Computer Graphics and Multimedia Systems Group
Institute for Vision and Graphics
University of Siegen

Master Thesis

Fluid Surface Reconstruction Using a Time-of-Flight Camera

Jan Christopher Grimm

14 September 2015

Supervisors:
Prof. Dr. Andreas Kolb, Dr.-Ing. Martin Lambers

Lehrstuhl für Computergraphik und Multimediasysteme
Institut für Bildinformatik
Universität Siegen

Masterarbeit

Rekonstruktion von Flüssigkeitsoberflächen mit einer Time-of-Flight Kamera

Jan Christopher Grimm

14. September 2015

Betreuer:
Prof. Dr. Andreas Kolb, Dr.-Ing. Martin Lambers

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Siegen, den 14. September 2015

Jan Christopher Grimm

Zusammenfassung

Time-of-flight Kameras liefern genaue Entfernungsinformationen, indem sie ein Lichtsignal emittieren und die Zeit messen, bis das Signal wieder bei der Kamera eintrifft. Diese Kameras sind besonders nützlich für Nahbereichsanwendungen innerhalb weniger Meter. Eine Flüssigkeitsoberfläche wie etwa Wasser mit solch einer Kamera aufzunehmen und zu rekonstruieren ist eine Herausforderung: Da das Wasser transparent ist, ist es nicht direkt sichtbar und kann nicht direkt gemessen werden. Dennoch hat es einen messbaren Effekt auf das Lichtsignal, das von der Kamera emittiert wird.

In dieser Abschlussarbeit stellen wir eine Methode zur Rekonstruktion von Flüssigkeitsoberflächen vor, die sich auf Entfernungsinformationen einer einzelnen Kamera stützt. Unser Analyse-durch-Synthese Ansatz nutzt ein physikalisch basiertes Modell der Interaktion zwischen Wasser und dem Kamerasignal. Die hauptsächliche Motivation für dieses Modell ist die Lichtbrechung an der Wasseroberfläche. Das Modell erzeugt synthetische Entfernungsbilder aus einem die Wasseroberfläche repräsentierenden Höhenfeld. Wir zeigen, wie diese simulierten Entfernungen zusammen mit einem Optimierungsalgorithmus genutzt werden können, um die Oberfläche aus einer Messung unserer Kamera wiederherzustellen. Wir beginnen mit einem einfachen Modell, das aufgrund von Messungen verfeinert wird.

Wir beschreiben einen Versuchsaufbau, der es uns erlaubt sowohl dynamische als auch statische Wasseroberflächen aufzunehmen. Für diesen Aufbau betrachten wir mögliche Fehlerquellen die allen time-of-flight Kameras gemein sind. Wir erklären auch die nötigen Kalibrierungsschritte für die Kamera selbst. Indem wir steigende Wasserhöhen mit diesem Aufbau messen, zeigen wir die Notwendigkeit der Verfeinerung unseres initialen Modells.

Wir implementieren die Oberflächenrekonstruktion als einen proof of concept Prototypen. Das Hauptziel unserer Implementierung ist nicht Performanz, sondern die Korrektheit der Ergebnisse. Anschließend evaluieren wir die Rekonstruktionsergebnisse von simulierten und echten Datensätzen. Die simulierten Daten dienen hauptsächlich der Überprüfung, dass die Optimierung stabil ist. Echte Daten werden für flache Wasseroberflächen und Wellen ausgewertet. Wie wir sehen werden, wird das Kamerarauschen ein limitierender Faktor für die Rekonstruktion von Realdaten.

Abstract

Time-of-flight cameras provide accurate distance information by emitting a light signal and measuring the time until the signal arrives back at the camera. These cameras are particularly useful for close range applications within a few meters. Capturing and reconstructing the surface of a liquid such as water with a camera like this is a challenging task: Since the water is transparent, it is not directly visible, and thus cannot be measured directly. However, water still has a measurable effect on the light signal emitted by the camera.

In this thesis, we propose a method for liquid surface reconstruction which relies on the distance information from a single time-of-flight camera. Our analysis-by-synthesis approach uses a physics-based model of the interaction between water and the camera signal. The main motivation for this model is the refraction of light at the water surface. The model creates synthetic distance images from a height field representing the water surface. We demonstrate how these simulated distances can be used in conjunction with an optimization algorithm to reconstruct the surface from a measurement of our camera. We start with a simple model, which is refined based on the results of our experiments.

We describe an experiment setup which allows us to capture both dynamic and static water surfaces. For this setup, we consider potential error sources that are common to all time-of-flight cameras. We also explain the necessary calibration steps for the camera itself. By measuring increasing water heights using this setup, we demonstrate the need to refine our initial model.

We implement the surface reconstruction as a proof of concept prototype. The main goal of our implementation is not performance, but the correctness of the results. We then evaluate the reconstruction results for simulated and real data sets. The simulated data serves mostly to check the stability of the optimization. Real data is evaluated for flat water surfaces and waves. As we will see, the camera noise becomes a limiting factor in the reconstruction of real data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Depth cameras are specialized cameras for gathering range information. When a regular camera captures an image or video sequence, each pixel provides intensity or color information. In contrast to this, a depth camera measures distances. Each pixel of a depth camera image contains distance information. This information can be used to reconstruct a three-dimensional model of the scene that was captured. An example application field for these cameras is close range sensing in robotics. In recent years, entertainment solutions like the Microsoft Kinect have made depth cameras more ubiquitous. A time-of-flight camera is a special type of depth camera which calculates the distance information from the travel time of light. To do this, the camera actively emits a light signal.

In this thesis we describe a novel method for reconstructing liquid surfaces from time-of-flight camera data. Our method uses a single camera to acquire range information. We focus our efforts on the reconstruction of water surfaces. Since water is a transparent medium, it cannot be measured directly. Instead, we rely on the interaction between the light emitted by our camera and the water surface. Reconstructed water surfaces could be used to verify physical simulation results, such as a fluid particle simulation. Existing methods use regular cameras for the reconstruction. Our method presents a first attempt at using a time-of-flight camera for reconstruction. Just proving the possibility of utilizing such a camera for this purpose was one of our goals, as we did not find any previous attempts at this at the time of writing.

The main idea of our approach is to use the refraction of light at the water surface as the basis of a model. The aim of this model is to replicate the influence of water on the signal of a time-of-flight camera. We rely mainly on refraction at the water surface, which changes the direction of the light signal. Light also travels at a lower speed in water. Our model is used by an optimization algorithm to match synthetic distance images with real measurements. This analysis-by-synthesis approach requires the model to match reality as accurately as possible. By refining the model, the reconstruction results are improved indirectly. The main challenges when applying our method on real data are camera noise and undesirable reflections.

The following chapters describe and evaluate our method in greater detail. Chapter 2 contains basic concepts which are used throughout this thesis, like our notational conventions. Here we introduce time-of-flight depth cameras and describe how these cameras work. We also describe the simple pinhole camera model that is commonly used to as a theoretical representation of a camera. The chapter concludes with a comparison to another reconstruction method based on an array of regular cameras.

In Chapter 3 we take a close look at the actual problem we are trying to solve: Reconstructing the water surface using only range information from a single camera. After establishing our goals, we examine possible interactions between a light signal and a water surface. From these interactions we derive a refraction-based model for the distance measurement, which can be used to simulate the effect of a water surface on a time-of-flight camera image. We explain how this model can be used together with an optimization algorithm to reconstruct the water surface from a measurement.

Chapter 4 then describes the setup of our experiment. The main motivation for many choices during the design of the experiment was the elimination of error sources. We describe these error sources, some of which are time-of-flight camera specific, and how they have been addressed. We also explain the necessary calibration steps for such a camera. Our experiment setup is described in some detail. At the end of the chapter, there is an overview of the different types of experiments that were conducted using our setup. These experiments cause a refinement of our initial model.

For the software side of our method, Chapter 5 provides most of the details. This includes reasoning about why some software libraries were chosen. An overview of our implementation of the surface reconstruction concludes this chapter.

The evaluation of our method takes place in Chapter 6. We evaluate the reconstruction of both simulated and real water surfaces. For real water surfaces, we further distinguish between static water and dynamic water, which presents its own set of problems. The limitations of our approach are also explained at the end of this chapter.

The final Chapter 7 provides a summary of this thesis, as well as an outlook on possible ways to improve our method in the future.

Chapter 2

Foundation

As we have established in Chapter 1, our goal is to reconstruct a water surface from data captured by a time-of-flight camera. Before we can go into greater detail on how water can potentially interact with the camera signal, we first need to take a step back and take a look at how the camera measures the range data. We can also examine the difference between our approach and an approach based on regular cameras. This chapter starts with an overview of the notational conventions used throughout this thesis. Then, we explain the general working principle of time-of-flight depth cameras. As we will see, the camera uses a light signal to calculate the distance towards a scene. After explaining the principle of a time-of-flight camera, we will describe the pinhole camera model that is commonly used to represent the perspective projection. With this model we are able to calculate the viewing direction at each individual pixel. Lastly, we present an approach which attempts the same goal by using an array of regular cameras. We also take a look at how our method differs from this prior work. In Chapter 3, we will develop a reconstruction method for water surfaces captured by a time-of-flight camera. The model we use for the reconstruction will take the interaction between water and the camera signal into account. It also relies on the availability of viewing directions for each pixel, which we can obtain using the pinhole camera model.

2.1 Notational Conventions

The notation we use for our mathematical symbols and expression is for the most part fairly standard. Since there are some exceptions that warrant explanation, we will describe the conventions we use in some detail. In case there is any confusion in a later chapter, the reader can refer back to this section and hopefully clear up the confusion.

Table 2.1 shows some examples of our conventions. To distinguish between vectors and scalar values, vectors are printed in bold. Unless otherwise noted, the vectors consist of three scalar components. Most of the time, they identify positions in a Cartesian coordinate system. Normalized vectors are used for directions in this coordinate system. The matrices we use here are always two-dimensional, and are denoted with capital letters. Their two-dimensional nature implies a neighborhood relationship between elements. One use of these matrices is to represent images, as these are also laid out in two dimensions.

The individual components of a matrix can be either scalar or vector values. We use bold font again to distinguish between these alternatives. Note that a single matrix cannot contain both scalar and vector values. When talking about an individual element of a matrix, we use a subscripted row and column indices to access that element. Elements of a matrix use the same symbol in lower case instead of the capital letter. Individual elements of a vector can also be

Type of expression	Notation
Scalar value	s
Vector	$\boldsymbol{p}$
ith element of a vector	p_i
Scalar product between vectors	$\boldsymbol{p} \cdot \boldsymbol{q}$
Normalized vector	$\hat{\boldsymbol{v}}$
Matrix of scalar values	M
Element at the ith row and jth column of the same matrix	$m_{i,j}$
Matrix of vector values	$\boldsymbol{N}$

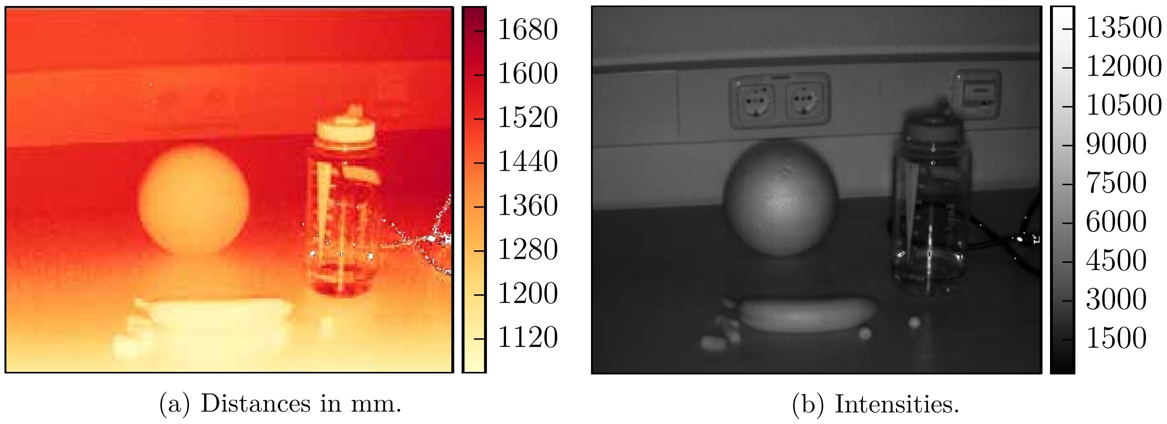
Table 2.1: Notation overview.

accessed with an index. These elements are not printed in bold to help distinguish them from the vector to which they belong.

Functions use similar conventions: The function name describes the result type of the function. For the individual function parameters we use the same conventions as before. Together they describe the domain of the function.

2.2 Time-of-Flight Camera Principles

Time-of-flight cameras can be used to gather distance information for each pixel of a captured scene. In addition to this, some cameras are also capable of producing intensity images. Figure 2.1 shows two views of the same example scene. The distance image contains distance information for each pixel, whereas the intensity image results mostly from the reflectivity of the scene. Pixels that contain no reliable information have been marked white, as can be seen on the bottle and the cable behind it.



(a) Distances in mm. (b) Intensities.

Figure 2.1: Distance and intensity image of the same scene.

The camera that was used for the majority of the work presented in this thesis is pictured in Figure 2.2. This camera is a prototype from pmdtechnologies GmbH called the Photonic Mixer Device (PMD) DigiCam. It offers a resolution of 352 × 288 pixels, using a specialized

CMOS-sensor. For our application we changed the default lens to a lens with a focal length of 12.8 mm. This results in a narrower field of view of roughly 40°, but also meant that we had to remove the protective front plate of the camera.

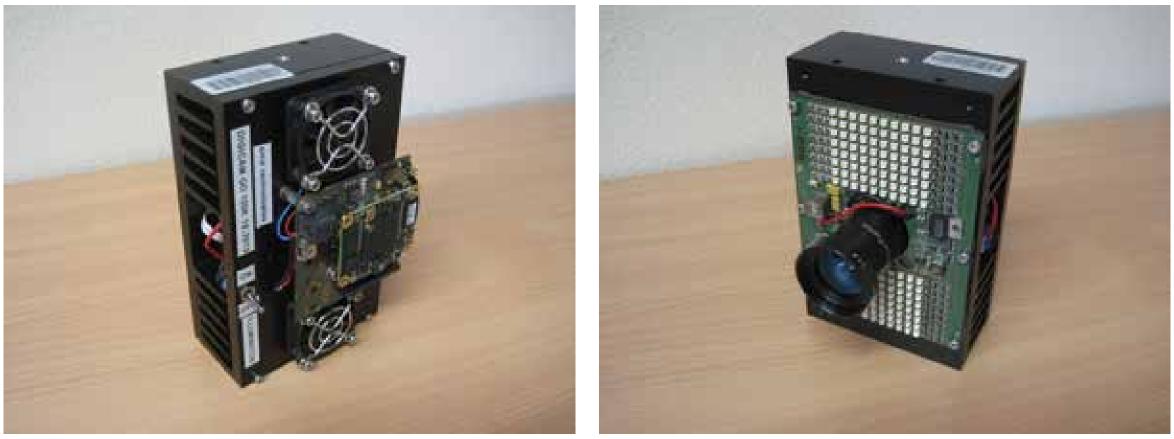

(a) Back view. (b) Front view.

Figure 2.2: The PMD DigiCam.

The rectangular grids in Figure 2.2b above and below this lens are LED arrays. These LED arrays emit infrared light at a wavelength of 850 nm and are used to actively illuminate the scene. When the camera is operated for longer periods of time, these illumination units can become quite warm. Thus, they require active cooling fans, which can be seen in the back view of the camera in Figure 2.2a. The body of the camera also functions as a heat sink in conjunction with these fans.

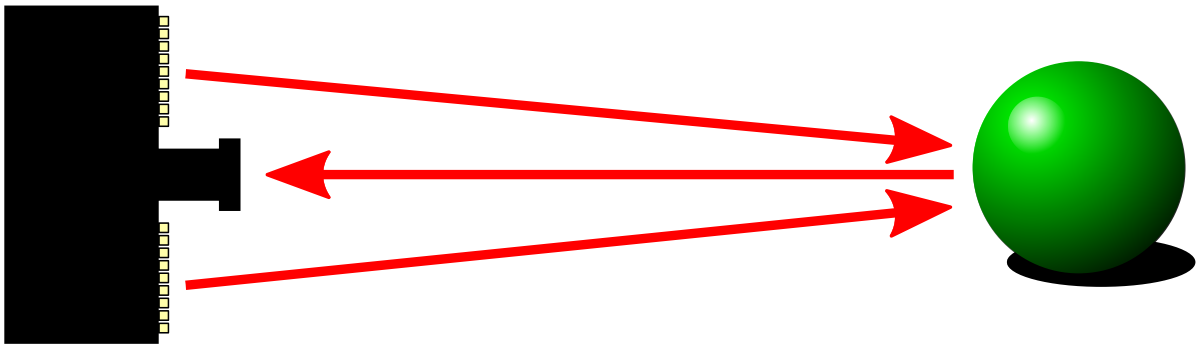


Figure 2.3: The time-of-flight camera principle.

The following description is based on [Lan00]. The working principle of a time-of-flight camera can be explained by a simplified model. Figure 2.3 shows an overview of this process. The camera actively illuminates the scene it is about to capture by emitting a light signal. Traveling at the speed of light c, the signal is reflected from the scene and arrives back at the camera sensor. For each pixel of the sensor, the camera measures the time between signal emission and the return of the signal. In theory, this time difference is all that is needed to calculate the distance to each corresponding point in the scene.

In practice, the travel time is not measured directly. Instead, the phase delay between two signals is used. To measure this phase delay, the camera modulates the intensity of the emitted light signal, resulting in a continuous wave. This wave is characterized by its modulation frequency f. An example value for f would be 30 MHz, which leads to a wavelength $\lambda = c/f$ of roughly 10 m. The wave is used as the reference signal on which the time-of-flight calculation is based. The reference signal is described as a sinusoidal wave

$$s(t) = \cos(\omega t), \tag{2.1}$$

where t is the time and $\omega = 2\pi f$ is the angular frequency of this wave.

The signal that arrives back at the camera reflects the changes applied to the reference signal during the travel time and interaction with the scene. While the frequency remains the same, the phase of the signal has shifted by a phase delay φ as a direct result of the travel time. Due to varying reflectivity of the scene, the reflected signal has a different amplitude a. A background intensity bias b is also always present as an offset. Taking all this together, the reflected signal arriving back at the camera is modeled as

$$r(t) = b + a\cos(\omega t + \varphi). \tag{2.2}$$

The camera sensor does not measure these two signals in isolation. Instead, the correlation

$$c(\tau) = r(t) \otimes s(t) = \lim_{t_i \to \infty} \frac{1}{t_i} \int_{-\frac{t_i}{2}}^{\frac{t_i}{2}} r(t)s(t+\tau)\,\mathrm{d}t \tag{2.3}$$

of the two signals is calculated directly in hardware. These specialized sensors are also referred to as 'smart pixels'. The correlation function $c(\tau)$ can be evaluated for a certain integration time t_i. The parameter τ represents a controlled phase interval for which $c(\tau)$ is sampled.

Equation (2.3) is simplified to

$$c(\tau) = b + \frac{a}{2}\cos(\omega\tau + \varphi) \tag{2.4}$$

by [Lan00]. [Hah12] describe a more detailed derivation of this simplification. This new correlation function is then sampled at different intervals. Most commonly, the four intervals $\tau_0 = 0$, $\tau_1 = \pi/2$, $\tau_2 = \pi$ and $\tau_3 = 3\pi/2$ are used. Sampling of an example correlation function with these intervals is shown in Figure 2.4.

When the correlation function is sampled at these intervals, the result is a series of intensity values

$$c_0 = c(\tau_0) = b + \frac{a}{2}\cos\varphi, \tag{2.5}$$

$$c_1 = c(\tau_1) = b - \frac{a}{2}\sin\varphi, \tag{2.6}$$

$$c_2 = c(\tau_2) = b - \frac{a}{2}\cos\varphi, \tag{2.7}$$

$$c_3 = c(\tau_3) = b + \frac{a}{2}\sin\varphi. \tag{2.8}$$

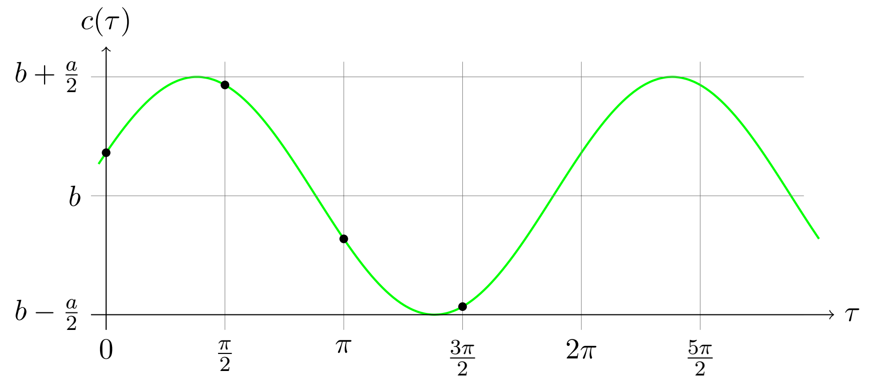


Figure 2.4: The correlation function, sampled at four intervals.

These raw values are used to compute the phase delay, but they also provide a way to calculate the amplitude of the signal, as well as an intensity bias.

To get the phase delay, we can subtract c_1 from c_3, as well as c_2 from c_0. This eliminates the background offset b in both cases. Division of the two resulting terms removes the amplitude a as well, leaving only $\sin\varphi/\cos\varphi = \tan\varphi$. From this, the phase delay can be calculated as

$$\varphi = \arctan\frac{c_3 - c_1}{c_0 - c_2}, \tag{2.9}$$

the result of which is an angle in the range $(-\pi/2, \pi/2)$.

One way to visualize the arctan function is by means of a vector (x, y) in polar coordinates. $\arctan(y/x)$ returns the angle between this vector and the positive x-axis for positive x values. A shortcoming of the arctan function is that it cannot inspect the sign of x and y. The result is that for negative x values, the angle cannot be placed in the correct quadrant of the polar coordinate system. To correct this, many programming languages provide the atan2 function.

