Robust Online Environment Reconstruction with Multiple Cameras for Cobot Collision Avoidance

Meijer, M.F.

Award date:
2020

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Visualization Research Group

Robust Online Environment Reconstruction with Multiple Cameras for Cobot Collision Avoidance

Master's Thesis

M.F. Meijer

Supervisors:
dr. A.C. Jalba
dr. A. Saccon

Coaches:
ir. M.J.R. Bemelmans (VinciTech)

Thesis Committee:
dr. K. Buchin

Eindhoven, Monday 6th July, 2020

Preface

This thesis is the result of the work I did for my final master project of the Computer Science and Engineering program on the Eindhoven University of Technology.

I would like to thank my supervisors A. Jalba and A. Saccon for their invaluable advice, guidance, extensive feedback, and supervision during my master's project. I would also like to thank my coach, M. Bemelmans for his feedback and valuable insights, especially regarding depth cameras and the Cobot platform. Without their feedback on my progress reports and creative ideas, this work would not have been possible. Additionally, I would like to thank K. Buchin for being part of the assessment committee of this thesis. My thanks to B. Blom, during our lunches your insights and our discussions helped me immensely in progressing this work. Finally, I would like to thank my family and friends for their unwavering support despite the fateful events last year. It means everything to me.

M.F. Meijer
Eindhoven, Monday 6th July, 2020

Abstract

Collaborative robots (Cobots) are used in, for instance, the assembly and packaging industry to help workers complete difficult or impossible repetitive tasks. To improve cobots they should be able to complete their tasks without colliding into dynamic objects. To avoid collision the cobot needs to be aware of the environment. In a previous project, an online dynamic environment reconstruction algorithm was developed that takes a point cloud captured by a depth camera as input and uses radial basis function (RBF) interpolation to create an environment description. The cobot controller can use this environment description to avoid static and dynamic objects in the environment.

An issue with this approach is that the captured point cloud contains shadows caused by the objects in the scene. To improve the environment description multiple cameras are introduced. A framework and methodology are presented for the alignment of point clouds with the help of fiducial markers and precise manual refinement. To capture high-quality point clouds the depth noise was reduced by calibrating the camera. Depth noise resulting from stereophotogrammetry algorithm used by the depth cameras and “phantom points”, points with an incorrect depth value, causes problems in the reconstructed environment.

Analysis of the point cloud density showed that to achieve stable computation times of the environment description, the aligned point clouds of multiple cameras can be sampled uniformly by a voxel-grid filter. This stability is demonstrated in an experiment where the point cloud density was changed over time. To speed up the computation time of the environment description the original algorithm is replaced with parallelized counterparts. This improved the computation time by a factor of three. This improvement allows the introduction of more cameras and improves the environment description without loss of efficiency.

Contents

Nomenclature

A list of the variables and notation used in this thesis is defined below. The definitions and conventions set here will be observed throughout unless otherwise stated. For a list of acronyms, please consult page vi.

Point Cloud Alignment

A,B,C,…	Coordinate frames
$\mathbf{p}$	Arbitrary point
${}^{A}\mathbf{p}$	Coordinates of point $\mathbf{p}$ w.r.t. to A
$\mathbf{o}_B$	Origin of coordinate frame B
${}^{A}\mathbf{o}_B$	Coordinates of the origin $\mathbf{o}_B$ w.r.t. to coordinate frame A
${}^{A}\mathbf{R}_B$	Orientation (as a 3×3 rotation matrix) of the origin $\mathbf{o}_B$ w.r.t. to coordinate frame A
${}^{A}\mathbf{H}_B$	Homogeneous transformation from coordinate frame B to A

CSRBF variables

ω	Weight of a RBF center
ϕ	Radial basis function
σ	CSRBF support radius.
c	Arbitrary RBF center
f	Implicit function approximated by n interpolants
g	Constraint value assigned to a center
h	Saturation value of the CSRBF

Other Symbols

k	Condition number of a matrix

List of Abbreviations

6DoF	Six Degrees of Freedom
ArUco	Augmented Reality University of Córdoba
BCSR	Block Compressed Sparse Row
CG	Conjugate Gradient
Cobot	Collaborative Robot
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CSRBF	Compactly Supported Radial Basis Function
FOV	Field of View
GPU	Graphics Processing Unit
ICP	Iterative Closest Point
IR	Infrared
LASER	light Amplification by Stimulated Emission of Radiation
LED	Light Emitting Diode
LIDAR	Laser Imaging, Detection, and Ranging
PaLinSo	Parallel Linear Solver
RADAR	Radio Detection and Ranging
RAM	Random Access Memory
RBF	Radial Basis Function
RMSE	Root Mean Square Error
ROI	Region of Interest
SD	Standard Deviation
SDF	Signed Distance Function
SDK	Software Development Kit
SONAR	Sound Navigation Ranging
SPD	Symmetric Positive Definite
ToF	Time of Flight
TSDF	Truncated Signed Distance Function
UML	Universal Modelling Language
VRAM	Video Random Access Memory

Chapter 1

Introduction

1.1 Research Setting

The use of robots in industrial production has been growing since, in 1961, the first industrial robot, the Unimate[1], was placed in a General Motors factory to perform metalworking and welding tasks. Because industrial robots are heavy and work at high velocities they are dangerous and they are often placed in cages to protect the persons working around them. An example is shown in the left of Figure 1.1 where workers are protected from industrial bread palletizing robots by fences. The motion of these robots is halted when the fences are opened.

Figure 1.1: *Left:* Caged industrial robots palletizing bread, *Right:* A lightweight cobot produced by VinciTech.

There are still tasks in industrial production lines that can not be automated and thus have to be performed by human workers. Parts of these task can be repetitive or cumbersome and could be performed by a robot. For this purpose two professors from Northwestern University, J. E. Colgate

[1] `https://robots.ieee.org/robots/unimate/`

and M. Peshkin, invented the Collaborative Robot or Cobot in 1996[1]. They designed a cobot that is able to assist factory workers with placing a heavy machine part in limited maneuvering space at a truck assembly plant of General Motors. Nowadays several companies are developing cobots and one such company is VinciTech[2]. They are developing a lightweight modular robotic platform that can be used in collaboration with human workers. A configuration of the platform can be found in Figure 1.1 on the right.

The first cobots were not autonomous and required human interaction to complete their task. To improve the cobot it should operate autonomously and collaborate with human workers when required. The environment of a cobot is unstructured and consists of dynamic and static obstacles. For a cobot to safely collaborate with humans it should never hit any obstacle in the environment, especially humans. Ideally the cobot should be able to complete the task it is assigned while avoiding these obstacles, unless interaction with the obstacle is required. To accomplish this the cobot needs a method to capture the surrounding environment. A method to capture a dynamic environment is with depth cameras that can generate a three-dimensional point cloud of the environment many times per second. A point cloud is a set of three-dimensional points in space.

A point cloud of the environment at a given time instance is not enough for the controller of a cobot to avoid dynamic obstacles in the path of the task it is currently performing. To give the cobot controller optimal flexibility it needs a continuous surface representation to avoid collision and plan ahead. There are many methods to generate a continuous surface from a point cloud [1]. Since the cobot must be able to avoid dynamic obstacles the surface reconstruction must be updated constantly. The reconstructed surface should also be free of high frequency and high amplitude noise because that can cause the cobot to behave unpredictable when it is close to that surface. This unpredictability should be avoided because it will cause a negative perception of the cobot by the persons working with it. In previous work, done by T. Bosch [2] for the Dynamics and Control group of the TU/e in collaboration with the company VinciTech, an online algorithm is described that generates such a continuous surface representation from a point cloud. This work will build on further on the previous work which is summarized in the next section.

1.2 Previous Work

In the master project of T. Bosch [2] an online dynamic surface reconstruction pipeline has been developed. The controller strategy developed as part of the master's project of S. Driessen [3] then uses the surface description as input for the obstacle avoidance tasks in the cobot controller. The controller strategy has been formulated as a quadratic program of the avoidance task, a reach task, and other necessary constraints. To avoid obstacles while completing its task the quadratic

[1] https://patents.google.com/patent/US5952796
[2] http://www.vincitech.nl/

program is solved for every control iteration and the solution is the next control command for the cobot.

1.2.1 Processing Pipeline

The thesis of T. Bosch [2] describes the surface reconstruction algorithm as a processing pipeline that consists of two parts. After the acquisition of a point cloud with a depth camera the first part of the pipeline reconstructs the surface. The surface is reconstructed with radial basis function (RBF) interpolation. In the second part the reconstructed surface is visualized. The visualization is not necessary for the cobot controller but is used for validation of the correctness of the reconstructed surface. An activity diagram[1] of this pipeline can be found in Figure 1.2. The activities responsible for the surface reconstruction are tinted white while the activities responsible for visualization are tinted grey.

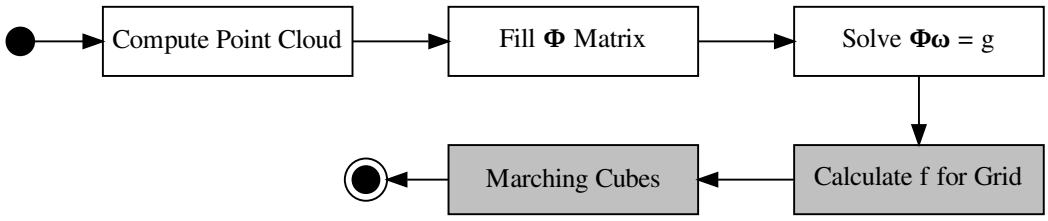

Figure 1.2: Activity diagram of the processing pipeline from [2]. The white steps are responsible for surface reconstruction, the grey steps are used for visualization of the surface.

This pipeline consists of five sequential steps which have the following responsibilities.

1. *Compute point cloud*: Part of surface reconstruction, align all points in the point cloud to the cobot frame and generate the off-surface points.
2. *Fill Φ matrix*: Part of surface reconstruction, build the linear system of equations $\Phi\omega = g$
3. *Solve $\Phi\omega = g$*: Part of surface reconstruction, Solve the linear system of equations, find ω
4. *Calculate f for grid*: Part of surface visualization, evaluate f for all positions of a three-dimensional regular grid.
5. *Marching Cubes*: Part of surface visualization, generate a surface mesh from the grid positions with the marching cubes algorithm.

In this work the focus will be on the three steps responsible for the surface reconstruction. The steps used for the visualization of the reconstructed surface are not necessary for cobot control and are used unmodified. The next sections will explain the three steps in more detail.

[1] The unified modelling language (UML) activity diagram is a specialization of the flowchart that offers basic support for modelling concurrency with the split and join nodes. This feature will be used in later chapters where parallel tasks are discussed. See `https://en.wikipedia.org/wiki/Activity_diagram`.

1.2.2 Point Cloud Acquisition and Off-surface Point Generation

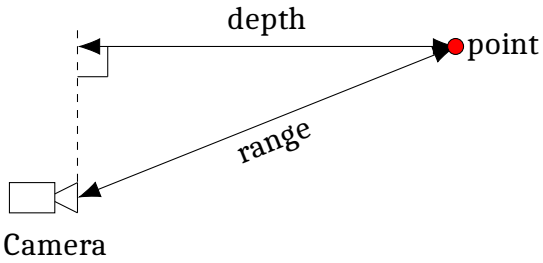

Figure 1.3: The range is the distance between a point (red) and the camera center. The depth on the other hand is perpendicular projection of the point on view plane of the camera.

First a distance or range image is acquired with a depth camera. The camera used in [2] uses active stereophotogrammetry to estimates the distance of each point and then calculates a depth image from the range image. In Figure 1.3 the difference between distance or range and depth is illustrated. The calculated depth image is filtered to reduce the number of depth pixels. From the depth image a point cloud is calculated which consists of points in three-dimensional space. In Figure 1.4 the transition of a simple scene from image to point cloud is shown.

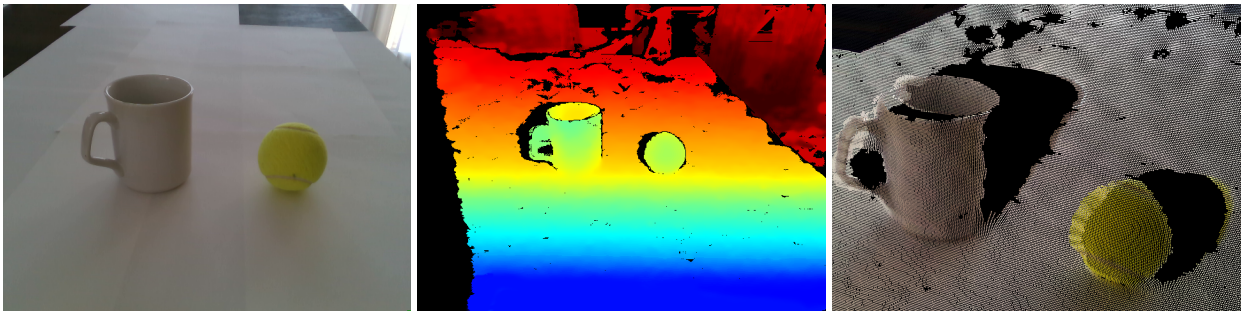

Figure 1.4: *Left:* A color frame of a scene with a coffee mug and a tennis ball captured with an Intel RealSense D435 depth camera. *Middle:* The depth image generated from left and right images captured with an Intel RealSense depth camera. *Right:* A render of the point cloud calculated from the depth image.

The surface will be reconstructed from the points of the point cloud. However because the point cloud will only contain points on the side of an object that is visible to the camera it is not possible to properly reconstruct the object. As is clear in the rightmost image in Figure 1.4 the surface of the coffee mug and tennis ball has no thickness, instead there is a shadow behind the surface of these objects with no points. To deal with this problem the thickness of the surface is simulated by placing points on the camera rays at a constant distance d behind the surface points. These surface and off-surface points are then used as the support points to reconstruct the environment surface. A two dimensional example of the placement of off-surface points is shown in Figure 1.5.

It is clear that this approach has its drawbacks. The surfaces are not accurately represented, planning a path behind the surface may still cause collision with an object when it is substantially

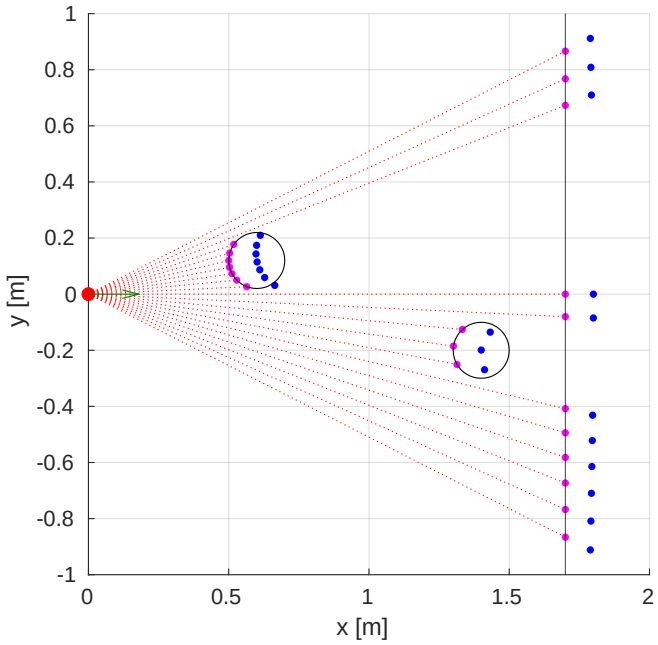

Figure 1.5: A 2D point cloud of 2 circular obstacles and a wall sampled by a depth camera, denoted by the red dot at $(0, 0)$, with the green arrow representing the view direction. The magenta points are surface points sampled by the camera, the blue points are off-surface points placed on the camera rays behind the surface points.

thicker than the assigned thickness. Another issue is that the cobot controller might plan a path through an object that is in the shadow cast by objects in front of it. In Figure 1.5, for instance the circles casts a shadow on the wall and the cobot could thus collide with the wall.

The controller strategy used in [3] represents the destination targets and obstacles in the environment as artificial potential fields. The destination target is defined as an attractive point while the surface of the environment is defined as a repulsive [4]. In Figure 1.6 the principle of the potential field method is illustrated.

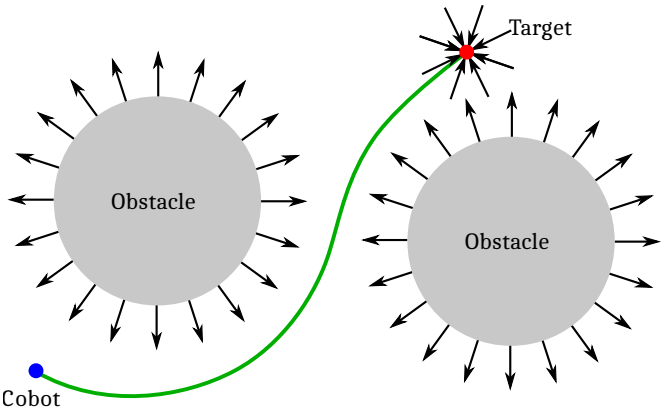

Figure 1.6: Artificial potential field method: The cobot (blue) moves to the target (red) because of the attractive velocity (arrows) it is assigned. Obstacles are assigned repulsive velocity and the cobot follows a path around the obstacles (green) instead of a direct path.

1.2.3 Surface Reconstruction with Radial Basis Function Interpolation

Distance to Surface. A way to have the environment surface act as a repulsive surface is by determining the shortest distance to the surface at some point p in the environment. The shortest distance can then be used to calculate the repulsive acceleration in p. For p the distance function $DF(p, S)$ is defined as the shortest distance to a surface S. If S is a closed surface the signed distance function $SDF(p, S)$ can be used

$$SDF(p, S) = \begin{cases} DF(p, S) & \text{outside } S \\ -DF(p, S) & \text{inside } S \end{cases} \tag{1.1}$$

The $SDF(p, S)$ is defined to have the value $DF(p, S)$ outside, and the value $-DF(p, S)$ inside the surface. Because a point does not need to be affected by distant objects the truncated distance function $TSDF(p, S)$ is used where the value saturates to a constant when is $DF(p, S)$ larger than a threshold value h, the definition is as follows

$$TSDF(p, S) = \begin{cases} -h & SDF(p, S) < -h \\ SDF(p, S) & -h \leq x \leq h \\ h & SDF(p, S) > h \end{cases} \tag{1.2}$$

The threshold value h is named the saturation value. A plot of the SDF and the TSDF of the unit circle are given in Figure 1.7.

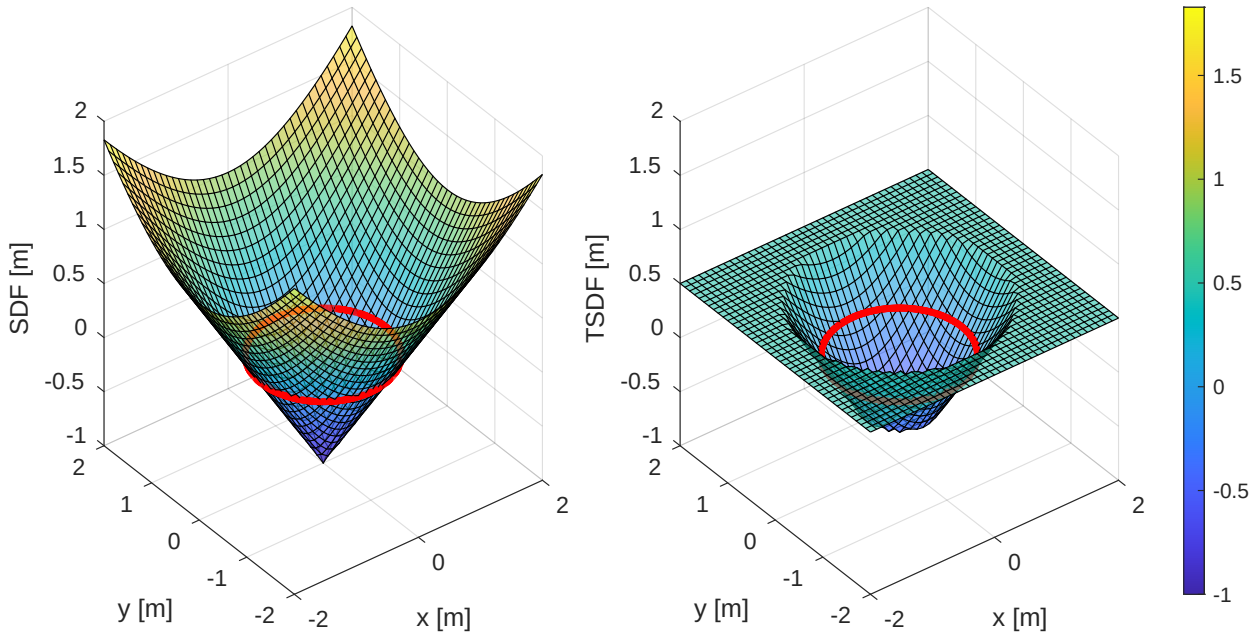

Figure 1.7: *Left:* The value of the signed distance function $SDF(x, y)$ of an unit circle centered around the origin. *Right:* The same plot but for the value of the truncated signed distance function $TSDF(x, y)$ and $h = 0.5$.

Implicit Surface Representation. When starting with the point cloud captured by the camera, S is actually not yet known. Instead, there is only the set C of surface and off-surface points. A surface needs to be reconstructed from C. To have a surface definition that resembles the behavior of the TSDF the surface can be defined as an implicit function. An implicit function has the form

$$f(x, y, z) = 0, \tag{1.3}$$

when the implicit function is 0 we are on the surface, inside the surface the value is a function of $-DF(p, S)$ and outside a function of $DF(p, S)$. There are different iso-surfaces or level-sets. For instance $f = 1$ describes the surface where the distance is a function of $f + 1$. A demonstration of this is Figure 1.8 where different level-sets of the implicit function of the unit circle $\sqrt{x^2 + x^2} - 1 = 0$ are shown.

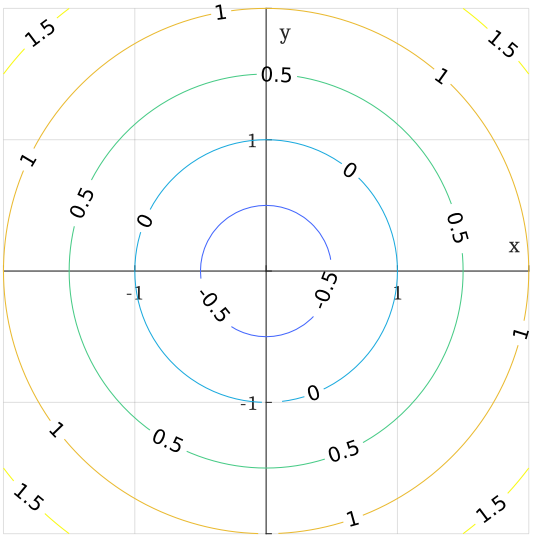

Figure 1.8: The level-sets $f = \{-0.5, 0, 0.5, 1, 1.5\}$ for the implicit function $\sqrt{x^2 + x^2} - 1 = 0$.

Radial Basis Function Interpolation. There are many ways to create an implicit surface from the point cloud C, reference [1] gives an exhaustive overview. The way to create an implicit surface from the point cloud that was used in [2] is RBF interpolation. In RBF interpolation the implicit function at an arbitrary point p in space is defined by the weighted sum of all interpolants of all centers $c_i \in C$,

$$f(p) = \sum_{i=1}^{n} \omega_i \phi(||p - c_i||), \tag{1.4}$$

with ω_i the weight of the interpolant and $\phi(||p - c_i||)$ a radial basis function of the distance between p and c_i. A radial basis function can be any radial function like, for instance, the Gaussian or the bump function. When the radial function is smooth the result of RBF interpolation is a smooth surface. This is required because otherwise the cobot controller would generate an erratic unpredictable path which is undesirable behaviour when collaboration with humans is required.

Wendland RBFs. Because the controller uses the potential field method the second order derivative of the surface is used to define the repulsive acceleration. Therefore the radial basis function that is chosen should at least be in C^2, the set of twice differentiable functions. Another requirement is that the radial function is of compact support, i.e. it only has a value not equal to h in a small radius around c. This is because the intention is to have a ϕ that it resembles the TSDF. The category of radial basis functions that satisfy the last requirement are the Wendland compactly supported radial basis functions (CSRBFs)[5]. The specific Wendland CSRBF that is C^2 smooth in $\mathbb{R}^3$ is $\phi_{3,1}$ and it is defined by the following piece-wise polynomial function

$$\phi_{3,1} = \begin{cases} (1-r)^4(4r+1) & \text{if } r \leq 1, \\ 0 & \text{otherwise} \end{cases} \tag{1.5}$$

with $r = ||p - c_i||$, i.e. the distance between p and interpolation center c_i. A plot of $\phi_{3,1}$ is shown in Figure 1.9.

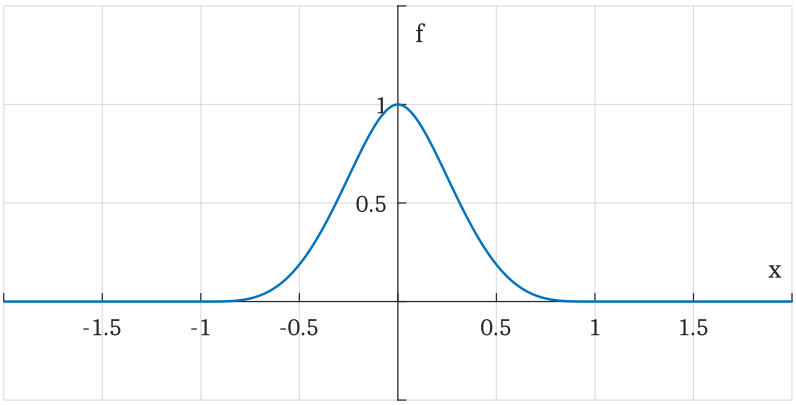

Figure 1.9: A plot of the Wendland piece-wise polynomial function $\phi_{3,1}$.

The Wendland RBF $\phi_{3,1}$ is of compact support because it is only non-zero between $(-1, 1)$, however we want this to be within $(-\sigma, \sigma)$ where the variable σ is the support radius of the CSRBF. To accomplish this r is replaced by $\frac{r}{\sigma}$ in the polynomial part of 1.5 as follows

$$(1 - \frac{r}{\sigma})^4(4\frac{r}{\sigma} + 1). \tag{1.6}$$

With the ϕ part of 1.4 known only the weights ω have to be found. The weights are found by solving the system of linear equations

$$g(c_i) = \sum_{j=1}^{n} \omega_i \phi(r_{ij}), i \in \{1, 2, \ldots, n\} \tag{1.7}$$

for all centers c_i and $r_{ij} = ||c_i - c_j||$. The value $g(c_i)$, which we will name the constraint for c_i, is the value that f will have in c_i. In Figure 1.10 the effect of different constraint values for centers is demonstrated. For two one-dimensional centers, c_1 at (0) and c_2 at (0.7) the constraint value is

set to $g(c_1) = 1$ respectively $g(c_2) = 1.5$. After solving the system of linear equations in 1.7 for the weights the function f in 1.4 is evaluated for $x = [-1, 2]$.

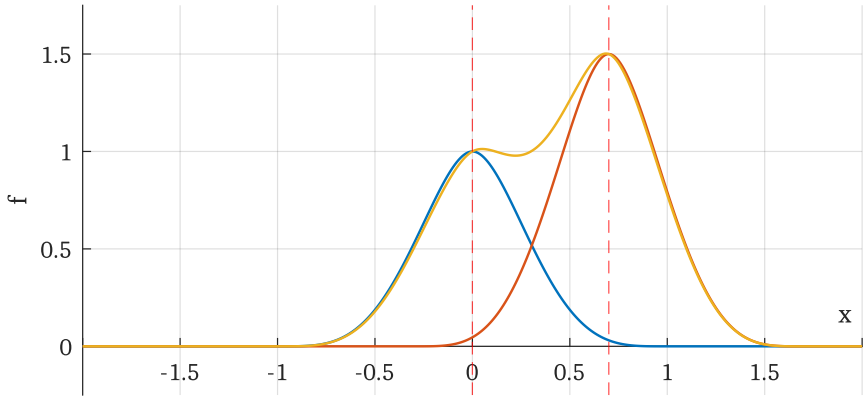

Figure 1.10: A plot of two Wendland CSRBFs, one at $c_1 = (0)$ (blue) and the other at $c_2 = (0.7)$ (orange) with $g(c_1) = 1$, $g(c_2) = 1.5$, $\sigma = 1$. The resulting value for $f(p)$ is shown in yellow.

