JOHANNES KEPLER
UNIVERSITY LINZ

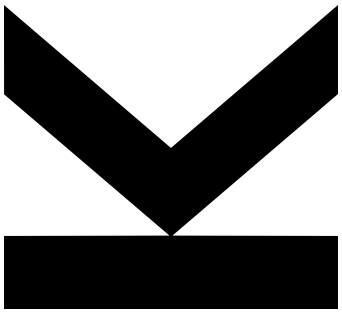
Author
Markus Kiesenhofer
k01255246

Design and Implementation of a Deep Neural Network for Surface Reconstruction from 2D Images

Submission
Institute of Computational Perception

Thesis Supervisor
Dr. **Josef Scharinger**

December 2022



Master Thesis

to obtain the academic degree of

Master of Science

in the Master's Program

Artificial Intelligence

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Kurzfassung

Diese Arbeit fasst eine zweistufige Rekonstruktionsmethode für eine spezielle Oberflächenstruktur, die in der Luftfahrtindustrie verwendet wird, zusammen. Die Erstellung von Bildern erfordert zusätzliche Informationen (Lichtposition / -intensität, Kameraposition, Oberflächenstruktur, Farbe usw.) über die Szene, um sinnvoll zu funktionieren. Parameter zur Rekonstruktion der 3D-Szene werden zum Beispiel verwendet, um die Oberflächenstruktur zu optimieren. Je mehr Unbekannte gleichzeitig vorhanden sind, desto schwieriger wird es, die Szene in 3D zu optimieren. Einige angepasste Methoden ermöglichen eine Optimierung mit vielen unbekannten Parametern. Zusätzlich werden synthetische Oberflächen verwendet, um ein angepasstes neuronales Netzwerk zu trainieren. Nach dem Training wird das neuronale Netzwerk verwendet, um reale Oberflächen vorherzusagen.

Abstract

This work summarises a two step reconstruction method for a special surface structure used in the aerospace industry. Rendering images needs additional information (light position/intensity, camera position, surface structure, color, etc.) about the scene to work in a meaningful manner. Parameters for reconstructing the 3D scene are used e.g. to optimize the surface structure. The more unknowns in parallel exist, the harder it will be to optimize the scene in 3D. Some customized methods enable an optimization with a plenty of unknown parameters. In addition, synthetic surfaces are used to train a customised neural network. After training, the neural network is used to predict real surfaces. The transition from synthetic surfaces to real surfaces needs some generalization of the network. A special function creates synthetic surfaces with the use of a probabilistic approach similar to real surfaces and the neural network is able to learn the task.

Contents

1 Introduction

Inverse graphics to compute 3D properties is a challenging task and as old as computer graphics itself. A novel approach called Differentiable Rendering is getting more and more attention. Forward rendering is the process for calculating a 2D image from a 3D scene. Unfortunately, calculating the gradients to re-engineer the 3D scene was long not possible. The forward function has a discrete step involved, which makes calculating gradients impossible. A new approach called Differentiable Rendering (DR) tackle this issue and produces promising results for predicting 3D scenes from 2D images. [1] surveys prominent approaches for backpropagating images. With DR it is possible to optimize the location of lights and cameras. Additionally, objects could be optimized e.g. optimizing material parameters, deform an object, rotate and translate the object, optimize the texture, etc. DR could help industries in different areas. The general approach allows a wide range of use cases. Profactor, an Austrian research company, uses DR for detection of manufacturing failures. Facebook developed a library [2] called Pytorch3D to accelerate research in DR and further improvements in DR lead to better results. The mentioned facts and improvements for 3D reasoning motivates to take a detailed inspection into DR. This work should also give 3D information about 2D images. DR could help to understand the 3D world and get a solution for a specific problem.

A rendering pipeline consists of two important function parts. The input is fed into a rasterizer and afterwards a shading model outputs a 2D image from a 3D scene. A renderer is in general a combination of a rasterizer and a shader. Prominent shading model are e.g. "Phong-shading", "Gouraud-shading", "smooth-shading" and "flat-shading". All these shading techniques are relative simple and they are not photo-realistic. Nevertheless, for some cases a phong-shading model leads to a reasonable solution. Toy examples and tutorials e.g. in Pytorch3D use simple shading techniques like Phong. For real applications a simple shading model is not realistic enough to perform well. Physically based renderer (PBR) are used as part of solution. As the name is already explaining PBRs use physical

models for shading and they produce a more realistic picture compared to simple shading techniques. However, PBR uses also approximations in rendering. There is always a tradeoff between efficiency and quality.

Before DR the rasterizer was discontinuous and the reason for hindering backpropagation. New rasterizer models enable continuity via approximation or probabilistic approaches. SoftRas [3] is a popular approach for a rasterizer in the DR field. In section 2 is an overview about prominent rasterizer and also a comparison about them.

1.1 Motivation

FACC is a company, which is building parts for the aerospace industry. The Freesurf sensor (figure 1.2) was built as a tool for quality assurance. It is used for cabin interiors to identify the roughness of a cap. Figure 1.1 illustrates a cabin-room of a plane. The mentioned cap is located above the seats and inside is usually some package from the passengers. These caps are produced by FACC and it turned out to become a critical part in quality assurance. Some light is shining onto the caps and even they have the same color, they look different. Clients from FACC complain about the different illustration from the caps. The reason is the difference between the roughness of caps. The Freesurf sensor from Profactor was designed to tackle this problem and it should determine the roughness of every cap. Having the roughness of each cap would be helpful to stick together caps with the same profile.

Figure 1.1: cabin interior of a plane. Figure from `https://unsplash.com/`

This work should use new machine learning approaches to estimate the cap material. The idea is to feed the images into a Neural Network and get the normal vectors back as a result. Having normal vectors from the cap material would help to determine the roughness.

1.2 Surface estimation with the Freesurf Sensor

Inspired by DR, this work uses main ideas and approaches to estimate normal vectors from a relative flat surface (cabin-cap). A sensor called "Freesurf" engineered from the company "Profactor" outputs twelve different images with one camera and twelve light sources. Each image corresponds to one active light source. The goal is to train a Neural Network, which is able to predict the normal vectors from the input of these twelve images. After training the Neural Network should be able to predict the surface correctly. The trained Neural Network in evaluation mode should be able to produce results with low computational power. The rendering function and the Neural Network is chosen to be efficient enough for that case.

Figure 1.2: Freesurf sensor from Profactor. Figure from `https://www.profactor.at/`

A lot of parameters can be used as arguments for DR methods. Nevertheless, some parameters have influence to others e.g. light intensity and material parameters. If the intensity of a light source is unknown as well as material parameters, it is not possible to find a local minimum. Therefore optimization is hardly possible. The same is for light intensity and light position. If a light source is far away, it needs more light intensity to produce the same illumination on an object and vice versa. Some parameters cannot be optimized in parallel, because of interdependency. One challenge for estimating the normal vectors of a cabin-cap is to find a solution for this interdependence. The location of light, material parameter and light intensity is unknown. Only an initial guess exists for the light locations. Optimizing everything together would not lead to a global optimum.

Additionally, the Freesurf sensor has no perfect point lights. A lot of models (e.g. Phong shading, physically based renderer [4], etc.) in theory assume point light sources, which simplifies the problem. The sensor lights are not perfect point lights and the design creates shadow effects. Some hurdles occur in practice to apply the DR theory on real world problems.

The suggested methods in section 4 to solve the cabin-cap task produces promising results as seen in section 7. However, the evaluation methods have to less information to assess the quality of results, because the surface structure of the cabin-cap object is unknown and therefore no ground truth exists to compare the predicted surface. Synthetic surfaces produced via a customized python function could handle the uncertainty in evaluation, if the synthetic surfaces are similar to the real cabin-cap surfaces.

1.3 Research question

This work should give answer to the following questions:

1. How to handle a computer vision task without having a perfect point light source? The Freesurf sensor produces a sensor specific illumination, which differs from sensor to sensor.
2. How to transfer a neural network, which is trained with synthetic samples into real world problems and how can synthetic samples be created for the cabin-cap task?
3. If only unlabeled image data exist, what can be done to evaluate the performance of a trained neural network?

1.4 Contribution

The surface reconstruction from 2D images by using data from the Freesurf sensor needs some new techniques to predict the normal vectors. A customized method (subsection 4.5) simulates shadow effects, which occurs by the use of Freesurf sensor. The sensor specific design introduces some shadow effects.

In addition, no labeled data was available at the beginning and the amount of unlabeled data was very low. Training a neural network was not possible with such a low amount of data samples. Thus, a customized function (subsection 4.10) was developed to create synthetic surfaces, which are able to simulate real samples. The neural network, which is trained by synthetic samples, is also able to predict real samples.

Data representation in 3D is very cost intensive. A 3D data representation called pixel-to-height (figure 4.4), which needs only one value to represent a 3D point, inspired from bumpmapping [5] is developed to simulate surfaces. This data representation can be used for any kind of surface structure, which is similar to the cabin-cap surface.

1.5 Outline

This work is subdivided into several chapters, which cover:

Chapter 2: A literature summery about state of the art Differentiable Rendering techniques and useful libraries for implementation.

Chapter 3: The evolution of specific computer vision architectures to solve the ImageNet challenge. Important improvements are summarised in the chapter.

Chapter 4: This chapter includes all important methods, which are used to solve the cabin-cap problem.

Chapter 5: The implementation chapter covers important descriptions to understand the programmed code.

Chapter 6: This chapter explains the usage of the optimization and training method.

Chapter 7: Experimental results for the optimization and training method are located in the chapter.

Chapter 8: Describes alternative approaches to solve the cabin-cap task.

Chapter 9: The chapter summarizes the findings, experimental results about the methods for solving the cabin-cap task.

2 Differentiable Rendering and shading techniques

Differentiable Rendering (DR) is a relative new field in computer vision. A lot of work and approaches have been developed in the last years. DR improves the 3D understanding for 2D images. A rendering function takes scene and material parameters (shape of object, camera position, lightning, materials, normal vectors etc.) and outputs a 2D image (RGB and depth map). Before DR it was not possible to get gradients for this computational process, because of discrete operations inside the function. Backpropagating images allows to reconstruct 3D scenes. [1] surveys promising approaches. Depending on different tasks, some approaches are more suitable then others. The different algorithms depend also on the representation of the rendered 3D object. Four main representations are used for 3D objects: Meshes, voxels, point clouds and an implicit form. [1] describes all pros and cons for the different data representations. To get gradients from the rendering process 2 ways are used for rasterization. First it is possible to get the gradients via approximation of the discrete operations, second way is to approximate the forward rendering function before backpropagating. Figure 2.1 illustrates a general differentiable rendering pipeline. The discrete functions are located inside the rasterizer. The rasterizer inspect every face of an object and distinguish between foreground and background pixel. A foreground pixel is seen from the camera perspective and a backward pixel is occluded for the camera perspective. The discrete distinction between them includes somewhere a discrete step.

Different publicity available frameworks to implement an algorithm for DR are already existing. These frameworks are promising for implementations. Especially Pytorch3D [2] is a library to compute the gradients in the 3D domain. It supports data representations like triangle meshes and point clouds. The documentation for Pytorch3D is well prepared and it has a very active community. The differentiable rendering algorithm called "Soft Rasterizer" [3] looks promising. The "Soft Rasterizer" is a differentiable renderer, which

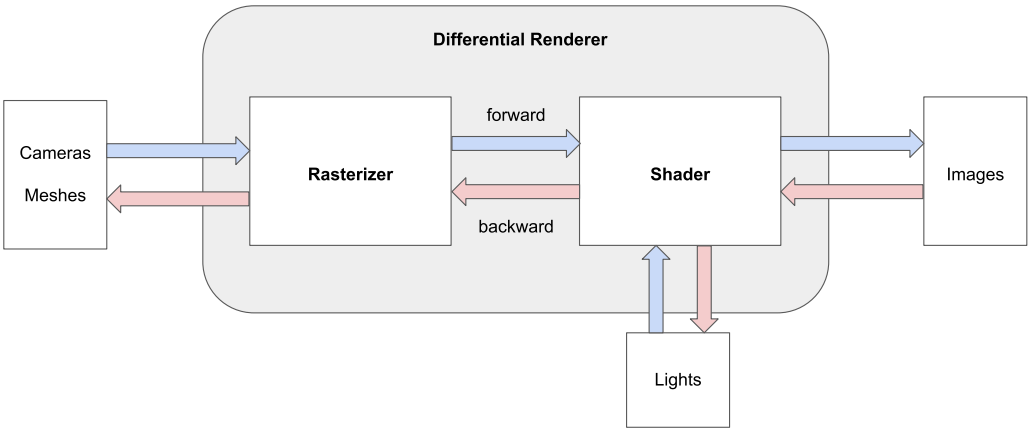


Figure 2.1: A differential rendering pass in general

is working in a general way and therefore it is suitable for many kinds of applications. Beside Pytorch3D Kaolin [6] is a library for writing DR-algorithms efficiently with Python. Kaolin supports more different rasterizers and less shaders compared to Pytorch3D.

For rendering a 2D image different kind of methods are used. One method is rasterization (e.g. Soft Rasterizer) and for most examples rasterization is a good trade off between producing good-looking images and computational effort. Images produced via rasterization depends also on the shader. In Pytorch3D [2] different shaders are available e.g. Phong shading, Gouraud shading, etc.

Currently there are more different rasterizer available with different approaches. One main distinction to separate them is the following: 1) Rasterizers with approximation of gradients, 2) Rasterizers with making the forward rendering function continuous to compute the gradients exactly. The second class produces better results as seen in the evaluation section of [3].

Another rendering function called physically based rendering (PBR) used by [7] provides a more accurate way to represent materials and how they are interacting with light. The Filament renderer introduced by [4] is built with the PBR method. The goal was to create

a renderer for mobile devices, which is able to create realistic photos. Some tradeoffs between efficiency and quality is made to support low and medium GPUs.

2.1 Approximate Differentiable Renderers

Several approaches exist to get the gradients from a 2D image. One way is to compute gradients via approximations. Different possibilities are existing to approximate the gradients. As seen in subsection 2.3 approximations do not lead to the best solution for creating the scene backward. Below are two common methods described to get approximated gradients.

OpenDR

Graphic renderers are usually not built to inverse the rendering process. OpenDR [8] is the first publicly-available framework for differentiating the forward render process. In a very simple description the forward process f to render an image I_{obs} is described with the following formula:

$$I_{obs} = f(\Theta) \tag{2.1}$$

where Θ are all model parameters in the scene including e.g. 3D body shape, lightning, camera parameters etc. OpenDR focuses on the reverse problem, also called inverse rendering. It approximates the model parameters Θ, which are: 1) Appearance A, 2) Geometry V and 3) Camera C. It is possible to compute the gradients with first order Taylor approximations. By introducing an intermediate variable U, which indicates the 2D projection of a 3D vertex, it is easy to see how the gradient flows to the different model parameters. The partial derivatives make it possible to optimize the scene (figure 2.2).

Before OpenDR only domain specific methods were available to inverse the rendering process. For each problem exists a separate algorithm to solve the inverse rendering pipeline. Some developed models for specific purposes were used for:

- Face modeling [9],
- 3D shape estimation [10],
- Human pose and shape [11]
- and many more.

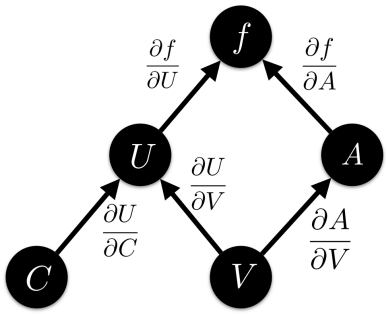


Figure 2.2: Partial derivative structure of the OpenDR renderer. Figure from [8]

OpenDR is the first general method. The framework is used to improve these novel fields of differentiable rendering. For implementation OpenDR is based on Numpy, Scipy, OpenCV and to get the autodifferentation part OpenDR uses the framework Chumpy. The time needed to compute the inverse rendering problem with OpenDR depends on user specific parameters e.g. triangles, amount of pixels etc.

The novel method initiated more research in future and Differentiable Rendering becomes more prominent in 3D reasoning. Despite these improvements in the field of inverse graphics, OpenDR produces noisy 3D predictions, because of approximation and getting only non-zero values for the gradients in a small area [12].

Neural 3D Mesh Renderer (NMR)

The Neural 3D Mesh Renderer [13], also called "Neural Renderer" or "NMR", was proposed after OpenDR. The NMR is also a renderer, which approximates the gradients, but in a different way compared to OpenDR.
To describe how the approximation is done, consider a simple triangle with 3 vertices and 1 face in object space. First the triangle is transformed into screen space via differentiable operations [14]. Second, an image is produced via traditional rasterization. Rasterization is a discrete function and cannot be differentiated.

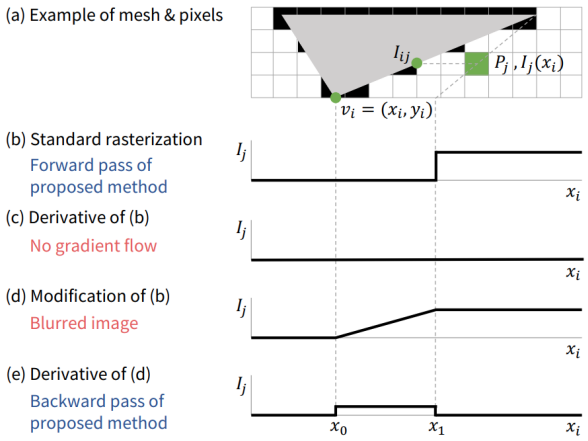


Figure 2.3: Illustration NMR method. Figure from [13]

The rasterization method produces a sudden change in pixel colour. This intense change from one pixel to the neighboring is discontinuous and makes it not possible to compute the gradients. Considering the triangle example figure 2.3 presents insights about the gradient flow. A change in colour at position x_1 outputs a step function (b), which is not differentiable (c). NMR proposes to smooth the transition (d) to get gradients via interpolation (e). This method is an approximation, but it is possible to backpropagate images.

2.2 Truly Differentiable Renderers

A truly Differentiable Renderer does no approximations about gradients. Usually the forward rendering function is changed in a way to be continuous. This allows to flow gradients to every parameter, even occluded objects / faces. Evaluation [3] between truly and approximated Differentiable Renderer indicates the improvement over approximated Differentiable Renderer.

Soft Rasterizer

A truly Differentiable Renderer computes the forward rendering function exactly with no discontinuity. This enables to compute the gradients correctly with no approximation. The first truly Differential Renderer was proposed from [3], called "Soft Rasterizer" (in short: "SoftRas").

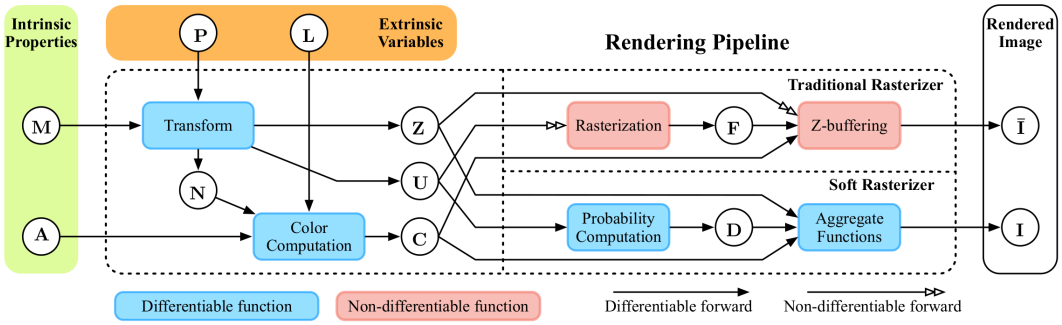
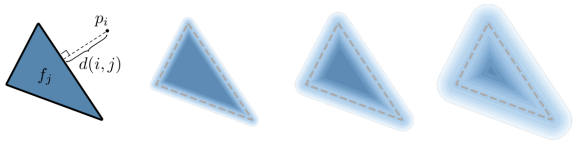


Figure 2.4: Comparisons between the standard rendering pipeline (upper branch) and truly differential rendering pipeline (lower branch). Notation of Variables: camera position **P**, lightning conditions **L**, meshes **M**, appearance **A**, mesh normal **N**, view-dependent depth **Z**, image-space coordinate **U**, color **C**, rasterization result **F** and probability maps **D**. Approach from [3]

.

Instead of using "rasterization" and "z-buffering", SoftRas uses a "probability density function" and an "aggregate function". For more details about the rendering process figure 2.4 shows an overview. Above in red is the Traditional Rasterizer and below in blue the Soft Rasterizer. Red blocks are non-differentiable and the blue ones are differentiable functions. Rasterization is a method including sampling. Sampling is discontinuous, therefore it is not possible to compute exact gradients. The same holds for z-buffering. Probability functions combined with aggregate functions produce a smooth transition between faces of the mesh. A paramter σ controls the smoothness. If σ is zero, the Soft Rasterizer is working in the same way as the Traditional Rasterizer. A SoftRas can be considered as a traditional rasterizer with $\sigma \to 0$. Figure 2.5 shows 4 different probability maps with different σ.

A big advantage to approximated rasterizer approaches e.g. NMR is the fact, that gradients from one pixel can flow to every face in a mesh even occluded faces. An occluded face is not visible in the image. This is not possible with OpenDR [8] and NMR [13].
The mesh M and the appearance A are intrinsic variables, which describe the object in the



(a) ground truth (b) $\sigma = 0.003$ (c) $\sigma = 0.01$ (d) $\sigma = 0.03$

Figure 2.5: Probability maps of a triangle (a) original face; (b)-(d) probability maps generated with different σ. Figure from [3]

.

scene. Extrinsic variables are the camera position P and the lightning L. These variables are inputs for the rasterizer to produce the output image I. The other variables are used to compute the internal state and help to compute the gradients. An exact description can be found in [3].

2.3 Evaluation between truly and approximate DRs

Comparing SoftRas (truly DR) with NMR (approximate DR) shows the benefits of Truly Differentiable Renderes. [3] evaluates the performance of the different renderes to compare them in a meaningful manner. One result is produced with "single-view mesh reconstruction" and the other experimental result is "image-based shape fitting". SoftRas produces in both experiments a quite better result.

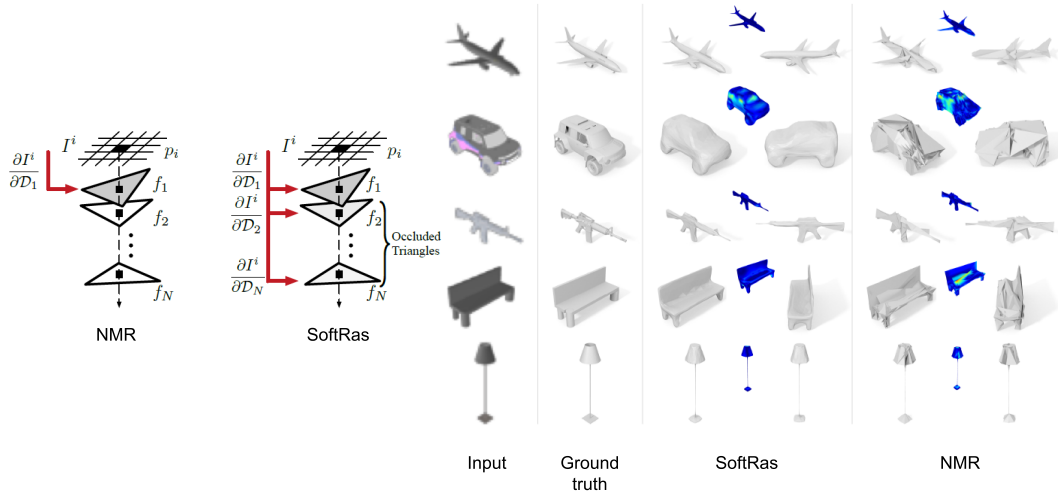


Figure 2.6: comparism SoftRas vs. NMR, figure from [3]

Figure 2.6 shows the different methods on the left side and experimental results for mesh reconstruction on the right side. SoftRas allows to flow gradients to occluded triangles. The smooth transition between triangle faces results in better mesh reconstruction. Overall SoftRas is not the only truly differentiable renderer e.g. DIB-R from [12] is very similar to SoftRas. Nevertheless, Pytorch3D [2] uses only one Rasterizer, which is SoftRas.

Table 2.1: experimental comparison between different objects, data from [3]

Category	Airplane	Bench	Dresser	Car	Chair	Display	Lamp	Speaker	Rifle	Sofa
NMR	0.6172	0.4998	0.7143	0.7095	0.4990	0.5831	0.4126	0.6536	0.6322	0.6735
SoftRas(sil.)	0.6419	0.5080	0.7116	0.7697	0.5270	0.6156	**0.4628**	0.6654	**0.6811**	0.6878
SoftRas(full)	**0.6670**	**0.5429**	**0.7382**	**0.7876**	**0.5470**	**0.6298**	0.4580	**0.6807**	0.6702	**0.7220**

Table 2.1 confirms the results with qualitative numbers. The table has two different rows of results for SoftRas. The first row is using only the alpha channel, which is the silhouette of the image. The second row uses as input all four channels, which are RGB and the alpha channel. Softras outperforms the NMR with an average of 4.5 points. The comparison is made with the mean Intersection over Unions on 10 categories on the ShapeNet dataset.

2.4 Useful libraries to implement Differentiable Renderer

Implementing a Differentiable Renderer is not straightforward. A lot of work is already done to simplify writing code with Differentiable Renderers. In this section two important libraries will be introduced. The libraries are not used for solving the task in this work. Nevertheless, it could help for getting into the field of DR and they can be used to implement first algorithm with less amount of time. These libraries can be used, if it is necessary to implement a rasterizer. As seen later, a rasterizer is not needed to solve the cabin-cap task.

Accelerating 3d deep learning with Pytorch3d

Pytorch3D [2] was developed from Facebook engineers. The library is built on Pytorch, which is a powerful framework for deep learning. Pytorch3D is implemented using Pytorch tensors. It can handle minibatches, uses autodifferentiation and supports GPUs to accelerate computations. The framework is easy to use with an understanding of Pytorch. It has some operations to manipulate 3D objects represented with triangle meshes. Additionally a good documentation is available on their website (`https://pytorch3d.org/`). Some tutorials explain the power of Pytorch3D. Currently, there is no support for ray traycing renderers. The focus is on rasterization with different shaders.

Pytorch3D is extendable to implement different Differentiable Renderer. It is easy to implement a SoftRas renderer as explained in section 2.2 and compute gradients with the autodifferention from Pytorch. The following parameters can be optimized with the framework:

- camera, extrinsic values e.g. rotation, translation of position
- camera, intrinsic values
- meshes, properties e.g. vertices, faces, normals

A class called "PointLights" can be defined to represent lightsources in the scene with: "PointLights(location=(0,0,0))". This command creates a point light at location $[x, y, z] = [0, 0, 0]$. Optionally, the light has some parameters to define ambient, specular and diffuse component. There is also a class called "FoVPerspectiveCameras" to define the

camera. Other classes exists for creating meshes, renderer, shaders, losses, visualizations, etc. All of them are easy to define similar to the "PointLights" class. The Tutorials (`https://pytorch3d.org/tutorials/`) are really helpful, if someone wants to start with Pytorch3D.

The library has also some predefined functions for loading objects and creating meshes. Some objects exist as toyexamples to play around with Pytorch3D.

Kaolin: A PyTorch Library for Accelerating 3D Deep Learning Research

Koalin from [6] is the second library, which will be described here. It is also built on Pytorch and looks promising to handle 3D deep learning approaches. An extension to the Pytorch Dataloader and Dataset makes it easy to load 3D data, similar to load MNIST data in Pytorch. Kaolin supports polygon meshes, pointclouds, voxel grids and depth images. Kaolin is developed for accelerating the field of 3D deep learning research. It is not only a Differentiable Renderer, it provides also 3D datasets, large amount of different models, loss functions for 3D tasks, several DR (e.g. Neural Renderer, Soft Renderer and DIB-R), a functionality to visualize 3D results and it is highly optimized. Very comfortable looks the class "DifferentiableRenderer". This python class provides a modular opportunity to use different shaders, lighting, rasterization and projection methods.

One main key feature of Kaolin is the ability to visualize all important 3D data representation (triangles, voxels and point clouds). This is a powerful tool to inspect the produced data and to get more insights in general. Kaolin seems to be a good alternative to Pytorch3D.

3 Research in computer vision using Neural Network architectures

Since 2010, a significant increase in research for computer vision using Neural Networks can be obtained. One of the most relevant progress is made, because of the ImageNet project [15]. A provided dataset from ImageNet is used to train a neural network to classify images and the ImageNet challange becomes very popular in the field of computer vision. A lot of research and new architectures are announced since the start of ImageNet and the project allows to compare the different neural network architectures in a meaningful manner. A test server evaluates the error made from different approaches. For computer vision tasks with the use of Neural Networks ImageNet is a good way to find some useful techniques. Figure 3.1 shows the state of the art models by evolving in time.

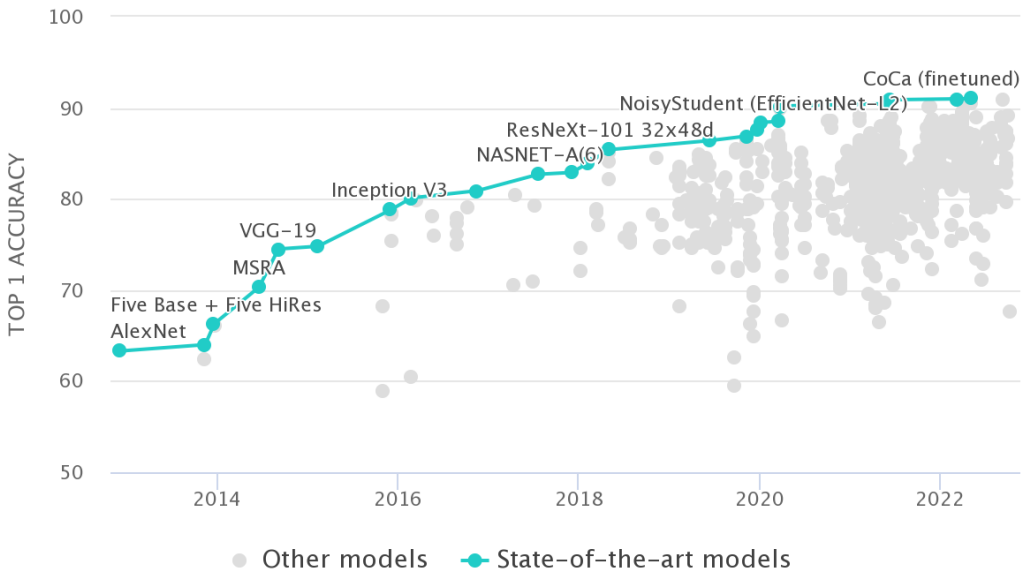


Figure 3.1: Top 1 accuracy, image classification on ImageNet dataset. Image from website `https://paperswithcode.com/sota/image-classification-on-imagenet` (accessed 05.12.2022)

On closer inspection, a trend in development is due to very deep neural networks and to high complex models. However, large neural networks are more difficult to train and tend to overfit. The problem of overfitting is often addressed by using dropout [[16]]. Additional, a large neural network needs dramatically more computational resources. Increasing the network size lead to an inefficient use of computing power [[17]]. Therefor some techniques are introduced and one of the most important methods were "Residual Networks" (ResNet) and the use of "inception modules". A significant increase in accuracy on ImageNet classification was made with introducing them.

