Technische Universität München
TUM School of Computation, Information and Technology

Persistent Learning for Semantic Indoor Mapping in Dynamic Environments

Maximilian Denninger

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz: Prof. Dr.-Ing. Eckehard Steinbach

Prüfer*innen der Dissertation:

1. Priv.-Doz. Dr. Rudolph Triebel
2. Prof. Dr. Bastian Leibe

Die Dissertation wurde am 28.03.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 09.11.2023 angenommen.

Abstract

Unlike current robots, future service robots are expected to exhibit artificial intelligence to act on their environment autonomously. Such autonomy can only be achieved if mobile agents are able to plan and navigate freely in indoor environments. Here a major task is mapping the environment, and a central aspect of persistent world mapping and interaction is to deeply understand the environment not only geometrically but also in its semantics. Creating 3D maps of indoor spaces has become therefore increasingly important. Such scenes provide a basis for the navigation planning of robots.

Creating these 3D scenes, however, is still an open research question, usually requiring the capture of many pictures of a scene to reconstruct all visible surfaces well. This demand for multiple views hinders the application in scenarios where a mobile robot enters a new environment and immediately requires a 3D map of the scene, as a movement might already lead to a collision. Furthermore, all the raw data a robot usually receives are depth views from depth sensors, yielding a partial reconstruction of the scene. Inspired by nature, however, which shows that successful navigation is possible without relying on depth sensors, we advocate the sole use of color images to obtain a complete 3D model of the scene beyond the visible surfaces. For this reconstruction, we propose to leverage prior information using persistently trained neural networks. As a by-product, we avoid artificial limitations of depth sensors like their limited range, varying noise modes depending on the scene and its illumination, calibration and synchronization issues, and last but not least, the extra cost or electricity demand of an added depth sensor.

Leveraging deep learning, we propose in this work an approach to tackle the problem of 3D scene reconstruction from single color images in indoor spaces. We abstain from using depth sensors to increase cost-effectiveness and reduce

resource demand. We approach this problem by first analyzing two different ways of representing 3D scenes; at first, we use a volume grid-based approach storing truncated signed distance values (TSDF) in the voxels. Secondly, we show how to implicitly encode a 3D space by querying a neural network for a position in 3D, delivering a TSDF value combined with a semantic label for this particular position. In order to map a color image to these 3D spaces, we propose to first project the visible camera frustum inside a cube, aligning it with the pixels in the input image. The projected data in the cube is a compressed and encoded latent representation of both the grid-based and implicit compression methods mentioned before, which are then used as training data. We then design a novel tree net architecture to map 2D features to 3D, allowing us to reconstruct the encoded 3D scenes, which can then be decoded into full 3D scenes. This approach enables the reconstruction of 3D scenes, including the non-visible spaces with only a single color image, avoiding the use of 3D sensors or the inconvenience of recording several images of a scene while further providing a 3D semantic segmentation that can be used in navigation and planning tasks.

Furthermore, we present BlenderProc enabling easy data generation for training vision-based deep learning methods, ranging from scene reconstruction over 6D pose estimation to semantic segmentation. BlenderProc has been warmly welcomed by the community with nearly 2k GitHub stars and is used in the 2020 & 2022 ECCV Workshop on 6D Object Pose Estimation. Additionally, we introduce SDFGen, a tool able to create training data for this challenging problem of TSDF generation of entire 3D scenes. We evaluate the two presented approaches on the real-world Replica dataset and in the wild to show that extended 3D scene reconstruction and semantic segmentation from single images are possible.

Zusammenfassung

Im Gegensatz zu heutigen Robotern verfügen künftige Serviceroboter über künstliche Intelligenz: was ihnen erlaubt autonom auf ihre Umgebung zu reagieren. Diese Autonomie kann nur erreicht werden, wenn diese mobilen Agenten in der Lage sind, in Innenräumen frei zu planen und zu navigieren. Wichtig dabei ist die persitente Kartierung der Umgebung um eine Interaktion zwischen dem Roboter und seiner Umwelt zu ermöglichen. Um dies zu erreichen, muss die Umgebung in ihrere Geometrie und Semantik verstanden werden. Die Erstellung von 3D-Karten von Innenräumen ist dabei immer wichtiger geworden. Diese karten bilden die Grundlage für die Navigationsplanung von den Robotern von morgen.

In der Forschung gibt es bei der Erstellung der 3D-Karten immer noch offene Fragen: so brauchen bisherige Methoden viele Bilder einer Szene, um alle sichtbaren Oberflächen rekonstruieren zu können. Dieser hohen Bedarf an Bildern behindert die Anwendung in verschiedenen Szenarien. So zum Beispiel benötigt ein mobiler Roboter in einer neuen Umgebung aber sofort eine 3D-Karte, da eine Bewegung unmittelbar zu einer Kollision führen kann. Viele Roboter nutzen daher Tiefensenoren, um eine partielle Rekonstruktion der Szene zu erhalten. In dieser Arbeit zeigen wir, dass - von der Natur inspiert - eine erfolgreiche 3D Rekonstrukton ohne Tiefensensoren, welche für eine Navigation eingesetzt werden könnte, möglich ist. Deswegen verwenden wir ausschließlich Farbbilder, um ein vollständiges 3D-Modell der Szene jenseits der sichtbaren Oberflächen zu erzeugen. Für diese Rekonstruktion werden neuronale Netze trainiert; dadurch werden künstliche Beschränkungen von Tiefensenoren vermieden: es entfallen die begrenzte Reichweite und die unterschiedlichen Rauschmodi je nach Szene und Beleuchtung, aber auch Kalibrierungs- und Synchronisationsprobleme. Auch die Kosten und der zusätzliche Strombedarf werden vermieden.

Der Ansatz dieser Arbeit nutzt Deep Learning, um das Problem der 3D-Szenenrekonstruktion aus einzelnen Farbbildern in Innenräumen anzugehen. Wie bereits dargestellt, wird durch den Verzicht auf Tiefensenoren die Kosteneffizienz erhöht und der Ressourcenbedarf reduziert. Dabei analysieren wir zunächst zwei verschiedene Arten der Darstellung von 3D-Szenen; zum einen verwenden wir einen auf einem Volumengitter basierenden Ansatz, bei dem truncated signed distance fields (TSDF) in den Voxeln gespeichert werden. Zu anderen zeigen wir, wie man einen 3D-Raum implizit codieren kann, indem man ein neuronales Netz an einer Position im 3D-Raum auswertet und dann einen TSDF-Wert in Kombination mit einem semantischen Bezeichnung erhält. Um ein Farbbild auf einen 3D-Raum abzubilden, schlagen wir vor, zunächst das sichtbare Kamerafrustum in einen Würfel zu projizieren und diesen dabei mit den Pixeln des Eingangsbildes auszurichten. Der Inhalt dieses projizierten Würfels, in codierter und latenter Form dargestellt, verwenden wir als Trainingsdaten, die wir durch den Einsatz der Volumengitter und impliziter Komprimierungsmethode erhalten. Anschließend entwerfen wir eine neuartige Baumnetzarchitektur, um die Merkmale von 2D auf 3D abzubilden. Damit können wir die 3D-Szene aus nur einem Farbbild, einschließlich der nicht sichtbaren Bereiche, vollständig rekonstrukieren. Die Verwendung von Tiefensensoren oder die Aufnahme mehrerer Bilder einer Szene kann dadurch vermieden werden. Zugleich wird eine semantische 3D-Segmentierung erzeugt, die für Planungs- und Navigationsaufgaben förderlich sein kann.

Abschließend stellen wir zwei weitere Möglichkeiten vor, um Daten für das Training zu generieren: BlenderProc und SDFGen. BlenderProc ermöglicht eine einfache Generierung von Daten für das Training von bildbasierten Lernmethoden. Diese Daten können in Ansätzen zu Szenenrekonstruktion über die 6D-Positionsabschätzung bis hin zur semantischen Segmentierung eingesetzt werden. BlenderProc wurde von der Community mit offenen Armen aufgenommen und hat mittlerweile rund 2.000 Github-Sterne. Es wird mittlerweile auch auf dem 2020 & 2022 ECCV-Workshop zur 6D-Objektposenschätzung eingesetzt. SDFGen ist ein Tool, das bei der Erstellung von Trainingsdaten für das anspruchsvolle Problem der TSDF-Generierung von 3D-Szenen zur Anwendung kommt. Wir evaluieren die beiden vorgestellten Ansätze auf dem realen Replica-Datensatz und in freier Wildbahn. Hier zeigen wir, dass sowohl eine 3D-Szenenrekonstruktion als auch eine semantische Segmentierung aus einzelnen Bildern möglich ist.

Acknowledgments

My time at the German Aerospace Center (DLR) uniquely shaped me as a person and as a researcher. I want to especially thank my PhD-advisor and supervisor, Rudolph Triebel, who supported me with all the crazy and outlandish ideas I had over the past five years. Providing guidance when I did not know how to proceed while providing a space to learn new things and explore the unknown. Nothing is more potent and motivating than the power of trust, and I am immensely thankful for that. Further, I want to thank Tim Bodenmüller for being my mentor for the past years, rereading my thesis several times, and always having supporting words for me when I needed them.

I also thank my colleagues, particularly Martin Sundermeyer, who discussed different 3D representations or architecture choices for hours. And Dominik Winkelbauer, who helped me develop a fast and easy way of converting complex meshes into truncated signed distance fields while joining Martin as a core developer of our open-source pipeline BlenderProc. I am both happy and lucky to have worked with such bright minds on such complex problems. I further want to thank my Deep Learning group at the DLR for discussing different architectures and losses, particularly Klaus Strobl for his endless questioning and Antonin Raffin for his dissecting views.

Finally, thank you to my Mum, Dad, and partner for your love and support through these past years. I am so grateful to have you in my life, supporting the crazy working hours and long nights pondering over this work.

November 2022,

Maximilian Denninger

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Contribution and Objectives 3
1.4 Structure of the Thesis 7
1.5 Publications 9

2 Related Work **11**
2.1 Synthetic and Real Datasets 11
2.2 Synthetic Dataset Generation 14
2.3 3D Object Reconstruction 15
2.4 3D Scene Reconstruction 16
2.5 3D Scene Compression 19
2.6 3D Scene Segmentation 20

3 Scene Representations **23**
3.1 Meshes 23
3.2 Point Clouds 25
3.3 Volumes 26
3.4 Viewport Alignment 30
3.5 Implicit 32
3.6 Semantic Information 34
3.7 Scene Representation in Robotics 35

4 Scene Compression **37**
4.1 Truncated Signed Distance Fields 37
4.2 Implicit Representations 41

4.2.1 Network Design 41
4.2.2 Loss Shaping 44
4.2.3 Compressing Millions of Blocks 47

5 Scene Reconstruction **51**
5.1 Tree Architecture 51
5.2 TSDF Volume Reconstruction 55
5.3 Implicit TSDF Reconstruction 57
5.4 Loss Shaping 58
5.4.1 TSDF Volume Grid 59
5.4.2 Tree Loss Shaping 60
5.4.3 Implicit Representation 62

6 Synthetic Data Generation **65**
6.1 BlenderProc 65
6.1.1 BlenderProc 1.X 66
6.1.2 BlenderProc 2.X 70
6.1.3 Advanced BlenderProc Features 72
6.1.4 Camera Sampling 72
6.2 SDFGen 77
6.2.1 TSDF volume 77
6.2.2 Implicit TSDF 81
6.3 Generated Datasets 83
6.3.1 External Datasets 84
6.3.2 2D Images 85
6.3.3 TSDF Scene Information 87

7 Experiments **89**
7.1 Scene Compression 89
7.1.1 Metrics 90
7.1.2 Truncated Signed Distance Fields 92
7.1.3 Implicit representations 96
7.2 Scene Reconstruction 103
7.2.1 TSDF Volume Grid Compression (SVR-GC) 103
7.2.2 Implicit Representation (SVR-IC) 110
7.3 Our Approach compared to Related Work 120
7.4 Scene Reconstruction in the Wild 124

8 Conclusion **127**

A Surface Normal Generation **131**

Bibliography **139**

List of Figures

1.1 Our mobile robot Rollin' Justin 3
1.2 A visual representation of the structure of this work 5
1.3 Overview of our entire proposed pipeline. 7

2.1 Publicly available 3D scene datasets. 13

3.1 A rendered mesh compared to its wireframe 24
3.2 Closeness in space does not mean connectedness in a mesh . 26
3.3 A point cloud of a regular textured chair 27
3.4 Different forms of TSDF volumes 28
3.5 Marching cube on TSDF volumes 29
3.6 The projection of a camera frustum into a cube is depicted here. 30
3.7 Mapping a scene into the projected cube 31
3.8 Point cloud of TSDF values around a chair 33
3.9 Ambiguity of semantic labels 34

4.1 Autoencoder to compress a TSDF volume 39
4.2 Plot of the Gaussian part used in eq. (4.1) 39
4.3 Result of our $\mathcal{L}_{\text{tsdf}}$ 40
4.4 Neural network for mapping an input point to a TSDF value 42
4.5 Combination of a latent vector with an input point 42
4.6 Our final implicit compression model 43
4.7 Plot of the exponential loss function $\mathcal{L}_{\text{surface}}$ 44
4.8 Boundary loss weights 45
4.9 Overfitting on the TSDF values 46

5.1 Simple two-branch architecture mapping a 2D image to 3D 52

5.2 A multi-branch architecture transforming 2D features into 3D 53
5.3 Our novel tree-net architecture 54
5.4 The multipath approach in the last tree layer 55
5.5 Our proposed SVR-GC method 56
5.6 Our novel SVR-IC method 58
5.7 A 2D top-down map of our scene loss shaping 59
5.8 Our novel inner tree-loss 61
5.9 The scene loss shaping for our SVR-IC method 62

6.1 BlenderProc 66
6.2 BlenderProc with the physics module 68
6.3 GitHub stars for BlenderProc 72
6.4 Camera sampling in the 3D-FRONT dataset. 74
6.5 CNN's texture focus 75
6.6 Texture randomization 75
6.7 Material properties in BlenderProc 76
6.8 Distance to a triangle 78
6.9 TSDF calculation on thin objects 80
6.10 Blocked implicit TSDF generation 82
6.11 Segmented implicit TSDF point cloud 83
6.12 Input to SVR-GC 86

7.1 Ablation on TSDF volume-based compression 95
7.2 hyperparameter optimization for the implicit compression . 98
7.3 Ablation results on the implicit compression method 101
7.4 Results for the Ablation on SVR-GC 106
7.5 SVR-GC on the SUNCG dataset 107
7.6 SVR-GC on the Replica dataset 109
7.7 Results of SVR-IC on the 3D-FRONT dataset 112
7.8 Ablation results of SVR-IC 115
7.9 SVR-IC on the Replica dataset 118
7.10 SVR-GC and SVR-IC compared to other methods 122
7.11 Fixed tilt for SVR-IC and P3DSR 123
7.12 Fixed tilt for SVR-GC and SVR-IC comparison 124
7.13 SVR-IC in the wild 125

A.1 U-Net architecture for SVR-GC 132

A.2 Four examples of our U-Net on the Replica dataset 133
A.3 U-Net architecture for SVR-IC 135
A.4 Four examples of our U-Net on the Replica dataset 137

List of Tables

2.1 Features present in open-source tools for synthetic data generation 14
2.2 Comparison of Scene Reconstruction methods 19

6.1 Comparison of the SUNCG and 3D-FRONT dataset 84
6.2 Our generated image datasets 87
6.3 Our generated 3D scene datasets 88

7.1 Volume-based results of our TSDF grid compression on SUNCG 93
7.2 Surface-based results of our TSDF grid compression on SUNCG 94
7.3 A volume-based evaluation of our implicit compression on the 3D-FRONT dataset 99
7.4 The surface-based results of our implicit compression on the 3D-FRONT dataset 100
7.5 Volume-based results of SVR-GC on the SUNCG dataset . 104
7.6 Surface-based results of SVR-GC on the SUNCG dataset . . 105
7.7 Volume-based results of SVR-GC on the Replica dataset . . 108
7.8 Surface-based results of SVR-GC on the Replica dataset . . 110
7.9 Voxel-based evaluation of SVR-IC on 3D-FRONT 113
7.10 Surface-based results of SVR-IC on 3D-FRONT 114
7.11 Volume-based results of SVR-IC on the Replica dataset . . 117
7.12 Surface-based evaluation of SVR-IC on the Replica dataset 119
7.13 Evaluation of related work to our methods 121

A.1 Surface normal results on the SUNCG and the Replica dataset. 132
A.2 Surface normal results on the 3D-FRONT dataset. 136
A.3 Surface normal results on the Replica dataset 136

Chapter one

Introduction

Understanding our surroundings is a fundamental skill of human nature. Transferring this to mobile robots is the primary objective of this work.

1.1 Motivation

Scene reconstruction and mapping of indoor spaces is a meaningful tool for mobile robots. It will enable them to operate in human homes as they can understand and memorize their environment. These environments usually follow strict and reproducible rules, as humans heavily design and structure indoor spaces [154]. This structuring extends even beyond the shape of a room to also include the systematic shape of objects. For example, if only a single image of a chair is provided, we humans can imagine the occluded parts of this chair. A robot equipped with these implicit rules, which structure our interior spaces, should be able to learn how to reproduce said objects and structures on its own. So a mobile robot that enters an entirely new space can build an internal map with as little prior knowledge as possible. Such an internal map is divided into occupied and free space. These free spaces are crucial for planning navigation paths and are necessary to fully understand the robot's environment. Such free space is separated from the

occluded space by the surfaces of the objects in the scene, meaning that the free space is everywhere where no objects occlude the space. In such a free space, a robot can freely move without colliding with its surroundings, allowing the usage of such an internal map for collision avoidance. So, it is almost impossible to plan valuable courses of action without knowing where this free space is. Most definitions of free and occluded space contain a distance measure to show for each point in space how far the closest surface is away. This information can provide a safety measure to tell the robot how fast it can move. If, for example, our robot Rollin' Justin [50] is close to the surface of a table, we will reduce its speed, while if Rollin' Justin is surrounded by free space, it can move as fast as we desire. To further improve the options that a planner in this map has, we strive to include a semantic segmentation of the environment. In this way, occluded space can be assigned to a semantic label to which humans can relate, such as a chair, table, or floor. These can help in a variety of different tasks. Such a task might be finding a particular object, for example, the remote control of a TV in a cluttered environment. It would most likely be placed near a television or a couch, so understanding the surrounding objects can drastically improve the search speed. This understanding allows Rollin' Justin to narrow down the search area and find the desired object more quickly.

1.2 Problem Statement

In fig. 1.1, our mobile robot Rollin' Justin looks onto a dining table surrounded by large dining chairs. Constructing such a complete and complex 3D scene would require dozens of different views to use multi-view scene reconstruction approaches successfully [45, 102]. We try to avoid this, as a complete 3D scene reconstruction is highly time-consuming and sometimes even impossible. Just imagine that Rollin' Justin is on a rescue mission, where it does not have the time to reconstruct the whole scene completely. Here our robot needs to quickly and reliably build up a map of the environment on only limited data. Furthermore, our robot is limited to one color image of the scene, as objects might not be static between different shots. Its task is to reconstruct a 3D environment, including the hidden and occluded free space.

Figure 1.1: Our mobile robot Rollin' Justin looks into a scene. The challenge is now converting the sensory 2D input back to a 3D scene, in which planning and navigation tasks can be performed. This reconstruction task entails reconstructing the visible and non-visible surfaces in the scene to separate free and occluded spaces.

One of the biggest challenges of this task of scene reconstruction is the underlying understanding of object geometry, which is necessary to reconstruct a scene completely. So, our approach will have to learn how semantic groups of objects look like and how they fill the free space. As humans, we can intuitively guess the shape of a table from a single image and can even fill in the hidden and occluded parts which are not visible in this one specific photograph. In order to train a robot to do the same, we have to generate an extensive dataset, which encompasses all kinds of objects and an enormous assortment of images of different combinations of those. So Rollin' Justin can understand how interior objects look and how they fill the occluded space.

1.3 Contribution and Objectives

Based on this desire to reconstruct a 3D scene based on a single color image, we derive a plan to avoid taking dozens of images of our scene and train a neural network to do this task. We employ convolutional neural networks

(CNN) [91, 54] as they have shown promising results on similar tasks like mono depth estimation or semantic segmentation of images [44, 17, 124, 135]. Our derived plan to reconstruct a 3D scene based on a single color image is divided into four main stages, depicted in fig. 1.2.

At first, we answer the question of how to represent a 2D image; this question is straightforward as the vast majority of images are represented as 2D grids. However, in 3D, no known structure provides all advantages as 2D grids do for images. Nevertheless, one can use a 3D grid for storing the environment. Such a representation still suffers from the curse of dimensionality, meaning that the memory requirements rise quickly for resolutions providing enough detail. Alternatives, such as implicit representations, point clouds, or meshes, are also evaluated.

After selecting an appropriate scene representation, one has to consider how such an entire scene might be loaded into the memory of a graphics card for training a CNN. In order to achieve this, we decided to compress a 3D scene into smaller blocks, similar to how image compression algorithms divide a picture into pieces and store them in a compressed data format [163]. Instead of relying on a known compression algorithm, we followed the idea of modern approaches using a neural network for the compression of 2D images [69, 59]. This allows much greater compression factors than possible with classical methods while ensuring that another neural network can understand our desired representation, allowing us to use these compressed scenes as reconstruction targets. Furthermore, it easily allows us to combine structural information in these created encodings with semantic information.

The third step is to design a neural network to convert a 2D image into our compressed scene representation designed in the last step. For this, we designed a novel tree architecture, which separates extracted 2D features into different 3D slices, dividing the challenging task of creating a 3D space out of a 2D image into several linked subproblems. This tree architecture is paired with a novel loss-shaping mechanism to ensure that the attention of the convolutional neural network is focused on the surface of objects in the scene. This loss shaping avoids that our method focuses on reconstructing free areas perfectly while the surface of a chair is only roughly reconstructed. Furthermore, it allows to increase the loss of more complicated objects like chairs and tables and reduces it for free or entirely occluded space.

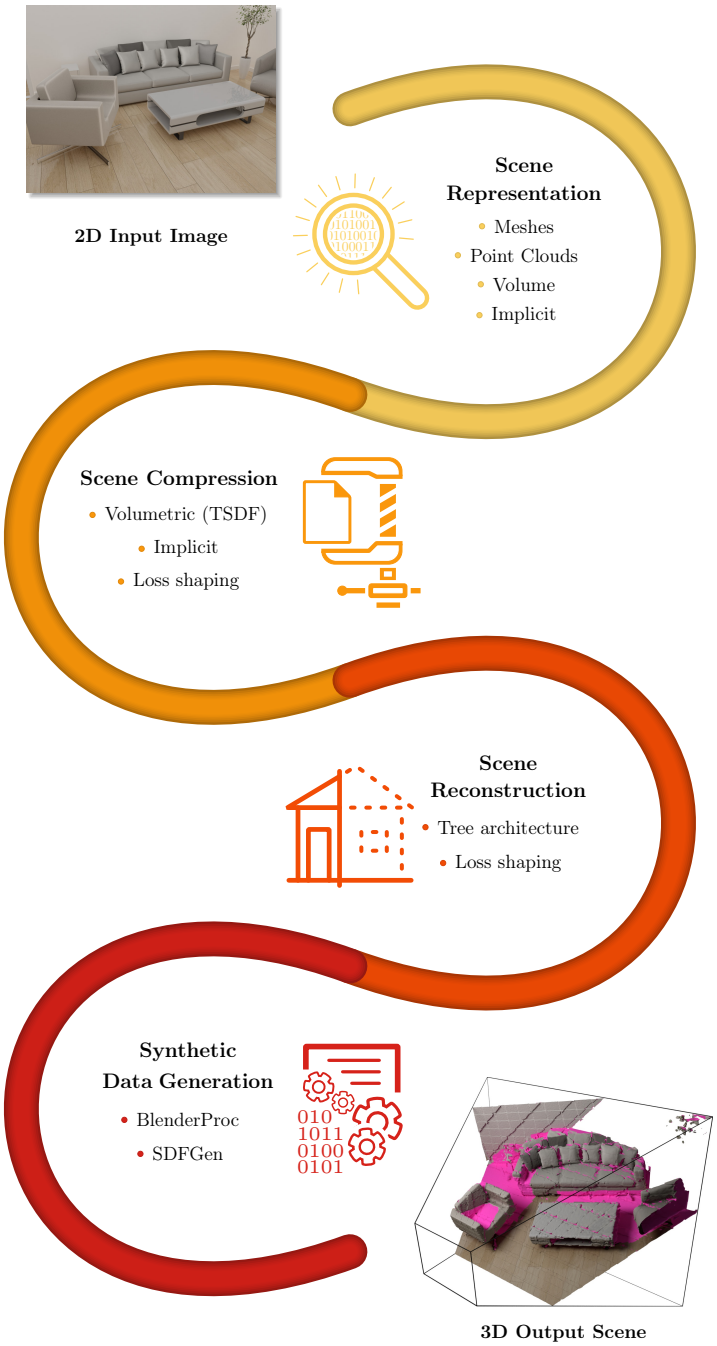


Figure 1.2: This visual representation shows the structure of this thesis. It is centered around reconstructing a 3D scene based on a color image. To achieve this, we first select a scene representation, which gets then compressed. This compressed scene is used as a reconstruction target in the scene reconstruction section, while lastly, the synthetic data generation is covered.

The final step of reconstructing a whole scene based on a 2D image is that we make sure that enough data for training the described methods exist. We decided to rely on simulated data as it does not require scanning thousands of homes and mapping them precisely, without error, assigning each object a consistent category. So, we created BlenderProc, an open-source pipeline that enables the uncomplicated generation of synthetic images of a 3D scene. Ensuring that the training of a neural network is possible, as enough image training data is available. This open-source tool now collected over 2k stars on Github and is used in the BOP challenge of the European Conference on Computer Vision (ECCV), demonstrating that synthetic data can be used to train models employed in the real world [75]. Of course, this only covers half of the required data as we are interested in reconstructing a 3D scene. Thus, the output of our network must likewise be a 3D scene, where scene here is a loose term for defining a function to decide if a given 3D point is either filled or free and additionally what the category of an occluded point would be. We created a software suite called SDFGen, which can convert a scene's mesh into a 3D representation. These desired representations can be used to train a neural network.

Our main contributions based on these four structural points plus an extensive evaluation are:

- A novel compression method for truncated signed distance fields in explicit and implicit representations
- A novel binary tree architecture that allows the transformation of 2D features into 3D
- Multiple new loss shaping techniques to ensure that compressing 3D scenes and reconstructing compressed 3D latent representations is achievable
- Designing and implementing new tools to create synthetic data for images (BlenderProc) and for truncated signed distance fields (SDFGen)

The final goal of this work is the complete reconstruction of a 3D scene in the current camera view of a single color image. This 3D reconstruction entails the scene's visible regions but also surfaces in the line of sight that are occluded by other surfaces. This detailed representation

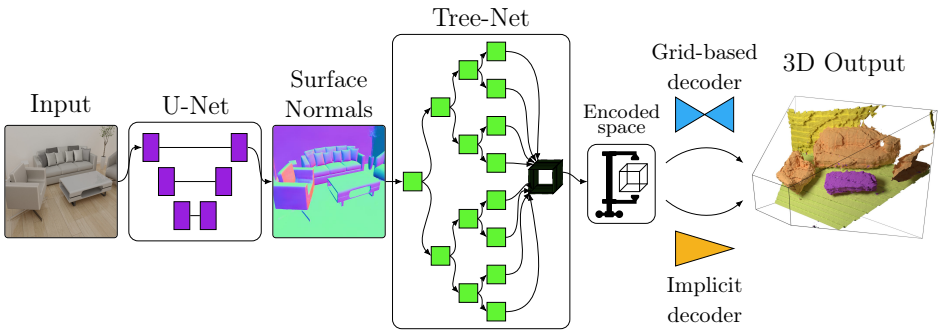


Figure 1.3: The entire proposed pipeline of our semantic 3D scene reconstruction method, transforming a color image into an occlusion map that can be visualized as mesh. The color image is first transformed into a surface normal with a simple U-Net architecture. Afterward, we rely on our novel tree-net design to transform a 2D color- and surface vector image into an encoded 3D space. We then explore two different 3D encodings, grid-based or implicitly, which enable decoding the entire scene.

enables us to determine the distance to the closest object for an arbitrary point in space. Finally, producing for each point not just the distance but also a semantic label increases our approach's usability and value. A full overview of the conversion from a single color image to our semantic 3D scene is shown in fig. 1.3.

1.4 Structure of the Thesis

This dissertation is structured following the pipeline from an input color image to a complete 3D scene reconstruction, as depicted in fig. 1.2.

Chapter 2 provides an overview of related work to this thesis by emphasizing how important datasets for the research community are and how such datasets can be synthetically generated. It also details how objects and scenes can be reconstructed and how others have tackled the problem of segmenting the scene into semantic parts.

Chapter 3 overviews possible scene representations and how they could be used in a scene reconstruction task, highlighting the advantages and disadvantages of possible mappings from the 2D to the 3D space.

Chapter 4 focuses on the compression of 3D blocks for later use as a reconstruction target, detailing how truncated signed distance fields can be compressed in a volumetric grid or in an implicit representation.

Chapter 5 provides a detailed explanation of transforming a 2D color image to a 3D compressed scene representation via a newly designed binary tree architecture design.

Chapter 6 discusses how the synthetic training data for the tasks described in the previous chapters are created while ensuring that all trained methods can be easily used in the wild.

Chapter 7 extensively evaluates the novel methods presented in previous chapters. It shows that scene compression is key to be able to increase the resolution of the 3D reconstruction and that learning the mapping from a 2D color image to an entire 3D scene is possible.

1.5 Publications

This thesis is in parts based on work that we have published in international journals and conferences, which we refer to in the respective sections. For the sake of completeness, find below a complete list of my prior publications.

Published works

- M. Denninger and R. Triebel. 3d semantic scene reconstruction from a single viewport. In *International Conference on Image Processing and Vision Engineering (IMPROVE)*, 2022 (Best Paper Award)
- M. Denninger and R. Triebel. 3d scene reconstruction from a single viewport. In *European Conference on Computer Vision*, pages 51–67. Springer, 2020
- M. Denninger, D. Winkelbauer, M. Sundermeyer, W. Boerdijk, M. Knauer, K. Strobl, M. Humt, and R. Triebel. Blenderproc2: A procedural pipeline for photorealistic rendering. In *The Journal of Open Source Software (JOSS)*, 2022
- M. Denninger, M. Sundermeyer, D. Winkelbauer, D. Olefir, T. Hodan, Y. Zidan, M. Elbadrawy, M. Knauer, H. Katam, and A. Lodhi. Blenderproc: Reducing the reality gap with photorealistic rendering. In *International Conference on Robotics: Sciene and Systems, RSS Workshop on Closing the Reality Gap in Sim2Real Transfer for Robotics*, 2020
- M. Denninger and R. Triebel. Persistent anytime learning of objects from unseen classes. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4075–4082. IEEE, 2018 (Nominated for Best Paper Award)
- M. Denninger, M. Sundermeyer, D. Winkelbauer, Y. Zidan, D. Olefir, M. Elbadrawy, A. Lodhi, and H. Katam. Blenderproc. *arXiv:1911.01911*, 2019

- J. Vogel, D. Leidner, A. Hagengruber, M. Panzirsch, B. Bauml, M. Denninger, U. Hillenbrand, L. Suchenwirth, P. Schmaus, M. Sewtz, et al. An ecosystem for heterogeneous robotic assistants in caregiving: Core functionalities and use cases. *IEEE Robotics & Automation Magazine*, 28(3):12–28, 2020

- D. Winkelbauer, M. Denninger, and R. Triebel. Learning to localize in new environments from synthetic training data. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5840–5846, 2021. doi: 10.1109/ICRA48506.2021.9560872

- M. Knauer, M. Denninger, and R. Triebel. Recall: Rehearsal-free continual learning for object classification. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022

- N. Y.-S. Lii, P. Schmaus, D. Leidner, T. Krueger, J. Grenouilleau, A. Pereira, A. Giuliano, A. S. Bauer, A. Köpken, F. Lay, M. Sewtz, N. Bechtel, S. Bustamante Gomez, M. Denninger, W. Friedl, et al. Introduction to surface avatar: the first heterogeneous robotic team to be commanded with scalable autonomy from the iss. In *Proceedings of the International Astronautical Congress, IAC*. International Astronautical Federation, IAF, 2022

Chapter two 

Related Work

Reconstructing a scene from different sensory inputs has seen an extensive amount of research over the past couple of decades. Multiple options exist to achieve this task of scene reconstruction/completion by using single or multiple color or depth images. This chapter will present an analysis of different methods used in the past to understand a scene. Further, an analysis of possible datasets and how to create new synthetic datasets from existing ones is presented, as these build the foundation of every machine learning approach.

2.1 Synthetic and Real Datasets

Datasets are at the center of any machine learning approach of the past decade [118]. Understanding the difference, especially the advantages and disadvantages of such datasets, is crucial for the final performance of any method. One of the first datasets to offer 3D meshes to the public is the Princeton shape benchmark dataset consisting of 1,814 different objects [136]. It allowed the evaluation of different methods on a standard dataset similar to what the ImageNet dataset [31] has done for the image classification

task. Other object-based datasets, such as the IKEA dataset by Lim et al., focus more on 3D indoor assets [97]. However, these datasets could not offer a wide variety of object categories. This lack of diversity changed with the introduction of the ShapeNet dataset [19], comprising of several million models, where 51,300 models have been cleaned and annotated with category labels and alignments. These three datasets contain only single 3D objects, and for a full 3D scene reconstruction, one would need to combine several objects into an entire scene. This creation has been done by Handa et al. for the SceneNet dataset, which contains 61 different scenes with various objects [64]. They even offered a tool to create new scenes automatically. However, such tools are less realistic than scenes created by human beings. For this reason, Song used the scenes created by users on the website Planner5D and their models to build up the SUNCG dataset [143]. It consists of 45,622 scenes created by humans, ensuring that each scene is plausible and realistic.

So far, all the presented datasets consist of simplified 3D models, which have been purposefully designed to reduce the memory demand, allowing them to be rendered on a modern GPU. In contrast, 3D-FRONT and 3D FUTURE contain more detailed objects, allowing the rendering of more realistic scenes [48, 49]. Even though the objects in 3D FUTURE are more detailed than those in SUNCG, less work was spent on ensuring that the textures could be easily swapped. Furthermore, are not all objects in the 3D FUTURE dataset watertight, making it more challenging to use with a broader range of tools. The datasets, as mentioned earlier, are all synthetic, while there are also real datasets that have been recorded with RGB-D sensors. One of them is the Replica dataset [145]. However, it only contains 18 different scenes, so it is too limited to be used as a training dataset. Matterport3D by Chang et al. and the S3DIS by Armeni et al. are also recorded in the wild and entail 90 and five different buildings respectively [18, 76]. While Replica is a watertight dataset, Matterport3D still contains many gaps, making it hard to use as a general scene reconstruction target. We rendered an image for six of those 3D scene datasets and showed them in fig. 2.1.

3D FRONT [48]

SUNCG [143]

SceneNet [64]

Replica [145]

Matterport3D [18]

S3DIS [76]

Figure 2.1: A rendered scene of six popular 3D datasets is shown here. The top row contains two synthetic datasets called 3D-FRONT, and SUNCG [48, 143]. The small SceneNet and the Replica dataset are depicted in the middle row [64, 145]. Below that, the Matterport3D and S3DIS dataset are shown. The last two datasets were recorded in the wild [18, 76].

2.2 Synthetic Dataset Generation

Having access to a synthetic dataset is only the first step in using it for the training of a convolutional neural network. Most available datasets must be converted into a different representation, for example, into images or TSDF representations. Rendering a dataset like ShapeNet [19] into images can be done via a simple OpenGL pipeline [11, 139]. Hinterstoisser has done this rendering in OpenGL, for instance, segmentation [71], Dosovitskiy et al. for optical flow estimation [39] and Su et al. for viewport estimation [146]. Several works have used such an OpenGL pipeline to render objects, which are then pasted on a random background image [122, 42, 153]. However, OpenGL pipelines create artifacts, which are hard to overcome when using trained approaches in the real world [138]. Similarly to that, much research has been done exploring the possibility of using a game engine to generate datasets [127, 128]. However, these rely on the assumption that a game engine emulates the natural world and is not in fact, a fast simulation that tricks humans into believing it is real [125]. Furthermore, Movshovitz-Attias et al. and Hodan et al. demonstrated that photorealistic light transport rendering, also called physically based rendering (PBR), is better than a simple OpenGL pipeline with randomized copy-paste backgrounds [111, 74, 36]. Additionally,

Table 2.1: Main features present or not present in different open-source tools for generating synthetic training data. A meaningful difference is the availability of physically based rendering (PBR), which does not allow for real-time rendering but provides more realistic data.

	NDDS	NViSII	Habitat	Stillleben	Kubric	BlenderProc
semantic segmentation	✓	✓	✓	✓	✓	✓
depth rendering	✓	✓	✓	✓	✓	✓
optical flow	⊗	⊗	✓	⊗	✓	✓
surface normals	⊗	⊗	✓	✓	⊗	✓
object pose	✓	⊗	✓	✓	⊗	✓
bounding box	✓	⊗	✓	⊗	⊗	✓
physics module	✓	⊗	✓	✓	✓	✓
camera sampling	✓	⊗	✓	✓	✓	✓
uses an open-source renderer	✓	⊗	⊗	✓	✓	✓
real-time	✓	⊗	✓	✓	⊗	⊗

Hinterstoisser et al. showed how crucial a matching in size between the foreground and background features is [72]. Li et al. has done PBR rendering for intrinsic image decomposition, and Zhang et al. used it for semantic segmentation, normal estimation, and boundary detection [94, 184]. Further, PBR rendering has been used by Hodan et al. for 6D pose estimation [74], showing that PBR rendering can be used in a variety of scenarios. With SceneNet RGB-D, McCormac et al. proposed a dataset of five million images rendered with NVIDIA's OptiX module [117], providing PBR images for semantic segmentation [105]. However, these approaches did not use a well-documented open-source pipeline, making it impossible for future researchers to use them. This changed with BlenderProc by Denninger et al. [35], one of the first open-source implementations providing sufficient documentation and examples to use the included PBR renderer for their own research goals and the research goals of many others [75, 41, 5, 15, 170, 85, 86]. BlenderProc is further detailed in section 6.1. Other open-source solutions were proposed, such as NDDS, NViSII, Stillleben, Kubric, or Habitat, not all relying on a PBR renderer [156, 110, 134, 131, 55]. For a comparison of those six, see table 2.1.

2.3 3D Object Reconstruction

Reconstructing 3D objects from raw data has a long history and is a well-discovered research area. While classical methods could only fill minor gaps and holes in 3D point clouds, more advanced methods emerged, reconstructing entire backsides of objects using prior knowledge. Such classical methods might fit local surface primitives or use continuous energy minimization [112, 144, 185]. Many such methods would use a point cloud as input and fit a local surface into missing parts. Curless et al. proposed using a signed distance field (SDF) stored in a voxel grid, allowing to represent an object in a structured format similar to images [24]. However, these classical methods would struggle to predict occluded areas of ordinary objects, such as the leg of a chair or the lower side of a table. So, approaches have been proposed to fix this by leveraging symmetries in point clouds, and meshes [155, 140]. Even though the results are impressive, they are limited to predefined hand-crafted

priors.