Two more notations that will be used for 1.7 are the compact notation

$$g = \omega\Phi \tag{1.8}$$

and the matrix notation which is written as

$$\begin{bmatrix} \phi(r_{11}) & \phi(r_{12}) & \dots & \phi(r_{1n}) \\ \phi(r_{21}) & \phi(r_{22}) & \dots & \phi(r_{2n}) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(r_{n1}) & \phi(r_{n2}) & \dots & \phi(r_{nn}) \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_n \end{bmatrix} = \begin{bmatrix} g(c_1) \\ g(c_2) \\ \vdots \\ g(c_n) \end{bmatrix}. \tag{1.9}$$

Emulating the TSDF Shape. To illustrate how the Wendland CSRBF is changed to resemble a TSDF all parameters that are used to modify Wendland $\phi_{3,1}$ are illustrated in Figure 1.11.

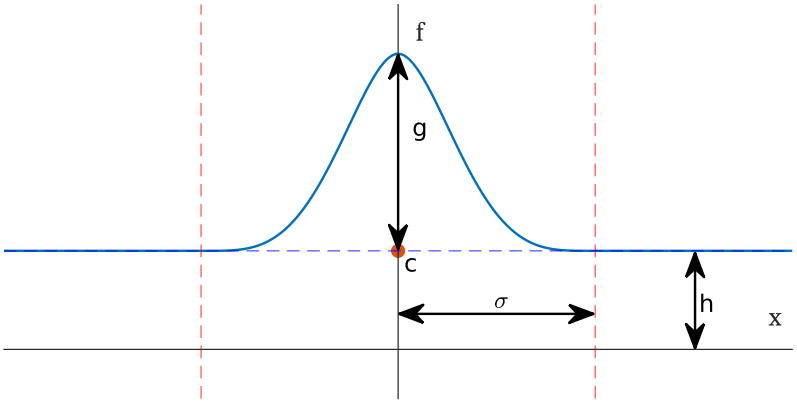

Figure 1.11: Parameters used to modify the Wendland CSRBF $\phi_{3,1}$.

To reiterate, for a center c the support radius σ controls the width, the saturation value h the horizontal offset outside the domain $(-\sigma, \sigma)$, and the constraint g is the height of the radial basis func-

tion.

What are the values chosen for the parameters of the CSRBF by [2] so that it resembles a TSDF? A list of the chosen values can be found in Table 1.1.

Parameter	Description	value
d	Distance of off-surface points	0.07 m
σ	Support radius of CSRBF	0.15 m
h	Saturation value of CSRBF	0.07
g_s	Constraint for surface points	-0.07
g_o	Constraint for off-surface points	-0.07
f_s	Surface level-set	0.01

Table 1.1: Parameters chosen for the CSRBF so it resembles a TSDF.

The value of $g = 0$ is not an option because then we have the trivial solution $\omega = 0$ and f will evaluate to 0 everywhere. For the surface and off-surface points a constraint value of $-h$ was chosen where h is the saturation value as in 1.2. This is done so the level-set $f = 0$ represents the surface. Off-surface points were placed at a distance of $0.07\,\mathrm{m}$ and the support radius was set to $0.07\,\mathrm{m}$. A plot of these constraints for the surface center c_s at (0) and the off-surface center c_o at (0.07) can be found in Figure 1.12.

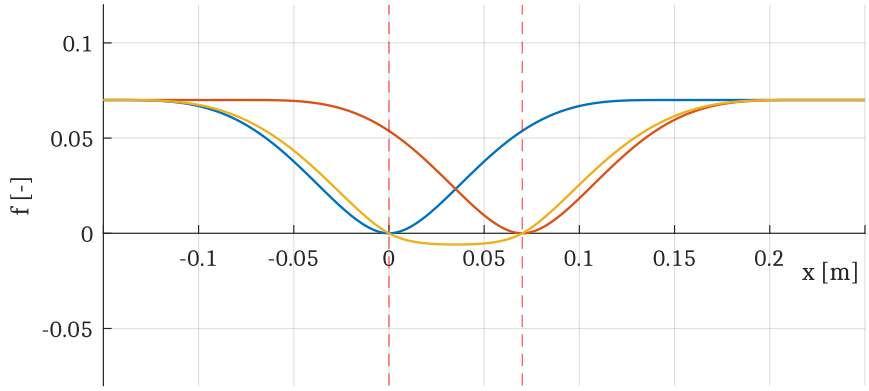

Figure 1.12: The plot of the centers c_s at (0) in blue and c_o at (0.07) in orange located at the dashed red vertical lines, and constraints $g(c_1) = -h$, $g(c_2) = -h$ with $\sigma = 0.15$. The resulting f is shown in yellow.

Potential Field from Surface. It is important that $f = 0$ represents the surface of the environment that is constructed because the repulsive force can then be calculated from the potential field U at a point p by taking the negative logarithm of $f(p)$:

$$U(p) = -\log(f(p)) \tag{1.10}$$

and to determine the direction of acceleration the controller uses the gradient of the potential field

$$\ddot{p} = -\nabla(U(p)) \tag{1.11}$$

Since the value of the negative logarithm goes to infinity at zero the force rapidly increases when the cobot moves closer to the surface effectively repelling the cobot, as seen in Figure 1.13.

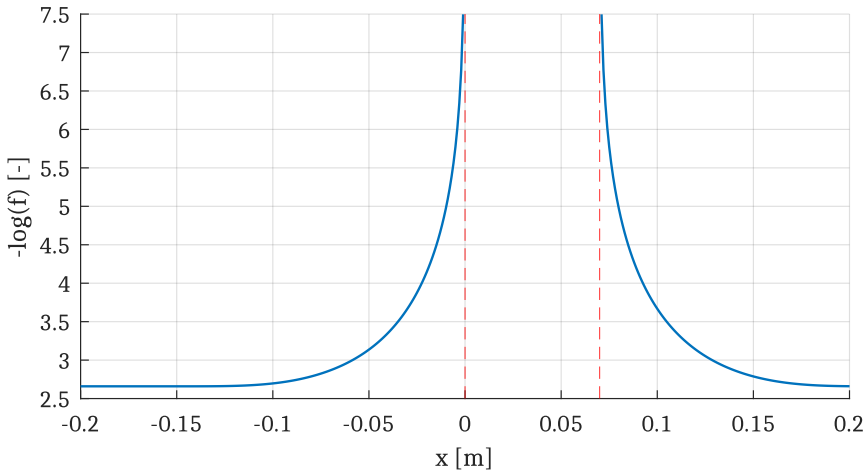

Figure 1.13: A plot of $U(p)$ for the f from Figure 1.12.

In the next example the off-surface centers are placed at $d = 0.13$. The saturation value h is set to d, the surface and off-surface centers are assigned $g(c) = -h$. The system of linear equations was solved for ω and the value of f was evaluated for the points in a two-dimensional grid and plotted together with the iso-surface lines for different values of f in Figure 1.14.

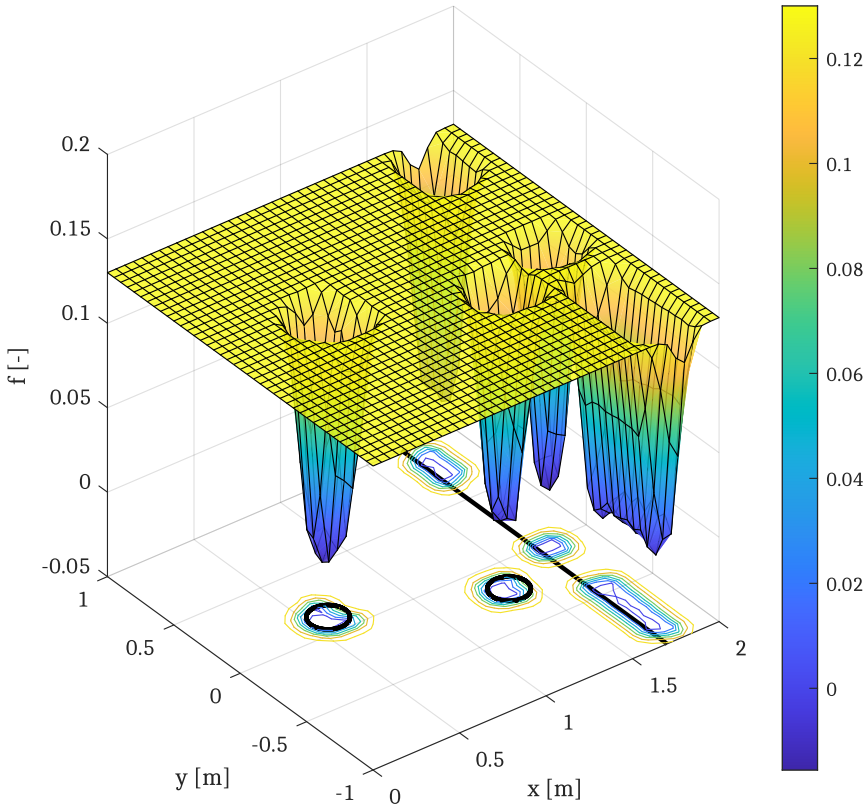

Figure 1.14: Evaluation of f on a grid points for the surface and off-surface centers acquired in figure 1.5. The circles and wall are visualized by the black lines.

In the last example the $f = 0.01\,\mathrm{m}$ level-set is reconstructed from the three-dimensional point cloud of a box with a soccer ball on top. The image can be found in Figure 1.15 and is rendered with the original pipeline. To make the visualization of the three-dimensional iso-surface, f is evaluated for a regular 3D grid. The grid points are spaced $0.04\,\mathrm{m}$ apart in x, y, and z direction. The values at the grid were used to generate an iso-surface mesh with the marching-cubes algorithm[6].

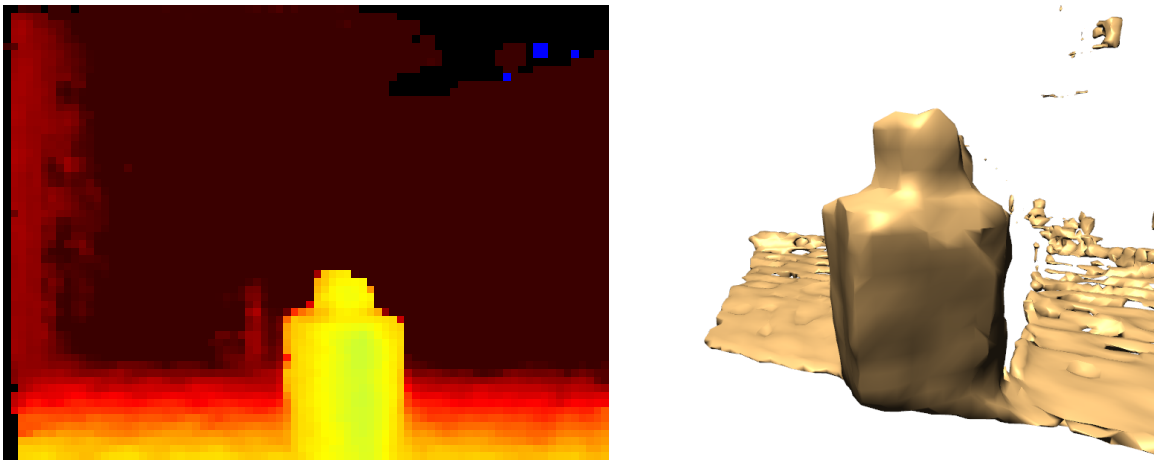

Figure 1.15: Scene of a soccer ball on a box captured by a depth camera. *Left:* depth image as captured by the camera. The dataset details are in D. *Right:* Iso-surface mesh of the $0.01\,\mathrm{m}$ level-set of f of the point cloud of the scene in the original pipeline with $d = 0.13\,\mathrm{m}$.

1.2.4 Computational Complexity of the Pipeline

In the development of the original pipeline substantial effort was put into the reduction of the computational complexity. The building of $\Phi\omega = g$ initially took $O(n_c^2)$ time with n_c the number of centers in the point cloud. Solving the system of linear equations with a direct solver took $O(n_c^3)$ time. The computational complexity was reduced to the numbers found in Table 1.2. The numbers refer to the first three tasks defined in Section 1.2.1.

Task	Name	Computational Complexity
1	Compute point cloud	$O(n_c)$
2	Fill Φ matrix	$\approx O(n_c)$
3	Solve $\Phi\omega = g$	$O(n_{nz}\sqrt{k})$

Table 1.2: Computational complexity of steps responsible for the surface reconstruction in the original pipeline from [2].

In the second step, where the system of linear equations is built, almost all effort is spent in the creation of the Φ matrix. As seen in the representation of Φ in 1.9 the matrix has $n_c \times n_c$ elements and to calculate all those elements the complexity is indeed $O(n_c^2)$. However since radial basis functions of *compact support* are used only the ϕ values for centers within the support radius σ

have to be added to Φ. The rest is of the elements of Φ are 0, i.e., the resulting matrix is very sparse. To quickly find the centers within a radius of σ around a center all the centers of C are added to a regular voxel-grid data-structure. The size of the voxels is $\sigma \times \sigma \times \sigma$. By applying spatial hashing to the center position of the voxel all neighboring voxels can be found in linear time. As long as the number of centers in the neighboring voxels is low, the average computational complexity approaches $O(n_c)$.

To improve the solving of the system of linear equations the transition was made from a direct solver to an iterative solver. The advantage of iterative solvers is that the number of iterations and the error can be set as a stop condition. When the the desired error or number of iterations is reached the solver stops. This is more efficient then a direct solver because the solution is approximated. Since the Φ matrix is symmetric and positive definite (SPD) a conjugate gradient (CG) solver can be used which is faster than a regular iterative solver. The computational complexity of the CG solver is $O(n_{nz}\sqrt{k})$ where n_{nz} is the number of non-zeros in the matrix and k the condition number of the matrix.

1.3 Research Objectives

In the original pipeline only a single depth camera is used to capture the point cloud. This inevitably causes shadows in the captured point cloud as was shown in Figures 1.4 and 1.5. These shadows can be problematic because as the surface shape is not known there and the cobot controller might still calculate a path where a collision happens. Another issue is that in the original pipeline reconstruction speed is around $10\,\mathrm{Hz}$ for point clouds with ≈ 13000 centers. Unfortunately the point cloud was not recorded in [2] and the point cloud can therefore not be used for future bench-marking. The original pipeline is also not robust, it will grind to a halt or fail when a subject gets too close, less than approximately $1\,\mathrm{m}$ to the camera.
To address these issues the following research objectives are formulated:

1. Create a framework that allows multiple aligned cameras to reduce the problem caused by shadows in the point clouds of individual cameras. The more cameras that capture the environment from different positions, the more accurate the environment reconstruction can be.

2. Improve the reconstruction speed of the pipeline to facilitate the addition of multiple cameras. The improvement in efficiency should be such that at least one but preferable more cameras can be added without penalty in efficiency. The new pipeline should be robust.

1.4 Contributions

In this work a framework is proposed that can quickly align the point clouds captured by multiple cameras with an automatic step with the help of fiducial markers and a manual step where precise refinement of the alignment is possible in six degrees of freedom. The alignments can be made persistent and reused in later experiments.

A new, more efficient, pipeline is introduced that allows multiple aligned cameras. The new pipeline consists of six steps. A top-level activity diagram of the new pipeline is shown in Figure 1.16.

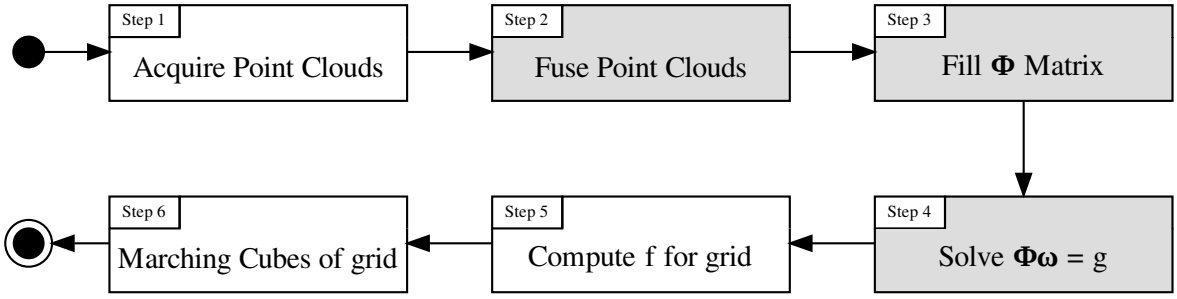

Figure 1.16: An activity diagram of the new processing pipeline. The grey steps, i.e. step 2, 3, and 4 are the main focus of this thesis.

Detailed activity diagrams will be used in Chapter 3 to illustrate where concurrency is applied to increase the efficiency. The steps in Figure 1.16 have the following description

1. Acquire the point cloud of multiple depth cameras.
2. Align and fuse the point clouds, apply the voxel-grid filter to the fused point cloud.
3. Compose the system of linear equations $g = \omega\Phi$.
4. Solve the system of linear equations with a parallel CG solver.
5. Evaluate f for all points in a grid.
6. Compute the iso-surface for the desired value with marching cubes.

The new pipeline can be used to capture and post-process the point clouds of multiple cameras in parallel with a latency of at most one frame between the frames captured by all cameras in parallel.

An analysis is done of the effect of fusing multiple point clouds on the point cloud density and the number of non-zeros that end up in the Φ matrix. The number of non-zeros is kept below an upper limit by applying a voxel-grid filter which can guarantee a certain level of uniformity of the point cloud. Because there were no opportunities to reduce the computational complexity of the steps in the pipeline further the approach is to increase the efficiency of the new pipeline by introducing parallel processing.

A specialized algorithm was developed to fill a sparse matrix concurrently. This method reduced the computation time per frame. Another improvement was to introduce an efficient parallel CG solver which further reduced the computation time. For a recorded reference dataset the new

pipeline is more than 3 times faster than the original pipeline. The new pipeline would also not halt when a subject was close to the camera, the frame rate is stable and an upper bound of the frame rate can be guaranteed. In Chapter 3 each of the steps will be discussed in detail.

1.5 Report Outline

To make this thesis more accessible the major research topics, the alignment of point clouds and the efficient surface reconstruction are assigned to separate chapters. These topics can be discussed independently. The alignment of point clouds is not part of the pipeline, it is a step that has to be done before the pipeline is employed. The alignment transformations are input for the pipeline. This topic is discussed in Chapter 2. The improvements to the original pipeline are discussed in Chapter 3. These chapters in itself follow the standard IMRaD, i.e. Introduction Method Results and Discussion, format for scientific reports. When writing these sections the guidelines from [7] were followed. The thesis ends with a brief conclusion chapter, Chapter 4, where there is reflection on the whole work in total.

Chapter 2

Alignment of Point Clouds

2.1 Literature Review, Concepts, and Models

2.1.1 RGBD Cameras

There are several types cameras available that can capture the environment in three dimensions. A summary of available models, their methodology and an evaluation of their accuracy can be found in [8]. The three methods used to capture environments in real-time are, *time-of-flight*, *active stereoscopy*, and *structured-light* cameras. The output of all three type of devices is a depth image which can be used to generate a point cloud. In Table 2.1 the accuracy of the depth measurement, the robustness of the camera in different types of environment (textured vs. untextered, low-light vs. well-lit etc.), and the ability to use multiple cameras in parallel to capture the same scene are listed.

Type	Accuracy	Environment Robustness	Multiple Cameras
Time-of-Flight	+	+	-
Active Stereoscopy	-	-	+
Structured Light	0	-	-

Table 2.1: The depth accuracy, robustness under different environmental conditions, and ability to use multiple cameras in parallel. Legend: + = good, 0 = normal, - = poor

Time-of-flight (ToF) cameras, such as the Microsoft Kinect v2, operate on the same principle as RADAR and SONAR, but use light instead of radio waves or sound to sample the environment. For each point where the distance is sampled an infrared LED or LASER projects light on the environment and the time is measured for the light to return to an infrared sensor in the camera. Since the speed of light is constant, the distance of each sampled point to the sensor can be calculated. ToF cameras are accurate and work well in a wide variety of environmental settings. Synchronous capture is not possible because of interference of the lasers but alternating capture is possible.

Cameras based on the principle of stereoscopy on the other hand use the same concept as the human eyes and brain to determine the distance of objects in the environment. Stereoscopy cameras

have two identical cameras positioned at a known distance facing the same direction as can be seen in Figure 2.1. Because the position of the left and right camera, C_l and C_r, the focal distance f of the camera, the distance between the cameras d_c, and the projections $p'_{l,r}$ of a point p on each $P_{l,r}$ are known, the three-dimensional position of p can be calculated with triangulation.

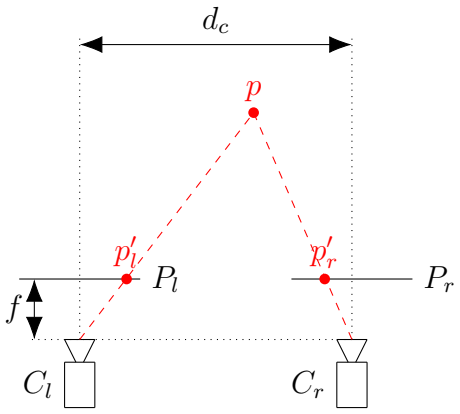

Figure 2.1: The principle of depth calculation with stereoscopy.

The disparity between p'_l and p'_r in the view planes P_l and P_r is determined with a correlation algorithm. The process of correlating patterns in the captured left and right frames is called stereophotogrammetry. Because parts of the environment can have the same color and have no discernible pattern some cameras project an invisible infrared pattern on the environment to help with the stereophotogrammetry. This method is called active stereoscopy. The error of active stereoscopy cameras grows faster than linear with the increase of distance and they have problems with untextured and poorly lit environments.

In structured-light cameras, like the Microsoft Kinect v1, one of the cameras as in Figure 2.1 is replaced by a laser that projects a structured infrared light pattern on the environment. Because the projection direction of the structured pattern is known beforehand the camera can triangulate based on the observed deformation of the pattern and generate a depth field with color information from the captured image. Structured-light cameras suffer of the same interference issues as ToF cameras and have the environment and accuracy issues of stereoscopy cameras.

These three techniques all have one limitation in common, they can only sample the distance at points in the environment that are visible to the camera. There is no information of what is behind these points. These unsampled parts of the environment are referred to as shadows and in the rightmost picture of Figure 1.4 the shadow behind the coffee mug and tennis ball is clearly visible. To reduce the issue caused by shadows multiple cameras are placed at different angles and their point clouds are fused. It is important that all point clouds that are fused are captured as accurately as possible with minimal distortion. The depth cameras are therefore calibrated before being used to capture point clouds.

2.1.2 Pose Estimation with Fiducial Markers

The estimation of a 3D pose is the process of estimating the position and orientation of an object from a 2D image captured by the camera and is a technique widely used in robotics and computer vision. If the exact dimensions of the object are known it is possible to estimate the pose from a 2D image by identifying corresponding points of the object in the image. While pose estimation can be done with just the point correspondences pose estimation can be made fast and more reliable with the help of fiducial markers. Fiducial markers have been successfully used to align point clouds acquired by multiple depth cameras [9], [10].

Several types of fiducial markers have been developed but the state of the art fiducial marker is the ArUco fiducial marker [11], [12]. The ArUco marker is a square marker that contains a unique binary code that can be used to identify the marker when multiple markers are present in the scene. When the corners of the marker are known the position of the center can be calculated and because the binary code is asymmetric the orientation orientation of the marker is also known. The pose estimation of the ArUco is fast and is used in several real-time augmented reality applications. ArUco markers can be detected with the freely available OpenCV library[1]. An example of an ArUco marker and its detection on a captured image can be found in Figure 2.2.

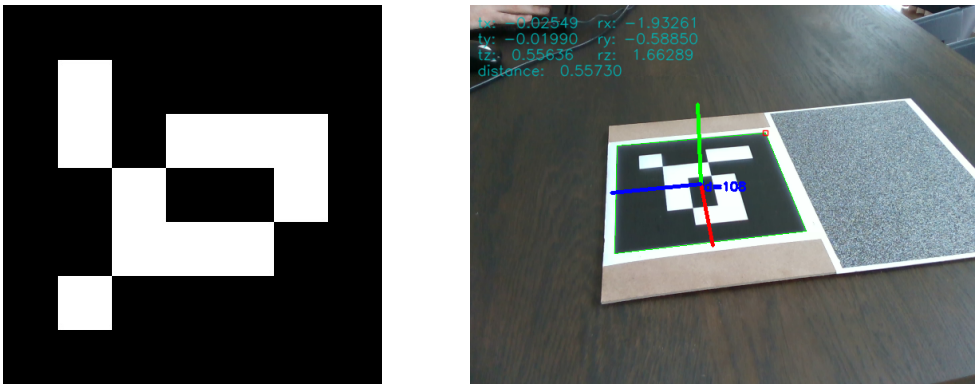

Figure 2.2: *Left:* An ArUco marker with a binary code that encodes for the identification number 108. *Right:* Pose estimation of the ArUco marker with pose translation and rotation vectors and pose frame shown on the captured image.

Because a camera is an imperfect optical instrument the captured image will be distorted. To accurately estimate the pose from a captured image the distortion caused by the camera needs to be known. This distortion is represented by the camera intrinsic matrix K which can be used to transform a point in the captured image to its undistorted counterpart, see [13] chapter 6. The camera intrinsic matrix

$$K = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

[1] https://docs.opencv.org/trunk/d5/dae/tutorial_aruco_detection.html

is composed of the following variables

- α_x and α_y are the focal length of the camera lens in the x- and y-direction expressed in pixels.
- $(x_0, y_0)^T$ is the optical center of the sensor (i.e. the principal point) expressed in pixels.
- s is the skew.

The camera intrinsic matrix can be found by pointing the camera at a checkered pattern of exactly known dimensions. In the case of the camera used in the experiments, the RealSense D435, this is not necessary because the RealSense library provides an interface that returns the camera intrinsic matrix.

Define the world frame as A and the camera frame as C, both frames in $\mathbb{R}^3$, then the relation between a 2D point q with pixel coordinate (u, v) in the camera plane and a 3D point p in the camera coordinate system is

$$\begin{bmatrix} q_u \\ q_v \\ 1 \end{bmatrix} = K \cdot \begin{bmatrix} {}^C\mathbf{R}_A & {}^C\mathbf{o}_A \end{bmatrix} \begin{bmatrix} {}^Cp \\ 1 \end{bmatrix}. \tag{2.2}$$

The rigid transformation matrix $\begin{bmatrix} {}^C\mathbf{R}_A & {}^C\mathbf{o}_A \end{bmatrix}$ is composed of the orientation ${}^C\mathbf{R}_A$, a 3×3 rotation matrix, and the position ${}^C\mathbf{o}_A$. The matrix ${}^C\mathbf{R}_A$ and point ${}^C\mathbf{o}_A$ define the orientation and the position of the camera center in world coordinates. The matrix $\begin{bmatrix} {}^C\mathbf{R}_A & {}^C\mathbf{o}_A \end{bmatrix}$ is known as the camera extrinsic matrix or the view matrix (in OpenGL).
In the new pipeline the marker frame A will be used as the world frame. From an image of a fiducial marker the pose ${}^C\boldsymbol{\xi}_A$ is estimated. A pose is a pair formed by a position ${}^C\mathbf{o}_A$ and an orientation represented by the 3×3 rotation matrix ${}^C\mathbf{R}_A$, the pose is defined as

$${}^C\boldsymbol{\xi}_A := \left({}^C\mathbf{o}_A, {}^C\mathbf{R}_A\right). \tag{2.3}$$

With the pose, a transformation from the marker coordinate system to the camera coordinate system can be constructed. First the homogeneous transformation matrix ${}^C\mathbf{H}_A$ is composed from the pose as follows

$${}^C\mathbf{H}_A := \begin{bmatrix} {}^C\mathbf{R}_A & {}^C\mathbf{o}_A \\ 0_{1\times3} & 1 \end{bmatrix}. \tag{2.4}$$

Not coincidentally, this matrix is the homogeneous representation of the camera extrinsic matrix. We want to map ${}^C\mathbf{p}$, the coordinates of $\mathbf{p}$ expressed in frame C to ${}^A\mathbf{p}$, the coordinates expressed in frame A. Let ${}^A\bar{\mathbf{p}}$ and ${}^C\bar{\mathbf{p}}$ denote the homogeneous representation of ${}^A\mathbf{p}$ and ${}^C\mathbf{p}$, then inverse matrix ${}^C\mathbf{H}_A^{-1}$ transforms a point in expressed in coordinate system C to that of A with

$${}^A\bar{\mathbf{p}} = {}^C\mathbf{H}_A^{-1}\, {}^C\bar{\mathbf{p}} \tag{2.5}$$

where

$$ {}^{C}\mathbf{H}_{A}^{-1} = \begin{bmatrix} {}^{C}\mathbf{R}_{A}^{T} & -{}^{C}\mathbf{R}_{A}^{T}\,{}^{C}\mathbf{o}_{A} \\ 0_{1\times 3} & 1 \end{bmatrix}. \tag{2.6} $$

When a point cloud is acquired from the camera all points in the point cloud are originally expressed in frame C. Equation 2.5 will be used to convert them to coordinates expressed in A.