3.1 Training deep neural networks with residual learning framework

Deep neural networks consists of dozens or even hundreds of stacked layers. As already obtained early for reccurrent Neural networks and LSTM networks [19] vanishing/exploding gradients could hinder learning. This problem was already solved by normalized initialization ([20], [21]). However, a so called degradation problem appears for deeper neural networks, which is not caused by overfitting (more details [18]). Therefor, a solution was introduced by identity mapping (also called skip connections or shortcut connections). Figure 3.2 shows one building block with identity mapping. The input is elementwise added to the output, if the input size and output size are the same. In case for different sizes, a linear projection matrix is multiplied to get the correct size. The effectiveness of Residual networks can be shown by comparison between two very similar networks. A plain network and a residual network with the same layers stacked together are compared. The only difference between them, is the identity mapping made by a residual network. The mentioned architecture for plain and residual network is depicted in [18]. Figure 3.3 shows the improvement using a residual network with its identity mapping.

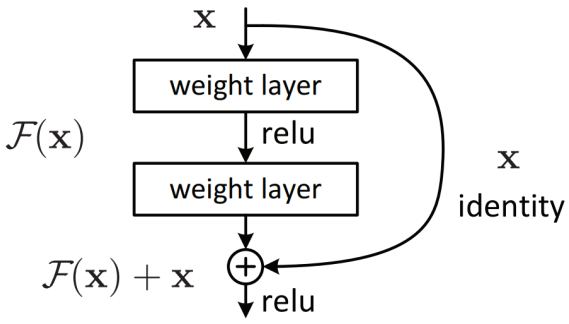


Figure 3.2: Residual learning: a building block with identity mapping. Figure from [18]

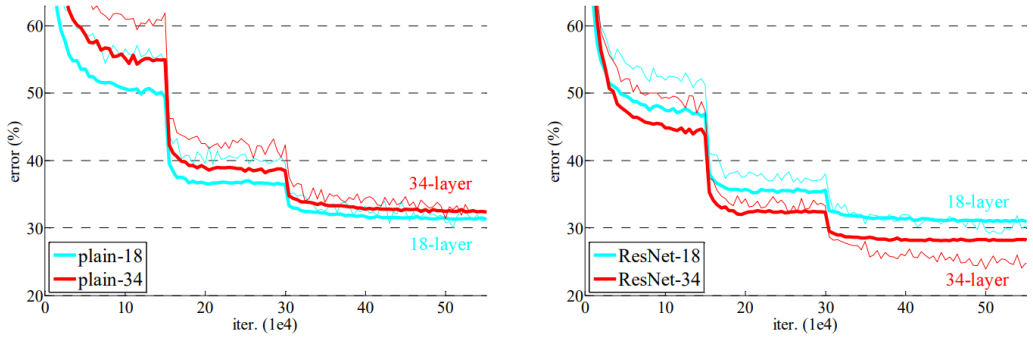


Figure 3.3: Training on ImageNet. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts. Figure from [18]

3.2 ResNext: Further improvements of residual networks

An improvement to residual networks is an aggregated residual network (ResNext) introduced by [22]. The main difference is the grouped (aggregated) convolutions, which are calculated in parallel. A computation of several convolutions layers side by side (parallel) was also made for AlexNet [23], which was the winner of ImageNet competition in the year 2012. However, ResNext uses grouped convolution operations to split a convolution with a high amount of channels to smaller groups with less channels. Experiments (figure 3.5) shows better results for classification of the ImageNet database. The concrete mechanism for a ResNext-block is shown in figure 3.4. One important hyperparameter for the neural network architecture is the cardinality, which stands for the amount of convolutions in parallel. ResNext has one big advantage to the networks introduced before. The design is more general and a few hyperparameters control the architecture of the model e.g. the parameters cardinality. Before ResNext the deepness (stacked layers) of

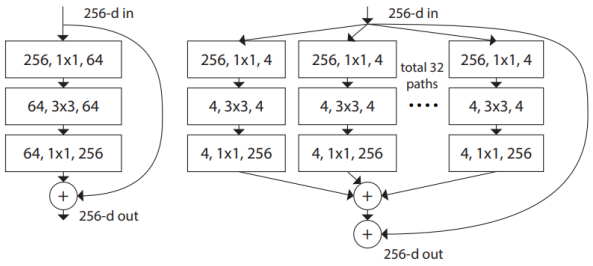


Figure 3.4: Left: A block of ResNet. Right: A block of ResNeXt with cardinality = 32, with roughly the same complexity. A layer is shown as (# in channels, filter size, # out channels). Figure from [22]

a model was a crucial metric. After ResNext it was shown, that the width (controlled via cardinality) is also important to consider.

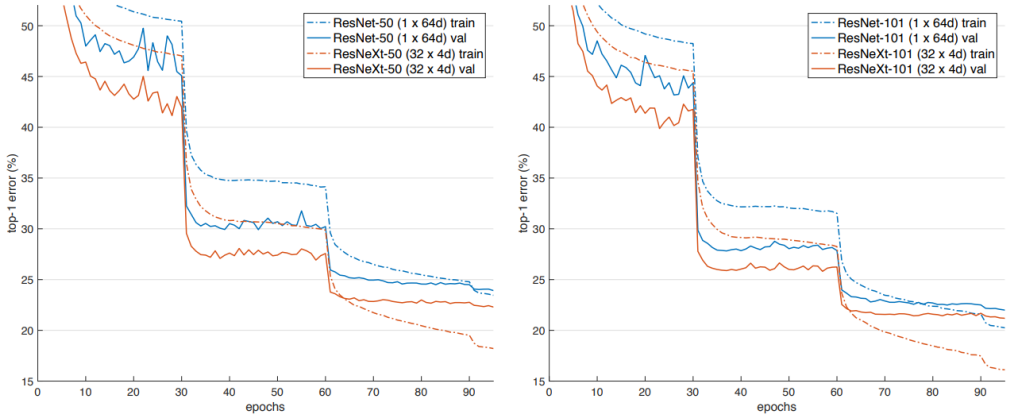
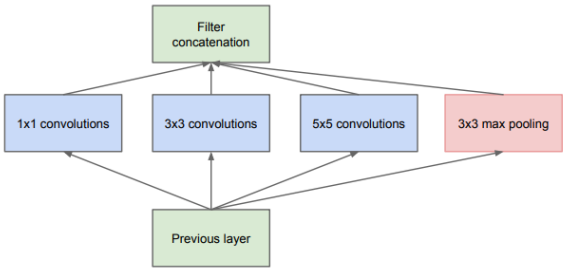


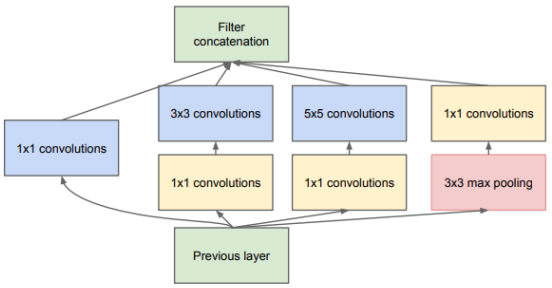
Figure 3.5: Training curves on ImageNet-1K. (Left): ResNet/ResNeXt-50 with preserved complexity (∼4.1 billion FLOPs, ∼25 million parameters); (Right): ResNet/ResNeXt-101 with preserved complexity (∼7.8 billion FLOPs, ∼44 million parameters). Figure from [22]

3.3 Inception Network: Going deeper with Convolutions

The inception modul uses different kinds of parallel convolutions e.g. filter size, dilation rates, etc. to learn features at multiple scales. The approach was introduced in the year 2014 [24] and has been widely used in many applications, including image recognition, object detection, and segmentation. Figure 3.6 shows the idea of inception, a distinction is made by the use of dimensionality reduction.



(a) Inception module, naïve version



(b) Inception module with dimensionality reduction

Figure 3.6: Inception module. Figure from [24]

3.4 CoCa: Contrastive Captioners

Contrasitive Captioners (CoCa) uses a contrasitve learning method to improve image to text models. The approach is mentioned in [25]. In image captioning, the goal is to automatically create a language description of an picture based on content. The CoCa network approach involves training a model to generate text for an image, as well as a set of negative text that are not related to the image. The network is then trained to minimize

the similarity between the created text and the negative text, which should help the model learn to generate more precise and descriptive text descriptions.

The approach of CoCa differs in its design compared to the methods before and it is not easy applicable for the cabin-cap problem.

4 Methods for surface prediction with a Freesurf sensor

A rendering function needs some parameters as additional input to work in a correct manner. Before training a neural network the parameters have to be determined. Here counts, a better parameter estimation leads to a more accurate result of normal vectors at the end. The following parameters have to be determined before training a Neural Network for predicting normal vectors:

Table 4.1: scene and material parameters

parameter	**variable**	**dimension**	**trainable**
roughness	p_{ro}	$\in \mathbb{R}$	yes
diffuse	p_d	$\in \mathbb{R}$	yes
reflectance	p_{re}	$\in \mathbb{R}$	yes
material $= (p_{ro}; p_d; p_{re})^T$	p_m	$\in \mathbb{R}^{3\times1}$	yes
shadow effects from sensor	p_s	$\in \mathbb{R}^{12\times H\times W}$	no
light position	p_l	$\in \mathbb{R}^{3\times12}$	yes
camera position	p_c	$\in \mathbb{R}^{3\times1}$	no

The scene parameters in table 4.1 are essential to train a Neural Network for estimating the normal vectors with a differentiable rendering function and they are sensor specific parameters (differs from sensor to sensor). This work is a detailed description with implementation to get all necessary parameters from scratch and train a neural network for estimating the normal vectors of a cabin-cap surface given images taken from the Freesurf sensor. The first step before training a neural network is to estimate some of the scene parameters. It involves also machine learning techniques to get them. The material parameters and shadow effects are completely unknown. An initial guess exists for all light positions and camera position. Table 4.1 displays one column named "trainable", which is an indication how to evaluate the parameter. The parameters which are marked

as "yes" will be optimized with methods described in section 4. Everything marked with "no" will be calculated (shadow effects: section 4.5, camera position see construction plan or table 4.2) and no optimization is happening to determine them.

4.1 Scene Parameter Optimization

Before training a Neural Network to predict normal vectors with images, the scene parameters from table 4.1 must be known. Some parameters can be calculated or determined and other parameters have to be optimized via an optimization loop as shown in figure 4.1. Every parameter which is marked as "yes" (table 4.1) cannot be computed easily. However, with an optimization loop it is possible to get the unknowns. A requirement for the iterative approach is a sufficient enough guess (difference between real parameter and guess/initial values should be low) for the initial values. The documentation for Pytorch3D [2] presents a method for optimizing scene parameters. The optimization loop in figure 4.1 is inspired from the Pytorch3D example. Nevertheless, the rendering function differs from the approach of Pytorch3D and also which parameters should be optimized.

A rendering function used in this project takes the scene parameters and the object structure as an input and predicts twelve images (Freesurf sensor example), where every image corresponds to one active light source. As seen in section 7, the object structure is at the beginning flat and every pixel value has a height of zero. After some iterations the structure deforms in a correct way to produce similar images as the ground truth. An optimization is only possible, if all scene parameters have a suitable initial guess. Material parameters, light positions and other scene parameters can also be optimized via this optimization loop. The error is calculated with a mean squared error function (MSE-Loss) between the predictions and the ground truth images. Back propagating the MSE-loss leads to optimizing the object structure and mentioned scene parameters. The described optimization loop is an approach to identify all necessary scene parameters for training a neural network and predicting normal vectors given the twelve images from the Freesurf sensor.

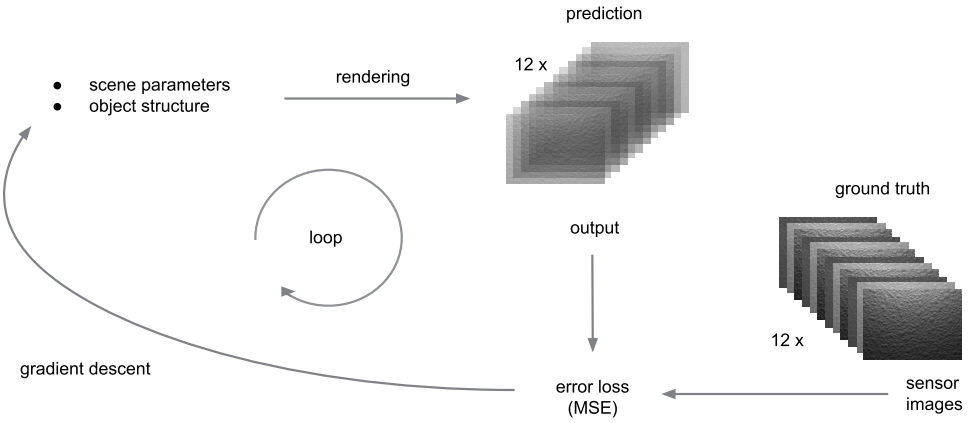


Figure 4.1: optimization loop to determine unknown parameters

Data and Setup

Figure 4.2 shows three images, which are output samples from the Freesurf sensor. Each column displays one image from three height levels and every level has four lightsources. First row in figure 4.2 shows the original images without any preprocessing and the second row illustrates a cropped version of the original images. Cropping the images is an important step for the methods later, because images from level 2 and 3 are effected from hard shadows. A hard shadow occurs, if objects hinder the light rays to reach the surface. As seen in the original images, hard shadow is occurring at the edge, because of the Freesurf sensor design. The cropped images are used later as an input for optimization and to identify scene parameters according to table 4.1.

Altogether, the sensor outputs twelve images per cap-sample. Every image is produced with one active light source. The position of every light source is known from the construction plan of the sensor and it is also an important scene parameter as well as the exact position of the camera. Unfortunately, the practice shows some difference between the calculated light sources and the real light source position. The reason for the difference is the design of the sensor. Mirrors are reflecting the light rays to the cap-surface. If the mirrors inside the sensor have a small variation in angle, the light rays will be reflected

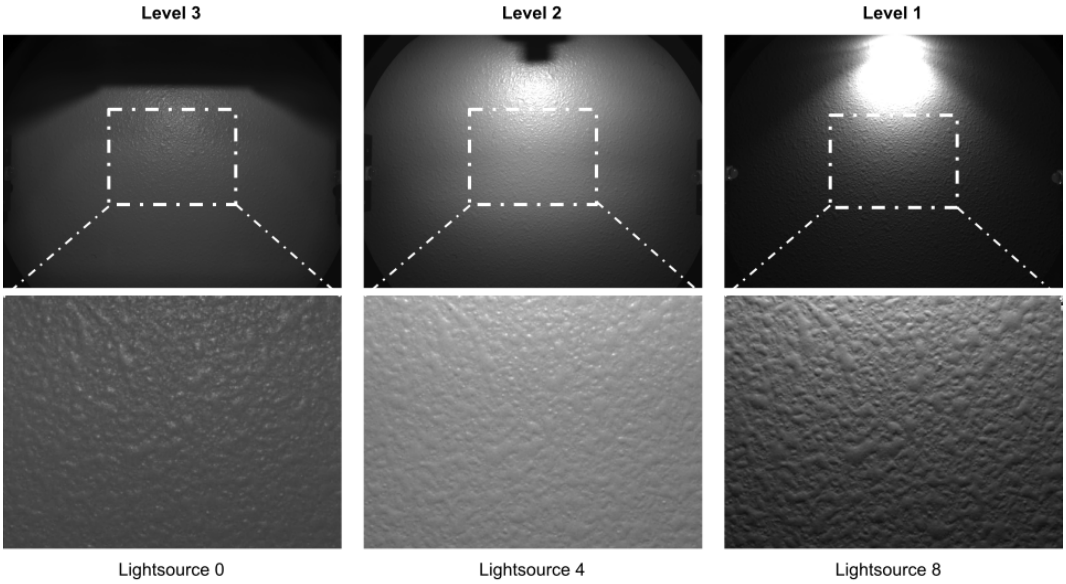


Figure 4.2: samples from Freesurf sensor

somewhere else. However, the known camera position can be used in practice for our rendering model later.

Figure 4.3 displays the positions of lights, camera and the cap-material. Diameter d_i corresponds to height h_i. Each height level consists of four light sources and the angle between them is 90 degrees.

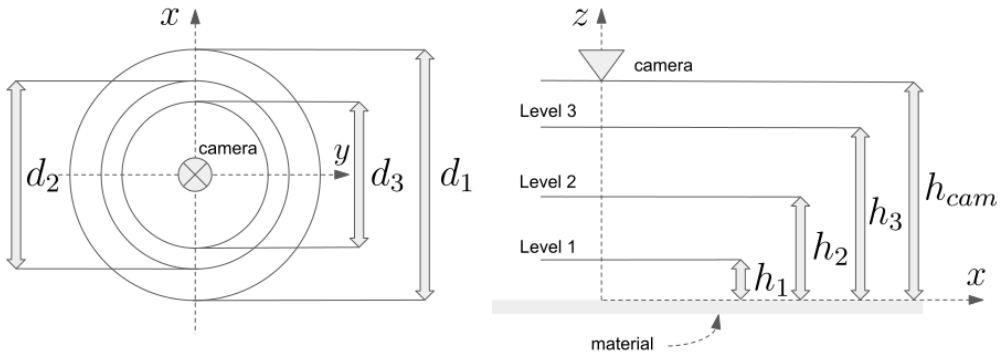


Figure 4.3: measurements inside the Freesurf sensor - camera and light positions.

Table 4.2: light and camera positions

component	coordinates	component	coordinates
$L0$	$(0, -r_3, h_3)^T$	$L7$	$(-r_2, 0, h_2)^T$
$L1$	$(r_3, 0, h_3)^T$	$L8$	$(0, -r_1, h_1)^T$
$L2$	$(0, r_3, h_3)^T$	$L9$	$(r_1, 0, h_1)^T$
$L3$	$(-r_3, 0, h_3)^T$	$L10$	$(0, r_1, h_1)^T$
$L4$	$(0, -r_2, h_2)^T$	$L11$	$(-r_1, 0, h_1)^T$
$L5$	$(r_2, 0, h_2)^T$	camera	$(0, 0, h_{cam})^T$
$L6$	$(0, r_2, h_2)^T$		

Table 4.2 represents the location of every light source and the camera position. The coordinate system for the table is depicted in figure 4.3. Lightsources from L_0 to L_3 are located at height level 3. The indices in coordinates represent the different height levels e.g. h_i and r_i stands for the i-th height level and for completeness $r_i = d_i/2$.

The mentioned positions are sensor specific parameters of the Freesurf sensor. Sensor parameters cannot be changed and they are fixed from sensor to sensor. A sensor parameter depends on the construction of the sensor. Even if two sensors exist, the sensor parameters can differ from each other. The reason is some discrepancy in production of the Freesurf sensor. As already mentioned before the sensor produces some shadow effects, because of using mirrors for reflecting light rays. The shadows are also a sensor specific parameter and will be calculated. Light sources have a light intensity in addition, which is usually important to determine for the rendering model. Intensity is the last sensor specific parameter for the Freesurf sensor. However, the light intensity can be neglected as seen in section 4.5. Having all sensor parameters as accurate as possible leads to a better result in rendering.

As a side note, distortion effects and other camera specific imprecise estimations are neglectable for the Freesurf sensor. Inspections before taking the sample images have led to the assumption.

4.2 DR setup for Freesurf sensor

Assuming a relative flat surface with the Freesurf sensor simplifies the rendering pipeline (figure 2.1). The prediction of the normal vectors does not need a rasterizer, because no occluded faces even exists. The decision to skip the rasterizer means a constraint in use cases. The built renderer is only able to predict surfaces with no occlusions, but it makes the whole rendering function more efficent. The suggested method below works only for surfaces with no occlusions. If the method is also used for a more complex surface with occlusions, a rasterizer as mentioned above must be added to the rendering pipeline before shading and it is not guaranteed to produce the same quality of results.

4.3 Data representation for 3D scenes

Three main representations are used to represent 3D data. Each representations has it's advantages and disadvantages. There is one big question about which data representation for 3D objects is more suitable combined with Differential Rendering. The three representations are: point clouds, voxels and polygon meshes.

Point clouds do not have surfaces, which makes it difficult to apply shading and lightning techniques on them. However voxels consist of surfaces, but they are not scaleable. Voxels have many parameters to represent a 3D object compared to polygon meshes. A polygon mesh can represent a large triangle with only 3 vertices and 1 face (parameters). It has also a surface and it is possible to apply simple translation and rotation operation on it. Because of the advantage compared to the other data representations for 3D scenes, polygon meshes are mostly used for Differential Rendering. More details about the different representations for 3D objects can be found at [13].

The setup with the Freesurf sensor allows even a more compact representation of the surface object (cabin-cap) as a polygon representation. Bumpmapping is a technique to reduce parameters and represent surface structure on a given object. [5] uses bumpmapping to represent surface structure on a given object and perform some calculations on GPU. Inspired from the idea bumpmapping is transformed to represent a cabin-cap sample. The inspired 3D data representation is called pixel-to-height, which describes the method exactly. Every pixel in the image is getting one height value. This height represents the

z-value of every pixel. Figure 4.4 illustrates the idea of a pixel-to-height representation. Usually the height H and the width W is the same as in image space e.g an image with 512×256 is equal to 131.072 height values. The variable $\boldsymbol{S}$ stands for the surface matrices, where every entry corresponds to one height value. A pixel-to-height representation needs per data point only one value in 3D space instead of 3 values in general. If the pixel distance p_d is known, the x- and y-coordinates can be easy calculated and only one parameter (height) is unknown. This object representation of the cap material simplifies the optimization problem in general, because each 3D point has only one value to optimize.

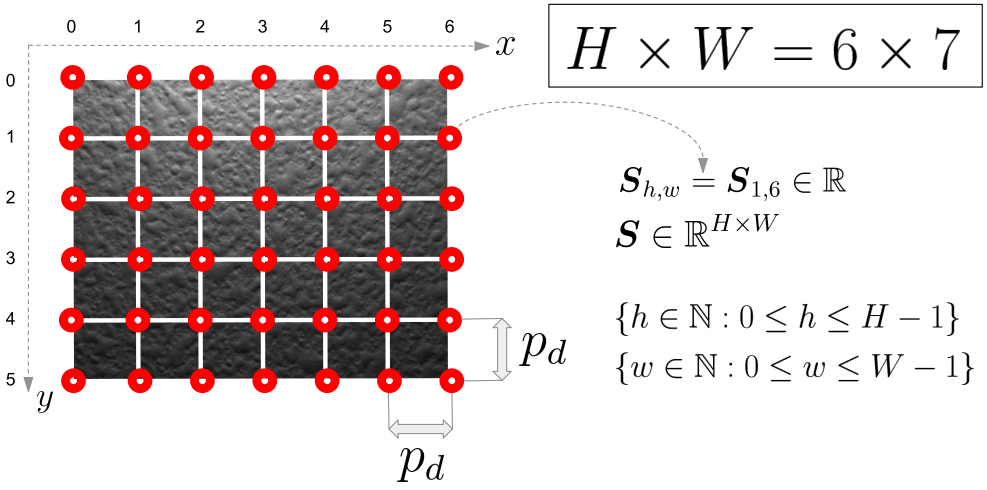


Figure 4.4: pixel to height representation - each red circle is one height value

4.4 Filament Renderer

This work uses parts of the Filament Renderer [4] from Google. The Filament Renderer is a pyhsically based renderer (PBR) adopted by Google to perform efficiently on mobile platforms. It combines two important properties in the field of DR. The Filament renderer is "physically based", which means it is more photo realistic, and it is efficient for mobile devices. Both features are important for this work. The official repository on Github `https://github.com/google/filament` is written in a large part with the programming

language "C++". The software for our project is written in python and uses the library pytorch for autodifferentiation, which means the original code from Google cannot be used. Therefore, for our project only the shading part is translated into python, because rasterization is not relevant. The Filament renderer has also a rasterizer model, if occluded faces appear.

```
# from filament light_punctual.fs/getDistanceAttenuation()
def get_distance_attenuation(
        posToLight: torch.Tensor,
        inv_falloff: torch.Tensor):
    distanceSquare = (posToLight ** 2).sum(dim=-1, keepdim=True)
    attenuation = get_square_falloff_attenuation(distanceSquare,
    inv_falloff)
    # Assume a punctual light occupies a volume of 1cm to avoid a division
    by 0
    return (attenuation / torch.clamp(distanceSquare, min=1e-4))
```

Listing 4.1: translated Python code example

A translated code example into python with pytorch tensors is depicted in listing 1. The translated python function has always a comment before, which indicates the original source code. The whole source code can be found at [4]. All necessary functions from the Filament renderer to solve the task for predicting the normal vectors are translated into python code in file named filament.py similar as the code part in listing 4.1.

4.5 Shadow effects

The Freesurf sensor has 12 light sources and they are located at known positions (table 4.2). The light sources emit light rays, which are not a perfect point lights. However, the Filament renderer assumes perfect point lights as input. A shadow mask helps the renderer to simulate shadow effects in practice. Shadow effects occur e.g. by reflecting light rays from mirrors. Every Freesurf sensor has its own shadows and they can be calculated with several image samples. An image sample contains 12 images corresponding to twelve light sources. The determination of shadows is more accurate, if more image samples exist. Figure 4.5 shows the Gaussian filtered median over all image samples for lightsource 7. Lightsource 7 has the most shadow effects compared to all other lightsources. The Gaussian filtered median gives more insights about shadow effects per lightsource. The shadows are computed with all sample images and every lightsource is linked to one shadow mask. Figure 4.6 illustrates the process to compute a Gaussian filtered median for one light source. Every image from different samples has several pixel values, the Gaussian filtered median is calculated with 2 steps as shown.

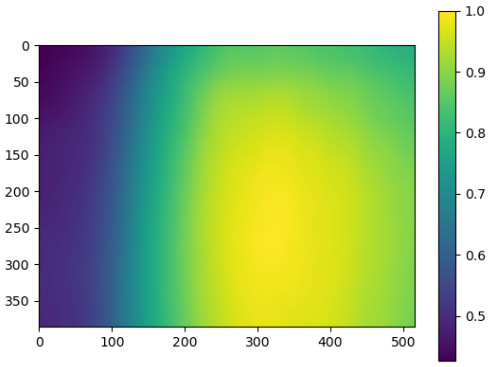


Figure 4.5: Gaussian filtered median calculated over all image samples - lightsource 7

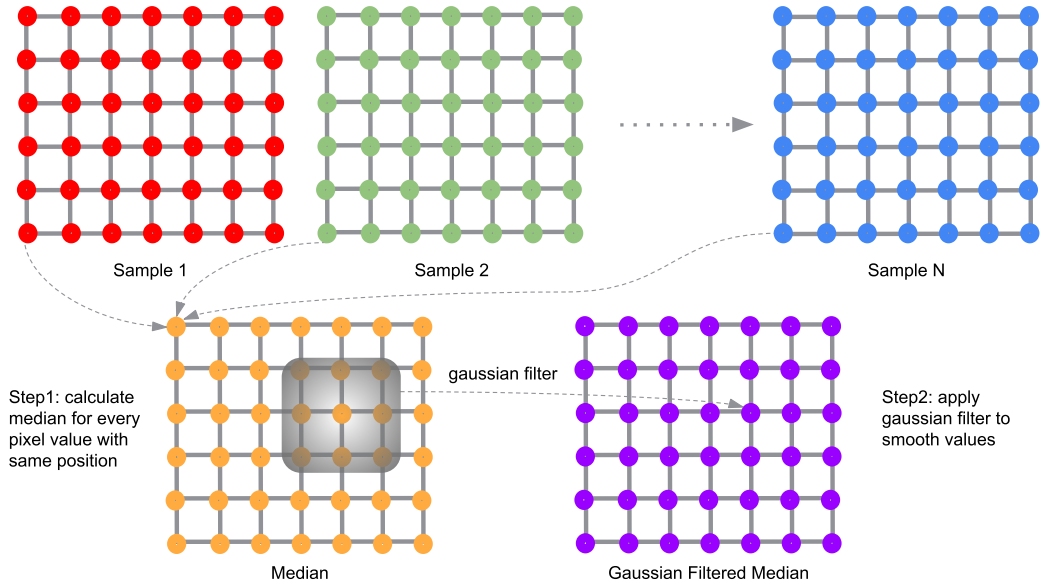


Figure 4.6: Gaussian filtered median (gfm) calculated with N image samples - calculation example for one specific lightsource

The shadow effects $p_s(L)$ for a specific lightsource L is proportional to the Gaussian filtered median $gfm(L)$. The prediction $pred_0(L)$ is the output of the Filament renderer for lightsource L, where 0 stands for zero height values. $pred_0(L)$ is the illumination result for a flat surface with a perfect point light. Equation 4.1 calculates elementwise the ratio between the illumination in real (Freesurf sensor, gfm) and a perfect point light ($pred_0$). The ratio is equal to the shadow effects. The point light illumination from the Filament renderer is converted to the real illumination,if the shadow effect p_s is multiplied with the output of the Filament renderer.

The Gaussian filtered median gfm is calculated with a python function called "getGfm(...)". More details can be found in section 5.

$$p_s(L) = \frac{gfm(L)}{pred_0(L)} \tag{4.1}$$

The shadow effects $p_s(L)$ is multiplied with the output of the Filament renderer and is important to simulate the shadows from the Freesurf sensor. If no shadow effects are

applied to the Filament renderer, the renderer is only able to render images with point lights and the prediction is not as accurate as possible.

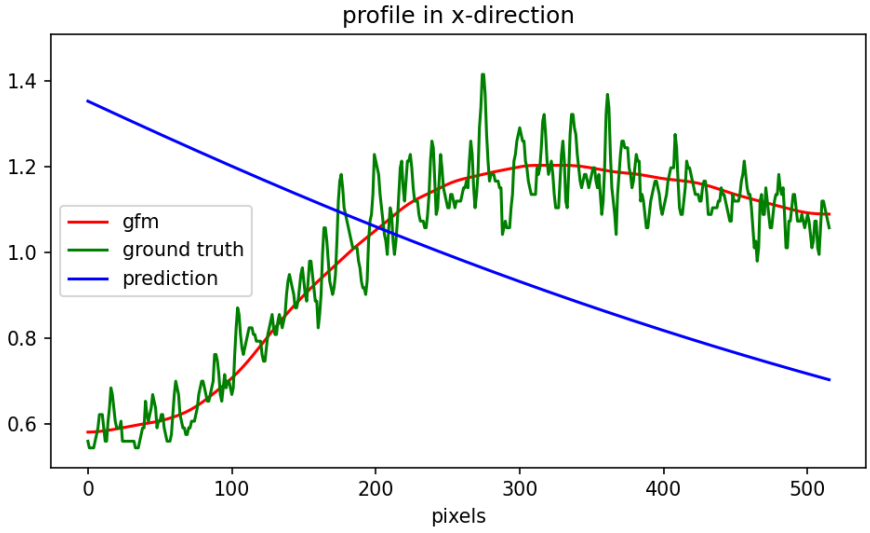


Figure 4.7: profile in x-direction with active light source 7

As mentioned before with lightsource 7 occurs the most shadow effects. Figure 4.7 displays a profile in x-direction. This profile is important to see clearly the shadow and it is used later on to find initial guesses before optimizing some parameters. The blue line called "prediction" is the intensity for a flat surface with zero height values at every pixel in x-direction without applying shadow on it. It simulates a perfect point light source, which is in practice not happening. The green line named "ground truth" demonstrates one image sample and the intensity of every value for a real image taken by the Freesurf sensor. The last red line called "gfm" represents the Gaussian filterd median and its values for every pixel. If the red and blue lines are not the same, shadow effects are the cause. By applying the shadow mask to the prediction from the Filament renderer, the blue and red line will get the same values, if all height values are zero. A mathematically proof is given in equation 4.2. The example seen in figure 4.1 is a prediction of a surface with zero heights, thus $pred^{'}(L) = pred_0^{'}(L)$. This condition results in $pred_s(L)$ (blue line) must be the same as $gfm(L)$ (red line) after applying the shadow effects.

