PÁZMÁNY
Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

THESIS WORK

Human Interpretable Visualization of 3D Sonar Data Using AI Technologies

Csatári, Dominik
Computer Science Engineering MSc

Supervisor: Dr. Hatvani, Janka
Faculty Mentor: Dr. Cserey, György Gábor

2024

PÁZMÁNY
Pázmány Péter Catholic University
1635

Institutional ID: FI79633
H-1088 Budapest, Szentkirályi Str. 28.
www.ppke.hu

THESIS PROPOSAL FORM

Name (Neptun ID): **Csatári Dominik** (FV1TW4)
Study program (ID): **Computer Science Engineering MSc** (IMNI-AMI)

Supervisor

Dr. Cserey György Gábor (inner supervisor)

Dr. Hatvani Janka (inner supervisor)

Thesis

Title:
Human Interpretable Visualization of 3D Sonar Data Using AI Technologies

Summary of the thesis:

For safer and more efficient rescues, divers need continuous, up-to-date visual information about the relevant area or object, aligned with their current orientation, even during the rescue itself.
Processing 3D point clouds is an active area of research across various fields, including Lidar imaging and 3D modeling. The focus of my thesis is to enhance diver situational awareness using modern deep learning techniques. By applying these methods and analyzing relevant datasets, I aim to improve the quality of sonar point clouds for different objects and assess the feasibility of potential advancements.

Detailed Task Description:
1. Literature review of existing methods for point cloud enrichment and sonar imaging.
2. Plan the adaptation of Lidar techniques for processing sonar data.
3. Develop a training dataset from existing point clouds. Adapt existing algorithms and transformation methods for processing sonar data.
4. Assess the algorithms on real-world sonar data.

Date of application: **19.09.2024**

Date of acceptance: **24.09.2024**

Date: Budapest, 05.12.2024

Declaration of Authenticity

I, the undersigned Dominik Csatári, student of the Faculty of Information Technology and Bionics at the Pázmány Péter Catholic University, hereby declare that I have prepared the present Thesis Work myself without any unauthorized help, and that I have only used the sources specified in the Thesis Work. I have clearly marked all parts that I have literally taken from other sources or have rewritten or translated while keeping the meaning of the original text, also indicating the source thereof. I have not submitted this Thesis Work to any other study programs.

Dominik Csatári

Abstract

In this Thesis Work, I explored the topic of shape completion, with a focus on its potential for enhancing sonar-generated point clouds. I discussed the challenges associated with sensors such as Mobile Laser Scanning (MLS) and sonar, in particular the issue of incomplete or sparse data. This is a critical issue as valuable information can be lost. I also presented recent solutions to this problem. In addition to introducing the general problems and mathematical formulations associated with this issue, I described several models, with a more detailed focus on two specific models. These models were PCN (Point Completion Network) and MVPCC (Multi-View Based Point Cloud Completion Network for MLS Data). I also provided a more detailed analysis of the architecture of these models. PCN follows an encoder-decoder structure while MVPCC is based on a Generative Adversarial Networks (GAN). I described the design of both models and explained the types of loss functions they use. I presented the MVPCC MLS dataset, which is based on real MLS data. I need these methods because my main goal is to perform a completion task on real sonar data.

I then introduced other datasets, such as ModelNet40 and ShapeNet, and trained the PCN model on both of them. With ModelNet40 I got great results for sparse inputs, even at a really high level of sparsity. The completion of incomplete inputs performed worse. From ShapeNet I only used the watercraft objects. There were 1910 objects in total. The model showed low loss after training, but its predictions lacked visible detail and clarity.

I also created my own implementations of MVPCC based on my projection and reprojection methods. For projection, I introduced two main types: pinhole camera projection and orthographic projection. Both were designed to create four projections from four virtual camera positions that I set up to capture the object from different viewpoints. For the orthographic projection, I created projections using point clouds for both the target

and the input. In addition, I created projections where the target was made directly from the mesh projections. This approach ensures that the entire object is covered, making it easier to mask the background. All projection and reprojection methods were successfully implemented. I trained it on both 1-channel inputs, representing depth maps, and 3-channel inputs, representing the original visible coordinates from a given perspective. Both applications generally produced a completion of the input, but the reprojection failed in both cases. In the future, I would like to improve these results by addressing the existing problems and exploring new ideas to complement my shortcomings.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. György Cserey, for sparking my interest in the topic of my thesis and for his professional support throughout the process. I am equally grateful to my supervisor, Dr. Janka Hatvani, for her significant contributions in guiding me to find solutions and for reviewing my thesis with valuable suggestions for improvement. I also wish to thank Pázmány Péter Catholic University for providing a quality education that greatly supported my studies. Lastly, I want to express my gratitude to my family - especially my grandmother for her thoughtful cooking - my friends, and my partner for creating a supportive environment that enabled me to focus on achieving my academic goals.

Soli Deo gloria!

Contents

Chapter 1

Introduction

1.1. Background

This Thesis Work is motivated by the sinking of the cruise ship Hableány in the Danube and aims to implement a system that will make it much easier to locate and rescue victims in a disaster, even in difficult circumstances. The whole of this project is to create such a system. The whole project consists of more phases and different parts. The scope of my implementation was to explore possible post-processing steps for densifying sparse point clouds or complete partial point clouds created with sonars of a given objects. In this way, we could enhance parts of the objects detected underwater to have a better understanding of its finer, more detailed shapes. Or, as another example, we can detect and visualize missing parts of an object to recover the original form of it.

A point cloud can be created from a measurement made with a sonar or a LiDAR sensor. Real-life data is usually incomplete or too sparse. These issues could limit the classification and segmentation tasks performed on the data. To counteract this, there is an area of research called shape completion. With the properties of neural networks, it tries to complete shapes based on their structure. Newer models even try to create detailed outputs, so no information is lost. These networks promise a lot of opportunities. After completion, a point cloud will have a better structure, giving more feedback of a sunken ship, or a collapsed bridge or finding mines in the water.

1.2. Outline

The summary of the chapters of my thesis work:

Related Works This chapter reviews related works on the functioning of sonar technology and outlines methods for performing shape completion on point clouds, generated by these sensors, based on the architectures described in the literature.

Methods This chapter outlines the tools I have used and the models I have implemented, including the mathematical formulations for projections and reprojections across various variations. It also covers the experimental setups that I have designed or adopted, based on insights from the literature.

Results This chapter presents the results obtained from performing the tasks outlined in the previous chapter, along with some reflections and reconsiderations based on the outcomes.

Summary In this chapter, I summarize my results, draw conclusions, and outline future plans for this project.

Chapter 2

Related Works

In this chapter, I will provide an introduction to sonar sensors, outlining their fundamental structure. Following this, I will discuss sonar systems and their application in generating depth maps or 3D point clouds. Additionally, this chapter will cover shape completion, including an overview of the principal architectures utilized for this purpose.

2.1. Sonars

2.1.1 Introduction of Echo Sounders

Sonar is commonly used as an echo sounder which in general terms means a device, which is able to track or map the bedding with ultrasonic waves under a body of water. In this general setting, these kinds of echo sounder have been in use for more than half a century [1]. For deciding upon the depth or the distance we need to travel till the bedding, a so called Single Beam Echo Sounder is useful. As the name suggests, it is such a hydrographic surveying device, which uses only a single acoustic beam. These kind of echo sounder's structure might be built with the following components:

- **Transducer:** These devices have a piezoelectric transmitter which generates mechanical energy (ultrasound) from an electrical signal. They also have a receiver part, which is able to convert ultrasounds into an electrical signal which can be later processed. Usually the same piezoelectric element processes the signals both way.

- **Pulse Transmission:** The generated pulse is sent out and after that, the device

waits for the return of an ultrasound from the bedding or elsewhere. This is usually compared to a conversation structure's having a "speaking" and "listening" part.

- **Reflection:** when the ultrasound wave hits the bedding a part of the energy is absorbed, part of it is scattered and part of it is reflected back, which can be detected with the transducer. It has a lower amplitude, with respect to the sent out signal. This returned signal is called an echo.

If we measure the time from transmitting an acoustic beam and receiving one with an amplitude above a given threshold, we are able to calculate the time of flight of the original ultrasound wave. With this we can calculate the distance with the following formula. [2.1] The speed of sound in water is calculated as 1500 $\frac{m}{s}$ [2].

$$\text{Depth} = \frac{\text{Speed of Sound} \cdot \text{Time of Flight}}{2} \tag{2.1}$$

2.1.2 Frequency

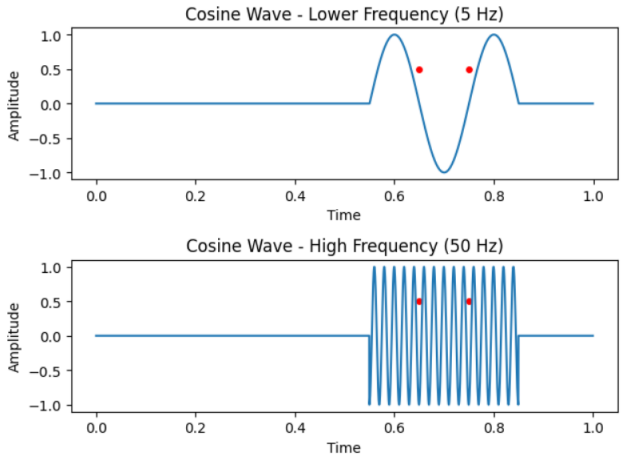

Figure 2.1: Two cosine waves showing 1-1 wave packets. The top, having a frequency of 5Hz and below it one with 50Hz. A red dots are added on both function at the same positions.

In ultrasonic devices, the number of emitted waves over a time period is quantified with the measure of frequencies. These devices are mostly used in a range from 20kHz up to several MHz even GHz in some applications. As for visualization I have created a Figure 2.1, which shows how one wave can detect two different points in the same troughs, not having any crest separating them. This is the case for wave having a smaller frequency. That is why wavelengths can be associated with the resolution of a

imaging process. Longer wavelengths (lower frequencies) produce poorer resolutions. In Figure 2.1, the wave in the upper plot cannot distinguish between the two red scatterers, while the response of the lower wave would produce two separable reflection peaks. However, high frequency waves scatter more, meaning that higher ratio of its energy gets absorbed, limiting the range they can insonate. This exchange between parameters can be understood more easily, telling different kind of resolutions apart.

Resolutions

As ultrasonic devices can be utilized for different types of tasks, it is important to generally define the area in which you decide to use them. You can emphasize different needs, for example range resolution. By this term we generally mean the capability of telling two detection points apart which are close to each other. If these targets are in the same beam line, the general definition of range resolution formulated by the following equation [3][4].

$$\text{Range Resolution [m]} = \frac{\text{Speed of Sound [m/s]} \cdot \text{Pulse Length [s]}}{2} \tag{2.2}$$

Usually, the transmitted ultrasound (because of its finite length) does not consist of a single frequency. It can be better described by a range of frequencies, the bandwidth. This results in an ambiguous pulse length, which is more detectable at higher frequencies. We can derive a connection between the frequency span of the signal (called bandwidth) and the range resolution from Equation 2.2 knowing that

$$\text{Pulse Length (Range)} = \frac{1}{\text{Bandwidth}}. \tag{2.3}$$

Therefore, if we lower the Pulse Length, we would get a better resolution, but also this comes with the cost of having a wider bandwidth, limiting the precision of the measurement [4].

Also, if we are using a multibeam sonar system, we might consider to account for having a great angular range resolution as well. In general, if the target is further away, it is harder to receive accurate information about its position. As the width of the beam is inversely related to the frequency, we can state that a narrower beam would result in a better angular resolution. Diving deeper, we can see that this resolution has a more complex relationship with the setup of our device, namely the beamforming part. Beamforming is the process of optimizing and focusing our ultrasonic energy to a given range. In this process we develop a main lobe, where the concentrated energy is. Note that every array system has side lobes, it is the matter of optimizing the arrangements [1][2][5].

2.1.3 Sonar Mapping Systems

As mentioned in Subsubsection 2.1.2, there are array-based mapping systems, providing greater opportunities for imaging purposes. We can distinguish two major types. Sidescan sonars and Multibeam Echo Sounders, MBES for short. Both techniques have a wide range of usage, as both of them promise a great resolution, having enough transducer elements. As for mapping the bedding, MBES provides a less rigid configuration, meaning the way the beamforming is chosen can be dependent by the current environment, giving us an adaptive parameter to work with. [1].

2.2. Shape Completion

In this section I will briefly talk about the problem statement for which shape completion offers a solution to. I will start with an introduction to point clouds, then work my way down to explaining the necessity of these architectures.

Point clouds are grouped points having 3 values (x, y, z), their spatial coordinates. They sometimes can have a $4th$ dimension, intensity. In our use case it is not that important as the transformed points are already trusted in a sense that they are usually based on a threshold value. However, in future work, this information could be used for differentiating materials. There is a basic rule in image processing or in information theory in general, called data processing inequality. It takes a random variable Z in a Markov Chain of $X \to Y \to Z$, meaning Z only depends on Y and is independent from X. This can be expressed with the joint probability

$$p(x, y, z) = p(x)p(y \mid x)p(z \mid y) = p(y)p(x \mid y)p(z \mid y). \tag{2.4}$$

This means that there is no process (in this isolated scenario) of Y, which can improve its information (I) regarding X. With mutual information we can state the following:

$$I(X; Y) \geq I(X; Z) \tag{2.5}$$

This means that post-processing cannot increase information. [6]

This statement fits most image processing algorithms or if we look at the 3D point cloud applications as an array of the point cloud's segmentation, it also fits for example interpolation techniques. However, recent works of the last decade show, that in post-processing we can gather information from X with different methods and with this

knowledge acquired we are able to increase the output object's information from our existing general knowledge [7], [8]. If a model like this behaves accordingly, we have a great probable assumption on X's wider information, rather than just a hallucinations of sort. These methods are for example super resolution, with its newly found and wildly spread methodology or point cloud completion in case of 3D point cloud studies [9]. Point cloud completion methods main tasks are making denser point clouds from a coarse input or generating parts of the object's point cloud, which were obscured at the time of creating the point cloud. Most of these processes are built on encoder-decoder architecture having a global feature vector created from a sparse point cloud input from which an improved point cloud is generated. The pooling method most of these processes use, creates a great core for the object. In Section 2.3, I will introduce some architectures made for shape completion.

2.3. Architectures

In this section, I am going to introduce the general purpose of an encoder-decoder architecture, then I will discuss various methods in more detail.

2.3.1 Encoder-Decoder

In shape completion studies, neural networks are extremely useful due to their ability to capture a wide range of variations. More precisely, the encoder–decoder architecture is used in most cases.

The encoder – decoder architecture is usually used for sequence-to-sequence learning. Its two main parts are the encoder and the decoder. The encoder from the input creates a vector of sort which is the input for the decoder. The decoder then creates an output sequence which is similar to the input sequence, however, some parts are changed due to the contribution of weights in each layer of this architecture. In our case the output point cloud will consist of more points. The structure can recognize more complex relationships within the given point cloud [10].

2.3.2 Generative Adversarial Networks

Generative Adversarial Networks or GANs for short are generative models that are able to generate data that is hardly distinguishable from real-world data. In a broader description, they are unsupervised learning methods, having two main networks: the generator

and the discriminator. The generator is a neural network that tries to generate an output that is as close to the target as possible. This closeness can be understood as the likelihood that the generator's output will deceive the discriminator. The discriminator's role is to distinguish between images produced by the generator and those taken from the dataset. The generator is fully aware of how to fool the discriminator when the discriminator suggests a possibility of 0.5, meaning its opinion on the input is that in cannot be distinguished from the target. In simple terms, the generator tries to generate examples, while the discriminator tries to tell whether it is real or fake. The input of the generator can be random noise, sampled from a Gaussian distribution, or distorted images of the target. The main intuition behind this model is that these two networks train together, hence they are competing against each other. To connect the two networks, we can use a loss function designed to penalize the generator when the discriminators binary guess (real or fake) catches the fake input. This loss function is called adversarial loss, which is described by Equation 2.6, where $G(I_{\text{in}})$ is the output of the generator given the input I_{in} and $D(x)$ is the output of the discriminator given the input $G(I_{\text{in}})$ or I_{gt}, where I_{gt} is the ground truth. The discriminator wants to minimize this loss, while the generator attempts to increase it. Derived from this, we can update the gradient of the generator with the formula shown in Equation 2.7, where m is the number of data inputs we have sampled [11].

$$\ell_{\text{adv}} = \mathbb{E}_{I_{\text{gt}}} \left[\log \left(D \left(I_{\text{gt}} \right) \right) \right] + \mathbb{E}_{I_{\text{out}}} \left[\log \left(1 - D \left(G \left(I_{\text{in}} \right) \right) \right) \right]. \tag{2.6}$$

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log \left(1 - D \left(G \left(z^{(i)} \right) \right) \right). \tag{2.7}$$

2.4. Point Completion Network

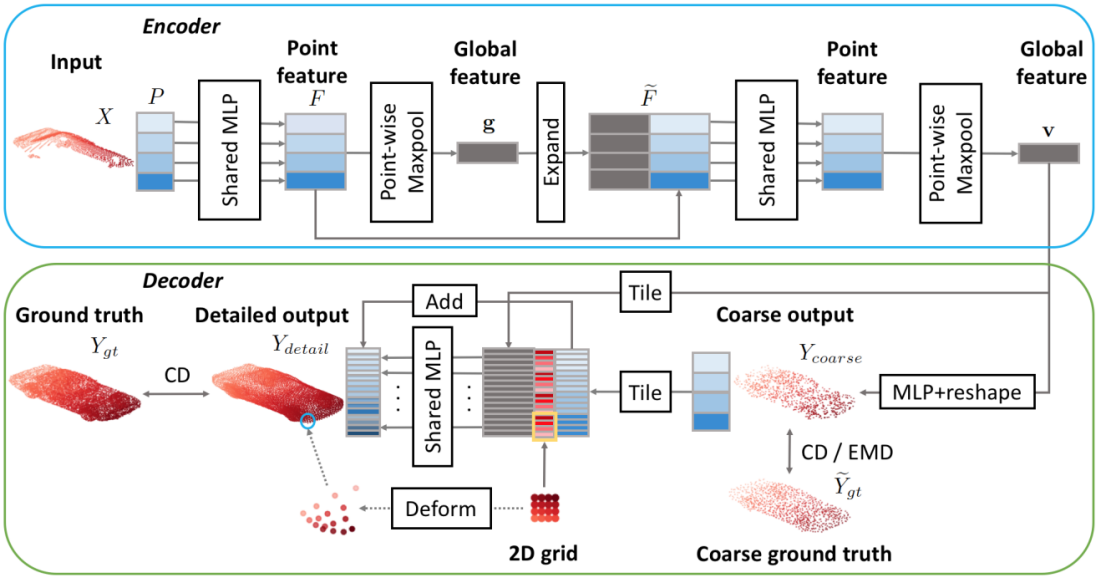

Figure 2.2: Encoder-Decoder architecture of PCN. Source: [7]

