Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Super Sampling of 3D Scans using Deep Learning

Master thesis

2023
Bc. Martin Melicherčík

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Super Sampling of 3D Scans using Deep Learning

Master thesis

Study programme:	Computer Science
Field of study:	Computer Science
Department:	Department of Applied Informatics
Supervisor:	doc. RNDr. Martin Madaras, PhD.
Consultant:	Mgr. Lukáš Gajdošech

Bratislava, 2023
Bc. Martin Melicherčík

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Martin Melicherčík
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Super Sampling of 3D Scans using Deep Learning

Annotation: Structured-light scanners work in a way projecting patterns encoding spatial indices of the scene in projector space, capturing the projected scene by a camera and processing the captured images in order to compute per-pixel depth. The processing is a sequential pipeline composed of several steps where each step is working on an image with predefined resolution. The time needed for the processing of a frame is dependent on the resolution of the processed images. Deep Learning Super Sampling is a technique proposed by Nvidia where a neural network is trained for upscaling rendered images. We would like to propose training of a neural network for upscaling computed point clouds that are output from the 3D scanner pipeline. The network would be trained in a deep learning manner using downscaled images from the pipeline. The trained network can be used for upscaling computed downscaled point clouds and restoring the original resolution while performing the processing pipeline faster. Alternatively, the application could be to upscale the image with the original resolution in order to get outputs with higher resolution. These two use cases can be used to make the camera frequency higher or to make a scanner producing point clouds with higher resolution.

Aim: Study relevant papers concerning the structured light scanners, convolutional neural networks (U-net, encoder-decoder, etc.), deep learning super sampling, and deep learning guided filter
Modify the pipeline of structured light scanner/camera in order to lower the resolution of the processing pipeline
Create the dataset composed of two sets of images (lower, higher resolution) that can be used for deep learning
Train the network and use it for upscaling of the resulting point clouds
Propose metrics for evaluation, and focus on quantitative evaluation of scanning speed, scan quality and thesis writing

Literature: Fast End-to-End Trainable Guided Filter, Huikai Wu, Shuai Zheng, Junge Zhang, Kaiqi Huang, 2019, arXiv:1803.05619

Deep Learning Techniques for Super-Resolution in Video Games, Alexander Watson
Department of Computing and Informatics, 2020, arxiv.org/abs/2012.09810

Keywords: Super Sampling, Deep Learning, 3D Scans, DLSS

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

Supervisor:	RNDr. Martin Madaras, PhD.
Consultant:	Mgr. Lukáš Gajdošech
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	11.10.2021
Approved:	13.10.2021

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Martin Melicherčík
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Super Sampling of 3D Scans using Deep Learning
Podvzorkovanie 3D skenov pomocou hlbokého učenia

Anotácia: Skenovanie za pomoci štruktúrovaného svetla je založené na projektovaní vzorov kódujucích priestorové indexy v scéne v priestore projektora, a táto scéna s indexami je snímaná kamerou, kde sú snímky s indexami spracúvané za účelom výpočtu hĺbky jednotlivých bodov obrazu. Spracovanie snímku je sekvencia operácii, kde každá operácia pracuje na obrázku s preddefinovaným rozlíšením. Časová náročnosť na spracovanie snímku je závislá na rozlíšení spracovaných obrázkov. DLSS je technika navrhnutá Nvidiou, kde je neurónová sieť natrénovaná na zvýšenie rozlíšenia renderovaných obrázkov. Chceli by sme navrhnúť natrénovanie neurónovej siete pre zvýšenie rozlíšenia vypočítaných mračien bodov, ktoré sú výstupom z 3D skenerov. Sieť bude trénovaná prístupmi hlbokého učenia s použitím snímkov z kamery so zníženým rozlíšením. Natrénovaná sieť môže byť použitá na zvýšenie rozlíšenia výsledného mračna bodov, ktorý bol vypočítaný zo snímok so zníženým rozlíšením, vďaka čomu by sme mohli urýchliť celý výpočet. Alternatívne, môžeme sieť použiť na zvýšenie rozlíšenia mračien bodov s pôvodným rozlíšením. Tieto dva prístupy môžu byť použité na zvýšenie frekvencie spracovania kamery alebo na produkciu mračien bodov vo vyššom rozlíšení.

Cieľ: Naštudovať relevatné články k skenovaniu pomocou štruktúrovaného svetla, konvolučným neurónovým sieťam (U-net, encoder-decoder, etc.), podvzorkovaniu pomocou hlbokého učenia a hlbokého filtra za pomoci riadiacej textúry
Modifikovať výpočet skenera/kamery tak, aby používal v jednotlivých krokoch obrázky s nižším rozlíšením
Vytvoriť dataset zložený z dvoch množín snímkov (nižšie a vyššie rozlíšenie), ktoré môžu byť použité na hlboké učenie
Natrénovať sieť na zvýšenie rozlíšenia výsledného mračna bodov
Navrhnutie metriky na evaluáciu, a zameranie sa na kvantitatívne vyhodnotenie rýchlosti skenovania, kvality skenov a písanie práce

Literatúra: Fast End-to-End Trainable Guided Filter, Huikai Wu, Shuai Zheng, Junge Zhang, Kaiqi Huang, 2019, arXiv:1803.05619

Deep Learning Techniques for Super-Resolution in Video Games, Alexander Watson
Department of Computing and Informatics, 2020, arxiv.org/abs/2012.09810

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Kľúčové slová: Super Sampling, Deep Learning, 3D Scans, DLSS

Vedúci: RNDr. Martin Madaras, PhD.
Konzultant: Mgr. Lukáš Gajdošech
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 11.10.2021

Dátum schválenia: 13.10.2021 prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgment: I would like to thank to supervisor of this thesis, doc. RNDr. Martin Madaras, PhD. for his willingness to help and his professional advice. Thanks also belong to Mgr. Lukáš Gajdošech for his guidance and consulting ideas.

Abstract

In this thesis, we work on increasing the resolution of depth maps obtained from a 3D camera based on structured light. The aim of the work is to create an output depth map with a higher resolution from a depth map with a lower resolution at the input. In addition to the input depth map, we also have a high-resolution intensity texture of the scene. The depth map super-resolution task is familiar, and several solutions are available based on deep learning models. Deep learning models use intensity textures as auxiliary information. Two models consisting of convolutional neural networks with which we work are FDSR and DKN. We deal with the pre-processing of data from the 3D camera we have chosen so that we can properly train and use the mentioned models. We adjust both models to meet our hardware limitations and make the training process stable. After the data pre-processing and applying the modified trained models, we evaluate the resulting depth maps using qualitative and quantitative metrics focused on the quality of the resulting 3D image. We also evaluate the time efficiency of our solution and its usability.

Keywords: Super Sampling, Deep Learning, 3D Scans, DLSS

Abstrakt

V tejto práci sa zaoberáme zvyšovaním rozlíšenia hĺbkových máp získaných z 3D kamery fungujúcej na princípe štruktúrovaného svetla. Cieľom práce je z hĺbkovej mapy s nižším rozlíšením na vstupe vytvoriť výstupnú hĺbkovú mapu s vyšším rozlíšením. Okrem vstupnej hĺbkovej mapy máme k dispozícii aj intenzitnú textúru scény vo vyššom rozlíšení. Problém zvyšovania rozlíšenia hĺbkových máp je známy a existuje viac prístupov založených na modeloch hlbokého učenia. Modely hlbokého učenia používajú intenzintné textúry ako pomocnú informáciu. Dva modely pozostávajúce z konvolučných neurónových sietí, s ktorými v práci robíme sú FDSR a DKN. Zaoberáme sa aj predspracovaním dát z nami zvolenej 3D kamery, aby sme mohli spomenuté modely správne natrénovať a použiť. Obidva spomenuté modely upravujeme tak, aby spĺňali naše hardvérové obmedzenia a aby bol proces trénovania stabilný. Po úprave dát a aplikovaní modifikovaných natrénovaných modelov vyhodnocujeme výsledné hĺbkové mapy pomocou kvalitatívnych a kvantitatívnych metrík zameraných na kvalitu výsledného 3D obrazu. Vyhodnocujeme aj časovú efektivitu nášho riešenia a jeho použiteľnosť.

Kľúčové slová: Podvzorkovanie, Hlboké učenie, 3D skeny, DLSS

Contents

List of Figures

List of Tables

Introduction

In addition to the standard 2D image-capturing cameras, there exist devices that provide 3D information about the captured scene. In this thesis, we will work with 3D data. Working with 3D data has many applications, such as validating produced components, analyzing archaeological findings, or robotic tasks like mapping and orientation in space. Devices that generate depth maps of scenes compute so-called depth maps, which are 2D matrices whose elements determine the distance of scene points to the camera sensor. Alongside the depth maps, 3D capture devices also provide textures, which are photos of the scenes. From the depth maps, knowing the camera calibration parameters, we can compute a point cloud, which is a set of points in a 3D space coordinate system.

The depth map processing is computationally demanding. We often want to eliminate the noise or smooth the object's surface. For real-time processing of depth maps, we often need GPU resources. The higher the resolution of the depth map, the longer it takes to process it. Many filters applied to the depth maps have time complexity dependent on the number of image pixels. In this thesis, we try to solve this problem by down-sampling the depth map from the device to lower resolution, processing it on the low resolution, and up-sample it back to high resolution. We assume this procedure can take less time than processing the high-resolution sample. Another motivation to make the depth map's resolution higher is simply to synthetically increase the resolution of the low-quality sample. This task is known as the depth map super-resolution task. There are more studies published on this task. Typical approaches use novel deep learning models that learn to up-sample depth maps preserving sharp scene details. In this thesis, we will examine two solutions of this task that propose differently built deep learning models: DKN [11] and FDSR [7]. We will modify these models to work with high-resolution, accurate data from the Photoneo MotionCam device [18]. The mentioned models were trained on a dataset created using a Microsoft Kinect device [13], which provides lower-resolution and less accurate data than ours. We will also propose procedures for the data pre-processing to obtain stable training. Several metrics will evaluate the results achieved by modified models.

In the first chapter, we will state our motivation and the formal definition of the task. The second chapter presents different 3D image capture devices and available datasets for our task. This chapter also introduces our device and the dataset we generated

for our project. The third chapter comprises the deep learning area overview and the related work on which we build our solution. The fourth chapter contains details of our implementation as data pre-processing methods or the training parameters. The last chapter summarizes the achieved results and the metrics used for evaluation.

Chapter 1

Motivation

These days 3D scanning is more and more popular thanks to the fast development of various technologies for capturing the 3D image of scenes. 3D scanners have many applications, such as orientation in space in the field of robotics, product validation for industrial use, analysis of archaeological findings, and others. In our thesis, we work with 3D data captured by the structured-light camera. 3D cameras provide several types of data. The essential data that give us information about the scanned scene are the depth map and texture. A depth map is a 2D matrix whose elements are values that express the distance of a point in a scanned scene from the camera sensor. The texture is a photo of a scanned scene. In our thesis, we will work mainly with the image types generated by the chosen scanner device. This first chapter will introduce the problem we will solve on the 3D data. We will also discuss the motivation for our task and its possible application.

As mentioned above, 3D scanners provide depth maps with corresponding scene textures. These raw data need to be somehow processed. Several filters and procedures can be applied to depth maps to improve quality. There often is some filtering pipeline that pre-processes depth maps for further usage. The time complexity of filters applied on depth maps depends on their resolution. The higher the resolution raw depth map is, the longer it takes to process it by the pipeline. Applying a pipeline on depth maps with lower resolution would speed up the process. However, simple down-sampling of the depth map from the scanner would nullify its advantage of precision. We could down-sample the depth map, apply a filtering pipeline, and then up-sample the filtered depth map back to high resolution while preserving its quality. This idea of up-sampling processed depth maps comes from the video game industry, where game producers want to offer players with less GPU computing power to enjoy high resolution and frame rate experience [25]. Different companies have developed several technologies based on deep learning techniques. Most known is DLSS (Deep Learning Super Sampling) by NVIDIA corporation [15]. This deep learning image up-sampling method can deliver a

4k resolution experience on commonly available gaming equipment. We were inspired by these successfully developed and used models and wanted to define this task under our conditions and chosen data character. To conclude, our task will be to take the input depth map, down-sample it, apply the filters and processing we need, and then up-sample the depth map back to high resolution preserving its sharp edges and surface details. We can use high-resolution intensity texture as additional information about the scene for final up-sampling.

There are several use cases where would depth map super sampling bring improvement. Speeding up real-time 3D image processing would make the whole production process of some products more effective. We could also optimize the process of 3D object fusion by a higher frame rate of the scanning device. Another example could be using a cheaper scanning device that produces lower-resolution depth maps instead of an expensive one that provides higher resolution. One could use a cheap device and artificially up-sample its depth map to high resolution.

1.1 Formal problem definition

Let us define the problem we are solving formally. The input data will be pair D_{HR}, I_{HR} that stands for high-resolution depth map and intensity texture of the scanned scene. Let W, H be the width and height of the input images in pixels. Then the resolution of input images is $H \times W$ pixels. The first sub-task we must define is the depth map down-sampling procedure. In this thesis, we will consider down-sampling as a procedure that gets depth map D_{HR} with resolution $H \times W$ pixels as input and gives depth map D_{LR} with resolution $H/s \times W/s$ where constant $s \in \mathbb{N}$ will be called a down-sample factor. The down-sample factor s determines the size of the $s \times s$ pixels square in D_{HR} that will be compressed to one pixel in D_{LR}. The low-resolution depth map D_{LR} will be input for this thesis's main task: depth map up-sampling. Additional data that can help us transform a low-resolution depth map into a high-resolution one is the high-resolution intensity texture from the scanner. Output of up-sampling procedure should be depth map O that has resolution $H \times W$ (like the input D_{HR}).

We decided to solve the above problem using deep learning techniques. That means we will design a machine-learning-based model that, while the training process, learns how to compute unknown missing pixel values while up-sampling from a low-resolution depth map to a high-resolution depth map. One reason for choosing this approach is an intuitive assumption that the calculations on low-resolution depth maps followed by the up-sampling process using the deep learning model will be faster than performing calculations on high-resolution depth maps. The other reason is that it is a commonly

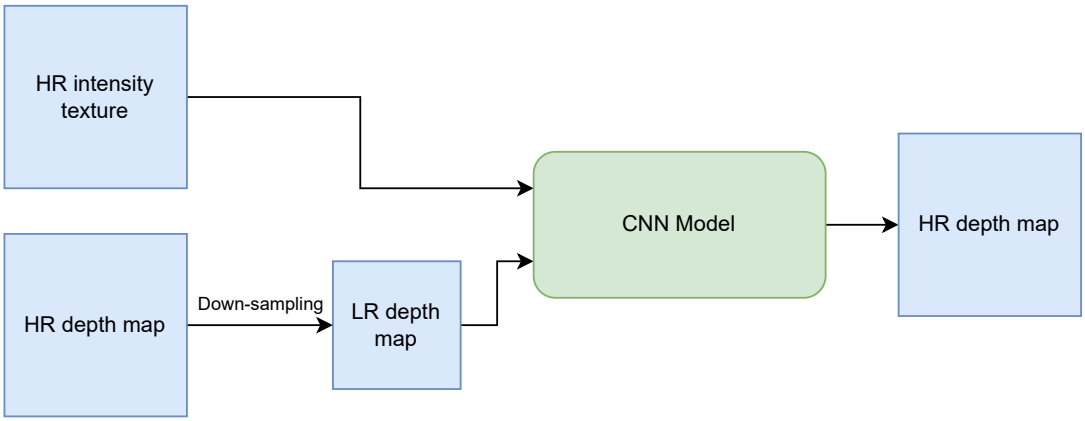


Figure 1.1: High level model diagram of depth map super sampling task solution approach.

used approach for image processing tasks like this one [24] [27]. We will use CNN (Convolutional Neural Network) architecture to solve the task. To visualize the CNN task solution, we present its high-level diagram in Fig. 1.1.