Neglecting light intensity after shadow effects

A very important additional fact for applying the shadow effects to the output of the Filament renderer is that light intensity can be neglected in the whole rendering pipeline. The mentioned shadow effects include also information about the light intensity and therefore the intensity for every light source is not fundamental anymore. The shadowed output prediction $pred_s(L)$ after multiplying shadow effects for a specific active light source L can be written as

$$pred_s(L) = \overbrace{pred'(L) \cdot p_i}^{pred(L)} \cdot \frac{\overbrace{gfm(L)}^{p_s(L)}}{\underbrace{pred'_0(L) \cdot p_i}_{pred_0(L)}} = \frac{pred'(L) \cdot gfm(L)}{pred'_0(L)} \tag{4.2}$$

The output prediction of the filament renderer $pred(L)$ is proportional to the light intensity p_i, this allows to write p_i separate as described in equation 4.2. The symbol $'$ illustrates a version without dependency on light intensity. Overall, the shadowed output prediction $pred_s(L)$ is independent of light intensity p_i.

4.6 Calculation of pixel distance p_d

The pixel distance p_d is essential for calculating normal vectors and distance vectors. Given the construction plan, the distance d_i between light sources is known. Fortunately, the Freesurf sensor produces sample images, where lights at level 1 can be seen in the image (figure 4.3). Figure 4.8 is a sample image from level 1 - two light sources can be seen in middle right and left. Counting pixel between these two light sources leads to a certain amount of pixels, additionally the distance in real space is known. Some basic math operations calculate the pixel distance p_d.

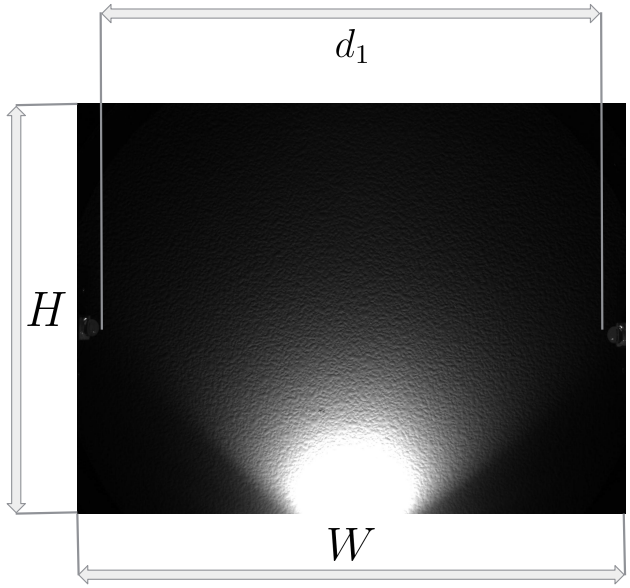


Figure 4.8: Original level 1 image sample with light source 9 and 11, d_1^{real} stands for d_1 in real space and d_1^{image} for image space. p_d is the ratio between real space and image space.

4.7 Important vectors for shading

Three vectors are essential as arguments for the shading technique used in Filament Renderer. Figure 4.9 illustrates these vectors in a point (pixel) on the material (surface). Normal vector N is perpendicular to the tangent plane, view vector V is the vector from a pixel on the surface to the camera (eye) and light vector L from the pixel to light source.

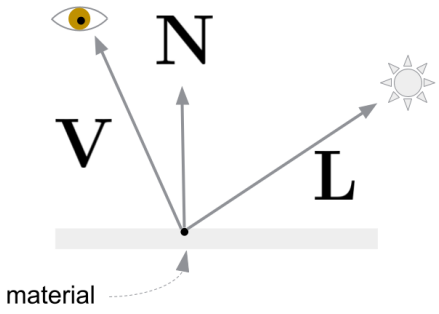


Figure 4.9: basic vectors used for different shading techniques: normal vector **N**, light vector **L** and view vector **V** (camera)

Determination of normal vector N

The surface matrix **S** represents all values in pixel-to-height representation. Converting matrix S into normal vectors is essential to apply shading techniques from the Filament renderer. Normal vectors from the surface are used as argument for Filament renderer and the task in this work is also to find

the normal vectors of the cabin-cap object. A fundamental math rule [26] computes normal vectors with equation 4.3.

$$\mathbf{N} = \begin{pmatrix} -\partial \mathbf{S}(x,y,z)/\partial x \\ -\partial \mathbf{S}(x,y,z)/\partial y \\ 1 \end{pmatrix} \tag{4.3}$$

Calculating partial derivatives for surface matrix $\mathbf{S}$ and concatenating them would solve the problem. Partial derivative is defined as a limit like ordinary derivatives. Let be $S : \mathbb{R}^3 \to \mathbb{R}$ a pixel-to-height surface function and $x, y, z \in \mathbb{R}$. The partial derivative [27] with respect to x for one specific pixel at position i, j is defined as

$$\frac{\partial \mathbf{S}_{i,j}(x,y,z)}{\partial x} = \lim_{h \to 0} \frac{\mathbf{S}_{i,j}(x+h,y,z) - \mathbf{S}_{i,j}(x,y,z)}{h} \tag{4.4}$$

where the limit to zero demonstrates a perfect partial derivative. The pixel-to-height representation allows a minimum h from one pixel to its neighbour, therefore h is equal to the pixel distance p_d. If the resolution is high, the approximation of the partial derivative is sufficient enough. All calculations are similar with respect to y, thus the normal vectors of pixel i, j could be calculated.

A python function called "get_normals(...)" calculates normal vectors in parallel. Section 5.2 views code snippets.

Determination of view vector V and light vector L

Calculating vector V and L is relatively simple and it is the same procedure for both of them. Given the pixel distance p_d and the surface matrix $\mathbf{S}$, it is possible to compute all 3D values for every pixel. The resulting matrix $\mathbf{S}^{3D}$ consists of $H \times W$ points in 3D. View vector $V_{i,j}$ and light vector $L_{i,j}$ for pixel i, j can be calculated with

$$V_{i,j} = \mathbf{S}_{i,j}^{3D} - p_c \qquad L_{i,j} = \mathbf{S}_{i,j}^{3D} - p_l^k \tag{4.5}$$

where k stands for the selected light source and p_i is the location of camera or light source.

A python function called "get_vectors(...)" calculates view and light vectors in parallel for every pixel. Section 5.3 views code snippets.

4.8 Interdependencies between light positions, light intensity and material parameters

One of the most challenging task is to solve the interdependency problem in the optimization loop. As mentioned in section 1.2 some parameters influence others and therefor it is not easy to find a global minimum. First of all it is important to understand, why interdependence is happening between parameters. Afterwards there could be a solution to tackle this issue in optimization.

Reason for interdependencies

Light position and light intensity are interdependent, because both of them influence the intensity in a pixel. The distance $d_{i,j}^k$ between light source k and an pixel i, j is quadratically inverse proportional [28] to its light intensity p_i:

$$\frac{1}{(d_{i,j}^k)^2} \sim p_i \tag{4.6}$$

The intensity of pixel i, j can also be influenced via material parameters p_m. The material parameters consist of three values, which is roughness p_{ro}, diffuse p_d and reflectance p_{re}. These parameters are used as input for the Filament renderer and they are important to render the given object. A detailed documentation about the Filament renderer is provided in [4]. Practice and theory show also an interdependence in material parameters to

light intensity and light position. As a short example to understand the interdependence, imagine a constant light intensity and also a constant light position. The brightness of a pixel i, j could also be changed via material parameter change. This scenario is also seen in practice and it demonstrates the interdependency. Altogether, three parameters are interdependent, which leads to a system of 3 interdependent variables (IV). An optimization problem with IVs cannot find a local minimum easily, if optimization between IVs is happening in parallel. As a consequence, there is a need for 2 constraints to shift the 3 IVs to 1 IV.

Solution for interdependencies

Light intensity is not known and it is not used in the optimization method (see section 4.5). Multiplying the shadow effects to the output of the the Filament Renderer will compensate the uncertainty and ensures the right illumination. An initial guess for material parameters is made with a tool implemented in this project and shown in figure 4.10 called "parameter-manipulation". The manipulation face is produced via a python library called Matplotlib and it simulates the illumination of a flat surface (zero height values) material before optimization. Matplotlib is able to create an interactive figure with buttons, text fields and sliders. These components can be used for interactions to find initial guesses for some important parameters. Figure 4.10 consists of 6 different areas denoted from A to F. Finding suitable guesses for material parameters area A,B and D are important to consider. Field A consists of several "radio buttons" from 0 to 11 and these buttons correspond to the different light sources. Selecting a light source leads to the corresponding profile shown in field B. A profile in x or y direction illustrates relevant data for guessing initial values. Given a 2D matrix the profile in x-direction represents the middle row values and the y-direction stands for the middle column values. For that reason field B displays 3 profiles in x- and y-direction. The profile named "ground truth" illustrates the grayscale values of a real image, which should be optimized afterwards and "gfm" stands for the Gaussian filtered median. The third profile line named "prediction" symbolizes the output of the Filament renderer for a flat surface and with applying shadow effects on it. The prediction line can be manipulated with the sliders depicted in field D. The values from the slider are going into the Filament renderer as arguments and the prediction line will be shifted accordingly. Material parameters can change the output and therefore the prediction line. An inspection of the diagrams (field B) while moving the slider value shows the change in prediction. First step is to take a look at all light sources (field A) and change

the parameter sliders to reach on average the gfm line. Of course it is not exactly possible to find the best value, but it is sufficient enough and some correction is made via applying the shadow effect afterwards. The slope and curve of the prediction line can be changed with the slider "rough", "reflectance" and "diffuse". A change in these values results in a better estimation, if the slope/curve of prediction line and gfm line is somehow similar.

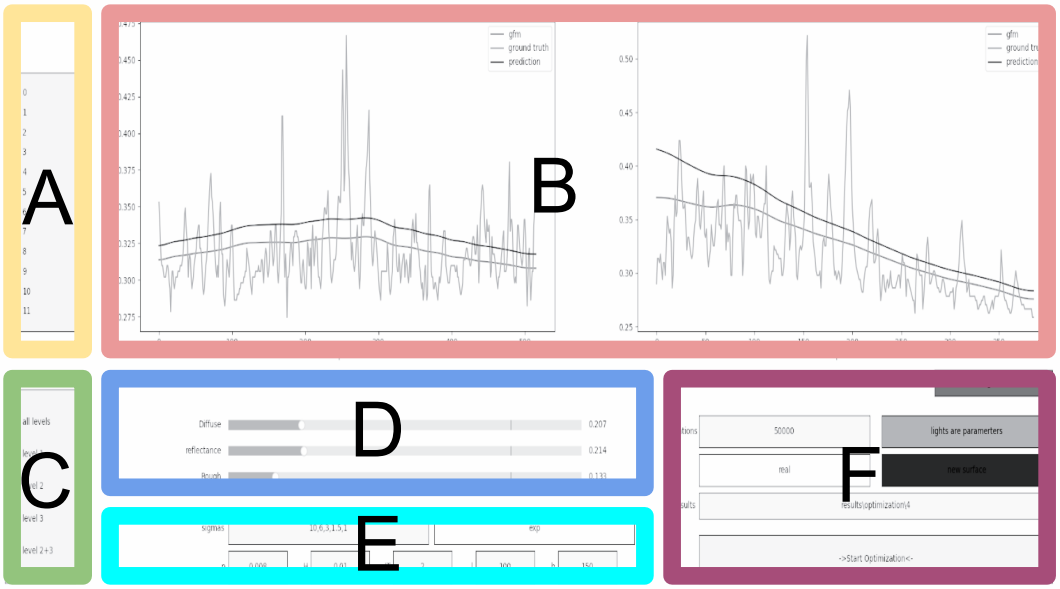


Figure 4.10: A: select light source, B: illumination profile, C: light source level selection, D: slider for material parameters, E: textfields for entering hyperparameters used for synthetic surface creation and button for changing regularization function for light positions, F: boolean button for optimizing light positions, boolean button for creating synthetic cap-samples, textfield for amount of iterations during optimization, textfield for saving results, button for creating new synthetic surface sample, button for starting optimization and button for change height-profile view to image view.

After the adjustments via sliders of material parameters, a sufficient guess for them exists. However, optimizing the parameters in parallel is still difficult, because of interdependeny. By multiplying the shadow effects the light intensity during optimization is not relevant anymore and therefore the system with 3 IVs becomes a problem with 2 IVs. The values for the material parameters have the most uncertainty, because in practice it is hard to adjust the 3 values and find an optimal guess. This is the reason for considering the third IV, which are the light positions. The initial guess for light positions can be seen in the construction plan (see figure 4.3). As mentioned in sections before, the light positions are not perfectly known, because mirrors reflecting the light rays and therefore the virtual

light positions have some discrepancy. However, the actual position of a light source is in practice located in the surrounding area of the perfect positions. A regularization term could handle this uncertainty and ensure light positions near to the actual location. Therefore the second constraint consists of regularization term, which forces light positions to be in a certain area via optimization. The regularization term is

$$\lambda \cdot F(\text{Norm}(L^k_{gt} - L^k_{pred}) \tag{4.7}$$

where λ is a hyperparameter between 0 to 1, F a function to choose from $F(x) = [e^x, x^2, |x|]$ (button in field F), Norm stands for the norm of a vector, L^k_{gt} is the constant position of light source k from the construction plan and L^k_{pred} stands for the prediction (during optimization) of light source k. The regularization can be seen as a soft constraint in the 3 IV problem. The 2 constraints above solve the interdependency problem.

4.9 Training of a neural network to predict cabin-cap surface structures

A trained neural network predicts the cabin-cap surface given images taken from the Freesurf sensor. However, training such a neural network is not possible with the real images from the Freesurf sensor, because there exist too less samples and the ground truth surface structure is not known. Therefor the second task is to find a workaround to train a neural network without having fair enough data provided.

For that reason synthetic surface samples are used to get more examples and to have a known ground truth for training. With the optimization loop explained in figure 4.1 and evaluation methods in section 4 all relevant scene parameters are known. The scene parameters combined with the Filament renderer (section 5.4) can be used as a transformation from surface structure to image representation. Figure 4.11 illustrates a training loop for a neural network given synthetic surfaces. If synthetic surfaces exist, it is possible to train a neural network with synthetic image samples. The transformation is able to transform a synthetic surface to a synthetic image and the gradients via backpropagation can be calculated with the use of pytorch autodifferentiation. The synthetic surface and

the predicted surface is represented with a pixel-to-height representation and the mean squared error (MSE) calculates the surface loss L^s between the two surface structures. The first intuitive idea is to train the neural network only with the surface loss L^s ($\sigma = 1$), but it turns out (section 7- experimental results) that the surface loss alone is hardly able to learn the correct surface structure. Therefor, an additional loss (gradients loss L^g) is added to the total loss via a weighted sum with the hyperparameter σ. Using a weighted combination of this two loss function lead to better results in general. However, the weighted sum introduces a new hyperparameter, which should be avoided if possible. The reason for a better result can be described due to information gain using also the gradients in the surface structure. Neighbouring pixel height values influence the surface slope.

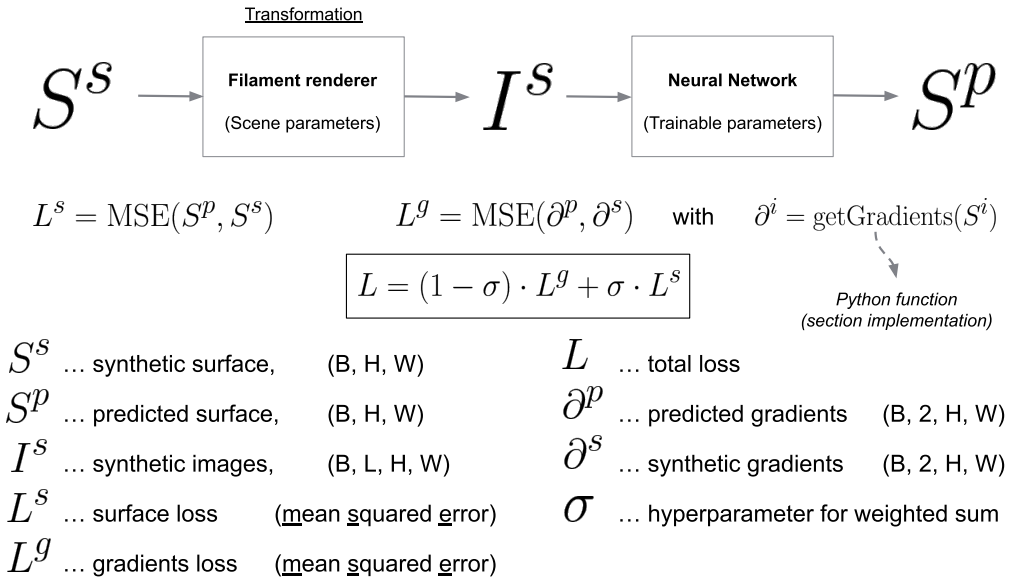


Figure 4.11: Training loop for a neural network in surface space to predict a cabin-cap surface. The Filament renderer is used as a transformation from surface structure to 12 image samples. The scene parameters are constant and they are used as additional arguments to render the images. Notation for shape sizes: 1) B: batch size, 2) H: image height, 3) W: image width and 4) L: amount of lights (default=12).

Beside the first intuitive idea of constructing a training loop in surface space as seen in figure 4.11, some other possibilities could be used as alternative approach for a training loop. Encoder-decoder architecture are often used to train networks [[29]] by using dimensionality reduction (latent space). The cabin-cap problem could be seen as a problem in encoder decoder style. Figure 4.12 shows an overview about the modified training

loop in image space. The image space is used as in- and output of the encoder decoder architecture and the latent space should represent the surface space in pixel-to-height representation. The encoder is a trainable neural network, which predicts the surface with image samples as input and the decoder is a transformation with the Filament renderer similar to figure 4.11. As seen in section 7 the experimental results with the modified encoder-decoder structure are significant better compared to the normal training loop as shown in figure 4.11 and the method intdroduces no additional hyperparameter. The main difference between the two training loops are the input spaces of the loss function, because the first training loop calculates the loss in surface space and the second (encoder-decoder) loop in image space.

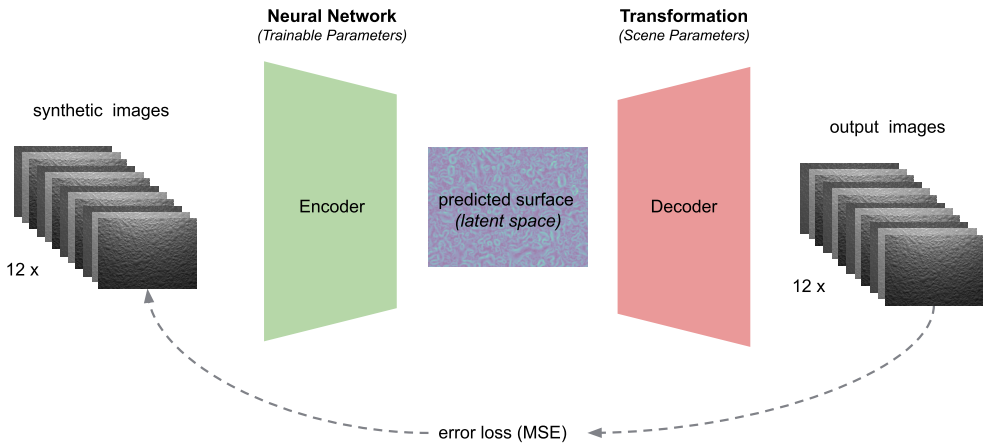


Figure 4.12: Training loop as encoder decoder network for training a neural network to predict a cabin-cap surface. The Filament renderer is used as a transformation from surface structure to 12 image samples. The scene parameters are constant and they are used as additional arguments to render the images. The synthetic images are created by applying the transformation to the synthetic surfaces.

As mentioned before the encoder decoder architecture (figure 4.12) with loss (MSE) function applied in image space performs better than the usual training loop (figure 4.11) with loss (MSE) function in surface space. The reason for the improvement is also due to information gain of the input by using the image space representation, because the surface representation uses only information about the height of every pixel. However, using

the image representation contains much more information about the illumination of the surface with different light sources and therefor the Neural Network is able to predict a better surface structure in general.

A critical part in the training loop is also the performance of the transformation and the quality of synthetic surfaces. The transformation depends on the scene parameters. If the scene parameters are close to the real values, the transformation with the Filament renderer lead to well transformed image samples. Additionally, the quality of a function, which is able to create synthetic surfaces similar to real surfaces is important to get training data. A probabilistic creation of surface samples lead to a infinite amount of surfaces and the neural network is able to generalize the problem.

4.10 Synthetic surface creation

As explained in section 4.9 the creation of synthetic surface samples is a critical mechanism while training a neural network. The spectrum of synthetic surfaces should overlap with the spectrum of real surface samples. As a convention a spectrum defines all possible surface structures, which occurs by creation (synthetic) or in real world (cabin-cap samples). Figure 4.13 shows the spectrum of synthetic and cabin-cap surfaces. If the spectrum of synthetic surfaces is higher than the spectrum of cabin-cap surfaces, the neural network generalize more and it is able to predict more accurate the surface structure given a cabin-cap image sample.

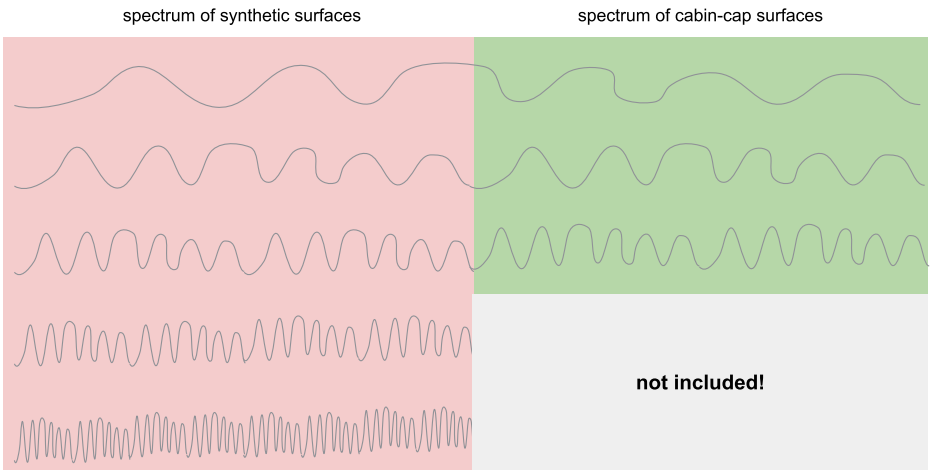


Figure 4.13: Surface spectrum with different frequencies - the cabin-cap spectrum is smaller than the synthetic spectrum.

The example with different frequencies should give more insights about the general surface spectrum. A surface spectrum depends not only on frequencies, the creation of surface is more complex. Inspecting the real cabin-cap images and first experimental results during optimization (step 1) give insights about important parameters for synthetic surface creation.

Random walk method

A combination of several small hills forms a flat surface into a cabin-cap surface. The hills has different heights and they are more or less expanded. Expanded means the amount of pixel, where one hill occurs. A hill could have any form and they are bulky. In addition, a hill could occur on other hills and they form a combination of hills. The customized "random walk method" is able to simulate the bulky hills. Figure 4.14 shows the bulkiness of several hills. As seen, the hills can have any form and large hills are around 50 pixels wide.

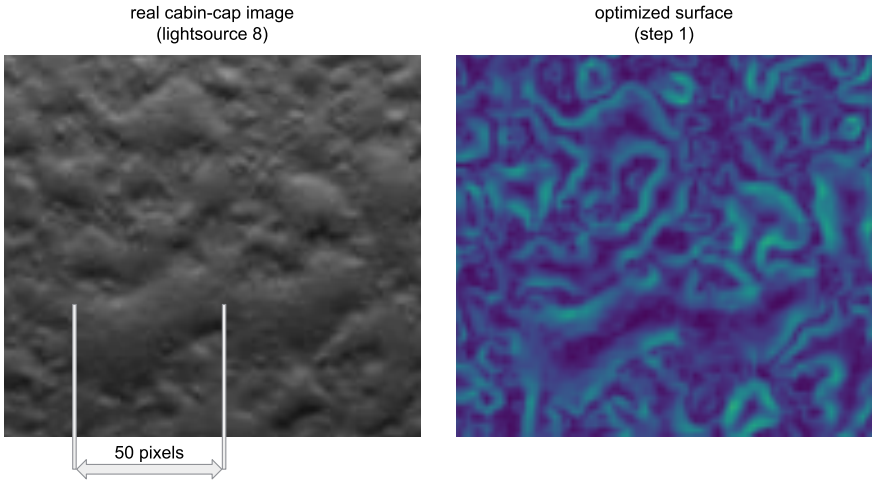


Figure 4.14: Illustration of bulky hills. Left image: real cabin-cap image taken with Freesurf sensor and active lightsource 8. Right image: optimized surface with optimization loop (figure 4.1) - if the surface slope is high, the pixel is brighter.

A creation of such bulky hills requires a customized function, which is able to create a high spectrum of different bulky expansions. A customized method called "random walk" is able to simulate similar expansions of hills as seen in figure 4.14. The random walk method distinguish between two different states. State 1 means a hill in place and state 0 stands for no hills. The returned result of random walk is a matrix with shape size $H \times W$ and states of 0 and 1. Figure 4.15 demonstrates the random walk method. Red pixels are in state 1 and white pixels in state 0. The state is changed from white to red, if the pixel is visited during the process. The random walk begins at the black point and the walk ends, if the amount of actions are equal to the amount of step size S. The next possible step consists of four different actions (left, right, up and down) to take and every action has a probability of 25 percent to take. As seen on the left side a random walk can have any expansion with one starting point. The variation is higher, if more starting points exist (right side), because they can interact with each other and form different expansions. The random walk is looped over more iterations e.g. 4 times and with different parameters, resulting in a high spectrum of variation for expansions. For getting a high spectrum of different expansions the step size S can vary between a value low l and high h (hyperparameter during creation of synthetic surfaces). A uniform distribution in each loop is sampled to

get different step sizes between l and h. The amount of starting points P depends on the amount of pixels and it is usually a percentage integer value over the amount of pixels.

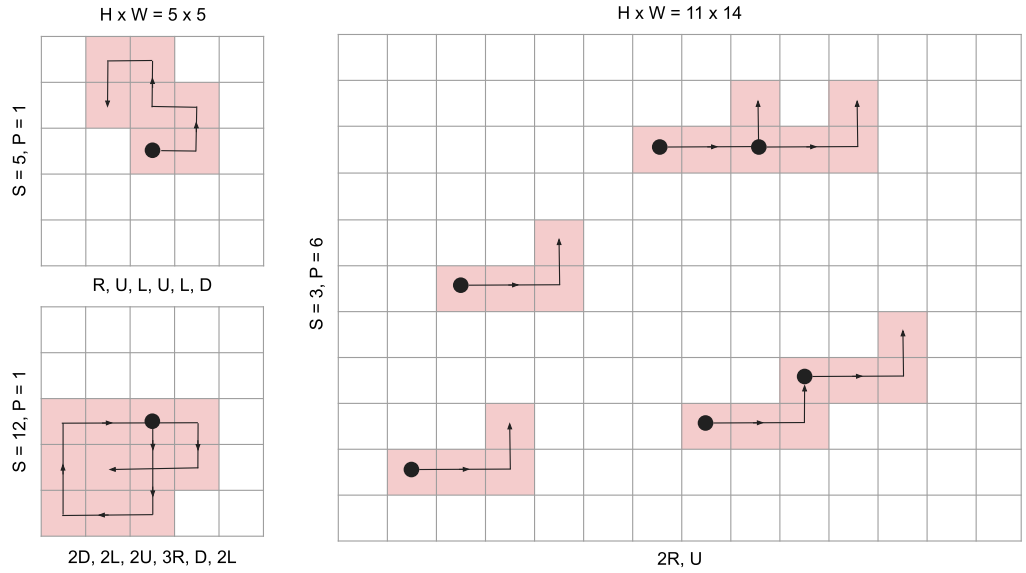


Figure 4.15: Demonstration of random walk method - Step size S means the amount of random steps from the black starting point and starting points P stands for the amount of starting points. The actions in direction right R, left L, up U and down D are random steps to the next pixel. Height H and width W controls the area size.

The spectrum of the random walk can be controlled via different hyperparameters:

- Low l and high h value of a uniform distribution $\mathbf{U}(l, h)$, which is used to sample the step size S (amount of actions). The uniform distribution has one special characteristic, because it is only able to sample integer values. Therefor no floating point numbers allowed. Before starting a random walk iteration the step size S is sampled from the uniform distribution.
- Probability p indicates the probability of a pixel to be a starting point P.
- The parameter I controls the amount of loops (iterations).

Examples in figure 4.16 show some outputs of a random walk method based on different hyperparameters. The variation of parameter probability p and iteration I results in more or less expanded bulky hills. The use of only one iteration I produces similar patterns for

every bulky hill, because all bulky hills are produced with the same random walk path and no other paths exist. If iteration I is getting higher, the bulky hills consists of hills produced with an aggregation of different random walk paths. The spectrum of bulky hills is higher, if iteration I is higher and also if probability p is higher. As already mentioned probability p controls the amount of starting points P. The more staring points exist in general, the higher is the chance for interactions between hills. An interaction of hills leads to a new form of hills and therefor to a higher spectrum. However, controlling the spectrum could also be done with parameter low l and high h, these parameters controls the step size S. A variation of l and h was not made for the random walk example, but they have also effects to the spectrum in creation. A small step size leads to very small expanded hill, which can also be seen in real examples (figure 4.14) and vice versa.

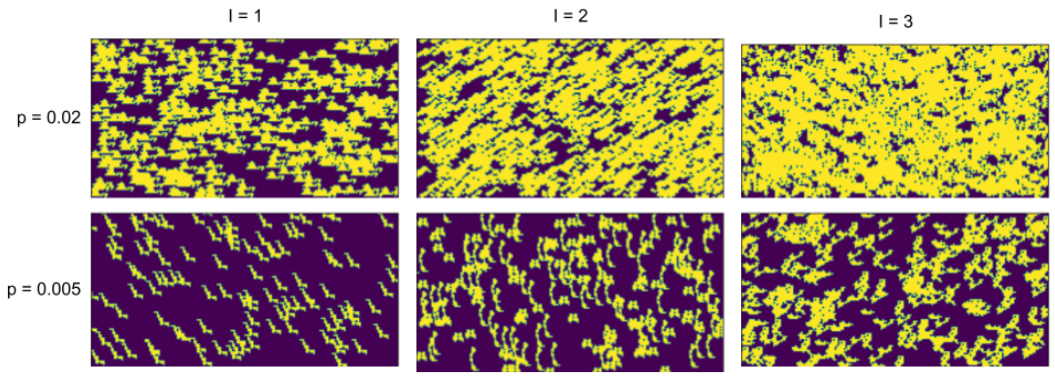


Figure 4.16: Hyperparameters for creation: $l = 49$, $h = 50$ (results in a fixed step size of 50), $H = 100$, $W = 200$, variation of parameters p and I as seen in image.

Bulky hills expansion to synthetic surface

The random walk method in section 4.10 produces a matrix with entries of 0 and 1, where 0 means no hill and 1 means a hill occurs. The output of this method is not a synthetic surface, it gives only insights about the occurrence of a bulky hill. A discrete step from 1 to 0 cannot be considered as a real surface. Nevertheless, the information about occurrence of a hill can further be used to create synthetic surfaces. Applying a Gaussian filter to the output of a random walk method lead to a continuous surface instead of a discrete step. Figure 4.17 shows two different profile heights, which are produced after applying a Gaussian filter. A profile height line represents the middle row values of a surface matrix. The standard deviation of a Gaussian kernel σ controls the smoothness between state 0 and state 1. If σ is high, the surface is smoother compared to a low σ-surface.