The use of data-driven approaches allowed for a better generalization to more scenes without defining human-crafted priors, further improving the field of 3D object reconstruction. 3D ShapeNets by Wu et al. is a method designed for classification, which can also predict the missing shape of presented objects [172]. This symbiosis improved the overall performance. One problem with using volume grids as an output is the limited resolution. Several methods have been proposed to solve this by using different octree levels, predicting with increasing precision if an octree is filled or free [152, 129, 167]. The issue is that later layers cannot correct a missed detail during the prediction. In contrast to that, Matryoshka Networks predict the missing difference to the last layer of one matryoshka block in finer detail, building up an object as the output from several blocks. Similar to the namesake of this architecture, an object is then just the combination of all different matryoshka blocks [126]. Dai et al. proposed to first predict a rough estimate with a resolution of 32 based on a point cloud scan. The finer details of this prediction get then filled in by combining it with a model from a database filled with detailed objects [27]. Fully end-to-end Han et al. train a neural network to first predict a global prediction using an LSTM to incorporate multiple depth images. This global prediction is then locally refined to ensure that the missing resolution of the LSTM does not hurt the final output [63]. Alternatively, one can find the closest CAD model out of a database and deform it into the currently visible object. A particular challenge here is that this also requires estimating the camera and object pose simultaneously [158]. Guemeli et al. even use a differentiable renderer to optimize the pose even further [57]. Nonetheless are, all of these approaches limited to a single object; even if multiple single objects in a scene can be predicted, the output does not represent the entire scene, just a selection of objects.

2.4 3D Scene Reconstruction

Reconstructing an entire 3D scene can be understood as reconstructing a room layout consisting of fixed structures, such as walls and floors and the

furniture placed. Many works have been proposed to extract only the room layout of a single photo [138, 103, 29], neglecting all interior objects in the scene. This layout and an additional simple object detection approach are now even possible on mobile phones with RoomPlan by Apple [3]. However, reconstructing a highly detailed 3D map of the current camera view is still an open research question. The approaches able to reconstruct a scene based on a single image can generally be separated into two different sets. One is more about reconstructing the scene independent of the structures present in the scene, and the other focuses on solving the subproblems in this challenge one by one. We will first look at solving the different subproblems, as done in Im2cad. Here, a room layout is built up, and then an object detection mechanism is used to find all objects. These objects are found using a bounding box detector, for which objects from a database are aligned to the image. In the final step, both are combined in a scene optimization step [81]. Huang et al. improved this by comparing a rendered image of the final reconstructed scene with the input, where the rendering also produced an object mask image, surface normal image, and a depth map [79]. Instead of relying on rendering the content, Zhang et al. use a novel implicit scene graph neural network exploiting the implicit local object features for a better 3D object pose [182]. Similarly, Total3DUnderstanding by Nie et al. [114] uses an AtlasNet [56] to reconstruct the 3D shape of the objects detected in the 3D bounding boxes, enabling the reconstruction of the scene in several steps. Mesh-RCNN also uses a two-step process for first extracting bounding boxes and then estimating the 3D shape of the objects. They, however, do not propose a way to reconstruct the room layout [53]. Kuo et al. used a CAD library with a shape embedding to find the closest matching object, making the predicted furniture pieces appear more realistic even if they do not entirely match the input image [88, 89]. When relying on a CAD library, one can go even a step further and ensure that the scene is physically plausible, ensuring that a mobile robot can be used in it, as is done by Han et al. [62].

The second set of approaches uses an image and converts it into a complete 3D map of the current view. This map is usually constructed end-to-end, allowing for a bigger corporation between the single structures in the scene. One of the first methods to do this is Voxlets by Firman et al., which relies on Random Forest to reconstruct tabletop scenes [46]. Kim et al. presented then one of the first methods relying on deep learning to convert an RGB image directly into

a truncated signed distance field (TSDF). Similarly to this, Shin et al. predict the multi-layer depth and segmentation maps, reconstructing the scene in the camera view. Enforcing consistency by using an epipolar feature transformer rendering the scene from a different view [137]. However, the goal of this work was more focused on mono-depth estimation than on reconstructing entire hidden structures. This changed with SingleViewReconstruction with a grid-based compression (SVR-GC) by Denninger et al., which uses a novel neural network shaped as a binary tree to allow the direct prediction of a compressed latent representation. This representation can be converted with a decoder to a high resolution TSDF volume, allowing the prediction of hidden and occluded spaces in the current camera view [33]. In an extension of their first work, Denninger et al. [34] improved on their first results with SVR-IC by switching the volume-based encoding with a neural network that implicitly encodes the TSDF value for queried positions. This implicit representation further allowed the inclusion of semantic labels. Instead of just predicting the shape of the surface, Dahnert et al. proposed a holistic 3D scene reconstruction approach, which predicts the instances and semantics of the present objects. This prediction is made by lifting the 2D features into a 3D space, allowing the backpropagation of the final reconstruction's loss [26]. In contrast to these methods, Worldsheet by Hu et al. tries to map one single sheet over the current camera view, allowing for the generation of unseen views with moderate pose changes [78]. An overview of a selection of these methods is shown in table 2.2.

These methods, described above, reconstruct an entire scene in contrast to mono-depth estimation. Early works relied on Markov Random Fields for the depth estimation [132, 133]. After that, one of the first approaches using deep learning in this field was by Eigen et al. [44], drastically improving the estimation performance. A consecutive work by Chakrabarti et al. estimated depth as a combination of depth derivatives of different orders [17]. The next evolution after this has been a fully convolutional neural network for mono-depth estimation [83, 90], allowing for a better flow of information. These fully convolutional neural networks then got extended to enforce geometrical constraints using conditional random fields (CRFs) [93, 99, 166, 98, 174]. Most recent works focused on depth contours as CNNs tend to smooth the edges of objects [123, 124].

A pretty recent addition to this field is the work of Mildenhall et al. called

Table 2.2: A comparison between different 3D scene reconstruction methods. We display the input and output used by the various techniques and describe if the output could be used for a navigation setting, where a TSDF value is useful. Furthermore, we highlight which methods provide a semantic or instance segmentation and if the output contains all objects present in the scene or just objects detected by a bounding box detector.

	Voxlets [46]	Total3D [114]	Panoptic [26]	SVR-GC [33]	SVR-IC [34]
Input image type	Depth	RGB	RGB	RGB	RGB
Output	TSDF grid	Mesh	TSDF grid (256^3)	TSDF grid (512^3)	implicit TSDF
TSDF availability	✓	⊗	✓	✓	✓
Semantic segmentation	⊗	✓	✓	⊗	✓
Instance segmentation	⊗	✓	✓	⊗	⊗
Full surface reconstruction	✓	⊗	⊗	✓	✓

Nerf [107], which allows the interpolation between existing camera views of a scene. This use of existing camera views requires calibrated camera poses of a set of images to allow any interpolation or extraction of a 3D model. Nonetheless, many derived works have shown the advantages of such methods for creating highly detailed 3D models of scenes [6, 20, 104, 108, 176, 147].

2.5 3D Scene Compression

Representing 3D spaces is memory intensive, and many works have been presented to tackle this problem [33, 16, 150]. While meshes are highly relevant for computer graphics, they do not work well as reconstruction targets. Volume representations, on the other hand, define for each point on a grid if it is inside or outside of an object and can be easily aligned with an input image [33]. This volume representation can be extended further to contain the truncated signed distance (TSDF) to the closest surface, which can then be converted back to a mesh with a marching cubes algorithm [100]. Such a TSDF volume is used by Denninger et al. to represent an entire scene seen from one camera viewport [33]. Liao et al. then extend this mesh generation from TSDF volumes to work in an end-to-end setting by allowing the gradient to flow through the marching cubes algorithm [95].

In contrast to strict volume grids, in an implicit representation, a network can be queried for any location in 3D for a TSDF value, as shown by Park et al. with DeepSDF [116]. Yifan et al. improved upon this concept by splitting it into the base SDF, and additional displacement for finer details [180]. Extending this further, Chen et al. use multiple residual outputs to increase the object reconstruction performance even further [21]. These implicit representations were used by Denninger et al. [34] to encode an entire scene. Tackling the computational demand of rendering objects inside of an SDF approach, Takikawa et al. proposed to use an octree combined with different levels of details of an object to shorten render times drastically [150]. Most methods struggle with reconstructing fine details on the inside of objects. This challenge was approached by Chibane using unsigned distance fields [22]. Wang et al. add an additional spline positional encoding to improve the 3D input space [168], while Chabra et al. proposed to split the scene into different parts and encode each part separately, which is similar to the work of Denninger et al. [16, 33].

Alternatively, Williams et al. presented Neural Kernel Fields, reconstructing implicit 3D shapes based on learned kernel parameters from data, which then fit the input points on the fly by solving a simple positive definite linear system [169]. Another solution was shown by Peng et al., in which they introduced a differentiable point-to-mesh layer using a differentiable formulation of Poisson Surface Reconstruction [119].

2.6 3D Scene Segmentation

Three-dimensional scene reconstruction often goes hand in hand with scene segmentation, as most approaches already need to understand object categories to reconstruct the hidden parts successfully. Song et al. proposed a method focusing on the semantic reconstruction of a depth image, producing an output volume with a resolution of $240 \times 144 \times 240$ [143]. With ScanComplete, Dai et al. presented an approach to complete a scan and segment the complete scenes. This completion and segmentation are done iteratively, decreasing the voxel size from 18.8cm^3 to 4.7cm^3 in three steps [28]. 3D-SIS

does not complete the scan but uses color information to improve the semantic segmentation results. Furthermore, it can produce instance segmentation for the objects in a scanned scene [76]. Hou et al. show with RevealNet the challenge of semantic instance completion, where they focus on completing detected instances of objects using again color information and an incomplete scan [77]. Tackling the problem of noisy outputs, Avetisyan et al. propose to rely on a CAD model database, where objects and the layout get detected in a 3D scan, and by using a graph neural network, the relationships between these detected objects can be improved [4].

Chapter three

Scene Representations

The first step in achieving a scene reconstruction based on a single image is choosing a well-suited 3D representation, as the 3D representation guides and structures the whole problem. However, representing a 3D scene is challenging as different aspects must be balanced. In some representations, for example, the surface is the centerpiece, while it is only implicitly decoded in others. These different representations have been intensely discussed in the literature over the past decades [14, 58, 164, 1]. The most relevant representations for our purposes are meshes, point clouds, volumes, and using an approximator for an implicit representation. All four will be discussed in the following chapter.

3.1 Meshes

The probably most well-known 3D data structure is a two-dimensional embedded Riemannian manifold, also called a mesh [13, 159]. It offers a boundary representation, which can be easily stored on a disc. It can also be quickly rendered into images, as most modern GPUs are designed to rasterize meshes into color images directly. We depict such a mesh in fig. 3.1, where a rendered

A regular textured object.

The underlying mesh of the chair.

Figure 3.1: The image of a chair is depicted on the left, while the underlying graph structure building up the chair's mesh is shown to the right.

mesh is depicted on the left, and the underlying wireframe is shown on the right. Such a mesh is used in many fields of application ranging from gaming to architecture visualization [13]. Each mesh here is represented by a set of vertices $\mathbb{P}$ and edges $\mathbb{E}$, which connect them. The edges $\mathbb{E}$ together form the polygons allowing meshes to represent a surface. Here, it is essential that each vertex has at least two connecting edges and that each polygon made out of a series of edges is closed. So that the start vertex is reached again after traversing all edges of this polygon. In most applications, general polygons are converted to triangles to make the processing easier [115]. This can be done quickly by repeatedly removing three adjacent vertices from the current polygon. We repeat this until no points are left in the polygon. This process then provides us with a set of triangles $\mathbb{T}$.

However, the biggest drawback of meshes is that they are hard to use as a learning target in combination with color images as an input [165]. Because there is no natural mapping between the image pixels and the vertices in the mesh. In contrast, in semantic segmentation, where each input pixel is mapped to a class, one can design architectures preserving this local structure. This preservation is done in fully convolutional architectures where each convolution uses the local neighborhood of an input pixel. While such a network still uses pooling operations to reduce the spatial dimension, it also uses skip connections to keep fine details. For meshes, however, this is impossible as it requires first compressing the entire image content to a latent representation and then building up a mesh based on the latent vector.

The difficulty here is representing a whole scene in one latent vector space. This compression might work for single objects, as shown in several works [177, 175]. Nevertheless, no work has shown that it is possible to do this sufficiently for a complete scene, as the latent space complexity gets too big. Alternatively, one could try to deform an existing ellipsoid-formed mesh into the desired shape [165]. However, this also does not generalize to complete scenes, as it assumes that the object is closed and watertight [165].

Some methods can directly work on meshes, as they can also be viewed as an undirected graph. They achieve this by encoding the position of each vertex and its relation to the neighboring connected vertices [164]. Here a convolution dynamically encapsulates the surrounding vertices and their information together in each network layer. This process is more complex than the straightforward task of applying a fixed 3×3 matrix to an input image. However, as we are trying to reconstruct meshes, we cannot rely on methods that only take meshes as input.

In conclusion, we do not rely on meshes in this work, as there is no straightforward way of mapping them from the incoming input image domain to the output mesh/graph domain. Without this transformation, we cannot find a mapping of our 2D image to a 3D scene.

3.2 Point Clouds

Like meshes, point clouds offer a representation that focuses only on the boundary of the scene objects. However, in contrast to meshes, these are not watertight and cannot be easily used for rendering or physics simulations. They contain the same information but do not offer edge information between different vertices and, therefore, no polygons. In order to make up for that, they usually have a higher point density than meshes. Using the same graph methods defined in the last section is much more challenging without these edges [12, 9, 171]. In general, the neighborhood is assumed by closeness for each point instead of relying on edges. This can lead to a wrong link, as seen in fig. 3.2. Here, the knees of a camel are so close that they would be connected if one relied on just the proximity.

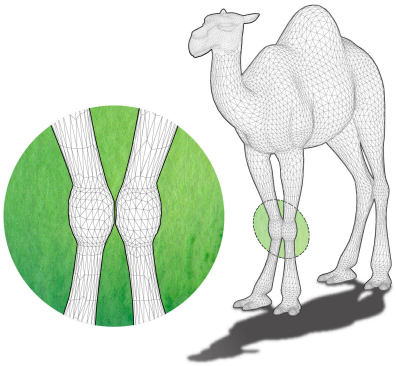


Figure 3.2: The knees of the depicted camel show that close points in a mesh do not necessarily mean that those points are connected [65].

In fig. 3.3, we show a point cloud and its corresponding mesh. Such a point cloud has the same challenges as a mesh as it is hard to use as an input for a neural network and is not easy to use as an output target. Nonetheless, point clouds have been used heavily in research, even though they have these limitations [121], especially in the case where point clouds are the input, and a classification value is the desired output [109]. However, point clouds are rarely used as reconstruction targets, except if a point cloud is an input. As there is no pixel alignment between the input point cloud and output images. This missing alignment is similar to the mesh case, as one could view the vertices of a mesh as a point cloud and drop the edges and polygons.

For the same reason as for meshes, we do not use point clouds to describe our scenes, as the formulation as a reconstruction target would require a latent compression again, limiting too much the expressiveness of our method.

3.3 Volumes

Volumes offer a dense representation of a given 3D enclosed space, similar to 2D images. Such volumes are often defined inside of a 3D cube. In this work, we assume that each volume has an internal grid, where the resulting voxels have a fixed side length. Each of them might have one of several possible states. In the simplest version, each voxel only contains a binary

A regular textured object.

The corresponding point cloud.

Figure 3.3: A point cloud of the regular textured chair on the right is depicted on the left here. The points are only distributed on the surface of the object. This distribution can be achieved by randomly sampling points on all the triangles.

label, where the states are either free or occupied [24]. In a more complex solution, one saves a truncated distance to the closest polygon in each voxel, called a truncated signed distance field (TSDF) $V : \Omega_v \rightarrow [-\sigma_{\text{tsdf}}, \ldots, \sigma_{\text{tsdf}}]$ where $\Omega_v = [0, \ldots, 511]^3$ for a side resolution of 512 [24, 113]. It is called signed as the sign of each value indicates if the space is occluded or free. Generally, we use a negative value for occluded space and a positive value for free voxels. Furthermore, these values are truncated to avoid saving the distance to a surface across the room, increasing the computation demand to generate these distances without providing more insight. We formulate the closest distance $\mathrm{d}_{\mathbf{x}}$ from a 3D point $\mathbf{x}$ to every possible triangle $\mathbb{T}$. Here, in this scenario, this point $\mathbf{x}$ is at the center of every voxel $\mathbf{v}$ in the TSDF voxel grid V. This distance $\mathrm{d}_{\mathbf{x}}$ is clipped by the truncation threshold σ_{tsdf} to be in the range of $[-\sigma_{\text{tsdf}}, \sigma_{\text{tsdf}}]$, as defined in eq. (3.1). The full definition of the truncated signed distance function can be seen in eq. (3.2).

$$\text{clip}(\mathbf{x}, \sigma_{\text{tsdf}}) = \max(-\sigma_{\text{tsdf}}, \min(\sigma_{\text{tsdf}}, \mathbf{x})) \tag{3.1}$$

$$V[\mathbf{v}] = \mathrm{d}_{\mathbf{x}} = \text{clip}\left(\min_{\forall t \in \mathbb{T}} [\mathrm{d}(\mathbf{x}, t)], \sigma_{\text{tsdf}}\right), \forall \mathbf{v} \in \Omega_v \tag{3.2}$$

These TSDF volumes can be visualized by converting the volume into a mesh by an algorithm called marching cubes, which calculates for each given voxel

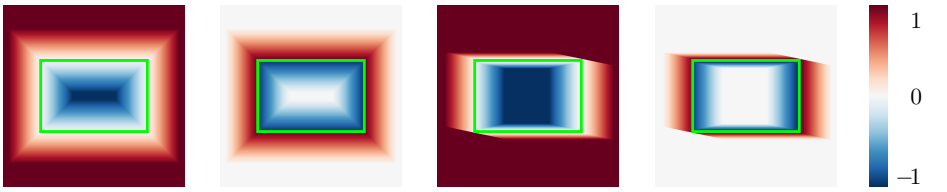


Figure 3.4: The two on the left represent a full TSDF, and the two on the right are projected TSDF volumes, using a camera that projects beams into the scene. Both also have a flipped version, where the free and occupied space is zero.

and its neighborhood one of the 27 possible resulting shapes, which are then scaled based on the given distance values [100]. We visualize this in fig. 3.5. Here the mesh of a chair is first converted into a TSDF volume and then converted to a mesh with the marching cubes algorithm. We do this for a side resolution of 128 and 512 to show possible differences in the object detail. The screws on the lower side of the chair are only distinctly visible for a resolution of 512 and vanish entirely for a comparatively lower resolution of 128.

However, not all TSDF volumes have the same structure; different approaches have used different setups. In this work, we use a full TSDF representation, meaning that each voxel contains the distance to the closest surface. This stands in contrast to a projected TSDF representation, in which the distance values are only calculated along the camera view line [24, 84]. The advantage in such a projected representation is that the TSDF calculation is much more straightforward, as one has only to calculate a distance image from the current camera viewport and repeat this process several times, removing the objects one by one. The biggest drawback in the projected representation is the big possible jump in values in a TSDF volume. These jumps are harder to learn for neural networks as moving the TSDF volume by just one voxel generates substantial differences in the loss, even though the actual change is neglectable. This substantial difference, in turn, increases the loss, focusing the weight change during training on the wrong parts. This shift inconsistency does not happen in a full TSDF representation.

One can also use a flipped TSDF representation. Here the distance value is zero for free and occluded space and gets bigger towards the surface of an object. This creates a strict boundary on the surface, where values suddenly

A regular textured object

A rendered image of a reconstructed mesh based on a TSDF volume with a resolution of 128

The same as in b but with a resolution of 512

A detailed shot of the reconstructed mesh with a resolution of 128

Similar to d but with a higher resolution of 512

Figure 3.5: The reconstructed meshes for different resolutions are presented, which can be computed with the marching cubes algorithm [100]. There is a big difference in detail on the upper part of the legs between the resolutions 128 and 512.

change from truncated max to its negative self. The biggest challenge with this representation is the difficulty of using a marching cubes algorithm afterward. All four combinations of TSDF volumes are depicted in fig. 3.4.

The most significant advantage these TSDF volumes have is that they can be efficiently mapped from a 2D image space to a 3D image space. Furthermore, they are used in planning tools as they provide for each point in space a distance to the closest polygon, which is valuable information if the main objective is to avoid collision with other objects [40]. Nonetheless, TSDF volumes have been limited successful in deep learning tasks as the desired output resolution to achieve satisfactory results is usually too limited. For example, a space with a resolution of 128 has $128^3 = 2.097.152$ output values, roughly double the amount of a $640 \times 480 \times 3$ image. If the resolution gets

even higher, up to 512, for example, we already need around 134.2 million values per space, which means that each scene needs 536 MB. In order to deal with such a high resolution, we propose to use compression. For more details, see section 4.1.

3.4 Viewport Alignment

One of the most significant advantages of our method is the possible alignment of the 2D input image with the 3D TSDF volume, by transforming our scene into the camera frame. For that, we transform vertices used for training from world coordinates $\mathbf{x}_w$ into the camera frame using the camera matrix $K_{\text{extrinsic}}$, i.e., $\mathbf{x}_s = K_{\text{extrinsic}}\mathbf{x}_w$. The camera matrix $K_{\text{extrinsic}}$ is just the inverse of the rotation and translation of the camera in world coordinates. Then, to map the camera frustum to a cubical 3D volume, a perspective projection $K_{\text{intrinsic}}$ is applied to our points $\mathbf{x}_s$ in the camera frame. The resulting projection points $\mathbf{x}_p = K_{\text{intrinsic}}$ are then in the range $[-1, 1]^3$. In eq. (3.3), the projection matrix $K_{\text{intrinsic}}$ is defined [47].

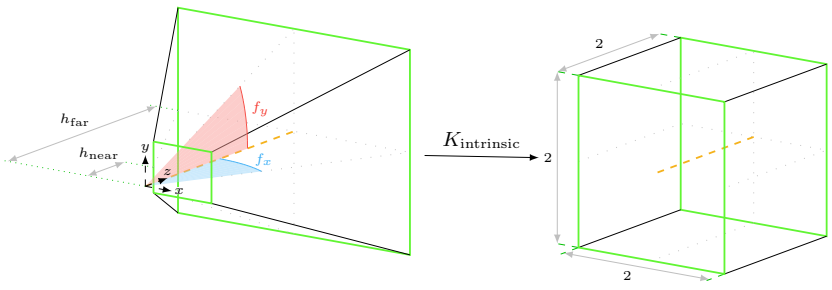


Figure 3.6: The camera frustum depicted on the left is transformed into the projected cube. For this, we position the camera at the center of the coordinate system. Here, the opening angles in x and y are shown. Additionally, the near and far clipping distance h_{near} and h_{far} are defined. This frustum gets then projected with the projection matrix $K_{\text{intrinsic}}$ into a cube with a side length of 2.

$$K_{\text{intrinsic}} = \begin{bmatrix} \frac{1}{\tan(f_x)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(f_y)} & 0 & 0 \\ 0 & 0 & \frac{h_{\text{near}} + h_{\text{far}}}{h_{\text{near}} - h_{\text{far}}} & \frac{2 \cdot h_{\text{near}} \cdot h_{\text{far}}}{h_{\text{near}} - h_{\text{far}}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{3.3}$$

It uses the horizontal open-angle f_x and vertical open-angle f_y of the camera. Additionally, we use a near h_{near} and far h_{far} clipping plane. These parameters are also visually depicted on the left side in fig. 3.6. This projection matrix allows a direct mapping between the input pixels in an image with the voxels in our projected cube, as each camera ray starts from the near-clipping plane and goes to the far-clipping plane. So each pixel can be mapped to an axis-aligned line in the projected cube. This alignment is crucial as it allows a convolutional neural network to rely on its local feature extraction. Without it, it would be necessary to compress the entire scene into one latent representation. However, one drawback of this projection is that objects further away from the camera are more compressed in the depth dimension than objects closer to the camera. This can be seen in fig. 3.7, where the

A scene inside of a camera frustum before the perspective projection $K_{\text{intrinsic}}$ is applied.

After the application of $K_{\text{intrinsic}}$, the scene is inside a cube with a side length of 2.

Figure 3.7: An application of the projection matrix $K_{\text{intrinsic}}$ on an exemplary scene from the 3D-FRONT dataset is shown. Transforming each vertice into the projected cube with a side length of 2. The compression along the camera's viewing direction is stronger for objects further back than for objects closer to the camera.

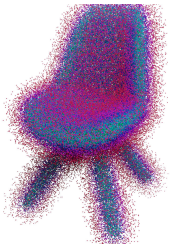
projection matrix $K_{\text{intrinsic}}$ is applied on a selected camera frustum of our synthetic scenes and maps the content into a projected cube. The objects closer to the h_{far} are compressed further in the camera view direction than, for example, the white chair in the front of the scene. A possible approach to abate this is discussed in section 6.2.2.

3.5 Implicit

In an implicit representation, a network architecture saves the object in the weights. The network can then be evaluated for any given 3D coordinate $\mathbf{x}$, resulting in a signed distance value for that position. This means that, in contrast to all prior methods, it does not have an explicit representation, relying on specific positions in 3D space to form the outer hull. Without this discrete representation, any object could be modeled with arbitrary detail, as it combines the advantages of point clouds with the more implicit representation of surfaces in volumes. These distances are valid for any point in the current query volume, making it more similar to TSDF volumes than to point clouds or meshes, as these two only represent the surface itself and not the corresponding volume.

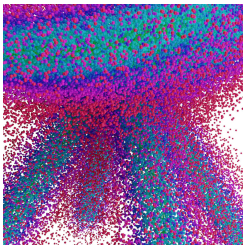
There are two options to visualize this implicit representation. In figure fig. 3.8, both are depicted. In the middle of the top row, a colored point cloud is used to visualize the TSDF distance to the surface. These points can be used to train a neural network, which is done for the right image in the top row. In order to create this mesh, it is necessary to evaluate the trained neural network millions of times, which is much more demanding than rendering a mesh on a modern graphics card. There are several options to do this. The first one uses the TSDF volume from section 3.3 by evaluating each grid position. Depending on the chosen resolution, it might require millions of evaluations before the volume can be transformed into a mesh with a marching cube algorithm. Secondly, it is also possible to use a ray-casting approach, in which a ray is cast through the scene for each pixel in an image. This ray starts at the camera's location and evaluates the neural network for this position. We then add the current maximum signed distance to our

(a) A regular textured chair rendered with blender



(b) A TSDF distance colored point cloud of the chair

(c) A mesh reconstructed from a trained neural network



(d) A close up of (b)

(e) A close up of (c)

Figure 3.8: Two different visualizations for implicit representations are shown here, as an implicit representation requires the querying of a neural network, we show the training data and a possible result here. We show the original mesh on the left, while a colored TSDF point cloud is depicted to its right. The colors represent the TSDF distance to the surface of the mesh. Additionally, a neural network was trained with this point cloud, and during inference, a TSDF volume is filled with its predictions. The TSDF volume can then be visualized with a marching cubes algorithm.

ray and calculate the TSDF distance for the resulting point. This process is repeated until the signed distance value is smaller than the threshold value, indicating that the surface is reached. However, in our tests, the calculations for one ray took too long to create a real-time rendering application and we had to resort to the marching cubes algorithm for the visualization.

As mentioned in section 3.4, the alignment with the input image is a crucial element of this work. We first use the same camera transformation $K_{\text{extrinsic}}$ and projection matrix $K_{\text{intrinsic}}$ on the point cloud. We then voxelize the point cloud shown in fig. 3.8 with a fixed resolution. This voxelization enables us to align the input image with our voxelized point cloud blocks. Furthermore, as we need to represent this point cloud block in a latent representation to use it as a reconstruction target later, we can directly and efficiently integrate

additional information like semantic categories for each point in this latent representation.

3.6 Semantic Information

Another representation dimension independent of the scene's general structure is a human interpretable label. This can be done by assigning semantic labels to points in our space, enabling us to tell different objects apart based on their semantic label. This enhances the value of our prediction, as a downstream task can use this semantic information. Such a semantic label would be represented as a word. However, such words are human-made and only roughly describe an object's nature. They do not have precise definitions. So, objects might lie on the boundary between different categories. Multiple good examples highlight how complex this problem truly is: the difference between

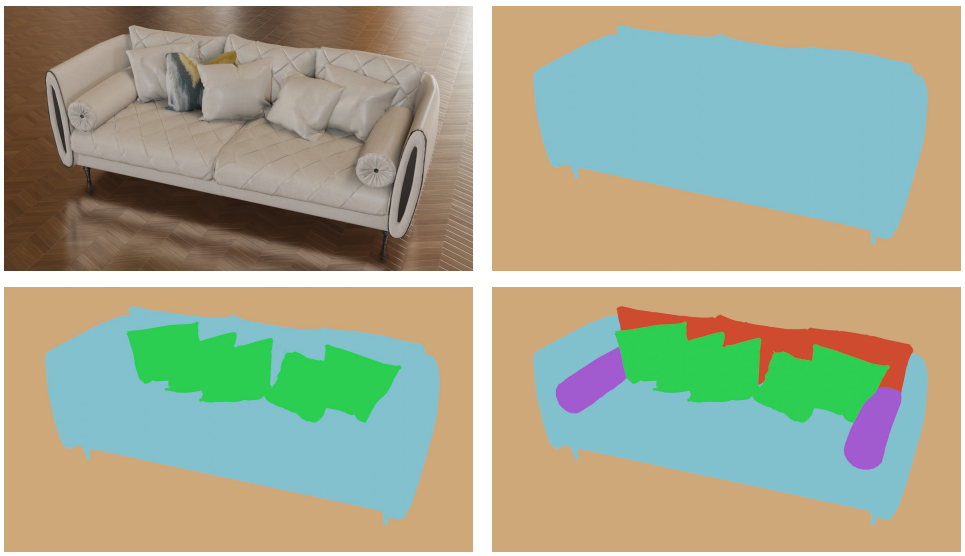


Figure 3.9: Possible configurations for a pillow class for one particular couch are shown here. In the top left, a rendered image created with blender is shown, and surrounding it, different possible segmentations are presented. These segmentations illustrate how the pillow class can be interpreted differently, leading to inconsistent data and, ultimately, wrong predictions in the trained model. The model is from the 3D-FRONT dataset [48, 49].

a desk and a table might only be defined by the surrounding objects, not the object itself. Similarly, the pillow class is hard to define as some pillows might be part of the couch while others are placed on top for decoration. This confusion of the pillow class can be seen in fig. 3.9.

Tackling this ambiguity is one of the biggest challenges of semantic segmentation. We are now presented with two options, as highlighted in the last sections. These are voxel grids and implicit representations, as they have advantages over other representations in our use case. Therefore, we have to choose if we incorporate the semantic information into our voxel grids or in our implicit representation. While theoretically possible to also predict a category class for each voxel, it quickly poses a problem to the memory demand of such a system. The reason is that each block would need not just to predict one TSDF value but multiple additional values depending on the number of chosen categories. This increase would drastically reduce the possible output resolution, so we did not pursue this direction. In contrast, in an implicit representation adding semantic information is a simple matter of adding a category for each point during the prediction step of the compressed latent vector. For more details on this see section 4.2.1.

3.7 Scene Representation in Robotics

Each application has different objectives in the extensive field of 3D scene reconstruction with various methods ranging from SLAM to the methods described in this work. As we target a 3D scene reconstruction, which can be applied to future mobile robots, we design our approach with this specific set of objectives in mind. Our 3D scene reconstruction could be used for planning a route, in which the distance to the surfaces could be used as guidance for the safety system. In such a scenario, detecting all occluded areas in the scene is more important, as missing one might lead to a collision, damaging our robot. However, if our method by accident predicts free space as occluded, in the worst case, we are unable to move at all, which is more desirable than a damaged robot. This means that when evaluating our method on the precision and recall. We will pay more attention to a higher recall, as

missing objects, as detailed above, is a much more crucial feature than filling free space by accident, measured by the precision.

Chapter four

Scene Compression

> *You're trying to take something that can be described in many, many sentences and pages of prose, but you can convert it into a couple lines of poetry and you still get the essence, so it's that compression. The best code is poetry.*
>
> — **Satya Nadella**
> CEO Microsoft

The overarching goal of this work is to achieve highly detailed scene reconstructions from single images. As we discussed the advantages and disadvantages of possible scene representations in chapter 3, we focus only on volume grids and implicit representations in this chapter.

4.1 Truncated Signed Distance Fields

In order to achieve high-resolution Truncated Signed Distance Fields (TSDF), we propose to compress these volumes. This compression is necessary as

prior research only works on low-resolution TSDF volumes, reaching only a resolution of 32 to 128 [177, 173]. In contrast to these works, we reach a resolution of 512, resulting in $512^3 = 134,217,728$ voxels; this is 512 times more per scene than for $64^3 = 262,144$. We have to rely on this compression, as each volume would need roughly 536.87 MB. Training with such big targets is impossible on modern hardware as just the last few layers would already need more memory than most GPUs offer. Our plan is, therefore, the compression of a 3D scene as a latent representation, which is then used by a second network as a reconstruction target. This network then converts a 2D image into this compressed latent representation. Furthermore, as this compression is built dynamically during the training of the compression network, we can emphasize the shape of our latent space through loss shaping. Like image compression algorithms, we break the scene into blocks, and each block gets compressed into a latent vector. So, we have a grid of unique latent vectors for a complex scene, where the surface is the most prominent part of the latent vector information. We focus on the surface while reconstructing the latent values by increasing the loss for the voxels close to the surface with a novel loss function.

We propose compressing this huge TSDF volume from a resolution of 512^3 to a size of 64×32^3, achieving a compression factor of 64. Then each scene is only roughly 8.38 MB big, making them much easier to be used as a reconstruction target. As we compress in a block-wise fashion, each block of size 32^3 is compressed to a vector of length 64. For the compression of these blocks, we use an autoencoder.

The first step is to increase the input spatial resolution from 16^3 to 30^3 by including more of the surrounding area. This increased input size ensures that the boundaries of our output are better reconstructed as it understands what lies behind the sharp boundary of our block. The block's input resolution is determined by our four 3D Convolutions, all performed with no padding. So, the spatial resolution is reduced by two for every step as we use a filter size of 3. Together with the pooling, we end up with 64 values for our $30^3 = 27,000$ input values. We use four 3D Convolutions again in combination with transposed 3D Convolutions, increasing the spatial dimension back to 16^3. We then iterate over the whole volume and compress each block to a 64 long vector. The resulting vectors are stacked again to form a 3D volume.

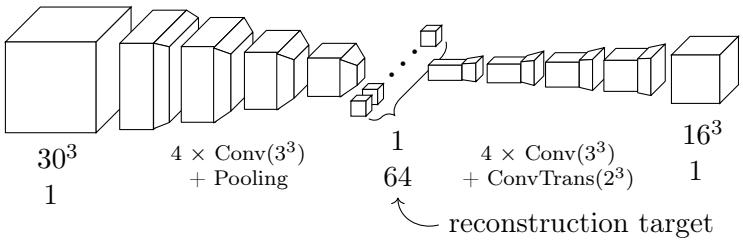


Figure 4.1: Approximated structure of the autoencoder to compress TSDF volumes, which is applied on each 16^3 block plus padding (30^3) on the 512^3 input space. The result is the encoded latent vector with 64 values in the middle. With the decoder on the right, we can reconstruct the scene based on the latent values.

As mentioned in section 3.3, we rely on a complete TSDF volume approach. This means that there are no sharp jumps in our TSDF volume, making it easier for the autoencoder, as the output has a natural smoothness, which only stops at the truncation threshold σ_{tsdf}. However, to circumvent that the autoencoder has to reconstruct the σ_{tsdf} value perfectly, we introduce a clipping of the output values after the last layer $o_{\text{last layer}}$, which does not have an activation function. This clipping is already defined in eq. (3.1) and returns the final output o. By clipping, we avoid that a value above the targeted σ_{tsdf} introduces a significant loss and distracts the network of our desired task of reconstructing the values around the surface well.

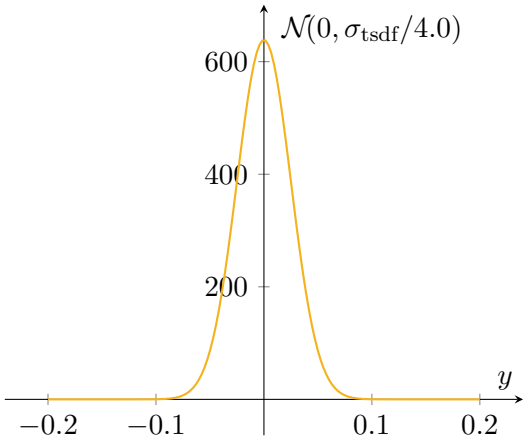
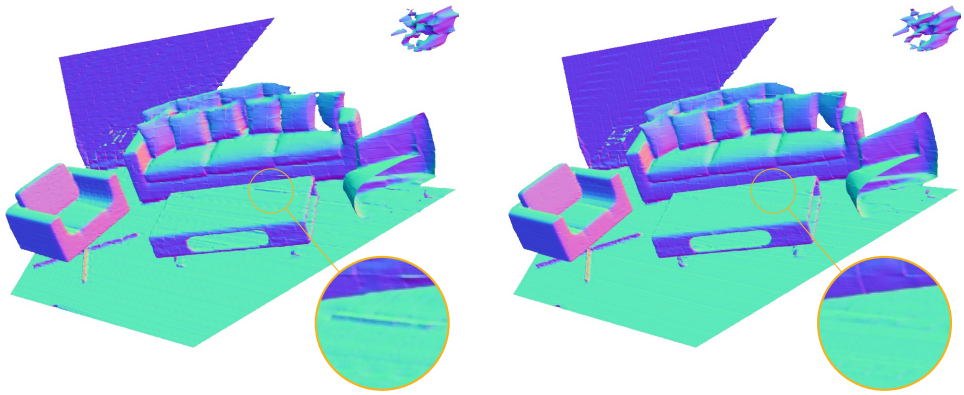


Figure 4.2: The Gaussian part of eq. (4.1) is depicted, for this visualization, we set σ_{tsdf} to 0.1. This function steadily increases all loss values in proximity to the boundary.

