task becomes significantly more difficult, since voxel occupancies possess considerably less information. Not only are the point-to-surface distances lost in the occupancies, but the signs could also be inaccurate: a point outside the shape in the SDF grid may become “inside” in the voxel grid. One can observe a quality drop from the visual results shown in Figure 4.12. Even our method cannot always produce faithful reconstructions due to the ambiguities, e.g., the rod in the first row could be rectangular or circular, and the edges in the second row could be smooth or sharp - both would yield identical voxel grids. However, our learning-based approach is able to narrow down the possible geometries and topologies by observing local neighborhoods. As shown by the quantitative results in Table 4.3, our method outperforms other MC variants and the trilinear interpolant on all measures, except for NC, by a substantial margin.

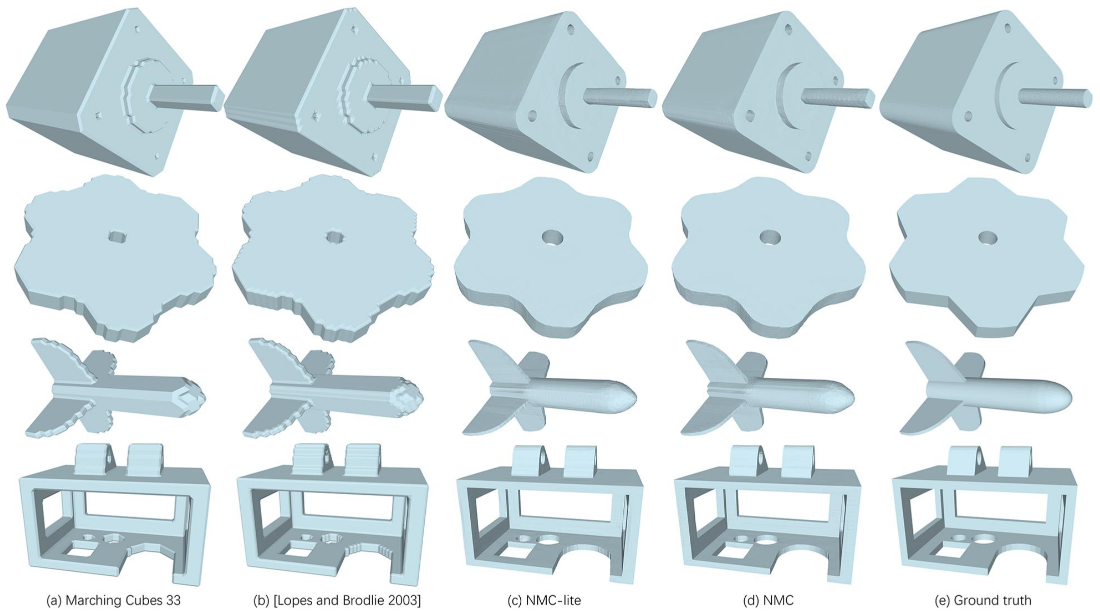


Figure 4.12: Results of reconstructing 3D meshes from binary voxel/occupancy inputs at 64^3 resolution. The shapes in the first two rows are from the ABC test set, and the last two rows from Thingi10K.

64^3 resolution	CD($\times 10^5$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	26.860	0.085	0.921	11.196	0.018	5,826.08	11,655.52
Lopes2003	26.829	0.084	0.921	14.601	0.017	23,302.73	46,608.90
Trilinear	26.826	0.084	0.921	14.866	0.017	-	-
NMC-lite	**9.302**	**0.443**	0.930	0.559	**0.365**	22,185.94	42,915.64
NMC	9.341	0.438	**0.931**	**0.528**	0.356	42,043.03	84,087.85
32^3 resolution	CD($\times 10^4$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	9.636	0.036	0.882	11.764	0.018	1,532.70	3,065.30
Lopes2003	9.632	0.036	0.883	14.723	0.017	6,130.84	12,261.58
Trilinear	9.641	0.035	**0.884**	14.820	0.017	-	-
NMC-lite	**5.909**	**0.237**	0.871	**0.901**	**0.112**	5,236.79	9,975.67
NMC	6.029	0.232	0.871	0.910	0.109	9,469.84	18,933.65

Table 4.3: Quantitative comparisons on ABC test set with binary voxel input.

64^3 resolution	CD($\times 10^5$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	3.195	0.795	0.945	3.763	0.099	5,517.51	11,044.35
Lopes2003	3.084	0.805	0.953	4.567	0.087	22,224.23	44,135.98
Trilinear	3.076	0.811	0.956	5.211	0.084	-	-
NMC-lite	**2.470**	**0.893**	**0.972**	0.330	**0.722**	22,991.80	44,109.17
NMC	2.477	**0.893**	**0.972**	**0.312**	**0.722**	40,951.73	81,910.41
32^3 resolution	CD($\times 10^4$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$	#V	#T
MC33	10.519	0.540	0.882	4.046	0.040	1,284.98	2,569.73
Lopes2003	10.473	0.547	0.893	4.596	0.038	5,163.28	10,281.15
Trilinear	10.431	0.555	0.897	5.180	0.037	-	-
NMC-lite	**8.425**	0.807	**0.935**	0.600	**0.542**	5,423.92	10,263.13
NMC	8.454	**0.808**	0.933	**0.596**	0.539	9,161.94	18,327.88

Table 4.4: Quantitative comparison results on Thingi10K with SDF input.

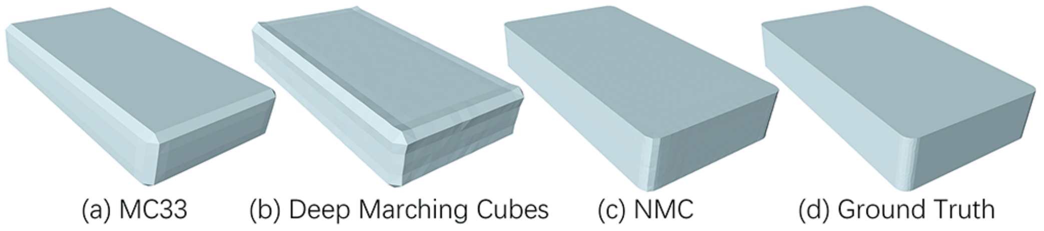


Figure 4.13: Comparing NMC with MC33 and Deep Marching Cubes (DMC) [139] on feature preservation.

Generalizability. To demonstrate generalizability of our networks, which were trained on ABC, we test them on the first 2,000 valid shapes from Thingi10K, using exactly the same network weights as those in the previous experiments. Tables 4.4 and 4.5 show quantitative comparison results on SDF and binary voxel inputs, respectively. Some qualitative results are given in Figures 4.9 and 4.12. We can observe a similar performance boost over the other methods in comparison. Note however that in Table 4.5, our method does not significantly outperform other methods at the 32^3 input voxel resolution. This may be due to NMC being overfit to the ABC training set, since the shape resolution (32^3) is getting close to the size of the receptive field of our voxel processing network (15^3).

Comparison to DMC In Figure 4.13, we compare NMC with DMC [139] on feature-preserving mesh reconstruction. Since the network architecture of DMC is not designed to perform general isosurface extraction, we train their network to *overfit* on a single input shape with 65,536 uniformly sampled points as supervision. As we can see, even with such an overfitting, DMC is still unable to recover sharp features, which is mainly due to its adoption of only few classical MC tessellations representing simple topologies. Related to this, while DMC is trained to minimize point-to-triangle distances, it does not provide the tessellations to support sharp edges. As a result, the reconstructed geometry near edges is “bulging” in order to minimize distances to the training points.

64^3 resolution	CD($\times10^5$)↓	F1↑	NC↑	ECD($\times10^2$)↓	EF1↑	#V	#T
MC33	25.538	0.069	0.907	7.411	0.017	5,939.62	11,881.67
Lopes2003	25.526	0.068	0.908	11.948	0.015	23,757.44	47,517.48
Trilinear	25.510	0.068	0.909	12.598	0.015	-	-
NMC-lite	**6.055**	**0.495**	**0.923**	0.606	**0.328**	22,540.88	43,272.05
NMC	6.108	0.493	**0.923**	**0.602**	0.314	40,430.06	80,861.75
32^3 resolution	CD($\times10^4$)↓	F1↑	NC↑	ECD($\times10^2$)↓	EF1↑	#V	#T
MC33	9.247	0.028	0.865	8.632	0.017	1,553.93	3,107.50
Lopes2003	**9.246**	0.028	**0.867**	12.344	0.015	6,215.99	12,431.69
Trilinear	9.256	0.028	**0.867**	12.709	0.015	-	-
NMC-lite	9.998	**0.258**	0.852	**0.946**	**0.096**	5,261.82	9,971.62
NMC	10.177	0.256	0.852	0.957	0.093	9,043.78	18,083.90

Table 4.5: Quantitative comparisons on Thingi10K with binary voxel input.

Input and output complexities. When making comparisons, the input resolutions to all methods are identical, but the output complexities do vary, as shown in Tables 4.1-4.5, in terms of the average triangle and vertex counts. With the same tessellation templates, hence comparable output complexities, NMC-lite outperforms Lopes2003 on all fronts, both quantitatively and qualitatively, offering clear evidence for the effectiveness of our data-driven approach for isosurfacing. The new tessellation templates designed for NMC are more refined, resulting in higher triangle counts, but also improved reconstruction quality, as shown in Figures 4.9 and 4.12.

NMC vs NMC-lite. Quantitatively, the performances of NMC and NMC-lite are quite similar, proving the robustness of our representation design. However, the visual quality of NMC results tends to be better than that of NMC-lite, at the expense of almost doubling the triangle counts. Thus, if a lower output complexity is desired, one may choose NMC-lite over NMC. But since NMC-lite employs simpler tessellation templates, it may not be able to recover specific fine shape features, such as the thin structures in the examples from the first and third rows of Figure 4.9, where the cubes with Case 3.2 were not well reconstructed. Also, we have observed that the *triangle quality* resulting from NMC is generally better than that from NMC-lite (e.g., see Figure 4.1), since the NMC tessellation templates were designed to better avoid thin/sliver triangles.

Training and testing times. Network training takes about 3 days on one Nvidia RTX 2080 Ti GPU for SDF processing and 2 days for binary voxel inputs. We tested inference time on the entire ABC test set with 64^3 inputs: the average is 0.006 second per shape for MC33 (implemented in scikit-image [243]), and 0.762 second for NMC (with 0.719s spent on network forwarding in PyTorch [188] and 0.042s for meshing in Cython [12]). Note that currently, our network is still quite crude, as we prioritize accuracy over speed. Possible speed-up could be achieved via neural architecture search and network pruning.

64^3 resolution	CD($\times 10^5$)↓	F1↑	NC↑	ECD($\times 10^2$)↓	EF1↑
MC33	16.611	0.710	0.942	3.360	0.100
Lopes2003	16.545	0.714	0.947	3.692	0.093
NMC (trained on clean data)	**15.340**	0.769	0.941	0.574	0.502
NMC (trained on noisy data)	15.627	**0.802**	**0.951**	**0.359**	**0.640**

Table 4.6: Quantitative comparison on ABC test set with noisy SDF input.

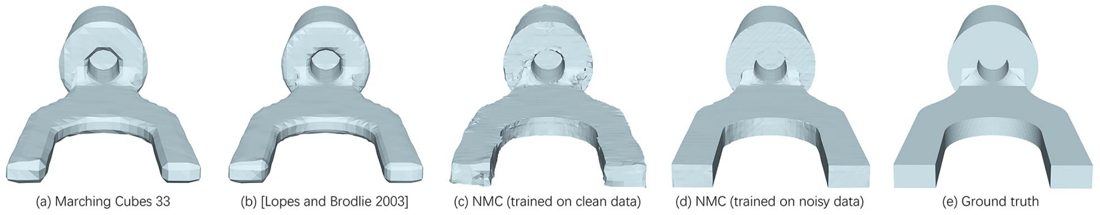

(a) Marching Cubes 33
(b) [Lopes and Brodlie 2003]
(c) NMC (trained on clean data)
(d) NMC (trained on noisy data)
(e) Ground truth

Figure 4.14: Results of reconstructing 3D meshes from a *noisy* SDF grid input at 64^3 resolution.

Noisy inputs. Finally, we show that NMC can also learn to extract *clean* meshes from noisy grid inputs when the network is trained on such data, such as those generated by current neural implicit models [38, 109, 163]. To test this capability, we run a state-of-the-art neural implicit model, SIREN [227], on the ABC dataset to fit each shape, but with only 4,096 training points per shape. The sparsity of the training points makes the output implicit fields necessarily noisy, as can be observed from Figures 4.14(a-b).

In our previous experiments, we trained NMC on clean data from ABC and assumed that the testing inputs were also clean. A model trained this way may fail when the input is noisy, as shown in Figure 4.14(c). To remedy this, we divide the noisy inputs into 80% training and 20% testing as before, and use the noisy inputs to train the NMC model from scratch. The re-trained NMC improves significantly on inputs from the noisy test set, as shown in Figure 4.14, demonstrating that our method can adapt to different inputs (such as voxels and noisy data), if trained on them.

Table 4.6 shows a quantitative comparison involving NMC trained on clean vs. noisy data. We note that Chamfer Distance (CD) is rather sensitive to outliers, e.g., SIREN may generate blobs in the empty region that are far away from the shape, which can impact CD heavily. In comparison, F1 is a more robust quality measure to outliers, as discussed in [237].

Comparison of different smoothness terms Due to considerable ambiguities in possible shapes represented by binary voxels, we need an extra smoothness term to regularize the generated surfaces. We reuse notations from Section 4.3.5 for E_s, F_s^u, and H_s^u, and $\mathbb{1}(\cdot)$. Further, let $F_s^{uv} = F_s^v - F_s^u$, $H_s^{uv} = H_s^v - H_s^u$, $[F_s^u]_x$ be the x coordinate of F_s^u, and $[F_s^{uv}]_{yz} = \sqrt{[F_s^{uv}]_y^2 + [F_s^{uv}]_z^2}$. We have experimented with the following settings:

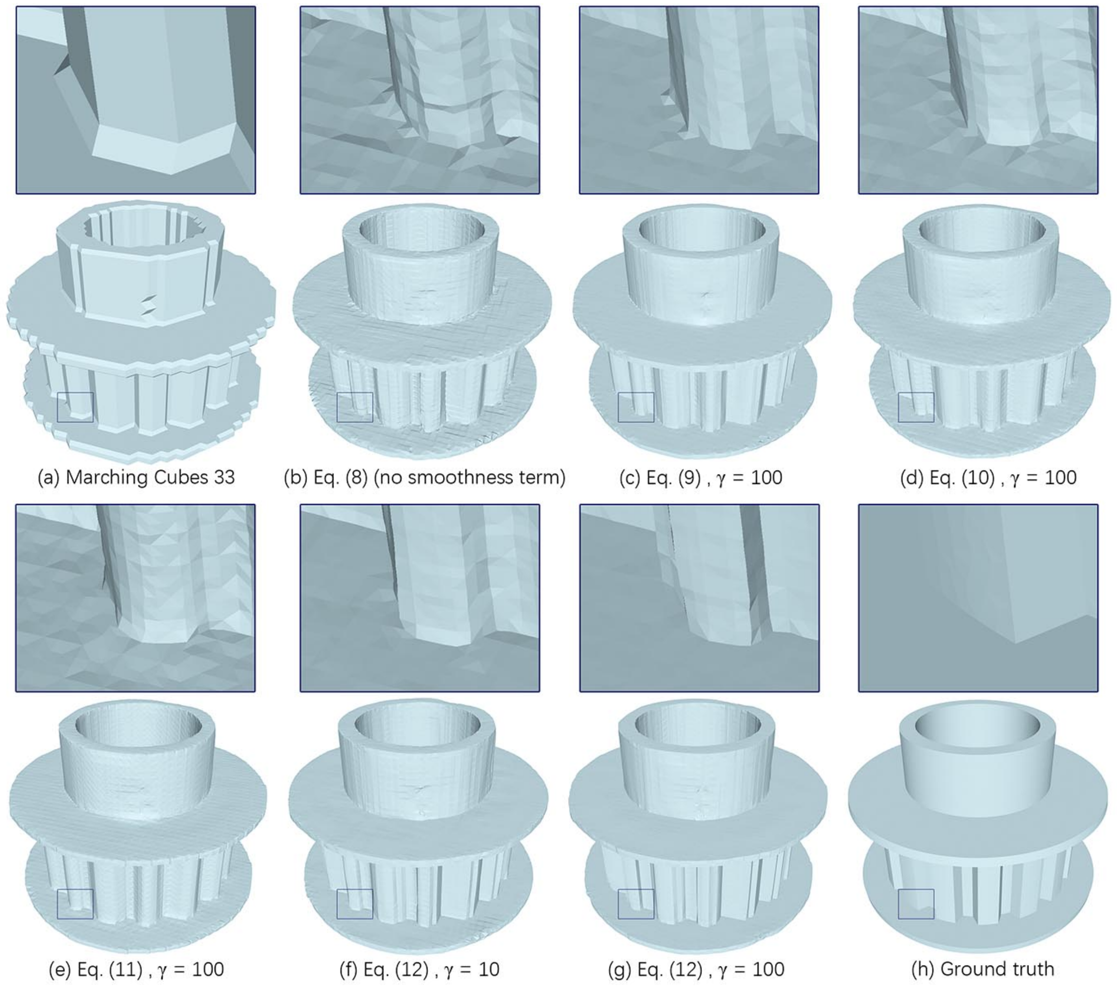


Figure 4.15: Visual comparisons of different smoothness terms on ABC test set with binary voxel input at 64^3 resolution.

64^3 resolution	CD($\times 10^5$)$\downarrow$	F1$\uparrow$	NC$\uparrow$	ECD($\times 10^2$)$\downarrow$	EF1$\uparrow$
Eq. (8)	$^{\underline{4}}$9.355	$^{\underline{1}}$0.445	$^{\underline{3}}$0.932	$^{\underline{5}}$0.633	$^{\underline{4}}$0.328
Eq. (9), $\gamma = 100$	$^{\underline{2}}$9.329	$^{\underline{3}}$0.435	$^{\underline{3}}$0.932	$^{\underline{4}}$0.615	$^{\underline{3}}$0.353
Eq. (10), $\gamma = 100$	$^{\underline{6}}$9.539	$^{\underline{3}}$0.435	$^{\underline{2}}$0.933	$^{\underline{3}}$0.612	$^{\underline{5}}$0.320
Eq. (11), $\gamma = 100$	$^{\underline{5}}$9.518	$^{\underline{5}}$0.434	$^{\underline{1}}$0.934	$^{\underline{2}}$0.562	$^{\underline{6}}$0.313
Eq. (12), γ=10	$^{\underline{3}}$9.341	$^{\underline{2}}$0.438	$^{\underline{5}}$0.931	$^{\underline{1}}$0.528	$^{\underline{2}}$0.356
Eq. (12), $\gamma = 100$	$^{\underline{1}}$9.327	$^{\underline{6}}$0.427	$^{\underline{6}}$0.923	$^{\underline{6}}$0.669	$^{\underline{1}}$0.359

Table 4.7: Comparison of different smoothness terms on the ABC test set. The underlined superscripts show rankings of the quantitative performances, where the overall best performing row is highlighted in bold.

$$L^{(1)}_{smooth} = 0 \quad \text{(No smoothness term).} \tag{4.8}$$

$$L^{(2)}_{smooth} = \mathbb{E}_{s\sim\mathcal{S}} \sum_{(u,v)\in E_s} \| \ F^{uv}_s - H^{uv}_s \ \|^2_2. \tag{4.9}$$

$$\begin{aligned} L^{(3)}_{smooth} &= \mathbb{E}_{s\sim\mathcal{S}} \sum_{(u,v)\in E_s} \left(L^x_y + L^x_z + L^y_x + L^y_z + L^z_x + L^z_y\right), \\ \text{where } L^x_y &= \left(\ [H^{uv}_s]_x \cdot |[F^{uv}_s]_y| - [F^{uv}_s]_x \cdot |[H^{uv}_s]_y| \ \right)^2. \end{aligned} \tag{4.10}$$

$$\begin{aligned} L^{(4)}_{smooth} &= \mathbb{E}_{s\sim\mathcal{S}} \sum_{(u,v)\in E_s} \left(L^x_{yz} + L^y_{xz} + L^z_{xy}\right), \\ \text{where } L^x_{yz} &= \left(\ [H^{uv}_s]_x \cdot [F^{uv}_s]_{yz} - [F^{uv}_s]_x \cdot [H^{uv}_s]_{yz} \ \right)^2. \end{aligned} \tag{4.11}$$

$$L^{(5)}_{smooth} = \mathbb{E}_{s\sim\mathcal{S}} \sum_{(u,v)\in E_s} \| \ \mathbb{1}(|F^{uv}_s| < \sigma) \cdot H^{uv}_s \ \|^2_2. \tag{4.12}$$

The smoothness term $L^{(2)}_{smooth}$ in Eq. (4.9) minimizes the differences between the edge gradients on the predicted mesh and those on the GT. $L^{(3)}_{smooth}$ and $L^{(4)}_{smooth}$ try to preserve the absolute angles of the edges. In an ideal situation, $[H^{uv}]_x/|[H^{uv}]_y| = [F^{uv}]_x/|[F^{uv}]_y|$, therefore $[H^{uv}]_x \cdot |[F^{uv}]_y| - [F^{uv}]_x \cdot |[H^{uv}]_y| = 0$. Eq. (4.10) minimizes the squared error of such terms. Eq. (4.11) is a variant of Eq. (4.10), while Eq. (4.12) is equivalent to Eq. (4.7) in Section 4.3.5.

The performances of the different smoothness terms are shown in Table 4.7 and exhibited visually in Figure 4.15. Based on the visual results and overall quantitative performances, we have adopted $L^{(5)}_{smooth}$ in our work, with $\gamma = 10$ and $\sigma = 0.0002$.

4.5 Conclusions

In this paper, we show for the first time that the mesh reconstruction quality by Marching Cubes (MC), one of the most classical algorithms in computer graphics, can be significantly boosted by machine learning. In Neural Marching Cubes (NMC), we introduce the first MC-based approach capable of recovering sharp geometric features without requiring additional inputs, such as normal information. Trained on automatically generated "ground-truth" meshes, our method shows superior performance in preserving various geometric features such as sharp/soft edges, corners, thin structures, etc., compared to other isosurfacing algorithms that take uniform grids of signed distances or binary occupancies as inputs. We also designed an efficient parameterization to represent a triangle mesh, compatible with neural

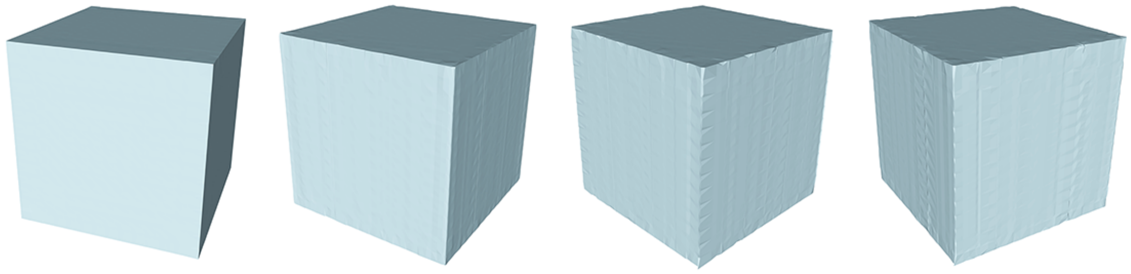


Figure 4.16: NMC may produce artifacts when the input is oriented at an “unusual” angle relative to the training shapes. From left to right: reconstructions of a cube that is axis-aligned, then rotated by $\frac{1}{14}\pi$, $\frac{2}{14}\pi$, and $\frac{3}{14}\pi$.

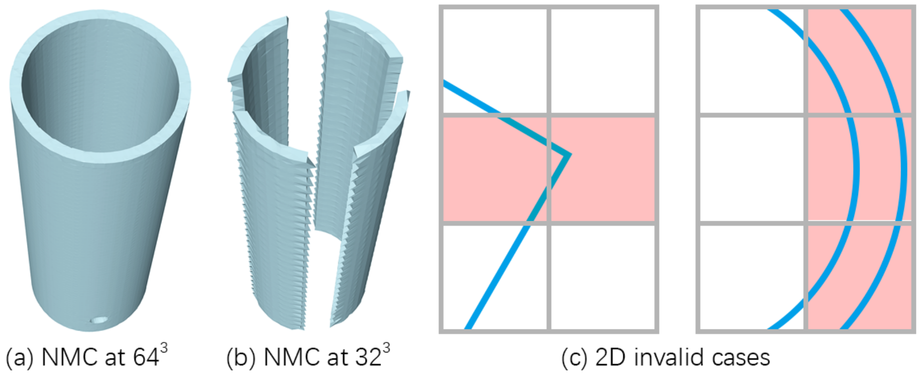


Figure 4.17: NMC cannot account for certain topological cases (deemed “invalid”), e.g., multiple intersections along an edge as highlighted in red (c). The reconstruction failure in (b) is due to similar invalid cases in 3D.

processing, so that our NMC network can directly output the meshes without postprocessing.

The main limitation of our method in terms of isosurfacing is its sensitivity to rotation, as shown in Figure 4.16. This is mainly due to the dataset we train the network on, as the shapes in the ABC dataset are mostly aligned with the coordinate axes. This data characteristic also motivated our definition of the smoothness term in Eq. (4.7). Performing random rotation augmentation on the training data is a viable solution, but would require longer training time and larger networks to fit. Second, as we follow the common assumption in MC that if the two vertices of a cube edge have different signs, then there is one and only one intersection point, several topological cases (as shown in Figure 4.17) cannot be represented. Adding more intersection points should cover most of such cases, and the numbers and the positions of the edge vertices can be learned from data.

Another limitation is that we do not have a built-in mechanism to avoid self-intersections in the output meshes. When testing on 64^3 SDF inputs from ABC, 32.7% of the meshes produced by NMC contain self-intersections, but they involve only 0.0086% of the triangles, which translate to about 7.39 triangles or around two separate intersections, per shape. For NMC-lite, the corresponding numbers are 29.6%, 0.0078%, and 3.40, respectively. Most of the intersections happen in cases where the structure in a cube cannot be represented by our tessellation templates, such as those in Figure 4.17(b). Last but not the least, our current model does not allow the learning of an arbitrary *tessellation style*, e.g., meshes whose triangles are all close to being equilaterals. The challenge is on how to prepare the proper training meshes to work under our designed templates.

Our proposed representation is not constrained to isosurfacing. It is a general shape representation that can be adopted to other tasks such as shape upsampling and generation, pointing to a worthy direction for future work. On the other hand, even when the input is a uniform grid, the output mesh does not have to be uniform. A simple plane only requires a few triangles to model, but a curved surface needs more. Therefore, learning to adaptively allocate vertices and triangles according to feature complexity could yield more efficient algorithms and control the explosion of triangle counts in MC, as reported in BSP-NET [36].

Chapter 5

Neural Dual Contouring

5.1 Introduction

Polygonal mesh reconstruction from discrete inputs such as point clouds and voxel grids has been one of the most classical and well-studied problems in computer graphics [54, 14]. Current solutions to the problem are predominantly model-driven, often relying on assumptions such as those related to shape characteristics (e.g., watertightness, zero genus, etc.), surface interpolants (e.g., trilinearity), sampling conditions, surface normals, and other reconstruction priors. It is only recently that a few *data-driven* meshing methods have emerged. However, they have mostly focused on learning point set triangulations [197, 220, 149]. One exception is Neural Marching Cubes (NMC) [39], a learning-based Marching Cubes (MC) approach for mesh reconstruction from a voxel grid of signed distances or binary occupancies. In comparison to the original MC algorithm [157] and its best-known variant, MC33 [40], NMC uses tessellation templates with more adaptive mesh topologies and learns local shape priors from training meshes. As a result, NMC generalizes well to a broader range of shape types and excels at preserving sharp features, two long-standing issues in existing MC work. On the other hand, the NMC tessellation templates are necessarily more complex than those of MC and MC33. As a result, NMC typically outputs 4-8 times the number of triangles and incurs 100× or more compute time to reconstruct a mesh.

In this paper, we introduce *Neural Dual Contouring* (NDC), a new data-driven approach to mesh reconstruction based on dual contouring (DC) [110]. The key motivation for building our learning framework upon DC rather than MC is that it provides a *more natural* and *more efficient* means of reproducing sharp features. As shown in Figure 5.2, NDC only needs to predict *one* mesh vertex per grid cell (i.e., a cube) and one quad for each cell edge intersected by the underlying surface. In contrast, NMC requires 23 edge, face, and interior vertices per grid cell [39, Fig.5].

A traditional drawback of the classical DC, as compared to MC, is that it requires gradients (i.e., surface normals) as input to compute a suitable vertex location within each cell. Our data-driven approach does not have this drawback. NDC employs a neural network

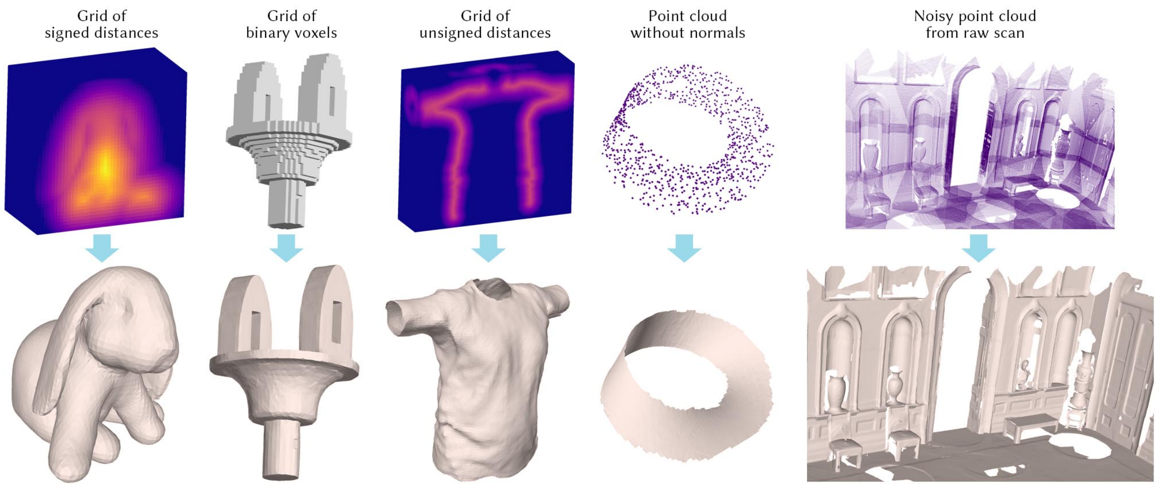

Grid of signed distances
Grid of binary voxels
Grid of unsigned distances
Point cloud without normals
Noisy point cloud from raw scan

Figure 5.1: Neural dual contouring (NDC) is a unified data-driven approach that learns to reconstruct meshes (bottom) from a variety of inputs (top): signed or unsigned distance fields, binary voxels, non-oriented point clouds, and noisy raw scans. Trained on CAD models, NDC generalizes to a broad range of shape types: CAD models with sharp edges, organic shapes, open surfaces for cloths, scans of indoor scenes, and even the non-orientable Mobiüs strip.

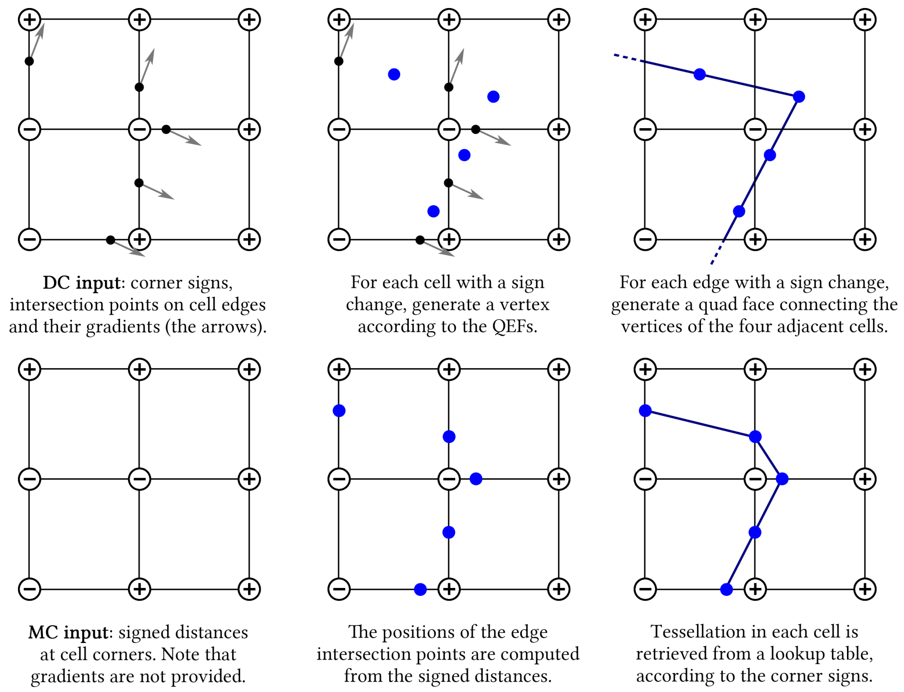

DC input: corner signs, intersection points on cell edges and their gradients (the arrows).
For each cell with a sign change, generate a vertex according to the QEFs.
For each edge with a sign change, generate a quad face connecting the vertices of the four adjacent cells.
MC input: signed distances at cell corners. Note that gradients are not provided.
The positions of the edge intersection points are computed from the signed distances.
Tessellation in each cell is retrieved from a lookup table, according to the corner signs.

Figure 5.2: **Dual Contouring (DC)** vs. **Marching Cubes (MC)** – visualized in 2D on different inputs that were sampled from the same underlying shape, DC (top) reconstructs a sharp feature (as an intersection between faces, in the top-right cell), while MC (bottom) does not.