2.2 Experimental Approach and Method

2.2.1 Camera Calibration

To align point clouds it is necessary to capture point clouds that are an accurate representation of the sampled surface. The first step in this process is to calibrate the cameras. To accomplish this Intel provides a calibration procedure for the RealSense cameras[1]. The calibration procedure calibrates two aspects of the camera, the depth noise and the depth accuracy.

The depth noise is defined by the measure: sub-pixel root mean square error, or the sub-pixel RMSE. First the camera is pointed to a flat surface parallel to the view plane of the camera. Then a plane P is fitted through all pixels in a region of interest (ROI) within the depth image. The plane-fit RMSE is calculated, this is a measure of the deviation in distance of all pixels in the ROI to P. With $p_i = (x_i, y_i, z_i)$ a point in the point cloud, and $p_{P_i} = (x_{P_i}, y_{P_i}, z_{P_i})$ the projection of p_i on the fitted plane along the plane normal, the plane-fit RMSE is calculated by

$$ RMSE_{plane\text{-}fit} = \sqrt{\frac{\sum_1^n (z_{Pi} - z_i)^2}{n}}. \tag{2.7} $$

The plane-fit RMSE is an absolute value. To compare the plane-fit RMSE value over different camera settings and between different devices the plane-fit RMSE is normalized to a relative value, the sub-pixel RMSE. Instead of the z distance in meters the z distance in device pixels, denoted as $\hat{z}$ is used in 2.7 by converting both z_i and z_{P_i} with

$$ \hat{z} = \frac{d_c f}{z}. \tag{2.8} $$

The variable d_c is the distance between the left and right center of the stereoscopic camera in meters and f is the stereoscopic focal length in pixels, see Figure 2.1 for a graphical explanation. To minimize both the depth noise of the cameras first the value is measured, then the calibration procedure is performed. If the measured calibration values are an improvement the values are stored in calibration table in the memory of the camera.

[1] https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/RealSense_DepthQualityTesting.pdf

2.2.2 Point Cloud Alignment

To align the point clouds from two cameras, a setup is made as shown in Figure 2.3. In this setup there are four frames of reference; frame A from the ArUco marker, frame B is the base frame of the cobot, and frame $C1$ and $C2$ are the frames from the two RealSense cameras.

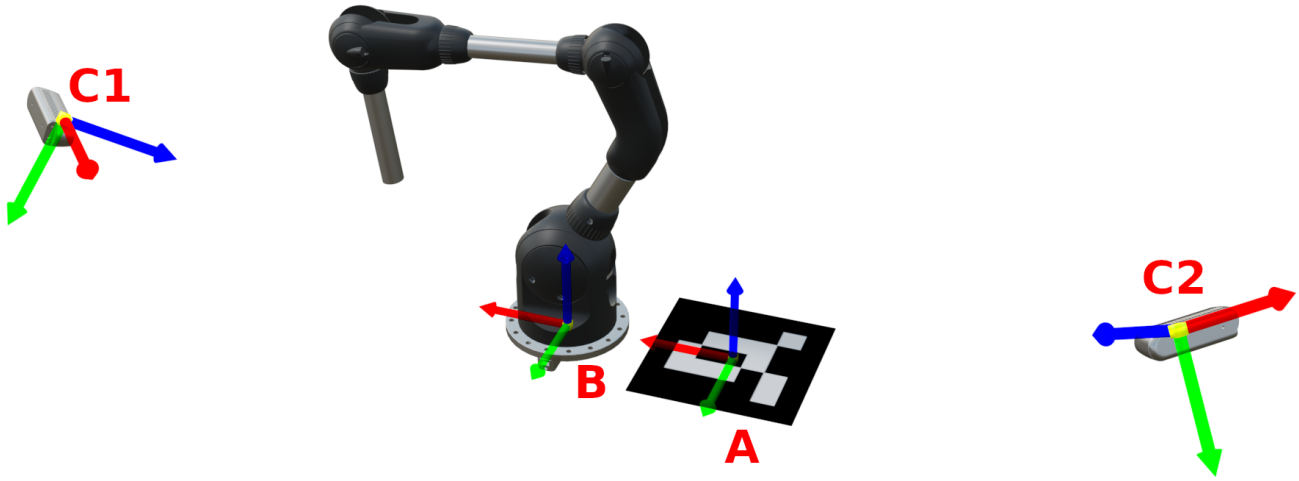

Figure 2.3: Capturing an ArUco marker with two RealSense cameras. Frame A is from the ArUco marker, frame B is the base of the cobot, and frame $C1$ and $C2$ are frames from each RealSense camera. The encoding of the axes is red is the X-axis, green is the Y-axis, and blue is the Z-axis.

Because there was no possibility to have a permanent experimental setup with cameras in exact fixed position the requirement arose to automate the point cloud alignment as much as possible. To align multiple point clouds an alignment procedure was developed. It consists of the steps shown in Figure 2.4.

Figure 2.4: Steps for the alignment of a pair of point clouds.

The responsibilities of each step in Figure 2.4 from left to right is defined as follows:

- Capture an infrared (IR) frame of the ArUco marker with both cameras C_1 and C_2.
- Estimate the poses ${}^{C1}\boldsymbol{\xi}_A$ and ${}^{C2}\boldsymbol{\xi}_A$ from the ArUco marker.
- Compose the homogeneous transformation matrices ${}^{C1}\mathbf{H}_A$ and ${}^{C2}\mathbf{H}_A$ and transform the points of the clouds with 2.5.
- Visualize both the point clouds in the `RBF_Viz` viewer.
- Manually refine ${}^{Cn}\boldsymbol{\xi}_A$, $n \in \{1, 2\}$ so that the alignment of the point clouds is optimal.

The manual refinement is done aligning the ArUco marker center to the world origin and the ArUco board to the world XZ plane. The point clouds are then continuously visualized in the `RBF_Viz` viewer. The poses ${}^{Cn}\boldsymbol{\xi}_A$, $n \in \{1, 2\}$ are refined by changing the values of the vectors ${}^{Cn}\mathbf{o}_A$ and ${}^{Cn}\mathbf{R}_A$. First the centers of the ArUco frame are aligned by manually adjusting the values of ${}^{Cn}\mathbf{o}_A$, finally the plane of the ArUco marker are aligned by adjusting the values of ${}^{Cn}\mathbf{R}_A$. This method can easily be extended, for each camera that is added align it to the ArUco frame or one of the cameras that is already aligned. It is important to only adjust the alignment of the camera that is added.

2.3 Results

2.3.1 Camera Calibration

To improve the quality of the depth measurement the two Intel RealSense D435 depth cameras that were used in the experiments were calibrated. The $RMSE_{sub\text{-}pixel}$ was measured before and after calibration with the depth quality tool and a textured target provided by the calibration documentation[1]. The values of the $RMSE_{sub\text{-}pixel}$ before and after the calibration can be found in Table 2.2. To differentiate between the cameras they are referred to by their serial number.

	$RMSE_{sub\text{-}pixel}$ [pixel]	
Camera serial Nº	Before Calibration	After Calibration
817512070909	0.21	0.06
832112072402	0.15	0.03

Table 2.2: The sub-pixel RMSE at a distance of $1.500\,\mathrm{m}$ measured before and after calibration. The measured values are an average of 15 samples.

The calibrated improvement after calibration is substantial, the improvement is 3.5 and 5 times respectively. With the $RMSE_{sub\text{-}pixel}$ values known for the cameras the values of the $RMSE_{plane\text{-}fit}$ can be calculated for different depths. Figure 2.5 shows a plot for the $RMSE_{sub\text{-}pixel}$ values of each camera.

In practice however much more noise was observed in both the spatial as the temporal domain than can be explained with the $RMSE_{plane\text{-}fit}$. To investigate the spatial depth noise a flat untextured (white wall) and flat passive textured (oak panel) were aligned to the camera plane. A depth image was captured and the depth values of a vertical column of pixels were plotted in Figure 2.6. Because the RealSense cameras use stereophotogrammetry for depth estimation they perform better on textured targets than untextured targets. To be able to perform depth estimation of untextured surfaces the camera has an infrared projector that projects an IR dotted pattern onto the scene that the photogrammetry algorithm then uses to detect disparities.

[1] `https://www.intel.com/content/dam/support/us/en/documents/emerging-technologies/intel-realsense-technology/RealSense_DepthQualityTesting.pdf`

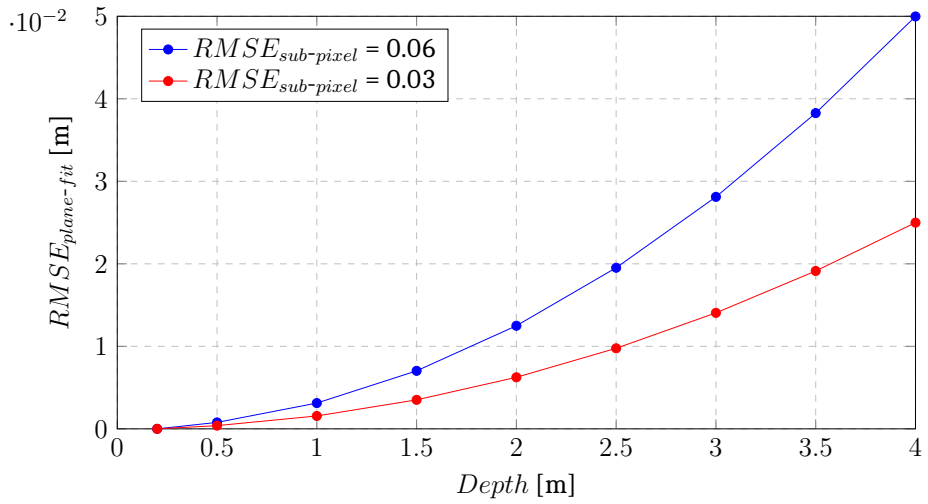

Figure 2.5: The calculated $RMSE_{plane\text{-}fit}$ for different depth values for a $RMSE_{sub\text{-}pixel}$ of 0.03 and 0.06 pixels, a depth image resolution of 640×480, and a horizontal FOV of 79.6.

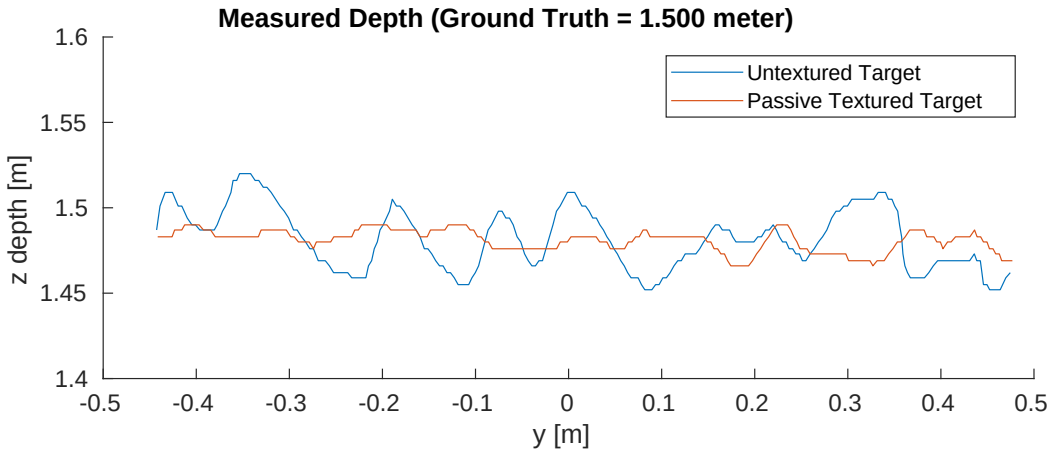

Figure 2.6: A column of pixels of a depth image of a stationary flat surface, the ground truth is $1.500\,\mathrm{m}$. The data was captured with camera with serial №817512070909.

Of this column of 240 pixels the mean, standard deviation, and the root mean square error for an estimator equal to the ground truth $(1.500\,\mathrm{m})$ were calculated in the textured and untextured case. The results are in Table 2.3.

	Target Type	
Measure	Untextured	Passive Textured
Mean [m]	1.482	1.481
SD [m]	0.018	0.006
RMSE$(\hat{\theta} = 1.500\,\mathrm{m})$ [m]	0.026	0.020

Table 2.3: The mean, standard deviation (SD), and RMSE for the estimator $\hat{\theta}$ equal to the ground truth of $1.500\,\mathrm{m}$, for the plot in Figure 2.6.

As can be seen there is a bias of ≈ -0.02 meter with the ground truth of 1.500 meter. The depth

quality tool has a method to remove the bias by setting the ground truth in the camera memory. This feature, when tested on the camera with serial №817512070909, did not work and the depth quality tool reported an error. Hopefully this bug will be fixed in the future. The standard deviation for the passive textured target is 0.006 and this is very close to the value of 0.007 in Figure 2.5 for a depth of $1.500\,\mathrm{m}$. The standard deviation of depth noise of an untextured target however is three times higher. In the $RMSE$ we see the effect of the bias in combination with the deviation of the values.

The next experiment was to measure the depth noise in the time dimension. For the same textured and untextured targets at a distance of $1.500\,\mathrm{m}$ the depth was measured for a duration of 60 seconds at a capture rate of 30 Hz. The depth of a point in the center of the depth image is show in Figure 2.7.

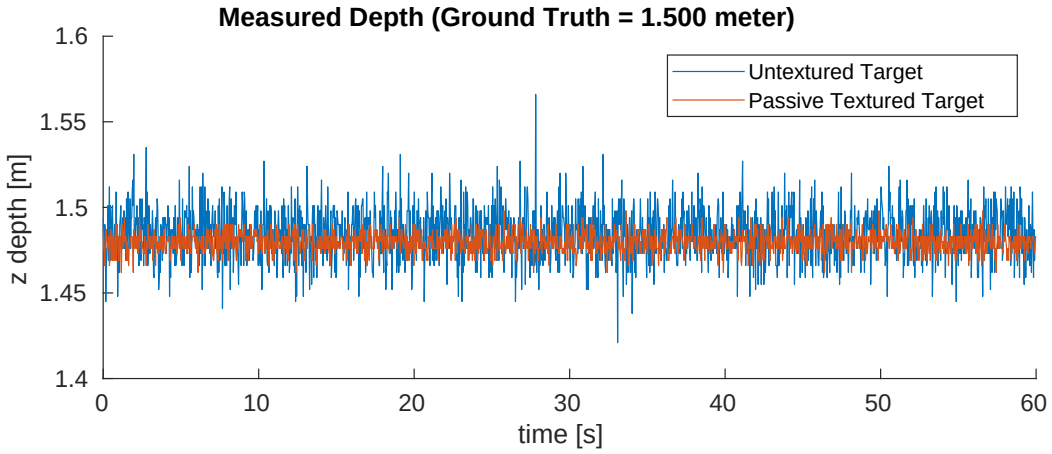

Figure 2.7: The depth value of a center pixel of a depth image of a stationary flat surface, the ground truth is $1.500\,\mathrm{m}$. The data was captured with camera with serial №817512070909.

The same measures as before, the mean, SD, and RMSE were calculated for this signal and are given in Table 2.4.

	Target Type	
Measure	Untextured	Passive Textured
Mean [m]	1.485	1.480
SD [m]	0.015	0.006
RMSE($\hat{\theta} = 1.500\,\mathrm{m}$) [m]	0.022	0.021

Table 2.4: The mean, standard deviation (SD), and RMSE for the estimator $\hat{\theta}$ equal to the ground truth of $1.500\,\mathrm{m}$, for the plot in Figure 2.7.

The values are close to the values in Table 2.3 as is to be expected. The depth noise in the time domain is of high frequency however, the value of the same pixel in a depth image can vary a lot between successive frames.

To reduce the depth noise in the temporal dimension the RealSense SDK offers a temporal post-

processing filter[1]. The temporal filter is an exponential moving average filter. It is applied with the default settings in the next experiment. A discrete Fourier transform was applied to the measured values and the measured values after application of the temporal filter in Figure 2.8.

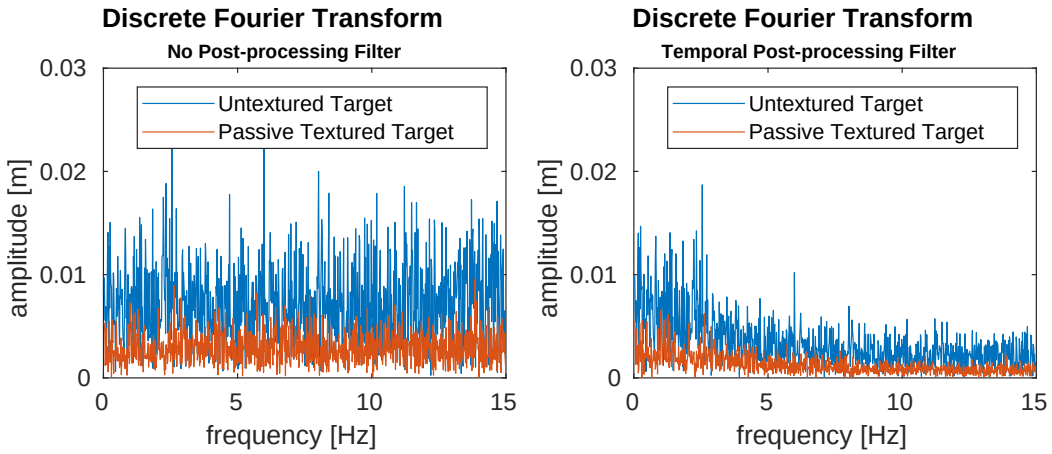

Figure 2.8: *Left:* Discrete Fourier transform of the signal in Figure 2.7. *Right:* Discrete Fourier transform of the signal after filtering with the temporal post-processing filter.

The temporal filter indeed reduces the high frequency depth noise. In the left plot the all frequencies are evenly present in the measured signal while in the right plot the higher frequencies are suppressed. The filtered signal that is shown in Figure 2.9 reflects this as well, the amplitude of the noise is roughly halved compared to Figure 2.7.

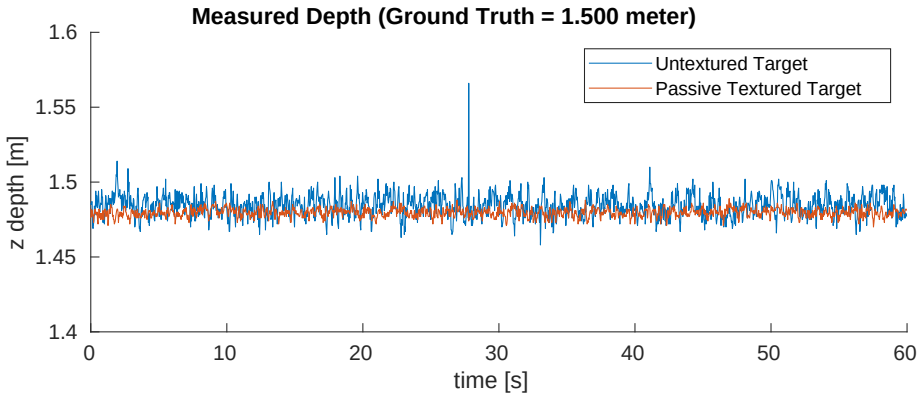

Figure 2.9: The depth value of a center pixel of a depth image of a stationary flat surface, the ground truth is $1.500\,\mathrm{m}$

Table 2.5 shows that the use of the temporal post-processing filter positively affects the value of SD and to a lesser extent the RMSE.

[1] `https://github.com/IntelRealSense/librealsense/blob/master/doc/post-processing-filters.md`

Measure	Target Type	
	Untextured	Passive Textured
Mean [m]	1.484	1.480
SD [m]	0.008	0.003
RMSE($\hat{\theta} = 1.500\,\mathrm{m}$) [m]	0.017	0.021

Table 2.5: The mean, standard deviation (SD), and RMSE for the estimator $\hat{\theta}$ equal to the ground truth of $1.500\,\mathrm{m}$, for the plot in Figure 2.9.

2.3.2 Point Cloud Alignment

Because it was not possible to have a permanent experimental setup with multiple cameras in exact fixed positions, the requirement arose to automate the point cloud alignment as much as possible. A point cloud alignment framework was developed that automates the process as much as possible while allowing manual refinement of the alignment. The alignment can also be made permanent by storing the matrix ${}^{C}\mathbf{H}_{A}^{-1}$ to disk for each camera. The matrix can then be loaded later and be applied to any dataset, recorded or captured. This is ideal in the situation where the fiducial marker should not be in the scene or where the cameras remain in a static position for a long period. The refined camera alignment can be loaded from disk and the manual refinement of the alignment is only needed once.

To align two point clouds the following steps are taken in the `RBF_viz` tool. Two RealSense cameras are pointed at a clearly visible ArUco marker. The marker should be quite large, the marker should be at least 100×100 pixels in the captured image that is used for pose estimation [12]. Avoid very acute angles between the camera view axis and the marker plane, pose estimation is less accurate at very acute angles. When `RBF_viz` is started the point cloud are automatically aligned as seen in Figure 2.10. The point clouds are color coded to distinguish the point clouds of each camera.

In the tool the left infrared image is used for the pose detection. This is done because the color image captured by the camera is captured with a different imager than the stereoscopic imager used for the capture of depth images. The color imager has a different horizontal and vertical FOV and is not used in the photogrammetry algorithm. In Figure 1.4 this difference in FOV can be seen between the color image on the left and the depth image on the right.

If there is some misalignment of the fiducial marker in both point clouds with respect to the world origin and orientation `RBF_viz` provides manual controls to align the point cloud with six degrees of freedom (6DoF). In Figure 2.11 the center of the marker was aligned to the world origin and then the marker board was aligned to the XZ world plane.

Then the right camera was aligned in the same manner, see Figure 2.12. The values of the 6DoF are shown in the panel on the left. These values are translations and rotations on top of the internal matrix ${}^{C}\mathbf{H}_{A}^{-1}$.

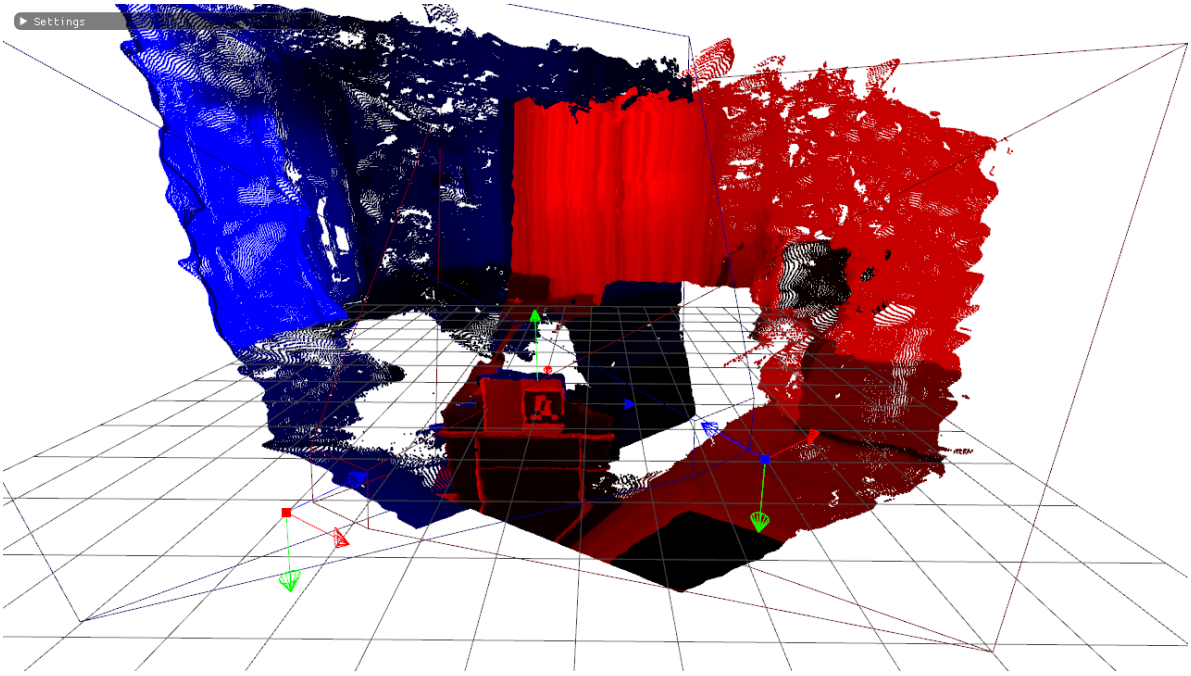

Figure 2.10: Automatic alignment to the ArUco marker. The XYZ axis of frames are encoded as red, green, and blue. The frame on the left, with the red origin, is of the camera that records the red point cloud on the right. The frame on the right, with the blue origin, is of the camera that records the blue point cloud on the left. The middle frame is the world origin and axis directions. The skeletons of the camera frustums are also visualized.

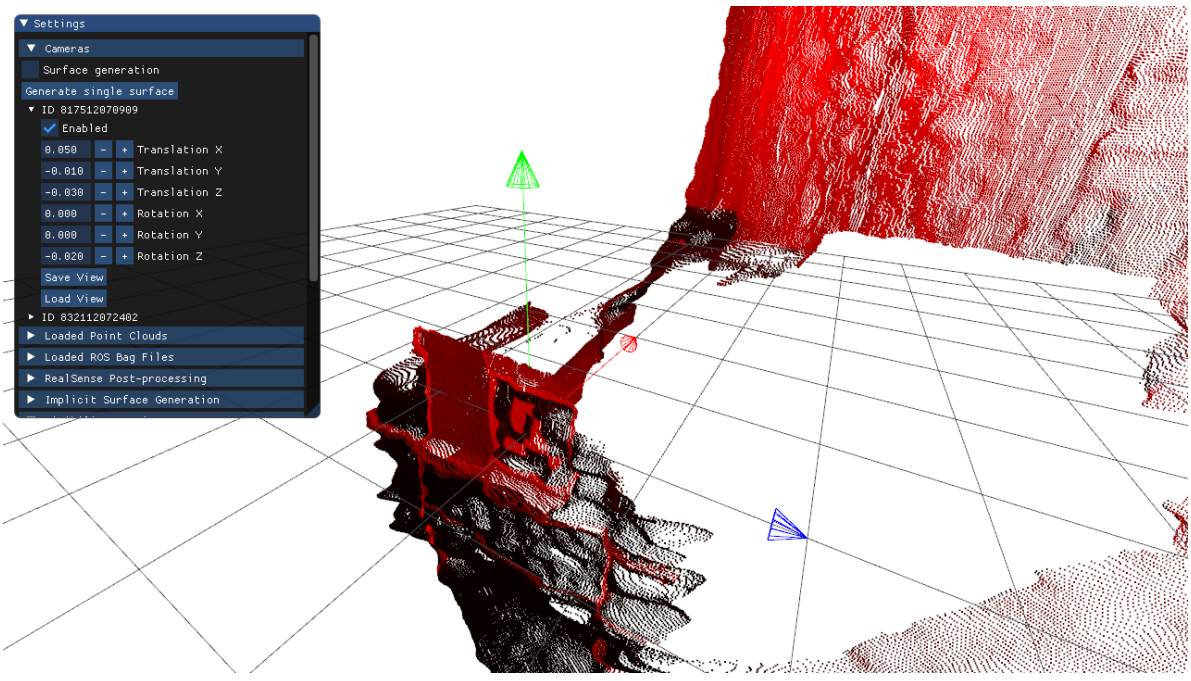

Figure 2.11: The orientation and position of the left camera aligned to the world frame. In the left the panel with the 6DoF are shown.

Finally the full matrix ${}^{C}\mathbf{H}_{A}^{-1}$ with all manual translations and rotations applied can be stored to disk. An example listing of this matrix of the left camera can be found in Appendix B.