As the mean over the $16^3 = 4096$ values does not necessarily indicate how well the surface is reconstructable with a marching cubes algorithm [100], we employ a loss shaping on the surface boundaries to focus the attention of the network. By increasing the loss for values close to the surface of our objects, we can improve the final surface reconstruction. Here, we rely on a Gaussian function to strengthen the loss smoothly in relation to the distance to the surface. In eq. (4.1), we define our used loss $\mathcal{L}_{\text{tsdf}}$ for this reconstruction. Here σ_{tsdf} is the threshold at which we clip our TSDF values. In our tests, we use a value of 0.1, roughly 5% percent, as our world after the projection spans from $[-1, -1, -1]$ to $[1, 1, 1]$.

$$\mathcal{L}_{\text{tsdf}}(o, y) = \|o - y\|_1 \cdot \left(1 + \mathcal{N}\left(0, \frac{\sigma_{\text{tsdf}}}{4}\right)(y) \cdot \frac{4}{\sigma_{\text{tsdf}}}\right) \tag{4.1}$$

The variance of our Gaussian of $\sigma_{\text{tsdf}}/4$ was found experimentally. The same experiments also favored a scaling with $4/\sigma_{\text{tsdf}}$ for the Gaussian. This loss function is plotted in fig. 4.2. In fig. 4.3, we display the surface normals of a compressed and then decompressed scene to show the benefits of the Gaussian loss $\mathcal{L}_{\text{tsdf}}$. Without our loss shaping, the scene compression introduces surface



No surface loss is employed in this scene compression result.

The surface loss $\mathcal{L}_{\text{tsdf}}$ is used in to train the encoder.

Figure 4.3: The 3D scene presented in the overview figure, see see fig. 1.2, is blockwise compressed and decompressed. We display here the surface normals of this resulting decompressed scene, on the left, without the newly introduced Gaussian loss, and on the right, with the surface loss $\mathcal{L}_{\text{tsdf}}$. Without this loss, visual artifacts are visible on the surfaces in the scene. Any minor error in these scene compression results will reduce the final accuracy of our method.

noise, which will hurt the final performance of the reconstruction. This scene is the same as in fig. 1.2.

4.2 Implicit Representations

In contrast to section 4.1, we highlight an alternative to a fixed grid structure by relying here on an implicit representation of our scene [179, 16]. This means that instead of a 3D grid, we can query our neural network for each position p in the scene and retrieve the calculated TSDF value. This enables us to define our resolution depending on the current task and is only limited by two things. First, by the time we want to spend on querying the network, and second, the capacity of the network to represent the details of the surface.

4.2.1 Network Design

For this, we design a simple, fully connected neural network, which gets as input the current query position p and outputs a TSDF value. At first, the input is transformed with a Fourier transformation from three dimensions into 128, which is inspired by Tancik et al. [151]. To understand this, we split this process into two steps. First, we use a basic mapping to transform our three elements into a Fourier transformed space, with the eq. (4.2). However, this only produces six outputs, three for the cosine and three for the sinus. We can increase this by using a Gaussian Fourier feature mapping by defining a matrix $\mathbf{B} \in \mathcal{R}^{m \times 3}$. Here m is the desired size of our Fourier transformation. In our experiments, we use 64 as m to achieve 128 final elements. The values of $\mathbf{B}$ are sampled from a normal distribution $\mathcal{N}(0, \sigma_{\text{Fourier}})$. The resulting mapping can be seen in eq. (4.3).

$$F_{basic}(x) = [\cos(2\pi x),\ \sin(2\pi x)]^T \tag{4.2}$$

$$F(x) = [\cos(2\pi \mathbf{B} x),\ \sin(2\pi \mathbf{B} x)]^T \tag{4.3}$$

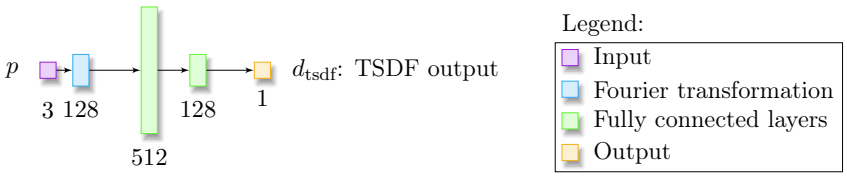


Figure 4.4: The shown network compresses one point p into an implicit TSDF representation. First, the point gets mapped with a Fourier Transformation followed by three layers of a fully connected network, producing the TSDF output. This network itself can only learn one scene representation at a time.

In fig. 4.4, we show our designed architecture. After we transform our p from 3 dimensions into 128 with the Fourier transform, we add two fully connected layers of size 512 and size 128. Both layers use as an activation function a ReLU. Finally, we go down to a size of one and use no activation function. With this, we can train our network for a given scene. However, it would mean that we need one network per scene. To avoid this, we add a latent vector at the beginning, which encodes the scene inside our 512-long latent vector l. We concatenate this latent vector with our Fourier-transformed input and feed it to the first layer, as shown in fig. 4.5.

We train this network in a two-step process. At first, the latent vector l is initialized with zeros. We then optimize only the latent vector for a set of points with their corresponding TSDF values. This optimization is done for a few hundred steps, after which we freeze the latent vector l and perform the second step, in which we train the network weights with the same points, corresponding TSDF values, and newly generated latent vector l. These two steps are now repeated until the network converges. As highlighted in section 3.6, our goal with this architecture is to additionally predict semantic

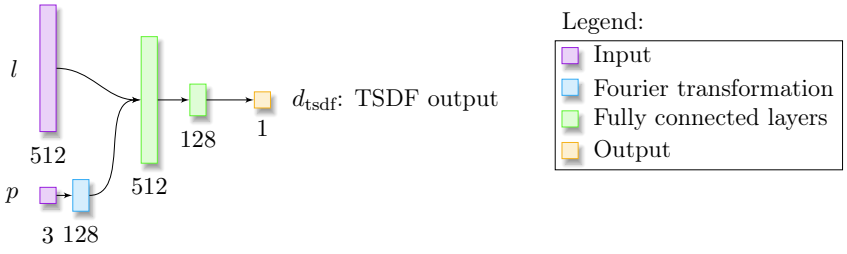


Figure 4.5: This model has an additional latent vector l as an input to our network defined in fig. 4.4. Allowing us to save the scene representation inside the latent vector and not exclusively inside the weights of the network itself.

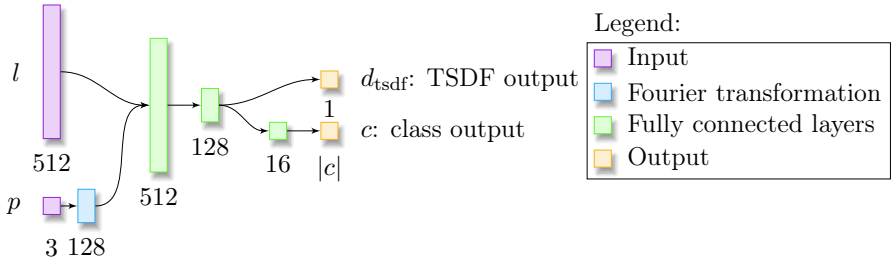


Figure 4.6: Our final compression network can convert a given point and latent vector to its TSDF value and semantic label represented as a probability distribution over several categories.

segmentation. So, we added another head to our network by adding another hidden layer just for the class output c. This layer has a size of 16 and connects with the class output c, which has as many elements as we have classes. Our final architecture can be seen in fig. 4.6.

This newly designed network allows us to compress scenes into a single latent vector. So the next step is to voxelize an entire scene into different blocks and compress each block. This voxelization has several reasons; for one, our designed network cannot represent an entire scene inside our 512 long latent vector l, as this would remove any possible mapping between the input image domain and the output reconstruction domain. This mapping is crucial to ensure that the information can flow in a localized way through the network without additionally learning to propagate the information to the correct positions. Additionally, using only one latent vector per scene would mean that our reconstruction network must map everything down into one latent vector. Breaking it up into several blocks makes it easier for the scene compression network as the blocks contain fewer details and can be seen as building blocks of our scenes. The final reconstruction network mapping an image to this result can also exploit this as it only has to predict a specific block for a particular location. So, our scene is divided with a side resolution of 16, giving us $16^3 = 4096$ single blocks. The points in each block are scaled to the interval between -1 and 1.

4.2.2 Loss Shaping

To guarantee that the surface in each block is well reconstructed, we use a similar loss shaping technique as in section 4.1. So, we first define the difference between the TSDF output value d_{tsdf} and the TSDF target value y_{tsdf} as $\mathcal{L}_{\text{dist}}$ in eq. (4.4). The error is defined as an absolute loss, which minimizes the difference between our predicted value d_{tsdf} and our target value y_{tsdf}.

$$\mathcal{L}_{\text{dist}} = |y_{\text{tsdf}} - d_{\text{tsdf}}| \tag{4.4}$$

The Gaussian distribution presented in section 4.1 has one drawback, through its wide range of values around zero, which have equally high loss strength values, the attention is not focused too sharply on the boundary. We fix this by using a sharper distribution defined in eq. (4.6). Here, the focus lies on the higher values close to zero, increasing the focus more on the object's surface rather than values just close to the surface. This focus is rather beneficial as we still try to have a high resolution, meaning that the critical points used in the marching cubes algorithm have a small distance to the surface.

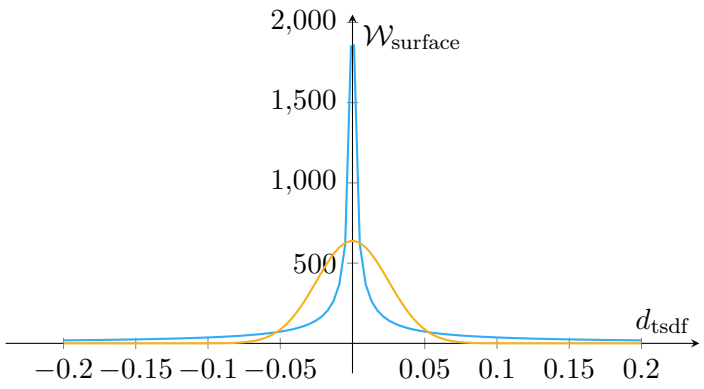


Figure 4.7: This graph represents in blue the $\mathcal{W}_{\text{surface}}$ of eq. (4.5), we use a θ_{surface} of 37.27 and a ϵ of 0.001. We also depict the Gaussian distribution in yellow from fig. 4.2.

$$\mathcal{W}_{\text{surface}} = \frac{\theta_{\text{surface}} \cdot \sigma_{\text{tsdf}}}{|y_{\text{tsdf}}| + \epsilon} \tag{4.5}$$

$$\mathcal{L}_{\text{surface}} = \mathcal{L}_{\text{dist}} \cdot \mathcal{W}_{\text{surface}} \tag{4.6}$$

We compare the Gaussian distribution defined in eq. (4.1) to our new distribution in fig. 4.7. The Gaussian distribution is depicted in yellow, and the new distribution is in blue.

This voxelize process creates sharp boundaries between the different voxels. To guarantee that these boundaries of each block are reconstructed well, we change two things. First, we increase the input range for each block by incorporating points from neighboring blocks. So, we increase the size of each block by a factor of b. This increases the side length of each block in our case for $b = 1.1$ from 2 to 2.2. These points give the network an indication of what happens beyond the current block's boundary and ensure that the final reconstruction has smooth transitions. On top of that, we introduce a boundary shaping loss to ensure further that the accuracy of the surface reconstruction at the boundary is higher than without it. This loss also helps in the final scene reconstruction as it increases the gradient on these points and, with that, also increases the corresponding gradient relative to the latent vector l, raising its magnitude during training. Moreover, higher latent values during our scene reconstruction mean a higher loss and an additional focus on the boundaries ensures a smooth transition between blocks. To construct this boundary loss $\mathcal{L}_{\text{boundary}}$, we start by calculating the closest distance $dist_{\text{boundary}}(p)$ for each point p to one of the six sides of our boundary cube. The maximum absolute value of our cube is b. This can be seen in eq. (4.7). Our $dist_{\text{boundary}}(p)$ is now inverted by subtracting it from one. After that,

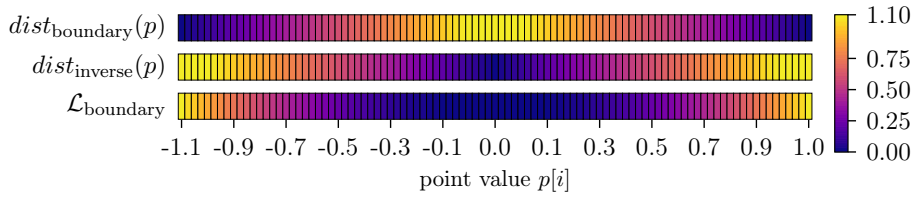


Figure 4.8: The different weights for the three parts defined in eqs. (4.7) to (4.9) are shown here, where we use a boundary size b of 1.1.

we square it to sharpen the loss around the boundary even more, which can be seen in eq. (4.8). The final value then scales the $\mathcal{L}_{\text{dist}}$ in eq. (4.9). The change of these values can be seen in fig. 4.8.

$$dist_{\text{boundary}}(p) = \min\left(b + \min_{i\in[1,2,3]} p[i],\ b - \max_{i\in[1,2,3]} p[i]\right) \tag{4.7}$$

$$dist_{\text{inverse}}(p) = \left(1 - dist_{\text{boundary}}(p)\right)^2 \tag{4.8}$$

$$\mathcal{L}_{\text{boundary}} = \mathcal{L}_{\text{dist}} \cdot dist_{\text{inverse}}(p) \tag{4.9}$$

Lastly, we need to add the category loss $\mathcal{L}_{\text{category}}$ to predict the class correctly.

$$\mathcal{L}_{\text{category}} = \sum_{i=1}^{\|\mathbb{C}\|} y_c[i] \cdot \log(c[i]) \tag{4.10}$$

This category loss gets then combined with the others in:

$$\mathcal{L}_{\text{combined}} = \mathcal{L}_{\text{dist}} + \mathcal{L}_{\text{surface}} + \mathcal{L}_{\text{boundary}} + \theta_{\text{category}} \cdot \mathcal{L}_{\text{category}} \tag{4.11}$$

Here the θ_{category} is set after extensive tests to 30.32.

One last challenge remains, our network could overfit on the TSDF values as

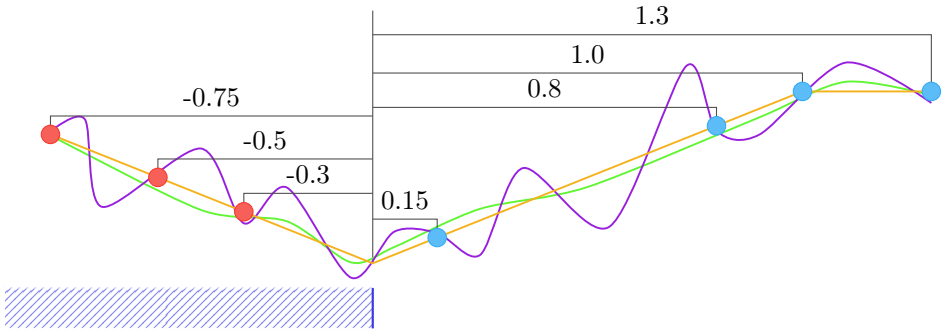


Figure 4.9: Possible overfitting on the TSDF values in a one-dimensional scenario if the gradient is not restricted. On the bottom of the figure, a dark blue block depicts our object, and the distances to the object's surface are depicted above. The negative TSDF points are red, while the positive points are blue. In yellow the optimal TSDF prediction values are shown, in green a best-case prediction, while in purple a failed prediction is shown. This failed prediction contains high-frequency signals and has overfitted to the data.

shown in fig. 4.9. As our space is only as densely sampled as our input data, we want to avoid that TSDF values between two arbitrary points show any sign of high-frequency changes. These high-frequency changes could arise as the network tries to overfit on the TSDF values and is not limited to a specific gradient. So, we introduce a loss on the gradient $\mathcal{L}_{\text{gradient}}$ with respect to the input point p itself. Thus, we subtract from our gradient $\delta\mathcal{L}_{\text{combined}} \,/\, \delta p$ the maximum value g the gradient should take. This maximum value g is the maximum change between two points in space, defined in eq. (4.12). This change relates precisely to the TSDF change $dist_{\text{tsdf}}(p_i, p_j)$ defined in eq. (4.13), as the TSDF values get clipped and have an orientation towards the surface. Additionally, the change in TSDF space can be smaller than the one in euclidean space, as seen between the two points on the right in fig. 4.9.

$$dist_{\text{Euclidean}}(p_i, p_j) = \|p_i - p_j\|,\ \forall p_i, p_j \in \mathbb{P} \tag{4.12}$$

$$dist_{\text{tsdf}}(p_i, p_j) = |y_{\text{tsdf}}[p_i] - y_{\text{tsdf}}[p_j]|,\ \forall p_i, p_j \in \mathbb{P} \tag{4.13}$$

$$dist_{\text{tsdf}}(p_i, p_j) \leq dist_{\text{Euclidean}}(p_i, p_j),\ \forall p_i, p_j \in \mathbb{P} \tag{4.14}$$

So, we set the maximum change of $dist_{\text{Euclidean}}(p_i, p_j)$ as the upper bound, which is the same as the $\delta\mathcal{L}_{\text{combined}} \,/\, \delta p$. After the subtraction, we apply a ReLU to remove all negative values, indicating a gradient smaller than the distance between two points. In the end, we scale this loss with our gradient factor θ_{gradient}, as seen in eq. (4.15).

$$\mathcal{L}_{\text{gradient}} = \theta_{\text{gradient}} \cdot ReLU\left(\frac{\delta\mathcal{L}_{\text{combined}}}{\delta p} - g\right) \tag{4.15}$$

4.2.3 Compressing Millions of Blocks

The final challenge of this implicit representation is the required computation time. As we use these blocks as reconstruction targets, we need to compress at least 90,000 scenes, where each scene consists of 16^3 blocks, meaning we need to compress $16^3 \times 90,000 = 368,640,000$ blocks. Each block takes 1,15 seconds on an NVIDIA Geforce RTX 3090. For all blocks, this would then

take roughly 12.6 years. As this was beyond our time budget, we needed to improve this drastically. Nevertheless, the challenge is that each latent vector needs to be optimized as described in section 4.2.1. So, we perform $1,400$ optimization steps per latent vector l, to get from a zero-initialized vector to our compressed latent representation.

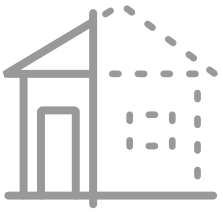
Our first optimization focuses only on reusing common blocks. So, we split our scene into three parts based on a TSDF volume grid with a resolution of 256, created simultaneously with the semantic TSDF point cloud. These three parts are the boundary, the non-boundary, and the free and filled voxels. We then spend most time compressing the boundary voxels, as they contain a surface inside them. For the non-boundary voxels, the process is more straightforward, as the precision of the reconstructed TSDF values or semantic classes is not as crucial as for the boundary voxels. For the free and filled voxels, we precalculate the latent vectors l once and use them. Correspondingly, all of them get the same 'void' class assigned.

This splitting into different parts already avoids compressing a big chunk of the data, but we still need to improve it further. Thus, to compress the boundary voxels faster, we first calculate a database of $1,483,472$ blocks, which equates to 750 scenes with the whole $1,400$ steps. We then search for the block in the database most similar to our current block and use its latent vector as initialization instead of the zero vector we used before. This comparison of two blocks is made by voxelizing our TSDF points with a side resolution of four, giving us 64 different sub-blocks, and for each sub-block, we calculate the mean TSDF value. Then we add the probability distribution of the points' categories to these 64 values, giving us 74 elements in total for ten categories. The resulting vector characterizes our current block. Now, we only need to find the block in our database closest to our new 74-long vector and use it as initialization. This search is done via KD-Tree [8].

By finding and reusing these precalculated latent vectors, we can reduce the number of necessary optimization steps from $1,400$ down to 750 steps. Further, we check every 125 steps if the absolute average error over the reconstructed TSDF values is below 0.02 and the average class accuracy is above 95%. If that is the case, we conclude this compression and move on to the next block. We also build up a database for the non-boundary blocks. This database enables us to reduce the number of steps drastically to 160;

here, we check every 20 steps if the reconstructed TSDF value is below 0.03. Luckily, we only have to predict half of the blocks, as most are free or filled voxels, and secondly, we can speed up the single prediction of a latent compression by using a closer initialization, increasing the compression speed per block to 0.3 seconds. So, the time on a single NVIDIA Geforce RTX 3090 is reduced to only 1.27 years or roughly nine days on 50 GPUs. This speed-up approach gives us a compressed latent representation, which we can use in our scene reconstruction as a target.

Chapter five



Scene Reconstruction

> *Vision is the art of seeing what is invisible to others.*
>
> — **Jonathan Swift**
> Poet

Reconstructing an entire 3D scene based on a single color image is challenging, as the solution space is enormous. This work focuses solely on indoor scenes in an attempt to limit the solution space; as such indoor spaces are highly structured and constrict the possible range of solutions. This chapter will highlight how a single color image can be transformed into a 3D scene representation, including occluded areas in the scene, e.g. behind a table. We use the encoded 3D scene representation discussed in chapter 4.

5.1 Tree Architecture

In order to reconstruct an entire scene based on a single color image, we proposed in prior sections to first project our camera frustum into a cube. This cube then gets voxelized into single blocks, and each block gets compressed

into a latent vector, either using the voxel grid or the implicit compression. So, we are left with a compressed 3D volume of latent representations. The task is now to convert a color image into a 3D latent representation.

Our first naive approach is to use a general convolutional neural network that converts at some point in the network the 2D content into 3D by transforming the 2D feature channels into a new axis in 3D. This transformation can be done by reshaping a 32^3 tensor to a $32^3 \times 1$ tensor. However, this approach has one major drawback it requires the network to learn the transformation of 2D features to a specific 3D location in one step. This is difficult as single convolutional layers are conceptually designed to perform single easy-to-do operations [92, 87]. Only combining several of those convolutions makes it possible to target complex problems [92]. We try to solve this by introducing intermediate steps into this 2D to 3D conversion. This conversion is done by splitting a convolutional neural network's usual linear path into two different ones. The idea here is that we create two paths for the information to flow, similar to AlexNet [87], where in our version, one path takes care of the front of the scene and the other one of the back. So, after some initial 2D feature extraction layers, their feature output is used as the input to two different branches. In the end, these two branches are joined to form a 3D volume, while each branch is responsible for either the front or the back of our scene. So, in the fusing step of these two branches, one branch forms the front from the start to the middle of our newly created Z-axis or depth axis, while the second branch is used for the middle to the end part. This splitting allows the first branch to be responsible for detecting features closer to the camera, while the second one does the opposite. We show this architecture in fig. 5.1.

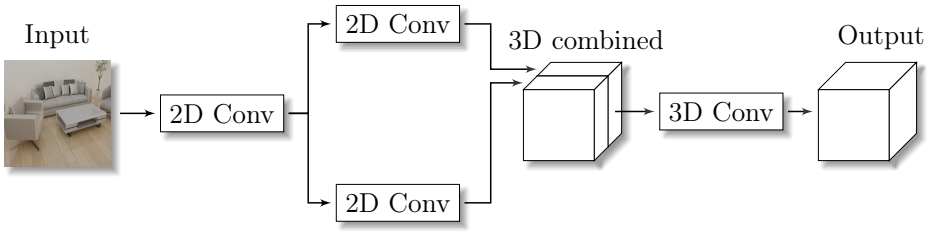


Figure 5.1: A simple branch splitting is displayed in this figure, enabling 3D reconstruction. The upper branch in this image is responsible for the rear end, and the lower branch is for the front part. Each `2D Conv` block can consist of multiple 2D convolutions, while a `3D Conv` block has multiple 3D convolutions. The output is the encoded latent representation from chapter 5.

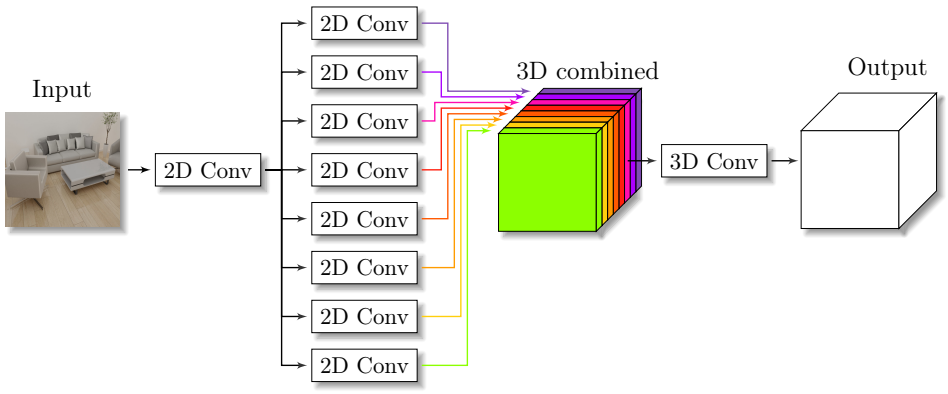


Figure 5.2: This depiction combines eight different branches of 2D convolutions into eight depth slices. This combination allows a more granular depth feature prediction, as the two-branch network shown above.

We can improve this further by using more than just two branches. In fig. 5.2, we show this for eight different branches, which then learn to represent a particular depth slice. Making each depth slice smaller enables the network to separate specific features into particular depth slices more smoothly. However, the drawback is that we need four times more computations than the two-branch network. So, we can combine certain parts of this process to reduce the computational overhead and streamline the process. This reduction is achieved by combining the two-branch network with the idea of multiple branches by applying a simple binary tree structure. At first, the input path is split into two branches, conceptually representing the front and back of our scene. Then, this process is repeated to create a binary tree with different layers, so each node is split further, allowing the network first to tackle the problem of separating a scene into front and back and then progressively into smaller depth slices. This binary tree can be seen in fig. 5.3.

One last drawback of the proposed method is that the binary tree only creates a 3D volume with a feature channel size of one. In order to remove this bottleneck, we create a multi-path mapping of the 2D features to the 3D volume. This mapping is done by increasing the output of each leaf node so that we can separate the feature channels of one leaf's output over the 3D outputs. So, each 3D combined volume uses the same path from the tree root to the leaf for a particular slice along the Z-axis but uses a different slice of the leaf node. In fig. 5.4, we show this for only two 3D volumes with eight leaf nodes, meaning that each leaf node has to split its feature channels

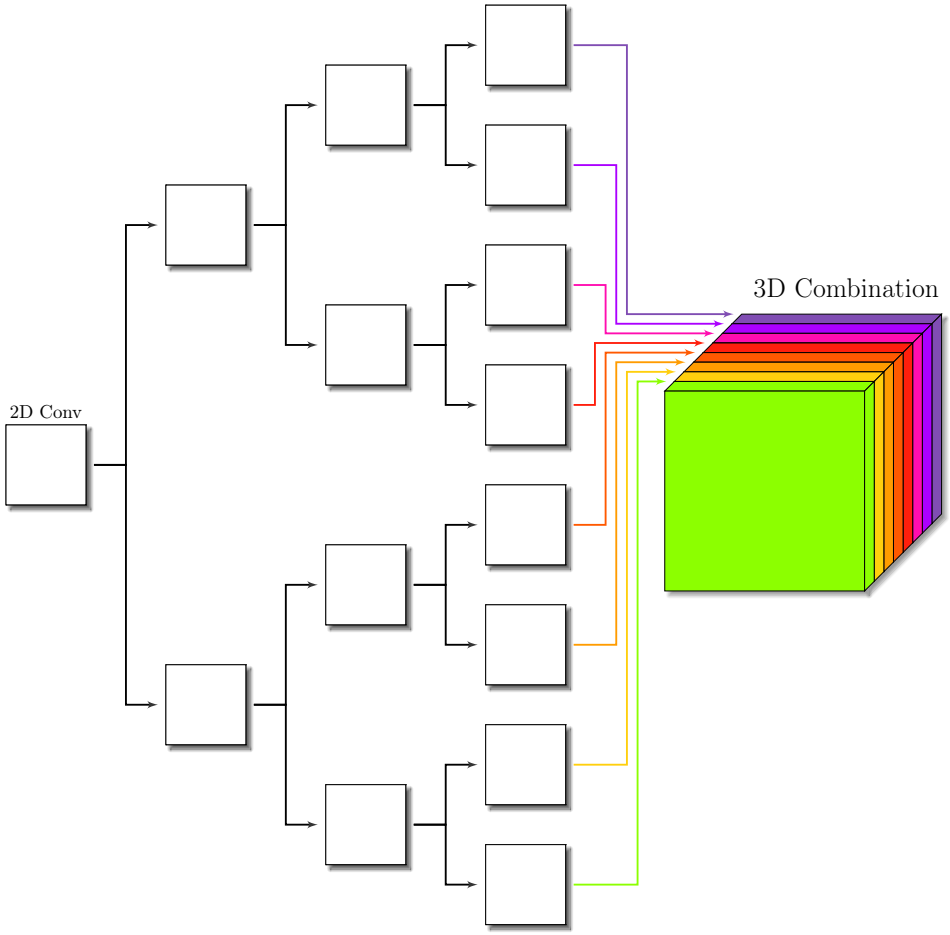


Figure 5.3: Our tree-branching architecture: After the feature extraction on the color image is done, the features are the input to two branches. These branches are then split several times sequentially to form a binary tree. The feature channels of the leaves are then combined to form a new axis in 3D. This splitting divides the mapping problem from 2D to 3D into several smaller problems, making it easier for a neural network to learn.

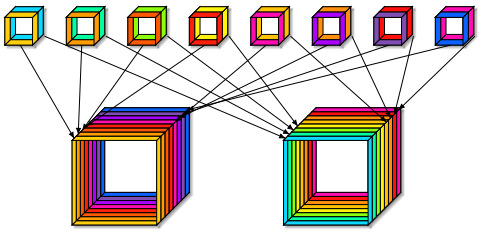


Figure 5.4: This figure illustrates how the multipath approach in the last leaves of the tree works. Here, the upper row represents the leaves of our tree architecture. Each leaf has eight feature channels in this example, which are evenly split over the resulting volumes. In this case, only two volumes are used.

over the two 3D volumes. We then use the first half of each 2D result in the left 3D volume, while the second half is used in the right one. These multiple paths allow us to increase the amount of 3D volumes and remove the bottleneck.

5.2 TSDF Volume Reconstruction

We start by designing an architecture capable of reconstructing the compressed TSDF grid-based volume, presented in section 4.1. We coin this architecture Single View Reconstruction with an TSDF grid-based compression (SVR-GC). It uses a color image and a surface normal vector image to reconstruct a $32^3 \times 64$ encoded TSDF volume. Both images have a resolution of 512^3. The surface normal image has a corresponding normal vector for each pixel in the image. These surface normal images have been successfully used in mono depth estimation tasks, as they provide continuity information about the current scene [183]. These are usually not available in the real world, so we train a simple U-Net architecture [130] to infer the surface normal image for a given color image. The compressed output can be decoded after the network is applied to the image using the autoencoder to achieve a final TSDF volume with a resolution of 512^3.

We assume for now that the training data is already generated for details; see chapter 6. In the first step of our architecture, we reduce the spatial

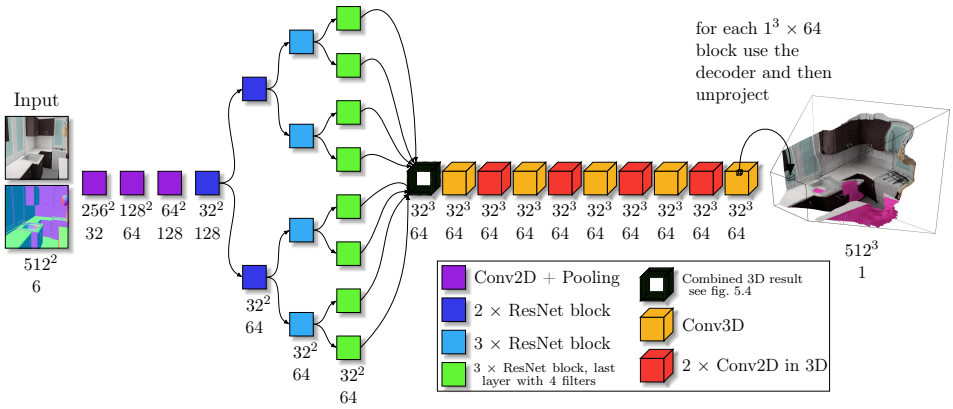


Figure 5.5: A compressed form of our proposed architecture is shown here. An exemplary RGB and normal image are used as input to the network on the left. On them, several convolutions and pooling operations are done to the spatial size down to 32^2. We then split the path of the network in two, where one represents the front and the other the rear part of our scene. This split is done two more times. The resulting depth slices are then combined into a 3D structure with 64 channels. On that, we perform some 3D convolutions and use the auto decoder to decode the output of the tree net. Note that our real model has an additional tree layer, where we repeat the second tree layer once more.

dimension of the input down to the output dimension of 32 by using four sets of 2D convolutions, each combined with a max pooling.

Our network's task is scene reconstruction, which requires the network to understand the relationship between different objects in the scene, even if these objects are far apart in the image. We solve this by increasing the receptive field of general 2D convolutions. Our alteration is inspired by an inception layer, in which each input is forwarded through several independent convolutions [149]. These results are then concatenated, and an activation function is applied. Usually, each path of an inception layer has a different filter size. As we do not want to increase the computation further, we only increase the dilation rate and keep the filter size the same for all convolutions. So, we split the desired amount of feature channels into three parts with a size of $50\%, 25\%$, and 25%, with corresponding dilation rates of $1, 2$, and 4. This splitting ensures that, for example, 25% of the used feature channels use a dilation rate of four. All convolutions in this network use a same padding,

which appends zero values in the outer frame of the features.

After extracting the 2D features, we use the newly introduced tree architecture from fig. 5.3. The root node uses two ResNet blocks [70] with a feature channel size of 128. Each ResNet block contains two convolutions, with a skip connection skipping these two. The result of the two convolutions and the block's input is summed up and passed through a ReLU. Using these ResNet blocks here allows the network to more easily get trained with more layers as the loss can be better backpropagated. Each convolution in a ResNet block uses our inception layer-inspired dilated convolutions to ensure a proper combination of spatial features. The rest of the layers in the tree use a feature channel size of 64, primarily to reduce the architecture's memory footprint. In the fourth layer, we use the same feature channel size but increase the number of ResNet blocks from two to three. The final layer consists of a total of 16 leaf nodes. Again it uses three ResNet blocks, and after those, a last 2D convolution with filter size one is used to increase the filter amount to 128. This is necessary to use our multi-path splitting shown in fig. 5.4. After combining the 2D features into a 3D volume with the multi-path splitting, we perform four sets of 3D convolutions paired with a planar 2D convolution in all directions. All with the same feature channel size of 64. The final 3D convolution added does not have an activation, as every other layer uses a ReLU as an activation function [51]. A compressed form of the architecture for transforming an input image into a reconstructed 3D scene can be seen in fig. 5.5.

5.3 Implicit TSDF Reconstruction

After discussing how to transform a 2D image to a TSDF voxel grid, we want to repeat this for the implicit representation, which we call Single View Reconstruction with an implicit compression (SVR-IC). Here, we need to slightly adapt the architecture from section 5.2, as the output has a lower resolution of 16 instead of 32. This resolution decrease makes each block bigger. To compensate, we increase the feature channel size from 64 to 512, reaching the same total number of feature values as SVR-GC. This decrease

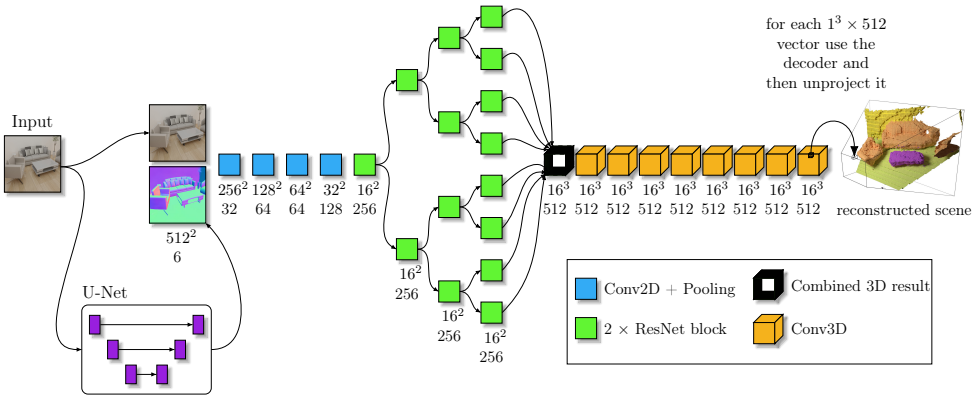


Figure 5.6: This presents our proposed architecture going from a single color image to a compressed implicit scene representation, which is then decompressed with a decoder. A simple U-Net architecture produces the surface normal image from the color input image. In contrast to SVR-GC, the compressed latent resolution is only half as big at 16, while the feature channel size is eight times bigger.

in spatial resolution also means we have one additional 2D convolution plus max-pooling before the tree net architecture. Additionally, we also store semantic information in each block. So, our input image with a resolution of 512^3 is passed through five convolutions with max-pooling operations to reduce the spatial size to 16^3 and increase the feature channel size to 256. We also use one less tree layer as we only need to get to 16 depth slices, resulting in three tree layers with eight leaf nodes. Each node in the tree uses two ResNet blocks. Finally, we only use conventional 3D convolutions. These changes result in the architecture seen in fig. 5.6.

5.4 Loss Shaping

The regression on 4D volumes is a challenging problem, as the number of values to regress is exceptionally high. In both architectures for SVR-GC and SVR-IC, the output has $32^3 \times 64 = 16^3 \times 512 = 2,097,152$ values to regress. For comparison, the input only has $512^3 \times 3 = 786,432$ values. So, a focus has to be set to ensure that the essential values are better reconstructed than the unimportant ones. By essential values, we refer to the latent values of the

vectors representing surface boundaries in contrast to a latent vector, which represents free or occluded space. The goal is to shape the loss so that the reconstruction of latent vectors representing boundary blocks is prioritized higher than for free or occupied blocks. In order to achieve this loss shaping, we propose to increase the loss on blocks in which more complex content is stored.