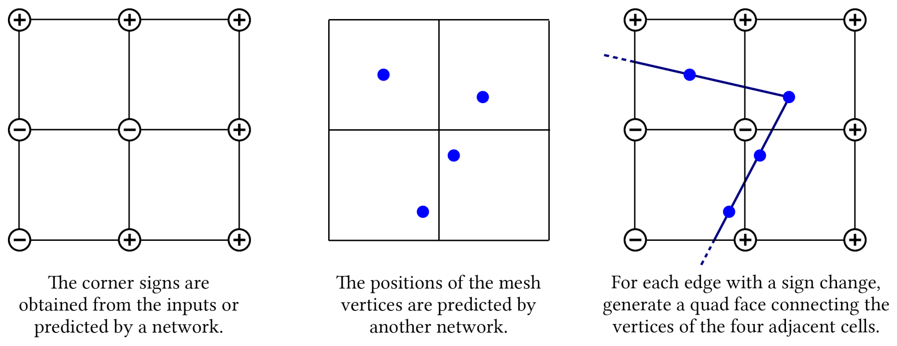

The corner signs are obtained from the inputs or predicted by a network.
The positions of the mesh vertices are predicted by another network.
For each edge with a sign change, generate a quad face connecting the vertices of the four adjacent cells.

Figure 5.3: **Neural dual contouring (NDC)**

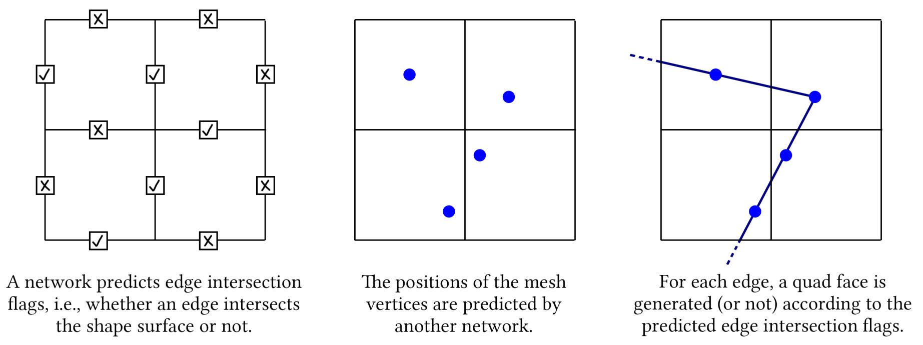

A network predicts edge intersection flags, i.e., whether an edge intersects the shape surface or not.
The positions of the mesh vertices are predicted by another network.
For each edge, a quad face is generated (or not) according to the predicted edge intersection flags.

Figure 5.4: **Unsigned neural dual contouring (UNDC)**

trained on example 3D surface data to predict the vertex locations (Figure 5.3). Our neural network learns to compute whatever gradients and/or contexts that are useful to reproduce the training surfaces, and thus can operate on a voxel grid *without gradients as input*.

Another key feature of DC is that its meshing only requires knowing whether a cell edge is intersected by the output surface or not [137]. We can thus train our network to predict an intersection or crossing *flag* per edge, in addition to vertex locations, without accounting for signs at cell corners (Figure 5.4). We refer to this version of our network as *unsigned* NDC, or UNDC for short. With the *sign-agnostic* UNDC, we can forgo both the input requirement on signed distances and the output requirement that the resulting mesh is closed and watertight, as for MC and its variants.

Our learning model is built with 3D convolution neural networks (CNNs) separately trained for vertex prediction and the prediction of cell corner signs (NDC) or edge crossings (UNDC)[1]. Our network training is supervised with an L2 reconstruction loss against pseudo ground-truth vertices computed by DC and binary cross entropy loss for sign/crossing pre-

[1]Note that in the rest of the paper, we use the term NDC to refer to both our overall dual contouring based learning framework *and* the specific network that reconstructs meshes based on sign prediction (Figure 5.3). On the other hand, the term UNDC is used exclusively to denote the sign-agnostic version of our method (Figure 5.4).

dictions. As in NMC, our CNNs are designed with limited receptive fields to ensure generalizability.

We train our NDC networks on a CAD dataset, ABC [122], and we test them on ABC and four other datasets to assess generalizability: 1) Thingi10K [302], a dataset of 3D-printing models, 2) FAUST [19], a dataset of human body shapes, 3) MGN [17], a dataset of clothes as open surfaces, and 4) Matterport3D [27], a collection of scenes with noisy RGB-D depth images. Quantitative and qualitative evaluations on isosurfacing using voxel data as input suggest that NDC clearly outperforms MC33 and several variants of NMC in terms of mesh reconstruction quality, feature preservation, triangle quality, and inference time, when using signed (distances or binary voxels) grids as inputs. At the same time, NDC produces 4-8 times fewer mesh elements using 3-20 times less inference time, compared to NMC. Further experiments with point cloud inputs suggest that UNDC outperforms both classical non-learning based methods, such as Ball Pivoting [15], Screened Poisson reconstruction [118], and recent reconstructive neural networks such as SIREN [227], Local Implicit Grids [107], and Convolutional Occupancy Networks [193]. Qualitative and quantitative results show significant improvements for NDC in terms of reconstruction quality, feature preservation, and inference time. Our main contributions can be summarized as follows:

- We propose the first data-driven approach to mesh reconstruction based on dual contouring. Unlike classical DC, which optimizes vertex locations within the confines of individual cells using a handcrafted Quadratic Error Function (QEF) [76], NDC predicts vertex locations using a learned function, which eliminates the need for gradients in the input and accounts for local *contextual* information inherent in the training data.
- A unified learning model that is applicable to a larger variety of inputs than previous meshing methods. As shown in Figure 5.1, the allowed inputs include signed/unsigned distance fields, binary voxels, and un-oriented point clouds.
- A significant, 23:1, reduction in representational complexity by NDC over NMC translates to across-the-board gains, in terms of simplicity of the network architecture, as well as reduction in network capacity, training and inference times, and more; see Table 5.1 for a summary.
- A sign-agnostic network, UNDC, that can produce *open*, even *non-orientable*, output surfaces; see Figure 5.1.

5.2 Related work

The literature on mesh reconstruction is extensive and so we refer to several surveys for full coverage [54, 14]. In this section, we focus on techniques for isosurfacing (i.e., mesh extraction from discrete volume data) and surface reconstruction from point cloud data, with a focus on the recent data-driven approaches most closely related to our work. Then in

	NMC	NDC
Output	5 (bool) + 51 (float) per cube	1 (bool) + 3 (float) per cube
Network	3D ResNet	6-layer 3D CNN
Tessellation	Manually designed, 37 unique cases per cube	≤ 1 vertex per cell; ≤ 1 quad per edge; see Figure 5.3
Output vertex count	$\approx 8\times$ MC	$\approx$ MC
Output triangle count	$\approx 8\times$ MC	$\approx$ MC
Data preparation	Sample dense point cloud in each cube; minimize chamfer distance via back propagation; complex and time-consuming	Sample only vertex signs, intersection points and normals; then apply Dual Contouring; Fast and easy to compute.
Implementation	Need to consider all cube tessellation cases; difficult to implement	Could be a nice undergraduate assignment
Regularization	Need a complex regularization term for voxel input	No regularization term needed
Trainging time	(On ABC training set) 4 days per network	(Same setting) $<$ 12 hours per network
Inference speed	(64^3 SDF input) $>$ 1 second per shape	(Same setting) 30+ shapes per second
Inherent issues	Self-intersections, thin triangles with small angles	Non-manifold edges and vertices

Table 5.1: Comparing various aspects of NMC vs. NDC.

Section 5.2.3, we formally define dual contouring (DC), establish notations used throughout the paper, and compare DC to marching cubes.

5.2.1 Isosurfacing and differentiable reconstruction

The marching cubes (MC) approach for isosurfacing from discrete signed distances was first proposed concurrently by [157] and [273]. Since then, many variants have followed, including the best-known MC33 [40], which correctly enumerated all possible topological cases for mesh tessellations, based on the trilinear interpolation assumption. Indeed, most of the MC follow-ups made the same assumption and are unable to recover sharp features. This issue was resolved by neural marching cubes (NMC) [39], which combines deep learning with MC for the first time, building on the premise that feature recovery can be learned from training meshes.

Our work is inspired by NMC. In NDC, we combine deep learning with dual contouring (DC) [110] to bring key advantages of classical DC over MC into a learned mesh reconstruction model, without requiring any additional inputs (e.g. gradients). In addition to improved efficiency and reconstruction quality (see Section 5.4), our method also represents the first unified mesh reconstruction framework that can take on all the input types shown in Figure 5.1. To the best of our knowledge, no previous methods were designed to reconstruct meshes from *unsigned* distance fields.

Several recent works, including deep marching cubes (DMC) [139], MeshSDF [201], and Deformable Tetrahedral Meshes (DefTet) [71], propose *differentiable* mesh reconstruction schemes. While both these methods and NDC bring deep learning to mesh reconstruction, their focuses and strengths are quite different. DMC, MeshSDF, and DefTet all target end-to-end differentiability, while offering limited capabilities to reconstruct geometric and topological details. They also encode *global* features for their predictions, which can hinder both scalability, reconstruction quality (as downsampling is necessary during training), and generalizability. In contrast, our work focuses on learning a refined meshing model applicable to a variety of inputs. We target fine-grained quality criteria related to feature preservation and surface quality. Our learning model is also local, hence highly scalable and generalizable to diverse shape types and classes.

5.2.2 Mesh reconstruction from point clouds

Many methods have been proposed for surface mesh estimation from unorganized points. Following the taxonomy in [14], previous works can be characterized based on the underlying priors, e.g., smoothness [118], visibility [48], dense sampling [2], primitives [215], and learning from data [260]. Among the methods based on data priors, some compose surfaces explicitly from patches extracted from examples [69, 189, 221]. Others learn implicit priors, either for entire objects [185, 163, 38, 192, 42] or for patches [8, 86, 107, 164, 193, 227, 260, 94]. Both NMC [39] and NDC are in the latter category: they learn implicit priors for local regions.

Surface reconstruction methods also differ in whether they can work for input point clouds without normals [5, 235], whether the output mesh interpolates the input points via triangulation [197, 220, 149], and whether they can produce open surfaces from partial scans, e.g., via an advancing front scheme [45, 15]. Of course, normals can be estimated in a preprocessing step (e.g., using [20]), and open surfaces can be created from watertight reconstructions in a postprocessing step (e.g., using SurfaceTrimmer in [118]). However, these separate steps rely on heuristic algorithms with parameters that are difficult to tune (e.g., size of neighborhood for normal estimation, density of points for surface trimming, etc.). By comparison, our UNDC includes all these steps in a single learned process that can produce open, even non-orientable, meshes directly from unoriented point cloud inputs, with fast inference. Also, our method is non-interpolatory, hence insensitive to sampling non-uniformity and noise (with noise augmentation in training). In Section 5.4.6, we compare UNDC with several representative learning-based reconstruction networks [227, 193, 107] whose results are most competitive to ours. Technical details about these works are described in Section 5.4.6.

5.2.3 Dual Contouring (DC)

[110] introduced *Dual Contouring* to convert a Signed Distance Field $\Phi : \mathbb{R}^3 \to \mathbb{R}$ into a polygonal mesh $\mathcal{M} = (\mathcal{V}, \mathcal{F})$; see Figure 5.2 (top). This is achieved by discretizing the function on a lattice $\mathcal{G} = (\mathcal{X}, \mathcal{E})$. It first samples the Φ at the grid vertices $\mathcal{X}$ and determines their signs $\mathcal{S}$. Then, it finds the zero crossings $\mathcal{V}^{\mathcal{E}}$ of the Φ on the lattice edges spanning vertices with opposite signs. Next, it computes the gradients of the Φ at those crossings, which provide surface normals $\mathcal{N}^{\mathcal{E}}$. Finally, it creates quadrilateral polygonal faces $\mathcal{F}$ that are *dual* to the lattice edge crossings $\mathcal{E}$. In what follows we have $|\mathcal{X}| = M \times N \times K$ lattice vertices, $|\mathcal{E}| = (M-1) \times (N-1) \times (K-1) \times 3$ lattice edges, and we index $\mathcal{X}$ by (m, n, k), while we refer to edges as $(i, j) \in \mathcal{E}$. Dual contouring assumes as input:

$$\mathcal{S} \in \mathbb{B}^{|\mathcal{X}|}, \qquad \mathcal{S} = f_{\mathcal{S}}(\Phi, \mathcal{G}), \qquad \text{(grid signs)} \tag{5.1}$$

$$\mathcal{V}^{\mathcal{E}} \in \mathbb{R}^{|\mathcal{E}|\times 3}, \qquad \mathcal{V}^{\mathcal{E}} = f_{\mathcal{V}^{\mathcal{E}}}(\Phi, \mathcal{G}), \qquad \text{(edge vertices)} \tag{5.2}$$

$$\mathcal{N}^{\mathcal{E}} \in \mathbb{R}^{|\mathcal{E}|\times 3}, \qquad \mathcal{N}^{\mathcal{E}} = f_{\mathcal{N}^{\mathcal{E}}}(\Phi, \mathcal{G}), \qquad \text{(edge normals)} \tag{5.3}$$

where, analogously to marching cubes [157], $\mathcal{S}$ are the signs of Φ on the lattice vertices, that is $f_{\mathcal{S}} : \text{sign}(\Phi(\mathcal{X}))$, $f_{\mathcal{V}^{\mathcal{E}}}$ computes the zero-crossings of Φ along the lattice edges, and $f_{\mathcal{N}^{\mathcal{E}}} : \nabla\Phi(\mathcal{V}^{\mathcal{E}})$ are the gradients of Φ measured at $\mathcal{V}^{\mathcal{E}}$. Given these quantities, dual contouring generates a polygonal mesh, consisting of *quad* faces and corresponding vertices:

$$\mathcal{F} \in \mathbb{B}^{|\mathcal{E}|}, \qquad \mathcal{F} = f_{\mathcal{F}}(\mathcal{S}), \tag{5.4}$$

$$\mathcal{V} \in \mathbb{R}^{|\mathcal{X}|\times 3}, \qquad \mathcal{V} = f_{\mathcal{V}}(\mathcal{V}^{\mathcal{E}}, \mathcal{N}^{\mathcal{E}}), \tag{5.5}$$

where, with a slight abuse of notations, we use the same nomenclature $f_{\mathcal{F}}$ for a polygonal face (i.e. tuple of vertex indices) and the Boolean value that determines whether the face should be created.

Dual faces $\mathcal{F}$ are created *only* whenever lattice edges connect lattice vertices of *opposite* signs $\mathcal{S}$:

$$f_{\mathcal{F}} : \text{xor}(\mathcal{S}_i, \mathcal{S}_j), \quad (i, j) \in \mathcal{E}. \tag{5.6}$$

Vertices are created by triangulating, a-la [76], the planar constraints defined on the edges of each voxel in the lattice:

$$f_{\mathcal{V}} : \arg\min_{\mathbf{x}} \sum_{\mathbf{e} \in \mathcal{G}_{mnk}} ({\mathcal{N}^{\mathcal{E}}}_{\mathbf{e}} \cdot (\mathbf{x} - {\mathcal{V}^{\mathcal{E}}}_{\mathbf{e}}))^2, \tag{5.7}$$

where $\mathcal{G}_{mnk}$ refers to the voxel rooted at $\mathcal{X}_{mnk}$, and $\mathbf{e}$ iterates the 12 edges of the voxel.

Comparison to MC DC has the drawback that it assumes the availability of the function's gradients $\mathcal{N}^{\mathcal{E}}$. This perhaps justifies why it has not been as popular as MC, which *only* requires signs (5.1) and zero-crossings (5.2). Nonetheless, the mesh creation mechanism of dual contouring is *significantly* simpler than the one in MC, where the former involves simple Boolean operations, while the latter involves enumerating all possible combinations and employing look-up tables that define the corresponding topology. Further, note that MC tends to discard high frequency information (i.e. sharp corners), DC is capable of preserving such details to a much better extent.

5.3 Method

In this paper, we introduce a learning framework, neural dual contouring (NDC), that achieves the simplicity and sharp features of DC without requiring function gradients in the input. Given any common input representation $\mathcal{I}$ (e.g. point cloud, signed or unsigned distance functions, or voxelized grids), NDC can be formalized by a simple generalization of Equations (5.1, 5.4, 5.5). In particular, we introduce *two* different techniques, illustrated with a 2D example in Figure 5.3 and Figure 5.4, and detailed in what follows.

The first, and default, variant of our method, which reconstructs meshes based on sign prediction, is simply referred to as NDC. It can be algebraically formalized as:

$$\mathrm{NDC}(\mathcal{I}) = \begin{cases} \mathcal{S} &= f_{\mathcal{S}}(\mathcal{I}, \mathcal{G};\ \theta), \\ \mathcal{V} &= f_{\mathcal{V}}(\mathcal{I}, \mathcal{G};\ \theta), \\ \mathcal{F} &= \mathrm{xor}(\mathcal{S}_i, \mathcal{S}_j). \end{cases} \tag{5.8}$$

The logic controlling whether a face should be generated is identical to classical DC, while vertices and signs are predicted by neural networks (with trainable parameters θ) that receive as input $\mathcal{I}$. At the same time, the input requirements of NDC are closer to the ones of MC and NMC: we do not require the availability of normals as in classical DC since we do not perform explicit optimizations for vertex locations using (5.7). Instead, vertex positions are predicted with a network trained from examples.

The second variant, named UNDC , with U denoting "unsigned", is similar to NDC, but it *directly* predicts the existence of dual faces $\mathcal{F}$ rather than resorting to sign prediction:

$$\mathrm{UNDC}(\mathcal{I}) = \begin{cases} \mathcal{V} &= f_{\mathcal{V}}(\mathcal{I}, \mathcal{G};\ \theta), \\ \mathcal{F} &= f_{\mathcal{F}}(\mathcal{I}, \mathcal{G};\ \theta). \end{cases} \tag{5.9}$$

The key advantage of this variant is that it can produce surface crossings without having to rely upon differences of inside/outside signs at grid cell vertices. This feature allows UNDC

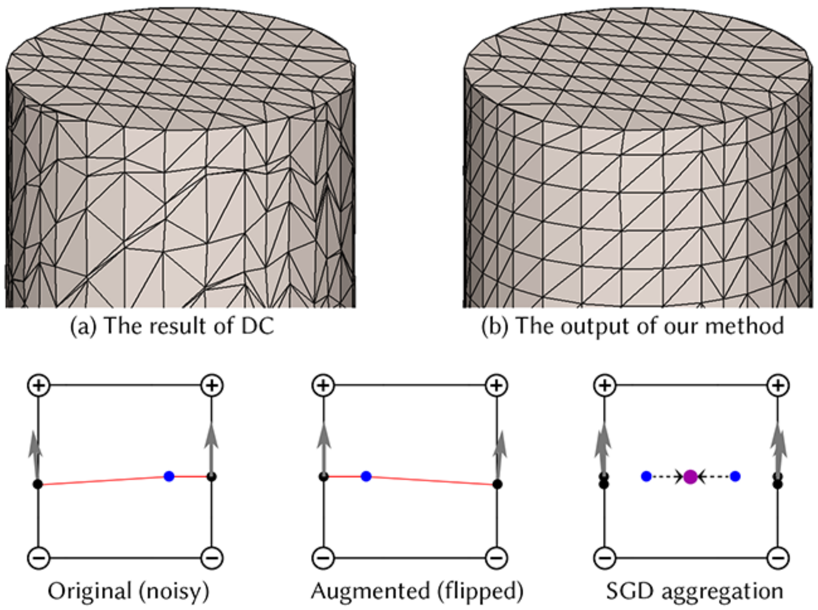


Figure 5.5: **Training data preparation with data augmentation** – The ground truth meshes computed using classical DC (a) can be noisy. With proper augmentation for the training data (see bottom), our NDC network can be trained to output meshes with better tessellation quality (b).

to operate on *unsigned* distance fields or *non-oriented* point clouds (we employ the prefix *U* to indicate this variant's ability to operate on *unsigned* inputs). It also allows UNDC to produce mesh faces, likely in the form of *thin sheets*, in regions where the underlying object parts are *thinner* than one voxel. Clearly, such thin parts are not representable by differences of grid vertex signs, and as a result, methods including MC, NMC, as well as NDC, would not be able to reconstruct them at all; see Figure 5.8 in Section 5.4. Additionally, UNDC can produce open surfaces with boundaries directly for input data representing partial surfaces. These advantages are in contrast to other methods like MC and variants that guarantee their outputs to be watertight and can represent only solid objects without thin features.

5.3.1 Encoders

Let us now consider the design of $f_{\mathcal{V}}$, $f_{\mathcal{S}}$, and $f_{\mathcal{F}}$ for different types of input $\mathcal{I}$: ① signed/unsigned distance functions, ② voxelized occupancy, and ③ point clouds.

Distance Function Inputs When a Signed Distance Function (SDF) Φ is provided as input, our model $f_{\mathcal{V}}$ first samples the function Φ at the grid vertices $\mathcal{X}$ into a floating point tensor of shape $|\mathcal{X}|$. We then use a 3D CNN to process this tensor; the 3D CNN has 6 layers, with the first 3 layers having kernel size 3^3 and the last 3 layers having kernel size 1^3, an overall receptive field of 7^3. We employ hidden layers with 64 channels to make the network computationally efficient (i.e. 37 fps) as it has few network weights (i.e. 1MB). Leaky ReLU

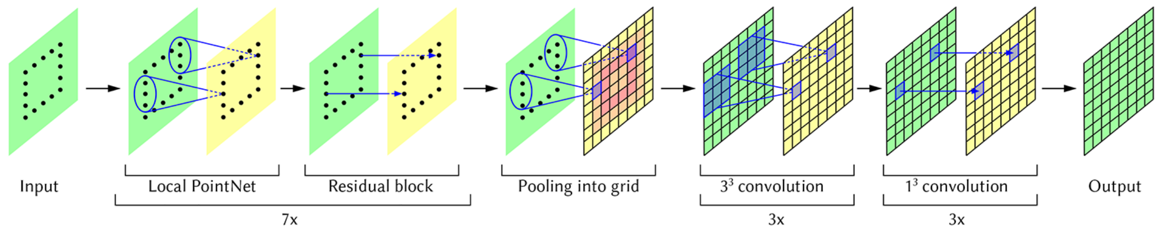


Figure 5.6: The architecture of our point cloud processing network for UNDC.

activation functions are employed everywhere except at the output layer where sigmoids are used. Note that when NDC operates on SDFs, $f_{\mathcal{S}}$ is extremely efficient as it just requires the computation of a sign at lattice locations similarly to classical dual contouring. Finally, the architecture of $f_{\mathcal{F}}$ for the UNDC model is the same as $f_{\mathcal{V}}$ in the NDC model.

Voxelized Occupancy Inputs For this class of inputs, we use a network with almost the same architecture as for SDF input, but with a small modification to enlarge the receptive field to 15^3 (i.e. employ 7 rather than three 3^3 convolutional layers). Our rationale is that voxelized occupancies are heavily quantized, and a larger receptive field would allow the network to develop stronger *priors* to cope with the larger degree of ambiguity in the data.

Point Cloud Inputs For point cloud inputs, we devise a local point cloud encoder network divided into two parts: ① point cloud processing and ② regular grid processing. The former is implemented as a dense PointNet++ [196], while grid processing has three 3^3 convolution layers and three 1^3 convolution layers, hence of a similar architecture to the one used for inputs represented as grids. The network architecture is shown in Figure 5.6. The local PointNet in Figure 5.6 is similar to the set abstraction layer in PointNet++, with the number of local clusters being the same to the number of input points. For each point p_i in the input point cloud, we find a local cluster with K points, and then apply PointNet [195] using relative coordinates of those K points with respect to p_i. In PointNet++, the local cluster is found by setting a radius r, so that any points whose distances to p_i are smaller than r will be selected into the cluster. This may bring issues such as setting appropriate r values and handling situations when a cluster has too many or too few points. Therefore, we use a simpler approach to avoid the issues. We find the K nearest neighbors ($K = 8$ in our experiments) of p_i to form the cluster, by using a KD-tree for efficient computation. Afterwards, we concatenate the relative coordinates of each point with its features, apply two fully-connected layers with leaky ReLU activation, and use max-pooling to aggregate the features of all K points into the feature of p_i. The residual block in Figure 5.6 is a standard residual block [95] for fully connected layers. The "Pooling into grid" module in Figure 5.6 is essentially a local PointNet as described above. The difference is that it uses the centers of the cells in the grid as query points to find the local clusters in the input point cloud via KNN. Since obtaining the features for all cell centers in a 3D gird is very expensive ($O(N^3)$), we only compute features for the cells that are close to the input point cloud, i.e., the cells that are within 3 units (manhattan distance) to the closest point in the point cloud, assuming the size of each cell is 1 unit. All hidden layers in our network has 128 channels. We use the same loss functions as UNDC for training the networks.

5.3.2 Training data preparation

To obtain the training data, we place random 3D mesh objects in the grid $\mathcal{G}$ to compute signs, intersection points, and corresponding ground truth normals[2], and then apply classical DC to obtain the ground truth vertex predictions. However, this process can result in aliased normals, which can lead to poorly positioned vertices after the optimization in (5.7); see Figure 5.5-(left).

While this situation might seem problematic at first, we make the same observation made by [130] in this setting. In particular, the use of an L2 reconstruction loss, coupled with data augmentation, leads to a zero-mean distribution in the vertex positions predicted by our model; see Figure 5.5-(b). To achieve this zero-mean distribution of optimization residuals, we augment the training data by rotation (by $\pi/2$ around the Euclidean axes), mirroring, and (global) sign inversion. Note this augmentation is *not* done within a mini-batch, but rather, we rely on stochastic gradient descent for aggregating data towards a zero-mean residual configuration progressively over the course of training.

5.3.3 Training losses

Given ground truth data, note that all the sub-networks within the NDC and UNDC models can be trained *separately*, leading to a simpler training setup where no hyper-parameter tuning between losses becomes necessary. Note that we leverage our input data to only supervise the prediction made by the networks in a narrow-band around the input surface, with binary masks $\mathcal{M}_\mathcal{S}$, $\mathcal{M}_\mathcal{V}$ that evaluate to one if we are within the narrow-band, and zero otherwise. This is because surfaces should only be created in the proximity of either changes in the sign of Φ, in occupancy for voxelized inputs, or proximity of the input points for point cloud inputs. We start with a simple L2 reconstruction loss of pseudo ground-truth vertices (i.e. as computed by dual contouring):

$$\mathcal{L}_\mathcal{V}(\theta) = \mathbb{E}_{(\mathcal{I},\mathcal{M}_\mathcal{V},\mathcal{V}_{\text{gt}})\sim\mathcal{D}} \sum_{m,n,k} \left[\|\mathcal{M}_\mathcal{V} \odot (f_\mathcal{V}(\mathcal{I},\mathcal{G};\ \theta) - \mathcal{V}_{\text{gt}})\|_2^2 \right],$$

[2]Note that these intersection points and normals were utilized to create the pseudo-ground truth at training time; they are not used at test time.

where $\odot$ is the Hadamard product on $\mathcal{G}$. For NDC , we supervise the prediction of signs via Binary Cross Entropy (BCE):

$$\mathcal{L}_{\mathcal{S}}(\theta) = \mathbb{E}_{(\mathcal{I},\mathcal{M}_{\mathcal{S}},\mathcal{S}_{\text{gt}})\sim\mathcal{D}} \sum_{m,n,k} \left[\mathcal{M}_{\mathcal{S}} \odot \text{BCE}\left(f_{\mathcal{S}}(\mathcal{I},\mathcal{G};\ \theta), \mathcal{S}_{\text{gt}}\right)\right].$$

Finally, for UNDC the loss $\mathcal{L}_{\mathcal{F}}(\theta)$ is analogous to $\mathcal{L}_{\mathcal{S}}(\theta)$, and hence we do not repeat its definition.

The definitions of $\mathcal{M}_{\mathcal{S}}$, $\mathcal{M}_{\mathcal{V}}$, and $\mathcal{M}_{\mathcal{F}}$ are as follows. We assume the size of each cell is 1 unit.

$\mathcal{M}_{\mathcal{V}}$ - NDC For a grid cell, if its corner vertices have different signs in the ground truth SDF, we set its corresponding entry in $\mathcal{M}_{\mathcal{V}}$ to 1. The other entries in $\mathcal{M}_{\mathcal{V}}$ are left 0. The definition applies for all kinds of inputs.

$\mathcal{M}_{\mathcal{V}}$ - UNDC For a grid cell, if any of its edges intersects the ground truth shape, we set its corresponding entry in $\mathcal{M}_{\mathcal{V}}$ to 1. The other entries in $\mathcal{M}_{\mathcal{V}}$ are left 0. The definition applies for all kinds of inputs.

SDF grid input - $\mathcal{M}_{\mathcal{S}}$ - NDC NDC directly use the signs of the input as $\mathcal{S}$, therefore it does not need $\mathcal{M}_{\mathcal{S}}$.

SDF and UDF grid input - $\mathcal{M}_{\mathcal{F}}$ - UNDC For an edge in a grid cell, if both of its end vertices have signed distances less than 1, we set its corresponding entry in $\mathcal{M}_{\mathcal{F}}$ to 1. The other entries in $\mathcal{M}_{\mathcal{F}}$ are left 0.

Binary voxel input - $\mathcal{M}_{\mathcal{S}}$ - NDC For an occupied grid cell, if it is adjacent to an unoccupied cell (in its 3^3 local neighborhood), we set the corresponding entries for all its 8 vertices to 1. The other entries in $\mathcal{M}_{\mathcal{S}}$ are left 0.

Binary voxel input - $\mathcal{M}_{\mathcal{F}}$ - UNDC For an edge in a grid cell, if all of its four adjacent cells are occupied, we set its corresponding entry in $\mathcal{M}_{\mathcal{F}}$ to 1. The other entries in $\mathcal{M}_{\mathcal{F}}$ are left 0.

Point cloud input - $\mathcal{M}_{\mathcal{F}}$ - UNDC In the point cloud networks, we only compute features for the cells that are close to the input point cloud, i.e., the cells that are within 3 units (manhattan distance) to the closest point in the point cloud. Therefore, the corresponding edges stored in those cells are set to 1 in $\mathcal{M}_{\mathcal{F}}$. The other entries in $\mathcal{M}_{\mathcal{F}}$ are left 0.

5.3.4 Post-processing

When UNDC is operating on sparse or noisy point clouds, the function $f_{\mathcal{F}}(\mathcal{I},\mathcal{G};\ \theta)$ that predicts grid edge crossings can make mistakes, leading to small holes in the output mesh. Empirically, the holes are typically small and isolated (see Table 5.8), and so we can use a simple post-processing step to close them. We employ our tensor representation of the mesh $\mathcal{F} \in \mathbb{B}^{|\mathcal{E}|}$ to determine boundary edges from $\mathcal{M}$, and we flip Boolean entries in $\mathcal{F}$ that would result in three/four edges to change their boundary state; see Figure 5.7. These post-

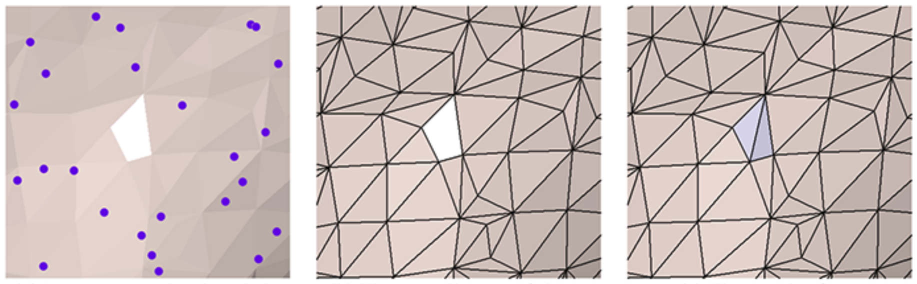

(a) Input point cloud and the mesh before post-processing (b) The tessellation of the mesh before post-processing (c) The mesh after post-processing

Figure 5.7: **Post-processing UNDC outputs** – The post-processing step can close small holes by adding quad faces.

processing steps are executed on the GPU, resulting in a negligible impact on the overall inference time.

5.3.5 Training details

Each network is trained for 400 epochs (for SDF/voxel inputs) or 250 epochs (for point cloud inputs) with a batch size of 1 (shape). We use Adam optimizer [120] with a learning rate of 0.0001, beta1= 0.9 and beta2= 0.999 for optimization. The learning rate is halved every 100 epochs.

5.4 Results and evaluation

We ran a series of experiments with NDC and UNDC to evaluate their performance in comparison to previous methods for a variety of input types, including SDFs, unsigned distance fields (UDF), binary voxel grids, point clouds, and depth image scans.

5.4.1 Datasets, training, and evaluation metrics

In all of our experiments, we train NDC and UNDC on the ABC dataset [122] following the protocols in NMC [39]. The ABC dataset consists of watertight triangle meshes of CAD shapes, which are characterized by their rich geometric features including both sharp edges and smooth curves, as well as their topological varieties. We only use the first chunk of ABC dataset for our experiments. We split the set into 80% training (4,280 shapes) and 20% testing (1,071 shapes). During the data preparation, we obtain meshes over 32^3 and 64^3 grids to train our network. We evaluate the methods on the *test set* of ABC.

Generalization To assess generalization capabilities of the methods, we evaluate on four other datasets, also following the experimental settings as in NMC [39]. The additional test sets include 2,000 shapes from Thingi10K dataset [302], a dataset of 3D-printing models; 100 shapes of human bodies from FAUST dataset [19], a dataset of organic shapes;

several shapes in MGN [17], a dataset of clothes with open surfaces; and several rooms from Matterport3D [27], a dataset containing scans of indoor scenes acquired with depth cameras. In all cases, we evaluate NDC and UNDC after training on the ABC training set without any fine-tuning.

5.4.2 Metrics

We evaluate surface reconstructions quantitatively by sampling 100K points uniformly distributed over the surface of the ground truth shape and the predicted shape, and then computing a suite of metrics that evaluate different aspects of the reconstruction. The metrics are divided into five groups.

Reconstruction accuracy We use Chamfer Distance (CD) and F-score (F1) to evaluate the overall quality of a reconstructed mesh. The metrics are good at capturing significant mistakes such as missing parts, but may not be informative for evaluating the visual quality. Therefore, we introduce other metrics to evaluate sharp feature preservation and surface quality.

Sharp feature preservation We follow NMC [39] and use Edge Chamfer Distance (ECD) and Edge F-score (EF1) to evaluate the preservation of sharp edges. For a given shape, points are sampled near sharp edges and corners to form a set of edge samples. The ECD and EF1 between two shapes are simply the CD and F1 between their edge samples.