2.4.1 Architecture

Point Completion Network (PCN) is an early model for creating denser point clouds from the input in the form of an encoder – decoder. It is a tool for shape completion for point cloud inputs. It was developed because there was a high demand for completing unfinished partial or incomplete point clouds. Incompleteness in this case means barely or not recognizable structure of a point cloud. Before this method, the primary approach for shape completion involved voxelizing the 3D point clouds and then applying convolutional networks to the resulting voxel grid. However, these methods were limited by the resolution of the grid. Additionally, directly using the 3D point cloud input requires less memory and as it operates on raw point cloud data it keeps the initial geometric information. [7]

For 3D point clouds these networks face two complex task: creating a great feature extractor and loss function as it needs to be permutation invariant. Direct cases of the tasks mentioned above are for example robustness to noise and sparsity and generalization. The problem statement is as follows: We have a sparse 3D point cloud, X, gathered with a sensor such as LiDAR or sonar. Y is the point cloud representation of the real scene, a dense and complete structure. Our goal is to predict Y based on X. For this

purpose, this model suggests a neural network which gives a generic solution for multiple categories. [7]

As mentioned above, the structure of this model is based on the encoder – decoder architecture. From the input point cloud the network creates a feature vector $\boldsymbol{v}$ which has k dimensions ($k = 1024$). For creating $\boldsymbol{v}$, first some multilayer perceptron (MLP) converts X to $\boldsymbol{F}$, from which maxpooling (MP) extracts a global feature vector $\boldsymbol{g}$ containing the coarse, important geometric information. As it is shown in Figure 2.2, $\boldsymbol{g}$ is concatenated with $\boldsymbol{F}$. After the shown MLP and MP layers, the final $\boldsymbol{v}$ feature vector is created. Vector $\boldsymbol{v}$ is the output of the encoder and the input for the decoder. The decoder is a multistage point generation pipeline. This pipeline is unique as it is effective at predicting a sparse set of points that represents the global geometry and also it performs quite well at predicting local surfaces. This is performed in two different stages in the model, estimating the coarse (Y_{coarse}) and the detailed (Y_{detail}) shape [7].

For creating detailed output the network uses a local folding operation. It takes an input q_i from Y_{coarse}, and the global feature vector $\boldsymbol{v}$. Then in creates a grid of points centered around q_i. Afterwards, with the help of a shared MLP a 3D local patch is created from the 2D grid input [7].

2.4.2 Loss Function

As I have mentioned in the introduction of PCN, it is a challenging task to create an appropriate loss function given we are working with 3D points. There are two loss functions that are used in most of the cases. These functions measure the distance between the ground truth and the output. These are the Chamfer Distance (CD) and the Earth Mover's Distance (EMD) [7].

$$d_{CD}(S_1, S_2) = \frac{1}{|S_1|} \sum_{x \in S_1} \min_{y \in S_2} \|x - y\|_2^2 + \frac{1}{|S_2|} \sum_{y \in S_2} \min_{x \in S_1} \|x - y\|_2^2 \tag{2.8}$$

$$d_{EMD}(S_1, S_2) = \min_{\phi : S_1 \to S_2} \frac{1}{|S_1|} \sum_{x \in S_1} \|x - \phi(x)\|_2^2 \tag{2.9}$$

Where $\phi : S_1 \to S_2$ is a bijection.

Chamfer Distance shown in Equation 2.8, calculates the average closest point distances. S_1 and S_2 are the two cloud points. PCN uses the symmetric version of it CD. It is also

important to state, that this distance measure does not require S_1 and S_2 to be the same size. This means a really dense point cloud of a rough shape can have a small distance with an object. It is a part of the loss function that is great for determining rough differences between the approximated skeleton and the ground truth. For telling apart detailed differences between point clouds, Earth Mover's Distance can be used. The general form of EMD is shown in Equation 2.9. EMD finds a bijection which has the minimum distance between the two sets. In practice, it is too expensive to find the optimal bijection, that is why in most cases they use an iterative approach. Unlike CD, here the two sets are required to have the same size. The loss function is the weighted sum of these two methods. Distance for the coarse and detailed solutions is measured separately. [7]

In summary, CD is high when the global structure is different, while EMD is more evenly distributed, thus panelizes details as well. In the model's metric space EMD requires one-to-one correspondence while CD requires one-to-many, which is easier to deliver, that is why generally CD is smaller.

2.4.3 Dealing with Noise

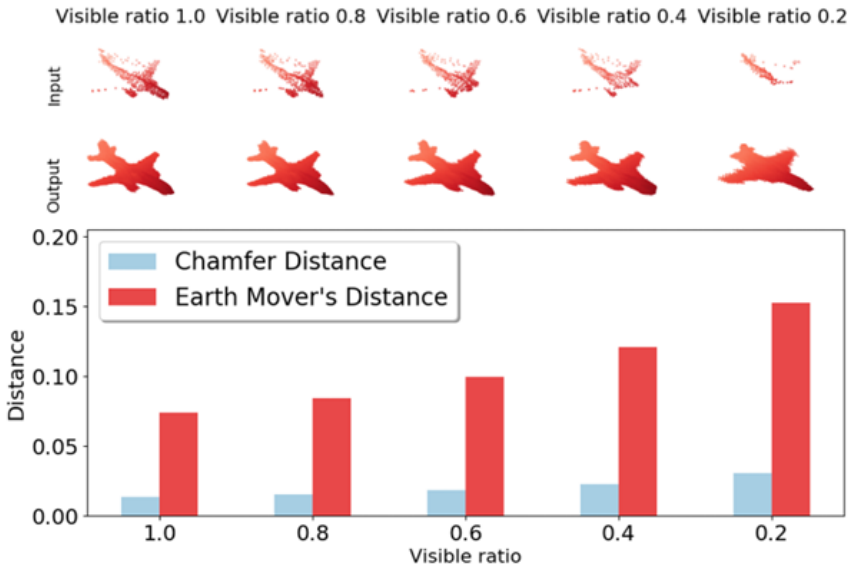

Figure 2.3: CD and EMD losses over inputs with decreasing visibility. Source: [7]

PCN promises robustness to noise, given its structure. The noise in their own measurement is Gaussian noise with a standard deviation of 0.01. The results they have measured are shown in Figure 2.3 with the corresponding input and output. In this

scenario PCN did well, however for various other objects they can get way worse predictions. For example, for inputs where they have multiple objects or the object has a thin structure, this architecture performs poorly. This can be seen in Appendix A in Figure A.1.

2.5. Variational Relational Point Completion Network

2.5.1 Architecture

VRCNet has a unique structure compared to other models used in the field of shape completion. It has two important stages: probabilistic modeling and relational enhancement. Both of these sub-networks are built on a encoder-decoder architecture, referenced as PMNet and RENet. This network was created to perform shape completion focused on detailed outputs. Most methods generate global shape skeletons, while this method tries to improve local details as well. VRCNet promises great generalizability and robustness for real–world data [12].

The network focuses on predicting the sparse skeleton, as fine details and local features could only be recovered if the core of the model is set. It is important in cases, when we only have a partial input. For this general purpose of predicting the skeleton of a shape, thin method proposes PMNet structure. This prediction process is different from other methods, as it uses probabilistic modeling to predict embedded global features and learned latent distributions. This structure is assisted with a dual-path parallel pipeline. So both reconstruction and completion are available. Reconstruction is done with a variatonal auto-encoder. The completion path works on a similar basis. The two paths share weights at multiple steps but they do not at the distribution inference layers. Both work as an autoencoder should, both paths learn to approximate the coarse structure from the global features and the latent distribution. From this we have the generated skeleton of the shape from a sparse or incomplete input. [12]

After this, comes the part which enhances the finer details in the structure. It is called Relational Enhancement Network. This part of the whole network focuses on founding local relations, features that could be enhanced in the final output of the model. This structure applies a Point Self-Attention kernel, defines a neighborhood and observes relational structures on these sets. This is the core idea behind RENet. Its structure is based on a hierarchical encoder-decoder architecture and uses Edge-aware Feature

expansion module which I will introduce in Subsection 2.6 [12].

As there are various parts of the system, different loss functions are needed. It has three main parts. For reconstruction path the Kullback–Leibler divergence is used. For reconstruction loss they chose Chamfer Distance which I have described in Equation 2.8. Completion path and Relational paths share the same idea here, the whole loss funciton is a weighted sum of these methods [12].

Their experience was evaluated with CD and F-score as the CD metric in itself could be misleading. The results of CD losses on different categories with multiple models is attached in Appendix A in Figure A.4. Also, I have attached Figure 2.4 which shows how different methods performed. As their results show, these methods based on their claims, outperform each existing methods on various categories and even on real-world data [12].

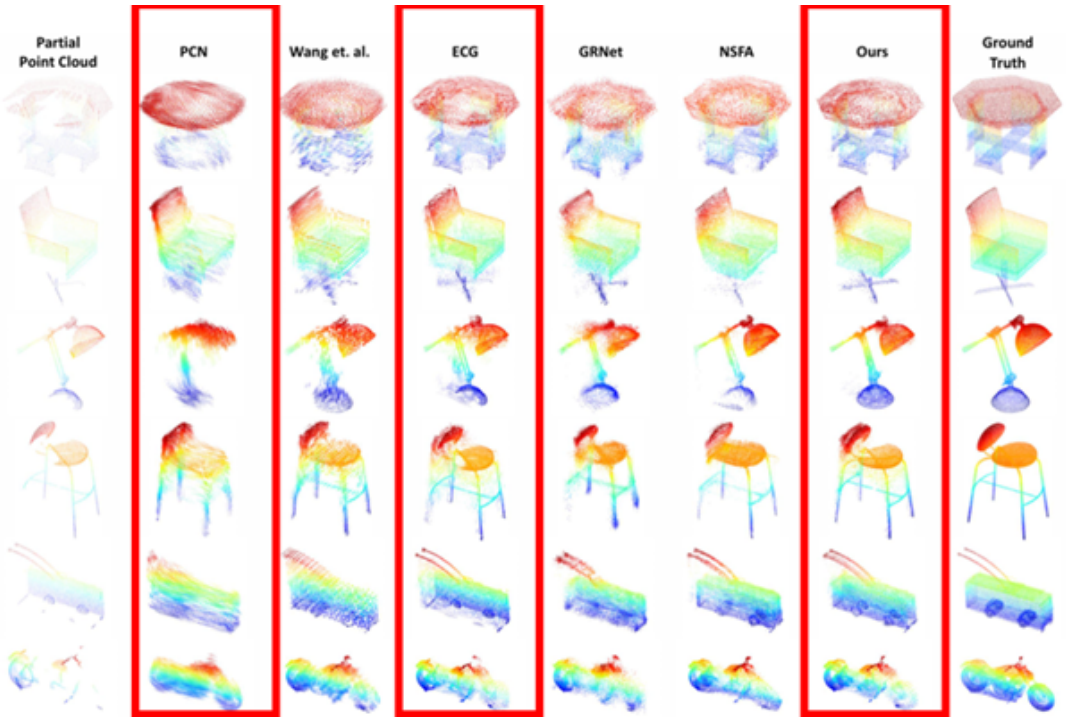

Figure 2.4: This figure compares results of different Shape Completion Methods. The results of PCN, ECG and VRCNet are highlighted. Source: [12]

2.6. Edge-aware Point Cloud Completion with Graph Convolution

This method was created for the same shape completion task as the others presented previously, but promises edge-aware point completion with graph convolution. This architecture also consists of two stages: skeleton generation for the input shape and resolution enhancement with correctly added points. In this way, the details or information of the output can be increased. [13]

As described in Section 2.4, this method also generates global features and a coarse skeleton of the shape from the input, just as the PCN. Then, local features are created with a deep hierarchical encoder with graph convolution. It also implements an Edge-aware Feature Expansion that preserves local details at the upsampling steps rather than just using folding mechanism which was the case with PCN [7] [13].

The choice of using graph convolution was convenient, as this methodology can be used with unordered points, having a flexible structure to learn neighborhoods, giving an opportunity to have a better understanding of the fine structure. The idea of skeleton generation and detail refinement structures are similar to those shown previously. However, the graph structure applies two strategies to define neighborhoods: ball query for distance-based ordering and k-nearest-neighbors for connectivity optimization. These are just the main elements of the model; the whole structure has a higher complexity. The results of this method on different categories are shown in Figure A.4 and in Figure 2.4 [13].

2.7. Multi-View Based Point Cloud Completion Network

This model, called Multi-View Based Point Cloud Completion Network or MVPCC for short, was developed for high-resolution MLS input. The MVPCC network was designed to generate dense and detailed 3D object models. The input to this model can be a sparse or incomplete representation of an object. By passing this input through the model, we aim to perform completion. Unlike the previously mentioned models, this one operates in the image domain. It generates projections from different viewpoints of the object and then the model works with these projected images [8].

The schematic illustrations of the components of this model can be seen in Figure 2.5,

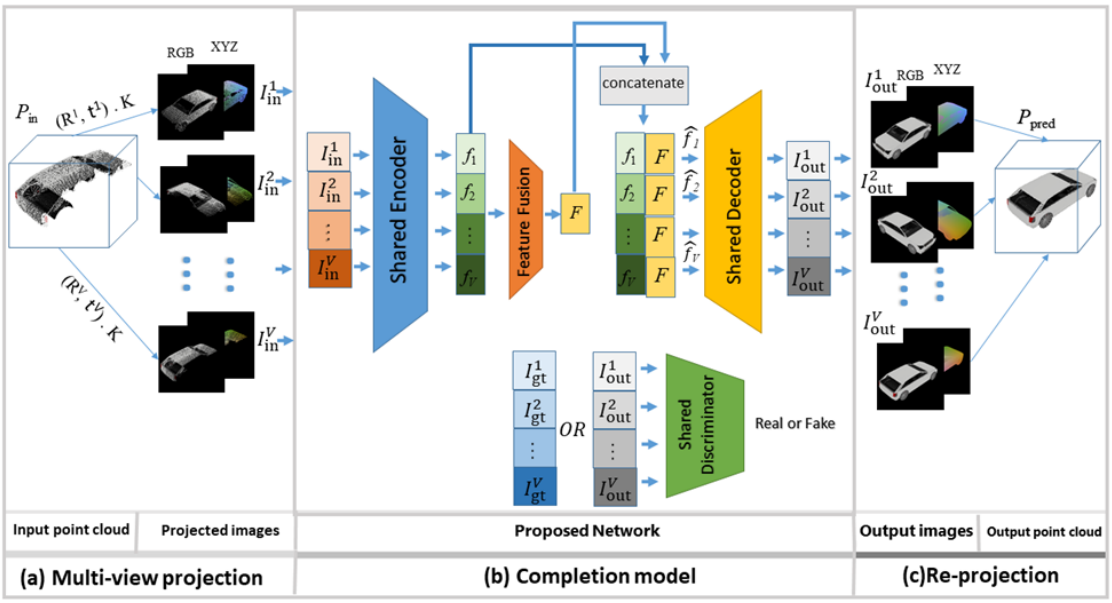

Figure 2.5: Architecture of MVPCC model. Source: [8]

where the number v on (a) Multi-view projection corresponds to the number of viewpoints that were used to create projections. The number of viewpoints is adjustable; however, the research paper suggests using four viewpoints, as this yielded the best results in their experiments [8]. The multi-channel image input enables the CNN to fill in missing structural information. It can also be applied to fill in colors or textures as shown in Figure 2.5, although this functionality is not utilized in my implementation [8].

It has numerous advantages over the other models. For example, we do not need to know the exact number of points as an input or output. We work on projections, so they can be altered and reprojected accordingly. This model is capable of addressing the issue of inhomogeneous point density and placement. The research paper also states that it outperforms state-of-the-art models generating local fine geometries [8]. This could be the result of the loss, we are trying to minimize, as all of the previously mentioned methods use Chamfer Distance (CD) as the training loss, minimizing the mean of local point-to-point distances between the predicted and ground truth point clouds. This approach does not necessarily ensure an accurate representation of shape similarity [8].

2.7.1 Architecture

The architecture of the model described in the research paper is shown in Figure 2.5. The number of projections is determined by how many virtual cameras are we working with.

In this model each image has six-channels for RGB and XYZ coordinates. The model is specifically designed to perform inpainting and completion, with the expected outcomes are shown in its design. The model is based on a GAN structure, which was previously introduced in Subsection 2.3.2. In this version the input is a sparse or incomplete image. The expected outcome is the completed version of each projection.

The Generator's structure consists of three main phases: encoding, feature fusion and decoding. The shared encoder downsamples the input images separately. Then to emphasis each image features, the model uses eight residual blocks to create a feature cube of each input image in the size of $64 \times 64 \times 256$. Then, in the feature fusion phase the extracted features are fused into a global cube having the same size as one of the local cubes. It is designed in a way, that we expect it to have a transmission of shared characteristics. The last phase which is the decoding phase, starts with creating an $f_{\hat{v}}$ cube for which we have $f_{\hat{v}} = [f_v, F]$, where f_v is the corresponding local feature cube and F is the global feature cube. Then, for each $f_{\hat{v}}$ we separately upsample them to the size of the original inputs using dilated convolutions. Then, as an output we will have v six-channel outputs, where v is the number of images that were used as inputs. In this architecture, we have expressed global characteristics and connections between each image in the fusion and concatenation steps. After this, we can use the outputs of the generator as the inputs for the discriminator along with the target inputs. This model's discriminator is based on PatchGAN structure [14]. From this structure we can ensure, that by training both networks simultaneously, we will have a reasonable loss showing how these network compete and become better.