Chapter 2

3D scanning overview

For working with 3D data, we need to know terms and definitions. In this chapter, we will state essential definitions from the field of 3D scanning, examine different known technological approaches to 3D image capture and look at their properties. We will also introduce the 3D scanning device we have chosen to work with and analyze the data that it generates.

2.1 3D capture devices

If we want to define some task on depth maps processing, we should define the properties of the data. These properties are closely connected to the purpose of the task. For example, in some tasks, we can prefer the frame rate of the device producing depth maps over its precision on the scanned object edges. The properties of depth maps are primarily affected by the technological approach used for their calculation. There are several types of depth maps when comparing their creation method. This chapter is based on the Photoneo company survey of 3D capture methods [19]. In the 3D image capture, we distinguish between two main technological approaches: active and passive systems. Passive systems do not use an additional light source to capture the 3D image. The most familiar passive technology is the stereo vision system. In a stereo vision system, we use multiple camera sensors to compute the depth map of the scanned scene. This method has one significant disadvantage, which is dependence on texture. When we, for example, scan one white wall system can not determine how far it is. Active systems, except for camera sensors, use an additional light source to lighten the scene. They can lighten some known projections and compute depth values from their deformation. Active systems' depth map computation process does not depend on scene texture. In this section, we will examine the most familiar types of devices that are active systems. We will analyze the properties of depth maps produced by these devices and discuss tasks for which they are suitable.

Time-of-Flight camera

ToF cameras consist of one sensor and an additional light source, in most cases infrared LED, that enlightens the scene in pulses. The distance of each pixel is computed from the measured time during which a light signal emitted from the light source hits the scanned object and returns to the sensor. An advantage of this technology is the high frame rate. However, the 3D image is noisy, and depth sensors used in ToF devices generally have a small resolution. Another advantage of ToF devices is their calibration process which is quite simple compared to other 3D scanning devices. ToF cameras are suitable for middle-distance scanning tasks without high resolution and require a high frame rate. A good example can be the navigation of a robot in 3D space. The robot needs real-time information about objects in its way, but the precise obstacle surface structure is optional.

Active stereo systems

Active stereo systems use multiple ToF cameras, often combined with regular cameras, for scene texture capture. These devices, like ToF cameras, have a high frame rate and take advantage of combining information from more than one depth sensor, which increases output precision. Active stereo systems are not dependent on texture. The properties of the output depth map are similar to single ToF cameras but can provide a more precise image with an additional texture of the scene. The most familiar and widely used example of an active stereo system is Microsoft Kinect [13]. This device was designed for 3D motion capture, and its primary purpose is to track human gestures for video games and control the system. These tasks are middle-distance tasks that do not require enormous precision. Kinect devices are also trendy in robotics for orientation in space.

Structured light scanners

3D scanners based on structured light technology are devices that enlighten scenes with several known pattern projections and, from their deformation, compute depth maps. The projection light source is often laser which provides high precision. Structured light scanners produce high-resolution depth maps depending on the used camera sensor. Depth maps produced by this type of device contain less amount of noise. They also provide sharper edges of scanned objects and more accurate object surfaces than ToF cameras or active stereo systems. One big disadvantage of structured light technology is that it only works for static scenes because of multiple projection samples that need to be processed by the camera sensor. Tasks that these scanners are good for require high precision and capture static scenes. An example of the task can be a scan of

archaeological findings or product validation in the industrial field.

Parallel structured light scanners

Parallel structure light is one of the latest technologies available on the market. It was developed and patented by Slovak company Photoneo [18]. Its main contribution to 3D scanning is taking advantage of high precision and high frame rate structure light. The depth map capture method is based on laser projections enlightened to the scene and patterns constructed on a custom-design CMOS camera sensor. Photoneo MotionCam-3D [18] is suitable for many tasks. It can be used for dynamic scenes, where we need the high quality scans. The main advantage of this technology is less noise, high precision, and sharp object edges, preserving a high frame rate.

2.2 Data structures

For 3D object representation, we need to define suitable data structures. This section will define the ones we use in our implementation. 3D scanners provide several data types, and the most important for us is the depth map. The depth map is a 2D matrix whose elements are values that express the distance of a point in a scanned scene from the camera sensor. Depending on scanner configuration or manufacturer preset, this value can be in different units. Depth maps can contain not defined pixels. There are areas of the scenes that are not visible to camera or the laser projection does not reach. For these, the device can not compute the depth values, so it marked them as unknown. The most common value for not defined pixels is value 0.

Another data type from the scanner is scene texture. The texture is a photo of the scanned scene. We can obtain a 3-channel RGB or intensity texture, which is a 1-channel grayscale image. Textures are mostly provided in higher resolution than depth maps, and various reasons can cause that. Some datasets are generated using two devices: a camera and a depth sensor. The majority of depth sensors provide lower resolution compared to high-resolution cameras. That can be solved by aligning a low-resolution depth map with a high-resolution texture and cropping the window. Devices that obtain textures and depth maps using one camera sensor also often provide depth maps with lower resolution. That may be solved simply by the downsampling of high-resolution texture.

If we know the calibration parameters of a 3D scanning device, we can compute a so-called structured or organized point cloud. The structured point cloud is a 2D matrix whose elements are coordinates of points in 3D space. When we use this data structure, we work with 3D coordinates of points in space. An advantage we have is the relative placement of points in a 2D grid which can be helpful in many tasks.

There are devices that, with their technology, can not provide structured point clouds. They generate points in 3D space relative to their position. A set of points in 3D space is called a point cloud or an unorganized point cloud.

2.3 Available datasets

There are few available datasets that contain pairs of the depth map and RGB or intensity texture. They differ in depth map creation methods. The most familiar datasets mentioned in papers that we have studied are NYUv2 [14], Middlebury [21], and RGB-D-D [7] datasets. The NYUv2 dataset [14] was created using Microsoft Kinect [13], and its samples are scans from video sequences of interior spaces like kitchens, offices, living rooms, and others. The Middlebury [21] dataset provides several sets of data. Some are generated by stereo vision, others with a structured light scanner. Dataset sample scenes contain various arranged objects with complex surfaces. The last mentioned RGB-D-D [7] dataset was created using two devices: the ToF camera and the classic RGB camera. Textures and depth maps are aligned and have the exact resolution obtained by cropping the window from the texture. RGB-D-D dataset also consists of various scenes from interior spaces like the previous ones.

All three mentioned datasets have one property in common: they are densely defined. Depth maps in these datasets contain only small, not defined holes or areas that are simple to fill in from their surrounding defined pixels. Objects in scenes of datasets created by ToF camera or Kinect do not have very sharp edges, and their samples are generally quite noisy.

Photoneo MotionCam-3D

In this thesis, we have chosen a 3D camera from Photoneo - MotionCam-3D [18] as a device for generating our data. There are more reasons for this choice. Firstly, we have access to this device, and it uses the latest technology for 3D image capture. There are not many publications that work with this type of data. Thus, it is still an unexplored area of research. Depth maps from Photoneo MotionCam-3D are less noisy than previously mentioned familiar datasets and more precise on object edges. Solving tasks on data like this will be an interesting challenge. Tasks that these devices are used for often require high accuracy. For that reason, depth maps contain many more undefined pixels. That is caused by scanners preferring to determine that it is not sure about depth value over giving inaccurate information. One of the biggest challenges we face in this thesis is handling large undefined areas of depth maps. Intensity texture from the scanner camera can provide helpful information about undefined regions. For

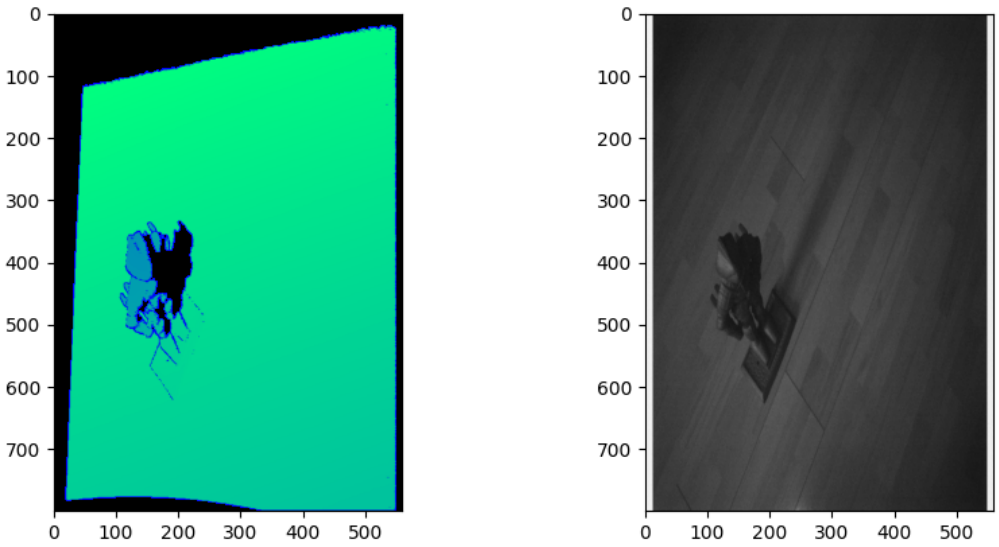


Figure 2.1: Samples from Photoneo MotionCam-3D. Depth map is visualized using RGB spectre. Black pixels are undefined. The defined depth values are mapped linearly to color scale from blue to green color (Left), Intensity texture (Right).

illustration, we present samples from scanner data in Fig. 2.1. We can observe that the depth map contains a lot of undefined (black) pixels. In Fig. 2.2. we demonstrate sharp edges in a 3D point cloud computed from a depth map from Fig. 2.1.

2.3.1 Our dataset

All datasets presented in previous sections contain scenes composed of various objects placed in indoor environments. Our dataset will be generated by the Photoneo MotionCam-3D [18]. Thus we want to create the dataset scenes according to the known

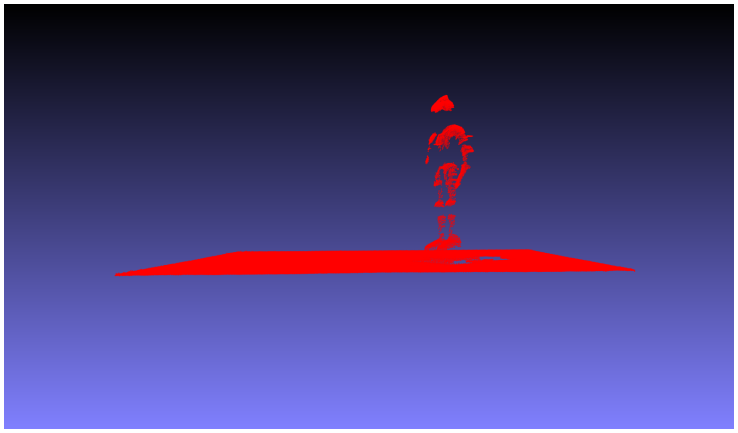


Figure 2.2: 3D point cloud computed from depth map shown in Fig. 2.1.

device's use case. A good example of the Photoneo MotionCam-3D use case can be 3D object fusion. If we consider a 3D object fusion task, we have a set of scenes that contain one object on a flat surface from different views. We try to align and merge them into one 3D object to form its most precise virtual representation. Based on the device we use and its object fusion use case, we generated a dataset comprising scenes composed of one object placed on a flat surface. We used five objects with relatively complex surfaces: A warrior figure, a 3D-printed bear figure, a David statue model, an HDD disc, a rotor, and a cogwheel. We created a 1200 sample-sized dataset using a Photoneo MotionCam-3D. The described type of dataset sample scene is shown in Fig. 2.1.

Chapter 3

Deep learning approach

In the previous chapter, we formally defined the problem we are trying to solve in this thesis and stated that our chosen solution approach would be a deep-learning-based model. In this chapter, we will state some basic definitions from the machine learning area and describe the main concepts of deep learning techniques we have chosen. We will also go through some related work that has been done on the depth map super sampling task. Ultimately, we will present the chosen models and discuss how they should be modified to suit our data and training hardware requirements.

3.1 Overview

In general, machine learning applications include algorithms that parse data, learn from them and apply the acquired knowledge to make a decision or estimate values. The goal of the machine learning problem-solving approach is to design a model (algorithm) that gets large amounts of data and learns to solve defined tasks during the so-called training process. After the training process model should be able to give satisfactorily accurate output for previously unseen input data. The training process of some models consists of giving the model the training dataset sample by sample and evaluating its outputs by proper error function. The model modifies itself after evaluating every output computed on a given sample. We can repeat the training process with the same dataset multiple times in so-called training epochs to improve model output accuracy. There are two main types of machine learning algorithms: supervised learning and unsupervised learning. These two differ in the way that algorithm learns. Dataset samples of the former learning approach contain the so-called ground truth, a known wanted output of the training model. Ground truth can be interpreted as a supervisor's data that shows us what our ideal output should look like. The model computes the error of its output by adequately defining the distance from the ground truth. The latter learning approach does not have ground truth data. The training process can

differ on the definition of the task. Examples of unsupervised learning tasks can be clustering points in space and reinforcement learning, which is a method of learning based on awarding models for reasonable steps and punishing for bad ones. In our thesis, we will use supervised learning. The reason is apparent. The ground truth in our task will be a high-resolution depth map. Several types of algorithms use supervised learning. We will naturally use ones that are appropriate for image-processing tasks. They will be described in the following sections.