Surface quality As in many other papers, we use Normal Consistency (NC) to evaluate the quality of the surface normals. However, NC is similar to CD and F1 in that it mainly captures significant mistakes and neglects small mistakes which contribute significantly to visual artifacts. Therefore, we break down NC to show the percentage of inaccurate normals (% Inaccurate Normals, or %IN) according to a threshold. To compute %IN, for each point sampled from shape A, we find its closest point in the points sampled from shape B, and then compute their angle. If the angle is larger than the threshold, the point from A is labeled as having an inaccurate normal. %IN (gt) is the percentage of points sampled from the ground truth shape that have inaccurate normals. %IN (pred) can be obtained similarly on points from the predicted shape.[3] Another aspect of mesh quality is the number of small angles in the reconstructed triangles. Therefore, we also report the percentage of small angles that are smaller than a threshold, as % Small Angles, or %SA.

[3]Note that these two metrics are the surface normal counterparts of the two terms assembling the symmetric Chamfer Distance.

64^3 SDF input	CD↓ ($\times10^5$)	F1↑	NC↑	ECD↓ ($\times10^2$)	EF1↑	#V	#T	Inference time
NMC	4.365	0.878	0.976	0.340	0.766	42,767	85,544	1.148s
NMC-lite	4.356	0.878	0.975	0.338	0.767	21,933	43,877	1.135s
DC-est	4.673	0.827	0.958	3.810	0.167	**5,459**	10,969	0.421s
MC33	4.873	0.788	0.950	5.759	0.103	5,473	**10,954**	**0.005s**
NMC*	4.400	0.874	0.972	0.409	0.715	42,767	85,544	0.158s
NMC-lite*	4.386	**0.875**	0.973	0.416	0.725	21,933	43,877	0.153s
NDC	4.463	0.867	0.970	0.338	0.745	**5,459**	10,969	0.027s
UNDC	**0.930**	0.873	**0.974**	**0.328**	**0.746**	5,584	11,295	0.051s
UNDC (UDF)	0.960	0.868	0.971	0.379	0.735	5,692	11,420	0.053s

128^3 SDF input	CD↓ ($\times10^5$)	F1↑	NC↑	ECD↓ ($\times10^2$)	EF1↑	#V	#T	Inference time
NMC	4.129	0.882	0.979	0.204	0.806	175,926	351,867	8.991s
NMC-lite	4.117	0.882	0.979	0.231	0.808	88,419	176,853	8.984s
DC-est	4.132	0.879	0.977	2.215	0.266	22,088	44,213	1.765s
MC33	4.144	0.870	0.972	4.247	0.193	**22,048**	**44,107**	**0.030s**
NMC*	4.116	0.882	0.978	0.257	0.779	175,926	351,867	1.126s
NMC-lite*	4.114	0.882	0.979	0.283	0.785	88,419	176,853	1.112s
NDC	4.131	0.881	0.978	0.214	0.802	22,088	44,213	0.207s
UNDC	**0.789**	**0.890**	**0.983**	**0.149**	**0.813**	22,578	45,411	0.410s
UNDC (UDF)	0.792	0.889	0.983	0.227	0.810	22,874	45,715	0.409s

Table 5.2: Quantitative evaluation on **ABC** with **SDF** (signed or unsigned) inputs at two resolutions, evaluated on the test set split, using mesh quality metrics, output complexity, and inference times.

Triangle & vertex counts We count the number of vertices (#V) and triangles (#T) in the output shape to reveal the fidelity-complexity trade-off. Note that while our method generates quad faces, we always randomly split each quad into two triangles for evaluations and visualizations.

Inference time We report inference times (seconds per shape) for the methods tested on the ABC test set. Timings are collected on the same machine with one NVIDIA GTX 1080ti GPU.

5.4.3 Reconstruction from SDF

We first test NDC and UNDC on mesh reconstruction from grids of signed distances, and compare them to Marching Cubes 33 (**MC33**) (an improved version of MC to guarantee topological correctness in each cube [40, 133]), classical DC with estimated normals (**DC-est**), and two versions of Neural Marching Cubes (**NMC** and **NMC-lite**) [39].

DC-est takes the same SDF input as our method, obtains gradient values at grid points by local differentiation over the SDF, and runs the classical DC [110] as described in Figure 5.2 by estimating the intersection points and their gradients via linear interpolation. Since NMC and NMC-lite use large networks to ensure the quality of the output meshes,

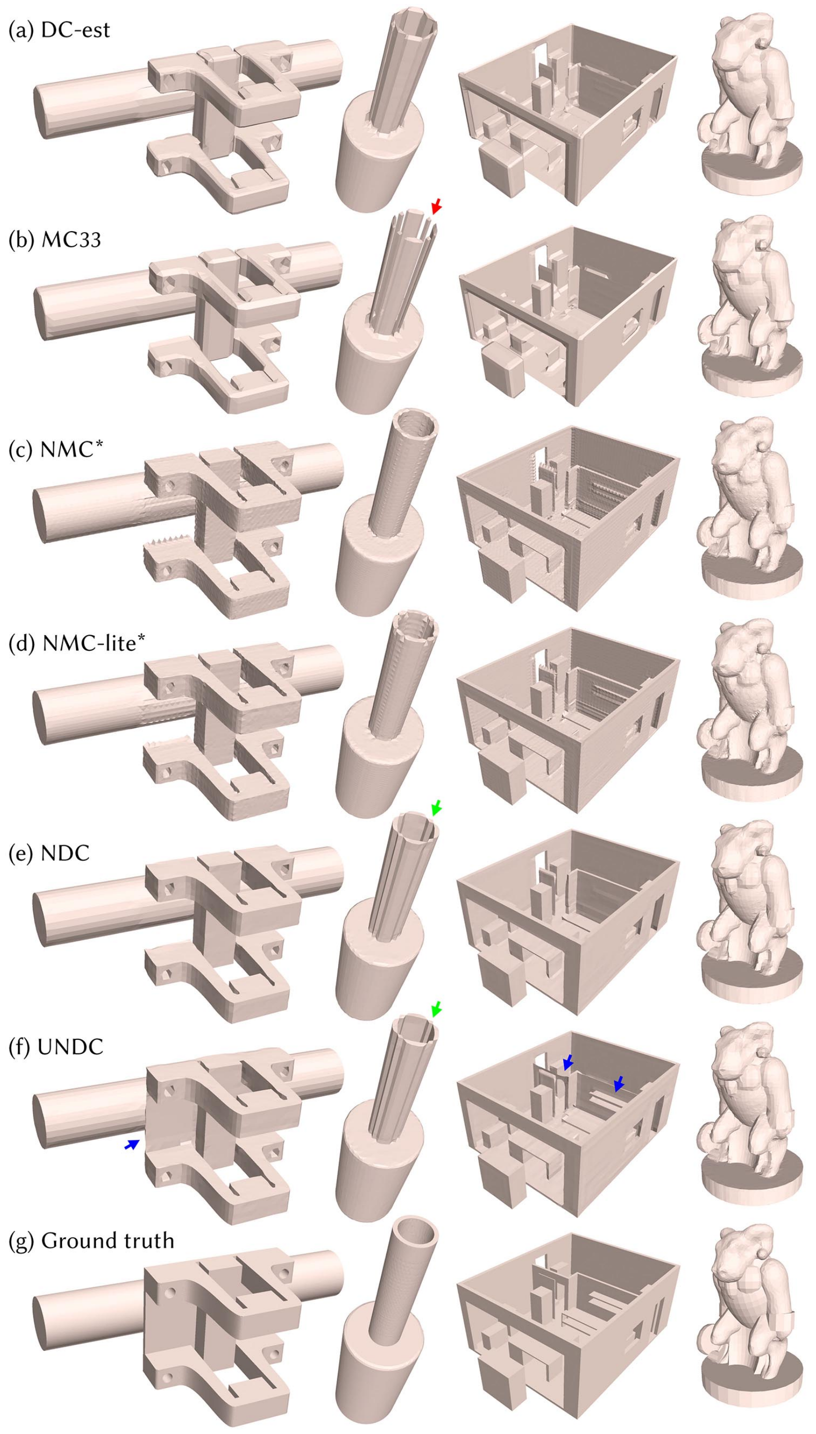


Figure 5.8: Mesh reconstruction results from **SDF** grid inputs at a relatively low resolution of 64^3. The shapes in the first three columns are from **ABC** test set, and the last column from **Thingi10K**. Zoom in to see various surface artifacts and artifacts near edges on NMC-lite* and NMC* results, broken meshes from MC33 (red arrows), and non-manifold edges from NDC and UNDC (green arrows). Pay special attention to the thin sheets (blue arrows) reconstructed by the sign-agnostic UNDC, which correspond to parts of the ground truth shape that are thinner than one voxel. In contrast, none of the other methods (a-e) could even recover any of these thin parts.

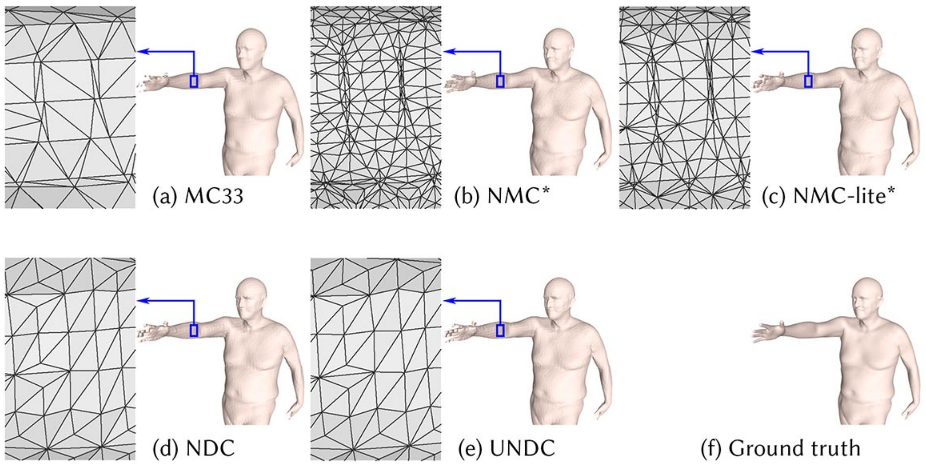


Figure 5.9: Mesh reconstruction results from **SDF** grid inputs at 128^3 resolution on the **FAUST** dataset; see insets to compare triangle quality.

128^3 SDF input	CD↓ ($\times 10^5$)	F1↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T	% IN $> 5°$	% SA $< 10°$
MC33	2.421	0.890	2.657	0.197	22,324	44,656	19.08	2.43
NMC*	2.613	0.902	0.269	0.760	169,211	338,427	20.99	0.77
NMC-lite*	2.651	0.902	0.254	0.772	89,260	178,527	17.04	1.74
NDC	2.300	0.901	0.215	0.792	**22,295**	**44,631**	**12.52**	**0.24**
UNDC	**0.757**	**0.904**	**0.189**	**0.795**	22,478	45,043	12.66	0.29
UNDC (UDF)	0.748	0.903	0.222	0.785	22,784	45,395	13.19	0.28

Table 5.3: Quantitative results on **Thingi10K** with **SDF** input.

128^3 SDF input	CD↓ ($\times 10^5$)	F1↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T	% IN $> 5°$	% SA $< 10°$
MC33	0.453	0.985	0.086	0.387	12,551	**25,076**	**34.28**	4.23
NMC*	0.385	0.990	0.146	0.552	83,024	166,038	44.58	1.18
NMC-lite*	0.381	0.991	0.119	0.567	50,207	100,404	38.33	2.63
NDC	0.397	0.989	0.044	0.530	**12,538**	25,100	38.38	**0.11**
UNDC	**0.362**	**0.992**	**0.038**	**0.574**	12,609	25,258	37.38	0.16
UNDC (UDF)	0.365	0.991	0.045	0.549	12,682	25,293	38.72	0.21

Table 5.4: Quantitative results on **FAUST** with **SDF** input.

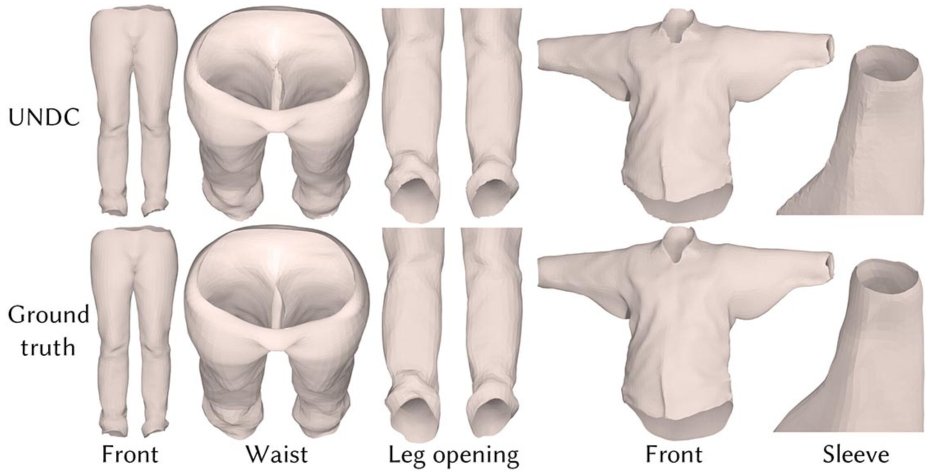


Figure 5.10: Qualitative results of mesh reconstruction from **UDF** inputs at 128^3 resolution on two cloth shapes from the **MGN** dataset. Note the open surfaces reconstructed by our sign-agnostic method **UNDC**.

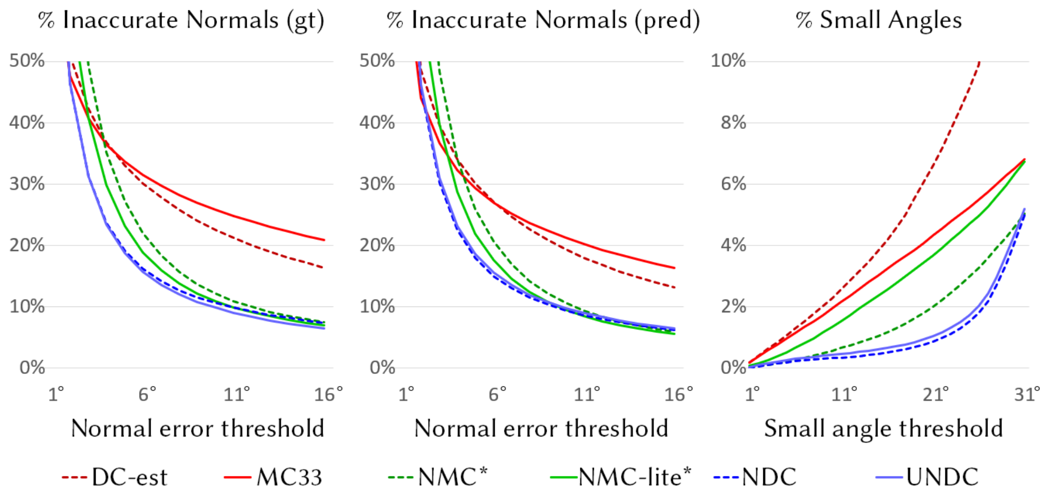


Figure 5.11: Some plots of surface quality (via % of Inaccurate Normals) and triangle quality (via % of Small Angles), on **ABC** test set with 64^3 **SDF** input. NDC and UNDC consistently outperform other isosurfacing methods.

which makes the inference significantly slower, we replace the large backbone networks in them with our 6-layer CNN to obtain **NMC*** and **NMC-lite***, in order to have a fair comparison on reconstruction quality with respect to the inference time. In addition, we include the results of UNDC when the input is a grid of unsigned distances as **UNDC (UDF)**. Since there is no method to reconstruct meshes from grids of unsigned distances to our knowledge, we do not compare it with other methods.

We show visual results in Figure 5.8. We report the quantitative results on the ABC test set in Table 5.2. To make the paper compact, we reduce the sizes of the tables by removing some less representative metrics. The full tables can be found at the end of this chapter: results for the ABC test set are in Table 5.9, Thingi10K in Table 5.10, and FAUST in Table 5.11.

Reconstruction accuracy NDC and UNDC consistently outperform model-driven MC33 and DC-est in terms of CD and F1. Clearly, normal estimation is not expected to be accurate, especially near sharp features. The results of DC-est are similar to or slightly better than those of MC33, as shown in Figure 5.8 and Table 5.2.

Although the network size has been significantly reduced in NMC* and NMC-lite*, they usually have slightly better results than NDC, since, given the same input resolution, Marching Cubes methods are able to reconstruct more inner structures inside each cube with their abundant tessellation templates, while NDC cannot due to its simple tessellation design. However, NMC and NMC-lite are significantly worse than UNDC in CD, even with their original large networks, due to the fact that UNDC can reconstruct thin structures which NMC methods and NDC cannot. Visual results in Figure 5.8 show some examples. Note in the first and the third columns (blue arrows), some thin structures are not reconstructed by any methods other than UNDC. In the fourth column and in Figure 5.9 , we show that even on smooth shapes, NDC and UNDC can preserve more details, such as the crevices, compared to MC33. However, in the second column (green arrows), we show failure cases from NDC and UNDC, where the walls of the tube are merged together with non-manifold edges, as a result of their simpler tessellations in each cube – this problem does not occur in NMC*.

Sharp features As shown in Tables 5.2, 5.3, and 5.4, NDC and UNDC consistently outperform MC33, DC-est, NMC*, and NMC-lite*, in ECD and EF1, for feature preservation. The only exception is the EF1 in Table 5.4, which may be due to NDC's tessellation being too simple to handle the fine structures of human shapes, e.g., the fingers.

Normal quality Since the compared methods generally have similar point-wise reconstruction accuracies, the quality of the generated surfaces in terms of visual appearance can be a better differentiator. We consistently observe that after switching to smaller networks, NMC* and NMC-lite* tend to generate noisy surfaces even over flat regions, as shown in Figure 5.8 (c-d). We use %IN (IN = inaccurate normals) as a means to quantify the quality of surface normals, which correlate with surface quality. Figure 5.11 shows the %IN-threshold curves on the ABC test set, where the normal errors of NDC and UNDC are noticeably less than those from other methods for small threshold values, which is consistent with our visual observation that NMC* and NMC-lite* outputs exhibit more surface artifacts.

Triangle quality In Figure 5.11, we show %SA-threshold (SA = small angles) curves for various methods on the ABC test set, showing that NDC outperforms the others in triangle quality, with UNDC coming close. A visual comparison can be found in Figure 5.9.

Triangle and vertex count The #V and #T in Table 5.2 show that the vertex and triangle numbers of NDC and UNDC are very similar to those of MC33. NMC and NMC-lite produce more vertices and triangles due to their complex cube tessellation templates.

Inference time With no deep networks involved, MC33 is undoubtedly the fastest, as shown by the inference times in Table 5.2. With our light network design, NDC and UNDC are next in line in terms of speed. Since NDC does not need sign predictions, it is half the size of and twice as fast as UNDC, running in real time on an NVIDIA GTX 1080ti GPU. With the newer RTX 3090 being twice as fast as GTX 1080ti, we expect UNDC to also run in real time on a higher-end GPU. In comparison, the original NMC and NMC-lite require more than a second to test on an input grid of 64^3. Even after we replace their networks with our light designs, NMC* and NMC-lite* are still 2-4 times slower than UNDC and NDC, due to their more complex cube tessellations. Finally, the inference time for classical DC is far from optimal as we employed our own implementation of DC-est with an unoptimized QEF solver.

Robustness to translation and rotation We highly encourage the readers to watch the video https://youtu.be/HwKMpeKgYcc where we test the methods on a shape while moving and rotating the shape inside the sampling grid. It clearly shows that NDC and UNDC are the most robust compared to others.

Varying input grid resolutions We report quantitative results obtained by the various methods on 64^3 and 128^3 input resolutions in Table 5.2. When increasing the input resolution, the gap of reconstruction accuracy diminishes, as reflected by CD and F1. But NMC* and NMC-lite* will always produce significantly more vertices and triangles, and take more time to process a shape.

Generalizability To show the generalizability of our method, we show the quantitative results on Thingi10K and FAUST in Table 5.3 and 5.4, respectively. Visual results can be

64^3 Voxel input	CD↓ ($\times 10^5$)	F1↑	NC↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T	Inference time
MC33	26.862	0.085	0.921	11.342	0.018	5,826	11,656	**0.005s**
NMC*	9.452	0.422	0.927	0.698	0.346	42,045	84,089	0.156s
NMC-lite*	9.428	0.420	0.927	0.604	0.356	21,431	42,862	0.154s
NDC	9.387	**0.428**	0.930	0.567	**0.360**	**5,345**	**10,726**	0.055s
UNDC	**9.139**	**0.428**	**0.931**	**0.564**	0.359	5,365	10,772	0.055s

Table 5.5: Quantitative results on **ABC** test set with **binary voxel** input.

found in Figure 5.8 and 5.9. All data-driven methods are only trained on the ABC training set. These results are consistent with our analysis above. Specifically, in terms of surface quality, we show %IN (pred) with error greater than 5° in Table 5.3 to show that NMC* and NMC-lite* have more surface artifacts. However, on organic shapes from FAUST, MC33 outperforms deep learning methods, as shown by %IN (pred) in Table 5.4. It makes sense, because Marching Cubes was originally designed to reconstruct smooth shapes, and none of the deep learning methods are trained on smooth shapes. We also report %SA with angles less than 10° in Table 5.3 and Table 5.4 for Thingi10K and FAUST, to show that our method produces fewer small-angle triangles.

5.4.4 Reconstruction from UDF

As shown in the last rows of Table 5.2, 5.3, and 5.4, the results on UDF are similar to those on SDF, but are usually worse due to the lack of signs. Still, UNDC is able to recover the shapes reasonably well by just observing the changes of unsigned distances in nearby cells. The visual results on UDF are very similar to the results on SDF when tested on the three datasets in the previous experiment. Therefore, we show the results of reconstructing clothes with open surfaces in MGN dataset [17] from grids of unsigned distances in Figure 5.10. Note that in our experiments with UDF, we do not compare with prior works, since, to the best of our knowledge, UNDC is the only method that can reconstruct meshes from UDF.

5.4.5 Reconstruction from binary voxels

Reconstructing meshes from binary voxels is clearly more challenging than from SDF grids. Learning from data is absolutely necessary in this scenario if one wishes to produce plausible outputs, as reflected in Table 5.5, where MC33 is significantly worse than all others in all metrics, except for vertex and triangle count. The results on Thingi10K and FAUST, and on 128^3 inputs are at the end of this chapter: the results for the ABC test set are in Table 5.12, Thingi10K in Table 5.13, and FAUST in Table 5.14. They show the same pattern as Table 5.5 and demonstrate the generalizability of our method. We show visual results in Figure 5.12. Many observations in Section 5.4.3 still apply here: our method is significantly faster than NMC* and NMC-lite* (Inference time), produces significantly less

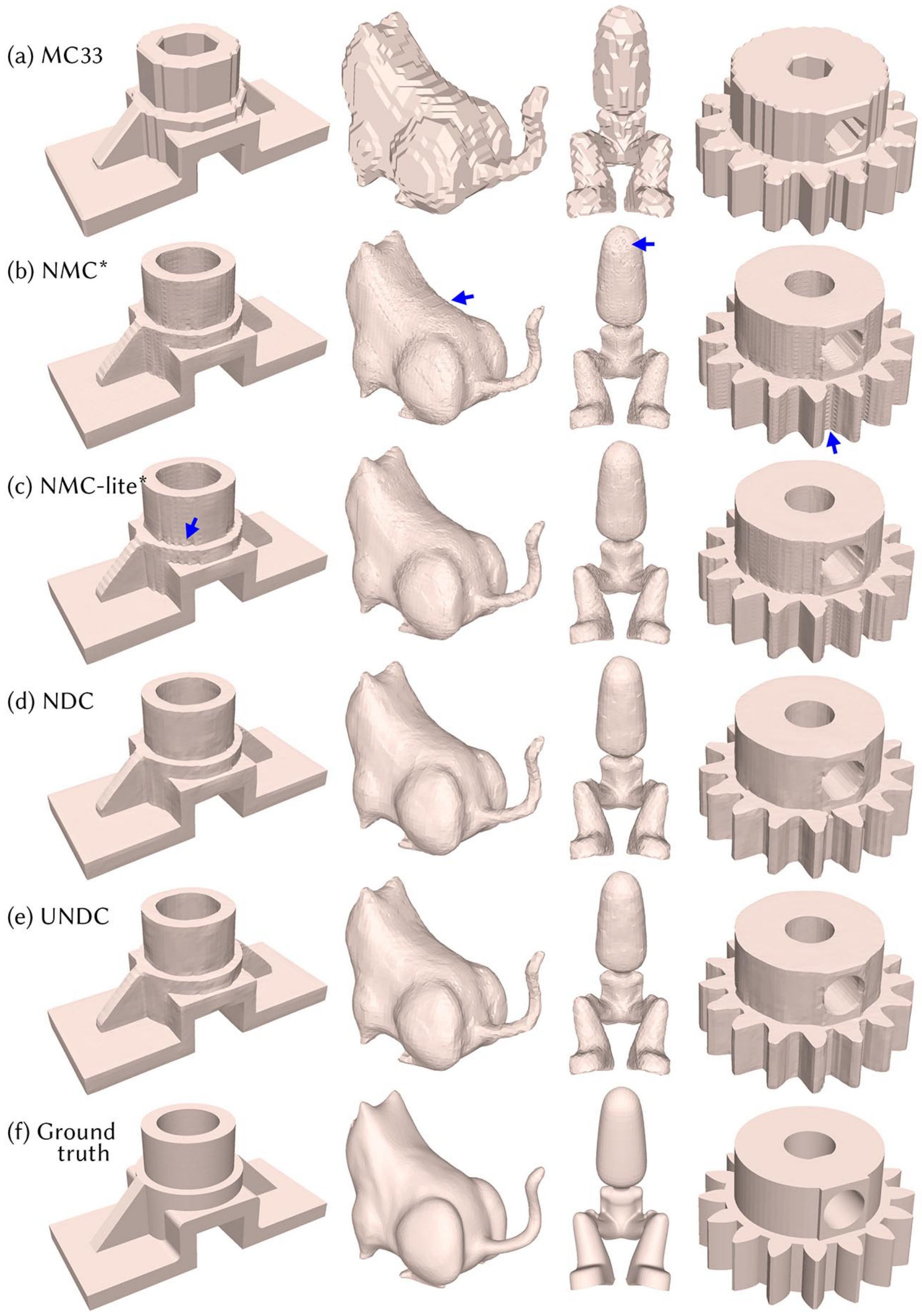


Figure 5.12: Mesh reconstruction results from **binary voxel** (**occupancy**) inputs at 64^3 resolution. Zoom in to see some surface artifacts by NMC-lite* and NMC*, marked with blue arrows. The shapes in the first column are from **ABC** test set, and the last three columns from **Thingi10K**.

vertices and triangles (#V, #T), has better normal quality (NC), and can better preserve sharp edges and corners (ECD, EF1). Moreover, since binary voxels are more challenging than SDF grids, NMC* and NMC-lite* are underfitting, with reconstruction accuracy worse than NDC and UNDC, as reflected by CD and F1.

5.4.6 Reconstruction from point clouds

We test UNDC on the task of reconstructing meshes from point clouds. UNDC *does not* require normals as input, while most other methods do, making direct comparisons difficult to perform. To give competing methods a slight advantage, we provide normals to methods that require them, and re-iterate that our method *does not* leverage this additional information.

Baselines We compare against five methods, including classical Ball-pivoting [15] and screened Poisson [118]) surface reconstruction, as well as three deep learning methods like SIREN [227], Local Implicit Grids (LIG) [107], and Convolutional Occupancy Networks (ConvONet) [193]. The latter is the only method that does not require point normals, and we test its two variations proposed in the original paper: ConvONet-3plane which uses the 3-plane (xy, yz, xz) setting, and ConvONet-grid that uses the 3D grid setting. Note that we compare with SIREN rather than SAL [5] since SIREN is built upon SAL and has shown better performance in their paper.

Ball-pivoting [15] and Screened Poisson surface reconstruction (**Poisson**) [118]. These are classic methods for reconstructing meshes from point clouds, and they require point normals as part of the input. Ball-pivoting does not create new vertices - it only connects the existing vertices into consistently oriented triangles. Poisson constructs an implicit field according to the points and normals, then extract the surface with an octree structure. In our experiments, we use the implementation in Open3D [301] for these two methods, and use a maximum depth of 8 for the octrees in Poisson.

SIREN [227] is a method that overfits an neural implicit function to a given shape, therefore it takes much longer to process a shape than all other methods, since each time it needs to train a neural network from scratch. It requires point normals as part of the input. In our experiments, we use the official code released by the authors. We find that after training SIREN, there is a 10% possibility that the output shape is covered by a shell, which cannot be easily removed since it is close to the actual shape and connected to it in many pieces. Therefore, for those shapes we have to re-train the network for several times. Nonetheless, we report the inference time in the tables assuming all shapes are successfully trained in the first go.

Local implicit grid (**LIG**) [107] is a method that first divides the input point cloud into small overlapping blocks, and then reconstruct the part in each block by optimizing a neural implicit field, and finally put the implicit fields together to reconstruct the entire

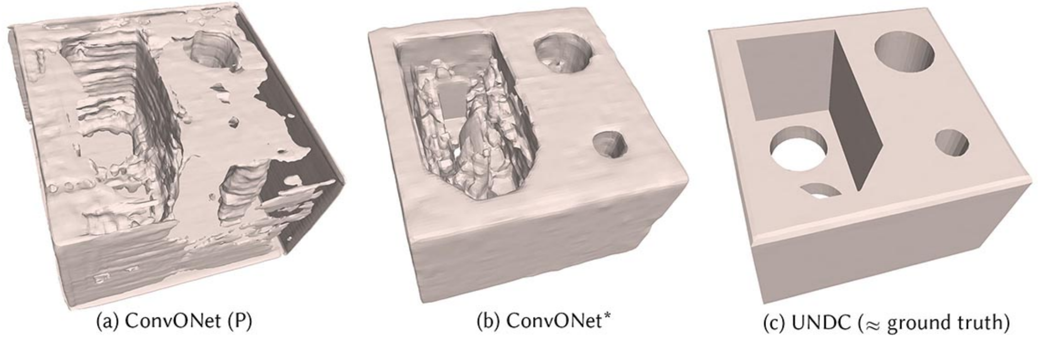


Figure 5.13: Pre-trained ConvONet vs. ConvONet with our local backbone.

shape. It requires point normals as part of the input. In our experiments, we use the official code released by the authors. The authors have released pre-trained network weights on ShapeNet [28], therefore we denote this method with pre-trained weights as **LIG (P)**. We also train this method on ABC training net for a fair comparison, denoted as **LIG**. We use 320^3 output resolution for both models.

Convolutional occupancy networks (**ConvONet**) [193] is a method to reconstruct an implicit field from point clouds. It does not require point normals as input, therefore is the only method that takes the exact same input as our method. In our experiments, we use the official code released by the authors. The authors have introduced many network configurations in their paper, and we choose two representative ones in our experiments. In **ConvONet 3plane**, we use the 3-plane (xy, yz, xz) setting with the resolution of each plane 128^2. In **ConvONet grid**, we use the 3D grid setting with the grid resolution 64^3. We train both networks on ABC training net for a fair comparison. We also use the network weights released by the authors, pre-trained on synthetic scenes with objects from ShapeNet [28], denoted as **ConvONet (P)**. We use 256^3 output resolution for all three models.

It is worth noting that unfortunately, all the networks in ConvONet are *non-local*, that is, their receptive fields need to cover the entire shape in order to properly decide which side is inside and which side is outside in the output implicit field. We tried to directly apply our backbone network in ConvONet, denoted as **ConvONet***, but as expected, the training has failed. This is because our network is local, and ConvONet cannot decide inside/outside for a local patch, therefore generates many artifacts in the featureless regions, as shown in Figure 5.13. Note also that the pre-trained ConvONet tends to turn single-face walls into thin volumetric plates in Figure 5.13.

We test all methods with 4,096 input points. The output grid size of UNDC is 64^3. We train all data-driven methods on the ABC training set for a fair comparison. We illustrate these results in Figure 5.14. Quantitative results on the ABC test set are provided in Table 5.6, while results on Thingi10K and FAUST, which reveal a similar pattern and trend,

point cloud (4,096)	CD↓ ($\times 10^5$)	F1↑	NC↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T	Inference time
Ball-pivoting (+n)	3.080	0.791	0.944	0.556	0.269	**4,096**	**7,439**	1.292s
Poisson (+n)	4.705	0.727	0.939	4.138	0.067	11,241	22,496	1.476s
SIREN (+n)	1.340	0.814	0.969	2.636	0.152	97,219	194,543	168.595s
LIG (+n)	3.413	0.721	0.947	11.868	0.022	149,860	299,166	66.866s
ConvONet 3plane	18.073	0.536	0.935	4.113	0.105	75,342	150,689	2.692s
ConvONet grid	8.844	0.488	0.939	9.701	0.036	74,171	148,337	2.404s
UNDC	**0.893**	**0.873**	**0.974**	**0.289**	**0.757**	5,578	11,261	**0.194s**

Table 5.6: Quantitative results on **ABC** test set with **point cloud** input. (+n) indicates that the method additionally requires point normals as input.

can be found at the end of this chapter: results for the ABC test set are in Table 5.15, Thingi10K in Table 5.16, and FAUST in Table 5.17.

Analysis As shown in Table 5.6, UNDC outperforms all other methods in all reconstruction quality metrics. SIREN has the closest results to ours in terms of reconstruction accuracy, but it has to be trained for, i.e., “overfit to”, each input shape. ConvONet, whose networks are not local, does not generalize well even within the ABC dataset. LIG is local and therefore expected to generalize better. However, its algorithm only considers the space around the input points and ignores the empty space. As a result, LIG generates many artifacts in the empty space, which are called “back-faces” in their paper, and a post-processing step is required to remove them. The post-processing step is not perfect, as shown in the first and fourth columns in Figure 5.14. UNDC and Ball-pivoting are the only two methods that directly output a mesh without iso-surface extraction, therefore they have the least numbers of vertices and triangles, and are the only two methods that can generate sharp features, as shown by ECD and EF1 in Table 5.6. As for inference time, UNDC is the fastest and significantly faster than all other methods compared, even the classical Ball-pivoting and Poisson.