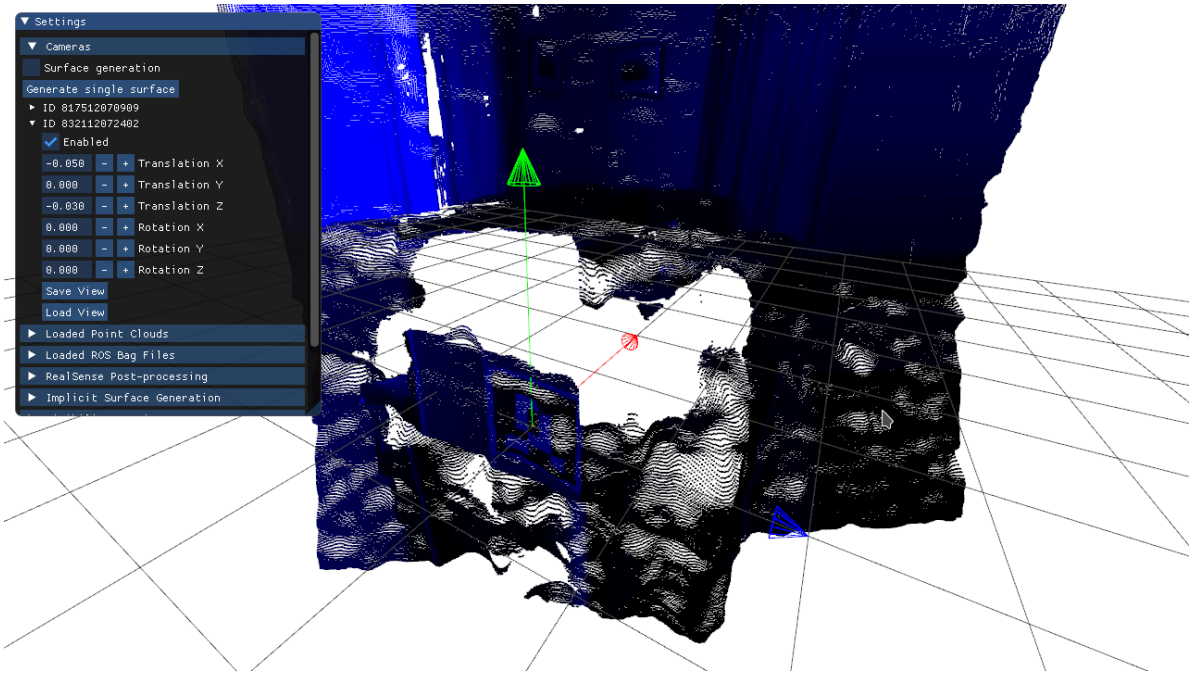

Figure 2.12: The orientation and position of the right camera aligned to the world frame.

2.4 Discussion and Recommendations

2.4.1 Camera Calibration

Interpretations: Calibration of the RealSense depth cameras definitely pays off and helps improve the depth accuracy of the camera. The standard deviation of the depth of points on a textured surface is close to the value that is expected from the $RMSE_{sub\text{-}pixel}$ value measured with the depth quality tool. With calibration the depth noise is still substantial however, especially on untextured surfaces because the stereophotogrammetry algorithm can not determine the disparities it needs to calculate the depth. Our experiment showed that the depth noise of untextured surfaces was approximately $3\times$ higher than on a passively textured surface. It was demonstrated that the application of the temporal post-processing filter helps in reducing the temporal aspect of the depth noise. Reduction of temporal and spatial depth noise is necessary to have the cobot not react to the noise on objects that it needs to avoid collision with.

Implications and limitations: The use of the temporal post-processing filter does not come without a cost. According to the documentation[1] the filter introduces visible blurring and smearing artefacts on moving objects, and therefore is best-suited for static scenes. This is indeed the case, to demonstrate this a stream of an arm moving in front of a wall was recorded and played back with and without the temporal filter. In Figure 2.13 the artifacts caused by the temporal filter are visualized.

The noise at a distance of ≈ 1.5 meter from a calibrated camera is quite substantial, for instance,

[1] `https://github.com/IntelRealSense/librealsense/blob/master/doc/post-processing-filters.md`

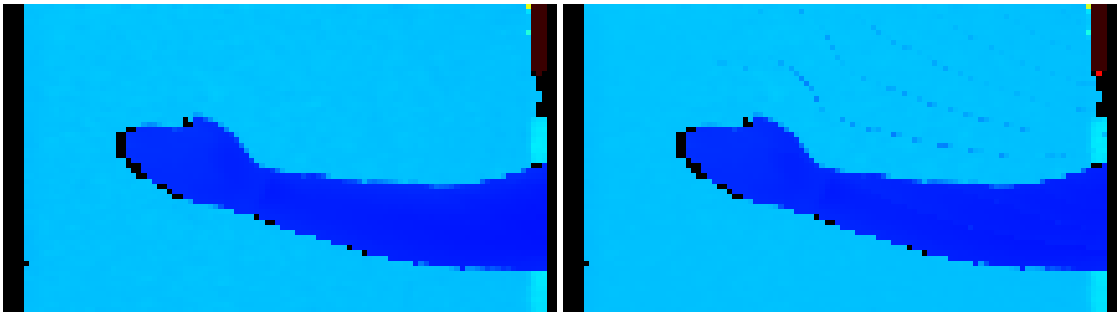

Figure 2.13: The same depth image from a stream captured of an arm moving downwards ($v \approx 2\,\mathrm{m/s}$) in front of a flat surface. On the left no temporal filter is used on the stream, on the right the temporal filter is used on the stream.

consider the scene in Figure 2.14.

Figure 2.14: Color, infrared, and depth image captured by a calibrated RealSense camera with ArUco marker and textured target at a distance of approximately 1.6 meter.

The tennis balls on the floor are not detectable in the depth image and are thus invisible in the point cloud. Even though the tennis balls are lying on a textured surface and their color contrasts with this surface. Most human wrists are thinner than a tennis ball and would thus not be properly detected in all cases. Another issue is that the $RMSE_{sub\text{-}pixel}$ is a relative value, the absolute error grows worse the further the acquired depth point is from the camera as is shown in 2.5. In the pipeline also a $8\times$ decimation post-processing filer is applied to the depth image reducing it from 640×480 to 80×60 resolution which hugely reduces the detail in the point cloud.

Another issue with the point clouds captured by the RealSense D435 camera is the presence of "phantom points". Phantom points are points, usually captured at the edge of an object, that takes a depth value between that of the object and that of the background. In Figure 2.15 such a case is illustrated where phantom points are visible around the edge of the subject in the scene. In the reconstruction of the surface no temporal filtering is used. The temporal filter unfortunately worsens the issue with phantom points. Phantom points are much more prominent in a surface reconstructed from point clouds post-processed with the temporal filter.
Phantom points are problematic because the cobot controller will attempt to avoid collision with these points. The phantom points are also very dynamic in nature and could result in erratic move-

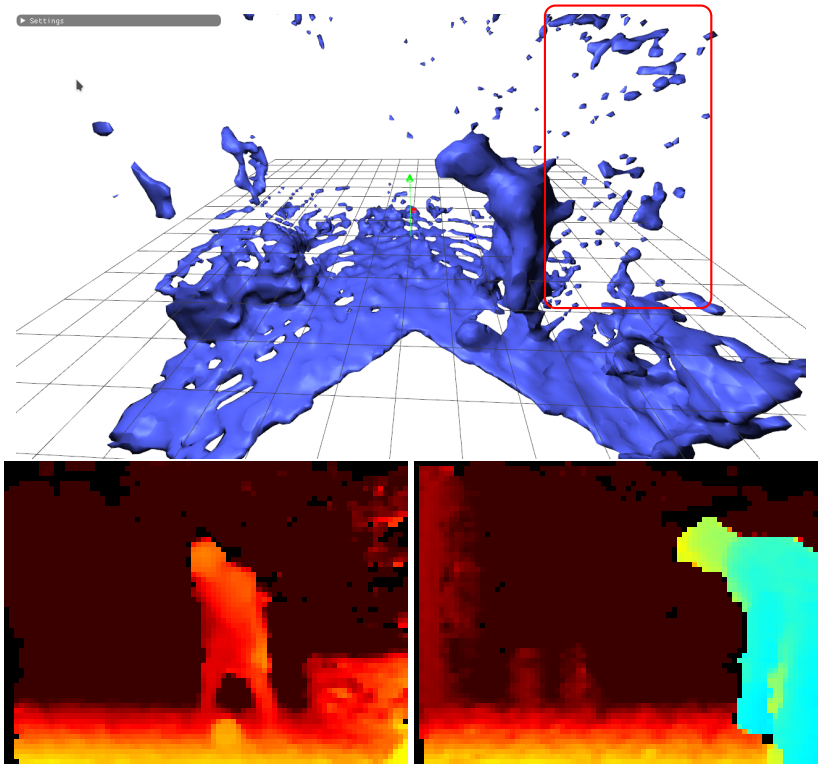

Figure 2.15: Phantom points. *Top:* Fused point cloud that shows phantom points in the area annotated by the red rectangle. *Bottom:* The depth images of the left camera and the right camera of the surface.

ment of the cobot which is undesirable.

Recommendations:

Better temporal filter: Because the temporal post-processing filter provided by the RealSense SDK causes artifacts it might be interesting to introduce a filter that has a better responsiveness. An efficient filter that is tuned for high precision and responsiveness is the 1€ filter [14]. This filter outperforms an exponential moving average filter in the area of responsiveness or lag which causes the artifacts.

External IR Projector: The hardware of the RealSense D435 camera calculates the depth by solving the correspondence problem between the simultaneously captured left and right images. When the disparity for each pixel is known the depth map can be calculated by triangulation. The RealSense D435 works well in bright, well-lit environments but has issues with, for instance, textureless surfaces like indoor dimly lit white walls. To improve depth perception in those cases the RealSense uses active stereo vision. In active stereo vision an IR projector is used to overlay a semi-random texture over the scene. The depth precision can be increased further by using external projectors to help the sterophotogrammetry algorithm[1].

Use ToF or LIDAR cameras: When more depth precision is required a time-of-flight (ToF camera

[1] `https://dev.intelrealsense.com/docs/projectors`

like the Kinect v2 can be used to increase the precision [8]. The Kinect v2 has the limitation that multiple Kinect can not easily be used to capture in parallel to reduce the shadow in the captured point cloud because the cameras projectors interfere. The new Azure Kinect DK[1] allows multiple cameras to capture frames with an interval of just $160\mu s$[2].
Intel will soon release its first laser imaging, detection, and ranging (LIDAR) camera, the RealSense L515[3], capable of capturing depth images at 30 Hz. The depth precision and depth range of the L515 camera is higher than that of the RealSense D400 series according to the provided datasheet. Multiple L515 cameras can be used to capture the same scene and synchronize with a hardware synchronization cable.

Remove Phantom Points: None of the above techniques eliminate phantom points however. It might be possible to reduce the phantom points from captured depth images by applying an edge-preserving post-processing filter as is provided in the RealSense SDK. It is certainly interesting to research if other types of image filters exist that can be tuned to remove phantom points. Another approach for the reduction of noise, the removal of phantom points, and otherwise improve the quality of depth images or point clouds is to train a neural network and use that to improve the quality [15].

2.4.2 Point Cloud Alignment

Interpretations: With the help of fiducial markers it is possible to quickly obtain a rough alignment of point clouds captured by multiple cameras. The alignment error of the point clouds is still quite large but can be corrected with a simple manual refinement procedure in 6DoF. The alignment parameters can be made persistent and can be used in another session when the cameras poses remain the same.

Implications and Limitations: Because it was not possible to have a permanent camera rig the need arose to quickly align multiple point clouds. In the previous pipeline all values were hard-coded. As only one camera was used this was a acceptable approach but when introducing multiple cameras a correct precise alignment is necessary to have a correct reconstructed surface. The new framework is quick and provides precise refinement while still allowing manual intervention to correct misalignment errors. The manual procedure is made as simple as possible by providing the ability to tint the point cloud and to turn the individual point clouds on or off. It is also possible to use an orthogonal camera to view a point cloud in the direction of any of the world axes without perspective distortion.
To get accurate results the size of the marker in the captured picture should be at least 100×100. There were a lot of issues when the marker was to small or if the marker was observed at an acute

[1] https://docs.microsoft.com/en-us/azure/kinect-dk/depth-camera
[2] https://docs.microsoft.com/en-us/azure/kinect-dk/multi-camera-sync
[3] https://www.intelrealsense.com/lidar-camera-l515/

angle. Care should be taken that all cameras capture the marker properly. Initially the color imager of the RealSense was used to capture the marker, this is not correct. One of the infrared imagers has to be used because these are also used to reconstruct the point cloud. Capturing the marker correctly might pose a challenge when multiple cameras are used scattered around the area.

Recommendations: It is possible to fully automate the alignment of the point clouds by using an algorithm like the iterative closest point (ICP) algorithm [16]. The point clouds captured by depth cameras only partially overlap in our case and this can cause the ICP algorithm to fail. There is a variant of ICP however, Sparse ICP [17], that is less sensitive to partially overlapping point clouds and outliers. Unfortunately the fidelity of the point cloud acquired by the RealSense D435 is way too low to get Sparse ICP to work. When the switch to a camera model with less depth noise is made this might be interesting to investigate further.

Chapter 3

Efficient Surface Reconstruction

3.1 Literature Review, Concepts, and Models

3.1.1 Point Cloud Density

The computation time of step 3 (fill the Φ matrix) and step 4 (Solve the system of equations $g = \omega\Phi$) from Figure 1.16 is dependent on the number of non-zeros that end up in the Φ matrix. A measure of how many non-zeros end up in the matrix Φ for a center c is counting the neighbors in a sphere with a radius of σ around c. An illustration of this measure can be found in Figure 3.1.

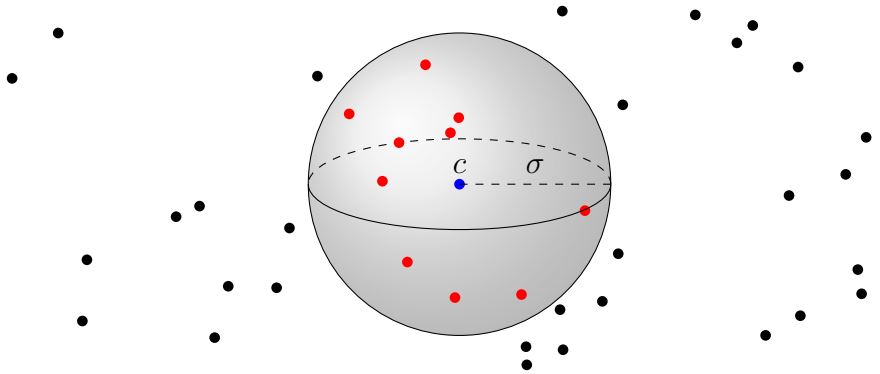

Figure 3.1: For a center c (blue) the number of neighboring centers in a sphere with radius σ around c can be counted. The neighbors inside the sphere are depicted by the red centers.

The number of neighbors within σ can be calculated for each center c in the point cloud and can be visualized. In Figure 3.2 the number of neighbors is visualized by a color scale for the point cloud from Figure 1.4.
We can observe from this figure that centers close to the camera have a lot of neighbors while points further from the camera have less. This can easily differ an order of magnitude within a short distance range. Also centers on the edges of the point cloud have a lower number of neighbors as the part of the sphere outside the depth camera frustum is guaranteed to not contain any neighbors.

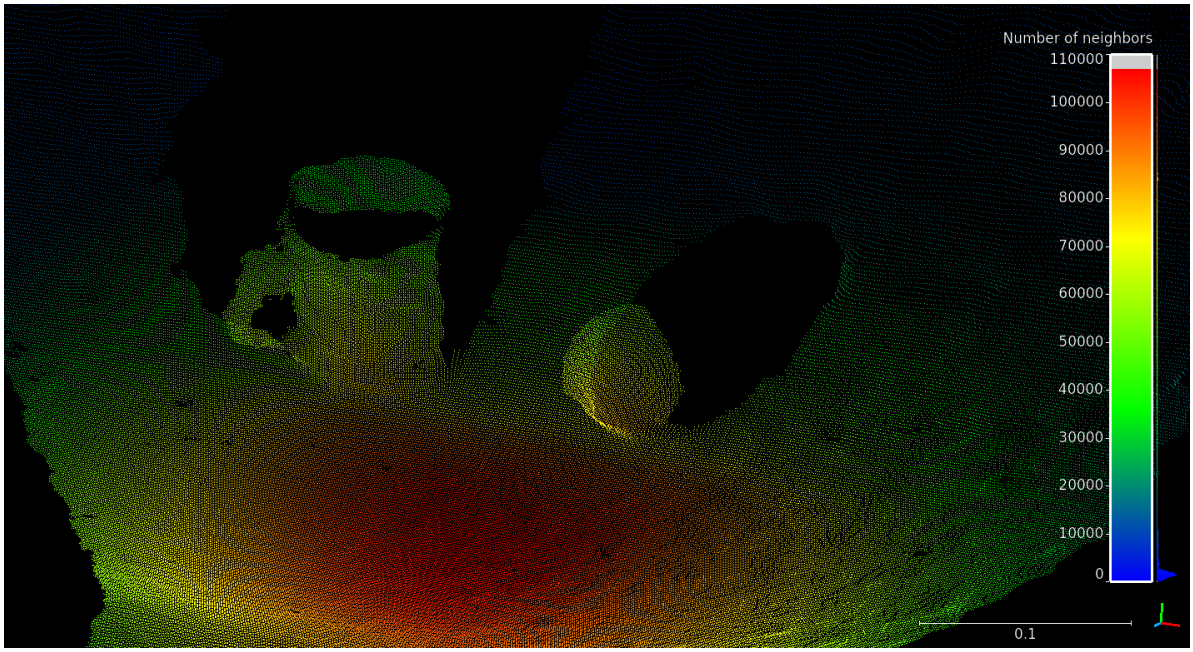

Figure 3.2: Visualization of the number of neighbors within a radius $\sigma = 0.17m$ for the points of the point cloud of Figure 1.4.

Let C be a depth camera and P a plane perpendicular to the view direction of the camera as seen in Figure 3.3. Then let d_c be the distance between the view center of the camera and the closest point on P and let $\alpha = (\alpha_x, \alpha_y)$ be the field of view angle of the camera.

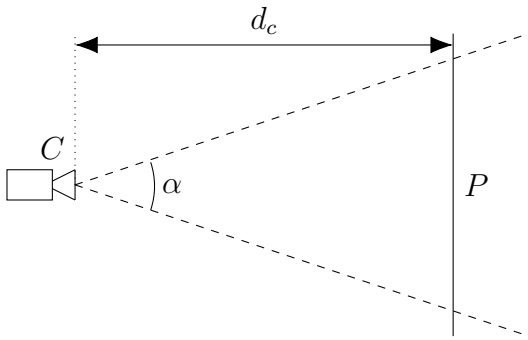

Figure 3.3: The d_c is the distance between view center of a depth camera C and the center of a plane P perpendicular to the view direction of the depth camera.

Then the vertical size s_y of the part viewed of plane P is given by

$$s_y = 2d_c \tan\left(\frac{\alpha_y}{2}\right). \tag{3.1}$$

If the number of pixels in the depth image is $n = n_x \times n_y$ then we can estimate the distance d_p between centers on the plane at depth d_c with

$$d_p = \frac{s_y}{n_y}. \tag{3.2}$$

Because depth images have square pixels the value d_p is equal in x and y direction. An illustration of d_p is shown in Figure 3.4.

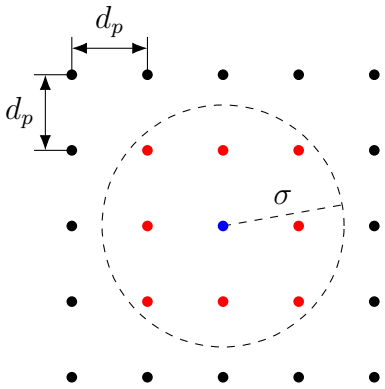

Figure 3.4: The d_p is the distance between to adjacent sampled depth points of a point cloud in the flat plane P perpendicular to the view direction of depth camera. The number of neighbors (red) for a center (blue) in the sphere with radius σ can be calculated by counting the points with a distance to c smaller than or equal to σ.

When the value of d_p is known the number of neighbors for centers with a distance greater than σ from the border of P can be calculated and an average contribution of non-zeros added to Φ per point can be given.

3.1.2 Sparse Matrix Storage Formats

Later in this chapter an algorithm will be described that computes the elements of the sparse matrix Φ from Equation 3.6. A sparse matrix is a matrix where most of the elements are zero. To store a sparse matrices several sparse matrix storage formats were developed [18]. The most popular of these is the compressed row sparse storage (CSR) format [19]. To store the matrix

$$\begin{bmatrix} 3 & 8 & 0 & 0 & 0 & 0 \\ 11 & 0 & -8 & 0 & 0 & 0 \\ 0 & 7 & 1 & 0 & 0 & 0 \\ 0 & 0 & 5 & -4 & -2 & 0 \\ 0 & 0 & 5 & 0 & -7 & 8 \\ 0 & 0 & 0 & 14 & 9 & 6 \end{bmatrix} \tag{3.3}$$

in CSR format three arrays are needed. These are the the *value*, *column indices*, and the *row indices* arrays. For the matrix in this example they will be filled as in Figure 3.5.
The *value* array contains all the non-zero values, row by row. The *column indices* array contains the column indices of the values in the *value* array. In all the examples 0-based indices are used. The *row indices* array contains start indices of each row in the *value* and *column indices* array. For example, the second element in the *row indices* array is the index where the row starts in the *value*

values =	3	8	11	-8	7	1	5	-4	-2	5	-7	8	14	9	6
column indices =	0	1	0	2	1	2	2	3	5	2	4	5	3	4	5
row indices =	0	2	4	6	9	12									

Figure 3.5: The matrix in Equation 3.3 stored in the CSR storage format.

and *column indices* array. This storage method is only efficient if the matrix is sparse enough, that is when the number of non-zeros n_{nz} for a $n \times n$ matrix is

$$n_{nz} < (n(n-1)-1)/2. \tag{3.4}$$

In the next section the solver from the original pipeline is replaced with the parallel solver proposed in [[20]. This solver only accepts a generalization of the CSR storage format, the block compressed sparse row (BCSR) storage format. The matrix is first divided in blocks. If the block size is 2×2 then the matrix in Equation 3.3 is is divided into 3×3 blocks as shown in Equation 3.5. The blocks are then stored in the $2 \times m_{nz}$ ***block value*** array with m_{nz} the number of non-zero blocks. A block is non-zero when one of its elements is non-zero. The *block value* array will be filled as in Figure 3.6.

$$\left[\begin{array}{cc:cc:cc} 3 & 8 & 0 & 0 & 0 & 0 \\ 11 & 0 & -8 & 0 & 0 & 0 \\ \hdashline 0 & 7 & 1 & 0 & 0 & 0 \\ 0 & 0 & 5 & -4 & -2 & 0 \\ \hdashline 0 & 0 & 5 & 0 & -7 & 8 \\ 0 & 0 & 0 & 14 & 9 & 6 \end{array}\right] \tag{3.5}$$

block values =	3	8	0	0	0	7	1	0	0	0	5	0	-7	8	
	11	0	-8	0	0	0	5	-4	-2	0	0	14	9	6	
block column indices =	0	1	0	1	2	1	2								
block row indices =	0	2	4	6											

Figure 3.6: The matrix in Equation 3.3 stored in the BCSR storage format.

The BCSR format then has two more arrays, the *block column indices* and the *block row indices* arrays which are the same as their CSR counterparts, the *column indices* and the *row indices* arrays, except that they contain indices of blocks instead of individual matrix elements. The BCSR storage format offers advantages on both CPU and GPU under certain conditions [20].

3.1.3 Previous Pipeline

In Section 1.2.1 a brief overview was given of the processing pipeline proposed in the thesis of T. Bosch [2]. In this chapter the improvement of the computation time of the steps in Figure 1.2 are discussed that are responsible for the reconstruction of surface. The details of these three steps are described in more detail below.

Compute Point Cloud. In this step all points of the point cloud P, except the points that have no depth, are transformed with Equation 2.5. The values of the matrix ${}^{C}\mathbf{H}_{A}^{-1}$ were hard-coded into the implementation. For the depth camera used in the experiment, the Intel RealSense, any point depth image where the stereoscopy algorithm is not able to estimate the depth the point is assigned a z-value of 0. These points are omitted from P because they have no purpose in the reconstruction of the environment. The computational complexity of this step is $O(n)$, with n the number of points in P, as only one pass over the points in P is necessary.

Fill Φ Matrix. In this step all entries of the matrix Φ are calculated. As seen in Equation 1.7 and Equation 1.9 the matrix is $n \times n$ with n the number of points in P. Each element in Φ is determined by calculating $\phi(r_{ij})$ where $i \in 1, 2, \ldots, n$ is used as row index of the matrix and $j \in 1, 2, \ldots, n$ is used as the column index of the matrix. Because this will result in an algorithm with a computational complexity of $O(n^2)$ the algorithm was made more efficient with spatial hashing.

For each point we are only interested in the points within the support radius σ of the point. To retrieve the points within the support radius σ efficiently the points are added to an axis aligned regular voxel-grid. The voxel-grid subdivides a part of $\mathbb{R}^3$ with cubic cells or voxels of size σ^3. Each voxel then has a reference to a linked list with all points in the voxel, see Figure 3.7 for an example.

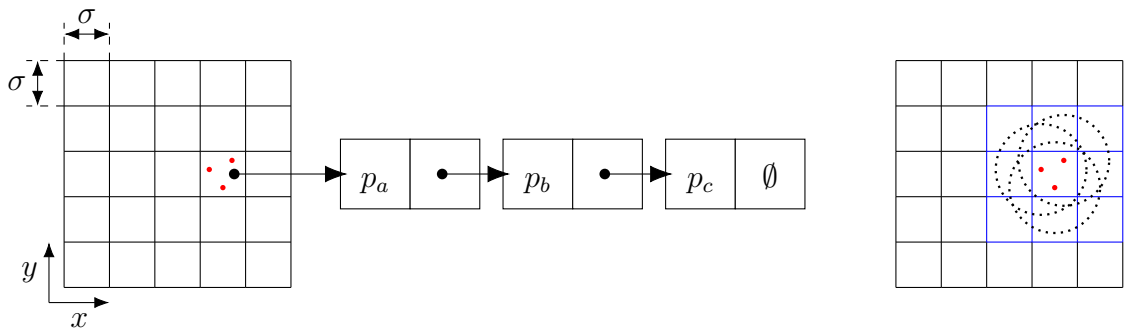

Figure 3.7: Example of spatial hashing. *Left:* The axis aligned regular voxel grid with voxels of size σ^3. Each voxel has a reference (black arrow) to a linked list. Here, for instance, the linked-list of voxel $(4, 3)$ contains the three points (red) p_a, p_b and p_c from the point cloud. *Right:* To get all points within σ (dotted circles) of the points in voxel $(4, 3)$ only the linked list of the neighboring voxels (blue border) need to be retrieved.

It is important that the voxels are of size σ^3, this ensures that all points within a distance of σ can be found by accessing the linked lists of only the adjacent voxels. The listing of Algorithm 1 shows the implementation.

Algorithm 1: Fill Φ matrix from point cloud P.

```
1  FillPhi(P, Φ)
   Input: The point cloud P and the empty matrix Φ.
   Result: The filled matrix Φ
2  Populate the voxel-grid G with all points of P
3  Reserve 500 non-zeros per row for Φ
4  foreach p_i ∈ P do
5      Get the set Q, all linked lists of the neighboring voxels.
6      foreach q_j ∈ Q do
           // Because Φ is symmetric only the upper triangular part has to be filled
7          if i < j then
8              r_ij ← ||p_i − q_j||
9              if r_ij ≤ σ then
                   // Wendland RBF
10                 Φ_ij ← φ_{3,1,σ}(r_ij)
11     end
12 end
13 return Φ
```

Populating the voxel-grid can be done in linear time and the same holds for the reservation of space for the non-zeros per row on line 3. As the Φ matrix is symmetric only the upper triangular part of Φ has to be filled which is ensured by the check on line 7. The algorithm uses the Eigen SparseMatrix implementation[1] which requires only one triangular half to be filled for symmetric sparse matrices. If the number of points stored in the linked list of the neighboring voxels of the current voxel is low, i.e. close to one per voxel, iterating over the points of the neighboring voxels has a computational complexity approaching $O(1)$. This is because each voxel has at most 26 neighbors in the three-dimensional case. When this is the case Algorithm 1 will have a computational complexity approaching $O(n)$ [2].

Solve $\Phi\omega = g$. In this step the vector g is filled with the constraint values for surface and off-surface points. The constraint value of both the surface and off-surface points are set to the negative saturation value $h = -d$. Then to solve the linear system of equations the Eigen conjugate gradient (CG solver is used[2]. The CG method is the most popular iterative method for solving linear equations in the form of

$$Ax = b \tag{3.6}$$

where x is an unknown vector and b is a known vector [21]. The matrix A has to be square, symmetric, and positive-definite (SPD). A matrix is positive-definite if for all non-zero vectors x

$$x^T Ax > 0. \tag{3.7}$$

[1] https://eigen.tuxfamily.org/dox/group__TutorialSparse.html
[2] https://eigen.tuxfamily.org/dox/classEigen_1_1ConjugateGradient.html

The principle of the CG method is based on the fact that the gradient of the quadratic equation

$$f(x) = \frac{1}{2}x^T Ax - b^T x. \tag{3.8}$$

is

$$\nabla f(x) = Ax - b \tag{3.9}$$

when A is a SPD matrix. If we set $\nabla f(x)$ to 0 we again obtain 3.6. Thus the minimum of $f(x)$ is a solution of $Ax = b$. To find this solution the CG solver starts at a point x_0 and takes an initial step into the direction where f decreases the most quickly, i.e. $-\nabla f$. In the next iteration the step is in the conjugate gradient direction of the previous step.
The number of iterations the Eigen solver takes is controlled by two stop conditions, the maximum number of iterations and the maximum tolerance. If one of these stop conditions is met the solver finishes. The tolerance t corresponds to the residual and at iteration i is defined by

$$t_i = \frac{||Ax_i - b||}{||b||}. \tag{3.10}$$

The residual can be interpreted as how far we are from the actual value of b. Because the tolerance is calculated by taking the norm of the residual $Ax_i - b$ and normalizing it by $||b||$ the tolerance is a relative error. This limits the use of the tolerance because no absolute error can be guaranteed. Obviously $||b||$ can not be 0 because then $t_i = \infty$. The computational complexity of the CG method is $O(n_{nz}\sqrt{k})$ with n_{nz} the number of non-zeros in A [21]. The variable k is the condition number of A, a measure that indicates how much the output of an equation changes as a result of a change of the input. To improve the computation time the Eigen CG solver uses the Jacobi preconditioner by default. The Jacobi preconditioner or diagonal preconditioner is a vector of the diagonal of A

$$P = diag(A) \tag{3.11}$$

with the condition that A is a square matrix [21]. In our case P will contain only ones because that is the value of Equation 1.6 for $r = 0$.