Deep learning

Deep learning is a subarea of machine learning. Deep learning models try to learn in a way that is inspired by the human brain. Their architecture is based on the neural network concept. In the human brain, there are neurons that are connected by synapses. They create network structures that use electric pulses to activate individual neurons. By activating the right neurons, some muscles can be moved, the task can be solved, and a decision can be made. Neural networks in computer science try to use this concept to create a mathematical model that learns this way to provide correct output. Neural networks in artificial intelligence are networks of nodes - neurons that are variously interconnected. Neurons are arranged in layers. In general, the neural network has an input, hidden, and output layer. The deep learning models have more hidden layers. The "deeper" the network is, the more computationally demanding it is to train the model. There is no neural network that is complex as a human brain. Deep learning models are designed for specific tasks. The architecture of these models varies on the types of layers, neuron activation functions, connections between layers, and output evaluation while training. The training process of deep neural networks consists of modifying weights and filters that neurons apply to given input data. Each training data sample model evaluates its output precision using the so-called loss function. It back-propagates the computed error through all network layers when determining how far it is from the wanted output. The neurons modify themselves according to this error and are ready to process the following training sample. The goal of training is to reduce the error as much as possible. After the training model should be able to provide satisfactorily accurate output for previously unseen input data. Several types of neural networks depend on the data they work with. In our thesis, we will use convolutional neural networks that are designed to work with image data. They are widely used for image processing tasks we try to solve. In the next section, we will describe this concept and try to fit it into our task.

3.2 Image processing

Image processing tasks are defined on image input data which are 2D matrices of pixel values. The model output type differs depending on the task. It can be a number, text, or output image. When working with images, we need to use appropriate types of layers in the deep network. The most common layer type for processing the image data is the convolution layer. The convolution layer applies the operation of convolution on the image matrix. The mentioned operation slides the kernel window (filter) along the input 2D matrix. These filters are modified during the training process. In one convolution layer, we can apply several filters on the image. After convolution, we often apply the activation function on every element of the layer output. The activation function decides whether the neuron (element of 2D output) should be activated. Deep networks that contain convolution layers are called convolutional neural networks (CNN). These networks can also contain other layers that work with 2D input and produce 2D output. For example, a pooling layer slides the window along the 2D input. According to the defined condition, one pixel will be put into the layer output for every position. The pooling layer down-samples the input data. Many other layers can be used in CNN architecture for feature extraction, down-sampling, and data concatenation. Layers in CNN models can be variously interconnected. We can fork or join some network branches by combining their outputs using mathematical operations.

In our thesis, we will use a convolutional neural network that inputs low-resolution depth maps and gives us up-sampled high-resolution depth maps as output. We can use the high-resolution intensity texture of the scanned scene as an additional input which our CNN will use for enhancing edges or surface structure precision. In the next section, we will discuss concrete CNN model architecture design and the related work on which we can build our solution.

3.3 CNN model architecture

The designing of CNN architecture includes determining which layers we will use and how the layer's inputs and outputs will be connected. The depth map super-resolution task defined in the previous chapter is a common problem, and several CNN models are available that try to solve it using different architectural approaches. For designing CNN architecture, we need to know the input sample format. Most recent CNN-model-based solutions for super sampling tasks assume that each depth map has its corresponding intensity or RGB texture in a few cases. In general, we will assume that the training input sample for our CNN model will be a triplet of the low-resolution depth map, HR intensity texture, and the high-resolution depth map used as ground truth. The first two mentioned images are inputs from which the model computes the output image

that can be compared to ground truth and evaluated. When we know the input sample format, we can build our architecture on it. We have studied several publications that propose different CNN models [7] [11] [26] [27] for depth map super-resolution. All of the studied architectures have a few concepts in common. We will introduce these concepts in the following sections and discuss their properties.

High-level architecture

The high-level architecture of all published architectures we have studied has a similar structure determined by the two input images. Models generally have two main branches composed of convolutional and other layers: the depth map and texture branches. These two branches interact with each other at specific network depths. It can be that image from the texture branch is element-wisely added to an image in the depth branch, the images are concatenated, or another merging method is used. The method of merging the branches varies for different models. The idea of merging the texture branch with the depth branch is based on the fact that essential image details like edges and surface structure are present in the high-resolution texture image. We want to correct inexact values of unknown depth values using known texture image pixel values.

Up-sampling layer

All models have a common property, the so-called up-sampling layer [24]. The up-sampling layer is a network layer located in the depth branch responsible for the concrete expanding procedure applied to the depth image. The up-sampling layer's output has a higher resolution than its input image. The CNN model architecture is, in fact, strongly affected by the type of chosen up-sampling method. Some models do simple deterministic up-sampling of depth images in the first layers of the network and then solve super-resolution task by correcting inexact expanded depth values. Another approach provides a model that does the opposite. That is the up-sampling layer as one of the very last network layers. The whole deep network learns to prepare low-resolution images for the following expanding procedure. Some models do so-called continuous up-sampling, meaning they do up-sampling multiple times at different network depths. This approach is often used when the training model is designed for multiple possible factors of up-sampling. Models are usually trained for specific constant up-sampling factors. An example of this can be just a super sampling of 2 times smaller resolution to the high-resolution one. The position and frequency with which the up-sampling layer is used in the deep network are strongly connected to the specific up-sampling method. In general, two types of methods are used in recently published models: ML (machine learning) based and algorithmic.

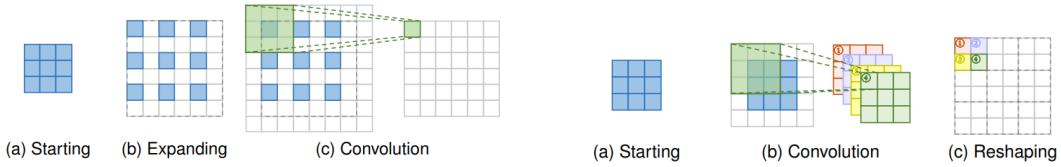


Figure 3.1: Deconvoluton (left) and Pixel shuffle (right) up-sampling methods [24].

Models with ML-based up-sampling layers have trainable parameters. They use parameters that modify themselves during the training process and try to up-sample the depth map suitably for further values refinement in the concrete network. The most familiar ML-based up-sampling methods are deconvolution (or transposed convolution) and pixel shuffle [24]. The deconvolution is the inverse of the convolution operation. It increases the image resolution by evenly expanding the image by inserting zeros between pixels and performing convolution along the expanded image. The pixel shuffle up-sampling method applies convolution with a proper number of filters that determines the up-sampling factor. The number of filters is the block size that will replace one pixel from low-resolution when up-sampling to high-resolution. To better understand the concepts, deconvolution and pixel shuffle are visualized in Fig. 3.1.

The second type of up-sampling is an algorithmic method. These methods do regular deterministic operations on images. We can mention bi-linear interpolation, bi-cubic interpolation, and the nearest neighbor method. These are used the by the majority of recent models. Bi-linear and bi-cubic interpolation are simply image-expanding methods that use the distance-weighted average of the four nearest (for bi-linear) or nine nearest (for bi-cubic) pixel values to estimate a new pixel value. The nearest neighbor method takes the exact value of the nearest available pixel for every unknown pixel after the expanding image procedure. These methods have no trainable parameters.

To conclude, when designing the CNN model for the depth map super-resolution task, we have to answer the high-level questions. The first one is where in our network will the up-sampling layer be placed: first layers, continuous or last layers. The second question is what method will be used for expanding the depth image in the up-sampling layer: bi-linear, bi-cubic, nearest or ML-based. The selection of the up-sampling method depends on the nature of our data and the task-specific requirements on output depth maps.

Evaluation

In the machine learning overview section, we have described the main concepts of the supervised deep learning area. Evaluation of results is part of the CNN model design process. The model modifies its weights and parameters during the training process to provide more precise results. For proper training, the model needs to determine how

good its actual result is compared to the known ground truth. For every deep learning model, we need to define the evaluation metric that will be used for computing the distance of some achieved result to the ideal one. It is often challenging to create such a metric because of the data nature and its specific properties. The result of our task is a high-resolution depth map image. We need to define a function that takes two depth maps and returns a number that expresses their distance. In machine learning theory, we call functions like this a loss function. The computed error value is for every sample back-propagated through each network layer so it can modify its parameters during training. Most of the models that we have studied in our research use L_1 and L_2 loss function [7] [11] [26]. The former is an absolute mean error of all depth map pixels, and the latter is the mean squared error of depth map pixels. These loss functions work well for most applications but can cause problems when there are some more important areas in the image. We will examine this problem and its possible solutions in further sections.

3.4 Related work

We studied several publications that propose deep-learning-based models for depth map super-resolution task [11] [7] [26]. These models work with the scene's texture as an additional input image. The training process and results evaluation is made on publicly available datasets described in the first chapter. The authors of the most recent publications attribute the most significant importance to the datasets created using the Kinect device [13], as it probably is the most available and familiar product on the market. Kinect is also usable for many robotic real-time tasks. We aim to solve depth map super-resolution task on more precise data from Photoneo MotionCam-3D [18]. These data have different nature, as discussed in the first chapter. We will try to use the most recent and precise models for our data. We expect that the proposed models have to be modified to fit our requirements.

We started our related work research with publications that have their implementation publicly available. At the time of our related work papers studies, we found three publications that met this requirement: FDSR [7], DKN [11], and DCT [26]. The DCT model publication was the most recent and provided survey of older models inference on the most familiar datasets created using the Kinect. According to this survey and the comparison of the results proposed in the article, the FDSR, DKN, and DCT were the most precise models. The accuracy difference between these three architectures was by order of magnitude greater than the older models. Based on these facts, we decided to use the three architectures as the ones on that we will build our task solution. Later as our work progressed, we decided to reduce the triplet of used models to just FDSR

and DKN because we could not put the published DCT model implementation to work and train the model with significant results.

In 2023 Zhong and Liu published a survey containing all depth map super-resolution task solution approaches [27]. They probably contacted many researchers and asked them to publish their implementations for making order in this research area. The new implementations appeared on familiar portals and are now available. Analyzing these newly added implementations and putting them to work goes beyond the scope of this thesis.

In this thesis, we will build on FDSR and DKN architectures. These are based on entirely different ideas. The DKN architecture is based on an image-filtering approach. That means it up-samples the input depth map using some simple method and then tries to create a filter that enhances its details. During the training process, DKN learns how to create a custom filter that will be applied to an inexact expanded depth map. The FDSR approach is different. It is based on continuous residual learning that uses structure from intensity texture for enhancing depth image. At specific network layers, the texture branch merged into the depth branch. The following sections will describe the FDSR and DKN models and discuss their properties.

3.4.1 DKN model

The description of the DKN model in this section is based on its publication text [11]. DKN stands for *Deformable Kernel Network*. This approach to a depth map super-resolution task comes from a joint image filtering idea. The main idea is to expand the image by some simple up-sampling method and then apply a filter to enhance precision. Apply filter means sliding along the image with the kernel window. The kernel window determines the set of neighbors for each pixel. We can compute a more exact value from kernel pixels. Each pixel from the neighborhood determined by the kernel window has a weight contributing to the result pixel value (e.g., a weighted average of neighboring pixels). All weights of kernel pixels should sum to 0. Each pixel's kernel window weight values can be determined by guidance image (in our case, intensity texture) or just from the depth image structure. Familiar examples of filters that do not use machine learning and use this concept are bilateral filter [22] or guided filter [6]. The DKN approach uses deep learning techniques to determine the kernel pixels and their weights. The main difference between DKN and the classic joint image filtering approach is that the filter uses a constant kernel for each image pixel in the joint image. A kernel like this picks the same relatively positioned neighbor pixels of each pixel to compute its filtered value. The DKN model uses the CNN model for picking neighbor pixels of each pixel and their weights. To describe this concept, we provide Fig. 3.2, where we can see the DKN approach in the figure on the right. The kernel comprises pointers to pixels

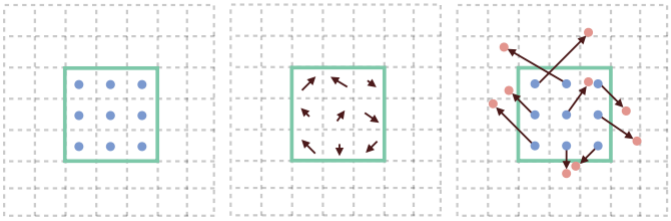


Figure 3.2: Classic joint image filtering kernel (left), learned offsets for neighborhood pixels positions (middle), neighbor pixel locations with their offsets (right) [11].

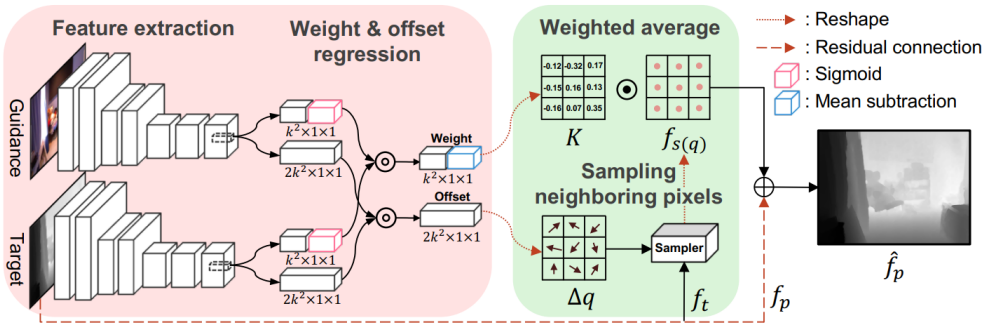


Figure 3.3: DKN model architecture [11].

that the CNN model chose for computing filtered value. The CNN model also computes kernel weights for the chosen pixels. We described the principal concept of the DKN approach and its output which is a weighted kernel for each depth image pixel.

Now we will examine the CNN architecture. As we mentioned in the CNN architecture overview, the DKN model comprises two main branches: texture and depth. The network can be divided into three consecutive parts: feature extraction, weight & offset regression, and weighted average. The three network parts forming the DKN model architecture are visualized in Fig. 3.3.

In the feature extraction part, seven convolutional layers with RELu activation function and batch normalization are applied on both input images. Some of the convolutions down-sample the image, so the result of this network part is $128 \times 1 \times 1$ feature map for receptive field 51×51 pixels. That means the model will pick neighbor pixels from area 51×51 for each pixel and their weights.

The second part of the DKN network is weight & offset regression. This network part is responsible for computing the weights of pixels in the kernel and their offsets. We apply one additional convolution layer for computing weights on outputs from previous network parts. The output of the convolution layer will be $k^2 \times 1 \times 1$ feature map, where k stands for kernel window side size in pixels. The weight values in the kernel should sum to value 0. The default DKN setting is $k = 3$. That means the filtered value of every pixel will be computed using nine pixel values picked from its 51×51 pixels neighborhood window. The offset is also computed by one additional

convolution layer applied to data from the previous network part. The output of offset computing convolution will be $2k^2 \times 1 \times 1$ feature map. The feature map will contain x and y offsets that point to pixel locations in the receptive field 51×51. Both weight and offset feature maps are computed from outputs of the texture branch and depth branch of the network. Outputs of additional convolutions applied separately in the branches are merged by element-wise multiplication of the feature maps providing a final output of this network part.