5.4.7 Reconstruction from noisy real scans

We test UNDC on reconstructing meshes from raw scan data in Matterport3D [27]. The raw scan data contains depth images and camera parameters – we convert them into noisy point clouds as the input to our network. Since the point clouds represent large scenes, we first crop the scene into overlapping patches, and then run our network to obtain grids of edge intersection flags and vertex locations. Finally, we put together the predicted grids to form a large grid of the scene, and then run the meshing algorithm in Figure 5.4 to obtain the output mesh. Our network is larger than that of the previous experiment to accommodate for point cloud noise. Specifically, we replace the three 3^3 conv3d layers in Figure 5.6 with 8 residual blocks [95]. We train the network on the same ABC training

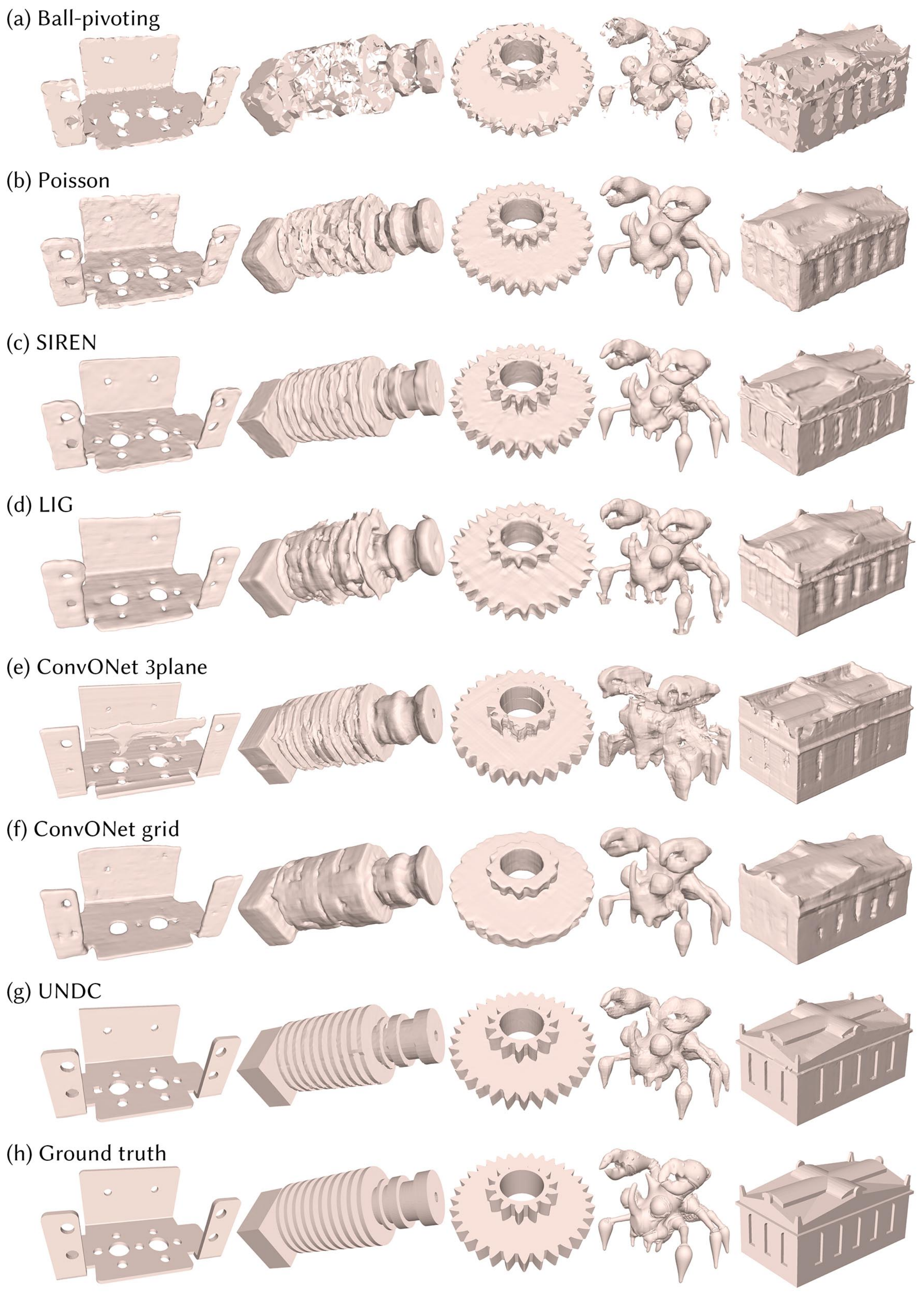


Figure 5.14: Results of reconstructing 3D meshes from **point cloud** inputs of 4,096 points. Please zoom in to observe the surface details. The shapes in the first two columns are from **ABC** test set, and the last three columns from **Thingi10K**.

set but with heavy data augmentation (random scaling and translation, in addition to the augmentations mentioned in the paper). During training, we also augment the input point clouds with Gaussian noise ($\sigma = 0.5$, assuming each cell of the output grid is a unit cube) to simulate the real noise from scan data.

Baselines We compare UNDC with ConvONet, since it does not require ground truth point normals and is designed to reconstruct large scenes. We use the pre-trained weights provided by the authors for synthetic scenes with objects from ShapeNet [28], denoted as ConvONet (P). Additionally, we compare to Poisson with estimated point normals. We show visual comparisons in Figure 5.15. Note that different from the experiments in most other works on deep learning scene reconstruction, which test their methods on sampled points from the "ground truth" meshes, we test on raw scan data, which is a more realistic setting. We do not report quantitative results since the "ground truth" meshes provided with the dataset were reconstructed by Poisson, one of the methods we are comparing against.

Analysis ConvONet is seen to significantly underperform compared to Poisson and UNDC, especially falling short in terms of surface quality and detail preservation. Therefore, we mainly compare UNDC with Poisson. Generally, UNDC produces less surface noise, which is especially obvious in the first row of Figure 5.15. The improvements are also observable in the other two rows, but they are less obvious due to the zoom-out to reveal the entire scenes. Since UNDC is trained on data with noise augmentation, it learns, to some extent, to remove noise.

Also, UNDC only reconstructs what is given in the input point cloud. In contrast, Poisson creates an implicit field of the scene, which could potentially inpaint the missing regions. However, such inpainting is not always desirable, and Poisson needs to trim the output mesh to remove surfaces that are generated in empty regions using a post-process (SurfaceTrimmer) that depends on careful tuning of parameters. If the trimming density threshold is too small, it may leave "bubble" artifacts as indicated by red arrows in Figure 5.15. If it is too large, it may accidentally trim the objects, as indicated by purple arrows in Figure 5.15. At the default setting shown, UNDC tends to produce more holes in the output, but avoids creating bubble artifacts, see the last row of Figure 5.15.

While water-tightness can be beneficial as a prior, it can lead to poor reconstruction of thin surfaces; this can be observed from the blue arrows in Figure 5.15. One thing worth special attention is the bottom-most red arrow in the last row of Figure 5.15. The umbrella surface is thinner than a voxel. However, Poisson forces inside-outside by creating a bubble on top of the umbrella, so that the bottom of the bubble can form the surface of the umbrella. This creates an odd boundary after trimming.

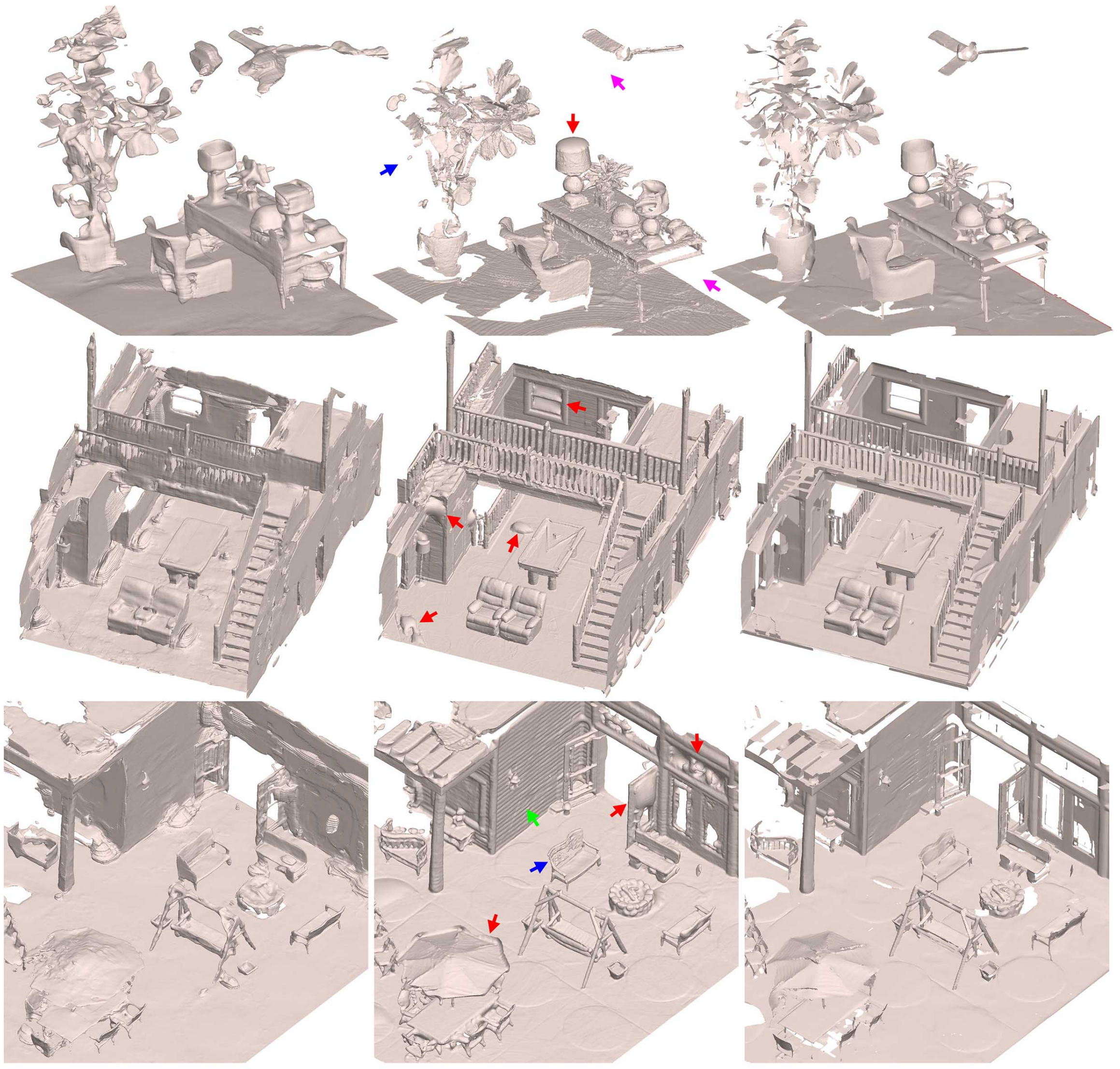


Figure 5.15: Qualitative comparison between ConvONet, Poisson, and **UNDC** on reconstructing rooms in **Matterport3D** from **raw scan data**, where some walls and roofs are removed for better visualization. Colored arrows bring attention to regions where Poisson should be contrasted against UNDC. Red arrows: "bubble" artifacts caused by the watertightness prior to Poisson; purple arrows: objects or parts incorrectly trimmed; blue arrows: poor reconstruction of thin surfaces. Green arrow in the bottom row points out an instance of better preservation of surface details by Poisson (the strip patterns are *not* noise or reconstruction artifacts); the flip side of this, however, is surface noise, as seen over Poisson reconstruction in the first row.

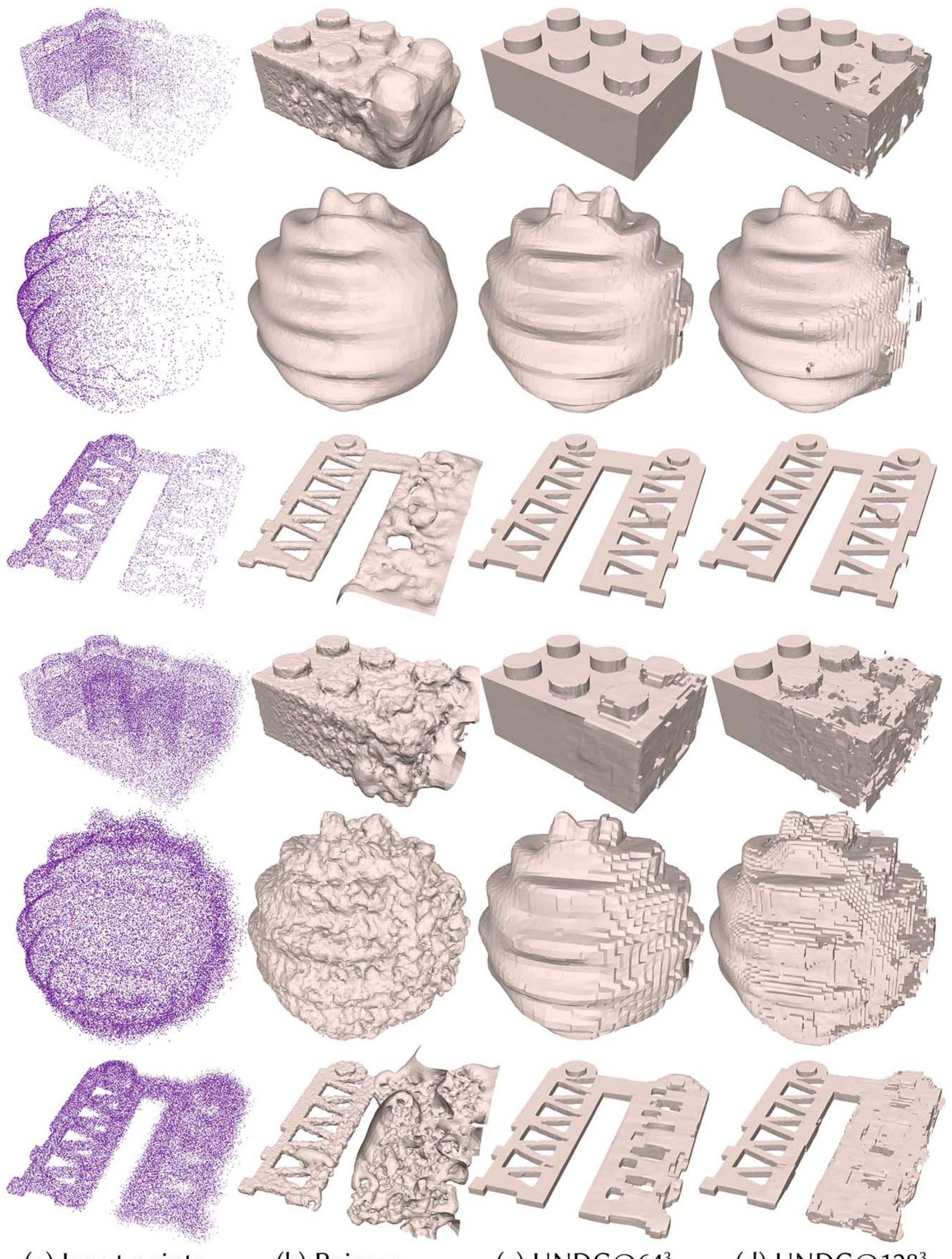


Figure 5.16: Qualitative comparison between Poisson and **UNDC** on mesh reconstruction from **point clouds** with density or noise variations *across the same shape, from its left to its right, as shown.* The input point clouds in the first three rows have decreasing point density from left to right, while the inputs in the last three rows have increasing noise. UNDC@64^3 and UNDC@128^3 produce output grid sizes of 64^3 and 128^3, respectively.

Number of input points	Gaussian noise levels	CD↓ ($\times 10^5$) Poisson	UNDC	F1↑ Poisson	UNDC	NC↑ Poisson	UNDC
1,024	None	34.872	**2.510**	0.248	**0.806**	0.799	**0.944**
2,048	None	16.406	**1.226**	0.387	**0.850**	0.847	**0.962**
4,096	None	8.653	**0.987**	0.539	**0.867**	0.879	**0.970**
4,096	$\sigma = 0.2$	9.814	**1.179**	0.480	**0.840**	0.863	**0.962**
4,096	$\sigma = 0.5$	14.017	**2.061**	0.331	**0.717**	0.821	**0.935**
16,384	$\sigma = 0.2$	5.286	**0.936**	0.636	**0.866**	0.889	**0.971**
16,384	$\sigma = 0.5$	8.738	**1.236**	0.444	**0.813**	0.840	**0.955**
65,536	$\sigma = 0.2$	2.299	**0.905**	0.741	**0.872**	0.930	**0.973**
65,536	$\sigma = 0.5$	4.567	**1.079**	0.552	**0.836**	0.880	**0.962**

Table 5.7: Comparing reconstruction results of **UNDC** (output grid size at 64^3) and Poisson on **point cloud** inputs from **ABC** test set, with varying point counts and noise levels to test the robustness of our method.

Robustness to varying point density and noise We use synthetic data to study the robustness of UNDC on point clouds with varying density and noise. We train our network with point clouds whose point counts were randomly selected between 2,048 and 32,768, where each point cloud is augmented with Gaussian noise whose σ is randomly sampled from $[0, 0.5]$. We then evaluate the trained network on point clouds of varying density and noise, and compare it to Poisson reconstruction with estimated point normals.

Some quantitative results are shown in Table 5.7, where UNDC evidently outperforms Poisson. In Figure 5.16, we show visual results where the point density or noise varies *within* each input point cloud. Note that UNDC at 128^3 output resolution tends to produce worse results than at 64^3 output resolution. This is because relatively, point sparsity and noise level are both more significant at higher-resolution grids, due to the smaller cell sizes.

5.5 Conclusions

We introduce neural dual contouring, a new data-driven approach to mesh reconstruction based on dual contouring. The volumetric version of our approach, NDC, takes the same input as MC and NMC, and it can better preserve sharp features while using approximately the same number of vertices and triangles as classical MC, which is 3-7 times reduction compared to NMC. The surface version of our approach, UNDC, is sign agnostic; it is therefore able to reconstruct open surfaces and thin structures from unsigned distance fields or unoriented point clouds. Both NDC and UNDC are designed as local networks using limited receptive fields, thus can generalize well to new datasets. Extensive experiments demonstrate the superior performance of our approach on multiple datasets over state-of-the-art methods, whether learned (e.g., NMC, SIREN, LIG, ConvONet) or traditional (e.g., MC33, Poisson, Ball-Pivoting).

Input	64^3 SDF	64^3 SDF	64^3 SDF	$4,096$ points	$4,096$ points
Method	NDC	UNDC	UNDC	UNDC	UNDC
Post-processing	No	No	Yes	No	Yes
Non-manifold-3	0.0 (0.000%)	125.3 (1.116%)	135.4 (1.206%)	139.4 (1.242%)	168.0 (1.496%)
Non-manifold-4	20.1 (0.183%)	31.2 (0.278%)	31.7 (0.282%)	28.8 (0.257%)	30.4 (0.271%)
Boundary-1	0.0 (0.000%)	56.7 (0.505%)	29.6 (0.264%)	116.4 (1.037%)	41.1 (0.366%)

Table 5.8: Statistics on non-manifold and boundary edges produced by NDC and UNDC. The methods are tested on ABC test set with 64^3 output resolution. Non-manifold-3 denotes non-manifold edges with 3 adjacent faces, and Non-manifold-4 denotes those with 4 adjacent faces. Boundary-1 refers to boundary edges, defined as edges with only one adjacent face.

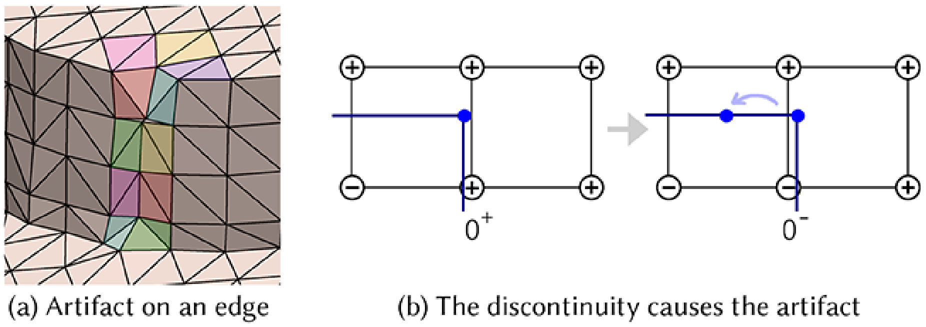


Figure 5.17: The edge artifacts and the cause. The quad faces are colored differently to show that the artifacts are not caused by *random* quad splitting.

Limitations One limitation of our approach is that it can produce non-manifold meshes. Specifically, since DC and its descendants produce only one vertex per grid cell, they may create meshes with vertices and edges shared by multiple surface patches in cases where MC would output multiple disconnected components within one cell; see the second column of Figure 5.8 where the green arrows indicate non-manifold edges created by NDC and UNDC. These cases happen fairly rarely (see statistics in Table 5.8) and are easily detected and fixed by splitting vertices/edges or “tunnelling" through them, using the techniques described in [177] or [213], for manifold dual contouring.

However, UNDC can also produce open surfaces (with edges connected to one face) or non-manifold fins (where edges are shared by three faces). Creating open surfaces is generally good, as it allows reconstruction of thin features and partial inputs (e.g., note the thin sheets indicated by the blue arrows in the row of UNDC results in Figure 5.8 better approximate the ground truth). However, boundaries and fins may cause problems for downstream tasks that assume manifoldness as a pre-condition. Hence, UNDC may not be the best meshing solution for all applications.

Another limitation is that the output of NDC is not completely invariant to orientation. Although NDC is empirically less sensitive to rotations than NMC, we still see that NDC occasionally generates coherence artifacts on sharp edges as an object rotates. One example is shown in Figure 5.17 (a). The artifact occurs when one or more vertices of the cube have

SDF values very close to 0. It cannot be easily avoided since it is due to the continuity of neural networks. See the illustration in Figure 5.17 (b). When the SDF value of the vertex gradually moves from positive (outside) to negative (inside), the input to the network (the SDF values) changes smoothly, but the output needs to change in a discontinuous way in order to produce the ground truth. Since most neural networks are continuous, the output of the network will be continuous. Therefore the network will generate artifacts when such transitions occur.

Future works Besides fixing the issues above, it would be interesting to incorporate the NDC framework into an end-to-end system for recovering surfaces from neural representations inferred from multiple images, possibly using Neural Radiance Fields [167]. Or, UNDC could potentially be used with differentiable rendering to reconstruct one-sided surfaces from a sparse set of images acquired from cameras inside a scene. These and other NDC extensions are promising topics for future work.

64^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	Inference	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
SDF grid input	($\times 10^5$)			($\times 10^2$)				time	$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	4.365	0.878	0.976	0.340	0.766	42,767	85,544	1.148s	2.15	3.79	12.11	1.28	2.50	10.71	0.74	2.09	4.94
NMC-lite	4.356	0.878	0.975	0.338	0.767	21,933	43,877	1.135s	2.16	3.78	11.61	1.33	2.56	10.31	1.51	3.64	6.74
DC-est	4.673	0.827	0.958	3.810	0.167	**5,459**	10,969	0.421s	1.68	9.32	30.17	0.64	6.62	27.12	2.29	6.08	15.24
MC33	4.873	0.788	0.950	5.759	0.103	5,473	**10,954**	**0.005s**	**1.05**	13.48	31.50	**0.40**	9.72	27.03	1.95	4.09	6.53
NMC*	4.400	0.874	0.972	0.409	0.715	42,767	85,544	0.158s	2.01	4.39	22.03	1.17	2.98	20.68	0.57	1.81	4.63
NMC-lite*	4.386	**0.875**	0.973	0.416	0.725	21,933	43,877	0.153s	2.05	4.32	18.92	1.23	**2.97**	17.56	1.35	3.44	6.34
NDC	4.463	0.867	0.970	0.338	0.745	**5,459**	10,969	0.027s	2.45	4.66	16.20	1.48	3.52	**15.03**	**0.33**	**0.76**	**4.16**
UNDC	**0.930**	0.873	**0.974**	**0.328**	**0.746**	5,584	11,295	0.051s	1.65	**3.71**	**15.75**	1.52	3.69	15.61	0.44	0.93	4.41
UNDC (UDF)	0.960	0.868	0.971	0.379	0.735	5,692	11,420	0.053s	1.80	3.93	16.10	1.70	4.08	16.21	0.36	0.89	4.21
128^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	Inference	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
SDF grid input	($\times 10^5$)			($\times 10^2$)				time	$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	4.129	0.882	0.979	0.204	0.806	175,926	351,867	8.991s	2.10	2.98	8.25	1.26	1.82	6.97	0.72	1.84	4.17
NMC-lite	4.117	0.882	0.979	0.231	0.808	88,419	176,853	8.984s	2.12	2.97	7.76	1.28	1.84	6.53	1.46	3.35	5.99
DC-est	4.132	0.879	0.977	2.215	0.266	22,088	44,213	1.765s	1.40	5.10	17.11	0.34	2.85	14.54	1.62	4.36	13.10
MC33	4.144	0.870	0.972	4.247	0.193	**22,048**	**44,107**	**0.030s**	**0.88**	7.81	18.73	**0.18**	4.95	15.42	1.75	3.63	5.77
NMC*	4.116	0.882	0.978	0.257	0.779	175,926	351,867	1.126s	1.90	3.22	15.25	1.14	2.00	13.99	0.60	1.68	4.01
NMC-lite*	4.114	0.882	0.979	0.283	0.785	88,419	176,853	1.112s	1.91	3.15	12.60	1.18	**1.97**	11.35	1.37	3.26	5.77
NDC	4.131	0.881	0.978	0.214	0.802	22,088	44,213	0.207s	2.20	3.11	9.62	1.31	1.99	**8.43**	**0.23**	**0.49**	**3.49**
UNDC	**0.789**	**0.890**	**0.983**	**0.149**	**0.813**	22,578	45,411	0.410s	1.32	**2.06**	**8.90**	1.30	2.04	8.77	0.34	0.65	3.74
UNDC (UDF)	0.792	0.889	**0.983**	0.227	0.810	22,874	45,715	0.409s	1.36	2.11	8.93	1.31	2.09	8.88	**0.23**	0.55	3.51

Table 5.9: Quantitative comparison results on ABC test set with SDF and UDF grid input.

64^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
SDF grid input	($\times 10^5$)			($\times 10^2$)				$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	2.434	0.895	0.974	0.284	0.735	40,952	81,911	1.58	3.95	17.57	1.34	3.40	16.92	0.96	2.78	6.59
NMC-lite	2.485	0.895	0.974	0.308	0.738	22,051	44,109	1.57	3.94	16.51	1.38	3.48	15.97	1.89	4.77	9.12
MC33	3.192	0.795	0.945	3.918	0.099	5,518	11,044	**0.76**	14.30	37.41	**0.55**	11.14	33.54	2.63	5.45	8.70
NMC*	2.777	**0.890**	**0.969**	0.391	0.662	40,952	81,911	1.47	4.92	31.21	1.21	4.23	30.53	0.75	2.37	6.12
NMC-lite*	2.760	**0.890**	**0.969**	0.404	0.674	22,051	44,109	1.52	**4.82**	27.10	1.28	**4.21**	26.42	1.68	4.47	8.49
NDC	2.481	0.877	0.966	0.390	**0.695**	**5,473**	**11,027**	1.88	5.39	23.59	1.57	5.04	**23.14**	**0.35**	**1.12**	**5.87**
UNDC	**0.899**	0.878	0.967	**0.369**	0.693	5,529	11,175	1.62	5.00	**23.52**	1.57	5.17	23.56	0.41	1.24	6.08
UNDC (UDF)	0.938	0.870	0.962	0.407	0.669	5,640	11,297	1.89	5.53	24.38	1.87	5.93	24.67	0.39	1.30	5.92
128^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
SDF grid input	($\times 10^5$)			($\times 10^2$)				$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	2.340	0.902	0.980	0.170	0.805	169,210	338,426	1.48	2.65	10.90	1.29	2.28	10.46	0.92	2.50	5.76
NMC-lite	2.398	0.902	0.980	0.163	0.810	89,260	178,527	1.48	2.63	9.98	1.32	2.30	9.59	1.83	4.46	8.26
MC33	2.421	0.890	0.972	2.657	0.197	22,324	44,656	**0.48**	7.47	21.65	**0.27**	5.46	19.08	2.43	5.04	7.92
NMC*	2.613	0.902	0.978	0.269	0.760	169,211	338,427	1.34	3.00	21.42	1.15	2.57	20.99	0.77	2.25	5.50
NMC-lite*	2.651	0.902	0.979	0.254	0.772	89,260	178,527	1.37	2.94	17.49	1.20	2.54	17.04	1.74	4.35	7.91
NDC	2.300	0.901	0.979	0.215	0.792	**22,295**	**44,631**	1.53	2.81	12.88	1.36	2.50	**12.52**	**0.24**	**0.75**	**5.07**
UNDC	0.757	**0.904**	**0.981**	**0.189**	**0.795**	22,478	45,043	1.31	**2.50**	**12.71**	1.29	**2.48**	12.66	0.29	0.85	5.28
UNDC (UDF)	**0.748**	0.903	0.980	0.222	0.785	22,784	45,395	1.35	2.63	13.23	1.30	2.61	13.19	0.28	0.90	5.08

Table 5.10: Quantitative comparison results on Thingi10K with SDF and UDF grid input.

128^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
SDF grid input	($\times 10^5$)			($\times 10^2$)				$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	0.376	0.991	0.989	0.041	0.645	83,023	166,034	0.18	1.86	25.45	0.13	1.52	25.15	1.37	4.28	10.11
NMC-lite	0.374	0.991	0.989	0.038	0.639	50,207	100,402	0.17	1.83	25.36	0.14	1.57	25.16	2.63	7.22	13.91
MC33	0.453	0.985	0.984	0.086	0.387	12,551	**25,076**	0.33	2.97	**35.52**	**0.15**	**1.73**	**34.28**	4.23	8.83	13.92
NMC*	0.385	0.990	0.983	0.146	0.552	83,024	166,038	0.25	2.45	44.67	0.16	2.02	44.58	1.18	3.78	9.49
NMC-lite*	0.381	0.991	0.984	0.119	0.567	50,207	100,404	0.22	2.32	38.51	0.16	2.00	38.33	2.63	7.25	13.60
NDC	0.397	0.989	**0.985**	0.044	0.530	**12,538**	25,100	0.21	2.33	38.56	**0.15**	1.92	38.38	**0.11**	**1.18**	**8.81**
UNDC	**0.362**	**0.992**	**0.985**	**0.038**	**0.574**	12,609	25,258	**0.18**	**2.11**	37.35	0.19	2.10	37.38	0.16	1.27	8.91
UNDC (UDF)	0.365	0.991	0.984	0.045	0.549	12,682	25,293	0.21	2.25	38.57	0.25	2.37	38.72	0.21	1.47	9.14

Table 5.11: Quantitative comparison results on FAUST with SDF and UDF grid input.

64^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	Inference	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
Binary voxel input	($\times 10^5$)			($\times 10^2$)				time	$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	9.327	0.440	0.930	0.546	0.373	42,044	84,088	0.715s	6.24	9.84	28.57	4.56	7.88	26.89	0.08	0.55	2.23
NMC-lite	9.285	0.440	0.929	0.562	0.373	21,457	42,916	0.729s	6.32	9.95	27.99	4.68	8.09	26.38	0.14	1.43	4.38
MC33	26.862	0.085	0.921	11.342	0.018	5,826	11,656	**0.005s**	**4.51**	17.42	40.09	**1.47**	16.07	40.27	**0.00**	**0.00**	**1.21**
NMC*	9.452	0.422	0.927	0.698	0.346	42,045	84,089	0.156s	6.25	10.53	33.14	4.47	8.45	31.44	0.02	0.21	1.52
NMC-lite*	9.428	0.420	0.927	0.604	0.356	21,431	42,862	0.154s	6.35	10.46	31.08	4.58	8.43	29.38	0.11	1.09	4.13
NDC	9.387	**0.428**	0.930	0.567	**0.360**	**5,345**	**10,726**	0.055s	6.14	10.11	**29.21**	4.39	8.00	**27.25**	0.21	0.38	2.52
UNDC	**9.139**	**0.428**	**0.931**	**0.564**	0.359	5,365	10,772	0.055s	6.02	**9.94**	29.51	4.36	**7.98**	27.65	0.21	0.39	2.53
128^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	Inference	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
Binary voxel input	($\times 10^5$)			($\times 10^2$)				time	$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	5.447	0.663	0.959	0.410	0.692	174,257	348,519	5.405s	3.91	5.74	22.16	2.50	4.12	20.76	0.10	0.67	2.19
NMC-lite	5.444	0.663	0.958	0.417	0.693	87,419	174,844	5.449s	3.96	5.81	21.69	2.56	4.25	20.36	0.20	1.69	4.60
MC33	9.800	0.212	0.944	11.690	0.023	22,775	45,557	**0.030s**	**2.71**	12.52	36.74	**1.01**	11.78	37.10	**0.00**	**0.00**	**0.96**
NMC*	5.465	0.659	0.956	0.652	0.664	174,255	348,515	1.129s	3.90	6.30	27.20	2.44	4.64	25.88	0.02	0.27	1.51
NMC-lite*	5.460	0.658	0.957	0.398	0.685	87,369	174,743	1.125s	3.93	6.20	25.23	2.48	4.56	23.87	0.15	1.31	4.40
NDC	**5.451**	**0.661**	0.960	0.316	**0.686**	**21,848**	**43,715**	0.403s	3.85	5.69	**21.01**	2.40	3.97	**19.42**	0.13	0.25	2.55
UNDC	5.458	**0.661**	**0.961**	**0.315**	0.682	21,877	43,757	0.404s	3.80	**5.63**	21.31	2.37	**3.93**	19.73	0.14	0.26	2.54

Table 5.12: Quantitative comparison results on ABC test set with binary occpuancy grid input.