2.7.2 Loss function

To ensure that the output is as close to the ground truth as possible, they have used a combined loss. The loss is a combination of six losses: adversarial loss, smooth L1 loss, perceptual loss, style loss, binary cross entropy loss and Total Variation loss.

- **Adversarial loss:** The same loss as described in Subsection 2.3.2 in Equation 2.6. This loss is computed for each viewpoint.

- **Smooth L1 loss:** This is simply *Mean Absolute Error*, however it is sensitive to outliers, where it penalizes bigger distances between output and ground truth with a bigger penalty.

- **Perceptual loss:** *Perceptual loss* or also called *VGG loss* is a loss term dependent on the great feature recognition ability of *VGG Network*. *VGG loss* is usually used to encode the input in a feature extractor; then we can define a distance between the extracted features of the output and the target. For this, *Mean Squarred Error* is a great choice. In this model, *perceptual loss* is strictly used for RGB channels [15].

- **Style loss:** *Style loss* is also used only for the RGB channels of the output and the target. It determines stylistic distance between the two images [16].

- **Binary cross entropy:** Our projection image consists of two main parts: the background and the object clutter. As the object part is more important, we need to adjust our loss to penalize those predictions on a higher degree that are wrong for the object. This can be done by masking non-zero parts of the ground truth image and the output image, then measuring the difference between them. A great measure for this can be BCE, which penalizes greater distance with a bigger penalty.

- **Total Variation loss:** This loss is performed only on the output image not compared to the target. Total Variation loss promotes smooth images in a way, that it looks in all the points neighborhoods and penalizes difference between them. This regularization is useful, but we have to use it cautiously.

In the combined loss of this model, all the above losses are presented, but they are also weighted. These weights for the losses are hyperparameters of the model. The combined loss can be expressed with Equation 2.10, where the regularization parameters for the losses are: $\lambda_1 = 50;\ \lambda_2 = 0.1;\ \lambda_3 = 250;\ \lambda_4 = 0.1;\ \lambda_5 = 0.1;\ \lambda_6 = 0.1/S$, where S refers to the image size. In their experiment $S = 256 \times 256$ [8].

$$\ell_G = \lambda_1 \ell_{L1} + \lambda_2 \ell_{\text{adv}} + \lambda_3 \ell_{\text{perc}} + \lambda_4 \ell_{\text{sty}} + \lambda_5 \ell_{\text{Lcro}} + \lambda_6 \ell_{\text{TV}} \tag{2.10}$$

2.7.3 Dataset

Synthetic Data

The research paper also describes the datasets it had used or created as part of this project. For training the model they have used synthetic data. For example they have used ShapeNet, which dataset has a lot of meshes [17]. From these meshes we are able to sample point to create a point cloud. To resemble real MLS data, we have to sample in a fashion which creates a point cloud similar to real-world data. This has to be done because synthetic data also have structures inside. For this, creating the ground truth

3D point cloud, they have used an approach called SK-PCN [18]. This method was also applied when creating the incomplete or partial inputs, so only visible point were on the projections. The training data consisted of four object categories from ShapeNet: car, bus, motorcycle and train. In the real-world application of MLS systems the creation of point clouds happens without capturing the bottom of the objects as that is not visible. This consideration was used when placing the virtual cameras around the objects, to represent the same manner. The synthetic dataset used 4918 distinct models. From these more partial sample were created, which resulted with a train dataset with 91600 objects [17] [8].

MLS Data

The research paper also describes a dataset, consisting of real-world MLS data. As this is real-world test set, it is mostly partial. After the measurement, the raw MLS data was pruned and the vehicles had been extracted and separated, creating the dataset. The segmentation task was performed with an assigned 3D annotator tool to ensure accuracy [19]. The total dataset consists of 424 object samples.

Chapter 3

Methods

In this chapter, I am going to summarize the methods and datasets I have used in my experiments. The methods involve mathematical descriptions of the data transformation procedures that I have performed and their implementations, the models I have used and the description of their implementation and the architectures of my model and my code.

3.1. File Formats

Throughout my project, I have used different databases. Because of this, I have come across different file formats, representing a point cloud or a mesh. I will briefly introduce each format.

3.1.1 Point Cloud File Formats

Polygon File Format

Polygon File Format is a file format for point cloud data with the extension *.ply*. It is for point clouds, objects are simply sums of their vertices. Its main part consists of the header and the points. Deeper connections can also be expressed by this format; faces, colors and normal directions. I have only used the stored vertices [20].

Point Cloud Data File Format

Point Cloud Data File Format is a file format for point cloud data with the extension *.pcd*. This file format mainly consists of its header and the so called fields, describing *xyz* coordinates and their colors or surface normal, if it is also attached. Its advantages over other formats are fast saving and loading. This is why in many cases this type of data storage could be seen [21].

3.1.2 Mesh File Formats

OBJ File Format

This is a file format with the extension *.obj* is for meshes, rather used in 3D printing. It encodes surface geometry of the given model while it is also able to store textures and colors. The points these formats store are rather a rough structure of the object rather than the object surface itself. So if we want to recover the point cloud corresponding to the object, we cannot use these points. We have to sample points from the meshes to recover the structure [22].

STL File Format

This format also could be found in 3D printing with the extension *.stl*. An *stl* file is a mesh object, consisting of a series of 3D vertices, and lists of vertices that describe individual polygons, the faces of the object [23].

3.2. Sampling Methods

In this section, I am going to briefly describe each method I have used for sampling points from a point cloud or from a mesh.

3.2.1 Random Sampling

This technique is simple and in most cases effective. By taking a point cloud having x points, the algorithm randomly chooses y point for which $x > y$ is true. The points are random, so the density of the point cloud is not the same throughout the structure, which is great for resembling a real-world data recording as most of the scanners do not create point clouds having the same density everywhere. However, for training, it might not be the best solution as having randomly chosen points does not guarantee us having a full cover of the original structure. Moreover, with this technique, we cannot sample directly from the mesh, we need a given set of points to sample from.

3.2.2 Simplifying Quadratic Decimation

This is a technique used to reduce the number of polygons or points in the dataset, used in the original implementation of PCN [7]. This is a method, where you do not apply the reduction process directly to the number of points you desire. Instead, you have to give a face count which is the number of faces the mesh will have. The name Quadratic

suggests that quadratic error metric is used to decide which faces to keep or remove. This method directly simplifies the mesh, after which you can sample the simplified mesh or the points corresponding to the newly modified faces.

3.2.3 Farthest Point Sampling

This method was used in the implementation of MVPCC [8] to sample points. The technique takes the mesh of the object as an input, and samples a given number of points from it. It first chooses a point that is on the mesh and then samples the farthest point to it in the next iteration from the remaining points. This process is repeated until we have sampled the given number of points.

3.2.4 Uniform Downsampling

This method also samples points from the mesh directly. It samples uniformly, so it keeps the original shape of the object. Also, it is uniform to the triangle sizes of the mesh, it samples more points from larger triangles and less from the smaller triangles. This technique helps to keep even finer details when sampling a small number of points and also keeps the shape robustly at important parts. This technique was also used in PCN [7].

3.3. Datasets

3.3.1 ModelNet40

For proof of concept, most of the pre-trained models were trained with this dataset, due to the diversity it shows classwise in a simple form. The objects themselves are simple, as each object is represented with 2048 points. As the name suggests, it includes 40 categories, ranging from household objects to vehicles and planes. It is great for showcasing the possibilities of a network [24].

3.3.2 ShapeNet

ShapeNet is a widely used large-scale dataset, having 55 common object categories and 51300 unique 3D models. The high diversity of objects is reflected in the taxonomy. For example, the common object category *watercraft* is divided into 7 subcategories and includes nearly 2000 objects in total. The diversity of this common category is shown in

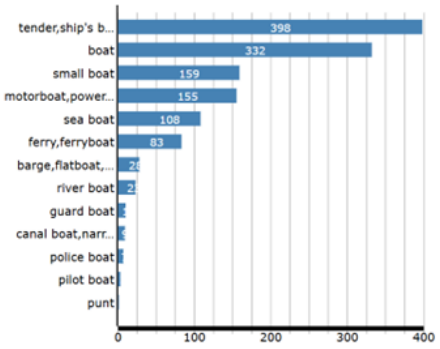

Figure 3.1: This graph shows the diversity of Shapenet's taxonomy of boats as a subgroup of watercraft. Source: [17]

the following Figure 3.1. The main categories of watercraft are attached in Appendix A in Figure A.2 [17].

All of these objects were stored in *.obj* format described in Section 3.1. The meshes were normalized to fit inside the bounding box of $[-1, 1]^3$ in 3D space. Also, another important feature of this dataset is that the shapes of the objects have an orientation, defined by the up- and front vectors, which vectors define the upper side and the front side of the mesh. Moreover, these vectors are aligned, so the objects are oriented in the same direction [17].

3.3.3 My Dataset

For this dataset I have gathered 50 *stl* files of ships, boats and ship parts. The file format is described in Section 3.1. For generating the 3D point clouds, I have chosen the middle points of each triangle, because the original vertices were ordered too regularly. From this, I can control the number of points each point cloud contains. The size depends on how many points represent the shape or the object. These 3D point clouds are the ground truths. From this I have created both partial and sparse clouds as inputs. In case of sonar data a partial point cloud would mean that the object is scanned from a single direction, or it is laying on the riverbed. For creating partial point clouds I have written a function `filtered_points = cut(points, plane_normal, plane_point, cut_direction)`, where

- **points** is the 3D point cloud
- **plane_normal** is the normal vector (n_x, n_y, n_z) of the plane which cuts the point cloud

- **plane_point** is a point (p_1, p_2, p_3) on the plane
- **cut_direction** determines which side of the plane to keep. Valid values are l for left side and r for right side.

In Figure 3.2 I have attached 2 plots showcasing what the partial point cloud looks like next to the original. I have generated this point cloud with a plane_normal $(1, 1, 1)$ and with a plane_point being $(0, 0, 0)$.

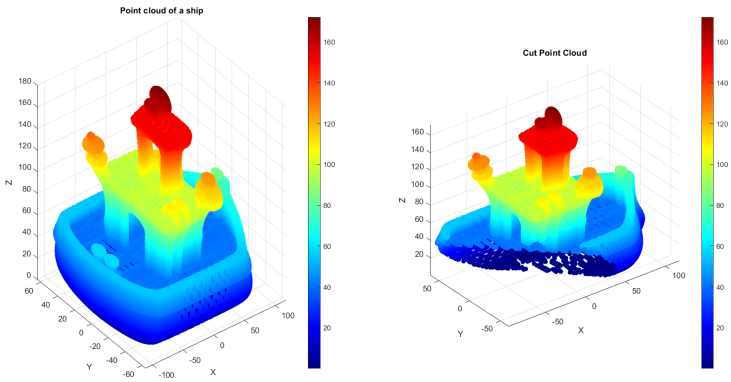

Figure 3.2: On the **left** the original point cloud of the ship can be seen, while on the **right** the newly created partial point cloud is shown.

For creating sparse point clouds, I have written a function that keeps a given percentage of the input point cloud. Also, for some of the points that I have kept, I gave an offset value of a given distance to generate some noise. The number of points with offset can be given. As an example, I have attached Figure A.3 in Appendix A, which shows a newly generated sparse point cloud with half of the points having a small offset.

3.3.4 Real World Data

To measure the performance of the models, I have used real-world data created with an ultrasound device. Sonar multibeam systems generate point clouds by typically retaining only the strongest reflection for each angle, representing the closest surface. The positions of these reflections are then calculated in 3D space using GPS/IMU data [25]. Alternatively, if all received data is stored, lower-intensity reflections can be filtered out later using thresholding. The resulting 3D point cloud, shown in Figure 3.3, was manually refined by removing the background and other artifacts.

This real-world example was recorded by our research group in a water tank, from a 3D-printed ship model. Images were taken using a vetenary ultrasound machine, Sonoscape

with a linear L745 probe. The position of the probe was tracked using a camera and visual markers, aiding the reconstruction of the 3D model from 2D B-mode ultrasonic images. Lower intensity scatterers were filtered from the images, resulting in the point cloud depicted in Figure 3.3. This example point cloud will be used to evaluate whether the best models I train can successfully perform shape completion on this data.

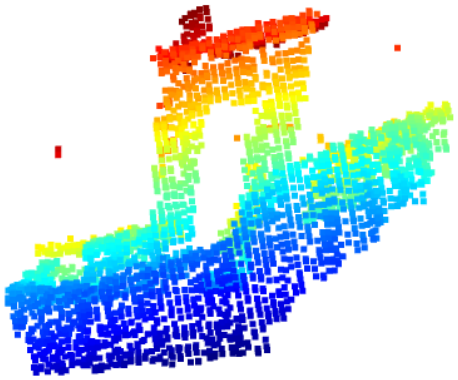

Figure 3.3: This is a pruned version of a real measurement created with a ultrasound than transformed to point cloud.

3.4. Environment

Computer For computations, data preprocessing and training I have used a laboratory PC from PPCU. It had a 12^{th} Gen Intel(R) Core(TM) i9-12900K CPU, a NVIDIA GeForce RTX 3090 GPU. I have used this PC with CUDA 11.8 in Python 3.11.

CUDA All the models I have implemented were Neural Networks. There are different structural elements, that I have used in each, however, there is a common ground, which is that each is computationally heavy. Neural Networks require a lot of operations computed on matrices. For these calculations, I have used GPUs. For this, I need to communicate with them. For this low level communication, I have used CUDA(Compute Unified Device Architecture), which translates high level code to low level operations for the computer. This can be utilized in NVIDIA GPUs. It is designed to do general purpose computations in parallel, which are great for matrix operations.

In my coding language, Python, I need to find an interface, which supports GPU acceleration. There are two major interfaces for it. TensorFlow [26] and PyTorch [27]. I

have used Pytorch, which is shown along other important and useful libraries in Table 3.1.

Python As on the laboratory PC I manage multiple implementations of different models and multiple projects at the same time, I needed different environments for each, so the Python versions and the libraries will not get conflicted. For this, I have used a local environment, which is a built-in way in VSCode. It creates a *.venv* folder in my project folder, where I can manage my libraries with a package manager. For this purpose, I have used PIP [28]. This is a package manager, version 24.0 for Python. I have attached a brief description of the most important libraries regarding my implementations in Table 3.1.

Library	**Description**
Torch	Framework for neural networks with GPU acceleration and tools for datasets and data loading. Provides dataloaders for neat organization of data for training [27].
Numpy	Used for data preprocessing, matrix operations, and eliminating loops for faster computations [29].
Matplotlib	Library for creating figures and plots for training results and depth maps [30].
Open3D	Toolset for visualizing and modifying 3D point clouds with features like point cloud objects, virtual cameras, and uniform point sampling [31].
Fpsample	Optimized FPS (Farthest Point Sampling) implementations. Types: Vanilla FPS, KDTree-based, QuickFPS [32].

Table 3.1: Summary of the Python Libraries I have used.

3.5. PCN

For PCN I have implemented the PCN model via learning3d package [33]. With the help of this package, I have also implemented its examples for training and testing. I have later modified these examples to fit my implementation. These codes are accessible in Appendix B. I have trained two different models to test the capabilities and find out how it performs the completion task. First, I have trained the model with ModelNet40 dataset. Then, I have trained it on watercraft derived from ShpapeNet.

3.5.1 Using ModelNet40 for Training

1. **Testing sparse input from ModelNet40.** I have tested different sparsity levels, then evaluated with the metric CD loss to compare how close my estimate was to the ground truth. The test set was based on categories that the network had learned on, but the objects themselves were different.

2. **Testing incomplete input from ModelNet40.** For this exercise I have utilized the cut function described in Section 3.3. After creating the incomplete structures, I have tested PCN's pre-trained model at this completion task. I have also evaluated on CD loss.

3. **Testing on my dataset.** For this I chose examples from my dataset covering all the types included in it by structural mean. I have also evaluated using CD loss.

In addition to standard evaluations, I will also qualitatively assess the accuracy of the approximated structure from a viewer's perspective. It is an important step, as a low CD loss in itself does not mean a good fit between the ground truth and the generated point cloud.

3.5.2 Using ShapeNet for Training

After training the network with ModelNet40, I have also trained it with a larger training set with objects having more variety to see how the model handles it. For this, I have used the watercraft from ShapeNet dataset described in Section 3.3. As it is shown in Figure 3.1, the taxonomy of watercraft is large, giving a great variance in shapes and sizes given all of its categories. From the original dataset I have acquired 1910 watercraft. For creating point clouds from the meshes, I had to sample them. For sampling I have tried three methods: Random Sampling, Quadratic Decimation and FPS. These methods are described in Section 3.2. After choosing the most useful way to sample in this application, I have prepared the input data and the target for training. For target I have sampled 16384 points from each mesh. For the input, I have generated sparse and incomplete inputs. The incomplete point clouds had been cut with the function described in Subsection 3.3.3. The sparse point cloud contained 12000 points. This is not much sparser than the original point cloud; however, in this experiment my main goal was to see how does the model act. After preprocessing the point clouds, I have generated a PyTorch dataset. After this, I have created the dataloaders with a batch size of 16. Then, I have trained the learning3d PCN model through 150 epochs with

Adam optimizer with a learning rate of 0.001 and pocketed the best result based on the smallest Chamfer Distance measured on the validation set. Also, for the model, I have set the embedded feature vector to a size 512. This embedded feature vector is the vector storing the high-level features detected in the input data, referenced as the global vector in Figure 2.2 in Chapter 2 Section 2.4.