The weighted average part uses the given kernel with weights and offsets from previous network parts and applies it to the input sample pixels. The offsets are computed fractionally, so the choice of the pixels that will be used in the kernel is estimated by bi-linear interpolation. The result value for every pixel is the dot product of the kernel weights matrix and picked pixels arranged in the kernel size matrix. By default, a residual connection of the input depth image is also used. The input image is element-wisely added to the weighted average image output.

DKN model was trained on NYUv2 dataset [14], which was created using a Kinect device [13]. The depth maps from this dataset are in lower resolution and do not have as sharp edges as our Photoneo data. The model works with a pre-expanded depth image. The low-resolution depth map is firstly expanded by bi-cubic interpolation and then put into the model for further processing. The authors also experimented with the nearest neighbor expanding method, which generally had worse results, so they chose the bi-cubic method as the expanding approach. DKN model uses L_1 loss function that provides evenly distributed attention to all pixels of depth maps when computing sample error.

3.4.2 FDSR model

The description of the FDSR model in this section is based on its publication text [7]. FDSR stands for *Fast Depth Super Resolution*. This approach is based on continuous learning considering high and low-frequency components of input images. High-frequency components of the image are areas that are more important in the image structure manner. Examples of a high-frequency component can be edges or some important object surface structure. The low-frequency components are lesser important. Examples of low-frequency components can be planes or smooth object surface parts. The FDSR model learns to extract high-frequency components of a scanned scene from high-resolution texture and uses this information while learning the depth map up-sampling procedure. FDSR model architecture has two main branches. Authors call them *High-Frequency* branch and *Multi-scale Reconstruction* branch. The high-resolution texture is input for the former, and low-resolution depth map for the latter branch. The architecture is visualized in Fig. 3.4.

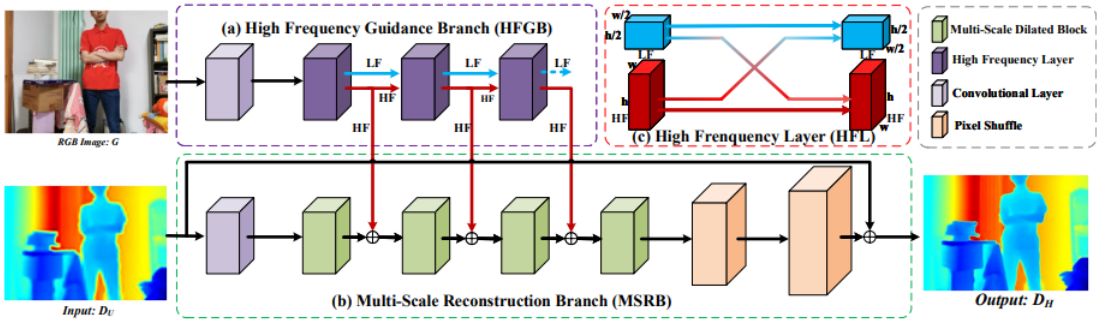


Figure 3.4: FDSR model architecture [7].

The high-Frequency branch starts with feature extraction performed by one convolutional layer. The next part of the high-frequency branch is composed of three consecutive high-frequency layers (HFL) that use the special operation of octave convolution to determine the input image's high-frequency components. HFL separates image input into two channels: low-frequency and high-frequency components. The output high and low-frequency components of one HFL are embedded into the following HFL, so the following HFL generates its current high and low-frequency components using previous ones.

Multi-scale Reconstruction branch starts with convolution for feature extraction. Then four multi-scale dilated blocks (MSDB) follow. MSDB applies two dilated convolutions on a depth image. Dilated convolution is one with a kernel with inflated holes between its elements. During convolution, it skips image pixels according to the constant parameter. For example, dilated convolution can skip every second pixel of the image. MSDB uses two different skipping offsets for exploiting contextual at different receptive fields. After dilated convolutions, one standard convolution is applied to integrate concatenated features. After every MSDB block, the high-frequency features from the corresponding HFL are concatenated to the output. The final fourth MSDB output is up-sampled by the pixel shuffle method applied twice.

At the beginning of depth, the branch input image is split by convolution to multiple channels. After the MSDB blocks are combined with HFL features, the pixel shuffle is applied to merge multiple channels back to high resolution. There is also one residual connection between the input depth map and the up-sampled final depth map. FDSR model like DKN works with the pre-expanded low-resolution depth map. The authors use the bi-cubic interpolation method to prepare the sample for the model. FDSR uses L_1 loss function for error computing. This model was trained on the NYUv2 dataset.

Conclusion

The two models described above were the best-performing models at the time of our related work research. FDSR [7] and DKN [11] are built on entirely different principles. Therefore, it will be interesting to apply them to our data from Photoneo MotionCam-3D [18]. Models were trained on the NYUv2 dataset [14], which is significantly different from our data. The NYUv2 samples are data from a Kinect device [13] that provides lower-resolution depth maps with inexact values compared to the Photoneo MotionCam-3D. Kinect data also contain fewer holes in depth maps. The authors of NYUv2 provide a method that fills the depth map holes using completely defined textures. We tried to apply the implementation [14] of the method published with NYUv2, but we could not put it to work on our data. DKN and FDSR models work with filled depth maps, so we need to design a method for filling them. Another problem that we will be facing is so called edge error that was introduced in the FDSR model article. The authors used a metric that looked at depth values near edges in depth maps and found that it is significantly greater than the error computed by the loss function. For the Kinect data, this is not a big problem because one can not expect big precision of the samples. The frame rate is more important. The depth maps are very noisy, so the edge error is negligible in many applications. The Photoneo 3D scanning use cases often require big precision, and high edge error can cause problems. We will try to solve the outlined and other raised problems in our implementation by experiments and models modifications.

Chapter 4

Implementation

This chapter will describe our solution implementation of a depth map super-resolution task applied to Photoneo MotionCam-3D [18] data. The implementation is the fundamental part of our work on this thesis. As Fig. 1.1 suggests, our implementation includes a few models through which the input data flow. During our research and implementation progress, we ran into several problems based on the nature of the Photoneo data. We will examine our research path, including its dead ends and observations. We will also describe our results evaluation approach, training process, and fine-tuning of training parameters to fit our needs. The full implementation is in the Python scripting language. We use Numpy citenumpy and PyTorch [17] frameworks. Our implementation is available on GitHub [12].

4.1 Data preparation

The only data that is provided to us are intensity texture and depth map of the scanned scene, both in high resolution. For the training depth map super-resolution CNN model, we need a dataset whose samples contain HR intensity texture, LR depth map, and HR depth map as ground truth. The data must also be in a proper format for the model implemented in the PyTorch framework [17]. Dataset samples can also contain additional data that can help us during training. In this section, we will describe the preparation of the dataset and the structure of its samples. We will start with the task requiring low-resolution depth maps that must be included in each sample.

4.1.1 Depth map down-sampling

In our problem's formal definition, we defined the depth map down-sampling procedure as our sub-task. We defined down-sampling determined by scale factor $s \in \mathbb{N}$ as collapsing $s \times s$ square blocks to one pixel. Various deterministic algorithms can do depth map down-sampling. Authors in the majority of publications use methods based

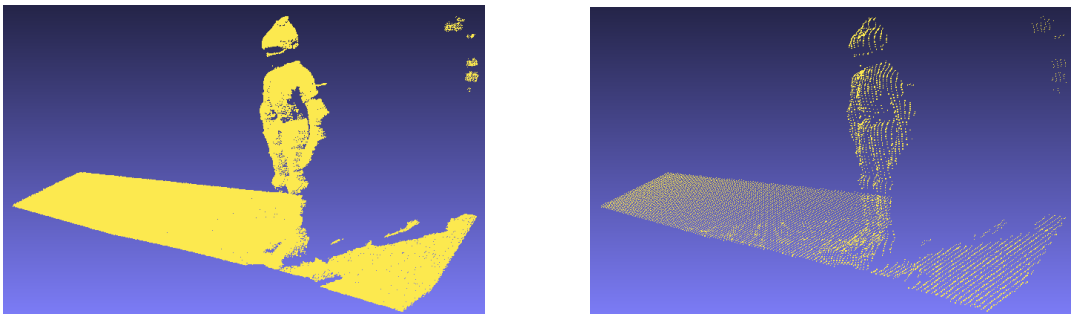


Figure 4.1: Point cloud computed from input HR depth map (left) and Point cloud computed from output LR depth map (right) when scaling factor $s = 4$ used.

on bi-cubic interpolation or the nearest neighbor approach. In the 2016 publication [8], authors found that applying the nearest neighbor on data from structured light scanner devices that provide higher precision is better. For depth maps from ToF devices, they used a method based on bi-cubic interpolation. Inspired by this study, we used a method based on the nearest-neighbor approach. The method [4] divides a high-resolution depth map into $s \times s$ pixels-sized blocks. For every block, we compute minimal value b_{min}, maximal value b_{max}, and (average) mean value b_{mean}. With these values, we do the following: We find out if the given block contains an edge. Whether the block contains an edge depends on the edge threshold parameter ε. If the condition $(b_{max} - b_{min}) < \varepsilon$ is met, the block does not contain any edge. Its values, in some approximation, form a flat surface. If the condition is not met, the block has an edge. We mark pixels with values above the b_{mean} value as foreground values and the others as the background ones. If the given block contains an edge, the result value representing the block in a low-resolution image will be the median of foreground pixels values. If the given block does not contain any edge, the resulting value will be the median of all pixels in the block. We empirically set the $\varepsilon = 0.1$ value by observing method behavior on our input data.

We also tried other, more familiar methods of image down-sampling, but the one described above works the best for our data. The bi-cubic-interpolation-based methods blur sharp edges that our precise Photoneo MotionCam-3D [18] provides. The nearest neighbor simple methods like picking a random pixel from the given block or taking the maximum pixel value can work well when a scale factor is a small number. When choosing the block-representing value of 4×4 or more pixels sized block, it is better to use a more sophisticated method.

We provide Fig. 4.1 to demonstrate the implemented down-sampling method. We computed 3D point clouds from the method input high resolution and output low-resolution depth map when scaling factor $s = 4$ was used. We can observe that the depth map lost its density but not the sharpness of the object edges.

4.1.2 Depth map filling

Both models that we want to use assume that depth maps do not contain any holes. That means all pixels in depth maps should be defined and contain meaningful, valid depth values. Depth maps from the Photoneo MotionCam-3D [18] contain a large number of undefined pixels. The Preset of the Photoneo devices often ignore big distance value because of potential inexact values. When using these data, we need to treat the undefined areas of depth image somehow. The first idea that comes to mind when solving a problem like this is to ignore the undefined regions during training. We could compute the sample error by applying the loss function on pixels from defined regions of the depth image. We tried this and found there was a problem with this approach. When applying convolution operations on the depth image, the sliding kernel treated values of undefined regions as normal. Undefined pixel located within kernel diameter distance from the defined ones influenced their values. The undefined pixels were not restricted or limited by the learning process, so their depth values randomly oscillated during the training. Based on this observation, we included the undefined pixels in the training process. The undefined pixels have all zero values. If we want to incorporate them into the learning process, we need to fill them with meaningful depth values that will goodly influence the estimation of defined pixel values. For this, we propose a filling procedure that gets a depth map with undefined regions as input and provides a filled depth map as the output. The dataset sample will then comprise a filled low-resolution depth map, high-resolution intensity texture, and filled high-resolution depth map as ground truth. The most important depth values of the CNN model output depth map will be the defined regions.

During the training process, evaluation, or providing results of the CNN model, we must preserve information about every sample's defined and undefined pixels. We introduce a new image data structure into all dataset samples for this, and we call it a definition map. The definition map is a 2D binary matrix with the dimensions of a high-resolution depth map. Its elements contain value 0 if the pixel at the same position in the high-resolution depth map is not defined and value one if the corresponding depth map pixel is defined. During the training process depth map will be variously modified. The definition map provides us with stable information about defined regions that will be the most important regions of the final model output. The definition map is pre-computed during dataset creation for every sample before the filling procedure.

Filling method

Now we will introduce and describe our simple depth-map-filling method. This procedure is inspired by a more sophisticated one for depth maps filling [2]. The implementation of the method from the publication was not available, so we implemented its simpler

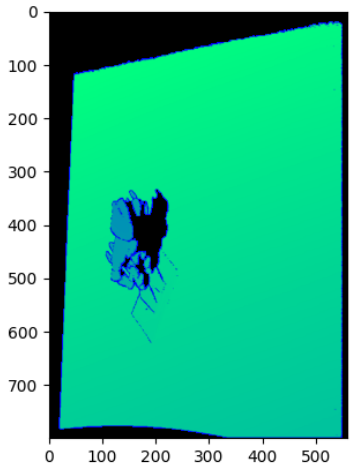


Figure 4.2: Depth map with undefined pixels (black).

version that should fit our needs. In the figure Fig. 4.2, we see a typical input depth map with undefined regions. We can observe that the whole defined scene depth image is surrounded by a big undefined region. This phenomenon repeats in every sample of our dataset. There are also small undefined areas near the warrior figure. These undefined regions are often caused by shadows or glossy surfaces of the scanned object in the scene. Usually, there are more than 50 holes in the depth map. Primarily we divide the undefined regions into two types: undefined background holes and near object holes. We determine the undefined region as a background hole if some of its pixels are adjacent to the image border. Regions with no pixel adjacent to the image border are near object holes. Our filling method treats the two types differently.

The first step of the filling procedure is generating a binary map of holes. We get the input depth map and create a map of the same dimensions where the pixel has a value of 1 when there is an undefined value of 0 in the input depth map. Pixels that are in the input depth map defined will have zero value. On this hole map, we apply the labeling function from scikit-image python library [23]. This method labels every continuous region of 1-valued pixels with a unique number. Output from this step is a map of holes where pixels of each continuous hole have the same value unique for the hole. Then we go through all holes, and for each one, do the following: We look if the border pixels of the image contains pixels with hole label value. If at least one pixel is found, we treat the hole as the background hole and set all its pixels to one the same value. We will comment on the value choice later. If the hole is classified as a near-object hole, we find its defined border pixels. Then we will fill the hole row-wise like this: for every hole row, we find pixels that intersect with the defined border pixels found in the previous step. We take the maximum of these values and fill the whole

hole row. To better understand the described filling procedure, we provide Fig. 4.3.

In the filling method described above, we use the outer border pixels of the hole for its filling. For that, we need to find the pixels first. We propose the method for finding the outer border pixels of a hole as follows: We take one concrete hole map. Pixels of this map with value 1 represent where the uniquely labeled hole is and 0 where it is not. We expand this hole region of ones in the two row-wise directions (right and left) by one pixel. The border pixels of this expanded hole are the outer border pixels of the hole in the depth map image (they all have their defined depth value). The hole region expanding process is done by NumPy roll operation that shifts the input matrix by one pixel in a concrete direction. We apply NumPy roll on the concrete hole map separately in the right and left directions and merge the resulting maps into one that contains a hole expanded by one pixel at the right and left hole borders. For extracting only the border pixels of the expanded hole map, we do the following procedure: We apply cumulative sum operation on the expanded hole map in the right and left directions separately. The pixels of the cumulative sum result images with value 1 (first that are not zero in that concrete direction) are the outer border pixels of the hole in the depth map. The process of finding the outer border pixels of the hole is illustrated on an example in the Fig. 4.4.