An aggregation between different σ-surfaces lead to a more realistic synthetic surface in general. The aggregated profile line on the right side demonstrates the use of combining different surfaces.

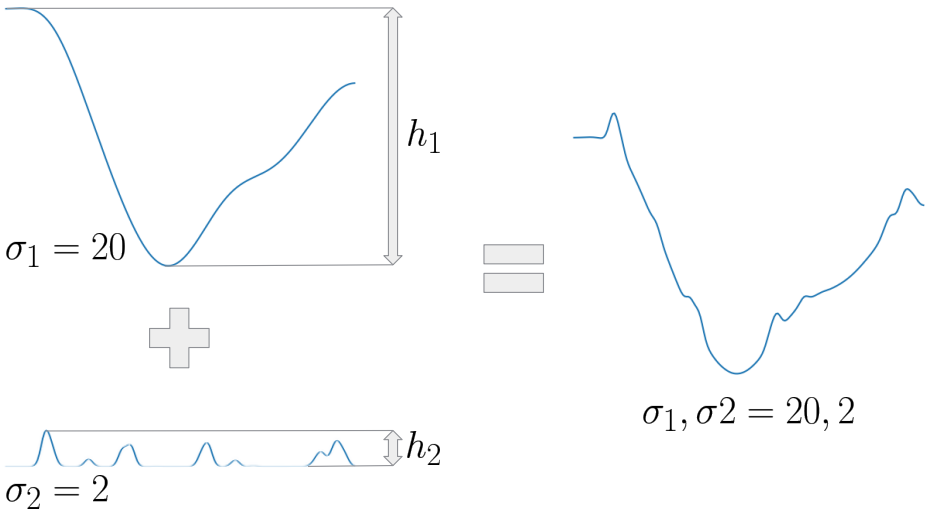


Figure 4.17: Aggregation of different σ-surfaces to produce a more realistic synthetic surface in general. The height of h_i depends on σ_i

As seen in figure 4.17 complex surfaces can be described with combinations of simple surfaces. A combination of different surfaces lead to a realistic sample, which is similar to a real cabin-cap sample by comparison. However, the aggregation of several simple surfaces has one special characteristic. The height h_i is set to σ_i, because a more expanded hill (high σ) could be higher as a small expanded hill (low σ). The practice shows better results with this assumption in synthetic surface creation. After aggregation of simple surfaces the maximum height is set to a similar value as the maximum of a cabin-cap surface. The optimization in step 1 give insights about the maximum height value h_{max} of a cabin-cap surface, which is around 0.01 cm. Altogether, a synthetic surface can be described with the following formula

$$\hat{S}(\sigma_1, \ldots, \sigma_N) = \left(\sum_{i=1}^{N} \frac{S_i(\sigma_i)}{\max(S_i(\sigma_i))} \cdot h_i \right) \cdot h_{\max} \tag{4.8}$$

with

$$S_i(\sigma_i) = \text{GaussianFilter}\left(\text{RandomWalk}(\ldots), \sigma_i\right) \tag{4.9}$$

Notation of variables

$\hat{S}$ ... a synthetic surface matrix in pixel-to-height representation of shape size $H \times W$.

$S_i(\sigma_i)$ ... a simple synthetic surface matrix in pixel-to-height representation of shape size $H \times W$. The matrix is created with a Gaussian filter and a random walk method. The three dots represent a placeholder for all parameters referred to the random walk method.

σ_i ... standard deviation of a Gaussian kernel, which controls the smoothness of bulky hills.

h_i ... maximum height value for a simple surface. (Default $h_i = \sigma_i$)

$h_{\max}$ ... maximum height value for an aggregated synthetic surface. (Default 0.01 cm)

Notation of functions

$\max(\ldots)$ ... returns maximum value of matrix with arbitrary size.

GaussianFilter$(\ldots)$ ... Gaussian filter function, which is able to smooth surfaces. More details can be found at [30].

RandomWalk$(\ldots)$ ... method as explained in section 4.10

Quality of synthetic surface creation

The creation of synthetic surfaces depends on many parameters and in practice it is hard to find the optimal parameters to create similar surfaces as a cabin-cap surface. However, the matplotlib figure 4.10 is able to compare synthetic images with real images while manipulating hyperparameters for synthetic surface creation. A detailed usage is described in section 6. Therefor, a way to find the right hyperparameters for the creation of synthetic surfaces could be made by comparison between real images and synthetic images. In practice it is hard to find the best parameters with a simple trial and error method.

Nevertheless, a high spectrum of synthetic surface creation could include samples similar to real surfaces. As explanation, a spectrum in case of synthetic surface creation means all possible synthetic surfaces, which can be created with synthetic surface creation. Some randomness is already included by design and more variation can be added on top by additional probability functions. An illustration (figure 4.18) between different spectrums could explain the idea in more detail. An initial guess for synthetic surfaces have to be sufficient enough to reach the real surface after expansion with probabilistic functions or a higher expansion (adding additional randomness) could be made to reach the spectrum of real surfaces. As seen on the right situation the spectrum of synthetic surfaces is not overlapping with the real surface spectrum. A solutions could be made with a better initial guess or a higher expansion.

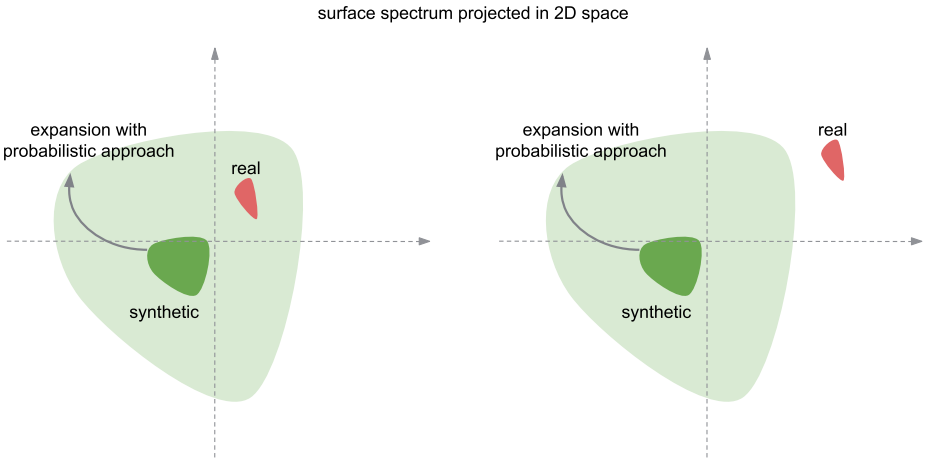


Figure 4.18: Illustration of a surface spectrum for synthetic (green) and real (red) surfaces. The spectrum is projected from a high dimensional space into 2D space. Left situation shows an overlapping situation and the right situation demonstrates a situation, where the real surface spectrum is outside the synthetic surface spectrum.

An expansion of synthetic surface creation is made via a probabilistic approach. As explained in equation 4.8 the creation depends on several hyperparameters (including also the parameters for "RandomWalk" function). The expansion could be made by adding a variation term to the chosen hyperparameters e.g. $h_{\max}$ is equal to $0.01 + v$. The variation v is sampled from a one dimensional normal function: $v \sim \mathcal{N}(\mu = 0, \sigma)$. Consequently, the expansion of a synthetic surface spectrum is higher, if more variation is added to the different parameters.

The initial guess is made with a comparison between real and synthetic surfaces using the matplotlib figure from 4.10. An example for comparison between synthetic images and real images is shown in figure 4.19. The first row shows three image samples produced via a synthetic surface and a discrepancy can be observed clearly. However, expanding the spectrum with a probabilistic approach could lead to a overlap between synthetic spectrum and real spectrum. If an overlap is reached, a trained neural network from figure 4.9 is able to predict real surfaces sufficient enough. With the use of a synthetic surface creation infinite samples are available for training.

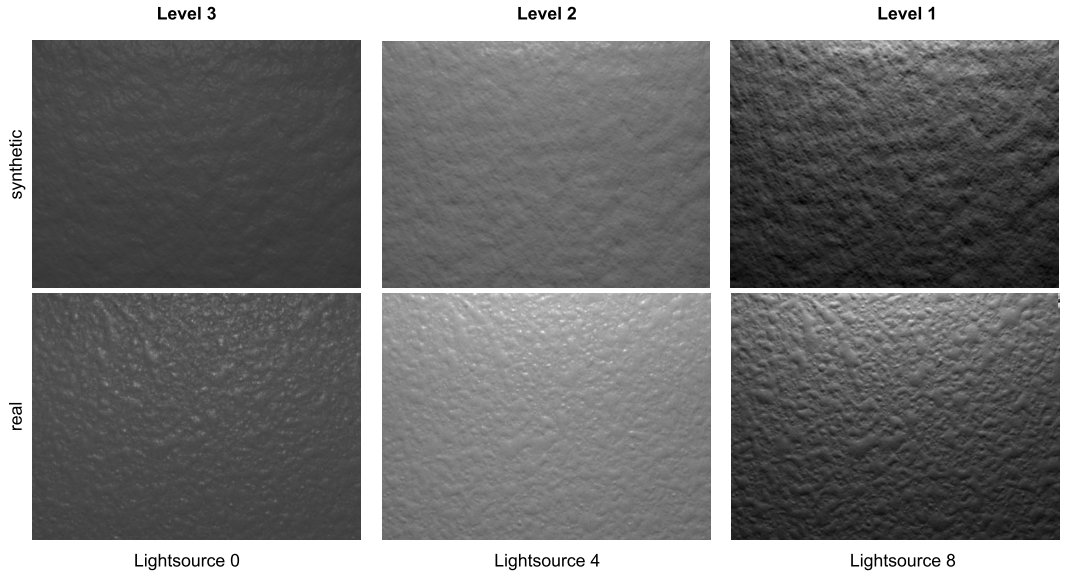


Figure 4.19: First row consits of synthetic images at 3 height levels created with the following parameters: $p = 0.008$, $h_{\max}$, $I = 2$, $l = 100$, $h = 150$, $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5) = (10, 6, 3, 1.5, 1)$. The synthetic surfaces are transformed with the Filament renderer (figure 4.9) into synthetic images. Second row demonstrates a real image example from a cabin-cap sample.

4.11 Execution time for different synthetic surface creations

The execution time for creating a synthetic surface is a critical value during training a neural network. The time for creation depends on the hyperparameters and the used hardware e.g. type of GPU, etc. Additionally, the implementation of surface creation is critical, because GPUs allows to parallelize operations and this ability can be used for creation to get a lower execution time. Table 4.3 shows the execution time of a random walk method as described in section 4.10. Starting points P and the process of random walk is parallized. Therefore, the execution time depends not on the probability p. As reminder probability p controls the amount of starting points P.

Table 4.3: Measured execution time for the random walk method. Probability p, iterations I and step size S are hyperparameters from 4.10. The table shows some execution times t for different variations of hyperparameters. Time t is the mean result over 100 measurements in milliseconds with the same value of hyperparameters. Used Hardware: 1.Processor: AMD Ryzen 5 5600H with Radeon Graphics, 3302 Mhz, 6 Core(s), 12 Logical Processor(s), 2.GPU: NVIDIA GeForce RTX 3060 Laptop GPU

p	I	S	t
0,005	3	50	16,55
0,005	3	100	17,91
0,005	6	50	33,43
0,005	6	100	37,37
0,01	3	50	16,18
0,01	3	100	17,68
0,01	6	50	35,26
0,01	6	100	36,15

A synthetic surface is created with the random walk method as seen in formula 4.8. Hence the execution time for surface creation depends on the random walk execution time and a additional part. The amount of standard deviations N_σ is crucial for the execution time, because the amount of summations corresponds to N_σ. Table 4.4 shows the execution time for surface creation, which depends also on N_σ.

Table 4.4: Measured execution time in milliseconds for synthetic surface creation. Probability p, iterations I, step size S and amount of standard deviations N_σ are hyperparameters from 4.10. The table shows some execution times t for different variations of hyperparameters. Time t is the mean result over 100 measurements with the same value in hyperparameters. Used Hardware: 1.Processor: AMD Ryzen 5 5600H with Radeon Graphics, 3302 Mhz, 6 Core(s), 12 Logical Processor(s), 2.GPU: NVIDIA GeForce RTX 3060 Laptop GPU

p	I	S	N_σ	t
0,005	3	50	1	17,01
0,005	3	50	2	52,54
0,005	3	100	1	19,71
0,005	3	100	2	57,03
0,005	6	50	1	36,56
0,005	6	50	2	90,49
0,005	6	100	1	40,41
0,005	6	100	2	95,07
0,01	3	50	1	18,45
0,01	3	50	2	52,64
0,01	3	100	1	18,96
0,01	3	100	2	57,35
0,01	6	50	1	37,55
0,01	6	50	2	88,91
0,01	6	100	1	37,75
0,01	6	100	2	94,50

As a summary about execution time for surface creation, the time for creating a synthetic surface is acceptably for training a neural network. The time for training different neural networks is less than 1 hour with the hardware as mentioned in table 4.3 and 4.4. The duration is observed from different test in practice.

4.12 Customized Neural Networks used for surface prediction

A lot of different computer vision tasks are widely solved and the ImageNet project mentioned in section 3 is prominent for its success. Some new ideas came up during the project time e.g. residual networks, inception modules, etc. [[15]] The design of a customized Neural Network for predicting the cabin-cap surface called "SurfaceNet" as shown in figure 4.9 is inspired from residual networks 3.1, which was a breakthrough architecture for the ImageNet project in the year 2016. Since 2016 a lot of novel approaches were introduced and therefore residual networks are not the most recent architecture in general. Nevertheless, residual networks are easy to design and the experimental results are satisfying as shown in section 7. Additionally, the chosen architecture should run on the available hardware, which consists of a GPU from NVIDIA (GeForce RTX 3060 Laptop).

The so called SurfaceNet was designed for a domain specific task and the general architecture is shown in figure 4.20. SurfaceNet has 12 input channels (input dimensionalty) corresponding to the 12 images taken from the Freesurf sensor and one output channel (output dimensionality), because it should predict a surface matrix in pixel-to-height representation. At the beginning there is always a layer with kernelsize 7 to transform the input dimensionalty to the dimensionality of mid channels C_{M} similar to [18]. After the first convolution layer L BlockNet modules are stacked together as shown and they consist of 3 dimensionalty reductions. A reduction in dimensionalty means a BlockNet layer with input channels of C and output channels of $C//2$. The operation "//" should indicate an integer division e.g. $5//2 = 2$. Every block, which is filled with red color indicates a dimensionalty reduction for the first applied BlockNet layer and a green block indicates no reduction in dimensionality. At the end a simple convolution with kernelsize 3 is made to reduce the channel size to one. The general architecture is designed to have a low amount of parameters to choose from, which is important to compare different

Neural Networks. Concrete 3 parameters can be seen as hyperparameter for building a SurfaceNet: 1) mid channels C_M, 2) amount of layers L and 3) BlockNet architecture.

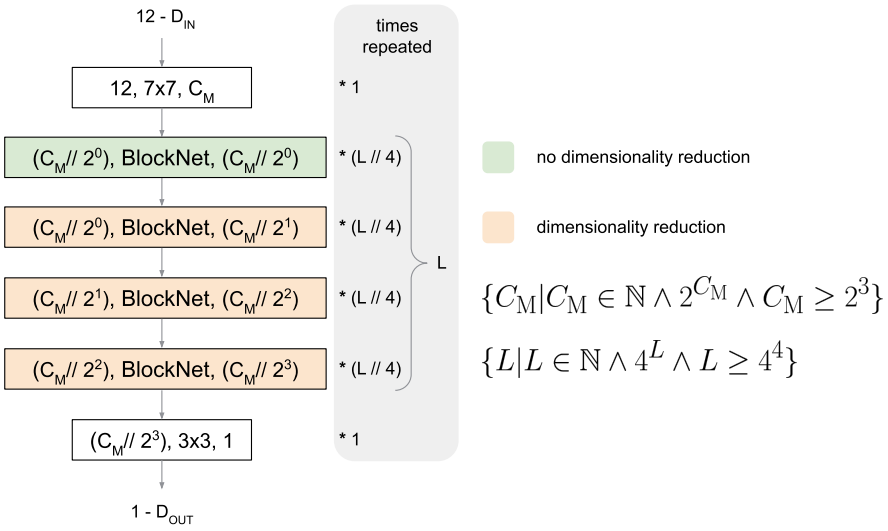


Figure 4.20: General architecture of SurfaceNet. C_M stands for the amount of mid channels between the convolution layers or "BlockNet" Network. The layers L determine the amount of stacked layers without the first and last convolution. A layer is shown as (# in channels, filter size, # out channel) and zero padding is always chosen to have no change in sizes between layers.

The BlockNet architecture is essential for the performance and some modules are more suitable than others. The idea for creating such a general architecture with modules (BlockNet) inside came from [18], [22] and [17]. SurfaceNet has two different BlockNet modules to choose from

1. ResNextBlock... from [22]. If cardinality is equal to 1, it can be considered as residual block introduced by [18] and
2. ConvBlock... same as Resblock without skipconnections.

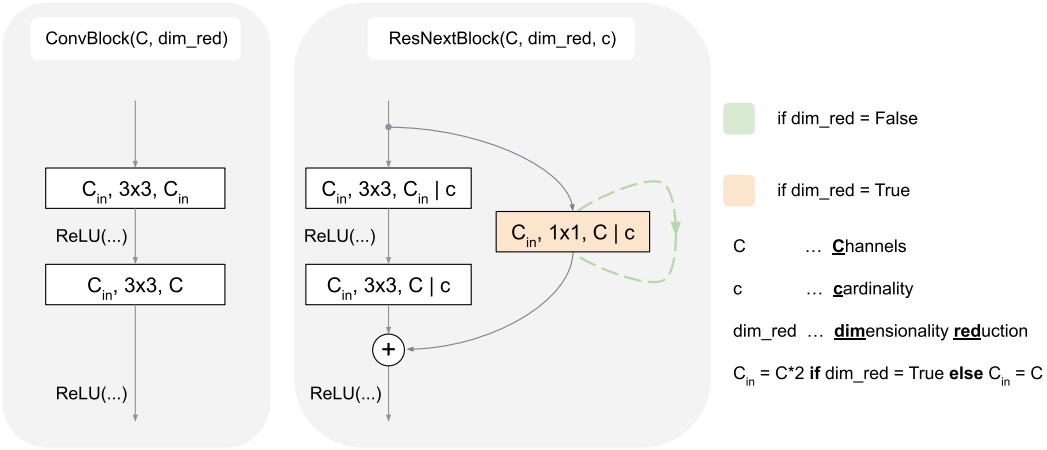


Figure 4.21: 2 different BlockNet modules: 1) ConvBlock and 2) ResNextBlock. ResNextblock is similar to the block introduced by [22]. A layer is shown as (# in channels, filter size, # out channel | cardinality c) and zero padding is always chosen to have no change in sizes between layers.

Figure 4.21 shows 2 different modules, which can be used as BlockNet architecture for SurfaceNet. The so called ConvBlock consists of two convolutional layers without any skipconnections and grouped convolutions as in [22]. It is used as baseline to measure the performance of adding skipconnections and grouped convolutions. The ResNextBlock has a hyperparameter called cardinality, which stands for the groups of convolutions. A cardinality of 1 is equal to the usual convolutional layer and the ResNext block can be used as ResNet block. A dimensionalty reduction means the amount of input channels are two times the output channels. This is done with a boolean flag called "dim_red". Additionally a 1 times 1 convolution must be added to reduce the dimension of the skipconnection. For visualization red and green color is used to distinguish between the two possibilities.

The SurfaceNet is a modular architecture designed for predicting cabin-cap surfaces and it has a small amount of hyperparameters. Depending on the available hardware a wider (increase of mid channels C_M) or deeper (increase of layers L) network could be used, which results in more complexity. However a too deep network could also avoid progress in learning depending on its initialization.

5 Implementation

The implementation for the cabin-cap problem to estimate the surface structure given images (from the Freesurf sensor) is coded with programming language python. The whole code is uploaded to the platform "Github". The repository is under `https://github.com/Kiesi1991/Surface-Reconstruction.git` available. In addition every function has a detailed description at the beginning, where all input arguments and the return variables are defined. The code listings has some invisible parts, which are not displayed. If code parts are hidden, three dots (...) symbolize a hidden code part. Additionally the shape of pytorch arrays has some variables, which are important to know:

H . . . height of input image

W . . . weight of input image

L . . . amount of light sources (Default=12)

B . . . batchsize during optimization (Default=1) and trainingt the SurfaceNet(...) (Default=2)

Below are important code samples to understand the overall idea of implementation.

5.1 Implementation of getGfm(...) function

The python function called getGfm(...) is used for calculating the Gaussian filtered median(gfm). It takes one input argument and outputs a torch tensor, which is the gfm. The input "path_real_samples" represents the folder path to real samples taken from the Freesurf sensor. Every subfolder represents one sample and the subfolder consists of 12 grayscale images in png format. Each grayscale image in a sample folder is numerated between 0 to 11 (e.g. "7.png"), the number stands for the active light source. The mentioned

folder structure is important for the scan(...) function (code line 7). The scan function scans all folders and creates a python dictionary. The keys of the python dictionary are strings from "0" to "11", corresponding to the active light source. The value from the python dictionary is a list with numpy arrays as elements. After scanning the real samples a for-loop calculates for each light source the median and applies the Gaussian filter afterwards. Variable gfm stores the results in a pytorch tensor and is returned at the end from the function. The Gaussian filter smooths a 2D image and the sigma variable controls the smoothness. The Gaussian filter is imported from the python library "scipy" and a detailed documentation is also available there.

```
1  def getGfm(path_real_samples):
2      '''
3      calculates gaussian filtered median (gfm) for samples in a folder
4      :param path_real_samples: (string), directory, which should be scanned
5      :return: (1, 1, H, W, L, 1), gaussian filtered median (gfm)
6      '''
7      images = scan(path_real_samples)
8
9      gfm = None
10     for im_nr in sorted(images.keys()) :
11         im = torch.from_numpy(gaussian_filter(torch.median(images[im_nr],
       dim=2)[0], sigma=10, mode='reflect'))
12         images[im_nr] = im
13         if gfm == None:
14             gfm = images[im_nr].unsqueeze(-1)
15         else:
16             gfm = torch.cat((gfm, images[im_nr].unsqueeze(-1)), dim=-1)
17
18     return gfm.unsqueeze(0).unsqueeze(0).unsqueeze(-1)
```

5.2 Implementation of getNormals(...)

The function getNormals(...) calculates normal vectors for each pixel in parallel given the surface matrix **S** and the pixel distance p_d. The function needs two arguments and outputs a pytorch tensor. The use of pytorch tensors allows to calculate normal vectors in parallel. Basic math operations of tensors are applied elementwise on GPU. As described in section 4.7, the implementation is straight forward. However, there is one detail to take

account of. Line 8 is calculating the difference quotient in x-direction from a pixel to its neighbour pixel in parallel. The implementation of calculating the result creates a matrix with a shape size of (B, H, W-1). To solve the discrepancy and get the original shape size, line 9 calculates the difference quotient from the other direction and line 10 concatenates both of them. The same procedure is deployed for y-direction.

```
1  def getNormals(surface, pd=0.0031):
2      '''
3      calculates normal vectors given a surface matrix.
4      :param surface: (B, H, W), surface matrix in pixel-to-height
       representation, every entry contains a height value in z-direction
5      :param pd: (float), distance between pixels
6      :return: (B, 1, H, W, 3), normalized normal vectors for every pixel
7      '''
8      dfdx = (surface[..., 1:] - surface[..., :-1]) / pd
9      dfdx1 = ((surface[..., -1] - surface[..., -2]) / pd).unsqueeze(2)
10     dfdx = torch.cat((dfdx, dfdx1), dim=2).unsqueeze(3)
11
12     dfdy = ((surface[:, 1:, ...] - surface[:, :-1, ...]) / pd)
13     dfdy1 = ((surface[:, -1, ...] - surface[:, -2, ...]) / pd).unsqueeze(1)
14     dfdy = torch.cat((dfdy, dfdy1), dim=1).unsqueeze(3)
15
16     z = torch.ones_like(dfdx)
17     normals = torch.cat((-dfdx, -dfdy, z), dim=3)
18     return normalize(normals.unsqueeze(1))
```

5.3 Implementation of getVectors(...)

This function calculates vectors between the pixel positions and a target position in a parallel way. Three inputs and an optional argument called norm is required. The argument surface represents the surface matrix **S**, targetLocation is the target position in 3D and the variable pd stands for the pixel distance p_d. Lines 13 and 14 calculate the x and y-position for each pixel height value (z-direction), given the pixel distance p_d. A simple elementwise difference between pixel positions and target position computes the vectors between them. The function normalize(...) returns the normalized vectors.

```
1  def getVectors(surface, targetLocation, pd=0.0031, norm=True):
2      '''
```

```
    calculates vectors between target positions and pixel positions.
    :param surface: (B, H, W), surface matrix in pixel-to-height
representation, every entry contains a height value in z-direction
    :param targetLocation: (1, (1 or L), 1, 1, 3), target position for
calculating vectors e.g. light positions, camera position
    :param pd: (float), distance between pixels
    :param norm: (boolean), if TRUE output is normalized
    :return: (normalized) vector between pixel positions and target
position(s)
    '''
    device = surface.device
    b, h, w = surface.shape

    X = torch.linspace(-(w // 2), (w // 2), steps=w).unsqueeze(0).to(device
) * pd
    Y = torch.linspace(-(h // 2), (h // 2), steps=h).unsqueeze(1).to(device
) * pd

    X = X.repeat(h, 1).unsqueeze(0).repeat(b, 1, 1).unsqueeze(-1).to(device
)
    Y = Y.repeat(1, w).unsqueeze(0).repeat(b, 1, 1).unsqueeze(-1).to(device
)
    Z = surface.unsqueeze(3)

    surfacePoints = torch.cat((X, Y, Z), dim=3).unsqueeze(1)
    V = targetLocation - surfacePoints
    if norm:
        return normalize(V)
    else:
        return V
```

5.4 Implementation of function filamentRenderer(...)

The function called filamentRenderer(...) calculates all necessary input variables, which are used for a function named evaluate_point_lights(...). evaluate_point_lights(...) applies the Filament renderer and outputs the rendered images as pytorch tensor. As mentioned above the python code of Filament renderer is a translated version from the original github repository [4]. Only necessary functions for rendering the cabin-cap images are used.

Everything, which is translated and used can be found at a file called filament.py. The perceptual roughness (line 18) is defined as the squared roughness and it is an important input for evaluate_point_lights(...). The parameter "f0" (line 19) is a different formulation about the material parameter reflectance and is also used as argument. The variable "light_attenuation" stands for the attenuation of light (line 25) and it is defined as the normalized inverse squared distance away from the light position. Additionally normal vector **N**, view vector **V** and light vector **L** are used as inputs or a combination (dot product, line 28 and 29) of them.

```
1  def filamentRenderer(surface, camera, lights,
2                       rough=0.5, diffuse=0.5, reflectance=0.5):
3      '''
4      calculates intermediate variables for Filament renderer and apply
       Filament renderer
5      :param surface: (B, H, W), surface matrix in pixel-to-height
       representation
6      :param camera: (1, 1, 1, 1, 3), camera position
7      :param lights: (L, 3), light positions
8      :param rough: (int(pytorch Parameter)), material parameter
9      :param diffuse: (int(pytorch Parameter)), material parameter
10     :param reflectance: (int(pytorch Parameter)), material parameter
11     :return: (1, 1, H, W, L, 1), rendered output of Filament renderer
12     '''
13     lights = lights.unsqueeze(1).unsqueeze(1).unsqueeze(0)
14     light_dir = getVectors(surface, lights, norm=False).permute(0,2,3,1,4).
       unsqueeze(1)#B,1,H,W,L,3
15     light_dir = tfunc.normalize(light_dir, dim=-1)
16
17     roughness = (torch.ones((1, 1, 1, 1, 1, 1)).to(surface.device) * rough)
       .to(surface.device)
18     perceptual_roughness = roughness ** 0.5
19     f0 = 0.16 * reflectance ** 2
20
21     N = getNormals(surface)[:, :, :, :, None, :]  # 1,1,H,W,1,3
22     V = getVectors(surface, camera).permute(0, 2, 3, 1, 4).unsqueeze(1)  #
       1,1,H,W,1,3
23     L_ = getVectors(surface, lights, norm=False).permute(0, 2, 3, 1, 4).
       unsqueeze(1) # 1,1,H,W,L,3
24
25     light_attenuation = 1/(torch.linalg.norm(L_, axis=-1, keepdims=True)
       **2)
```

```
26      L = normalize(L_)
27
28      NoL = (N * L).sum(dim=-1, keepdim=True)
29      NoV = (N * V).sum(dim=-1, keepdim=True)
30
31      return evaluate_point_lights(...)
```

5.5 Implementation of function optimizeParameters(...)

The function "optimizeParameters(...)" iteratively performs an optimization loop, apply plotting functions, initialize optimization model and store optimized parameters. It combines all important parts, which are necessary for optimizing a real cabin-cap sample.

1. preparations (line 4-10): This code block calls some important functions and defines necessary variables before optimization. Variable "samples" is a pytorch tensor of shape size (B, 1, H, W, L) and stores the samples from directory path "path_real_samples". Shape size B is equal to the amount of real samples in folder "path_real_samples" and every real sample is in one subfolder of "path_real_samples". This is an important structure for the called function "getRealSamples(...)". Function "getLightNumeration" returns the start and end numeration, which indicates the relevant light sources for optimization. It could be the case to optimize e.g. only with level 1 light sources. At the end of preparations a function called "getScene(...)" returns relevant variables (camera position, light positions and a surface matrix with zero height values) for starting the optimization. The camera and light position was derived from the construction plan.

2. model, loss function and optimizer (line 12-20): The class OptimizeParameters(...) creates a model for optimization. A detailed description about the class is given below. As criterium for optimization a MSE (Mean Squared Error) loss function is used, it compares every pixel value from the real sample images with the predicted samples.

