Camera Tracking Using a Dense 3D Model

Erik Bylow

Master's thesis
2012:E41

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Mathematics

Centrum Scientiarum Mathematicarum

Abstract

In this thesis we present a method for tracking a depth sensor and building a 3D model in real-time. The tracking will be done using the current estimated 3D model. We will show that this approach gives more accurate results and is more robust than the well-known KinectFusion approach. It will also be shown how to colourise the 3D model in real-time.

In this work we will study different error metrics such as the projective point-to-point metric and projective point-to-plane metric. Also different norms will be evaluated to find out which gives the best result. We will also show how to represent a 3D model by using a so called signed distance function.

Acknowledgement

I would like to thank my supervisors Fredrik Kahl and Jürgen Sturm for helping me with this thesis and for that they have together with Daniel Cremers given me the opportunity to do my thesis as an exchange student at the Technology University of Munich.

I would also like to thank the rest of the computer vision group in Munich for welcoming me and helping me with all different kinds of problems.

Contents

1 Introduction **7**
1.1 Motivation . . . 7
1.2 Goal . . . 8
1.3 Method . . . 8
1.4 Outline . . . 9

2 Basics **11**
2.1 Notations . . . 11
2.2 Pinhole Camera Model . . . 12
2.3 Depth Images . . . 14
2.4 Fusion of Depth Images . . . 16
2.5 Signed Distance Function . . . 17
2.6 Visualization . . . 20
2.7 Rigid Body Motion . . . 22
2.8 Registration . . . 23
2.9 Related Work . . . 24

3 Reconstruction of 3D Surfaces **27**
3.1 Projective Point-To-Point Metric . . . 27
3.2 Projective Point-To-Plane Metric . . . 32
3.3 Estimating the Signed Distance Function . . . 38
3.4 Colourising . . . 41

4 Tracking **45**
4.1 Introduction To the Approach . . . 45
4.2 Optimisation . . . 47
4.2.1 L_2-norm . . . 47
4.2.2 L_1-norm . . . 51
4.2.3 Truncated L_2-norm . . . 58
4.3 Summary . . . 58

5 Experimental Results **59**
5.1 Qualitative Results . . . 59

5.2 L_2-Norm 63
5.3 L_1-Norm 65
5.4 Truncated L_2 66

6 Conclusion And Future Work **69**

7 Appendix **75**

Chapter 1

Introduction

1.1 Motivation

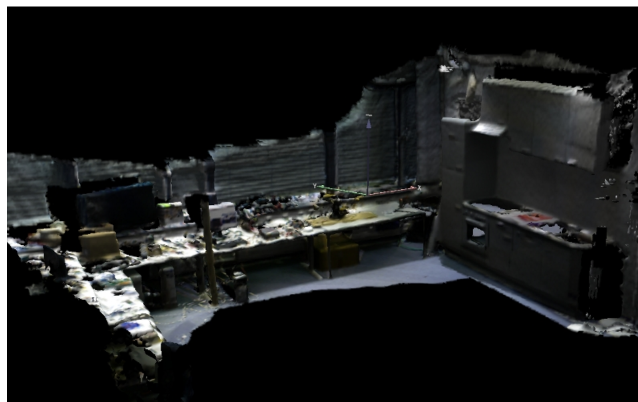
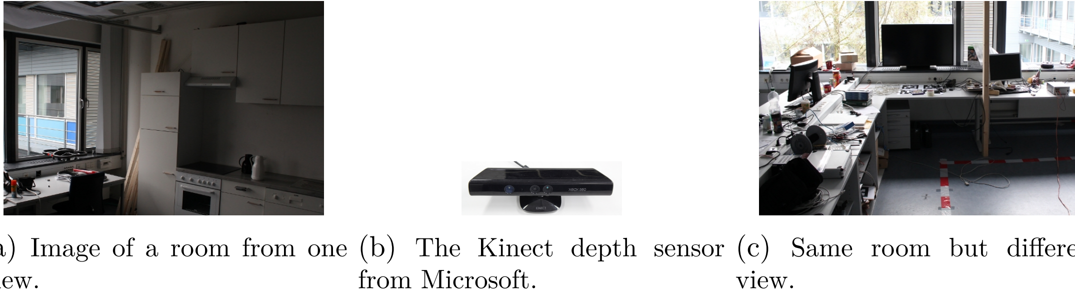


Figure 1.1: A 3D model of a room.

In several tasks, it can be useful to create dense 3D models in real-time. One example is in robotics where the robot can use a 3D model of the environment calculated in real-time in order to navigate in the room. An example of how a 3D model of a room might look like is given in Figure 1.1.

Other applications you can think of concerning 3D models can be internet shopping, if you can create a 3D model of your room, you can use this to see how new furnitures would fit into your room without going to IKEA.

It can also be useful if you want to renovate your room, by using the 3D model one could change colours and change the room to see how the room would look like after the renovation before it has even started.



(a) Image of a room from one view. (b) The Kinect depth sensor from Microsoft. (c) Same room but different view.

Figure 1.2: Here one can see two different views of the room and the Microsoft Kinect in the middle.

1.2 Goal

The goal with this thesis is to build a complete 3D model in real-time by using images from different views of the same object, as in Figure 1.2a and Figure 1.2c, using a depth camera, Figure 1.2b, which generates a surface from each image. To do this one needs to estimate the camera position and at the same time build the 3D model.

If the position of the camera is known, then it is possible to estimate the surface. On the other hand, if the position of the surface is known, then one can find where the camera is.

The problem is that in general neither the position of the camera nor the position of the surface is known in advance.

The goal with this work is to estimate the camera movement and build the 3D model in real-time.

1.3 Method

To build the 3D model we use the method presented by Levoy and Curless in [10]. They use a 3D volume to represent the surface by using a so called Signed Distance Function (SDF).

To build the 3D model, one needs to know how the camera is rotated and translated. Our approach is to use the SDF to define an error function. That way we use the surface that we already have in order to find the camera position.

The main contribution with this work is that we track our camera against the complete dense surface model. In particular we will look at different error metrics and different norms to evaluate how good the different norms and error metrics are. Furthermore, we present a method for including colours in the 3D reconstruction.

1.4 Outline

The outline for the rest of this thesis is:

Chapter 2: Basics - Here we present basic stuff such as the camera model, depth images, rigid-body motions etc. We will also present how the geometry can be represented and at the end we present related work.

Chapter 3: Reconstruction of 3D Surfaces - In this chapter we present our approach to dense 3D reconstruction, assuming the trajectory of the camera is known. We present different metrics for how one can approximate the signed distance function.

Chapter 4: Tracking - In this chapter we present our method for estimating the trajectory of the camera. We present different error functions and how to minimize them.

Chapter 5: Experimental Results - Here we show how our tracking works for different datasets, we do also demonstrate that it works for live data.

Chapter 6: Conclusion And Future Work - Here we summarize the results of our thesis and possible extensions and improvements of our method.

Chapter 7: Appendix - In this chapter a detailed evaluation of our method is provided. We compare it in detail between KinFu and RGB-D SLAM.

Chapter 2

Basics

In this chapter we introduce basic concepts which are necessary to understand this thesis, such as the pinhole camera model and depth images. We will also introduce the notations we will use in the remainder of this thesis.

2.1 Notations

Here we present notations which we will use in the rest of this report.

A 3D point in $\mathbb{R}^3$ will be denoted by

$$X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

We will also use homogeneous coordinates to represent a point. The homogeneous coordinates will be denoted by

$$\bar{X} = \lambda \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

We will always assume that $\lambda = 1$ if nothing else is said.

To denote a 3D point with respect to the cameras frame of reference we will write

$$X_L = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

A 3D point in the global frame of reference will be denoted by

$$X_G = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

We will also work with vertices in a voxel grid which we will use to represent our SDF. A vertex in the voxel grid will be written

$$V_{i,j,k}$$

where (i, j, k) are non-negative integer coordinates for the vertex in the voxel grid. We will always assume that the global coordinates for the vertex $V_{i,j,k}$ are known.

We will denote vectors in $\mathbb{R}^N$ by boldface, for instance

$$\mathbf{v} \in \mathbb{R}^3.$$

A depth image will be denoted by

$$I_d$$

and with

$$I_d(i, j)$$

we mean the value at pixel (i, j).

2.2 Pinhole Camera Model

We will in this work use the so called pinhole camera model with focal length f, which is the distance between the image plane and the camera center. We will also use the pixel coordinates c_x and c_y, which are the pixel coordinates where the principal axis meets the image plane. In the model, a 3D point will be projected onto the image plane, illustrated in Figure 2.1.

By triangulation, illustrated in Figure 2.2, we can calculate the 3D coordinates for the point (x, y) in the image plane by

$$\frac{x}{z} = \frac{x'}{f} \iff x = \frac{zx'}{f}. \tag{2.1}$$

With help from this we can define some functions we will use in the rest of this report. For simplicity we will treat the pixel coordinates as real valued, even though an image consists of discrete pixels. It can be continuously represented by for instance doing interpolation between the neighbouring pixels.

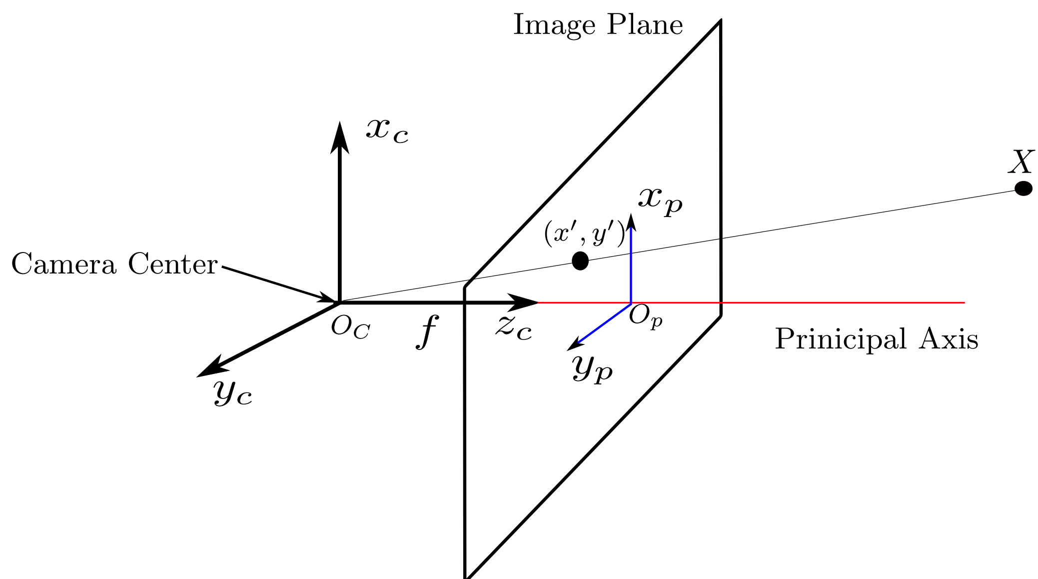


Figure 2.1: A sketch of the camera model we will use.

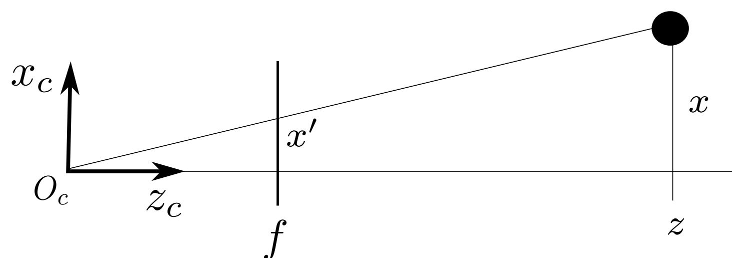


Figure 2.2: Here are two similar triangles (x', f, O_c) and (x, z, O_c).

(a) The depth image of a teddy bear. (b) The colour image of the teddy bear.

We define here a function ρ which transforms pixel (i, j) in a depth image to a 3D coordinate. To see where the equation comes from one can look at Figure 2.1.

Definition 1. Let ρ be the vector valued function $\rho : \mathbb{R}^2 \times \mathbb{R} \rightarrow \mathbb{R}^3$ which takes the pixels (i, j) to its 3D coordinates $X = (x, y, z)$:

$$\rho(i, j, I_d(i, j)) = (\tfrac{(i-c_x)z}{f_x}, \tfrac{(j-c_y)z}{f_y}, z)^T$$

where $(i, j) \in I_d$, $z = I_d(i, j)$ and f_x, f_y, c_x and c_y are intrinsic camera parameters.

We also defines a function which projects a 3D point $X = (x, y, z)$ onto the image plane.

Definition 2. Let π be the vector valued function $\pi : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ which projects the 3D-point $X = (x, y, z)$ onto the image plane:

$$\pi(x, y, z) = (\tfrac{f_x x}{z} + c_x, \tfrac{f_y y}{z} + c_y)$$

where f_x, f_y, c_x and c_y are intrinsic camera parameters.

2.3 Depth Images

In this work we will use so called depth sensors. From the depth sensors we will use we will receive one normal colour image and one so called depth image as seen in Figure 2.3a and 2.3b. The colour image is just a normal image one gets from an ordinary hand held digital camera or the camera on a cell phone. The speciality with a depth sensor is the depth image, from which you get information about the distance between the surface and the camera which you do not get from the colour image.

To get an idea of what an depth image is, we start with an illustrative figure of a depth image, Figure 2.3.

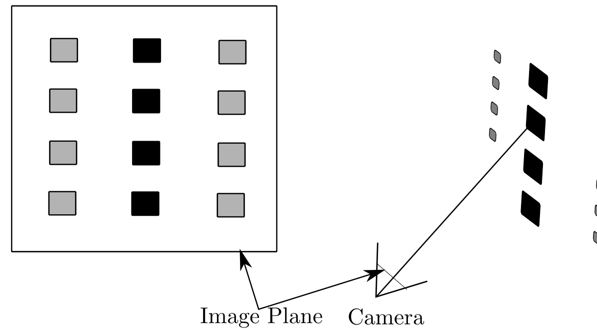


Figure 2.3: An illustrative image of a depth map. The depth is visualized by the intensity of the colour, darker pixels corresponds to points closer to the camera.

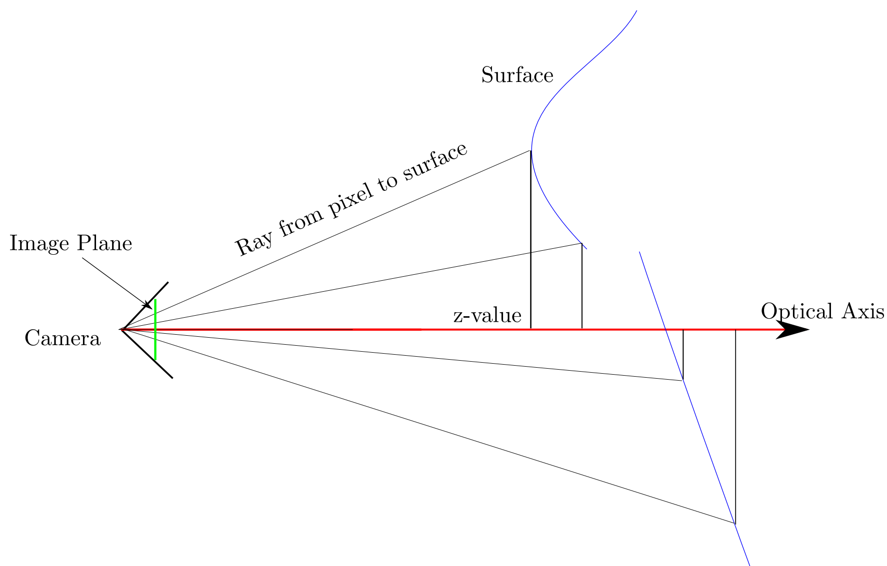


Figure 2.4: The ray between a pixel and the camera center hits the surface. The distance between the camera center and the surface point is then projected to the principal axis. The projected value is the so called z-value, which is stored in the corresponding pixel.

In the depth image I_d, each pixel has a so called z-value. The z-value is the distance between the camera center and the surface point projected to the principal axis, as seen in Figure 2.4.