3.6. Projections and Reprojections

For creating projections I have implemented multiple processes. By the mathematical formulation of these transformations, we can divide them into two groups. The first group is for creating depth maps, which is based on pinhole camera projections. The second group is based on projecting to planes, where the plane's origins are based on the position of the virtual camera. In my project code I have referenced to pinhole camera projection as projection1.py and to the orthographic projection as projection2.py attached in Appendix B. In the projected 2D images the pixel values can be set as the distance of its 3D counterpart from the camera, resulting in a depth map. The second solution is to store in these projections the original 3D coordinates of the projected points as 3 different channels.

3.6.1 Pinhole Camera Projection

Projection

For my implementation of MVPCC I had to create depth map projections from the original point cloud, from 4 viewpoints. The pinhole camera projection assumes that light enters the camera through a single small pinhole, from where light travels to the 2D sensor array. This transformation from 3D to 2D can be managed by matrix multiplications. For this, we need to define intrinsic and extrinsic matrices. The intrinsic matrix maps the 3D camera points to the sensor array, or image plane. The matrix itself is an upper triangle matrix as shown in Equation 3.1, where f_x and f_y are the focal lengths (distance of the pinhole from the sensor plane, ideally f_x and f_y), s is skew(0 in my application), while c_x and c_y define the optical center in pixel coordinates ,where the perpendicular ray from the pinhole hits the sensor plane (which in my case were $c_x = \frac{width}{2}$ and $c_y = \frac{height}{2}$,respectively). Conversion of 3D camera-centered point to 2D homogeneous point can be

expressed by the Equation 3.2, where p_c is the original coordinate [34].

$$\mathbf{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

$$\tilde{x}_s = K p_c \tag{3.2}$$

Another matrix that is needed for the transformation is the extrinsic matrix, which basically describes at what angle and location the camera is positioned in space. It will be used to transform the coordinates of the point cloud to the camera coordinates. It is composed of two parts, the rotation matrix and the translation vector. This can be expressed by Equation 3.3.

$$[R \mid t] = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & | & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & | & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & | & t_3 \end{bmatrix} \tag{3.3}$$

The matrix itself is changed to be a 4×4 matrix, when we are working with homogeneous coordinates. This change is expressed by concatenating an extra row to the matrix in the form of [0 0 0 | 1]. It is a useful tool, from knowing the virtual camera's pose, its position and direction, we can define the extrinsic matrix [35].

For each point cloud, I have created four projections. For each of them I have saved the extrinsic and intrinsic matrices as well, so at the reprojection phase, I can easily reproject the object, knowing the settings and poses of each virtual camera.

Reprojection

After projecting a 3D point cloud from 4 viewpoints, I have also stored the extrinsic and intrinsic matrices as mentioned in the previous Subsubsection 3.6.1. With the help of these matrices, I am able to create a reprojection, similar to the original 3D point cloud. The given 4 viewpoints are chosen in a way that they are supposed to cover as much depth of the object from their angle as possible. However, 4 viewpoints are not enough to totally restore the original object, so I will encounter a loss of information. The viewpoints are chosen in a way that they are able to capture smaller details and the overall structure of a given object, but they might fail to reproject really fine details or some parts of the ships. In my case, I have chosen the viewpoints to be a bit higher than

the highest point of the ships, so it can capture depth from an upper angle. This might result in the loss of the ship's hull, but I had to compromise for computational efficiency.

3.6.2 Orthographic Projection

Projection

This projection assumes parallel rays are being casted onto the projection plane, resulting in an affine transformation. For this, I needed to define the plane from which the rays start. For defining the four viewpoints, which are the same as defining a point of the planes, I have used a fixed structure. As I was working on ShapeNet, I had many benefits from it: the ShapeNet objects were normalized, so all of the points representing the object were in the boundaries of $[-1, 1]^3$ centered in the origin. Given this, I was able to generalize the plane parameters for the whole dataset. The equation of a plane is

$$\mathbf{n} \cdot (P_0 - P_a) = 0 \tag{3.4}$$

,

where $\mathbf{n}$ is the normal vector, P_0 is an arbitrary point on the plane, and P_a is another point in the space laying on the plane. In my case, P_0 is one of the virtual camera's position. They are placed at $(1, -1, 1)$, $(-1, -1, 1)$, $(1, -1, -1)$, $(-1, -1, -1)$. The normal vector $\mathbf{n}$ of the plane describes the direction perpendicular to the plane, along which the rays are casted. For the projection I will have another variable called d, corresponding to the distance of the plane from the origin $(0, 0, 0)$, calculated as

$$d = \mathbf{n} \cdot P_0. \tag{3.5}$$

Then I can compute the distance of each point Q in the point cloud from the plane as

$$t = \frac{d - (\mathbf{n} \cdot Q)}{\|\mathbf{n}\|^2} \tag{3.6}$$

where t is the distance of the points from the plane. Now I can calculate the 3D coordinates of the points on the plane with

$$Q_{\text{3Dproj}} = Q + t \cdot \mathbf{n} \tag{3.7}$$

After this, to capture the X, Y and Z coordinates or depths of the original vertices I had projected, I need to map the Q_{3Dproj} points to a 2D grid. For this I rotated Q_{3Dproj}

points from the projection plane to the xy plane, and sampled it on a grid. This is done by a simple coordinate transformation

$$Q_{\text{2Dproj}} = \mathbf{R} \cdot Q_{\text{3Dproj}}, \tag{3.8}$$

where $\mathbf{R}$ is an arbitrary orthonormal basis including $\mathbf{n}$.

After this conversion, I can start constructing the grid. For construction, I needed to define the resolution. For default values I have chosen $\Delta x = \Delta y = 0.001$. The bounding box was defined by x_{min}, x_{max}, y_{min}, y_{max}. These are the minimum and maximum values for both x and y. To ensure that later the model will get the same sized input, I had to pad these projections, as not all projections have the same minimum and maximum values. After creating the grid, the next phase of the process was to do the mapping of the coordinates. For mapping, I have decided upon to try out two strategies. I will highlight them in the next paragraph.

First mapping strategy As projection comes with a loss of information, I needed to make sure whenever there are more than one point projected to one point in the 2D projection, I greedily grab the closest to the plane. Also, I have neglected points that are distant, according to their neighborhood. The distance and the neighborhood were two fixed parameters, 0.01 and 1 accordingly. As a result, I will get 4 projections from different viewpoints, each having 3 channels containing the original point's coordinates.

The second mapping strategy was performed via mapping with ray tracking. I have implemented another form of selecting points that are visible from the plane's perspective. For this method, I have used the original mesh. This method intuitively, creates rays from the plane's perspective at those points, that had been projected to it via orthographic projection. Then it traces them, and creates hit points where the rays hit the object. From these hit points I could gather those, that are visible. As all hit points are on the mesh, as they have been sampled from it, I have to sample those, that only hit the mesh once. Simplifying it, I can choose those points, that are at the same distance from the plane as the hit distance or at least their absolute difference is smaller than epsilon, where epsilon is a fixed number. In my calculations, epsilon was set to 0.0001.

Also, I have created another form of implementation for ray casting. In this case, I have created a projection of the mesh to be the target, instead of sampling more points compared to the input. After this, I have sampled points from the mesh and chose those that

are visible from the mesh perspective to be the input projections. This projection technique can be found in the implementation of my network in *raycasting_projections.py* attached in Appendix B.

Reprojection

The reprojection of this method is simpler, than the reprojection of the previous method introduced in Subsection 3.6.1. In this case, I have stored the original x, y, z coordinates. So by adding the non-zero values to a list x, y, z accordingly and summing all 4 viewpoints, theoretically I can reproject my original point cloud with minimal losses and no artifacts.

3.7. Original MVPCC Implementation

In this section, I will briefly summarize the implementation process of MVPCC. After that, I am going to explain its network in more depth than introduced in Chapter 2 Section 2.7. Lastly, I am going to write about my experimental setup for the original model.

3.7.1 Implementation

For implementing the original model, I had to follow the implementation requirements written down in the GitHub Repository of the code [36]. First, the code requires a Linux operating system, as the code has parts, which are parallelized. This process can be done in the operating system they have used, called Ubuntu 20.04. To create an environment for this model, I had to use WSL (Windows Subsystem for Linux). This allowed me to create Ubuntu environment in my Windows operating system. After this, I have installed and setup PIP, Python 3.8 and and venv in Ubuntu. For gathering all the Python packages, I have used in the project, I have created a virtual environment with venv in WSL. After the environment setup, I had cloned the repository from GitHub [36]. All of these processes were accommodated in VSCode, code editor.

3.7.2 The Network

The Neural Networks, that I have described in Chapter 2 in Section 2.7, were implemented with PyTorch, built to assist GPU utilization. There are four neural networks, that together form the completion network. BaseNetwork, InpaintingGenerator, Discriminator and ResNetBlock. All models apply layers, activation functions, or transformations from

the torch package `torch.nn` which gathers basic building blocks for the networks [27]. I have used the following building blocks:

- **Convolutional Layers:** I have used convolutional layers to perform convolutions on the input matrices with the help of `Conv2D(in_channels,out_channels, kernel_size, stride, padding)` function, where the $in\_channels$ and the $out\_channels$ are the number of input and output channels, kernel size is the size of the convolutional filter, stride gives the number of pixels that we shift the kernel with and padding gives whether we want to pad the border or not [27].

- **ReLU:** ReLU or rectified linear unit is a non-linear activation function commonly used in neural networks. The function itself is $ReLU(x) = max(0, x)$, which non-linearity is needed for neurons activation [27].

- **Leaky ReLU:** Leaky ReLU is also an activation function with the non-linear function: $LeakyReLU(x) = max(0, x) + negative\_slope \times min(0, x)$. This small slope for negative input values helps to address the dead neuron problem of ReLU activations, which in some cases can be harmful for the model [27].

- **Sequential Blocks:** It is a container module, providing the functionality to stack layers together in a given order. This is useful, because when I apply forward function, I only need to call the sequential block, not all the layers one by one.

- **Reflection Padding:** This is a padding type with the command `torch.nn.ReflectionPad2d(padding)`, where `padding` is the size of the padding. The padding in this case is performed with the boundaries of the input being projected [27].

- **Batch Normalization:** This is a normalization layer widely used in all neural network applications. It is designed to normalize the inputs for layers to a mean of 0 and a standard deviation close to 1 [37].

- **Instance Normalization:** This is a normalization layer, mostly used in GANs. This process tries to remove instance-specific contrast information. This helps generalization [38].

As I have mentioned earlier in this Section, that the completion network of MVPCC consist of four networks. I am going to introduce them and present their main tasks.

- **BaseNetwork:** This is a class inheriting $nn.Module$, which is a Torch base class. This inheritance lets us define organized layers, used in neural networks and also gives us a forward propagation function, which simplifies the forward propagation steps. In addition, it also helps with saving and loading the model's weights which is a must, for testing and and making predictions. The BaseNetwork equipped with these functionalities is used as a base model for the other networks used in this project extended with a function for initializing weights. For weight initialization, there are four options we can choose from, normal, Xavier [39], Kaiming [40] and orthogonal initialization. Weight initialization is an important procedure before training, as badly chosen weights can cause vanishing or exploding gradients. For my implementation purposes the normal, Xavier and Kaiming initializations were important.

  - *Normal weight initialization:* In this case the values of the weights are initially random, sampled from a normal (Gaussian) distribution. In BaseNetwork, normal initialization is set up with a normal distribution with 0 mean and standard deviation of 0.02. This initialization procedure helps to break symmetry between neurons, thus helping to improve the training dynamics.

  - *Xavier weight initialization:* Xavier or also called Glorot weight initialization is used to perform the same task as normal initialization does. However, it has more steps to it. First, it also initializes its weight by sampling from a distribution (normal or uniform), than it scales the weights proportional to the number of inputs and outputs to the layers, as shown in Equation 3.9. So the weights are sampled from $U(-a, a)$. *Gain* is set too 0.02 in this project. This scaling process helps to keep variance of activations consistent across layers, helping to create a smoother gradient flow [39].

$$a = \text{gain} \times \sqrt{\frac{6}{\text{fan}_{in} + \text{fan}_{out}}} \tag{3.9}$$

  - *Kaiming weight initilaization:* Kaiming or also called He initialization was developed for networks that use ReLU or LeakyReLUs as their activation functions. This method also samples from a Gaussian distribution with a mean of 0, however its standard deviation is defined with $\sqrt{\frac{2}{n}}$, where n is the number of inputs to the neuron. This can be changed with modifying the argument *mode* from fan_{in} to fan_{out}. This choice is dependent on our end goal, whether we want to stabilize the forward or backward pass. In the BaseNetwork they

chose to initialize this method with fan_{in} [40].

- **ResnetBlock:** This is the Residual Block used in the model, which inherits directly from $nn.Module$. It's main task is to perform operations with a given numbers of layers in its structure, than to apply a skip connection, adding together the input x and the $modified\ x$ to form the output. This is performed so the depth of the network can be modified and also, to create superior flow for the gradient at the backpropagation step [41].

- **InPaintingGenerator:** This model is the generator part of their GAN model. In its own, its built as encoder-decoder architecture. The model's structure is described in Chapter 2 in Section 2.7. The layers and activation functions were provided by inheriting the BaseNetwork and applying a Torch package provided for this purpose.

- **Discriminator:** The other half of the GAN, the Discriminator was designed in the same way as in another model called PatchGAN [14]. In this application the discriminator has 5 convolutional layers. Each channel works with the same kernel sizes and padding with different strides. Each channel doubles the out_channel's number compared to the in_channel's. All of them use a Leaky ReLU with a slope of 0.2 for negative values. The last convolutional layer's out_channel is 1, with a size of $n \times n$ where the matrix represents the number of patches the network considers. Instead of giving back one value of whether the input was fake or real, it suggests a real or fake consideration for these patches, so it is a more detailed feedback [14].

3.7.3 Experimental Setup

To examine the possibilities of this model I have tested in a controlled setup provided on GitHub [36]. As for my future plans, I only needed to perform shape completion. That is the reason why I have only tested that part, where the network is trained with an input channel of three, for X,Y,Z coordinates mapped to 2D planes. For this, I have tried the given MLS dataset, which I have introduced in Chapter 2 in Subsection 2.7.3. In this controlled setup I have expected great results with the pre-trained model, which was trained on the synthetic dataset as mentioned in Chapter 2 Subsection 2.7.3. To see the results of this completion setup, I have chosen an incomplete object representing a car from the given real-world MLS dataset. Then, I have predicted its structure, on the

pre-trained model. As the last step, I have visualized the completed point cloud to see how the model performs.

3.8. My Implementations of MVPCC

In this section, I am going to describe different approaches that I have designed to perform completion task. For these approaches, I have used two models. One, which I have created based on the original model and the original model with few modifications.

3.8.1 Description of My Code Structure

For my implementation of MVPCC, I had two considerations in mind: can the code be structured in a way that it fits my purpose for my dataset and can it be simplified? As the model was written for six-channel inputs, the expected output would also have six channels consisting of XYZ coordinates of the projected points original coordinates and 3 channels for RGB. My dataset also had textures and colors, however my implementation's purpose is to use this for sonar generated point cloud data, which is created underwater, hence it has no colors. So, for this I have created two implementations: the first, which creates depth maps from a given pinhole camera projection and uses **1-channel inputs** and another model, which has **3-channel inputs** corresponding to XYZ. The 3-channel input is similar to the original implementation; however, I have used orthographic projection.

My second reason for creating my implementation was to see if I can simplify the structure of the model. As the input data has simpler structure compared to the original 6-channel input, my assumption was that a simpler model could be able to capture the complexity of this completion task and generate complete structures from sparse or incomplete input. For this simpler structure, I have decided not to use GAN, if not necessary, instead I have tried to create and experiment with a model similar to the original implementation's Generator. This is an Encoder-Decoder architecture described in Chapter 2 in Subsection 2.3.1. For this, I have created different models to fit the input and output sizes described in the previous paragraph dimensions.

3.8.2 MVPCC Implementation for 1-Channel Inputs

As for all the models, I have implemented, there were built in a style, so it is more interpretable, easy to read and understand. For my own implementation, I also wanted to

apply the same design considerations, so it could be scalable at any level, if needed. For this, I have tried to apply module-based design, so functions corresponding to different tasks our grouped separately. For this, I have created different *.py* files so I can manage my code. In *main.py*, I have my main function, which generates the datasets from the provided data stored in *.pcd* extenisions. For this generation process I need functions, to create sparse and incomplete point clouds. These functions can be found in *auxiliary_functions.py*. After this, I need to create projections from the ground truth data input and also from the created sparse or incomplete point clouds as well. For this, I use functions from *projection.py* to create projections and also store them in a scalable way at the correct folders. Its scalability is shown for example in the generation of incomplete or sparse input. I can choose to create multiple different incomplete point cloud from a given input just by modifying the variable `number_of_incomplete_samples` in the main function. Similarly, sparser point clouds could be achieved the same way.

After this, I can train the model. For training I have grouped the necessary functions in *train.py*. The main function in *train.py* is called *train* with the arguments: *dataset size*, *number of incomplete samples*, giving the number of different incomplete samples per target, *number of chunks*, which gives the number of chunks I have divided one epoch to, the learning rate for the optimizer, batch size and the scheduler, changing the learning rate while training. First, in this function I create a model imported from *models.py*, with an optimizer. For optimizers there are two choices, either to use Adam or SGD with scheduler.

Note that the number of chunks is necessary because the model and datasets are too large to load the entire dataset at once. This limitation arises from constraints on cache and GPU memory capacity. That is why function *train* has 2 loops: an outer loop, looping through the epochs and an inner loop, which loops through the chunks one epoch is divided to. In the inner loop with the dataset segment, I create a dataloader for training and testing to see if there is any signs for overfitting. I pass these dataloader to a function *training_loop*. The training loop has a loop which passes through the given dataset fed in batch sizes. The same is done for evaluation, however in that case I use `torch.no_grad()`, so it is only a feedback of the networks loss. In the training loop I also use `torch.nn.utils.clip_grad_norm_`, which clips the gradient and normalizes it, so it prevents the exploding gradient problem. These modules are accessible through Appendix B.