5.4.1 TSDF Volume Grid

For the loss generation in SVR-GC, we use the 512^3 TSDF volume grid as starting point. At first, we search for the free blocks in the camera's line of sight. For this search, we only have to walk along the Z-axis away from the camera and check if each voxel is free, meaning their TSDF value is above zero. All free blocks in the camera's line of sight before any surface is detected are set to the status ρ_{Free}. The first block, which TSDF value is below zero on this ray, gets a value of ρ_{Surface}. This surface loss ρ_{Surface} is much higher than the ρ_{Free}, as can be seen in the legend of eq. (5.1). We then employ a flood fill algorithm [142] to find more free blocks which are reachable from any of the already defined free and visible blocks. These get a value scaled by the distance to the closest free visible block, called $\rho_{\text{NonVisibleFree}}$. Blocks

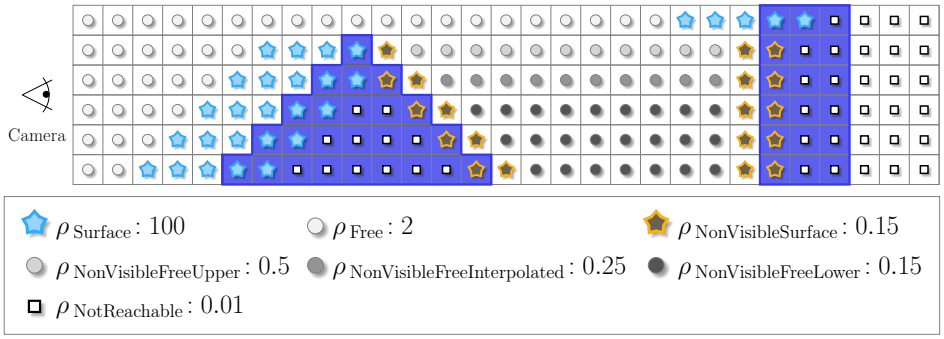


Figure 5.7: In this top-down 2D map of a scene, two objects are depicted. The two objects in dark blue depict different structures in the scene. For these two objects, the loss factors are set by the symbols defined in the legend below. It also contains the weight values used in our approach. The only exception is the $\rho_{\text{NonVisibleFreeInterpolated}}$, which is interpolated between the upper and lower value based on the distance to the closest ρ_{Free}.

close to the visible blocks get a higher value than blocks further away. We argue that it might be more straightforward to predict the free space behind a small object rather than a larger one. So, we decline the value linearly starting from the $\rho_{\text{NonVisibleFreeUpper}}$ down to the $\rho_{\text{NonVisibleFreeLower}}$ for at most 7% of the spatial resolution, which correlates roughly to 36 blocks. All surface blocks that the flood fill algorithm reaches, which are not visible, are set to $\rho_{\text{NotVisibleSurface}}$. As the loss grid is averaged down to match the output resolution of 32^3, settings only one value to ρ_{Surface} would not significantly affect the loss. To fix this, we propose to change the 64 values before and 32 values after it to the same loss of ρ_{Surface}, significantly increasing the amount of ρ_{Surface} values and, therefore, their importance. All unchanged values are set to $\rho_{\text{NotReachable}}$.

That defines our training loss $\mathcal{L}_{\text{scene}}$ for the output $o^{\mathbb{F}}$ with the ground truth $y^{\mathbb{F}}$ in eq. (5.1).

$$\mathcal{L}_{\text{scene}} = \sum_{i}^{32}\sum_{j}^{32}\sum_{k}^{32} \mathcal{W}_{\text{scene}}^{\text{GC}}[i,j,k] \sum_{c^{\mathbb{F}}}^{64} \| o^{\mathbb{F}}\left[i,j,k,c^{\mathbb{F}}\right] - y^{\mathbb{F}}\left[i,j,k,c^{\mathbb{F}}\right] \| \quad (5.1)$$

Each summed value of the absolute distance between the output $o^{\mathbb{F}}$ and the ground truth $y^{\mathbb{F}}$ is weighted with the $\mathcal{W}_{\text{scene}}^{\text{GC}}$. The normalizing factors are removed as they only change the magnitude of the loss. An example of this can be seen in fig. 5.7, where a camera looks into a scene with two different objects colored in dark blue. W e also mention our used values in it, indicating that a visible surface ρ_{Surface} should get a high loss value of 100 compared to non-reachable space $\rho_{\text{NotReachable}}$ with only 0.01. The free space on the right, which is not visible but reachable from the camera on the left, gets decreased values compared to free space. As mentioned before, the final 512^3 loss grid is averaged to a resolution of 32^3. This reduction makes it possible to use it without additional memory overhead while also shaping the loss during training to focus on the essential parts of the scene.

5.4.2 Tree Loss Shaping

This loss shaping is used not only on the final output of SVR-GC but also inside the tree. This loss shaping inside the tree speeds up the training,

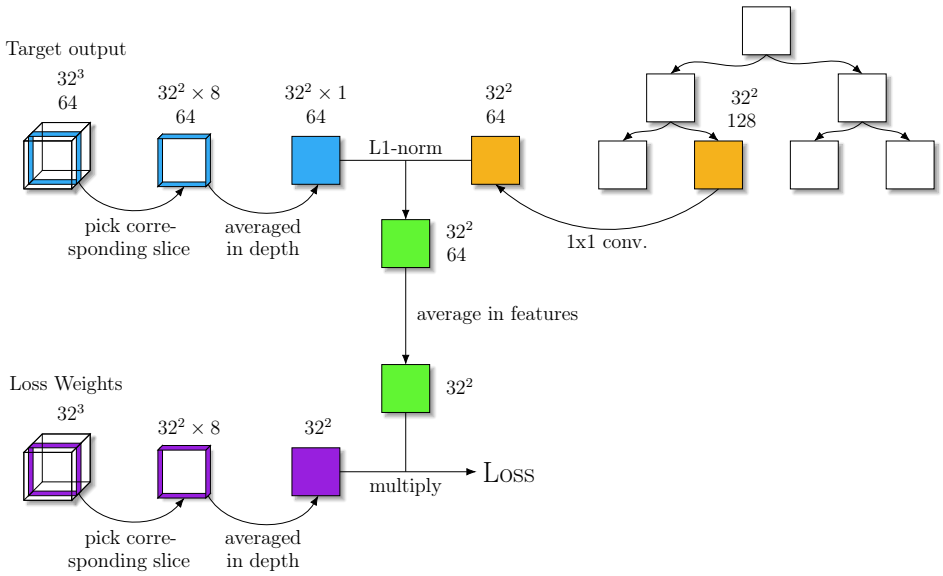


Figure 5.8: The application of the tree-loss is depicted here. For each node in the tree, the corresponding depth layers from the output are averaged in the depth dimension. The result is then compared to a reshaped tensor from the tree node. This process already enforces a sense of the encoded 3D structure in the tree.

as we already enforce the depth splitting described above during training. So, for the first split in the layer below the root, we take the result of the left node and branch it into a 1×1 convolution to change the number of feature channels to 64, so they can be matched to the target output. We then take the target's output, use only its first half corresponding to the depth slice of that left node and average it in the depth dimension. After this, the L1 loss is computed between the averaged target and the averaged feature output. Before averaging the $32^3 \times 64$ to one value, we first average them across the feature dimension, reducing it to 32^3, on which we can add the averaged loss shaping weights produced in the last section. This weighting already strengthens essential features during the tree splitting. We show this in fig. 5.8. This process is then repeated for each tree layer and each node, where each node is responsible for a different depth slice. Nodes further down the tree and closer to the leaves get smaller depth slices, while nodes more to the right in our tree get depth slices further in the back of our scene. We then sum the losses per layer level in the tree to one value and weigh them according to their layer height. In the tests, we used the following

weights $[0.2, 0.3, 0.5, 0.8]$, where the smallest value was for the top level, and the highest value was for the leaf nodes. The resulting value is scaled by 0.4 and added to the final loss.

5.4.3 Implicit Representation

The implicit representation is stored as a compressed grid for our SVR-IC approach so that each latent vector represents a different compressed scene block. In order to keep the focus of the network on the essential blocks, we use a similar loss shaping as proposed in the TSDF volume case. However, we perform some changes to improve the loss weights further. The first change is that we only use a resolution of 256 for the loss grid to save computation time. We then increase the non-visible free space range from 7% to 59%, which corresponds to 150 values at a resolution of 256. We further separate between ρ_{surface} and $\rho_{\text{true surface}}$, where the $\rho_{\text{true surface}}$ is reserved solely for the first voxel in the camera's line of sight, which is occupied. The ρ_{surface} is then filled in for 32 values before and 16 values after a $\rho_{\text{true surface}}$ value. Additionally, we do not only consider the camera's line of sight but use a sphere with a radius of 16 around each $\rho_{\text{true surface}}$ value and set them to ρ_{surface} as well. This additional step increases the loss on thin objects in all

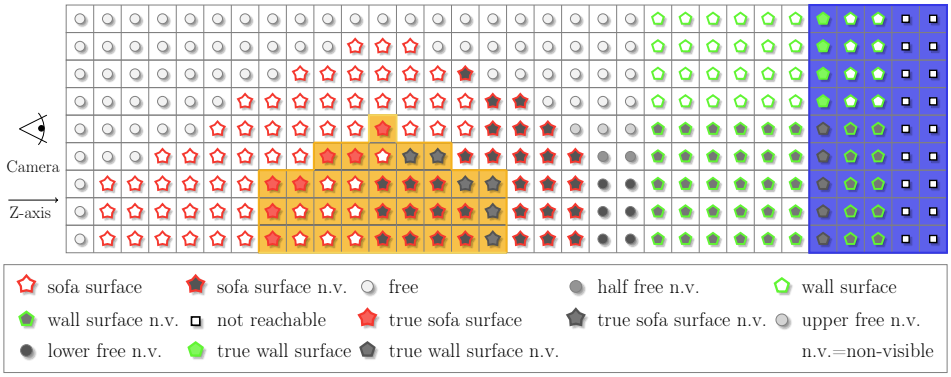


Figure 5.9: A top-down 2D map of a scene is shown. The yellow shape on the left is a sofa, and the blue block on the right is a wall. They determine the loss values. In contrast to before, the surface values are independent of the camera direction and spread in all directions, increasing the loss, particularly around thin objects.

directions equally instead of just in the viewing direction. The exact process is repeated for the $\rho_{\text{non-visible surface}}$ values, splitting into $\rho_{\text{non-visible true surface}}$ and $\rho_{\text{non-visible surface}}$ and also relying on the sphere to get a better and broader loss shaping in all directions.

Finally, we also combine each loss value with a category label, allowing us to focus the attention of the network on specific categories. For example, we increase the loss shaping around objects by a factor of five and keep it the same for structures like walls or floors. Furthermore, this loss is computed on the fly during training, allowing these values to change after generation. In the end, the 256^3 generated loss values are replaced by the designated weights and averaged to a resolution of 16^3. So, our training loss $\mathcal{L}_{\text{scene}}$ for the output o^{F} with the ground truth y^{F} is defined in eq. (5.2).

$$\mathcal{L}_{\text{scene}} = \sum_{i}^{16}\sum_{j}^{16}\sum_{k}^{16} \mathcal{W}^{\text{IC}}_{\text{scene}}[i,j,k] \sum_{c^{\text{F}}}^{512} \| o^{\text{F}}\left[i,j,k,c^{\text{F}}\right] - y^{\text{F}}\left[i,j,k,c^{\text{F}}\right] \| \quad (5.2)$$

We take the sum of the absolute difference between each output o^{F} and ground truth y^{F} and weigh it without dynamic loss weight $\mathcal{W}^{\text{IC}}_{\text{scene}}$, which we generated prior to the training. This loss weight is only used on the final output and not on intermediate results inside of the tree, as in the SVR-GC variant.

Chapter six

Synthetic Data Generation

> *Data is a precious thing and will last longer than the systems themselves.*
>
> — **Tim Berners-Lee**
> Computer scientist

6.1 BlenderProc

Before our method is trained, we need to create a dataset. This dataset must allow us to learn to reconstruct an entire 3D scene from a single color image. Sadly, we cannot rely on existing datasets such as the Matterport3D dataset [18], in which 90 buildings have been recorded, or five buildings for the S3DIS dataset [76]. The issue with these is that they are not hole-free and, therefore, unsuitable for use as a reconstruction target, as it is impossible to estimate the distance to a surface that does not exist. Furthermore, manually matching color images with real scenes is exceptionally labor-intensive when capturing a wide range of various interior spaces. This process is particularly labor-intensive as we need a complete 3D scene reconstruction of the visible

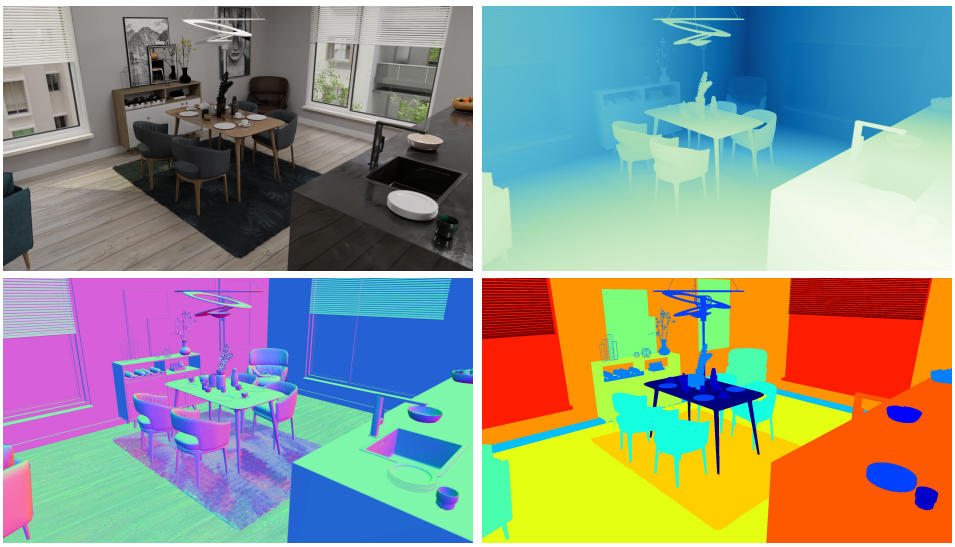


Figure 6.1: A possible output from BlenderProc: The rendered color image is depicted in the top left. The distance image is shown to the right, and below both, the surface normal vector image and a possible semantic segmentation are shown.

and occluded areas in the scene for each color image. This 3D model must contain all surface areas reachable from the camera, and their quality would set the upper boundary for our approach. So, even the backside of a chair or a couch visible in the image has to be modeled. As there are no real datasets available fulfilling our requirements and the effort to create a dataset in the real world is too tedious, we decided to rely on simulation. One of the biggest challenges there is simulating realistic color images. Thus, we created BlenderProc[10], a new open-source tool that allows the generation of synthetic photorealistic images, as shown in fig. 6.1.

6.1.1 BlenderProc 1.X

The first public release of BlenderProc is designed as a modular-based framework, which gets configured via `YAML` config files. A typical BlenderProc run would start with an initialization phase, which starts by downloading Blender [10]. After the download, all required pip packages inside Blender's python environment get installed. This process is entirely automatized to make the

usage independent of the chosen platform. This independence means that BlenderProc supports Linux-based systems, Windows, and Mac OS. After this automatic setup, Blender's internal state machine is set up to render our scenes. This setup entails configuring the camera and setting Cycles [10] as our used rendering engine on all available GPUs. After completing the setup, an object or an entire scene is loaded into Blender via BlenderProc's API. For this, we offer various loaders for standard formats like `OBJ`, `PLY`, or `STL`. On top of that, it is also possible to load complete scenes from datasets like SUNCG, SceneNet, 3D-FRONT, IKEA, Pix3D, Replica, ShapeNet, AMASS, and all eleven datasets supported in the BOP challenge [143, 64, 48, 49, 97, 148, 145, 19, 101, 73, 75]. Furthermore, BlenderProc supports non-standard datasets like all assets, which can be downloaded from BlenderKit, PolyHaven, or AmbientCG [38, 181, 30]. After the objects have been loaded via BlenderProc's API, we offer options to place them in the scene. Allowing the loading of an interior scene from the SUNCG dataset, SceneNet dataset, or 3D-FRONT dataset and then adding an object from the ShapeNet dataset to it. The ShapeNet object could be randomly placed above a bed in mid-air, from where it is dropped onto the bed with the integrated physics [23], see fig. 6.2. This use of multiple objects dynamically to build a scene allows for the creation of a variety of different datasets for a multitude of various problems. The original goal was to generate images for the scene reconstruction task, but a tool like BlenderProc can be easily used for multiple problems. Ranging from 6D pose estimation, as BlenderProc is used for a challenge in the European conference on computer vision (ECCV), called Benchmark for 6D Object Pose Estimation challenge or short BOP challenge [75], to object segmentation [41, 5, 15] and camera localization [170].

After placing all objects, we must position a camera in the scene. This positioning can be done in numerous ways. For example, for the basket on the bed, the camera could be positioned on a sphere centered around the basket while focusing on the basket itself. We also check that the basket is visible and that no walls obstruct the view. Finally, we can render the scene; for this, BlenderProc offers various renderers. Besides rendering color information, with a physical-based lighting simulation [120, 10], it is also possible to render the surface normals, the depth information, the optical flow, or semantic segmentation. This rendering is done for each camera pose,

Figure 6.2: A basket from the ShapeNet dataset [19] is first placed above the bed and then dropped with the physics capabilities of BlenderProc onto the bed's surface, resulting in a random placement. The scene with the bed is from the SceneNet dataset. [64].

and all data related to one pose is saved in an `HDF5` container, ensuring no mixups.

A significant focus of BlenderProc `1.X` lies on the specific `YAML` config files, which are used to control the flow of the program. Before any of these modules can be used in the `YAML` config file, BlenderProc has to be set up, and it only requires the `blender_install_path` and the required pip packages. This setup can be seen in code snippet 6.1.

Code 6.1: The setup for BlenderProc in the `YAML` config files.

```
1  "version": 3,
2  "setup": {
3    "blender_install_path": "/home/<env:USER>/blender/",
4    "pip": ["h5py"]
5  },
```

After that, the modules are defined in sequential order. Each file starts with the Initializer Module, which sets up Blender for rendering with Cycles. We also define a global value accessible in all modules, namely the `output_dir`. This global variable makes it easier for each writer to access the desired `output_dir`. The input value to the output directory variable is controlled via the arguments handed over in the command line. We also use such command line arguments in the next module called the Object Loader, which loads an `OBJ` file specified via the first given argument: `<args:0>`. This loading can be seen in lines six and seven in code snippet 6.2. After loading,

we need to position the camera using the camera sampling module. It samples the camera's location uniformly in a box, where the camera is rotated to look at the mean center of all objects. This rotation is achieved using the config files' getter and setter system. After setting the five camera poses, we need to render them, as shown in lines 24 to 28 in code snippet 6.2. We additionally specify to render the surface normals and the distance images in the same step. In the end, we store all collected information in `HDF5` containers.

Code 6.2: Loading an object, placing a camera and rendering a scene in the `YAML` config files of BlenderProc.

```
"modules": [
  { "module": "main.Initializer",
    "config":{
      "global": { "output_dir": "<args:1>" }
    } },
  { "module": "loader.ObjectLoader",
    "config": { "path": "<args:0>" } },
  { "module": "camera.CameraSampler",
    "config": {
      "cam_poses": [
        {
          "number_of_samples": 5,
          "location": {
            "provider": "sampler.Uniform3d",
            "min": [-10, -10, 12], "max": [10, 10, 8]
          },
          "rotation": {
            "format": "look_at",
            "value": { "provider": "getter.POI" }
          }
        }
      ]
    } },
  { "module": "renderer.RgbRenderer",
    "config": {
      "render_normals": True,
      "render_distance": True
    } },
  { "module": "writer.Hdf5Writer" }
]
```

As shown here, it is possible in the `YAML` config files to refer to internal states using the `getter.Entity` and the `EntityManipulator`. A `getter`

can generally return a list of entities or just a single one. It can also return the attribute of an entity in the scene. However, these getters still have multiple drawbacks. They are often overly complex and are not adaptable to all circumstances, requiring the user to set custom properties for loaded objects to find them later. So, we decided to depart from our specific `YAML` config files and move to a more accessible API via python. In python, we can work with newly designed object classes, which is more straightforward and familiar to the average user than relying on our custom-designed `YAML` config files.

6.1.2 BlenderProc 2.X

As highlighted in the last section, BlenderProc `1.X` was designed to have a rigid sequential pipeline configured via `YAML` files. For BlenderProc 2, the goal was to remove those obstacles and liberate the possible use cases. However, this meant restructuring the entire project, away from fixed modules that execute a particular task down to functions that influence Blender's state machine. So, the setup is now done by executing the `bproc.init()` command, making BlenderProc much easier to use. Furthermore, we released BlenderProc as a pip package, allowing to pip install BlenderProc. After which, one can directly use our command-line tool named `blenderproc`.

The code snippet 6.2 from above in the `YAML` config style boils down to the code snippet 6.3 in python:

Code 6.3: The setup for BlenderProc with a python config file.

```
import blenderproc as bproc

bproc.init() # set up blender for BlenderProc

objs = bproc.loader.load_obj("SCENE_PATH")

# define a light and set its location and energy level
light = bproc.types.Light()
light.set_location([5, -5, 5])
light.set_energy(1000)

# Find point of interest, all cam poses should look towards it
```

```
poi = bproc.object.compute_poi(objs)
# Sample five camera poses
for i in range(5):
    # Sample random camera location above objects
    location = np.random.uniform([-10, -10, 8], [10, 10, 12])
    # Compute rotation going from location towards poi
    rot_matrix = bproc.camera.rotation_from_forward_vec(poi - location)
    cam2world_matrix = bproc.math.build_transformation_mat(location, rot_matrix)
    bproc.camera.add_camera_pose(cam2world_matrix)

# activate surface normal and depth rendering
bproc.renderer.enable_normals_output()
bproc.renderer.enable_depth_output(activate_antialiasing=False)

# render the whole pipeline
data = bproc.renderer.render()
# write the data to a .hdf5 container
bproc.writer.write_hdf5("OUTPUT_DIR", data)
```

We rely on the same principles but give users more freedom of choice. For example, they can add their own sampling method for the location and are not bound by the prior defined modules. That means any sampling method offered, for example, by `NumPy` [67], can be used. Furthermore, a user can design their own writer, as the `data` is just a dictionary mapping each image type to a list of images for each camera pose.

These design goals make BlenderProc an open-source success, which enabled us to find bugs more efficiently and get feedback on our new features. Additionally, the community added their own requested features to BlenderProc, making it a more accessible and better tool overall. One of our users, for example, added the possibility to save an image sequence directly as `GIF`, while another provided Windows support. Since October 28, 2019, BlenderProc has been an open-source project, and over the years, we have collected over 1700 stars on Github. The progression gain can be seen in fig. 6.3.

6.1.3 Advanced BlenderProc Features

BlenderProc is a versatile tool that can generate synthetic color images of interior scenes. In this section, we highlight some of the most valuable and more advanced features.

6.1.4 Camera Sampling

To ensure that the rendered images generalize well to the real world and close the *sim2real* gap, we need to sample camera poses realistically. We start by sampling a random location on one of the floor objects in our scene and add a randomly sampled height and viewing direction. For this pose, we check if any objects are too close to the camera and if the average distance for the current view is acceptable. In this work, we set this minimum required distance to 1.25 meters and the interval for the average distance to be 1.5 to 3.5 meters. This distance check ensures that the camera is placed not too close to any object avoiding collisions and that there is more than just empty space in front of it. Some scenes might not have windows or doors, or

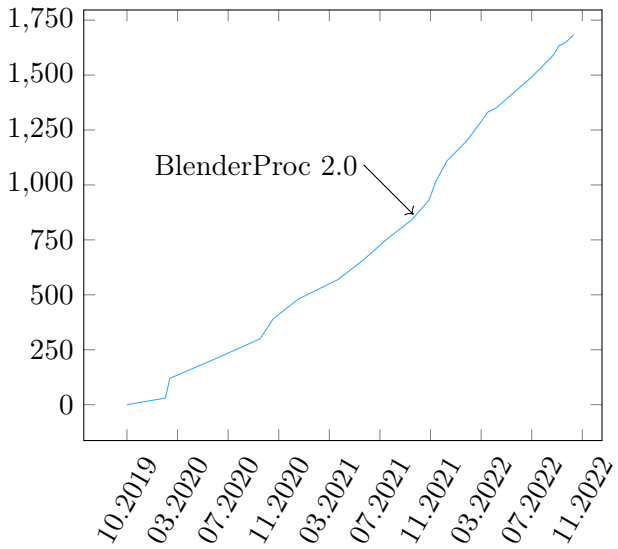


Figure 6.3: GitHub stars over time for the open-source project BlenderProc, showing how the popularity grew long after the first version's release.

sometimes an entire wall is missing, so we remove all camera views, which look out into the open space. We further improve the realism by detecting objects in the camera view at a low resolution of 10 by 10, mapping each pixel to a specific object. We assign each object a score and sum these up for the generated image; objects which are part of a priorly selected set of interesting objects get a score of four, and the other a score of one. These interesting objects enrich the likelihood that a camera pose is chosen. We have used these categories as interesting: bed, table, sofa, or chair. All those objects are more likely to be photographed by a human than a blank wall.

$$\sigma_{\text{interesting score}} = \frac{1}{I_{\text{height}} \times I_{\text{width}}} \sum_{i=1}^{I_{\text{height}}} \sum_{j=1}^{I_{\text{width}}} \mathbb{1}(I[i,j] \in \mathbb{C}_{\text{interesting}}) \quad (6.1)$$

Additionally, we calculate a scene variance score $\sigma_{\text{scene score}}$ by multiplying the inverses of the class's probabilities in our 10×10 image together, as can be seen in eq. (6.2). Here, a pixel from the image grid is denoted as $I[i,j]$, with i and j being the indices and C_{present} contains all classes presented in the image.

$$\sigma_{\text{scene score}} = \prod_{c \in \mathbb{C}_{\text{present}}} \left(\frac{1}{I_{\text{height}} \times I_{\text{width}}} \sum_{i=1}^{I_{\text{height}}} \sum_{j=1}^{I_{\text{width}}} \mathbb{1}(I[i,j] = c) \right)^{-1} \quad (6.2)$$

This product of inverse probabilities is high if there are a lot of different classes and low if one class dominates the scene. So, we try to maximize this value as it indicates that many different types of categories are present. In the end, this scene variance score $\sigma_{\text{scene score}}$ is multiplied by the averaged interesting score from above. We use the camera poses, which have a combined score of $\sigma_{\text{interesting score}}$ and $\sigma_{\text{scene score}}$ above 0.8. This value was found experimentally and gave enough camera poses per scene while excluding shots with few individual objects.

As this check favors parts of the scene with a high density of interesting objects, we wanted to strike a balance between these and getting the scene covered. So, we convert a depth image with a side resolution of 15 pixels into a point cloud for each new camera pose and check its chamfer distance to all previous camera poses. For a full definition of the chamfer distance, see

Figure 6.4: Camera sampling in the 3D-FRONT dataset. Half-transparent red camera frustums depict the camera poses, only sampled viewing furniture. Rooms that do not contain any furniture do not get any cameras.

section 7.1.1. We first try to find a new camera pose with a maximum overlap of 0.5 meters. If this fails for 50,000 tries, we reduce this overlap to 0.25 meters. This sampling might fail in more miniature scenes, with insufficient space for more camera views.

Combining all of these measurements ensures that we look at the interesting parts of a scene and not at an empty white wall while also ensuring that we do not take 20 images of the same couch from the same viewpoint. An example of this can be seen in fig. 6.4, where the cameras are depicted as half-transparent red short camera frustums. These only appear in rooms with objects; additionally, they point at the interesting objects in the scene while covering the whole room.

Texture Sampling

Geirhos et al. have shown that convolutional neural networks often have a bias towards textures in contrast to the shape of an object [52]. An example of this behavior is shown in fig. 6.5. In it, the content image, namely a cat, is overlayed with a texture image of an elephant. Now, a network pre-trained on Image-net will classify this cat with the texture of an elephant as an Indian elephant. However, changing the texture does not change the perceived class for humans, as the shape is a more vital semantic indicator than the texture. Based on these results, we try to generate a scene in which the texture is

Texture image	Content image	Overlayed image
81.4% **Elephant**	71.1% **tabby cat**	63.9% **Elephant**
10.3% indri	17.3% grey fox	26.4% indri
8.2% black swan	3.3% Siamese cat	9.6% black swan

Figure 6.5: An example of the importance of texture randomization. Here, a cat is overlayed with an Indian elephant skin texture, producing a texture shape cue conflict. The network uses the texture to classify the instance, not the shape itself. The images and results are from Geirhos et al. [52]

not directly linked to the semantic category of an object. By removing the tail for the network to assign a category based on a texture, we force the network to evaluate the shape, bringing it closer to how humans evaluate our

Figure 6.6: BlenderProc can randomly switch objects' materials in a scene. The top left image shows the original image, and the other three shots show the same content but with randomized materials. This randomization removes any texture correlation between the image and the output categories.

surroundings. Therefore, narrowing the gap between how convolution neural networks interpret the world and how we do it.

For this, we use two different datasets collected from two public domain web pages, the first being ambientCG and the second one being poly haven [30, 181]. BlenderProc offers download scripts for both. From these, we sample a new material, relying on a bidirectional scattering distribution function (BSDF) shader [7]. This BSDF shader allows us to model real materials as closely as possible. These sampled materials are replacing existing materials randomly in the scene. Such a replacement can be seen in figure fig. 6.6, where the wooden floor is replaced by a brick, cobblestone, or black wood material, removing all texture cues a network could use and forcing it to learn the shape of an object.

Randomizing Material Properties

Instead of swapping the material, it is also possible to randomly change the properties of the existing materials. Figure 6.7 shows an extreme case where all surfaces are made reflective. Below, we show an instance where all materials are randomly made reflective with different roughness values assigned to the BSDF shader. These small changes increase the network's

Figure 6.7: Different shader properties: Only reflective surfaces are used in the top right, and on the bottom, a mix of reflective and rough surfaces is shown.

resilience against lighting and material changes in the real world.

6.2 SDFGen

In sections 4.1 and 4.2, we compress a scene based on the given TSDF values and available category labels. The generation of these TSDF spaces required the development of a new algorithm, as existing methods only worked for water-tight objects and not a combination of objects forming a scene.

6.2.1 TSDF volume

In section 4.1, we compress a scene based on a given TSDF volume. The goal is to calculate a full TSDF volume as mentioned in section 3.3. So, the distance to the closest surface has to be calculated for each voxel in our voxel grid $V : \Omega_v \mapsto [-\sigma_{\text{tsdf}}, \ldots, \sigma_{\text{tsdf}}]$ where $\Omega_v = \{0, \ldots, 511\}^3$. For a resolution of 512, we then have $512^3 = 134,217,728$ TSDF values to calculate. If we do this for $100,000$ scenes and assume that each TSDF value takes ten nanoseconds, we would need roughly 425 years to calculate all TSDF values, assuming each scene has $100,000$ triangles. In this test, we used a NVIDIA Geforce RTX 3090. So, our main objective is to make this algorithm as fast as possible while ensuring that each calculated distance is the smallest to all triangles in a scene.

Distance to a Triangle

We start by optimizing the distance calculation between a point to a triangle, as the most time is spent on it. If each scene only contains 100,000 triangles, we need to perform this calculation 10^{18} times. So, we define our set of 3D triangles as $\mathbb{T}$, which we map into our camera frame using our predefined transform $K_{\text{intrinsic}} \times K_{\text{extrinsic}}$ mentioned in section 3.3. We then calculate for each center point p of each voxel v the distance d_{tsdf} to the closest point on a triangle $t \in \mathbb{T}$ and truncate the absolute distance at a maximum value σ_{tsdf} with the clipping function defined in eq. (3.1), i.e.

$$d_{\text{tsdf}}[p] = \text{clip}\left(\min_{\forall t \in \mathbb{T}} \text{d}(p, t), \sigma_{\text{tsdf}}\right) \tag{6.3}$$

At first, we precompute ten vectors for each triangle t, namely the normal vector $\mathbf{n}$ of the triangle plane $\mathcal{P}$, and the vectors $\mathbf{n}^{\perp}$ that are orthogonal to the edges of t and lie inside $\mathcal{P}$. The last six vectors are the $\mathbf{n}^{+}$ and $\mathbf{n}^{-}$ that are parallel to the edges of t and lie inside $\mathcal{P}$. These vectors are displayed in fig. 6.8, where a triangle t with all corresponding normals is depicted.

To determine the distance d_{tsdf}, we first project our point p into the triangle plane $\mathcal{P}$. We check with the normals on the triangle's edges $\mathbf{n}^{\perp}$ if the point is inside the triangle t or outside. If the point is inside, the distance is simply the distance between the triangle plane $\mathcal{P}$ and the point p. If it is not, it is either closest to one of the triangle's edges or closest to one of its corners. As we need to check each edge with its corresponding $\mathbf{n}^{\perp}$ on its own, we immediately know which edge border normals $\mathbf{n}^{+}$ and $\mathbf{n}^{-}$ to use. If a point is now between two of these border planes, we know that the closest distance is now to one of these sides. If it is not, we need to check the distance to the two points on either side of an edge. This method is also shown in algorithm 1.

We evaluate the speed of our newly designed algorithm by comparing it to the open-source GeometricTool box designed by David Eberly [43]. So, we replaced our distance measurement algorithm presented in algorithm 1 with their suggested algorithm and can report that it now takes 8.586 ns for each distance calculation instead of 14.951 ns with their method. This reduction is a speed improvement of roughly 74 %. The main reason is that we use

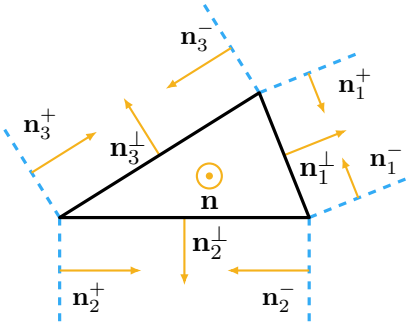


Figure 6.8: Triangle t with all normals in yellow that are used to efficiently compute $\text{d}(p, t)$. The corresponding orthogonal planes are shown in blue and dashed.

Algorithm 1 In this distance calculation algorithm we use three different variable colors, which correspond to the main, the edge and the border planes.

```
1:  procedure CalculateDistance(Point p)
2:      plnDist ← mainPln.distTo(p)
3:      for nr ∈ [1, 2, 3] do
4:          if edgePln[nr].distTo(p) < 0 then
5:              # Outside, check border planes
6:              if borderPln[nr][1].distTo(p) < 0 then
7:                  # Dist to left point
8:                  return sgn(plnDist) · ‖p − borderPln[nr][1].p‖2
9:              else if borderPln[nr][2].distTo(p) < 0 then
10:                 # Dist to right point
11:                 return sgn(plnDist) · ‖p − borderPln[nr][2].p‖2
12:             else
13:                 # Dist to edge
14:                 return sgn(plnDist) · edgeLine[nr].distTo(p)
        return plnDist
```

far fewer if-branches in our designed method, as in the approach by David Eberly, and each if-check on a modern CPU, where the branch prediction fails, takes much longer. Furthermore, we precalculate the normal vectors of the plane and the edges beforehand, removing some overhead.

After generating these 134 million TSDF values, they have to be saved to disc. However, each file would roughly have a size of 536 MB, meaning that a dataset of 100,000 scenes would need 53 TB of storage. As this is massively inefficient, we decided to quantize the data and store the TSDF values in signed 16-bit int values. This quantization only reduces the size by half, so we further used a "GZIP" compression ending up with 28.8 MB on average for a file. This size reduction allows us to store 246,000 files in 7.14 TB.

Flood Fill and Octree Splitting

We use some heuristics to speed up the TSDF calucation, avoiding the processing of the 135 million voxels. At first, we calculate the intersection between all triangles and voxels in our scene using an octree [106]. We then use a flood fill algorithm [142], starting from the voxels that intersect with at least one triangle. For each new voxel, we visit all the closest triangles of the already visited voxels in a $5 \times 5 \times 5$ area around the current voxel. If any of these voxels have an intersecting triangle, they are also added to the

collection. Based on this collection of triangles, we compute the closest one and its distance to the current voxel. We further improve the efficiency by setting it to the truncation threshold σ_{tsdf} if all visited neighboring voxels have their distance also set to the truncation threshold. Additionally, every voxel must get the correct sign to tell if it is inside or outside an object. We use the normal of each triangle to determine if the point is above or below our polygon indicating if the point is inside or outside of an object. However, one corner case does not work, which is illustrated in fig. 6.9. Here, the closest point is on the edge of the upper and lower triangle. However, for the inside-out check, the upper one is used. The given sign would be incorrect as the point lies below the upper triangle but is still outside the object. To avoid this wrong assignment, we go over all triangles with the same closest distance. If one of those triangles marks the point as outside, we also mark the voxel as outside and give it a positive sign.

Instead of comparing each of the 135 million voxels with the 20,000 triangles, we only need to check roughly 18.1 polygons per voxel. This speed-up makes the entire process over a thousand times faster and ensures that we only need a few minutes per scene. As this can be run efficiently in parallel on multiple CPUs distributed over many nodes, we can generate enough scenes to train a network in a few days.

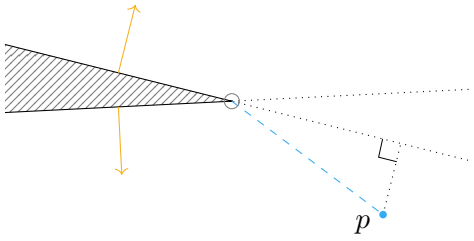


Figure 6.9: Two polygons in a top view, which share the circled edge. The distance to this edge in blue shows the distance to the closest point on both polygons. The area in gray shows where the object is filled.

6.2.2 Implicit TSDF

In this section, we create the training data for the implicit representation. As mentioned before, we do not have a strict grid guiding where all points are located. This increases the resolution and means that the network's architecture is the only limiting factor. Now to train such a network, a set of points with the corresponding TSDF value and semantic label are required. The goal is to sample points around all reachable surfaces in the scene. Each point knows how far it is away from the closest surface and the surface's category. Furthermore, we want to decompress the projection in the Z-direction so that objects are not compressed to the back of the cube.

We first create a TSDF volume grid with a low resolution of 128 using the method described in the last section, which is depicted in a) of fig. 6.10. The resulting TSDF volume grid is used to sample 400 so-called anchor points. Each anchor point has to be in the free space of our TSDF volume. It also cannot be too close to a surface, requiring that the distance is at least 10% of the truncation threshold σ_{tsdf}. With these anchor points, we can sample points on the surface of all triangles in our scene. Here, we require at least three anchor points to be visible for each new point. This visibility check ensures that only surface parts are included, which are reachable from the current camera position. So, complete triangles are discarded, while there might be triangles that are only used partially. An example might be a wall behind a couch, the parts of the triangle surrounding the couch are reachable, and the parts in close contact with the sofa are not. This check allows us to separate these surfaces without splitting them into different pieces.

Our next step corrects the Z-compression introduced through the camera projection $projectionMatrix$. The camera projection $projectionMatrix$ defined in section 3.4 warps the scene inside a cube with a side length of two. Here, objects further away from the camera experience are compressed while objects closer to the camera are expanded. We deal with this compression by first transforming the content into the range of zero to one and then applying a square root function in Z-direction, i.e.

$$p[2] = \sqrt{(p[2]+1)/2} \cdot 2 - 1 = \sqrt{2 \cdot p[2] + 2} - 1 \tag{6.4}$$

After that, we transform it back to a range of minus one to one. We do not change the values for X and Y, as these should align with the pixels in the color image. This decompression can be seen in d) in fig. 6.10. One disadvantage of this correction is that all flat surfaces are converted into curved ones. It does not hurt the scene compression, as a network can easily learn curved structures. Still, it makes the generation process more tricky, as we can no longer use the triangles for the distance calculation, as they are only defined by their points and not by their curved surface.

To create our final TSDF points, we take random surface points and add a random direction vector with a maximum length of $2 \cdot \sigma_{\text{tsdf}}$, see e) in fig. 6.10. For each resulting point, we need to estimate the closest distance to a surface, defined by our surface points. This is done by a nearest neighbor search, where we find the closest point on the surface for each randomly placed point.