3.2 Experimental Approach and Method

To improve the efficiency of the reconstruction of the implicit surface only the steps in Figure 3.8 are considered. The steps needed for visualization of a level set of the implicit surface are not considered in this chapter as they are only used for validation but they are not necessary as input for the cobot controller.
The following sections will describe in detail how these steps work and how they were adapted so that their efficiency was improved.

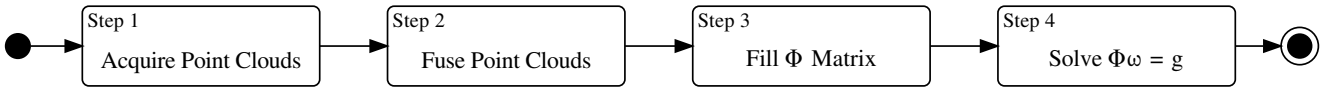

Figure 3.8: Top-level activity model of the first four steps of the pipeline.

3.2.1 Step 1: Capturing and Post-processing Multiple Point Clouds

The first step is responsible for the acquisition of the point clouds and their post-processing with image and depth filters. When capturing multiple point clouds of a dynamic environment it is important that the point clouds are captured at the same time. If there is too much delay between the captured point clouds the reconstructed environment is not accurate. To help accomplish this with the cameras used in this experiment, the RealSense D435, Intel has published a white paper [22] with recommendations. One recommendation to keep the latency between captured point clouds as low as possible is to capture and process each point cloud in parallel. As long as the capture of each depth image is triggered at the same time the maximal latency with this method is at most one frame. This recommendation results in the detailed activity diagram shown in Figure 3.9.

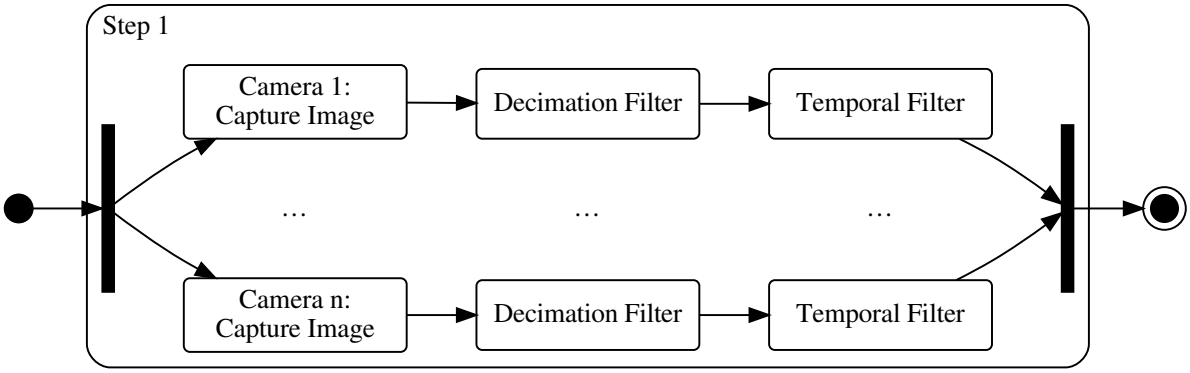

Figure 3.9: Detailed activity model of step 1 for m cameras.

For each camera in the set of cameras $C = \{C_1, \ldots, C_m\}$ an activity is started at the same time in parallel to capture a depth image. The decimation and and other desired post-processing filters, like for instance the temporal filter (optional), are applied to the depth image. Finally we wait until all parallel activities are finished before continuing to next step.

Another recommendation is to synchronize the internal clocks of the cameras via a synchronization cable. One of the cameras is set as the master, the others are set as slave. The clock of the master will then be used by all cameras that are set as slave to capture images at exactly the same time.

3.2.2 Step 2: point cloud fusion and compression

The detailed activities of step 1 are shown in Figure 3.10. In step 1 the points of all point clouds were captured at a certain time by the cameras and are stored in m sets $P = \{P_1, \ldots, P_m\}$. These point clouds are the input for step 2.

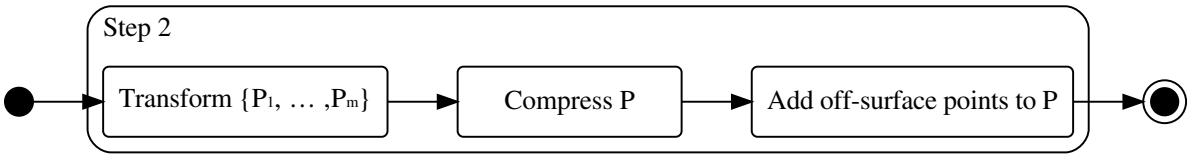

Figure 3.10: Detailed activity model of step 2 for m tasks.

Alignment of Point Clouds

The activity *"Transform P"* transforms all points of a point cloud from the frame of its camera to the frame of the ArUco marker with Equation 2.5. This activity then removes invalid points from P. Points are considered invalid when they have a depth value of 0 meter or when the are positioned outside the extent of the voxel-grid. In this activity for each point the vectors with the direction of the camera that captured the point is calculated.

Voxel-grid Filter

The next activity, *"Compress P"* reduces the number of points n_p in P. The point cloud acquired by each RealSense camera in the experiments consists of $n_p = 60 \times 80 = 4800$ points after the $8\times$ decimation filter is applied. If, for instance, for each point in the point cloud an off-surface point is added this will result in $2n_p$ points. When another camera is added to the pipeline the total number of points increases to $4n_p = 19200$. Obviously an increase in number of points will result in an increase in the calculation time. If compression can reduce the number of points in the point cloud without compromising the ability to generate an accurate surface it is a good approach to keep the computation time low.

Another aspect that decreases the surface reconstruction speed is when the number of neighbors for many points in point cloud n_d is high. When multiple point clouds are fused there are parts of surfaces that are sampled by multiple cameras. In Figure 3.11 such a situation is illustrated.
As can be seen the two point clouds, one tinted blue and the other tinted red, are overlapping. They overlap, for instance, on the surface of the table or the front of the tennis ball and coffee mug. When the number of neighbors n_d are visualized for all points, we see that the number of neighbors for individual points increases significantly in these overlapping areas.
To reduce the increase in computation time caused by increased n_p and n_d step 2 will compress the point cloud. Ideally the compression should have the following properties [23]:

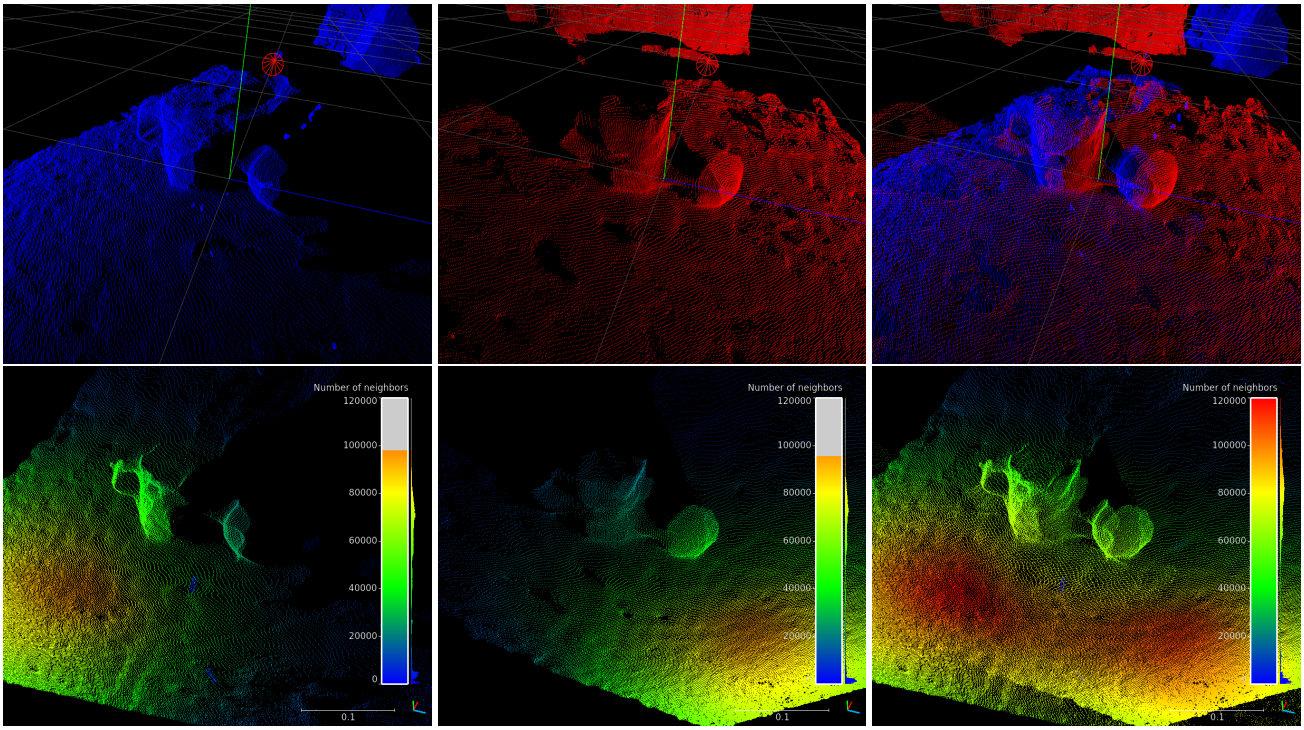

Figure 3.11: The scene of Figure 1.4 captured by two depth cameras. *Top row:* On the left the point cloud from the left dept camera tinted blue, in the middle the point cloud of the right image tinted red, and on the right both point clouds aligned. *Bottom row:* the same views as the top row but now colored with the same relative color scale for the number of neighbors. On the right side of the color bar a histogram is displayed.

- **Uniform:** The parts of the point cloud where the surface is flat should be uniformly sampled with centers spaced as far from each other as possible such that the flat surface can still be reconstructed. There should be no clustering of points as this slows down further computations.
- **Feature preserving:** The details and features on the sampled surface should remain after compression or have minimal distortion, for instance, close to the camera noise level.
- **Computational efficiency:** The compression algorithm must operate in real-time. Algorithms that can easily be adopted to a parallel version have an advantage.

One of the most straightforward point cloud compression algorithms is the voxel-grid filter as described in [24]. The voxel-grid filter is very computationally efficient, at least two magnitudes faster than other point cloud filtering methods described in [25].

For the voxel-grid filter a part of $\mathbb{R}^3$ is divided into an axis aligned regular grid of cubes or voxels. Based on their coordinates the points of the point clouds are assigned to a voxel. Then for each voxel the geometric center or centroid is calculated by averaging the position of the points in the voxel. For the placement of the off-surface points each point has a direction vector that is the direction of the camera that captured the point. The camera direction vector of the centroid is

the average of the camera direction vectors of all points assigned to the voxel. In Figure 3.12 an illustration of the working of voxel-grid filter is given.

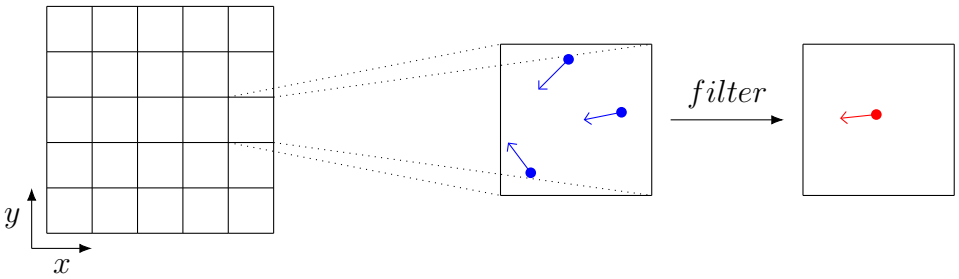

Figure 3.12: The principle of the voxel-grid filter: *Left:* Example of a axis aligned regular voxel-grid. *Middle:* Points (blue) and the vectors that are the direction to their camera are added to voxels based on their coordinates. *Right:* The centroid (red) of the points and the average camera direction are calculated and the original points are discarded.

Looking back at the ideal properties, the result of the voxel-grid filter are *uniform* because of the uniformity of the voxels. Applying a voxel-grid filter to a point cloud is also referred to as uniform sampling of a point cloud [24].
The voxel-grid filter is not feature preserving as the filter does not have feature detecting logic. When the voxel size is increased details in the surface are smoothed out. The voxel size is also bound by the CSRBF support radius σ. When the voxel size is too large the distance between filtered points may become larger than σ and CSRBF interpolation is not possible any more resulting in holes in the surface. If the voxel size is set to the noise level of the camera at the distance of the cobot base frame the filter will smooth out noise. The implementation of the voxel-grid is given in Listing 2.
The voxel-grid data structure is initialized when the program is started. If the set P contains n points then the loop on lines 2 to 15 takes $O(n)$ time. If the algorithm is implemented so that it keeps track of the indices of voxels that are accessed the statements on line 16 and 17 can be done in $O(n)$ time. It is not trivial to parallelize the voxel-grid filter. The points in P are not sorted in any way. If we divide P in subsets $P_1, \ldots, P_n$ where n is the number of threads and assign each subset to a thread then it can happen that a voxel is accessed by multiple threads a the same time. This would require locking of the voxels which introduces an efficiency penalty. The computation time of Algorithm 2 is so low however that it was not necessary to create a parallel version of the algorithm. The contribution of the voxel-grid filter to the total computation time and the choice of the voxel size will be discussed in Section 3.3.4.

3.2.3 Step 3: Efficient Filling of the Sparse Matrix Φ

To solve Equation 1.7 we first have to fill the matrix Φ. When RBFs of compact support are used and the support radius σ is chosen small enough the matrix Φ becomes sparse.

Algorithm 2: Voxel-grid filter for point cloud P

```
1  VoxelGridFilter(P, voxelGrid)
   Input: The set P of points and the list voxelGrid, the empty regular voxel-grid.
   Result: Filtered point cloud P
2  foreach p ∈ P do
3      Calculate the index i in voxelGrid based on the coordinates of p
4      Get the voxel v at voxelGrid[i]
5      if v is empty then
6          v.centroid ← p
7          v.cameraDirection ← p.cameraDirection
8          v.count ← 1
9      else
           // Get the number of points previously added
10         n ← v.count
           // Average the centroid position with that of p
11         v.centroid ← (n · v.centroid + p)/(n + 1)
           // Average the camera direction with that of p
12         v.cameraDirection ← (n · v.cameraDirection + p.cameraDirection)/(n + 1)
13         v.count ← n + 1
14     end
15 end
16 P ← all centroids stored in voxelGrid
17 clear voxelGrid
18 return P
```

To improve the efficiency of this step of the pipeline an activity diagram was made that breaks down the activities of this step. The idea is to leverage parallel processing for the algorithm in Listing 1 that is responsible for the filling of the Φ matrix. In Figure 3.13 a breakdown is shown.

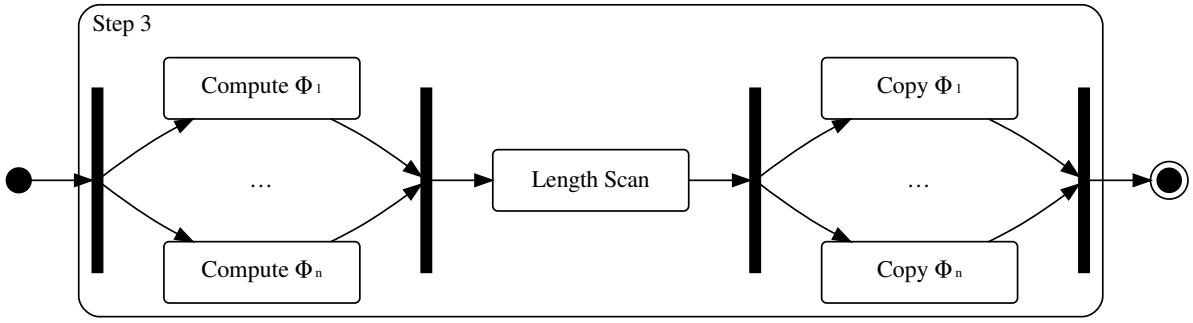

Figure 3.13: Detailed activity model of step 3 for n tasks.

The matrix Φ is split in n equal parts and each part is assigned to a parallel activity of type "Compute Φ" and "Copy Φ". Each part is a set of sequential rows of Φ. If the size of Φ is $m \times m$ then activity i, with $i \in \{1, \ldots, n\}$ and $\lceil m/n \rceil \geq n$, is assigned the rows with indices in the range

$$\left[(i-1)\left\lceil \frac{m}{n} \right\rceil + 1, \min\left\{m, i\left\lceil \frac{m}{n} \right\rceil\right\}\right]. \tag{3.12}$$

When the parallel activities are started we wait until all are finished, this is indicated by the vertical join bar in the activity diagram in Figure 3.13.

Listing 3 shows how the activity diagram is implemented. Each parallel activity owns partial *block value*, *block column indices*, and *block row indices* arrays that belong to the part of Φ that the activity will fill. Each activity fills their partial arrays and then we wait for all the parallel activities to finish. When these activities are finished the length of all partial arrays of the parallel tasks are known. The memory for the full arrays of Φ are then reserved and parallel copy tasks fill their part of the *block value*, *block column indices*, and *block row indices* arrays.

Algorithm 3: Compute the non-zero elements of Φ in parallel.

```
1  FillPhiParallel(Φ, P)
   Input: Empty sparse matrix Φ, list P with points
   Result: Filled Φ matrix
2  Populate the voxel-grid G with all points of P
3  do in parallel
4  |  Calculate the range r of indices in P to assign to current parallel task with Equation 3.12
5  |  FillPartOfPhi(Φ, G, P, r, s_b)
6  end
7  Wait for parallel activities to finish
8  Reserve memory for block and index arrays of Φ based on the sum of their sizes
9  Calculate the start indices for arrays based on the prefix sum of their sizes
10 do in parallel
11 |  Copy partial block and index arrays arrays to arrays of Φ
12 end
13 Wait for parallel activities to finish
14 return Φ
```

The computational complexity of `FillPhiParallel` depends on the computational complexity of the parallel tasks. As seen in the description of Listing 1 populating the spatial grid takes $O(n)$ time. The reserving of memory of Φ takes linear time and the calculation of the start indices is bound by the number of parallel activities. The parallel activities from line 10 to 12 are at most $O(n_{nzb})$ with n_{nzb} the total number on non-zero blocks in Φ. The parts of Φ are filled by the parallel activity `FillPartOfPhi`. Details of this activity are given in Listing 4.
On line 11 and 15 Algorithm 4 uses the index k to address the elements of a block. The elements of a block are numbered in row-major order and for a 2×2 block are

$$\begin{array}{|c:c|}\hline 0 & 1 \\ \hdashline 2 & 3 \\ \hline \end{array} \tag{3.13}$$

The block index k can then calculated from the row index i and the column index j of Φ with

$$k = (i \bmod s_b)s_b + (j \bmod s_b), k \in \{0, \ldots, s_b - 1\} \tag{3.14}$$

Algorithm 4: Compute the non-zero elements of the row indices in R

```
1  FillPartOfPhi(G, P, r, s_b)
   Input: Range r of row indices assigned to this thread, the spatial hash-grid G, list with all points P,
          and the block size s_b
   Result: Filled B
2  allocate memory for arrays B and M
3  l_prev ← −1
4  l ← 0
5  initialize M with −1
6  foreach p_i ∈ R do
7      Get the set Q from G, the set of all points of neighboring voxels.
8      foreach q_j ∈ Q do
           // i is the row index and j is the column index in Φ
9          if M(⌊j/s_b⌋) ≤ l_prev then
10             Block does not exist, create it and fill elements with zeros
11             k = (i mod s_b)s_b + (j mod s_b), k ∈ {0, ..., s_b − 1}
12             calculate Φ_ij as in Algorithm 1, line 8-10 and assign to element k of block B(l)
13             M(⌊j/s_b⌋) ← l
14             l ← l + 1
15         else
               // block exists
16             k = (i mod s_b)s_b + (j mod s_b), k ∈ {0, ..., s_b − 1}
17             fill element k of block B(⌊j/s_b⌋)
18         end
19     end
20     if (i mod s_b) = (s_b − 1) then
21         l_prev ← l − 1
22     end
23 end
```

where the variable s_b is the block size. The working of Algorithm 4 will be demonstrated by an example. Assume the matrix in 3.5 is part of a larger matrix and we want to store this part with `FillPartOfPhi`. For convenience the matrix in 3.5 is repeated here first.

$$\left[\begin{array}{cc:cc:cc} 3 & 8 & 0 & 0 & 0 & 0 \\ 11 & 0 & -8 & 0 & 0 & 0 \\ \hdashline 0 & 7 & 1 & 0 & 0 & 0 \\ 0 & 0 & 5 & -4 & -2 & 0 \\ \hdashline 0 & 0 & 5 & 0 & -7 & 8 \\ 0 & 0 & 0 & 14 & 9 & 6 \end{array}\right] \tag{3.15}$$

The values of the block array B, scratch array M, length counter l, and previous length counter l_{prev} after the execution of line 2 to 5 is

$$
\begin{aligned}
\mathrm{B} &= \\
\mathrm{M} &= \begin{array}{|c|c|c|}\hline \text{-1} & \text{-1} & \text{-1} \\ \hline \end{array} \\
l = 0, &\quad l_{prev} = -1
\end{aligned}
\tag{3.16}
$$

After processing the the first two values in the first block of the matrix the state becomes

$$
\begin{aligned}
\mathrm{B} &= \begin{array}{|cc|cc|cc|cc|cc|}\hline 3 & 8 & & & & & & & & \\ 11 & 0 & & & & & & & & \\ \hline \end{array} \\
\mathrm{M} &= \begin{array}{|c|c|c|}\hline 0 & \text{-1} & \text{-1} \\ \hline \end{array} \\
l = 1, &\quad l_{prev} = -1
\end{aligned}
\tag{3.17}
$$

When the whole first row is processed the check on line 19 is executed. Because this is not the last row of the individual blocks the condition is false and the state does not change. Processing all elements of the second row however will cause line 20 to be executed and l_{prev} is set.

$$
\begin{aligned}
\mathrm{B} &= \begin{array}{|cc|cc|cc|cc|cc|}\hline 3 & 8 & 0 & 0 & & & & & & \\ 11 & 0 & \text{-8} & 0 & & & & & & \\ \hline \end{array} \\
\mathrm{M} &= \begin{array}{|c|c|c|}\hline 0 & 1 & \text{-1} \\ \hline \end{array} \\
l = 2, &\quad l_{prev} = 1
\end{aligned}
\tag{3.18}
$$

Processing the third and fourth row of the matrix gives

$$
\begin{aligned}
\mathrm{B} &= \begin{array}{|cc|cc|cc|cc|cc|}\hline 3 & 8 & 0 & 0 & 0 & 7 & 1 & 0 & 0 & 0 \\ 11 & 0 & \text{-8} & 0 & 0 & 0 & 5 & \text{-4} & \text{-2} & 0 \\ \hline \end{array} \\
\mathrm{M} &= \begin{array}{|c|c|c|}\hline 2 & 3 & 4 \\ \hline \end{array} \\
l = 5, &\quad l_{prev} = 4
\end{aligned}
\tag{3.19}
$$

It becomes clear that because array M is not cleared but reused for every iteration of the outer loop we have to store the value of $l-1$ in l_{prev} to determine if the block already exists or not. Finally the state is

$$
\begin{array}{l}
\mathrm{B} = \\
\\
\mathrm{M} = \\
l = 7, \quad l_{prev} = 6
\end{array}
\begin{array}{|cc|cc|cc|cc|cc|cc|cc|}
\hline
3 & 8 & 0 & 0 & 0 & 7 & 1 & 0 & 0 & 0 & 5 & 0 & -7 & 8 \\
11 & 0 & -8 & 0 & 0 & 0 & 5 & -4 & -2 & 0 & 0 & 14 & 9 & 6 \\
\hline
\end{array}
\qquad (3.20)
$$

$$
\begin{array}{|c|c|c|}
\hline
2 & 5 & 6 \\
\hline
\end{array}
$$

and the array B is filled with the blocks that were assigned to this parallel process. The *block column indices* and *block row indices* are maintained as well, they will contain the values of the block row and block column indices of the inserted blocks. The computational complexity of `FillPartOfPhi` is $O(n_{nz})$ with n_{nz} the number of non-zero elements added to Φ by this parallel activity. If the number of points in neighboring voxels and the voxel itself is low the computational complexity approaches $O(n)$ with n the number of points assigned to the parallel task. This is because each voxel has at most 26 neighbors and voxels at the border of the voxel-grid have less neighbors than that. Finally line 11 in Listing 3 of `FillPhiParallel` is responsible for combining the B, partial *block column indices* and partial *block row indices*.

The functionality of `FillPhiParallel` and `FillPartOfPhi` is added to the `SparseMatrix` implementation of the programming library of the Parallel Linear Solver, or PaLinSo for short. The PaLinSo library contains the CG solver that will be discussed in the next xection. The programming interface for clients of the PaLinSo library is kept relatively simple with the filling of parts of the matrix done via a call-back interface. The number of parallel activities can be set by the client via a parameter.

3.2.4 Step 4: Parallel Conjugate Gradient Solver

To improve the efficiency of the solving of the linear system of equations $\Phi\omega = g$ the approach was to look at a CG solver that is capable of efficient parallel processing. The CG method is hard to parallelize because of its low arithmetic intensity and high memory-bandwidth requirements. The PaLinSo[1] solver is a scalable parallel iterative CG solver that can utilize the CPU or GPU to solve a wide range of linear problems. On the CPU the solver uses low-level SSE instructions to accelerate sparse-matrix vector multiplications beyond the efficiency possible with higher-level programming languages. It is also possible to change the number of threads employed so that the optimal number can be chosen for the current linear problem. PaLinSo utilizes the BCSR storage format to map blocks to sparse-matrix vector multiplications with the multiple block-row mapping. This mapping optimizes memory-bandwidth so that the amount of wasted computational resources is decreased. It is important however to chose the right block size as for different linear problems different block sizes are optimal. To choose the optimal configuration for the PaLinSo

[1] `https://code.google.com/archive/p/palinso/source`

solver we will look at the block size and the number of threads. The linear problem is too small to benefit from the use of the GPU. To let the CG solver converge faster we precondition the solver with the Jacobi preconditioner.

3.3 Results

3.3.1 Step 1: Synchronized Acquisition

Ideally all depth cameras capture a depth image at the same time. To synchronize the shutters of RealSense cameras a synchronization cable was constructed as described in the Intel RealSense multiple camera white paper [22] and the integrity of the cable was tested. With software commands one camera was set to master and the other was set to slave and synchronized capture was started. The master sends the signal to capture a depth image to all slaves, the slaves respond with an acknowledge signal. In our experiment the RealSense software development kit (SDK) however reported that the master did not receive the acknowledge within the required 2 milliseconds.

Since the synchronization cable did not work the approach without hardware synchronization as described by the Intel RealSense multiple camera white paper [22] is used. While the cable was thoroughly checked for faults the master camera reported that the slave camera did not respond within the required time of $2\,\mathrm{ms}$. The capture of frames is started in parallel activities and a supervisory activity collects all frames and computes the environment representation. Listing 5 shows the supervisory activity.

Algorithm 5: Compute the non-zero elements of the row indices in R

```
1 MainLoop(Cameras)
  Input: The set Cameras with started depth cameras
2 foreach camera ∈ Cameras do
3 |   Start the parallel activity Capture(camera), see Listing 6
4 end
5 while true do
6 |   barrier1.wait()
7 |   barrier2.wait()
8 |   Calculate and visualize the implicit surface, i.e. step 2-5 from Figure 1.16
9 end
```

First a depth image capture and post-processing activity is started for each camera on line 2-4. The while loop after that uses two synchronization barriers. The barrier is a synchronization primitive used to synchronize multiple parallel activities. An activity can only pass a barrier when all other activities have also reached the same barrier [26]. When started the `Capture` parallel activity halts at `barrier1` as can bee seen in Listing 6.