64^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
Binary voxel input	($\times 10^5$)			($\times 10^2$)				$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	6.081	0.491	0.922	0.573	0.342	40,431	80,862	5.42	11.57	41.64	4.63	10.39	40.70	0.15	0.87	3.10
NMC-lite	6.056	0.490	0.920	0.604	0.341	21,635	43,272	5.54	11.71	40.44	4.83	10.71	39.65	0.22	1.91	5.93
MC33	25.523	0.069	0.907	7.542	0.017	5,940	11,882	**3.86**	21.40	52.50	**1.64**	20.06	52.99	**0.00**	**0.00**	2.30
NMC*	6.256	0.471	0.916	0.772	0.306	40,382	80,764	5.49	13.03	46.36	4.53	11.69	45.36	0.03	0.34	**2.13**
NMC-lite*	6.226	0.471	0.917	**0.625**	0.321	21,577	43,154	5.58	12.79	44.13	4.64	11.53	43.18	0.17	1.46	5.51
NDC	6.185	0.477	0.921	0.681	**0.322**	**5,373**	**10,808**	5.29	12.03	**42.19**	4.44	10.80	**41.00**	0.19	0.49	3.78
UNDC	**6.070**	**0.478**	**0.923**	0.651	0.321	5,401	10,855	5.08	**11.78**	42.52	4.36	**10.76**	41.50	0.20	0.51	3.81
128^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
Binary voxel input	($\times 10^5$)			($\times 10^2$)				$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	3.162	0.726	0.957	0.404	0.645	168,218	336,440	2.95	6.08	32.64	2.53	5.37	32.07	0.19	1.06	3.23
NMC-lite	3.163	0.726	0.956	0.414	0.650	88,499	177,003	2.99	6.17	31.55	2.59	5.55	31.07	0.32	2.33	6.33
MC33	8.473	0.169	0.934	7.328	0.026	23,198	46,400	**1.82**	15.34	48.95	**1.09**	15.21	49.98	**0.00**	**0.00**	**2.04**
NMC*	**3.184**	0.721	0.951	0.654	0.604	168,118	336,240	2.95	7.25	38.76	2.46	6.45	38.22	0.04	0.44	2.28
NMC-lite*	3.197	0.721	0.953	0.435	0.638	88,411	176,826	2.97	6.98	36.21	2.49	6.23	35.65	0.24	1.81	6.04
NDC	3.192	**0.724**	0.959	0.427	**0.643**	**22,109**	**44,246**	2.85	5.94	**30.72**	2.39	5.12	**29.90**	0.14	0.37	3.88
UNDC	3.205	**0.724**	**0.960**	**0.369**	0.639	22,157	44,318	2.74	**5.83**	31.13	2.34	**5.09**	30.35	0.14	0.37	3.84

Table 5.13: Quantitative comparison results on Thingi10K with binary occpuancy grid input.

128^3 resolution	CD↓	F1↑	NC↑	ECD↓	EF1↑	#V	#T	% inaccurate normals (gt)			% inaccurate normals (pred)			% small angles		
Binary voxel input	($\times 10^5$)			($\times 10^2$)				$> 80°$	$> 30°$	$> 5°$	$> 80°$	$> 30°$	$> 5°$	$< 10°$	$< 20°$	$< 30°$
NMC	0.760	0.970	0.965	0.328	0.347	82,406	164,806	0.69	5.47	70.31	0.29	4.29	70.06	0.34	2.07	6.55
NMC-lite	0.754	0.970	0.964	0.296	0.334	49,699	99,393	0.70	5.79	69.04	0.32	4.79	68.87	0.48	4.09	11.54
MC33	6.928	0.064	0.905	0.455	0.085	14,175	28,348	**0.52**	25.51	95.40	**0.10**	22.46	95.46	**0.00**	**0.00**	5.23
NMC*	0.816	0.967	0.944	0.760	0.160	82,337	164,668	1.02	9.46	79.88	0.55	8.49	80.00	0.07	0.82	**4.44**
NMC-lite*	0.801	0.967	0.953	0.345	0.311	49,643	99,279	0.75	8.27	75.80	0.29	7.26	75.75	0.39	3.21	10.85
NDC	0.766	**0.969**	**0.966**	**0.169**	0.330	**12,411**	**24,833**	0.66	5.45	**67.71**	0.31	**4.23**	**67.38**	0.03	0.63	8.07
UNDC	**0.760**	**0.969**	**0.966**	0.177	**0.353**	12,467	24,930	0.59	**5.30**	68.22	0.33	4.39	68.01	0.05	0.62	8.16

Table 5.14: Quantitative comparison results on FAUST with binary occpuancy grid input.

point cloud (4,096)	CD↓ ($\times 10^5$)	F1↑	NC↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T	Inference time
Ball-pivoting (+n)	3.080	0.791	0.944	0.556	0.269	**4,096**	**7,439**	1.292s
Poisson (+n)	4.705	0.727	0.939	4.138	0.067	11,241	22,496	1.476s
SIREN (+n)	1.340	0.814	0.969	2.636	0.152	97,219	194,543	168.595s
LIG (P) (+n)	4.747	0.709	0.939	10.786	0.023	148,927	297,766	61.176s
LIG (+n)	3.413	0.721	0.947	11.868	0.022	149,860	299,166	66.866s
ConvONet (P)	38.926	0.207	0.844	1.522	0.057	127,247	254,627	4.598s
ConvONet 3plane	18.073	0.536	0.935	4.113	0.105	75,342	150,689	2.692s
ConvONet grid	8.844	0.488	0.939	9.701	0.036	74,171	148,337	2.404s
UNDC @ 64^3	**0.893**	**0.873**	**0.974**	**0.289**	**0.757**	5,578	11,261	**0.194s**

Table 5.15: Quantitative results on ABC test set with point cloud input. (+n) indicates that the method additionally requires point normals as input. UNDC @ 64^3 means that the output grid size of UNDC is 64^3.

point cloud (4,096)	CD↓ ($\times 10^5$)	F1↑	NC↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T
Ball-pivoting (+n)	2.329	0.787	0.936	0.602	0.236	**4,096**	**7,455**
Poisson (+n)	12.799	0.744	0.938	3.439	0.052	11,498	23,010
SIREN (+n)	1.419	0.834	0.962	2.059	0.144	94,797	189,637
LIG (P) (+n)	4.453	0.691	0.929	8.471	0.019	146,554	292,847
LIG (+n)	5.991	0.748	0.943	8.266	0.021	145,269	290,354
ConvONet (P)	39.822	0.209	0.826	1.306	0.051	120,475	241,068
ConvONet 3plane	18.272	0.484	0.903	3.222	0.090	74,514	149,033
ConvONet grid	6.032	0.476	0.928	8.249	0.024	73,745	147,484
UNDC @ 64^3	**0.927**	**0.873**	**0.965**	**0.400**	**0.686**	5,543	11,159

Table 5.16: Quantitative results on Thingi10K with point cloud input. (+n) indicates that the method additionally requires point normals as input. UNDC @ 64^3 means that the output grid size of UNDC is 64^3.

point cloud (4,096)	CD↓ ($\times 10^5$)	F1↑	NC↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T
Ball-pivoting (+n)	0.906	0.932	0.965	0.372	0.131	4,096	7,652
Poisson (+n)	0.724	0.966	0.975	0.467	0.246	11,330	22,646
SIREN (+n)	0.697	0.951	**0.986**	0.211	**0.504**	51,215	102,443
LIG (P) (+n)	1.449	0.876	0.964	1.307	0.107	79,337	158,605
LIG (+n)	2.533	0.871	0.962	1.772	0.077	80,845	161,226
ConvONet (P)	17.334	0.312	0.849	1.244	0.027	47,716	95,424
ConvONet 3plane	23.809	0.389	0.868	1.046	0.053	46,211	92,427
ConvONet grid	3.506	0.574	0.945	4.618	0.029	41,710	83,418
UNDC @ 64^3	0.532	0.970	0.965	0.345	0.206	**3,146**	**6,308**
UNDC @ 128^3	**0.413**	**0.985**	0.978	**0.095**	0.437	12,681	25,202
point cloud (16,384)	CD↓ ($\times 10^5$)	F1↑	NC↑	ECD↓ ($\times 10^2$)	EF1↑	#V	#T
Ball-pivoting (+n)	0.545	0.977	0.977	0.144	0.316	16,384	31,767
Poisson (+n)	0.397	0.987	0.987	0.118	0.528	45,325	90,630
SIREN (+n)	0.707	0.953	0.988	0.263	0.562	51,132	102,270
LIG (P) (+n)	1.140	0.902	0.969	1.170	0.160	78,821	157,622
LIG (+n)	2.215	0.895	0.966	1.792	0.120	80,399	160,445
ConvONet (P)	20.218	0.250	0.865	1.345	0.024	51,449	102,873
ConvONet 3plane	24.682	0.390	0.869	0.985	0.048	47,922	95,851
ConvONet grid	3.563	0.568	0.947	5.474	0.029	42,218	84,433
UNDC @ 128^3	0.368	0.991	0.983	0.050	0.566	**12,665**	**25,387**
UNDC @ 256^3	**0.353**	**0.993**	**0.989**	**0.020**	**0.767**	51,043	101,733

Table 5.17: Quantitative results on FAUST with point cloud input. (+n) indicates that the method additionally requires point normals as input. UNDC @ 64^3 means that the output grid size of UNDC is 64^3.

Chapter 6

MobileNeRF: Exploiting the Polygon Rasterization Pipeline for Efficient Neural Field Rendering on Mobile Architectures

6.1 Introduction

Neural Radiance Fields (NeRF) [167] have become a popular representation for novel view synthesis of 3D scenes. They represent a scene using a multilayer perceptron (MLP) that evaluates a 5D implicit function estimating the density and radiance emanating from any position in any direction, which can be used in a volumetric rendering framework to produce novel images. NeRF representations optimized to minimize multi-view color consistency losses for a set of posed photographs have demonstrated remarkable ability to reproduce fine image details for novel views.

One of the main impediments to wide-spread adoption of NeRF is that it requires specialized rendering algorithms that are poor match for commonly available hardware. Traditional NeRF implementations use a volumetric rendering algorithm that evaluates a large MLP at hundreds of sample positions along the ray for each pixel in order to estimate and integrate density and radiance. This rendering process is far too slow for interactive visualization.

Recent work has addressed this issue by "baking" NeRFs into a sparse 3D voxel grid [97, 288]. For example, Hedman et al. introduced Sparse Neural Radiance Grids (SNeRG) [97], where each active voxel contains an opacity, diffuse color, and learned feature vector. Rendering an image from SNeRG is split into two phases: the first uses ray marching to accumulate the precomputed diffuse colors and feature vectors along each ray, and the second uses a light-weight MLP operating on the accumulated feature vector to produce a view-dependent residual that is added to the accumulated diffuse color. This precomputation and deferred rendering approach increase the rendering speed of NeRF by three orders of

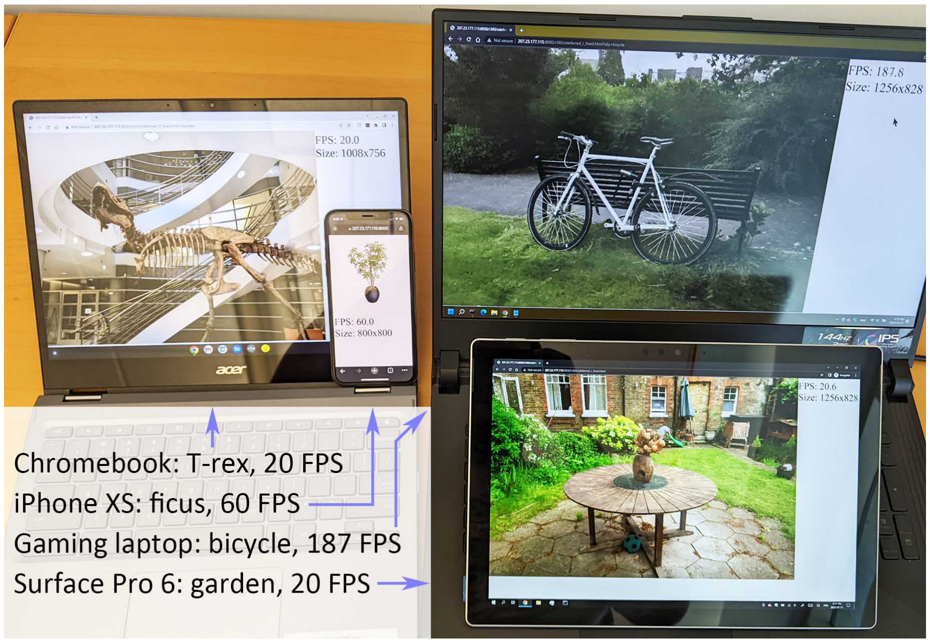


Figure 6.1: **Teaser** – We present a NeRF that can run on a variety of common devices at interactive frame rates.

magnitude. However, it still relies upon ray marching through a sparse voxel grid to produce the features for each pixel, and thus it cannot fully utilize the parallelism available in commodity graphics processing units (GPUs). In addition, SNeRG requires a significant amount of GPU memory to store the volumetric textures, which prohibits it from running on common mobile devices.

In this paper, we introduce MobileNeRF, a NeRF that can run on a variety of common mobile devices at interactive frame rates. The NeRF is represented by a set of textured polygons, where the polygons roughly follow the surface of the scene, and the texture atlas stores opacity and feature vectors. To render an image, we utilize the classic polygon rasterization pipeline with Z-buffering to produce a feature vector for each pixel and pass it to a lightweight MLP running in a GLSL fragment shader to produce the output color. This rendering pipeline *does not* sample rays or sort polygons in depth order, and thus can model only binary opacities. However, it takes full advantage of the parallelism provided by z-buffers and fragment shaders in modern graphics hardware, and thus is 10$\times$ faster than SNeRG with the same output quality on standard test scenes. Moreover, it requires only a standard polygon rendering pipeline, which is implemented and accelerated on virtually every computing platform, and thus it runs on mobile phones and other devices previously unable to support NeRF visualization at interactive rates.

Contributions In summary, MobileNeRF:

- Is 10× *faster* than the state-of-the-art (SNeRG), with the same output quality;
- Consumes less memory by storing *surface* textures instead of volumetric textures, enabling our method to run on integrated GPUs with limited memory and power;
- Runs on a web browser and is *compatible* with all devices we have tested, as our viewer is an HTML webpage;
- Allows real-time *manipulation* of the reconstructed objects/scenes, as they are simple triangle meshes.

6.2 Related work

Our work lies within the field of view-synthesis, which encompasses many areas of research: light fields, image-based rendering and neural rendering. To narrow the scope, we focus on methods that render output views in *real-time*.

Light fields [132] and Lumigraphs [83] store a dense grid of images, enabling real-time rendering of high quality scenes, albeit with limited camera freedom and significant storage overhead. Storage can be reduced by interpolating intermediate images with optical flow [16], representing the light field as a neural network [3], or by reconstructing a Multi-Plane Image (MPI) representation of the scene [194, 303, 166, 66, 262]. Multi-sphere images enable larger fields of view [22, 4], but these representations still only support limited output camera motion

Other approaches leverage explicit 3D geometry to enable more camera freedom. While early methods applied view-dependent texturing to a 3D mesh [55, 23, 53], later methods incorporated convolutional neural networks as a post-processing step to improve quality [160, 96, 238]. Alternatively, the input geometry can be simplified into a collection of textured planes with alpha [144]. Point-based representations further increase quality by jointly refining the scene geometry while training the post-processing network [127, 208, 125]. However, as this convolutional post-processing runs independently per output frame it often results in a lack of 3D consistency. Furthermore, unlike our work, they require powerful desktop GPUs and have not been demonstrated to run on a mobile device. Finally, unlike the vast majority of the methods above, our method does not need reconstructed 3D geometry as input.

It is also possible to extract explicit triangle meshes via differentiable inverse-rendering [71, 173, 46]. DefTet [71] differentiably renders a tetrahedral grid with occupancy and color at each vertex, and then compositing the interpolated values at all intersected faces along a ray. NVDiffRec [173] combines differentiable marching tetrahedra [222] with differentiable rasterization to perform full inverse rendering and extract triangle meshes, materials, and lighting from images. This representation enables elaborate editing and scene relighting. However, it incurs a significant loss in view-synthesis quality. Furthermore, while real-time

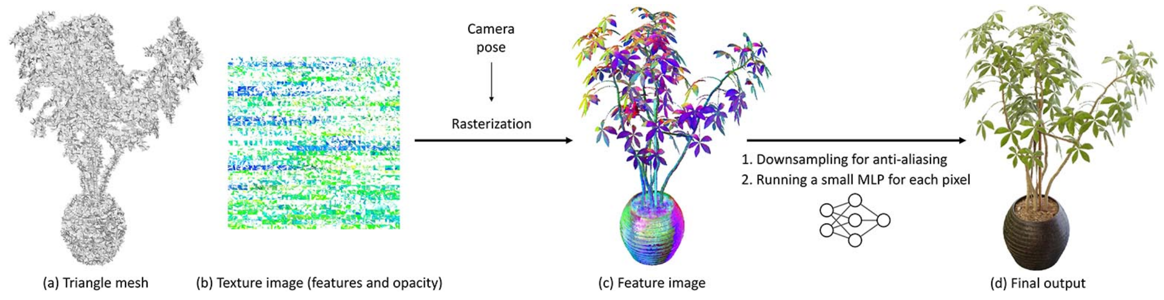

Camera pose
Rasterization
1. Downsampling for anti-aliasing
2. Running a small MLP for each pixel
(a) Triangle mesh
(b) Texture image (features and opacity)
(c) Feature image
(d) Final output

Figure 6.2: **Overview (rendering)** – We represent the scene as a triangle mesh textured by deep features. We first rasterize the mesh to a deferred rendering buffer. For each visible fragment, we execute a neural deferred shader that converts the feature and view direction to the corresponding output pixel color.

rendering is possible with simple lighting, global illumination (GI) is computationally infeasible on mobile hardware. In contrast, our method simply caches the outgoing radiance, which does not need expensive compute to model GI effects, and also results in higher view-synthesis quality.

NeRF [167] represents the scene as a continuous field of opacity and view-dependent color, and produces images with volume rendering. This representation is 3D consistent and reaches high quality results [247, 11]. However, rendering a NeRF involves evaluating a large neural network at multiple 3D locations per pixel, preventing real-time rendering.

Recent works have improved the training speed of NeRF. For example, by modeling the opacity and color of entire ray segments instead of just points [145] or by subdividing the scene and modeling each sub-region with a smaller neural network [199]. Recently, significant speed-ups have been achieved by decoding features fetched from a 3D embedding with a small neural network. This embedding can either be a dense voxel grid [230, 114], a sparse voxel grid [212], a low-rank decomposition of a voxel grid [29], a point-based representation [277], or a multi-resolution hash map [172]. These 3D embeddings can also be used without a trained decoder, for example by directly storing diffuse colors [154] or by encoding view-dependent colors as spherical harmonics [212]. While these approaches drastically speed up training, they still require a large consumer GPU for rendering.

Rendering performance can further be increased by *post-processing* a trained NeRF. For example, by reducing the network queries per pixel with learned sampling [175], by evaluating the network for larger ray segments [266], or by subdividing the scene into smaller networks [200, 270, 199]. Alternatively, pre-computation can speed up rendering, by storing both scene opacity and a latent representation for view-dependent colors in a grid. FastNeRF [75] uses a dense voxel grid and represents view-dependence with a global spherical basis function. PlenOctrees [288] uses an octree representation, where each leaf node stores both opacity and spherical harmonics for colors. SNeRG [97] uses a sparse grid rep-

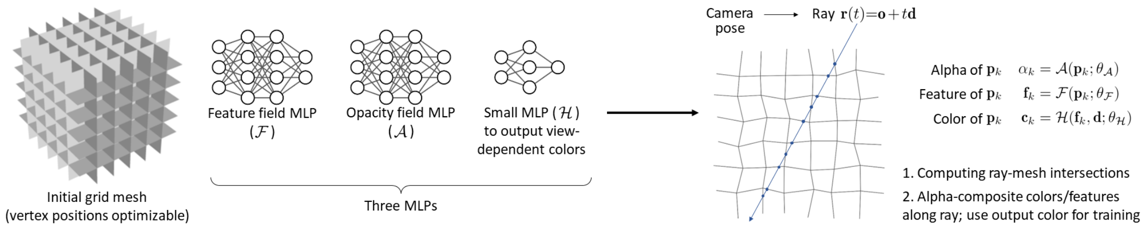

Initial grid mesh
(vertex positions optimizable)
Feature field MLP
($\mathcal{F}$)
Opacity field MLP
($\mathcal{A}$)
Small MLP ($\mathcal{H}$)
to output view-dependent colors
Three MLPs
Camera pose
Ray $\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$
Alpha of $\mathbf{p}_k$ $\alpha_k = \mathcal{A}(\mathbf{p}_k;\theta_{\mathcal{A}})$
Feature of $\mathbf{p}_k$ $\mathbf{f}_k = \mathcal{F}(\mathbf{p}_k;\theta_{\mathcal{F}})$
Color of $\mathbf{p}_k$ $\mathbf{c}_k = \mathcal{H}(\mathbf{f}_k,\mathbf{d};\theta_{\mathcal{H}})$
1. Computing ray-mesh intersections
2. Alpha-composite colors/features along ray; use output color for training

Figure 6.3: **Overview (train)** – We initialize the mesh as a regular grid, and use MLPs to represent features and opacity for any point on the mesh. For each ray, we compute its intersection points on the mesh, and alpha-composite the colors of those points to obtain the output color. In a later training stage, we enforce *binary* opacity, and perform super-sampling on features for anti-aliasing.

resentation, and evaluates view-dependence as a post-process with a small neural network. Among these real-time methods, only SNeRG has been shown to work on lower-powered devices without access to CUDA. As our method directly targets rendering on low-powered hardware, we primarily compare with SNeRG in our experiments.

6.3 Method

Given a collection of (calibrated) images, we seek to optimize a representation for *efficient* novel-view synthesis. Our representation consists of a polygonal mesh (Figure 6.2a) whose texture maps (Figure 6.2b) store features and opacity. At rendering time, given a camera pose, we adopt a two-stage *deferred rendering* process:

- **Rendering Stage 1** – we rasterize the mesh to screen space and construct a *feature image* (Figure 6.2c), i.e. we create a deferred rendering buffer in GPU memory;
- **Rendering Stage 2** – we convert these features into a color image via a (neural) deferred renderer running in a fragment shader, i.e. a small MLP, which receives a feature vector and view direction and outputs a pixel color (Figure 6.2d).

Our representation is built in three *training* stages, gradually moving from a classical NeRF-like continuous representation towards a discrete one:

- **Training Stage 1 (Section 6.3.1)** – We train a NeRF-like model with *continuous* opacity, where volume rendering quadrature points are derived from the polygonal mesh;
- **Training Stage 2 (Section 6.3.2)** – We *binarize* the opacities, as while classical rasterization can easily discard fragments, they cannot elegantly deal with semi-transparent fragments.

- **Training Stage 3 (Section 6.3.3)** – We *extract* a sparse polygonal mesh, bake opacities and features into texture maps, and store the weights of the neural deferred shader.

The mesh is stored as an OBJ file, the texture maps in PNGs, and the deferred shader weights in a (small) JSON file. As we employ the standard GPU rasterization pipeline, our real-time renderer is simply an HTML webpage.

As representing continuous signals with discrete representations can introduce aliasing, we also detail a simple, yet computationally efficient, anti-aliasing solution based on super-sampling (Section 6.3.4).

6.3.1 Continuous training (Training Stage 1)

As Figure 6.3 shows, our *training setup* consists of a polygonal mesh $\mathcal{M}{=}(\mathcal{T}, \mathcal{V})$ and three MLPs. The mesh topology $\mathcal{T}$ is fixed, but the vertex locations $\mathcal{V}$ and MLPs are optimized, similarly to NeRF, in an auto-decoding fashion by minimizing the mean squared error between predicted colors and ground truth colors of the pixels in the training images:

$$\mathcal{L}_{\mathbf{C}} = \mathbb{E}_{\mathbf{r}} \|\mathbf{C}(\mathbf{r}) - \mathbf{C}_{\text{gt}}(\mathbf{r})\|_2^2. \tag{6.1}$$

where the predicted color $\mathbf{C}(.)$ is obtained by alpha-compositing the radiance $\mathbf{c}_k$ along a ray $\mathbf{r}(t){=}\mathbf{o} + t\mathbf{d}$, at the (depth sorted) quadrature points $\mathcal{K}{=}\{t_k\}_{k=1}^{K}$:

$$\mathbf{C}(\mathbf{r}) = \sum_{k=1}^{K} T_k \alpha_k \mathbf{c}_k, \quad T_k = \prod_{l=1}^{k-1}(1 - \alpha_l) \tag{6.2}$$

where *opacity* α_k and the view-dependent *radiance* $\mathbf{c}_k$ are given by evaluating the MLPs at position $\mathbf{p}_k{=}\mathbf{r}(t_k)$:

$$\alpha_k = \mathcal{A}(\mathbf{p}_k; \theta_{\mathcal{A}}) \qquad \mathcal{A} : \mathbb{R}^3 \to [0, 1] \tag{6.3}$$

$$\mathbf{f}_k = \mathcal{F}(\mathbf{p}_k; \theta_{\mathcal{F}}) \qquad \mathcal{F} : \mathbb{R}^3 \to [0, 1]^8 \tag{6.4}$$

$$\mathbf{c}_k = \mathcal{H}(\mathbf{f}_k, \mathbf{d}; \theta_{\mathcal{H}}) \qquad \mathcal{H} : [0, 1]^8 \times [-1, 1]^3 \to [0, 1]^3 \tag{6.5}$$

The small network $\mathcal{H}$ is our *deferred neural shader*, which outputs the color of each fragment given the fragment feature and viewing direction. Finally, note that (6.2) does not perform compositing with volumetric density [167], but rather with opacity [3, Eq.8]. For real-world scenes, we further incorporate the distortion loss $\mathcal{L}_{dist}$ introduced by [11, Eq. 15] to suppress *floaters* and *background collapse.*

Polygonal mesh Without loss of generality, we describe the polygonal mesh used in *Synthetic* 360° scenes, and provide the configurations for *Forward-Facing* and *Unbounded* 360° scenes in Section 6.3.7. 2D illustrations can be found in Figure 6.4. We first define a *regular* grid $\mathcal{G}$ of size $P{\times}P{\times}P$ in the unit cube centered at the origin; see Figure 6.4a. We instantiate $\mathcal{V}$ by creating one vertex per voxel, and $\mathcal{T}$ by creating one quadrangle (two triangles) per grid edge connecting the vertices of the four adjacent voxels, akin to Dual Contouring [110, 35]. We locally parameterize vertex locations with respect to the voxel centers (and sizes), resulting in $\mathcal{V}{\in}[-.5,+.5]^{P\times P\times P\times 3}$ free variables. During optimization, we initialize the vertex locations to $\mathcal{V}{=}\mathbf{0}$, which corresponds to a regular Euclidean lattice, and we regularize them to prevent vertices from exiting their voxels, and to promote their return to their neutral position whenever the optimization problem is under-constrained:

$$\mathcal{L}_{\mathcal{V}} = \sum_{\mathbf{v}\in\mathcal{V}} (10^3\, \mathcal{I}(\mathbf{v}) + 10^{-2}) \cdot ||\mathbf{v}||_1, \tag{6.6}$$

where the indicator function $\mathcal{I}(\mathbf{v}){\equiv}1$ whenever $\mathbf{v}$ is outside its corresponding voxel.

Quadrature As evaluating the MLPs of our representation is computationally expensive, we rely on an acceleration grid to limit the cardinality $|\mathcal{K}|$ of quadrature points. First of all, quadrature points are only generated for the set of voxels that intersect the ray; see Figure 6.5a: Then, like InstantNGP [172], we employ an acceleration grid $\mathcal{G}$ to prune voxels that are unlikely to contain geometry; see Figure 6.5b. Finally, we compute intersections between the ray and the faces of $\mathcal{M}$ that are incident to the voxel's vertex to obtain the final set of quadrature points; see Figure 6.5c. We use the barycentric interpolation to back-propagate the gradients from the intersection point to the three vertices in the intersected triangle. For further technical details on the computation of intersections, we refer the reader to Section 6.3.6. In summary, for each input ray $\mathbf{r}$:

$$\tilde{\mathcal{B}} = \text{intersect}(\mathbf{r}, \mathcal{G}) \tag{6.7}$$

$$\mathcal{B} = \{b \in \tilde{\mathcal{B}} \mid \mathcal{G}[b] > \tau_{\mathcal{G}}\} \tag{6.8}$$

$$\mathcal{K} = \text{intersect}(\mathbf{r}, \{\mathbf{t} \in \mathcal{T} \mid \mathbf{t} \cap \mathcal{B}\}) \tag{6.9}$$

where $(a{\cap}b){=}\mathit{true}$ if a intersects b, and the acceleration grid is supervised so to upper-bound the alpha-compositing visibility $T_k\alpha_k$ *across* viewpoints during training.

$$\mathcal{L}_{\mathcal{G}}^{\text{bnd}} = \sum_k \max(\cancel{\nabla}[T_k\alpha_k] - \mathcal{G}[\mathbf{p}_k], 0) \tag{6.10}$$

where $\cancel{\nabla}[.]$ is the stop-gradient operator that prevents the acceleration grid from (negatively) affecting the image reconstruction quality. This loss performs a *stochastic upper-bound*, as we initialize $\mathcal{G}[*]{=}\mathbf{0}$, and $\mathcal{G}[\mathbf{p}_k]$ receives gradients whenever $T_k\alpha_k{>}\mathcal{G}[\mathbf{p}_k]$. Similarly to Plenoxels [212], we additionally regularize the content of the grid by promoting its pointwise sparsity (i.e. lasso), and its spatial smoothness:

$$\mathcal{L}_{\mathcal{G}}^{\text{sparse}} = \|\mathcal{G}\|_1^1 \qquad \mathcal{L}_{\mathcal{G}}^{\text{smooth}} = \|\nabla\mathcal{G}\|_2^2 \tag{6.11}$$

6.3.2 Binarized training (Training Stage 2)

Rendering pipelines implemented in typical hardware *do not* natively support semi-transparent meshes. Rendering semi-transparent meshes requires cumbersome (per-frame) sorting so to execute rendering in back-to-front order to guarantee correct alpha-compositing. We overcome this issue by converting the smooth opacity $\alpha_k \in [0,1]$ from (6.3) to a discrete/categorical opacity $\hat{\alpha}_k \in \{0,1\}$. To optimize for discrete opacities via photometric supervision we employ a *straight-through estimator* [13]:

$$\hat{\alpha}_k = \alpha_k + \cancel{\nabla}[\mathbb{1}(\alpha_k > 0.5) - \alpha_k] \tag{6.12}$$

Please note that the gradients are transparently passed through the discretization operation (i.e. $\nabla\hat{\alpha} \equiv \nabla\alpha$), regardless of the values of α_k and the resulting $\hat{\alpha}_k \in \{0,1\}$. To stabilize training, we then co-train the continuous and discrete models:

$$\mathcal{L}_{\mathbf{C}}^{\text{bin}} = \mathbb{E}_{\mathbf{r}} \|\hat{\mathbf{C}}(\mathbf{r}) - \mathbf{C}_{\text{gt}}(\mathbf{r})\|_2^2 \tag{6.13}$$

$$\mathcal{L}_{\mathbf{C}}^{\text{stage2}} = \tfrac{1}{2}\mathcal{L}_{\mathbf{C}}^{\text{bin}} \; + \tfrac{1}{2}\mathcal{L}_{\mathbf{C}} \tag{6.14}$$

where $\hat{\mathbf{C}}(\mathbf{r})$ is the output radiance corresponding to the discrete opacity model $\hat{\alpha}$:

$$\hat{\mathbf{C}}(\mathbf{r}) = \sum_{k=1}^{K} \hat{T}_k \hat{\alpha}_k \mathbf{c}_k, \quad \hat{T}_k = \prod_{l=1}^{k-1}(1 - \hat{\alpha}_l) \tag{6.15}$$

Once (6.14) has converged, we will apply a fine-tuning step to the weights in $\mathcal{F}$ and $\mathcal{H}$ by minimizing $\mathcal{L}_{\mathbf{C}}^{\text{bin}}$, while fixing the weights of others.

6.3.3 Discretization (Training Stage 3)

After binarization and fine-tuning, we convert the representation into an explicit polygonal mesh (in OBJ format). We only store quads if they are at least partially visible in the training camera poses (i.e. non-visible quads are discarded). We then create a texture image whose size is proportional to the number of visible quads, and for each quad we allocate a $K \times K$ patch in the texture, similarly to Disney's Ptex [24]. We use K=17 in our experiments, so that the quad has a 16×16 texture with half-a-pixel boundary padding. We then iterate over the pixels of the texture, convert the pixel coordinate to 3D coordinates, and *bake* the values of the discrete opacity (i.e. (6.3) and (6.12)) and features (i.e. (6.4)) into the texture map. We quantize the $[0,1]$ ranges to 8-bit integers, and store the texture into (losslessly compressed) PNG images. Our experiments show that quantizing the $[0,1]$ range with 8-bit

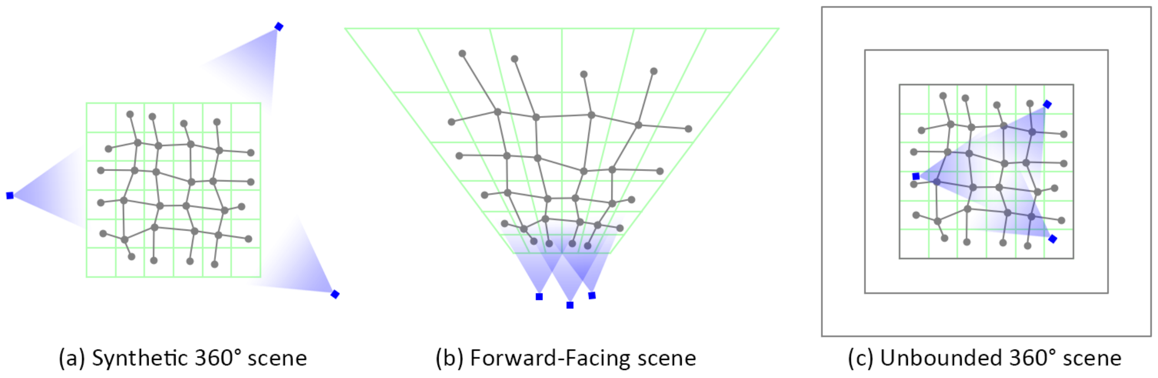


Figure 6.4: **Configurations of polygonal meshes** – The meshes employed for the different types of scenes. We sketch the distribution of camera poses in training views.