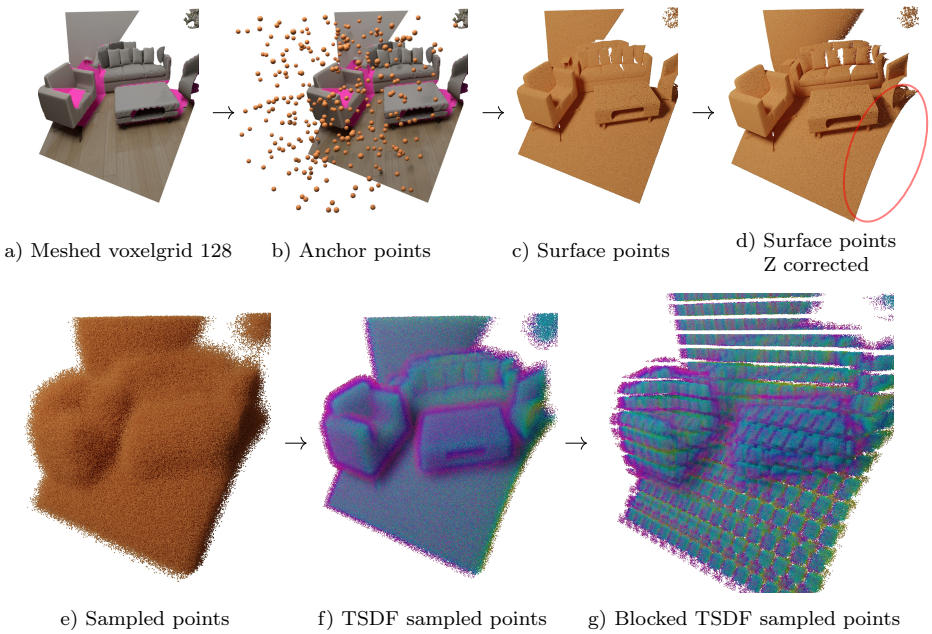


Figure 6.10: Creating a blocked TSDF point cloud around the reachable surface. First, the tool described in section 6.2.1 creates a low-resolution voxel grid a), and in its free space, anchor points are sampled b). These anchor points are used to determine if a sampled point on the surface is reachable c). We then remove the Z-compression d) and sample new points around our surface points e). Finally, we determine the distance to the curved surface f) and divide it into different blocks g).

We cannot use the surface point for the distance calculation as it is highly likely that another point on the surface will be closer than the starting point, and this distance vector will not be perpendicular to the surface. Thus we sample around this surface point on the corresponding curved triangle to find a closer point. This search is done until the distance converges, and the result is shown in f) in fig. 6.10. The category for this TSDF point is copied from the corresponding triangle. The result of this algorithm is a point cloud that stores the TSDF value and category per point shown in fig. 6.11.

The scene compression presented in section 4.2 needs a block of the current scene as input. So, we voxelize the scene with a resolution of 16 while expanding each block by a factor of b, including more of the neighboring voxels. This voxelized input is depicted in g) in fig. 6.10

6.3 Generated Datasets

With BlenderProc and SDFGen, we are equipped to generate the necessary synthetic training data for this work. This creation entails a detailed process of how the different kinds of data, such as images, volume grids, encoded

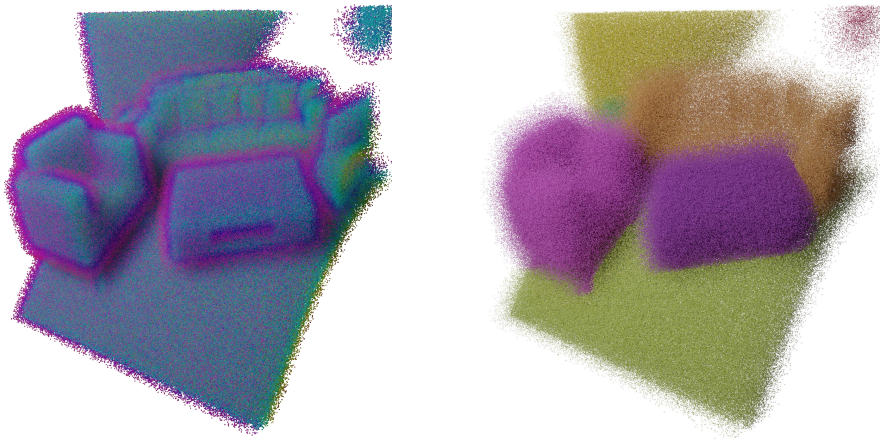


Figure 6.11: The output of the implicit TSDF generation is shown here. We visualize the TSDF values on the left, while on the right, the categories for each point are displayed. Each color represents its own category.

volume grids, and point clouds, are produced.

6.3.1 External Datasets

This work heavily relies on the SUNCG [143], the 3D-FRONT [48, 49], and the Replica dataset [145]. The SUNCG dataset as mentioned in section 2.1 was created by users of the Planner5D website and consists of 45,622 scenes, which the users themselves designed. These users used 2,550 unique objects to design various interior scenes. Additionally, we employ the 3D-FRONT dataset [48] as a training dataset, which consists of 6,812 different scenes and uses 16,563 unique objects from the 3D FUTURE dataset [49]. 3D-FRONT was designed by Alibaba for the purpose of synthetic data generation. In contrast, the SUNCG dataset models were designed to be useable in a web application, drastically limiting each object's detail level. These different design goals lead to very different average amounts of vertices and faces, shown in table 6.1. The amount of vertices and faces for the 3D FUTURE objects is nearly four times as high, creating much finer detail on each object and closing the *sim2real* gap further.

Both training datasets needed refinement to be useable in our complex data generation pipelines, as our methods require the normals of the objects to point outwards and the models to be watertight. This meant we fixed

Table 6.1: The SUNCG dataset [143] is compared here against the 3D-FRONT dataset [48], which uses the 3D FUTURE [49] models. While the SUNCG dataset has drastically more scenes, the object models have fewer vertices and faces on average, indicating a lower detail level compared to the 3D FUTURE models.

Dataset	SUNCG	3D-FRONT
Amount of unique objects	2,593	16,563
Amount of scenes	45,622	6,814
Average amount of vertices	2,509.4	9,338.4
Average amount of faces	1,917.1	10,382.0

38 models in the SUNCG dataset by hand to ensure that they work with our pipeline. For the 3D-FRONT dataset, we removed 4619 objects from the 16,563 objects as the normals of the objects did not consistently point outwards, therefore breaking the TSDF generation. A manual repair was impossible as too many objects were broken. Furthermore, we fixed the categories of 711 objects, which were mislabeled, e.g. a flower pot was labeled as a childbed. As the main objective is scene reconstruction with added annotations, we reduced the number of categories to ten, removing the need for a distinction between, for example, pillows and the couch they are on. This also means that we combined all types of chair, such as a lounge chair, book-chair, computer-chair, footstool, sofa stool, bed-end stool, stool, classic Chinese chair, and dressing chair, etc. in our chair category. Resulting in ten different categories: *void*, *table*, *wall*, *bath*, *sofa*, *cabinet*, *bed*, *chair*, *floor*, *lighting*, where we use *void* for all objects, which do not fit in the rest.

For the evaluation, we rely on the Replica dataset. In contrast to the other two datasets, it is a recording of the real world and does not consist of simulated objects in synthetic 3D scenes. As it only has 18 different scenes with 35 rooms, it is impossible to use it to train our method. However, it has enough variety for a real-world evaluation, allowing us to test our methods in a realistic setting. These 18 rooms have been recorded with a custom build rig with an IR projector and four cameras, the resulting images have been fused with KinectFusion [113], and the resulting TSDF volume was converted to a mesh with the marching cubes algorithm [100].

6.3.2 2D Images

Using BlenderProc, we need to create six different types of image sets. At first, we focus on creating color images with the SUNCG dataset. The resulting dataset is called $\mathbb{I}_{\text{SUNCG}}$. In order to render any information in a scene, a camera has to be positioned in the SUNCG rooms. We rely here on the SUNCGToolbox [143], which samples camera poses at a camera height of 1.55 meters, while the horizontal opening angle of the camera is fixed to 57.2958°. Furthermore, the camera's tilt is fixed to 78.69°, making the scene reconstruction problem easier as the floor always has the same slope in the

final reconstruction. With the fixed hyperparameters, new camera poses are sampled. Each pose gets a scene coverage score $\sigma^{SUNCG}_{\text{scene score}}$, which is defined in eq. (6.5) [143].

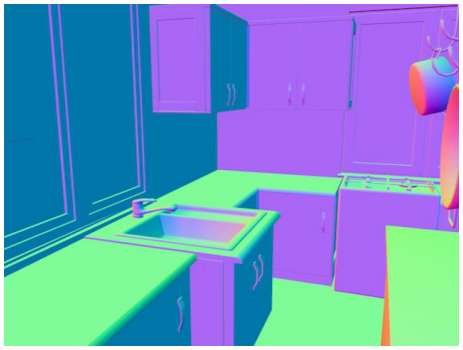
$$\sigma^{SUNCG}_{\text{scene score}} = \sum_{c \in C_{\text{present}}} \log\left(\frac{\sum_{i=1}^{I_{\text{height}}} \sum_{j=1}^{I_{\text{width}}} \mathbb{1}(I[i,j] = c)}{0.1 \times I_{\text{height}} \times I_{\text{width}}}\right) \tag{6.5}$$

This $\sigma^{SUNCG}_{\text{scene score}}$ is the sum over all instances c in the current scene C_{present}. For each instance c, the logarithm of the relative amount of pixels in the image I is divided by the number of 10% of all pixels. Each image I has a resolution of $I_{\text{width}} = 640$ and $I_{\text{height}} = 480$, and only camera poses with more than three instances in the camera view are considered [143].

Using this scene coverage score, we generate $127,706$ camera poses. For each camera pose, we render with BlenderProc the color images and call this set of images $\mathbb{I}_{\text{SUNCG}}$. In the same run, we also create the surface normal images and call them $\mathbb{I}^{\perp}_{\text{SUNCG}}$. One of the generated color images with the corresponding surface normal image is shown in fig. 6.12.

We also generated camera poses in the 3D-FRONT dataset, now using BlenderProc to sample poses inside the rooms. For a full definition of the camera sampling, see section 6.1.4, especially eq. (6.1) and eq. (6.2). Using this scene variance score $\sigma_{\text{scene score}}$, we can generate $84,508$ color images.

A rendered color image of a SUNCG scene. The glass windows are frosted to avoid rendering the outside as well.



The corresponding surface normal image.

Figure 6.12: A color image with its corresponding surface normal image in a SUNCG scene is shown here. The color image was rendered with BlenderProc and will be the input to the scene reconstruction network.

Table 6.2: The generated different image datasets are defined here. They are split into three different types of images: color, surface normals, and color images with swapped textures. They are generated for the SUNCG dataset, the 3D-FRONT dataset, and the Replica dataset.

Dataset	SUNCG	3D-FRONT	Replica
Color image $\mathcal{I}$	$\mathbb{I}_{\text{SUNCG}}$	$\mathbb{I}_{\text{3D-FRONT}}$	$\mathbb{I}_{\text{Replica}}$
Surface normal image $\mathcal{I}^{\perp}$	$\mathbb{I}^{\perp}_{\text{SUNCG}}$	$\mathbb{I}^{\perp}_{\text{3D-FRONT}}$	$\mathbb{I}^{\perp}_{\text{Replica}}$
Textured swapped color image $\mathcal{I}^{\circlearrowleft}$	–	$\mathbb{I}^{\circlearrowleft}_{\text{3D-FRONT}}$	–

These generated images define our 3D-FRONT color image set $\mathbb{I}_{\text{3D-FRONT}}$, and for each color image, we generate four different versions with different sampled textures from the ambient CG textures [30]. So, this generation produces $422,540$ images for the swapped texture image set $\mathbb{I}^{\circlearrowleft}_{\text{3D-FRONT}}$ for the 3D-FRONT dataset. Each can be mapped to one of the $84,508$ surface normal images $\mathbb{I}^{\perp}_{\text{SUNCG}}$.

To evaluate our approach, we generate 360 color images $\mathbb{I}_{\text{Replica}}$ and surface normal images $\mathbb{I}^{\perp}_{\text{Replica}}$ on the Replica dataset. This generation leads to 20 images per scene. We generate a point cloud from the depth image for each camera pose and compute the distance to all existing camera poses, rejecting those too close to existing ones. This removal allows us to spread the camera poses around the whole scene while still focusing the attention on the more interesting objects. We rely on the same sampling variance score as for the 3D-FRONT dataset. The five created image-based datasets are depicted in table 6.2.

6.3.3 TSDF Scene Information

We now need the corresponding 3D scenes represented as TSDF information for the color image sets defined in the past section. Using SDFGen, we use the camera poses defined before for SUNCG and 3D-FRONT and generate the corresponding TSDF information based on the meshes provided in both datasets. For the TSDF calculation, we use the projection matrix $K_{\text{intrinsic}}$

defined in eq. (3.3), where we set the near plane h_{near} to 1 meter and the far plane h_{far} to 4 meters. This setting gives a sufficient range for reconstructing indoor spaces.

The SUNCG dataset camera poses are converted with SDFGen into voxel grids, containing the TSDF distance to the closest surfaces. The set of $127,706$ voxel grids is named $\mathbb{V}_{\text{SUNCG}}$, where each voxel grid has a resolution of 512^3, resulting in roughly 135 million values per scene, which are stored as detailed in section 6.2.1. With the autoencoder defined in section 4.1, we can compress these voxel grids to encoded voxel grids with a resolution of $32^3 \times 64$, which are used as the scene reconstruction regression targets. These compressed and encoded voxel grids are called $\mathbb{V}^{\text{F}}_{\text{SUNCG}}$.

For 3D-FRONT, we use the implicit encoding strategy as compression, which is described in section 6.2.2. This implicit compression would require we sample the TSDF distance for arbitrary points during training. As this is technically infeasible, we sample first points around the surface, for which the closest TSDF distance and category of the nearest surface are saved. We do this for $2,000,000$ points for $89,029$ scenes. The resulting set of point clouds with TSDF value and category label is called $\mathbb{T}_{\text{3D-FRONT}}$.

For the evaluation of our approach, we rely on the Replica dataset. We use the 360 camera poses defined for the color images $\mathbb{I}_{\text{Replica}}$ and surface normal images $\mathbb{I}^{\perp}_{\text{Replica}}$ to create the TSDF volume with a resolution of 512^3 $\mathbb{V}_{\text{Replica}}$ and the $2,000,000$ TSDF distance points with semantic segmentation $\mathbb{T}_{\text{Replica}}$ for each scene.

Table 6.3: The two different input modalities, firstly a TSDF voxel grid using the SUNCG dataset and secondly a point cloud with TSDF and category values for the 3D-FRONT dataset, are shown here. Both are processed in a block-wise fashion to produce an encoded volume. For the Replica dataset, we only need the voxel grid and the point cloud, as a compressed form is unnecessary.

Dataset	SUNCG	3D-FRONT	Replica
TSDF voxel grid V	$\mathbb{V}_{\text{SUNCG}}$	–	$\mathbb{V}_{\text{Replica}}$
TSDF and category point cloud T	–	$\mathbb{T}_{\text{3D-FRONT}}$	$\mathbb{T}_{\text{Replica}}$
Encoded TSDF volume V^{F}	$\mathbb{V}^{\text{F}}_{\text{SUNCG}}$	$\mathbb{V}^{\text{F}}_{\text{3D-FRONT}}$	–

Chapter seven

Experiments

This chapter will highlight the experimental evaluation of the methods presented in this work. We will start by evaluating the scene compression approaches and look closely at their strengths and weaknesses. After this, we first focus on a TSDF grid-based 3D scene reconstruction followed by a similar architecture where the focus is to include semantic information. This inclusion is done by implicitly representing the input space, allowing the querying of any position for a distance and semantic value while this implicit space is still encoded in blocks.

All our methods presented in this work use TensorFlow 2.6 and are trained on the cluster in the Institute of Robotics and Mechatronics of the German Aerospace Center (DLR). This cluster contains 46 different GPUs ranging from 8× NVIDIA Geforce GTX 1080 to 16× NVIDIA Geforce RTX 3090.

7.1 Scene Compression

To evaluate the entire pipeline from image to 3D scene reconstruction, we first start by evaluating the scene compression. In particular, the TSDF volume grid, which is compressed with an autoencoder, and afterward the gridless implicit space representation, which relies on a simple neural network.

Firstly, we show that a voxelized TSDF volume can be compressed by using an autoencoder. Each 16^3 block gets compressed into a 64 long latent vector, later used as a reconstruction target. Secondly, we demonstrate how to use an implicit representation for encoding such a scene, where arbitrary query positions are possible instead of fixed query positions like on a grid. This implicit representation removes the resolution requirements set by using a voxelized TSDF volume. So, the resolution is bound only by the neural network's capacity to compress the scene. Furthermore, it allows the inclusion of additional information, such as semantic information, into the latent encoding.

7.1.1 Metrics

A general metric over TSDF volume grids is the IOU, which is the same as the Jaccard similarity coefficient [82]. Here, we divide the intersection where all target and output values have the same sign through the union of the same. We define the IOU between the prediction d_{tsdf} and ground truth y_{tsdf} of the TSDF values as eq. (7.1).

$$IOU\left(y_{\text{tsdf}}, d_{\text{tsdf}}\right) = \frac{\|(y_{\text{tsdf}} \leq 0) \cap (d_{\text{tsdf}} \leq 0)\|}{\|(y_{\text{tsdf}} \leq 0) \cup (d_{\text{tsdf}} \leq 0)\|} \tag{7.1}$$

This IOU is a good indication of the reconstruction ability, as the sign in a TSDF volume indicates if we are inside or outside of an object. The goal is to have a high IOU, which indicates that the occluded space was correctly estimated for points in the grid. Furthermore, the precision Prec, recall Rec, and F-score are evaluated; all three are depicted in eqs. (7.2) to (7.4).

$$\text{Prec}(y_{\text{tsdf}}, d_{\text{tsdf}}) = \frac{\|(y_{\text{tsdf}} \leq 0) \cap (d_{\text{tsdf}} \leq 0)\|}{\|(d_{\text{tsdf}} \leq 0)\|} \tag{7.2}$$

$$\text{Rec}(y_{\text{tsdf}}, d_{\text{tsdf}}) = \frac{\|(y_{\text{tsdf}} \leq 0) \cap (d_{\text{tsdf}} \leq 0)\|}{\|(y_{\text{tsdf}} \leq 0)\|} \tag{7.3}$$

$$\text{F-Score}(y_{\text{tsdf}}, d_{\text{tsdf}}) = \frac{2 \times \text{Prec}(y_{\text{tsdf}}, d_{\text{tsdf}}) \times \text{Rec}(y_{\text{tsdf}}, d_{\text{tsdf}})}{\text{Prec}(y_{\text{tsdf}}, d_{\text{tsdf}}) + \text{Rec}(y_{\text{tsdf}}, d_{\text{tsdf}})} \tag{7.4}$$

The precision indicates how well the correctly predicted space is part of the

total predicted space, including the wrong predictions. So, a high precision means that not much of the free space was wrongly classified as occluded. In contrast to precision, recall measures how much of the occluded area was correctly detected. This recall value is crucial for any application of this method, as missing objects in the reconstruction can lead to damaging collisions, while wrongly predicting free space as occluded only limits the possible range of movements of a robot in such a space. The harmonic mean between this recall and precision is called F-Score. It combines the measurement of the occluded area with attention to the correctly predicted free area, see eq. (7.3).

One last metric to evaluate the TSDF values in the grid is the ⌀L1 loss:

$$\text{⌀L1} = \|d_{\text{tsdf}} - o\| \tag{7.5}$$

which indicates how far away each TSDF value from the ground truth is. This loss is the absolute average difference over the 32^3 values per block, and we only perform this check on the voxelized TSDF grid. Minimizing this value to zero is the overall goal of the compression method. The ⌀L1 and IOU indicate the performance of the compression method. A high IOU shows that the method can reconstruct the overall surface, while a low ⌀L1 loss shows that each value in the block is well reconstructed.

In order to assess the surface reconstruction, we first have to decode the surface of the scene and then calculate the distance between the ground truth mesh and the en- and decoded mesh. This mean distance is calculated by averaging the distance of the ground truth mesh vertices $\mathbb{P}_{\text{GT}}$ to the closest point in the predicted mesh $\mathbb{P}_{\text{pred}}$, as shown in eq. (7.6).

$$\text{CD}_{\text{GT}}(\mathbb{P}_{\text{GT}}, \mathbb{P}_{\text{pred}}) = \frac{1}{\|\mathbb{P}_{\text{GT}}\|} \sum_{y \in \mathbb{P}_{\text{GT}}} \min_{x \in \mathbb{P}_{\text{pred}}} \|x - y\|_2 \tag{7.6}$$

$$\text{CD}_{\text{pred}}(\mathbb{P}_{\text{GT}}, \mathbb{P}_{\text{pred}}) = \frac{1}{\|\mathbb{P}_{\text{pred}}\|} \sum_{x \in \mathbb{P}_{\text{pred}}} \min_{y \in \mathbb{P}_{\text{GT}}} \|x - y\|_2 \tag{7.7}$$

$$\text{CD}(\mathbb{P}_{\text{GT}}, \mathbb{P}_{\text{pred}}) = \text{CD}_{\text{GT}}(\mathbb{P}_{\text{GT}}, \mathbb{P}_{\text{pred}}) + \text{CD}_{\text{pred}}(\mathbb{P}_{\text{GT}}, \mathbb{P}_{\text{pred}}) \tag{7.8}$$

The CD_{GT} metric is part of the chamfer distance CD [2], defined in eq. (7.8). We are more interested in this chamfer distance CD_{GT} as we concentrate

more on how well the predicted mesh reconstructed the ground truth mesh. The reason for this is similar to the focus on the recall instead of the precision, as we are more interested in correctly reconstructing the occluded space than accidentally predicting free space as occluded. We, therefore, omit predicted parts which are not in the ground truth; one extreme example of this might be an entire predicted chair, which is not in the ground truth data. As the overarching goal of this work is an application in robotics, missing occluded areas is much worse than occluding more areas than in the ground truth. We still evaluate the correct occlusion rate with the IOU, the prediction-focused chamfer distance $\mathrm{CD}_{\mathrm{pred}}$, and the full chamfer distance CD. This chamfer distance focused on the ground truth $\mathrm{CD}_{\mathrm{GT}}$ can also be viewed as an averaged Hausdorff distance [68], in which the maximum distance between one point cloud to another is searched. The chamfer distance for the prediction $\mathrm{CD}_{\mathrm{pred}}$ is defined in eq. (7.7). These metrics will enable the validation of our methods and will show what changes enable them to perform that well.

7.1.2 Truncated Signed Distance Fields

We start by evaluating the performance of compression methods used on the TSDF volume grids and validate how the changes we have done improve the performance.

Setup

The autoencoder described in section 4.1 is trained on the SUNCG TSDF voxel grids $\mathbb{V}_{\mathrm{SUNCG}}$ described in section 6.3.3 . Here, we use a subset of 270 different TSDF voxel grids, as each contains $32^3 = 32768$ blocks, even though we remove 99% of the filled and empty blocks. This removal ensures a better distribution between blocks with a surface or which are filled or empty. We train the autoencoder for 45 epochs with an Adam Optimizer with a learning rate of 0.0003 and a batch size of 256. As data augmentation, we rely only on using a random flip in one of the three axes of the input block, which gets encoded. This random flipping mainly increases the variability in the data and stabilizes it against unusual encoding targets.

Table 7.1: Results of our TSDF grid compression method evaluated on the SUNCG dataset. These results are obtained by evaluating the voxel-based metrics, which show that the combination of our proposed improvements has the best result. Even though the average L1 distance is smaller if our $\mathcal{L}_{\text{tsdf}}$ is removed, the surface performance measured by the recall and the IOU is more important.

Method	Precision	Recall	F-Score	⌀IOU	⌀L1
Without $\mathcal{L}_{\text{tsdf}}$	99.59	99.75	99.67	99.35	**6.128e-5**
Without added border	99.31	**99.90**	99.60	99.22	10.74e-5
Without random flip	**99.72**	99.61	99.66	99.33	9.371e-5
default	99.64	99.77	**99.71**	**99.42**	9.521-5

Results

In order to evaluate the TSDF grid compression, we train several networks and evaluate these on the SUNCG dataset; we use 68 scenes for validation, giving us around 2.2 million blocks to compress correctly. Our first training is the default configuration proposed in section 4.1, which relies on our Gaussian loss defined in eq. (4.1). As seen in the last row of table 7.1, it achieves an IOU of 99.42% and an F-Score of 99.71%.

Without this Gaussian loss $\mathcal{L}_{\text{tsdf}}$, the IOU performance drops to 99.35% and the F-Score to 99.67%. For the IOU, this increases the error from 0.58% to 0.65%, which means that not using our proposed Gaussian loss increases the error by 12%. However, using the Gaussian loss increases the overall ⌀L1 loss over all grid points. Nevertheless, our surface reconstruction has a higher priority than the correctness of voxels far from the surface. If instead of relying on a 30^3 input TSDF voxel grid, we reduce it to the same size as the output of 16^3, we can see how the IOU is even more affected than when removing the Gaussian loss $\mathcal{L}_{\text{tsdf}}$. Then the IOU drops from 99.42% to 99.22%, corresponding to an error increase of 34.4%. The reason for this is the missing continuity information on the borders of the 32^3 blocks. Lastly, we check the effects of removing the data augmentation of randomly flipping the TSDF voxels. Here, the IOU and F-Score changes are comparable to

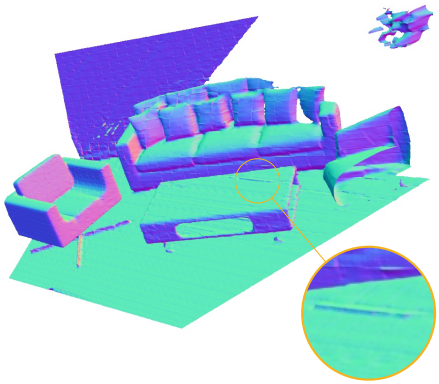
Table 7.2: Our TSDF grid compression method evaluated with surface-based metrics on the SUNCG dataset. These results show that our proposed approach improves through the added suggestions, resulting in an overall surface error of less than 1.417 millimeters. Especially the random flipping drastically improves the results.

Method	CD_{GT}	CD_{pred}	CD
Without $\mathcal{L}_{tsdf}$	0.001537	0.001279	0.001408
Without added border	0.001860	0.001681	0.001770
Without random flip	0.001800	0.001597	0.001699
default	**0.001417**	**0.001258**	**0.001337**

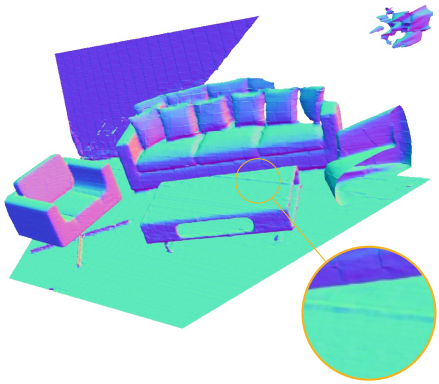
removing the Gaussian loss $\mathcal{L}_{tsdf}$, but the precision slightly improves. One reason might be that the training data more closely resembles the validation data, as the most prominent effect is on the blocks, where the surface points away from the camera. These blocks are under-represented in the training data and are more represented when randomly flipping the blocks, resulting in a possible performance drop of blocks facing the camera.

In table 7.2, the chamfer distances for the default method and three ablation results are presented. The results are similar to the voxel grid-based evaluation. So, adding a border to the autoencoder brings the biggest improvement overall. On the other hand, the gain of the Gaussian loss is more focused on the ground truth-focused chamfer distances CD_{GT}. However, the overall error to the ground truth is below 2 millimeters and is far better than any application we desire to work on will need. For the final compression of our proposed method, the error is even below 1.5 millimeters, as seen in the last row of table 7.2.

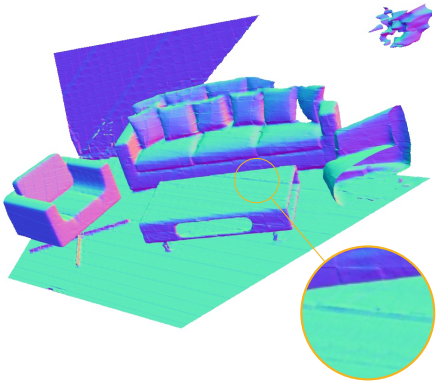
Besides evaluating the quantitative results, we also provide qualitative results on the scene presented in the main overview in fig. 1.2. We focus on our proposed default method and its three ablation results, as seen in fig. 7.1. The scene presented here was first TSDF voxelized with SDFGen and then compressed and decompressed with the different autoencoders presented in tables 7.1 and 7.2. On the decoded TSDF voxel grid, we used a marching



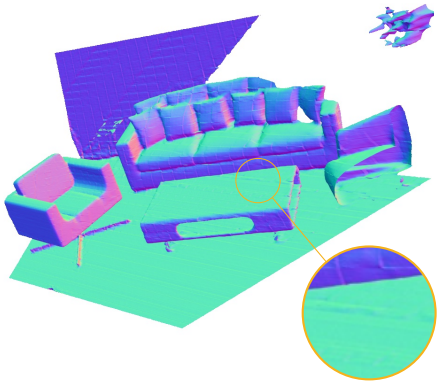
No surface loss $\mathcal{L}_{\text{tsdf}}$ is used in this scene compression result



The random flipping as augmentation is removed here



The additional borders are not used, reducing the input to 16^3



Our default scene compression method

Figure 7.1: The 3D scene depicted in the overview figure is blockwise compressed and decompressed; see fig. 1.2. We display the effects of our proposed approaches and show how using our introduced methods improves the results. Removing the suggested Gaussian loss as shown in the top left or not adding data augmentations in the form of random flipping the TSDF blocks introduces visible artifacts in the reconstruction. The same can be said for reducing the size of the input space from 30^3 to 16^3.

cubes algorithm to extract the surface, which we rendered in BlenderProc, focusing on the surface normals of the reconstruction to see every minor detail. In particular, the zoomed-in bit on the coffee table shows evident artifacts on the boundary of the voxel, which are lessened when using the final method.

These results show that it is possible to compress a 512^3 TSDF volume by a factor of 64 down to $32^3 \times 64$ without introducing major artifacts or losing finer details. For comparison, JPEG, a standard image compression method, offers a compression factor between 10 and 15 without more significant losses [162]. Even more recent image compression methods, such as AVIF [25] and HEVC [80], only offer compression factors between 20 and 30. This TSDF grid scene compression enables us to reconstruct full 3D scenes at a high resolution, as seen in section 7.2.1.

7.1.3 Implicit representations

In this subsection, we want to evaluate the performance of the surface reconstruction and classification of compressing a scene with an implicit representation and give evidence for our design choices.

Setup

We use 100 different scenes recorded in the 3D Front dataset $\mathbb{T}_{\text{3D-FRONT}}$, described in section 6.3.3, each containing a point cloud with 2.000.000 points. Each point has a corresponding TSDF value and category assigned to it. This point cloud gets split into a grid with a resolution of 16, resulting in $16^3 = 4096$ blocks, roughly 1.600 blocks of those need to be compressed. The rest is either completely filled or empty. This splitting process is done by a small program called TSDFBlocker, allowing us to define overlapping blocks, as defined in fig. 4.8. The data is then loaded and forwarded through the compression network defined in section 4.2.1.

We use an Adam optimizer with a learning rate of 0.00627, a batch size of eight different blocks, and 2048 points per block. Each point is considered a single batch with a shared latent vector l, giving us a total batch size of $8 \times 2048 = 16384$. After 250 iterations of optimizing a batch of latent vectors

and data blocks, the training is done, and no further changes can be expected. The validation is done on 500 scenes from the 90,000 generated scenes from the 3D-FRONT dataset $\mathbb{T}_{\text{3D-FRONT}}$, which are not in the training set.

Hyperparameter Sampling

One crucial step of this network design is the hyperparameter choice. The reason for this is the needed time to compress an entire scene in the end, as highlighted in section 4.2.3. This compression can take several years if not done correctly. So, to ensure that we have the best hyperparameter set possible for the lowest computational demand, we set up a C-MAES black box sampler [66] and trained 7.259 networks. This vast amount of networks is only possible as the training of one network does not take longer than 2 hours. To ensure a quick validation, we used five different scenes to validate the compression score in this hyperparameter tuning.

The final design choice is presented in section 4.2.1. In order to achieve this, we show in fig. 7.2 how certain hyperparameters affect the validation performance. In the top row, the latent dimension of the latent vector l is changed. We report the IOU and class accuracy for a latent dimension of 128, 256, and 512. The difference for the IOU is greater than for the class accuracy, indicating that the segmentation can be done without a high latent dimension. As the results for 256 and 512 are comparable, we evaluate the 256 variant further in table 7.3. In the second row of fig. 7.2, we evaluate the number of hidden layers on the network accuracy, while one layer is not enough. A third layer does not provide any benefit under the current limitation of total multiplications per latent optimization. Lastly, we show how big the first layer has to be to get satisfactory results. In this scenario, it is also clear that 512 is the optimum number of weights. The same was done for the size of the second layer, the size of the Fourier mapping, the learning rate for the generator, and the latent optimization, the number of optimization steps for the generator, and the latent vector, and the number of weights in the hidden layer before the classification is done. Lastly, we also tried SIREN [141] as an activation function but did not find any improvement over RELU, especially as SIREN is considerably more computationally heavy.

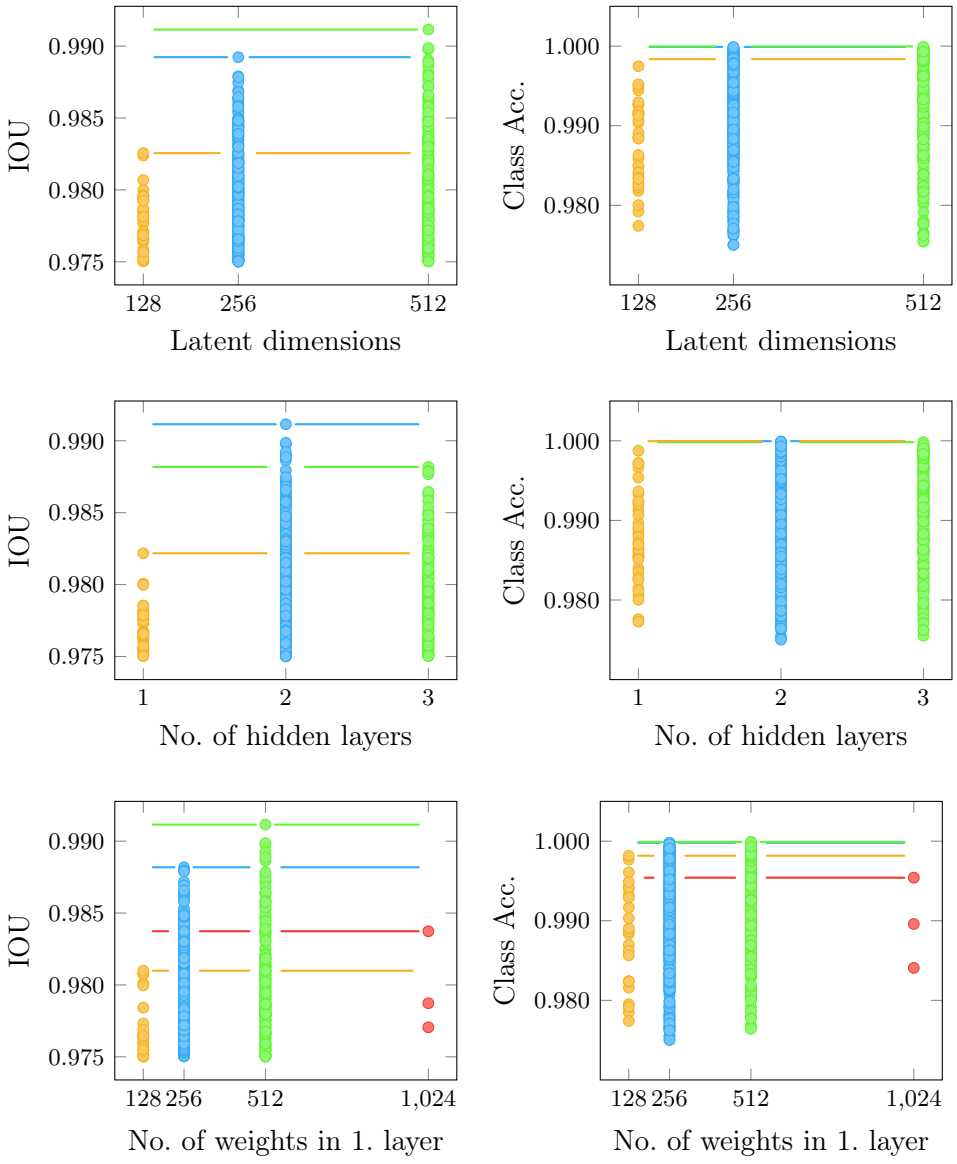


Figure 7.2: Results of our hyperparameter optimization. Each row contains a different hyperparameter. Starting from the top to the bottom, we show the dimension of the latent vector l, the number of layers in the network, and the number of weights in the first layer. On the left, the IOU is displayed on the validation set, and the corresponding classification result is on the right. For the IOU, the results show how certain hyperparameters limit the possible accuracy, while for the class accuracy, the effect is neglectable.

Table 7.3: Results of our implicit segmentation TSDF method are evaluated on the 3D-FRONT dataset. They show that our proposed changes improve the outcome, resulting in a IOU reconstruction of 97.93% and a classification accuracy of 99.67%. If the class accuracy cannot be computed, we denote this with X.

Method	Precision	Recall	F-Score	⌀IOU	⌀L1	Class Acc.
Without $\mathcal{L}_{\text{surface}}$	95.76	95.49	95.62	91.74	0.00339	99.57
With $\mathcal{L}_{\text{tsdf}}$	97.08	97.53	97.30	94.81	0.00304	99.65
Without $\mathcal{L}_{\text{boundary}}$	98.98	98.94	98.96	97.92	0.00149	99.65
Without $\mathcal{L}_{\text{gradient}}$	98.93	98.95	98.94	97.91	0.00133	97.73
With $\|l\| = 256$	98.83	98.76	98.79	97.63	0.00195	98.94
No categories	98.82	98.75	98.78	97.68	**0.00129**	X
default	**99.00**	**98.96**	**98.98**	**97.93**	0.00130	**99.67**
TSDF autoencoder	99.59	99.75	99.67	99.35	0.00061	X

Results

The evaluation of the implicit TSDF surface compression and classification is done as described in the setup on 500 validation scenes. We first create 16^3 latent vectors for all 16^3 blocks in the 500 scenes for a given network. The network described in section 4.2, is called `default` in the following. Based on this default network, we made some ablations to highlight the strength of our design choices. Our first change is the removal of the surface loss $\mathcal{L}_{\text{surface}}$, which means that all TSDF values are only optimized after a simple L1 loss. Compared to the default method, this decreases the average IOU in a 512^3 voxel grid spanning the scene from 97.93% down to 91.74%, as displayed in table 7.3. The error increases by a factor of four as it goes up from 2.07% to 8.26%.