$\operatorname{atan2}(y, x)$ is applied to two arguments, which represent the same vector components. By inspecting the sign of its arguments, the function can place the resulting angle in the correct quadrant. atan2 returns an angle in the range $(-\pi, \pi]$, which covers all quadrants of the polar coordinate system.

Rewriting Equation (2.9) in terms of atan2 gives us

$$\varphi = \operatorname{atan2}(c_3 - c_1, c_0 - c_2), \tag{2.10}$$

which means that the phase delay can take any value between $-\pi$ and π, covering the full period of the reference signal. If the range is instead desired to be $(0, 2\pi]$, we can add 2π to all negative phase delays.

Once the phase delay is known, the distance

$$d = \frac{c\varphi}{2\omega} \tag{2.11}$$

can be calculated. Division of the phase delay by the angular frequency leads to the total time-of-flight. This term is divided by two to account for the fact that the light had to travel the same distance twice, once from the camera to the scene, and once from the scene back to the camera. The final distance is the result of multiplying this time by the speed of light.

A consequence of measuring the time-of-flight indirectly using the phase delay is that only values within the range $(0, 2\pi]$ are unambiguous. Considering again that the light needs to travel back and forth, this means that only objects within half the wavelength of the reference signal can be detected. For a modulation frequency of 30 MHz, this unambiguity range is $c/(2f) \approx 5\,\mathrm{m}$. For any object outside this range, the phase delay is not measured correctly, which results in an incorrect distance as well. Increasing the modulation frequency increases the spatial resolution, at the expense of shortening this unambiguity range.

The above process was focused on individual pixel values. In reality, the process is repeated for each pixel of the image sensor. This means that the camera captures four raw images which represent the correlation function, sampled four times per pixel. From these four raw images, a single phase image is calculated, which then results in a single distance image. The amplitude and intensity bias likewise form a complete image.

By summing up the four raw values and dividing by four, the background intensity bias

$$b = \frac{c_0 + c_1 + c_2 + c_3}{4} \tag{2.12}$$

is retrieved. This bias is a constant offset that is present in every raw value. In a grayscale image it can also be used for calibrating the camera. This image will show the same scene as the distance image, illuminated by the reference signal. Since this reference signal is near infrared, the image can look quite different from what one would expect. A color pattern that is uniformly reflective in infrared will show up as a uniform area in this image.

The amplitude a can also be extracted from the four raw values in the following way:

$$a = \frac{\sqrt{(c_3 - c_1)^2 + (c_0 - c_2)^2}}{2}. \tag{2.13}$$

This amplitude can then be used to filter out pixels that are overexposed, or pixels where the returned signal was too weak to carry any meaningful information. These pixels are then flagged as invalid by the camera software, since their distance information cannot be relied on.

2.3 Pinhole Camera Model

In computer graphics, the camera is often described by an idealized pinhole camera model. An advantage of this model is its simplicity. It describes the relationship between pixel coordinates of an image and a three-dimensional camera coordinate system. Even though real cameras are typically not pinhole cameras, the model can also used for them. As we will see in Chapter 4, for a real camera lens, distortion effects need to be corrected as well. For our theoretical model of a time-of-flight camera, the pinhole camera model is sufficient.

The pinhole camera model is described by a set of parameters. These parameters are also called the extrinsic parameters of the camera, since they only depend on the camera itself, not on the scene. Conceptually, the camera coordinate system and the image plane are aligned in the following way: The x and y axis of the camera coordinate system are aligned with the u and v axis of the image plane. The origin of the camera coordinate system represents the focal point of the camera. The origin of the image plane is the top left corner of the image. The image plane is positioned at a certain focal length f along the z axis of the camera coordinate system. The intersection between image plane and z axis is called the principal point (c_x, c_y). The aspect ratio of the image can be controlled by using different focal lengths f_x and f_y for the x and y axis. All of these values have to be provided in pixel units.

The perspective projection described by this camera model can be expressed as a matrix

$$C = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

which maps camera coordinates to image coordinates. Multiplying C with a vector $\boldsymbol{x}$ results in a vector $\boldsymbol{p}$. If this vector is divided by its z component, its first two components become the coordinates of the projected point on the image plane.

Likewise, the inverse camera matrix

$$C^{-1} = \begin{bmatrix} \frac{1}{f_x} & 0 & -\frac{c_x}{f_x} \\ 0 & \frac{1}{f_y} & -\frac{c_y}{f_y} \\ 0 & 0 & 1 \end{bmatrix} \tag{2.15}$$

can be used to convert points on the image plane to vectors in the camera coordinate system. For this purpose, the pixel coordinates of the point are extended by a z component, which is set to 1. Multiplying C^{-1} with this vector results in a vector $\boldsymbol{v}$, which describes the viewing direction from the camera origin to this pixel on the image plane. This process can be used to calculate the viewing direction of any pixel if the intrinsic parameters of the camera have been obtained by a calibration.

The relationship between the camera and the scene is described using a rotation and translation. These extrinsic parameters provide a mapping from world coordinates to camera coordinates. If the camera and scene are not moved relative to each other, it is also possible to perform all calculations directly in camera coordinates.

2.4 Prior Work

Capturing and reconstructing a water surface is a task that has been attempted before. We will highlight how our approach differs from these surface reconstruction methods by example of the most recent attempt we could find. To the best of our knowledge, no method involving a time-of-flight camera has been published so far.

The setup of [Din+11] relies on a camera array to acquire images of a moving water surface. The cameras themselves are regular cameras which capture intensity information. Nine such cameras are arranged in a 3×3 grid and placed above a small glass tank. This tank contains both the water and a checkerboard pattern, which is placed on the floor of the tank. Figure 2.5 shows the fully assembled setup.

To reconstruct the water surface, the checkerboard corners have to be tracked in each frame. The refraction of the moving water surface distorts the position of these corners. The same distortion can also mean that a camera loses track of a checkerboard corner. According to

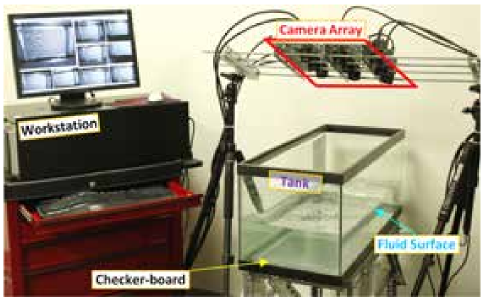


Figure 2.5: The camera array positioned above the tank. The image was taken from [Din+11].

[Din+11], an advantage of using multiple cameras is that in case this happens to a single camera, another camera can fill in the gap. The tracked corners are then used to calculate a distance map, which in turn allows the authors to reconstruct the water surface. The main influence on the measured images is the refraction.

The cameras used in their setup can capture images at a rate of 30 frames per second. By delaying the capturing of some of the cameras and interleaving the results, the effective frame rate is increased to 60 frames per second. This is done in order to reduce motion artifacts. The authors compare their setup to camera arrays with 64 or even 128 cameras. For these arrays, computing clusters are necessary to allow the recording of data. Even though a single workstation can record data from the nine cameras used here, an array of six external hard drives is still required.

Our approach differs from this method in several crucial ways. First, by using a time-of-flight camera, we are able to directly capture distance information, instead of having to calculate it from point correspondences. This also frees us from tracking a checkerboard pattern placed on the floor of the setup. Instead, any reflective floor plane can be used. However, a flat floor is still a requirement. Since we can acquire distance information for each pixel, the resolution of our camera is the equivalent of the resolution of the checkerboard pattern used by [Din+11].

The second import difference is that we only use a single camera. This simplifies the acquisition of the data, which means we can use any laptop or desktop workstation to record our images instead of specialized equipment. Since we avoid having to track checkerboard corners over time, the major drawback of using a single camera and losing track of a corner does not concern us. A simple reason for using just a single depth camera was availability. Calibration of the camera is also easier compared to calibrating a camera array.

Our approach uses similar information to reconstruct the water surface. We also base our model mainly on the refractive properties of water.

Chapter 3

Fluid Surface Reconstruction

In Chapter 2, we explained the working principle of a time-of-flight camera. The main takeaway from this chapter is that the camera uses a light signal to illuminate a scene and calculate the distance. We also saw an example of a method for reconstructing water surfaces using regular cameras. Our goal now is to define a new method for using a single time-of-flight camera for the same purpose. In this chapter, we start with an overview of the problem that we wish to solve. We then examine possible influences of the water on the camera signal. From these observations, we derive a model to describe these interactions. This model can be used to generate synthetic distance images for a simulated water surface. It is camera-independent and possible to use with any time-of-flight camera. In a second step, we describe how an optimization algorithm can be used to obtain the water surface from a distance measurement. The model is used to match simulated distances with real distances. In Chapter 4, we describe the experiment setup that was used to test our surface reconstruction. An example implementation of the model and the reconstruction is presented in Chapter 5. The results of using this implementation in conjunction with our experiment are discussed in Chapter 6.

3.1 Problem Definition

As we saw in Chapter 2, a time-of-flight camera captures images which contain distance information for each individual pixel. It calculates this information by emitting a signal and measuring the phase delay between signal emission and the return of the reflected signal. This phase delay is proportional to the actual time-of-flight, from which the distance is reconstructed. A transparent medium, such as water, cannot be measured directly by this type of depth camera. This is due to the fact that by definition the transparent medium lets most of the light pass through, instead of directly reflecting it. However, the presence of a transparent medium in a scene can still have a measurable influence on the distance information.

In this thesis our goal is to utilize this effect to reconstruct the surface of a water layer. We represent this water surface as a height field, containing a scalar height value for each pixel. Our only sources of information about the water surface are distance images captured by a single time-of-flight camera. The presence or absence of water in these images has a measurable effect on the distances. Moving water also changes the distance information. Given such a distance image, our method must reconstruct a representation of the water surface.

The first and foremost concern with regards to this reconstructed water height field is correctness. We need to be able to reliably reconstruct a water surface using only the distance information. This also means that we have to work around some of the shortcomings of the camera we use. At the same time, our method has to remain general enough to be usable with any time-of-flight camera. It should also be possible to exchange water for another transparent medium. However, for the sake of convenience we restrained all our experiments to water.

Since the camera is capable of recording consecutive frames, moving water surfaces can also be used as a goal for the reconstruction. While the ability to reconstruct water surfaces in real time would be a nice feature, the correctness of the results has to take precedence over the performance of our method..

To enable the reconstruction of water heights from distance information, we employ a simple model based on a physical property of water: Broadly speaking, refraction at the water surface is the main motivation of our model.

3.2 Physical Properties of Water

Water can interact with light in various ways. In this section, we will concentrate on these main interactions:

- Refraction
- Reflection
- Absorption

Refraction causes a change in direction of the light as it moves from one medium to another. Reflection represents the portion of the light that is mirrored from the surface instead of entering the second medium. Absorption results in a loss of intensity as light travels through a medium. These properties are not unique to water: Any transparent medium can exhibit some or all of these interactions. In what follows, we take a closer look at these phenomena. We will also briefly discuss how these properties could affect our goal of reconstructing the water surface from a time-of-flight camera image. Because these are fairly basic concepts, they can be found in many introductory physics textbooks. We used [Har07] and [LLT13] for this section.

3.2.1 Refraction

Refraction of light is a physical phenomenon that can be observed whenever light travels from one transparent medium to another. A good example of this effect can be seen in Figure 3.1. When looking at a straw in a glass of water, the straw appears bent. The straw itself is of course entirely straight. This optical illusion is caused by refraction at the water surface. Here, the light coming from the straw changes direction. A ray of light going from the straw towards the observer is no longer a straight line. Instead, the ray is divided into two distinct segments: The first segment lies underwater, while the second segment is the part of the ray which travels through air.

Refraction can take place at every intersection of two media along a ray of light. It is caused by differences in the speed of light for each medium. The speed of light $c = 299\,792\,458\,\mathrm{m\,s^{-1}}$ is a universal physical constant and refers to the speed of light in vacuum. However, the speed of light while traveling through a transparent medium is lower than this constant. This adjusted speed of light is referred to as the phase velocity v of light in the medium. The ratio of the speed of light in vacuum and the phase velocity is called the index of refraction $\eta = \frac{c}{v}$, or

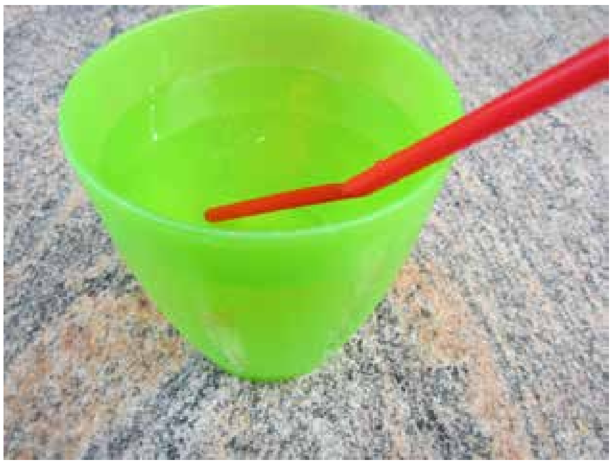


Figure 3.1: Refraction example.

refractive index of the medium. A higher index of refraction means that light traverses this medium at a lower speed. In the example, light travels faster in air than it does in water.

The direction of a ray of light after refraction depends on the refractive indices of both media. We will refer to these indices as η_1 and η_2 for the first and second medium, respectively. A ray of light traveling along direction $\hat{\boldsymbol{v}}$ in the first medium is refracted in direction $\hat{\boldsymbol{r}}$. This refraction takes place at the boundary between the two media. The angle of incidence θ_1 is the angle between the surface normal $\hat{\boldsymbol{n}}$ of the boundary and $\hat{\boldsymbol{v}}$. Conversely, the refracted angle θ_2 is measured between $\hat{\boldsymbol{n}}$ and $\hat{\boldsymbol{r}}$. Figure 3.2 illustrates how these directions and angles relate to each other.

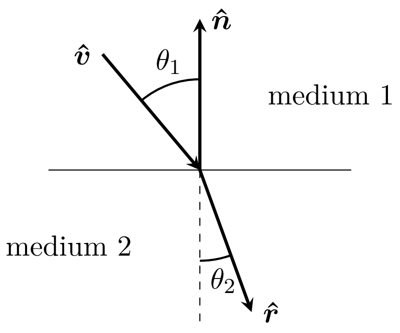


Figure 3.2: Refraction at the intersection of two media.

The relationship between the two angles θ_1, θ_2 and the two refractive indices η_1, η_2 is described by

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{\eta_2}{\eta_1}. \tag{3.1}$$

This equation is also known as Snell's law.

If η_1, η_2, $\hat{\boldsymbol{n}}$ and $\hat{\boldsymbol{v}}$ are known, $\hat{\boldsymbol{r}}$ can be calculated using this equation. One thing to note is that the medium with the higher refractive index is also the one with the smaller of the two angles. In our example, water has a higher index of refraction. As a result of this, a ray of light entering the water is refracted closer towards the surface normal.

Different transparent materials often also have different refractive indices. In addition, the refractive index depends on the wavelength of the light. Figure 3.3 contains an example plot of the refracted angle as a function of the incident angle for two fixed refractive indices.

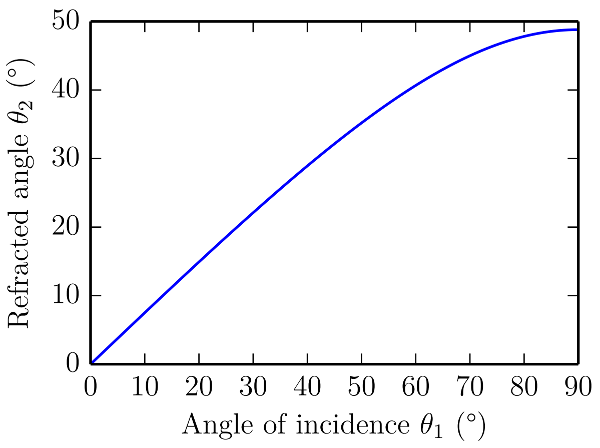


Figure 3.3: Refraction for $\eta_1 = 1.0$, $\eta_2 = 1.329$.

Refraction is the main component of the model we will use to generate synthetic images. For the signal of a time-of-flight camera, it can change the actual path towards a point of the scene. Additionally, the change to the speed of light in the second medium also has the potential to influence the overall measured distance.

3.2.2 Reflection

When a ray of light is refracted at the intersection of two media, an additional phenomenon takes place: Part of the incoming light is reflected off the surface. The light wave now traveling in the second medium is actually only a part of the original light wave. The difference between the two is explained entirely by the reflection. The reflected part of the ray of light is mirrored on the opposite side of the surface normal, as can be seen in Figure 3.4. This also means that the angle between the reflected part and the normal is exactly the same as the incident angle.

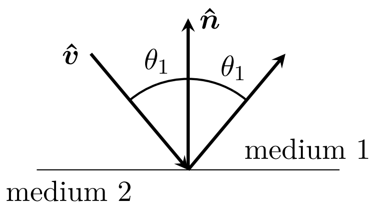


Figure 3.4: Reflection at the intersection of two media.

The percentage of the light intensity that is reflected is referred to as the reflectance r. Conversely, the part of the light intensity that passes into the second medium is called the transmittance t. The sum of these two terms always is always one.

Calculating the reflectance is usually done by solving a set of equations called the Fresnel equations. The notation used here was taken from [LLT13]. These equations describe the reflectance for two different polarizations of the same light wave.

The first of the two reflectances is that of a light wave polarized in a plane orthogonal to the plane formed by the incoming light and the intersection normal. This polarization is also referred to as s-polarization. The reflectance

$$r_s = \left(\frac{\eta_1 \cos\theta_1 - \eta_2 \cos\theta_2}{\eta_1 \cos\theta_1 + \eta_2 \cos\theta_2}\right)^2 \tag{3.2}$$

for this light wave depends on the refractive indices, as well as the angle of incidence and the angle of refraction.

The other polarization is a polarization in the plane formed by the incoming light and the intersection normal. This polarization is referred to as p-polarization. The corresponding reflectance

$$r_p = \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1}\right)^2 \tag{3.3}$$

depends on the same parameters as r_s.

The actual reflectance r for the intensity of a ray of light is given by

$$r = \frac{r_s + r_p}{2}, \tag{3.4}$$

which is the mean of the two polarized reflectance. The transmittance

$$t = 1 - r \tag{3.5}$$

can easily be calculated from this.

Figure 3.5a shows how these different terms relate to each other. The equations were evaluated for fixed indices of refraction. Since the angle of refraction can then be calculated from the angle of incidence, we can plot the transmittance and the various reflectances for this angle of incidence directly. The most important thing to note about this figure is that for small angles between the normal and the incoming light, the reflectance is actually very small. At the same time, the high transmittance in this part means that most of the light passes through the intersection. At about 50°, this begins to change. As the angle between the light and the surface becomes more and more shallow, the reflectance and transmittance rapidly change place. The same effect can be observed at a glass window, for example. Looking straight out, the reflection is barely noticeable when compared to the outside view. But when looking at the window from the side, the reflection starts to become more pronounced.

For our surface reconstruction, reflectance is considered mostly as a source of errors. It restricts the viewing angle of the camera, because we are mostly interested in the light that passes on to the floor. If the camera is looking at the surface from a shallow angle, the water surface acts more and more like a mirror. Our model is not equipped to deal with reflection, since this would require an exact knowledge of the surrounding environment. Instead, we will see in Chapter 4 how we tried to reduce the influence of reflection on our measurements.

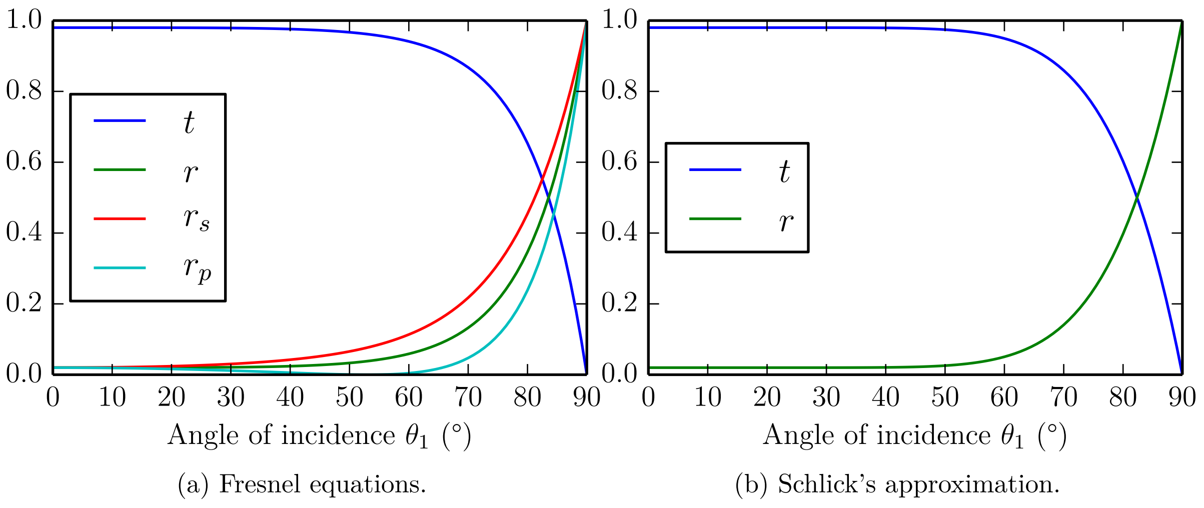


Figure 3.5: Transmittance and reflectance for $\eta_1 = 1.0$, $\eta_2 = 1.329$.

If evaluating these equations becomes to costly, an approximation can be used. A common approximation for the reflectance is that proposed by [Sch94]. This approximation is also called Schlick's approximation. A special case for Equations 3.2 and 3.3 is when the angle of incidence is zero, which means the light enters the second medium from the normal direction. In this case, the reflectance becomes

$$r_0 = \left(\frac{\eta_1 - \eta_2}{\eta_1 + \eta_2}\right)^2. \tag{3.6}$$

Schlick's approximation of the reflectance

$$r \approx r_0 + (1 - r_0)(1 - \cos\theta_1)^5 \tag{3.7}$$

uses just this special case and the angle of incidence. Figure 3.5b shows the reflectance and transmittance that is the result of using this approximation and the same refractive indices as Figure 3.5a. Especially for steep angles of incidence below 50°, this approximation is quite close to the actual value.