When `MainLoop` reaches `barrier1` the `Capture` activities can proceed and each camera captures a depth image. `MainLoop` waits at `barrier2` until the post-processing of all depth images is fin-

Algorithm 6: Main loop for the reconstruction of the environment.

```
1 Capture(camera)
  Input: A camera object
2 while true do
3 |   barrier1.wait()
4 |   acquire a point cloud from camera
5 |   Apply decimation and temporal filter, see Figure 3.9.
6 |   barrier2.wait()
7 end
```

ished and all point clouds are calculated. When all `Capture` activities synchronize at `barrier2` the `MainLoop` activity can proceed to calculate the surface representation of the point clouds and surface visualization while the `Capture` activities wait at `barrier1` again. When the `MainLoop` activity finishes we have executed the full sequence of step 1 through 5 and can start the next iteration.

To verify the claim that with the described method the time stamps of captured frames never misalign by more than one frame [22] the global time stamps of each captured frame are recorded for a length of time and it was checked that

$$\Delta T < 2/f_s. \tag{3.21}$$

The variable f_s is the sample rate and ΔT is the difference in time between two parallel captured frames. In this experiment $f_s = 30\,\mathrm{Hz}$ and thus $2/f_s = 0.066\,67\,\mathrm{s}$. The results are found in Table 3.1.

Measure	ΔT **[s]**
Average	0.01999
Minimum	0.01311
Maximum	0.04666

Table 3.1: Different measures for 98 samples of ΔT.

For all 98 samples the value of ΔT was indeed below $2/f_s$ since the maximum value of the set is smaller than $2/f_s$.

3.3.2 Step 2: Fusion, Compression, and the Number of Non-zeros in Φ

Relation between the point cloud density and the computation time: In experiments done in [2] it was seen that the closer objects are to the camera the lower the reconstruction speed. The distance of objects from the camera was kept above a minimum so that the required reconstruction speed of $0.1\,\mathrm{s}$ could be achieved. This shortcoming of the original pipeline makes it not very useful in a practical situation and the intention is to add more depth cameras and increase the environment reconstruction speed.

In this section we look if compression can increase the reconstruction speed so that more depth cameras can be added. We then investigate if the voxel-grid filter can keep the reconstruction speed constant when objects are close to the depth camera. In the first experiment the surfaces of several planar point clouds are generated with increasing distance between the points d_p as shown in Figure 3.14.

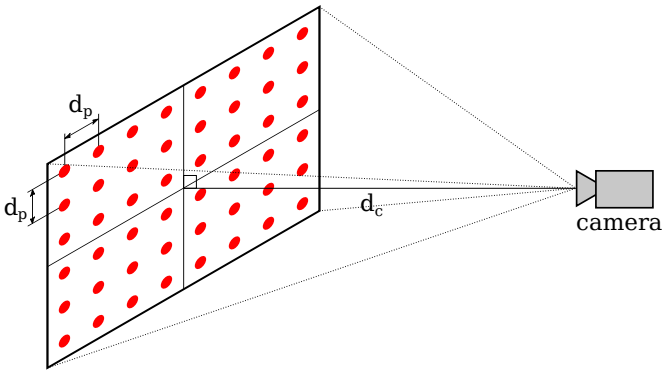

Figure 3.14: Example of a point cloud parallel to the camera view plane. The points of the point cloud are the red dots. Distance between adjacent centers is d_p, distance of camera center to plane is d_c.

Because size of the point cloud is 80×60 points, because we know the vertical FOV or α_y of the camera, and we know d_p we can calculate the camera distance d_c with Equation 3.1. For the values of d_p used the values of d_c were calculated and the number of neighbors counted as in Figure 3.4. The results can be found in Table 3.2.

d_p **[m]**	d_c **[m]**	number of neighbors $\sigma = 0.17\,\mathrm{m}$
0.005	0.244	3625
0.010	0.488	901
0.015	0.732	405
0.020	0.976	225
0.025	1.221	145
0.030	1.465	101
0.035	1.709	69
0.040	1.953	61

Table 3.2: Value of d_c calculated from d_p with Equation 3.1. The last column contains the number of neighbors for a point near the center of the plane within a radius of $0.17\,\mathrm{m}$ around the point.

The values in this table demonstrate an important fact, the number of neighbors grows hyperbolically with a singularity when $d_p = 0$. Hyperbolic growth of computational complexity is extremely bad because, unlike exponential growth, it reaches its singularity in finite-time. The total number of non-zeros that end up in Φ are plotted against the set of point clouds where $d_p = \{0.005, 0.010, 0.015, 0.020, 0.025, 0.030, 0.035, 0.040\}$ in Figure 3.15.

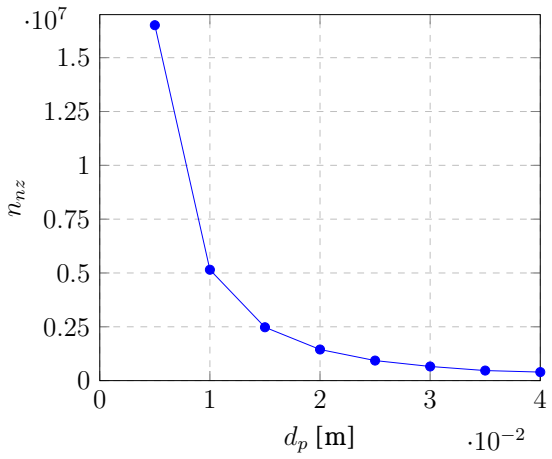

Figure 3.15: Point distance plotted against the number of non-zeros added to Φ in step 2.

The dataset described above was used to measure the performance of the original pipeline from [2] and the computation time T was measured for step 3 and 4. The computation time explodes for low values of d_p as can be seen in Figure 3.16.

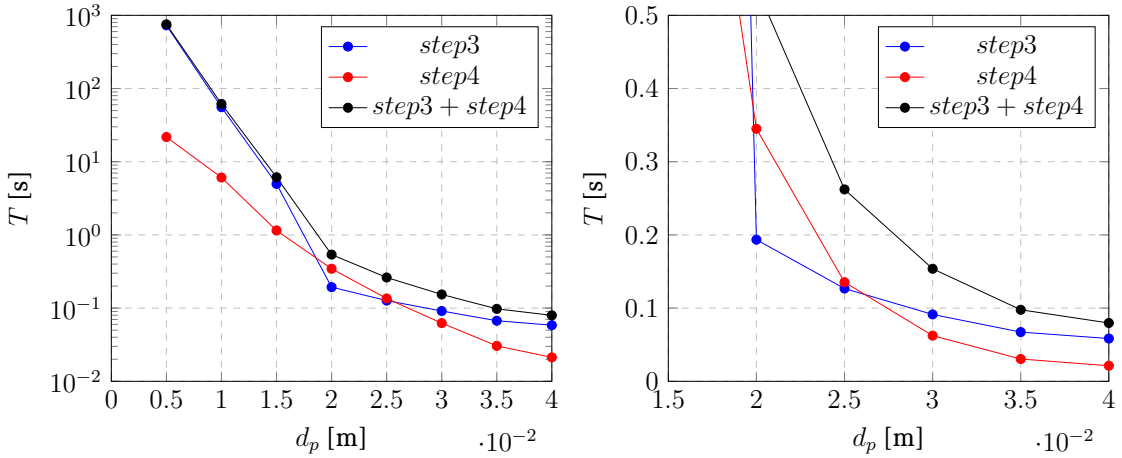

Figure 3.16: Computation time of step 3, step 4, and the sum of step 3 and 4 plotted against d_p. *Left:* The full plot, be aware that the vertical axis has logarithmic scale. *Right:* Zoomed in plot of $d_p = [0.015, 0.04]$.

In this experiment to have a surface reconstruction time of $10\,\mathrm{Hz}$ a d_p of at least $0.03\,\mathrm{m}$ is necessary which is equivalent to a camera distance of $1.465\,\mathrm{m}$. The point cloud density becomes even worse when more cameras are added and point clouds overlap as shown in Figure 3.11. To maintain or even improve the reconstruction speed the voxel-grid filter was added and the voxel-size was set to $0.03\,\mathrm{m}$. Figure 3.17 shows the result of this experiment.
The reconstruction time in this experiment can approximately be kept less than or equal to the the processing time of that of the point cloud where $d_p = 0.03\,\mathrm{m}$ by applying a voxel-grid filter with a voxel size of $0.03\,\mathrm{m}$. The plot in Figure 3.17 behaves the same as the plot on the right in Figure 3.16 for $d_p \geq 0.03\,\mathrm{m}$. For for $d_p < 0.03\,\mathrm{m}$ the voxel-grid filter starts to collapse points to

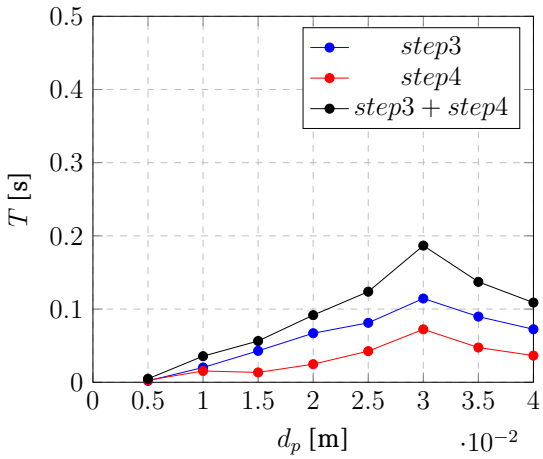

Figure 3.17: Computation time of step 3, step 4, and the sum of step 3 and 4 plotted against d_p for a voxel-grid filter with voxel size $0.03\,\mathrm{m}$.

their centroids. Instead that T explodes for $d_p < 0.03\,\mathrm{m}$ there is a maximum at $d_p = 0.03\,\mathrm{m}$. The smaller d_p the higher the compression and since the number of points in the dataset is constant the reconstruction time will decrease with the decrease of d_p.

In the next experiment we will compare the performance of the original solution from [2] with the final pipeline with compression on the same stream of two aligned point clouds. The voxel-grid voxel size was set to $0.03\,\mathrm{m}$, $\sigma = 0.17\,\mathrm{m}$, and off-surface points were generated in front of the surface on the camera rays. In the stream a subject moves close to one or both cameras to stress the pipeline. An impression of the most important frames of the stream is shown in Figure 3.18.

Figure 3.18: Interesting frames from the dataset recorded to test compression. The top row is the left depth camera, the bottom row the right camera. Column 1: frame 0, empty scene. Column 2: frame 176, pose in center. Column 3: frame 279, hand for left camera. Column 4: frame 411, body for right camera. Column 5: frame 504, hands for both cameras.

For both the original pipeline and the fully multi-threaded pipeline with PaLinSo and compression the time to calculate each frame was recorded for as long as possible and Figure 3.19 shows the result.

The computation time per frame of the original pipeline and parallel pipeline without compression explodes the moment a subject moves close to one of the cameras. For the last frames of their plot

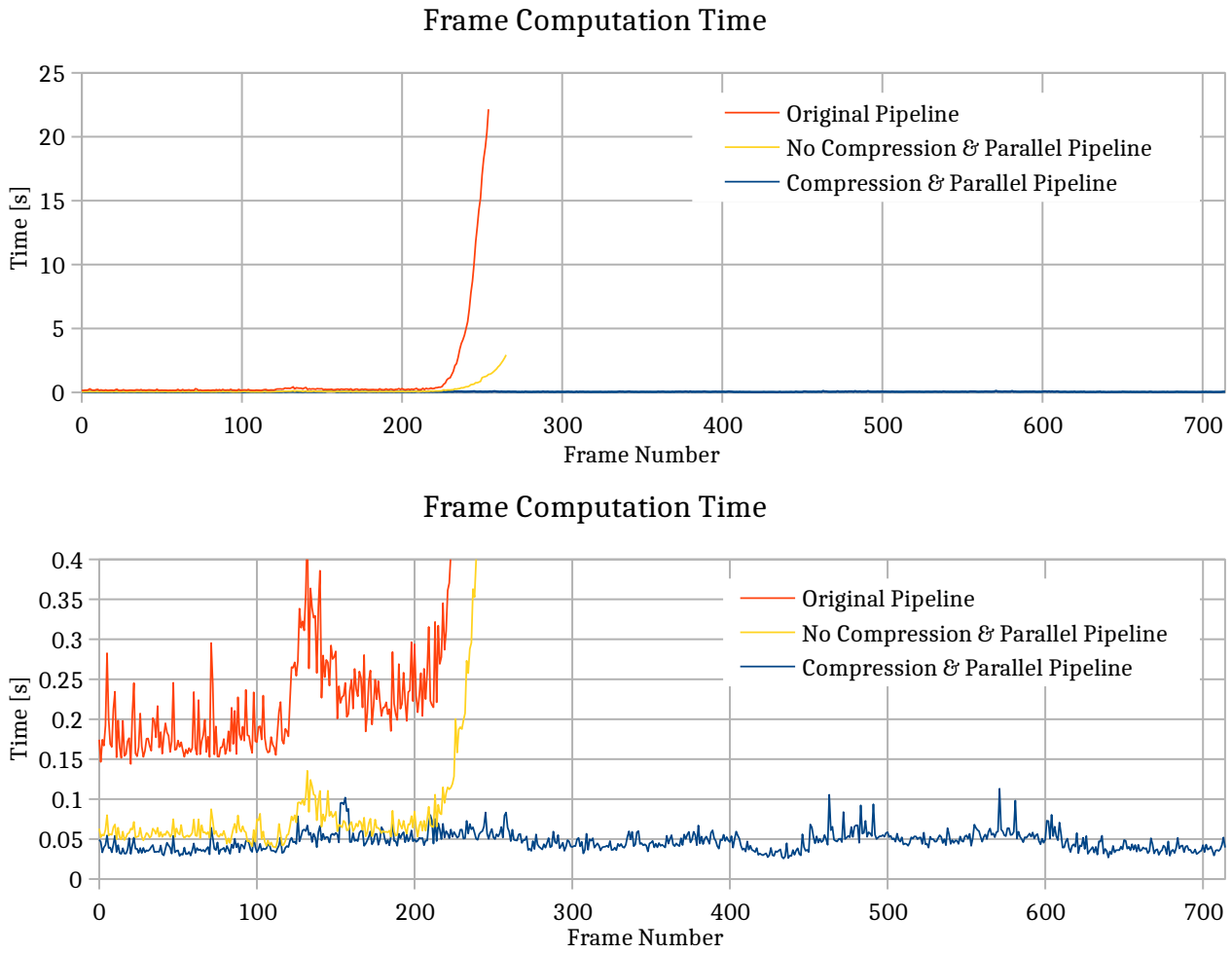

Figure 3.19: *Above:* Computation time for each frame for the original and the new parallel pipeline. The moment a subject moves close to the camera the computation time explodes. The blue line is entirely on the x-axis. *Below:* Zoomed in version of the graph.

both the Eigen CG and the PaLinSo solver were not able to converge to the tolerance of $1 \cdot 10^{-4}$ within 1000 iterations. The processing of the stream was stopped in those cases. The new pipeline on the other hand has a computation time per frame close to the x-axis. In zoomed it becomes clear that even with the extreme cases that are in the stream the parallel pipeline with compression has a stable frame computation time of $\approx 0.05\,\mathrm{s}$ per frame with a few outliers to around $0.1\,\mathrm{s}$. The original pipeline on the other hand has higher computation times per frame with much more variance and it grinds to a halt when a subject moves close to one of the cameras. The parallel pipeline with compression has a slightly higher frame computation time and explodes at the same moment the original pipeline does. In Figure 3.20 the compression ratio of each frame in the stream is plotted for the new parallel pipeline with compression. The compression ratio is defined as the the number of points after compression divided by the number of points before compression.

As can be seen the compression ratio is low, i.e. $< 10\%$ when subjects are not close to the camera. The compression ratio increases when the subject moves close to the camera around frame 279, 411, and 504 as highlighted in Figure 3.18.

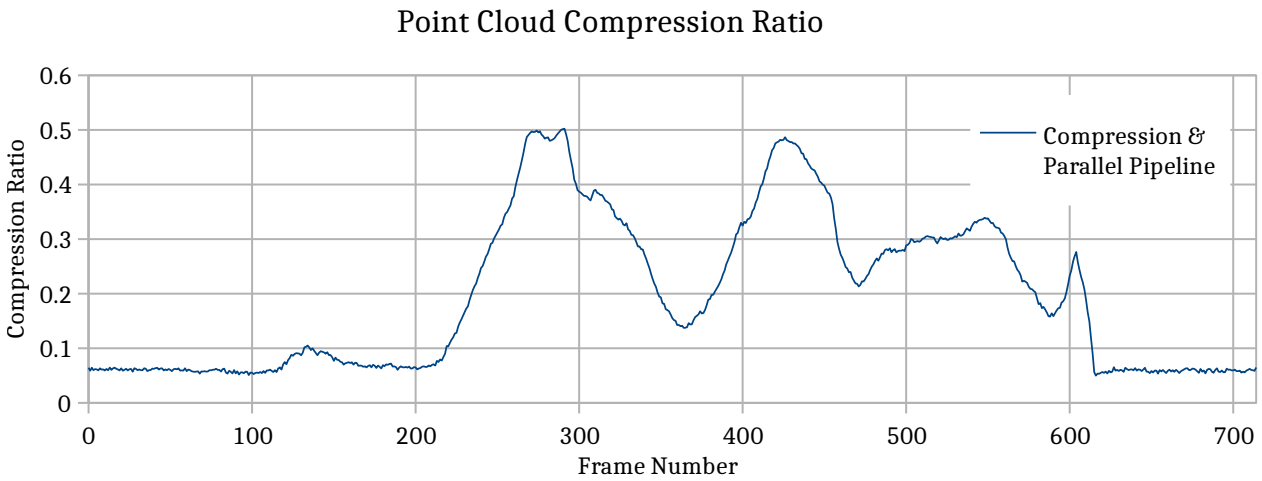

Figure 3.20: Compression ratio for the frames in the stream.

To show the effect of compression the of the "soccer ball on box" scene of dataset in Appendix D with the soccer ball at $0.8\,\mathrm{m}$ is reconstructed with the original pipeline and the CSRBF parameters from Table 1.1. The reconstructed surfaces are shown in Figure 3.21. To increase the accuracy of the generated surface mesh the marching cubes voxel size was set to $0.01\,\mathrm{m}$.

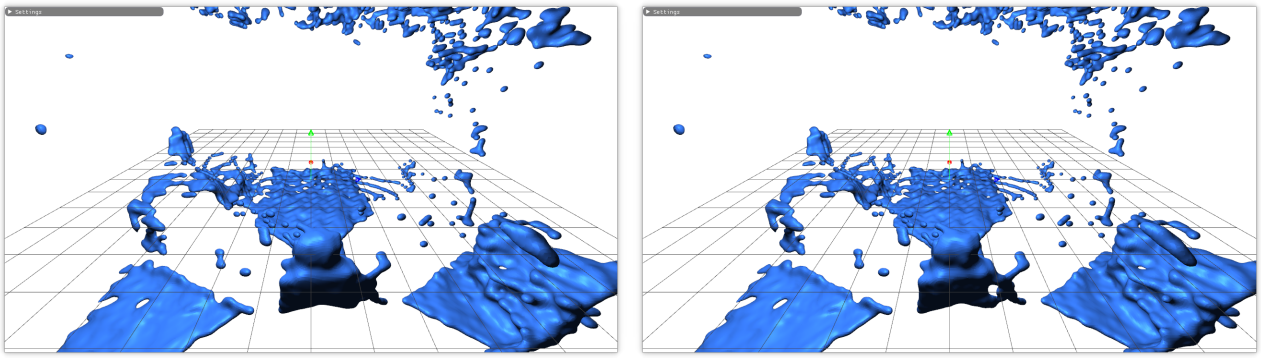

Figure 3.21: Comparison of the original pipeline with the original CSRBF parameters and the dataset in Appendix D. *Left:* No voxel-grid filter. *Right:* With voxel-grid filter. The point clouds are almost identical.

The size of the voxels of the voxel-grid filter was set to $0.04\,\mathrm{m}$. This resulted in a compression ratio of 0.136. As can be seen there is some slight distortion in the surface of soccer ball but this seems to be lower than the noise on surrounding objects in the environment like the cabinet to the right of the soccer ball. Of both reconstructed surfaces f was evaluated for a planar grid parallel to the ground and through the center of the soccer ball. The centers in a slab of $0.05\,\mathrm{m}$ above and below the plane were projected on the plane and plotted together with the level-sets $f = \{0.01, 0.02, 0.03, 0.04, 0.05, 0.06\}$. The result is shown in Figure 3.22. As can be seen the number of centers used to reconstruct the ball is substantially lower while the level-sets of both look practically the same.

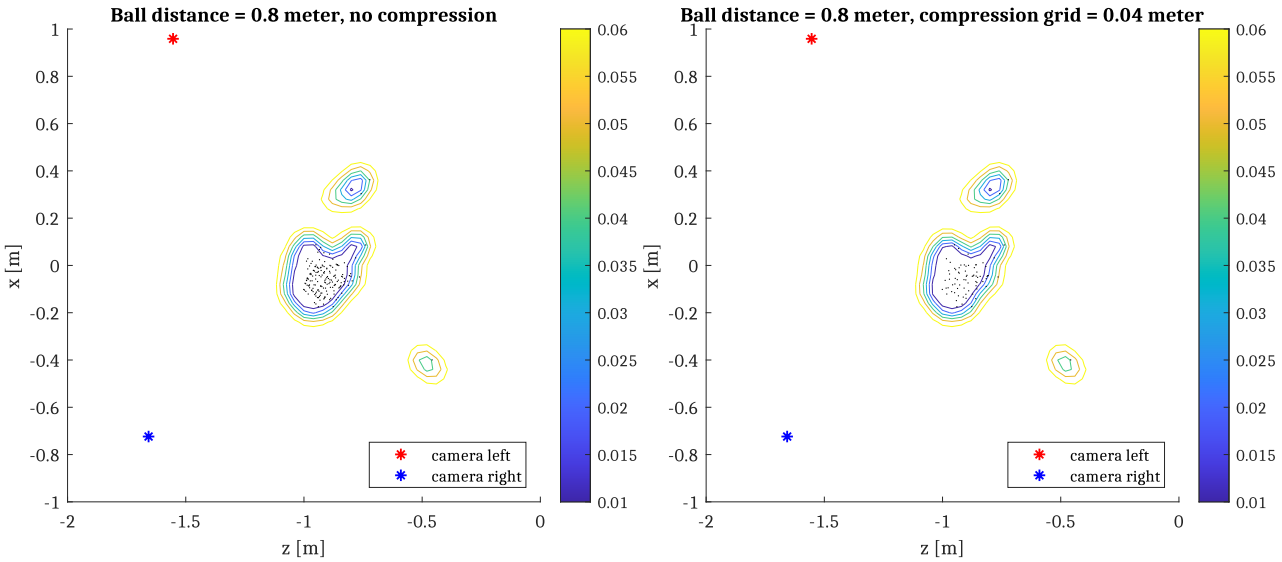

Figure 3.22: Iso-surface levels of a horizontal plane through the center of the soccer ball. The centers are the black dots. *Left:* No voxel-grid filter. *Right:* With voxel-grid filter. The level-sets look very similar while fewer centers are used in the creation of the right level-sets.

3.3.3 Choosing the Optimal BCSR Block Size for Step 4

The block size used for the BCSR matrix Φ that is part of the linear system 1.8 solved by the PaLinSo CG solver is configurable. The goal of the experiment in this section is to determine the optimal block size to be used in further experiments. To determine the optimal block size the Φ matrix of the dataset from Section 3.3.2 where $d_p = \{0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.4\}$ were stored to disk in matrix market exchange format from within the new pipeline. The values of these matrices were loaded in the test suite of the PaLinSo library. For each of the matrices the computation time for a BCSR block size of 1, 2, 4, and 8, and a number of threads of 0 (no separate thread), 1, 2, 4, 8, 16, and 32 threads were recorded. The hardware in Appendix A was used which has 4 physical and 8 logical cores. The results can be found in Table 3.3. A BCSR block size of 2 is optimal in combination with 8 threads especially for decreasing d_p which means more non-zeros. The experiments in the next sections will therefore all be with a block size of 2.

3.3.4 Multi-threading of Step 3 and 4

The main goal of this experiment is to measure the effect of the amount of parallel activities on the efficiency of step 3 and 4 on a reference dataset and find the optimal configuration. We test the following thread configurations: all activities are done on the same thread, i.e. single-threaded, the activity is done on 1 separate thread, 2 threads, 4 threads, and 8 threads. The voxel-grid filter in step 2 is set to a voxel size of $0.03\,\mathrm{m}$. Another goal is to benchmark the original pipeline against the new pipeline. Compression was also added to the first step of the original pipeline and benchmarked. An aligned point cloud stream was recorded by two depth cameras, in the dataset

block size	number of threads	d_p [m]						
		0.01	0.015	0.02	0.025	0.03	0.035	0.04
		T [s]						
1	0	11.237	2.379	0.747	0.314	0.141	0.075	0.047
1	1	11.531	2.377	0.790	0.317	0.156	0.073	0.048
1	2	7.271	1.569	0.546	0.210	0.097	0.046	0.032
1	4	6.524	1.384	0.582	0.179	0.088	0.045	0.028
1	8	6.353	1.327	0.585	0.165	0.077	0.039	0.026
1	16	6.255	1.380	0.606	0.169	0.091	0.042	0.028
1	32	6.315	1.506	0.616	0.173	0.105	0.044	0.036
2	0	3.575	0.788	0.246	0.107	0.053	0.030	0.020
2	1	3.418	0.787	0.246	0.108	0.056	0.030	0.021
2	2	2.129	0.464	0.165	0.069	0.043	0.021	0.013
2	4	1.922	0.411	0.122	0.051	0.025	0.015	0.011
2	8	1.897	0.396	0.115	0.049	0.025	0.013	0.008
2	16	2.165	0.424	0.127	0.059	0.027	0.015	0.009
2	32	2.142	0.464	0.143	0.062	0.032	0.017	0.011
4	0	3.019	0.689	0.255	0.111	0.050	0.031	0.019
4	1	2.989	0.696	0.256	0.113	0.056	0.033	0.020
4	2	2.043	0.502	0.180	0.084	0.034	0.021	0.013
4	4	2.071	0.462	0.160	0.071	0.030	0.020	0.011
4	8	1.885	0.461	0.168	0.080	0.030	0.026	0.012
4	16	1.972	0.682	0.167	0.078	0.032	0.021	0.011
4	32	2.121	0.445	0.170	0.083	0.034	0.023	0.013
8	0	3.526	0.920	0.313	0.138	0.068	0.038	0.027
8	1	3.357	0.975	0.313	0.159	0.074	0.040	0.029
8	2	2.329	0.624	0.222	0.093	0.047	0.028	0.023
8	4	2.080	0.510	0.202	0.085	0.042	0.023	0.025
8	8	1.968	0.569	0.208	0.110	0.042	0.020	0.020
8	16	1.994	0.613	0.210	0.136	0.042	0.023	0.020
8	32	2.281	0.834	0.203	0.149	0.045	0.025	0.021

Table 3.3: Computation times of solving the linear system of equations, i.e. step 3, with PaLinSo for different block sizes, number of threads, and value of d_p. The computation times are conditionally formatted, red is bad and the greener the value the better.

a subject at a distance of $\approx 1.5\,\mathrm{m}$ from the camera moves an arm. The dataset has 724 aligned frames captured by 2 depth cameras. All measurements can be found in Appendix C. Several measures were calculated for the frames in the dataset. The first is the number of points of each frame and the plot is in the left of Figure 3.23.

The difference in number of points for the compressed and the uncompressed point cloud is not big, around 2.3% on average. This is quite low and not many points are lost. It will have a noticeable effect on the processing time of point clouds however. In the right of Figure 3.23 the number of non-zeros that are added to Φ are shown.