For a pixel (i, j), we thus get a z-value z_{ij}. This makes it possible to calculate the 3D coordinates $X = (x, y, z)$ corresponding to the pixel (i, j) by computing

$$x = \frac{(i - c_x)z_{ij}}{f_x}$$
$$y = \frac{j - c_y z_{ij}}{f_y}$$
$$z = z_{ij}$$

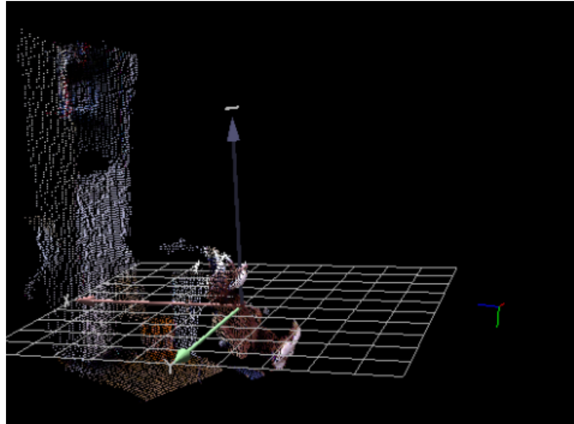


Figure 2.5: A point cloud generated from a depth image of a teddy bear. The colours are taken from the corresponding colour image for visualization purposes.

where f_x and f_y are the focal lengths and c_x and c_y are the pixel coordinates for the optical center.

In the sequel, we will instead use our defined function ρ

$$(x, y, z) = \rho(i, j, I_d(i, j)).$$

The difference between a depth image and a normal image is thus the ability to compute the exact 3D point. In a colour image one gets from a hand held camera, one can only compute the line between the pixel (i, j) and the camera center at which the 3D point lies, the z-value gives us where on the line the 3D point is.

In Figure 2.5 we see an example of a point cloud generated from the Kinect. Note that by doing the calculations above, we only get the local 3D coordinates in the camera reference system. To get the global coordinates, we need to know how the camera is rotated and translated with respect to a global coordinate system.

2.4 Fusion of Depth Images

The goal is to estimate a complete 3D model from depth images. If we look at Figure 2.6 we see three different point clouds from different parts of the same room. By fusing all images one would like to obtain a 3D model of the whole scene, as illustrated in Figure 2.7.

To achieve this we need to know the global configuration $[\mathbf{R}\ \mathbf{t}]$ of the camera so that we can calculate the global 3D coordinates X_G by

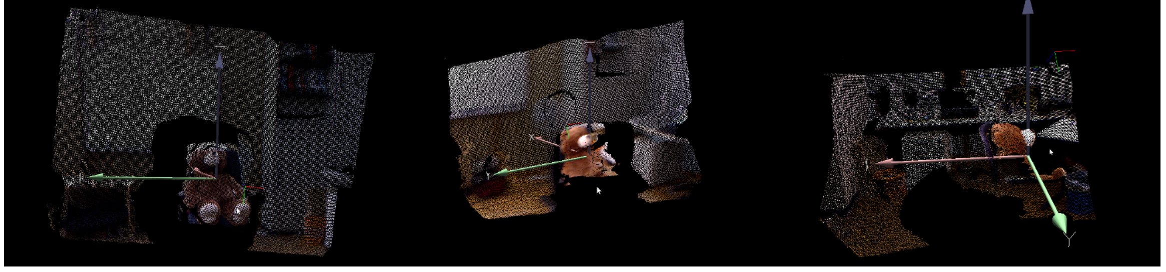


Figure 2.6: Resulting point clouds generated from three depth images. The goal with fusion of these images is to get a 3D model of the whole scene from all these images.

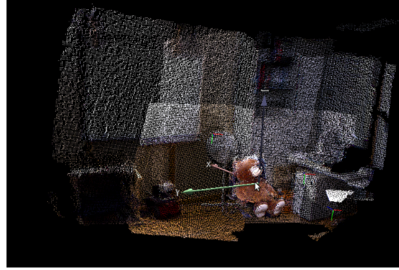


Figure 2.7: Resulting point cloud generated from a couple of depth images, giving a whole 3D model of what the camera sees from different views.

$$X_G = \mathbf{R}X_L + \mathbf{t} \tag{2.2}$$

$$X_L = \rho(i, j, I_d(i, j)) \tag{2.3}$$

where (i, j) are pixel coordinates in the depth image I_d.

Hence, if we know the rotation and translation for all cameras we can easily calculate the global position for all the 3D points we obtain from the depth images.

2.5 Signed Distance Function

To fuse all the depth images into a 3D model we need a way to represent the geometry seen from all different camera positions.

To represent the geometry of the surface we rely on the work by Levoy and Curless, [10]. They use a so called Signed Distance Function to represent the geometry.

Definition 3. A signed distance function ψ is a function which gives the signed distance d between a point $X \in \mathbb{R}^3$ and the closest point X_s on a surface S.

$\psi : \mathbb{R}^3 \to \mathbb{R}$

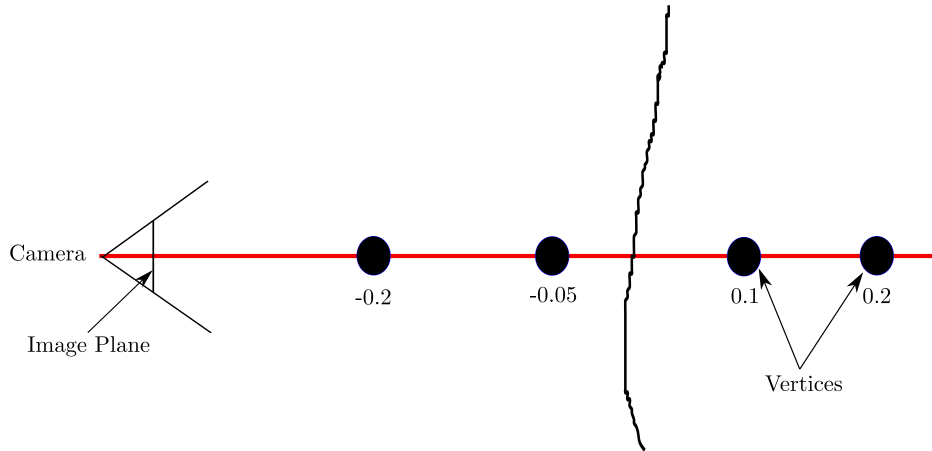


Figure 2.8: A sketch of how a signed distance function works, the distance between the black points and the curve is negative in front of the surface and positive behind.

$\psi(X) = d$

where $d < 0$ if X is outside the surface and $d > 0$ if X is inside the surface.

If we look on a two dimensional example in Figure 2.8 we can see how different vertices have different signs and distances, depending on how close and on which side of the curve the vertices are on.

The geometry in a signed distance function is implicitly represented by its zero crossing.

The reason we choose to use the method proposed by Levoy and Curless is that there are no restrictions on how the geometry may look like and the SDF gives us also the possibility to define an error function which we will minimise to find the rotation and translation of the camera.

Example:

There are some geometric objects which can be exactly described implicitly and gives a good example of how the geometry can be represented on a computer with a SDF and a voxel grid. Since we will be working in 3D, we can look at the equation for a sphere with radius r

$$\psi : \mathbb{R}^3 \to \mathbb{R}$$
$$\psi(x, y, z) = \sqrt{x^2 + y^2 + z^2} - r.$$

For a point $(x, y, z) \in \mathbb{R}^3$, $\psi(x, y, z)$ is either positive, negative or zero. If $\psi(x, y, z) < 0$, then $\sqrt{x^2 + y^2 + z^2} < r$ and if $\psi(x, y, z) > 0$, then $\sqrt{x^2 + y^2 + z^2} > r$ and finally, if $\psi(x, y, z) = 0$, then $\sqrt{x^2 + y^2 + z^2} = r$. Thus, all points $(x, y, z) \in \mathbb{R}^3$ fulfilling $\psi(x, y, z) = 0$ lie on the surface of the sphere.

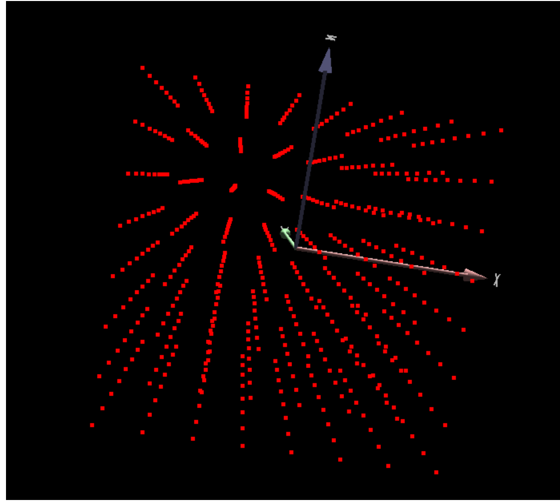


Figure 2.9: This is how the voxel grid looks, it has a fixed resolution with equally distanced vertices and a fixed place in space.

Hence, $\psi(x, y, z)$ is an implicit representation of a sphere and for all points inside the sphere $\psi(x, y, z) < 0$ and for all points outside the sphere $\psi(x, y, z) > 0$. In other words, for an arbitrary point $(x, y, z) \in \mathbb{R}^3$, $\psi(x, y, z)$ gives the signed distance between (x, y, z) and the surface.

We will now illustrate how to represent ψ on a computer. As we can see in the example image, Figure 2.8, we can use vertices and assign each vertex a distance between the vertex and the surface. That would then be a discrete version of ψ. Since we are working with three dimensional objects, we need a discretisation in 3D. The discretisation can be seen in Figure 2.9. We will call this a voxel grid, each vertex in the voxel grid will be assigned the signed distance between the vertex and the surface.

To represent ψ with this voxel grid, we do the following calculations for each vertex:

$$s = \sqrt{x^2 + y^2 + z^2} - r \tag{2.4}$$

where (x, y, z) are the coordinates for the vertices.

The value s is then stored in the corresponding vertex. The vertices which are inside the sphere will have negative values and the vertices which are outside the sphere will have positive values and the vertices which are on the surface will have the distance zero.

This is visualized by plotting the vertices which are inside the surface in yellow and the other in red in Figure 2.10

This example with the sphere shows how the geometry can be represented implicitly by a SDF and also how one can represent it on a computer by using a voxel grid.

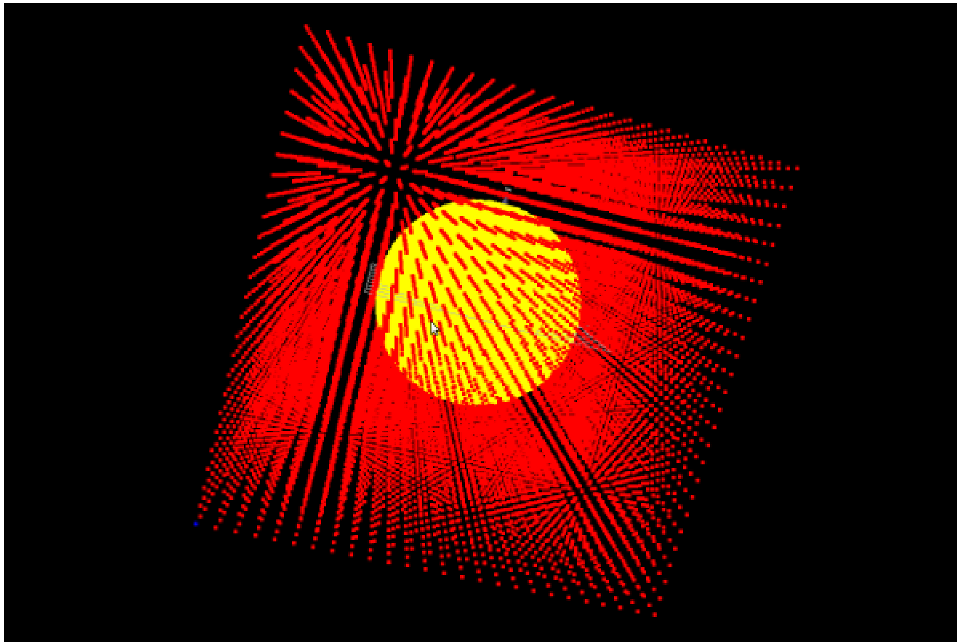


Figure 2.10: The vertices with negative distance are plotted yellow, they are the vertices inside the sphere with radius r.

2.6 Visualization

To visualize the surface represented by our SDF we use the well-known Marching Cubes algorithm, [14]. By using the fact that in the signed distance function a vertex has a negative sign if the vertex is in front of the surface and positive if it is behind, one can easily find a zero crossing by searching for vertex pairs with different signs. The zero crossing is then found by linear interpolation between these two vertices. A two dimensional sketch of the idea behind Marching Cubes is seen in Figure 2.11

In the three dimensional case one finds the zero crossing in the voxel grid and then draws triangles where the function is 0.

The approach is to look at each voxel, or cube, defined by eight neighbouring vertices in the voxel grid, illustrated in Figure 2.12. In this voxel one finds which vertices have different signs. If two neighbouring vertices have different signs, the surface goes between these two vertices and we find where by doing interpolation between the vertices. The triangles are then drawn between these interpolated points, as illustrated in Figure 2.13.

In [7] it is well described how Marching Cubes works and how one can easily implement it.

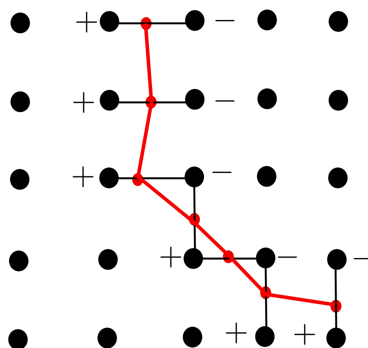


Figure 2.11: An example in 2D of how a curve can be extracted from a signed distance function. By finding the zero-crossing between vertices with different signs, one can extract the curve by drawing lines between the zero-crossings.

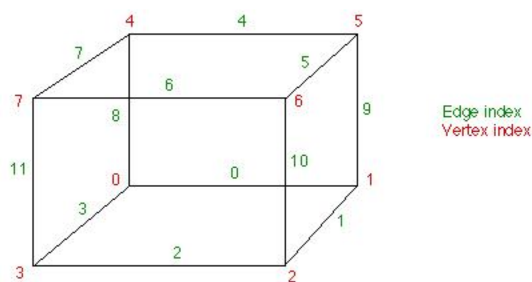


Figure 2.12: An illustration of a cube in the voxel grid with its local vertices and edges. Image obtained from http://paulbourke.net/geometry/polygonise/, accessed 2012-11-19.

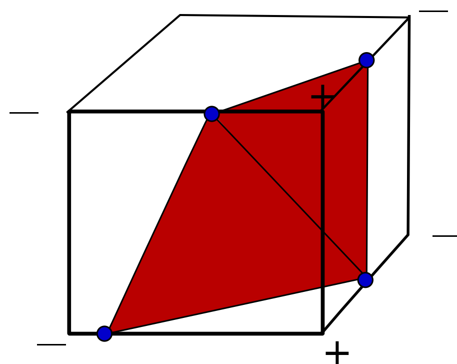


Figure 2.13: An illustration of how triangles can be drawn between the zero crossings in a voxel.

2.7 Rigid Body Motion

In this work we assume a static environment, meaning we assume that the objects we want to reconstruct do not move. The only thing we assume moving is the camera and what we want to find is a rigid body motion of the camera.

In [15] a rigid-body motion is defined as follows

Definition 4. Rigid-body motion or special Euclidean transformation. A map $g : \mathbb{R}^3 \to \mathbb{R}^3$ is a rigid-body motion or a special Euclidean transformation if it preserves the norm and the cross product of any two vectors.
1. norm: $||g(\mathbf{v})|| = ||\mathbf{v}||, \forall \mathbf{v} \in \mathbb{R}^3$.
2. cross product: $g(\mathbf{u}) \times g(\mathbf{v}) = g(\mathbf{u} \times \mathbf{v}), \forall \mathbf{u}, \mathbf{v} \in \mathbb{R}^3$.
The collection of all such motions or transformations is denoted by SE(3).

In a rigid-body motion, the distance between two points and the orientation is preserved when it is rotated and translated. That is also the reason to why we must assume a static environment so that the distance between two points is the same when the camera sees the points from different views. In [15] and in [6] much more is written about rigid-body motions.

By using homogeneous coordinates a rigid-body motion can be described by a matrix

$$\mathbf{P} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \in \mathbb{R}^{4\times 4}$$

where $\mathbf{R}$ is a rotation matrix in the *special orthogonal group* $SO(3)$ and $\mathbf{t}$ the translation in $\mathbb{R}^3$.

For simplicity we will write $[\mathbf{R}\ \mathbf{t}]$ and omit the last row, but when we write $\mathbf{P}\bar{X}$ we mean the 4×4-dimensional matrix.