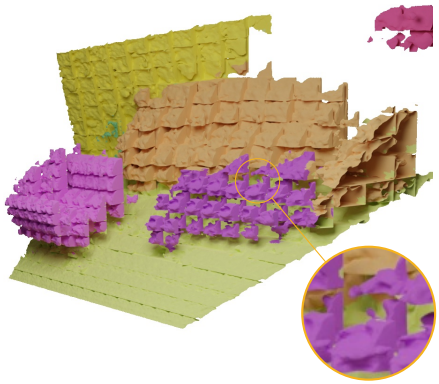
Similar results are obtained if we switch the $\mathcal{L}_{\text{surface}}$ with the Gauss loss $\mathcal{L}_{\text{tsdf}}$ described in eq. (4.1). The IOU performance is better than without any surface loss but considerably worse than with our proposed exponential loss $\mathcal{L}_{\text{surface}}$. On the other hand, if we remove the boundary loss $\mathcal{L}_{\text{boundary}}$ or the

Table 7.4: The implicit compression method is evaluated by comparing the reconstructed mesh to the ground truth mesh. We compare the chamfer distance on the ground truth CD_{GT}, the prediction CD_{pred}, and the combined one CD. These results show that our changes improved the surface reconstruction to a difference between the ground truth and the compressed version of 8.7 millimeters.

Method	CD_{GT}	CD_{pred}	CD
Without $\mathcal{L}_{surface}$	0.017750	0.033747	0.025748
With $\mathcal{L}_{tsdf}$	0.011579	0.026540	0.019060
Without $\mathcal{L}_{boundary}$	0.009075	0.009644	0.009360
Without $\mathcal{L}_{gradient}$	0.009129	0.010270	0.009699
With $\|\|l\|\| = 256$	0.008987	0.023282	0.016134
No categories	0.008768	0.010861	0.009814
default	**0.008283**	**0.0091423**	**0.008712**
TSDF autoencoder	0.001715	0.001577	0.001646

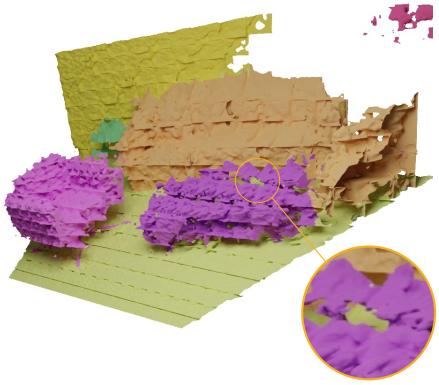
gradient loss $\mathcal{L}_{gradient}$, the changes are less pronounced. We also show the results for the scenario in which we increase the possible compression further by reducing the latent dimension to 256. This decreases the F-Score from 98.98% down to 98.79 an 18.6% increase in error. On the other hand, if we train a method without the possibility of classifying the input points into our ten different classes, we obtain a marginally better average L1 distance but lose the option to classify the output. We further evaluate how the TSDF grid-based autoencoder compares to the implicit method in the last row of table 7.3. The compression performance is considerably higher with the autoencoder than with the implicit method, but it does not provide an easy way to represent a classification label. The advantage of the implicit method is the unbound possible precision, which in the TSDF grid-based approach is bound by the input's resolution.

In table 7.4, the created voxel grid was compared to the ground truth. However, in table 7.4, the reconstructed mesh created with the marching

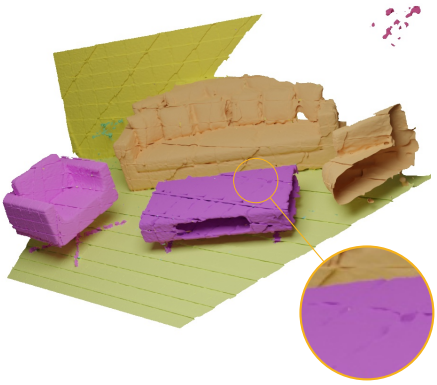


No surface loss $\mathcal{L}_{\text{surface}}$ is used in this scene compression result

The latent space is limited to 256



This example uses the Gauss loss $\mathcal{L}_{\text{tsdf}}$ instead of the $\mathcal{L}_{\text{surface}}$



Our default scene compression method

Figure 7.3: The 3D scene depicted in the overview figure is blockwise compressed and decompressed; see fig. 1.2. We display the effects of our proposed approaches and show how using our introduced methods improves the results. Removing the suggested Gaussian loss as shown in the top left or not adding data augmentations in the form of random flipping the TSDF blocks introduces visible artifacts in the reconstruction. The same can be said for reducing the size of the input space from 30^3 to 16^3.

cubes algorithm is compared to the ground truth mesh, focusing more on the mesh than the filled and empty space. The same ablation methods as before are used. The chamfer distance CD on the reconstructed surface has an error of 8.7 millimeters in the default setting, while without any surface loss, the error goes up to 25.7 millimeters. The Gauss loss $\mathcal{L}_{\text{tsdf}}$ reduces this error from 25.7 millimeters down to 19.0 millimeters but is still double as high as the exponential loss $\mathcal{L}_{\text{surface}}$. The focus on the surface also shows the importance of the boundary and gradient loss, as both increase the error roughly by a millimeter. Reducing the latent dimension down to 256 doubles the chamfer distance from 8.7 to 16.1 millimeters. These results show how vital our changes are to ensure that the scene is well-compressed to offer a valid starting point for the scene reconstruction task.

Besides a quantitative evaluation, we offer in fig. 7.3 a qualitative evaluation, where we compress and decompress the scene presented in fig. 1.2. The surfaces in this figure are colored-coded based on the encoded category. Without the surface loss, the reconstructed mesh dramatically decreases in quality, especially the couch in the background losses all details, as seen in the top left of fig. 7.3. Reducing the latent dimension introduces flying artifacts in the scene and visible surface imperfections on the objects. While performing better as no surface loss, the Gauss loss still focuses too much on the neighboring area instead of directly on the points closest to the surface. If this had been used for the scene reconstruction task, it would have been the upper limit of the output and, therefore, considerably degraded the approach's performance.

7.2 Scene Reconstruction

In this section, we will evaluate the 3D scene reconstruction from single color images. We focus here on two approaches, relying on the same tree net architecture, as described in section 5.1. At first, we will focus on the volume grid-based 3D scene compression as encoding, named SVR-GC. Then we show how the results change when we switch to the scene reconstruction method using the implicit and semantically segmented surface representation, called SVR-IC. We then compare our approaches to the related work of Total3DUnderstanding [114] by Nie et al. and P3DSR [26] by Dahnert et al.. Finally, we will evaluate this on images in the wild.

7.2.1 TSDF Volume Grid Compression (SVR-GC)

Setup

Our novel tree network relying on the TSDF volume-based compression SVR-GC is trained on $128,005$ images $\mathbb{I}_{\text{SUNCG}}$ from the SUNCG dataset with the corresponding encoded TSDF volumes $\mathbb{V}^{\mathbb{F}}_{\text{SUNCG}}$. During training, we do not use the generated surface normals, only the ground truth ones. The model is trained with the Adam optimizer using a learning rate of $1e{-}4$ for $200,000$ steps with a batch size of four.

Results

We evaluate our trained method on $1,000$ images from the SUNCG dataset and 360 images from the Replica dataset. As highlighted in section 6.3.1, the limited amount of scenes in the Replica dataset restricts the number of possible images. We only rely on the color image in these experiments using our trained U-Net for the surface normal generation. We first evaluate our method on the SUNCG dataset and report the volume-based results in table 7.5 and the surface-based results in table 7.6.

Besides the results for our default method SVR-GC shown in the second-to-last row, we also show some ablation results. Our default SVR-GC achieves

Table 7.5: Results of SVR-GC on the 1,000 scenes from the SUNCG dataset. This table contains the volume-based metrics evaluated on the 512^3 TSDF volume. Our default method performs best in IOU and recall, which we deem most important in this work.

Method	Precision	Recall	F-Score	IOU
No augmentations	**85.77**	74.93	78.38	66.67
No surface normals	74.34	62.00	64.07	49.62
Reduced no. of 3D layers	84.91	74.75	77.92	66.01
Tree height of three	84.92	73.30	77.00	64.91
Tree height of five	84.20	73.70	76.83	64.59
Default SVR-GC	84.65	**77.00**	**79.21**	**67.54**
No loss shaping $\mathcal{W}_{\text{scene}}^{\text{GC}}$	86.65	78.12	80.66	69.67

an IOU 67.54% and a recall of 77.00% while only having a chamfer distance to the ground truth of 13.15 centimeters on average. Removing the color augmentations reduces the IOU while more drastically affecting the recall performance, which is crucial for an application in mobile robotics. We want to be sure which areas of a scene are really occupied, measured by the recall, as it can lead to collision damages when occlusions are missed. Interestingly, while the IOU is reduced, the chamfer distance to the ground truth CD_{GT} is smaller at 12.85 centimeters. The removal of the surface normals as an input drastically decreases the performance on the volume-based and surface-based metrics, as the guidance for flat surfaces is missing and cannot be learned indirectly through the data. If we reduce the number of 3D layers in the tree network from 13 to seven, we only see a minor drop in performance, indicating that speeding up the prediction is possible without hurting the prediction. We also looked into adapting the height of our tree network by either removing one tree layer or adding the second layer again. However, this change does not improve the accuracy, indicating that four is the best tree height for this use case, except for the chamfer distance CD_{GT}, which decreases to 12.70 centimeters. Lastly, we remove the loss shaping $\mathcal{W}_{\text{scene}}^{\text{GC}}$ from the loss calculation as done in eq. (5.1). This removal slightly increases

Table 7.6: The surface-based metrics are evaluated here on the SUNCG dataset. The meshes are reconstructed from the 512^3 grid via the marching cubes algorithm and are then compared to the ground truth.

Method	CD_{GT}	CD_{pred}	CD
No augmentation	0.1285	0.1482	0.1383
No surface normals	0.2406	0.3127	0.2766
Reduced no. of 3D layers	0.1417	**0.1528**	0.1473
Tree height of three	0.1513	0.1909	0.1711
Tree height of five	**0.1270**	0.2017	0.1643
Default SVR-GC	0.1315	0.1626	**0.1470**
No loss shaping $\mathcal{W}^{GC}_{scene}$	0.2644	0.0893	0.1768

the IOU and even the recall but drastically hurts the chamfer distance CD_{GT} by doubling it.

In fig. 7.4, we perform a visual ablation study, where we change certain aspects of our method and evaluate the effect on the performance. The top row of the figure contains the color input image and the ground truth scene. In the row below that, our default SVR-GC result is presented; to the right, we removed the loss shaping techniques. This removal smooths the output as each compressed block gets treated the same, even if they contain valuable information. In this instance, the gap between the kitchen island and the stove is filled. If we remove the color augmentations, we see no significant difference in prediction strength. However, if we reduce the number of 3D layers, the finer details are lost, like the indentation for the sink. Removing the normals as an additional input ruins the scene reconstruction. The main reason for this is likely the small feature extractor at the beginning of the network. Changing the height of our tree architecture from four to three or five does not affect the reconstruction of this scene much.

In fig. 7.5, four scenes from the SUNCG dataset are qualitatively tested with our SVR-GC model. We selected scenes from four different room types to show the wide variety of scenes in the SUNCG dataset. These results show

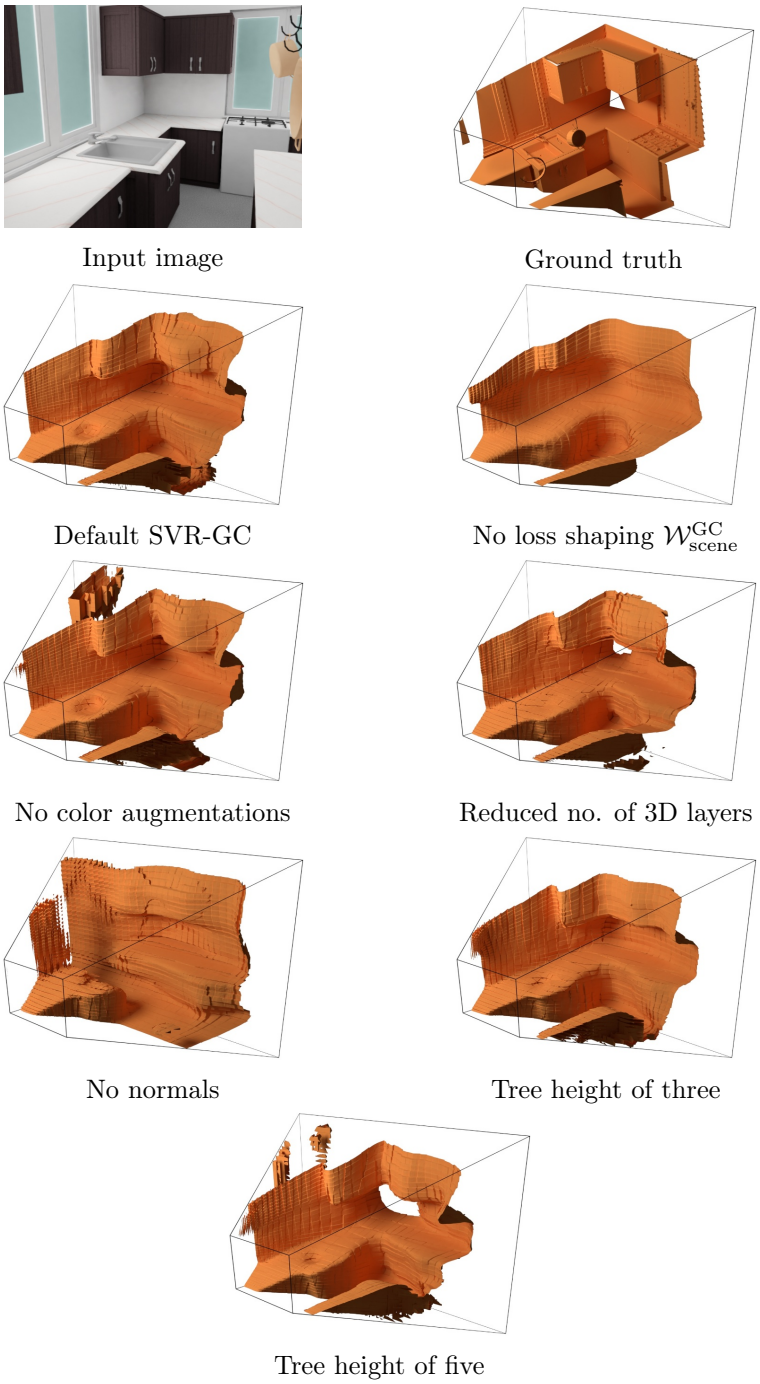


Figure 7.4: The scene reconstruction with the volume-based compression SVR-GC tested on one scene from the 3D-FRONT dataset. The top row contains the input image and the corresponding ground truth. Below the input image is the prediction of our default SVR-GC approach. The other six reconstructions are ablation results for the same scene.

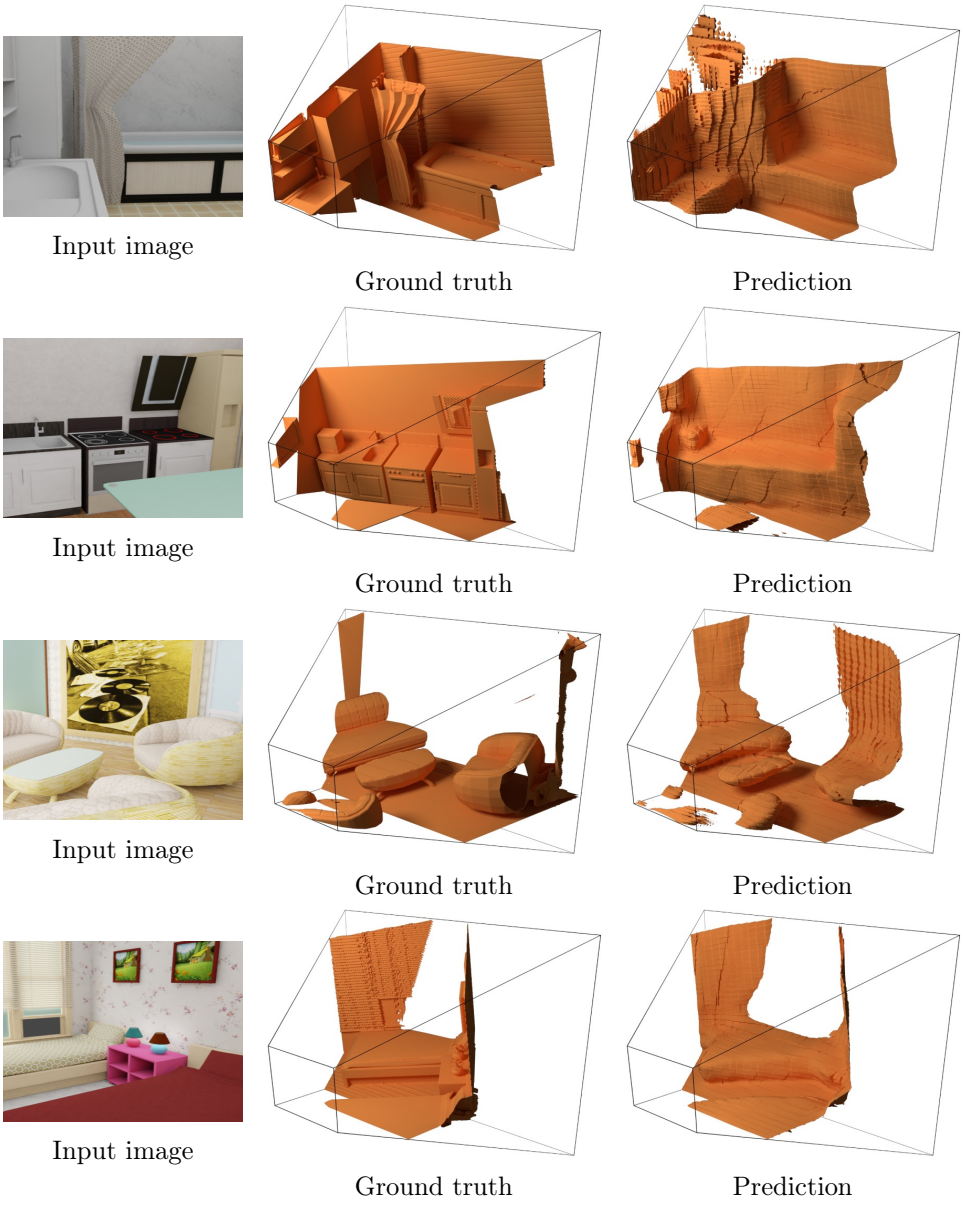


Figure 7.5: Four example scenes from the SUNCG dataset reconstructed with the SVR-GC method. These show four different types of rooms and how the model performs well in each. The filled bathtub in the first image is reconstructed well, even though it struggled with the finer details of the curtain. SVR-GC managed to reconstruct the frosted table inside the kitchen, even though it is paper-thick. The reconstruction of the living room shows how it can work with unusual table shapes, even though it omits the armrests for the chair in the back. The beds in the last row are well-reconstructed.

Table 7.7: In this table, the volume-based metrics on the Replica dataset are evaluated. Our default SVR-GC method performs best in IOU and recall on this challenging dataset. Only the method without the loss shaping performs better than the default method. However, as the results without loss shaping contain no fine details, they cannot be compared to the others.

Method	Precision	Recall	F-Score	IOU
No augmentation	87.08	64.33	72.37	58.04
No surface normals	73.57	60.06	63.34	47.38
Reduced no. of 3D layers	**87.53**	65.54	73.36	59.29
Tree height of three	86.62	65.03	72.66	58.28
Tree height of five	86.19	65.95	73.17	58.89
Default SVR-GC	86.79	**67.30**	**74.10**	**60.01**
No loss shaping $\mathcal{W}_{\text{scene}}^{\text{GC}}$	89.58	68.44	75.31	61.71

how our model can reconstruct the overall shape of the objects well, even though it sometimes struggles with thin or dynamic objects such as curtains or table legs. However, in the second row, a kitchen scene is shown, in which the reconstruction contains the thin frosted table in the front.

After evaluating the test images from the SUNCG dataset, we now evaluate SVR-GC on the real world Replica dataset. A quantitative evalatuion on the 360 images is done in table 7.7 and table 7.8. Our SVR-GC method achieves an IOU of 60.01% with a recall value of 67.30% with a chamfer distance of 51.68 centimeters. The augmentations improve the results, as, without them, the IOU is at 58.04% and the recall at 64.33%. Similarly to the SUNCG dataset, removing the surface normals as an input reduces the IOU drastically. Reducing the number of 3D layers slightly increases the precision while worsening everything else. Changing the tree height has similar effects as for the SUNCG dataset. Such that a network with a tree height of five decreases the IOU while also slightly improving the chamfer distance. In fig. 7.6, we show some quantitative results on the Replica dataset. The image in the first row contains a couch, which is correctly reconstructed, even though the coffee table before is neglected. Below is another couch

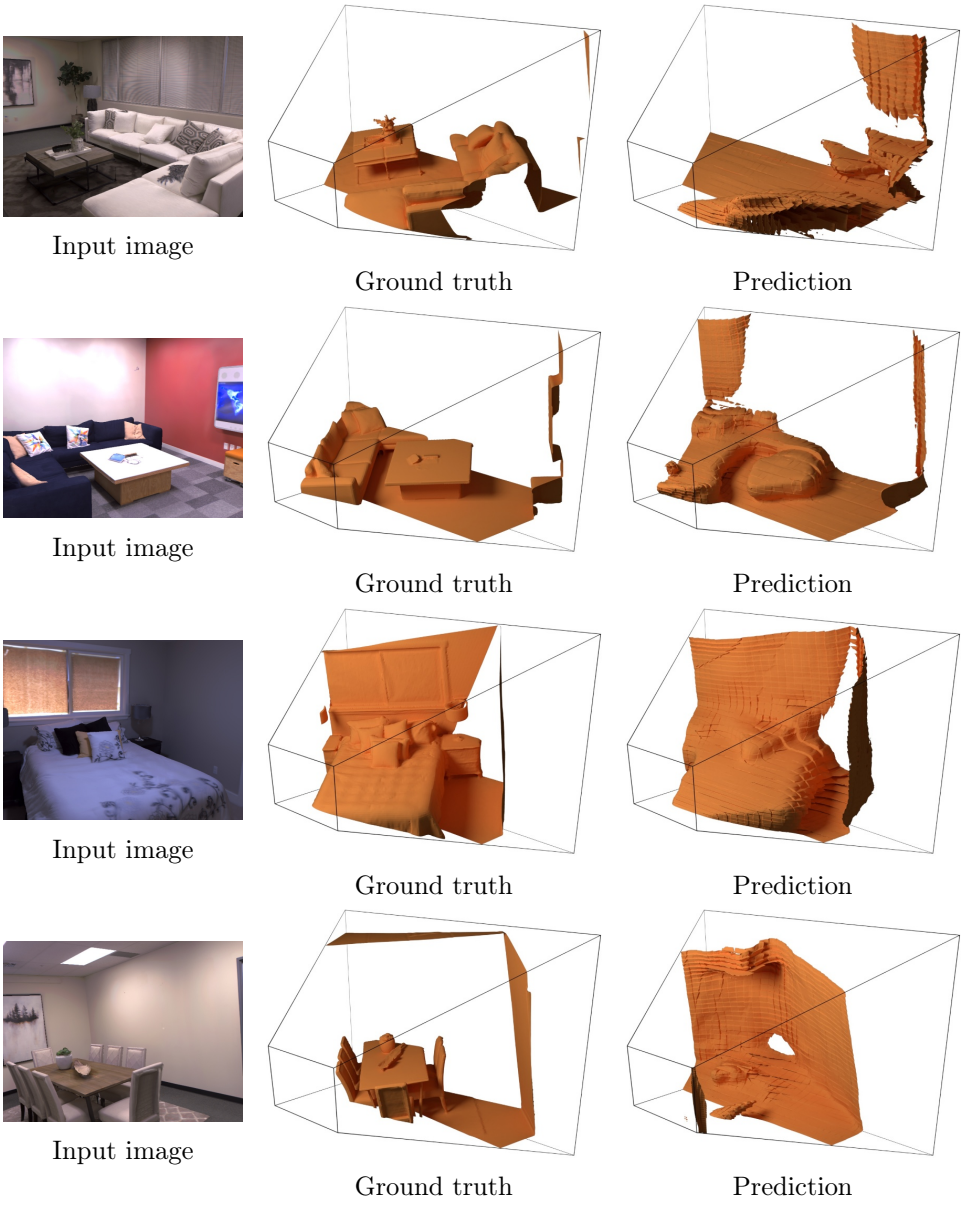


Figure 7.6: Some quantitative results on the Replica dataset are depicted here. The reconstruction performances on the two couches in the first two rows are good. However, the couch table is only reconstructed in the lower one and forgotten in the upper one. In the third row, a scene with a bed is reconstructed, impressively it assumes the space next to the bed to be free, even though this space is not visible. A failure case is presented in the last row, where the table and chairs are not correctly predicted, most likely because they are too thin.

Table 7.8: The surface-based reconstruction is evaluated on the Replica dataset in this table. Our SVR-GC method only performs 0.85 centimeters worse on the chamfer distance $\mathrm{CD}_{\mathrm{GT}}$ than the best-performing ablation method. Not using our novel loss shaping increases the error by more than 46% on the chamfer distance to the ground truth.

Method	$\mathrm{CD}_{\mathrm{GT}}$	$\mathrm{CD}_{\mathrm{pred}}$	CD
No augmentation	0.5499	**0.3665**	0.4582
No surface normals	0.4806	0.4337	0.4571
Reduced no. 3D layers	0.5189	0.3749	0.4449
Tree height of three	0.5373	0.4373	0.4873
Tree height of five	**0.4639**	0.4076	0.4358
Default SVR-GC	0.4654	0.3962	**0.4308**
No loss shaping $\mathcal{W}^{\mathrm{GC}}_{\mathrm{scene}}$	0.6803	0.3443	0.5123

from an office space, where the more sturdy coffee table and a blue couch are well-reconstructed. Even though the reconstructed gap between the couch table and the couch is tiny. The reconstruction of the bed in the third row correctly detects the free space behind the bed and reconstructs it well, even though the floor is not visible in this image. A failure case is shown in the last row, where a table with some chairs misses entirely in the reconstruction. As SVR-GC sometimes struggles with thin objects, especially if they are too dissimilar to objects in the simulated SUNCG dataset. Nonetheless, these results show how well a reconstruction of real-world scenes is possible when training only on synthetic data.

7.2.2 Implicit Representation (SVR-IC)

Setup

The tree network (SVR-IC) described in section 5.3 is trained on the $89,029$ encoded implicit TSDF volumes $\mathbb{V}^{\mathbf{F}}_{\text{3D-FRONT}}$ with the corresponding color

$\mathbb{I}_{\text{3D-FRONT}}$ and surface normal $\mathbb{I}^{\perp}_{\text{3D-FRONT}}$ images. We do not use the generated surface normal images for training only during testing. The batch size is set to two, while the learning rate of the used Adam optimizer is 1e−4. We train the network for $600,000$ steps and evaluate the resulting approach on the 3D-FRONT dataset $\mathbb{T}_{\text{3D-FRONT}}$ and on the Replica dataset $\mathbb{T}_{\text{Replica}}$.

Results

Our final tree net architecture (SVR-IC) described in section 5.3 is evaluated on $1,000$ images $\mathbb{I}_{\text{3D-FRONT}}$ from the 3D-FRONT dataset and on the 360 images from the Replica dataset.

In fig. 7.7, we show four different scenes from the 3D-FRONT dataset. The tree network only gets the color image and a predicted surface normal image from the U-Net. The final prediction is presented in the right column, while the ground truth is depicted in the center. This ground truth is the decompressed scene created with the implicit TSDF network evaluated in section 7.1.3. The first row in this figure contains the scene presented in the main overview in fig. 1.2. In its prediction, the room is well reconstructed, even though the armchair is incorrectly labeled as a couch, while the ground truth is labeled as a chair. It appears the closest matching object to this armchair the network has seen before was labeled as a couch. One noteworthy thing is the struggle with thin objects, most notably in the second row, where the stand of the table behind the green chair is not reconstructed. The issue is that the compression algorithm already struggles with thin objects as only the space inside an object gets negative TSDF values, which is quite limited for delicate entities. This struggle can be seen in the first row, where the white couch table at the wall is not well depicted in the ground truth.

The last two rows of fig. 7.7 show the successful reconstruction of a scene with a bed and one with an L-shaped couch. These four scenes already highlight that most objects in the 3D-FRONT dataset are more commonly found in living and bedrooms, as it provides a wide range of couches and beds but lacks a good collection of kitchens or bathrooms.

Evaluating our proposed tree network with the implicit encoding quantitatively on the $1,000$ images from the 3D-FRONT dataset leads to the results presented in table 7.9 and table 7.10. We show the voxel-based evaluations for our default method in the second-to-last row of table 7.9 and the evaluation

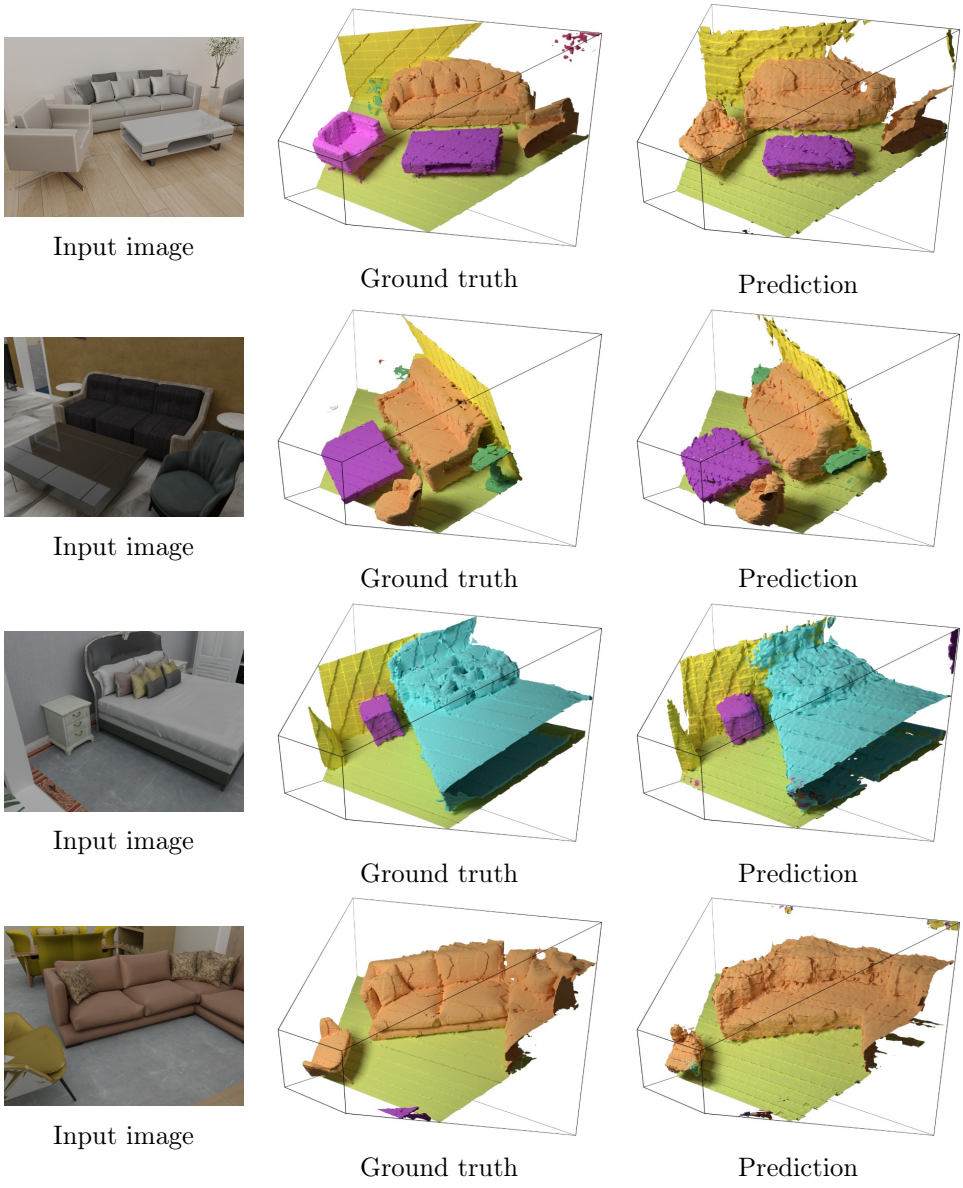


Figure 7.7: This figure shows the reconstruction performance of the tree architecture combined with the implicit TSDF encoding. These four examples are taken from the 3D-FRONT dataset. On the left, the input to the reconstruction network is shown, while in the center, the ground truth is depicted. The reconstructed mesh of the predicted TSDF latent space is shown in the right column. We used the default method with the predicted surface normal image. The surface colors represent the categories.

on the reconstructed mesh in the same row in table 7.10. For the voxel-based results, we sample the implicit space with a resolution of 512 along each axis and calculate the precision, recall, F-Score, IOU, and class accuracy as defined in section 7.1.1. The reconstructed mesh is the back-projected marching cubes result of this TSDF voxel grid. We report the values for the ground truth surface normals and surface normals predicted with the U-Net. We can see that using the predicted normals decreases the performance, but only in a small range, while still allowing an application in robotics. Our default approach gets a recall value of 86.3%, while the IOU is at 77.8% and the class accuracy at 81.8%. However, the consistency of the class predictions can be seen in fig. 7.7, where instances are consistently labeled as the same category even if the predicted category is wrong, not showing much random label noise. On the other hand, the reconstructed mesh closely matches the ground truth and has an average distance error of 7.2 centimeters for the CD_{GT} with the predicted surface normals. The combined chamfer distance is slightly worse

Table 7.9: The voxel-based evaluation on the 1,000 images from the 3D-FRONT dataset are presented in this table. In the second to last row, we show our default configuration described in section 5.3. The evaluated metrics used here are defined in section 7.1.1. We particularly focus on the Recall and the IOU, as described above. Further, each column contains the values for using the ground truth surface normals and the predicted surface normals. We also provide an ablation study for some selected hyperparameters.

Method	Precision		Recall		F-Score		IOU		Class Acc.	
Surface Normal	GT	pred	GT	pred	GT	pred	GT	pred	GT	pred
reduced 3D filters	89.2	88.1	90.7	**86.5**	89.6	86.4	82.2	77.6	86.2	82.7
no inception	88.6	87.0	89.6	84.3	88.7	84.9	80.9	75.2	84.0	79.3
no tree loss	89.5	88.2	88.7	85.8	88.7	86.4	80.8	77.4	84.4	81.3
no wall weight	90.8	88.3	90.9	86.0	**90.5**	86.5	83.4	77.6	**88.1**	**83.0**
tree height 2	90.9	**89.4**	89.8	84.7	90.0	86.4	82.9	77.4	86.8	82.4
tree height 4	**91.0**	88.6	88.9	83.5	89.6	85.3	82.3	75.9	86.4	80.2
default SVR-IC	90.1	87.8	**91.3**	86.3	90.4	**86.5**	**83.5**	**77.8**	86.7	81.8
no loss shaping	92.8	90.5	91.5	85.8	91.9	87.4	85.8	79.0	89.7	84.7

at 11.6 centimeters. This is because surfaces behind the walls and objects are sometimes mispredicted, increasing the CD_{pred}. A small example of this incorrect prediction can be seen in the last row of fig. 7.7, where the floor is partially reconstructed below the couch. This reconstruction of the lower side is challenging as some objects have gaps below them, like the bed in the row above this couch, forcing the network to learn if such a gap exists and then propagate it properly.

These tables also contain some ablation results, where we changed the architecture slightly to show the strength of our design choices. In the first row, we reduce the number of filters per layer in the 3D convolutions from 512 to 256. Therefore, reducing the necessary amount of multiplications. Comparing the results for the voxel-based evaluation to the default configuration, we see no major differences. The same is true for the results for the chamfer distance calculations in table 7.10, where we see the CD_{GT} reduces by a few millimeters. We further evaluate the effect of our inception layers increasing

Table 7.10: The results on the reconstructed surface of the implicit TSDF scene reconstruction method on the 3D-FRONT dataset are shown here. Each predicted mesh is compared to the scene decoded with the implicit compression network. Our focus is more on the CD_{GT} than on the entire chamfer distance, as we are more interested in how well the ground truth is predicted and less in how well the prediction matches the ground truth.

Method	CD_{GT}		CD_{pred}		CD	
Surface Normal	GT	pred	GT	pred	GT	pred
reduced 3D filters	0.050	0.074	0.120	**0.152**	0.085	**0.113**
no inception	0.050	0.080	0.137	0.180	0.093	0.130
no tree loss	0.052	0.073	0.144	0.164	0.098	0.118
no wall weight	0.047	0.074	0.115	0.163	0.081	0.118
tree height 2	0.049	0.077	0.121	0.155	0.085	0.116
tree height 4	0.047	0.076	0.129	0.175	0.088	0.125
default SVR-IC	**0.045**	**0.072**	**0.114**	0.160	**0.080**	0.116
no loss shaping	0.077	0.111	0.050	0.094	0.063	0.103

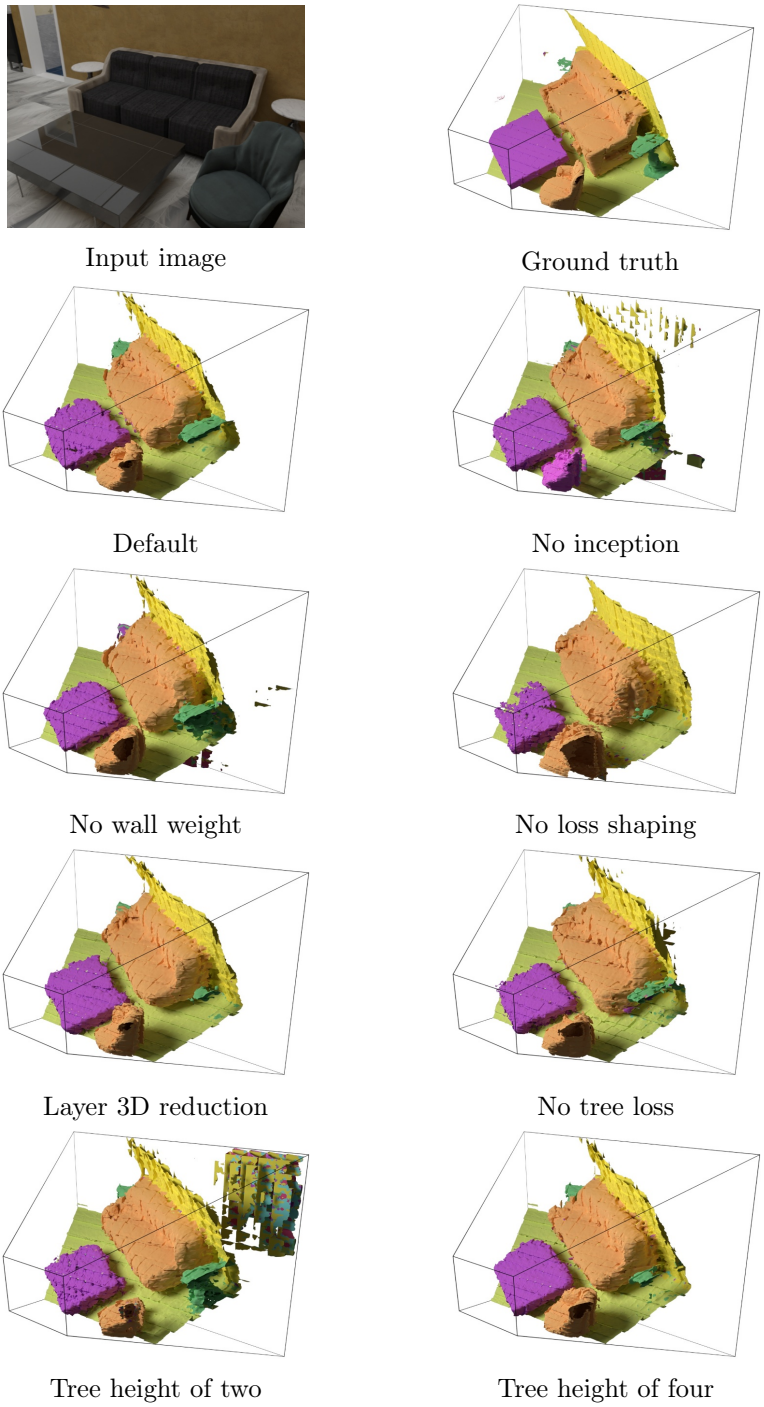


Figure 7.8: The different ablation study results are shown here. The first row contains the input image to the network and the ground truth. Each row below shows the ablation and the default method results. Our default method reconstructs the scene best while maintaining minimal surface noise and ensuring that the green tables are reconstructed well.

the receptive field using dilated convolutions. Here, we see a more considerable drop in performance as the IOU drops from 77.8% on the predicted surface normals down to 75.2%, indicating how vital our inception layers are. On the other hand, removing the inner tree loss defined in section 5.4.3 reduces the IOU slightly while increasing the error on the chamfer distance by more than a centimeter. If we instead remove the additional loss on non-wall and non-floor objects, we only see a slight drop in performance. We further evaluated the effects of the tree's height on the performance and can show that a tree height of three the IOU and CD_{GT} is better than for trees with a height of two or four.