3.2.3 Absorption

Absorption is another influence that a transparent medium like water can exert on a ray of light. Energy is lost due to direct collision of the incoming photons with the atoms of the medium. Additionally, some part of the light is also scattered in a different direction. Taken as a whole, the process of absorption means that as light travels through a medium like water, more and more of its intensity is lost.

The absorbing property of a medium due to these processes is described by the attenuation coefficient α, which is commonly expressed in units of cm^{-1}. This coefficient combines the direct absorption as well as the scattering effect. The exact value is wavelength dependent.

Given an initial intensity i_0 and an attenuation coefficient α, the remaining intensity for a distance x into the absorbing medium is described by

$$i(x) = i_0\, e^{-\alpha x}. \tag{3.8}$$

Figure 3.6 shows the effect of absorption on the intensity for a fixed attenuation coefficient. The intensity drops to a small fraction of its initial value for increasing distances. The absorption determines how far a ray of light can penetrate into a medium. The wavelength dependence of the attenuation coefficient can for example be seen when diving underwater: At a depth of a few meters, the light takes on a distinctive blue and green hue. This is due to the fact that the wavelengths from this part of the spectrum can reach farther into the water when compared to longer wavelengths. Red and near infrared light is more strongly absorbed.

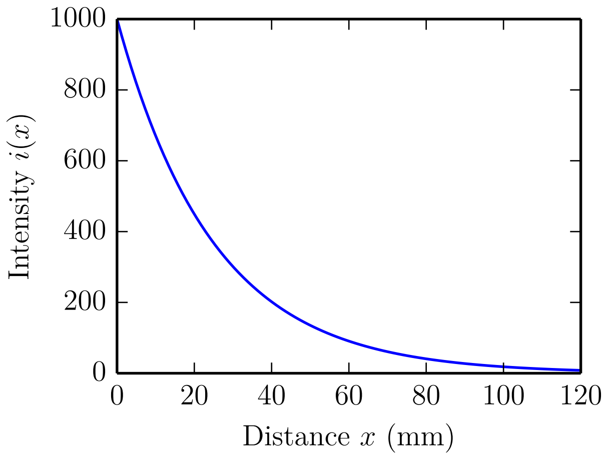


Figure 3.6: Absorption for $\alpha = 0.04\,\mathrm{cm}^{-1}$ and $i_0 = 1000$.

Instead of the attenuation coefficient α, sometimes the extinction coefficient $\kappa(\lambda)$ is used. The two values can be converted by using

$$\alpha = \frac{2\kappa(\lambda)\omega}{c} = \frac{4\pi\kappa(\lambda)}{\lambda}. \tag{3.9}$$

The extinction coefficient is the imaginary part of the complex refractive index. The real part of the complex refractive index is the index of refraction. This representation is sometimes used to describe refraction in terms of a complex wave.

For the camera signal, the absorption has an immediate negative effect: By lowering the intensity of the signal, the camera noise becomes more pronounced as the signal has to travel through a thicker layer of water. The precise effect of this is of course camera specific, and also dependent on the wavelength of the emission units.

3.3 Refraction-Based Distance Measurement Model

We have seen the different influences a transparent medium like water can have on the camera signal. Of these influences, refraction seems the most likely candidate to use as a basis of our distance measurement model. The goal of this model is to mimic the effect of water on the measured distances as accurately as possible. At the same time, the model has to be general enough to be used with any time-of-flight based range sensing device instead of being tied to a specific camera. The goal of our distance measurement model is the ability to convert water heights to a synthetic distance image. Here, we will describe the model that we used for this purpose. In a second step, we will later explain how such a model can be used for the reconstruction of water surfaces from distance data, which represents the opposite direction.

Our model assumes that the distance at each pixel is calculated from a single ray of light. This ray is made up of two parts: The distance d_a in air and the distance d_w in water. The first part of the ray starts at the camera center $\boldsymbol{c}$ and travels along viewing direction $\hat{\boldsymbol{v}}$. At the intersection $\boldsymbol{s}$ between the ray and the water surface, the ray is refracted in direction $\hat{\boldsymbol{r}}$. This underwater part of the ray is stopped once it hits the floor at floor point $\boldsymbol{f}$. An overview of this model can be seen in Figure 3.7.

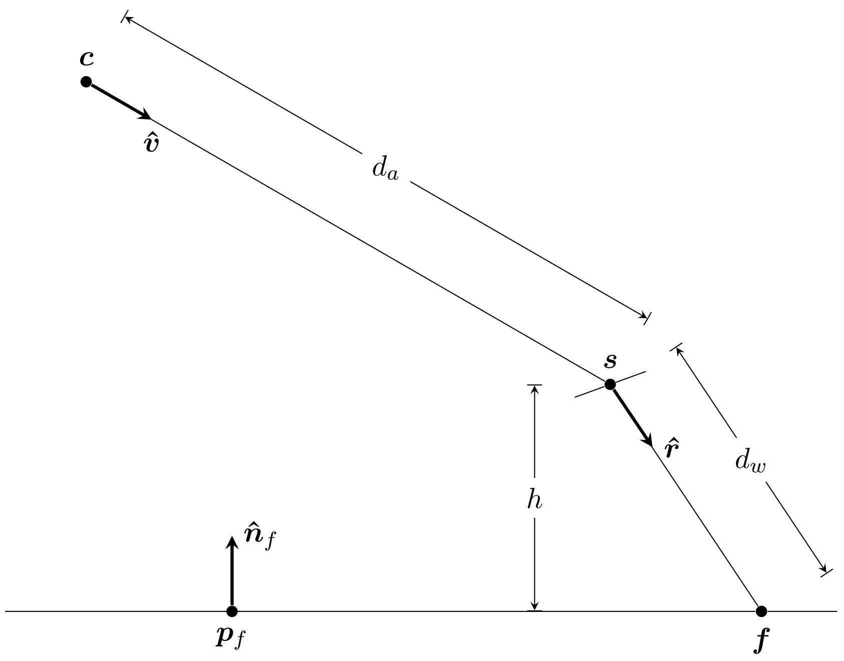


Figure 3.7: Model overview.

The floor can be described as a plane in normal form, defined by the floor normal $\hat{\boldsymbol{n}}_f$ and the support vector $\boldsymbol{p}_f$, a point on the floor plane. Any point $\boldsymbol{p}$ which fulfills the following equation is inside this floor plane:

$$(\boldsymbol{p} - \boldsymbol{p}_f) \cdot \hat{\boldsymbol{n}}_f = 0. \tag{3.10}$$

The viewing direction can be obtained from the pixel coordinates and the intrinsic camera parameters after the camera has been calibrated, like we saw in Chapter 2. To get the camera center, floor normal and floor point we can use a depth image of the experiment setup without any water in the container. To calculate the total distances, we will need the position of the floor and surface intersection points.

For a fixed height h, the surface intersection point is the intersection of the light ray with a plane parallel to the floor plane. This surface plane shares the normal of the floor plane. For its support vector

$$\boldsymbol{p}_s = \boldsymbol{p}_f + h\hat{\boldsymbol{n}}_f \tag{3.11}$$

we can use the support vector of the floor plane offset by the height in the direction of the floor normal. Thus, the surface plane contains any point $\boldsymbol{p}$ for which

$$(\boldsymbol{p} - \boldsymbol{p}_s) \cdot \hat{\boldsymbol{n}}_f = 0 \tag{3.12}$$

holds true. It is important to note that this plane is only used to compute the intersection between ray and water surface. The actual water surface normal is computed in a separate step.

The surface intersection point can also be expressed as

$$\boldsymbol{s} = \boldsymbol{c} + d_a \hat{\boldsymbol{v}}, \tag{3.13}$$

which is the point at distance d_a along the ray in air. If we insert this point as $\boldsymbol{p}$ in Equation (3.12) and solve for d_a, we get

$$d_a = \frac{(\boldsymbol{p}_s - \boldsymbol{c}) \cdot \hat{\boldsymbol{n}}_f}{\hat{\boldsymbol{n}}_f \cdot \hat{\boldsymbol{v}}}. \tag{3.14}$$

Together with Equation (3.13) we now have both the distance in air and the surface intersection point for a fixed height. All that remains is calculating the refraction direction, floor intersection and distance in water.

To compute the refraction direction, we need the viewing direction, the surface normal, as well as the refractive indices of both water and air. The viewing direction is already known. For the surface normal, we can use finite differences and the surface position of the neighboring pixels to approximate the gradient of the surface in horizontal and vertical direction. The surface normal $\hat{\boldsymbol{n}}_s$ can then be obtained by normalizing the cross product of these two vectors. The normal calculation marks the only time our model requires information about neighboring pixels. At the same time, calculating the normal by finite differences is only one possible way of many to obtain this information. Other methods that rely on a larger neighborhood could also be used as a replacement.

The index of refraction for air can be approximated as $\eta_a \approx 1.0$. For water, the index of refraction η_w is camera specific, as different time-of-flight cameras will emit light at different wavelengths. The ratio $\eta = \eta_a/\eta_w$ determines the actual direction of the refraction. Recall that Snell's law describes how the angles involved in refraction and the refractive indices relate to each other. [Fol96] shows how to reformulate Equation (3.1) in order to calculate the refraction direction. Adapted for our notation, this formula can be rewritten as

$$\hat{\boldsymbol{r}} = \eta \hat{\boldsymbol{v}} - \left(\eta (\hat{\boldsymbol{n}}_s \cdot \hat{\boldsymbol{v}}) + \sqrt{1 - \eta^2 (1 - (\hat{\boldsymbol{n}}_s \cdot \hat{\boldsymbol{v}})^2)} \right) \hat{\boldsymbol{n}}_s. \tag{3.15}$$

Apart from changing the variable names, the only other change is the direction of vector $\hat{\boldsymbol{v}}$: In [Fol96] a vector pointing in the opposite direction $-\hat{\boldsymbol{v}}$ is used.

Using Equation (3.15), we can compute the refraction direction for a single pixel from the viewing direction, so long as we also have the surface normal and the ratio of the two refractive indices. This underwater part of the ray intersects with the floor in the floor intersection point

$$\boldsymbol{f} = \boldsymbol{s} + d_w \hat{\boldsymbol{r}} \tag{3.16}$$

at a distance d_w from the surface intersection point. Similar to how we arrived at Equation (3.14), we can insert this point into Equation (3.10) describing the floor plane and solve for d_w:

$$d_w = \frac{(\boldsymbol{p}_f - \boldsymbol{s}) \cdot \hat{\boldsymbol{n}}_f}{\hat{\boldsymbol{n}}_f \cdot \hat{\boldsymbol{r}}}. \tag{3.17}$$

At this point we have all the pieces we need. We calculate a surface intersection point for a fixed height and a viewing direction. This also yields the length of the part of the ray which travels through air. Using the surface intersection points of neighboring pixels, we approximate a surface normal, which is used to compute the refraction direction. Together with the floor intersection point, we now have the lengths of both the underwater and in air parts of the ray. As a last step, we assume that the lower speed of light in water alters the measured underwater distance. To incorporate this aspect into our model, we scale the underwater distance by the index of refraction. Thus, the total distance

$$d = d_a + \eta_w d_w \tag{3.18}$$

can be formed. It is mostly dependent on static information such as the viewing direction and the relative position of camera and floor. The only variable we can adjust is the height at each pixel. Notice also how this height influences the distance for the neighboring pixels: Since the normal is approximated from the horizontal and vertical direct neighbors, changes in height at these pixels also change the direction of the normal. This in turn results in a different refraction direction.

The model as described by Equation 3.18 represents a linear relationship between an increasing underwater distance and the overall total distance. As we will see in Chapter 4, the actual distance measurement for increasing water height show a nonlinear response. For this reason, we need to refine our model to include a correction function $f(d_w)$, resulting in a total distance of

$$d = d_a + f(d_w). \tag{3.19}$$

This function takes the underwater distance calculated by our model and returns a corrected underwater distance. The precise nature of this correction is unspecified at this point. In Chapter 6, we will see a possible example for the correction. We refer to this refined model as the nonlinear model, in contrast to the simpler linear model.

Our model thus far only operates at the level of each individual pixel. This means that for each pixel we compute a total distance. As we already saw, this distance depends not only on the height at this precise pixel, but also on the surface points of its neighbors. We refer to the complete height field as H, which is a two dimensional image comprised of individual pixel height $h_{i,j}$ at each pixel (i, j). Similarly, the distance image is referred to as D and consists of pixel distances $d_{i,j}$. The water surface $\boldsymbol{S}$ can also be thought of as a two dimensional structure. In contrast to the other images, the components of the water surface are not scalar values, but vectors $\boldsymbol{s}_{i,j}$ describing the position of the surface in space for a pixel (i, j). Each of these images can also be thought of as an $a \times b$ matrix. The matrices share the same resolution, which is determined by the camera itself.

Figure 3.8 shows a summary of the individual stages of our refraction-based model. To the left of each stage, the input variables upon which the stage depends are shown. The output to the right of each stage shows the result of this particular stage. Some of these results are reused as input to later stages. Note how the output of the surface intersection stage is just a single point of the surface. Calculating the surface normal requires information about the surface for the neighborhood of the pixel, which is why this stage depends on $\boldsymbol{S}$ instead of just $\boldsymbol{s}$.

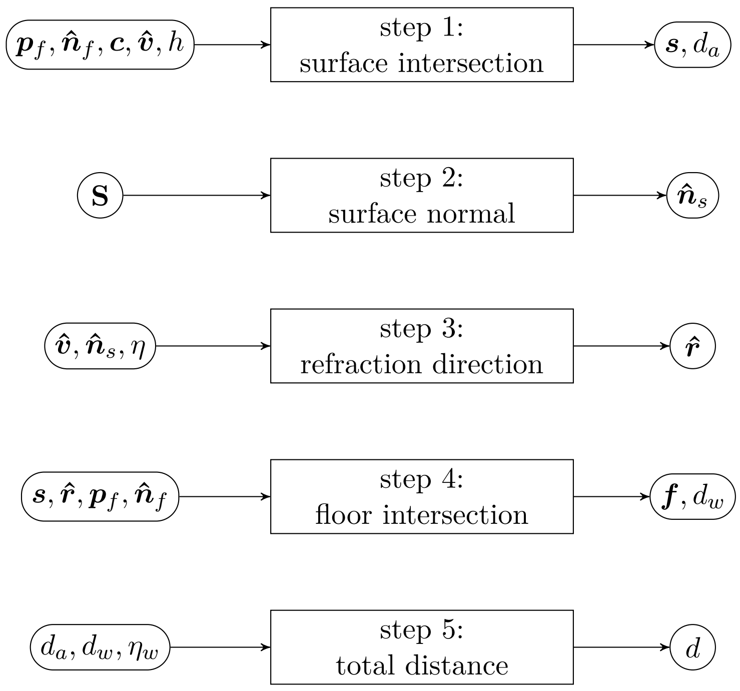

$\boldsymbol{p}_f, \hat{\boldsymbol{n}}_f, \boldsymbol{c}, \hat{\boldsymbol{v}}, h$
step 1: surface intersection
$\boldsymbol{s}, d_a$
$\mathbf{S}$
step 2: surface normal
$\hat{\boldsymbol{n}}_s$
$\hat{\boldsymbol{v}}, \hat{\boldsymbol{n}}_s, \eta$
step 3: refraction direction
$\hat{\boldsymbol{r}}$
$\boldsymbol{s}, \hat{\boldsymbol{r}}, \boldsymbol{p}_f, \hat{\boldsymbol{n}}_f$
step 4: floor intersection
$\boldsymbol{f}, d_w$
d_a, d_w, η_w
step 5: total distance
d

Figure 3.8: Refraction-based distance measurement model stages.

3.4 Surface Reconstruction

Our model provides us with a way to simulate the influence of water height changes on the distances. If we use the surface height field as an input, our model calculates a distance image that reflects these changes. Our goal now is to utilize this model to reconstruct a water surface. This means applying the model to a problem in the opposite direction: When presented with a distance image of a water surface, we must find out which water heights were responsible for the formation of this image. A closed form solution of this inverse problem is not easy to find, since we do not know the actual length of the individual distances that make up the total distance. There are too many unknown variables. Instead, we opted to use an optimization algorithm for reconstructing a height field from a distance image.

From the perspective of optimization, our refraction-based distance measurement model can be viewed as a set of nonlinear equations, one for each pixel. The output of these equations for a height image H is a distance image D. Due to calculating the normal from the surface neighborhood, the equations are not independent for each pixel. However, most information like the relative position of camera and floor or the viewing directions remains static. The only actual variable that we can tweak is the height for each individual pixel. The goal of an optimization algorithm is then to find the corresponding height field for a distance image.

The distance image is used as an optimization target and provides a way to evaluate the optimization error. The optimization algorithm tries to find a height field that minimizes this error. Our model provides us with a synthetic distance image, which we can compare to the optimization target. If our model accurately reflects the actual influence of the water on the distances, the result of the optimization should be the correct set of water heights.

For optimization, we were faced with a choice between three possible approaches. We could try to optimize at the level of individual pixels, a local pixel neighborhood or the entire image. Each approach has its advantages and drawbacks.

For a single pixel, the normal can no longer be calculated from its neighbors, and would have to be an additional optimization variable instead. The advantage of this approach is that it is easier to parallelize. Also, the low number of variables allow some optimization algorithms to be used that are not suitable for a more global approach. The biggest disadvantage of the single pixel method is that there is no guarantee that the resulting heights form a consistent water surface. To overcome this, some additional steps would have to be taken at the end of the optimization to ensure this requirement.

Optimizing a local pixel neighborhood can be seen as a hybrid between the local and global approaches. It is now possible to use the heights of other pixels to approximate the normal. This means that within a single region, the water surface can be kept consistent using additional optimization constraints. If the neighborhood is sufficiently small, the number of available optimization routines is also not restricted. Consistency of the entire water surface is still compromised at the borders of the region. Here, additional work is required to make sure no sharp edges remain. Overlapping the regions could potentially alleviate this somewhat, but this comes at the cost of additional computations. Individual regions could be optimized in parallel, since there is no direct dependency between them, similar to how individual pixels could be optimized in the local approach.

If the entire height field is optimized at the same time in a global approach, it becomes easier to ensure that the water surface has no sharp edges and is in fact smooth. During optimization, pixel heights that deviate to far from their neighbors can be discouraged. The resulting water surface then requires no additional steps to enforce this requirement. A downside is the large number of variables: Each pixel represents a height for the optimization algorithm to adjust. The resolution of the camera thus determines the number of variables available to the optimization algorithm. For many optimization routines, there is a practical limit to how many variables they can deal with. This approach restricts the number of available optimization routines when compared to the local and the hybrid approach. The global optimization could also be a potential performance bottleneck. Parallelization of the optimization seems much harder for this approach, since only one optimization is actually performed.

For this thesis, we chose to proceed by using the global optimization approach. The advantage of being able to ensure a smooth water surface across the entire image outweighs the disadvantage of the limited number of optimization routines. This is of course only true as long as at least one optimization algorithm can be found that can cope with the large number of variables. Since the resolution of time-of-flight camera sensors is likely to increase in the future, there may come a point when our global approach is no longer practical. The drawbacks of the other approaches by no means make them completely inapplicable. Their biggest advantage can be seen in the fact that they consist of multiple independent optimizations. For increasing image resolutions, this could provide an import performance benefit over our chosen approach. Nevertheless, we elected to use the global optimization approach. This approach puts more emphasis on the correctness of the result and ease of ensuring the smoothness of the water surface, at a potential performance cost. As mentioned before, this performance cost could prove to be too large for future devices.

Before, we have considered the distance image D and the height field H as two-dimensional matrices. From the perspective of an optimization algorithm, the two-dimensional structure is not actually relevant. In the following explanation of the surface reconstruction, we will use flattened, one-dimensional versions of these structures instead. Flattening the images is revertible, but will make the optimization easier to explain. All of the images have the same resolution $a \times b$, for a total number of $p = ab$ pixels. In place of the distance image a distance vector $\boldsymbol{d}$ is used. This vector contains p entries, one entry d_i for each pixel i. In the same way, we replace the height field with a height vector $\boldsymbol{h}$, containing a height h_i for each pixel i.

Instead of the imperative description of our refraction-based model, it also possible to see each entry of the distance vector as a function $d_i(\boldsymbol{c}, \hat{\boldsymbol{v}}_i, \eta, \hat{\boldsymbol{n}}_f, \boldsymbol{p}_f, \boldsymbol{h})$ which maps its input parameters to a single output distance for a pixel i. Since the camera center, viewing directions, index of refraction, floor normal and floor support vector remain fixed during optimization, it is possible to simplify this function to $d_i(\boldsymbol{h})$. If the process is repeated for every pixel, the model represents a vector-valued function $\boldsymbol{d}(\boldsymbol{h})$ which maps the input height vector to the output distance vector. While the setup parameters are important for the output of this function, for the purpose of optimization we view them as constants.

The measured distance image M, flattened to a vector $\boldsymbol{m}$ is used as the optimization target. The vector also contains p entries because the measured image determines the resolution of the other images.

There are different ways we can use an optimization algorithm to find a solution to our problem. All of them revolve around defining an error function that describes how close the solution is to an ideal solution. For our problem of matching synthetic distance data to measured distance data, we can simply take the component-wise difference $\boldsymbol{m} - \boldsymbol{d}(\boldsymbol{h})$ as a starting point for our error function: For the correct height vector, the difference should approach zero at each pixel.