A rigid-body motion can also be compactly represented by the so called twist coordinates

$$\boldsymbol{\xi} = (\omega_1, \omega_2, \omega_3, v_1, v_2, v_3).$$

To go from the twist coordinates $\boldsymbol{\xi}$ to a matrix representation one computes

$$\mathbf{P} = \begin{bmatrix} e^{\hat{\boldsymbol{\omega}}} & \frac{(I-e^{\hat{\boldsymbol{\omega}}})\hat{\boldsymbol{\omega}}\mathbf{v}+\boldsymbol{\omega}\boldsymbol{\omega}^T\mathbf{v}}{||\boldsymbol{\omega}||} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \tag{2.5}$$

where

$$\hat{\boldsymbol{\omega}} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix} \text{ and } \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}. \tag{2.6}$$

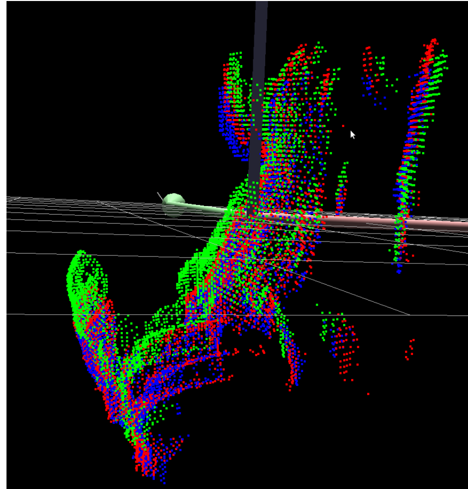


Figure 2.14: An example of how ICP works, start with two point-clouds, the red and the green. Then you want to minimize the error so that the two point-clouds fits as good as possible, as the red and blue.

2.8 Registration

To find the rotation and translation of the camera a common approach is to use Iterative Closest Point, (ICP), which was introduced by Besl and McKay in [4]. The general idea behind ICP is to align two point clouds to each other as good as possible, as seen in Figure 2.14. One has some form of error metric, sum of squared distances for instance and defines an error function. The goal is to find the correct rotation $\mathbf{R}$ and translation $\mathbf{t}$ of the camera by minimising the error function

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=0}^{M} ||(\mathbf{R}X_i + \mathbf{t}) - \hat{X}_i||^2. \tag{2.7}$$

where $\hat{X}_i$ are the 3D points which one is trying to fit the points X_i against. To do ICP one thus needs a way of finding these corresponding point pairs $(X_i, \hat{X}_i)$ so that the error metric is correct. One way of choosing point pairs is to take the closest point in the corresponding point cloud. Other methods also use the normals for each point so that one takes the closest point only if the angel between the normals are below a certain threshold. Other alternatives are to include colours if possible. Furthermore, one can measure the distance differently, one can for example use the point-to-point metric, but one can also use the so called point-to-plane method to estimate the difference between the point and the corresponding plane. Point-to-plane metric and point-to-point metric will be described in this work. In [19] a lot of different versions of ICP are described and evaluated.

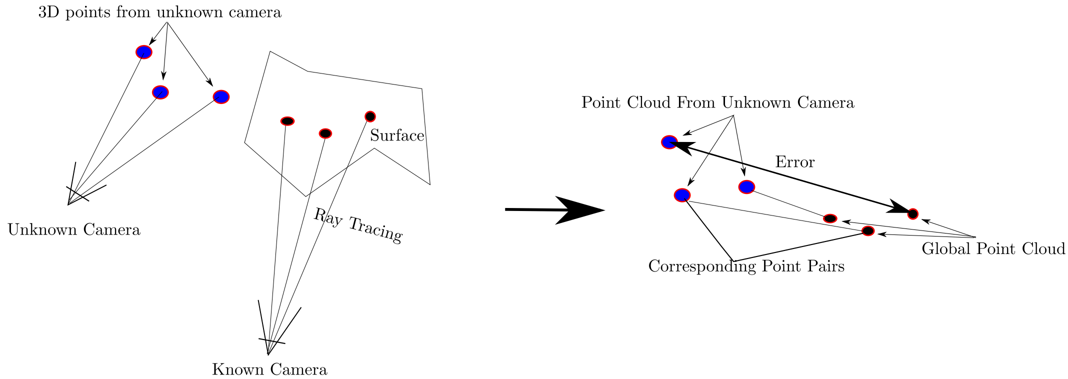


Figure 2.15: The global point cloud is estimated from the last image frame I_d^{n-1} and the last known camera position.

2.9 Related Work

Our work is in many aspects similar to the work done by Newcombe *et. al.* in [16] and [12]. To represent the surface they also use a voxel grid together with a signed distance function, just as we do. They use also the signed distance function to track the camera movement. The difference is that they use the SDF to generate a global point cloud by doing a surface prediction, illustrated in Figure 2.15. When they have created the global point cloud with help from the SDF, they use the projective data association algorithm [5] to find which points belong to each other. Thereafter they do standard ICP to estimate the correct rotation and translation of the camera.

KinectFusion uses only the information they have from the previous camera to estimate the global movement of the the camera. In our work, we make use of all the images we have obtained instead of just using the last frame as KinectFusion does. Furthermore, we do not use ICP but instead we track the camera directly against the 3D model. Our idea is that this should make the tracking more stable. Another advantage is that our approach is simpler, no data association between points in point clouds is needed for instance.

Another work recently presented is the Master Thesis [18] of Daniel Canelhas, Örebro University. The surface representation is also built on [10]. His approach to track the camera is the same as we have. This means the signed distance function is used to define an error function which is minimized in order to find the camera pose.

The difference is that in this thesis we will go deeper into the tracking and compare different norms and error metrics. We will also include colours in the reconstruction.

Endres *et al.* presents in [11] a different approach to tracking and fusion of the depth images. In the tracking part, instead of optimizing against point clouds or the global model, they use sparse feature points in the colour images. They find corresponding points in two consecutive frames and reconstruct these points to 3D points. Then a rigid-body motion is obtained by using three of these corresponding points if they are reconstructed

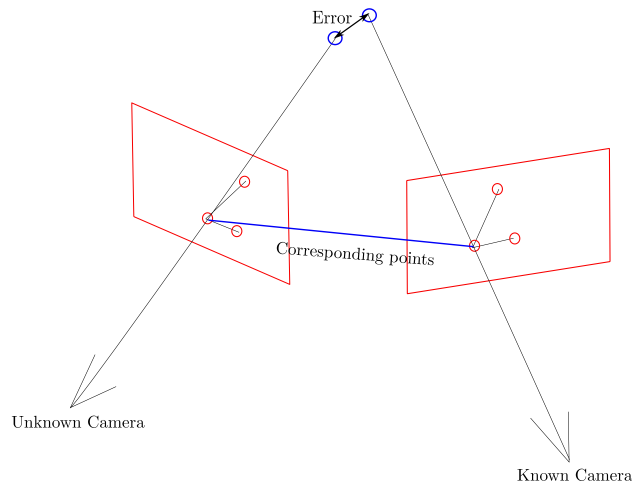


Figure 2.16: The found corresponding points are reconstructed in 3D and the error between them is calculated and if the error is small enough it can be used for estimating the rigid-body motion of the camera.

close enough. An illustration of how it works is shown in Figure 2.16

Since the tracking is made by using frame-to-frame, the tracking easily drifts away. Therefore, in order to create a global consistent trajectory they add additional constraints between non-consecutive images and use a graph optimizing library called g^2o.

The last step when the trajectory is found is to compute a representation of what the camera sees. To do this they use a 3D occupancy grid, which is a voxel representation where each voxel is either occupied or unoccupied. To make this memory efficient the voxels are represented by using so called octrees. More information about octrees and the representation of geometry using occupancy grid can be found in the work by Wurm *et al.* [23].

The advantage with this totally different approach is that they use colours. Tracking approaches like ICP and ours are prone to fail when there is not enough structure in the scene. The solution to the error function one minimizes cannot be uniquely determined. RGB-D SLAM does not suffer from this weakness if there are enough visual features in the image. On the other hand, if there is only one colour in the image, but a lot of structure, then ICP and our approach will succeed but RGB-D SLAM will fail.

Chapter 3

Reconstruction of 3D Surfaces

In this chapter we will present our approach to fuse the depth images into a complete 3D model and our approach to colourise it. We start by defining two different metrics to estimate the distance between the vertices in our voxel grid and the surface. Then we present a method to find a signed distance function which can represent the geometry seen from all cameras and at last we will present a simple method for how one can put colours onto the surface.

3.1 Projective Point-To-Point Metric

In chapter 2 we saw an example for how a sphere could be represented using a signed distance function and a voxel grid. The approach is to estimate the distance between each vertex and the surface on the sphere. This distance can be estimated exactly since we have a analytical expression for the signed distance function for a sphere. In general we cannot expect to have that, so we need a method for estimating the distance between the vertices and the surface no matter how the surface looks like.

There exists a method called the *Fast Marching Method*, (FMM), which is well described in [3]. FMM estimates the exact distance for all vertices to the surface. However, it is not fast enough for our application since we want to do real-time reconstruction. Therefore, we will instead look at projective metrics which are much faster to compute and it can also be made in parallel, which is suitable for real-time applications.

The first metric we consider is the projective point-to-point metric which is also used by Levoy and Curless [10].

The distance will be between a point X in the voxel grid and a point Y on the surface. The trick now is to choose the point Y on the surface appropriate. Ideally, we want to measure

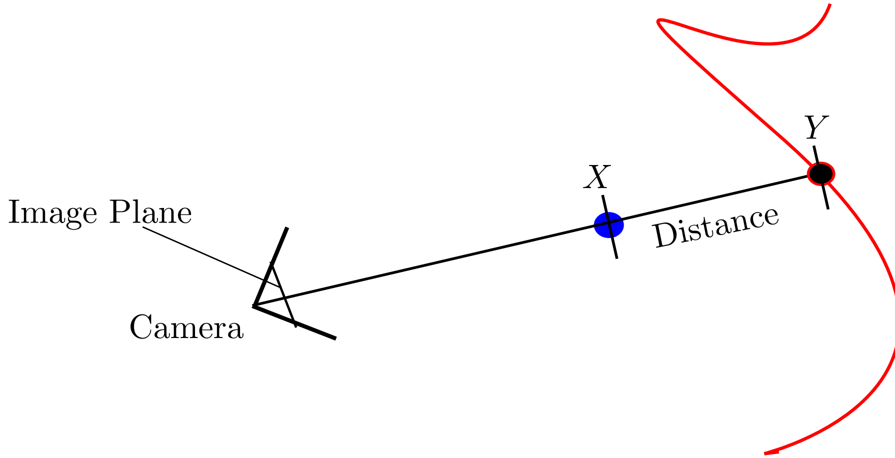


Figure 3.1: The idea behind projective point-to-point metric is to measure the distance between a point X and the point Y along the ray between X and the camera center.

the distance between the point X and the point Y on the surface which lies closest to X, because then we would get the exact signed distance.

However, it is not efficient to estimate the true distance for each vertex. Instead we choose the point Y on the surface so that Y lies on the line between the camera and the point X, as in Figure 3.1.

The distance between the point X and the point Y on the surface is then the length of the line between them.

In general this will not be the true distance, but it is much faster to compute and gives a good enough approximation if one includes weights and truncations, as we will see soon.

We start with explaining how the projected distance is estimated for each vertex. In this chapter we will use the functions which are defined in chapter 2.

Our goal now is to estimate the projected point-to-point distance for each vertex in the voxel grid. Assume for now that we want to find the projected distance for the vertex $V_{i,j,k}$.

What we need to get the projected distance as seen in Figure 3.1 is first of all the position of the vertex in the local coordinate system defined by the camera. Since the vertex $V_{i,j,k}$ has a fixed position in the space, we know what global coordinates, X_G, the vertex $V_{i,j,k}$ has. For the moment, the camera matrix $\mathbf{P}$ is also assumed to be known.

Then we can calculate the local coordinates X_L for $V_{i,j,k}$ by

$$\bar{X}_L = \mathbf{P}^{-1}\bar{X}_G \tag{3.1}$$

which is also illustrated in Figure 3.2. Now we need to find the point on the surface, corresponding to the vertex $V_{i,j,k}$ and the coordinates for that point in the local coordinate system, as seen in Figure 3.3. Since Y_L lies on the ray between the camera center and X_L,

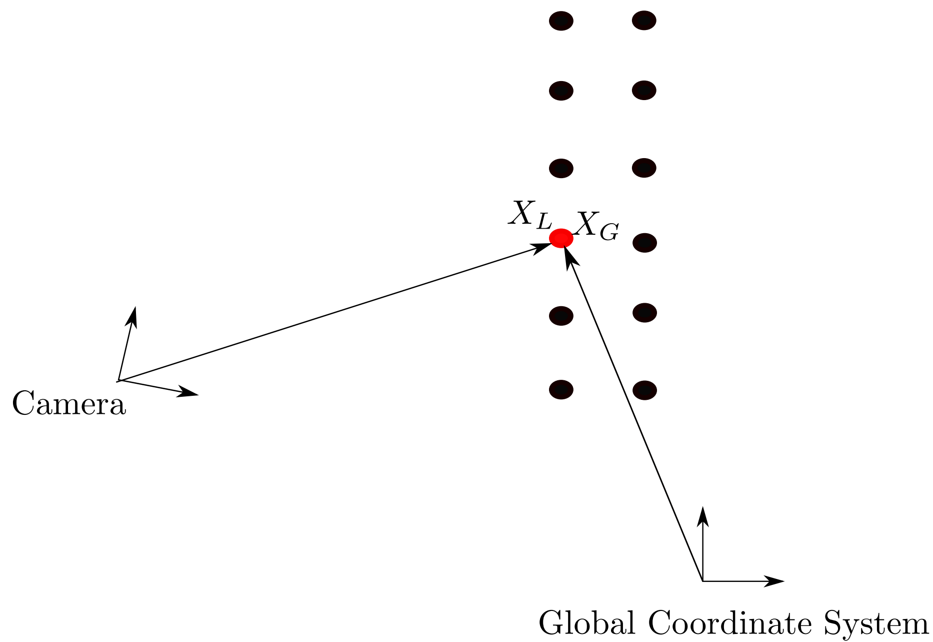


Figure 3.2: Each vertex can be described with the coordinates in the global system, but also with coordinates in the local camera system.

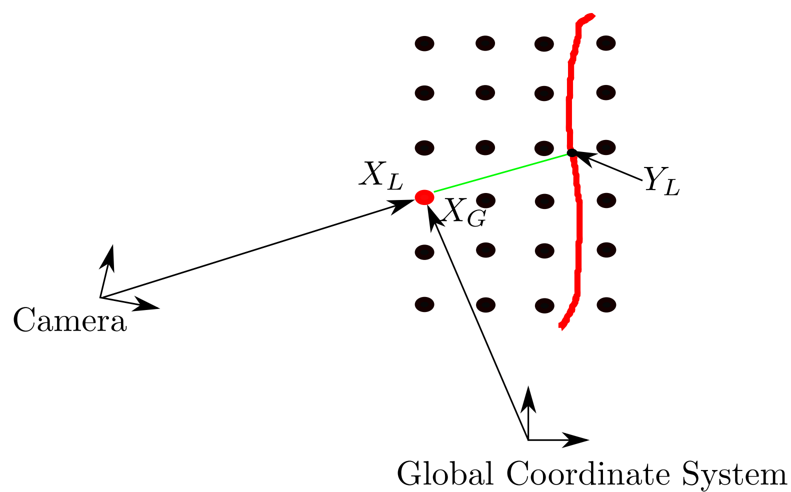


Figure 3.3: Here one can see the local and global coordinates X_G and X_L for the vertex and the local coordinates for the point Y_L on the surface.

Y_L will be projected to the same pixels as X_L.

By using our function π we can calculate the pixels corresponding to Y_L by projecting X_L onto the image plane

$$(x, y) = \pi(X_L). \tag{3.2}$$

From the pixel coordinates (x, y), we can now find the z-value by

$$z = I_d(x, y). \tag{3.3}$$

This is the z-coordinate for the surface point Y_L we are looking for. To find the coordinates for Y_L we use our function ρ

$$Y_L = \rho(x, y, z). \tag{3.4}$$

Now it is simple to compute the projected signed distance for vertex $V_{i,j,k}$ by taking

$$||X_L|| - ||Y_L||. \tag{3.5}$$

If X_L is in front of the surface

$$||X_L|| - ||Y_L|| < 0 \tag{3.6}$$

and if X_L is behind the surface then

$$||X_L|| - ||Y_L|| > 0. \tag{3.7}$$

Hence, the distance is signed.

We can now define an algorithm, Algorithm 1, which computes the projected signed distance for a vertex with index (i, j, k).