After training, I can predict upon an input object with the module *predict*. Here, after given the input and the ground truth if exists, I will get the output of the model which are the completed depth maps. I can also see the MSE loss between the predicted output and the target as a feedback. After this, I have an option to reproject both the target and the predicted output, given the target's extrinsic and intrinsic matrices. Also, I can visualize just the depth maps as well. These visualization tools both for reprojection or plotting are grouped in the modules *predict* and *reproject*.

3.8.3 MVPCC Implementation for 3-Channel Inputs

For my second implementation, I have used the same or at least similar functions and modules. The general module-based design is the same as shown in the previous Subsection 3.8.2. For the similar but modified modules and functions and objects, that serve the same purpose as before, I have assigned the number "2" in their names, so they are distinguishable. I have only made small changes so the functions and the neural network itself could accommodate the 3-channel inputs. The bigger changes were projection and reprojection, for which I have created new modules called *projection2.py* and *reprojection2.py*. This accommodates the Python implementation of the projection and reprojection processes introduced in Subsection 3.6.2. Also, for the training process I have introduced a new function, which saves the dataloaders and load them whenever needed. This functionality optimizes the code in a way that the dataloaders do not have to be reestablished at the beginning of each epoch. These modules are accessible through Appendix B.

Also, for the second implementation, I have created a function to save a predicted projection image converted to RGB every 10^{th} epoch, so I would have a sense of the model's performance during training. Moreover, I would see how fast does it converge or does the outcome looks visually good, besides from minimizing the loss functions. I have also written a function to accommodate all orthographic projection solutions as described in Subsection 3.6.2.

3.9. Experimental Setup for My Implementations

In my experimental setup I have used two models, one for the depth map estimation which was a replication of sort of the original model and the original InpaintingGenerator model for the 3-channel inputs. The only change I had made in these models, was the number of

residual blocks. I will highlight this number at each presented experimental setup along with the hyperparameters of each model. Another difference between each experimental setup is the losses that I have chosen for training. For most cases, I have used combined loss, which had parts that I have already introduced in Chapter 2 in Section 2.7. Other parts that I have implemented were:

- **Mean Squared Error:** This is a loss which is the squared differences between the input and the target.
- **Object Loss Mean Squared Error:** This is the difference between the input and the target at points, where the target has an object point, meaning the background for this calculation was masked out.
- **Feature Matching Loss:** This is a loss function, that can be used in application of GAN models. It is based on the utilization of the convolutional layers. After passing a batch of inputs through the Discriminator and returning the output, I have tried to return the feature maps, which are the convolutional layers as well, as those are the natural choice of statistics of the Discriminator, when making a decision upon whether the input is generated or original. Then, I have compared the target-passed feature maps to the generated input's feature maps with MSE and used the loss as a part of the loss function. This is theoretically a great choice for loss as the feature maps should also be close to each other, when they are getting closer on a high-level basis [42].

3.9.1 Experiment Setups for 1-Channel Inputs

For the 1-channel inputs corresponding for one depth map per view, I have created multiple setups with the model introduced in Section 3.7. I have created projections with the techniques presented in Section 3.6.1. I have chosen four viewpoints for each point clouds. As for the target, I have sampled 16384 points, and 800 points for the input. I have sampled both with uniform sampling method introduced in Section 3.2. I have saved all the necessary extrinsic and intrinsic matrices with the projections. I have prepared the 4 viewpoints with angles to cover a great part of the objects, so reprojection can happen with the smallest possible loss of information. The images of the depth maps were created with width of 640 pixels and a height of 480. As for variations on the target, I chose to try two ways defining the background: It is either set to 0 or set to -1. These changes will be highlighted at the introduction of each experiment's result. As a proof

of concept experiment, I have tried sparse inputs, however I have written functions to create numerous incomplete inputs as described in Section 3.8.2.

I have combined MSE, Object Loss MSE, Perceptual Loss and TV Loss for the combined loss each with a weights, that will be introduced in Chapter 4. In my first setup I have used a lower number of epochs (set to 10) with all the data I have gathered, to see whether the network is able to set its weight the right way when seeing the data. Aware of these results, I have made changes in my model's parameters accordingly, described in Chapter 4.

3.9.2 Experiment Setups for 3-Channel Input

For this experiments I have used the exact neural networks, that I have introduced in Section 3.7. For this implementation I have used a combination of different losses, that I will highlight in Chapter 4. For creating the inputs I have used three types of projections as introduced in Section 3.6.2, for projecting the point cloud or the mesh itself to the plane.

3-Channel Input without Pruning Point Cloud

This experiment was designed to evaluate the projection method, that was used to create the inputs. This evaluation was based on how the model performed for this input-target pair. The model was trained with differently weighted losses and a different number of epochs to completely avoid mistakes caused by any other factor. This experiment was performed on a target point cloud of 100000 points and an input point cloud of 1000 points. The target and the input had a size of $4 \times 900 \times 900 \times 3$, where 4 is the number of viewpoints.

3-Channel Input with Pruning Point Cloud

This experiment was designed to evaluate the projection method, that was used to create the inputs. This evaluation was based on how the model performed for this input-target pair. The model was trained with differently weighted losses and a different number of epochs to completely avoid mistakes caused by any other factor. Also, I have tried to modify the background of target. I have set it to -1 from 0 and normalized data, so each coordinate is in the $[0, 1]$ range. I have also tried to change the previously discussed (in Section 3.7) weight initialization methods to see how does it contribute to the output.

This was done on a target point cloud of 100000 points and on an input point cloud of 1000 points. The target and the input had a size of $4 \times 900 \times 900 \times 3$, where 4 is the number of viewpoints.

3-Channel Input Pruned with Ray Tracking

This experiment design was created to see whether the pruned point cloud performs better given the same circumstances as before. So, the same initialization and preprocessing techniques were applied, except the projection phase, where I have applied ray tracking to use only points that are visible from that given perspective. I have also applied this technique to smaller resolution projections to see whether that has an effect on the training's performance. The target and the input had a size of $4 \times 900 \times 900 \times 3$, where 4 is the number of viewpoints. For the smaller resolution projections the input size was set to $4 \times 100 \times 100 \times 3$ with resolution set to 0.01 which resolution's role is described in Section 3.6.2.

3-Channel Input with Mesh Projection Target

This experiment design was set to see whether the change of projection technique would outperform the previously introduced techniques. This is the last orthographic projection technique I have introduced in Section 3.6.2. It was trained on the same model setups as previously introduced variations. The model was trained on two types of data input: one with the background set to -1 and one with the background set to zero and other coordinates normalized between $[0, 1]$. The target of this model was not necessarily set by the number of points but rather the whole mesh was projected. At the projected points, the coordinates of the original visible point coordinates were stored. The input points were 800 sampled points from the mesh from which at each projection the visible points were chosen. The target and the input had a size of $4 \times 256 \times 256 \times 3$, where 4 is the number of viewpoints.

3.9.3 Folders for Storing and Loading

I have also created folders for different types of data, so it would become easier to scale and read my project. For incomplete, sparse and general depth maps I have created the folders *Incomplete_depth_maps*, *sparse_depth_maps*, *complete_depth_maps* to store the depth maps and also the matrices regarding the projection and the camera settings. For projections creating a 3-channel inputs, I have created two folders; *XYZ_projections* for

the target projections and *XYZ_projections_sparse* for sparse input projections. I have also saved all my trained models in the folder *trainedmodels*, with file extensions *.pth*, which stores the weights and biases for my trained torch models. These folders are also accessible as described in Appendix B.

Chapter 4

Results

In this chapter I am going to describe the results I have obtained by completing the tasks I have proposed in Chapter 3.

4.1. PCN

4.1.1 Testing Sparse Input from ModelNet40

For this task I have expected the model to perform well for most of the objects, as the coarse of the model will still be fed to the network. I tested at three levels: 80, 400, and 800 points. I did not choose fewer points, as they would not have provided a clear representation of the structure. I did not choose more points, as above 800, the structure becomes too clear for the network to recognize. I have tested on all of the categories. For showcasing the results, I have chosen two structures. Their ground truth 2048 point point clouds can be seen in Figure 4.1 and n Figure 4.2.

The CD losses for different inputs are included in Table 4.1. The Chamfer Distance between the ground truth and the generated point cloud decreases by increasing the input point cloud's sizes. This is the expected behavior. However, this could very well be the cause of the fact that the network has less points to predict as it could choose the input as a coarse for its prediction if the network learned to do that at training phase. The network does not create similar CD losses compared to the results summarized in Figure A.4. This is because this network was trained on a smaller dataset (with objects having less points) than the one I have introduced in Chapter 2 in Section 2.4. The results met my expectations. From the experiment, I have attached two figures. One of

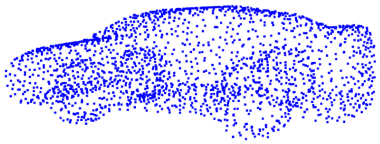

Figure 4.1: Ground Truth of the first object

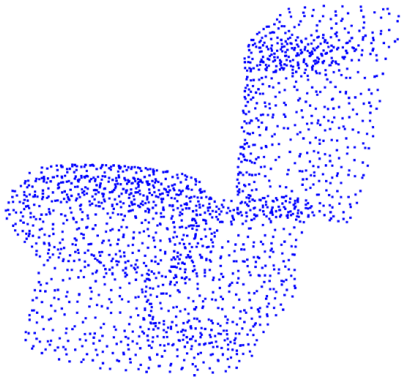

Figure 4.2: Ground Truth of the second object

the car in Figure 4.3 and one in Appendix A in Figure A.5.

	80 points	**400 points**	**800 points**
Car	**0.0949**	**0.0489**	**0.0407**
Toilet	**0.0844**	**0.0446**	**0.0373**

Table 4.1: This table summarizes the CD losses for the two categories I chose to show.

4.1.2 Testing Incomplete Input from ModelNet40

In this task, I have created partial inputs with the modified version of *cut* function introduced in Chatper 3 Section 3.3. Before I cut, the input was 2048 points. The results of CD losses were: **0.2063** for the car, **0.1701** for the toilet and **0.1771** for an object with low complexity (table). These distances are higher than the distances measured in Section 4.1.1. Working with partial input is a harder task in general, the network shows no sign of creating a great coarse for the model either. The partial inputs are included in Appendix A in Figures A.6, A.7 and A.8.

4.1.3 Testing on My Dataset

From my dataset, I chose to show 2 results which represent the whole behavior of the network. My input was the sparse point cloud of the ship shown in Section 3.3. I chose not to have any offsets for the points. The input was 1024 points. I have attached the result of it in Figure 4.4. The CD loss was **0.5310**. As this is an unseen category and a complex structure the result fits my expectation. In the figure the result is sideways compared to the input.

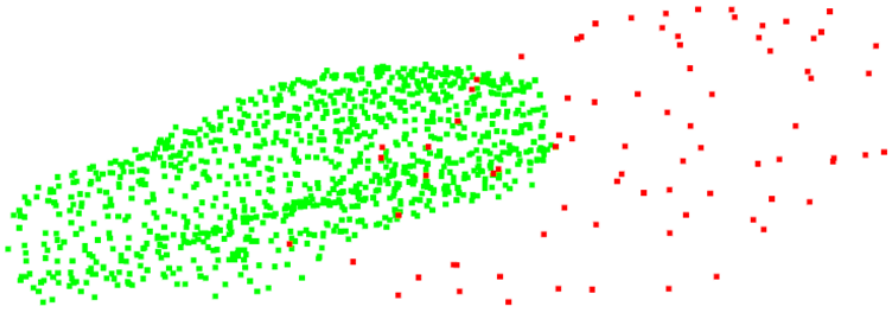

Figure 4.3: This Figure shows the 80 point input for the car and generated point cloud from it. The red point cloud is the input and the green is the output.

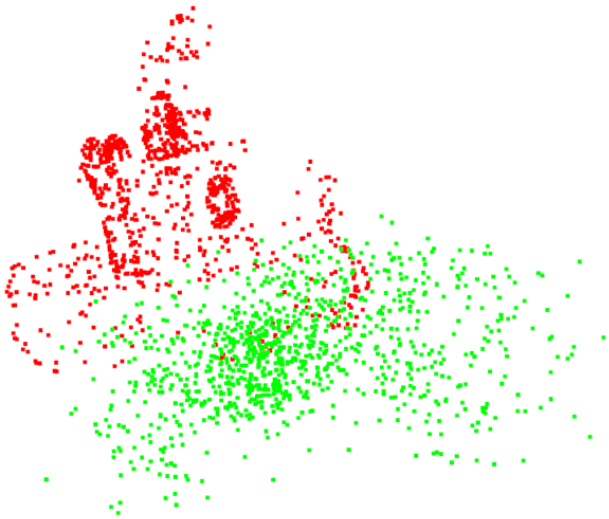

Figure 4.4: Figure of the input ship (red) and the reconstructed point cloud (green). The reconstructed point cloud is sideways.

For another test, I have chosen a general input, a partial part of a boat's hull. The CD loss was **0.5271**. The images of the input and the generated point cloud is attached in Appendix A in Figure A.9. As these inputs were an unseen category and had bigger distances between each point the network failed to generalize. This behavior might be due to the nature of this network, introduced in Figure A.1.

4.1.4 Training and Testing Sparse Input on ShapeNet

As I have introduced the experiment in Chapter 3 Section 3.5, I have trained learning3D's implementation on sparse input to see how the network generalizes. I have successfully created my own dataloaders and managed to train the network. I have trained it with the settings described in Section 3.5. After training, I visualized some results to see whether the training only minimized the loss or actually created a visually appealing output. I took a random sample from the training example. In Figure 4.5 and 4.5 I have attached the two inputs and their results as well. The inputs are red, and the predicted results are green. The estimated CD losses are **0.016** and **0.0105**. As it shows, the Chamfer Distance is minimized; however, the model failed to generalize. For two totally different point clouds, the model created roughly the same point clouds. Only roughly, as the length and the density of the point cloud was not the same, but changed according to the input.

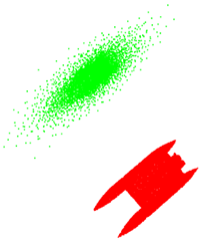

Figure 4.5: A sample input object for PCN trained on ShapeNet in **red** and the output in **green**.

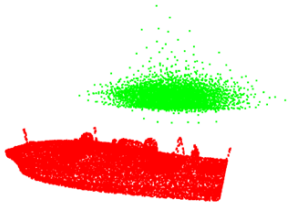

Figure 4.6: A sample input object for PCN trained on ShapeNet in **red** and the output in **green**.

Compared to the results I have measured at the model which was trained on ModelNet40 and tested with a ship input(4.4), the results are improved regarding Chamfer Distance by a factor of at least 10, consistently. However, as shown on the attached figures, the model has generalization issues as the created point cloud does not resemble any individual shape for the input ships. It is a cluster of points that statistically is the best cluster for minimizing CD loss.

Upon receiving these results, I have decided to create a model, where I use input point clouds, where only those points are sampled that are visible from the outside. I have made these changes, so the model would not focus on the internal structure. I have created these point clouds with the reprojection of orthographic projections introduced

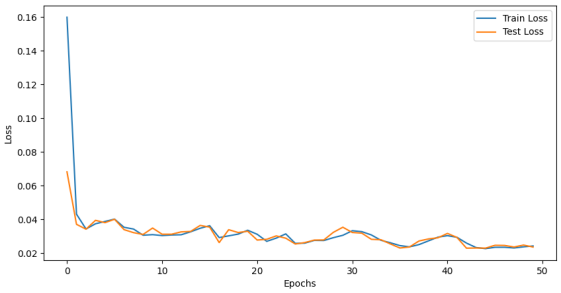

Figure 4.7: Training results for sampled input point clouds sized 2500, through 50 epochs.

in Chapter 3 in Section 3.6.2. After this, I have sampled 2500 and 1024 points from these points for a 100 objects. I have split these datasets to training and validation sets with the ratio 0.9 to 0.1. I have trained the PCN for both types. The training loss through epochs and the validation loss was decreasing for both cases. I have attached a Figure 4.7, showing how the losses decrease in the case of 2500 point inputs. The same results can be seen in the other training results for 1024 point clouds, for which I have attached the figure in Appendix A in Figure A.10. Unfortunately, despite training results, the model is still not able to generate visually great objects.

The results of this PCN model's show, that the model itself performs fine, with well separated input classes, having small variety at each class. Also performs well on higher sparsity levels in these controlled settings. However, trained with a class having a high variety, the model's capability is limited by its loss function.

4.2. Orignal MVPCC Implementation

For this implementation I have used a pre-trained model of the original model as described in Chapter 3 in Section 3.7 with the MLS Dataset, described in Chapter 2 in Section 2.7. I have successfully implemented the WSL for experiment and after that I managed to compile the code for testing. I have used a sparse input of a car and have passed through the completion network. After this, I have saved the output of the completion network. The input and the output can be seen in Figure 4.8 and 4.9. As it is shown, the input is sparse and also seem to have occluded parts. The output is really dense and looks like it kept the shape even at fine details such as the cars mirrors and wheels. As a result of performing the completion task from more views and on images, the results seem to be dense. This is caused because I did not had to provide a number of output points the model should densify to, the model can decide upon this number itself. This is a great

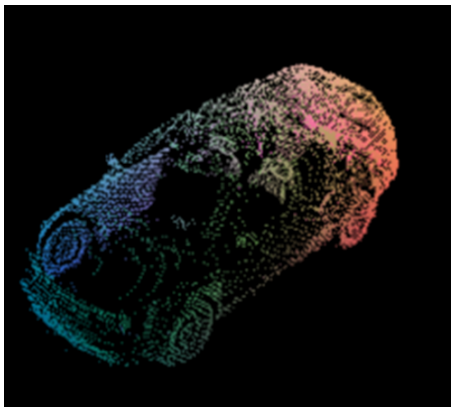

Figure 4.8: Sparse input point cloud of a car sampled from the MLS dataset.