Filling results

We proposed the depth map filling method that provides us the whole defined depth image for further processing. This method was designed to fit our requirements. We require filling the holes with meaningful depth values that positively affect defined pixels near the holes during training. The filling method described above is not designed for filling the image with fine details preservation or the most realistic filled object surface. The meaningful depth values mean that the values should be close to reality with some approximation. We aim to keep the depth values of undefined areas stable and close to reality. The values do not need to be precise because, in the end, we set them back to an undefined state. The only values that are of our concern are the defined ones.

In the filling method section, we mentioned that our method fills the background hole with a single value. This value is the maximum depth value of all samples in the dataset. This setting of background hole pixel values forms a plane behind the whole scanned scene, and it gives some reference to the model during training. We simply put a synthetic wall perpendicular to the camera angle behind the scanned scene. To illustrate the filling method results, we propose Fig. 4.5 and Fig. 4.6. Our experiments show that idea of approximate holes filling with the synthetic background keeps the training process more stable, and the impact on the defined values is positive.

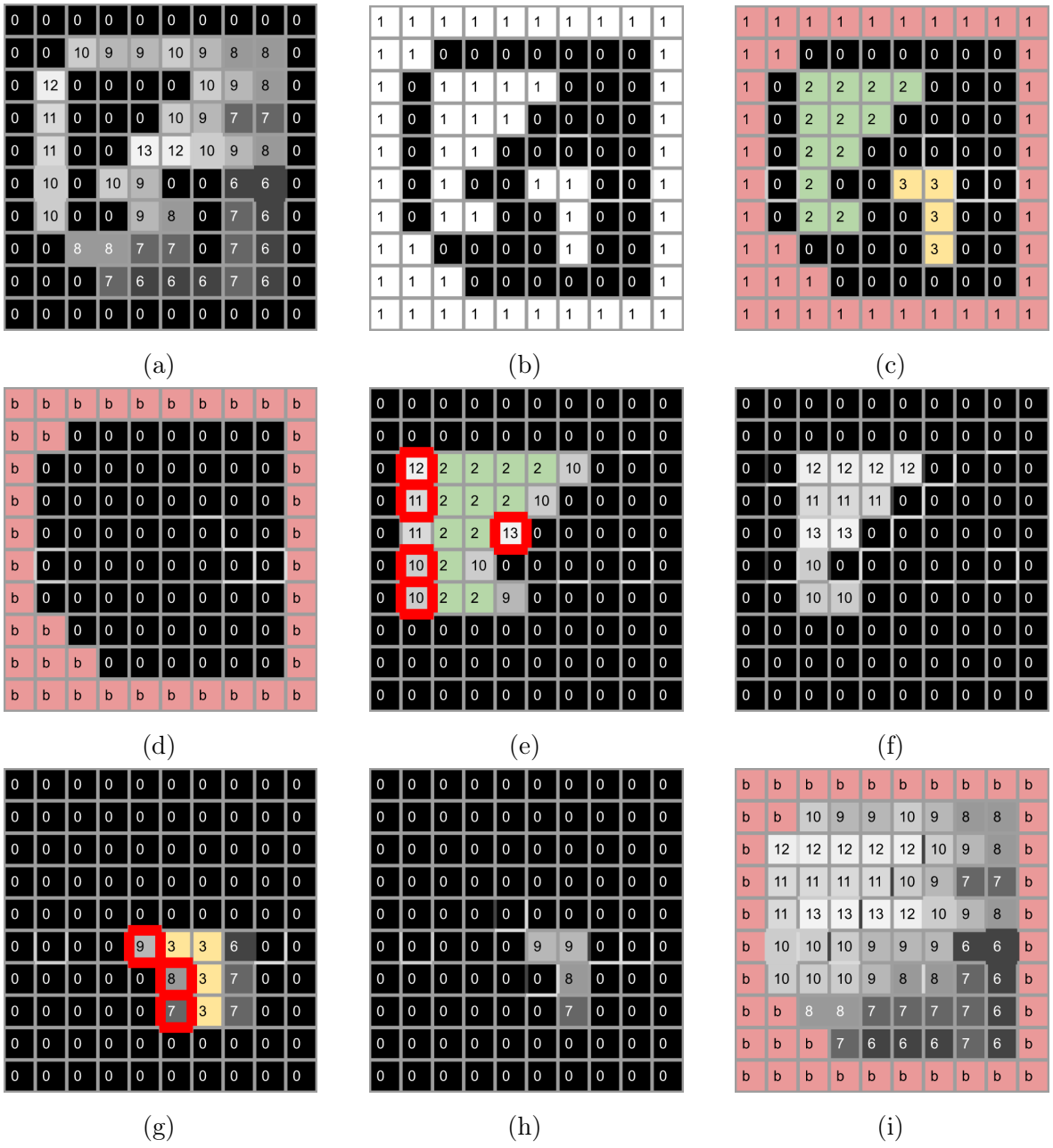


Figure 4.3: From the input (a), we generate hole map (b). Then we label distinct holes with unique labels and get labels map (c). Then we go through all holes, starting with the background hole that is filled with one value (d). Then we go through holes 1 and 2. We determine the defined outer border pixels of the holes and find the maximum value for every row (e), (g). When we have the maximal values for the rows, we fill them row-wise (f), (h). In the end, we merge all filled holes into one whole-defined depth image (i).

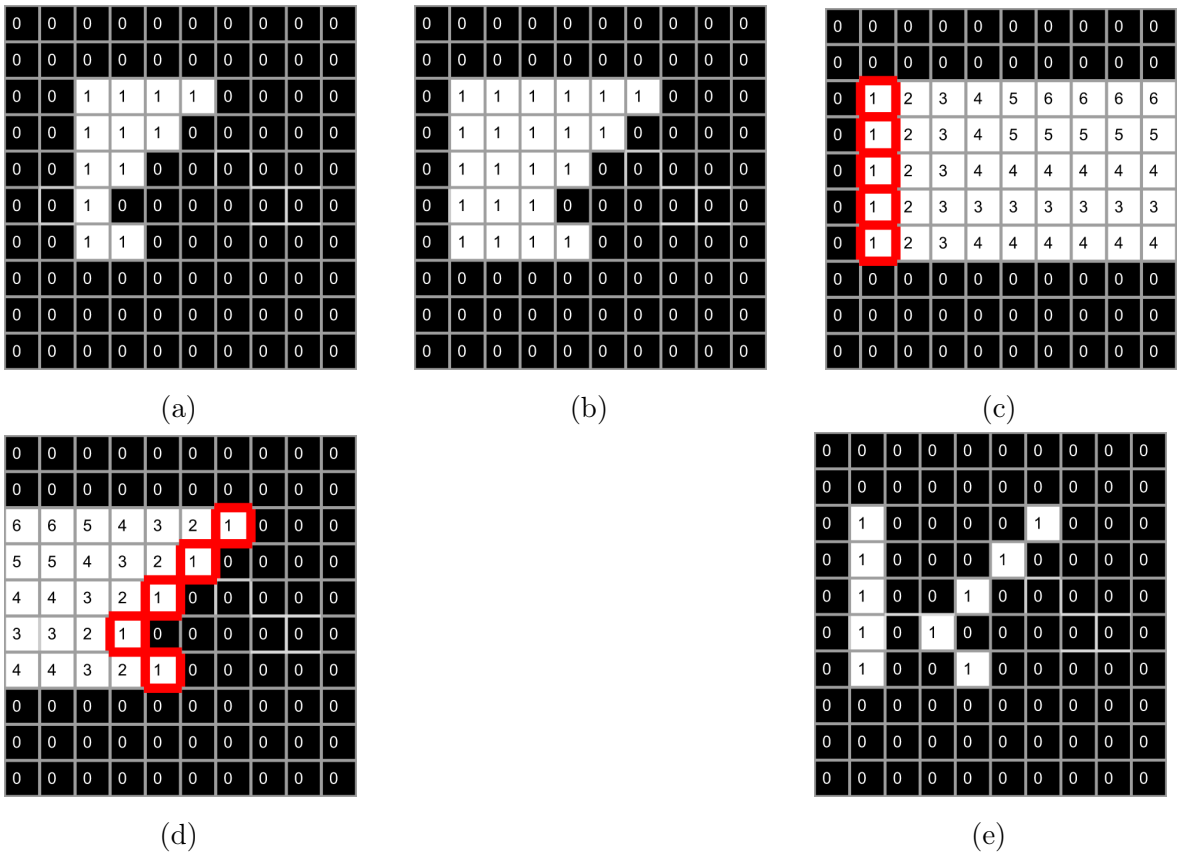


Figure 4.4: On the concrete hole map (a), we apply NumPy roll operation separately in left and right directions and merge resulting maps into expanded hole map (b). Then we separately apply the cumulative sum operation on (b) in the right (c) and left (d) directions. In the end, we merge the 1-valued pixels from the cumulative sum results into a hole row outer border map (e). This map is used in the filling process of depth map illustrated in the Fig. 4.3.

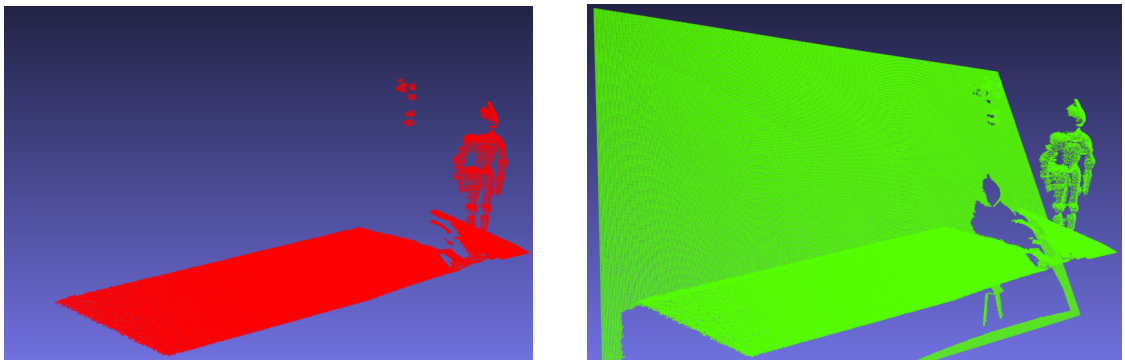


Figure 4.5: 3D point cloud computed from unfilled (a) and filled (b) depth map. We can observe a synthetic plane behind the scene.

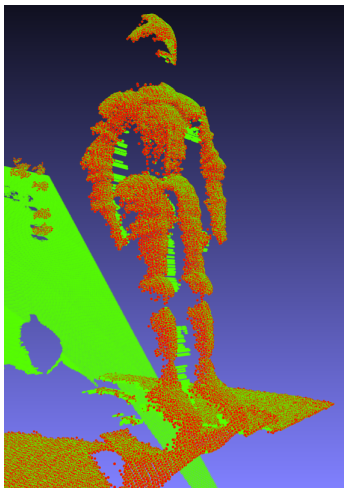


Figure 4.6: 3D point cloud computed from the filled depth map. We can observe row-wise filled near-object holes. Green points are filled, and red are the defined ones.

4.1.3 Texture augmentation

CNN models that we will implement use high-resolution intensity textures for extracting details of the scanned scene, such as edges or surface structures. If we want to use intensity texture as a guidance image, it has to fulfill the task's requirements. There has to be a correlation between the intensity texture and depth map that we are up-sampling. For example, if some edge is not captured in the intensity texture, it would not be preserved precisely in the up-sampled depth map. On the contrary, if there is an edge in the intensity texture and the edge is missing in the depth map, it will also break the correlation of the images. Based on the idea of preserving the correlation between the depth map and intensity texture, we decided to modify the input intensity textures to correlate with the filled depth maps. Our approach is the same as with the depth maps. Firstly, we multiply the intensity texture with the definition map element-wise to create the same undefined areas in the intensity texture as in the depth maps. Then we apply the same filling method to the image. This provides us with the intensity texture that has a bigger correlation in the filled depth values than the not modified one. According to our experiments, this modification of intensity textures improved the training process. The leading cause of this phenomenon is probably the fact that we do not push the model to estimate useless depth values of the undefined regions. It is some kind of simplification of the useless regions of the depth image to avoid the computation of their complicated surface structure values. We modify the intensity textures while creating the dataset so every sample contains only the modified one.

Depth map up-sampling

In the previous chapter, we analyzed the architecture of CNN models designed for depth map super-resolution task. We found out there was an up-sampling layer in all models responsible for simple depth map expansion to high-resolution ones. FDSR and DKN models work with pre-upsampled depth maps. That means the input depth maps provided to the model are high-resolution, and models modify inaccurate depth values estimated by the simple expanding method. Both models we will work with use bi-cubic interpolation for up-sampling input depth maps. FDSR and DKN were trained on the NYUv2 dataset, which was created using a Kinect device [13]. According to an older publication on depth map super-resolution task, it is better to use the nearest neighbor for pre-upsampling input images generated by structured light scanners. The nearest neighbor should be more precise on the blurred edges than when using interpolation-based methods.

There is also possible to use an end-to-end trainable up-sampling layer based on pixel shuffle or deconvolution operation [8]. The most recent publications work with the nearest neighbor or bi-cubic interpolation as their pre-upsampling methods [27]. We tried to combine FDSR and DKN with the deconvolution layer applied before the whole architecture. However, the training process was unstable, and we reached worse results than with bi-cubic or nearest neighbor pre-upsampling methods. We achieved the best results by using the nearest neighbor method, so we decided to use it in our thesis as the pre-upsampling method. We use the nearest neighbor pre-upsampling while creating a dataset for FDSR and DKN models.

4.1.4 Data normalization

In order to make the training process more stable and generalize the trained models for various scanned scenes, we decided to normalize our data. The input sample consists of a filled low-resolution depth map, augmented intensity texture, and filled ground truth high-resolution depth map. Normalization of our intensity texture is done by scaling all possible 16-bit grayscale values to interval $[0, 1]$. We assume that all samples are filled with our proposed filling method. The normalization of depth maps is a little bit different. The scanned scene used depth values interval can differ. We solved this problem by finding the minimum and maximum depth value of all dataset samples. We scale all depth maps by the difference between the found maximal and minimal value, and then we subtract the minimal value. That means we put all depth values of depth maps to interval $[0, 1]$. This normalization of intensity textures and depth maps makes the model more general and stabilizes training. Normalization is applied on samples when getting a concrete sample from the dataset so the samples are stored in the original scale.