Algorithm 1: Estimating the Projected Point-To-Point Distance

Input: (i,j,k)

`// Each vertex at position` (i, j, k) `has known global 3D coordinates`

1 $X_G \leftarrow (i, j, k)$

`// Compute the coordinates in the cameras local coordinate system`

2 $\bar{X}_L \leftarrow \mathbf{P}^{-1}\bar{X}_G$

`// Project` X_L `onto the image plane`

3 $(x, y) \leftarrow \pi(X_L)$

`// Find the value in the image at that point.`

4 $z \leftarrow I_d(x, y)$

`// Compute the projected point-to-point distance`

5 $d(i, j, k) = ||X_L||$ - $||\rho(x, y, z)||$

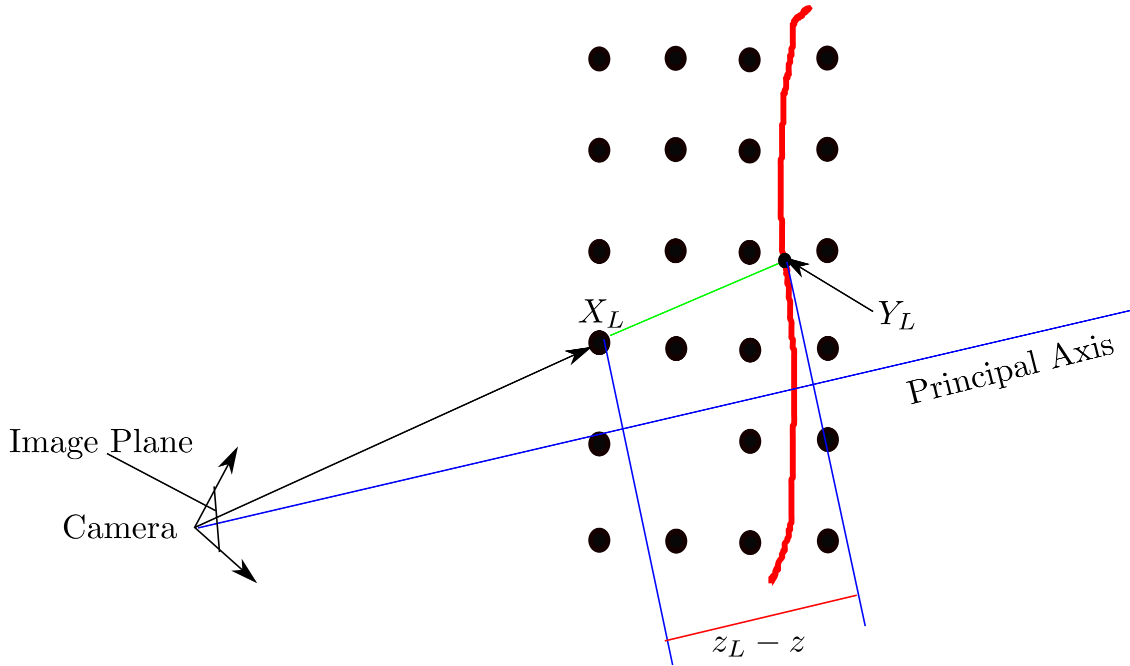


Figure 3.4: By measuring the distance along the principal axis instead, one gets an approximated point-to-point distance.

Another possibility to approximate the signed point-to-point distance is by just using the z-coordinates for X_L, z_L, and the z-value for Y_L, z, as seen in Figure 3.4.

In the last row of Algorithm 1 we then compute

$$d(i, j, k) = z_L - z \tag{3.8}$$

instead, where z_L is the local z-coordinate for the vertex $V_{i,j,k}$ and z the local z-coordinate for Y_L.

The advantage with this simplified version is that it is faster to compute.

Now we have defined a method for computing the distance between the surface and a vertex. By applying this method to all our vertices, we do get the projected distance for all the vertices.

Note that we will only measure the distance for those vertices which are projected onto image plane and in front of the camera.

By looking at Figure 3.5, one sees that the projective point-to-point metric might not always be the best choice. Therefore, we now introduce the point-to-plane metric.

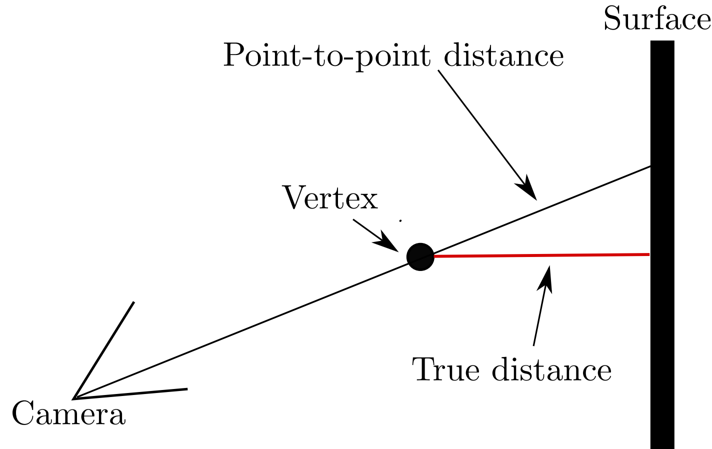


Figure 3.5: When measuring the distance against a plane, the projected point-to-point will fail to give an accurate distance.

3.2 Projective Point-To-Plane Metric

We will now introduce another projective metric which is called point-to-plane. The idea with the point-to-plane metric is that it gives a more accurate measure of the distance between a point and the surface, when the surface is locally planar. Point-to-plane is presented in [9] to improve ICP, we propose the following method to estimate the point-to-plane distance for each vertex.

To estimate the point-to-plane metric one projects the vector between the surface point and the vertex point onto the normal at the surface. This is illustrated in Figure 3.6. If the surface is a wall, the measured distance will be exact. As for the point-to-point metric, we want to estimate the point-to-plane distance for each vertex in the voxel grid in order to represent the signed distance function.

To calculate the projected point-to-plane distance for $V_{i,j,k}$ we need again find the corresponding point on the surface Y_L and then find the normal for Y_L. To find the corresponding point on the surface for vertex $V_{i,j,k}$ we do exactly as we do for the point-to-point metric.

The difference now is that we want to estimate the normal at Y_L which we can use to compute the point-to-plane metric, as illustrated in Figure 3.7. Since we are assuming that the surface around Y_L is locally planar, we want to find the neighbouring points to Y_L so that we can estimate the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ which lie in the (assumed) plane and take the cross product between $\mathbf{v}_1$ and $\mathbf{v}_2$ to get the normal.

By using the pixels (x, y) which Y_L is projected to, we assume that the neighbouring pixels $(x+1, y)$ and $(x, y+1)$ corresponds to neighbouring points to Y_L. By using the function

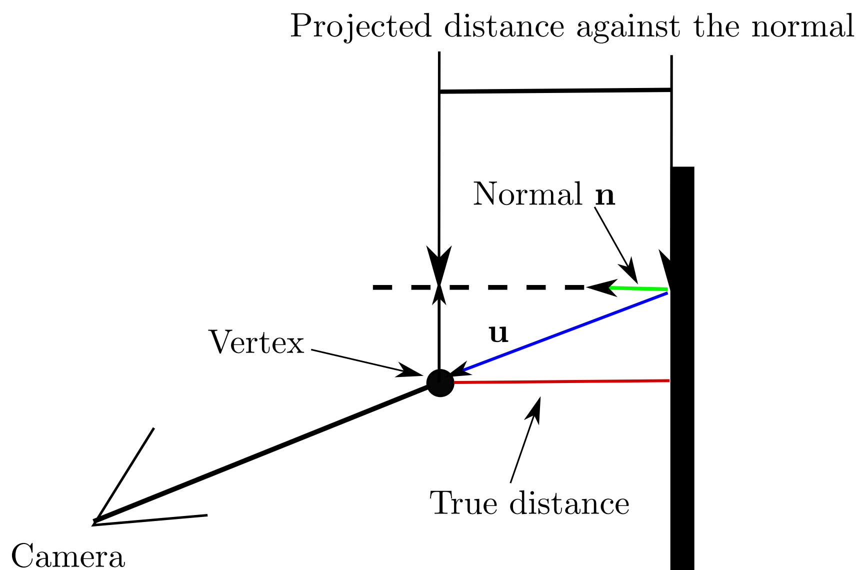


Figure 3.6: The projection of the vector **u** onto the normal gives a better approximation of the distance between the vertex and the plane.

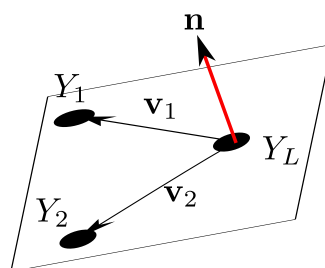


Figure 3.7: For the surface point Y_L, we want to find neighbouring points in order to define the vector $\mathbf{v}_1$ and $\mathbf{v}_2$ which can be used to estimate the normal.

ρ, we calculate two neighbouring points Y_1 and Y_2 from the pixels $(x+1, y)$ and $(x, y+1)$

$$Y_1 = \rho(x+1, y, I_d(x+1, y)) \tag{3.9}$$
$$Y_2 = \rho(x, y+1, I_d(x, y+1)). \tag{3.10}$$

Now we use Y_L, Y_1 and Y_2 to find the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$

$$\mathbf{v}_1 = Y_1 - Y_L \tag{3.11}$$
$$\mathbf{v}_2 = Y_2 - Y_L. \tag{3.12}$$

To estimate the normal $\mathbf{n}$ for the plane which $\mathbf{v}_1$ and $\mathbf{v}_2$ lie in, the cross product between $\mathbf{v}_1$ and $\mathbf{v}_2$ is computed

$$\mathbf{n} = \frac{\mathbf{v}_1 \times \mathbf{v}_2}{||\mathbf{v}_1 \times \mathbf{v}_2||}. \tag{3.13}$$

What we need to find now is the vector $\mathbf{u}$ between X_L and Y_L

$$\mathbf{u} = X_L - Y_L. \tag{3.14}$$

In Figure 3.6 it is the blue vector we are seeking. Now we have everything to estimate the projected point-to-plane metric. By projecting the vector $\mathbf{u}$ onto the normal $\mathbf{n}$, we get a new vector $\mathbf{s}$

$$\mathbf{s} = \frac{\mathbf{u}^T\mathbf{n}}{\mathbf{n}^T\mathbf{n}}\mathbf{n} = (\mathbf{u}^T\mathbf{n})\mathbf{n}. \tag{3.15}$$

To get the point-to-plane distance we calculate the length of the resulting vector $\mathbf{s}$

$$||\mathbf{s}|| = ||(\mathbf{u}^T\mathbf{n})\mathbf{n}|| = |(\mathbf{u}^T\mathbf{n})|||\mathbf{n}|| = |(\mathbf{u}^T\mathbf{n})|. \tag{3.16}$$

Hence, the length of the vector $\mathbf{s}$ is simply the absolute value of the scalar product between the normalized normal $\mathbf{n}$ and the vector $\mathbf{u}$.

To get the sign right we just compute

$$\mathbf{u}^T\mathbf{n} \tag{3.17}$$

with appropriate sign of $\mathbf{n}$.

We can summarize the method to calculate the point-to-plane distance for a vertex $V_{i,j,k}$ by Algorithm 2.

By applying this to each vertex in the field of view, we can estimate the signed distance function with the point-to-plane metric instead of the point-to-point metric.

Algorithm 2: ESTIMATING THE POINT-TO-PLANE DISTANCE FOR A VERTEX

Input: (i, j, k)

// The global coordinates for vertex (i, j, k)

1 $X_G \leftarrow (i, j, k)$

// Transform the global point X_G to the local camera system

2 $\bar{X}_L \leftarrow \mathbf{P}^{-1}\bar{X}_G$

// Calculate the pixel coordinates

3 $(x, y) \leftarrow \pi(X_L)$

// Compute corresponding point on the surface

4 $Y_L \leftarrow \rho(x, y, I_d(x, y))$

// Compute the neighbouring points on the surface

5 $Y^1 \leftarrow \rho(x + 1, y, I_d(x + 1, y))$

6 $Y^2 \leftarrow \rho(x, y + 1, I_d(x, y + 1))$

// Calculate the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$

7 $\mathbf{v}_1 \leftarrow Y^1 - Y_L$

8 $\mathbf{v}_2 \leftarrow Y^2 - Y_L$

// Compute the normal

9 $\mathbf{n} \leftarrow \frac{\mathbf{v}_1 \times \mathbf{v}_2}{||\mathbf{v}_1 \times \mathbf{v}_2||}$

// Compute $\mathbf{u}$

10 $\mathbf{u} \leftarrow X_L - Y_L$

// Compute the signed distance d

11 $d = \mathbf{u}^T\mathbf{n}$

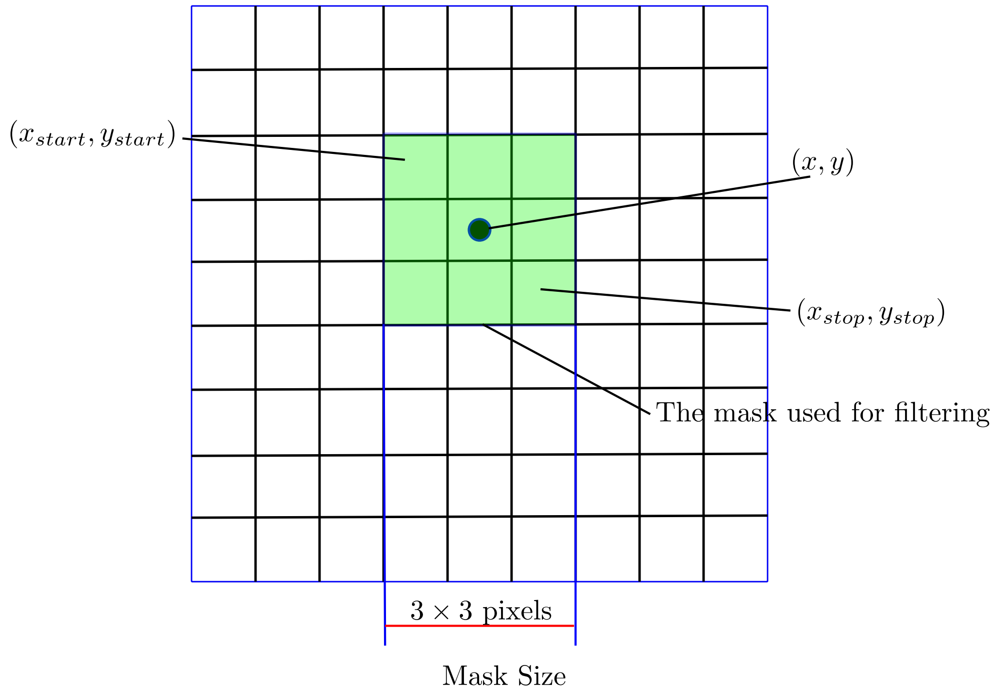


Figure 3.8: The thresholded average is taken for the pixels within the mask.

In the depth images there will be noise, the random error of the depth measurement increases the further away the object is. In [13] they show that the error in the depth measurement for the Kinect sensor varies between a few millimetres up to about 4 cm at the maximum range of the sensor. To get a better approximation of the normals we therefore filter the image before the normals are estimated. A commonly used filter applied to depth images is the edge preserving bilateral filter presented by Tomasi and Manduchi in [21]. There exists an open-source implementation in the OpenCV library for bilateral filtering. The drawback with this implementation is that it also smooths pixels without any depth information. Therefore, instead of implementing our own bilateral filter, we do a simpler filtering where we take the average of the neighbouring pixels which z-value does not lie to far away from the center value. In Figure 3.8 one can see how the mask is defined and the algorithm is described in Algorithm 3.

We have now seen three different metrics we can use to estimate the distance between the vertices and the surface, we have two versions of point-to-point metric and one version of point-to-plane.

Even though projected point-to-plane gives better approximation than projective point-to-point, we can still see in Figure 3.9 that projective metrics can give large errors for some vertices when the surface is just next to the vertex but not along the projected ray.