3. optimization loop (line 22-44): The optimization loop consists of several important parts. The first part (line 26-32) is used for setting the gradients to zero for all light positions. This is important at the beginning, because before optimizing the light

positions the surface matrix and material parameters should be optimized. Both of them are unknown and for light positions there exists a good initial guess. After performing several iterations, the light positions are also important to optimize. This happens by setting the "requires_grad" value to True (line 30). Variable "err" calculates the error, which should be backpropagated and used for performing an optimization step. The error has two summed up parts: 1) MSE-Loss, 2) Regularisation term for light positions. An important constraint for solving the IV problem is a regularisation for light positions (see section 4.8). The variable "distance_err" stands for this regularisation and variable "lam" controls the influence of regularisation. If it is high, more penalty exists for light positions far away from the origin and vice versa. Line 40-42 performs the optimization step with setting gradients to zero. After the optimization step a hidden code part is not displayed in detail here. This hidden code part consists of several plotting functions to get intermediate results during optimization.

```
1  def optimizeParameters(path_real_samples, path_results, lr, iterations,
       selected_lights, para_lights, rough, diffuse, reflectance,
       regularization_function, lam):
2      '''(...)'''
3      (...)
4      # preparations before starting optimization
5      (...)
6      device = 'cuda' if torch.cuda.is_available() else 'cpu'
7      samples = getRealSamples(path_real_samples)
8      start, end = getLightNumeration(selected_lights)
9      camera, lights, surface = getScene(
10                                batch=samples.shape[0])
11     (...)
12     # define optimization model
13     model = OptimizeParameters(surface, (lights, para_lights), camera,
14                                rough=rough, diffuse=diffuse,
15                                reflectance=reflectance)
16     # transfer model parameters to "device"
17     model.to(device)
18     # define loss function and optimizer
19     mse = torch.nn.MSELoss()
20     optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_decay
       =0.0)
21     (...)
22     # optimization loop with gradient steps
```

```
23      for iteration in tqdm(range(iterations)):
24              pred = model.forward()
25              (...)
26              # set requires_grad value for light positions to False for
27              # the first iteration till iteration is grater equal to
28              # plot_every
29              if iteration >= plot_every and para_lights:
30                  model.lights.requires_grad = True
31              else:
32                  model.lights.requires_grad = False
33              # calculate errors between real cabin-cap images and
34              # predictions from Filament renderer
35              # + apply regularisation
36              err = mse(pred[..., start:end+1].to(device),
37                          samples[..., start:end+1].to(device))
38                          + lam * distance_err
39              # set gradients to zero and apply optimization step
40              optimizer.zero_grad()
41              err.backward()
42              optimizer.step()
43
44              (...)
45
46      return {...}
```

5.6 Implementation of nn.Module class OptimizeParameteres(...)

This python class inherits from the pytorch class nn.Module and consists of initialization, forward rendering pass and some plotting functions.

class OptimizeParameters(...) and def __init__(...)

The initialization is straight forward and consists of four main categories.

1. scene parameters (line 17-21): If a variable is defined with the function Parameter(...), it will be optimized during training. The light positions (self.light, line 22) can be trainable or not, it depends on the parameter mask from figure 4.10 - area E. The button "lights are parameters" is an input chosen from the user.

2. material parameters (line 23-26): Three material parameters are used for characterizing an object and they are all trainable.

3. Gaussian filtered median (line 28-29): The function called getGfm() returns the Gaussian filtered median. The variable self.gfm is important for calculating shadow effects in the forward rendering pass.

4. shadow effects (line 31-33): The variable self.shadowing is a boolean and it indicates, if shadowing effects are applied (True) or not (False).

```
1  class OptimizeParameters(nn.Module):
2      def __init__(self, surface, lights, camera,
3                   shadowing = True,
4                   rough=0.5, diffuse=0.5, reflectance=0.5):
5          '''
6          initialization of class OptimizeParameters
7          :param surface: (B, H, W), surface matrix in pixel-to-height
       representation
8          :param lights: (tuple) -> (lights, boolean), lights: (L,3) light
       positions for all 12 light sources, boolean: if True lights is
       Parameter else is not a Parameter
9          :param camera: (1, 1, 1, 1, 3), camera position
10         :param shadowing: (boolean), if True shadow effects are applied to
       output of Filament renderer
11         :param rough: (int), material parameter
12         :param diffuse: (int), material parameter
13         :param reflectance: (int), material parameter
14         '''
15         super().__init__()
16
17         # scene parameters
18         self.surface = Parameter(surface)
19         self.lights_origin = lights[0]
20         self.lights = Parameter(lights[0]) if lights[1] else lights[0]
21         self.camera = camera
22
23         # material parameters
24         self.rough = Parameter(torch.tensor(rough))
25         self.diffuse = Parameter(torch.tensor(diffuse))
26         self.reflectance = Parameter(torch.tensor(reflectance))
27
28         # gaussian filtered median
```

```
29          self.gfm = getGfm()
30
31          # shadow effects
32          self.shadowing = shadowing
33          self.shadow = None
34
35          (...)
```

def forward(...)

Line 10 calculates the mean of all height values for the surface matrix and takes the elementwise difference. This is an important operation to keep the average distance between camera position and pixel values at a constant level. The output is calculated with line 11-13 by applying the "filament_renderer(...)" function. If shadow effects are applied, the prediction $pred_0$ for zero height values are calculated with line 15-20 and afterwards line 21 calculates the shadow effects p_s.

```
1       def forward(self):
2           '''
3           forward rendering function with applying Filament Renderer.
4           :return: if shadowing=True the function outputs a rendered
5           pytorch tensor multiplied with shadow effects, else the
6           function outputs a rendered pytorch tensor without applying
7           shadow effects.
8           '''
9           device = self.surface.device
10          surface = self.surface - torch.mean(self.surface)
11          output = filament_renderer(surface, self.camera.to(device),
12                                     self.lights, self.rough,
13                                     self.diffuse, self.reflectance)
14          if self.shadow is None:
15              output0 = filament_renderer(surface,
16                                          self.camera.to(device),
17                                          self.lights,
18                                          # values could be changed
19                                          rough=0.5, diffuse=0.5,
20                                          reflectance=0.5)
21              self.shadow = (self.gfm.to(device) / output0).detach()
22          if self.shadowing:
23              return (output * self.shadow).squeeze(-1)
24          else:
25              return output.squeeze(-1)
```

def plotImageComparison(...)

The function called "plotImageComparison(...)" saves 12 images, corresponding to 12 active light sources. Every image has two subimages, where the left subimage shows a real cabin-cap sample image and the right subimage is a rendered image from the forward(...) rendering pass.

def plotDiagrams(...)

"plotDiagrams(...)" produces several images with insights during optimization. This function is called after some iterations and stores the plotting results in a given folder. The following diagrams are produced and stored:

angles.png: An image, where every pixel value demonstrates the angle between normal vector and a vector in z-direction (0, 0, 1),

error.png: Sum of MSE (Mean Squared Error) and distance (regularisation) error while optimization,

height-profile.png: A height profile in x- and y-direction (middle row and column of surface matrix $\mathbf{S}$,

l_to_origin.png: A line diagram with 12 lines corresponding to the 12 light sources, every line illustrates the distance between origin (position in construction plan) and current position.

l_to_zero.png: Same as l_to_origin.png, but instead origin distance is used, position zero (0, 0, 0) is used for calculating distance lines,

material-parameters.png: Change of material parameters while optimization.

def createParametersFile(...)

The created textfile with "createParametersFile" gives important information about concrete values during optimization. This file is also used later to compare different settings and to search hyperparameters. The file has always at the beginning a name and after that name, some values etc. exist to show their current state.

def saveParameters(...)

Saving intermediate parameters during training and at the end of optimization is an important procedure. The saved parameters are used in step 2 of the cabin-cap problem, which is training a neural network to predict normal vectors given some cabin-cap real sample images. Function "saveParameters(...)" saves all necessary parameters to perform step 2 and it stores also other parameters to get a better understanding, what is happening during training.

5.7 Implementation of randomWalk(...)

The randomWalk(...) function is the implementation of the random walk method, which was introduced in section 4.10. The function arguments are described in detail at listing 4.10. The step size S is sampled from a uniform distribution (line 16). Variable low l and high h corresponds to the minimum and maximum margin for the uniform distribution. Probability p is used for declaration of starting points (line 18-19), thus every pixel value has a probability p to be a starting point P. In addition, several actions (up, down, right, left) for a random walk could be chosen and a 25 percent chance is used to determine the path. The random path is made with line 21. Line 23-24 allows a parallel computation on GPU, because the matrix (called surface) is used to "walk" in parallel with as many parameters as needed. The surface matrix is zero padded to allow any direction to go without reaching the end of the matrix. The for-loop (line 26 -35) executes the random path and stores the states into the surface matrix. At the end (line 37) a clipping function is used to clip the maximum value to 1, because a state can be visited any times and if a state is visited, it should have a value of 1 instead of 0.

```
1  def randomWalk(size, p, I, l, h):
2          '''
3          perform random walk method for bulky hill expansions.f
4          :param size: (tuple) -> (H:int, W:int), size of output matrix
5          :param p: (float), percentage of starting points based on
6           amount of pixels (H*W)
7          :param I: (int), number of loops (iterations)
8          :param l: (int), minimum length of step size S
9          :param h: (int), maximum length of step size S
10         :return: (numpy array) -> (H:int, W:int) output matrix
```

```
11          '''
12          result = np.zeros(size)
13          for _ in range(I):
14              # Return random integers from `low` (inclusive) to
15              `high` (exclusive) for step size
16              S = np.random.randint(low=l, high=h)
17              # create starting points for random walk
18              starting_points = np.random.choice(np.array([1.0, 0.0]),
19                                      size=size, p=[p, 1.0 - p])
20              # sample S actions: 1=right, 2=left, 3=up, 4=down
21              actions = np.random.randint(4, size=S) + 1
22              h, w = size
23              surface = np.zeros((h+S*2, w+S*2))
24              surface[S:-S, S:-S] = starting_points
25              x, y = 0, 0
26              for action in actions:
27                  if action == 1:
28                      x += 1 # right
29                  if action == 2:
30                      x -= 1 # left
31                  if action == 3:
32                      y -= 1 # up
33                  if action == 4:
34                      y += 1 # down
35                  surface[S+y:(h+S+y), S+x:(w+S+x)] += starting_points
36              result += surface[S:-S, S:-S]
37          return np.clip(result, 0.0, 1.0)
```

5.8 Implementation of createSurface(...)

To create a synthetic surface with the use of random walk method it is necessary to perform several operations. The createSurface(...) function controls the expansion of the synthetic spectrum (line 21-26 and 40-42) as described in section 4.10. The clipping function is used to avoid outliers, which can be appear by sampling from a normal distribution e.g. a strong negative sample for "sigma_var" (line 21-23) could lead to a negative value as argument for the Gaussian filter (line 29-30). A negative "sigma" for the Gaussian filter should be avoid, because it crashes the creation of the synthetic surface. As initialization a surface in pixel-to-height representation is created with zero height values (line 16). The

randomWalk(...) function (line 28) returns bulky hills with 0 (no hill) and 1 (hill) pixel values, which results in different expansions of hills. A Gaussian filter (line29-30) is able to smooth the discrete hill expansion to a continuous surface based on its argument "sigma" (standard deviation). A python for-loop (line 17-36) over all "sigmas" calculates different simple synthetic surfaces as described in figure 4.17 and aggregate them together. The maximum height of a synthetic surface can be controlled with line 44. After the process of creating a synthetic surface, the function returns the synthetic surface as a pytorch tensor with mean height values equal to zero.

```
1  def createSurface(resolution, sigmas = (10,6,3,1.5,1),
2                    p = 0.008, H = 0.008, I=2, l=100, h=150,
3                    variation = 0.02):
4      '''
5      create synthetic surface with the use of random walk method
6      :param resolution: (tuple) -> (H:int, W:int), size of output matrix
7      :param sigmas: (tuple) -> (float, ..., float), looped standard
       deviations for Gaussian Filter
8      :param p: (float), percentage of starting points based on amount of
       pixels (H*W)
9      :param H: (float), maximum height of synthetic surface
10     :param I: (int), number of loops (iterations)
11     :param l: (int), minimum length of step size S
12     :param h: (int), maximum length of step size S
13     :param variation: (float), variation of standard deviations (percent)
14     :return: (pytorch tensor) -> (H, W), synthetic surface pytorch tensor
15     '''
16     surface = np.zeros(resolution)
17     for sigma in sigmas:
18         # variation of standard deviation sampled from normal
19         # distribution
20         # np.clip is used to control outliers from sampling
21         sigma_var = np.clip(np.random.normal(0, sigma*variation),
22                             a_min=-(sigma*variation*2),
23                             a_max=sigma*variation*2)
24         p_var = np.clip(np.random.normal(0, p * variation),
25                         a_min=-(p*variation*2),
26                         a_max=p*variation*2)
27         # apply random walk method
28         surface1 = randomWalk(resolution, p+p_var, I, l, h)
29         surface1 = gaussian_filter(surface1, sigma=sigma+sigma_var,
30                                    mode='reflect')
```

```
31          # set maximum height to 1
32          surface1 /= surface1.max()
33          # set maximum height equal to sigma from gaussian_filter
34          surface1 *= (sigma+sigma_var)
35          # add surface structure to overall surface
36          surface += surface1
37      # set maximum height to 1
38      surface /= surface.max()
39      # variation of height sampled from normal distribution
40      H_var = np.clip(np.random.normal(0, H*variation),
41                      a_min=-(H*variation*2),
42                      a_max=H*variation*2)
43      # set maximum surface height to H + variation
44      surface *= (H+h_var)
45      # return synthetic surface (mean of synthetic surface = 0)
46      return (torch.from_numpy(surface) - torch.mean(torch.from_numpy(surface
        ))).float()
```

5.9 Implementations of SurfaceNet(...)

SurfaceNet(...) is the implementation of SurfaceNet network as described in section 4.12. The class inherits from BaseNet, which has basically some important plotting functions to get the results in section 7. The idea of this inheritance is to get a modular system for allowing a quick implementation with other network architectures. The class SurfaceNet has four hyperparameters, which have to be initialized. The four hyperparameters are also described in detail in section 4.12.

A more interesting part is the implementation of stacked BlockNet layers as depicted in figure 4.20. The first $L//4$ BlockNet layers without dimensionality reduction are stacked together in line 20-24, after that stacking process the next $L//4$ Blocknet layers with dimensionalty reduction are extended to the module list named self.mid_layers. This extension of self.mid_layers is repeated altogether 3 times, which results in L stacked BlockNet layers. The dimensionality reduction is happening with code line 27, 32 and 37, which identifies if the extended BlockNet layers is the first one or not. If the extended BlockNet Layers is the first one, a dimensionality reduction is happening by setting the boolean value called dim_red to "True" and the BlockNet module is able to perform

a reduction accordingly. An initialization for the first and last layers is done by code block 15-18. The padding for all convolutional operations are chosen such that the size is unchanged. Thus the padding is equal to $k//2$, where k is the kernel size and the operation // is again an integer division.

The forward pass, which is coded via the function forward(...) is relatively simple. Starting with the first convolution (line 50), then all stacked layers are passed through it (line 51-52) and the last layer (line 53) converts the intermediate result to one channel, which should represent the pixel-to-height surface matrix. Squeezing the result before returning (line 54) is necessary for representational purpose of a surface matrix.

```
1  class SurfaceNet(BaseNet):
2      '''
3      SurfaceNet inherits from BaseNet and it consists of the general
       architecture
4      '''
5      def __init__(self, layers=12, mid_channels=32, BlockNet=ResNetBlock,
       cardinality=1):
6          '''
7          :param layers: (int), amount of stacked layers
8          :param mid_channels:  (int), amount of mid channels CM
9          :param BlockNet: (nn.Module), module for general architecture
10         :param cardinality: (int), cardinality or amount of grouped
11         convolutions
12         '''
13         super().__init__(crop=50)
14         # first and last layer
15         self.begin = nn.Conv2d(12, mid_channels, kernel_size=7,
16                                padding= 7 // 2)
17         self.head = nn.Conv2d(mid_channels//2**3, 1, kernel_size=3,
18                                padding=3 // 2)
19         # stacked layers BlockNet layers
20         self.mid_layers = nn.ModuleList(
21                                [BlockNet(mid_channels,
22                                dim_red=False,
23                                cardinality=cardinality)
24                                for i in range(layers // 4)])
25         self.mid_layers.extend(nn.ModuleList(
26                                [BlockNet(mid_channels//2,
27                                dim_red=True if i==0 else False,
28                                cardinality=cardinality)
```

```
29                              for i in range(layers // 4)]))
30          self.mid_layers.extend(nn.ModuleList(
31                              [BlockNet(mid_channels//2**2,
32                              dim_red=True if i==0 else False,
33                              cardinality=cardinality)
34                              for i in range(layers // 4)]))
35          self.mid_layers.extend(nn.ModuleList(
36                              [BlockNet(mid_channels//2**3,
37                              dim_red=True if i==0 else False,
38                              cardinality=cardinality)
39                              for i in range(layers // 4)]))
40
41          self.relu = nn.ReLU()
42
43      def forward(self, x):
44          '''
45          forward pass of SurfaceNet
46          :param x: (B,12,H,W), pytorch tensor with 12 images
47          :return: (B,H,W), surface matrix in pixel to height
48          representation
49          '''
50          x = self.relu(self.begin(x))
51          for block in self.mid_layers:
52              x = block(x)
53          x = self.head(x)
54          return x.squeeze(1)
```

5.10 Implementation of BlockNet modules

This subsection consists of all BlockNet modules, which are implemented and used as experiment. BlockNet is a python class and inherits from pytorch nn.Module class. SurfaceNet from section5.9 takes as input a BlockNet module to specify its architecture in detail. Section 4.12 introduces two different types of BlockNet modules as specification for SurfaceNet.

class ResNextBlock(...)

The class ResNextBlock(...) is copied from [22] and it has two convolutions (line 13-16) and one skipconnection (line 17-18) as initialization parameters. If the boolean variable

called dim_red is True, the input channel is twice times the amount of channel C, else input channel and output channel are equal to argument C.

The forward function is relativly simple and the code snippet should be enough information to understand the idea.

```
class ResNextBlock(nn.Module):
    def __init__(self, C, dim_red=False, cardinality=1):
        '''
        :param C: (int), amount of channels
        :param dim_red: (boolean), dimensionality reduction,
        if True Cin=C*2, Cout=C, else Cin=Cout=C
        (Cin: input channels, Cout: output channels)
        :param cardinality: (int), cardinality to group convolution
        in parts
        '''
        super().__init__()
        C_in = C*2 if dim_red else C
        self.conv1 = nn.Conv2d(C_in, C_in, kernel_size=3,
                               padding=3 // 2, groups=cardinality)
        self.conv2 = nn.Conv2d(C_in, C, kernel_size=3,
                               padding=3 // 2, groups=cardinality)
        self.skipconnections = nn.Conv2d(C_in, C, kernel_size=1,
                               groups=cardinality)

        self.relu = nn.ReLU()
        self.dim_red = dim_red
    def forward(self, x):
        '''
        forward pass of ResNextBlock
        :param x: (B, C or C*2, H, W), input tensor
        :return: (B, C, H, W), output tensor
        '''
        fx = self.relu(self.conv1(x))
        fx = self.conv2(fx)
        skip = self.skipconnections(x) if self.dim_red else x
        return self.relu(fx+skip)
```

class ConvBlock(...)

The ConvBlock(...) class is a copied version of ResBlock(...) with two changes: 1) self.skipconnections is removed from the module and 2) cardinality is always 1. More information about implementation can be found at Github.

5.11 Implementation of DummySet(...)

The dataset module is very easy and simple as the name already indicates. However, the parameter called "amount_data" is responsible to control the length of the data. This number could also be very large, because there exists no limit for the amount of synthetic surfaces. Additional, the default hyperparameters are used for creating a synthetic surface (line 15). A change in hyperaparameter values for synthetic surface creation is only possible, if changing the default value in the createSurface(...) function. This is e.g. the case if other cabin-cap materials are used.

```
1  class DummySet(Dataset):
2      def __init__(self, resolution, amount_data=300):
3          '''
4          intialization of the dataset
5          :param resolution: ( tuple ) -> ( H : int , W : int )size of
6          synthetic samples
7          :param amount_data: length of dataset
8          '''
9          self.len = amount_data
10         self.resolution = resolution
11     def __getitem__(self, index):
12         '''
13         :param index: (int), index of sample
14         :return: synthetic surface sample in pixel-to height
15         representation
16         '''
17         return torch.tensor(createSurface(self.resolution).tolist()),
18                 index
19     def __len__(self):
20         return self.len
```

5.12 Implementation of update(...) and _forward(...)

This subsection shows parts of implementation about the training loops as described in section 4.9. Two important function are located in file training.py, which are important to understand.

The update(...) function set gradients to zero, apply backward pass and perform optimization step (line 13-15). A for-loop over a function called _forward(...) yields the error made by prediction.

```
1  @torch.enable_grad()
2  def update(network: nn.Module, data: DataLoader, loss: nn.Module,
3             opt: torch.optim.Optimizer) -> list:
4      # set network to train mode
5      network.train()
6      # initialize list for storing training error
7      errs = []
8      for iter, err in tqdm(enumerate(_forward(network, data, loss))):
9          # store training error
10         errs.append(err.item())
11         # set gradients to zero, apply backward pass and perform
12         optimization step
13         opt.zero_grad()
14         (err).backward()
15         opt.step()
16
17         (...)
18
19     return errs
```

A function named _forward(...) calculates the error made by the neural network (line 22-23 of line 38-39), which is used in the update(...) function as error signal to perform an optimization step. The dataloader returns in a for loop several synthetic surface samples (line 3), which are used as data points. The synthetic surface samples can be converted via the transformation function (Filament renderer with optimized scene parameters) to synthetic images (line 5-6) and the model uses the synthetic images as input to predict the surface (line 8). Additionally, the surface error (line 11-14) is calculated. However, the surface error is only used for the loss function, if no encoder decoder architecture is used. Nevertheless, the function returns always the surface error no matter, which kind of

training loop is selected. The reason is to record the surface error later as quality measure. A distinction is made with the if/else in line 15 and 24. If the encoder decoder architecture is selected with setting the Boolean variable _encoder_decoder to True, blockcode in line 16-23 will be executed otherwise blockcode 25-39 will be used. As already explained in section 4.9 the difference between this two training loops is the loss function. For encoder decoder architecture the image space is used to compute the loss function (line 22-23) and for the other architecture a weighted sum as total loss is calculated (line 39-39). The weighted sum consist of a gradients loss L^g (line 34-36) and a surface loss L^s (line 11-14). At the end _forward(...) returns a tuple with two elements: 1) error (calculated loss) and 2) MSE surface error between prediction and synthetic surface.

```
1  def _forward(network: nn.Module, data: DataLoader, metric: callable):
2      device = next(network.parameters()).device
3      for synthetic_surface, idx in data:
4          # transform synthetic surface to synthetic images
5          synthetic_images = transformation.forward(
6                                synthetic_surface.to(device)) # (B,L,H,W)
7          # predict surface with synthetic images as input
8          predicted_surface = model(synthetic_images) # (B,H,W)
9          # calculate loss between synthetic surface
10         # and predicted surface
11         surface_err = metric(
12                         predicted_surface[...,crop:-crop,crop:-crop],
13                         (synthetic_surface[...,crop:-crop,crop:-crop])
14                         .to(device))
15         if _encoder_decoder:
16             # apply decoder: transform prediceted surface
17             # to predicted images
18             predicted_images = transformation.forward(
19                         predicted_surface)
20             # calculate loss between synthetic images
21             # and predicted images
22             res = metric(predicted_images[...,crop:-crop,crop:-crop],
23                          synthetic_images[...,crop:-crop,crop:-crop])
24         else:
25             # calculate gradients for predicted surface
26             predicted_gradients = getGradients(
27                         predicted_surface[..., crop:-crop,
28                         crop:-crop])
29             # calculate gradients for synthetic surface
30             synthetic_gradients = getGradients(
```

```
31                          (synthetic_surface[...,crop:-crop,crop:-crop])
32                          .to(device))
33              # calculate gradient loss
34              gradients_err = metric(
35                          predicted_gradients,
36                          synthetic_gradients)
37              # calculate total loss
38              res = (1-_sigma) * gradients_err +
39                          _sigma * surface_err
40          yield res, surface_err.item()
```

5.13 Implementation of getGradients(...)

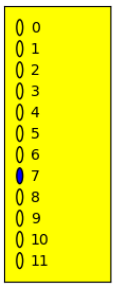
The getGradients(...) function is used to calculate the gradients in x and y direction, which is needed in the _forward(...) function for calculating the total loss in case of no encoder decoder architecture is selected. The implementation very similar to the function called getNormals(...) in section 5.2. More information about the implementaton can be found at Github. The repository is under `https://github.com/Kiesi1991/Surface-Reconstruction.git` available.

6 Usage

6.1 How to start scene parameter optimization?

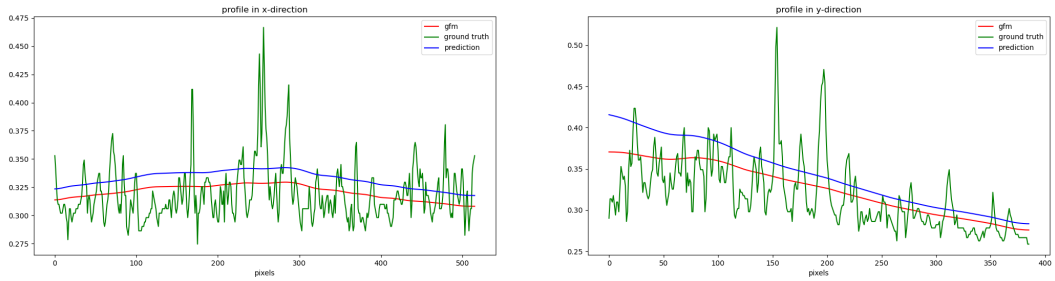
This section explains the usage to start optimization, given real images taken from the Freesurf Sensor or synthetic surfaces. Before starting the python file, a folder with real sample images must be generated. The default folder path is named "realSamples", but it is possible to specify other locations as well. By running the file parameter-manipulation.py the matplotlib figure from 4.10 appears. As mentioned before, this figure is used to adjust the parameters (area D) and to specify several parameters etc.

Area A - select light source



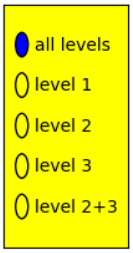
Area A is used to switch between active light sources and change the diagrams in area B to the corresponding active light.

Area B - compare and display results



Three profile lines in x- and y-direction are displayed in area B. A profile line means always the middle row or column from a 2D matrix. The lines are used to get more insights into shadow effects and to adjust some parameters. The line called ground truth shows the gray-scale values from a 2D real cabin-cap sample image and "gfm" stands for the calculated Gaussian filtered median with function "getGfm(...)" over several real sample images. These lines are fix and they cannot be influenced via parameter changes. However, the blue line named prediction illustrates the profile for a shadowed output with applying the Filament renderer. This line is influenced from parameter change and it should be changed to somehow fit to the "gfm" line (in red).

Area C - select light source levels for optimization

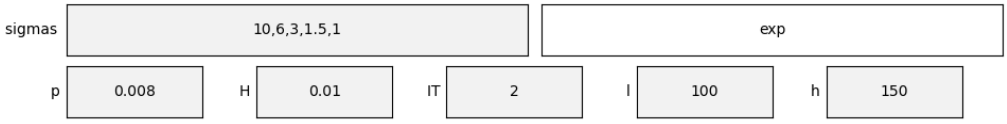


In general the optimization could happen with all light source, but it could also happen with a part of them. With area C it is possible to choose, which light source level should be used for optimization.

Area D - adjust parameters before optimization

Area D has 3 sliders to adjust the material parameters, a change in a parameter leads to a new prediction line in area B. The default values are good initial guesses for the Freesurf sensor. Nevertheless, every Freesurf sensor has its own shadow effects and therefore a different Freesurf sensor has other parameters for optimization. This is the reason for using the matplotlib figure with adjustments in parameters.

Area E - hyperparameters for synthetic surface creation and regularization function



The creation of synthetic surfaces depends on several hyperparameters (hyperameter listing 4.10). Input field "sigmas" corresponds to the input arguments as seen in equation 4.8. The other important hyperparameters are

p: probability p (hyperameter listing 4.10),

H: maximum height value (equation 4.8),

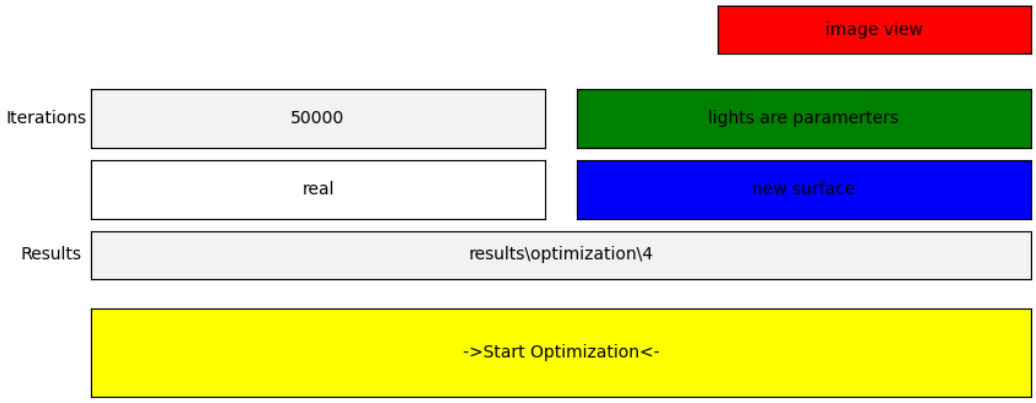
IT: iterations I (hyperameter listing 4.10),

l: minimum step size S (hyperameter listing 4.10) and

h: maximum step size S (hyperameter listing 4.10).

Additional, a button located on the top right position is used to select the regularization function for the light positions (equation 4.7).

Area F - boolean buttons



The boolean buttons in area F are red, if they are False and green for True. Area F consists of 2 boolean buttons. The first button called "lights are parameters" is used for setting the gradients on and off. If the button is green (on), light positions are also optimized during optimization and if the button is red (off), light positions are constant. The second boolean button called "image view" changes the profile view in area B to the image view. The image view shows either a real image cabin-cap sample or a synthetic image (shadowed output from Filament renderer).
The field named Iterations in area E shows how many loops of optimizations should be performed and "Folder results" stands for the path, where intermediate/final results during optimization are stored. As default value 50.000 iterations are set. On click of button "real" change the real sample to a synthetic sample created with hyperparameters from area E and the profile line "ground truth" in area B illustrates the synthetic profile sample. Additional, a blue button called "new surface" creates a new synthetic surface with the hyperparameters entered in area E. This button can be used to inspect different synthetic samples. For comparing synthetic surfaces, the image view is a better choice.
After adjusting the parameters and selecting everything mentioned before, the "Start Optimization" button can be clicked to start the optimization. Please wait until the optimization is finished, it is possible to take a look at the directory path of "Folder results" to inspect intermediate results during optimization.

6.2 How to start training SurfaceNet

For training a SurfaceNet start the python file called "training.py" with a python interpreter. Before starting the process some important variables can be modified. At the beginning of the file several variables are located, which are necessary to define. Open "training.py" will show the following code lines:

```
# resolution of images
resolution = (386, 516)
# path, where results are stored
path_results = os.path.join('results', 'trainNN20')
# path, where optimized parameters are stored
path_optimized_parameters = os.path.join(
                    'results', 'optimization', '1', '0')
# path, where real images samples are stored
path_real_samples = 'realSamples1'

# please enter parameters for SurfaceNet
# if more than 1 element in list, the training is looped accordingly
mid_channels=[64] # mid_channels C
layers=[8] # amount of BlockNet layers
blocks = [ConvBlock] # BlockNet: [ResNextBlock, ConvBlock]
cardinality = [1] # cardinality for ResNextBlock

# training parameters
num_iter = 2001 # amount of synthetic surface samples during training
batch_size = 2 # batchsize
lr = 1e-4 # learining rate
crop = 50 # all images will be cropped accordingly
encoder_decoder = [False] # use encoder decoder training loop
sigma = [0.98] # if sigma is 1 full surface loss
```

Every variable has its importance during training and therefor they can be specified separate. Here is a detailed description about them:

resolution: resolution of all images. This value must be the same as the resolution of the real image samples. (dtype = tuple)

path_results: defines the location, where intermediate/final results are stored. If the path do not exist, the folders will be created automatically. (dtype = str)

path_optimized_parameters: location of all optimized scene parameters from step 1. If no folder is available, please start "paramerter-manipulation.py" before "training.py". (dtype = str)

path_real_samples: location path, where real cabin-cap images are inside. The images should be named "L.png" and L stands for the number of active light seen in the figure. (dtype = str)

mid_channels: defines the amount of channels in SurfaceNet corresponding to variable C_M as depicted in figure 4.20. (dtype = list with integers, iterable)

layers: defines the amount of BlockNet layers similar as variable L in figure 4.20 (dtype = list of integers, iterable)

blocks: defines the BlockNet module used for training. (dtype = list of nn.Module, iterable)

cardinality: cardinality (groups of convolution), which is used by BlockNet is equal to ResNextBlock. (dtype = list of integers, iterable)

num_iter: number of samples used for training until loop ends. (dtype = int)

batch_size: batch size number (dtype = list of integers, iterable)

encoder_decoder: if boolean is True encoder-decoder training loop is used (figure 4.12), else normal training loop is used (figure 4.11). (dtype = list of booleans, iterable)

sigma: hyperparameter for weighted sum in case of using no encoder decoder architecture (dtype = list of float numbers, iterable

All lists (with dtype = iterable) are looped and the network is trained as many times as possible combinations. After initialization of the variables, the file "training.py" can be started and the training begins. Intermediate and final results can be obtained in the result folder, in addition the model is saved also in the result folder.