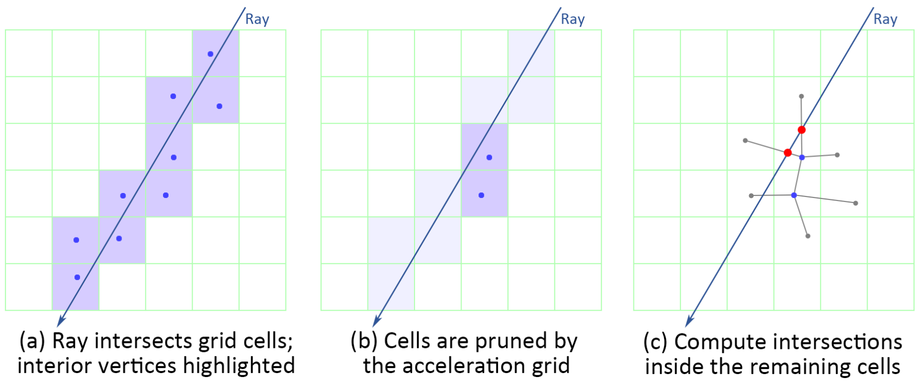

Ray
Ray
Ray
(a) Ray intersects grid cells; interior vertices highlighted
(b) Cells are pruned by the acceleration grid
(c) Compute intersections inside the remaining cells

Figure 6.5: **Quadrature points** – are obtained by (a) identifying cells that intersect the ray; (b) pruning cells that do not contain geometry; and, (c) computing explicit intersections with the mesh.

precision, which is not accounted for during back-propagation, does not significantly affect rendering quality.

6.3.4 Anti-aliasing

In classic rasterization pipelines, aliasing is an issue that ought to be considered to obtain high-quality rendering. While classical NeRF hallucinates smooth edges via semi-transparent volumes, as previously discussed, semi-transparency would require per-frame polygon sorting. We overcome this issue by employing anti-aliasing by super-sampling. While we could simply execute (6.5) four times/pixel and average the resulting color, the execution of the deferred neural shader $\mathcal{H}$ is the computational bottleneck of our technique. We can overcome this issue by simply averaging the features, that is, *averaging the input* of the deferred neural shader, rather than averaging its output. We first rasterize features (at

2× resolution):

$$\mathbf{F}(\mathbf{r}) = \sum_k T_k \alpha_k \mathbf{f}_k, \tag{6.16}$$

and then average sub-pixel features to produce the anti-aliased representation we feed to our neural deferred shader:

$$\mathbf{C}(\mathbf{r}) = \mathcal{H}\left(\mathbb{E}_{\mathbf{r}_\delta \sim \mathbf{r}}[\mathbf{F}(\mathbf{r}_\delta)],\ \mathbb{E}_{\mathbf{r}_\delta \sim \mathbf{r}}[\mathbf{d}_\delta]\right) \tag{6.17}$$

where $\mathbb{E}_{\mathbf{r}_\delta \sim \mathbf{r}}$ computes the average between the sub-pixels (i.e. four in our implementation), and $\mathbf{d}_\delta$ is the direction of ray $\mathbf{r}_\delta$. Note how with this change we only query $\mathcal{H}$ *once* per output pixel. Finally, this process is analogously applied to (6.15) for discrete occupancies $\hat{\alpha}$. These changes for anti-aliasing are applied in training stage 2 (6.14).

6.3.5 Rendering

The result of the optimization process is a textured polygonal mesh (where texture maps store features rather than colors) and a small MLP (which converts view direction and features to colors). Rendering this representation is done in two passes using a deferred rendering pipeline:

1. we rasterize all faces of the textured mesh with a z-buffer to produce a $2M \times 2N$ feature image with 12 channels per pixel, comprising 8 channels of learned features, a binary opacity, and a 3D view direction;

2. we synthesize an $M \times N$ output RGB image by rendering a textured rectangle that uses the feature image as its texture, with linear filtering to average the features for antialiasing. We apply the small MLP for pixels with non-zero alphas to convert features into RGB colors. The small MLP is implemented as a GLSL fragment shader.

These rendering steps are implemented within the classic rasterization pipeline. Since z-buffering with binary transparency is order-independent, polygons *do not* need to be sorted into depth-order for each new view, and thus can be loaded into a buffer in the GPU once at the start of execution. Since the MLP for converting features to colors is very small, it can be implemented in a GLSL fragment shader [97], which is run in parallel for all pixels. These classical rendering steps are highly-optimized on GPUs, and thus our rendering system can run at interactive frame rates on a wide variety of devices; see Table 6.2. It is also easy to implement, since it requires only standard polygon rendering with a fragment shader. Our interactive viewer is an HTML webpage with Javascript, rendered by WebGL via the `threejs` library.

6.3.6 Quadrature details

The regular-grid mesh $\mathcal{M}$ provides an efficient way for computing intersections between a ray and the mesh of size $P \times P \times P$ in $O(P)$ complexity, as shown in Figure 6.5.

First, we compute the set of voxels that are intersected by the ray. This involves solving $3P$ ray-plane intersections and using those intersection points to obtain at most $3P$ intersected voxels. This step is shown in Figure 6.5a and Eq. 6.7.

Then, we use the acceleration grid $\mathcal{G} \in \mathbb{R}^{P \times P \times P}$ to prune voxels that are unlikely to contain geometry, with respect to a threshold $\tau_{\mathcal{G}} = 0.1$. This step is shown in Figure 6.5b and Eq. 6.8.

Finally, we compute intersections between the ray and the faces of $\mathcal{M}$ that are incident to the voxel's vertex to obtain the final set of quadrature points. This step is shown in Figure 6.5c and Eq. 6.9.

During the first quarter of the training iterations, $\mathcal{G}$ may not be accurate, therefore we will keep all $3P$ intersected voxels regardless of $\tau_{\mathcal{G}}$, and keep $3P$ intersection points (Recall that if the mesh grid is a regular grid, there are at most $3P$ intersections). Then in the next quarter, we will use $\mathcal{G}$ to remove empty voxels and keep at most $3P/2$ non-empty voxels and $3P/2$ intersection points that are closest to the camera. In the rest of the training, we will keep $3P/4$. We also double the training batch size each time we halve the number of intersections.

For the concentric boxes in unbounded 360° scenes, we will compute their intersections and keep all of them.

6.3.7 Initial meshes

In this section we detail the polygonal meshes used for synthetic 360°, forward-facing, and unbounded 360° scenes, see Fig. 6.4 for 2D illustrations.

We will call the coordinate system of a regular mesh grid in a unit cube centered at the origin as the normalized coordinates, and we can apply transformations to obtain the grids in the world coordinates for different types of scenes. In the following, we will denote points in the normalized coordinates as $\mathbf{p} \in [-0.5, 0.5]$ and points in the world coordinates as $\mathbf{p}'$.

For synthetic 360° scenes, we apply scaling to the grid to put the object inside the grid.

$$\mathbf{p}' = w\mathbf{p}, \tag{6.18}$$

where $w = 2.4$ or 3, depending on the size of the object. We use a grid size of $P = 128$.

In forward-facing scenes, we apply transformation to concentrate more voxels close to the camera, as shown in Fig. 6.4 (b).

$$\begin{cases} \mathbf{p}'_z &= \exp(w(\mathbf{p}_z + 0.5)), \\ \mathbf{p}'_x &= u\mathbf{p}_x\mathbf{p}'_z, \\ \mathbf{p}'_y &= v\mathbf{p}_y\mathbf{p}'_z, \end{cases} \tag{6.19}$$

where w is set to a value so that $\mathbf{p}'_z = 25$ when $\mathbf{p}_z = 0.5$; $u = v = 1.75$. We use a grid size of $P = 128$.

In unbounded 360° scenes, we assume the cameras are inside the unit cube in the normalized coordinates, therefore we do not apply transformations. However, to model the surrounding environments, we add a set of $L + 1$ concentric boxes around the regular grid. The boxes have fixed positions and geometry, and their distances to the center are given by

$$d_i = (\exp(\frac{wi}{L}) + w - 1)/2w, \tag{6.20}$$

where i ranges from 0 to L. w is set to a value so that $d_L = 8$, therefore $d_i \in [0.5, 8]$. We use a grid size of $P = 128$, and $L = 64$.

6.3.8 Network and Training details

Training Our training stages are formalized as follows. In the first training stage, we optimize

$$\underset{\mathcal{V},\theta_{\mathcal{A}},\theta_{\mathcal{F}},\theta_{\mathcal{H}}}{\arg\min} \quad \mathcal{L}_{\mathbf{C}} + w_d\mathcal{L}_{\text{dist}} + \mathcal{L}_{\text{v}} \tag{6.21}$$

and

$$\underset{\mathcal{G}}{\arg\min} \quad \mathcal{L}_{\mathcal{G}}^{\text{bnd}} + w_{g1}\mathcal{L}_{\mathcal{G}}^{\text{sparse}} + w_{g2}\mathcal{L}_{\mathcal{G}}^{\text{smooth}}, \tag{6.22}$$

where $w_{g1} = w_{g2} = 10^{-5}$. w_d is set to 0.0 for synthetic 360° scenes, 0.01 for forward-facing scenes, and 0.001 for unbounded 360° scenes. In the second training stage, we optimize

$$\underset{\mathcal{V},\theta_{\mathcal{A}},\theta_{\mathcal{F}},\theta_{\mathcal{H}}}{\arg\min} \quad \mathcal{L}_{\mathbf{C}}^{\text{stage2}} + w_d\mathcal{L}_{\text{dist}} + \mathcal{L}_{\text{v}} \tag{6.23}$$

and Eq. 6.22. When the loss converges, we fix the weights of $\mathcal{V}$, $\theta_{\mathcal{A}}$, and $\mathcal{G}$ and optimize

$$\underset{\theta_{\mathcal{F}},\theta_{\mathcal{H}}}{\arg\min} \quad \mathcal{L}_{\mathbf{C}}^{\text{bin}}. \tag{6.24}$$

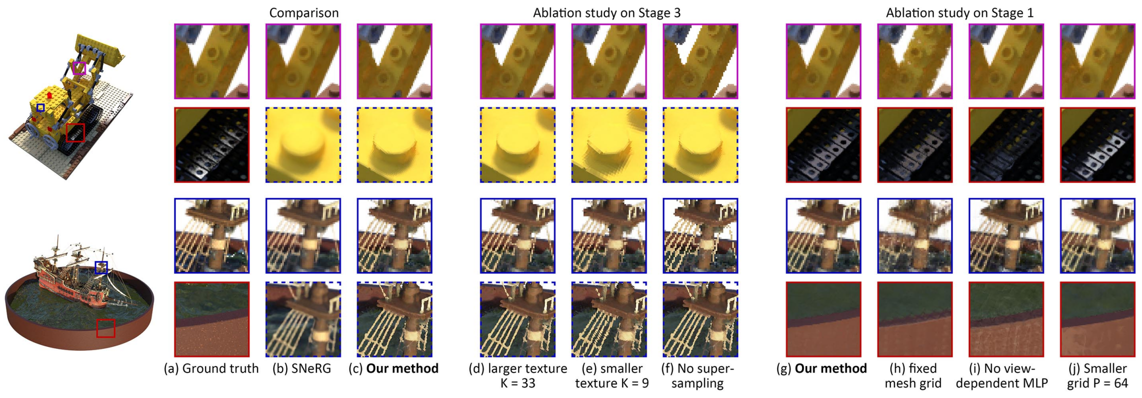


Figure 6.6: **Qualitative Results** – Comparisons to the state-of-the-art and ablation studies. With a *solid* line we denote zoom-ins of the rendered (800×800) image, while with a *dashed* line we move the camera to zoom-in onto the same detail.

Network architectures We adopt the MLP designed in NeRF as the network for both $\mathcal{A}$ and $\mathcal{F}$. We increase the hidden layer sizes from 256 to 384, since $\mathcal{A}$ and $\mathcal{F}$ are not used during inference, so we can afford more time on training. The small MLP $\mathcal{H}$ is the same as the small MLP used in SNeRG, with two hidden layers, each consisting 16 neurons.

Texture images Since the features to be stored are 8-dimensional, we use two PNG images to store them. Each PNG image has 4 channels, therefore two PNG images have a total of 8 channels to store 8-d features. To avoid having an extra image to store the binary alpha (opacity) channel, we squeeze the alpha channel into the first feature channel, so that the alpha is one when the first feature channel is non-zero, and zero when the channel is zero. Since phones have a hardware constraint that the texture size must be a power of 2 and at most 4096×4096, we split the large texture images into multiple 4096×4096 texture images.

6.4 Results and evaluation

We run a series of experiments to test how well MobileNeRF performs on a wide variety of scenes and devices. We test on three datasets: the 8 synthetic 360° scenes from NeRF [167], the 8 forward-facing scenes from LLFF [166], and 5 unbounded 360° outdoor scenes from Mip-NeRF 360 [11]. We compare with SNeRG [97], since, to our knowledge, it is the only NeRF model that can run in real-time on common devices. We also include extensive ablation studies to investigate the impact of different design choices.

The online demo is available at https://mobile-nerf.github.io, where all the results can be viewed on a web browser. The models in our online demo are the same as the ones used in our paper. The rendered images of our method are nearly identical whether they are rendered in Python (for computing quantitative metrics) or web browsers, see Figure 6.7.

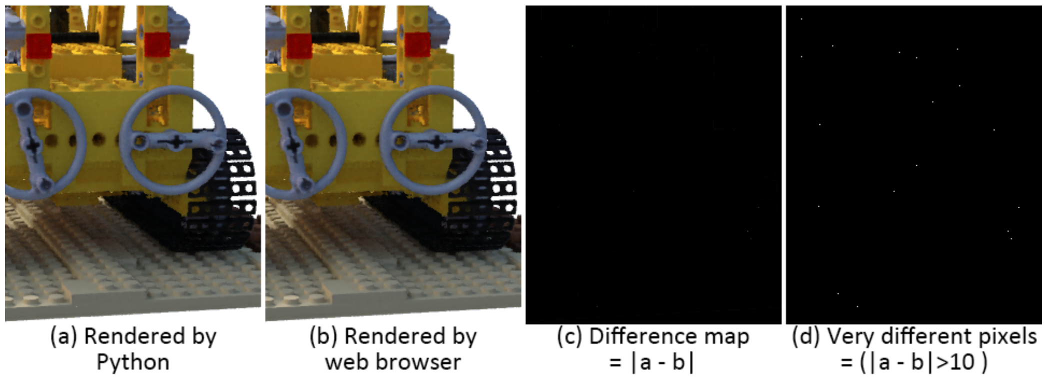

(a) Rendered by Python
(b) Rendered by web browser
(c) Difference map = |a - b|
(d) Very different pixels = (|a - b|>10)

Figure 6.7: Comparison between images rendered in Python and in a web browser. Image pixel value range is 0-255. Zoom in for details.

If one overlays the difference image and the rendered image, one can find that the few very different pixels are all on the boundary of a part, which indicates that they are likely caused by precision errors in rasterization.

For Surface Pro 6, Gaming laptop, and Desktop, we disable frame-rate limiting from vertical synchronization by starting the Chrome browser with the following arguments:

```
--disable-frame-rate-limit
--disable-gpu-vsync
```

However, for phones and Chromebook, we did not find a way to easily disable vertical synchronization, therefore the FPS is capped at 60.

6.4.1 Comparisons

To show the superior performance and compatibility of our method, we test our method and SNeRG on a variety of devices, as shown in Table 6.1. We report the rendering speed in Table 6.2. The rendering resolution is the same as the training images: 800×800 for synthetic, 1008×756 for forward-facing, and 1256×828 for unbounded. We test all methods on a chrome browser and rotate/pan the camera in a full circle to render 360 frames. Note that SNeRG is unable to represent unbounded 360° scenes due to its regular grid representation, and it does not run on phone or tablet due to compatibility or out-of-memory issues. We also report the GPU memory consumption and storage cost in Table 6.3. MobileNeRF requires 5x less GPU memory than SNeRG.

Rendering quality We report the rendering quality in Table 6.4, while comparing with other methods using the common PSNR, SSIM [257], and LPIPS [297] metrics. Our method has roughly the same image quality as SNeRG, and better than NeRF. Visual results are shown in Figure 6.6 (a-c). Our method achieves image quality similar to SNeRG when the

Device	Type	OS	GPU	Power
iPhone XS	Phone	iOS 15	Integrated GPU	6W
Pixel 3	Phone	Android 12	Integrated GPU	9W
Surface Pro 6	Tablet	Windows 10	Integrated GPU	15W
Chromebook	Laptop	Chrome OS	Integrated GPU	15W
Gaming laptop	Laptop	Windows 11	NVIDIA RTX 2070	115W
Desktop	PC	Ubuntu 16.04	NVIDIA RTX 2080 Ti	250W

Table 6.1: **Hardware specs** – of the devices used in our rendering experiments. The power is the max GPU power for discrete NVIDIA cards, and the combined max CPU and GPU power for integrated GPUs.

Dataset	Synthetic 360°		Forward-facing		Unbounded 360°
Method	Ours	SNeRG	Ours	SNeRG	Ours
iPhone XS	**55.89**	$0.0\frac{8}{8}$	**27.19**$\frac{2}{8}$	$0.0\frac{8}{8}$	$22.20\frac{4}{5}$
Pixel 3	**37.14**	$0.0\frac{8}{8}$	**12.40**	$0.0\frac{8}{8}$	9.24
Surface Pro 6	**77.40**	Unsupported	**21.51**	Unsupported	19.44
Chromebook	**53.67**	$22.62\frac{2}{8}$	**19.44**	$7.85\frac{3}{8}$	15.28
Gaming laptop	**178.26**	$8.30\frac{1}{8}$	**57.72**	3.63	55.32
Gaming laptop 🔌	**606.73**	$43.87\frac{1}{8}$	**250.17**	26.01	192.59
Desktop 🔌	**744.91**	207.26	**349.34**	50.71	279.70

Table 6.2: **Rendering speed** – on various devices in frames per second (FPS). The devices are on battery, except for the gaming laptop and the desktop which are plugged in, indicated with a 🔌. The mobile devices (first four rows) have almost identical rendering speed when plugged in. With the notation $\frac{M}{N}$ we indicate that M out of N testing scenes failed to run due to out-of-memory errors.

Dataset	Synthetic 360°		Forward-facing		Unbounded 360°
Method	Ours	SNeRG	Ours	SNeRG	Ours
GPU memory	**538.38**	2707.25	**759.25**	4312.13	1162.20
Disk storage	125.75	**86.75**	**201.50**	337.25	344.60

Table 6.3: **Resources** – memory and disk storage (MB).

camera is at an appropriate distance. When the camera is zoomed in, SNeRG tends to render over-smoothed images.

Polygon count Table 6.5 shows the average number of vertices and triangles produced by our method, and the percentage compared to all available vertices/triangles in the initial mesh. As we only retain visible triangles, most vertices/triangles are removed in the final mesh.

Shading mesh In Figure 6.2a and Figure 6.8, we show the extracted triangle meshes without the textures. Most triangle faces do not align with the actual object surface. This is

	Synthetic 360°			Forward-facing			Unbounded 360°		
	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓	PSNR↑	SSIM↑	LPIPS↓
NeRF	31.00	0.947	0.081	26.50	0.811	0.250	-	-	-
JAXNeRF	31.65	0.952	0.051	26.92	0.831	0.173	21.46	0.458	0.515
NeRF++	-	-	-	-	-	-	22.76	0.548	0.427
SNeRG	30.38	**0.950**	**0.050**	25.63	0.818	**0.183**	-	-	-
Ours	**30.90**	0.947	0.062	**25.91**	**0.825**	**0.183**	21.95	0.470	0.470

Table 6.4: **Quantitative Analysis** – For NeRF [167] and NeRF++ [295], we dash entries where the original papers did not report quantitative performance. For SNeRG, while one could extend the method to include the unbounded design from [11], implementing this is far from trivial. Our method can be easily adapted to work across all modalities.

	Synthetic 360°		Forward-facing		Unbounded 360°	
	V	T	V	T	V	T
Number	494,289	224,341	830,076	338,535	1,436,033	608,785
Percentage	1.964%	1.783%	3.298%	2.690%	4.891%	4.147%

Table 6.5: **Polygon count** – Average number of vertices and triangles produced, and their percentage compared to all available vertices/triangles in the initial mesh.

perhaps due to the ambiguity that good rendering quality can be achieved despite how the triangles are aligned. For example, the results of our method after Stage 1 in Table 6.6 is similar to other methods in Table 6.4. Therefore, better regularization losses or training objectives need to be devised if one wishes to have better surface quality. However, optimizing vertices does improve the rendering quality, as shown in Figure 6.6h.

Per-Scene metrics We provide per-scene breakdown for the quality metrics at the end of this chapter, in Table 6.8 6.9 6.10 6.12 6.13 6.14 6.16 6.17 6.18. We provide per-scene breakdown for rendering speed and storage cost in Table 6.11 6.15 6.19, where OOM (out-of-memory) indicates the device cannot run a testing scene due to GPU memory issues, and ICP (incompatible) indicates the device cannot run the method due to compatibility issues. The GPU memory and disk storage were tested on the Desktop.

6.4.2 Ablation studies

In Table 6.6, we show the rendering quality of our method at each stage, and report our ablation studies. The rendering quality gradually drops after each stage, because each stage adds more constraints to the model. In Stage 1, the performance drops significantly if we use a fixed regular grid mesh instead of having optimizable mesh vertices, or if we forgo view-dependent effects by directly predicting the color and alpha of each point. The performance drops slightly if the grid is smaller (P=64 vs. 128). If we remove the acceleration grid, we are not able to quadruple the batch size during training; the performance drops if we train this model the same number of iterations as our method. Note that the PSNR of this model

	Synthetic 360°		Forward-facing	
	PSNR↑	SSIM↑	PSNR↑	SSIM↑
Stage 1, **our method**	**32.13**	**0.955**	26.57	**0.839**
Stage 1, fixed mesh grid	29.87	0.938	25.43	0.797
Stage 1, no view-dependent MLP	29.91	0.935	25.91	0.824
Stage 1, smaller grid P=128 $\rightarrow$ 64	31.58	0.952	26.39	0.831
Stage 1, no acceleration grid	31.77	0.953	**26.61**	0.835
Stage 2, **our method**	**31.01**	**0.948**	**26.32**	**0.833**
Stage 2, no fine-tuning	30.80	0.946	26.25	0.832
Stage 2, only pseudo-gradients	29.70	0.935	26.01	0.820
Stage 2, binary loss	30.89	0.947	**26.32**	0.832
Stage 3, **our method**	30.90	0.947	25.91	0.825
Stage 3, larger texture K=17 $\rightarrow$ 33	**30.99**	**0.948**	**26.14**	**0.830**
Stage 3, smaller texture K=17 $\rightarrow$ 9	30.49	0.945	24.85	0.796
Stage 3, no supersampling	29.26	0.937	24.88	0.799

Table 6.6: **Ablation –** rendering quality.

	Speed in FPS			Space in MB	
Synthetic 360° scenes	Pixel 3	Surface Pro 6	Gaming laptop 🔌	GPU memory	Disk storage
our method	37.14	77.40	606.73	538.38	125.75
Larger texture $K = 33$	$32.48\frac{2}{8}$	59.15	589.20	1290.88	283.50
Smaller texture $K = 9$	37.74	94.62	617.74	**336.63**	**67.00**
No supersampling	51.81	**113.41**	**649.86**	440.25	125.75
No view-dependent MLP	**52.16**	96.76	638.30	538.38	125.75
Forward-facing scenes	Pixel 3	Surface Pro 6	Gaming laptop 🔌	GPU memory	Disk storage
our method	12.40	21.51	250.17	759.25	201.50
Larger texture $K = 33$	$12.88\frac{3}{8}$	18.79	241.52	2024.13	462.75
Smaller texture $K = 9$	12.70	23.61	257.64	**394.13**	**105.75**
No supersampling	16.97	**42.11**	**413.02**	645.00	201.50
No view-dependent MLP	**23.72**	28.06	385.65	759.25	201.50

Table 6.7: **Ablation –** rendering speed/memory.

is higher on forward-facing scenes. This is because the acceleration grid will remove cells that are not visible in the training images, thus cannot "inpaint" the objects and may leave holes. In Stage 2, if we do not perform the fine-tuning step that only optimizes $\mathcal{F}$ and $\mathcal{H}$ and fix the weights of others, the performance drops. If we only use the binary opacity with pseudo-gradients by applying $\mathcal{L}_{\mathbf{C}}^{\text{stage2}}=\mathcal{L}_{\mathbf{C}}^{\text{bin}}$ instead of Eq. 6.14, the performance drops. If we use a binary loss on the predicted opacity, e.g., $\mathcal{L}_{binary}= - \sum |\alpha_k - 0.5|$, instead of using the pseudo-gradients with $\hat{\mathbf{C}}(\mathbf{r})$, the performance drops slightly. In stage 3, when we use a larger texture size K=33 instead of 17, the performance improves, but the texture size will be quadrupled; the performance drops when we use a smaller texture size K=9.

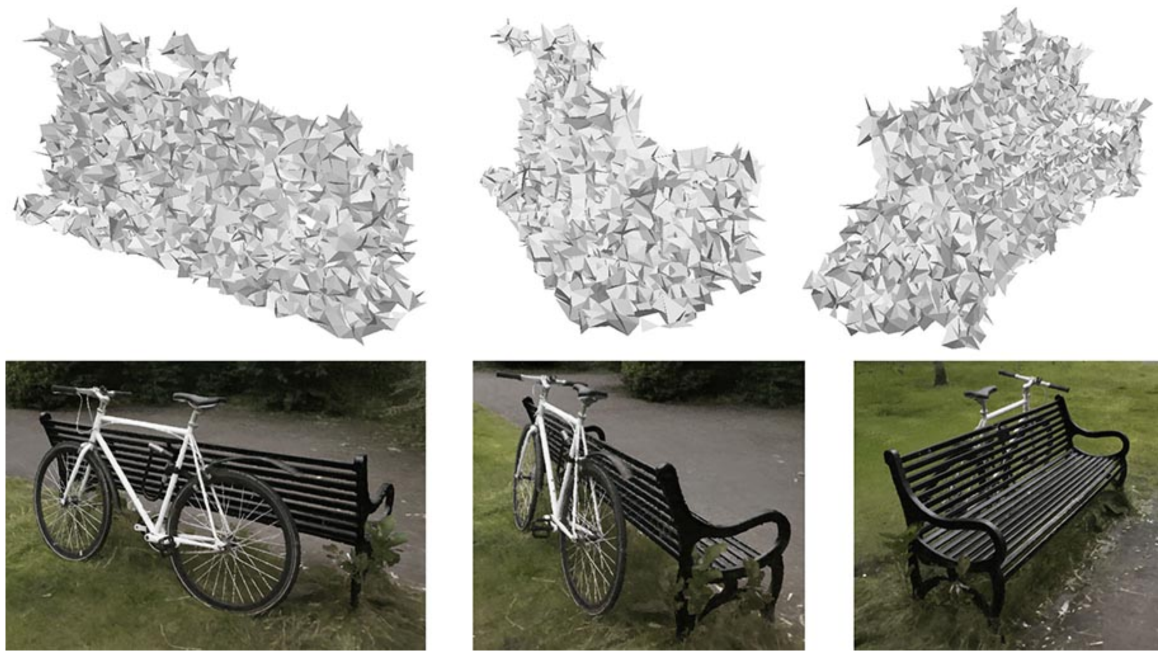


Figure 6.8: **Shading mesh** – not textured. The mesh corresponds to the bicycle (see Figure 6.1). We manually removed the background mesh to better show the geometry of the object. Zoom-in to see more details. In the bottom, we also show the rendered images of our method. Note how the coarse mesh is able to represent detailed structures such as the spokes of the wheels and the wires around the handles, thanks to high-resolution textures with transparencies.

If we remove the super-sampling step, the performance drops significantly. Visual results are shown in Figure 6.6. We omit some models because they do not have significant visual differences compared to our method. Notice the squared pixels of the texture images are clearly visible in the dashed-line boxes in (e) and almost invisible in (d). The aliasing artifacts are conspicuous in the solid-line boxes in (f). In Stage 1, if the grid vertices cannot be optimized, the results will be significantly worse, as shown in (h). Without the small MLP, the model cannot handle reflections, as shown in (i). Changing to a smaller grid size introduces some minor artifacts in (j). In Table 6.7. we show the rendering speed and space cost if we use a larger or smaller texture size, or if we remove the super-sampling step, or if we only perform the rasterization without using the small MLP to predict the view-dependent colors. One can find that the super-sampling step and the small MLP have the most significant impact.

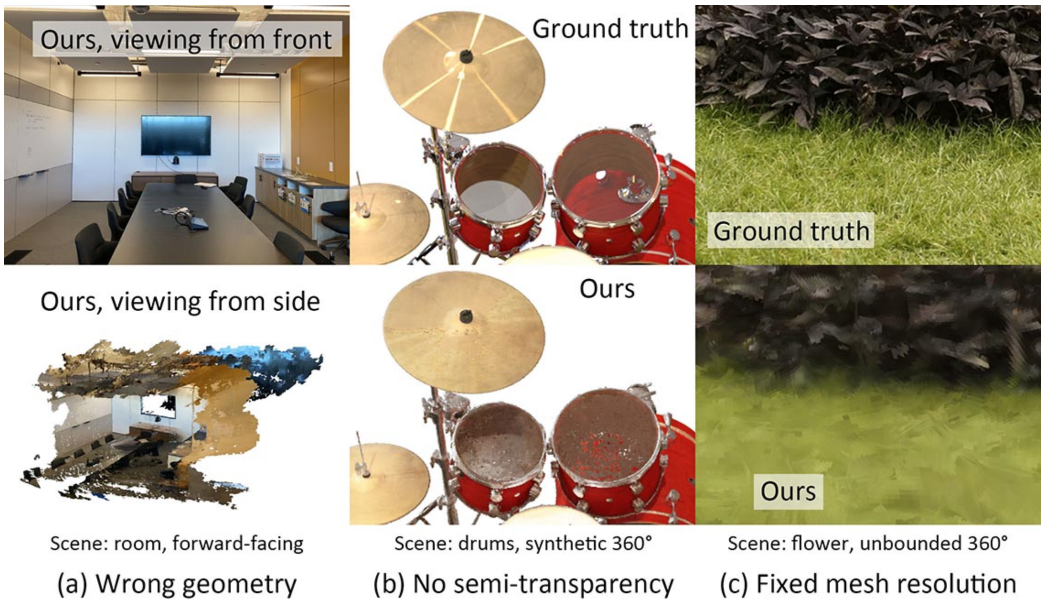


Figure 6.9: **Limitations** – (a) the monitor/table are hollow, because the reflections are modelled as real objects behind the monitor and below the table. (b) our method generates scattered small fragments in the semi-transparent parts. (c) the camera is too close to the scene and details in the grass cannot be represented at the chosen texture resolution.

6.4.3 Scene editing

The explicit mesh representation provided by MobileNeRF gives us direct editing control over the NeRF model without any complex architectural change (e.g. ControlNerf [128]), but in this paper we only superficially investigated these possibilities.

Our representation is a textured mesh with baked lighting, and thus can be used in any application that combines, renders, or manipulates such meshes. Figure 6.10 (a) shows a simple example where meshes learned from four different sets of photos are composited into a single scene. The scene, rendered in 1920×1080 resolution without super-sampling, runs at 150 FPS on the gaming laptop, and consumes 1.5 GB of GPU memory. Similarly Figure 6.10 (b)(c) show scenes where some parts or objects are edited or removed by manipulating the triangle meshes of the scenes in a 3D modeling software. The resulting renders do not account for differences in illumination between the captured photos or indirect illumination between different meshes. However, it suggests an easy way to create “photorealistic-looking” scenes from a library of objects captured using photos rather than painstaking 3D modeling.

https://youtu.be/kVy2W6afuyk shows three examples where we manipulate the learned NeRF objects interactively in real-time. We also highlight how easy it is to implement these operations with our mesh representation. In contrast, implementing those with classic NeRF is non-trivial.

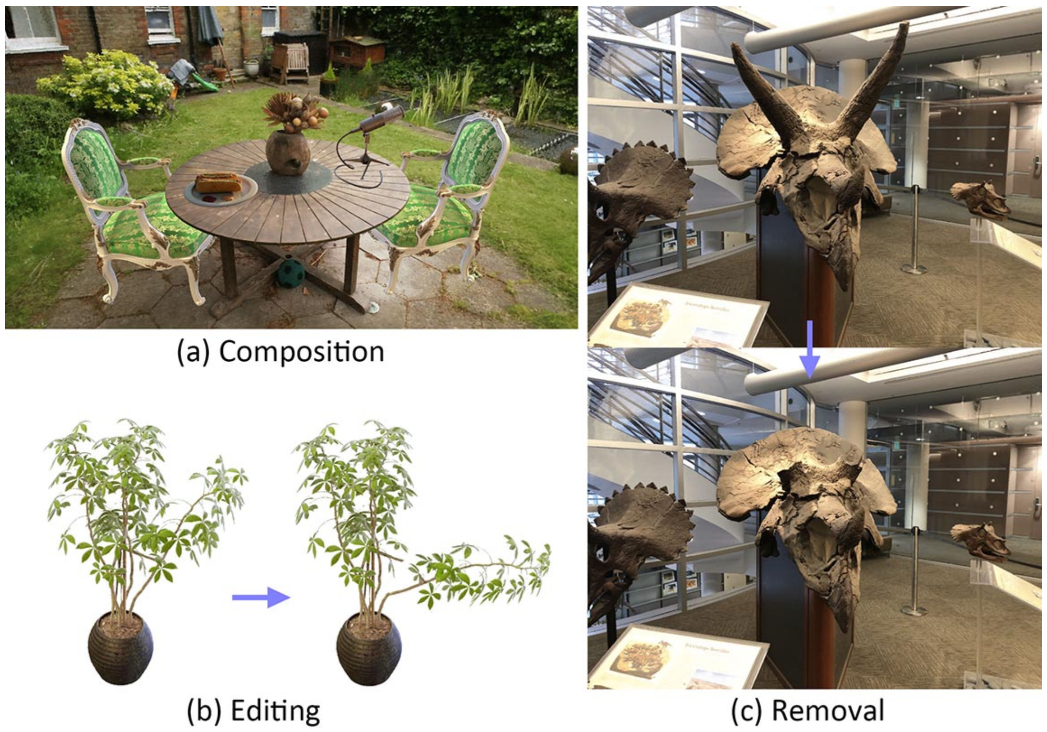


Figure 6.10: **Scene editing** – (a) four objects learned from the synthetic scenes are added into an unbounded scene. (b) a branch of the ficus is bent. (c) the horns are removed.