Algorithm 3: FILTERING AROUND PIXELS (x, y) WITH THRESHOLD δ

```
   // Find the value in the pixel we want to filter around
1  Z_C ← I_d(x, y)
2  sum ← 0
3  count ← 0
4  for i = x_start to i ≤ x_stop do
5      for j = y_start to j ≤ y_stop do
6          z ← I_d(x + i, y + j)
           // If the difference between the center point Z_C and z is below
              the threshold, we use z to compute the average
7          if (|z − Z_C| < δ) then
8              sum+ = z
9              count + +
   // Since we only take the average of the pixels which are close
      enough, we do not smooth over too sharp edges.
10 return sum/count
```

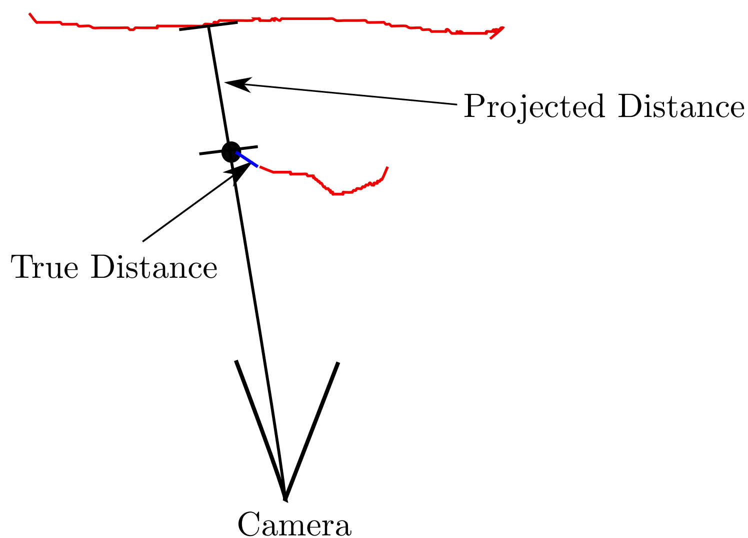


Figure 3.9: The projected distance clearly fails to give a good approximation of the closest distance to the surface.

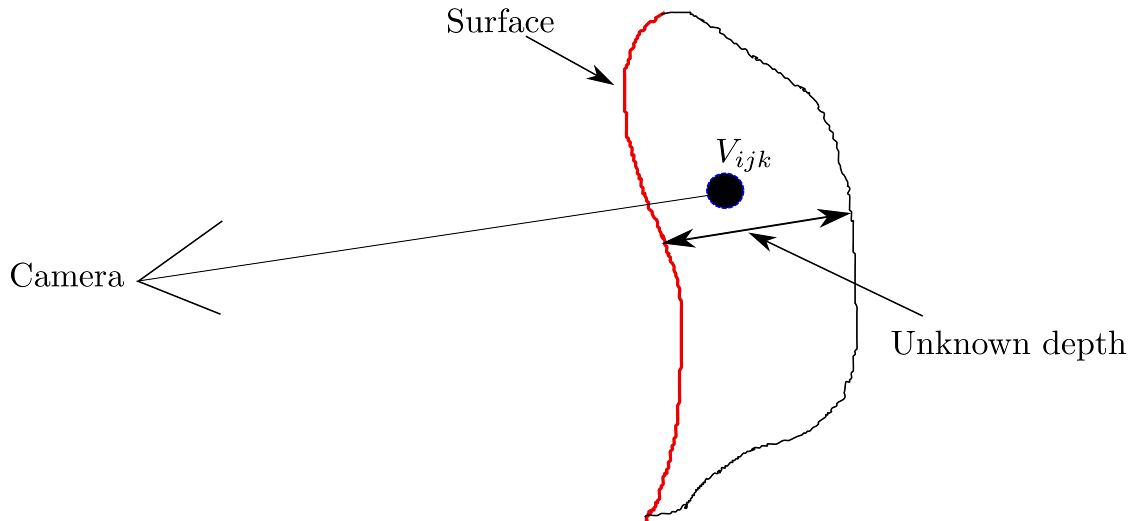


Figure 3.10: From the cameras field of view, we cannot see what is behind the surface.

3.3 Estimating the Signed Distance Function

In this section, we will propose a method to decrease the impact of the potentially large errors one gets while using a projective distance. This will be done by truncation of the distance function and also by the introduction of weights. The last part of this section will be about how to find a signed distance function which takes all depth images into account so that we can represent the complete geometry.

As we already have seen in Figure 3.9, there are some obvious drawbacks with projective metrics. Even though we use point-to-plane instead of point-to-point, there will still be some measures which are very bad compared to the true closest surface.

The first problem is when measuring the projected distance for the vertices behind the seen surface, we cannot be really sure about what is behind that surface since we do not know anything about the thickness of the object as illustrated in Figure 3.10.

In the work by Levoy and Curless, [10], it is suggested that for each measured distance $d_{i,j,k}$, corresponding to the the vertex $V_{i,j,k}$, we introduce the weight $w_{i,j,k}$. We can then compute a weighted distance $w_{i,j,k}d_{i,j,k}$. The weight makes it possible to handle uncertainties in the measurements, where for instance a vertex can be behind the object and nothing is known about the thickness of the object.

It is not enough to just introduce a weight function. We can still get measures which are highly erroneous in front of the surface as can be seen in Figure 3.9.

However, the closer the vertex is to the surface, the more accurate will the projected distance be. Therefore, to decrease the impact of highly erroneous measurements, Levoy and Curles [10] suggests that one shall truncate the projected distance by a certain threshold.

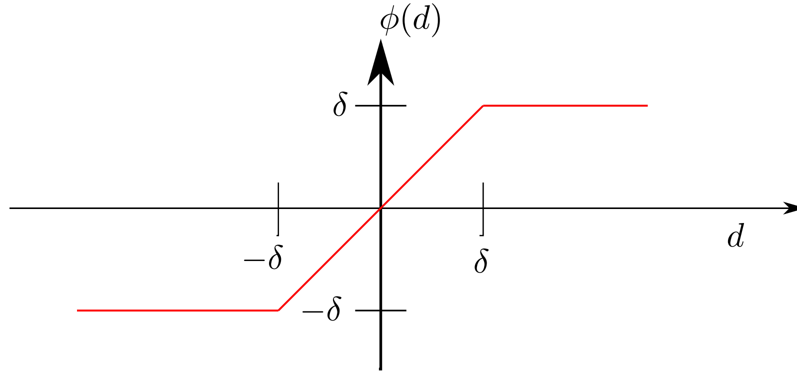


Figure 3.11: An illustration of how the truncated distance function looks like, when $|d_{i,j,k}| > \delta$, the distance is truncated.

This means that when $|d_{i,j,k}| > \delta$, d is set to either δ or $-\delta$.

The truncated signed distance function looks as follows

$$\phi(d_{i,j,k}) = \begin{cases} -\delta & \text{if} \quad d_{i,j,k} < -\delta \\ d_{i,j,k} & \text{if} \quad |d_{i,j,k}| \leq \delta. \\ \delta & \text{if} \quad d_{i,j,k} > \delta \end{cases} \tag{3.18}$$

We also have a corresponding weight to each measurement

$$w(d_{i,j,k}) = \begin{cases} 1 & \text{if} \quad d_{i,j,k} < \epsilon \\ e^{-\sigma(d_{i,j,k}-\epsilon)^2} & \text{if} \quad d_{i,j,k} \geq \epsilon \text{ and } d_{i,j,k} \leq \delta. \\ 0 & \text{if} \quad d_{i,j,k} > \delta \end{cases} \tag{3.19}$$

The weight is one as long as we are sure about the distance and thereafter the weight decreases exponentially until we reach δ when the weight is put to 0. The parameter ϵ decides how far behind the surface we want to go before we consider the uncertainty to be lower than one.

Note that for a true signed distance function

$$|\nabla\psi| = 1 \tag{3.20}$$

must hold.

This constraint will in general be violated by our approach when we truncate the distance and assign a weight to it.

A sketch for how the distance function and the weight function might look like can be seen in Figure 3.11 and Figure 3.12.

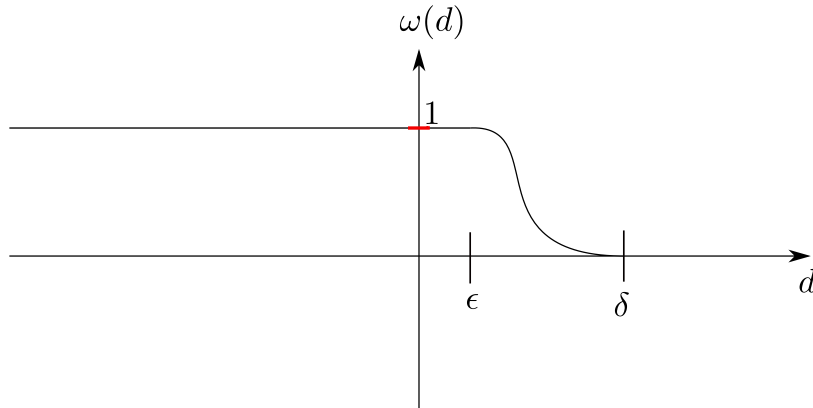


Figure 3.12: An illustration of how the weight function looks like, as long as $d_{i,j,k}$ is less than ϵ the weight is 1. Thereafter the weight is decreasing until δ, thereafter it is put to 0.

To estimate the truncated and weighted signed distance function ψ_n for a frame I_d^n when we also know the corresponding camera matrix $\mathbf{P}_n$, we start with choosing a metric d. We take either point-to-point or point-to-plane. Then for all vertices in the voxel grid, which are in the field of view, we calculate the truncated and weighted signed distance $\psi(i, j, k) = \phi(d_{i,j,k})w(d_{i,j,k})$ and assign that distance to the vertex.

Thus, we have an estimation of the truncated and weighted signed distance function ψ_n and now ψ_n is a representation for the surface seen from camera C_n.

In the sequel, when we refer to a signed distance function, we will mean a truncated and weighted signed distance function.

However, this far, we have only described how to estimate the signed distance function ψ_n for one image I_d^n. The goal with the fusion is obviously to integrate all depth images into one single global 3D model. This way a complete room can be reconstructed if each part of the room is seen in one of the images.

For each frame I_d^n we have estimated a signed distance function ψ_n. To get a representation of the whole geometry, we need to find an optimal signed distance function ψ^* which contains information from all SDFs we have estimated. This single SDF ψ^* shall then represent the geometry seen from all images.

Levoy propose in [10] a way to do this by formulating it as an optimization problem

$$E(\psi) = \sum_{i=0}^{N} w_i ||\psi - \psi_i||^2 \tag{3.21}$$

where N is the number of estimated SDFs.

We want to find the optimal SDF ψ^* which gives the lowest error. In this case, the solution is simple. The SDF which gives the lowest error is the weighted average of all the SDFs

ψ_n we have estimated. This is seen by taking the derivative of $E(\psi)$

$$\frac{\partial E}{\partial \psi} = 2\sum_{i=0}^{N} w_i(\psi - \psi_i) = 0 \tag{3.22}$$

$$\iff$$

$$\psi = \frac{\sum_{i=0}^{N} w_i \psi_i}{\sum_{i=0}^{N} w_i}. \tag{3.23}$$

This makes it fast to compute the optimal SDF, since it can be calculated as a running weighted average. This is done iteratively, after the first estimated SDF we calculate the second one. Then we take the weighted average of these two and obtains ψ_1^*. Then we estimate the third SDF and then we take this SDF ψ_3 and uses ψ_1^* and takes the weighted average between these two, resulting in ψ_2^*. This way, the optimal SDF ψ^* is computed as we receive images.

Levoy and Curless suggests the following updating procedure for each vertex to calculate the weighted average of the SDFs

$$D^{n+1} = \frac{D^n W^n + \phi(d^{n+1}) w(d^{n+1})}{W^n + w(d^{n+1})} \tag{3.24}$$

$$W^{n+1} = W^n + w(d^{n+1}). \tag{3.25}$$

D^n is the weighted, truncated and averaged distance obtained from the first n images and W^n is the total weight obtained. $w(d^{n+1})$ is the weight function defined in (3.19) and $\phi(d^{n+1})$ is the function defined in (3.18) and d^{n+1} is the estimated distance between the vertex and the surface seen from frame I_d^{n+1} obtained from either point-to-point or point-to-plane.

By calculating this running average, we calculate directly the optimal signed distance function instead of first estimating N different SDFs and then finding the optimal SDF. Since this updating procedure is done for each vertex and the vertices does not depend on each other, it is possible to do the calculations in parallel.

By using the depth images and the corresponding camera matrices for the Teddy bear data set received from [20] we can estimate the signed distance function as we have described above by using the point-to-point metric. When this is done for all 1376 images, we do Marching Cubes and draw the triangles. The result can be seen in Figure 3.13.

3.4 Colourising

By using the colour images received from the depth sensor, it is also possible to approximate the colours of the surface. We propose a very simple and straight forward way of estimating

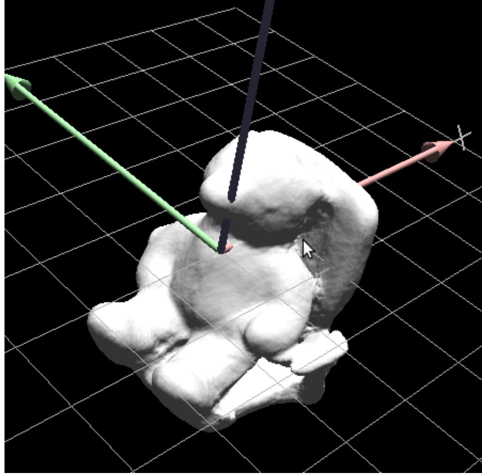


Figure 3.13: This is the result of fusing 1376 images from the data set Teddy bear by using point-to-point metric and doing Marching Cubes.

the colours. Each colour image contains a three dimensional vector

$$\mathbf{c} = (r, g, b) \tag{3.26}$$

where r is the intensity of red, g the intensity of green and b the intensity of blue. By estimating the colour intensities for each vertex which is close enough to the surface we can get a colourised 3D model.

The approach is simply to estimate the correct RGB-vector for each vertex. If the vertex is close enough to the surface, then we can use the pixels coordinates (x, y) used in the depth image to find the corresponding RGB-vector in the colour image I_c.

Algorithm 4: Approximating The Colour For A Vertex

Input: (i, j, k)
1. $(x, y) \leftarrow \pi(X_L)$
   `// Find the RGB-vector in the new colour image`
2. $\mathbf{c} = I_c(x, y) \in \mathbb{R}^3$
3. **if** V_{ijk} *is close to surface* **then**
   `// Updating the colour intensities for vertex` (i, j, k)
4. $R^{n+1}_{i,j,k} = \frac{R_{i,j,k}n+\mathbf{c}(0)}{n+1}$
5. $G^{n+1}_{i,j,k} = \frac{G_{i,j,k}n+\mathbf{c}(1)}{n+1}$
6. $B^{n+1}_{i,j,k} = \frac{B_{i,j,k}n+\mathbf{c}(2)}{n+1}$
7. $n++$

Every time the vertex is close to the surface, we will obtain a RGB-vector from the colour image. By computing a running average, we use the colour information from all the colour images. The algorithm for estimating the colours is shown in Algorithm 4.

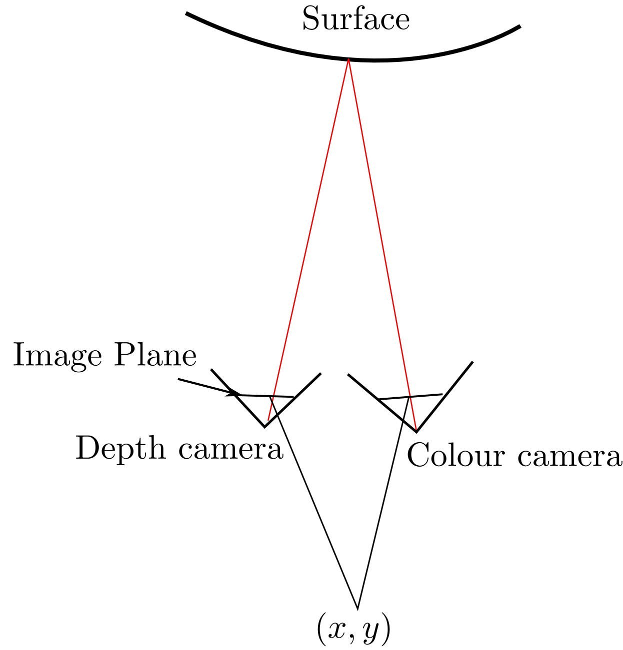


Figure 3.14: The pixel coordinates in the depth image is used to find the colour information in the colour image.

Another approach to update the RGB-vector for the vertices is by computing

$$R_{i,j,k}^{n+1} = \frac{R_{i,j,k}^{n} + \mathbf{c}(0)}{2} \tag{3.27}$$

$$G_{i,j,k}^{n+1} = \frac{G_{i,j,k}^{n} + \mathbf{c}(1)}{2} \tag{3.28}$$

$$B_{i,j,k}^{n+1} = \frac{B_{i,j,k}^{n} + \mathbf{c}(2)}{2}. \tag{3.29}$$

That way, the latest colour intensities gets a higher weight than the old intensities.

When extracting the surface, each corner is assigned a RGB-vector which gives the triangle its colour. An example is shown in Figure 3.15.

After that this had been implemented, we got to know about a work by Whelan *et al.* [22] where they also use voxel grid of the same size as the grid for the SDF to represent the colours, however, they do it in a more refined way by not updating vertices close to surface edges and also a more advanced weighting where they use the angle between the vertex and the camera to weight the colours so that vertices looked more "straight on" are weighted more than those seen from a bigger angle. This seems to give better results but to the cost of more complexity and computational time.