A scalar optimization algorithm uses a single scalar value to evaluate the optimization error. One way to arrive at a scalar value from our per pixel difference image is by means of the sum of squared errors. The optimization algorithm then tries to find a solution by minimizing this scalar error value. We can express this as

$$\arg\min_{\boldsymbol{h}} \sum_{i=1}^{p} \left[m_i - d_i(\boldsymbol{h})\right]^2. \tag{3.20}$$

In this case, since the minimum error value is zero, the problem could also be stated in a different way: Instead of minimizing the error function, we search for a set of heights $\boldsymbol{h}$ so that

$$\sum_{i=1}^{p} \left[m_i - d_i(\boldsymbol{h})\right]^2 = 0. \tag{3.21}$$

This is referred to as finding the root, or zero of the error function.

One problem with using a scalar value to evaluate the optimization error is that the influence of a single optimization variable on this error is not clear. This problem becomes more pronounced as the number of variables increases. Instead of relying on a scalar error function, it is also possible to use a vector-valued error function. Our component-wise difference can then be used directly as the error function

$$\boldsymbol{e}(\boldsymbol{h}) = \boldsymbol{m} - \boldsymbol{d}(\boldsymbol{h}), \tag{3.22}$$

since each entry i of the component-wise difference contains an error value $e_i(\boldsymbol{h})$. Finding the root of a vector-valued function in our case means that we are looking for a height vector $\boldsymbol{h}$ so that this error becomes zero for every pixel.

To find the root of a possibly vector-valued, nonlinear function, Newton's method is commonly used. This method evaluates the function for a number of iterations. Starting from an initial guess $\boldsymbol{h}_0$, the value of the next iteration after n iterations can be calculated as

$$\boldsymbol{h}_{n+1} = \boldsymbol{h}_n - J^{-1}\boldsymbol{e}(\boldsymbol{h}_n). \tag{3.23}$$

The iteration is typically stopped if the error becomes sufficiently small, the relative change between each iteration becomes smaller than a threshold, or a predetermined number of iterations has been reached.

The Newton iteration makes use of the Jacobian matrix

$$j_{i,j} = \frac{\partial e_i(\boldsymbol{h})}{\partial h_j} \tag{3.24}$$

of the error function, which contains a partial derivative of the error function for each pixel i with respect to the height variable of pixel j. This matrix is different for each step. In total, the Jacobian matrix consists of $p \times p$ entries. This means that the number of entries increases quadratically with the number of pixels.

Newton's method requires evaluating and inverting the Jacobian matrix for each iteration step. As the size of the Jacobian matrix increases, this quickly becomes impractical. For example, simply storing the Jacobian matrix for a 200×200 pixel image would require 1 600 000 000 entries. If each entry is stored using a 32 bit floating point number, the entire matrix would consume approximately 6.4 GB of memory. Evaluating and inverting a matrix of this size at each iteration step is obviously not feasible. As even todays cameras already surpass the resolution in our example, we need to find a way around this problem if we wish to continue with our global optimization approach.

In order to find the root of such a large scale, nonlinear system of equations, inexact Newton methods can be used. These optimization routines use an approximation of the Jacobian matrix instead of evaluating it directly. The result of this is that these methods are much more efficient at dealing with a large number of optimization variables. The inexact Newton methods proved sufficient for optimization at the image resolution of the camera we used for the majority of our work presented in this thesis. For this reason, we chose to continue using our global optimization approach in conjunction with optimization routines based on inexact Newton methods.

An important observation can be made about our distance error function $\boldsymbol{e}(\boldsymbol{h})$. Since the normal is calculated from a local neighborhood, the influence of a single pixel height on the entire distance image is limited. As a result of this, a majority of the entries of the Jacobian matrix are actually zero. This is because the partial derivative is formed with respect to a variable that is not actually present in the distance function of a pixel. A matrix that contains mostly the same entries opens up the possibility to use a sparse matrix for its representation. The advantage of a sparse matrix is that only entries which are different from the majority need to be stored, which dramatically reduces the size and memory requirements of our Jacobian matrix. As the inexact Newton methods worked adequately for our purposes, we never fully investigated the usefulness of using a sparse matrix for the Jacobian matrix. Evaluating the

Jacobian matrix instead of using an estimate has the potential to speed up the convergence of the optimization algorithm. It could be one avenue to increase performance, and might also make more optimization routines viable again for global optimization.

Real water surfaces have a property that we can use in our reconstruction method: They are smooth under certain conditions. As long as there are no breaking waves or extreme turbulences, the water heights around a pixel cannot vary greatly from the height at the pixel itself. Sharp edges and spikes are not possible. We use this information to enforce a smoothness constraint on our height image during optimization. The Laplace operator Δ can be used for this purpose. When applied to the height field H, it provides the second order derivative in horizontal and vertical direction. This derivative is closer to zero for a smooth height field, whereas sharp edges result in nonzero derivatives. For discrete images, the Laplace operator is commonly approximated by convolution of the image with a small kernel. Even though the Laplace operator is applied to a two dimensional height image, we can flatten the result for use with our error function. The result is a smoothness constraint $\boldsymbol{c_s}$.

An alternative to the Laplace operator is the Laplacian of the Gaussian (LoG), which combines a Gaussian filter with the Laplace operator. When discretized for a convolution kernel, the main advantage of the LoG is the larger size of the kernel. This is helpful for smoothing larger parts of the image. The LoG is also known as the Mexican Hat filter due to its distinctive shape.

By adjusting our error function with a scaled version of this smoothness constraint

$$\boldsymbol{e}(\boldsymbol{h}) = \boldsymbol{m} - \boldsymbol{d}(\boldsymbol{h}) + s\boldsymbol{c}_s, \tag{3.25}$$

we can discourage large variations in height for neighboring pixels. The scaling factor s allows us to influence the smoothness of the resulting surface heights.

Chapter 4

Experiment Setup

In Chapter 3, we developed a theoretical method for reconstructing a water surface from a time-of-flight camera image. To apply this method to a real data set, we need to be able to record water surfaces in a controlled environment. While the model we use for the reconstruction is camera-independent, we also have to deal with some characteristics of real cameras not reflected in our model. We first discuss these potential error sources, and how they relate to our goal. We also take a look at the necessary calibration steps for the camera. We then describe our experiment setup in more detail, keeping in mind some of these error sources. The chapter concludes with an overview of the different experiments that were conducted using this setup. We will also see the need for a refinement of our model based on one of these experiments. In order to reconstruct surfaces from our experiment measurements, we also need the implementation described in Chapter 5. The surfaces reconstructed from these measurements are discussed in Chapter 6.

4.1 Camera Data

Before we describe our experiment setup in more detail, we take a closer look at some camera specific error sources. These error sources are common to many time-of-flight cameras. We discuss how the error sources could potentially influence the type of scene we are interested in. Some of these influences have to be considered for the construction of our experiment. Others can be corrected in a camera specific way. After the discussion of the error sources, we describe the necessary calibration steps that were conducted for the PMD DigiCam.

4.1.1 Potential Error Sources

[Lef+13] provide an overview of the error sources that can falsify the distance information captured by time-of-flight cameras. According to the authors, these error sources can be divided into the following categories:

- Systematic error
- Intensity-related error
- Error from motion artifacts
- Sensor noise
- Error from multiple returns

Systematic errors are a result of a mismatch between the theoretical model and the actual physical camera. In case of the PMD DigiCam, we saw in Chapter 2 that the modulation function is assumed to be a perfect sinusoidal wave. The physical illumination unit produces a wave that is slightly different from the idealized model wave due to higher order dynamics. As a consequence of this mismatch between model and reality, the measured depth oscillates around the correct depth when compared over the unambiguity depth range. This oscillation is the reason the error is also referred to as the 'wiggling error'. Additionally, the model assumes that the light source and camera sensor are in the same physical spot. This is of course not possible for actual cameras. As we saw in Chapter 2, the illumination units are positioned to either side of the camera lens for the PMD DigiCam. To correct this systematic error, a depth calibration becomes necessary.

Intensity-related errors manifest themselves in the distance to objects of varying reflectivity. Darker portions of a scene appear closer than more reflective, lighter parts, even when in reality they are positioned at a similar distance. [Lef+13] speculate that this error might be a result of nonlinearity in the signal response of the sensor, or of scattering effects between the sensor and lens. For our purposes, the scene we capture is more or less uniformly reflective, with no regions of high contrast between light and dark. Nevertheless, as we discussed in Chapter 3, the absorption of light by water results in a reduced intensity. This becomes more noticeable as the path through water lengthens, in other words, as the water height is increased. In theory the intensity should have no bearing on the distance. This error source could mean that in practice our distance information is influenced by the change in intensity depending on the path length through water. This change is in addition to the influence of refraction on the direction and the index of refraction on the speed of light.

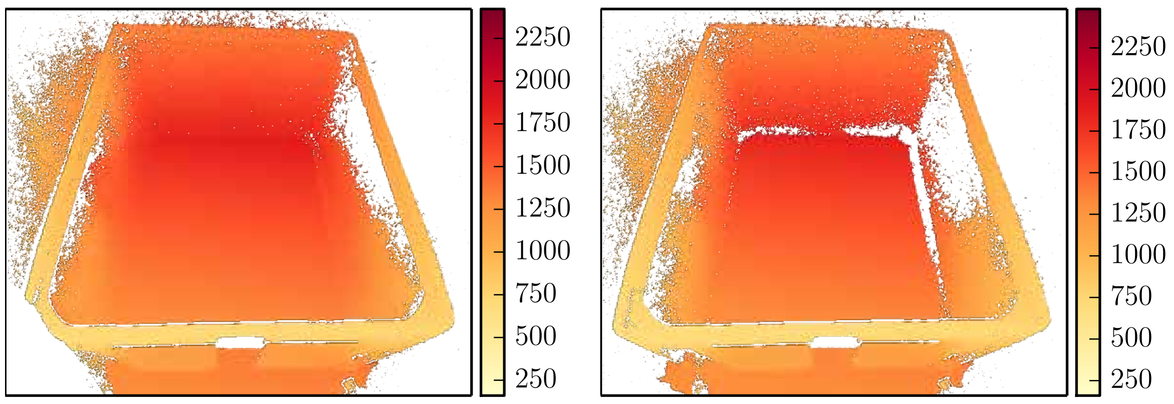
Because the correlation function is sampled multiple times sequentially, dynamic scenes can result in motion artifacts. While there are methods for correcting these artifacts, our implementation does not utilize these methods yet. For moving water surfaces, incorporating motion compensation could be a way to improve the reconstruction results.

Time-of-flight cameras are also affected by noise, similar to regular cameras. This noise is decreased by choosing a longer integration time, and increases as the signal strength is reduced. Longer integration times have the potential to introduce more motion artifacts. For static scenes, averaging over multiple frames can help in reducing the influence of the noise on the distance. This option is of course not available for dynamic scenes.

Errors stemming from multiple returns are the result of a mixture of reflections. Ideally, a scene pixel represents the direct reflection of the modulation signal back towards the camera. In a typical scene, there are two reasons for why this assumption might be violated. First, the comparatively low resolution of the sensor of a time-of-flight camera means that each pixel covers a larger solid angle. If a pixel is capturing the edge between two objects, the resulting distance is a mixture of the signals received from both objects. This mixed distance can take on any possible distance within the range of the camera, which is why these pixels are also called flying pixels. In our setup, we have avoided creating sharp edges between objects, as we only take images of a flat floor situated at the bottom of a container.

The second type of error based on multiple returns occurs when the light is not reflected directly, but instead is scattered within the scene multiple times. The effect is also called the multipath effect. The result is again that distance information for a pixel is a mixture of signals that no longer conform to the idealized model used to calculate the distance. These effects can result in sharp corners of a real scene taking on a more rounded appearance in the distances captured by the camera. Compensation methods for this type of error source often rely on capturing the same scene with different modulation frequencies, which is only possible for static scenes.

Since we also want to capture dynamic scenes, correcting this effect becomes a difficult task. As a result, we tried to design our experiment setup with this effect in mind. We chose to use a camera lens with a narrow field of view to avoid capturing the walls of the container. The water itself represents another potential source of multipath errors, since it can reflect part of the illumination signal. The description of the experiment setup details the other steps we took in order to avoid the multipath effect to the best of our ability.



(a) Empty container. (b) Container filled with water.

Figure 4.1: Distances in mm captured by the Kinect 2 prototype.

In order to test for the presence of multiple returns as a result of filling our setup with water, we used a different camera. The Kinect 2 prototype camera is able to detect the presence of multipath in a scene, which is not possible with the PMD DigiCam. Figure 4.1 shows two distance images captured with this prototype camera. Invalid pixels have been marked as white. In Figure 4.1a, the container is still empty, and no invalid pixels are detected where the floor meets the container walls. In Figure 4.1b, water has been added for a total water height of 5 cm. Comparing the water filled container to the empty container, we notice an increase in invalid pixels towards the edges formed by the walls and floor. This could be a result of the multipath effect. However the floor pixels away from the walls and towards the center of the floor show no sign of additional invalid pixels. For the PMD DigiCam, we positioned the camera in such a way that only this part of the floor was visible. In an image taken of a very low water height of 5 mm, the Kinect prototype shows the same absence of errors cause by multiple returns. Note that while this result is encouraging, it is not a guarantee that there is no multipath effect present with a different camera like the PMD DigiCam.

Time-of-flight camera software is able to flag some pixels as invalid if the distance at these pixels is deemed unreliable. For the PMD DigiCam, the reasons for flagging a pixel include oversaturation of the sensor, a signal that was too low, or an inconsistent signal as a result of motion between two exposures. Regardless of the source of the invalid pixels, our

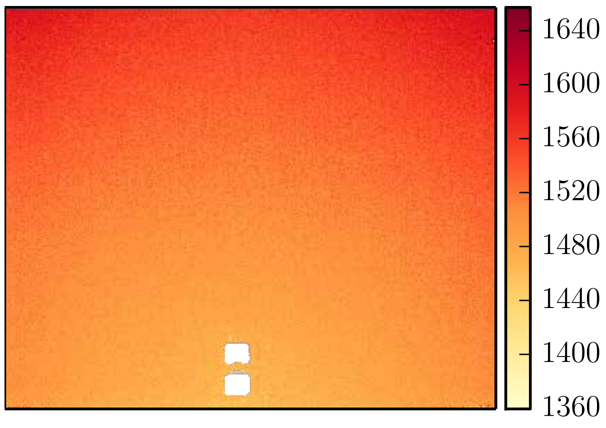


Figure 4.2: Direct reflection for the PMD DigiCam, distances in mm.

implementation must be able to reconstruct water surfaces even when some of the pixels contain no valid information.

The water surface itself is also a source of oversaturated and therefore invalid pixels. Figure 4.2 shows an example: The camera has been angled towards the water surface at a very steep angle. The two rectangular illumination units are visible towards the lower edge of the image. They have been flagged as oversaturated and contain no useful distance information. This is the result of direct reflection from the water surface, as the illumination units are not visible without water. For a straight downward view, the number of invalid pixels due to this effect is greater for both still water and moving waves. We captured most of our material with the camera tilted at a slightly greater angle to avoid this effect. For moving water surfaces, some of the invalid pixels can be explained by the water surface aligning with the optical axis of the camera again.

4.1.2 Calibration

In Chapter 2 we already saw how we can transform two-dimensional image coordinates to three-dimensional camera coordinates using a pinhole camera model. Real cameras are of course not actually equivalent to pinhole cameras. Their lenses introduce distortion effects which need to be corrected. As we will see in Chapter 5, we use the OpenCV [Its15] software library for this correction step. In OpenCV, the image distortion is estimated at the same time as the other intrinsic parameters that form the camera matrix of the idealized pinhole camera. The calibration algorithm used by OpenCV is based on [Zha00]. It allows for correcting radial and tangential distortion. The effect of radial distortion can be most readily observed in straight lines: In a distorted image, these lines will appear curved. Radial distortion is usually more visible towards the edges of an image. Tangential distortion is a consequence of imperfect alignment of the image sensor on the camera lens.

The radial and tangential distortion is modeled as a polynomial function which maps undistorted image coordinates (x_u, y_u) as used by the idealized pinhole camera to distorted image coordinates (x_d, y_d). The latter coordinates are those of an image taken by a real camera which exhibits distortion. If

$$r = \sqrt{x_u^2 + y_u^2} \tag{4.1}$$

is the distance of the undistorted image coordinates to the principal point of the pinhole camera, then the distorted image coordinates are given by

$$x_d = x_u(1 + k_1 r^2 + k_2 r^4) + 2p_1 x_u y_u + p_2(r^2 + 2x_u^2) \tag{4.2}$$

$$y_d = y_u(1 + k_1 r^2 + k_2 r^4) + 2p_2 x_u y_u + p_1(r^2 + 2y_u^2). \tag{4.3}$$

The radial distortion is characterized by the radial distortion coefficients k_1 and k_2. In a similar way, the tangential distortion coefficients p_1 and p_2 describe the tangential distortion. Note that the OpenCV implementation supports up to four additional radial distortion coefficients. These higher order coefficients are not used by us to retain compatibility with the software we used for calibrating the depth correction.

The distortion model describes how to convert from undistorted image coordinates of an idealized pinhole camera model to distorted image coordinates of a real camera. For the opposite direction, it is necessary to create a lookup table that maps distorted image coordinates to undistorted image coordinates using Equations (4.2) and (4.3). The pixels of the distorted image will not match the coordinates contained in this lookup table exactly, but lie somewhere in between. Because of this, interpolation is needed to produce an undistorted image. OpenCV offers a function that combines the lookup table generation with a bilinear interpolation to map a distorted image to its undistorted counterpart. Figure 4.3 shows the effect of applying this function to an intensity image of an example scene. Note how curved lines due to distortion are corrected in the undistorted image. Because of this, undistortion is sometimes also referred to as rectification. The effect is subtle, but more visible towards the edges. We marked some pixels with a yellow hue to highlight a line were this can be seen.

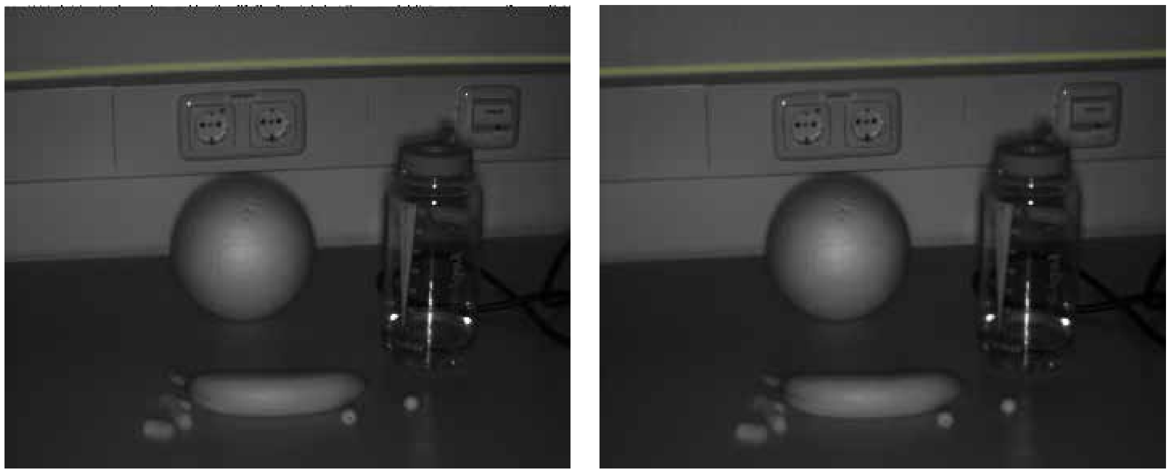

(a) Scene before undistortion. (b) Scene after undistortion.

Figure 4.3: Distorted and undistorted image of the same scene.

The intrinsic camera parameters depend only on the arrangement of camera lens and image sensor, as well as their respective properties. Modulation frequency and integration time have no influence on this calibration. By contrast, changing the focus of the lens influences both the intrinsic calibration and the depth correction. For this reason, we selected a fixed focus that fit our experiment setup well, and calibrated the camera for this focus point.

For the intrinsic calibration, we used a printed checkerboard pattern fixed on a flat piece of wood. We took a series of about 100 images in order to calibrate the camera matrix and distortion coefficients. For these images, it was important to cover the whole image with the checkerboard pattern. In addition to this, the checkerboard was angled away from the camera in several of the images, to better estimate perspective distortion. The checkerboard was also captured at varying distances, covering roughly the same space as our experiment setup. An example calibration image can be seen in Figure 4.4. For this calibration, we used only intensity images and no distance information. The resulting intrinsic parameters are summarized in Table 4.1.

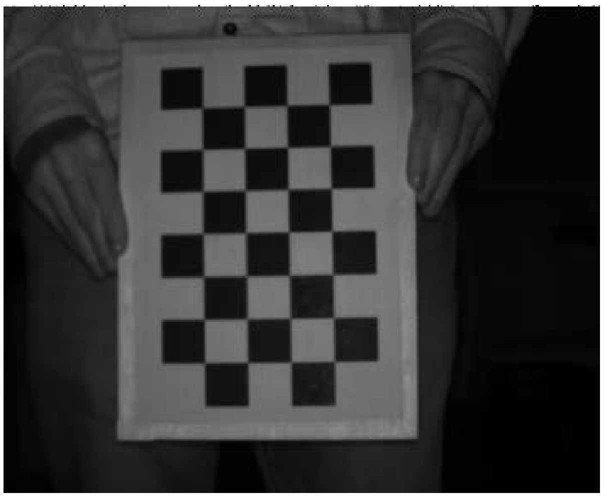


Figure 4.4: Example calibration image.

Focal length		Principal point		Distortion coefficients			
f_x	f_y	c_x	c_y	k_1	k_2	p_1	p_2
730.4809	729.6344	148.8965	117.64	-0.3845	-0.197	0.0081	0.0078

Table 4.1: Intrinsic calibration results.

Having calibrated our intrinsic parameters, we use the calibration result as a starting point to obtain a depth calibration. In contrast to the previous calibration, this calibration very much depends upon the selected camera parameters for integration time and modulation frequency.

The calibration tool [Sch11] we used for the depth correction is targeted at multi-camera setups, but can also be used for our single camera. More details about this tool are covered in [Lin+10]. We use fixed intrinsic parameters to speed up the calibration. The same checkerboard is used, but this time it is placed in the center of the image and moved away in 2 cm intervals. We cover a range of about 1 m with this checkerboard. For a complete calibration, the whole unambiguity range would have to be covered. For our purposes, covering the dynamic range of our measurements is sufficient.