7 Experimental Results

7.1 Evaluation of optimized parameters

This section shows results about the optimization method, which was described before. The main goal of optimizing a real cabin-cap sample is to get relevant parameters for training a neural network, which is able to estimate normal vectors. Different evaluation methods can be used as quality measurement.

Compare real images with rendered images

The first intuitive idea to evaluate the optimization method is to compare the real cabin-cap images with the rendered images. This evaluation is straight forward and it shows a first outcome. Figures 7.1, 7.2 and 7.3 illustrate two images, where it is possible to compare them. Nevertheless it is hard to see any significant differences between corresponding images. This evaluation cannot be used to get a sufficient information about the quality of the optimization method. However, comparing the images is an evidence about the overall functionality and it is even possible to get more insights with some understanding about computer vision.

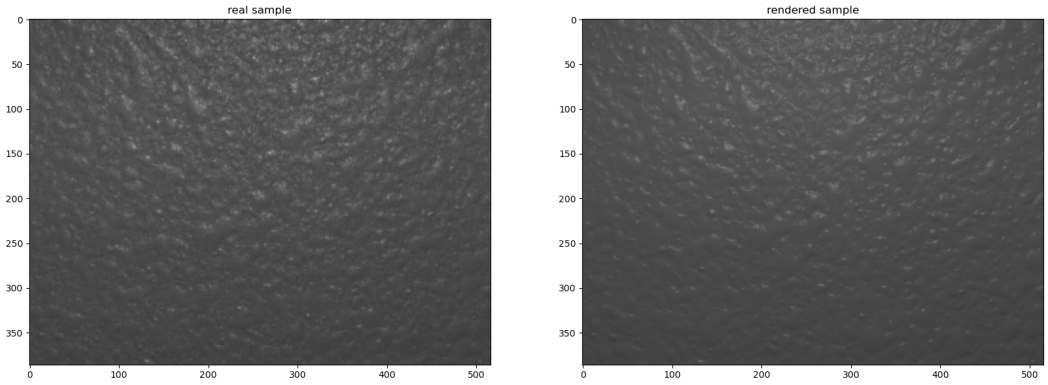


Figure 7.1: real vs. rendered image comparison, light source 0 - level 3, left subimage = real cabin-cap sample, right subimage = rendered image (output after 50.000 iterations)

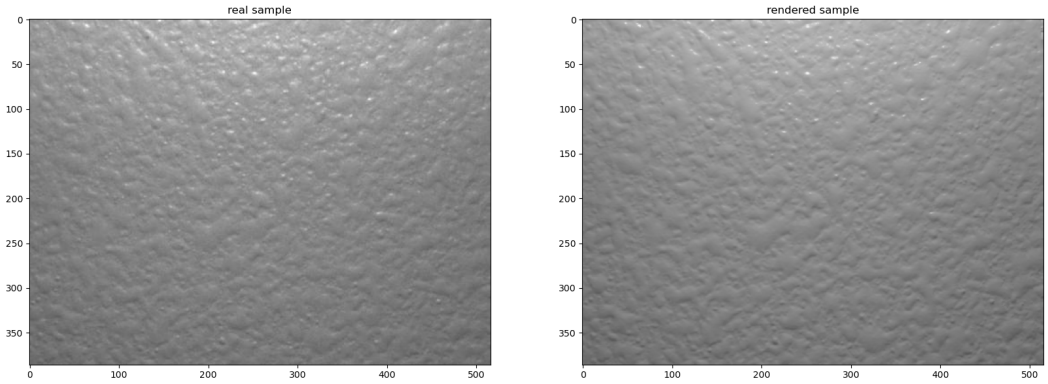


Figure 7.2: real vs. rendered image comparison, light source 4 - level 2, left subimage = real cabin-cap sample, right subimage = rendered image (output after 50.000 iterations)

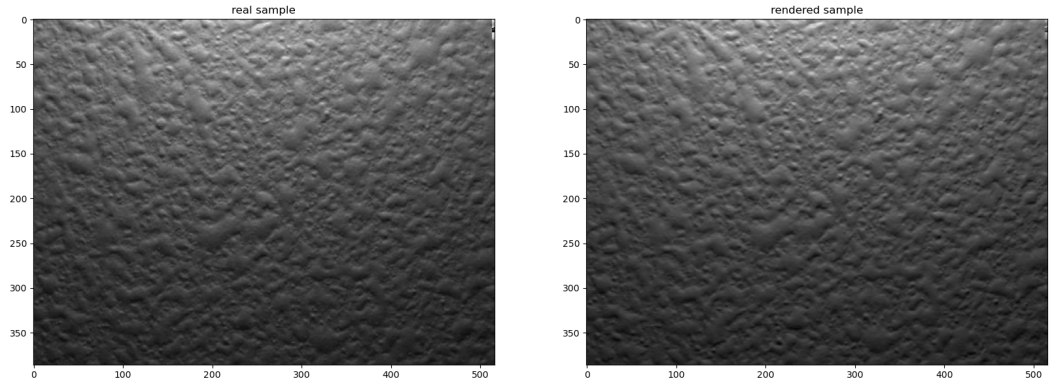


Figure 7.3: real vs. rendered image comparison, light source 8 - level 1, left subimage = real cabin-cap sample, right subimage = rendered image (output after 50.000 iterations)

If the bright areas of both images are very similar, it is more likely to have a good optimized scene. Bright areas can only occur, if the angle between light vector **L** and view vector **V** are very similar to the real scene and the slope of the surface correlates to the real surface slope. Therefor the comparison between real images and rendered images should concentrate on the bright areas. Figure 7.4 shows a comparison with no bright areas. This leads to the assumption that some parameters are not well optimized or in general something went wrong. In the case of figure 7.4 the optimization with only 6.250 iterations is the cause for no bright areas, because the optimization is not finished. This example should illustrate the effect of bright areas by comparison between real and rendered images.

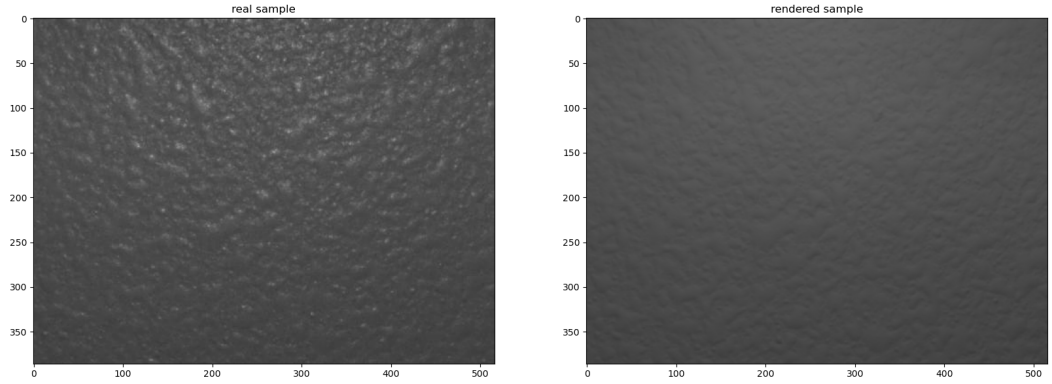


Figure 7.4: real vs. rendered image comparison, light source 0 - level 3, left subimage = real cabin-cap sample, right subimage = rendered image (output after 6.250 iterations)

Predicted surface angles

Figure 7.5 shows a heatmap, where every pixel stands for a angle value. The angle values are calculated with the surface normal vectors and a vector in z-direction: $(x, y, z)^T = (0, 0, 1)^T$. The used surface for calculation of the angles is an optimized surface after 50.000 iterations.

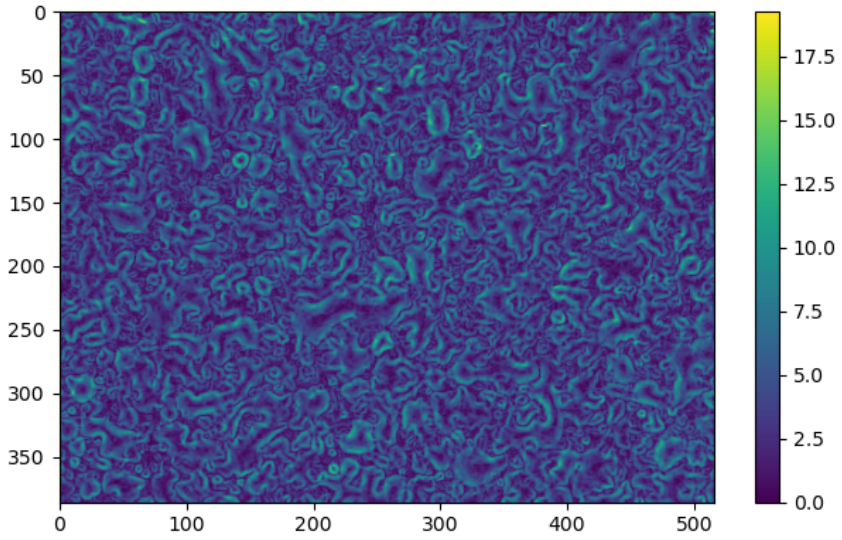


Figure 7.5: Heatmap, where every pixel stands for a angle value.

By inspecting the cabin-cap material, it is possible to roughly estimate the maximum angles of the normal vectors. It is obvious to see, the maximum is lower than 45 degrees and the heatmap shows angles between 0 degree and 18 degrees. The spread in angles for an optimized surface verifies the assumption. However, the maximum angle value is also important for other reasons. If the angles are to high, it could be the case that hard shadow effects occur. A hard shadow effect is not considered with the optimization method. Figure 7.6 demonstrates a hard shadow effect for an optimized surface. On the left side is a hill, which has a high slope. It is not possible for the light rays to reach a certain area. Therefor every pixel has a maximum angle, which is allowed to get no hard shadows. The allowed angles for light source level 1 are between 11,79 and 20,01 degrees depending on which pixel is considered. Therefor an optimization with level 1 light source could lead to some issues during optimization. However, the practice in optimization shows

not an significant influence about this effect but it is possible to optimize only with level 2 and 3. The section "usage" describes the parameter manipulation mask and with area C it is possible to select parts of light source levels for optimization. The Filament renderer has a material parameter called "diffuse" this parameter regulates also the illumination effect in areas, where no light rays reach the surface. The influence of material parameter diffuse and very small areas of hard shadows has little impact. As a hint for other surfaces, it is necessary to check the effect of hard shadows. The easiest way is to optimize the parameters with all light levels and also without level 1. If there is a significant discrepancy in results, hard shadows could be the reason for that.

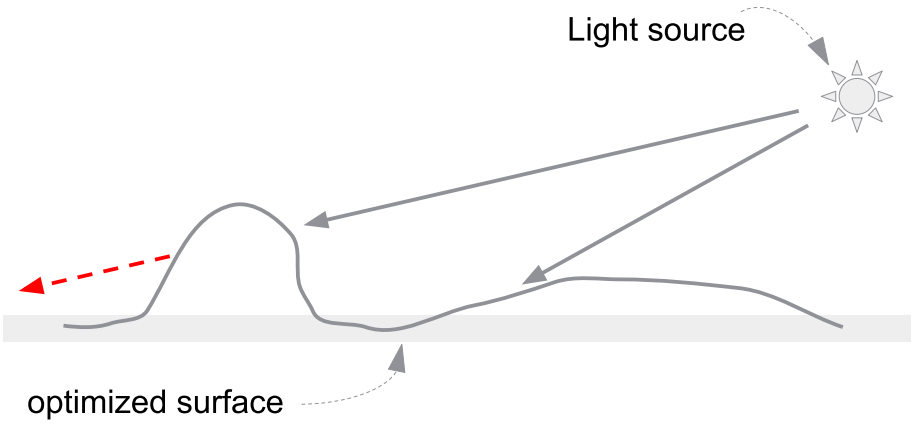


Figure 7.6: Hard shadow effects, which occur on a optimized surface.

Evolution of material parameters during optimization

Material parameters are hyperparameters before starting the optimization process. The material parameters influence the training time. If they are chosen well, the optimization could happen with less iterations. However the material parameters are also influenced from the multiplied shadow effects. The shadow effects are calculated with the output prediction at zero height values $pred_0$. This output prediction is also influenced by some additional chosen material parameters. The chosen material parameters for $pred_0$ could be randomly initialized, but they influence the predicted material parameters during optimization. The predicted material parameters differ, depending on the initialized material parameters for $pred_0$. For clarification, there are two different material parameters

by multiplying the shadow effects: 1) material parameters for $pred_0$ (constant by applying shadow effects) and 2) material parameters for the prediction *pred* during optimization (trainable). More information about the method can be found at section 4.5.

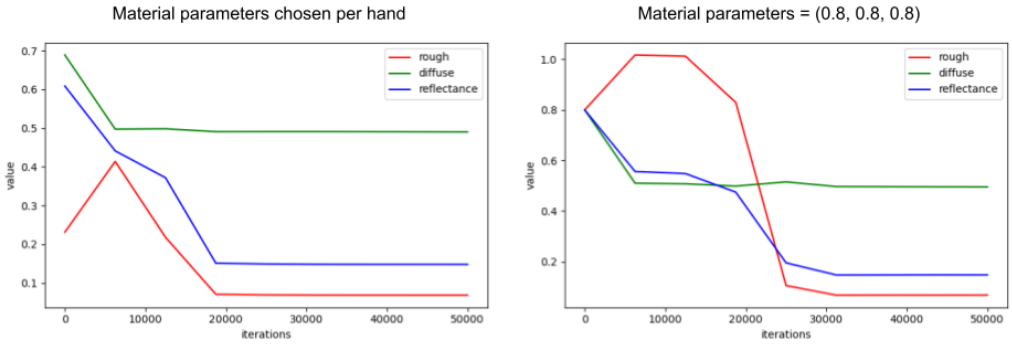


Figure 7.7: Evolution of predicted material parameters during optimization for a real sample - constant material parameters for $pred_0$: $(0.5, 0.5, 0.5)^T$

As seen in figure 7.7 the material parameters reach always the same values after optimization, but the initial guesses influence the duration of finding the optimal values. The left side shows an initialization per hand, which was made with the parameter manipulation mask. The values converge faster to the optimum compared to the right initialization. In worst case of initialization, it is even possible to fail and find no optimum during training. As mentioned above the optimal values at the end of optimization can differ, if the material parameters for $pred_0$ are different.

For a real cabin-cap sample exists no ground truth in material parameters, thus the results in figure 7.7 could only be an indication for a correct approach. However, as seen in the experimental result both situations (left and right diagram) converges to the same values. This could be considered as an evidence for a correct method. An even stronger evidence would be to know the ground truth for material parameters, which is true for synthetic samples as described in section 4.10. Figure 7.8 shows the evolution of material parameters for a synthetic sample and the dashed lines are the ground truth values. The optimized values converges nearly perfect to the real values. Therefore, this result is a strong evidence for the correctness of the optimized values.

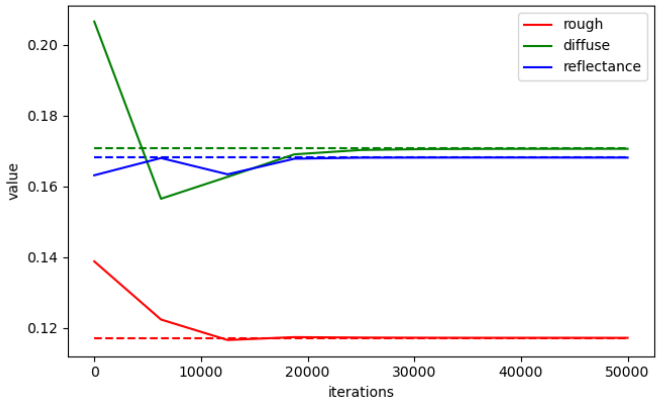


Figure 7.8: Evolution of material parameters during optimization for a synthetic sample - constant material parameters for $pred_0$: $(0.2, 0.2, 0.2)^T$, dashed line: ground truth value.

Optimized height profiles

Unfortunately for the real cabin-cap images there exists no ground truth for the optimized surface. Therefor it is hard to evaluate the quality of the optimized surface. Profiles in x- and y-directions can be used as information gain for the optimized surface. Figure 7.9 and 7.10 illustrate several profiles. Every profile is the result of different initial material parameters for ($pred$ and $pred_0$) after 50.000 optimization loops. The variation of initial values is given in table 7.1.

Table 7.1: initial values used for height profiles in figure 7.9 and 7.10. Second column indicates the initial predicted material parameters and third column the constant material parameters for $pred_0$

profile name	$(rough, diffuse, reflectance)$	$(rough_0, diffuse_0, reflectance_0)$
prediction 0	$(0.2, 0.2, 0.2)$	$(0.5, 0.5, 0.5)$
prediction 1	$(0.5, 0.5, 0.5)$	$(0.5, 0.5, 0.5)$
prediction 2	$(0.8, 0.8, 0.8)$	$(0.5, 0.5, 0.5)$
prediction 3	$(0.23, 0.68, 0.60)$	$(0.5, 0.5, 0.5)$
prediction 4	$(0.2, 0.2, 0.2)$	$(0.2, 0.2, 0.2)$
prediction 5	$(0.5, 0.5, 0.5)$	$(0.2, 0.2, 0.2)$
prediction 6	$(0.8, 0.8, 0.8)$	$(0.2, 0.2, 0.2)$

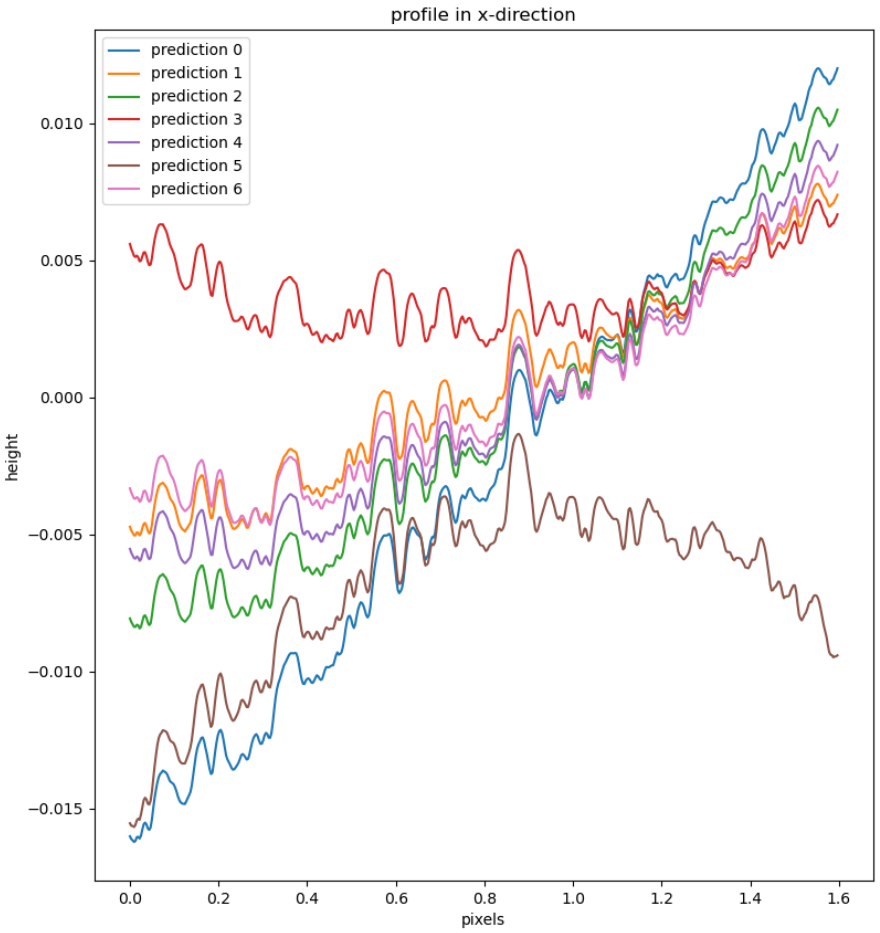


Figure 7.9: Different height profiles in x-direction based on initialization and material parameters for $pred_0$. The prediction was made with an exponential regularization function.

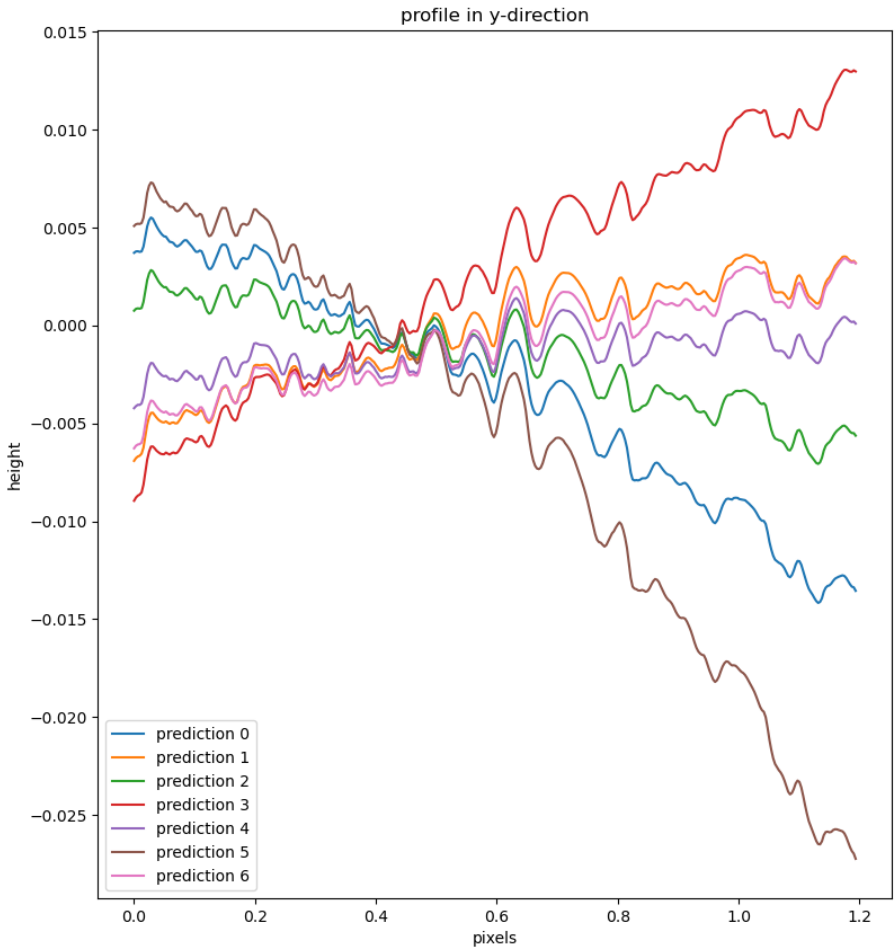


Figure 7.10: Different height profiles in y-direction based on initialization and material parameters for $pred_0$. The prediction was made with an exponential regularization function.

The profiles have more less the same surface structure. Hills and valleys occur on the same place. Nevertheless, a discrepancy can be observed between the predictions, which could be problematic. As already mentioned in section 1 (Introduction) the overall goal is to determine the normal vectors of the cabin-cap surface. Normal vectors can be computed by using equation 4.3, which is basically a vector with gradients in x and y directions. Thus, if the gradients from the profiles (figure 7.9 and 7.10) are similar, the normal vectors should be also the similar. Figure 7.11 shows the gradients from profile prediction 0 and 6. The similarity of gradients is an indication for a well optimized surface and the predicted surfaces can be used to get the normal vectors. The y-axis for the height profiles

just contains small numbers. Normal vectors are not really influenced and therefor the optimized surface structures with different initial values lead to the same structure in general. This fact is an evidence for the stability of the optimized surface structure.

Figure 7.11 contains only two predictions (prediction 0 and 6), because of visualization purpose. However, the other profile lines create same results.

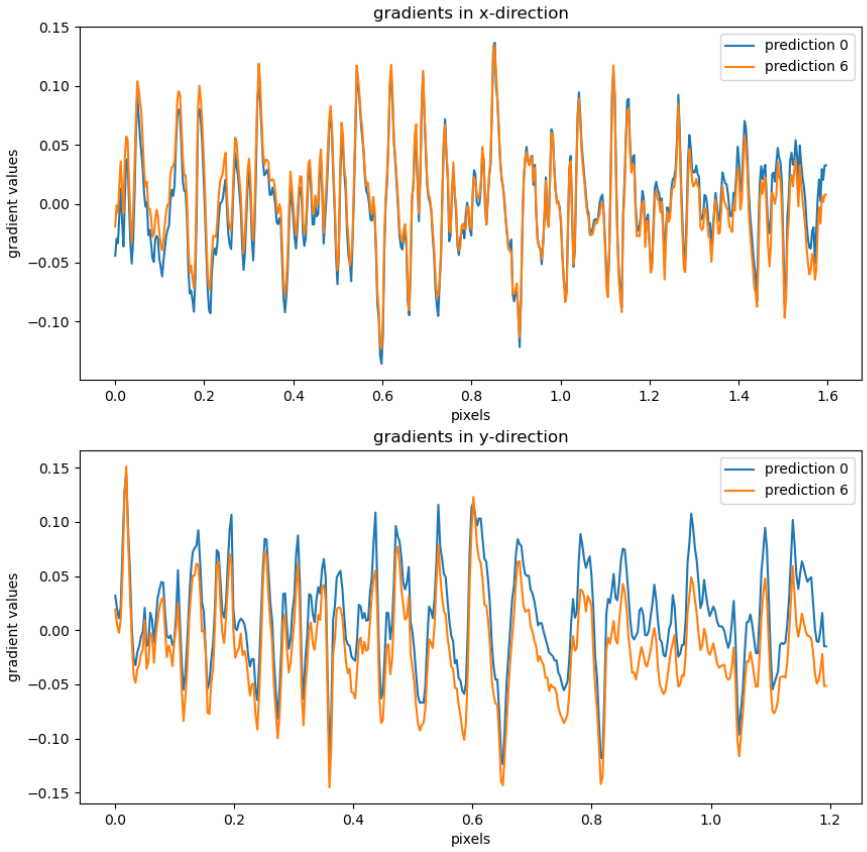


Figure 7.11: Gradients in x and y direction for the height profiles (prediction 0 and 6) seen in figure 7.9 and 7.10.

Synthetic surfaces as introduced in section 4.10 can also be used to verify the optimization method in general, because the ground truth surface is known for synthetic samples. Figure 7.12 and 7.13 shows a comparison between surface prediction and ground truth surface.

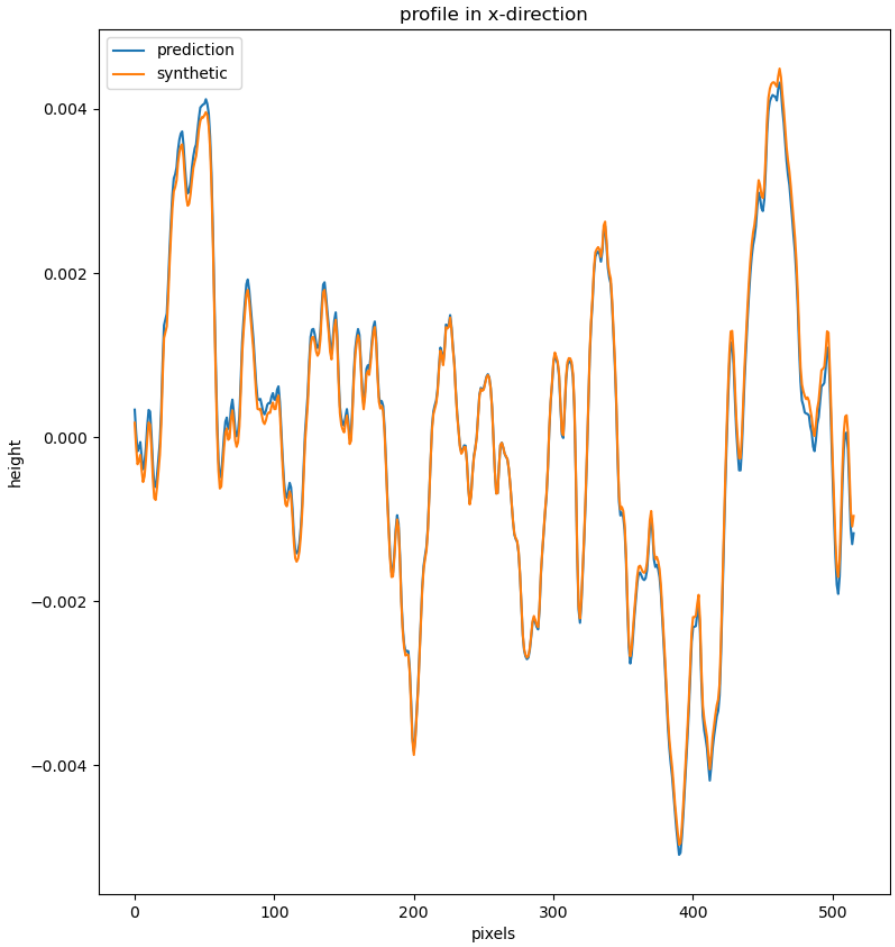


Figure 7.12: prediction profile (blue) vs. synthetic profile (orange) in x direction - the surface is created with function 5.8, Hyperparameters for creation: $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5) = (10, 6, 3, 1.5, 1)$, $h_{\max} = 0,01$, $p = 0.008$, $I = 2$, $l = 100$ and $h = 150$. The prediction was made with an exponential regularization function.

The profile lines in both directions are nearly perfect, which is a strong evidence for the optimization method.