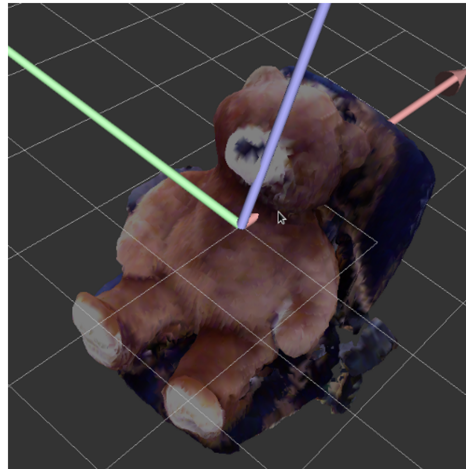


Figure 3.15: An example of how the colourising works. Each corner in the triangle is assigned a rgb-vector which will give the correct colour fro the triangle.

Summary

We have now seen different approaches to estimate the distance between a vertex $V_{i,j,k}$ and the surface, both the point-to-point metric and the point-to-plane metric. Furthermore, we have seen how we can reduce the error in the approximation by using a weight function and truncation of the distance.

We have also seen how to estimate a signed distance function ψ_n from a depth image I_d^n, with a known camera matrix $\mathbf{P}_n$. By applying the updating procedure described in equation (3.24) and (3.25), we can fuse all the SDFs into one single SDF which is a representation of the whole geometry seen so far.

We have also presented a simple approach to colourise the 3D model.

Chapter 4

Tracking

In the previous chapter we introduced a method for fusing the surfaces from the depth images into one single 3D reconstruction. As we see in Figure 3.13, the result looks good and the method works. However, it is based on a key assumption, that is, we must know how the camera is rotated and translated with respect to the global coordinate system. The reason is that we cannot estimate the distances between the surface and the vertices without the correct camera configuration.

4.1 Introduction To the Approach

When the depth images in Figure 3.13 are fused the camera pose is already known. To create the data set in [20] an external motion capture system was used. This gives a very accurate estimation of the camera pose, but it is expensive and ineffective. In this work, we aim to build a system which can do dense 3D reconstruction in real-time. Therefore, we need a completely different approach.

In 2 we saw that there exists well studied methods such as ICP and RGB-D SLAM. In particular, we saw that Microsoft has already presented a work called KinectFusion, [16], which can do dense 3D reconstruction in real-time. Just as KinectFusion we will use the signed distance function to define an error function which minimum gives us the correct camera configuration. However, in contrast to KinectFusion, we will use the signed distance function directly instead of using it to estimate a point cloud with ray tracing. Our approach minimizes the error between a point cloud and the surface instead of minimizing the error between two point clouds as in [16]. Moreover, we use the information from all the previous images instead of just the previous image to track the camera.

The main idea is, assuming we have found the cameras $C_0 ... C_n$, we make an initial guess

$$\mathbf{P}^0 = [\mathbf{R} \ \mathbf{t}] \tag{4.1}$$

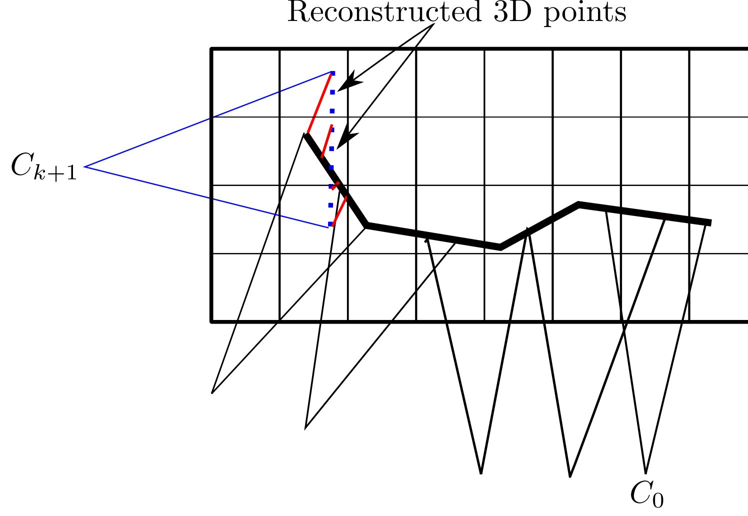


Figure 4.1: The basic idea behind our approach. The surface from frame I_d^{n+1} is reconstructed in the voxel grid, by finding the value in the SDF where a point is reconstructed, we get the distance between the point and the surface.

for camera C_{n+1} and reconstructs the point cloud from frame I_d^{n+1} by using our camera matrix $\mathbf{P}^0$.

Each point will then be reconstructed somewhere in the voxel grid as seen in Figure 4.1. From our signed distance function we can then estimate the distance between the points and the surface. We want to find the rotation and translation of the camera C_{n+1} so that as many points as possible are reconstructed on the surface. To find the correct rotation and translation, we define an error function where we use our SDF to find the distance between each point reconstructed in the voxel grid and the surface

$$
\begin{aligned}
E(\mathbf{R}, \mathbf{t}) = \sum_{i=0}^{M} \sum_{j=0}^{N} ||\psi(\mathbf{R}X + \mathbf{t})||^2 \\
\mathbf{R} \in SO(3) \\
\mathbf{t} \in \mathbb{R}^3 \\
(i, j) \in I_d^{n+1} \\
X = \rho(i, j, I_d^{n+1}(i, j)) \in \mathbb{R}^3 \\
M \text{ is the number of rows in the image} \\
N \text{ is the number of columns in the image.}
\end{aligned}
\tag{4.2}
$$

If the rotation $\mathbf{R}$ and the translation $\mathbf{t}$ are correctly estimated, then the points will be reconstructed on the surface and

$$\psi(\mathbf{R}X + \mathbf{t}) = 0.$$

Thus, by minimizing (4.2), we will find the correct rotation and translation of the camera.

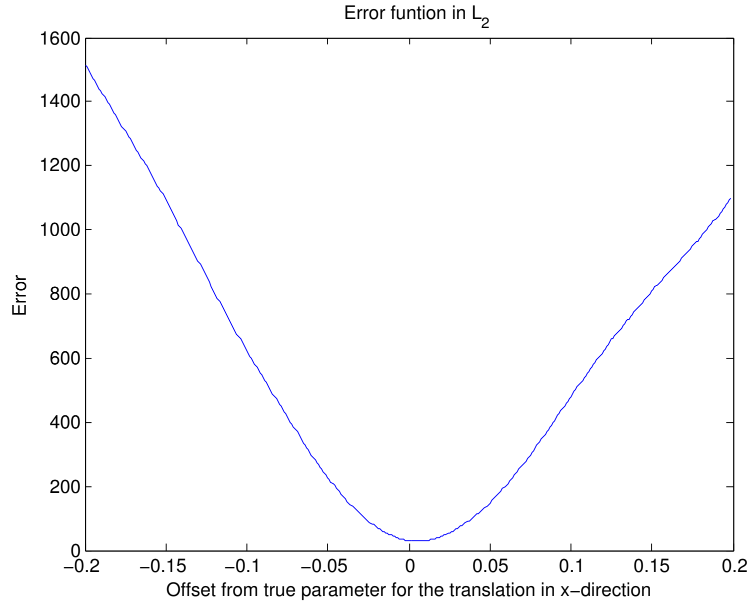


Figure 4.2: The error is plotted with different offsets from the true value of the parameter for the translation along the x-axis.

By using the ground truth data from [20] we can plot how the error function looks like with different offsets from the true value, as seen in Figure 4.2.

4.2 Optimisation

To find the correct rotation $\mathbf{R}$ and translation $\mathbf{t}$ for the camera we need to minimize (4.2).

We will look at some different norms, namely L_2, L_1 and truncated L_2. We start with the L_2-norm.

4.2.1 L_2-norm

In this part we will present a method for optimizing the error function in the L_2 norm. What we want to find is the rotation and translation

$$\mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \quad \mathbf{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_2 \end{pmatrix}. \tag{4.3}$$

To get a more compact representation of the rotation and translation we use that a rotation can be expressed as the exponential matrix of a skew symmetric matrix $\hat{\boldsymbol{\omega}}$ as defined in

chapter 2. We can shortly describe the rotation and translation by the 6-dimensional vector

$$\xi = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ t_1 \\ t_2 \\ t_3 \end{pmatrix}$$

where $(\omega_1, \omega_2, \omega_3)$ defines the rotation and (t_1, t_2, t_3) defines the translation. The translation $\mathbf{t}$ as defined in chapter 2 depends on $\hat{\boldsymbol{\omega}}$ and the vector $\mathbf{v} = (v_1, v_2, v_3)$ but we write $\mathbf{t}$ directly.

The goal with the optimization is to find the correct parameters for the rotation $\boldsymbol{\omega}$ and for the translation $\mathbf{t}$.

With help from the notation above we can rewrite the error-equation (4.2) as

$$E(\boldsymbol{\xi}) = \sum_{i=0}^{M} \sum_{j=0}^{N} ||\psi(\exp(\hat{\boldsymbol{\omega}})X + \mathbf{t})||^2 \tag{4.4}$$

$$(i, j) \in I_d$$

$$X = \rho(i, j, I_d(i, j)) \in \mathbb{R}^3$$

M is the number of rows

N is the number of columns in the image.

To get easier notation we will write

$$\psi(\boldsymbol{\xi}) \tag{4.5}$$

instead of

$$\psi(\exp(\hat{\boldsymbol{\omega}})X + \mathbf{t}) \text{ or } \psi(\mathbf{R}X + \mathbf{t}). \tag{4.6}$$

To minimize this function we need to find the rotation parameters $\boldsymbol{\omega}$ and the translation parameters $\mathbf{t}$ so that

$$\nabla E(\boldsymbol{\xi}) = 0. \tag{4.7}$$

This is not trivial to do by using our current error equation (4.4). Therefore we linearise the SDF ψ around our current guess for the rotation and translation $\boldsymbol{\xi^k}$. The linearisation of ψ around $\boldsymbol{\xi^k}$ looks like

$$\psi(\boldsymbol{\xi}) \approx \psi(\boldsymbol{\xi}^k) + \nabla\psi(\boldsymbol{\xi}^k)^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k). \tag{4.8}$$

By plugging this into the equation (4.4) we get the following approximation

$$E(\boldsymbol{\xi}) \approx \sum_{i=0}^{M} \sum_{j=0}^{N} ||\psi(\boldsymbol{\xi}^k) + \nabla\psi(\boldsymbol{\xi}^k)^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k)||_2^2. \tag{4.9}$$

Note that $\psi(\boldsymbol{\xi})$ is also dependent on the pixel coordinates (i, j) in (4.9) since each pixel generates a 3D point

$$X = \rho(i, j, I_d(i, j)). \tag{4.10}$$

We now aim to minimize the linearised equation (4.9) instead of the original equation (4.2). By taking the gradient of the linearised equation (4.9) and putting it to 0 we obtain

$$\nabla E(\boldsymbol{\xi}) \approx \nabla \sum_{i=0}^{M} \sum_{j=0}^{N} ||\psi(\boldsymbol{\xi}^k) + \nabla\psi(\boldsymbol{\xi}^k)^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k)||^2 = \tag{4.11}$$

$$\nabla \sum_{i=0}^{M} \sum_{j=0}^{N} (\psi(\boldsymbol{\xi}^k)^2 + 2\psi(\boldsymbol{\xi}^k)\nabla\psi^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k) + (\nabla\psi^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k))^T(\nabla\psi^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k))) = \tag{4.12}$$

$$\sum_{i=0}^{M} \sum_{j=0}^{N} \psi(\boldsymbol{\xi}^k)\nabla\psi + \nabla\psi\nabla\psi^T\boldsymbol{\xi} - \nabla\psi\nabla\psi^T\boldsymbol{\xi}^k = 0. \tag{4.13}$$

We can now define a 6×6 matrix $\mathbf{A}$ and 6×1 vector $\mathbf{b}$

$$\mathbf{A} = \sum_{i=0}^{M} \sum_{j=0}^{N} \nabla\psi(\boldsymbol{\xi}^k)\nabla\psi(\boldsymbol{\xi}^k)^T \tag{4.14}$$

$$\mathbf{b} = \sum_{i=0}^{M} \sum_{j=0}^{N} \psi(\boldsymbol{\xi}^k)\nabla\psi(\boldsymbol{\xi}^k). \tag{4.15}$$

With these definitions we can reformulate (4.13) as

$$\mathbf{b} + \mathbf{A}\boldsymbol{\xi} - \mathbf{A}\boldsymbol{\xi}^k = \mathbf{0} \tag{4.16}$$

$$\iff$$

$$\boldsymbol{\xi} = \mathbf{A}^{-1}(\mathbf{A}\boldsymbol{\xi}^k - \mathbf{b}) \tag{4.17}$$

$$\iff$$

$$\boldsymbol{\xi} = \boldsymbol{\xi}^k - \mathbf{A}^{-1}\mathbf{b}. \tag{4.18}$$

So the vector $\boldsymbol{\xi}$ which solves

$$\nabla E(\boldsymbol{\xi}) = \mathbf{0} \tag{4.19}$$

is

$$\boldsymbol{\xi} = \boldsymbol{\xi}^k - \mathbf{A}^{-1}\mathbf{b}. \tag{4.20}$$

If $\mathbf{A}$ is a positive definite matrix, then this newly found $\boldsymbol{\xi}$ is a global minimum to our linearised error function (4.9). Therefore we put

$$\boldsymbol{\xi}^{k+1} = \boldsymbol{\xi}^{k} - \mathbf{A}^{-1}\mathbf{b} \tag{4.21}$$

as our updated guess of the rotation and translation for the camera.

However, we are seeking for the minimum of the original error function (4.2). Therefore, when we have found $\boldsymbol{\xi}^{k+1}$, we repeat the process and linearise ψ around $\boldsymbol{\xi}^{k+1}$ and solve the the same equation which gives us a new vector $\boldsymbol{\xi}^{k+2}$.

Hence, the approach to find the camera configuration $\boldsymbol{\xi}$ for a new image frame I_d^{n+1} is to start with an initial camera guess $\boldsymbol{\xi}^0$ and then iteratively solve the equation (4.21).

We must start reasonably close to the true minimum to find it. Therefore we need to have a reasonable method of choosing an initial camera guess $\boldsymbol{\xi}^0$. Since depth sensors like the Microsoft Kinect has a frame rate of 30 Hz, we can assume that the distance between two frames will be small. Therefore, the initial camera guess is the found camera from the previous frame I_d^n. By denoting the rotation and translation for the last known camera by $\boldsymbol{\xi}_n$ and the rotation and translation for the camera we want to find by $\boldsymbol{\xi}_{n+1}$ the optimisation procedure thus looks as in Algorithm 12.

Algorithm 5: OPTIMISING IN L_2

1 $k = 0$
  `// We start by initialising with the rotation and translation from the previous camera.`
2 **Initialize:** $\boldsymbol{\xi}_{n+1}^{k} \leftarrow \boldsymbol{\xi}_n$
3 **while** *stop* = *false* **do**
4   $\boldsymbol{\xi}_{prev} \leftarrow \boldsymbol{\xi}_{n+1}^{k}$
    `// Compute the matrices A and b`
5   $\mathbf{A} = \sum_{i=0}^{M}\sum_{j=0}^{N} \nabla\psi(\boldsymbol{\xi}_{n+1}^{k})\nabla^T\psi(\boldsymbol{\xi}_{n+1}^{k})$
6   $\mathbf{b} = \sum_{i=0}^{M}\sum_{j=0}^{N} \psi(\boldsymbol{\xi}_{n+1}^{k})\nabla\psi(\boldsymbol{\xi}_{n+1}^{k})$
    `// Update` $\boldsymbol{\xi}$
7   $\boldsymbol{\xi}_{n+1}^{k+1} = \boldsymbol{\xi}_{n+1}^{k} - \mathbf{A}^{-1}\mathbf{b}$
8   $count++$
    `// Check if the stop criteria is fulfilled`
9   **if** $(|\boldsymbol{\xi}_{n+1}^{k+1} - \boldsymbol{\xi}^{prev}| < \epsilon$ *or* $count == MAX)$ **then**
10   stop = true
11    $\boldsymbol{\xi}_{opt} = \boldsymbol{\xi}_{n+1}^{k+1}$
12 **return** $\boldsymbol{\xi}_{opt}$

The algorithm terminates either when the updated $\boldsymbol{\xi}$ change less than the threshold ϵ or the number of iterations reaches its maximum. Therefore, we know that the algorithm will always terminate. The procedure is also illustrated in Figure 4.3.