Figure 4.9: Generated Output of a car sampled from the MLS dataset.

use case and seems to apply better on real-world completion tasks.

4.3. My Implementations of MVPCC

In this section, I am going to describe my results based on the methods and experimental setups I have introduced in Chapter 3 in Section 3.8.

4.3.1 MVPCC for 1-Channel Inputs

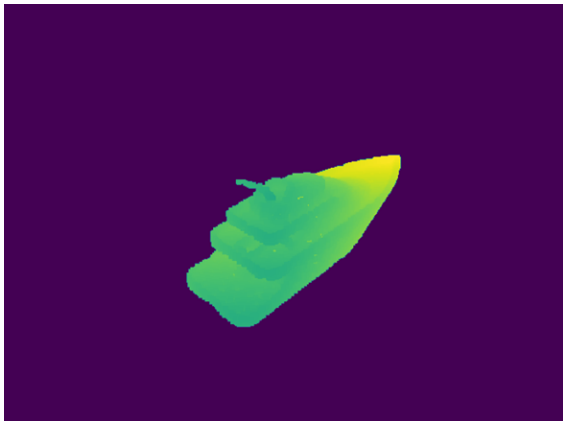

Figure 4.10: Target depth map of an object with the index **66**.

For the positions of the virtual cameras; each is first defined to look at the origin, with a front vector $(1, 1, 0)$ and a up vector $(0, 1, 0)$. Then, I have rotated each camera with a given number of pixel movements. This can be translated to degree, as Open3d documentation mentions, it moves 5.8178 pixels/degree [31]. I have chosen the following rotatoins: $(35°, 0°)$, $(-35°, -17°)$,$(-130°, 25°)$,$(141°, 35°)$. From these I have created

robust projections that covered the objects. The results can be seen in Figure 4.10. The figure shows the projection of ship 66 from the second virtual camera's perspective. In this example, the background is set to 0. These depth maps could be reprojected. The reprojections were not perfect even for the target, as it returned with small artifacts as shown in Figure 4.12 on the rear end of the object shown next to the whole reprojection of the target on Figure 4.11. The artifacts are visible, as we can see each point being reprojected from each direction, creating 'X' formations. However, this should not affect the results of the training.

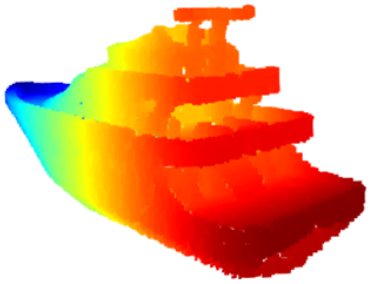

Figure 4.11: Distant view of the reprojection of a target object with the index **66**.

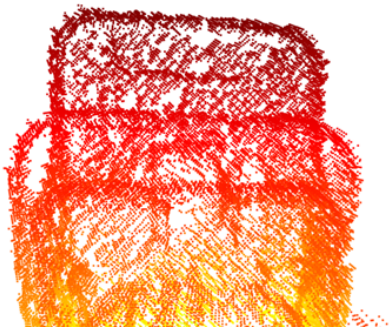

Figure 4.12: This figure shows the artifact formed by reprojection on ship indexed **66**.

Training First, I have trained on lower epochs and on all of the watercraft objects, contained in ShapeNet, with the background set to 0, using 4 residual blocks. However, this gave bad results, so I have decided to pick 256 objects out of the original 1910 ships, and train the network on these ships with higher epochs and background set to -1 to see whether the network is able to learn. I had also split the dataset to training and validation sets with a ratio of 0.8 to 0.2. I have used four losses in combined loss: object loss with a weight of 50, MSE loss with a weight of 5, perceptual loss with a weight of 1.5 and TV loss with a weight of 1. Perceptual loss or vgg loss was used for RGB channels in the original implementation, however I wanted to test its feature matching capabilities on a single channel input. With these losses, the the network was only able to identify the object, as shown in Appendix A in Figure A.11 on ship indexed 65. Also, I have attached a Figure A.12 in Appendix A, which shows the target of ship indexed 65, created with 16384 uniformly sampled points.

After these results I have decided to increase the epoch to 30, and to not use perceptual loss as it seemed like it enhanced the edges of the object. After these changes, I had

great results for the depth maps as shown in Figures 4.13, 4.14, 4.15, 4.16. The MSE loss between the output and the target object was **0.00725**. After receiving these results for the depth maps, I have decided to **try to enhance my real-world data**, introduced in Chapter 3 in Section 3.3 Subsection 3.3.4. After pruning the point cloud, I normalized its coordinates to the $[0, 1]$ range and adjusted its orientation to the correct alignment. Once this setup was complete, I generated the corresponding depth maps. One of the sparse depth map and the enhanced result of the corresponding view is attached in Appendix A in Figure A.13 and A.14. As shown in these figures, the model was able to improve the projection in various ways. It excelled at estimating distant points and their surroundings, and similarly, it handled close points effectively. The model filled the gap on the deck of the ship and enhanced the outlines of the bow section, demonstrating its ability to improve both distant and nearby details to some degree. Unfortunately, in neither cases was the reprojection successful. This issue might be caused by the wide variety of distances that the model needs to predict for each depth map separately.

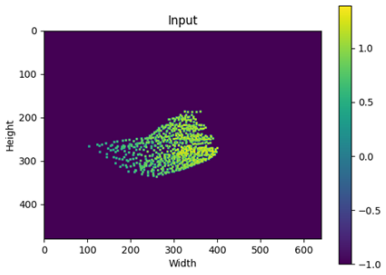

Figure 4.13: Projection of the input sample indexed **66** from the first view.

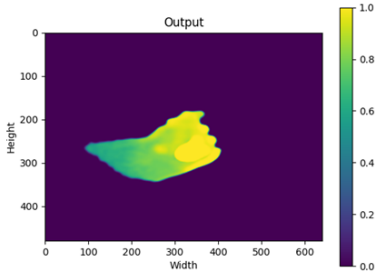

Figure 4.14: Output for sample indexed **66** from the first view.

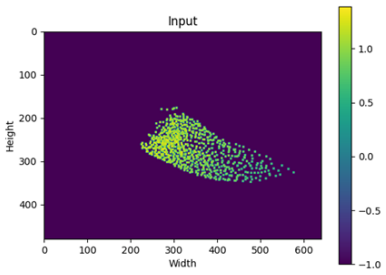

Figure 4.15: Projection of the input sample indexed **66** from the second view.

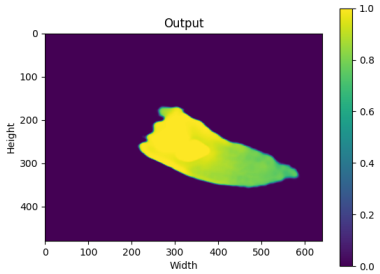

Figure 4.16: Output for sample indexed **66** from the first view.

4.3.2 My Implementation of 3-Channel Inputs for MVPCC

In this subsection, I am going to introduce and discuss my results for the experimental setups I have introduced in Chapter 3 in Section 3.8. I have distinguished four main types of input and target pairs regarding this model: 3-channel input without pruning, 3-channel input with pruning, 3-channel input pruned with ray tracking and 3-channel input with mesh projection targets.

Firstly, I will discuss my results regarding the **3-channel input without pruning and with pruning**. The Figure 4.17 shows a projection of ship indexed 0. The image had been scaled to RGB levels between 0-255 and the XYZ coordinate matrices had been visualized as a RGB image. As shown in the image, the point cloud is generally 'messy' both for the target and also for the inputs. This was true for training a network without pruning or with pruning with neighbors.

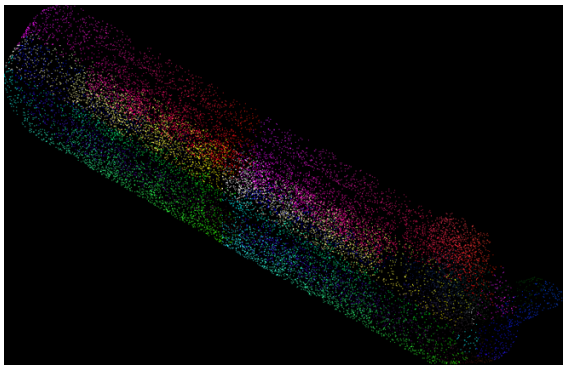

Figure 4.17: This figure shows a projection of ship indexed **0** generated with orthographic projection, without pruning.

For training I had chosen 256 objects. I had also split the dataset to training and validation sets with a ratio of 0.8 to 0.2. For the losses, I have tried different losses, with and without each part of the combined loss presented in Chapter 3 in Section 3.9. The general idea was to keep Object loss with a high weight compared to other losses. MSE and TV loss were mainly a part of the loss for the masking out the background, so if I have trained with a high number of epochs, I could keep the weights of MSE and TV loss low. I have trained with 50 epochs. I have set the number of residuaal blocks in the model to 2. The results validated my assumption, that I had made upon seeing Figure 4.17, as the model failed to generalize and make visually appealing outputs.

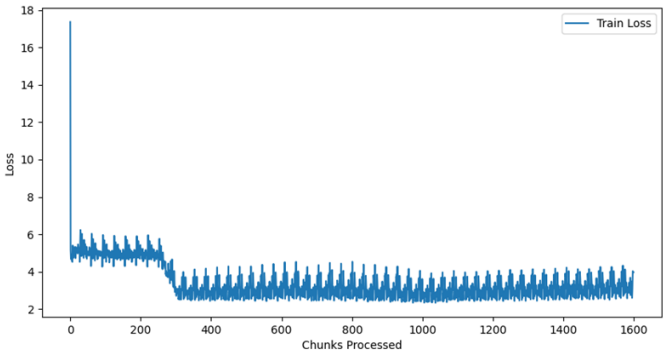

Figure 4.18: In this figure I have attached a plot showing how the training loss had changed trough out epochs and chunks.

As it is shown in Figure 4.18, the training loss is stuck, so it cannot gather enough information out of the input to create a similar image to the target. The output of a training sample indexed 65 through epochs is attached in Appendix A in Figures A.15, A.16 and A.17. As it is shown on the predictions the model makes through out training, it shows that the model fails to predict the correct form for the output and creates holes in the projections. This is probably caused by the MSE loss and the Object Loss. As the target itself has a lot of holes in its structure, neither loss can encounter this by creating a smooth surface as I do not compare my results to a smooth surface either. Moreover, it is also shown how the model failed to predict distant coordinates. As shown in A in Figure A.16 at the 10^{th} epoch, the model understood the outline of the object, however, for some parts it brought it to the front and forgot some other parts. It could have been caused by the fact that there are negative coordinates, so in the next experiment setups, I have decided to use a normalized point cloud, where each point coordinate is normalized between $[0, 1]$. As the model failed to identify negative numbers. I also set the background to 0 instead of -1.

As I have proposed, I have tried multiple **weight initialization methods**. I have tried Xavier, He and normal initializations. The main difference between them was at the beginning of each training, after that each converged to the same output. Normal weight initialization seemed to fit my problem the best that is why I used that in future experiments.

As an experiment to see whether the Discriminator worked well, I **had trained a model, where I have only used adversarial loss for training**. In this way, I was able to see how it affects the combined loss. For this I have printed out the fake loss

and real loss which are the losses of BCE, for output of Generator and target. At the beginning these were both around 0.5-0.6, meaning, the network was not able to decide upon whether these projected images are real or fake. But already at the first epoch at the second chunk the Generator learned how to fool the Discriminator. So, even though the value for real loss was really small, around 0.0002, the loss for the generated images was the same. However, it is not the same as learning features, as the corresponding generated images were bad, the model simply learned how to fool the Discriminator. I have attached one training sample image in Appendix A in Figure A.18 at the end of the first epoch. This projection already fools the Discriminator.

4.3.3 Training On 3-Channel Inputs Pruned with Ray Casting

For ray casting, I had used two approaches: selecting visible points with ray casting and projecting the mesh itself for the target. In both approaches I had used point coordinates normalized between $[0, 1]$. I also used normal weight initialization and in the model I have used 2 residual blocks.

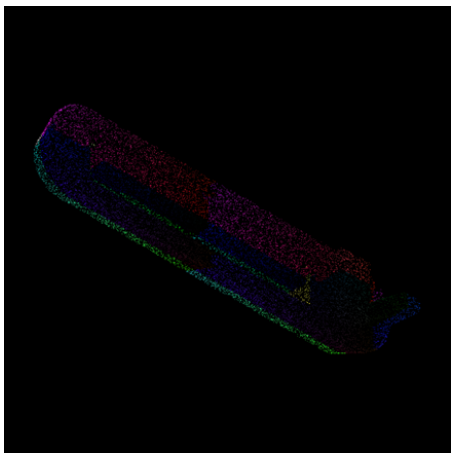

Figure 4.19: Projection of object indexed **0** with a resolution of 0.001.

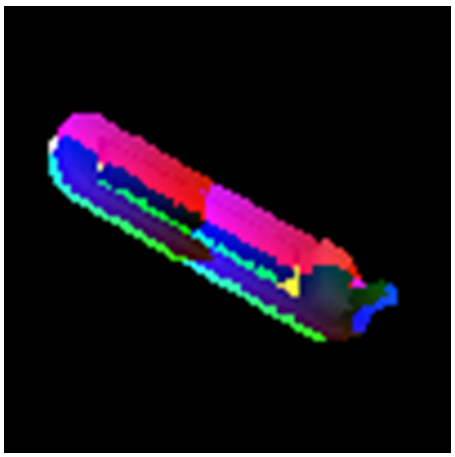

Figure 4.20: Projection of object indexed **0** with a resolution of 0.01

First, I will discuss my results regarding the experiment in which **3-channel input is created through pruned with ray tracking**. I have used the methods described in Chapter 3 in Section 3.6.2. First, I have chosen 100000 points for the target and 500 points for the input. I have chosen at each projection only the visible points. I have created two setups, one, where I set the resolution of grid to the projection image to 0.001 having a padded image size of $900 \times 900 \times 3$ for each projection. Then, I have set the resolution to 0.01, creating projections of size $100 \times 100 \times 3$. These projections can

be seen in Figure 4.19 and in Figure 4.20. After reprojecting both sets of projections, each produced good results in terms of visual quality, so I trained a model for both. For the projection with input size $4 \times 900 \times 900 \times 3$, I have also got outputs with holes in it, as the development of these structural artifacts can be seen in Appendix A in Figures A.19, A.20, A.21.

The **use of smaller resolution** grids only helped to improve performance, however it did not solve the problem. I have changed the resolution, so on the target I would have less holes, making it easier for the MSE and Object loss to converge. This change seemed to increase the performance, however the results still had holes or artifacts. The results of the training processes throughout the training can be seen on ship 65 in Figure 4.21. So even with this method, I was not able to perform an appealing reprojection.

Figure 4.21: These are the predictions for object indexed **65** after epochs 1, 10, 50, 100, 150 respectively.

To improve my results, I have decided to use the **mesh projections directly as targets** so that the ground truth projections do not have holes. I have implemented the projection techniques described in Chapter 3 Section 3.6.2. As these meshes are detailed, I had to simplify the structure of them. For this, I have used Quadratic Decimation as introduced in Chapter 3 Section 3.2. I have simplified the the face count from n to $\frac{n}{50}$. One of the results of this simplification process is attached in Appendix A in Figure A.22 performed on ship indexed 0. One of the results of the reprojection of sample object, indexed 0 can be seen in Appendix A in Figure A.23. In Figure 4.22 I have attached the projection of the mesh from a given perspective and the corresponding points I have sampled, visible from the camera's perspective. The values of the original x, y and z coordinate are separately shown on the plots.

I have trained my models for this experiment with 8 residual blocks, with 256 objects with a train/validation ratio of 0.9 to 0.1. I have experimented with point clouds of sizes 800 and 2500 for sparse input. I have trained these input/target pairs 6 times with

different combined losses to see how does the result changes visually. For example, my best setup was, where I had used combined loss consisting of object loss with a weight of 3, MSE loss with the weight of 2, adversarial loss with a weight of 10 and feature matching loss with a weight of 10 with the input point cloud having 2500 points. In these cases the model visually predicted really great projections, as even finer details had started to form after epochs 50. The result for a training sample given an input similarly sampled as show in Figure 4.22, is shown in Figure 4.23. This is the result after 100 epochs. To have a better understanding about the ground truth details I have attached the target projection's x plane in Appendix A in Figure A.24. Unfortunately, despite these promising results, the reprojection did not work as expected, as the generated point cloud turned out to be too messy. Even though the generated surface for each projection appeared sufficiently accurate in their respective images, the model seemed unable to establish stronger inner connections between them.

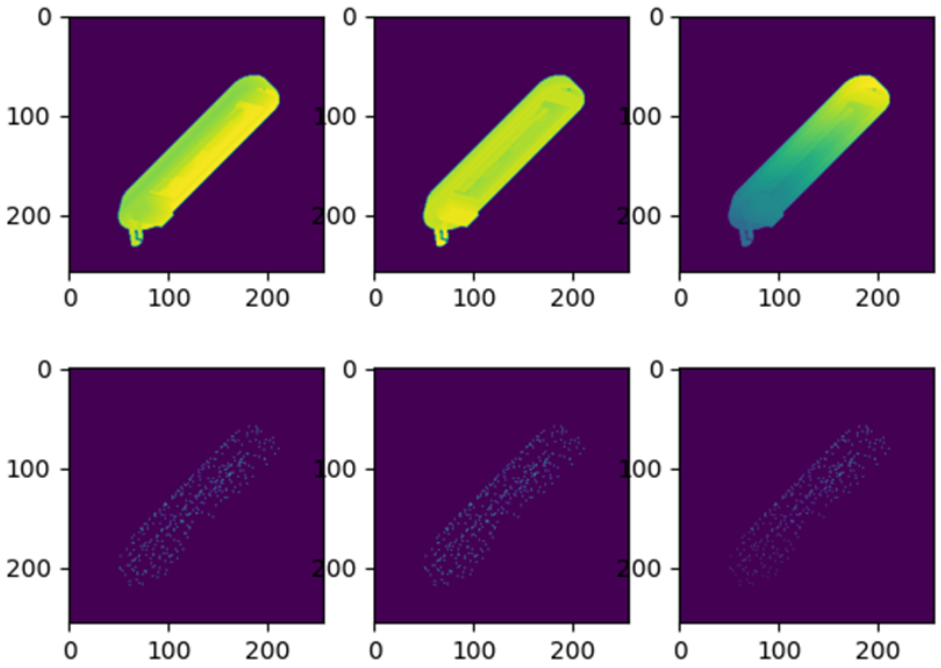

Figure 4.22: Projection images for a direct mesh projection (**first row**) and for the sampled points (**second row**) from a given perspective. The original x, y, z coordinates are plotted separately.