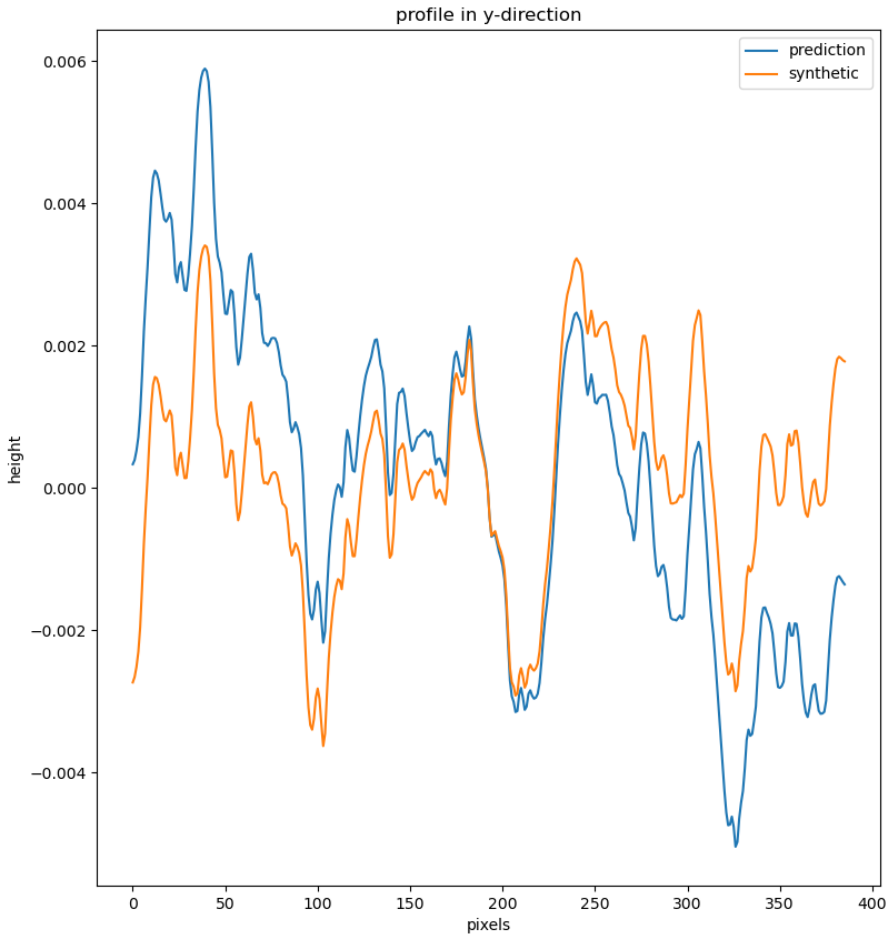


Figure 7.13: prediction profile (blue) vs. synthetic profile (orange) in y direction - the surface is created with function 5.8, Hyperparameters for creation: $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5) = (10, 6, 3, 1.5, 1)$, $h_{\max} = 0,01$, $p = 0.008$, $I = 2$, $l = 100$ and $h = 150$. The prediction was made with an exponential regularization function.

Summary: results of optimization method

The scene parameter optimization method was evaluated with different experimental results: 1) Image comparison, 2) Surface angles, 3) Evaluation of material parameters, 4) Predicted height profiles with real cabin-cap samples and 5) Predicted height profiles with synthetic surfaces. Altogether, results in 4) and 5) show the evidence of the scene parameter optimization. Especially, figure 7.11 is a proof for a correct optimization of

a real cabin-cap surface and results in figure 7.12 and 7.13 confirm the approach with synthetic surfaces, where the ground truth is very similar to the prediction.

7.2 Evaluation of surface prediction

Predicting a surface with the input of 12 cabin-cap image samples is the overall goal of this work. However, scene parameters are necessary for surface prediction, thus the results from section 7.1 are necessary for the training and evaluation loop. This section shows the experimental results by training different neural network architectures with the use of optimized scene parameters and also different training methods (loops) as described in section 4.9.

Notification: All results, which are shown in this section, uses the same parameters (except others mentioned) for synthetic surface creation: $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5) = (10, 6, 3, 1.5, 1)$, $p = 0.008$, $IT = 2$, $l = 100$, $h = 150$ and $v = 0.02$ (a description can be found at section 4.10, 4.10 and 4.10).

Validation error and surface error for model selection

The DummySet(...) introduced in section 5.11 returns always new synthetic surfaces by calling the function self.__getitem__(...), because with synthetic surface creation an infinite amount of data exist. Therefor, every synthetic surface sample is only used one time during training. This is the reason, why no epochs exist for the whole training procedure and instead of speaking about epochs, iterations are used. An iteration is one step in training with a defined batch size B. The distinction between training error and validation error as used for real world data with finite data points can be neglected, because every sample is only shown once the network during training. The overfitting problem as described in [31] is crucial for training Neural Networks especially, if less amount of data points are available. In the cabin-cap problem with the use of synthetic surface creation this effect is therefore not relevant and the training error (loss) is equal to the validation error.

The SurfaceNet with its BlockNet in it allows a variation of different architectures depending on the chosen hyperparameters.Thus selecting a model needs a performance measure to compare them. The validation error, which was mentioned before (equal to training

error) is an opportunity to compare the different network architectures. In addition, the surface error between synthetic surface and predicted surface can be used too. Table 7.2 lists 10 different architectures based on several parameters for model selection. The 10 architectures with its parameters are chosen to see a clear impact of the parameter variation.

Table 7.2: Different model architectures for model selecting. L: amount of layers, C_M: amount of mid channels, c: cardinality (grouped convolution) and BlockNet: ResNextBlock or ConvBlock. More infos about model architecture can be found in section 4.12

model	L	C_M	c	**BlockNet**
0	12	64	1	ResNextBlock
1	12	32	1	ResNextBlock
2	16	64	1	ResNextBlock
3	16	32	1	ResNextBlock
4	16	64	2	ResNextBlock
5	16	64	4	ResNextBlock
6	16	64	1	ConvBlock
7	12	64	1	ConvBlock
8	8	64	1	ConvBlock
9	8	32	1	ConvBlock

The model selection is made with comparison of the validation error during the training process. Altogether the training was made 1.000 iterations with a batch size of 2. Figure 7.14 shows the validation error for the ten architectures from table 7.2. Model 0-5 uses as BlockNet the ResNextBlock and model 6-9 are trained with the simple ConvBlock as BlockNet, which has no skipconnections. The results show, that ResNextBlock as BlockNet is more robust and the networks using ResNextBlock as BlockNet are able to learn easier with different combinations of parameters. Especially, if the network is deeper, a network with ConvBlock is not able to learn the task. The deepness of the SurfaceNet-network can be controlled via the parameter L, which stands for the amount of stacked layers (more details at section 4.12). Model 6 and 7 are even unable to learn anything as seen in figure 7.14, because the network is too deep and the gradients are vanished. The literature describes this phenomena in detail and that is one reason for using skipconnections, which were introduced in the ResNet paper (section 3.1 and [18]). However, if the parameter L is lower or equal to 8, a SurfaceNet with Convblock as BlockNet (model 8 and 9) is able to learn and it looks like it could get better results than a SurfaceNet with ResNextBlock (model 8). Interesting is also model 9, which needs more than 200 iterations to get out of its intermediate state and than continue learning. This is an indication of being at the maximum network deepness by using ConvBlock as BlockNet.

ResNextBlock as BlockNet (model 0-5) can handle deeper network structures as seen in figure 7.14 every network variation is able to learn, also a SurfaceNet with $L = 16$ (model 2-5). This is an important property, especially if the task is complex. As a side notification: "The complexity of SurfaceNet can be controlled via the parameter L (deepness) and C_M (width)." Figure 7.14 shows an improvement for the validation error, if L and C_M is high. However, the used hardware (1.Processor: AMD Ryzen 5 5600H with Radeon Graphics, 3302 Mhz, 6 Core(s), 12 Logical Processor(s), 2.GPU: NVIDIA GeForce RTX 3060 Laptop GPU) allows only a maximum of $L = 16$ and $C_M = 64$, which is in the cabin-cap task sufficient enough. Further improvements can be observed by a variation of parameter cardinality. The cardinality c is the main difference between ResNet and ResNext (section 3.1 and 3.2). Better results can be observed in the literature [22] and also in Figure 7.14. If the cardinality is 1 the ResNextBlock as BlockNet behaves like a Resnet module and if the cardinality is getting higher it acts like a ResNext module (grouped convolutions are applied). Model 4 shows an improvement by using a cardinality of 2 (model 4) instead of 1 (model 2). However, a cardinality of 4 (model 5) is even worse compared to model 2. Therefor, using cardinality could increase the performance, but it could also decrease the performance. The quality increase of using a cardinality equal to 2 (model 4) is not significant compared to a cardinaltiy of 1 (model 2). In contrast, using pytorch as library for implementation has one big disadvantage with grouped convolutions (means cardinalty is greater as 1). A bug in implementation (pytorch library) can lead to a high execution time during training (`https://discuss.pytorch.org/t/group-convolution-takes-much-longer-than-normal-convolution/92214`), which was also the case by training model 4. This is the reason to consider further only models with cardinality equal to 1.

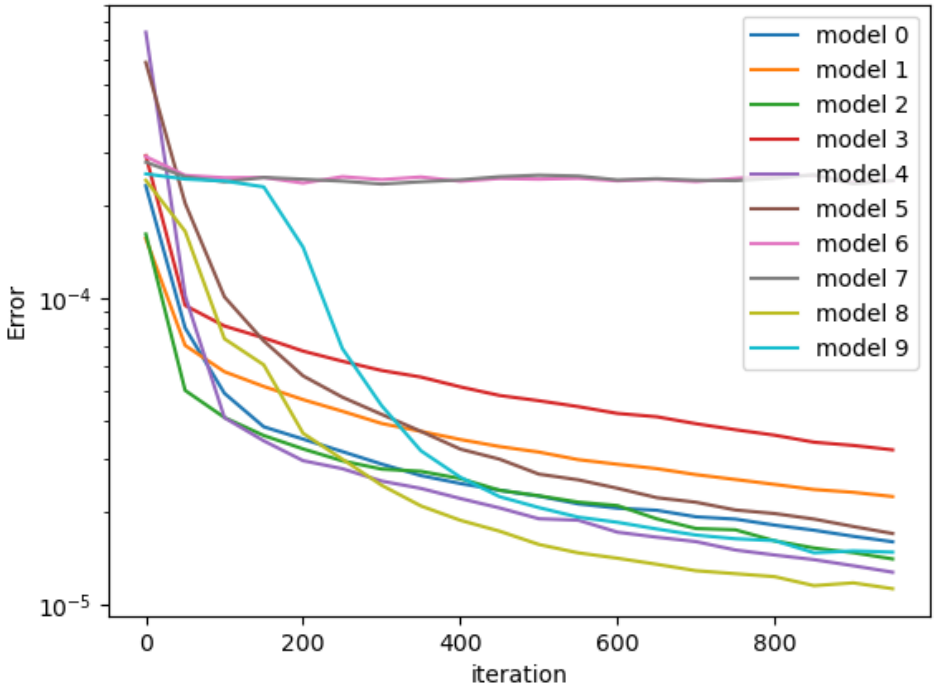


Figure 7.14: Validation error for different model architecture. A detailed hyperparameter for model selection is listed in table 7.2. Every model is trained with the encoder decoder training loop as explained in section 4.9. This diagram is made with the use of "compare-training.py" file.

A comparison between different validation errors (figure 7.14) during training is used for model selection. Concrete, two models are selected and trained for a longer period of time to choose the final architecture (figure 7.15). Model 8 (BlockNet = ConvBlock) and model 2 (BlockNet = ResNextBlock) are selected for that reason, because other models has a higher validation error or a cardinality greater than 1. As a side note, an interesting fact is also to see the lowest validation error for model 8 (figure 7.14), which is a very simple network without having any skipconnections. However, if the two network are trained for more iterations, model 2 outperforms model 8. The literature for ResNet [18] and for ResNext [22] is confirmed. Figure 7.15 shows the results for training model 2 and 8 for more iterations.

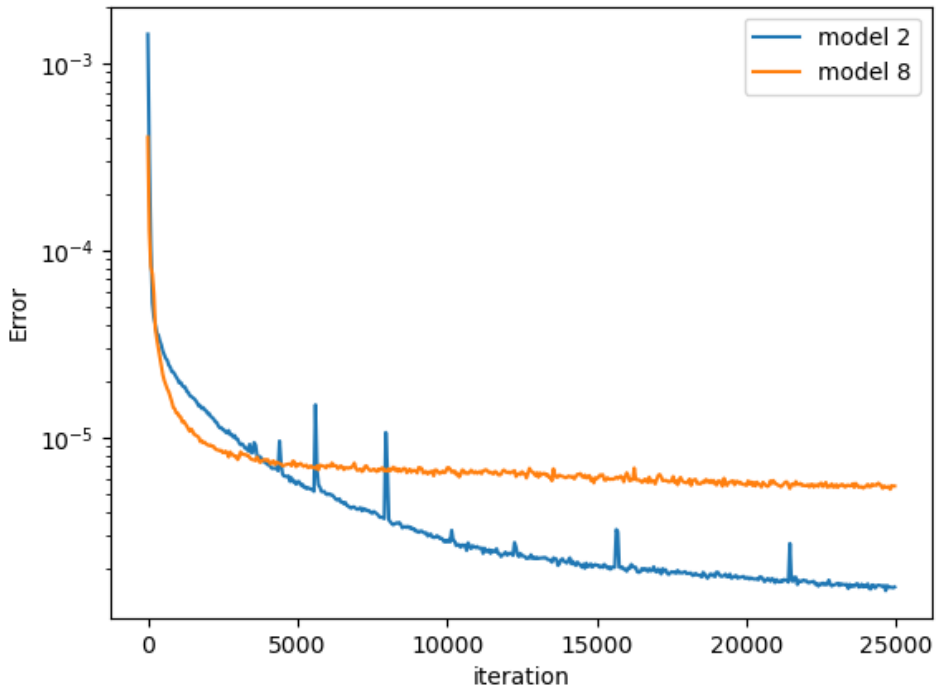


Figure 7.15: Validation error model 2 and 8 for more iterations. A detailed hyperparameter for model selection is listed in table 7.2. Every model is trained with the encoder decoder training loop as explained in section 4.9. This diagram is made with the use of "compare-training.py" file.

Comparing the different validation errors to get the most suitable network architecture is a valid possibility. But, the validation error in the encoder decoder training loop (explained in section 4.9) stands for the mean squared error (result of loss function) between synthetic images and predicted images (image space). The overall goal is to predict the surface (surface space) not the images, thus the validation error is a weak confirmation. Figure 7.16 displays the surface error during training, which is the mean squared error between synthetic surface and predicted surface. The surface error in combination with the validation error is a stronger evidence for the trained network. The surface error results are shown for model 2 and 8, which were selected before. In addition two models were trained without encoder decoder training loop (figure 4.11), which are named model 2.1 and 2.2. All four trained models has more less the same

surface error, except model 2.2 and therefor a distinction between 2, 2.1 and 8 cannot be made by comparison of surface errors. Nevertheless, a distinction can be made for model 2.2. The validation error is lower compared to the others, which could be an indication for a better surface prediction. Section 7.2 disproves the assumption for a better surface prediction with model 2.2. In addition, the method for model 2.1 and 2.2 introduces an extra hyperparameter σ, which can be avoided by using the encoder decoder structure. Extra hyperparamters should be avoided if possible. Therefore, models trained with encoder decoder training loop are more relevant. A comparison in validation error between model 2 and 8, which were trained with encoder decoder training loop, shows a lower validation error for model 2, therefore model 2 is the most relevant model in sections below. As a side notification: Model 2.1 and 2.2 cannot be compared via the validation error, because the loss functions differs from each other.

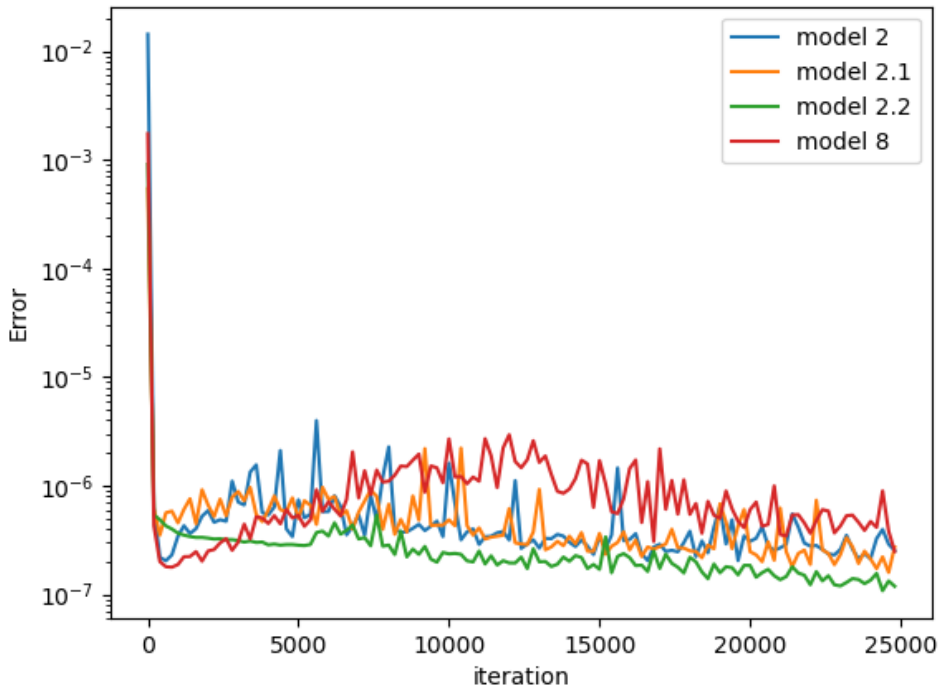


Figure 7.16: Surface error model 2, 2.1, 2.2 and 8 for more iterations. Model 2.1 and 2.2 has the same architecture as model 2, but it was trained without encoder decoder method. The exact training loop is depicted in figure 4.11. Model 2.1 was trained with $\sigma = 0.98$ and model 2.2 with $\sigma = 1.00$. A detailed hyperparameter for model selection is listed in table 7.2. Every model except model 2.1 and 2.2 is trained with the encoder decoder training loop as explained in section 4.9. This diagram is made with the use of "compare-training.py" file.

Compare real images with predicted images

Similar to subsection 7.1 this section shows the comparison between real cabin-cap images and the prediction made by a trained neural network architecture e.g. model 2. The trained neural network predicts usually a pixel-to-height surface matrix and therefor a transformation is made from surface to image. The transformation is done similar as described in figure 4.9 to get predicted images for comparison.

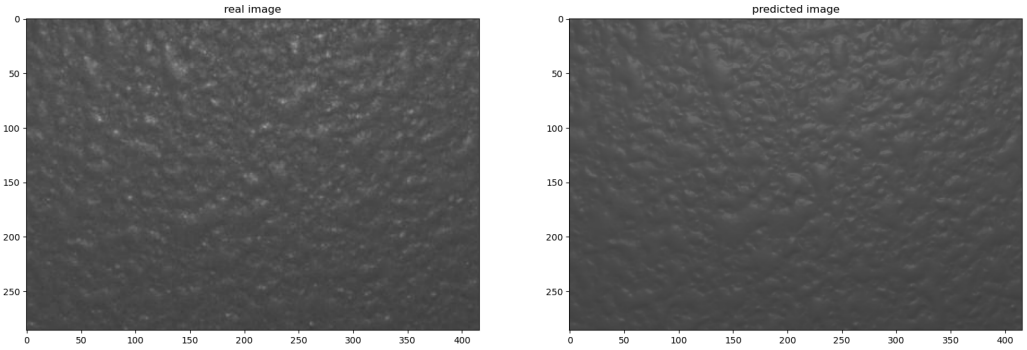


Figure 7.17: real vs. predicted image comparison, light source 0 - level 3, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 25.000 iterations with a batch size of 2), model architecture = model 2 (see section 7.2)

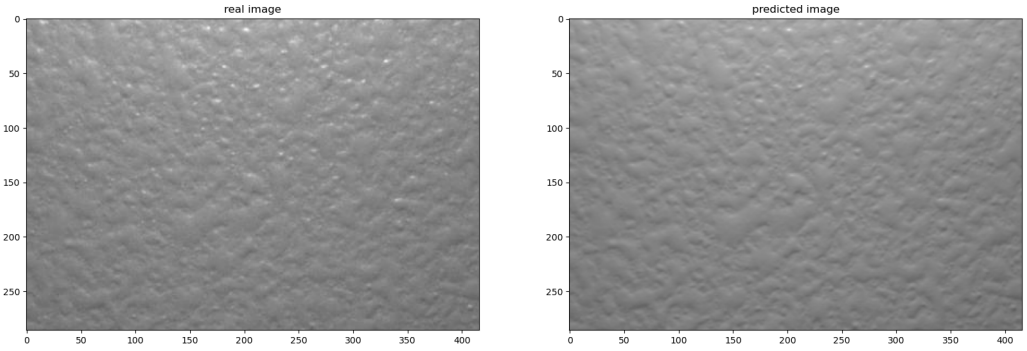


Figure 7.18: real vs. predicted image comparison, light source 4 - level 2, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 25.000 iterations with a batch size of 2), model architecture = model 2 (see section 7.2)

Figure 7.17, 7.18 and 7.19 shows the real image (left subimage) and the predicted image (right subimage). The predicted images are very similar to the real images and it is hardly possible to distinguish between reality and prediction. Nevertheless, by focusing on the bright pixels some difference can be observed. As already mentioned in section 7.1 comparing images could only verify, if the method is working in general and it is not a good quality measurement.

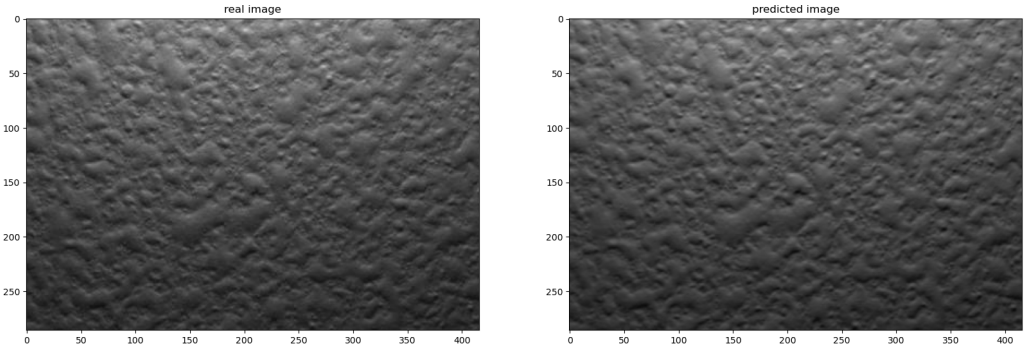


Figure 7.19: real vs. predicted image comparison, light source 8 - level 1, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 25.000 iterations with a batch size of 2), model architecture = model 2 (see section 7.2)

As already mentioned in section 7.2 model 2.2 has the lowest surface error. However, training the network only with the surface loss ($\sigma = 1$) lead to unstable surface regions as indicated with white circles in figure 7.20. In this regions are the height values extremely high or low, which is definitive a failure in prediction.

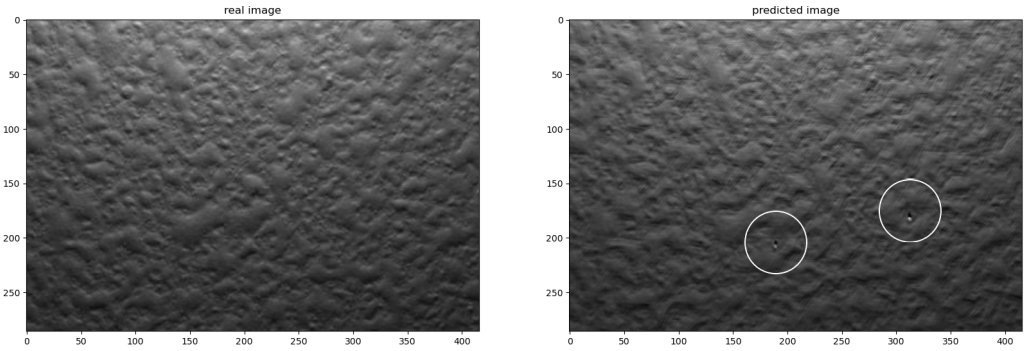


Figure 7.20: real vs. predicted image comparison, light source 8 - level 1, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 25.000 iterations with a batch size of 2), model architecture = model 2.2 (see section 7.2)

In addition, training a SurfaceNet(...) with encoder decoder training loop needs a low amount of iterations for predicting well surface structures compared to a SurfaceNet(...) trained without encoder decoder method. Networks, which are trained with encoder decoder training loop converges with significantly less iterations e.g. figure 7.21 and

7.22 compares the image predictions after 10.000 iterations. A clear discrepancy between predicted and real image can be observed for model 2.2.

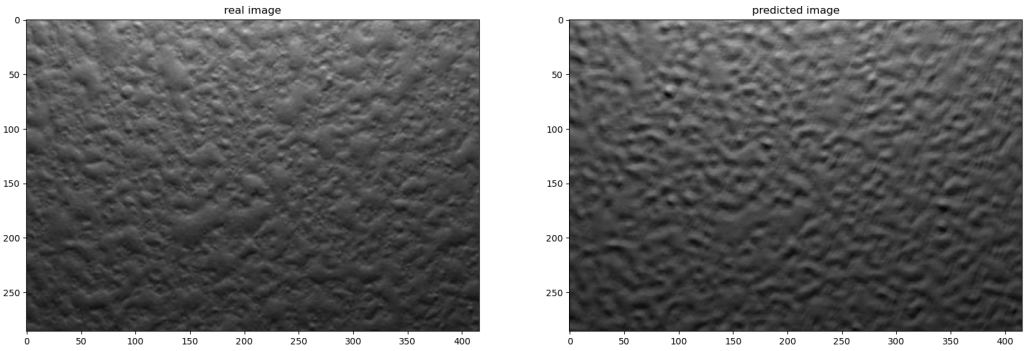


Figure 7.21: real vs. predicted image comparison, light source 8 - level 1, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 10.000 iterations with a batch size of 2), model architecture = model 2.2 (see section 7.2)

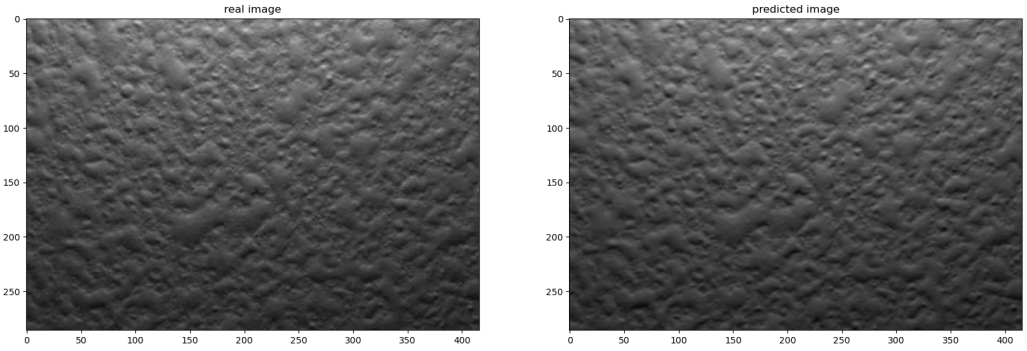


Figure 7.22: real vs. predicted image comparison, light source 8 - level 1, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 10.000 iterations with a batch size of 2), model architecture = model 2 (see section 7.2)

Further inspections shows, that even less than 500 iterations are enough for model 2 to predict a fair enough surface structure (figure 7.23).

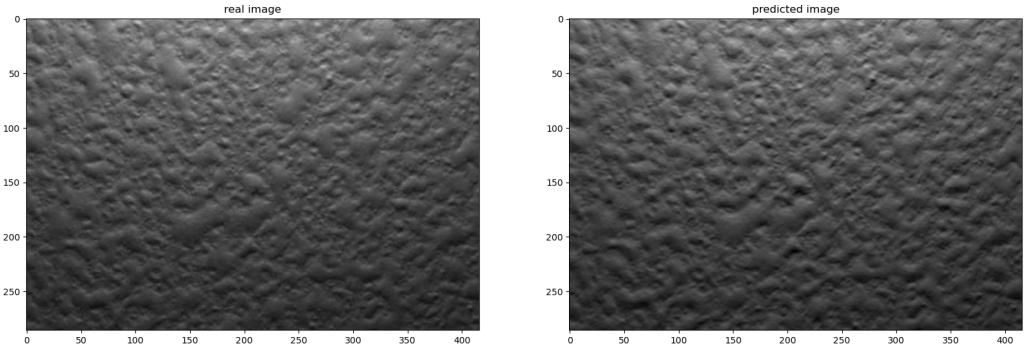


Figure 7.23: real vs. predicted image comparison, light source 8 - level 1, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 500 iterations with a batch size of 2), model architecture = model 2 (see section 7.2)

Predicted height profiles

For the cabin-cap surface task, height profiles are a good choice to understand the surface structure and compare different approaches. Similar as already introduced in section 7.1 a comparison of height profiles in x and y direction can be used to verify results made by prediction. Section 7.1 compares the optimized surface with the synthetic surface. In this section the comparison is made between the optimized surface (surface result from step 1 - optimization) and the predicted surface (prediction made with neural network). The optimized surface is one result, which is also shown in figure 7.9 and 7.10 and it is not the real ground truth surface. Nevertheless, an optimized surface can be used as comparison, because the optimization is sufficient enough.

Figure 7.24, 7.25, 7.26 and 7.27 show surface prediction profiles in x and y direction with the use of different model architecture as described in section 7.2. By comparison between the results, every prediction is a good choice. However, predicted profiles from model 2.1 (figure 7.25) has higher hills compared to the others.

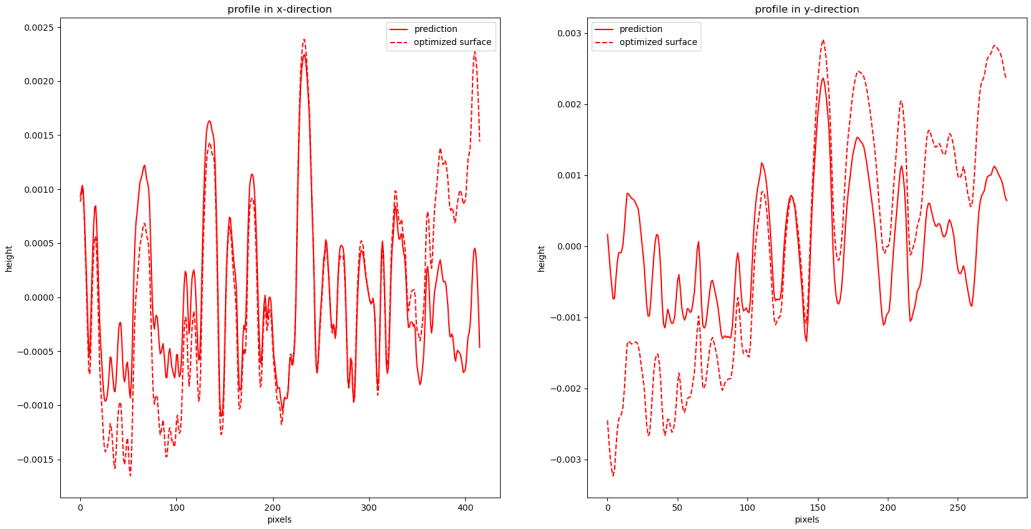


Figure 7.24: Height profiles with model 2 after 25.000 iterations of training and a batch size of 2.

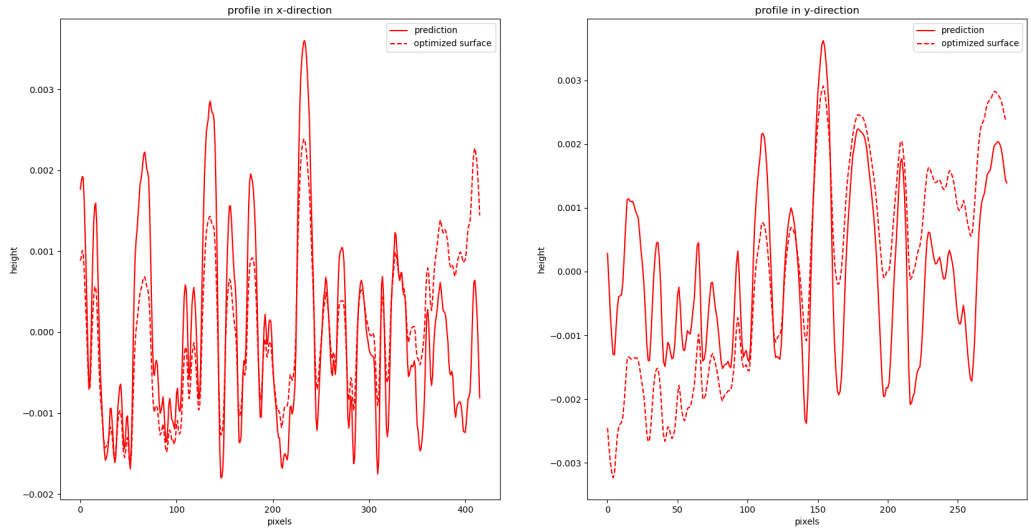


Figure 7.25: Height profiles with model 2.1 after 25.000 iterations of training and a batch size of 2.