At last, we evaluate the effect of our loss shaping on the performance. While it increases the IOU and the precision by a considerable margin, it decreases the recall and deteriorates the CD_{GT}, which is more important in this work as we want to reconstruct the ground truth mesh correctly. We are less concerned about predicted objects or surfaces not present in the ground truth. We also depict the effects of our ablation study on one example of the 3D-FRONT dataset in fig. 7.8. The default settings show superior performance, particularly when evaluated on the green tables to either side of the orange couch. The top surface of the table is only well reconstructed in the default settings case. While for the no inception test, the no wall weight, and the no tree loss test, the main criticism is increased surface noise. The importance of the loss shaping can be seen here too. Without it, all sharper boundaries of the objects are smoothed over, and finer details are lost entirely, such as the table or the armchair's shape. Changing the network architecture only has little effect on the total performance, even though for the tree with layer height two, the network predicts random surface noise behind the wall in the non-reachable space.

After the evaluation, on the 3D-FRONT dataset, we assess the performance on the real-life Replica dataset. This is done, to ensure that the simulated data we used for the training transforms well into the real world. Again, we evaluate the voxel-based metrics in table 7.11 and the surface-based metrics in table 7.12. Our default method has the best performing IOU and recall value, which are only slightly worse than for the 3D-FRONT dataset. The class accuracy is worse than before, based on the qualitative results in fig. 7.9. The main reason is that many objects are not correctly classified in the ground truth. The reason for this is the mapping between the Replica classes

Table 7.11: The volume-based metrics are evaluated in this table on the 360 images from the Replica dataset. This table contains the results for the ground truth and predicted surface normals while comparing the results of our default method in the second-to-last row against some ablation results. The best IOU and best recall value are achieved for our default method.

Method	Precision		Recall		F-Score		IOU		Class Acc.	
Surface Normal	GT	pred	GT	pred	GT	pred	GT	pred	GT	pred
reduced 3D filters	83.0	**85.9**	**86.8**	78.3	**83.7**	80.6	73.3	69.0	64.9	67.1
no inception	84.2	85.1	84.4	77.0	83.3	79.3	72.6	67.2	62.9	63.6
no tree loss	83.1	83.9	83.6	79.2	82.2	80.1	71.1	68.3	64.9	64.9
no wall weight	83.1	84.1	83.9	76.7	82.3	78.6	71.3	66.4	**70.8**	**68.9**
tree height 2	**84.5**	85.5	83.5	76.7	82.8	79.4	72.1	67.5	67.5	67.0
tree height 4	84.2	85.0	82.8	75.3	82.2	78.2	71.2	65.9	67.0	65.8
default SVR-IC	84.2	85.3	85.5	**79.4**	83.6	**80.9**	**73.3**	**69.5**	68.0	66.8
no loss shaping	87.1	87.2	85.7	78.0	85.3	80.8	75.5	69.4	77.5	75.5

and the reduced class set we use. All surfaces in red could not be mapped to one of our ten categories. It is also apparent that the coffee tables in the 3D-FRONT dataset are usually classified as cabinets in purple, while in the Replica dataset they are green tables.

Nonetheless, the reconstruction quality is exceptionally high, even though the network has never seen any of these instances during training. So, it learned a fundamental understanding of 3D space and interior objects and was able to reconstruct the main furniture pieces in the scene. This is especially interesting for the tables and chairs in the last row, where it was even able to reconstruct the sitting area of the chair, where only the backrest is visible. However, it estimated the table length wrong and tried to fit the chair at the end of the table into the reconstruction.

Some of the hyperparameters evaluated on the ablation study are more important on this dataset than on the training 3D-FRONT dataset. In particular, removing the wall weight reduces the IOU performances from 69.5% to 66.4%. Interestingly, the difference is less prominent if the ground

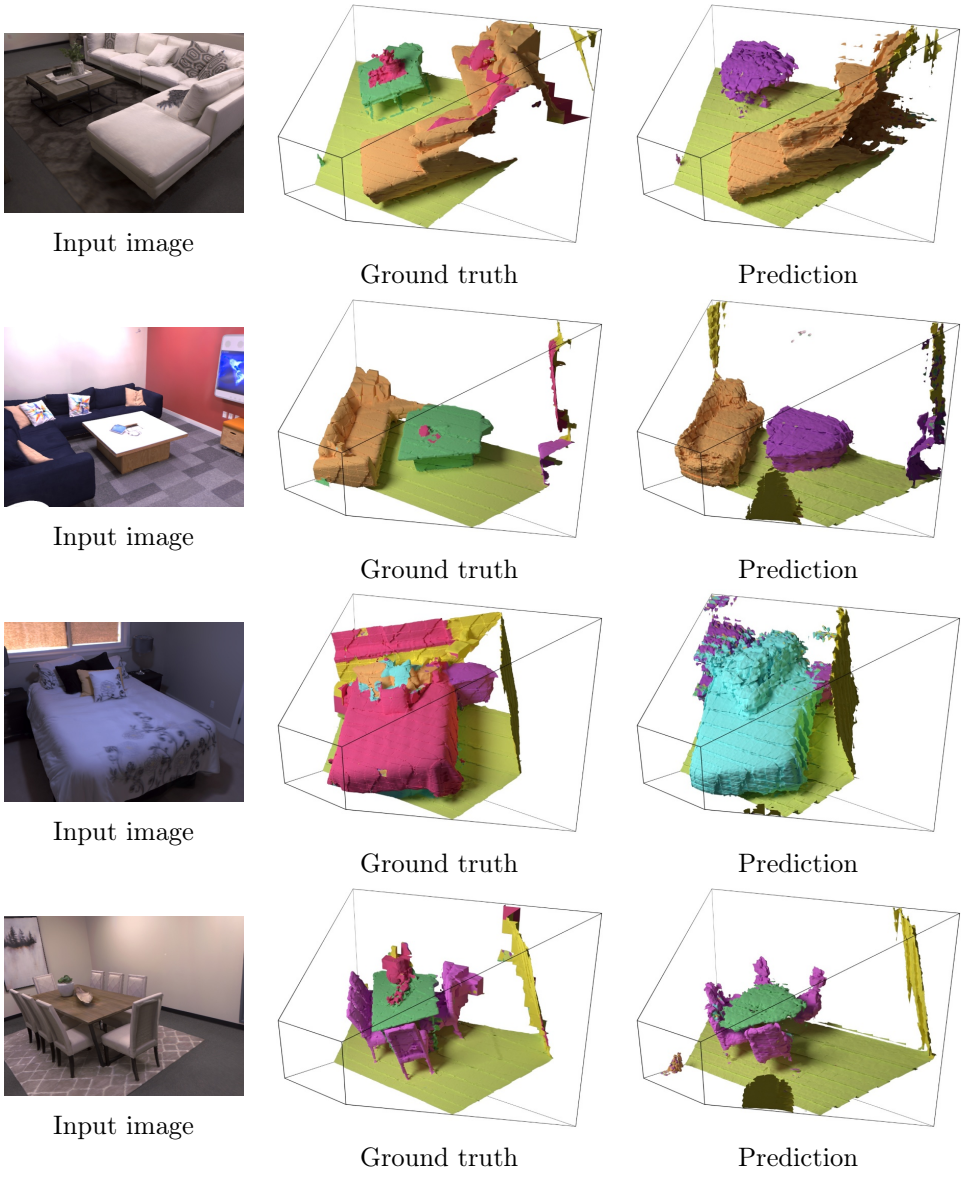


Figure 7.9: Four images from the Replica dataset, show the performance of our approach on real-life data. The reconstruction performance is only slightly worse than on the 3D-FRONT dataset, indicating that our method can bridge the gap between the simulation and the real world. The ground truth surfaces in red do not have a matching category in our reduced class set. Especially the results on the chairs and table in the last row are impressive, showing how the network learned to predict the shape of an object in 3D based only on the color information and the predicted surface normals.

Table 7.12: The surface-based reconstruction metrics in meters are evaluated on the 360 images from the Replica dataset in this table. We provide the results of the default approach and some ablation methods. When using the ground truth surface normals, the chamfer distance on the ground is only 8.4 centimeters, while it only decreases to 10.5 centimeters when relying on the predicted surface normals. The higher error for the chamfer distance on the prediction can mostly be attributed to flying surface noise behind the objects or inside the objects.

Method	CD_{GT}		CD_{pred}		CD	
Surface Normal	GT	pred	GT	pred	GT	pred
reduced 3D filters	0.088	0.107	0.211	**0.229**	0.149	**0.168**
no inception	0.085	0.110	0.214	0.253	0.149	0.182
no tree loss	0.085	**0.096**	0.244	0.254	0.164	0.175
no wall weight	0.092	0.111	0.226	0.262	0.159	0.187
tree height 2	0.086	0.112	0.234	0.248	0.160	0.180
tree height 4	0.088	0.111	0.235	0.261	0.162	0.186
default SVR-IC	**0.084**	0.105	**0.206**	0.233	**0.146**	0.169
no loss shaping	0.148	0.168	0.125	0.161	0.136	0.165

truth normals are used, indicating that the wall weight is crucial when the normals are not perfectly accurate. Furthermore, the reduction of 3D filters reduces the recall on the predicted surface normals from 79.4% to 78.3%. As this is not the case for the ground truth surface normals, we can assume that more 3D filters help correct errors made in the network's prior layers. The other hyperparameters behave similarly as on the 3D-FRONT dataset. When evaluating the surface reconstruction performance, we can see that the chamfer distance on the ground truth CD_{GT} is worse than on the 3D-FRONT dataset. The main reason is that the object instances are all unknown, while in the simulated case, the instance can be learned by heart. We still achieve a chamfer distance of 8.4 cm for the ground truth surface normals and 10.5 cm for the predicted surface normals. These distances are small enough for any navigation and planning application on a real robot, as Rollin' Justin would usually not navigate through open space with only 10 cm of free space

between itself and surrounding objects.

7.3 Our Approach compared to Related Work

In order to evaluate the performance of our proposed methods, we compare the tree net architecture relying on the TSDF volume grid compression, and the implicit position representation to Total3DUnderstanding [114] and P3DSR [26]. For this evaluation, we use the Replica dataset as it contains real-world scenes while providing a dense and hole-free mesh. As the TSDF volume grid scene reconstruction network and the P3DSR approach were only trained on scenes with a fixed height and tilt, we repositioned the camera by adapting the height and tilt of the existing camera poses. We then rendered new camera images and created the appropriate TSDF volumes. As we need a category label, we rely on the decompressed scene from the implicit method as ground truth data.

The results for this are collected in table 7.13. We first evaluate the four methods on the images from the Replica dataset with different camera heights and tilts ranging from 45° to 80°. We can only present the volume-based metrics for our methods, as Total3D and P3DSR do not create a 3D TSDF volume grid at the end, only a reconstructed mesh, on which we use the chamfer distance to evaluate the reconstruction accuracy. In this scenario, our method with the implicit encoding SVR-IC performs best with an IOU of 69.45% and a chamfer distance of the ground truth mesh with 10.53 centimeters. The second best method in this scenario is P3DSR, which was not trained for this range, but still performs well. Total3D does not perform great, with an average error of 30.57 centimeters, even though we provided the ground truth 2D annotations to reduce the error from the used 2D detector. Even though we provide the perfect categories for the pixel positions, they are not transformed well into 3D, leading to only half the performance in class accuracy. Our method SVR-GC was only trained with a fixed tilt of 78.69° and a height of 1.55 meters. So, the performance drops to 41.89 centimeters, as the range is now between 1.45 and 1.85 meters and 45° and 80° degrees.

In fig. 7.10, our methods SVR-GC and SVR-IC are compared visually against

Total3D and P3DSR. Both our presented methods perform better on the Replica dataset, as either Total3D or P3DSR. In the office scene in the first row, with a couch and a sturdy coffee table, our methods can reconstruct the couch and coffee table. At the same time, Total3D got the information about the couch being presented via the ground truth bounding box but placed the couch incorrectly in the scene, as it collides with the table in front of it. Our method SVR-IC miscategorized the coffee table for a cabinet, which might be a close match. For the bed scene, Total3D placed the bed below the floor and added small structures next to it, which might be the cabinets. P3DSR reconstructed the rough shape of the bed, even though it labeled the bed as floor. SVR-GC reconstructed the bed but could not detect the free space behind it, most likely because it was not trained on this camera tilt. The performance for a fixed camera tilt is better (see fig. 7.6). With our method SVR-IC, we can reconstruct the bed and label it correctly. The ground truth has a blanket on top, labeled as the void class. For the challenging scene

Table 7.13: Our own two methods and two related works evaluated on the real-world Replica dataset. We compare the volumetric-based metric on our methods and the surface-based ones on all of them. In this comparison, it is clear that our method works best in its designed tilt range with a chamfer distance to the ground truth of only 10.53 centimeters on average. Methods that do not provide a class accuracy are denoted with an X.

Angle	Method	Precision	Recall	⌀IOU	CD_{GT}	CD_{pred}	Class Acc.
[45° - 80°]	Total3D	-	-	-	0.3057	0.3545	32.77
[45° - 80°]	P3DSR	-	-	-	0.1728	0.3144	17.49
[45° - 80°]	SVR-GC	75.62	60.85	48.28	0.4189	0.4319	X
[45° - 80°]	SVR-IC	**85.30**	**79.41**	**69.45**	**0.1053**	**0.2331**	**66.78**
78.69°	SVR-GC	**86.79**	67.30	60.01	0.5168	0.3926	X
78.69°	SVR-IC	82.84	**76.65**	**65.46**	**0.1081**	**0.2526**	64.86
90°	P3DSR	-	-	-	**0.1858**	**0.2692**	10.74
90°	SVR-IC	79.97	48.41	41.67	0.3051	0.3751	**48.19**

with the multiple chairs around the dining table, all methods except SVR-IC perform poorly. Even for a fixed camera tilt, SVR-GC does not perform much better. SVR-IC, on the other hand, can detect the chairs, even though they are only a few centimeters thick. The only error in that prediction is that the chair at the back of the table should not be in the prediction as it is further away than our four-meter boundary. Lastly, we have an image of a white couch; again, only SVR-IC delivers satisfactory results.

In order to evaluate P3DSR correctly, we need to adapt the evaluation data, as its training data was created at a camera height of 90 centimeters and a fixed tilt of 90° degrees. On a side note, the authors of P3DSR also relied on BlenderProc and the 3D-FRONT dataset to generate their training data. On these adapted views, the performance for our method SVR-IC drops, as 90° is beyond the range of the training data. The main reason is that reconstructing a scene at a 90° angle is more complex than at a tilt. As the surfaces in the scene are now planar to the camera, making it more difficult

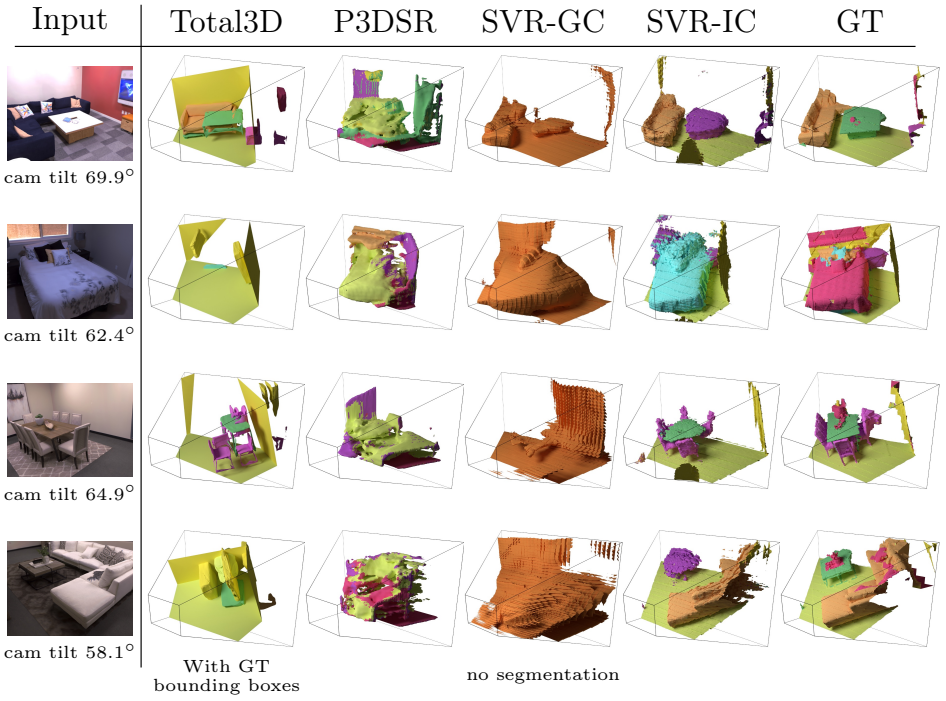


Figure 7.10: Results on the Replica dataset for four different color images. All methods only use a color image as an input. Our method SVR-IC shows the best reconstruction, followed by our other presented method SVR-GC.

to detect them correctly. This can be seen in table 7.13, as the couch in the top row of fig. 7.11 was correctly predicted. However, its depth could not be adequately estimated based on this frontal view; the table and chair are entirely neglected. In contrast, P3DSR struggles with correctly classifying the objects even though both methods were trained on the 3D-FRONT dataset. Nonetheless, is it able to predict the shape of the couch and chair better than with SVR-IC. We did not create another dataset to evaluate if retraining with another angle would help, as the creation of the dataset for the training of SVR-IC takes around 1.3 years of GPU hours.

The same data adaption has to be done for SVR-GC. So, we repositioned the camera to a height of 1.55 meters and a fixed tilt of 78.69° and recreated the 360 images. This adapted data only slightly changes the results for SVR-IC, as this camera tilt and height are covered in the original training data. On the other hand, it improves the volume-based result for SVR-GC, while decreasing the performance for the chamfer distance. The main reason might be that the output produces so much surface noise when the camera tilt is too different. So, the closest matching point is closer than with a proper surface reconstruction. An example of this can be seen in the last row of fig. 7.10, where the output surface is extremely noisy. Some qualitative results can

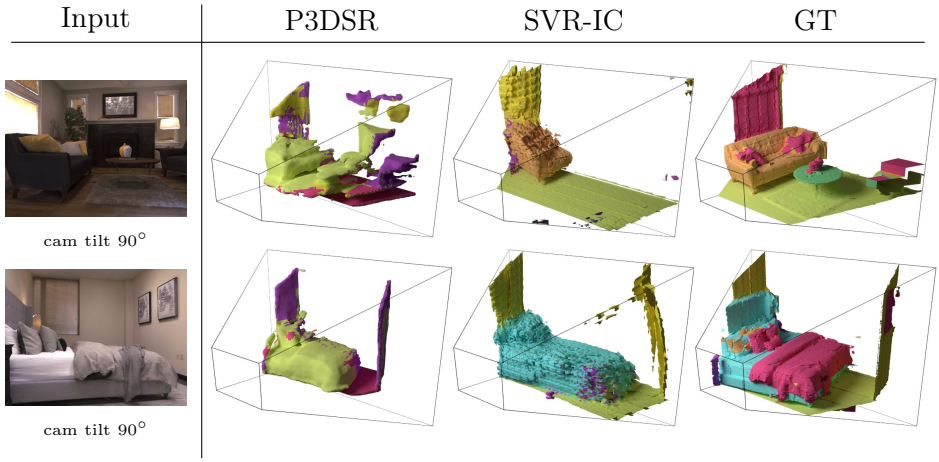


Figure 7.11: Four reconstructed scenes for images taken at a tilt of 90°, similar to the training data from P3DSR. Both methods only use the color image shown in the left column. Even though the reconstruction of the objects in P3DSR is acceptable, the classification of the objects failed. Both the bed and the couch were classified as floor.

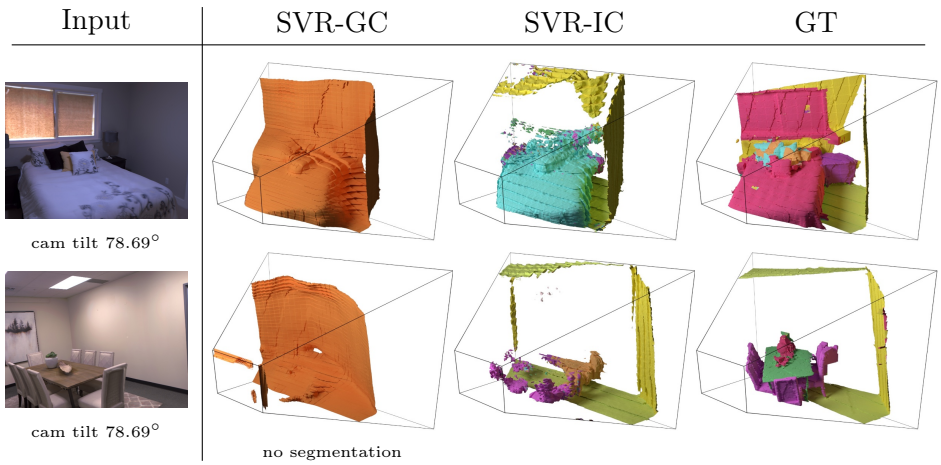


Figure 7.12: Results on the Replica dataset for four different color images. All methods only use a color image as an input. Our method SVR-IC shows the best reconstruction while providing a semantic segmentation. Only the detection of the backside wall in the upper row works better with SVR-GC.

be seen in fig. 7.10 and fig. 7.12. The input images, which have a camera tilt close to 78.69° are well reconstructed, while the image in the last row of fig. 7.10 has a big difference in tilt, breaking the prediction. However, if the camera tilt is at 78.69° degrees, the segmentation of large objects works well, like the bed in the first row of fig. 7.12. The SVR-GC method still struggles with finer objects, like the chair and table in the same figure.

These results show that our methods SVR-GC and SVR-IC can produce 3D scene reconstructions of different scenes in the Replica dataset.

7.4 Scene Reconstruction in the Wild

After evaluating our approach on the Replica dataset, where we can easily create a ground truth scene to compare, we want to see now how well SVR-IC works in the wild. For this, we recorded images in homes all around Bavaria. This means that after evaluating them on scenes from the Replica dataset, which has been recorded in the USA, and the 3D-FRONT dataset, which Alibaba, a Chinese company, designs. We now see how well SVR-IC works on German interiors. This work will not highlight the architectural difference

Figure 7.13: These are ten images recorded in five different homes and the SMiLE lab [160, 60] at the German Aerospace Center (DLR). We have a collection of different couches that have been reconstructed successfully. The image of the table and chair on the left in the second row is particularly difficult, as all objects are made only of thin pieces of wood. Nonetheless, they appear in the final reconstruction. The image below shows a glass coffee table, which was not present in the training data and is therefore not reconstructed well, even though it detected the small stand holding the table up.

and interior styles between different cultural zones; we refer to [61, 178, 157] for this. In fig. 7.13, we show the results on different images recorded in five homes and the SMiLE lab [160, 60] in the German Aerospace Center (DLR). These reconstruction results show that with our method SVR-IC it is possible to reconstruct 3D scenes. In particular, they show that our method can understand the nature of objects and how they occupy the scene. This understanding can be seen in the left image in the second row, where a dining set is reconstructed. Even though SVR-IC failed to reconstruct the back of the chair right behind the table, it still managed to detect the chair and place a sitting surface in mid-air. This reconstruction is remarkable, as the sitting surface is not visible in the single input image. In the last row on the right, we show an image of the SMiLE lab, where we reconstruct the couch and parts of the cabinet on the left. Our method slightly struggles with the curtains in the background as the reconstruction of its surface is particularly challenging. Overall, these results show that it is possible to reconstruct a 3D scene and segment it simultaneously using only one color image.

Chapter eight

Conclusion

In pursuit of economic well-being, the number of tasks solved by robots is growing without ceasing. These robots need to understand and reconstruct their environment to perform any kind of human task. For the most demanding tasks concerning scene interaction or understanding, it is necessary to equip the robots with the capability of semantic 3D scene reconstruction. These 3D scene reconstructions must be quick and precise while using as little information as possible. To enable the use in everyday systems, it would be preferable to use only color images without relying on depth images obtained by advanced cameras. In this thesis, we successfully produce a system capable of reconstructing 3D scenes using single color images in indoor environments.

So, we started this work by tackling the problem of single-view scene reconstruction, going beyond just reconstructing single objects from images. By going at it holistically, we were able to combine the full information in an image to reconstruct every object and structure in the scene. For this, we selected TSDF grid-based volumes and encoding a surface implicitly as our scene representations and compressed them as a latent vector using a neural network. We then designed a novel tree-net architecture enabling us to learn the transfer of 2D features into 3D while reconstructing the encoded 3D scenes.

Concretely, we highlighted in chapter 3 the possible scene representations and

discussed the difficulty of aligning meshes and point clouds pixelwise with a 2D image. We then explained how this could be done for TSDF volume grids and an implicit representation, encoding a scene in the weights of a neural network. This alignment between the color image and the scene could be achieved by projecting the 3D scene into a cube.

In chapter 4, we demonstrated how 3D scene information could be encoded in a latent representation using a neural network. We showed how such an autoencoder could compress a TSDF volume grid by a factor of 512. Additionally, a neural network is explained that can implicitly represent a surface by allowing the querying for positions in space and returning a TSDF value and segmentation label for them. This compression enabled us to encode highly detailed models while reducing the data size of our 3D scenes to fit them in the memory of a GPU. Furthermore, we could enrich such encodings with semantic labels, allowing us to reconstruct a scene and segment it simultaneously.

Chapter 5 established our novel tree-net architecture, allowing the transformation of learned 2D features to 3D, enabling the reconstruction of an entire 3D scene based on a single color image. We further provided a novel loss-shaping technique that allows the optimizer to focus on particular 3D surfaces that are more relevant to the task at hand.

In chapter 6, we described how we created the synthetic data necessary to train our proposed methods. At first, we detailed how the open-source software BlenderProc [35, 37] was invented and explained the difference between the first and second releases. We then elaborate on how TSDF values can be generated for non-water-tight objects in 3D scenes.

Lastly, we evaluated in chapter 7 our two novel methods for scene compression and scene reconstruction on the SUNCG, 3D-FRONT, and the real world Replica dataset. Furthermore, we assessed the quality of our reconstruction on images taken in the wild without the constraints typically imposed in a lab environment.

The results of this thesis enable the 3D reconstruction of scenes based on single color images. Our presented methods allow future robots in new environments to obtain a metric 3D reconstruction of the scene, removing the need to record multiple images from different vantage points. We can now imagine a world in which robots enter homes for the first time and are

able to plan and navigate freely in them without needing a team of engineers who first create a detailed map of the environment.

In the presented work, we are mere forerunners in the vast area of research of single-view whole-scene reconstruction. Future works might focus on improving the reconstruction precision, especially on delicate objects, by improving the scene compression or they extend the number of known classes. Lastly, one could imagine including a color reconstruction for non-visible surfaces by relying on a NeRF for compressing a scene. One could also imagine replacing our scene representation with meshes, even though their alignment to the color image using a deep neural network poses an unsolved problem. The question of how to represent a 3D scene for deep learning is fundamental and remains open for future researchers to tackle. We believe this area has the highest chance of possible innovations to make further progress in the field of scene reconstruction.

Appendix A

Surface Normal Generation

In chapter 5, we use a U-Net architecture [130] to predict a surface normal image for each color image. We design two U-Net architectures, one for the SVR-GC approach and the other for the SVR-IC method.

Surface Normal Prediction for SVR-GC

For the training of SVR-GC, we rely on the SUNCG dataset, which we also used to train this U-Net. We start with our color image with a resolution of 512^2, on which a convolution is applied to raise its feature channels to eight. The output of each convolution before a pooling operation is later used to be merged with the up-part in the U-Net. We then add a pooling operation to reduce the spatial size to 256, followed by two ResNet blocks [70]. Each ResNet block contains two convolutions, where both can be skipped. The used convolutions rely on the same inception layer defined in section 5.2. Here we use the first half of the feature channels with a dilation rate of one. The second half is equally split with a dilation rate of two and three. This inception layer increases the receptive field, ensuring the entire image is used. We then perform another pooling followed by three ResNet blocks with 64 feature channels each. Followed by another pooling reducing the spatial

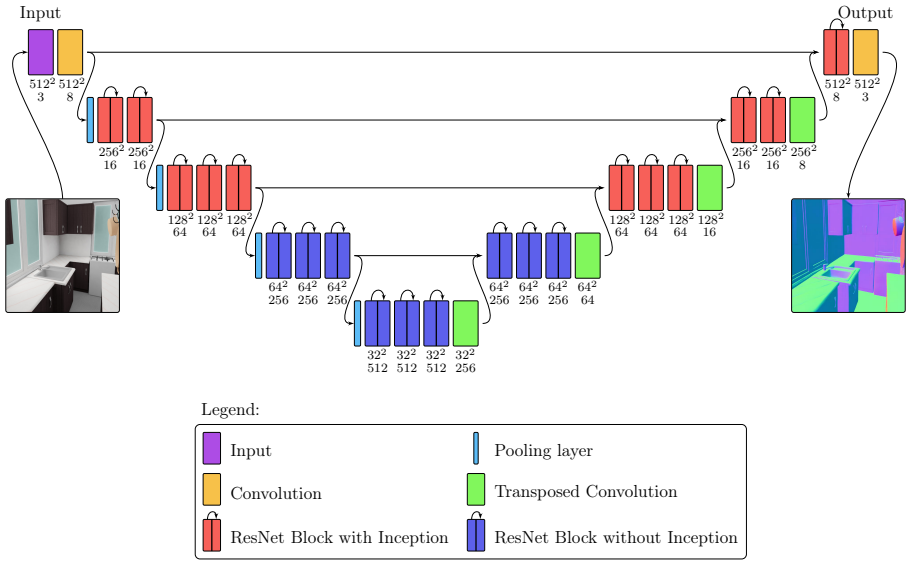


Figure A.1: Our U-Net architecture for SVR-GC. It transforms a color input image into its corresponding surface normal image, enabling a scene reconstruction approach that only relies on color images.

Table A.1: Surface normal results on the on SUNCG and the Replica dataset. In the top row, we show the results on the SUNCG dataset, where we achieve that 78.20% of the predicted pixels have an angle difference below 5°. However, this drops for the real-world Replica dataset to 39.85%.

Testing dataset	Mean Angle	Angel Difference				
		5°	11.5°	22.5°	30°	60°
SUNCG dataset	5.30°	78.20	85.40	90.98	93.28	98.98
Replica dataset	15.28°	39.85	62.65	78.40	84.07	94.90

dimension to 64, we use another three ResNet blocks without any Inception layers, as the spatial size is already relatively small. We use another last pooling operation for the lowest layer, followed by three ResNet blocks with a feature channel size of 512. From this lowest layer, we apply a transposed convolution with a stride of two to increase the spatial resolution back to 64 with 256 feature channels. The result of the transposed convolution is concatenated with the input to the lowest layer. We then apply three ResNet blocks followed by a transposed convolution with 256 feature channels. This

Figure A.2: Four examples of our U-Net on the Replica dataset. Even though we trained with the swapped texture dataset, the texture on the bed in the first row produces tiny details on the bedspread.

gets done once more with three ResNet blocks now using dilation again, as the spatial size is 128, followed by a transposed convolution up to a spatial size of 256. Our last layer contains two ResNet blocks followed by the last transposed convolution, which gets concatenated with the output of the first convolution. Its output gets passed through a ResNet block with eight feature channels, which are then finally mapped to three to produce our predicted surface normals. This entire architecture is shown in fig. A.1.

In table A.1, we compare the performance of our U-Net on the SUNCG training dataset, using the color $\mathbb{I}_{\text{SUNCG}}$ and surface normal $\mathbb{I}^{\measuredangle}_{\text{SUNCG}}$ images,

with the real-world Replica dataset. The images in both datasets are recorded with a fixed tilt of 78.69° and a height of 1.55 meters, making the task more straightforward as the floor's surface normal is always the same. Our trained U-Net then achieves an accuracy of 78% on predicting the surface normals of pixels that have an angle difference below 5° on the SUNCG dataset, indicating a high accuracy and prediction performance. However, this does not translate from the synthetic data to the real-world Replica dataset, where the performance drops drastically to 39.85%. Below an angle of 11.5°, our network still manages to get 62.65% correct on the Replica dataset, which is enough for our scene reconstruction network.

Besides a quantitative evaluation, we show a qualitative evaluation in fig. A.2. Our network manages to reconstruct the surfaces of the objects in the scene well, such as the bed or couch. However, it struggles with the blinds in front of the window or the changes in color on the white wall as shown in the third row. Our U-Net manages to predict the surface normals on the challenging chairs in the lowest row in fig. A.2, enabling a scene reconstruction of these chairs.

Surface Normal Prediction for SVR-IC

We use a similar U-Net architecture to predict surface normale images for our SVR-IC method. It uses as an input a squared RGB image with a side resolution of 512, where we rely on our dilated inception layers defined in section 5.2. Each inception layer splits the input in three and uses the first 50% with a dilation of one, while the other half is split in two and uses a dilation of two and four. Using such an inception layer, we increase the feature size from 3 to 32 in the first layer. After it, we use max pooling to reduce the spatial size to 256. The output of each convolution before the pooling operation in this U-Net will be later concatenated with the result of the transposed convolutions with the matching spatial resolution. After the pooling, we perform two inception layers with a feature channel size of 64, ending with another pooling operation. This setup gets repeated for the subsequent lower spatial resolution of 128 to reach 64. At a spatial resolution of 64, we only perform one inception layer with 128 feature channels. A pooling operation is then used as start in our deepest layer; here, we use six inception layers, the first and last two use 256 feature channels, while

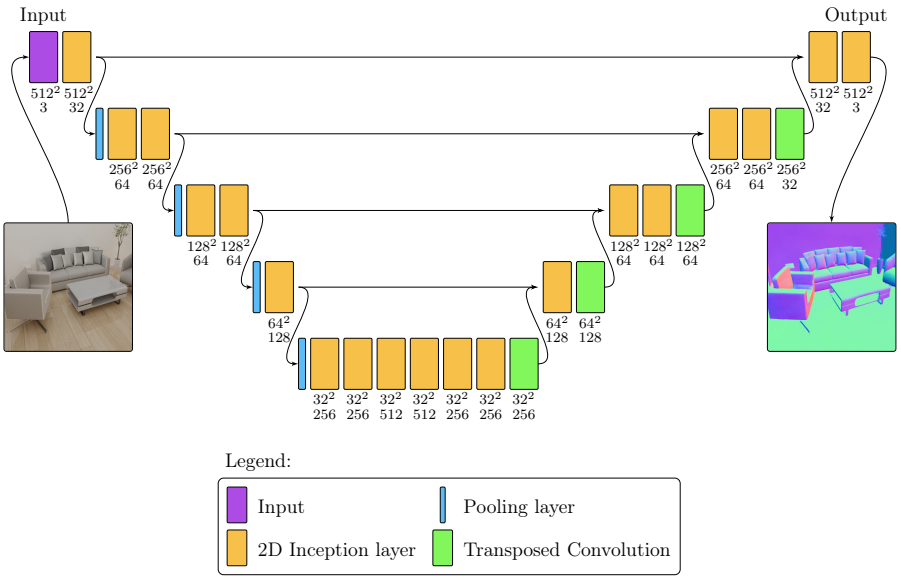


Figure A.3: Our U-Net architecture. It predicts the input for the surface normals in the SVR-IC method, enabling a method that only relies on color images.

Table A.2: Surface normal results on the 3D-FRONT dataset. The upper row shows the results using our proposed texture swapping, while the second row only uses the color and surface normal images for the training. In the last row, we reduce the inception layer capacity.

Training dataset or ablation	Mean Angle	Angel Difference				
		5°	11.5°	22.5°	30°	60°
$\mathbb{I}_{\text{3D-FRONT}}$ and $\mathbb{I}^{\circlearrowleft}_{\text{3D-FRONT}}$	5.96°	75.65	89.30	94.69	96.26	98.70
Only $\mathbb{I}_{\text{3D-FRONT}}$	7.78°	66.26	85.64	92.35	94.40	97.95
Reduced inception layer	6.65°	68.17	87.80	94.30	96.04	98.64

Table A.3: Surface normal results on the Replica dataset. The upper row uses both the normal color and texture-swapped color images, while the second row relies on the color images without texture-swapping. The last row contains the results for a reduced inception layer capacity.

Training dataset or ablation	Mean Angle	Angel Difference				
		5°	11.5°	22.5°	30°	60°
$\mathbb{I}_{\text{3D-FRONT}}$ and $\mathbb{I}^{\circlearrowleft}_{\text{3D-FRONT}}$	13.64°	44.08	67.72	81.48	86.36	95.63
Only $\mathbb{I}_{\text{3D-FRONT}}$	17.61°	35.03	59.36	74.66	80.42	92.74
Reduced inception layer	15.43°	37.11	63.13	78.92	84.34	94.61

the two in the middle have 512 feature channels. After this reduction to a spatial size of 32, we use a transposed convolution with a stride of two to increase the spatial resolution to 64. The output is concatenated with the result of the downstream part. We then use the same amount of inception layers and features channels as on the way down while replacing the pooling with a transposed convolution. These transposed convolutions have a fixed dilation rate of one. The output is an image with three channels containing the surface normal vector for each pixel in the input image. This architecture is visualized in fig. A.3.