The tool estimates the distance from the checkerboard pattern and provides a depth correction polynomial. If d_u is an uncorrected distance as captured by our camera, and (x, y) are the pixel coordinates of its pixel, then the corrected distances d_c is given by

$$d_c = d_0 + (1.0 + d_1)d_u + d_2x + d_3y + d_4d_u^2 + d_5d_u^3, \tag{4.4}$$

where $d_0, \ldots, d_5$ are the coefficients that are estimated by the tool. The tool also provides an upper and lower bound of the depth range that was covered by a single calibration.

We repeated this process for several different integration times at a modulation frequency of 30 MHz. Table 4.2 shows the calibration result for 1000 µs. The depth correction polynomial that is defined by this calibration result is shown in Figure 4.5 for a single pixel.

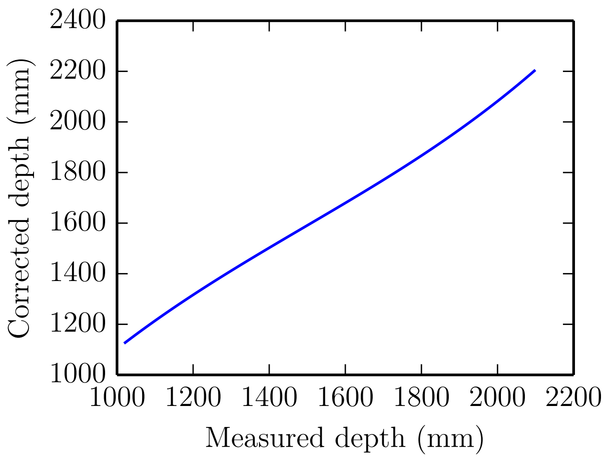


Figure 4.5: Depth correction polynomial.

Depth range		Depth correction coefficients					
d_{min}	d_{max}	d_0	d_1	d_2	d_3	d_4	d_5
1021.39	2096.82	−1012.69	2.4799	0.0064	−0.0859	−0.0017	3.874×10^{-7}

Table 4.2: Depth correction calibration results for 30 MHz and 1000 µs.

4.2 Experiment Goals and Constraints

In Chapter 3, we discussed a model that was an idealized representation of reality. This model relies mostly on refraction at the water surface. However, for our experiment setup we have to consider several other factors which could influence our results. Before we describe our setup in more detail, we give an overview of the constraints these factors impose on our experiment. The goal of our experiment setup is to eliminate as many of these error sources as possible, or at least lessen their influence when that is not possible.

Our model assumes an unobstructed view of the floor, where the only thing between the camera and the floor is a layer of water. This means that any container ideally has to be large enough to contain the field of view of the camera without also showing the walls. If that is not possible, only a subset of the pixels can be used. The distance between camera and floor also plays a role in this.

Areas close to the walls of a container are problematic for another reason as well: The multipath effect is very noticeable in these areas, since here the light can bounce back and forth several times, giving a rounded appearance to edges and corners that are in reality quite sharp. In our setup, we attempt to avoid the multipath effect as much as possible, since correcting this effect in dynamic scenes is difficult. This means that any indirect reflection is undesirable.

The only desired reflection is the direct reflection from the floor. However, for a container filled with water, there are several other potential sources of reflection, all of which are indirect. The container walls have potential to reflect light either before or after hitting the floor, even when there is not water inside the container.

Once there is a layer of water, additional sources of indirect reflection appear: The water surface itself is reflective at some angles and can possibly reflect light back out of the container, which could arrive at the ceiling or anything else overhead. Direct reflection of the water surface also dictates the camera angle: If this angle is too steep, the illumination units of the camera can become visible as oversaturated pixels, which will be marked as invalid.

It is also helpful to keep the camera noise to a minimum. One way to do this is to have a very reflective floor, which provides a strong signal. This also helps with the influence of the absorption property of water on the camera noise and the number of invalid pixels. Since our model assumes a completely flat floor plane, the floor of our experiment has to be flat as well.

To summarize, the main concern for the design of our experiment was the suppression of indirect reflection. In contrast to this, the direct reflection from the floor is the basis of our model and our measurements, so a highly reflective floor is desirable.

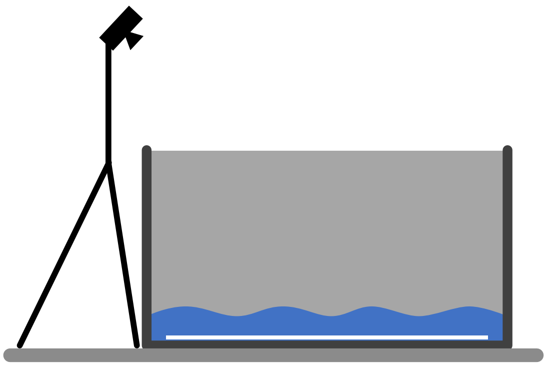
4.3 Setup of Experiment

For the setup of our experiment, we chose to address these concerns in various ways. The resulting setup consists of:

- A container to be filled with water
- A highly reflective floor
- A time-of-flight camera mounted on a tripod
- An infrared absorbing tent

A schematic of how these components were arranged can be seen in Figure 4.6a. The infrared absorbing tent is not included in this schematic. Optionally, this tent can be placed around the experiment in such a way that it covers the whole setup to further reduce reflections.

The container itself is a plastic box with a respective length, width and height of roughly 1.2 m, 1.0 m and 0.8 m. This size was chosen because it was at the same time small enough to be easy to obtain and large enough to contain the camera's field of view. The main concern here was to make sure that none of the walls were visible. Of course the container has to be watertight as well. Figure 4.6b shows the container we used.

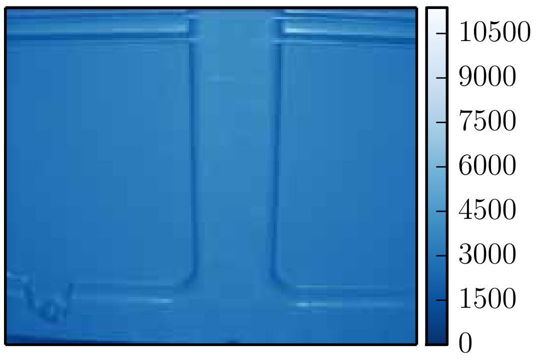


(a) Overview of the main components.

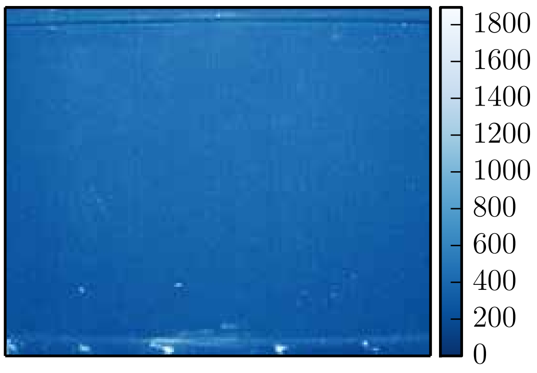
(b) The container.

Figure 4.6: Schematic overview and container.

In order to reduce reflections from the container walls, the inside of the container was coated with blackboard paint. Since no part of the container walls is actually visible in our setup, the main purpose of this coating is to suppress indirect reflection. Figure 4.7 shows the effect of this treatment on direct reflection. We use amplitude images to illustrate the absorption properties of the paint. Low amplitudes are used to flag pixels as invalid. Note that we have not marked any invalid pixels in these images. The outside wall has not been coated, so we use it for comparison here. The images were taken at a distance of about 1.3 m using an integration time of 800 µs.



(a) The uncoated outside of the container.

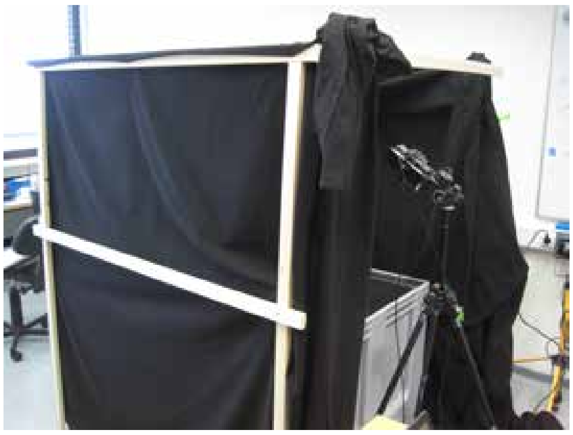


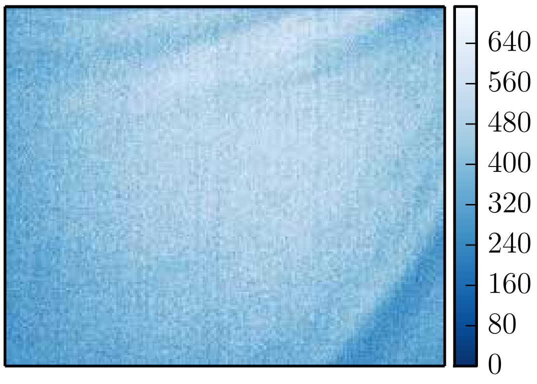
(b) The coated inside of the container.

Figure 4.7: Amplitude images of uncoated and coated container walls.

At the same distance and camera orientation, the outside wall is clearly reflective, as can be seen in Figure 4.7a. None of the wall pixels would be marked as invalid. Figure 4.7b shows the improvement provided by the paint: The overall amplitude of the signal is much lower. If we were to mark invalid pixels, a majority of the pixels on the wall would be invalid. The comparison shows that the paint succeeds in reducing direct reflections. We assume that indirect reflections are suppressed by this even more effectively. The effect also becomes more noticeably at greater distances and lower integration times.

Since the container has to be open at the top, the reflected camera signal can also reach the ceiling and anything overhead. A quick test of holding some objects above the water filled container confirmed this suspicion: Even though these objects were well outside the camera field of view, their reflections were clearly visible in the intensity and distance images. To make sure non of these indirect reflections influenced our results, we constructed a tent out of infrared absorbing cloth. This tent can be placed around the entirety of our experiment setup, as can be seen in Figure 4.8a. Of course it is also possible to remove the tent during an experiment. Figure 4.8b demonstrates the infrared absorbing properties of the black cloth using the same camera settings and relative positioning as Figure 4.7.


(a) The tent placed around the experiment.



(b) The infrared absorbing cloth.

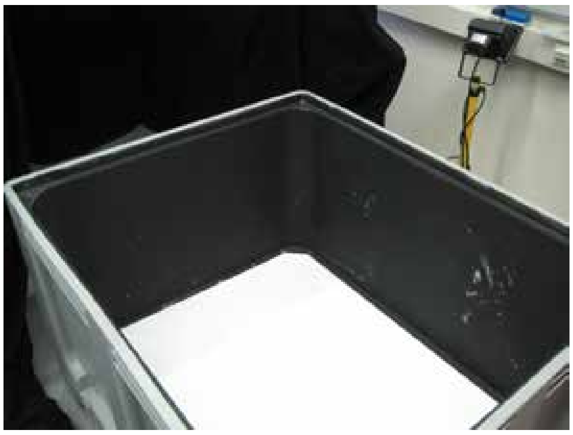
Figure 4.8: Tent constructed out of infrared absorbing cloth.

In contrast to the walls, it is actually desirable for the floor to be reflective, since better reflectivity means less invalid pixels. Although the container material is reflective for infrared light, it falls short when compared to other, more reflective materials. The original floor of the container also had small ridges in it, whereas an ideal floor would have to be completely flat and uniformly reflective. To construct a better floor, we applied matte white spray paint to an aluminium plate. When placed inside the box like in Figure 4.9a, this plate covers almost the entire original container floor. The camera field of view can be constrained to a smaller portion of this floor. When compared to the original container floor, this coated floor plate is both flatter and more reflective.

Figure 4.9b shows the assembled experiment setup when not covered by the infrared absorbing tent. With our setup in place, there are different types of experiments that can be carried out.

4.4 Types of Experiments

Our setup allows for several different types of experiments to take place. The results of reconstructing water surfaces from measurements captured during these experiments will be discussed in Chapter 6. First, we can vary the height of the water surface by pouring in


(a) The floor placed inside the container.

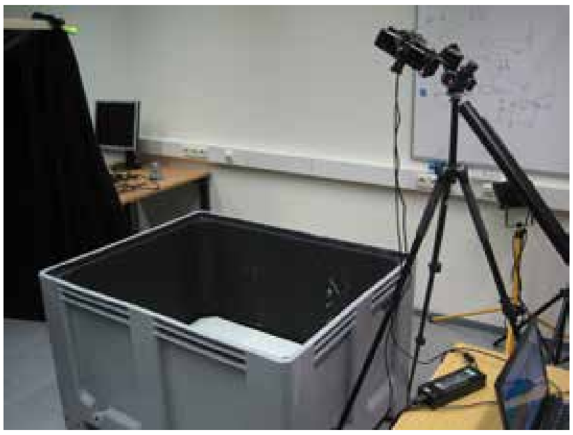

(b) The experiment setup, without tent.

Figure 4.9: The reflective floor and the assembled experiment setup.

controlled amounts of water. This allows use to test the influence of the water height on the distances, as well as on the amplitudes and the noise level. For static scenes of non-moving, flat water surfaces, we can average several consecutive frames to combat noise. If the water is completely flat, it is also easy to obtain a height measurement by simply using a ruler. This height can be used to verify the reconstructed flat surfaces.

For dynamic scenes of moving water, we generate waves from outside the field of view of the camera. One way to do this is to use an object like a length of wood, or dropping something small from overhead. Verifying these waves is more difficult than verifying flat water surfaces, since we have no easy way to measure the ground truth for these waves. For this type of experiment, we can also no longer rely on temporal averaging, since the scene changes to rapidly from frame to frame. The same reason prohibits the use of multipath correction, since this requires changing the modulation frequency of the camera.

One attempt at verifying the moving water surface involved placing small floating polystyrene pieces on the water surface. The distance to these balls can be tracked during the capturing of a wave, in hopes of using this information to check the correctness of the reconstructed wave. In the end, all these methods can only prove plausibility of the result. For a real ground truth, a different, known to be correct method would have to be used.

In Chapter 3 we already hinted at the possibility of a nonlinear influence of water on the measured distances. During our experiment involving increasing water heights, we confirmed this suspicion. Since the nonlinear distance influence can be demonstrated independently of the actual surface reconstruction, we have elected to discuss it here. The reconstruction results based on this experiment are discussed in Chapter 6. The measurement itself was the cause for extending our basic linear model with a nonlinear correction function for underwater distances in Chapter 3.

For the measurement we increased the water height in 5 mm steps starting with the empty container up to a height of 10 cm. We captured distance images using an integration time of 1000 µs and a modulation frequency of 30 MHz. The plot in Figure 4.10a shows the average distance for increasing water height. Based on our linear model, we would also expect a linear increase, as the angle between camera and water surface changes only very slightly. Instead, the plot clearly shows a curve that, after an initial steep incline, flattens for higher water surfaces.

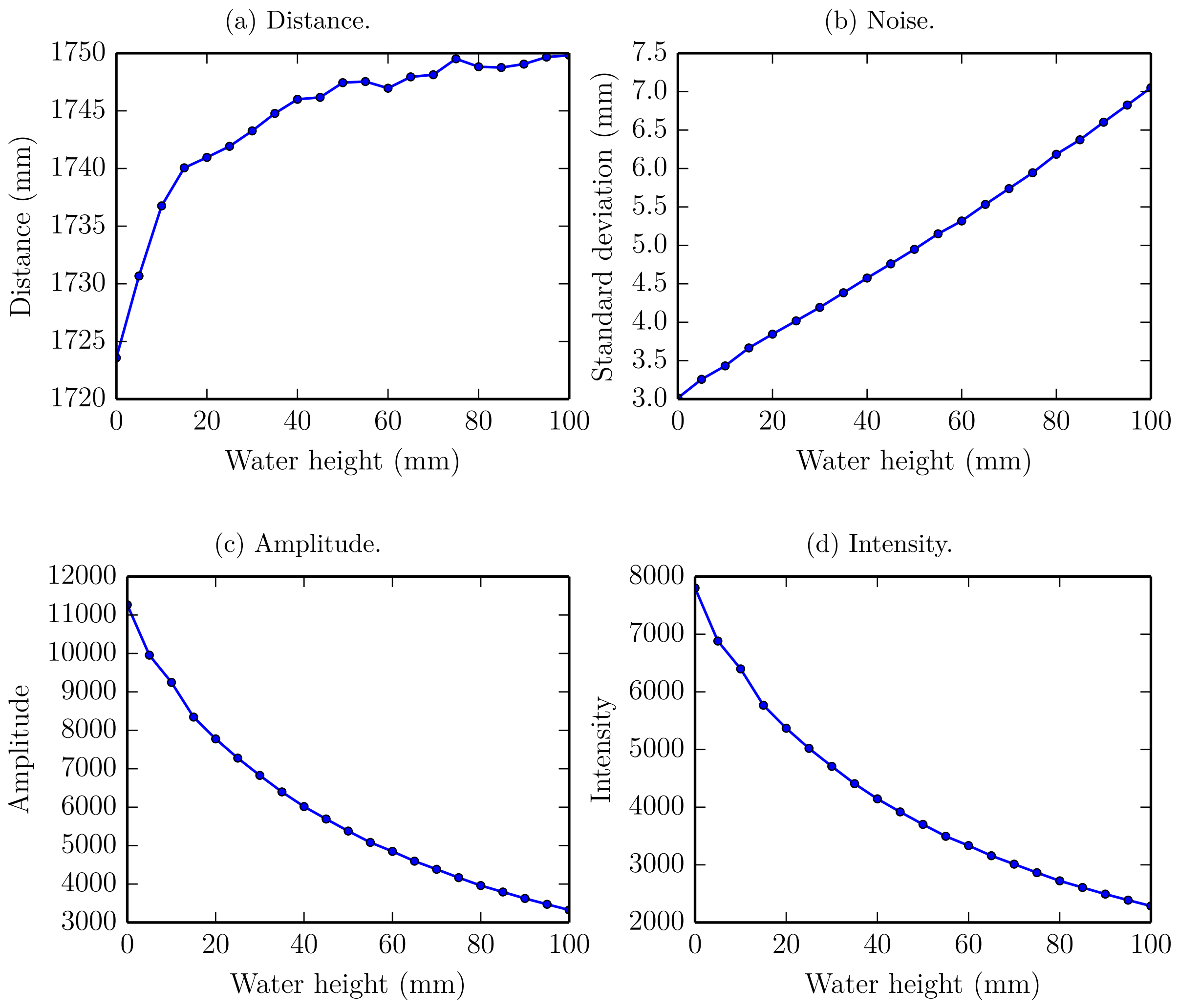


Figure 4.10: Results from measuring increasing water heights.

One possible explanation for this could be the scattering that occurs during absorption. This could cause the signal from a scene point to influence other neighboring signals. As one would expect, the absorption also leads to a decreasing amplitude and intensity, shown in Figures 4.10c and 4.10d. The intensity-related distance error would be another possible explanation. Another interesting takeaway from these measurements is the increasing noise: Figure 4.10b shows the average of the per pixel standard deviation for 100 frames. For higher water surfaces the noise could make reconstruction more difficult. Regardless of its source, the nonlinear distance measurement indicates a clear need for refinement of our model.

Chapter 5

Implementation Details

In Chapter 3, we developed a general approach for reconstructing water surfaces from time-of-flight camera data. We also saw some of the necessary calibration and preprocessing steps when dealing with a real camera in Chapter 4. In this chapter, we discuss an example implementation of the surface reconstruction method and necessary tools surrounding this method. The surface reconstruction itself is also not camera specific. Because we focused on using the PMD DigiCam, many of the tools are geared towards this camera. We start with an overview of the software libraries that were used for our implementation. Then we present the most important points about the implementation itself. The main goal was having a prototype that could be used to verify our refraction-based model on real data. In Chapter 6, we will discuss the results of reconstructing water surfaces with our implementation.

5.1 Programming Language and Library Choices

In this section, we provide an overview of the different software libraries that were used for this thesis. The order in which these are presented roughly follows the data flow. This means we start with the necessary parts to obtain the camera data and finish with the actual surface reconstruction.

The PMD DigiCam can be accessed and manipulated using a C/C++ Software Development Kit (SDK). This SDK can be used to interact with the camera itself or data captured by the camera. The PMD SDK provides calls to obtain distance information, as well as amplitude and intensity images. Invalid pixels are marked by flags. Querying and setting camera parameters such as integration time and modulation frequency is also supported. Additionally, a tool called ‘LightVis’ is provided, which can be used to capture data without using the C/C++ SDK. The camera data is stored in a proprietary binary format using the file extension ‘.pmd’. We extract the information we need from these files using the PMD SDK. To be able to use the data in other programming languages, we store it in a different intermediate format.

We chose the Hierarchical Data Format (HDF) file format [HDF15] for our intermediate files. The reason for this was simply that existing libraries allow both Python and C++ to handle HDF data files. We extract the distance, intensity, amplitude and invalid pixel information for the original PMD data files. Compared to the PMD files, the HDF files are slightly larger in size. We were willing to make this trade-off to facilitate interaction with the data in Python.

Python [Fou01] is a high level, interpreted scripting language. We chose to use it for our surface reconstruction implementation because it allowed us to arrive at a working prototype relatively quickly. The absence of a compilation step was also helpful when interacting with a data set. Since correctness took priority over performance, the use of a scripting language was acceptable to us. Another reason for using Python was the availability of specific libraries for numerical computations, optimization, visualization and camera calibration.

NumPy arrays, as described in [VCV11], are a multidimensional data structure for Python. The shape of these arrays describes their dimensionality. The data type is the type of each individual element, as they are homogeneous arrays. In low level compiled languages like C, the standard approach for numerical computations often involves using for loops to operate on individual elements. However, in high level interpreted languages like Python this approach can result in a performance bottleneck. NumPy arrays provide a way to execute numerical computations in Python without this drawback using *vectorization* and *broadcasting*.