4.2 CNN models

The project is written in Python with PyTorch [17] machine learning framework. We also use the NumPy library for the data pre-processing. We train the models and process all data on a device equipped with NVIDIA Quadro RTX 4000 [16]. Before the whole thesis project work, we set up the environment that provides a PyTorch framework together with GPU cores usage for effective processing of the image data. The CNN models that we chose for this thesis are FDSR and DKN. The models were chosen thanks to their topicality and the availability of their official implementation. The published DKN model implementation [10] is in Python and PyTorch. We analyzed the published code, found the crucial parts that include model architecture, and incorporated the acquired architecture into our project pipeline. The first problem that we faced was our training hardware limit. DKN model, by its design, needs a large amount of GPU memory to work. The Photoneo depth maps with intensity textures have higher resolution than the NYUv2 dataset Kinect device [13] samples. We had to somehow reduce the model size because it could not fit into GPU RAM. Our solution for this problem was halving the number of applied filters in all DKN model architecture convolution layers. After a few experiments, we observed that the training process was stable with the reduced model. Another minor adjustment of our data to the acquired DKN implementation was making the intensity texture a three-channel image. We do this by cloning the intensity texture three times to form three channels. This process is based on recommendations of the DKN model author when only one channel intensity texture is provided at the input. FDSR model implementation that is publicly available [3] is not written in PyTorch. This was unsuitable for us because we wanted to examine and modify the model for our data. We rewrote it using the PyTorch framework and validated both model implementations on the part of the NYUv2 dataset. When we observed that the models acted as expected, we started applying them to our Photoneo fusion-typed dataset.

Loss function

We first tried FDSR and DKN models with their default parameter settings, resulting in unpresentable results. The default loss function that both models used was L_1 loss. We used a small dataset whose samples were scenes composed of small warrior figure placed on the flat ground surface. The depth map from Photoneo 3D scanner has sharp edges and precise object surfaces. After training, we obtained a model with low L_1 error values, which looked like a good result. After 3D point cloud computation, we discovered that the warrior figure points were messed up, and the values were not as we expected from the low L_1 error. The reason for this was that the flat ground depth

values from the scene were precise enough, and the model was not forced to modify itself a lot during the training. The flat ground depth values had a small error. They pulled the overall mean absolute error to the low value because the biggest percentage of the depth image was covered by the flat ground. L_1 loss function gives the same attention to all image areas, which causes problems when working with our scenes designed for the fusion task. Authors of the FDSR model proposed the concept of edge error in their publication. The edge error is the overall error computed only from near-edged regions of the depth image. We implemented this concept in our project using the Canny edge detector from the sctikit-image python library [23]. We found the edges in the depth map and considered the 5-pixel radius neighborhood near the edges as the area for the error calculation. The edge error values were significantly higher than the overall L_1 error, as we expected. We tried to incorporate our Canny loss function into the training process by summing it to the L_1 loss or using it with various weight values. However, we did not achieve any presentable results. We had to develop a different approach. The idea of giving more attention to important areas of the depth map when computing the error inspired us to design our custom loss function, which we call *Object loss*.

Object loss function

The main idea behind our object loss function is to compute the sample error that gives significantly higher attention to the depth map pixels belonging to the scanned object. This loss function forces the model to modify itself more during training according to scanned object precision. The pixels of flat ground surrounding the object in the scene also contribute to the loss but with negligible weight. In the fusion task for which our data is created, our concern is nothing but the pixels that are in the area of the image containing the scanned object. The scene environment around the object will be removed in post-processing of the computed depth maps so their values do not need to be very precise.

For computing an object map, we must determine which pixels belong to the object and which do not. In our implementation, we compute an object map which is a binary matrix of the depth map dimensions that for every pixel contain the value one if the pixel belongs to the object and 0 if not. Using an object map, we compute weighted L_1 error that gives weight 1 to the object pixels and weight 0.01 to the non-object pixels. The constant 0.01 was determined empirically to achieve a stable training process.

Obtaining object map

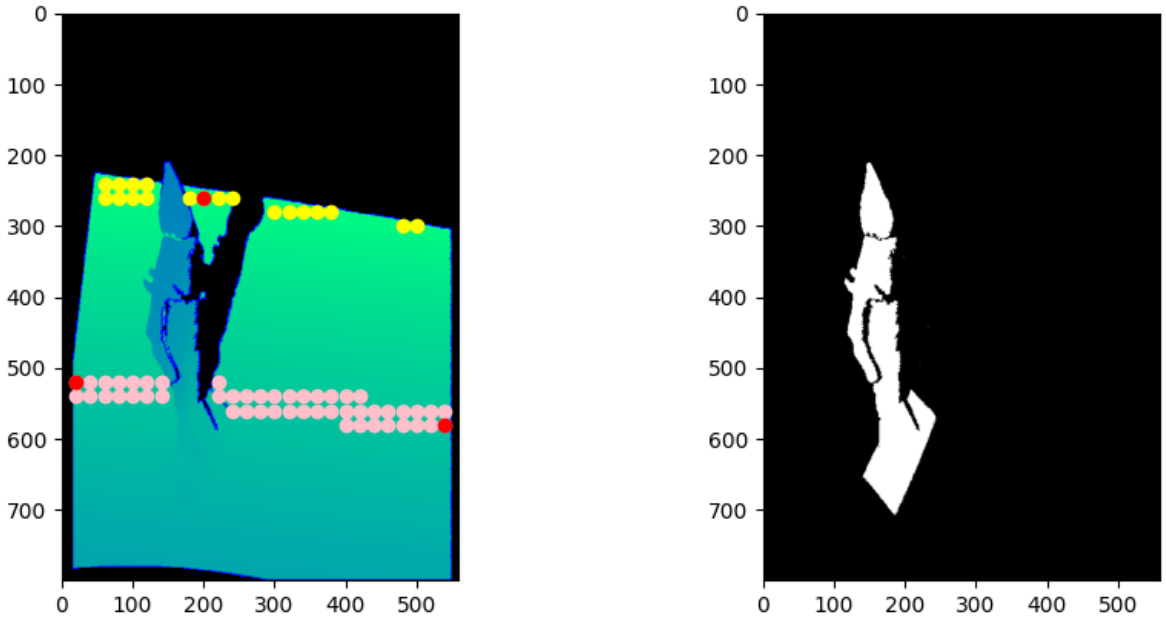
Obtaining the object map is derived from the composition of our scenes. As the previous text states, our scenes comprise one object on the flat ground surface. If we find the geometric parameters of the scene's flat ground in its coordinate system, we can remove

the flat ground and everything below it. If our scenes are correctly created, only the scanned object should be left in the depth image after removing the flat ground. The remained pixels, after this process, form an object map. Now the question is how to find a plane that fits the scene ground. We solved this issue by finding three pixels belonging to the scene's flat ground. These form a triangle which determines the searched plane. We choose the three ground points as follows: We assume that the biggest part of the scene contains the ground. That means the ground pixel depth values should be closer to the depth map mean value than the object depth values. Based on this assumption, we compute for every depth map pixel its squared deviation from the whole image mean. This image tells us how far from the mean every pixel is. Then we sort obtained distances and take the maximum from the lowest 5% as a threshold for determining whether the pixel belongs to the plane. If some pixel value has a squared deviation from the image mean that belongs to the lowest 5% of all pixels deviations, we consider it a flat ground pixel.

At first, we tried to select the three points randomly from the group of pixels determined by the threshold. This approach turned out to be inappropriate because the randomly selected pixels were often too close to each other. Therefore the computed plane was not precise enough to fit the ground. When we visualized the group of 5% mean-nearest pixels, we observed that they were all placed near the horizontal line of the scene ground. The line was, in most samples, placed near the center of the scene ground. The close neighborhood of the mentioned line is suitable for choosing 2 of three searched pixels. We can choose the most left and the most right one. The two pixels would create a long triangle base. To find a good plane, we must carefully choose the triangle's third vertex. The best would be a pixel near the scene's ground top or bottom border, far from the horizontal line from which we choose the triangle base vertices. Based on this reflection, we introduced a new group of pixels that should meet the described requirements. This group is formed of the most distant pixels of the depth map. We empirically found that pixels with the depth value belonging to the highest 2% of all depth map values belong to the scene ground near-top-border area. To set the threshold for determining if a pixel belongs in this group, we sort all image depth values and take values that separate the highest 2%.

To conclude, we have two groups of pixels from which we choose three vertices of the triangle. The two pixels that form the triangle base are chosen from ones with value belonging to the mean-nearest 5% of sample values. The third remaining pixel is chosen from pixels with values belonging to the highest 2% of sample values.

To maximize the triangle base length, we sort all possible near-mean group pixels by their y coordinate and take the most left and the most right as the result vertices. For maximizing the length of the other two triangle sides, we sort all possible maximal-depth group pixels by their y-coordinate and take the middle pixel as the resulting vertex.



(a) (b)

Figure 4.7: In figure (a), we show the input depth map where the yellow points represent the maximal-depth group of pixels of the grid, pink points represent the near-mean group of pixels of the grid, and the red points represent the chosen points to be vertices of the triangle determining scene ground plane. Figure (b) shows the output object map, which is the result of removing depth values that fit the plane determined by the triangle from (a).

To speed up the whole process of finding the triangle vertices pixels and maximizing their distance, we use just a subset of depth map pixels. We define a grid on the depth map composed of 20×20 pixel-sized blocks. We use only the grid line's intersection pixels. This reduces the whole working set of pixels to a few hundred.

After obtaining the three points, we have a triangle from which we can compute the plane. The three points provide two vectors from which we compute a normal vector of the wanted plane. Then we combine this vector with one of the three points and get the equation of the plane that fits the scene ground. In the end, we generate the output object map by removing every depth value that fits the plane and providing a binary map of the remaining pixels. The object map is for every sample pre-computed while creating the dataset. Every sample then contains a filled LR depth map, HR intensity texture, HR definition map, HR object map, and the HR depth map as ground truth. The object loss function then uses the object map for computing the weighted error. For visualization of the triangle pixels choice and an example of this process on the real data, we provide Fig. 4.7.

The object map obtaining method described above works well for our data because we created the sample scenes on our own, and we complied with the necessary standards

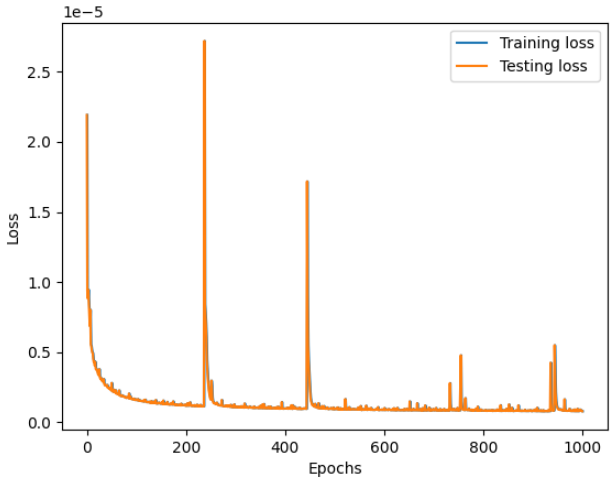
for the scenes. Some samples in the dataset also contain noise, like part of a mistakenly captured scene environment or ground surface irregularities. These samples cause some problems, but the triangle vertices are generally chosen correctly. The most common behavior on the not standard samples is that its output object map contains few artifacts alongside the scanned object, which is acceptable. The object map obtaining method is designed for this concrete data type we use for the 3D object fusion use case. The obtaining of ground-fitting plane can be obtained by other approaches too. A familiar approach is to place a known marker on the ground surface and compute the scene coordinate system from its deformation using the intensity texture of the scene.

Training

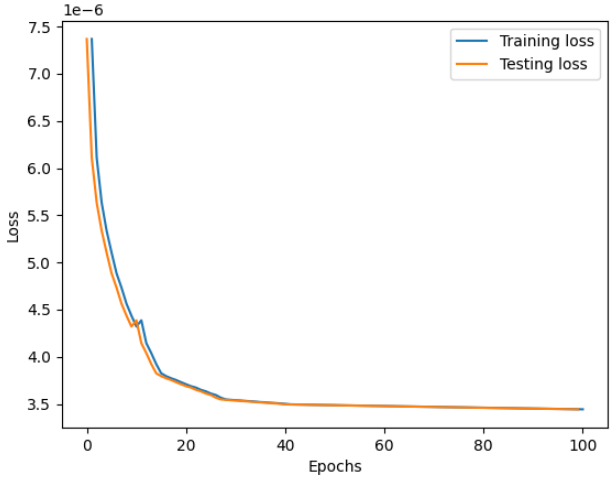
We train the models on the device equipped with NVIDIA Quadro RTX 4000 [16]. The mentioned GPU card uses CUDA architecture. All procedures that we do on the data are implemented in a way compatible with the GPU support. We set up the CUDA version 11.6 environment to work with Python and the PyTorch framework. Our dataset is composed of about 1200 samples generated by the Photoneo MotionCam-3D [18]. Every sample contains a filled LR depth map, HR intensity texture, HR definition map, HR object map, and the HR depth map as ground truth. We separate the 70% of the dataset samples as the training set and 30% as the testing set for the model validation. We train FDSR and DKN models, which both use Adam optimizer. The FDSR has a learning rate parameter set to 0.0005, and the DKN model uses a 0.0001. The FDSR model was trained in 1000 epochs, which took about 12 hours. The DKN model was trained in 100 epochs, which also took about 12 hours. We save the model checkpoint state after every 100 epochs. We use our custom object loss function for both models instead of the original L_1 loss. In Fig. 4.8, we show the learning curve of both models. We can see that the DKN model training process is more stable than the FDSR. Both models testing and training loss tends to decrease during the training.

Data post-processing

DKN and FDSR models trained on our dataset are imperfect, especially in areas near the scanned object edges that often contain wrongly estimated outlier values. To preserve the Photoneo image quality, we would remove the incorrect values. After applying our model to the input data, we compute the 3D point cloud and run the outlier removing procedure from Open3D python library [28] on it. This post-processing causes the loss of some vertices, but we value precision over the quantity of output 3D point cloud points. The amount of the lost points varies but often does not overcome 1% of the input point cloud size.



(a)



(b)

Figure 4.8: In figure (a), we show the progress of training and testing loss of the FDSR model. Figure (b) shows the same for the DKN model.

Chapter 5

Evaluation and results

In the previous chapters, we described the CNN models used. We Analyzed their behavior on our data generated by the Photoneo MotionCam-3D [18]. We reduced the number of filters used in the DKN model architecture to fit into our hardware conditions. We introduced the filling procedure that estimates undefined pixels of input depth maps. Alongside the filled depth maps, we modified high-resolution textures to preserve the correlation between the two input images. We also proposed the custom loss function that gives attention to scanned objects during training. After applying the trained model to the input data, we run post-processing on the computed 3D point cloud to remove the outlier points created by the imperfection of the model. All the data modifications and proposed procedures were used to improve the model's training process. After the training, we obtained models that can be applied to the Photoneo MotionCam-3D data. This chapter will examine the evaluation options for analyzing our results. We will review some possible applications of our trained models and examine their precision. Qualitative and quantitative metrics will evaluate the precision of achieved results.