In the first example, we render all 8 objects learned from the synthetic scenes at the same time, and we move the objects by using mouse to drag the objects. This is implemented by a single line of code with the *DragControls* class provided in the *threejs* library. *DragControls* is designed for manipulating meshes, which suits our needs exactly since our objects are meshes. We also cast real-time shadow of the objects by applying shadow mapping. This is implemented by having a directional light, an ambient light, and a plane below the objects to receive shadows. The drag control and the real-time shadow are also used in the following examples.

In the second example, we interactively deform the learned chair object to create new variations of chairs. To implement the deformation, we only need to deform the vertex positions of the meshes, and this is achieved by adding vertex deformation code in the vertex shader. Specifically, we implemented three operations: moving the chair up/down will lengthen or shorten its legs, moving the chair left/right will adjust its width, and moving the chair forward/backward will adjust the skew of its back.

In the third example, we render 9 ficus objects, which are considered "NeRF" objects, and a blue ball, which is a classic object with standard material used in classic rendering. We again change the vertex shader to make the leaves of the plants to be repelled by the blue ball.

6.5 Conclusions

We introduce MobileNeRF, an architecture that takes advantage of the classical rasterization pipeline (i.e. z-buffers and fragment shaders) to perform efficient rendering of surface-based neural fields on a wide range of compute platforms. It achieves frame rates an order of magnitude faster than the previous state-of-the-art (SNeRG) while producing images of equivalent quality.

Limitations Our estimated *surface* may be incorrect, especially for scenes with specular surfaces and/or sparse views (Figure 6.9a); it uses *binary* opacities to avoid sorting polygons, and thus cannot handle scenes with semi-transparencies (Figure 6.9b); it uses fixed mesh and texture resolutions, which may be too coarse for close-up novel-view synthesis (Figure 6.9c); it models a radiance field without explicitly decomposing illumination and reflectance, and thus does not handle glossy surfaces as well as recent methods [247]. Extending the polygon rendering pipeline with efficient partial sorting, levels-of-detail, mipmaps, and surface shading should address some of these issues. Also, the current training speed of MobileNeRF is slow due to NeRF's MLP backbone. The extension of MobileNeRF to fast-training architectures (e.g., Instant NGP [172]) constitutes an exciting avenue for future works.

	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
NeRF [167]	33.00	25.01	30.13	36.18	32.54	29.62	32.91	28.65	31.00
JAXNeRF [56]	33.88	25.08	30.51	36.91	33.24	30.03	34.52	29.07	31.65
SNeRG [97]	33.24	24.57	29.32	34.33	33.82	27.21	32.60	27.97	30.38
Ours	34.09	25.02	30.20	35.46	34.18	26.72	32.48	29.06	30.90

Table 6.8: **PSNR**↑ on **Synthetic** 360° **scenes**.

	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
NeRF [167]	0.967	0.925	0.964	0.974	0.961	0.949	0.980	0.856	0.947
JAXNeRF [56]	0.974	0.927	0.967	0.979	0.968	0.952	0.987	0.865	0.952
SNeRG [97]	0.975	0.929	0.967	0.971	0.973	0.938	0.982	0.865	0.950
Ours	0.978	0.927	0.965	0.973	0.975	0.913	0.979	0.867	0.947

Table 6.9: **SSIM**↑ on **Synthetic** 360° **scenes**.

	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
NeRF [167]	0.046	0.091	0.044	0.121	0.050	0.063	0.028	0.206	0.081
JAXNeRF [56]	0.027	0.070	0.033	0.030	0.030	0.048	0.013	0.156	0.051
SNeRG [97]	0.025	0.061	0.028	0.043	0.022	0.052	0.016	0.156	0.050
Ours	0.025	0.077	0.048	0.050	0.025	0.092	0.032	0.145	0.062

Table 6.10: **LPIPS**↓ on **Synthetic** 360° **scenes**.

				SNeRG [97]					
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
iPhone XS	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Pixel 3	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Surface Pro 6	ICP	ICP	ICP	ICP	ICP	ICP	ICP	ICP	-
Chromebook	28.06	OOM	OOM	26.11	27.08	16.48	26.99	11.01	22.62
Gaming laptop	4.94	10.27	OOM	8.10	9.41	2.05	21.65	1.69	8.30
Gaming laptop 🔌	37.66	51.06	OOM	45.52	60.20	13.81	87.67	11.17	43.87
Desktop 🔌	120.70	147.72	81.88	436.05	232.03	92.45	507.54	39.73	207.26
GPU memory	1254.00	4729.00	8243.00	1253.00	1253.00	1253.00	1251.00	2422.00	2707.25
Disk storage	141.00	44.00	43.00	67.00	114.00	134.00	22.00	129.00	86.75

				Ours					
	Chair	Drums	Ficus	Hotdog	Lego	Materials	Mic	Ship	Mean
iPhone XS	60.00	60.00	60.00	60.00	50.10	54.65	60.00	42.37	55.89
Pixel 3	41.68	38.71	43.09	35.59	29.56	32.35	52.65	23.52	37.14
Surface Pro 6	83.40	83.15	99.34	64.01	57.11	58.80	130.62	42.76	77.40
Chromebook	60.00	60.00	60.00	53.24	47.51	51.04	60.00	37.56	53.67
Gaming laptop	186.03	183.04	231.01	156.08	118.27	129.80	332.10	89.74	178.26
Gaming laptop 🔌	657.77	656.22	643.32	649.58	566.39	618.98	648.88	412.70	606.73
Desktop 🔌	810.99	789.30	882.23	707.27	629.70	659.95	970.35	509.48	744.91
GPU memory	451.00	590.00	450.00	456.00	723.00	721.00	322.00	594.00	538.38
Disk storage	107.00	120.00	80.00	88.00	199.00	191.00	50.00	171.00	125.75

Table 6.11: **Rendering speed** in frames per second (FPS), and **GPU memory and disk storage** in MB, on **Synthetic** 360° **scenes**.

	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
NeRF [167]	32.70	25.17	20.92	31.16	20.36	27.40	26.80	27.45	26.50
JAXNeRF [56]	33.30	24.92	21.24	31.78	20.32	28.09	27.43	28.29	26.92
SNeRG [97]	30.04	24.85	20.01	30.91	19.73	27.00	25.80	26.71	25.63
Ours	31.28	24.59	20.54	30.82	19.66	27.05	26.26	27.09	25.91

Table 6.12: **PSNR↑** on **Forward-facing scenes**.

	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
NeRF [167]	0.948	0.792	0.690	0.881	0.641	0.827	0.880	0.828	0.811
JAXNeRF [56]	0.958	0.806	0.717	0.897	0.657	0.850	0.902	0.863	0.831
SNeRG [97]	0.936	0.802	0.696	0.889	0.655	0.835	0.882	0.852	0.818
Ours	0.943	0.808	0.711	0.891	0.647	0.839	0.900	0.864	0.825

Table 6.13: **SSIM↑** on **Forward-facing scenes**.

	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
NeRF [167]	0.178	0.280	0.316	0.171	0.321	0.219	0.249	0.268	0.250
JAXNeRF [56]	0.086	0.207	0.247	0.108	0.266	0.156	0.143	0.173	0.173
SNeRG [97]	0.133	0.198	0.252	0.125	0.255	0.167	0.157	0.176	0.183
Ours	0.143	0.202	0.245	0.115	0.277	0.163	0.147	0.169	0.183

Table 6.14: **LPIPS↓** on **Forward-facing scenes**.

					SNeRG [97]				
	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
iPhone XS	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Pixel 3	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM	-
Surface Pro 6	ICP	ICP	ICP	ICP	ICP	ICP	ICP	ICP	-
Chromebook	9.75	6.02	OOM	9.68	OOM	5.12	8.68	OOM	7.85
Gaming laptop	7.77	1.28	0.80	8.46	1.14	0.67	4.72	4.18	3.63
Gaming laptop 🔌	52.40	14.45	6.15	54.47	12.43	8.77	32.87	26.51	26.01
Desktop 🔌	110.36	28.18	13.54	122.91	17.59	15.96	62.65	34.46	50.71
GPU memory	3594.00	3585.00	4729.00	3595.00	5903.00	3593.00	3595.00	5903.00	4312.13
Disk storage	149.00	288.00	408.00	162.00	704.00	321.00	251.00	415.00	337.25

					Ours				
	Room	Fern	Leaves	Fortress	Orchids	Flower	Trex	Horns	Mean
iPhone XS	29.82	25.10	OOM	30.02	OOM	26.28	26.30	25.59	27.19
Pixel 3	13.57	12.74	8.66	14.69	10.77	12.98	13.07	12.71	12.40
Surface Pro 6	22.92	20.32	13.84	29.13	17.10	22.30	22.53	23.92	21.51
Chromebook	20.70	18.95	14.65	23.16	16.79	20.06	20.08	21.12	19.44
Gaming laptop	64.27	55.88	37.11	76.29	48.72	60.60	59.65	59.26	57.72
Gaming laptop 🔌	281.01	252.70	170.66	303.77	222.54	260.44	258.45	251.82	250.17
Desktop 🔌	377.87	352.01	254.51	397.00	323.54	367.68	359.68	362.46	349.34
GPU memory	610.00	610.00	1143.00	473.00	1276.00	611.00	604.00	747.00	759.25
Disk storage	127.00	147.00	353.00	89.00	372.00	151.00	162.00	211.00	201.50

Table 6.15: **Rendering speed** in frames per second (FPS), and **GPU memory and disk storage** in MB, on **Forward-facing scenes**.

	Bicycle	Flower	Garden	Stump	Treehill	Mean
JAXNeRF [56]	21.76	19.40	23.11	21.73	21.28	21.46
NeRF++ [295]	22.64	20.31	24.32	24.34	22.20	22.76
Ours	21.70	18.86	23.54	23.95	21.72	21.95

Table 6.16: **PSNR↑** on **Unbounded** 360° **scenes**.

	Bicycle	Flower	Garden	Stump	Treehill	Mean
JAXNeRF [56]	0.455	0.376	0.546	0.453	0.459	0.458
NeRF++ [295]	0.526	0.453	0.635	0.594	0.530	0.548
Ours	0.426	0.321	0.599	0.556	0.450	0.470

Table 6.17: **SSIM↑** on **Unbounded** 360° **scenes**.

	Bicycle	Flower	Garden	Stump	Treehill	Mean
JAXNeRF [56]	0.536	0.529	0.415	0.551	0.546	0.515
NeRF++ [295]	0.455	0.466	0.331	0.416	0.466	0.427
Ours	0.513	0.526	0.358	0.430	0.522	0.470

Table 6.18: **LPIPS↓** on **Unbounded** 360° **scenes**.

			Ours			
	Bicycle	Flower	Garden	Stump	Treehill	Mean
iPhone XS	OOM	OOM	22.20	OOM	OOM	22.20
Pixel 3	9.44	8.61	10.49	8.54	9.12	9.24
Surface Pro 6	20.24	19.12	21.67	18.21	17.97	19.44
Chromebook	15.89	14.72	16.56	14.23	15.02	15.28
Gaming laptop	55.62	59.18	58.19	51.73	51.89	55.32
Gaming laptop 🔌	195.63	194.66	204.31	178.89	189.46	192.59
Desktop 🔌	280.24	282.02	295.74	265.90	274.58	279.70
GPU memory	1350.00	1081.00	808.00	1082.00	1490.00	1162.20
Disk storage	400.00	294.00	239.00	337.00	453.00	344.60

Table 6.19: **Rendering speed** in frames per second (FPS), and **GPU memory and disk storage** in MB, on **Unbounded** 360° **scenes**.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we introduce several deep learning algorithms that reconstruct meshes from various input sources, including voxels, point clouds, single images, and multi-view images. These methods encompass models that rely on encoding global shape features, encoding local features, and differentiable rendering. They provide a wide range of tools for high-performance mesh reconstruction and generation, as well as frameworks for future research and industrial applications.

In Chapter 3, we introduced BSP-Net, a shape decoder network that generates compact and structured polygonal meshes in the form of convex decomposition. Trained as a neural implicit representation, our network learns an explicit BSP-tree built on a set of planes, and in turn, a set of convex primitives, in an unsupervised manner. These planes and convexes are defined by weights learned by the neural network. Compared to state-of-the-art methods, meshes generated by BSP-Net exhibit superior visual quality, especially in sharp geometric details.

In Chapter 4, we introduced Neural Marching Cubes, the first iso-surfacing algorithm capable of recovering sharp geometric features without requiring additional inputs, such as normal information. We designed an efficient parameterization to represent a triangle mesh in a regular grid structure that is compatible with neural processing. Trained on automatically generated "ground-truth" meshes, our method shows superior performance in preserving various geometric features such as sharp/soft edges, corners, and thin structures.

In Chapter 5, we introduced Neural Dual Contouring, a new data-driven approach to mesh reconstruction based on Dual Contouring. The signed version of our approach, NDC, has similar performance with Neural Marching Cubes but produces 3-7 times fewer vertices and triangles. The unsigned version of our approach, UNDC, is sign agnostic, therefore able to reconstruct open surfaces and thin structures from unsigned distance fields or unoriented point clouds. Both NDC and UNDC are designed as local networks using limited receptive fields, thus can generalize well to new datasets. Extensive experiments demonstrate the superior performance of our approach on multiple datasets over traditional and deep-learning state-of-the-art methods.

In Chapter 6, we introduced MobileNeRF, an architecture that takes advantage of the classical rasterization pipeline (i.e. z-buffers and fragment shaders) to perform efficient rendering of surface-based neural fields on a wide range of compute platforms. Thanks to its explicit mesh representation with deep-feature textures, MobileNeRF achieves frame rates an order of magnitude faster than the previous state-of-the-art while producing images of equivalent quality.

7.2 Future Work

BSP-Net adopts an encoder-encoder structure, therefore, in principle, it can reconstruct meshes from all types of inputs as long as they can be encoded into a global shape latent code. However, recent evidence has suggested that relying solely on global features may lead to overfitting to the training category and inability to represent detailed geometries. NMC and NDC employ fully convolutional networks with limited receptive fields, therefore they are only aware of local features in the input, which gives them strong generalization ability across new shapes and new datasets, but in turn hinders their ability to perform global-aware inference such as shape completion and shape generation. MobileNeRF relies purely on differentiable rendering, and thus cannot benefit from any learned priors from large training corpuses. Therefore, unifying global features, local features, and test-time optimization such as differentiable rendering, is an interesting topic for future research on neural mesh reconstruction.

Reconstructing 3D representations from multi-view images has been a trending research topic these days. Our attempt, MobileNeRF, has an apparent undesirable trait that its output is a polygon soup, where the triangles neither align with the actual shape surface nor form manifold meshes. In addition, MobileNeRF uses fixed mesh and texture resolutions, which cannot adapt to the granularity of different objects in the scene. Besides, how to model semi-transparent objects, reflection, refraction, and other light effects remains an open and challenging research problem.

The primary focus of this thesis is on mesh reconstruction, where substantial information in the inputs is relayed to the mesh outputs. Generative models that can take sparse and ambiguous inputs, such as texts, to perform shape editing and novel shape synthesis constitute another valuable research field. Existing works in this field are predominantly based on differentiable rendering, neural fields, and optionally, differentiable iso-surfacing. However, these approaches based on differentiable rendering and large text-to-image models differ significantly from how humans imagine new shapes. It prompts us to explore more natural ways of creating 3D shapes with neural networks, and perhaps ways that offer compact meshes, as opposed to those extracted from neural fields.

Bibliography

[1] Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas J. Guibas. Learning representations and generative models for 3d point clouds. In *ICML*, 2018.

[2] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *SIGGRAPH*, page 415–421, 1998.

[3] Benjamin Attal, Jia-Bin Huang, Michael Zollhöfer, Johannes Kopf, and Changil Kim. Learning neural light fields with ray-space embedding networks. *CVPR*, 2022.

[4] Benjamin Attal, Selena Ling, Aaron Gokaslan, Christian Richardt, and James Tompkin. MatryODShka: Real-time 6DoF video view synthesis using multi-sphere images. *ECCV*, 2020.

[5] Matan Atzmon and Yaron Lipman. Sal: Sign agnostic learning of shapes from raw data. In *CVPR*, pages 2562–2571, 2020.

[6] Matan Atzmon and Yaron Lipman. Sald: Sign agnostic learning with derivatives. In *International Conference on Learning Representations*, 2020.

[7] Dejan Azinović, Ricardo Martin-Brualla, Dan B Goldman, Matthias Nießner, and Justus Thies. Neural rgb-d surface reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6290–6301, 2022.

[8] Abhishek Badki, Orazio Gallo, Jan Kautz, and Pradeep Sen. Meshlet priors for 3D mesh reconstruction. In *CVPR*, pages 2849–2858, 2020.

[9] Sai Praveen Bangaru, Michael Gharbi, Fujun Luan, Tzu-Mao Li, Kalyan Sunkavalli, Milos Hasan, Sai Bi, Zexiang Xu, Gilbert Bernstein, and Fredo Durand. Differentiable rendering of neural sdfs through reparameterization. In *SIGGRAPH Asia 2022 Conference Papers*, pages 1–9, 2022.

[10] Alan H Barr. Superquadrics and angle-preserving transformations. *IEEE Computer graphics and Applications*, 1(1):11–23, 1981.

[11] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. *CVPR*, 2022.

[12] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 2011.

[13] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[14] Matthew Berger, Andrea Tagliasacchi, Lee M. Seversky, Pierre Alliez, Gaël Guennebaud, Joshua A. Levine, Andrei Sharf, and Claudio T Silva. A survey of surface reconstruction from point clouds. In *Computer Graphics Forum*, volume 36, pages 301–329, 2017.

[15] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Cláudio Silva, and Gabriel Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE TVCG*, 5(4):349–359, 1999.

[16] Tobias Bertel, Mingze Yuan, Reuben Lindroos, and Christian Richardt. OmniPhotos: Casual 360° VR photography. *ACM Transactions on Graphics*, 2020.

[17] Bharat Lal Bhatnagar, Garvita Tiwari, Christian Theobalt, and Gerard Pons-Moll. Multi-garment net: Learning to dress 3d people from images. In *ICCV*, pages 5420–5430, 2019.

[18] Federica Bogo, Angjoo Kanazawa, Christoph Lassner, Peter Gehler, Javier Romero, and Michael J Black. Keep it smpl: Automatic estimation of 3d human pose and shape from a single image. In *European conference on computer vision*, pages 561–578. Springer, 2016.

[19] Federica Bogo, Javier Romero, Matthew Loper, and Michael J. Black. FAUST: Dataset and evaluation for 3D mesh registration. In *CVPR*, pages 3794–3801, 2014.

[20] Alexandre Boulch and Renaud Marlet. Fast and robust normal estimation for point clouds with sharp features. *Computer Graphics Forum*, 31(5):1765–1774, 2012.

[21] Alexandre Boulch and Renaud Marlet. Poco: Point convolution for surface reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6302–6314, 2022.

[22] Michael Broxton, John Flynn, Ryan Overbeck, Daniel Erickson, Peter Hedman, Matthew DuVall, Jason Dourgarian, Jay Busch, Matt Whalen, and Paul Debevec. Immersive light field video with a layered mesh representation. *ACM Transactions on Graphics*, 2020.

[23] Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. Unstructured lumigraph rendering. In *Proceedings of Computer graphics and interactive techniques*, 2001.

[24] Brent Burley and Dylan Lacewell. Ptex: Per-face texture mapping for production rendering. In *Computer Graphics Forum*, 2008.

[25] Yan-Pei Cao, Zheng-Ning Liu, Zheng-Fei Kuang, Leif Kobbelt, and Shi-Min Hu. Learning to reconstruct high-quality 3d shapes with cascaded fully convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 616–633, 2018.

[26] Rohan Chabra, Jan E Lenssen, Eddy Ilg, Tanner Schmidt, Julian Straub, Steven Lovegrove, and Richard Newcombe. Deep local shapes: Learning local sdf priors for detailed 3d reconstruction. In *European Conference on Computer Vision*, pages 608–625. Springer, 2020.

[27] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niebner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D data in indoor environments. In *3DV*, pages 667–676, 2017.

[28] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An information-rich 3D model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[29] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. *ECCV*, 2022.

[30] Ding-Yun Chen, Xiao-Pei Tian, Yu-Te Shen, and Ming Ouhyoung. On visual similarity based 3d model retrieval. In *Computer graphics forum*, 2003.

[31] Wenzheng Chen, Huan Ling, Jun Gao, Edward Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. Learning to predict 3d objects with an interpolation-based differentiable renderer. *Advances in Neural Information Processing Systems*, 32, 2019.

[32] Wenzheng Chen, Joey Litalien, Jun Gao, Zian Wang, Clement Fuji Tsang, Sameh Khamis, Or Litany, and Sanja Fidler. Dib-r++: Learning to predict lighting and material with a hybrid differentiable renderer. *Advances in Neural Information Processing Systems*, 34:22834–22848, 2021.

[33] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. Mobilenerf: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. In *The Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[34] Zhiqin Chen, Vladimir G Kim, Matthew Fisher, Noam Aigerman, Hao Zhang, and Siddhartha Chaudhuri. Decor-gan: 3d shape detailization by conditional refinement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 15740–15749, 2021.

[35] Zhiqin Chen, Andrea Tagliasacchi, Thomas Funkhouser, and Hao Zhang. Neural dual contouring. *ACM Transactions on Graphics*, 2022.

[36] Zhiqin Chen, Andrea Tagliasacchi, and Hao Zhang. BSP-Net: generating compact meshes via binary space partitioning. In *CVPR*, 2020.

[37] Zhiqin Chen, Kangxue Yin, Matthew Fisher, Siddhartha Chaudhuri, and Hao Zhang. BAE-NET: Branched autoencoder for shape co-segmentation. *ICCV*, 2019.

[38] Zhiqin Chen and Hao Zhang. Learning implicit fields for generative shape modeling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5939–5948, 2019.

[39] Zhiqin Chen and Hao Zhang. Neural marching cubes. *ACM Transactions on Graphics*, 40(6):1–15, 2021.

[40] Evgeni Chernyaev. Marching Cubes 33: construction of topologically correct isosurfaces. Technical Report CN/95-17, CERN, 1995.

[41] Julian Chibane, Thiemo Alldieck, and Gerard Pons-Moll. Implicit functions in feature space for 3d shape reconstruction and completion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6970–6981, 2020.

[42] Julian Chibane, Gerard Pons-Moll, et al. Neural unsigned distance fields for implicit function learning. *Advances in Neural Information Processing Systems*, 33:21638–21652, 2020.

[43] Christopher B Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. 3d-r2n2: A unified approach for single and multi-view 3d object reconstruction. In *European conference on computer vision*, pages 628–644. Springer, 2016.

[44] Paolo Cignoni, Fabio Ganovelli, Claudio Montani, and Roberto Scopigno. Reconstruction of topologically correct and adaptive trilinear isosurfaces. *Computers & Graphics*, 2000.

[45] David Cohen-Steiner and Tran Kai Frank Da. A greedy Delaunay-based surface reconstruction algorithm. *The Visual Computer*, 20(1):4–16, 2004.

[46] Forrester Cole, Kyle Genova, Avneesh Sud, Daniel Vlasic, and Zhoutong Zhang. Differentiable surface rendering via non-differentiable sampling. In *ICCV*, 2021.

[47] Steven A Coons. Surfaces for computer-aided design of space forms. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1967.

[48] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *SIGGRAPH*, pages 303–312, 1996.

[49] Lis Custodio, Tiago Etiene, Sinesio Pesco, and Claudio Silva. Practical considerations on marching cubes 33 topological correctness. *Computers & Graphics*, 2013.

[50] Angela Dai and Matthias Nießner. Scan2mesh: From unstructured range scans to 3d meshes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5574–5583, 2019.

[51] Angela Dai, Charles Ruizhongtai Qi, and Matthias Nießner. Shape completion using 3d-encoder-predictor cnns and shape synthesis. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5868–5877, 2017.

[52] François Darmon, Bénédicte Bascle, Jean-Clément Devaux, Pascal Monasse, and Mathieu Aubry. Improving neural implicit surfaces geometry with patch warping. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6260–6269, 2022.

[53] Abe Davis, Marc Levoy, and Fredo Durand. Unstructured light fields. *Computer Graphics Forum*, 2012.

[54] Bruno Rodrigues De Araújo, Daniel S. Lopes, Pauline Jepp, Joaquim A. Jorge, and Brian Wyvill. A survey on implicit surface polygonization. *ACM Computing Surveys (CSUR)*, 47(4):1–39, 2015.

[55] Paul Debevec, Yizhou Yu, and George Borshukov. Efficient view-dependent image-based rendering with projective texture-mapping. In *Eurographics Workshop on Rendering Techniques*, 1998.

[56] Boyang Deng, Jonathan T. Barron, and Pratul P. Srinivasan. JaxNeRF: an efficient JAX implementation of NeRF, 2020.

[57] Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton, and Andrea Tagliasacchi. Cvxnet: Learnable convex decomposition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 31–44, 2020.

[58] Yu Deng, Jiaolong Yang, Sicheng Xu, Dong Chen, Yunde Jia, and Xin Tong. Accurate 3d face reconstruction with weakly-supervised learning: From single image to image set. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[59] Tamal K. Dey and Joshua A. Levine. Delaunay meshing of isosurfaces. *The Visual Computer*, 2008.

[60] Akio Doi and Akio Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Transactions on Information and Systems*, 1991.

[61] Martin J. Dürst. Letters: additional reference to marching cubes. In *SIGGRAPH*, 1988.

[62] Philipp Erler, Paul Guerrero, Stefan Ohrhallinger, Niloy J Mitra, and Michael Wimmer. Points2surf learning implicit surfaces from point clouds. In *European Conference on Computer Vision*, pages 108–124. Springer, 2020.

[63] Wang et al. HF-NeuS: Improved surface reconstruction using high-frequency details. In *NeurIPS*, 2022.

[64] Tiago Etiene, Luis Gustavo Nonato, Carlos Scheidegger, Julien Tienry, Thomas J. Peters, Valerio Pascucci, Robert M. Kirby, and Cláudio T. Silva. Topology verification for isosurface extraction. *TVCG*, 2011.

[65] Haoqiang Fan, Hao Su, and Leonidas J. Guibas. A point set generation network for 3d object reconstruction from a single image. In *CVPR*, 2017.

[66] John Flynn, Michael Broxton, Paul Debevec, Matthew DuVall, Graham Fyffe, Ryan Overbeck, Noah Snavely, and Richard Tucker. DeepView: View synthesis with learned gradient descent. *CVPR*, 2019.

[67] Qiancheng Fu, Qingshan Xu, Yew-Soon Ong, and Wenbing Tao. Geo-neus: Geometry-consistent neural implicit surfaces learning for multi-view reconstruction. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[68] Henry Fuchs, Zvi M Kedem, and Bruce F Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 124–133, 1980.

[69] Thomas Funkhouser, Michael Kazhdan, Philip Shilane, Patrick Min, William Kiefer, Ayellet Tal, Szymon Rusinkiewicz, and David Dobkin. Modeling by example. *ACM Transactions on Graphics*, 23(3):652–663, 2004.

[70] Matheus Gadelha, Rui Wang, and Subhransu Maji. Multiresolution tree networks for 3d point cloud processing. In *ECCV*, 2018.

[71] Jun Gao, Wenzheng Chen, Tommy Xiang, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Learning deformable tetrahedral meshes for 3d reconstruction. *Advances In Neural Information Processing Systems*, 33:9936–9947, 2020.

[72] Lin Gao, Yu-Kun Lai, Jie Yang, Ling-Xiao Zhang, Leif Kobbelt, and Shihong Xia. Sparse data driven mesh deformation. *IEEE transactions on visualization and computer graphics*, 2019.

[73] Lin Gao, Tong Wu, Yu-Jie Yuan, Ming-Xian Lin, Yu-Kun Lai, and Hao Zhang. Tm-net: Deep generative networks for textured meshes. *ACM Transactions on Graphics (TOG)*, 40(6):1–15, 2021.

[74] Lin Gao, Jie Yang, Tong Wu, Yu-Jie Yuan, Hongbo Fu, Yu-Kun Lai, and Hao Zhang. Sdm-net: Deep generative network for structured deformable mesh. *ACM Transactions on Graphics (TOG)*, 38(6):1–15, 2019.

[75] Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien P. C. Valentin. Fastnerf: High-fidelity neural rendering at 200fps. In *ICCV*, 2021.

[76] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH*, pages 209–216, 1997.

[77] Kyle Genova, Forrester Cole, Avneesh Sud, Aaron Sarna, and Thomas Funkhouser. Local deep implicit functions for 3d shape. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4857–4866, 2020.

[78] Kyle Genova, Forrester Cole, Daniel Vlasic, Aaron Sarna, William T Freeman, and Thomas Funkhouser. Learning shape templates with structured implicit functions. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7154–7164, 2019.

[79] Rohit Girdhar, David F Fouhey, Mikel Rodriguez, and Abhinav Gupta. Learning a predictable and generative vector representation for objects. In *ECCV*, 2016.

[80] Shubham Goel, Georgia Gkioxari, and Jitendra Malik. Differentiable stereopsis: Meshes from multiple views using differentiable rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8635–8644, 2022.

[81] Shubham Goel, Angjoo Kanazawa, and Jitendra Malik. Shape and viewpoint without keypoints. In *European Conference on Computer Vision*, pages 88–104. Springer, 2020.

[82] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[83] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *SIGGRAPH*, 1996.

[84] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3d semantic segmentation with submanifold sparse convolutional networks. In *CVPR*, 2018.

[85] Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. Implicit geometric regularization for learning shapes. In *Proceedings of the 37th International Conference on Machine Learning*, pages 3789–3799, 2020.

[86] Thibault Groueix, Matthew Fisher, Vladimir G. Kim, Bryan C. Russell, and Mathieu Aubry. A papier-mâché approach to learning 3D surface generation. In *CVPR*, pages 216–224, 2018.

[87] Xianfeng Gu, Steven J Gortler, and Hugues Hoppe. Geometry images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, 2002.

[88] Haoxiang Guo, Shilin Liu, Hao Pan, Yang Liu, Xin Tong, and Baining Guo. Complexgen: Cad reconstruction by b-rep chain complex generation. *ACM Transactions on Graphics (TOG)*, 41(4):1–18, 2022.

[89] Haoyu Guo, Sida Peng, Haotong Lin, Qianqian Wang, Guofeng Zhang, Hujun Bao, and Xiaowei Zhou. Neural 3d scene reconstruction with the manhattan-world assumption. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5511–5520, 2022.

[90] Kunal Gupta and Manmohan Chandraker. Neural mesh flow: 3d manifold mesh generation via diffeomorphic flows. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, pages 1747–1758, 2020.

[91] Heli Ben Hamu, Haggai Maron, Itay Kezurer, Gal Avineri, and Yaron Lipman. Multi-chart generative surface modeling. *ACM TOG*, 2018.

[92] Christian Häne, Shubham Tulsiani, and Jitendra Malik. Hierarchical surface prediction for 3d object reconstruction. In *2017 International Conference on 3D Vision (3DV)*, pages 412–420. IEEE, 2017.

[93] Rana Hanocka, Amir Hertz, Noa Fish, Raja Giryes, Shachar Fleishman, and Daniel Cohen-Or. MeshCNN: A network with an edge. *ACM TOG*, 2019.

[94] Rana Hanocka, Gal Metzer, Raja Giryes, and Daniel Cohen-Or. Point2mesh: a self-prior for deformable meshes. *ACM Transactions on Graphics (TOG)*, 39(4):126–1, 2020.

[95] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[96] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics*, 2018.

[97] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. Baking neural radiance fields for real-time view synthesis. *ICCV*, 2021.

[98] Paul Henderson, Vagia Tsiminaki, and Christoph H Lampert. Leveraging 2d data to learn textured 3d mesh generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7498–7507, 2020.

[99] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[100] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 1991.

[101] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *NeurIPS*, 2016.

[102] Ka-Hei Hui, Ruihui Li, Jingyu Hu, and Chi-Wing Fu. Neural template: Topology-aware reconstruction and disentangled generation of 3d meshes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18572–18582, 2022.

[103] Eldar Insafutdinov, Dylan Campbell, João F Henriques, and Andrea Vedaldi. Snes: Learning probably symmetric neural surfaces from incomplete data. In *European Conference on Computer Vision (ECCV)*, pages 367–383. Springer, 2022.

[104] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017.

[105] Boyi Jiang, Juyong Zhang, Yang Hong, Jinhao Luo, Ligang Liu, and Hujun Bao. Bcnet: Learning body and cloth shape from a single image. In *European Conference on Computer Vision*, pages 18–35. Springer, 2020.

[106] Chiyu Jiang, Jingwei Huang, Andrea Tagliasacchi, and Leonidas J Guibas. Shapeflow: Learnable deformation flows among 3d shapes. *Advances in Neural Information Processing Systems*, 33:9745–9757, 2020.

[107] Chiyu Jiang, Avneesh Sud, Ameesh Makadia, Jingwei Huang, Matthias Nießner, and Thomas Funkhouser. Local implicit grid representations for 3d scenes. In *CVPR*, pages 6001–6010, 2020.

[108] Yue Jiang, Dantong Ji, Zhizhong Han, and Matthias Zwicker. Sdfdiff: Differentiable rendering of signed distance fields for 3d shape optimization. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1251–1261, 2020.

[109] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. DeepSDF: Learning continuous signed distance functions for shape representation. In *CVPR*, 2019.

[110] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of Hermite data. *ACM Transactions on graphics*, 21(3):339–346, 2002.

[111] Angjoo Kanazawa, Shubham Tulsiani, Alexei A Efros, and Jitendra Malik. Learning category-specific mesh reconstruction from image collections. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 371–386, 2018.

[112] Kacper Kania, Maciej Zieba, and Tomasz Kajdanowicz. Ucsg-net-unsupervised discovering of constructive solid geometry tree. *Advances in Neural Information Processing Systems*, 33:8776–8786, 2020.

[113] Abhishek Kar, Shubham Tulsiani, Joao Carreira, and Jitendra Malik. Category-specific object reconstruction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1966–1974, 2015.

[114] Animesh Karnewar, Tobias Ritschel, Oliver Wang, and Niloy J. Mitra. Relu fields: The little non-linearity that could. *ACM Transactions on Graphics*, 2022.

[115] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3907–3916, 2018.

[116] Yuki Kawana, Yusuke Mukuta, and Tatsuya Harada. Neural star domain as primitive representation. *Advances in Neural Information Processing Systems*, 33:7875–7886, 2020.