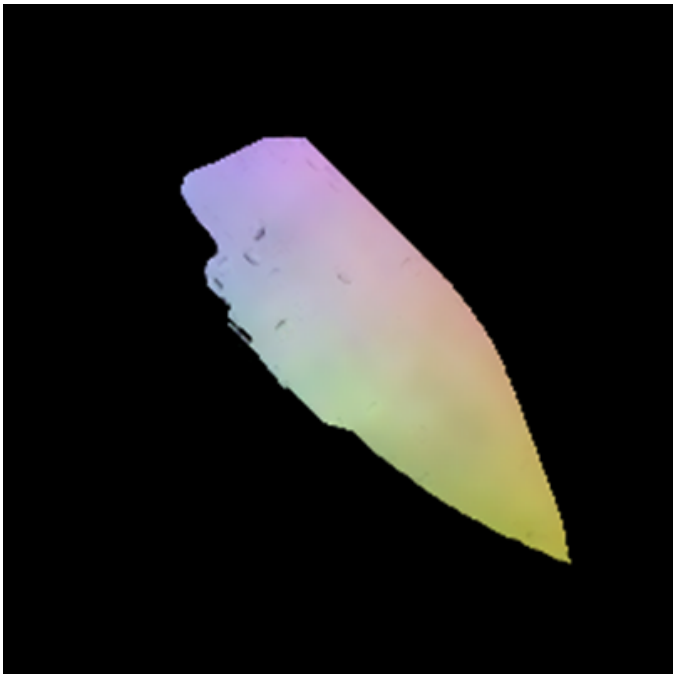

Figure 4.23: This is the output of the model trained with direct target mesh projections. This projection was generated of the output of the object, indexed **66**.

Chapter 5

Summary

5.1. Conclusion

5.1.1 PCN

ModelNet40

From the proposed methods, I have implemented the baseline model PCN. On this network I have tried three datasets, ModelNet40, my own dataset and ShapeNet. On the first dataset I tested both sparse and partial inputs. On the sparse input the network worked well, producing great dense point clouds. The outputs were great representations of the objects, even when the input had only 80 points. I quantitatively evaluated the results of each categories, and showcased the qualitative results on 2 objects.

For incomplete inputs the model performed worse. It had a hard time associating the input's structure with previously learned knowledge. As this network cannot define neighborhoods and smaller structural details, this behavior was expected.

In the qualitative evaluations neither a sparse input from a boat nor a partial input from a boat's hull achieved good results, as they were from unseen categories. However, the boat's hull is a general shape, for which I have expected better results. These results show the limitations of this model.

ShapeNet

As shown in chapter 4 Section 4.1, I managed to train and test the model, using the trained model. Unfortunately, the results were not great visually, but the network tried to minimize both the training loss and the validation loss. However, the final result

visually did not match my expectations. It had a lower CD loss for a given sample than the model trained on ModelNet40. It is important to note that the tested object had more points and the predicted shape also had more points, so there is less loss. I also tried training the model using the points on the visible surface of the object as input. Both models failed to generalize, although the losses were minimized. In my experimental setups, the PCN failed to generalize and failed to perform shape completion for my input data set.

5.1.2 MVPCC

1-Channel Inputs

In this experiment, I created a version of MVPCC where the input did not take three or six channel inputs, but rather a depth map from each viewpoint. The creation of these depth maps was based on my projection application. These processes proved to be successful, which I concluded after visually inspecting the reprojection of the target projections. The training part was difficult, but after trial and error the network predicted good quality depth maps that closely resembled the original targets. The prediction was worse for distant points, but sufficiently smooth to have a low loss compared to the target. Unfortunately, the reprojection process did not produce a good point cloud from the depth maps. However, I found this experiment successful in terms of proving the completeness of the proposed model. This success was also evident with real-world data, as when I ran the completion on the given input, the output showed noticeable improvements.

3-Channel Input

The results of the initial implementation were as expected. The pre-trained model worked well for the completion task. My modifications to create the 3-channel inputs via orthographic projections were successful, as I was able to reproject the original inputs and targets with minimal loss. However, training failed for the first two types of projections, despite the changes to the model hyperparameters and input types. The model generally tried to minimize loss, as shown in Chapter 4 Section 4.3, but was unable to produce a visually appealing output point cloud after reprojection. Despite the holes in the output of both models that I have trained, the model generally tries to create a smooth surface that highlights depth. These results justify the model's capabilities. Furthermore, the final model, where I used directly projected meshes as targets, performed exceptionally well. It was able to recover fine detail even from very sparse input. This success was

largely due to the setup of the GAN and the inclusion of the feature matching loss, which effectively steered the training process in the right direction.

5.2. Future plans

In the future, I would like to improve the results of my MVPCC implementations. I am interested in working with this model as I believe that creating projections and working within the image feature space is a more effective approach than working directly in 3D space. In addition, I would like to develop an accurate completion and reprojection of real sonar data. I also aim to improve my overall understanding of these architectures to gain a deeper insight into the problem and improve my ability to develop effective and optimized solutions. I also intend to use this model with more than four projections, as I believe that four views are generally sufficient for simpler structures such as cars, however, for more complex structures, such as ships, this number needs to be higher. After making these changes, it will be much easier to achieve better shape completion on real data.

Bibliography

[1] D. Jung, J. Kim, and G. Byun, "Numerical modeling and simulation technique in time-domain for multibeam echo sounder," *International Journal of Naval Architecture and Ocean Engineering*, vol. 10, no. 2, pp. 225–234, Mar. 2018, ISSN: 2092-6782. DOI: 10.1016/j.ijnaoe.2017.08.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2092678216305179 (visited on 12/13/2023).

[2] EdgeTech, *Application Note: Sidescan Sonar Beamwidth EdgeTech*, 2005. [Online]. Available: https://www.edgetech.com/wp-content/uploads/2019/07/app_note_beamwidth.pdf.

[3] O. R N A, P. D G, and Khomsin, "Analysis of angular resolution and range resolution on multibeam echosounder R2 Sonic 2020 in Port of Tanjung Perak (Surabaya)," *Geomatics International Conference 2020*, vol. IOP Conf. Series Earth and Environmental Science, no. 721, 2020. DOI: 10.1088/1755-1315/731/1/012032.

[4] International Hydrographic Bureau, *MANUAL ON HYDROGRAPHY* (Publication C-13). B.P. 445 - MC 98011 MONACO Cedex: International Hydrographic Bureau, Dec. 2010, vol. 5th edition, Chaoter 3 Depth Estimation. [Online]. Available: www.iho.int.

[5] A. E. Albright Blomberg, A. Austeng, R. E. Hansen, and S. A. V. Synnes, "Improving Sonar Performance in Shallow Water Using Adaptive Beamforming," *IEEE Journal of Oceanic Engineering*, vol. 38, no. 2, pp. 297–307, Apr. 2013, Conference Name: IEEE Journal of Oceanic Engineering, ISSN: 1558-1691. DOI: 10.1109/JOE.2012.2226643. [Online]. Available: https://ieeexplore.ieee.org/document/6401207 (visited on 12/13/2023).

[6] N. J. Beaudry and R. Renner, *An intuitive proof of the data processing inequality*, arXiv:1107.0740 [quant-ph], Sep. 2012. DOI: 10.48550/arXiv.1107.0740. [Online]. Available: http://arxiv.org/abs/1107.0740 (visited on 06/02/2024).

[7] W. Yuan, T. Khot, D. Held, C. Mertz, and M. Hebert, *PCN: Point Completion Network*, en, arXiv:1808.00671 [cs], Sep. 2019. [Online]. Available: http://arxiv.org/abs/1808.00671 (visited on 06/02/2024).

[8] Y. Ibrahim and C. Benedek, "MVPCC-Net: Multi-View Based Point Cloud Completion Network for MLS data," *Image and Vision Computing*, vol. 134, p. 104 675, Jun. 2023, ISSN: 0262-8856. DOI: 10.1016/j.imavis.2023.104675. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0262885623000495 (visited on 12/02/2024).

[9] C. Saharia, J. Ho, W. Chan, T. Salimans, D. J. Fleet, and M. Norouzi, *Image Super-Resolution via Iterative Refinement*, arXiv:2104.07636 [cs, eess], Jun. 2021. DOI: 10.48550/arXiv.2104.07636. [Online]. Available: http://arxiv.org/abs/2104.07636 (visited on 06/03/2024).

[10] Ahmadsabry, *A Perfect guide to Understand Encoder Decoders in Depth with Visuals*, en, Jun. 2023. [Online]. Available: https://medium.com/@ahmadsabry678/a-perfect-guide-to-understand-encoder-decoders-in-depth-with-visuals-30805c23659b (visited on 06/02/2024).

[11] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, *Generative Adversarial Networks*, arXiv:1406.2661, Jun. 2014. DOI: 10.48550/arXiv.1406.2661. [Online]. Available: http://arxiv.org/abs/1406.2661 (visited on 12/02/2024).

[12] L. Pan, X. Chen, Z. Cai, *et al.*, "Variational Relational Point Completion Network," en, in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Nashville, TN, USA: IEEE, Jun. 2021, pp. 8520–8529, ISBN: 978-1-66544-509-2. DOI: 10.1109/CVPR46437.2021.00842. [Online]. Available: https://ieeexplore.ieee.org/document/9577912/ (visited on 06/02/2024).

[13] L. Pan, "ECG: Edge-aware Point Cloud Completion with Graph Convolution," *IEEE Robotics and Automation Letters*, vol. 5, no. 3, pp. 4392–4398, Jul. 2020, ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2020.2994483. [Online]. Available: https://ieeexplore.ieee.org/document/9093117/ (visited on 06/02/2024).

[14] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, ISSN: 2380-7504, Oct. 2017, pp. 2242–2251. DOI: 10.1109/ICCV.2017.244. [Online]. Available: https://ieeexplore.ieee.org/document/8237506 (visited on 12/02/2024).

[15] J. Johnson, A. Alahi, and L. Fei-Fei, *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*, arXiv:1603.08155, Mar. 2016. DOI: 10.48550/arXiv.1603.08155. [Online]. Available: http://arxiv.org/abs/1603.08155 (visited on 12/02/2024).

[16] L. A. Gatys, A. S. Ecker, and M. Bethge, "Image Style Transfer Using Convolutional Neural Networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, ISSN: 1063-6919, Jun. 2016, pp. 2414–2423. DOI: 10.1109/CVPR.2016.265. [Online]. Available: https://ieeexplore.ieee.org/document/7780634 (visited on 12/02/2024).

[17] A. X. Chang, T. Funkhouser, L. Guibas, *et al.*, *ShapeNet: An Information-Rich 3D Model Repository*, arXiv:1512.03012 [cs], Dec. 2015. DOI: 10.48550/arXiv.1512.03012. [Online]. Available: http://arxiv.org/abs/1512.03012 (visited on 06/02/2024).

[18] Y. Nie, Y. Lin, X. Han, *et al.*, *Skeleton-bridged Point Completion: From Global Inference to Local Adjustment*, arXiv:2010.07428, Oct. 2020. DOI: 10.48550/arXiv.2010.07428. [Online]. Available: http://arxiv.org/abs/2010.07428 (visited on 12/03/2024).

[19] B. Nagy and C. Benedek, "3D CNN-Based Semantic Labeling Approach for Mobile Laser Scanning Data," en, *IEEE Sensors Journal*, vol. 19, no. 21, pp. 10 034–10 045, Nov. 2019, ISSN: 1530-437X, 1558-1748, 2379-9153. DOI: 10.1109/JSEN.2019.2927269. [Online]. Available: https://ieeexplore.ieee.org/document/8756228/ (visited on 12/03/2024).

[20] G. Turk, *The PLY Polygon File Format*, 1998. [Online]. Available: https://web.archive.org/web/20161204152348/http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html (visited on 12/03/2024).

[21] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *2011 IEEE International Conference on Robotics and Automation*, ISSN: 1050-4729, May 2011, pp. 1–4. DOI: 10.1109/ICRA.2011.5980567. [Online]. Available:

https://ieeexplore.ieee.org/document/5980567/authors (visited on 12/03/2024).

[22] W. Technologies, *Wavefront OBJ File Format*, eng, 1990. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml (visited on 12/03/2024).

[23] I. 3D Systems, *STL (STereoLithography) File Format Family*, eng, 1988. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000504.shtml (visited on 12/03/2024).

[24] Z. Wu, S. Song, A. Khosla, *et al.*, *3D ShapeNets: A Deep Representation for Volumetric Shapes*, arXiv:1406.5670 [cs], Apr. 2015. DOI: 10.48550/arXiv.1406.5670. [Online]. Available: http://arxiv.org/abs/1406.5670 (visited on 12/04/2024).

[25] J. Jung, Y. Lee, D. Kim, D. Lee, H. Myung, and H.-T. Choi, "AUV SLAM using forward/downward looking cameras and artificial landmarks," in *2017 IEEE Underwater Technology (UT)*, Feb. 2017, pp. 1–3. DOI: 10.1109/UT.2017.7890307. [Online]. Available: https://ieeexplore.ieee.org/document/7890307 (visited on 12/04/2024).

[26] M. Abadi, A. Agarwal, P. Barham, *et al.*, *TensorFlow, Large-scale machine learning on heterogeneous systems*, Nov. 2015. DOI: 10.5281/zenodo.4724125. [Online]. Available: https://github.com/tensorflow/tensorflow (visited on 12/05/2024).

[27] J. Ansel, E. Yang, H. He, *et al.*, *PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation*, Publication Title: 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24) original-date: 2016-08-13T05:26:41Z, Apr. 2024. DOI: 10.1145/3620665.3640366. [Online]. Available: https://pytorch.org/assets/pytorch2-2.pdf (visited on 12/03/2024).

[28] *Pypa/pip*, original-date: 2011-03-06T14:30:46Z, Dec. 2024. [Online]. Available: https://github.com/pypa/pip (visited on 12/05/2024).

[29] *Numpy/numpy*, original-date: 2010-09-13T23:02:39Z, Dec. 2024. [Online]. Available: https://github.com/numpy/numpy (visited on 12/03/2024).

[30] *GitHub - matplotlib/matplotlib: Matplotlib: Plotting with Python.* [Online]. Available: https://github.com/matplotlib/matplotlib (visited on 12/03/2024).

[31] *Isl-org/Open3D*, original-date: 2016-12-02T16:40:38Z, Dec. 2024. [Online]. Available: https://github.com/isl-org/Open3D (visited on 12/03/2024).

[32] AyajiLin, *Leonardodalinky/fpsample*, original-date: 2023-09-11T07:34:25Z, Dec. 2024. [Online]. Available: https://github.com/leonardodalinky/fpsample (visited on 12/03/2024).

[33] V. SaRoDe, *Vinits5/learning3d*, original-date: 2020-03-21T21:37:11Z, May 2024. [Online]. Available: https://github.com/vinits5/learning3d (visited on 06/02/2024).

[34] K. Simek, *Dissecting the Camera Matrix, Part 3: The Intrinsic Matrix ←*, Aug. 2013. [Online]. Available: https://ksimek.github.io/2013/08/13/intrinsic/ (visited on 12/03/2024).

[35] K. Simek, *Dissecting the Camera Matrix, Part 2: The Extrinsic Matrix ←*, Aug. 2012. [Online]. Available: https://ksimek.github.io/2012/08/22/extrinsic/ (visited on 12/03/2024).

[36] GeoComp-SZTAKI, *Sztaki-geocomp/Multi-view-3d-completion*, original-date: 2022-04-12T09:33:27Z, Apr. 2023. [Online]. Available: https://github.com/sztaki-geocomp/Multi-view-3d-completion (visited on 12/03/2024).

[37] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv:1502.03167 [cs], Mar. 2015. DOI: 10.48550/arXiv.1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167 (visited on 12/03/2024).

[38] D. Ulyanov, A. Vedaldi, and V. Lempitsky, *Instance Normalization: The Missing Ingredient for Fast Stylization*, arXiv:1607.08022 [cs], Nov. 2017. DOI: 10.48550/arXiv.1607.08022. [Online]. Available: http://arxiv.org/abs/1607.08022 (visited on 12/03/2024).

[39] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," en, in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ISSN: 1938-7228, JMLR Workshop and Conference Proceedings, Mar. 2010, pp. 249–256. [Online]. Available: https://proceedings.mlr.press/v9/glorot10a.html (visited on 12/03/2024).

[40] K. He, X. Zhang, S. Ren, and J. Sun, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, arXiv:1502.01852 [cs], Feb.

2015. DOI: 10.48550/arXiv.1502.01852. [Online]. Available: http://arxiv.org/abs/1502.01852 (visited on 12/03/2024).

[41] K. He, X. Zhang, S. Ren, and J. Sun, *Deep Residual Learning for Image Recognition*, arXiv:1512.03385 [cs], Dec. 2015. DOI: 10.48550/arXiv.1512.03385. [Online]. Available: http://arxiv.org/abs/1512.03385 (visited on 12/03/2024).

[42] Y. Zhang, Z. Gan, K. Fan, *et al.*, *Adversarial Feature Matching for Text Generation*, arXiv:1706.03850 [stat], Nov. 2017. DOI: 10.48550/arXiv.1706.03850. [Online]. Available: http://arxiv.org/abs/1706.03850 (visited on 12/04/2024).