Some metrics can be applied directly to the output depth map of our models, and other metrics can be applied to the 3D point cloud computed from the depth map. In the following sections, we will examine metrics suitable for these data types and use them for our purpose.

5.1 Depth map metrics

Depth map metrics compare two 2D matrices. Authors of FDSR and DKN models use root mean squared error (RMSE) to evaluate their output depth map compared to the ground truth high-resolution one. This metric is unsuitable for us because our models are modified to give low attention to the biggest part of the whole image. Using RMSE on our data does not provide beneficial information about the output depth map quality

Method	**RMSE**	**Object RMSE**	**Object loss**
FDSR	1.9537	4.4297	1.0427
DKN	2.2696	6.9191	2.3058
Nearest	3.0778	10.8156	7.0692

Table 5.1: Mean depth map metrics values of whole validation dataset.

because we are not interested in the error of the background plane but in the scanned object only. The more significant metric can be RMSE in combination with the object map. We can compute RMSE only from a set of object pixels. We call this metric a *Object RMSE* - RMSE applied to the object's pixels. We computed mean error values for the whole validation dataset to quantitatively analyze our results using these depth map metrics. The validation dataset contains 327 samples. We also computed the mean object loss of the whole validation dataset to link the Object metric described above and standard RMSE with the training process. Mean values were computed for FDSR, DKN, and simple nearest neighbor upscale. We examined the nearest neighbor method to show that FDSR and DKN models that we trained outperform it. All mentioned values stated in millimeters can be found in Tab. 5.1.

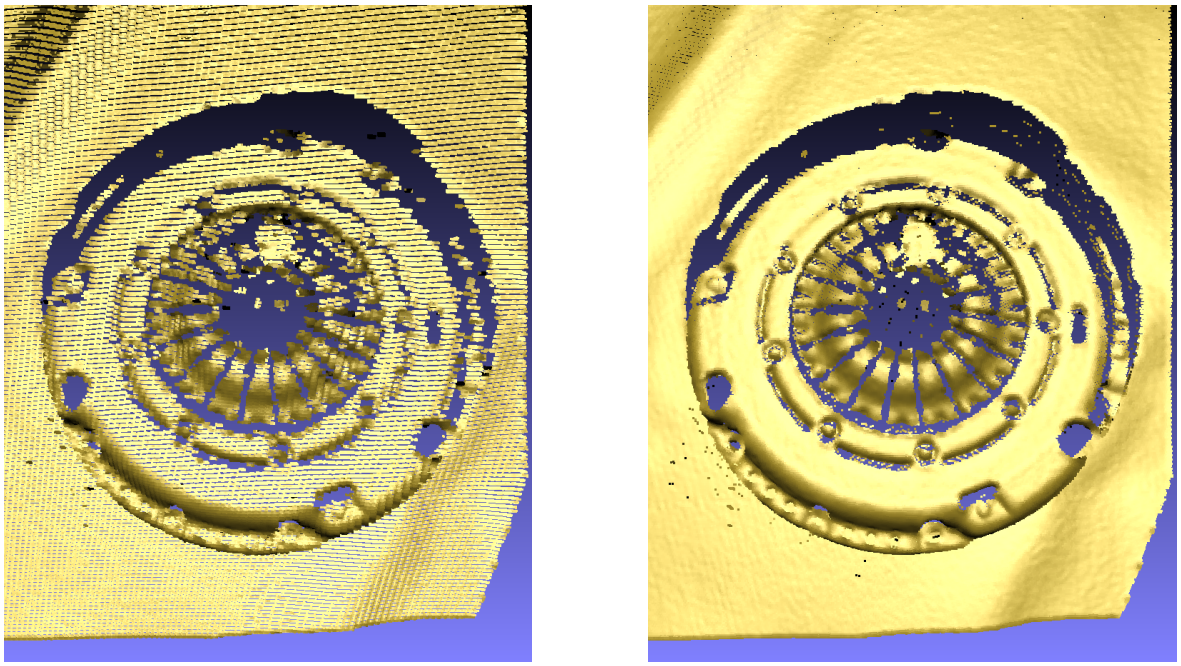
In Tab. 5.1, the Object RMSE metric has higher values than unmodified RMSE. This follows from the fact that in the RMSE, we compute the error value from all pixels, which means we also consider the scene environment. The error of the ground plane pixels is generally deficient, and it pulls the overall mean value down despite the high error object pixels. We can also notice that Object loss computed by weighted error favoring the object pixels above the rest is low for the trained models and higher for the nearest neighbor method. Object loss does not precisely highlight object error values like the Object RMSE, but it counts with all depth map pixels. This loss function is like an approximation of the Object RMSE, which makes the training process stable. Hypothetically it would be better to use Object RMSE as a training loss function, but it ignores the error of the non-object pixels. Ignoring pixel error values leads to unrestricted image error areas and, therefore, unstable training with a large set of outliers.

Object RMSE values from Tab. 5.1 tells us that the root mean squared error of the scanned object pixel values is about $0.5cm$ when using FDSR and DKN models and about $1cm$ for the nearest neighbor method. These values are too high to be usable. High error values near the object's edges probably cause these significant overall error values. Rather than having no precise points in the point cloud computed from the depth map, we remove the outliers in post-processing. The final error is then lower, but a few points are missing in the resulting point cloud. The outliers cleaning is done on an unstructured point cloud, so the RMSE metrics can not be applied afterward.

Depth map metrics described above do not provide useful information about result depth image quality because of the cleaning process of outliers present in depth maps. However, these metrics can be helpful in quick proof of the up-sampling model correctness. If the RMSE or Object RMSE is very high, we can assume that the resulting depth map is not precise enough for further processing. If we want to analyze the output depth map more accurately, we need a metric that is applied after the outliers cleaning process. This leads to unstructured 3D point cloud metrics that will be introduced in the following sections.

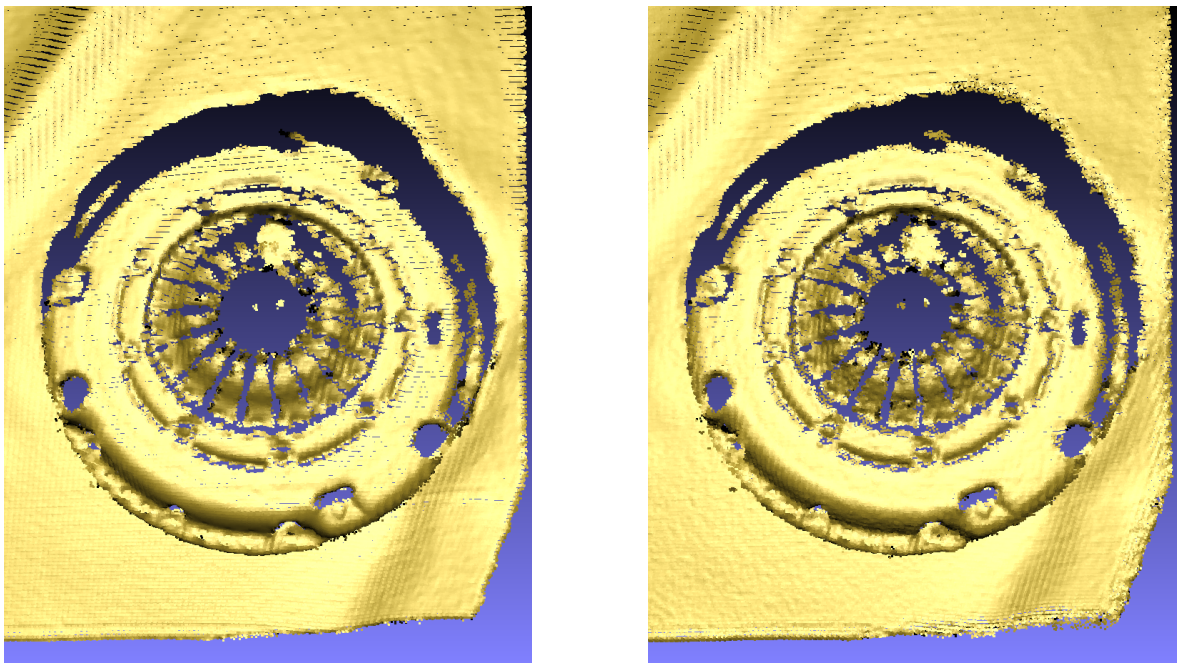
5.2 Point cloud metrics

After applying trained models on the input depth map, we can compute an unstructured 3D point cloud, which is a set of points in space. This point cloud allows us to analyze our output using different metrics. The first and most straightforward method for point cloud precision evaluation is analyzing the sample by hand. We can use proper software that provides an option to examine the point cloud in 3D space. We use Meshlab software [20] for the sample analysis. When we load the point clouds computed from high-resolution ground truth depth maps, low-resolution input depth maps, and DKN and FDSR output depth maps, we can compare them by applying filters and shading parameters, which help us to recognize differences. These differences are not observable in the raw depth map. 3D space with light and shading provides more options to examine the scanned object surface. We show photos of rotor object point clouds from the Meshlab software in Fig. 5.1 and Fig. 5.2. The whole scene in Fig. 5.1a is not precise. The nearest neighbor method that was used for generating the depth map implies the gridded structure of the scene surface. Every 4×4 pixel window of the showed image is determined by one low-resolution depth map pixel value. This causes the gridded surface structure. Fig. 5.2 shows the output of our trained models. It can be seen that models, to some extent, eliminated gridded scene surfaces and estimated more accurate pixel values. Examining up-sampled depth map quality by hand is a familiar approach researchers often use. An example of an area where people often evaluate results is the gaming industry. Game developers often get feedback on their concrete samples from game users. The result sample analysis done by the user is a meaningful metric for our task. As Fig. 5.1 and Fig. 5.2 show, we achieved relatively good results. By the deeper point cloud analysis we conclude that the user-based metric determines the DKN model as a better performing one.



(a) (b)

Figure 5.1: Figure (a) shows a photo of the point cloud computed from the low-resolution depth map. Figure (b) shows the high-resolution ground truth point cloud.



(a) (b)

Figure 5.2: Figure (a) shows the point cloud computed from the DKN model output depth map. Figure (b) shows the FDSR output.

Quantitative point cloud analysis

Alongside the people-feedback qualitative results analysis, it is also good to have some quantitative metrics. It can bring us more precise estimations about the model's output accuracy. We aim to compare two point clouds: model output and ground truth. We will use a standard point cloud comparison method called Hausdorff distance [9]. This metric computes the distance between 2 sets of points. Hausdorff's distance for one point from the other set is simply the longest distance to the closest point in the other set. We compute Hausdorff distance for every point of the model's output point cloud. We show the results of the Hausdorff distance metric applied on the output samples in Tab. 5.2. The table states the minimal, maximal, mean, and root mean squared error of Hausdorff distances. We can see that every up-sampling method contains outlier points with a large Hausdorff distance. Mean, and RMSE values fit into 2 millimeters deviation, which we consider acceptable. We see that the nearest neighbor up-sample has a low mean and RMSE. That means that values in the table should not be the only data we rely on when evaluating precision of models. We will examine all computed Hausdorff distances of the compared point clouds. This will be done by coloring points by RGB color specter Hausdorff distance mapping. Our RGB mapping will be from blue through green to red color. We will set the threshold for red points to 2 millimeters, meaning that red points in the colored point cloud will be considered inaccurate. Values that fit into 2 millimeters deviation will be mapped into the RGB specter. The most accurate points should be blue. As we mentioned before in the text, it is the scanned object that we want to be precise. Therefore, we extracted only the scanned object from the examined samples. The scene's environment was erased using our object map that identifies where the scanned object is. Visualization of Hausdorff distance using the RGB specter described above is shown in Fig. 5.3. In Fig. 5.3, we can observe that the simple nearest neighbor up-sample output point cloud contains the fewest red - inaccurate points. FDSR and DKN outputs contain more inaccurate points, especially near the sharp edges of the scanned object. The models have certain error near object edges bigger than simple nearest up-sampling. FDSR and DKN outputs obviously outperform the nearest method's overall surface precision. We can see that a significantly larger amount of scanned object's surface points of FDSR and DKN outputs are darker and purer blue than the nearest neighbor output ones. To conclude, FDSR and DKN model outputs provide a more precise object surface but have a certain error on the object edges. The nearest neighbor output also contains inaccurate points near edges, and the whole object surface is significantly less accurate than FDSR and DKN. The FDSR and DKN clearly outperform the simple nearest neighbor up-sampling method, which should prove our models' usability. By the deeper analysis using this metric we can not say clearly which of the models performs better but the DKN result

Method	min	max	mean	RMSE
FDSR	0.0000	9.8178	0.4650	0.7755
DKN	0.0000	10.3815	0.4449	0.7978
Nearest	0.0000	29.5586	0.4144	0.7455

Table 5.2: Results of Hausdorff distance metric applied on output point clouds of up-sampling methods. All samples were compared with ground truth high-resolution point cloud of the scene.