Vectorized operations like multiplication are applied to entire NumPy arrays at once. This avoids having to iterate over each individual element. Broadcasting extends this concept to arrays of different shape. If the shapes of a smaller and a larger array are compatible, the smaller array is broadcast to fit the shape of the larger array. An example of this would be adding a three component array to a $5 \times 5 \times 3$ array: The smaller array is added to each of the 5×5 elements of the outer two dimensions, which are also three component arrays. The actual code executed by NumPy during these vectorized operations is implemented in C. Thus, NumPy provides a way to express numerical computations in a high level language while retaining at least some of the performance benefits of a lower level language.

NumPy arrays can also contain masked entries to mark invalid values. Because other NumPy routines are aware of these masks, we can use them to handle invalid pixels in our distance images. We use NumPy for the majority of the calculations in our implementation.

SciPy [J+01] is a library which uses NumPy arrays as a basis to implement various useful tools for scientific computation. For our purposes, we used the optimization algorithms provided by this library, as well as some image processing and linear algebra tools. Most importantly, some of the optimization algorithms are large scale solvers that can be used to find a solution for a system of equations with a large number of variables.

Matplotlib [Hun07] is a Python library for creating plots and figures. We used it extensively for dataset exploration and to visualize the results of our implementation. A number of figures in this thesis have also been created using matplotlib.

OpenCV [Its15] is a library that provides a large number of routines for computer vision. Even though it is implemented in C++, Python bindings are also provided. We use OpenCV for calibrating some of the intrinsic camera parameters, but also for removing the distortion of the captured images.

5.2 Code Overview

Our implementation can be roughly separated into a part that does the actual surface reconstruction and a set of supporting tools for this task. The supporting tools are mostly geared towards handling camera data as well as evaluating the reconstruction results. We used Python for most of our implementation, but resorted to C++ where necessary or more practical.

A number of conversion tools can be used to convert the proprietary format used by the camera to the HDF intermediate format we use in our implementation, as well as regular image files or XML files. The XML files are necessary for the depth calibration tool. The conversion occurs offline after the data has been acquired. For capturing the camera data itself, we use both the LightVis tool as well as some tools of our own. Our custom tools are mainly used for calibration, where we wanted to acquire the same scene for a number of integration times or capture images at certain time intervals. Note that since these conversion and capturing tools all use the PMD SDK, they had to be implemented in C++.

For the intrinsic calibration, we used our own OpenCV based script, whereas we used the depth calibration tool for the depth calibration with fixed intrinsic parameters. We also converted the resulting XML output of the depth calibration tool to a different format used by our reconstruction implementation. The reconstructed surface heights can be visualized by both Matplotlib based plots and a custom tool we used internally to render a simple wireframe representation of the surface mesh. This is possible for both single frames as well as animated sequences of frames.

The surface reconstruction itself is implemented in a Python module using four classes.

CameraSettings The *CameraSettings* class contains all settings that are only relevant for the camera itself. By reading a configuration file generated from the calibration results it can be configured for any specific camera, and contains such information as per pixel viewing directions, camera parameters, as well as the relative position of camera and floor plane. In our implementation, we perform all calculations in camera space, and so the floor plane is characterized by a support vector and normal vector in this space.

CameraData The *CameraData* class uses the *CameraSettings* class to perform the necessary preprocessing steps on a camera data set. This includes masking of invalid pixels, undistortion and depth correction. A *CameraData* object can store single frames as well as animated frame sequences.

Simulator The refraction-based model discussed in Chapter 3 is implemented in the *Simulator* class. This class generates synthetic distance images from height field information. The model assumes a single point of refraction at the surface intersection. We use the *CameraSettings* class for information such as the relative position of the camera and floor plane or the viewing directions that are needed for the simulation. The *Simulator* class is used for both simulating the effect of changes to our model, as well as for optimization. It can also generate synthetic datasets from known height fields to test different optimization algorithms. The refraction is calculated using a fixed index of refraction, which can be adjusted for the specific camera used. [HQ73] contains a table of refractive indices and absorption coefficients for water at various wavelengths. We used the index of refraction listed for 850 nm, which is 1.329. The absorption coefficient is not used directly by our implementation.

Optimizer The optimization is performed by the *Optimizer* class. This class is mostly a wrapper around the optimization functions exposed by the SciPy library. The optimization is performed by using a distance image as an optimization target, which can be either synthetic or from a real data set. The *Optimizer* adjusts a field of heights and uses the *Simulator* to calculate a distance image according to our model. The error is evaluated between the target distances and the simulated distances. Additionally, a smoothness constraint placed on the height field limits the height difference between neighboring pixels. We used a convolution with a 3×3 kernel for the approximation of the Laplace operator, and a 5×5 kernel for the LoG. The SciPy library offers several different solvers for finding the root of a vector function. For our implementation, the choice of solvers is constrained to those that can handle large scale problems. These include the 'Anderson' and 'Krylov' solvers, both of which are inexact newton solvers. The results presented in Chapter 6 were produced using the 'Krylov' solver in conjunction with the LoG smoothness constraint.

Surrounding these core classes are a number of scripts that make use of them in order to reconstruct surface information from a dataset. Simulation of reconstruction on generated height fields is supported, which allows the evaluation of the error not just on the distance, but on the height itself. These synthetic data sets were mostly used to test the optimization algorithms directly. Our implementation requires that we capture the empty container before filling it with water. A plane is then fitted through this data set. After this initial data set is acquired, the camera and container must not be moved. Reconstruction of several consecutive frames is also supported.

Figure 5.1 shows a typical workflow enabled by our implementation. After acquiring a data set, we need to convert it in order to use it for the reconstruction, but also if we wish to calibrate the camera with it. The result of the calibration is a file containing the camera settings. Our surface reconstruction scripts can then reconstruct a water surface for a specific data set. Visualizing and evaluating the results can give us hints how we need to adjust our model or our implementation.

$$\begin{tikzcd}\boxed{\text{Acquisition}} \arrow[r] & \boxed{\text{Conversion}} \arrow[r] & \boxed{\text{Reconstruction}} \arrow[r] & \boxed{\text{Evaluation}}\end{tikzcd}$$

Figure 5.1: Workflow example.

All of the tools that were written in C++ are contained within CMake projects. CMake [Kit15] is a tool that enables the generation of project files for a large number of development environments and platforms. Using CMake for configuring the dependencies of our small tools enables some level of portability. For executing the scripts written in Python, a Python environment containing the NumPy, SciPy and OpenCV libraries is necessary. The code is documented using Doxygen [Hee15], which uses a special markup language to generate documentation from comments within the source code.

Chapter 6

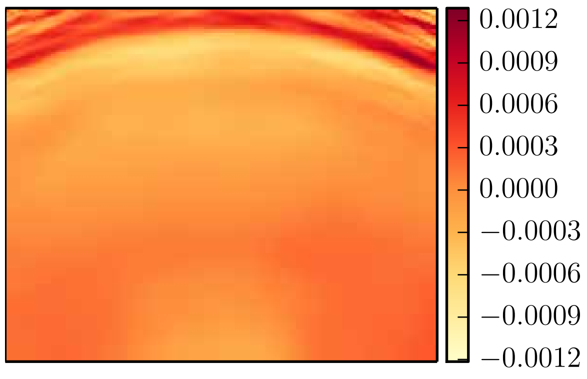
Result Evaluation

We developed a method for reconstructing fluid surfaces from time-of-flight camera data in Chapter 3. In Chapter 4, we described the setup we used to capture moving and flat water surfaces. We also saw the necessity of refining our model for a nonlinear influence of water on the distance. Our implementation of the reconstruction method was discussed in Chapter 5. In this chapter, we evaluate the surfaces reconstructed by our implementation. We begin with the reconstruction from synthetic data sets. The main goal here is to show the stability of the optimization itself. We then proceed with reconstruction of real water surfaces. For flat water surfaces, we discuss a possible distance correction function. For moving waves, we unfortunately have to revert to our basic linear model. Aside from showing the reconstruction of dynamic water surfaces, we also attempt to verify an example wave with a polystyrene floater. The chapter concludes with an overview of the limitations of our approach.

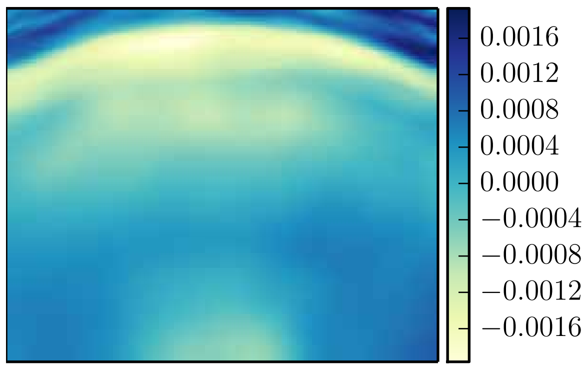
6.1 Simulation Results

Our *Simulator* class enables us to calculate distance information from water surface heights. It does this by implementing the refraction-based model we developed in Chapter 3. While our main use of the simulator is as a part of the optimization, we can also use it to generate synthetic data. The synthetic data is useful in order to test whether or not a given optimization algorithm is suitable for our purposes. Since we calculate the synthetic data from a known height field, we can evaluate the result not only with regard to the distance, but also with regard to the water surface height. Because we use the same simulation settings for generating the synthetic data and reconstructing the surface, this mostly tests the optimization algorithm and our surrounding implementation. In other words, the model we use for the simulated reconstruction is a perfect fit for the synthetic distance target. It is not necessarily the correct model to use with real water. Nevertheless, some characteristics of real data can also be tested: We can mark a number of pixels as invalid, and can add random noise to the calculated distances.

For the simulated distances discussed here, we used the same camera settings as for a real reconstruction based on our experiment setup. This means that the intrinsic parameters of our camera are the same, as well as the position and orientation of the floor plane. We only evaluate single frames instead of generating entire sequences of moving waves. The refraction-based model used in this section is the linear model, where we only multiply the underwater distance by the refractive index of water. The smoothness constraint itself is the LoG, approximated by a 5×5 convolution kernel. The optimization algorithm is the ‘Krylov’ large scale nonlinear solver provided by SciPy. For all error functions plotted here, the error is the difference between the reconstruction result and the ground truth.



(a) Distance error (mm) for smoothness $s = 1$.



(b) Height error (mm) for smoothness $s = 1$.

Figure 6.1: Simulation results for flat water at a height of 5 cm.

To begin with the most simple case, we examine the influence of different factors on a simulation of perfectly flat water at a height of 5 cm. Figure 6.1 shows the result of this simulation in the absence of any disturbing effects. The distance error depicted in Figure 6.1a is the basis of the decisions of the optimization algorithm. It tries to match a target distance image up to a specified precision. For a lower error tolerance, the number of necessary iterations increases. Since the real datasets we captured contain a certain amount of noise, the precision we require of the optimizer is somewhat limited. The height error shown in Figure 6.1b is distributed in the same pattern as the distance error. What is also visible is the relationship between distance error and height error: The height error is typically larger than the distance error, since the optimization can only evaluate the influence of height changes indirectly.

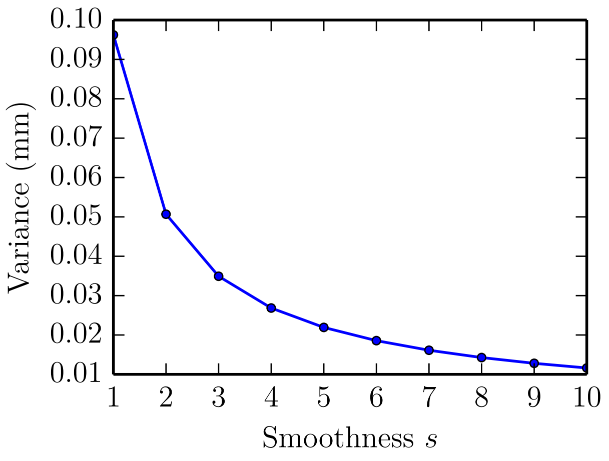
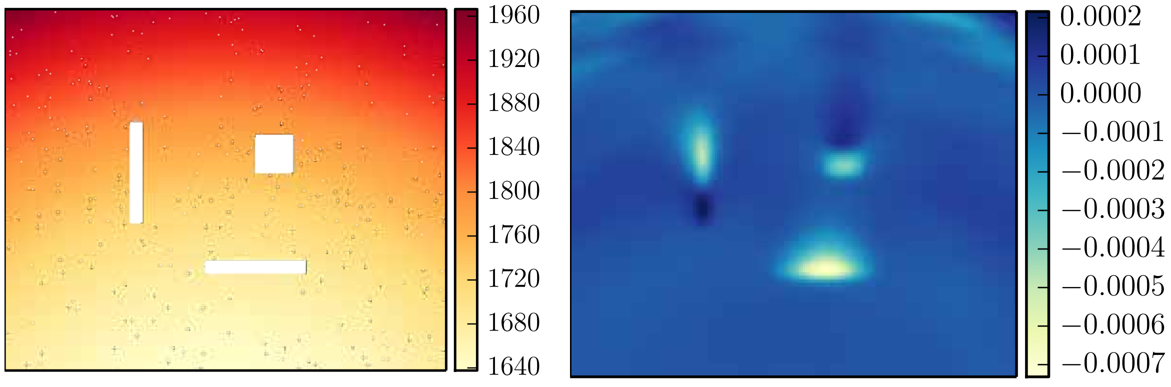


Figure 6.2: Influence of smoothness s on the height error variance.

Next, we introduce noise to our simulation of flat water. To do this, we add a random offset of up to 3 mm to each distance pixel. This offset can be both positive or negative. The heights remain untouched because we assume that the noise is produced by the camera, not the water surface. The reconstruction produces a flat surface at the correct height offset by a pattern. The pattern visible in the height error now reflects the random nature of the noise. By adjusting the smoothness, we can reduce the variance of this pattern, as can be seen in Figure 6.2. Our implementation requires that at least some level of smoothing is always present. Otherwise the optimization will not converge in the presence of noise.



(a) Target distances (mm).

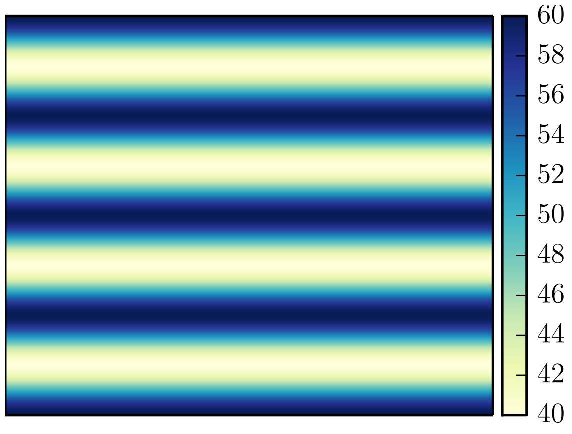
(b) Height error (mm) for smoothness $s = 1$.

Figure 6.3: Influence of invalid pixels on the simulation results for flat water.

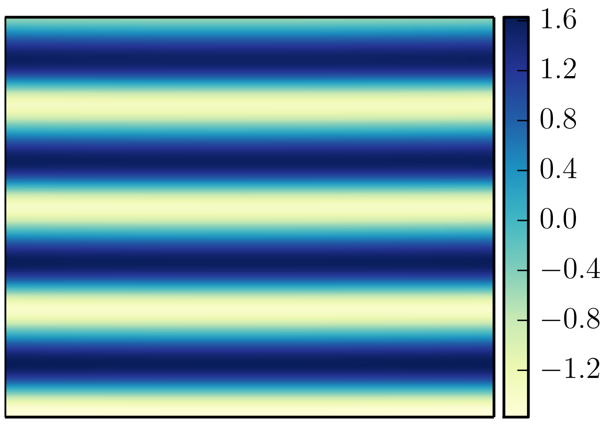
Invalid pixels are another source of errors for a real data set. To simulate these invalid pixels, we marked a number of random individual pixels as invalid in our synthetic distance image. Additionally, we added larger rectangular regions of invalid pixels to some parts of the image. Figure 6.3a shows a distance image that has been manipulated in this way. The small individual pixels are corrected by any smoothness value. For the larger regions, the optimization requires greater smoothness values and longer iterations to reach the inner parts of these regions. The height error in Figure 6.3b is the result of such a reconstruction with smoothness $s = 1$. In the real data sets we captured, individual invalid pixels and thin streaks are much more common than large extended areas.

Finally, we evaluated the reconstruction of a simulated wave. The wave discussed here is a sine wave that oscillates with an amplitude of 1 cm around a base height of 5 cm and can be seen in Figure 6.4a. Other waves can be used to produce similar results. Figure 6.4b shows the height error of the reconstructed wave. A problem introduced by the smoothness becomes evident in this result: The highest parts of the wave are too low, and conversely, the lowest parts are too high. This is a direct result of smoothing the reconstructed heights during optimization. Increasing the smoothness also increases this error.

To summarize our findings, we can say that the smoothness constraint presents a double edged sword. On the one hand, the smoothing is necessary to deal with the noise that is always present in a recording of a real camera. For static scenes of non-moving water, we can reduce the noise by averaging several frames. In a dynamic scene of moving water, this becomes



(a) Target heights (mm).



(b) Height error (mm) for smoothness $s = 1$.

Figure 6.4: Simulation results for a wave.

impossible. Enforcing some level of smoothness on the water heights also aids the convergence of the optimization algorithm. It is also a sensible constraint for a water surface to require no sharp edges. On the other hand, selecting a stronger smoothness can produce reconstructed waves that are flatter than they should be. The relative shape of the wave is not affected by this. Selecting a smoothness parameter requires a trade-off between these two extremes: If it is chosen too low, the noise begins to dominate the reconstructed result. If it is too high, the amplitude of waves is likely lower than reality, and smaller waves will get lost in the noise entirely.

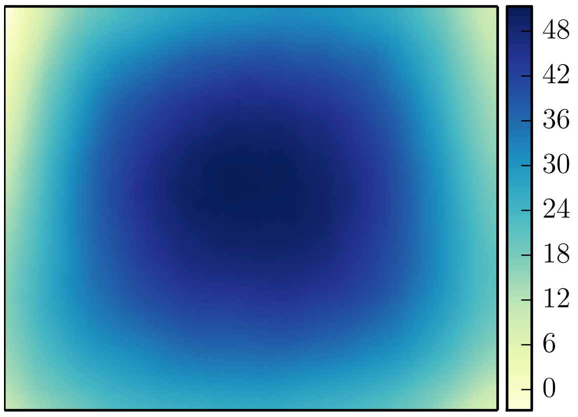
6.2 Experiment Results

We will now cover the reconstruction of real water surfaces captured using the experiment setup we described in Chapter 4. This section is divided into two parts which deal with different types of water surfaces. First, we are going to discuss the reconstruction of non-moving water surfaces at different heights above the floor. After this, we will focus on the reconstruction of dynamic scenes, where waves disturb the flat nature of the water surface.

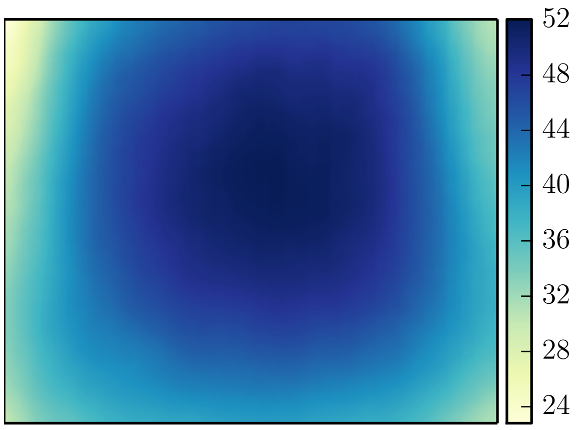
6.2.1 Still Water

An advantage of capturing non-moving water surfaces is that we can measure the actual height of the water using a simple ruler. We can also expect the water surfaces to be completely flat. Averaging several frames is a legitimate option for noise reduction in this case, since the actual scene is assumed to be static.

In Chapter 4 we already took a brief look at a dataset of still water heights. The water was increased in 5 mm steps up to a final height of 10 cm. The most noticeable effect was the nonlinearity of the distances: After rising steeply for the first couple of heights, the distances roughly follow a curve which becomes flatter as the water height increases. This effect can be observed both for individual pixels as well as for the average of larger blocks. We hypothesized possible reasons for this nonlinearity: The scattering that occurs as part of the absorption could be one explanation, as could be the camera itself. We also noticed an increase in noise for increasing water height which makes reconstruction of higher water surfaces more difficult compared to lower water levels.



(a) Linear model with slope 1.73.



(b) Linear model with slope 1.56 and per pixel offset.

Figure 6.5: Reconstructed heights (mm) for 50 mm of water.

The most simple refraction model assumes a linear underwater distance which is only changed by a coefficient that controls the slope. This model is motivated by the index of refraction which is related to the lower speed of light in water. If we use the index of refraction directly for this coefficient and reconstruct the surface for the water height of 50 mm, the resulting heights are much too high. Figure 6.5a shows a first attempt at improving this result: By increasing the underwater coefficient, we indirectly reduce the overall height of the reconstructed surface. This is due to the fact that for a linear model with a greater slope, an increase in water height has a greater influence on the overall distance. What is also visible is the fact that the reconstructed heights fall of dramatically towards the edges.

The curvature of the result is at least to some extent already present in the image taken of the empty container. If we use the difference between the captured container and our ideal floor plane as a per pixel offset for the distance, we can mitigate the effect of falling off towards the edges to some degree. This is shown in Figure 6.5b, where we also used a different coefficient to arrive at a height of 50 mm. In both images, the highest point is located towards the center of the image.