[117] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, 2006.

[118] Michael Kazhdan and Hugues Hoppe. Screened Poisson surface reconstruction. *ACM Transactions on Graphics*, 32(3):1–13, 2013.

[119] Petr Kellnhofer, Lars C Jebe, Andrew Jones, Ryan Spicer, Kari Pulli, and Gordon Wetzstein. Neural lumigraph rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4287–4297, 2021.

[120] Diederik P. Kingma and Jimmy Ba. Adam: a method for stochastic optimization. In *ICLR*, 2015.

[121] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature sensitive surface extraction from volume data. In *SIGGRAPH*, 2001.

[122] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. ABC: a big cad model dataset for geometric deep learning. In *CVPR*, pages 9601–9611, 2019.

[123] Ravikrishna Kolluri. Provably good moving least squares. *ACM Transactions on Algorithms (TALG)*, 4(2):1–25, 2008.

[124] Nikos Kolotouros, Georgios Pavlakos, and Kostas Daniilidis. Convolutional mesh regression for single-image human shape reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4501–4510, 2019.

[125] Georgios Kopanas, Julien Philip, Thomas Leimkühler, and George Drettakis. Point-based neural rendering with per-view optimization. In *Computer Graphics Forum*, 2021.

[126] Joseph George Lambourne, Karl Willis, Pradeep Kumar Jayaraman, Longfei Zhang, Aditya Sanghi, and Kamal Rahimi Malekshan. Reconstructing editable prismatic cad from rounded voxel models. In *SIGGRAPH Asia 2022 Conference Papers*, pages 1–9, 2022.

[127] Christoph Lassner and Michael Zollhofer. Pulsar: Efficient sphere-based neural rendering. *CVPR*, 2021.

[128] Verica Lazova, Vladimir Guzov, Kyle Olszewski, Sergey Tulyakov, and Gerard Pons-Moll. Control-nerf: Editable feature volumes for scene rendering and manipulation. *arXiv preprint arXiv:2204.10850*, 2022.

[129] Eric-Tuan Lê, Minhyuk Sung, Duygu Ceylan, Radomir Mech, Tamy Boubekeur, and Niloy J Mitra. Cpfn: Cascaded primitive fitting networks for high-resolution point clouds. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7457–7466, 2021.

[130] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. Noise2Noise: Learning image restoration without clean data. In *ICML*, pages 2965–2974, 2018.

[131] Jiabao Lei and Kui Jia. Analytic marching: an analytic meshing solution from deep implicit surface networks. In *ICML*, 2020.

[132] Marc Levoy and Pat Hanrahan. Light field rendering. *SIGGRAPH*, 1996.

[133] Thomas Lewiner, Hélio Lopes, Antônio Wilson Vieira, and Geovan Tavares. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of Graphics Tools*, 8(2):1–15, 2003.

[134] Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. Grass: Generative recursive autoencoders for shape structures. *ACM TOG*, 2017.

[135] Lingxiao Li, Minhyuk Sung, Anastasia Dubrovina, Li Yi, and Leonidas J Guibas. Supervised fitting of geometric primitives to 3d point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2652–2660, 2019.

[136] Manyi Li and Hao Zhang. D2im-net: Learning detail disentangled implicit fields from single images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10246–10255, 2021.

[137] Ruosi Li, Lu Liu, Ly Phan, Sasakthi Abeysinghe, Cindy Grimm, and Tao Ju. Polygonizing extremal surfaces with manifold guarantees. In *Proceedings of ACM Symposium on Solid and Physical Modeling*, pages 189–194, 2010.

[138] Xueting Li, Sifei Liu, Kihwan Kim, Shalini De Mello, Varun Jampani, Ming-Hsuan Yang, and Jan Kautz. Self-supervised single-view 3d reconstruction via semantic consistency. In *European Conference on Computer Vision*, pages 677–693. Springer, 2020.

[139] Yiyi Liao, Simon Donne, and Andreas Geiger. Deep marching cubes: Learning explicit surface representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2916–2925, 2018.

[140] Chen-Hsuan Lin, Chen Kong, and Simon Lucey. Learning efficient point cloud generation for dense 3d object reconstruction. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[141] Chen-Hsuan Lin, Oliver Wang, Bryan C Russell, Eli Shechtman, Vladimir G Kim, Matthew Fisher, and Simon Lucey. Photometric mesh optimization for video-aligned 3d object reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 969–978, 2019.

[142] Cheng Lin, Tingxiang Fan, Wenping Wang, and Matthias Nießner. Modeling 3d shapes by reinforcement learning. In *European Conference on Computer Vision*, pages 545–561. Springer, 2020.

[143] Cheng Lin, Changjian Li, Yuan Liu, Nenglun Chen, Yi-King Choi, and Wenping Wang. Point2Skeleton: Learning skeletal representations from point clouds. In *CVPR*, 2021.

[144] Zhi-Hao Lin, Wei-Chiu Ma, Hao-Yu Hsu, Yu-Chiang Frank Wang, and Shenlong Wang. Neurmips: Neural mixture of planar experts for view synthesis. *CVPR*, 2022.

[145] David B. Lindell, Julien N.P. Martel, and Gordon Wetzstein. Autoint: Automatic integration for fast neural rendering. *CVPR*, 2021.

[146] Gidi Littwin and Lior Wolf. Deep meta functionals for shape representation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1824–1833, 2019.

[147] Feng Liu and Xiaoming Liu. 2d gans meet unsupervised single-view 3d reconstruction. In *European Conference on Computer Vision*, pages 497–514. Springer, 2022.

[148] Hsueh-Ti Derek Liu, Vladimir G. Kim, Siddhartha Chaudhuri, Noam Aigerman, and Alec Jacobson. Neural subdivision. *ACM Transactions on Graphics*, 2020.

[149] Minghua Liu, Xiaoshuai Zhang, and Hao Su. Meshing point clouds with predicted intrinsic-extrinsic ratio guidance. In *ECCV*, pages 68–84, 2020.

[150] Shaohui Liu, Yinda Zhang, Songyou Peng, Boxin Shi, Marc Pollefeys, and Zhaopeng Cui. Dist: Rendering deep implicit signed distance function with differentiable sphere tracing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2019–2028, 2020.

[151] Shi-Lin Liu, Hao-Xiang Guo, Hao Pan, Peng-Shuai Wang, Xin Tong, and Yang Liu. Deep implicit moving least-squares functions for 3d reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1788–1797, 2021.

[152] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7708–7717, 2019.

[153] Shichen Liu, Shunsuke Saito, Weikai Chen, and Hao Li. Learning to infer implicit surfaces without 3d supervision. *Advances in Neural Information Processing Systems*, 32, 2019.

[154] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. Neural volumes: Learning dynamic renderable volumes from images. *SIGGRAPH*, 2019.

[155] Xiaoxiao Long, Cheng Lin, Peng Wang, Taku Komura, and Wenping Wang. Sparseneus: Fast generalizable neural surface reconstruction from sparse views. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXII*, pages 210–227. Springer, 2022.

[156] Adriano Lopes and Ken Brodlie. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *TVCG*, 2003.

[157] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH*, page 163–169, 1987.

[158] Yiming Luo, Zhenxing Mi, and Wenbing Tao. Deepdt: Learning geometry from delaunay triangulation for surface reconstruction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 2277–2285, 2021.

[159] Haggai Maron, Meirav Galun, Noam Aigerman, Miri Trope, Nadav Dym, Ersin Yumer, Vladimir G. Kim, and Yaron Lipman. Convolutional neural networks on surfaces via seamless toric covers. *ACM TOG*, 2017.

[160] Ricardo Martin-Brualla, Rohit Pandey, Shuoran Yang, Pavel Pidlypenskyi, Jonathan Taylor, Julien Valentin, Sameh Khamis, Philip Davidson, Anastasia Tkach, Peter Lincoln, Adarsh Kowdle, Christoph Rhemann, Dan B Goldman, Cem Keskin, Steve Seitz, Shahram Izadi, and Sean Fanello. Lookingood: Enhancing performance capture with real-time neural re-rendering. *ACM Transactions on Graphics*, 2018.

[161] Ricardo Martin-Brualla, Noha Radwan, Mehdi SM Sajjadi, Jonathan T Barron, Alexey Dosovitskiy, and Daniel Duckworth. Nerf in the wild: Neural radiance fields for unconstrained photo collections. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7210–7219, 2021.

[162] Sergey V. Matveyev. Approximation of isosurface in the marching cube: ambiguity problem. In *IEEE Visualization*, 1994.

[163] Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *CVPR*, pages 4455–4465, 2019.

[164] Zhenxing Mi, Yiming Luo, and Wenbing Tao. SSRNet: Scalable 3d surface reconstruction network. In *CVPR*, pages 970–979, 2020.

[165] Mateusz Michalkiewicz, Jhony K Pontes, Dominic Jack, Mahsa Baktashmotlagh, and Anders Eriksson. Implicit surface representations as layers in neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4743–4752, 2019.

[166] Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. Local light field fusion: Practical view synthesis with prescriptive sampling guidelines. *ACM Transactions on Graphics*, 2019.

[167] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, pages 405–421. Springer, 2020.

[168] Paritosh Mittal, Yen-Chi Cheng, Maneesh Singh, and Shubham Tulsiani. Autosdf: Shape priors for 3d completion, reconstruction and generation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 306–315, 2022.

[169] Kaichun Mo, Paul Guerrero, Li Yi, Hao Su, Peter Wonka, Niloy Mitra, and Leonidas J Guibas. Structurenet: Hierarchical graph networks for 3d shape generation. *SIGGRAPH Asia*, 2019.

[170] Kaichun Mo, Shilin Zhu, Angel X. Chang, Li Yi, Subarna Tripathi, Leonidas J. Guibas, and Hao Su. PartNet: A large-scale benchmark for fine-grained and hierarchical part-level 3D object understanding. In *CVPR*, 2019.

[171] Tom Monnier, Matthew Fisher, Alexei A. Efros, and Mathieu Aubry. Share With Thy Neighbors: Single-View Reconstruction by Cross-Instance Consistency. In *ECCV*, 2022.

[172] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Transactions on Graphics*, 2022.

[173] Jacob Munkberg, Jon Hasselgren, Tianchang Shen, Jun Gao, Wenzheng Chen, Alex Evans, Thomas Müller, and Sanja Fidler. Extracting triangular 3d models, materials, and lighting from images. In *CVPR*, 2022.

[174] Charlie Nash, Yaroslav Ganin, S. M. Ali Eslami, and Peter W. Battaglia. PolyGen: an autoregressive generative model of 3d meshes. *ICML*, 2020.

[175] Thomas Neff, Pascal Stadlbauer, Mathias Parger, Andreas Kurz, Joerg H Mueller, Chakravarty R Alla Chaitanya, Anton Kaplanyan, and Markus Steinberger. Donerf: Towards real-time rendering of compact neural radiance fields using depth oracle networks. In *Computer Graphics Forum*, 2021.

[176] Gregory M. Nielson. On marching cubes. *TVCG*, 2003.

[177] Gregory M. Nielson. Dual marching cubes. In *IEEE Visualization*, pages 489–496, 2004.

[178] Gregory M. Nielson and Bernd Hamann. The asymptotic decider: resolving the ambiguity in marching cubes. In *IEEE Visualization*, 1991.

[179] Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3504–3515, 2020.

[180] Chengjie Niu, Jun Li, and Kai Xu. Im2struct: Recovering 3d shape structure from a single rgb image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4521–4529, 2018.

[181] Michael Oechsle, Lars Mescheder, Michael Niemeyer, Thilo Strauss, and Andreas Geiger. Texture fields: Learning texture representations in function space. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4531–4540, 2019.

[182] Michael Oechsle, Songyou Peng, and Andreas Geiger. Unisurf: Unifying neural implicit surfaces and radiance fields for multi-view reconstruction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5589–5599, 2021.

[183] OpenAI. Gpt-4 technical report, 2023.

[184] Junyi Pan, Xiaoguang Han, Weikai Chen, Jiapeng Tang, and Kui Jia. Deep mesh reconstruction from single rgb images via topology modification networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9964–9973, 2019.

[185] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *CVPR*, pages 165–174, 2019.

[186] Despoina Paschalidou, Luc Van Gool, and Andreas Geiger. Learning unsupervised hierarchical part decomposition of 3d objects from a single rgb image. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1060–1070, 2020.

[187] Despoina Paschalidou, Ali Osman Ulusoy, and Andreas Geiger. Superquadrics revisited: Learning 3d shape parsing beyond cuboids. In *CVPR*, 2019.

[188] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pages 8024–8035, 2019.

[189] Mark Pauly, Niloy J. Mitra, Joachim Giesen, Markus H. Gross, and Leonidas J. Guibas. Example-based 3D scan completion. In *Symp. on Geometry Processing*, pages 23–32, 2005.

[190] Georgios Pavlakos, Luyang Zhu, Xiaowei Zhou, and Kostas Daniilidis. Learning to estimate 3d human pose and shape from a single color image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 459–468, 2018.

[191] Dario Pavllo, Graham Spinks, Thomas Hofmann, Marie-Francine Moens, and Aurelien Lucchi. Convolutional generation of textured 3d meshes. *Advances in Neural Information Processing Systems*, 33:870–882, 2020.

[192] Songyou Peng, Chiyu Jiang, Yiyi Liao, Michael Niemeyer, Marc Pollefeys, and Andreas Geiger. Shape as points: A differentiable Poisson solver. In *NeurIPS*, volume 34, 2021.

[193] Songyou Peng, Michael Niemeyer, Lars Mescheder, Marc Pollefeys, and Andreas Geiger. Convolutional occupancy networks. In *ECCV*, pages 523–540, 2020.

[194] Eric Penner and Li Zhang. Soft 3D reconstruction for view synthesis. *ACM Transactions on Graphics*, 2017.

[195] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[196] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. In *NeurIPS*, volume 30, pages 5105–5114, 2017.

[197] Marie-Julie Rakotosaona, Paul Guerrero, Noam Aigerman, Niloy J. Mitra, and Maks Ovsjanikov. Learning delaunay surface elements for mesh reconstruction. In *CVPR*, pages 22–31, 2021.

[198] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[199] Daniel Rebain, Wei Jiang, Soroosh Yazdani, Ke Li, Kwang Moo Yi, and Andrea Tagliasacchi. DeRF: Decomposed radiance fields. *CVPR*, 2021.

[200] Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. Kilonerf: Speeding up neural radiance fields with thousands of tiny mlps. In *ICCV*, 2021.

[201] Edoardo Remelli, Artem Lukoianov, Stephan R. Richter, Benoît Guillard, Timur Bagautdinov, Pierre Baque, and Pascal Fua. MeshSDF: differentiable iso-surface extraction. In *NeurIPS*, volume 33, pages 22468–22478, 2020.

[202] Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, Haiyong Jiang, Zhongang Cai, Junzhe Zhang, Liang Pan, Mingyuan Zhang, Haiyu Zhao, et al. Csg-stump: A learning friendly csg-like representation for interpretable shape parsing. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12478–12487, 2021.

[203] Daxuan Ren, Jianmin Zheng, Jianfei Cai, Jiatong Li, and Junzhe Zhang. Extrudenet: Unsupervised inverse sketch-and-extrude for shape parsing. In *European Conference on Computer Vision*, pages 482–498. Springer, 2022.

[204] Stephan R Richter and Stefan Roth. Matryoshka networks: Predicting 3d geometry via nested shape layers. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1936–1944, 2018.

[205] Gernot Riegler, Ali Osman Ulusoy, Horst Bischof, and Andreas Geiger. Octnetfusion: Learning depth fusion from data. In *2017 International Conference on 3D Vision (3DV)*, pages 57–66. IEEE, 2017.

[206] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10684–10695, 2022.

[207] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[208] Darius Rückert, Linus Franke, and Marc Stamminger. Adop: Approximate differentiable one-pixel point rendering. *arXiv preprint arXiv:2110.06635*, 2021.

[209] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. In *NeurIPS*, 2017.

[210] Shunsuke Saito, Zeng Huang, Ryota Natsume, Shigeo Morishima, Angjoo Kanazawa, and Hao Li. Pifu: Pixel-aligned implicit function for high-resolution clothed human digitization. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2304–2314, 2019.

[211] Shunsuke Saito, Tomas Simon, Jason Saragih, and Hanbyul Joo. Pifuhd: Multi-level pixel-aligned implicit function for high-resolution 3d human digitization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 84–93, 2020.

[212] Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *CVPR*, 2022.

[213] Scott Schaefer, Tao Ju, and Joe Warren. Manifold dual contouring. *IEEE TVCG*, 13(3):610–619, 2007.

[214] Scott Schaefer and Joe Warren. Dual marching cubes: primal contouring of dual grids. In *Pacific Conference on Computer Graphics and Applications (PG)*, 2004.

[215] Ruwen Schnabel, Patrick Degener, and Reinhard Klein. Completion and reconstruction with primitive shapes. *Computer Graphics Forum*, 28:503–512, 2009.

[216] John Schreiner, Carlos E. Scheidegger, and Claudio T. Silva. High-quality extraction of isosurfaces from regular and irregular grids. *TVCG*, 2006.

[217] R Schumacher. *Study for applying computer-generated images to visual simulation*, volume 69. Air Force Human Resources Laboratory, Air Force Systems Command, 1969.

[218] Gopal Sharma, Rishabh Goyal, Difan Liu, Evangelos Kalogerakis, and Subhransu Maji. Csgnet: Neural shape parser for constructive solid geometry. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5515–5523, 2018.

[219] Gopal Sharma, Difan Liu, Subhransu Maji, Evangelos Kalogerakis, Siddhartha Chaudhuri, and Radomír Měch. Parsenet: A parametric surface fitting network for 3d point clouds. In *European Conference on Computer Vision*, pages 261–276. Springer, 2020.

[220] Nicholas Sharp and Maks Ovsjanikov. Pointtrinet: Learned triangulation of 3d point sets. In *European Conference on Computer Vision*, pages 762–778. Springer, 2020.

[221] Chao-Hui Shen, Hongbo Fu, Kang Chen, and Shi-Min Hu. Structure recovery by part assembly. *ACM Transactions on Graphics*, 31(6):1–11, 2012.

[222] Tianchang Shen, Jun Gao, Kangxue Yin, Ming-Yu Liu, and Sanja Fidler. Deep marching tetrahedra: a hybrid representation for high-resolution 3d shape synthesis. *Advances in Neural Information Processing Systems*, 34:6087–6101, 2021.

[223] Yue Shi, Bingbing Ni, Jinxian Liu, Dingyi Rong, Ye Qian, and Wenjun Zhang. Geometric granularity aware pixel-to-mesh. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 13097–13106, 2021.

[224] Yawar Siddiqui, Justus Thies, Fangchang Ma, Qi Shan, Matthias Nießner, and Angela Dai. Retrievalfuse: Neural 3d scene reconstruction with a database. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12568–12577, 2021.

[225] Ayan Sinha, Jing Bai, and Karthik Ramani. Deep learning 3d shape surfaces using geometry images. In *ECCV*, 2016.

[226] Ayan Sinha, Asim Unmesh, Qixing Huang, and Karthik Ramani. Surfnet: Generating 3d shape surfaces using deep residual networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6040–6049, 2017.

[227] Vincent Sitzmann, Julien NP Martel, Alexander W. Bergman, David B. Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. In *NeurIPS*, volume 33, pages 7462–7473, 2020.

[228] Dmitriy Smirnov, Mikhail Bessmeltsev, and Justin Solomon. Learning manifold patch-based representations of man-made shapes. In *International Conference on Learning Representations*, 2020.

[229] Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view convolutional neural networks for 3d shape recognition. In *ICCV*, 2015.

[230] Cheng Sun, Min Sun, and Hwann-Tzong Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. *CVPR*, 2022.

[231] Jiaming Sun, Xi Chen, Qianqian Wang, Zhengqi Li, Hadar Averbuch-Elor, Xiaowei Zhou, and Noah Snavely. Neural 3d reconstruction in the wild. In *ACM SIGGRAPH 2022 Conference Proceedings*, pages 1–9, 2022.

[232] Qingyang Tan, Lin Gao, Yu-Kun Lai, and Shihong Xia. Variational autoencoders for deforming 3D mesh models. In *CVPR*, 2018.

[233] Jia-Heng Tang, Weikai Chen, Bo Wang, Songrun Liu, Bo Yang, Lin Gao, et al. Oct-field: Hierarchical implicit functions for 3d modeling. *Advances in Neural Information Processing Systems*, 34:12648–12660, 2021.

[234] Jiapeng Tang, Xiaoguang Han, Junyi Pan, Kui Jia, and Xin Tong. A skeleton-bridged deep learning approach for generating meshes of complex topologies from single rgb images. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition*, pages 4541–4550, 2019.

[235] Jiapeng Tang, Jiabao Lei, Dan Xu, Feiying Ma, Kui Jia, and Lei Zhang. Sa-convonet: Sign-agnostic optimization of convolutional occupancy networks. In *ICCV*, pages 6484–6493, 2021.

[236] Maxim Tatarchenko, Alexey Dosovitskiy, and Thomas Brox. Octree generating networks: Efficient convolutional architectures for high-resolution 3d outputs. In *Proceedings of the IEEE international conference on computer vision*, pages 2088–2096, 2017.

[237] Maxim Tatarchenko, Stephan R Richter, René Ranftl, Zhuwen Li, Vladlen Koltun, and Thomas Brox. What do single-view 3d reconstruction networks learn? In *CVPR*, 2019.

[238] Justus Thies, Michael Zollhöfer, and Matthias Nießner. Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics*, 2019.

[239] Shubham Tulsiani, Hao Su, Leonidas J Guibas, Alexei A Efros, and Jitendra Malik. Learning shape abstractions by assembling volumetric primitives. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2635–2643, 2017.

[240] Benjamin Ummenhofer and Vladlen Koltun. Adaptive surface reconstruction with multiscale convolutional kernels. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5651–5660, 2021.

[241] Mikaela Angelina Uy, Yen-Yu Chang, Minhyuk Sung, Purvi Goel, Joseph G Lambourne, Tolga Birdal, and Leonidas J Guibas. Point2cyl: Reverse engineering 3d objects from point clouds to extrusion cylinders. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11850–11860, 2022.

[242] Mikaela Angelina Uy, Jingwei Huang, Minhyuk Sung, Tolga Birdal, and Leonidas Guibas. Deformation-aware 3d model embedding and retrieval. In *European Conference on Computer Vision*, pages 397–413. Springer, 2020.

[243] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in python. *PeerJ*, 2014.

[244] Allen Van Gelder and Jane Wilhelms. Topological considerations in isosurface generation. *ACM Transactions on Graphics*, 1994.

[245] Gokul Varadhan, Shankar Krishnan, Young J. Kim, and Dinesh Manocha. Feature-sensitive subdivision and isosurface reconstruction. In *IEEE Visualization*, 2003.

[246] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[247] Dor Verbin, Peter Hedman, Ben Mildenhall, Todd Zickler, Jonathan T. Barron, and Pratul P. Srinivasan. Ref-NeRF: Structured view-dependent appearance for neural radiance fields. *CVPR*, 2022.

[248] Dan Wang, Xinrui Cui, Xun Chen, Zhengxia Zou, Tianyang Shi, Septimiu Salcudean, Z Jane Wang, and Rabab Ward. Multi-view 3d reconstruction with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5722–5731, 2021.

[249] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. Pixel2mesh: Generating 3d mesh models from single rgb images. In *Proceedings of the European conference on computer vision (ECCV)*, pages 52–67, 2018.

[250] Peng Wang, Lingjie Liu, Yuan Liu, Christian Theobalt, Taku Komura, and Wenping Wang. Neus: Learning neural implicit surfaces by volume rendering for multi-view reconstruction. *Advances in Neural Information Processing Systems*, 34:27171–27183, 2021.

[251] Peng-Shuai Wang, Yang Liu, Yu-Xiao Guo, Chun-Yu Sun, and Xin Tong. O-cnn: Octree-based convolutional neural networks for 3d shape analysis. *ACM Transactions On Graphics (TOG)*, 36(4):1–11, 2017.

[252] Peng-Shuai Wang, Yang Liu, and Xin Tong. Mesh denoising via cascaded normal regression. *ACM Transactions on Graphics*, 2016.

[253] Peng-Shuai Wang, Yang Liu, and Xin Tong. Dual octree graph networks for learning adaptive volumetric shape representations. *ACM Trans. Graph.*, 41(4), jul 2022.

[254] Peng-Shuai Wang, Chun-Yu Sun, Yang Liu, and Xin Tong. Adaptive o-cnn: A patch-based deep representation of 3d shapes. *ACM Transactions on Graphics*, 37(6):1–11, 2018.

[255] Weiyue Wang, Duygu Ceylan, Radomir Mech, and Ulrich Neumann. 3dn: 3d deformation network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1038–1046, 2019.

[256] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic graph cnn for learning on point clouds. *Acm Transactions On Graphics (tog)*, 38(5):1–12, 2019.

[257] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004.

[258] Chao Wen, Yinda Zhang, Zhuwen Li, and Yanwei Fu. Pixel2mesh++: Multi-view 3d mesh generation via deformation. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1042–1051, 2019.

[259] Francis Williams, Zan Gojcic, Sameh Khamis, Denis Zorin, Joan Bruna, Sanja Fidler, and Or Litany. Neural fields as learnable kernels for 3d reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18500–18510, 2022.

[260] Francis Williams, Teseo Schneider, Claudio Silva, Denis Zorin, Joan Bruna, and Daniele Panozzo. Deep geometric prior for surface reconstruction. In *CVPR*, pages 10130–10139, 2019.

[261] Francis Williams, Matthew Trager, Joan Bruna, and Denis Zorin. Neural splines: Fitting 3d surfaces with infinitely-wide neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9949–9958, 2021.

[262] Suttisak Wizadwongsa, Pakkapon Phongthawee, Jiraphon Yenphraphai, and Supasorn Suwajanakorn. Nex: Real-time view synthesis with neural basis expansion. *CVPR*, 2021.

[263] Markus Worchel, Rodrigo Diaz, Weiwen Hu, Oliver Schreer, Ingo Feldmann, and Peter Eisert. Multi-view mesh reconstruction with neural deferred shading. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6187–6197, 2022.

[264] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. Learning a probabilistic latent space of object shapes via 3d generative-adversarial modeling. In *NeurIPS*, 2016.

[265] Jiajun Wu, Chengkai Zhang, Xiuming Zhang, Zhoutong Zhang, William T Freeman, and Joshua B Tenenbaum. Learning shape priors for single-view 3d completion and reconstruction. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 646–662, 2018.

[266] Liwen Wu, Jae Yong Lee, Anand Bhattad, Yuxiong Wang, and David Forsyth. Diver: Real-time and accurate neural radiance fields with deterministic integration for volume rendering. *CVPR*, 2022.

[267] Rundi Wu, Chang Xiao, and Changxi Zheng. Deepcad: A deep generative network for computer-aided design models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6772–6782, 2021.

[268] Rundi Wu, Yixin Zhuang, Kai Xu, Hao Zhang, and Baoquan Chen. Pq-net: A generative part seq2seq network for 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 829–838, 2020.

[269] Shangzhe Wu, Christian Rupprecht, and Andrea Vedaldi. Unsupervised learning of probably symmetric deformable 3d objects from images in the wild. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1–10, 2020.

[270] Xiuchao Wu, Jiamin Xu, Zihan Zhu, Hujun Bao, Qixing Huang, James Tompkin, and Weiwei Xu. Scalable neural indoor scene rendering. *ACM Transactions on Graphics*, 2022.

[271] Zhijie Wu, Xiang Wang, Di Lin, Dani Lischinski, Daniel Cohen-Or, and Hui Huang. SAGNet: Structure-aware generative network for 3D-shape modeling. *SIGGRAPH*, 2019.

[272] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *CVPR*, 2015.

[273] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, 1986.

[274] Haozhe Xie, Hongxun Yao, Xiaoshuai Sun, Shangchen Zhou, and Shengping Zhang. Pix2vox: Context-aware 3d reconstruction from single and multi-view images. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 2690–2698, 2019.

[275] Yiheng Xie, Towaki Takikawa, Shunsuke Saito, Or Litany, Shiqin Yan, Numair Khan, Federico Tombari, James Tompkin, Vincent Sitzmann, and Srinath Sridhar. Neural fields in visual computing and beyond. In *Computer Graphics Forum*, volume 41, pages 641–676. Wiley Online Library, 2022.

[276] Qiangeng Xu, Weiyue Wang, Duygu Ceylan, Radomir Mech, and Ulrich Neumann. Disn: Deep implicit surface network for high-quality single-view 3d reconstruction. *Advances in Neural Information Processing Systems*, 32, 2019.

[277] Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. Point-nerf: Point-based neural radiance fields. *CVPR*, 2022.

[278] Yifan Xu, Tianqi Fan, Yi Yuan, and Gurprit Singh. Ladybird: Quasi-monte carlo sampling for deep implicit field based 3d reconstruction with symmetry. In *European Conference on Computer Vision*, pages 248–263. Springer, 2020.

[279] Siming Yan, Zhenpei Yang, Chongyang Ma, Haibin Huang, Etienne Vouga, and Qixing Huang. Hpnet: Deep primitive segmentation using hybrid representations. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2753–2762, 2021.

[280] Yaoqing Yang, Chen Feng, Yiru Shen, and Dong Tian. Foldingnet: point cloud auto-encoder via deep grid deformation. In *CVPR*, 2018.

[281] Yuan Yao, Nico Schertler, Enrique Rosales, Helge Rhodin, Leonid Sigal, and Alla Sheffer. Front2back: Single view 3d shape reconstruction via front to back prediction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 531–540, 2020.

[282] Lior Yariv, Jiatao Gu, Yoni Kasten, and Yaron Lipman. Volume rendering of neural implicit surfaces. *Advances in Neural Information Processing Systems*, 34:4805–4815, 2021.

[283] Lior Yariv, Yoni Kasten, Dror Moran, Meirav Galun, Matan Atzmon, Basri Ronen, and Yaron Lipman. Multiview neural surface reconstruction by disentangling geometry and appearance. *Advances in Neural Information Processing Systems*, 33:2492–2502, 2020.

[284] Jianglong Ye, Yuntao Chen, Naiyan Wang, and Xiaolong Wang. Gifs: Neural implicit function for general shape representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12829–12839, 2022.

[285] Li Yi, Vladimir G Kim, Duygu Ceylan, I Shen, Mengyan Yan, Hao Su, Cewu Lu, Qixing Huang, Alla Sheffer, Leonidas Guibas, et al. A scalable active framework for region annotation in 3D shape collections. *SIGGRAPH Asia*, 35(6), 2016.

[286] Kangxue Yin, Zhiqin Chen, Hui Huang, Daniel Cohen-Or, and Hao Zhang. LOGAN: Unpaired shape transform in latent overcomplete space. *ACM Transactions on Graphics*, 2019.

[287] Kangxue Yin, Hui Huang, Daniel Cohen-Or, and Hao Zhang. P2P-NET: Bidirectional point displacement net for shape transform. *ACM TOG*, 2018.

[288] Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. PlenOctrees for real-time rendering of neural radiance fields. In *ICCV*, 2021.

[289] Fenggen Yu, Zhiqin Chen, Manyi Li, Aditya Sanghi, Hooman Shayani, Ali Mahdavi-Amiri, and Hao Zhang. Capri-net: Learning compact cad shapes with adaptive primitive assembly. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11768–11778, 2022.

[290] Lequan Yu, Xianzhi Li, Chi-Wing Fu, Daniel Cohen-Or, and Pheng-Ann Heng. PU-Net: Point cloud upsampling network. In *CVPR*, 2018.

[291] Zehao Yu, Songyou Peng, Michael Niemeyer, Torsten Sattler, and Andreas Geiger. Monosdf: Exploring monocular geometric cues for neural implicit surface reconstruction. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[292] Jason Zhang, Gengshan Yang, Shubham Tulsiani, and Deva Ramanan. Ners: Neural reflectance surfaces for sparse-view 3d reconstruction in the wild. *Advances in Neural Information Processing Systems*, 34:29835–29847, 2021.

[293] Jingyang Zhang, Yao Yao, Shiwei Li, Tian Fang, David McKinnon, Yanghai Tsin, and Long Quan. Critical regularizations for neural surface reconstruction in the wild. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6270–6279, 2022.

[294] Jingyang Zhang, Yao Yao, and Long Quan. Learning signed distance field for multi-view surface reconstruction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6525–6534, 2021.

[295] Kai Zhang, Gernot Riegler, Noah Snavely, and Vladlen Koltun. Nerf++: Analyzing and improving neural radiance fields. *arXiv preprint arXiv:2010.07492*, 2020.

[296] Nan Zhang, Wei Hong, and Arie Kaufman. Dual contouring with topology-preserving simplification using enhanced cell representation. In *IEEE Visualization*, 2004.

[297] Richard Zhang, Phillip Isola, Alexei A. Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *CVPR*, 2018.

[298] Xiuming Zhang, Zhoutong Zhang, Chengkai Zhang, Josh Tenenbaum, Bill Freeman, and Jiajun Wu. Learning to reconstruct shapes from unseen classes. *Advances in neural information processing systems*, 31, 2018.

[299] Yuxuan Zhang, Wenzheng Chen, Huan Ling, Jun Gao, Yinan Zhang, Antonio Torralba, and Sanja Fidler. Image gans meet differentiable rendering for inverse graphics and interpretable 3d neural rendering. In *International Conference on Learning Representations*, 2020.

[300] Wenbin Zhao, Jiabao Lei, Yuxin Wen, Jianguo Zhang, and Kui Jia. Sign-agnostic implicit learning of surface self-similarities for shape modeling and reconstruction from raw point clouds. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10256–10265, 2021.

[301] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv preprint arXiv:1801.09847*, 2018.

[302] Qingnan Zhou and Alec Jacobson. Thingi10K: a dataset of 10,000 3D-printing models. *arXiv preprint arXiv:1605.04797*, 2016.

[303] Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. Stereo magnification: Learning view synthesis using multiplane images. *ACM Transactions on Graphics*, 2018.

[304] Chenyang Zhu, Kai Xu, Siddhartha Chaudhuri, Renjiao Yi, and Hao Zhang. SCORES: Shape composition with recursive substructure priors. *ACM TOG*, 2018.

[305] Chuhang Zou, Ersin Yumer, Jimei Yang, Duygu Ceylan, and Derek Hoiem. 3D-PRNN: Generating shape primitives with recurrent neural networks. In *ICCV*, 2017.