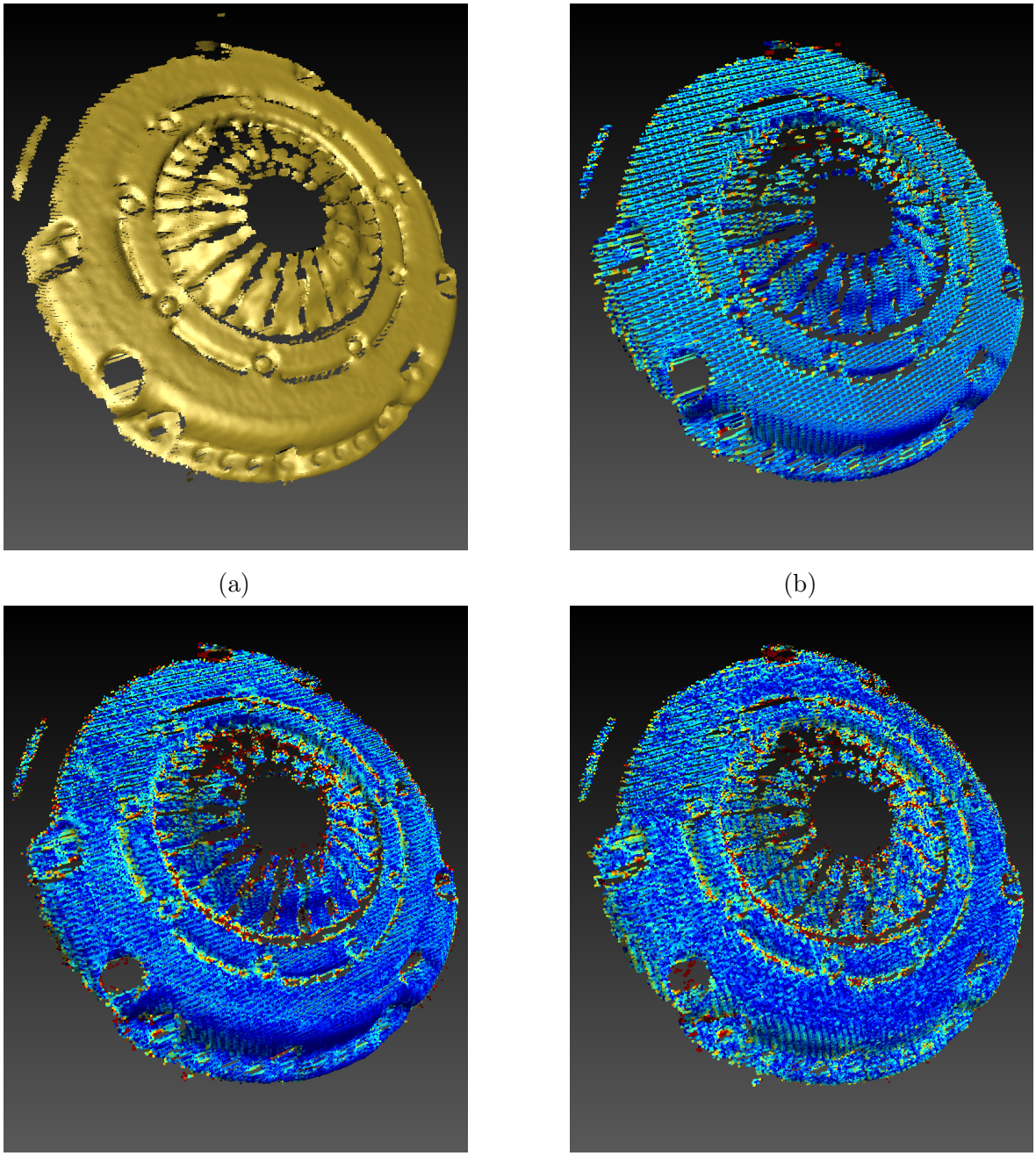
seems to be better.

5.3 Mesh metrics

We created our Photoneo dataset suitable for the 3D object fusion task. That means the output from the task solution will be a 3D object which is computed from an unstructured 3D point cloud of the object. The object's point cloud is created from multiple point clouds of scene scans taken from different views. The different view point clouds are aligned to form one object. The output point cloud of the views alignment is then cleansed from the outliers, noise, and the flat surface that is the object stated on. From the cleansed aligned point cloud, we can generate an object mesh composed not only of points but also of faces between them. The 3D object mesh can then be compared to a synthetic 3D mesh of the object designed in some CAD software. An example of suitable data for this validation metric is designing an appropriately complex object in the CAD software and printing it using a precise 3D printer. Scanning such an object and comparing the result with synthetic ground truth can give us a reference for the scanning process precision expectations.

There are several methods of obtaining appropriate scans for object fusion. Some kind of reference is often used for the aligning process while scanning, like stable camera position, markers for computing the translation matrix of the scene space, and others. The rotary table approach is the standard approach for taking a consistent sequence of scans from different views. When using this method, we place the scanned object on the flat base rotated by a motor controlled by scanning software. The software scans the object and rotates the base at the proper angle. This procedure is repeated multiple times, capturing all needed views for virtual object reconstruction.

In this thesis, we try to solve the depth map super sampling task. If we want to use the mesh metric described above to evaluate our results, we could compare five meshes computed from different data: low-resolution scene point clouds, ground truth high-resolution point clouds, high-resolution point clouds computed from up-sampling model output, low and high-resolution synthetic model of a scanned object designed in



(a) (b)

(c) (d)

Figure 5.3: Figure (a) shows a photo of a point cloud computed from a high-resolution ground truth depth map. Figure (b) shows the low-resolution sample point cloud. Figures (c) and (d) are outputs of DKN and FDSR models. Figures (b), (c), and (d) show Hausdorff's distance for every point when comparing the point clouds to the ground truth from the figure (a). Hausdorff distances of points are mapped to RGB spectre. The most blue points are the most accurate, and the red ones are inaccurate - not fitting into 2 millimeters maximal deviation. The Hausdorff distances in the accurate interval of 0 - 2 millimeters are linearly mapped from blue through green to red.

CAD software. If we find out that the mesh computed from our up-sampled depth map matches the synthetic model, it would prove the correctness of our models and allow us to work with low resolution in general and up-sample it with our models afterward. We were not able to get data for the 3D fusion task from the Photoneo MotionCam-3D [18]. Generating data that would fit our requirements included non-trivial intervention into technology and software available for us. Performing the 3D fusion experiment described above is a good type for future work that can be done with the output of this thesis.

Metrics conclusion

To conclude, we trained two models: FDSR [7] and DKN [11]. These CNN models take down-sampled depth maps as input and provide up-sampled high-resolution depth maps as output. We introduced the RMSE metric that showed these models outperform the simple nearest neighbor up-sample method. For a more precise analysis of the up-sampled depth maps, we computed unstructured 3D point clouds from them. We examined them in Meshlab software [20] and found that CNN models outperformed the nearest neighbor method. The surface of the scanned object was smoother and closer to ground truth in general. From the user's point of view, we consider the FDSR and DKN output point clouds useful for further processing. We analyzed model output point clouds with a quantitative approach too. We used Hausdorff distance [9] of point clouds to compare our FDSR and DKN point clouds with the ground truth. The Hausdorff distance was visualized by mapping the point's color to an RGB specter. Colored point clouds showed that FDSR and DKN models have a certain error on the object edges but clearly outperform the nearest neighbor method, which we expected to happen. There will always be some error present when using CNN models. There is, of course, space for improving DKN and FDSR precision by using a bigger dataset or fine-tuning training parameters. Using the mentioned metrics we can conclude that the DKN model provides better results than the FDSR. After all, we trained models usable for up-sampling depth maps from Photoneo MotionCam-3D [18]. The output depth maps are appropriate for further processing.

5.4 Time measurements

One of the motivations for the depth map super-resolution task was speeding up its processing pipeline. The main idea was to down-sample the input depth map, perform filtering in low resolution, and up-sample it back using our CNN model. If we want to evaluate the efficiency of our solution, we need a reference comprising of time

Resolution [px]	140x200	560x800	1120x800	1680x1200
Time [s]	0.054	0.068	0.091	0.184

Table 5.3: Measured time of the depth map processing by the Photoneo pipeline on different resolutions. The time value for the lowest resolution is estimated by fitting the second-degree polynomial function to known high-resolution values.

Model	FDSR	DKN
Time [s]	0.007	0.634

Table 5.4: Measured time of the depth map processing by FDSR and DKN models on depth maps with 140x200 resolution.

measurements of some real depth-map-filtering pipeline on concrete image resolutions. Our reference will be the Photoneo filtering pipeline. We did not have access to the setting parameters of the pipeline, but we were able to measure time for three different resolution runs. From the obtained time values, we estimated the running time value on our down-sampled depth map resolution. The estimated value was computed by fitting the second-degree polynomial function to three known values. The measured times of the pipeline with our low-resolution estimated value are stated in Tab. 5.3. From the measured times, we see that processing time tends to grow exponentially while increasing the depth map resolution. That means if we train models on higher resolution depth maps, we can bring more significant processing time improvement. The future work that would give us a more accurate image of the pipeline time improvement is a modification of the pipeline parameters to work on our low-resolution samples. Measuring the actual time values would be better than their estimation.

To analyze the possible processing time improvement by super-sampling depth map, we measured the time of our FDSR [7] and DKN [11] models. The models were trained on a dataset comprising depth maps with 560x800 resolution. We computed the average time from ten runs. The results are stated in Tab. 5.4. We can observe that the FDSR model is about 100 times faster than the DKN model. Compared to pipeline processing time, it is clear that the DKN can not be used for run-time improvement. The advantage of using DKN is the higher precision of results. The FDSR brings speed and can be used for speeding up the pipeline. The DKN can be used for the depth map quality improvement, which was our second motivation to super-sampling task. We can use cheaper devices providing low-resolution depth maps and up-sample them by the trained model.

The other procedures we must count when measuring the depth map processing time are down-sampling and filling. Our implementation of these two currently needs to be more effective for speeding up the filtering pipeline. Average time from five runs of our

depth map down-sampling in 1,68s. Our down-sampling procedure is not optimized yet for running entirely on the GPU. Improving this process is a suggestion for future work that can be done to improve our current solution and make it capable of speeding up the pipeline. The same for the down-sampling applies to the proposed filling procedure. Currently, it takes 0,36s to fill the input low-resolution depth map. We proposed and implemented the filling and down-sampling procedures to stabilize the training process that brings usable models with acceptable accuracy. If the 3D-capture device provided completely defined depth maps, our models could be used directly without the pre-processing.

For now, our models can be used to improve the depth map quality by increasing its resolution. If we optimize the depth map pre-processing procedures or the 3D-capture device provides us with a sample that does not need to be filled, we can use models to speed up the pipeline. We could also get more significant results by speeding up the filtering process by training models for higher resolution as the pipeline processing time grows exponentially with the increasing depth map resolution.

Conclusion

In this thesis, we were solving the depth map super-resolution task. We chose the Photoneo MotionCam-3D device [18] for generating the dataset suitable for the fusion task. This dataset is different from the available ones. We studied available publications on the task and decided to use two deep learning CNN models: FDSR [7] and DKN [11]. These models were originally trained on a dataset created using a Microsoft Kinect device [13], whose output data are in lower resolution and less accurate than the MotionCam-3Ds. We had to pre-process the MotionCam-3D data for the models to work correctly. We proposed a filling method for estimating the undefined depth map pixels. The depth map filling procedure ensured the stability of the training. We used the same filling procedure to augment intensity textures to enhance their correlation to depth maps. The input data were normalized before the training. To improve model precision and stabilization of the training, we proposed a custom loss function that gives attention to important areas of the depth maps. We trained modified models on our dataset and evaluated the achieved results with several quantitative and qualitative metrics. The achieved high-resolution depth maps clearly outperform the simple nearest neighbor method. User analysis by Meshlab software [20] and the quantitative analysis by Hausdorff's distance showed that point clouds computed from the model's output are usable for further processing. The DKN model achieved better results when considering the output depth map precision. We also measured the computing time of the models and found that the FDSR model is significantly faster. From the measured times and the outputs obtained by the models, we can say that the DKN model is better for the depth map quality improvement and the FDSR model for the speeding up the filtering pipeline. Based on the chosen evaluation metrics and the time measurements, our results satisfy the aimed goals of the thesis.

Bibliography

[1] AMD. Amd fidelityfx. Retrieved April 17, 2023, from `https://www.amd.com/en/technologies/radeon-software-fidelityfx`.

[2] Ji-Min Cho, Soon-Yong Park, and Sung-Il Chien. Hole-filling of realsense depth images using a color edge map. *IEEE Access*, 8:53901–53914, 2020.

[3] Runmin Cong. Fdsr. Retrieved April 17, 2023, from `https://github.com/kingcong/FDSR`.

[4] Huiping Deng, Li Yu, Juntao Zhang, Bin Feng, and Qiong Liu. Edge-preserving down/upsampling for depth map compression in high-efficiency video coding. *Optical Engineering*, 52(7):071509, 2013.

[5] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[6] K. He, J. Sun, and X. Tang. Guided image filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, 2013.

[7] Lingzhi He, Hongguang Zhu, Feng Li, Huihui Bai, Runmin Cong, Chunjie Zhang, Chunyu Lin, Meiqin Liu, and Yao Zhao. Towards fast and accurate real-world depth super-resolution: Benchmark dataset and baseline. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, page 9229–9238, 2021.

[8] Tak-Wai Hui, Chen Change Loy, and Xiaoou Tang. Depth map super-resolution by deep multi-scale guidance. In *Proceedings of European Conference on Computer Vision (ECCV)*, page 353–369, 2016.

[9] D.P. Huttenlocher, G.A. Klanderman, and W.J. Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, 1993.

[10] Beomjun Kim. Dkn. Retrieved April 17, 2023, from `https://github.com/jun0kim/DKN`.

[11] Beomjun Kim, Jean Ponce, and Bumsub Ham. Deformable kernel networks for guided depth map upsampling. *ArXiv*, abs/1903.11286, 2019.

[12] Martin Melichercik. Thesis implementation source code. Retrieved April 17, 2023, from `https://github.com/Meli-0xFF/depthmap_sr`.

[13] Microsoft. Microsoft kinect. Retrieved April 17, 2023, from `https://learn.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows`.

[14] Pushmeet Kohli Nathan Silberman, Derek Hoiem and Rob Fergus. Indoor segmentation and support inference from rgbd images. In *ECCV*, 2012.

[15] NVIDIA. Nvidia dlss 2.0: A big leap in ai rendering. Retrieved April 17, 2023, from `https://www.nvidia.com/engb/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/`.

[16] NVIDIA. Nvidia quadro rtx 4000. Retrieved April 17, 2023, from `https://www.nvidia.com/en-us/design-visualization/rtx-4000-sff/`.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, page 8024–8035. Curran Associates, Inc., 2019.

[18] Photoneo. Photoneo motioncam-3d. Retrieved April 17, 2023 from `https://www.photoneo.com/motioncam-3d/`.

[19] Photoneo. The revolution in machine vision. Retrieved April 17, 2023, from `https://www.photoneo.com/the-revolution-in-machine-vision/`.

[20] Guido Ranzuglia, Marco Callieri, Matteo Dellepiane, Paolo Cignoni, and Roberto Scopigno. Meshlab as a complete tool for the integration of photos and color with high resolution 3d geometry data. In *CAA 2012 Conference Proceedings*, page 406–416. Pallas Publications - Amsterdam University Press (AUP), 2013.

[21] Daniel Scharstein. Middlebury stereo datasets. Retrieved April 17, 2023, from `https://vision.middlebury.edu/stereo/data/`.

[22] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, page 839–846, USA, 1998. IEEE Computer Society.

[23] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.

[24] Zhihao Wang, Jian Chen, and Steven C. H. Hoi. Deep learning for image super-resolution: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(10):3365–3387, 2021.

[25] Alexander Watson. Deep learning techniques for super-resolution in video games. *ArXiv*, abs/2012.09810, 2020.

[26] Zixiang Zhao, Jiangshe Zhang, Shuang Xu, Zudi Lin, and Hanspeter Pfister. Discrete cosine transform network for guided depth map super-resolution. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, page 5697–5707, 2022.

[27] Zhiwei Zhong, Xianming Liu, Junjun Jiang, Debin Zhao, and Xiangyang Ji. Guided depth map super-resolution: A survey. *ACM Comput. Surv.*, 2023.

[28] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.