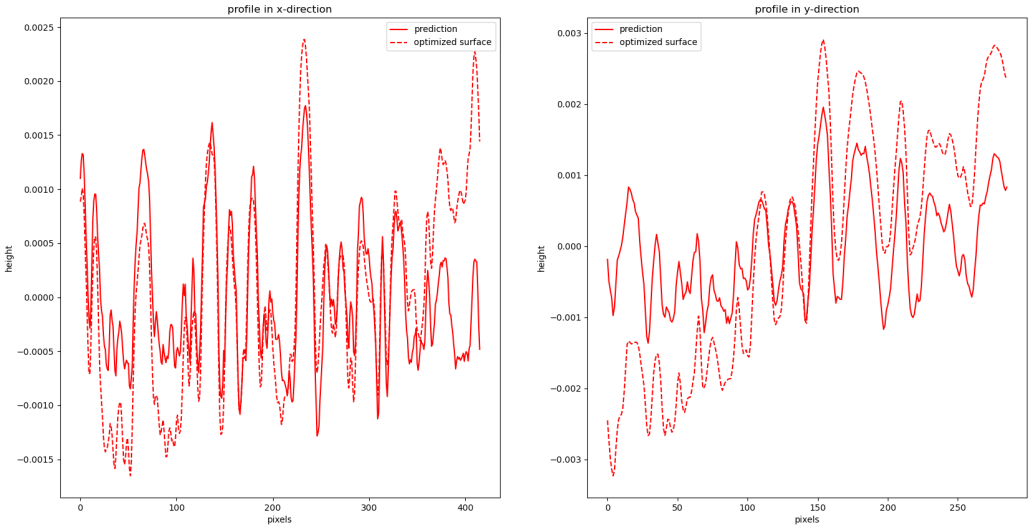


Figure 7.26: Height profiles with model 2.2 after 25.000 iterations of training and a batch size of 2.

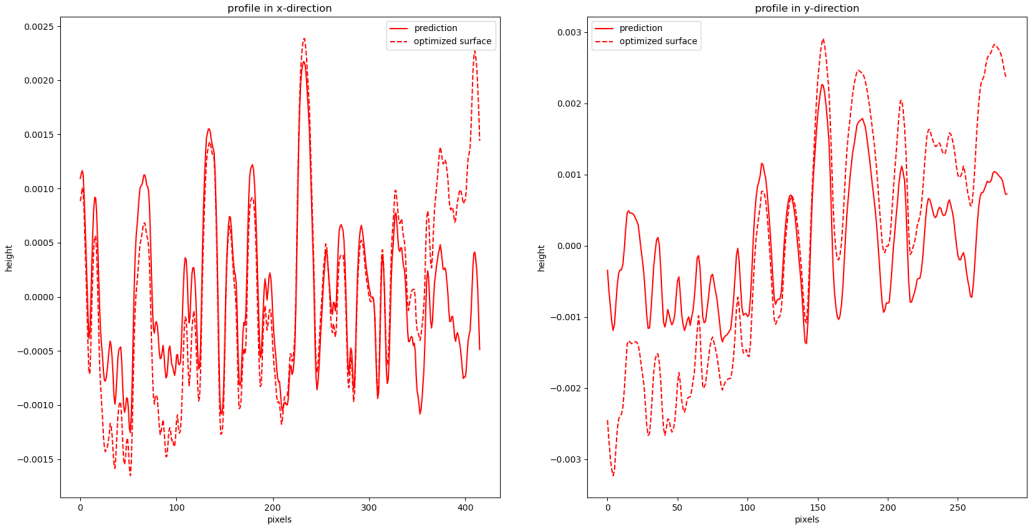


Figure 7.27: Height profiles with model 2 after 25.000 iterations of training and a batch size of 8.

Variation of synthetic surface creation

The experimental results before were all created with the following parameters in synthetic surface creation: $(\sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5) = (10, 6, 3, 1.5, 1)$, $p = 0.008$, $IT = 2$, $l = 100$, $h = 150$ and $v = 0.02$ (a description can be found at section 4.10, 4.10 and 4.10). This section covers the influence of parameter v, which is responsible for expansion of the synthetic surface spectrum.

Figure 7.28 shows the influence in validation error by a change in parameter v. If v decreases, less variation in synthetic surface creation happens and the model can learn faster the task. The validation error from model 2.0 decreases faster compared to the error from model 2, which is an indication for an easier task. If less different synthetic surfaces exist during training, the expectation is also to learn faster. However, less variation in synthetic surfaces means, that the network generalizes fewer. A higher generalization is important to use the trained network in similar tasks, but not the same. This is also the case with synthetic surfaces (during training) and real cabin-cap surfaces (during evaluation). Therefor, a generalization have to be high enough to transfer the synthetic task into the real task. Section 4.10 explains this on the basis of introducing spectrums.

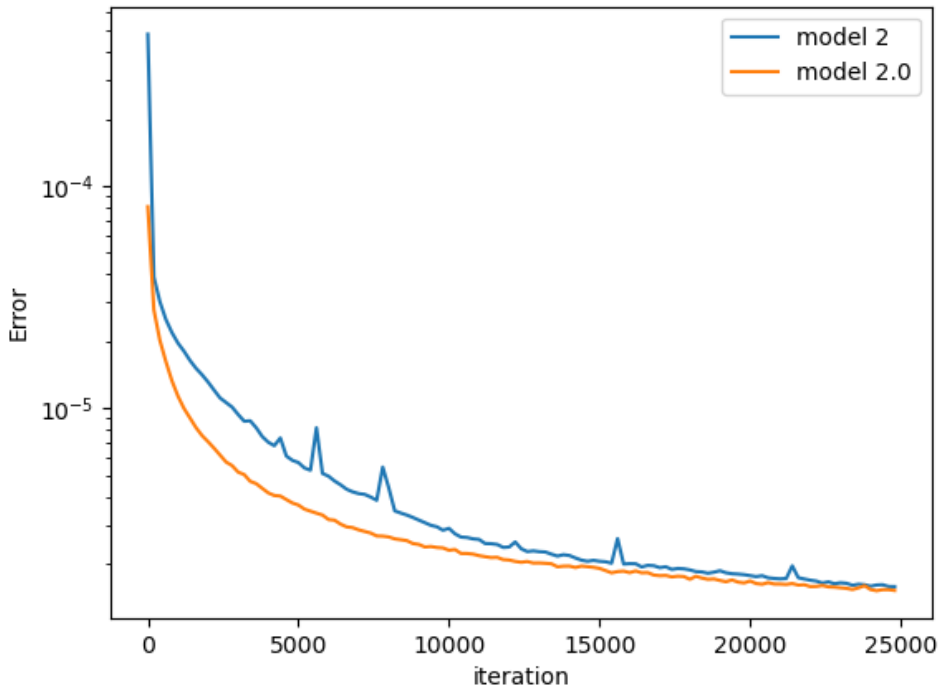


Figure 7.28: Validation error for model 2 ($v = 0.02$) and model 2.0 ($v = 0$). Model architecture: model 2 = model 2.0. A detailed hyperparameter for model 2 is listed in table 7.2. Both models are trained with the encoder decoder training loop as explained in section 4.9. The only difference is the hyperparameter v for synthetic surface creation. This diagram is made with the use of "compare-training.py" file.

By the inspection of the validation error, it is not possible to measure the quality of results for model 2.0, but it shows the difficulty to learn the task. Now, it is interesting, if the variation is still high enough for model 2.0 to get good results on the real cabin-cap samples. Thus, an image comparison between real and prediction is made similar to section 7.2. Figure 7.29 shows the results and the variation is still high enough as seen.

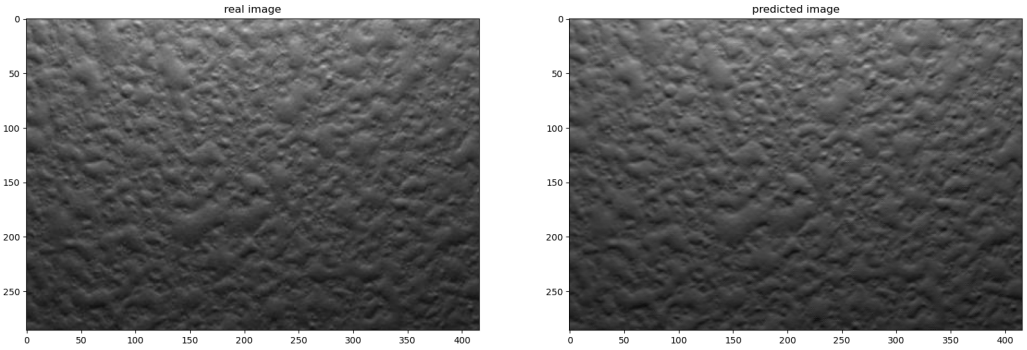


Figure 7.29: real vs. predicted image comparison, light source 8 - level 1, left subimage = real cabin-cap sample, right subimage = predicted image (transformed output after 25.000 iterations with a batch size of 2), model architecture = model 2.0 (see section 7.2)

To confirm the results made by model 2.0, the height profile is shown in figure 7.30, which are produced similar as in section 7.2.

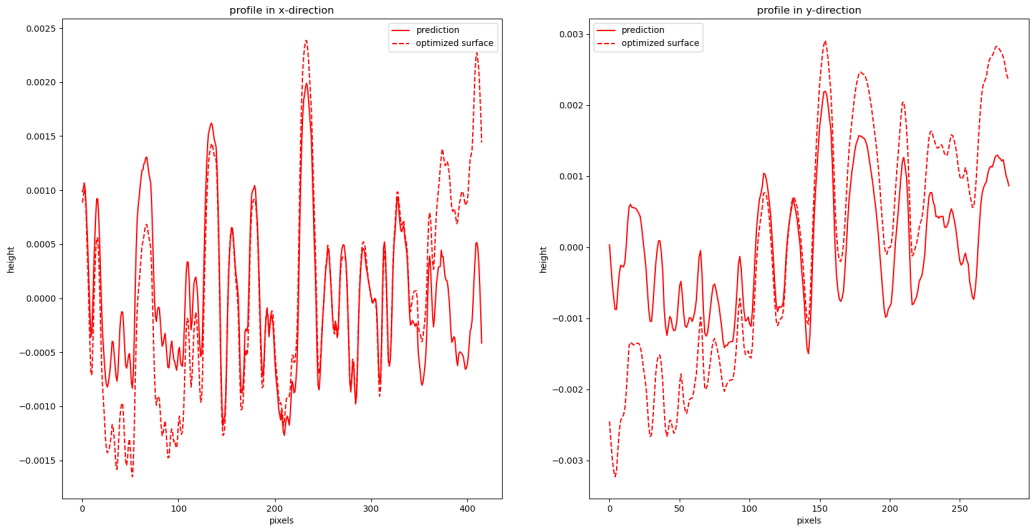


Figure 7.30: optimized profile (dashed red line) vs. predicted profile (red line) in x (left subimage) and y direction (right subimage). The surface prediction was made with model 2.0 ($v = 0$) after 25.000 iterations of training and a batch size of 2.

For model 2.0 is the variation of hyperparameters v set to zero and the task between synthetic and real surfaces can still be transferred as seen, because other variation exist for the method of synthetic surface creation. Some variation is also added by increasing the amount of different standard deviations σ. Model 2 and 2.0 has five different σ values. This lead to a higher variation, because the random walk method is used five times for surface creation. The parameter p controls the probability for a pixel to be a starting point, this process introduces also some variation during the random walk. Parameter l and h are used in a uniform distribution to sample the stepsize S, which means also some variation. And at the end, the random walk method samples between 4 actions (Up, Down, Right and Left) to perform one step. This means an extra variation.

This excursion about variation shows the influence of parameter v and explains the importance of variation to transform a network to a similar task. However, the synthetic surface creation has a lot of variation by design, which would be enough for the cabin-cap problem. Nevertheless, parameter v can be added for surface creation, if the prediction of more complex surfaces would be relevant.

Summary: results of surface prediction with a customized neural network

At the beginning it was shown, that some architectures are not able to learn the task due to vanishing gradient problem [19] e.g. model 6 and 7. However, if the models are able to learn, different kind of architectures are possible to learn the surface prediction task. Architectures, which perform better in the ImageNet project [15], produces also slightly better results in the cabin-cap task as seen e.g. by comparison between different BlockNets used for SurfaceNet(...). ResNextBlock creates a better prediction compared to ConvBlock in general. The comparison method between optimized height profile and predicted height profile can be used as verification about the functionality as seen in figure 7.24, 7.25, 7.26, 7.27 and 7.30. In addition, the comparison between real and predicted images shows the quality of the surface prediction.

Model 2 and 8 were trained with more iterations (25.000), because they were chosen as potential candidates for the final architecture. As seen in figure 7.15, the validation error for model 2 is getting lower than the error of model 8. Therefor, model 2 was also trained with the second training loop. As a reminder, two different trainings loops were introduced: 1) encoder decoder training loop as displayed in figure 4.12 and 2) a training loop with a combined loss function as seen in figure 4.11).

model 2: training loop 1

model 2.1: training loop 2 with $\sigma = 0.98$

model 2.1: training loop 2 with $\sigma = 1.00$

model 8: training loop 1

Altogether, 4 different final models exist and the methods before were not able to distinguish between them. Every experimental result looks very promising, but which one is really the best? For that question, the overall goal must be considered to get a distinction between the four final models. The goal of this work was to predict the normal vectors of a cabin-cap surface. As mentioned several times, the normal vectors can be calculated with the gradients in x and y direction (equation 4.3). Thus, the gradients of the predicted surface must be as accurate as possible to the real surface. Unfortunately, no ground truth exist for the real cabin-cap surface. However, a comparison can be calculated by using synthetic surfaces again and applying the mean squared error (MSE) between the predicted gradients and the synthetic gradients. Figure 7.31 shows the MSE between predicted and synthetic calculated gradients. Model 2.1 is the best model by comparing the gradient-MSE value. The reason could be to have a loss function with a weighted sum, which uses also the gradient loss during training. A drawback of using training loop 2 is the additional hyperparameter σ, which have to be chosen by hand. In contrast, the encoder decoder training loop 1 is more robust and produces similar results with a slightly higher gradient-MSE.

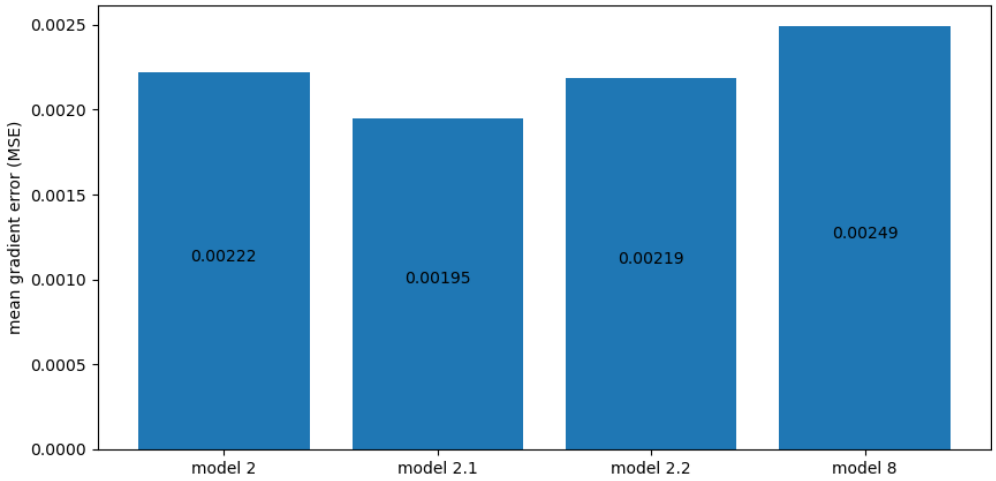


Figure 7.31: Mean squared error between predicted and synthetic gradients over 50 randomly generated synthetic surfaces.

8 Alternative approaches

This work introduces a workaround with synthetic surface creation to avoid using real data of cabin-cap samples for training a neural network (step 2). Additionally, an optimization method (step 1) is used to find important scene parameters (light position, material parameters, etc.). However, the described methods are not the only solution for the task, which was described in section 1. In this section some alternative approaches will be presented and also the reason for designing a complex workaround (synthetic surface creation) for the cabin-cap problem.

8.1 Methods to scan a surface structure

Optimize material parameters and predict a detailed surface structure about the cabin-cap material is not the only way for surface reconstruction. A lot of different approaches exist to scan objects and materials. The following methods can be used to measure the roughness, texture and surface structure:

Optical microscopy [32] uses a microscope to view a surface structure at high magnification by using a combination of lenses.

Atomic force microscopy [33] uses a cantilever with a sharp tip to scan a surface and measure the forces between the tip and the surface at a very high resolution (in nanometer space).

Scanning electron microscopy (SEM) [34] is a type of microscope that uses a focused beam of electrons to produce images of a sample. The electrons interact with atoms in the sample to generate signals that contain information about the sample's surface topography and composition. The position of the electron beam is combined with the detected signal intensity to produce an image. The resolution of SEMs can be

as high as 1 nanometer. SEMs can be used in high vacuum, low vacuum, or wet conditions, and at a range of temperatures.

X-ray reflectometry [35] is a technique used in chemistry, physics, and materials science to study surfaces, thin films, and multilayers. It is a surface-sensitive analytical technique that characterizes surfaces and thin films. The idea is to measure the reflected intensity of X-rays in specular direction and afterwards Fresnel reflectivity law [36] is used to analyze the surface profile.

Lidar (sometimes called 3D laser scanning) [37] is a method for measuring distances using lasers. It involves targeting an object or surface with a laser and measuring the time it takes for the reflected light to return to the receiver. Lidar can be used to create digital 3D representations of areas on the Earth's surface, ocean floor and also to scan other surfaces.

Image analysis [38] involves using software to process images of a surface to get various characteristics. Applications in image analysis are e.g. object recognition, image segmentation, motion detection, analysis to scan material, pose estimation, etc. The cabin-cap method falls under the category of image analysis too.

The surface reconstruction can be done via different methods as described here. The most suitable method depends on requirements of the task, such as accuracy, working environment, speed, costs, etc. However, the use of images from the Freesurf sensor restrict the possible methods to image analysis.

8.2 Alternative approaches with the use of a Freesurf sensor

This section give alternative approaches by using the Freesurf sensor. However, the methods are only ideas and some of them are hard to implement.

Determination of material parameters

Before explaining an alternative to determine the material parameters, it is important to give a short excursus about the material parameters fed into the Filament renderer. The material parameters are not physically correct parameters, they simulate the light rays reflected by an object and they are used as approximation. More details about the

impact of material parameters used by Filament renderer can be found at `https://google.github.io/filament/Filament.html#materialsystem`. However, with that information it is obvious to see the material parameters are virtual parameters for approximation and they can hardly determined by knowing the material in general of an object. By creating a "safe environment" it is possible to find the material parameters very accurate. A safe environment means to generate a scene, where everything is known except the material parameters. For this case a measurement system is needed with accurate position of light (important to have a pointlight sources) and camera plus an object of the same material with a well known surface structure. If such a scene is built, the material parameters can be determined and the uncertainty is close to zero. Compared to the optimization method (figure 4.1) the determination with a safe environment is more trustable.

Determination of light positions

The virtual light position can also be determined with measuring the real location and the angle of the real mirrors. As a reminder some mirrors reflect the light rays produced by the light source, which results in virtual light positions (based on the specific design of a Freesurf sensor). For the Filament renderer is the virtual light position from importance and it can be calculated with having the actual position of light source and the arrangement of all twelve mirrors. Measuring this per hand could lead also to a satisfying result for virtual positions instead of optimizing the light positions during the optimization process.

Determination of camera position

The camera position is well known and in this work the position is set to be equal to the position from the construction plan (CAD plan). Nevertheless, the position could have a discrepancy to the real world. An alternative method is to optimize the camera position as well as light positions or to measure the position per hand.

Determination of shadow effects

The shadow effects have to be calculated similar to the method described in section 4.5, because I have not found a different approach in literature, which shall not mean there is maybe a method related to the shadow problem statement. However, for calculating the shadow effects several real cabin-cap image samples are used to determine the Gaussian filtered median (Gfm). It is also possible to use a flat surface (e.g. a white paper) for the

determination of shadow effects, which could result in a better estimations. In general, the shadow effects can be determined more accurate by having more real samples, therefore the approach with using a flat surface will have not an dramatically effect by my guess.

Alternative approach for synthetic surface creation

Usually real world data is used to train a neural network instead of synthetic generated data, nevertheless labeling data means mostly a lot of work before having enough data to train a neural network. Especially in case of the cabin-cap samples a labeled data means to know the the ground truth surface structure. This information could only be determined with using a precise measurement system e.g. as described in subsection 8.1, which is able to scan the surface and return its surface structure.

A second approach could also be to take a high amount of images with the Freesurf sensor as unlabeled sample and optimize each surface structure given the unlabeled samples with a optimization loop similar to figure 4.1. This would result in labeled data, which can be used in the next step as training data. However the time to create such labeled data is very high.

9 Conclusion and Outlook

9.1 Optimization of scene parameters and cabin-cap surface

Optimizing scene parameters (material, positions, surface structure) with the Freesurf sensor leads to several tasks, which can be solved as mentioned in this work. The IV (interdependent variable) problem and the occurrence of shadow effects needs some special customized methods as solution. At the beginning of implementation it was challenging to find the reason, why optimization was not working in a correct way. However, the approaches for solving the IV problem and simulate shadow effects produces very well results as seen in section 7. The approach for shadow effects and to avoid IVs could also be useful for other tasks, beside the cabin-cap problem. It is very likely to have no perfect point light in reality and it is also likely to have some interdependency between parameters. In addition, the pixel-to-height representation is a modified version of bumpmapping [5], which could be a good alternative to traditional 3D representations.

Synthetic surfaces as described in section 4.10 are used to verify the optimization method. The optimized surface has some difference in height, which are not relevant for the normal vectors, because the y-axis has small numbers compared to the x-axis as seen in figure 7.9 and 7.10.

The optimization of scene parameters is a crucial part in this work and the results (for scene parameters) have to be very accurate, because the parameters are essential for the transformation between surface space to image space, which is used in the next step for predicting the surface structure with a neural network. Another way of determination is also to measure some of them e.g. light source position, camera position, etc. Nevertheless, measuring the scene parameters lead also to some failures, because every measurement system has a tolerance in accuracy and I would say it would be hard to get better parameters by measuring them.

Predicting normal vectors given input images from the Freesurf sensor is the overall goal. The optimization method is the first part for solving this task. A good estimation of parameters leads to a better result in the second step. Experimental results in section 7 confirms the quality of the methods and the scene parameters are efficient enough to use them as transformation between surface space and image space.

9.2 Surface prediction with a customized Neural Network

A neural network or machine learning approach needs always a lot of data, which is used to train the method. The cabin-cap problem has only a few image samples (less than 50 image samples were available at the project start) and a big question in designing some methods arises: How to implement a method with such a low number of data points? One of the first idea was to take more image samples, which would increase the amount of data. Nevertheless, more image samples would not solve the problem with unlabeled data, because no ground truth for the surface structure exist. All this facts cannot be easily solved by using the image samples for the Freesurf sensor. This was the reason for creating a function, which is able to create synthetic surfaces. Synthetic surfaces have some pros and cons. A big advantage for synthetic surfaces is, that an infinite amount of labeled data exist. However, if synthetic surfaces are not realistic enough compared to the real image samples, the network might not be able to learn the real problem. Therefor, the creation of synthetic surfaces must be sufficient enough. The customized method invented to create synthetic surfaces is relative complex and the probabilistic approach lead to a high spectrum of different surface structures.

Different training loops (section 4.9) are used for training the Neural Network. It is interesting to see a better performance by using the encoder decoder training loop (figure 4.12) instead of a more intuitive training loop (figure 4.11). An information gain could be observed by using the loss function in image space by the use of a encoder decoder loop, thus the network is able to learn faster (with less iterations). However, all results in section 7.2 are satisfying and all of them can be used to estimate the roughness of a cabin-cap material, which should be the overall goal.

9.3 Outlook

The cabin-cap problem arises in the production line of the company "FACC" and the Freesurf sensor was designed to tackle this issue. However, before this work it was not possible to get a well surface structure as output from sensor data. The trained neural networks is now able to predict the surface structure with data from the Freesurf sensor. Theoretically results shows the quality of the predicted surface and figure 7.31 can be used as quantification.

The whole code is located in the following Github repository: `https://github.com/Kiesi1991/Surface-Reconstruction.git`

In the repository are some ZIP-files, which includes

realSamples.zip: all cabin-cap image samples taken from Freesurf sensor

realSamples1.zip: one specific cabin-cap image sample, which was used to get scene parameters

scene-parameters.zip: results of scene parameters - optimization was made with realSamples.zip

trained-models.zip: includes all relevant trained neural networks: model 2, 2.0, 2.1, 2.2 and 8

Now it is time to confirm the results by using the trained neural networks in practice.

Bibliography

[1] Hiroharu Kato et al. "Differentiable rendering: A survey". In: *arXiv preprint arXiv:2006.12057* (2020) (cit. on pp. 1, 6).

[2] Nikhila Ravi et al. "Accelerating 3d deep learning with pytorch3d". In: *arXiv preprint arXiv:2007.08501* (2020) (cit. on pp. 1, 6, 7, 13, 14, 24).

[3] Shichen Liu et al. "Soft rasterizer: A differentiable renderer for image-based 3d reasoning". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 7708–7717 (cit. on pp. 2, 6, 7, 10, 11, 12, 13).

[4] R Guy and M Agopian. "Physically Based Rendering in Filament". In: *Website available at https://google.github.io/filament/Filament.html, accessed 19.09.2022* (2018) (cit. on pp. 3, 7, 29, 30, 37, 60).

[5] Morten S Mikkelsen. "Bump mapping unparametrized surfaces on the gpu". In: *Journal of graphics, gpu, and game tools* 15.1 (2010), pp. 49–61 (cit. on pp. 5, 28, 120).

[6] Krishna Murthy Jatavallabhula et al. "Kaolin: A pytorch library for accelerating 3d deep learning research". In: *arXiv preprint arXiv:1911.05063* (2019) (cit. on pp. 7, 15).

[7] Brent Burley and Walt Disney Animation Studios. "Physically-based shading at disney". In: *ACM SIGGRAPH*. Vol. 2012. vol. 2012. 2012, pp. 1–7 (cit. on p. 7).

[8] Matthew M Loper and Michael J Black. "OpenDR: An approximate differentiable renderer". In: *European Conference on Computer Vision*. Springer. 2014, pp. 154–169 (cit. on pp. 8, 9, 11).

[9] Volker Blanz and Thomas Vetter. "A morphable model for the synthesis of 3D faces". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 1999, pp. 187–194 (cit. on p. 9).

[10] André Jalobeanu, Frank O Kuehnel, and John C Stutz. "Modeling images of natural 3D surfaces: Overview and potential applications". In: *2004 Conference on Computer Vision and Pattern Recognition Workshop*. IEEE. 2004, pp. 188–188 (cit. on p. 9).

[11] Cristian Sminchisescu. "Estimation algorithms for ambiguous visual models: Three dimensional human modeling and motion reconstruction in monocular video sequences". PhD thesis. Institut National Polytechnique de Grenoble-INPG, 2002 (cit. on p. 9).

[12] Wenzheng Chen et al. "Learning to predict 3d objects with an interpolation-based differentiable renderer". In: *Advances in Neural Information Processing Systems* 32 (2019), pp. 9609–9619 (cit. on pp. 9, 13).

[13] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. "Neural 3d mesh renderer". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 3907–3916 (cit. on pp. 10, 11, 28).

[14] Steve Marschner and Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2018 (cit. on p. 10).

[15] Stanford Vision Lab. *ImageNet*. `https://www.image-net.org/`. accessed 20.10.2022. 2010 (cit. on pp. 16, 54, 113).

[16] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012) (cit. on p. 17).

[17] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9 (cit. on pp. 17, 55).

[18] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on pp. 18, 19, 54, 55, 98, 100).

[19] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on pp. 18, 113).

[20] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256 (cit. on p. 18).

[21] Yann LeCun et al. "Neural networks: Tricks of the trade". In: *Springer Lecture Notes in Computer Sciences* 1524.5-50 (1998), p. 6 (cit. on p. 18).

[22] Saining Xie et al. "Aggregated residual transformations for deep neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1492–1500 (cit. on pp. 19, 20, 55, 56, 73, 99, 100).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90 (cit. on p. 19).

[24] Christian Szegedy et al. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9 (cit. on p. 21).

[25] Jiahui Yu et al. "Coca: Contrastive captioners are image-text foundation models". In: *arXiv preprint arXiv:2205.01917* (2022) (cit. on p. 21).

[26] Kurt Endl and Wolfgang Luh. "Analysis". In: *Aula, Wiesbaden* 7 (1989), pp. 375–387 (cit. on p. 36).

[27] Wikipedia. *Partial Derivative*. `https://en.wikipedia.org/wiki/Partial_derivative`. accessed 19.09.2022. 2022 (cit. on p. 36).

[28] Wikipedia. *Inverse-square law*. `https://en.wikipedia.org/wiki/Inverse-square_law`. accessed 19.09.2022. 2022 (cit. on p. 37).

[29] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. "Segnet: A deep convolutional encoder-decoder architecture for image segmentation". In: *IEEE transactions on pattern analysis and machine intelligence* 39.12 (2017), pp. 2481–2495 (cit. on p. 41).

[30] Scipy. *scipy.ndimage.gaussian_filter*. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.gaussian_filter.html`. accessed 13.10.2022. 2022 (cit. on p. 49).

[31] Imanol Bilbao and Javier Bilbao. "Overfitting problem and the over-training in the era of data: Particularly for Artificial Neural Networks". In: *2017 eighth international conference on intelligent computing and information systems (ICICIS)*. IEEE. 2017, pp. 173–177 (cit. on p. 97).

[32] Wikipedia. *Optical microscope*. `https://en.wikipedia.org/wiki/Optical_microscope`. accessed 15.12.2022. 2022 (cit. on p. 116).

[33] Wikipedia. *Atomic force microscopy*. `https://en.wikipedia.org/wiki/Atomic_force_microscopy`. accessed 15.12.2022. 2022 (cit. on p. 116).

[34] Wikipedia. *Scanning electron microscope*. `https://en.wikipedia.org/wiki/Scanning_electron_microscope`. accessed 15.12.2022. 2022 (cit. on p. 116).

[35] Wikipedia. *X-ray reflectivity*. https://en.wikipedia.org/wiki/X-ray_reflectivity. accessed 15.12.2022. 2022 (cit. on p. 117).

[36] Wikipedia. *Fresnel equations*. https://en.wikipedia.org/wiki/Fresnel_equations. accessed 15.12.2022. 2022 (cit. on p. 117).

[37] Wikipedia. *Lidar*. https://en.wikipedia.org/wiki/Lidar. accessed 15.12.2022. 2022 (cit. on p. 117).

[38] Wikipedia. *Image analysis*. https://en.wikipedia.org/wiki/Image_analysis. accessed 15.12.2022. 2022 (cit. on p. 117).