In tables A.2 and A.3, we compare the effects of randomizing the textures, as described in section 6.1.4. For this evaluation, we rely on the same color

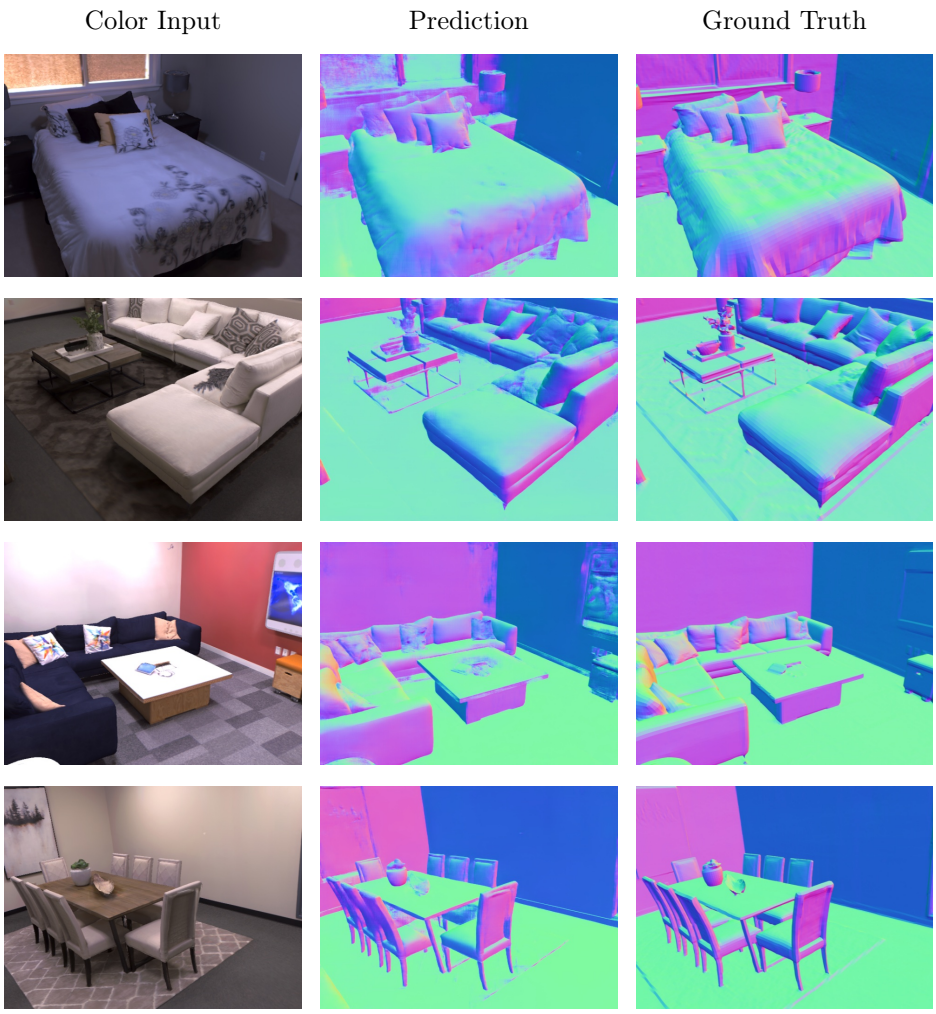


Figure A.4: Four examples of our U-Net on the Replica dataset. Even though we trained with the swapped texture dataset, the texture on the bed in the first row produces tiny details on the bedspread.

images as for evaluating the SVR-IC method in section 7.2.2. Our proposed texture swapping mechanism increases the performance on the 3D-FRONT dataset and the Replica dataset, enabling a mean angle of 13.64° while ensuring that 67% of all predicted surface normals have an error below 11.5°. On the training dataset, the performance is remarkable. On average, the error is below 6% while 75.7% of all vectors have an error below 5°. We also trained a network with reduced dilation rates inside the inception layers. Here, we only use the dilation rates one and two and split the features by half over the two values. This reduces the performance on the 3D-FRONT

dataset and the Replica dataset, which shows the importance of our dilated layers.

In fig. A.4, we demonstrate the performance of our trained U-Net. Even though our network was only trained on synthetic data, it performs well on the real-world Replica dataset. Nonetheless, the training with the $\mathbb{I}^{\circlearrowleft}_{\text{3D-FRONT}}$ does not remove all texture bias, as in the first row, the texture of the bedspread influence the surface normals. The texture of the blinds behind the bed confuses the normal prediction completely, leading to a broken scene reconstruction of this part, as can be seen in the third row of fig. 7.9. Strong textures on objects are hard to handle; another example is shown in the third row, where the screen on the wall and the images on the pillow confuse the network. Even with these minor mistakes, the overall surface normal reconstruction performance is excellent and is a valuable input to the scene reconstruction network. It is even capable of capturing fine details, such as the table and chair legs in the last row of fig. A.4.

Bibliography

[1] E. Ahmed, A. Saint, A. E. R. Shabayek, K. Cherenkova, R. Das, G. Gusev, D. Aouada, and B. Ottersten. A survey on deep learning advances on different 3d data representations. *arXiv preprint arXiv:1808.01462*, 2018.

[2] M. Akmal Butt and P. Maragos. Optimum design of chamfer distance transforms. *IEEE Transactions on Image Processing*, 7(10):1477–1484, 1998. doi: 10.1109/83.718487.

[3] Apple. Roomplan. Online, June 2022. URL `https://developer.apple.com/augmented-reality/roomplan/`.

[4] A. Avetisyan, T. Khanova, C. Choy, D. Dash, A. Dai, and M. Nießner. Scenecad: Predicting object alignments and layouts in rgb-d scans. In *European Conference on Computer Vision*, pages 596–612. Springer, 2020.

[5] S. Back, J. Lee, T. Kim, S. Noh, R. Kang, S. Bak, and K. Lee. Unseen object amodal instance segmentation via hierarchical occlusion modeling. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 5085–5092, 2022. doi: 10.1109/ICRA46639.2022.9811646.

[6] J. T. Barron, B. Mildenhall, M. Tancik, P. Hedman, R. Martin-Brualla, and P. P. Srinivasan. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5855–5864, 2021.

[7] F. Bartell, E. Dereniak, and W. Wolfe. The theory and measurement of bidirectional reflectance distribution function (brdf) and bidirectional transmittance distribution function (btdf). In G. H. Hunt, editor,

Radiation Scattering in Optical Systems, volume 0257, pages 154 – 160. International Society for Optics and Photonics, SPIE, 1981. doi: 10.1117/12.959611. URL https://doi.org/10.1117/12.959611.

[8] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975. ISSN 0001-0782. doi: 10.1145/361002.361007. URL https://doi.org/10.1145/361002.361007.

[9] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE transactions on visualization and computer graphics*, 5(4):349–359, 1999.

[10] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. URL http://www.blender.org.

[11] D. Blythe and T. McReynolds. Opengl implementations, 2005.

[12] J.-D. Boissonnat and B. Geiger. Three-dimensional reconstruction of complex shapes based on the delaunay triangulation. In *Biomedical image processing and biomedical visualization*, volume 1905, pages 964–975. SPIE, 1993.

[13] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, and B. Lévy. *Polygon mesh processing*. CRC press, 2010.

[14] D. R. Canelhas. *Truncated signed distance fields applied to robotics*. PhD thesis, Örebro University, 2017.

[15] A. Caporali, K. Galassi, R. Zanella, and G. Palli. Fastdlo: Fast deformable linear objects instance segmentation. *IEEE Robotics and Automation Letters*, 7(4):9075–9082, 2022. doi: 10.1109/LRA.2022.3189791.

[16] R. Chabra, J. E. Lenssen, E. Ilg, T. Schmidt, J. Straub, S. Lovegrove, and R. Newcombe. Deep local shapes: Learning local sdf priors for detailed 3d reconstruction. *European Conference on Computer Vision (ECCV)*, Mar. 2020.

[17] A. Chakrabarti, J. Shao, and G. Shakhnarovich. Depth from a single image by harmonizing overcomplete local network predictions. *Advances in Neural Information Processing Systems*, 29, 2016.

[18] A. Chang, A. Dai, T. Funkhouser, M. Halber, M. Niessner, M. Savva, S. Song, A. Zeng, and Y. Zhang. Matterport3d: Learning from rgb-d data in indoor environments. *International Conference on 3D Vision (3DV)*, 2017.

[19] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, J. Xiao, L. Yi, and F. Yu. Shapenet: An information-rich 3d model repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

[20] X. Chen, Q. Zhang, X. Li, Y. Chen, Y. Feng, X. Wang, and J. Wang. Hallucinated neural radiance fields in the wild. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12943–12952, 2022.

[21] Z. Chen, Y. Zhang, K. Genova, S. Fanello, S. Bouaziz, C. Häne, R. Du, C. Keskin, T. Funkhouser, and D. Tang. Multiresolution deep implicit functions for 3d shape representation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 13087–13096, 2021.

[22] J. Chibane, A. Mir, and G. Pons-Moll. Neural unsigned distance fields for implicit function learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, December 2020.

[23] E. Coumans. Bullet physics simulation. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336345. doi: 10.1145/2776880.2792704. URL `https://doi.org/10.1145/2776880.2792704`.

[24] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996.

[25] A. K. Cyril Concolato, Paul Kerr. Av1 image file format (avif). Standard, Alliance for Open Media, Feb. 2019.

[26] M. Dahnert, J. Hou, M. Nießner, and A. Dai. Panoptic 3d scene reconstruction from a single rgb image. In *Thirty-Fifth Conference on Neural Information Processing Systems*, 2021.

[27] A. Dai, C. Ruizhongtai Qi, and M. Niessner. Shape completion using 3d-encoder-predictor cnns and shape synthesis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[28] A. Dai, D. Ritchie, M. Bokeloh, S. Reed, J. Sturm, and M. Nießner. Scancomplete: Large-scale scene completion and semantic segmentation for 3d scans. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4578–4587, 2018.

[29] S. Dasgupta, K. Fang, K. Chen, and S. Savarese. Delay: Robust spatial layout estimation for cluttered indoor scenes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 616–624, 2016.

[30] L. Demes. Ambientcg. `https://ambientcg.com/`, Mar. 2022.

[31] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[32] M. Denninger and R. Triebel. Persistent anytime learning of objects from unseen classes. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4075–4082. IEEE, 2018.

[33] M. Denninger and R. Triebel. 3d scene reconstruction from a single viewport. In *European Conference on Computer Vision*, pages 51–67. Springer, 2020.

[34] M. Denninger and R. Triebel. 3d semantic scene reconstruction from a single viewport. In *International Conference on Image Processing and Vision Engineering (IMPROVE)*, 2022.

[35] M. Denninger, M. Sundermeyer, D. Winkelbauer, Y. Zidan, D. Olefir, M. Elbadrawy, A. Lodhi, and H. Katam. Blenderproc. *arXiv:1911.01911*, 2019.

[36] M. Denninger, M. Sundermeyer, D. Winkelbauer, D. Olefir, T. Hodan, Y. Zidan, M. Elbadrawy, M. Knauer, H. Katam, and A. Lodhi. Blenderproc: Reducing the reality gap with photorealistic rendering. In *International Conference on Robotics: Sciene and Systems, RSS Workshop on Closing the Reality Gap in Sim2Real Transfer for Robotics*, 2020.

[37] M. Denninger, D. Winkelbauer, M. Sundermeyer, W. Boerdijk, M. Knauer, K. Strobl, M. Humt, and R. Triebel. Blenderproc2: A procedural pipeline for photorealistic rendering. In *The Journal of Open Source Software (JOSS)*, 2022.

[38] P. Dlouhý, V. Duha, K. Húserková, M. Rygalová, A. Krhánek, M. Radjabov, and A. Bajwa. Blenderkit. `https://www.blenderkit.com/`, Mar. 2022.

[39] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. Van Der Smagt, D. Cremers, and T. Brox. Flownet: Learning optical flow with convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2758–2766, 2015.

[40] T. Duhautbout, J. Moras, and J. Marzat. Distributed 3d tsdf manifold mapping for multi-robot systems. In *2019 European Conference on Mobile Robots (ECMR)*, pages 1–8, 2019. doi: 10.1109/ECMR.2019.8870930.

[41] M. Durner, W. Boerdijk, M. Sundermeyer, W. Friedl, Z.-C. Márton, and R. Triebel. Unknown object segmentation from stereo images. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4823–4830. IEEE, 2021.

[42] D. Dwibedi, I. Misra, and M. Hebert. Cut, paste and learn: Surprisingly easy synthesis for instance detection. In *Proceedings of the IEEE international conference on computer vision*, pages 1301–1310, 2017.

[43] D. Eberly. Geometric tools. `https://www.geometrictools.com/Documentation/DistancePoint3Triangle3.pdf`, Sept. 1999.

[44] D. Eigen, C. Puhrsch, and R. Fergus. Depth map prediction from a single image using a multi-scale deep network. *Advances in neural information processing systems*, 27, 2014.

[45] J. Engel, T. Schöps, and D. Cremers. LSD-SLAM: Large-scale direct monocular SLAM. In *European Conference on Computer Vision (ECCV)*, September 2014.

[46] M. Firman, O. Mac Aodha, S. Julier, and G. J. Brostow. Structured prediction of unobserved voxels from a single depth image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5431–5440, 2016.

[47] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Introduction to computer graphics*, volume 55. Addison-Wesley Reading, 1994.

[48] H. Fu, B. Cai, L. Gao, L.-X. Zhang, J. Wang, C. Li, Q. Zeng, C. Sun, R. Jia, B. Zhao, et al. 3d-front: 3d furnished rooms with layouts and semantics. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10933–10942, 2021.

[49] H. Fu, R. Jia, L. Gao, M. Gong, B. Zhao, S. Maybank, and D. Tao. 3d-future: 3d furniture shape with texture. *International Journal of Computer Vision*, pages 1–25, 2021.

[50] M. Fuchs, C. Borst, P. Robuffo Giordano, A. Baumann, E. Kraemer, J. Langwald, R. Gruber, N. Seitz, G. Plank, K. Kunze, R. Burger, F. Schmidt, T. Wimboeck, and G. Hirzinger. Rollin' justin - design considerations and realization of a mobile platform for a humanoid upper body. In *2009 IEEE International Conference on Robotics and Automation*, pages 4131–4137, 2009. doi: 10.1109/ROBOT.2009.5152464.

[51] K. Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3):121–136, 1975.

[52] R. Geirhos, P. Rubisch, C. Michaelis, M. Bethge, F. A. Wichmann, and W. Brendel. Imagenet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=Bygh9j09KX`.

[53] G. Gkioxari, J. Malik, and J. Johnson. Mesh r-cnn. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9785–9795, 2019.

[54] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[55] K. Greff, F. Belletti, L. Beyer, C. Doersch, Y. Du, D. Duckworth, D. J. Fleet, D. Gnanapragasam, F. Golemo, C. Herrmann, T. Kipf, A. Kundu, D. Lagun, I. Laradji, H.-T. D. Liu, H. Meyer, Y. Miao, D. Nowrouzezahrai, C. Oztireli, E. Pot, N. Radwan, D. Rebain, S. Sabour, M. S. M. Sajjadi, M. Sela, V. Sitzmann, A. Stone, D. Sun, S. Vora, Z. Wang, T. Wu, K. M. Yi, F. Zhong, and A. Tagliasacchi. Kubric: a scalable dataset generator. 2022.

[56] T. Groueix, M. Fisher, V. Kim, B. Russell, and M. Aubry. Atlasnet: A papier-mâché approach to learning 3d surface generation. arxiv 2018. *arXiv preprint arXiv:1802.05384*, 2018.

[57] C. Gümeli, A. Dai, and M. Nießner. Roca: Robust cad model retrieval and alignment from a single image. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4022–4031, 2022.

[58] Y. Guo, H. Wang, Q. Hu, H. Liu, L. Liu, and M. Bennamoun. Deep learning for 3d point clouds: A survey, 2019. URL https://arxiv.org/abs/1912.12033.

[59] Z. Guo, Z. Zhang, R. Feng, and Z. Chen. Causal contextual prediction for learned image compression. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(4):2329–2341, 2021.

[60] A. Hagengruber and L. Suchenwirth. Smile2gether: A prototype of a holistic ecosystem for robotic care assistants. *Open-Access-Publikation im Sinne der CC-Lizenz BY 4.0*, page 227, 2022.

[61] T. Y. Ham and D. A. Guerin. A cross-cultural comparison of preference for visual attributes in interior environments: America and china. *Journal of Interior Design*, 30(1):37–50, 2004. doi: https://doi.org/10.1111/j.1939-1668.2004.tb00398.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1939-1668.2004.tb00398.x.

[62] M. Han, Z. Zhang, Z. Jiao, X. Xie, Y. Zhu, S.-C. Zhu, and H. Liu. Reconstructing interactive 3d scenes by panoptic mapping and cad

model alignments. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 12199–12206. IEEE, 2021.

[63] X. Han, Z. Li, H. Huang, E. Kalogerakis, and Y. Yu. High-resolution shape completion using deep neural networks for global structure and local geometry inference. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[64] A. Handa, V. Patraucean, V. Badrinarayanan, S. Stent, and R. Cipolla. Scenenet: Understanding real world indoor scenes with synthetic data. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Nov. 2015.

[65] R. Hanocka, A. Hertz, N. Fish, R. Giryes, S. Fleishman, and D. Cohen-Or. Meshcnn: A network with an edge. *ACM Trans. Graph.*, Sept. 2018. doi: 10.1145/3306346.3322959.

[66] N. Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[67] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585 (7825):357–362, Sept. 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

[68] F. Hausdorff. *Grundzüge der mengenlehre*, volume 7. von Veit, 1914.

[69] D. He, Y. Zheng, B. Sun, Y. Wang, and H. Qin. Checkerboard context model for efficient learned image compression. *CVPR 2021*, 2021. doi: 10.1109/cvpr46437.2021.01453.

[70] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[71] S. Hinterstoisser, V. Lepetit, P. Wohlhart, and K. Konolige. On pre-trained image features and synthetic images for deep learning. In

Proceedings of the European Conference on Computer Vision (ECCV) Workshops, pages 682–697, 2018. doi: 10.1007/978-3-030-11009-3_42.

[72] S. Hinterstoisser, O. Pauly, H. Heibel, M. Martina, and M. Bokeloh. An annotation saved is an annotation earned: Using fully synthetic training for object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.

[73] T. Hodan, F. Michel, E. Brachmann, W. Kehl, A. G. Buch, D. Kraft, B. Drost, J. Vidal, S. Ihrke, X. Zabulis, C. Sahin, F. Manhardt, F. Tombari, T.-K. Kim, J. Matas, and C. Rother. Bop: Benchmark for 6d object pose estimation. *European Conference on Computer Vision (ECCV)*, Aug. 2018.

[74] T. Hodaň, V. Vineet, R. Gal, E. Shalev, J. Hanzelka, T. Connell, P. Urbina, S. N. Sinha, and B. Guenter. Photorealistic image synthesis for object instance detection. In *2019 IEEE international conference on image processing (ICIP)*, pages 66–70. IEEE, 2019.

[75] T. Hodan, M. Sundermeyer, B. Drost, Y. Labbe, E. Brachmann, F. Michel, C. Rother, and J. Matas. Bop challenge 2020 on 6d object localization. *European Conference on Computer Vision Workshops (ECCVW)*, Sept. 2020.

[76] J. Hou, A. Dai, and M. Nießner. 3d-sis: 3d semantic instance segmentation of rgb-d scans. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4421–4430, 2019.

[77] J. Hou, A. Dai, and M. Nießner. Revealnet: Seeing behind objects in rgb-d scans. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2098–2107, 2020.

[78] R. Hu, N. Ravi, A. C. Berg, and D. Pathak. Worldsheet: Wrapping the world in a 3d sheet for view synthesis from a single image. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12528–12537, 2021.

[79] S. Huang, S. Qi, Y. Zhu, Y. Xiao, Y. Xu, and S.-C. Zhu. Holistic 3d scene parsing and reconstruction from a single rgb image. In *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[80] ISO 23008-12:2017. Information technology — high efficiency coding and media delivery in heterogeneous environments — part 12: Image file format. Standard, International Organization for Standardization, Dec. 2017.

[81] H. Izadinia, Q. Shan, and S. M. Seitz. Im2cad. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5134–5143, 2017.

[82] P. Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, Feb. 1912. URL `http://www.jstor.org/stable/2427226?seq=3`.

[83] A. Kendall and Y. Gal. What uncertainties do we need in bayesian deep learning for computer vision? *Advances in neural information processing systems*, 30, 2017.

[84] H. Kim, J. Moon, and B. Lee. Rgb-to-tsdf: Direct tsdf prediction from a single rgb image for dense 3d reconstruction. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6714–6720, 2019. doi: 10.1109/IROS40897.2019.8968566.

[85] M. Knauer, M. Denninger, and R. Triebel. Recall: Rehearsal-free continual learning for object classification. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022.

[86] M. W. Knauer, M. Denninger, and R. Triebel. Hows-cl-25: Household objects within simulation dataset for continual learning. *Zenodo.org*, 2022.

[87] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL `https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`.

[88] W. Kuo, A. Angelova, T.-Y. Lin, and A. Dai. Mask2cad: 3d shape prediction by learning to segment and retrieve. In *European Conference on Computer Vision*, pages 260–277. Springer, 2020.

[89] W. Kuo, A. Angelova, T.-Y. Lin, and A. Dai. Patch2cad: Patchwise embedding learning for in-the-wild shape retrieval from a single image. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12589–12599, 2021.

[90] I. Laina, C. Rupprecht, V. Belagiannis, F. Tombari, and N. Navab. Deeper depth prediction with fully convolutional residual networks. In *2016 Fourth international conference on 3D vision (3DV)*, pages 239–248. IEEE, 2016.

[91] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[92] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

[93] B. Li, C. Shen, Y. Dai, A. Van Den Hengel, and M. He. Depth and surface normal estimation from monocular images using regression on deep features and hierarchical crfs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1119–1127, 2015.

[94] Z. Li and N. Snavely. Cgintrinsics: Better intrinsic image decomposition through physically-based rendering. In *Proceedings of the European conference on computer vision (ECCV)*, pages 371–387, 2018.

[95] Y. Liao, S. Donne, and A. Geiger. Deep marching cubes: Learning explicit surface representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2916–2925, 2018.

[96] N. Y.-S. Lii, P. Schmaus, D. Leidner, T. Krueger, J. Grenouilleau, A. Pereira, A. Giuliano, A. S. Bauer, A. Köpken, F. Lay, M. Sewtz, N. Bechtel, S. Bustamante Gomez, M. Denninger, W. Friedl, et al. Introduction to surface avatar: the first heterogeneous robotic team to be commanded with scalable autonomy from the iss. In *Proceedings of the International Astronautical Congress, IAC*. International Astronautical Federation, IAF, 2022.

[97] J. J. Lim, H. Pirsiavash, and A. Torralba. Parsing ikea objects: Fine pose estimation. In *Proceedings of the IEEE international conference on computer vision*, pages 2992–2999, 2013.

[98] F. Liu, C. Shen, and G. Lin. Deep convolutional neural fields for depth estimation from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5162–5170, 2015.

[99] M. Liu, M. Salzmann, and X. He. Discrete-continuous depth estimation from a single image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 716–723, 2014.

[100] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4): 163–169, aug 1987. ISSN 0097-8930. doi: 10.1145/37402.37422. URL `https://doi.org/10.1145/37402.37422`.

[101] N. Mahmood, N. Ghorbani, N. F. Troje, G. Pons-Moll, and M. J. Black. AMASS: Archive of motion capture as surface shapes. In *International Conference on Computer Vision*, pages 5442–5451, Oct. 2019.

[102] R. Maier, R. Schaller, and D. Cremers. Efficient online surface correction for real-time large-scale 3D reconstruction. In *British Machine Vision Conference (BMVC)*, London, United Kingdom, September 2017.

[103] A. Mallya and S. Lazebnik. Learning informative edge maps for indoor scene layout prediction. In *Proceedings of the IEEE international conference on computer vision*, pages 936–944, 2015.

[104] R. Martin-Brualla, N. Radwan, M. S. Sajjadi, J. T. Barron, A. Dosovitskiy, and D. 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.

[105] J. McCormac, A. Handa, S. Leutenegger, and A. J. Davison. Scenenet rgb-d: Can 5m synthetic images beat generic imagenet pre-training on indoor segmentation? In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2678–2687, 2017.

[106] D. Meagher. Geometric modeling using octree encoding. *Computer graphics and image processing*, 19(2):129–147, 1982.

[107] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*, pages 405–421. Springer, 2020.

[108] B. Mildenhall, P. Hedman, R. Martin-Brualla, P. P. Srinivasan, and J. T. Barron. Nerf in the dark: High dynamic range view synthesis from noisy raw images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16190–16199, 2022.

[109] S. S. Mohammadi, Y. Wang, and A. Del Bue. Pointview-gcn: 3d shape classification with multi-view point clouds. In *2021 IEEE International Conference on Image Processing (ICIP)*, pages 3103–3107. IEEE, 2021.

[110] N. Morrical, J. Tremblay, Y. Lin, S. Tyree, S. Birchfield, V. Pascucci, and I. Wald. Nvisii: A scriptable tool for photorealistic image generation, 2021.

[111] Y. Movshovitz-Attias, T. Kanade, and Y. Sheikh. How useful is photo-realistic rendering for visual learning? In *European conference on computer vision*, pages 202–217. Springer, 2016.

[112] A. Nealen, T. Igarashi, O. Sorkine, and M. Alexa. Laplacian mesh optimization. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 381–389, 2006.

[113] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*, pages 127–136. Ieee, 2011.

[114] Y. Nie, X. Han, S. Guo, Y. Zheng, J. Chang, and J. J. Zhang. Total3dunderstanding: Joint layout, object pose and mesh reconstruction for indoor scenes from a single image. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 55–64, 2020.

[115] J. o'Rourke et al. *Computational geometry in C*. Cambridge university press, 1998.

[116] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019.

[117] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: A general purpose ray tracing engine. *ACM Trans. Graph.*, 29 (4), jul 2010. ISSN 0730-0301. doi: 10.1145/1778765.1778803. URL `https://doi.org/10.1145/1778765.1778803`.

[118] A. Paullada, I. D. Raji, E. M. Bender, E. Denton, and A. Hanna. Data and its (dis)contents: A survey of dataset development and use in machine learning research. *Patterns*, 2(11):100336, 2021. ISSN 2666-3899. doi: https://doi.org/10.1016/j.patter.2021.100336. URL `https://www.sciencedirect.com/science/article/pii/S2666389921001847`.

[119] S. Peng, C. M. Jiang, Y. Liao, M. Niemeyer, M. Pollefeys, and A. Geiger. Shape as points: A differentiable poisson solver. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[120] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016. ISBN 0128006455.

[121] C. R. Qi, H. Su, K. Mo, and L. 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 (CVPR)*, pages 652–660, 2017.

[122] M. Rad and V. Lepetit. Bb8: A scalable, accurate, robust to partial occlusion method for predicting the 3d poses of challenging objects without using depth. In *Proceedings of the IEEE international conference on computer vision*, pages 3828–3836, 2017.

[123] M. Ramamonjisoa and V. Lepetit. Sharpnet: Fast and accurate recovery of occluding contours in monocular depth estimation. In *Proceedings*

of the IEEE/CVF International Conference on Computer Vision Workshops, pages 0–0, 2019.

[124] M. Ramamonjisoa, Y. Du, and V. Lepetit. Predicting sharp and accurate occlusion boundaries in monocular depth estimation using displacement fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14648–14657, 2020.

[125] W. Richard, S. Grünvogel, and B. Grützmacher. How realistic is realism? considerations on the aesthetics of computer games. In *International Conference on Entertainment Computing*, volume 3166, pages 216–225, 09 2004. ISBN 978-3-540-22947-6. doi: 10.1007/978-3-540-28643-1_28.

[126] S. R. Richter and S. 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.

[127] S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. In *European conference on computer vision*, pages 102–118. Springer, 2016.

[128] S. R. Richter, Z. Hayder, and V. Koltun. Playing for benchmarks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2213–2222, 2017.

[129] G. Riegler, A. Osman Ulusoy, and A. Geiger. Octnet: Learning deep 3d representations at high resolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3577–3586, 2017.

[130] O. Ronneberger, P. Fischer, and T. 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.

[131] M. Savva, A. Kadian, O. Maksymets, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, et al. Habitat: A platform for embodied ai research. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 9339–9347, 2019.

[132] A. Saxena, S. Chung, and A. Ng. Learning depth from single monocular images. *Advances in neural information processing systems*, 18, 2005.

[133] A. Saxena, M. Sun, and A. Y. Ng. Make3d: Learning 3d scene structure from a single still image. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):824–840, 2008.

[134] M. Schwarz and S. Behnke. Stillleben: Realistic scene synthesis for deep learning in robotics. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10502–10508. IEEE, 2020.

[135] D. Seichter, M. Köhler, B. Lewandowski, T. Wengefeld, and H.-M. Gross. Efficient rgb-d semantic segmentation for indoor scene analysis. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 13525–13531, 2021.

[136] P. Shilane, P. Min, M. M. Kazhdan, and T. A. Funkhouser. The princeton shape benchmark. In *2004 International Conference on Shape Modeling and Applications (SMI 2004), 7-9 June 2004, Genova, Italy*, pages 167–178. IEEE Computer Society, 2004. doi: 10.1109/SMI.2004.1314504.

[137] D. Shin, Z. Ren, E. B. Sudderth, and C. C. Fowlkes. 3d scene reconstruction with multi-layer depth and epipolar transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2172–2182, 2019.

[138] P. Shirley, M. Ashikhmin, and S. Marschner. *Fundamentals of computer graphics*. AK Peters/CRC Press, 2009.

[139] D. Shreiner, B. T. K. O. A. W. Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009.

[140] I. Sipiran, R. Gregor, and T. Schreck. Approximate symmetry detection in partial 3d meshes. In *Computer Graphics Forum*, volume 33, pages 131–140. Wiley Online Library, 2014.

[141] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems*, 33:7462–7473, 2020.

[142] A. R. Smith. Tint fill. In *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '79, page 276–283, New York, NY, USA, 1979. Association for Computing Machinery. ISBN 0897910044. doi: 10.1145/800249.807456. URL https://doi.org/10.1145/800249.807456.

[143] S. Song, F. Yu, A. Zeng, A. X. Chang, M. Savva, and T. Funkhouser. Semantic scene completion from a single depth image. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Nov. 2016.

[144] O. Sorkine and D. Cohen-Or. Least-squares meshes. In *Proceedings Shape Modeling Applications, 2004.*, pages 191–199. IEEE, 2004.

[145] J. Straub, T. Whelan, L. Ma, Y. Chen, E. Wijmans, S. Green, J. J. Engel, R. Mur-Artal, C. Ren, S. Verma, A. Clarkson, M. Yan, B. Budge, Y. Yan, X. Pan, J. Yon, Y. Zou, K. Leon, N. Carter, J. Briales, T. Gillingham, E. Mueggler, L. Pesqueira, M. Savva, D. Batra, H. M. Strasdat, R. D. Nardi, M. Goesele, S. Lovegrove, and R. Newcombe. The Replica dataset: A digital replica of indoor spaces. *arXiv preprint arXiv:1906.05797*, 2019.

[146] H. Su, C. R. Qi, Y. Li, and L. J. Guibas. Render for cnn: Viewpoint estimation in images using cnns trained with rendered 3d model views. In *Proceedings of the IEEE international conference on computer vision*, pages 2686–2694, 2015.

[147] C. Sun, M. Sun, and H.-T. Chen. Direct voxel grid optimization: Super-fast convergence for radiance fields reconstruction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5459–5469, 2022.

[148] X. Sun, J. Wu, X. Zhang, Z. Zhang, C. Zhang, T. Xue, J. B. Tenenbaum, and W. T. Freeman. Pix3d: Dataset and methods for single-image 3d shape modeling. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[149] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions.

In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[150] T. Takikawa, J. Litalien, K. Yin, K. Kreis, C. Loop, D. Nowrouzezahrai, A. Jacobson, M. McGuire, and S. Fidler. Neural geometric level of detail: Real-time rendering with implicit 3d shapes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11358–11367, 2021.

[151] M. Tancik, P. P., Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. T. Barron, and R. Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *2020 Thirty-fourth Conference on Neural Information Processing Systems (NeurIPS)*, June 2020.

[152] M. Tatarchenko, A. Dosovitskiy, and T. 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.

[153] B. Tekin, S. N. Sinha, and P. Fua. Real-time seamless single shot 6d object pose prediction. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 292–301, 2018.

[154] J. A. A. Thompson and N. Blossom. *The handbook of interior design.* John Wiley & Sons, 2015.

[155] S. Thrun and B. Wegbreit. Shape from symmetry. In *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, volume 2, pages 1824–1831. IEEE, 2005.

[156] T. To, J. Tremblay, D. McKay, Y. Yamaguchi, K. Leung, A. Balanon, J. Cheng, W. Hodge, and S. Birchfield. NDDS: NVIDIA deep learning dataset synthesizer, 2018. `https://github.com/NVIDIA/Dataset_Synthesizer`.

[157] D. Tüfekçioğlu. A historical comparative analysis of european and asian interior spaces through cultural background. *New Trends and Issues Proceedings on Humanities and Social Sciences*, 4(11):222–231, 2017.

[158] S. Tulsiani, A. Kar, J. Carreira, and J. Malik. Learning category-specific deformable 3d models for object reconstruction. *IEEE transactions on pattern analysis and machine intelligence*, 39(4):719–731, 2016.

[159] J. Vince and J. A. Vince. *Mathematics for computer graphics*, volume 3. Springer, 2010.

[160] J. Vogel, D. Leidner, A. Hagengruber, M. Panzirsch, and A. Dietrich. Das projekt smile. In *Cluster-Konferenz Zukunft der Pflege*, pages 212–216, 2018.

[161] J. Vogel, D. Leidner, A. Hagengruber, M. Panzirsch, B. Bauml, M. Denninger, U. Hillenbrand, L. Suchenwirth, P. Schmaus, M. Sewtz, et al. An ecosystem for heterogeneous robotic assistants in caregiving: Core functionalities and use cases. *IEEE Robotics & Automation Magazine*, 28(3):12–28, 2020.

[162] G. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992. doi: 10.1109/30.125072.

[163] G. K. Wallace. The JPEG still picture compression standard. *Commun. ACM*, 34(4):30–44, 1991. doi: 10.1145/103085.103089.

[164] H. Wang and J. Zhang. A survey of deep learning-based mesh processing. *Communications in Mathematics and Statistics*, 10, 02 2022. doi: 10.1007/s40304-021-00246-7.

[165] N. Wang, Y. Zhang, Z. Li, Y. Fu, W. Liu, and Y.-G. Jiang. Pixel2mesh: Generating 3d mesh models from single rgb images. *ECCV2018*, Apr. 2018.

[166] P. Wang, X. Shen, Z. Lin, S. Cohen, B. Price, and A. L. Yuille. Towards unified depth and semantic prediction from a single image. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2800–2809, 2015.

[167] P.-S. Wang, Y. Liu, Y.-X. Guo, C.-Y. Sun, and X. Tong. O-cnn: Octree-based convolutional neural networks for 3d shape analysis. *ACM Trans. Graph.*, 36(4), jul 2017. ISSN 0730-0301. doi: 10.1145/3072959.3073608. URL `https://doi.org/10.1145/3072959.3073608`.

[168] P.-S. Wang, Y. Liu, Y.-Q. Yang, and X. Tong. Spline positional encoding for learning 3d implicit signed distance fields. *arXiv preprint arXiv:2106.01553*, 2021.

[169] F. Williams, Z. Gojcic, S. Khamis, D. Zorin, J. Bruna, S. Fidler, and O. 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.

[170] D. Winkelbauer, M. Denninger, and R. Triebel. Learning to localize in new environments from synthetic training data. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5840–5846, 2021. doi: 10.1109/ICRA48506.2021.9560872.

[171] N. Wongwaen, S. Tiendee, and C. Sinthanayothin. Method of 3d mesh reconstruction from point cloud using elementary vector and geometry analysis. In *2012 8th International Conference on Information Science and Digital Content Technology (ICIDT2012)*, volume 1, pages 156–159. IEEE, 2012.

[172] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3d shapenets: A deep representation for volumetric shapes. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1912–1920, 2015. doi: 10.1109/CVPR.2015.7298801.

[173] H. Xie, H. Yao, S. Zhang, S. Zhou, and W. Sun. Pix2vox++: Multi-scale context-aware 3d object reconstruction from single and multiple images. *International Journal of Computer Vision*, 128(12):2919–2935, 2020.

[174] D. Xu, E. Ricci, W. Ouyang, X. Wang, and N. Sebe. Multi-scale continuous crfs as sequential deep networks for monocular depth estimation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5354–5362, 2017.

[175] Q. Xu, W. Wang, D. Ceylan, R. Mech, and U. Neumann. Disn: Deep implicit surface network for high-quality single-view 3d reconstruction. *Advances in Neural Information Processing Systems*, 32, 2019.

[176] Q. Xu, Z. Xu, J. Philip, S. Bi, Z. Shu, K. Sunkavalli, and U. Neumann. Point-nerf: Point-based neural radiance fields. In *Proceedings of the*

IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 5438–5448, 2022.

[177] X. Yan, J. Yang, E. Yumer, Y. Guo, and H. Lee. Perspective transformer nets: Learning single-view 3d object reconstruction without 3d supervision. *Advances in neural information processing systems*, 29, 2016.

[178] W. Yantao. Research on the application of chinese traditional culture elements in modern interior design. 2019.

[179] S. Yao, F. Yang, Y. Cheng, and M. G. Mozerov. 3d shapes local geometry codes learning with sdf. *European Conference on Computer Vision (ECCV)*, Aug. 2021.

[180] W. Yifan, L. Rahmann, and O. Sorkine-Hornung. Geometry-consistent neural shape representation with implicit displacement fields. *arXiv preprint arXiv:2106.05187*, 2021.

[181] G. Zaal and R. Tuytel. Polyhaven. https://polyhaven.com/, Mar. 2022.

[182] C. Zhang, Z. Cui, Y. Zhang, B. Zeng, M. Pollefeys, and S. Liu. Holistic 3d scene understanding from a single image with implicit representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8833–8842, 2021.

[183] Y. Zhang and T. Funkhouser. Deep depth completion of a single rgb-d image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 175–185, 2018.

[184] Y. Zhang, S. Song, E. Yumer, M. Savva, J.-Y. Lee, H. Jin, and T. Funkhouser. Physically-based rendering for indoor scene understanding using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5287–5295, 2017.

[185] W. Zhao, S. Gao, and H. Lin. A robust hole-filling algorithm for triangular mesh. *The Visual Computer*, 23(12):987–997, 2007.