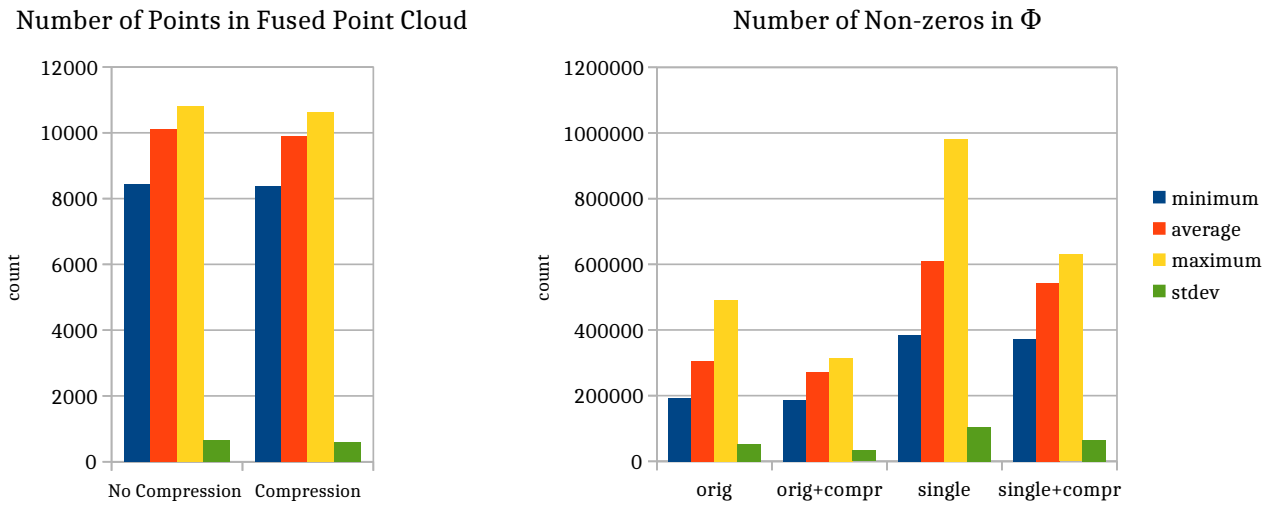

Figure 3.23: *Left diagram:* Minimum, average, maximum, and the standard deviation of the number of points of all the frames in the stream after step 2 with and without compression. *Right diagram:* Minimum, average, maximum, and the standard deviation of the number of non-zeros in Φ for the original pipeline (`orig`) and the new pipeline (`single`), both with and without compression (`compr`). The abbreviations on the horizontal axis are described in Appendix C.

The first important thing to notice is that the the values for the new pipeline are much higher, about twice as high. This is because the `SparseMatrix` data structure of the PaLinSo library has no special mode for symmetric matrices. The Eigen sparse matrix data structure used by the original pipeline requires only the upper or lower triangle to be filled. The compression has more effect on the number of non-zeros: while the number of points was compressed 2.3% on average the number of non-zeros are compressed 10.8%. The maximum number of non-zeros is reduced by 37.2%. Thus a small reduction in the number of points can have a significant reduction in the number of non-zeros that end up in the Φ matrix.

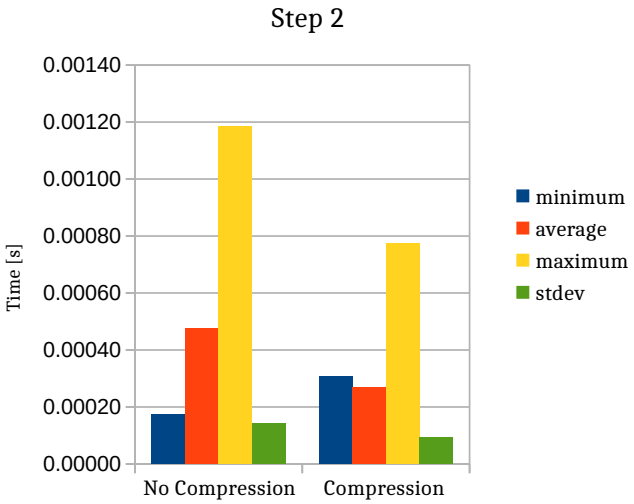

Figure 3.24: Minimum, average, maximum, and the standard deviation of step 2 with and without compression.

This is because the voxel-grid filter makes the point cloud more uniform, dense clusters of points

are reduced to a centroid per voxel decreasing the number of non-zeros in Φ. This reduction increases the number of point clouds that can be processed per second.

First we will compare the timings of step 2. The same implementation for step 2 was used for the original pipeline and the new pipeline. Compression for step 2 can be switched on or off in the new pipeline. In Figure 3.24 the measured computation times are displayed.

No effort has been put in creating a parallel version of step 2 because the computation time of step 2 is not a significant portion of the processing time of the rest of the pipeline. In Figure 3.25 the portion step 2 has on average without compression of the original pipeline computation time, and with compression on the new pipeline parallelized with 8 threads.

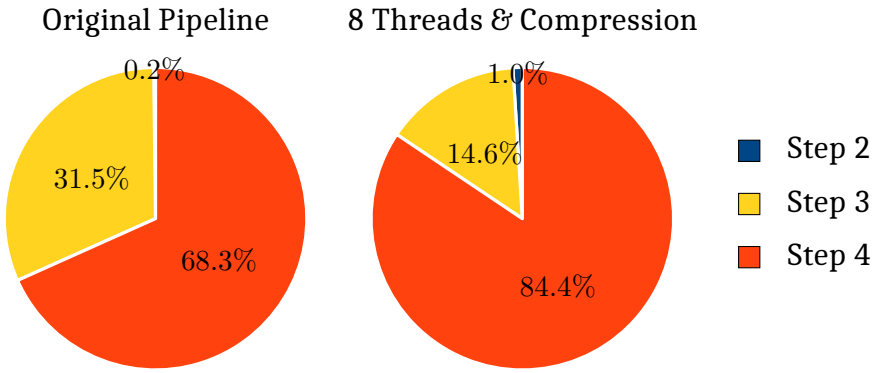

Figure 3.25: Pie charts of the timings of step 2, 3 and 4 of the original pipeline without compression and the new pipeline with compression running on 8 threads.

Then the computation times of step 3, the filling of the Φ matrix, for the original and the new pipeline were compared. The new pipeline is tested in single threaded mode, and in multi-threaded mode with 1, 2, 4, and 8 threads respectively. The results of this comparison can be found in Figure 3.26

Interesting to see is the effect of compression on outliers. In the original pipeline, both the maximum frame computation time as well as the standard deviation of the computation time are reduced, making the frame computation time is much more stable. The multi-threaded implementation of step 3 is in all cases faster than the implementation in the original pipeline or the single-threaded implementation.

On average the filling of Φ is a bit slower with 8 threads and compression than without compression. Most likely this is because the load is balanced differently. The parallel filling of Φ takes as long as the longest of all parallel activities. Because of the compression each parallel activity is assigned an other range of rows to calculate, if one of the parallel activities now receives a batch of rows that take longer to calculate then the whole activity takes more time. The parallel activities of step 3 showed a lot of variance in computation time. Load balancing might be an option to improve the efficiency of this activity further.

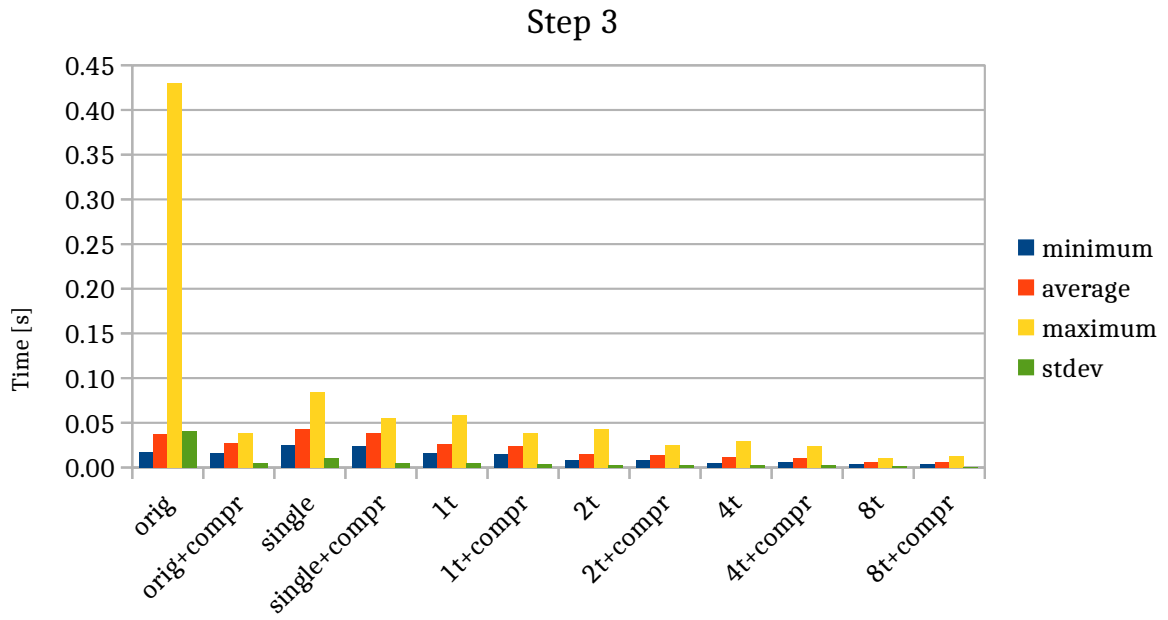

Figure 3.26: Minimum, average, maximum, and the standard deviation of step 3, for the original pipeline and the new pipeline in single-threaded and multi threaded mode with 1, 2, 4, and 8 threads. The abbreviations on the horizontal axis are described in Appendix C.

The computation time of the last activity, the solving of the linear equation, is also measured with the same configurations as the previous activity. The results are shown in Figure 3.27.

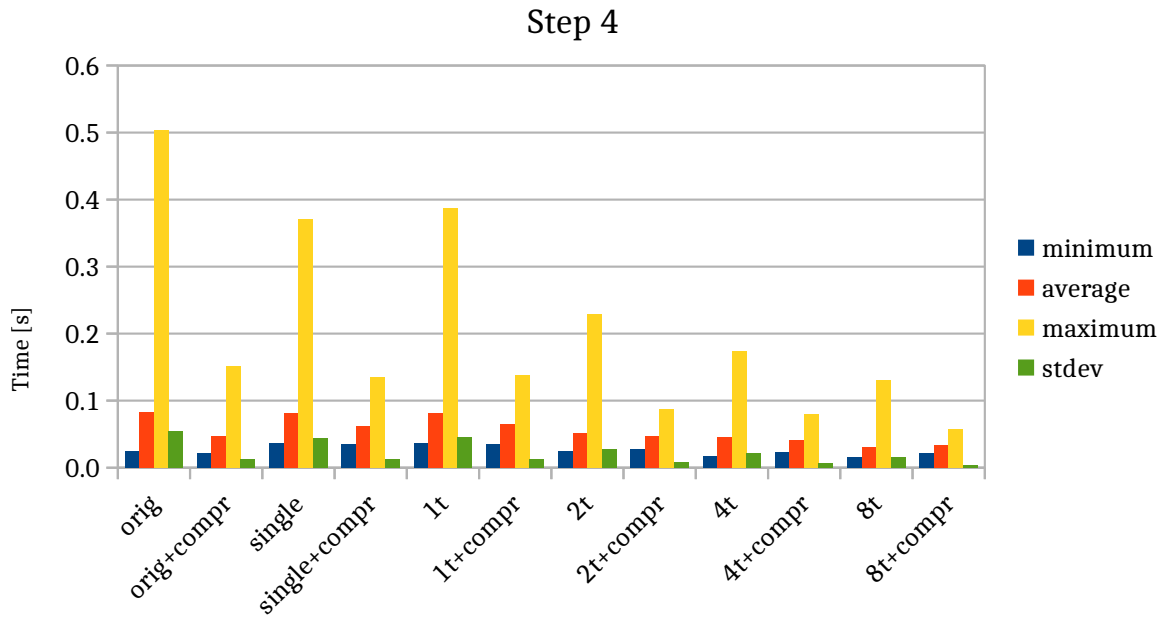

Figure 3.27: Minimum, average, maximum, and the standard deviation of step 4, for the original pipeline and the new pipeline in single-threaded and multi threaded mode with 1, 2, 4, and 8 threads. The abbreviations on the horizontal axis are described in Appendix C.

Also in the case of step 4 it holds that the more parallel activities the better. With step 4 we see, the same as with step 3, that solving the system of equations is a bit slower with 8 threads with compression than without compression. This might be because of an imbalance of arithmetic intensity

of the parallel activities as also here the duration is dictated by the duration of the parallel activity that takes the longest. Disabling compression will however impact the robustness. The PaLinSo solver, as the Eigen solver will grind to a halt when too many non-zeros are added because a subject or obstacle is near a depth camera. Compression also reduces the standard deviation of the frame rate, this is definitely an advantage because it makes the processing time per point cloud stable.

Figure 3.28, shows the total time of all three steps with and without compression. The maximum processing time of a frame is lower however and stays below $0.1\,\mathrm{s}$, thus a lower bound on the frame rate of $10\,\mathrm{Hz}$ can be guaranteed.

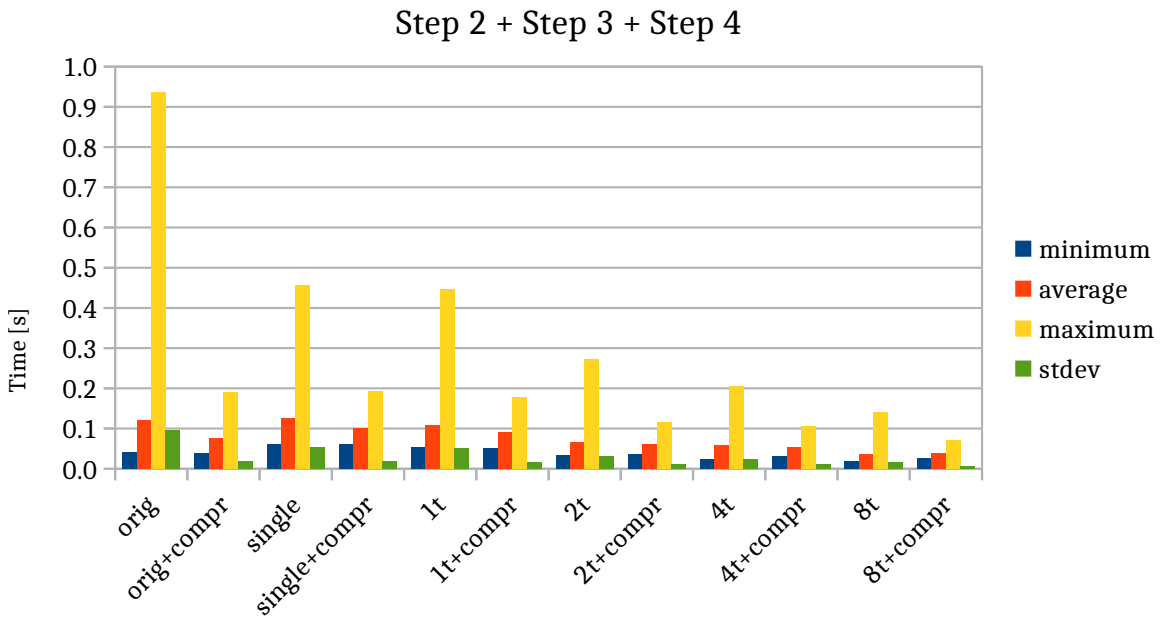

Figure 3.28: Minimum, average, maximum, and the standard deviation of the sum of step 2 to 4, for the original pipeline and the new pipeline in single-threaded and multi threaded mode with 1, 2, 4, and 8 threads. The abbreviations on the horizontal axis are described in Appendix C.

The Average processing time for the new pipeline with 8 threads and compression is $25.7\,\mathrm{Hz}$ on average with a lower bound of $14.3\,\mathrm{Hz}$. The standard deviation is low in this case which shows that the frame rate is quite stable. Figure 3.29 shows the cumulative computation time of the three steps for all modes.

The dataset consist of 724 depth image pairs sampled at a rate of $30\,\mathrm{Hz}$ by two depth cameras. The new pipeline is capable to process these images at a speed close to that which is a good improvement of efficiency. The original pipeline takes 3 times as much time to process the same frames. For this dataset a speed improvement of $3\times$ is achieved with 2 depth cameras.

To give an impression of the deviation and variance of the of the timings histograms with 10 bins were created for the timings of step 2, 3, and 4. The result is found in Figure 3.30.

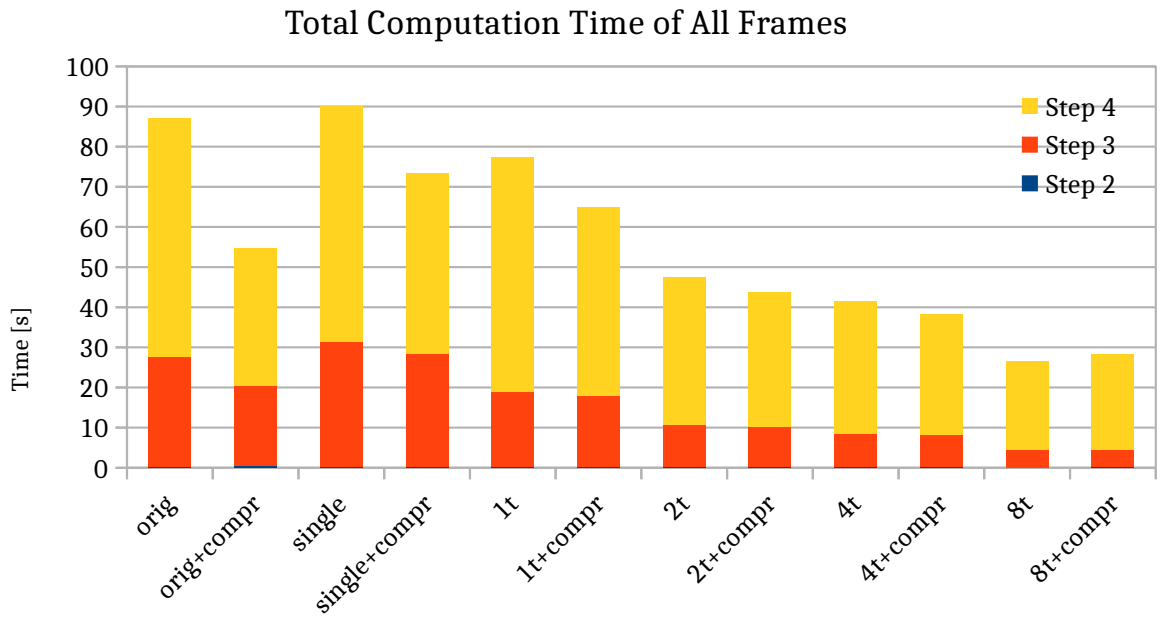

Figure 3.29: Total computation time to process all 724 frame with all methods. The timings of step 2 are so small that they are not visible. The abbreviations on the horizontal axis are described in Appendix C.

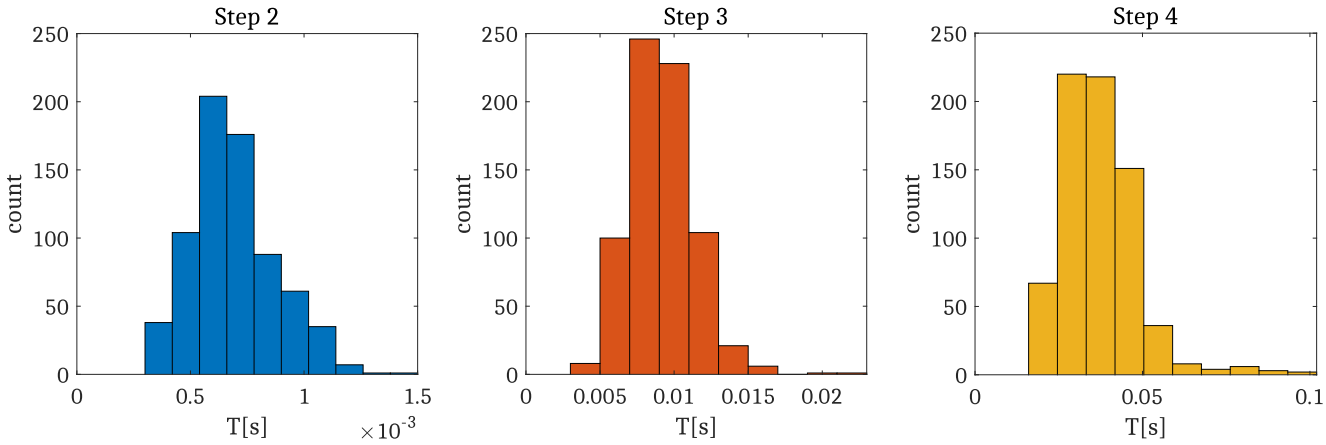

Figure 3.30: Histogram of the computation times of the new pipeline with compression and 8 threads.

3.3.5 Sparsity Analysis

To see how sparse the Φ matrix is the fill rate percentage was calculated for the dataset in Appendix D and the results are depicted in Table 3.4.

Measure	fill rate	fill rate compressed
minimum	0.54%	0.53%
average	0.59%	0.56%
maximum	0.84%	0.56%

Table 3.4: The relative sparsity in % of the Φ matrix for dataset in Appendix D.

As can be seen only around 0.6% of Φ is filled. By applying compression this value is kept closer to the average for the minimum and maximum values. For the dataset frame_9898190, two aligned point clouds capturing a subject at $\approx 1.5\,\mathrm{m}$, a bitmap of the sparsity pattern was made, see Figure 3.31. As the dataset had 10422 points this bitmap has $10422^2 = 108\ 618\ 084$ pixels. All matrix non-zero elements were set to black and the zeros were set to white.

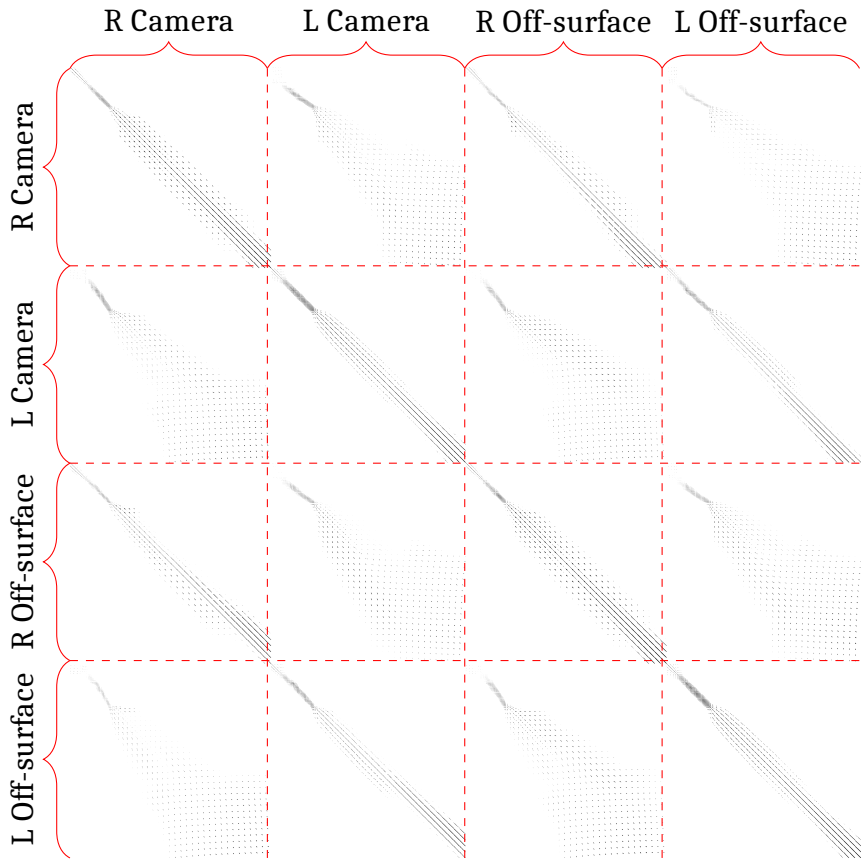

Figure 3.31: The sparsity pattern of the Φ matrix generated from dataset frame_9898190. On the image the horizontal and vertical ranges of pixels are annotated with curly braces to show what range is responsible for what points, those of the right (R), left (L) camera or their off-surface points.

No compression was applied in the generation of the image. A dense version of this Φ matrix stored with 32 bit floats would take ≈ 414 mega bytes of storage. The matrix has $946\ 926$ non-zero elements. The sparse version of the matrix stored in CSR storage format takes $2n_{nz} + n$ with n_{nz} the number of non-zeros and n the number of points. This would be $\frac{4}{1024^2}(2 \cdot 946926 + 10422) \approx 7$ mega bytes, substantially less than the dense version. A zoomed in crop of the top-left 100×100 pixels is shown in Figure 3.32.

As can be seen from this cropping the matrix is symmetric and the diagonal is filled. All values on the diagonal are 1 because that is the value of the Wendland radial basis function for $r = 0$. On the diagonal $r = 0$ because that is the distance between a point and itself.

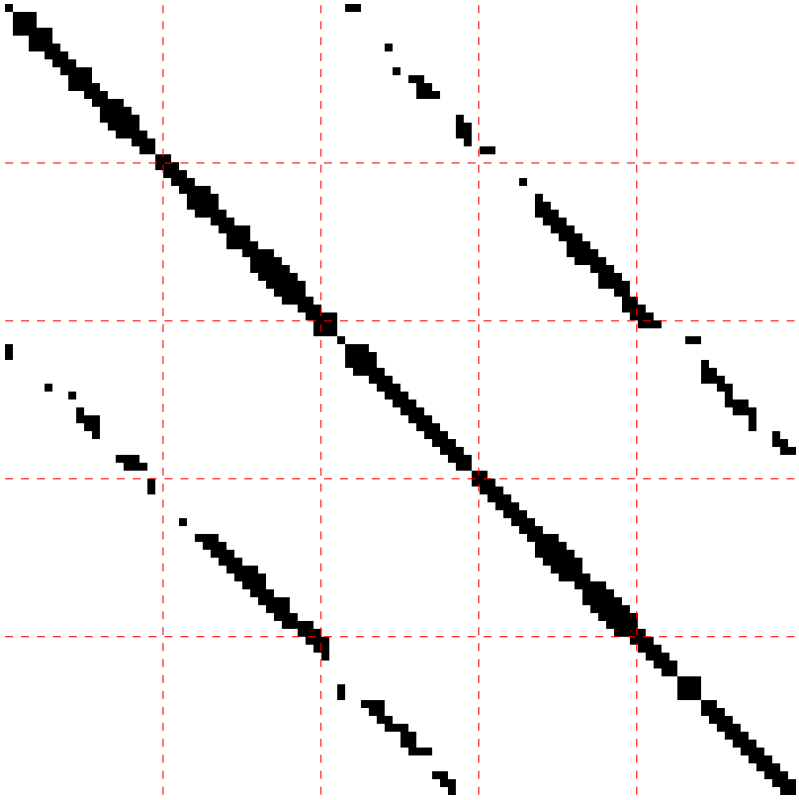

Figure 3.32: The top-left 100×100 pixels of the sparsity plot in Figure 3.31. The spacing of the dotted red grid is 20 pixels.

3.3.6 Validation

No extensive validation of the new pipeline was done in combination with a cobot simulation or an actual cobot. It is however possible to compare the original pipeline with the new pipeline. If the new pipeline produces the same results the previous work of T. Bosch [2] and [3] should be fully reproducible with the new pipeline. In Figure 3.33 two image of the same surface, one reconstructed by the original and the other reconstructed by the new pipeline are compared.

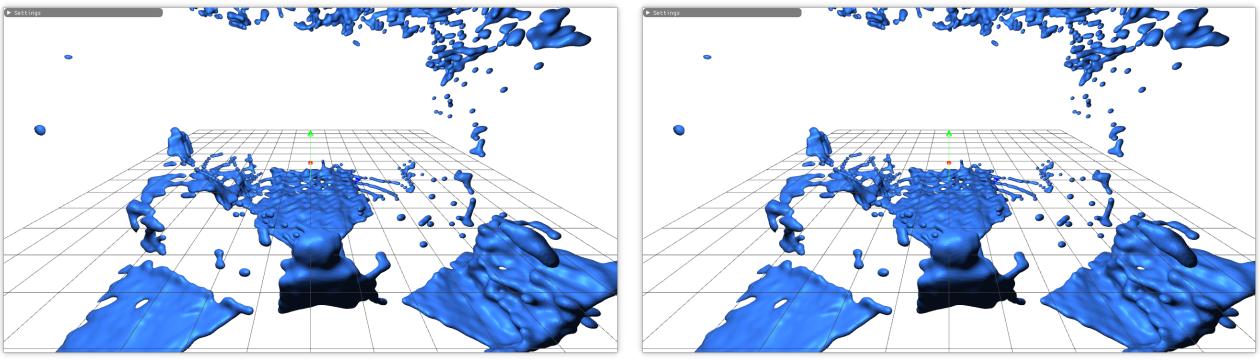

Figure 3.33: Comparison of the original pipeline and new pipeline with the original CSRBF parameters and the dataset in Appendix D. *Left:* Original pipeline. *Right:* New pipeline.

As far as can be seen with the naked eye there are no discernible differences. Again the value of

f was evaluated for a planar grid parallel to the ground and through the center of the soccer ball. The centers in a slab of $0.05\,\mathrm{m}$ above and below the plane were projected on the plane and plotted together with the level-sets $f = \{0.01, 0.02, 0.03, 0.04, 0.05, 0.06\}$.