Figure 6.6 shows the limitations of our linear model when applied to reconstructing the water heights at each interval. The linear distance model will always result in a linear total distance when simulating completely flat water surfaces, as Figure 6.6a shows. We can control the slope and overall position of this line, but it is immediately clear that this model is not a good fit for the nonlinear real distances. Figure 6.6b confirms this suspicion: The blue line represents the reconstructed water heights for each captured data set. The results were evaluated for a block of pixels in the center of the image due to the falloff towards the edges. The green line represents the ideal expected result of the reconstructed water height, since we increased the water level in a linear fashion. By adjusting the slope and offset of our linear model, we can at

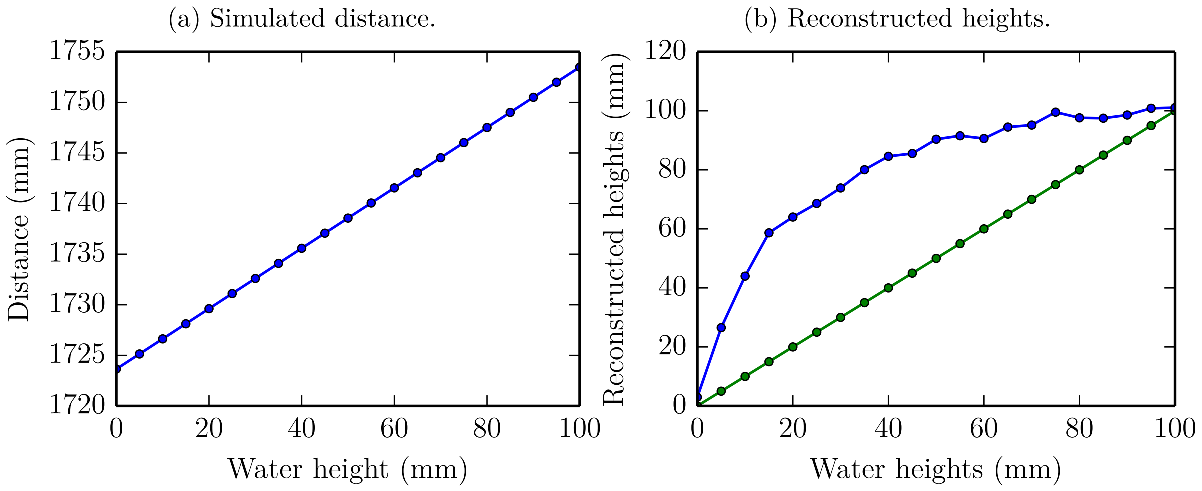


Figure 6.6: Linear model and reconstruction results for this model.

best improve the results for parts of the curve that are closer to linear. For the overall complete range, a nonlinear model of the water distance is needed.

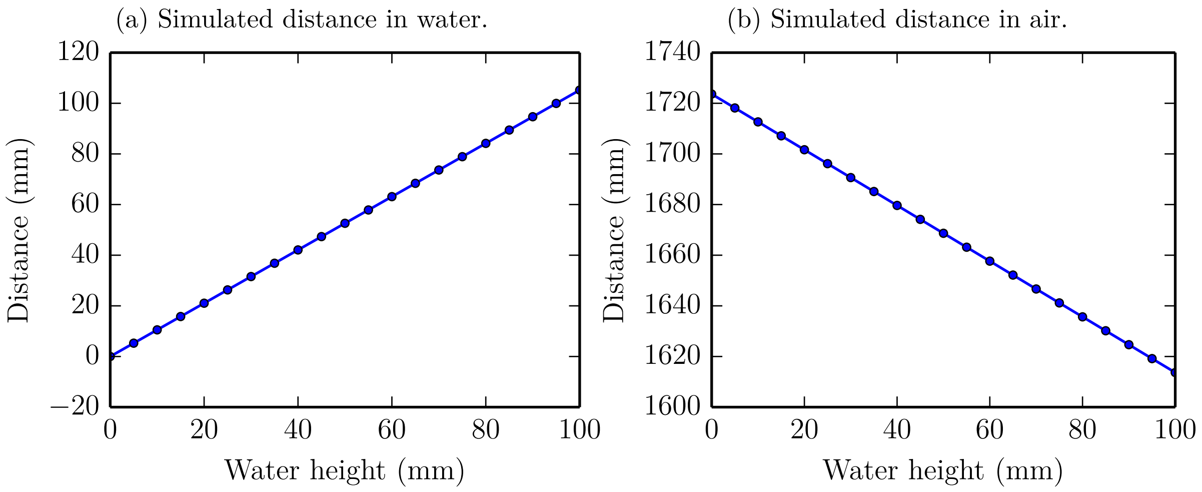
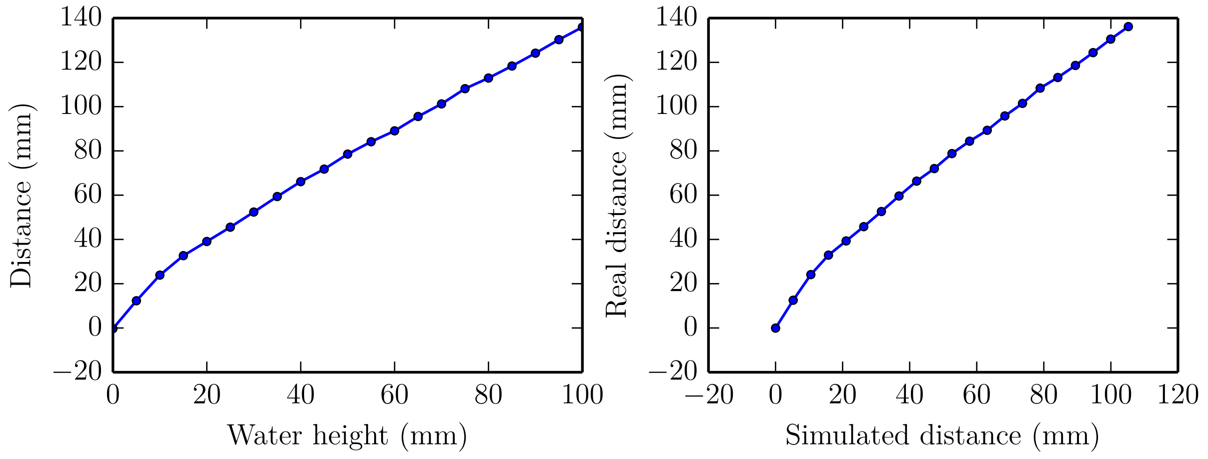


Figure 6.7: The two components of the simulated distance.

We assume that only the underwater part of the distance needs to be corrected, as the water should not influence the air part of the ray. Figure 6.7 shows the two components of our model as they would appear without any alteration for simulated flat water heights. As one would expect, the distance in water increases while the distance in air decreases if the water level rises. We then subtract the simulated distance in air from the measured distance. The result of this a curve that represents the distance in water that our simulator has to produce in order to fit the measured distance curve. As can be seen in Figure 6.8a, the curve depends on the water height. For correcting our simulated water distance, we plot the same result against the simulated water distance. This is shown in Figure 6.8b. The function represents a mapping of simulated distances to real distances.



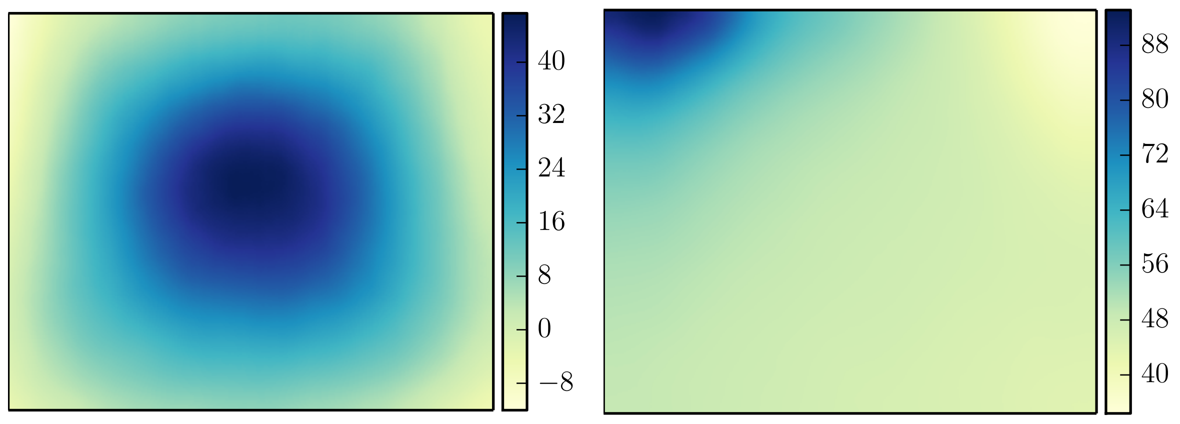
(a) Measured distance in water. (b) Correction target function.

Figure 6.8: Nonlinear water distance correction.

We use this function to correct our model for the nonlinear distances. The function is approximated by fitting the third degree polynomial

$$d_w^* = f(d_w) = ad_w^3 + bd_w^2 + cd_w + d \tag{6.1}$$

through the sample points, where d_w is the distance in water before correction, a, b, c, d are the coefficients for the fit and d_w^* is the corrected distance. $f(d_w)$ is an example of the arbitrary correction function we mentioned in Chapter 3. The process is first performed for the average of a block of pixels around the image center.



(a) Block fit results. (b) Per pixel fit results.

Figure 6.9: Reconstructed heights (mm) for 50 mm using a nonlinear model.

Figure 6.9a shows an example of reconstructed heights using this nonlinear correction. At first glance, the result seems worse than the results shown in Figure 6.5. This is certainly true for the parts toward the edges of the image. However, our linear model was only correct for this one example height because we adjusted the slope and the offset of it. For other heights along the curve, the adjusted linear model provided completely wrong results again. The average heights of the central block when reconstructed using our nonlinear model are shown in Figure 6.10a. Already, the nonlinear model provides a reconstruction much closer to the ground truth.

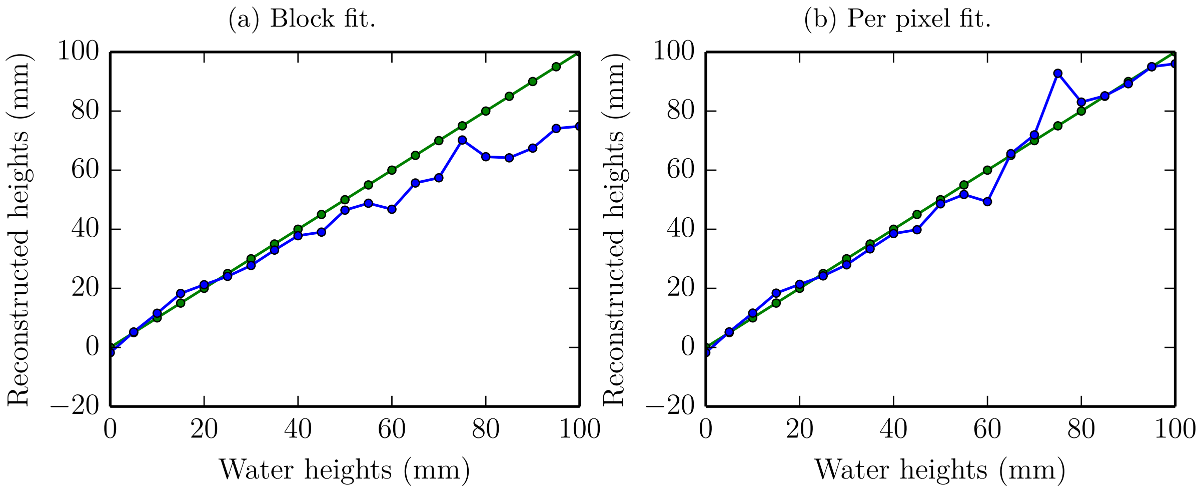
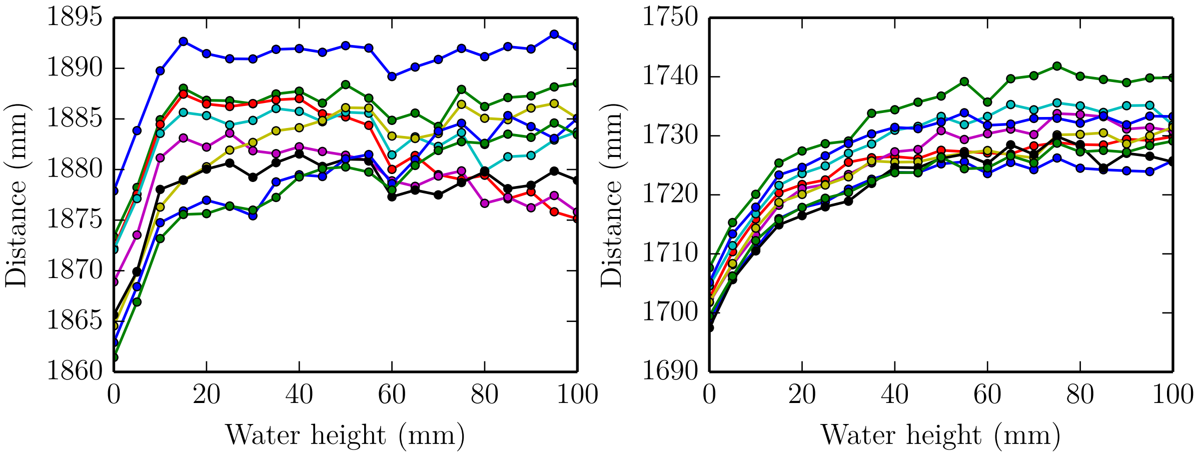


Figure 6.10: Reconstructed heights for corrected water distances.

However, fitting a function through an average of several pixels only provides a good correction for that subpart of the image. If we increase the block size or even use the entire image, the curvature of the result also increases, making the effect worse. Instead we repeated the process of fitting a curve through our underwater distance correction function on a per pixel basis. Before, we have corrected the distances in the same way for each pixel. An example reconstruction result using the per pixel correction can be seen in Figure 6.9b. Except for the corners of the image, large parts are actually flat and at the correct height. Figure 6.10b presents the reconstructed heights for all water heights. Except for some outliers, which might be explained by measurement errors, the reconstructed heights fit the ground truth better than any other method we tried.

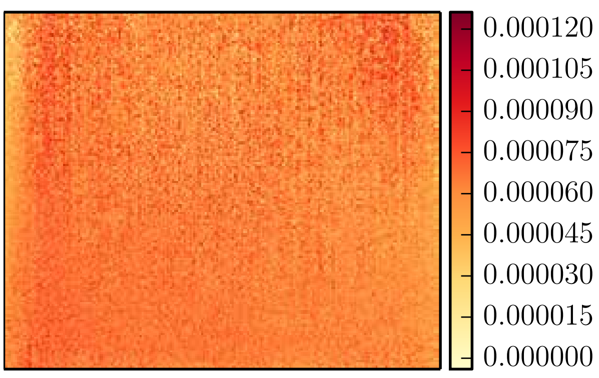
To investigate the cause of the incorrect values in the corners, we took a closer look at the measured distances for individual pixels. We sampled nine individual pixels in close proximity, once for a corner region and once for a region in the image center. The result is depicted in Figure 6.11. For the corner pixels in Figure 6.11a, it is obvious that our third degree polynomial cannot provide a good fit for the correction function. An explanation could be the close proximity to the wall of the container. Contrary to this, the center pixels in Figure 6.11b appear much more suited to our polynomial. Some of the outliers are also visible even at this very granular level. This explains why our per pixel correction function, and perhaps all our reconstruction attempts, work better on the center of the image.

Unfortunately, our nonlinear model is not easy to reuse on a different data set. Even small changes to the camera position result in very different reconstructed heights. This suggests

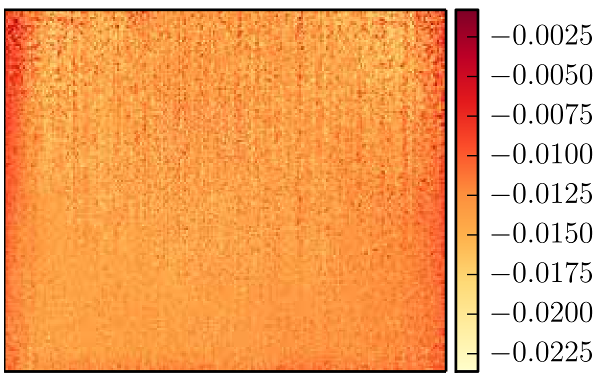


(a) Pixels in the corner. (b) Pixels in the center.

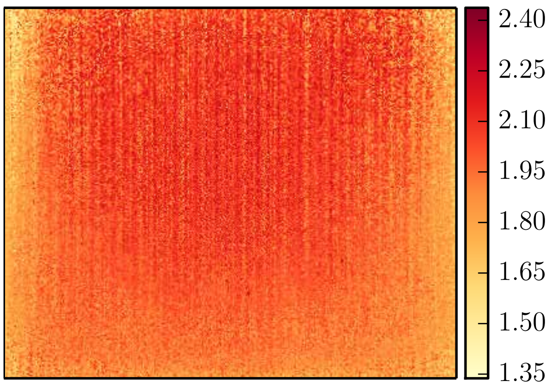
Figure 6.11: Distances for individual pixels.



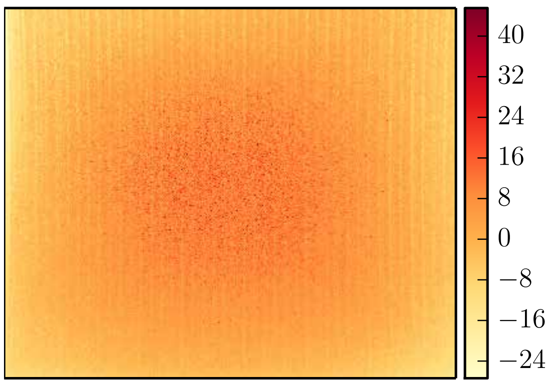
(a) Coefficient a.



(b) Coefficient b.



(c) Coefficient c.



(d) Coefficient d.

Figure 6.12: Per pixel correction coefficients.

that the correction function might also be dependent on the viewing direction. Taking a closer look at the distribution of the correction coefficients in Figure 6.12, we can see the hint of several other influences: In the parameters c and d, a radial component is visible. This could be related to the camera lens itself. A pattern of vertical lines is also visible in these images, which could be caused by the camera sensor. Lastly, the areas near the left and right sides indicate that the walls might still have an influence here despite all our precautions.

Another unfortunate consequence of the inability to transfer this correction directly to a new dataset is that we must resort back to our linear model for the reconstruction of waves. This model can still provide an approximation in a local neighborhood of a certain water height, but we can already presume that the reconstructed heights will be somewhat incorrect.

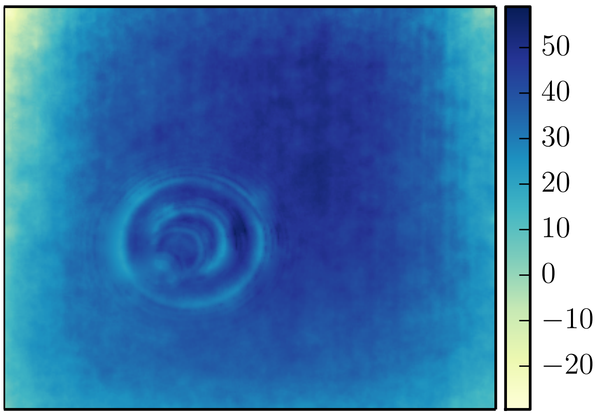
6.2.2 Moving Water

As we saw in the previous section, the linear refraction model is not sufficient for reconstruction of a range of flat water heights. Adopting a per pixel correction based on a third degree polynomial improved our results. Unfortunately, this correction appears to be viewing direction dependent, and is therefore not easy to apply to a data set where the camera position has been altered even slightly. Because of this, we reverted to using the linear model for the reconstruction of moving water. The results we are about to discuss were produced by using the index of refraction η_w as a coefficient for our underwater distance. The waves were created at a water height of 50 mm. We used an additional global offset to ensure the reconstructed flat water surface provided a similar starting height. This offset was adjusted manually for a block of pixels in the image center. The smoothness s was kept at 1 for these reconstructions.

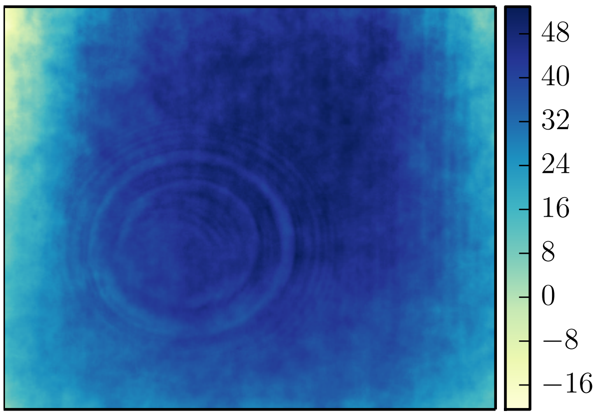
Figure 6.13 contains reconstructed frames from a variety of different wave sequences. To increase the visibility of the waves, we subtracted the difference image between empty container and the simulated distance to the floor plane. As we saw before, this results in a reduction of the falloff towards the edges. Keep in mind that since our model does not reflect the nonlinear influence of the water on the distance, the heights of these waves are unlikely to be correct. Nevertheless, we can still observe which parts of the wave are above or below the starting height. The general shape of the wave is still visible and should not be affected by the lack of nonlinear correction.

To generate the circular wave in Figures 6.13a and 6.13b, we used a single drop of water. As the concentric circle expands, it also becomes much flatter, which is what one would expect in this situation. The other waves were produced by placing a piece of wood in the container outside the camera field of view. We used this piece of wood to create vertical, horizontal and diagonal waves. Figures 6.13c and 6.13d show the same wave as it travels from the right edge of the image to the left. This wave was likely higher in reality than the circular wave, which can also be seen in the reconstructed heights. For completeness, waves traveling in vertical and diagonal directions are shown in Figures 6.13e and 6.13f.