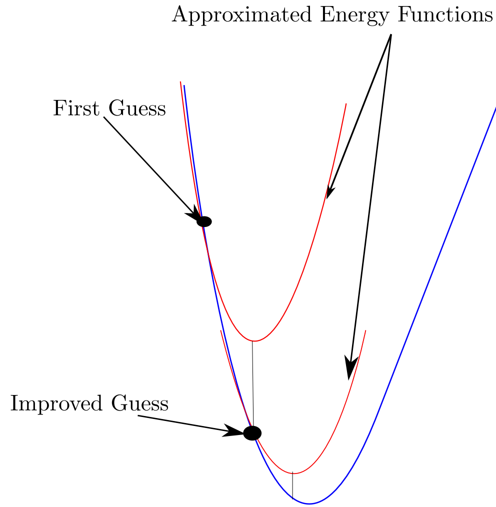


Figure 4.3: By optimising the approximated error-function we iteratively find a better solution to the original error-function.

It also important to notice that each matrix $\mathbf{A}_{ij}$ we get from the pixels (i, j) will have determinant

$$\det(\mathbf{A}_{ij}) = 0. \tag{4.22}$$

However,

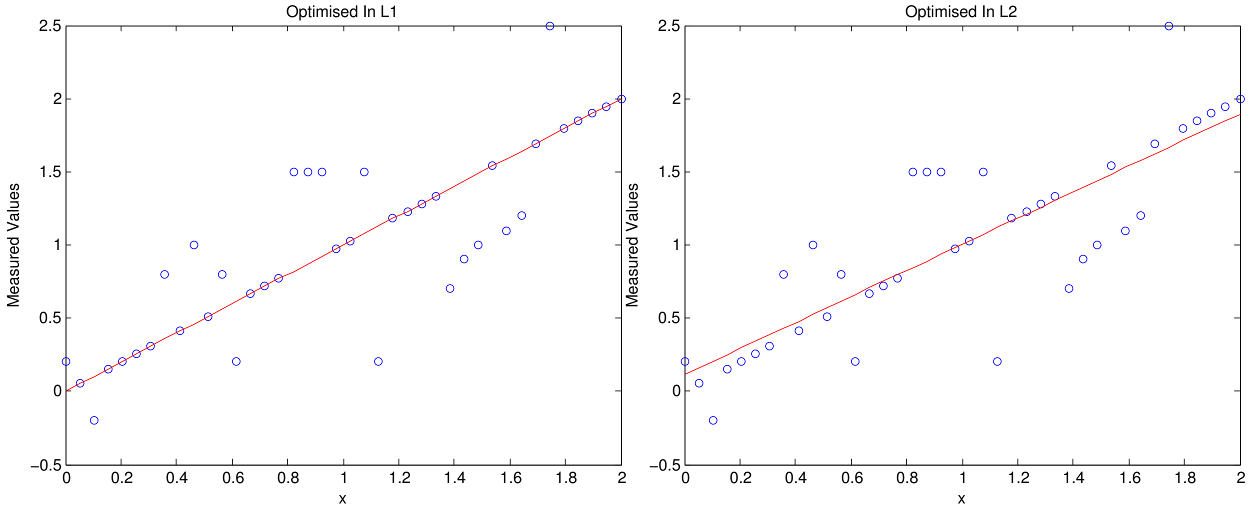
$$\det(\sum_{i=0}^{M}\sum_{j=0}^{N}\mathbf{A}_{ij}) \neq \sum_{i=0}^{M}\sum_{j=0}^{N}\det(\mathbf{A}_{ij}) \tag{4.23}$$

if not all matrices $\mathbf{A}_{ij}$ are linearly dependent. The image is of size 640×480 which is potentially 307 200 3D-points. The probability of getting a matrix $\mathbf{A}$ with determinant 0 is very low.

It will happen if the surface is a plane, then it is not possible to determine a unique rigid-body motion with this tracking approach. That one can see by realizing that if the whole surface lies in a plane, a wall for instance, then one can move the plane in a way so that it still lies in the wall. Then the error would still be zero (ideally), the minimum will not be unique and then the whole approach fails.

4.2.2 L_1-norm

We have now seen how to optimize the error function (4.2) in the L_2-norm using Taylor expansion and the Gaussian-Newton method. The advantage with this approach is that it is converging very fast, thus a good choice for real-time optimization. However, their are certain drawbacks, one is that if data is very erroneous, then the noise is squared and can



(a) The fitted line using the L_1-norm. (b) The fitted line using the L_2-norm.

have a potentially high impact. This is seen in a artificial experiment where a line is fitted to noisy data. In the L_2-norm one clearly sees how the error is affected by noise. On the other hand, when optimising in the L_1-norm, the line is much better fitted to the data.

In L_2, the sum of squared errors $||y - ax + b||_2^2$ is minimised using *polyfit* in Matlab. In the L_1-norm the sum of absolute errors $|y - ax + b|$, is minimised by using the $L_1 - Magic$ library [1].

In this part we introduce an error function using the L_1 norm instead. The reason is to investigate whether the tracking works better or not with this norm instead of the one introduced in the previous section.

We start with defining the error function 4.2

$$
\begin{aligned}
E(\boldsymbol{\xi}) &= \sum_{i=0}^{M} \sum_{j=0}^{N} |\psi(\boldsymbol{\xi})| \qquad (4.24) \\
\psi(\boldsymbol{\xi}) &= \psi(\mathbf{R}X + t) \\
X_{ij} &= \rho(i, j, I_d(i, j)) \\
&\text{M is the number of rows in the image} \\
&\text{N is the number of columns in the image.}
\end{aligned}
$$

The idea is the same, the rotation and translation $\boldsymbol{\xi}$ of the camera shall ideally reconstruct all the points for the frame I_d on the surface, giving an error of 0.

How the error function might look like is illustrated in Figure 4.4.

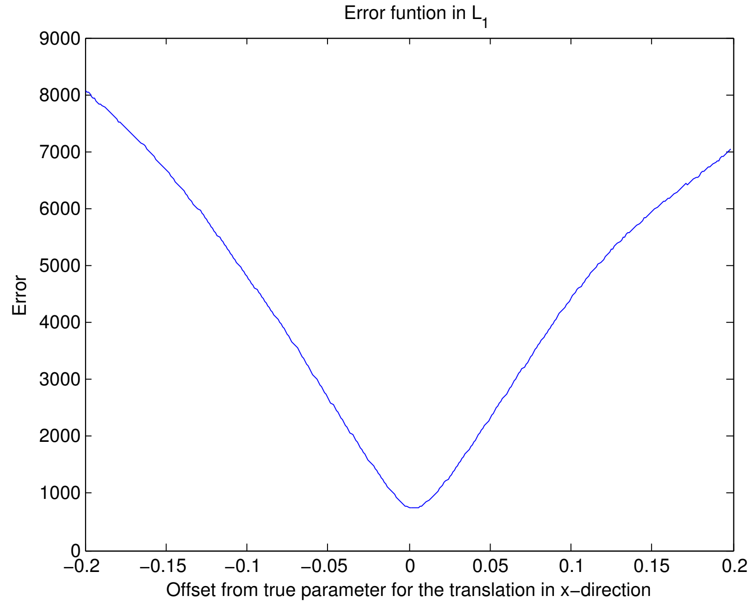


Figure 4.4: The error function in the L_1-norm where the error corresponds to the offset from the true parameter.

We can with help from (4.24) formulate the optimization problem as

$$\min_{\boldsymbol{\xi}} \sum_{i=0}^{M} \sum_{j=0}^{N} |\psi(\boldsymbol{\xi})|. \tag{4.25}$$

The problem now is to find a way of optimizing (4.24). One of the problems is that $|\psi(\boldsymbol{\xi})|$ will not be differentiable everywhere. A common approach is described by Pock and Chambolle in [8] where they solve error functions in the L_1 norm by using the so called *primal-dual algorithm*. In [8] they are solving variational problems, which means the goal is to find a functional which minimizes the error function. In this work we are searching for parameters which minimizes a function.

To get an optimization problem we can solve, we start by realising that

$$|\psi(\boldsymbol{\xi})| = \max_{|p|\leq 1}(p\psi(\boldsymbol{\xi})). \tag{4.26}$$

Substituting this into the equation (4.24) yields

$$E(\boldsymbol{\xi}) = \sum_{i=0}^{M} \sum_{j=0}^{N} \max_{|p_{ij}|\leq 1}(p_{ij}\psi(\boldsymbol{\xi})). \tag{4.27}$$

Then we see that

$$\max_{|p_{ij}|\leq 1} \sum_{i=0}^{M} \sum_{j=0}^{N} (p_{ij}\psi(\boldsymbol{\xi})) \leq \sum_{i=0}^{M} \sum_{j=0}^{N} \max_{|p_{ij}|\leq 1}(p_{ij}\psi(\boldsymbol{\xi})). \tag{4.28}$$

By observing that the solution to

$$\max_{|p|\leq 1}(p\psi(\boldsymbol{\xi})) \tag{4.29}$$

is

$$p = \frac{\psi(\boldsymbol{\xi})}{|\psi(\boldsymbol{\xi})|} \; \textit{if} \; \psi(\boldsymbol{\xi}) \neq 0 \tag{4.30}$$

we get

$$\max_{|p_{ij}|\leq 1} \sum_{i=0}^{M}\sum_{j=0}^{N} p_{ij}\psi(\boldsymbol{\xi}) = \tag{4.31}$$

$$\sum_{i=0}^{M}\sum_{j=0}^{N} \frac{\psi(\boldsymbol{\xi})}{|\psi(\boldsymbol{\xi})|}\psi(\boldsymbol{\xi}) = \tag{4.32}$$

$$\sum_{i=0}^{M}\sum_{j=0}^{N} \max_{|p_{ij}|\leq 1}(p_{ij}\psi(\boldsymbol{\xi})), \tag{4.33}$$

$$\text{if } \psi(\boldsymbol{\xi}) \neq 0.$$

Since we have equality in equation (4.28), we can formulate our original problem (4.25) as a primal-dual problem

$$\min_{\boldsymbol{\xi}} \; \max_{|p_{ij}|\leq 1} \sum_{i=0}^{M}\sum_{j=0}^{N} p_{ij}\psi(\boldsymbol{\xi}). \tag{4.34}$$

A more general and similar optimisation problem is presented in [8] and [17]

$$\min_{x\in C}\max_{y\in K}\langle Ax, y\rangle + \langle g, x\rangle - \langle h, y\rangle \tag{4.35}$$

where X and Y are finite dimensional vector spaces, $C \subset X$ and $K \subset Y$. A is a linear continuous operator and $\langle g, x\rangle$ and $\langle h, y\rangle$ are point wise linear terms.

This formulation is widely used in the computer vision community when it comes to variational problems such as denoising and image segmentation etc.

The difference between a variational optimization problem and our is that in a variational problem the elements in X and Y are functionals, in our optimization problem $X = \mathbb{R}^6$ and $Y = \mathbb{R}$. A method for solving these kinds of problems is presented in [17] and the algorithm looks like in Algorithm 6.

Where Π_K and Π_C projects the variables y and x back to K and C which are convex and closed subsets which the parameters x and y shall lie in. In our case $\boldsymbol{\xi} \in \mathbb{R}^6$ and $p_{ij} \in \{x \in \mathbb{R} : |x| \leq 1\}$. Another comment about the algorithm above is that one is alternating between gradient ascent and steepest descent to find the saddle-point. In the first row, one

Algorithm 6: OPTIMISATION ALGORITHM FOR PRIMAL-DUAL

1 Initialization: Choose $\tau, \sigma > 0$, $\theta \in [0,1]$
2 Choose $(x^0, y^0) \in C \times K$
3 Iterations ($n \geq 0$): Update $x^n, y^n, \tilde{x}^n$ as follows:

$$\begin{cases} y^{n+1} = & \Pi_K(y^n + \sigma(A\tilde{x}^n - h)) \\ x^{n+1} = & \Pi_C(x^n - \tau(A^* y^{n+1} + g)) \\ \tilde{x}^{n+1} = & 2x^{n+1} - x^n \end{cases}$$

takes a step in the direction of steepest ascent and in the second one takes a step in the direction of steepest descent. That is

$$\nabla_y(\langle A\tilde{x}, y\rangle + \langle g, \tilde{x}\rangle - \langle h, y\rangle) = A\tilde{x} - h \tag{4.36}$$

$$-\nabla_x(\langle Ax, y\rangle + \langle g, x\rangle - \langle h, y\rangle) = -(A^* y + g). \tag{4.37}$$

The third line in Algorithm 6 is an extrapolation step which guarantees convergence.

Our optimization looks as follows

$$\min_{\boldsymbol{\xi}} \max_{\substack{p_{ij} \\ |p_{ij}| \leq 1}} \sum_{i=0}^{M} \sum_{j=0}^{N} \psi(\boldsymbol{\xi}) p_{ij}. \tag{4.38}$$

We see that A is the identity operator I and h and g are 0. However, it is not straightforward to implement the optimization procedure because our term $\psi(\boldsymbol{\xi})p_{ij}$ is not linear in $\boldsymbol{\xi}$. Therefore, we again linearise around our current guess $\boldsymbol{\xi}^k$

$$\psi(\boldsymbol{\xi})p_{ij} \approx (\psi(\boldsymbol{\xi}^k) + \nabla_{\boldsymbol{\xi}^k}\psi^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k))p_{ij}. \tag{4.39}$$

Now it is linear in $\boldsymbol{\xi}$ and p_{ij} and the gradients are

$$\nabla_{p_{ij}}((\psi(\tilde{\boldsymbol{\xi}}^k) + \nabla_{\tilde{\boldsymbol{\xi}}^k}\psi^T(\tilde{\boldsymbol{\xi}} - \tilde{\boldsymbol{\xi}}^k))p_{ij}) = \psi(\tilde{\boldsymbol{\xi}}^k) + \nabla_{\tilde{\boldsymbol{\xi}}^k}\psi^T(\tilde{\boldsymbol{\xi}} - \tilde{\boldsymbol{\xi}}^k) \tag{4.40}$$

$$\nabla_{\boldsymbol{\xi}}((\psi(\boldsymbol{\xi}^k) + \nabla_{\boldsymbol{\xi}^k}\psi^T(\boldsymbol{\xi} - \boldsymbol{\xi}^k))p_{ij}) = \nabla_{\boldsymbol{\xi}^k}\psi p_{ij}. \tag{4.41}$$

By using this linearisation and these gradients we can adjust the proposed method for our optimization problem and our alorithm is illustrated in Algorithm 7.

Since we do not have any constraints on the set which $\boldsymbol{\xi}$ shall belong to, no reprojection is needed. For the dual variables p_{ij}, we project them back to $K = \{x \in \mathbb{R} : |x| \leq 1\}$ as shown in Algorithm 8.

As initialization for $\boldsymbol{\xi}$, we again start from the previous known camera position. For the dual $\mathbf{p}$, it is initialized as a zero-matrix of dimension $M \times N$, which is the same size as the image.

Algorithm 7: OUR ALGORITHM FOR PRIMAL-DUAL

1 Initialization: Choose $\tau, \sigma > 0$

```
// p is a matrix of the same size as the image
```

2 Choose $(\boldsymbol{\xi}^0_{n+1}, \mathbf{p}^0) \in \mathbb{R}^6 \times \mathbb{R}^{M \times N}$

3 Iterations $(k > 0)$:

4 Update $\boldsymbol{\xi}^k_{n+1}, p^k_{ij}, \tilde{\boldsymbol{\xi}}^k_{n+1}$ as follows:

$$\begin{cases} p^{k+1}_{ij} = & \Pi_K(p^k_{ij} + \sigma((\psi(\tilde{\boldsymbol{\xi}}^k) + \nabla_{\tilde{\boldsymbol{\xi}}^k}\psi^T(\tilde{\boldsymbol{\xi}}^k - \tilde{\boldsymbol{\xi}}^k))) = \\ & \Pi_K(p^k_{ij} + \sigma(\psi(\tilde{\boldsymbol{\xi}}^k))) \\ \boldsymbol{\xi}^{k+1}_{n+1} = & \boldsymbol{\xi}^k_{n+1} - \tau(\sum_{i=0}^M \sum_{j=0}^N p^{k+1}_{ij} \nabla\psi(\boldsymbol{\xi}^k_{n+1})) \\ \tilde{\boldsymbol{\xi}}^{k+1} = & 2\boldsymbol{\xi}^{k+1}_{n+1} - \boldsymbol{\xi}^k_{n+1} \end{cases}$$

Algorithm 8: PROJECTION FUNCTION

1 Π_K :

2 **if** $p_{ij} < -1$ **then**

3 $\quad p_{ij} = -1$

4 **if** $p_{ij} > 1$ **then**

5 $\quad p_{ij} = 1$

This is then updated iteratively until we reach the stopping criterion. For the stopping criteria we define the gap δ as the difference between the error for the original error function and the one we are minimising

$$\delta = |\sum_{i=0}^{M}\sum_{j=0}^{N}|\psi(\boldsymbol{\xi})| - \sum_{i=0}^{M}\sum_{j=0}^{N} p_{i,j}\psi(\boldsymbol{\xi})|. \tag{4.42}$$

We save the old δ_{old} and compare it with updated gap δ. If the difference

$$|\delta - \delta_{old}| < \epsilon \tag{4.43}$$

then the algorithm terminates, or when the number of iterations reaches its maximum we also stop.