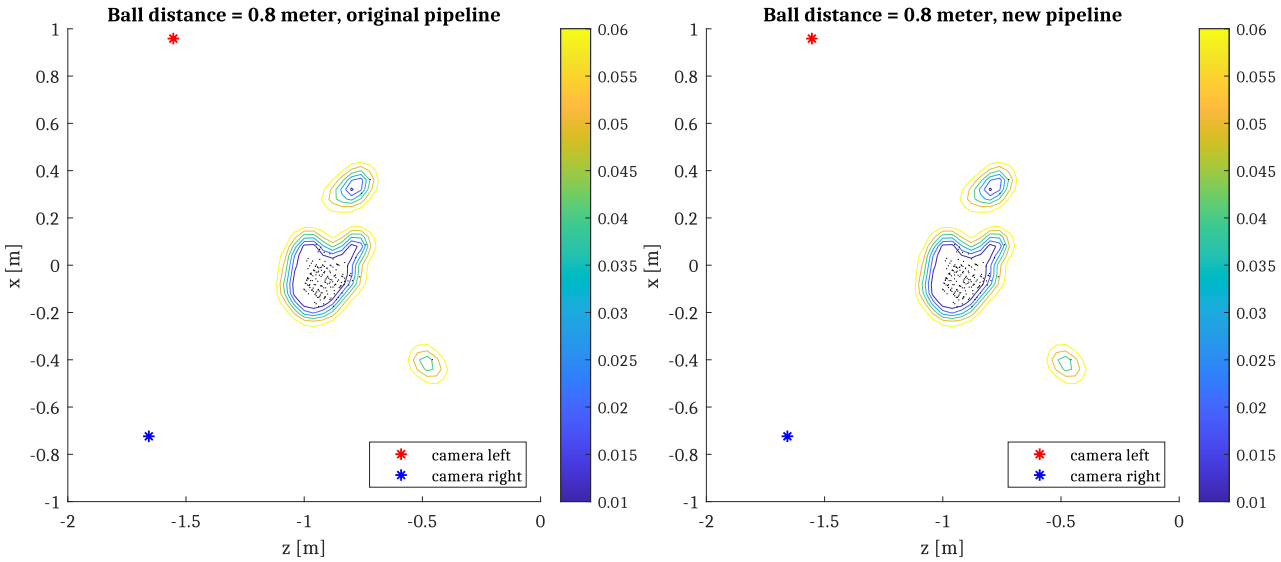

Figure 3.34: Iso-surface levels of a horizontal plane through the center of the soccer ball. The centers are the black dots. *Left:* Original pipeline. *Right:* New pipeline.

Again there is no discernible difference in the two plots, the original and the new pipeline produce exactly the same results.

3.4 Discussion and Recommendations

Interpretations: The introduction of more depth cameras to the pipeline does not only increase the computation time linearly by introducing n new points to the problem. The computation time can grow even harder when the point cloud density increases as aligned point clouds overlap. The same rapid increase in computation time is seen when an object is close to a depth camera. This chapter demonstrated that this problem can be solved with a voxel-grid filter tuned to the desired maximal point cloud density. If the voxel size of the voxel-grid filter is set to the distance between points at a distance where the cobot is located the voxel-grid filter will not cause distortion of the reconstructed environment at that distance. The distortion as distances closer to the camera is also low as demonstrated bu the distortion of the surface of a soccer ball at $0.08\,\mathrm{m}$. A more extensive validation of the distortion might be necessary to to make statements of different ratios of compression.

The original pipeline was changed so it could use a parallel CG solver. The solver is the most efficient when a BCSR block size of 2 is used. On top of this the calculation of the Φ matrix was made more efficient by parallelization. Both the calculation of Φ and the solving of the linear system of

equations is faster and when utilizing 8 cores on the used hardware (see A). The total computation time of the new parallel pipeline is a third on average than that of the original pipeline on the dataset in Appendix D. The resulting surfaces of the original and new pipeline are the same, no difference could be seen with the naked eye.

Implications and limitations: The proposed new pipeline is efficient and robust enough to process the input of two depth cameras at a resolution of 80×60 at an average frame rate of $25.7\,\mathrm{Hz}$ with a low standard deviation. This is a good result and closely satisfies our research goal of the improvement of efficiency. This means that point clouds captured by two Intel RealSense at the default frame rate of $30\,\mathrm{Hz}$ can be processed with not much frame drop.
Using two depth cameras reduces the shadow in the fused point cloud considerably, but the problem is not gone. To minimize this problem even further a multiple camera rig of at least 4 cameras as shown in Figure 3.35 is often used see [27], [28], [29]. If the average surface reconstruction frame rate is relaxed, to for instance $10\,\mathrm{Hz}$, it might be possible to use 4 cameras on more efficient hardware than was used for the benchmarks in this chapter.

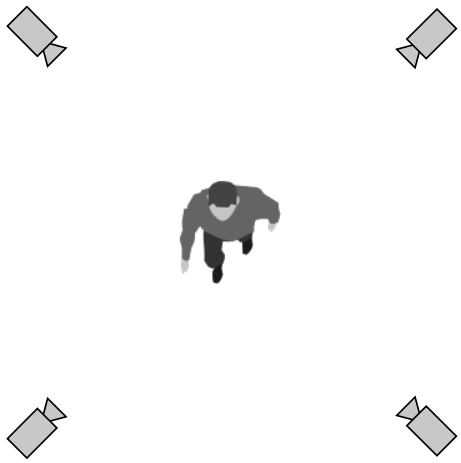

Figure 3.35: Quadruple camera rig to reduce the problems caused by shadows even further.

If multiple cameras are used to solve the problem of shadows then the CSRBF parameters of the original pipeline in Table 1.1 and Figure 1.12 and the proposed parameter in Section 3.2.2 will not work because off-surface points can be placed at a position where another camera, that sees the other side of the surface, knows that there should be no point.

Recommendations:
Weighted Compression: The further a sampled depth point is from its camera the lower its precision. When applying the voxel-grid filter the points of several cameras can be binned to the same voxel. An improvement can be to weight each point based on their distance from their camera when calculating the centroid for the voxel. Points close to their camera are assigned a high weight while points far from their camera have little weight. This might improve the accuracy of the reconstructed surface.

Feature Preserving Compression: A flat surface only needs a few uniformly spaced points to be reconstructed correctly while a detailed surface needs more support points to be reconstructed. An interesting approach could be to look in neighboring voxels and determine the curvature of the surface and reduce the number of points even more when the surface is flat. There are specialized feature preserving compression algorithms [25], [30] but it will be hard to beat the efficiency of the voxel-grid filter.

Load Balancing of Step 2: For the dataset in Appendix D the minimum computation time and the maximum computation time or the parallel activities to fill the matrix Φ differed by a factor 4 on average. This is high, it means that the computation time can be reduced by balancing the load of the parallel activities. Because it is not known beforehand how many non-zeros have to be calculated per row the optimal job size of the activities is not known beforehand. An approach could be to divide the job into many smaller batches and put them in a synchronized queue. When an parallel activity is finished with a batch it can pop a job from the queue if it is not empty. The parallel activity is finished when the queue is empty.

Total Pipeline on GPU: Using the PaLinSo solver on the GPU to solve the linear system of equations is not efficient because it takes time to transfer the point cloud data from RAM to VRAM and back. If the whole pipeline is implemented on the GPU these round trips are not necessary and this bottleneck is gone. First investigation is needed if the pipeline lends itself for efficient GPU implementation. If this is the case porting it to the GPU may take substantial time. Another option would be to investigate if an field-programmable gate array (FPGA) solution is viable.

Remove Off-surface points: When many cameras are used the placement of off-surface points is not desirable because they can be placed where other cameras see that they are incorrect. An option is to remove them and have only surface points with a constraint of 0. This frees up computation time that can be used to add more cameras or calculate proper normals. For model predictive control the normal direction of the surface has to be know in the points of the point cloud. In [1] section 2.2.1 and [31] an efficient method for the calculation of oriented surface normals are described. Some filtering and smoothing of the normals might be necessary.

Chapter 4

Conclusions and Recommendations

This chapter will briefly reflect on the work as a whole. For detailed conclusions and recommendations please refer the sections in Chapters 2 and 3.

The first research objective, the introduce multiple cameras to reduce the problem caused by shadows is successful. A method was developed to accurately and quickly align point clouds. When point clouds are poorly aligned the resulting fused point cloud is not very meaningful. The alignment method is easily extendable with more cameras. While two cameras reduce the problem with shadows but they are still present. Camera rigs of at least four cameras are recommended. When having more then two cameras the visibility of the fiducial marker might be an issue, the introduction of multiple markers could be a solution.
The noise of the type of depth cameras used is substantial but is manageable with calibration. The noise increases with distance however so it might be worthwhile to look at the alternatives discussed when more fidelity is required. One type of noise is bothersome and that is the presence of 'phantom points' in the point clouds. These are bad because they can cause the cobot a avoid obstacles that are not there. There are strategies to remove them from the point cloud and a strong recommendation is to investigate their removal as multiple cameras will worsen the issue even more.
Off-surface points pose another issue and will cause more problems when multiple cameras are used. The solution is simple, remove them. If normals are needed for model predictive control there are citations given with a simple and efficient method to obtain normals in the recommendations of Chapter 3.
Another problem is the resolution of the depth image. The acquired depth images are just 60×80 points in size. That means that human appendages at a distance of $\approx 1.5\,\mathrm{m}$ are sampled every $\approx 0.03\,\mathrm{m}$. Is that sufficient? Holes have been observed in subjects at this distance. Is the current resolution enough to have reliable collision avoidance?

The second research objective was to improve the reconstruction speed and make the pipeline robust. It became clear that point cloud density and uniformity was the main cause of high com-

putation times. So much so that computation of the surface would take extremely long per frame. Introducing more cameras would worsen the issue by causing overlap of the point clouds. This research showed that by making the point cloud more uniform where it was very dense will create an approximate upper bound on the frame rate. The frame rate is also more stable.
The pipeline was made more efficient by a factor of 3 by introducing parallel processing. A specialized algorithm is introduced to fill matrices of the BCSR storage type in parallel which is much faster than its single threaded counterpart. Also a highly optimized parallel CG solver was introduced further optimizing the pipeline. A recommendation is to use better hardware and more threads and test what the limits are of the pipeline. It seems that four depth cameras should be no problem.

Bibliography

[1] M. Berger, A. Tagliasacchi, L. M. Seversky, P. Alliez, G. Guennebaud, J. A. Levine, A. Sharf, and C. T. Silva, "A Survey of Surface Reconstruction from Point Clouds", *Computer Graphics Forum*, vol. 36, no. 1, pp. 301–329, 2017, ISSN: 14678659. DOI: 10.1111/cgf.12802.

[2] T. Bosch, "Vision Based Obstacle Representation for Cobot Collision Avoidance in Unstructured Environments", *Eindhoven University of Technology, Department of Mechanical Engineering, Dynamics and Control Group*, 2019.

[3] S. Driessen, "Reactive Cobot Control in Unstructured Dynamic Environments", *Eindhoven University of Technology, Department of Mechanical Engineering, Dynamics and Control Group*, 2019.

[4] R. Geraerts, A. Kamphuis, I. Karamouzas, and M. Overmars, "Using the corridor map method for path planning for a large number of characters", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5277 LNCS, pp. 11–22, 2008, ISSN: 03029743. DOI: 10.1007/978-3-540-89220-5-2.

[5] H. Wendland, "Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree", *Advances in Computational Mathematics*, vol. 4, no. 1, pp. 389–396, 1995, ISSN: 10197168. DOI: 10.1007/BF02123482.

[6] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3D surface construction algorithm", in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH '87*, vol. 21, New York, New York, USA: ACM Press, 1987, pp. 163–169, ISBN: 0897912276. DOI: 10.1145/37401.37422. [Online]. Available: http://portal.acm.org/citation.cfm?doid=37401.37422.

[7] A. M. Penrose and S. B. Katz, *Writing in the Sciences - Exploring Conventions of Scientific Discourse*. St. Martins Press, Inc., 1998, p. 34, ISBN: 0-312-11971-2.

[8] S. Giancola, M. Valenti, and R. Sala, *A Survey on 3D Cameras: Metrological Comparison of Time-of-Flight, Structured-Light and Active Stereoscopy Technologies*, ser. SpringerBriefs in Computer Science. Cham: Springer International Publishing, 2018, pp. 89–90, ISBN: 978-3-319-91760-3. DOI: 10.1007/978-3-319-91761-0. [Online]. Available: http://link.springer.com/10.1007/978-3-319-91761-0.

[9] E. Ujkani, J. Dybedal, A. Aalerud, K. B. Kaldestad, and G. Hovland, "Visual Marker Guided Point Cloud Registration in a Large Multi-Sensor Industrial Robot Cell", in *2018 14th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA)*, IEEE, Jul. 2018, pp. 1–6, ISBN: 978-1-5386-4643-4. DOI: 10.1109/MESA.2018.8449195. [Online]. Available: https://ieeexplore.ieee.org/document/8449195/.

[10] T. Madeira, M. Oliveira, and P. Dias, "Enhancement of RGB-D Image Alignment Using Fiducial Markers", *Sensors*, vol. 20, no. 5, p. 1497, Mar. 2020, ISSN: 1424-8220. DOI: 10.3390/s20051497. [Online]. Available: https://www.mdpi.com/1424-8220/20/5/1497.

[11] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and R. Medina-Carnicer, "Generation of fiducial marker dictionaries using Mixed Integer Linear Programming", *Pattern Recognition*, vol. 51, no. October, pp. 481–491, 2016, ISSN: 00313203. DOI: 10.1016/j.patcog.2015.09.023.

[12] F. J. Romero-Ramirez, R. Muñoz-Salinas, and R. Medina-Carnicer, "Speeded up detection of squared fiducial markers", *Image and Vision Computing*, vol. 76, no. June, pp. 38–47, 2018, ISSN: 02628856. DOI: 10.1016/j.imavis.2018.05.004.

[13] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge: Cambridge University Press, Mar. 2004, ISBN: 9780521540513. DOI: 10.1017/CBO9780511811685. [Online]. Available: https://www.cambridge.org/core/product/identifier/9780511811685/type/book.

[14] G. Casiez, N. Roussel, and D. Vogel, "1 € filter", in *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*, New York, New York, USA: ACM Press, 2012, p. 2527, ISBN: 9781450310154. DOI: 10.1145/2207676.2208639. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2207676.2208639.

[15] M. Rakotosaona, V. La Barbera, P. Guerrero, N. J. Mitra, and M. Ovsjanikov, "PointCleanNet: Learning to Denoise and Remove Outliers from Dense Point Clouds", *Computer Graphics Forum*, vol. 39, no. 1, pp. 185–203, Feb. 2020, ISSN: 0167-7055. DOI: 10.1111/cgf.13753. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13753.

[16] P. Besl and N. D. McKay, "A method for registration of 3-D shapes", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 2, pp. 239–256, Feb. 1992, ISSN: 0162-8828. DOI: 10.1109/34.121791. [Online]. Available: http://ieeexplore.ieee.org/document/121791/.

[17] S. Bouaziz, A. Tagliasacchi, and M. Pauly, "Sparse iterative closest point", *Eurographics Symposium on Geometry Processing*, vol. 32, no. 5, pp. 113–123, 2013, ISSN: 17278384. DOI: 10.1111/cgf.12178.

[18] E. Gallopoulos, B. Philippe, and A. H. Sameh, *Parallelism in Matrix Computations*, ser. Scientific Computation. Dordrecht: Springer Netherlands, 2016, pp. 1–473, ISBN: 978-94-017-7187-0. DOI: 10.1007/978-94-017-7188-7. [Online]. Available: http://link.springer.com/10.1007/978-94-017-7188-7.

[19] Y. Saad, "Iterative Methods for Sparse Linear Systems", *Iterative Methods for Sparse Linear Systems*, 2003. DOI: 10.1137/1.9780898718003.

[20] M. Verschoor and A. C. Jalba, "Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs", *Parallel Computing*, vol. 38, no. 10-11, pp. 552–575, Oct. 2012, ISSN: 01678191. DOI: 10.1016/j.parco.2012.07.002. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2012.07.002.

[21] J. R. Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain", *Science*, 1994.

[22] A. Grunnet-Jepsen, P. Winer, A. Takagi, J. Sweetser, K. Zhao, T. Khuong, D. Nie, and J. Woodfill, "Using the Intel ® RealSense TM Depth cameras D4xx in Multi-Camera Configurations", 2018. [Online]. Available: https://dev.intelrealsense.com/docs/multiple-depth-cameras-configuration.

[23] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, and E. Steinbach, "Real-time compression of point cloud streams", in *2012 IEEE International Conference on Robotics and Automation*, IEEE, May 2012, pp. 778–785, ISBN: 978-1-4673-1405-3. DOI: 10.1109/ICRA.2012.6224647. [Online]. Available: http://ieeexplore.ieee.org/document/6224647/.

[24] J. C. Rangel, V. Morell, M. Cazorla, S. Orts-Escolano, and J. García-Rodríguez, "Object recognition in noisy RGB-D data using GNG", *Pattern Analysis and Applications*, vol. 20, no. 4, pp. 1061–1076, 2017, ISSN: 14337541. DOI: 10.1007/s10044-016-0546-y.

[25] X.-F. Han, J. S. Jin, M.-J. Wang, W. Jiang, L. Gao, and L. Xiao, "A review of algorithms for filtering the 3D point cloud", *Signal Processing: Image Communication*, vol. 57, no. May, pp. 103–112, Sep. 2017, ISSN: 09235965. DOI: 10.1016/j.image.2017.05.009. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0923596517300930.

[26] Y. Solihin, *Fundamentals of Parallel MULTICORE Architecture*, 1st. Chapman & Hall/CRC, 2015, ISBN: 1482211181. DOI: 10.1201/b20200.

[27] M. Dou, S. Khamis, Y. Degtyarev, P. Davidson, S. R. Fanello, A. Kowdle, S. O. Escolano, C. Rhemann, D. Kim, J. Taylor, P. Kohli, V. Tankovich, and S. Izadi, "Fusion4D: Real-time performance capture of challenging scenes", *ACM Transactions on Graphics*, vol. 35, no. 4, 2016, ISSN: 15577368. DOI: 10.1145/2897824.2925969.

[28] M. Dou, P. Davidson, S. R. Fanello, S. Khamis, A. Kow-Dle, C. Rhemann, V. Tankovich, and S. Izadi, "Motion2Fusion: Real-time volumetric performance capture", *ACM Transactions on Graphics*, vol. 36, no. 6, 2017, ISSN: 15577368. DOI: 10.1145/3130800.3130801.

[29] S. Meerits, V. Nozick, and H. Saito, "Real-time scene reconstruction and triangle mesh generation using multiple RGB-D cameras", *Journal of Real-Time Image Processing*, vol. 16, no. 6, pp. 1–13, 2017, ISSN: 18618200. DOI: 10.1007/s11554-017-0736-x.

[30] J. Elseberg, D. Borrmann, and A. Nüchter, "Efficient processing of large 3D point clouds", *2011 23rd International Symposium on Information, Communication and Automation Technologies, ICAT 2011*, no. October, 2011. DOI: 10.1109/ICAT.2011.6102102.

[31] S. Meerits, D. Thomas, V. Nozick, and H. Saito, "FusionMLS: Highly dynamic 3D reconstruction with consumer-grade RGB-D cameras", *Computational Visual Media*, vol. 4, no. 4, pp. 287–303, Dec. 2018, ISSN: 2096-0433. DOI: 10.1007/s41095-018-0121-0. [Online]. Available: http://link.springer.com/10.1007/s41095-018-0121-0.

Appendix A

Hardware and Software

For reference the hardware used in the measurements in Chapter 3 is given in Table A.1.

Component	Value
Computer Model	HP Zbook Studio G3
CPU	Intel Core i7-6700HQ CPU @ 2.60GHz
GPU	Quadro M1000M, 4096 MB VRAM
RAM	16 Gb @ 2.133GHz
Depth Camera	Intel RealSense D435

Table A.1: Hardware used in this thesis.

The software and libraries that are used in the new pipeline and the viewer `RBF_Viz` are listed in Table A.2.

Software Package	Version
OS	Debian GNU/Linux 10 (buster)
OpenGL	version 3.3.0
Realsense SDK	version 2.2.13
Eigen	version 3.3.7
OpenCV	version 3.2.0
PCL	version 1.9.1

Table A.2: The software used in this thesis.

Appendix B

Point Cloud Alignment Matrix File

An example of an point cloud alignment matrix stored to disk in the matrix market exchange format[1]. A human readable format that is easy to parse.

```
%%MatrixMarket matrix coordinate  real general
4 4 16
1 1 -4.31202501e-01
1 2 -1.33644104e-01
1 3 8.923024535e-01
1 4 -1.04172730e+00
2 1 -2.17633117e-02
2 2 -9.87140775e-01
2 3 -1.58365518e-01
2 4 2.242448032e-01
3 1 9.019926786e-01
3 2 -8.77070501e-02
3 3 4.227490425e-01
3 4 -5.47817409e-01
4 1 -0.0e+00
4 2 0.0e+00
4 3 -0.0e+00
4 4 1.00e+00
```

[1] https://math.nist.gov/MatrixMarket/formats.html

Appendix C

Computation Times for Step 2, 3, and 4

Abbreviation	Description
orig	Original pipeline, the value is measured with the original pipeline as presented in [2]
new	New pipeline, the value is measured with the new parallel pipeline
compr	Compression with voxel-grid filter was applied to the point cloud
single	Sinle-threaded, everything is executed on the main thread
1t, 2t, 4t, 8t	Multi-threaded with either 1, 2, 4, or 8 threads

Table C.1: Clarification of used abbreviations.

Measure	No Compression	Compression
Minimum	8422	8380
Average	10119.1	9887.5
Maximum	10820	10616
St. Dev.	643.2	602.5

Table C.2: Number of points in fused point cloud after step 2.

Measure	Original	Original + compression	New	New + compression
Minimum	192716	186678	385432	373356
Average	304464	271557	608928	543114
Maximum	491329	315045	982658	630090
St. Dev.	52271	32804	104542	65608

Table C.3: Number of non-zeros in Φ after step 3 for original [2] and new pipeline.

Measure	Step 2	Step 2 + Compression
Minimum	0.00017	0.00031
Average	0.00047	0.00027
Maximum	0.00118	0.00077
St. Dev.	0.00014	0.00009

Table C.4: Timings of step 2 with and without compression.

Measure	orig	orig+compr	single	single+compr	1t	1t+compr
Minimum	0.01675	0.01625	0.02470	0.02436	0.01569	0.01532
Average	0.03783	0.02753	0.04311	0.03857	0.02607	0.02414
Maximum	0.43052	0.03900	0.08416	0.05552	0.05870	0.03832
St. Dev.	0.04056	0.00444	0.01010	0.00549	0.00524	0.00440
Measure	2t	2t+compr	4t	4t+compr	8t	8t+compr
Minimum	0.00868	0.00842	0.00531	0.00588	0.00339	0.00369
Average	0.01469	0.01354	0.01122	0.01083	0.00621	0.00567
Maximum	0.04350	0.02564	0.02970	0.02438	0.01042	0.01239
St. Dev.	0.00324	0.00266	0.00308	0.00255	0.00120	0.00094

Table C.5: Computation times of step 3.

Measure	orig	orig+compr	single	single+compr	1t	1t+compr
Minimum	0.02428	0.02212	0.03668	0.03528	0.03656	0.03566
Average	0.08197	0.04740	0.08104	0.06227	0.08062	0.06499
Maximum	0.50380	0.15072	0.37071	0.13465	0.38666	0.13831
St. Dev.	0.05436	0.01273	0.04390	0.01199	0.04485	0.01244
Measure	2t	2t+compr	4t	4t+compr	8t	8t+compr
Minimum	0.02453	0.02761	0.01738	0.02375	0.01498	0.02172
Average	0.05078	0.04630	0.04566	0.04152	0.03033	0.03280
Maximum	0.22843	0.08784	0.17416	0.07934	0.12985	0.05659
St. Dev.	0.02750	0.00803	0.02157	0.00722	0.01572	0.00414

Table C.6: Computation times of step 4.

Activity	orig	orig+compr	single	single+compr	1t	1t+compr
Step 1	0.209	0.364	0.213	0.365	0.191	0.338
Step 2	27.389	19.930	31.212	27.923	18.874	17.477
Step 3	59.346	34.318	58.669	45.086	58.369	47.050
Step 1+2+3	86.945	54.612	90.094	73.374	77.434	64.865
Activity	2t	2t+compr	4t	4t+compr	8t	8t+compr
Step 1	0.185	0.338	0.215	0.365	0.153	0.292
Step 2	10.637	9.804	8.123	7.842	4.498	4.107
Step 3	36.765	33.520	33.057	30.058	21.962	23.751
Step 1+2+3	47.587	43.661	41.395	38.265	26.613	28.150

Table C.7: Total computation times of all frames in the dataset.

Appendix D

Reference Dataset

For several of the experiments in this document the reference dataset in this appendix was used. The first part consists of a series of static depth images captured by two depth cameras of a soccer ball on a box. The setup of the scene can be found in Figure D.1.

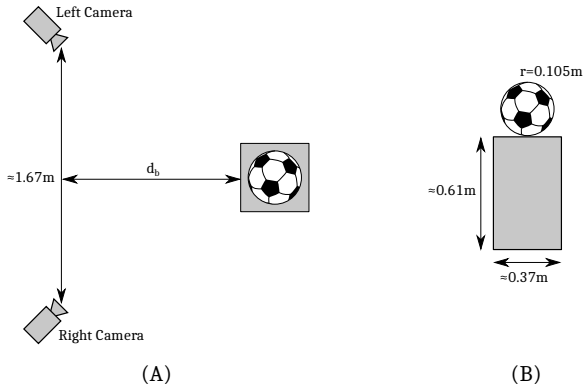

Figure D.1: Camera setup of reference dataset

Depth images and 3D point clouds (in PLY format) were captured for $d_p \approx \{0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8\}$ meter. In Figure D.2 2 such depth images are shown.

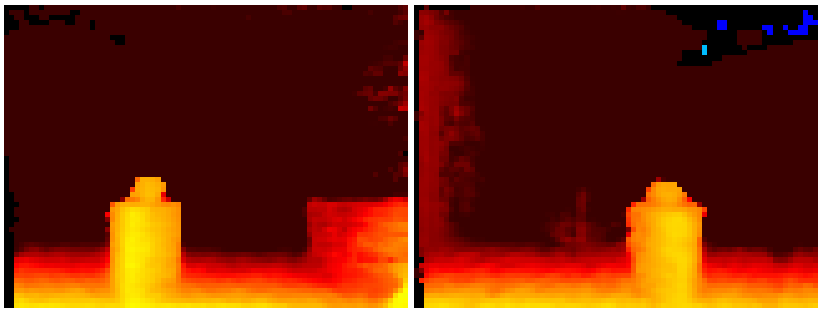

Figure D.2: Left and right depth images of the soccer ball scene with $d_b \approx 1.6\,\mathrm{m}$

The second part consists of two depth streams of a subject D.3 that moves his outstretched arm from right to left. The setup can be found in Figure D.3.

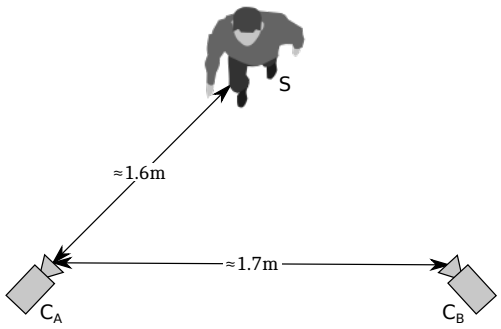

Figure D.3: Camera setup of reference dataset

Two typical depth images of the dataset are shown in Figure D.2.

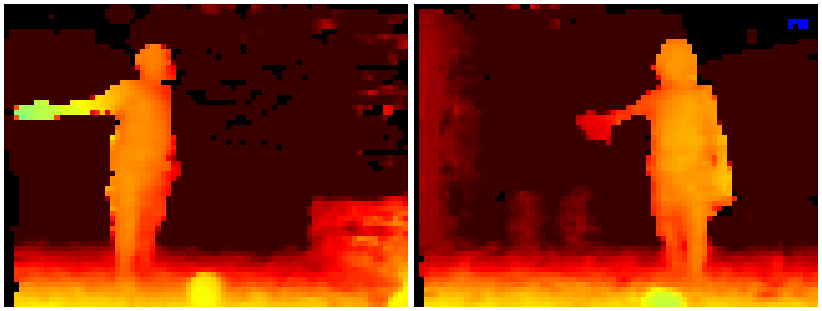

Figure D.4: Left and right depth images of the person scene with $d_b \approx 1.6\,\mathrm{m}$

The dataset also has two stored view files in matrix market exchange format which can be used to align the point cloud of the left and the right cameras.