Appendix A

Appendix

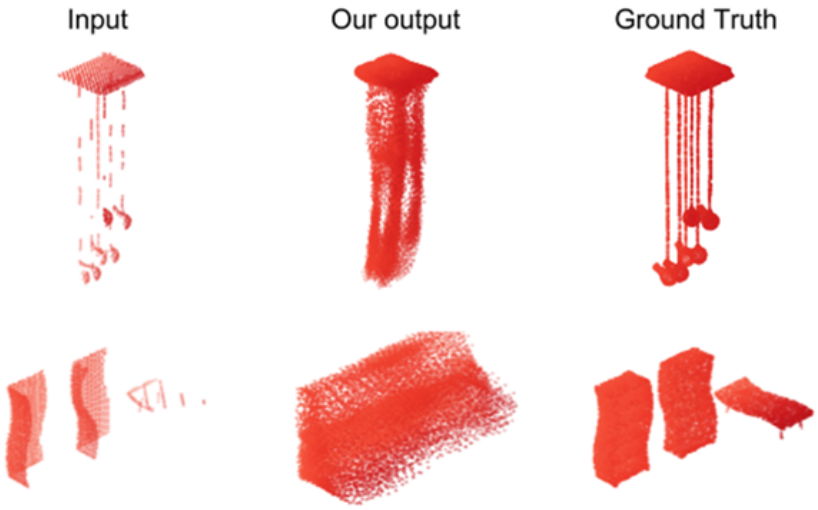

Figure A.1: Scenarios with multiple objects and thin structures. For these examples PCN has a hard time of reconstructing the original shapes. Source: [7]

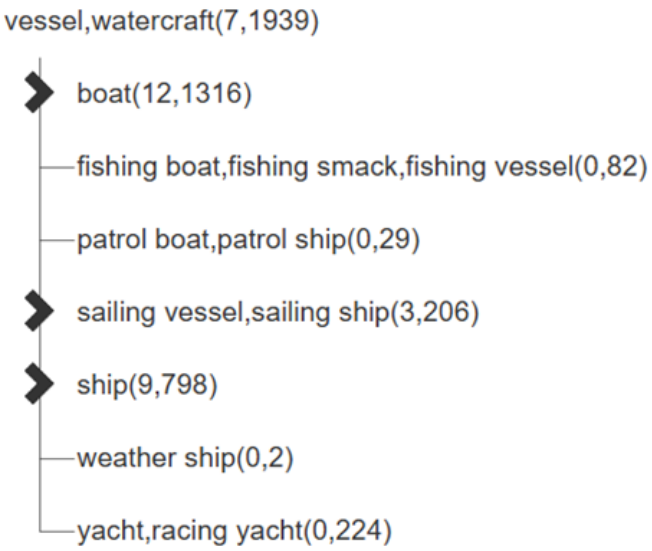

Figure A.2: This list shows the diversity of Shapenet's taxonomy of the common category watercraft. Source: [17]

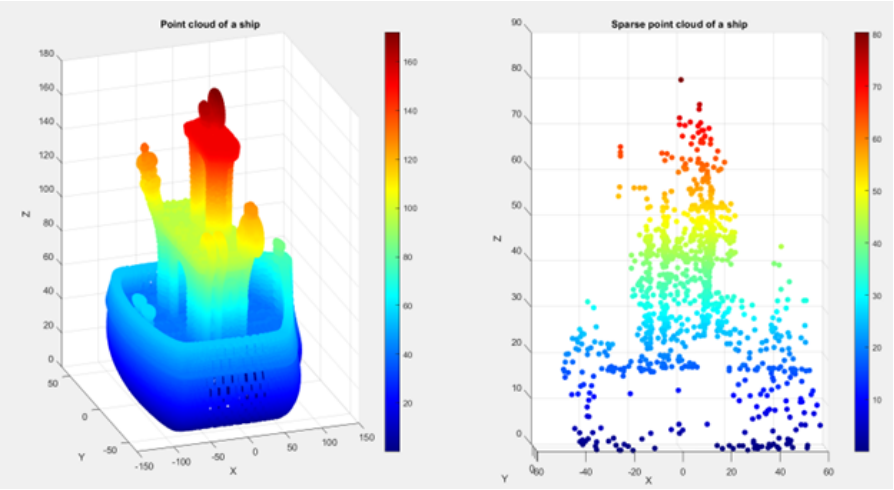

Figure A.3: On the **left** the original point cloud of the ship can be seen, while on the **right** the newly created sparse point cloud is shown with some points having an offset value.

Method	*airplane*	*cabinet*	*car*	*chair*	*lamp*	*sofa*	*table*	*watercraft*	*bed*	*bench*	*bookshelf*	*bus*	*guitar*	*motorbike*	*pistol*	*skateboard*	Avg.
PCN [30]	2.95	4.13	3.04	7.07	14.93	5.56	7.06	6.08	12.72	5.73	6.91	2.46	1.02	3.53	3.28	2.99	6.02
TopNet [21]	2.72	4.25	3.40	7.95	17.01	6.04	7.42	6.04	11.60	5.62	8.22	2.37	1.33	3.90	3.97	2.09	6.36
MSN [14]	2.07	3.82	2.76	6.21	12.72	4.74	5.32	4.80	9.93	3.89	5.85	2.12	0.69	2.48	2.91	1.58	4.90
Wang et. al. [23]	1.59	3.64	2.60	5.24	9.02	4.42	5.45	4.26	9.56	3.67	5.34	2.23	0.79	2.23	2.86	2.13	4.30
ECG [15]	1.41	3.44	2.36	4.58	6.95	3.81	4.27	3.38	7.46	3.10	4.82	1.99	0.59	2.05	2.31	1.66	3.58
GRNet [28]	1.61	4.66	3.10	4.72	5.66	4.61	4.85	3.53	7.82	2.96	4.58	2.97	1.28	2.24	2.11	1.61	3.87
NSFA [31]	1.51	4.24	2.75	4.68	6.04	4.29	4.84	3.02	7.93	3.87	5.99	2.21	0.78	1.73	2.04	2.14	3.77
VRCNet (Ours)	**1.15**	**3.20**	**2.14**	**3.58**	**5.57**	**3.58**	**4.17**	**2.47**	**6.90**	**2.76**	**3.45**	**1.78**	**0.59**	**1.52**	**1.83**	**1.57**	**3.06**

Figure A.4: This table compares results of different methods including PCN, ECG and VRCNet on different categories. The results are Chamfer Distances multiplied with a constant of 10^4. Source: [12]

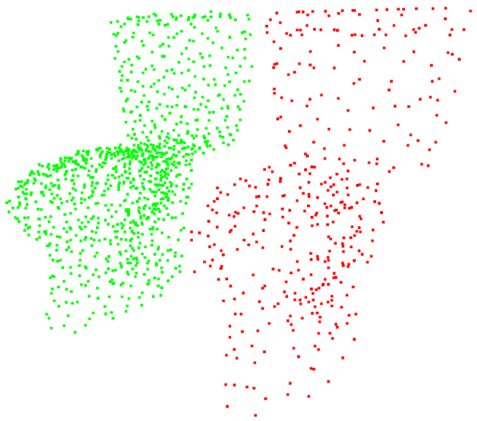

Figure A.5: This Figure shows the 400 points input for the toilet and generated point cloud from it. The red point cloud is the input and the green is the output.

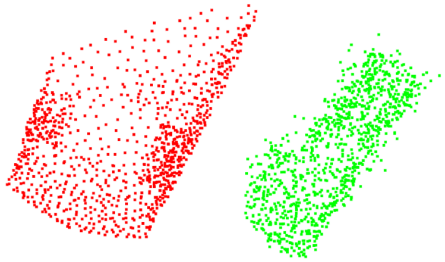

Figure A.6: Partial point cloud of a car. The red point cloud is the input and the green point cloud is the generated output.

Figure A.7: Partial point cloud of a toilet. The red point cloud is the input and the green point cloud is the generated output.

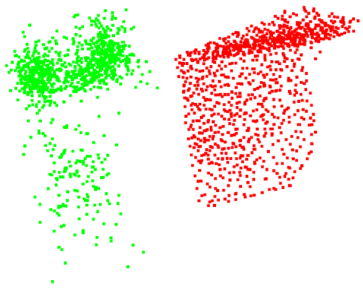

Figure A.8: Partial point cloud of a table. The red point cloud is the input and the green point cloud is the generated output.

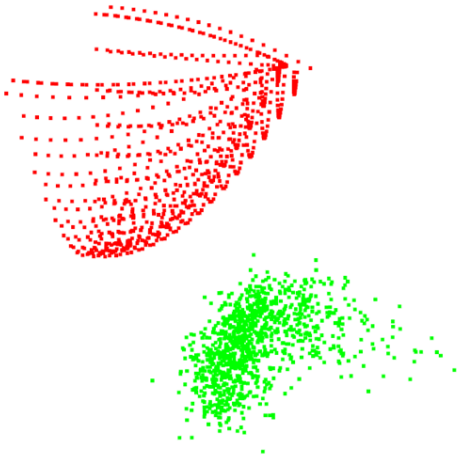

Figure A.9: Reconstruction of a partial input. The input is red and the generated point cloud is green. The green result is sideways.

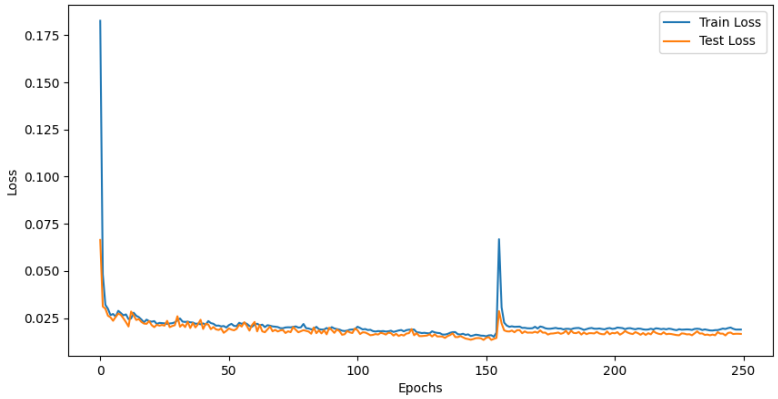

Figure A.10: Training results for sampled input point clouds sized 1024, through 250 epochs.

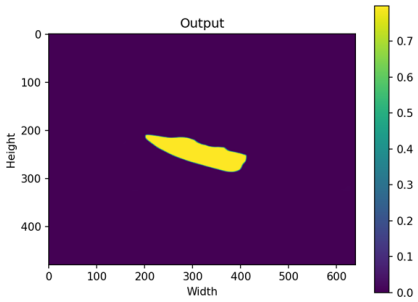

Figure A.11: Output projection of the first model for ship **65**.

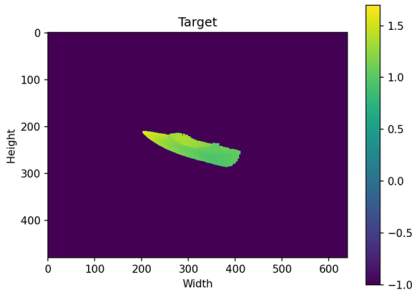

Figure A.12: Projection of a target object with index **65**.

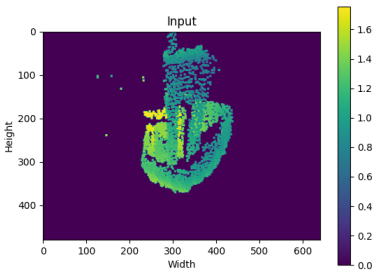

Figure A.13: One of the projection of the **real-world point cloud** input.

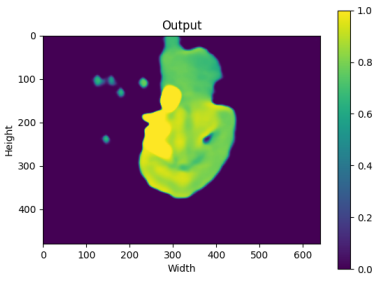

Figure A.14: One of the output of the **real-world point cloud** input projection.

Figure A.15: Prediction of model with not pruned inputs after 1 epoch for object indexed **65**.

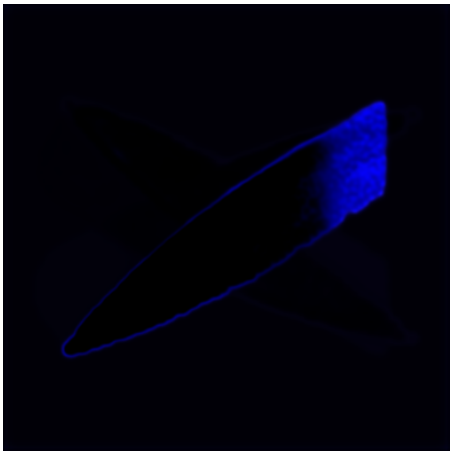

Figure A.16: Prediction of model with not pruned inputs after 10 epoch for object indexed **65**.

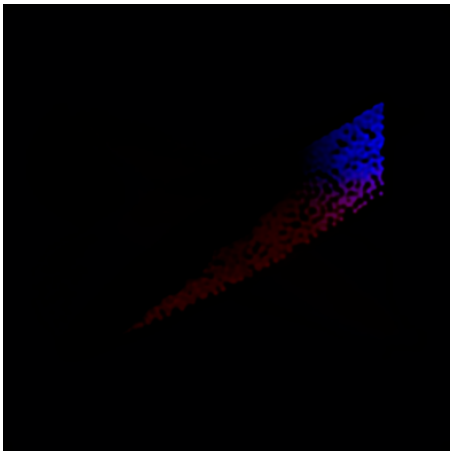

Figure A.17: Prediction of model with not pruned inputs after 50 epoch for object indexed **65**.

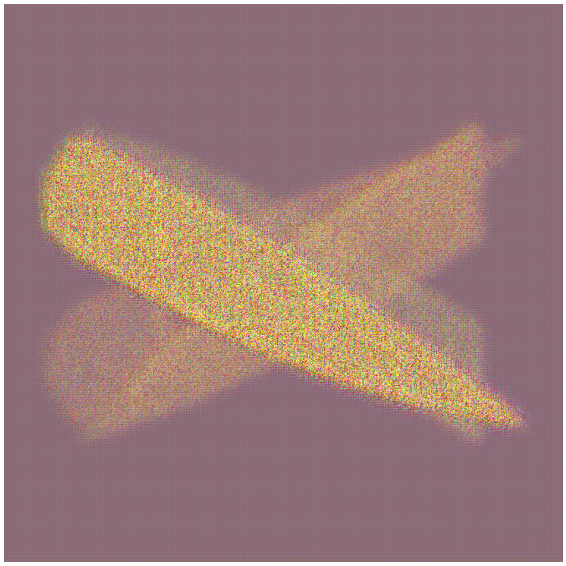

Figure A.18: This figure shown the predicted output for input of object **65**, after the model trained only with adversarial loss.

Figure A.19: Prediction of model with ray casting pruning inputs after 1 epoch for object indexed **65**.

Figure A.20: Prediction of model with ray casting pruning inputs after 10 epoch for object indexed **65**.

Figure A.21: Prediction of model with ray casting pruning inputs after 50 epoch for object indexed **65**.

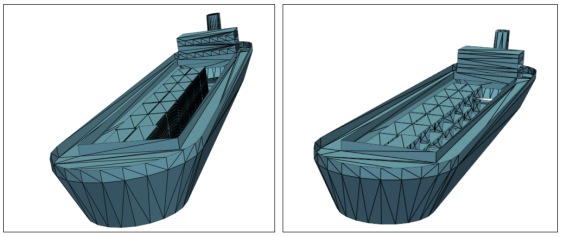

Figure A.22: The simplification process of ship indexed **0** simplified with Quadratic Decimation.

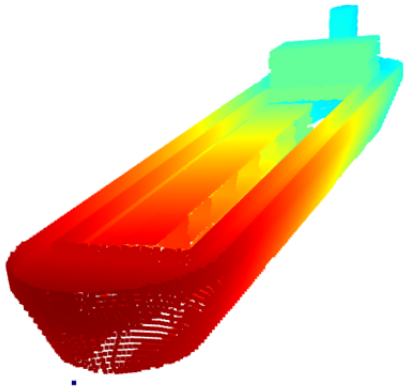

Figure A.23: The results of reprojecting the proejctions created with direct mesh projection. Sample ship indexed **0**.

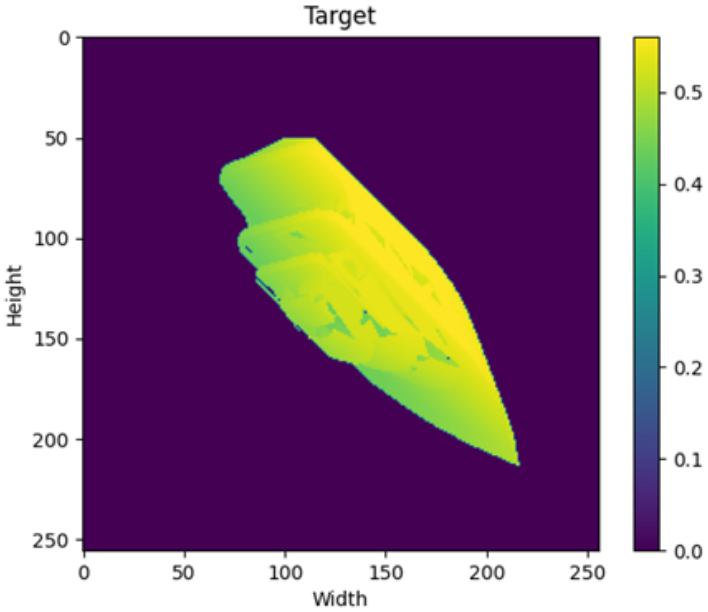

Figure A.24: This is the x plane of the target object's projection from a given viewpoint. The object is indexed **66**.

Appendix B

Code

All of my codes that I have written throughout my thesis work can be found in my GitHub repository: https://github.com/csataridominik/ThesisWork.