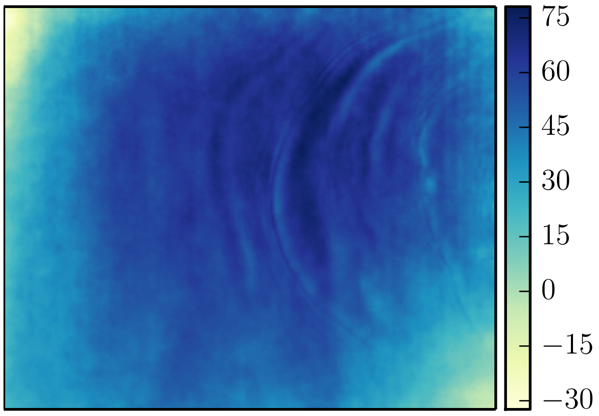
Even though we expect the absolute heights of the waves reconstructed using a linear model to be most likely incorrect, we can still use them to demonstrate a possible method for evaluating the reconstructed heights. This method is based on placing small pieces of floating polystyrene pieces inside the water. These floaters will follow the surface and move up and down during a wave. To prevent them from moving too far from their starting position, we tie them to small anchors using pieces of string. The length of the string determines how far the floater



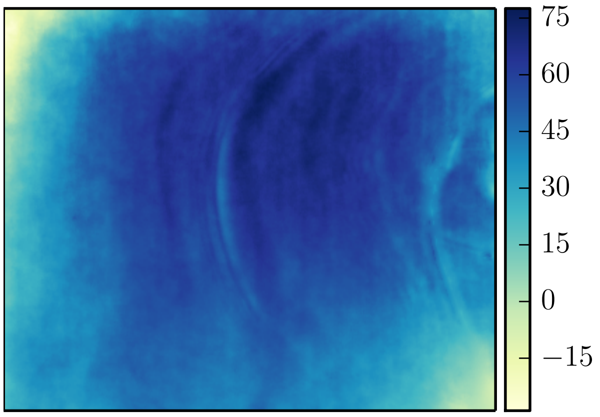
(a) Circular wave.



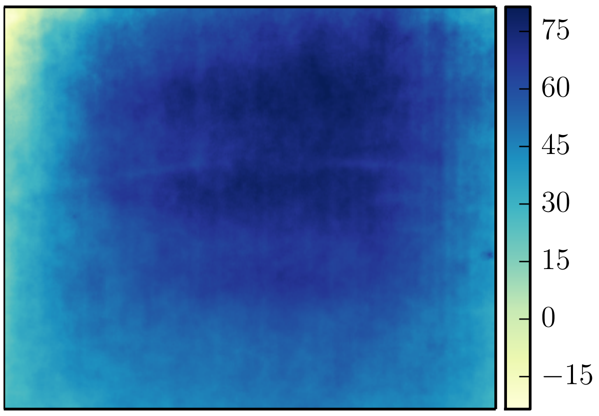
(b) Circular wave.



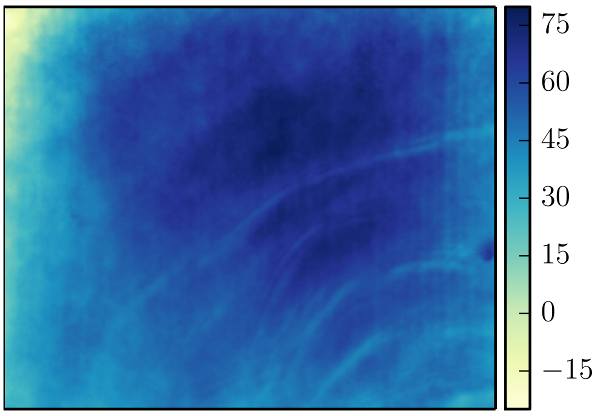
(c) Wave in horizontal direction.



(d) Wave in horizontal direction.



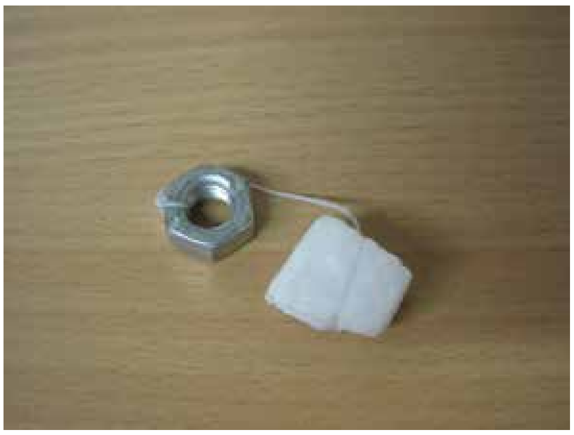
(e) Wave in vertical direction.



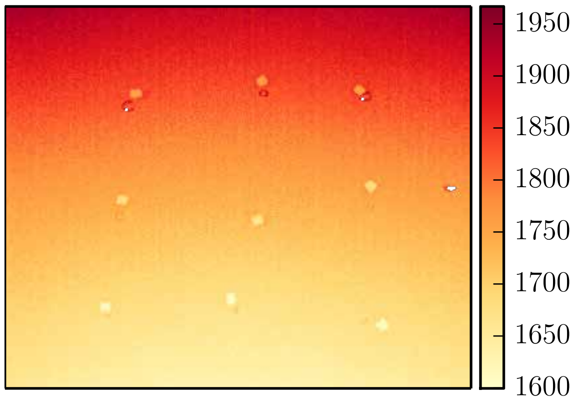
(f) Wave in diagonal direction.

Figure 6.13: Reconstructed heights (mm) for different waves.

can move, both in height and position. An example of such a floater with its anchor is pictured in Figure 6.14a. In Figure 6.14b, a total of nine such floaters have been placed in the container. The floaters are visible in the distance image and are closer than the background floor. We will focus our attention on the middle floater because the results in the middle of the image have proved most reliable thus far.



(a) Floater with anchor.



(b) Floaters placed in the container. Distances in mm.

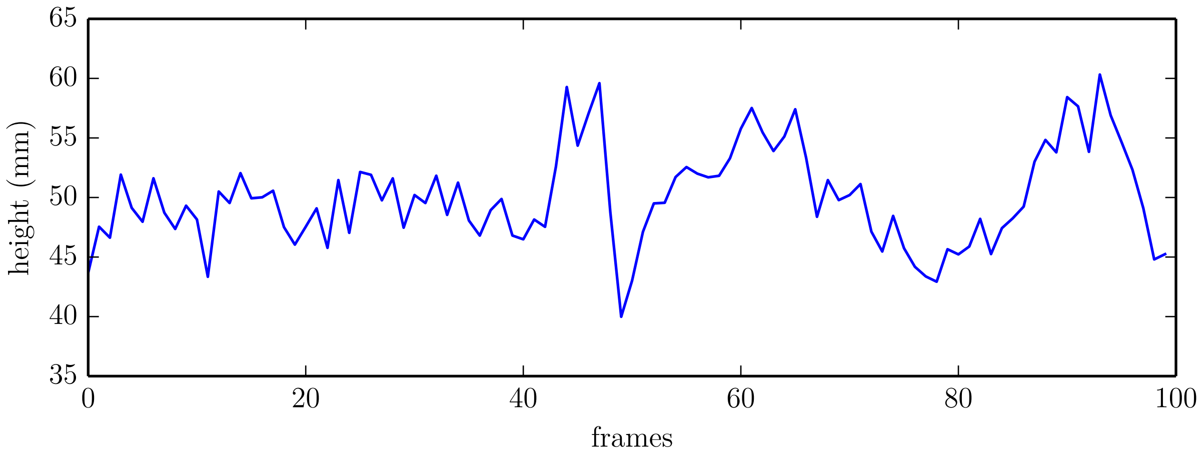
Figure 6.14: Floating polystyrene for validation of reconstruction results.

The distance measurement can be used to track the floater in a very rudimentary fashion. We use a simple manually adjusted threshold to figure out the position of the floater in each frame. We use a 3×3 pixel neighborhood around the centroid of the tracked floater. The wave discussed here is a wave traveling from the bottom to the top edge. For some of the frames during the peak of the wave, we manually adjusted the centroid position because our very simple tracking method failed on these frames. Figure 6.15a contains a plot of the relative height change of the tracked floater with respect to the floor plane. The plot was adjusted due to the assumption that the average floater height was at 5 cm.

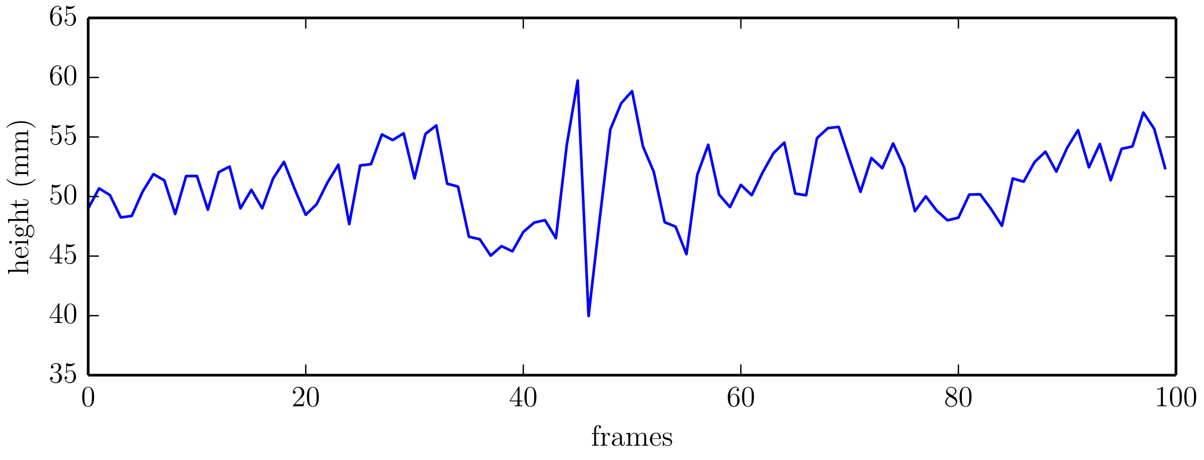
The distance measurement for the floater is also subject to the usual effects such as camera noise. The small neighborhood size is a consequence of the physical size of the floater, but also of the fact that the wave passes a single point in the image quite rapidly. We can see the first peak of the wave arriving shortly after frame 40. Two more peaks appear to follow the first, slightly smaller than the first. Between these, the height of the floater drops below the average height. The sequence before frame 40 contains mostly undisturbed water, where the camera noise should be the only influence.

We reconstructed the wave using the same linear model as before. A 3×3 pixel neighborhood 30 pixels to the left of the floater was chosen for comparing the reconstructed heights to the tracked floater. This time, we adjusted the global offset of our linear model so that the average reconstructed height was 50 mm for this neighborhood. We kept the index of refraction as our water distance coefficient, but did not use the difference image to correct the curvature towards the edges. The adjustment using a global offset also means that other parts of the reconstructed height image are likely to be at an incorrect height.

The reconstructed heights for this neighborhood are plotted in Figure 6.15b. The first peak of the wave is also visible in this plot. Interestingly enough, the actual height of this peak appears to be very similar to that of the tracked floater. However, the following two peaks are much less pronounced in this plots. Were it not for the fact that we know of these peaks from the tracked floater plot, the two following peaks might also be mistaken for noise. Their general shape is still possible to see, if one is looking for it.



(a) Water heights for a tracked polystyrene floater.



(b) Water heights for a reconstructed wave.

Figure 6.15: Water heights over time.

Overall, we would caution against seeing these plots as proofs for our method, simply because of the fact that in the previous section we already showed the shortcomings of the linear model. Instead, they can only provide a way for us to evaluate the general shape of the wave. From this example, we can tell that the overall shape of our reconstructed wave appears to be correct. Individual heights of peaks, however, are likely to not be right. Considering the general noise level, the results are better than expected.

6.3 Limitations

As we have seen, our method enables us to reconstruct water heights for both simulated and real distance data sets. However, there are also clear limits to what our implementation can and cannot do. In this section, we will summarize these limitations, some of which have already been brought up and some of which are new.

Our initial model for the influence of water on the time-of-flight camera signal assumed only a linear influence caused by the slower speed of light in water and the change in direction due to refraction. Our experiment involving increasing water heights demonstrated a need for nonlinear correction of the water distance. Ideally, such a correction would be performed the same way for each pixel. For our measurements, this was not successful, forcing us to use a correction function for each individual pixel.

While this per pixel correction of the water distance improved the reconstruction results, we were unable to reuse the correction for a data set captured on a different day. This suggests that the correction also depends on the viewing direction. Capturing a series of increasing water heights is also very time consuming, as we need to wait for the water to calm after each increase. The dependency between viewing direction and distance correction also implies that the camera must not be moved at all after the initial calibration with the empty container. The same goes for the floor plane and the container itself.

The camera noise also increases the difficulty of the reconstruction. Absorption causes the signal strength to decrease as the water height increases. This in turn also results in a higher noise level for rising water levels. For a reconstructed sequence of a flat water surface, the camera noise results in a kind of background 'wobbling'. Increasing the smoothness can flatten out the surface, but has the same effect on reconstructed waves. The increased noise at higher water levels limits our approach to water surfaces of only a few cm. Depending on the integration time, there is always a point where the signal loss due to absorption causes too many invalid pixels. This could be seen as a hard limit to our range of available water heights, but is of course also camera dependent.

The reconstructed waves are likely to be at incorrect absolute heights due to a combination of these two shortcomings: We had to revert to the linear model to reconstruct the waves, which has proved to be less accurate for the reconstruction of even flat water heights. The smoothing that is always necessary to some degree will also flatten the waves. If we had a more reliable way to evaluate the water height of a wave than our polystyrene floaters, we could figure out an ideal value for the smoothness in combination with a certain linear or nonlinear model. Correcting waves that are too flat by scaling them could also be conceivable. As it is, our experiment and implementation lacks a ground truth measurement for moving water.

Our implementation makes no attempt at correcting motion artifacts. We rely completely on the camera software to mark those artifacts as invalid.

The setup of our experiment was chosen to eliminate as many unwanted reflections as possible. We kept the experiment covered by the infrared absorbing tent and coated the walls of the container in blackboard paint. Our refraction based range image model does not take multiple reflectance into account at all. As we saw during the nonlinear underwater distance correction, despite our precautions, the corners at the top of the image show some signs of reflection from the container walls. These parts of the image are also the parts that respond the worst to our nonlinear correction.

Performance wise, our implementation is still far removed from real time: On a 3 GHz ‘AMD Phenom II X4 960T’ CPU, synthesizing a distance image from a set of heights takes roughly 0.1 s. Depending on the number of iterations performed by the optimization algorithm, this function needs to be called several hundred times during the reconstruction of a single frame. As a result, the reconstruction of a frame can take 20 to 30 s for a typical frame, or even longer if more iterations are required. Reconstructing a sequence that took place during for example 5 s in real time takes several minutes for our implementation. We can save a few iterations by using the previous frame as an initial value for the next, and can speed up the reconstruction slightly by splitting the work on multiple CPU cores. This still does not close the gap towards real time performance sufficiently. However, given that our implementation was mainly intended as a prototype to show the working principle, we are not too surprised by this result. Increasing the performance poses its own challenges, while we were mostly concerned with finding a solution that works correctly.

Chapter 7

Conclusion and Future Work

In this thesis, we have described our analysis-by-synthesis approach towards reconstructing liquid surfaces from distance information obtained by a time-of-flight camera. The basis of our reconstruction is a physics-based model that mimics the effect of water on the distance images. Our model in its most simple form takes into account the change of direction due to refraction and the different speed of light in water. By using our model in conjunction with an optimization algorithm, we can reconstruct the water heights that were responsible for a given distance image.

We carefully designed and built an experiment setup in order to capture data sets of real water surfaces while suppressing as many sources of errors as possible. We used this experiment setup to verify our model on real data. Comparing the synthesized distances with real measured distances lead to a refinement of our model: Our initial assumption of a linear influence on the distance by the water turned out to be false. Instead, correcting the water distance using a nonlinear correction function became necessary for accurate reconstruction of flat water heights.

The implementation of our reconstruction model is a simple proof-of-concept prototype. We focused mostly on getting a working implementation early on to use in conjunction with our experiment setup. The use of a scripting language enabled us to make adjustments quickly and try out various changes to our model. Our implementation can be used to reconstruct both static and dynamic scenes of water. It relies on a large scale nonlinear solver to reconstruct the heights of all pixel simultaneously. We used simulated data sets to verify the stability of our chosen optimization algorithm.

While other approaches rely on multiple cameras to reconstruct the distance indirectly from a tracked checkerboard, our approach has the advantage of using only a single camera that can measure the distance directly. We require no tracking of feature points. Instead, only an initial calibration with the empty container is necessary before water is added. As far as we are aware, our approach is also the first to use a time-of-flight camera for this specific purpose instead of a regular camera.

We evaluated the reconstruction results for flat water heights as well as dynamic waves. For still water, we derived a nonlinear correction function for the underwater distances. This correction improved the reconstruction results of increasing flat water heights. Unfortunately, it seems that the viewing direction has an additional influence on the distances. This prevented us from using the correction on another data set.

For moving water, we reverted back to our non-refined linear model. We demonstrated examples that showcase different types of waves reconstructed with our method. Additionally, we evaluated the shape of a reconstructed wave by means of a polystyrene floater. It appears that while our method does not provide completely accurate waves as of yet, the general shape of the reconstructed waves are at least plausible. We also summarized important shortcomings of our method. These include the limitation to just a few cm of water due to increasing noise.

Overall, we have shown a first proof of concept that it is indeed possible to reconstruct information about a liquid surface like water from the distance measurement of a time-of-flight camera. There is however still room for improvement. For one thing, all of our reconstruction results are based on a specific camera. It would be interesting to see if the results can be reproduced by a completely different time-of-flight camera. This could also show which of the problems are camera-specific, a result of our setup or part of the interaction between water and the light signal. It could also reveal whether or not our camera calibration was precise enough. Cameras using a different wavelength for the emission signal might also be useful for higher water levels.

Our nonlinear correction relies on capturing increasing water heights, but is not reusable on a different data set. To use this correction on a moving surface at a specific height, one could measure flat heights above and below this height at certain intervals. Then, after reducing the water level again, it would be possible to test the nonlinear correction method on moving water.

The dependency of the distance on the viewing direction is another interesting aspect that could be used for improvement. One way would be to incorporate this into the model as a correction based on the distance from the image center, instead of performing the correction for each pixel. It would also be interesting to see the effect of changes in viewing direction and camera position for a fixed surface, moving the camera instead of increasing the water. This could be used to check how the distance changes if the camera is moved closer to the surface while keeping the relative angle between surface and camera the same.

Since noise is a limiting factor of our approach, smoothing of individual frames is one immediately obvious avenue of improving the results. Smoothing could be performed as a preprocessing step in addition to the smoothness constraint we introduced. For time-of-flight cameras, it is possible to smooth the raw images of the correlation function before calculating the distance images. Because motion artifacts can occur while capturing moving water, correcting these artifacts might also improve results.

The reconstruction results in Chapter 6 indicate that our experiment setup itself could also be the cause of some of our problems. Ideally, the walls should be far removed from the camera field of view to prevent multiple reflections. The floor also has to be perfectly flat and uniformly reflective for an ideal setup.

For verifying reconstructed waves, a reliable ground truth measurement is needed. Ultrasonic sensors can be used to measure a point on the surface if positioned a few cm above the water. Such a sensor could fulfill the same function as the polystyrene floater.

An optimization that can perform the reconstruction in real time seems to be far removed at this point. The ability to see the effect of changes to the water surface instantaneously could however prove very valuable when refining the model.

Bibliography

[Din+11] Yuanyuan Ding et al. „Dynamic fluid surface acquisition using a camera array“. In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2478–2485.

[Fol96] J.D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley systems programming series. Addison-Wesley, 1996. ISBN: 9780201848403.

[Fou01] Python Software Foundation. *Python*. [Online; accessed 12 September 2015]. 2001–. URL: `http://www.python.org/`.

[Hah12] Uwe Hahne. „Real-time depth imaging“. PhD thesis. Technical University of Berlin, 2012.

[Har07] Ulrich Harten. *Physik: Einführung für Ingenieure und Naturwissenschaftler*. Springer-Verlag, 2007.

[HDF15] HDFGroup. *HDF Hierarchical Data Format*. [Online; accessed 12 September 2015]. 2015. URL: `http://www.hdfgroup.org/HDF5`.

[Hee15] Dimitri van Heesch. *Doxygen*. [Online; accessed 12 September 2015]. 2015. URL: `http://www.doxygen.org`.

[HQ73] George M Hale and Marvin R Querry. „Optical constants of water in the 200-nm to 200-μm wavelength region“. In: *Applied optics* 12.3 (1973), pp. 555–563.

[Hun07] J. D. Hunter. „Matplotlib: A 2D graphics environment“. In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[Its15] Itseez. *OpenCV: Open source computer vision*. [Online; accessed 12 September 2015]. 2015. URL: `http://www.opencv.org/`.

[J+01] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 12 September 2015]. 2001–. URL: `http://www.scipy.org/`.

[Kit15] Kitware. *CMake*. [Online; accessed 12 September 2015]. 2015. URL: `http://www.cmake.org`.

[Lan00] Robert Lange. „3D time-of-flight distance measurement with custom solid-state image sensors in CMOS/CCD-technology“. PhD thesis. University of Siegen, 2000.

[Lef+13] Damien Lefloch et al. „Technical foundation and calibration methods for time-of-flight cameras“. In: *Time-of-Flight and Depth Imaging. Sensors, Algorithms, and Applications*. Springer, 2013, pp. 3–24.

[Lin+10] Marvin Lindner et al. „Time-of-flight sensor calibration for accurate range sensing“. In: *Computer Vision and Image Understanding* 114.12 (2010), pp. 1318–1328.

[LLT13] Stephen G. Lipson, Henry S. Lipson, and David S. Tannhauser. *Optik*. Springer-Verlag, 2013.

[Sch11] Ingo Schiller. *MIP - MultiCameraCalibration*. [Online; accessed 12 September 2015]. 2011–. URL: `http://www.mip.informatik.uni-kiel.de/tiki-index.php?page=Calibration`.

[Sch94] Christophe Schlick. „An inexpensive BRDF model for physically-based rendering“. In: *Computer graphics forum*. Vol. 13. 3. 1994, pp. 233–246.

[VCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. „The NumPy array: a structure for efficient numerical computation“. In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30.

[Zha00] Zhengyou Zhang. „A flexible new technique for camera calibration“. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22.11 (2000), pp. 1330–1334.