The reason we choose this stopping criteria is that the difference

$$|\sum_{i=0}^{M}\sum_{j=0}^{N}|\psi(\boldsymbol{\xi})| - \sum_{i=0}^{M}\sum_{j=0}^{N} p_{i,j}\psi(\boldsymbol{\xi})| \tag{4.44}$$

will converge to 0 since we are maximising with respect to $\mathbf{p}$. How the final optimisation algorithm for L_1 looks like is shown in Algorithm 9.

Algorithm 9: OPTIMISE IN L_1

```
1  count ← 0
   // Initialise with the last known camera configuration
2  ξ^0 ← ξ_n
3  ξ̃^0 ← ξ^0
   // p is initialised as a zero matrix of the same size as the image
4  p^0 ← 0
5  while stop is true do
6      δ_prev = δ
7      p_ij^{k+1} = Π_K(p_ij^k + σ(ψ(ξ̃^k)))  // ψ(ξ̃^k) is also dependent on (i, j)
8      ξ^{k+1} = ξ^k − τ(Σ_{i=0}^M Σ_{j=0}^N p_ij^{k+1} ∇ψ(ξ^k))
9      ξ̃^{k+1} = 2ξ^{k+1} − ξ^k
       // Compute the gap
10     δ = |Σ_{i=0}^M Σ_{j=0}^N |ψ(ξ^{k+1})| − Σ_{i=0}^M Σ_{j=0}^N p_{i,j} ψ(ξ^{k+1})|
11     if (|δ − δ_prev| ≤ ϵ || count == MAX) then
12         stop = false
13         return ξ^{k+1}
```

4.2.3 Truncated L_2-norm

Since we will get noise from the images it might be interesting to see whether we can improve the tracking by truncation of the L_2 norm. The idea is quite simple, assume we have an error function

$$E(\mathbf{x}) = \sum_{i=0}^{N} ||r(\mathbf{x})||_2^2 \tag{4.45}$$

we want to minimize in the L_2-norm where

$$r : \mathbb{R}^N \to \mathbb{R}. \tag{4.46}$$

By looking at Figure 4.4b, we can see that for some measures the error is big and will have a big impact on the solution. In the truncated L_2, we simply reject those errors which are above a certain threshold δ. Thus, (4.45) would look like

$$E(\mathbf{x}) = \sum_{i=0}^{N} ||r(\mathbf{x})||_2^2 \tag{4.47}$$
$$\text{subject to: } ||r(\mathbf{x})||_2 < \delta.$$

The idea is that errors shall not affect the solution. In our case, we consider only points which are reconstructed within a threshold δ from the surface and the others will be rejected. Except from that, the optimisation procedure is the same as for L_2.

4.3 Summary

We have in this chapter presented 3 different methods to track the camera movement directly using the 3D model, namely optimising in the L_2-norm, L_1-norm and the truncated L_2-norm. All these methods works iteratively, we linearise the SDF ψ around the current camera guess $\boldsymbol{\xi}^k$ and optimise the linearised error function and then finds a better guess $\boldsymbol{\xi}^{k+1}$ which we use in the next iteration.

Chapter 5

Experimental Results

In this part we will evaluate the tracking against different benchmarks from [20]. All different norms will be evaluated together with different error metrics. The purpose is to see which of the norms and metrics gives the best results and to find out where the tracking has problems. A comparison between KinFu, [2], and RGB-D SLAM, [11], will also be done. To calculate the absolute trajectory error between the ground truth and our estimated trajectory we use a script provided by [20]. It will also be demonstrated how our method works using live data. For the point-to-point metric we use the simplified version where only the difference along the principal axis is calculated.

5.1 Qualitative Results

Here we start with showing some examples from live data to demonstrate that the procedure really works. We will show results both with and without colours, also we will demonstrate that both the point-to-point metric and the point-to-plane metric works. We will here only consider L_2, since the $primal - dual$ algorithm is implemented on the CPU and not fast enough for real-time reconstruction. The L_2 optimisation is on the other hand implemented on a GPU and works therefore much faster. The evaluation will be made on a computer with the graphic card GeForce GTX 560 Ti, which has 340 cores and 1 GB of memory.

It has also been found empirically that the tracking works slightly better when one calculates the matrices $\mathbf{A}_{ij}$ and $\mathbf{b}_{ij}$ only for those points $\rho(i, j, I_d(i, j))$ which are reconstructed in voxels which have a weight larger than zero. For all results here, the matrices $\mathbf{A}_{ij}$ and $\mathbf{b}_{ij}$ are only calculated for those points which are reconstructed in voxels with weight larger than zero.

With the point-to-point metric and without any colourising and a resolution of 256^3 vertices in the voxel grid, the algorithm performs on average a frame-rate of 60 Hz. As a comparison, the open-source Kinfu-implementation performs on about 15 Hz on the same computer.

Figure 5.1: A 3D reconstruction made by point-to-point metric and 256^3 vertices.

An example of a how a 3D reconstruction might look like from our system is shown in Figure 5.1. By including colours the result looks much better. The speed is slowed down a bit but performs still above 40 Hz. The result of the same scene is demonstrated in Figure 5.2. To colourise the surface in Figure 5.2 we used the alternative method presented in 3.4, where the last image has a higher weight.

An example which shows the accuracy of the tracking is shown in Figure 5.3, where a reconstruction of the screen is made. On the screen the colour image which shows what the camera sees is shown. In that colour image one can actually see the screen again and even on that screen one can see the structures of what is on the screen. Also here we used the alternative approach where the last colour image has a higher weight than the others to estimate the colours, otherwise the image on the screen would be smeared out.

Especially, if we look at the trajectory in Figure 5.4, we can see that the camera is moving quite a lot and it is not always facing the screen. Nonetheless, the colourising gives very good results, which shows the accuracy of the tracking. Especially the borders on the terminals are sharp, even the terminals on the screen in the colour image.

Also by using point-to-plane and colourise using a running average one does also obtain nice 3D reconstructions. This is well illustrated in Figure 5.6. If one does not filter the image to calculate the normals, also the point-to-plane metric works well above 40 Hz.

For larger volumes it does also works good, in Figure 5.5 the point-to-point metric is used.

Figure 5.2: A colourised 3D-model.

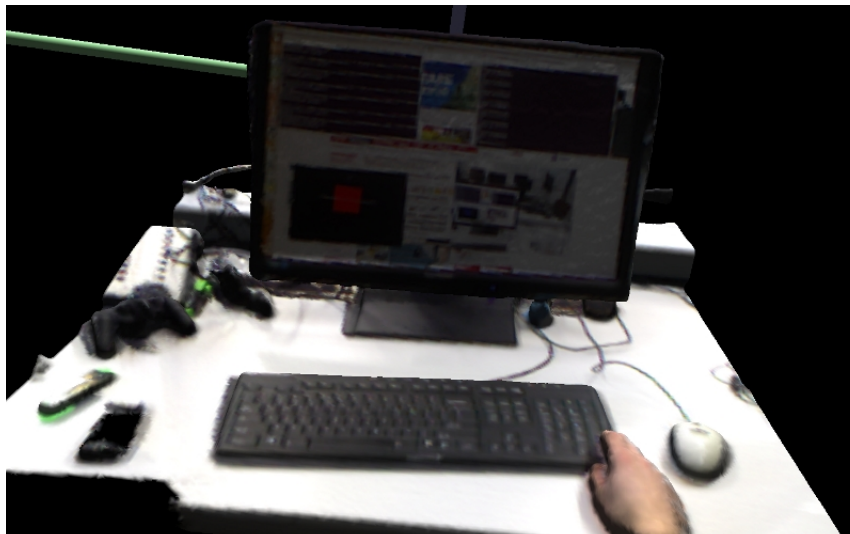


Figure 5.3: A coloured reconstruction of the screen, where one can see the screen again in the colour image in the right corner. Notice that the borders of the terminals on the screen are sharp. Even for the terminals on the image of the screen in the lower right corner.

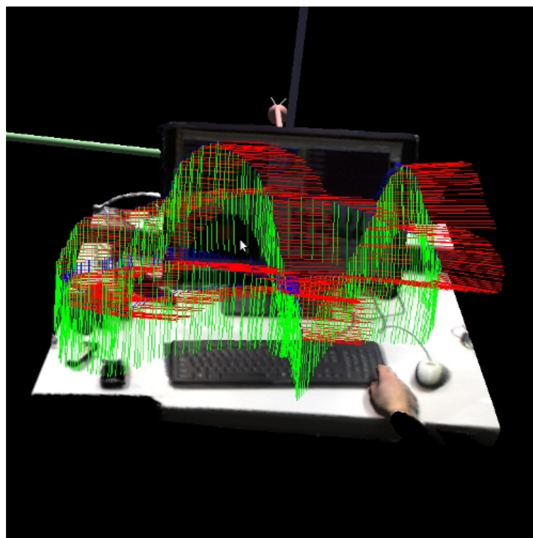


Figure 5.4: The estimated trajectory of the camera, in the left part, the camera does not see the screen.

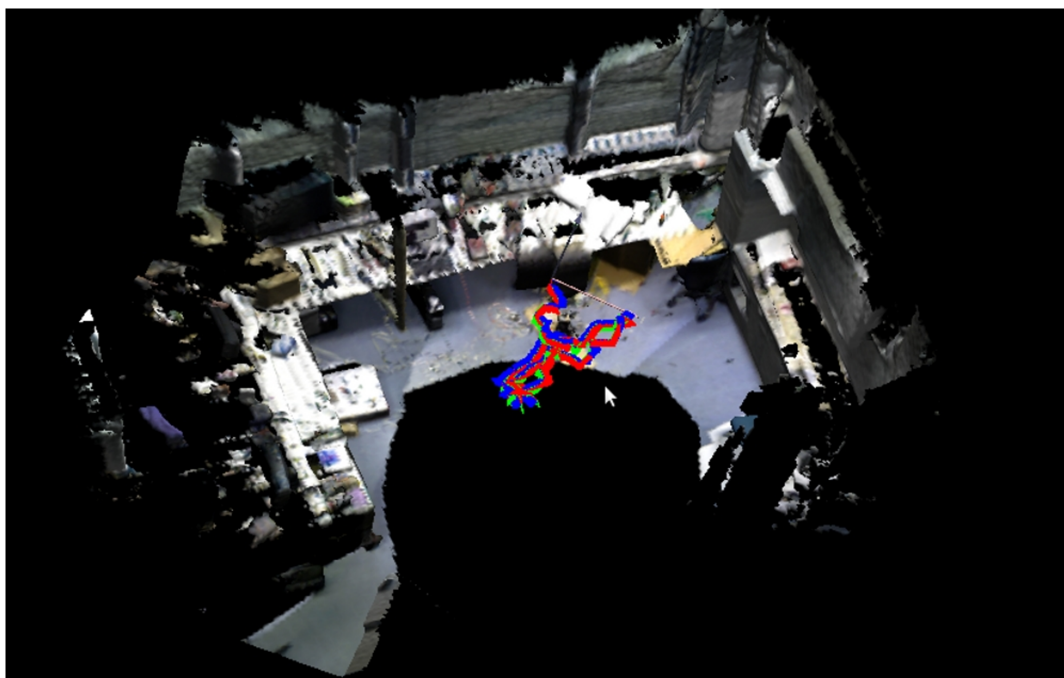


Figure 5.5: A large scale reconstruction of a lab.

Figure 5.6: Using point-to-plane and a running average of the colours, this 3D reconstruction is obtained. If one looks at the screen one can see another 3D reconstruction of the author.

5.2 L_2-Norm

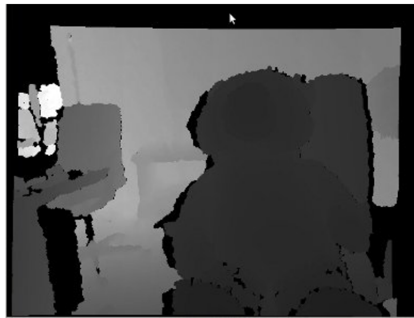
We have in the previous part seen that our approach works fine. Both with point-to-plane metric and point-to-point metric, also the colourising works good.

In this part we evaluate how the tracking works for the L_2-norm. The tracking will be tested both by using the point-to-point metric and the point-to-plane metric. Furthermore, we will test the tracking with different resolutions on the voxel grid. To get consistency in the tests, the variables for the truncation δ and the distance ϵ we go behind the surface before the uncertainty decreases are kept constant. The image will also be filtered using our implementation with a mask size of 5×5 and a threshold of 5 cm

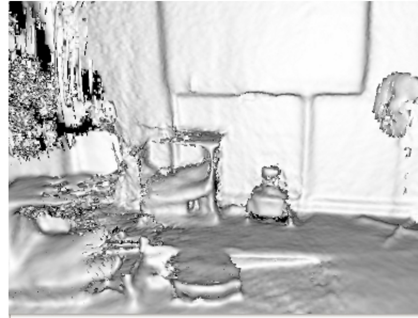
$$\delta = 0.3 \text{ m}$$
$$\epsilon = 0.025 \text{ m}.$$

To compare our method with KinectFusion, we use the PCL-implementation called KinFu available from [2] and the evaluation tool which comes with it. We will use the benchmarks provided by Sturm *et al* [20] together with the evaluation tool for calculating the absolute trajectory error between the estimated trajectory and the ground truth.

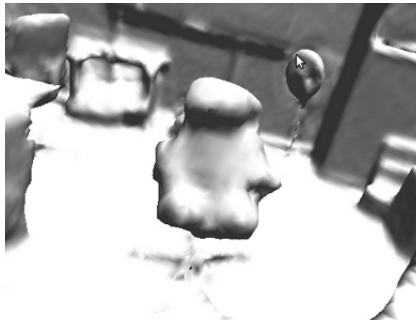
To get a fair comparison the size of the volume is the same for both our method and for KinFu for each dataset. We do change the size between the datasets since the datasets covers surfaces of different sizes. But for each dataset, the size of the volume is the same. The same holds for the starting position which is adjusted between the datasets, but kept constant for each dataset so that both KinFu and our method starts from the same position.



(a) The depth image from the Teddy bear set.



(b) The resulting 3D-reconstruction from KinFu, the bear is gone.



(c) The teddy bear reconstructed using our method.

Figure 5.7: Here one can see the difference between our method and Kinfu. In 2.3a one can see what the camera sees and in 5.7b the resulting reconstruction. In 5.7c one can see the reconstruction using our method. Our reconstruction is after 1376 images and the KinFu after about 1200 images.

We will use quite challenging datasets to really test the algorithms so we can find out where the problems are and to see what strengths and weaknesses the different approaches have.

By looking at how the scene looks like for the KinFu algorithm in Figure 5.7b and how it looks for our in Figure 5.7c, one can clearly see the difference.

Here we show the results in Table 5.1 and Table 5.2 for the evaluation of our method and KinFu for different datasets. Only the Root Mean Square Error for the estimated trajectory is presented, the interested reader can find more detailed results in the appendix where we also compare our method to RGB-D SLAM.

From the evaluation, it is clear that our method outperforms KinFu in practically every aspect.

On the harder datasets Teddy, Room and Desk2, our method manage to estimate the trajectory with a much lower error than KinFu. Interesting is that we outperform KinFu without using standard methods like coarse-to-fine manner and we do not have a real bilateral filter.

One can also observe that the results are typically better for a resolution of 256^3 vertices

Method	Res.	Teddy	F1 Desk	F1 Desk2	Room
KinFu	256	0.155356 m	0.058550 m	0.425655 m	Failed
KinFu	512	0.314937 m	0.067729 m	1.069398 m	Failed
Point-To-Plane	256	**0.076112 m**	0.094144 m	0.226816 m	0.266557 m
Point-To-Plane	512	0.142598 m	0.143094 m	0.257331 m	0.105750 m
Point-To-Point	256	0.089038 m	**0.036481 m**	**0.058547 m**	0.186188 m
Point-To-Point	512	0.124477 m	0.037537 m	0.065838 m	**0.071330 m**

Table 5.1: The root-mean square absolute trajectory error for KinFu and our method for different resolutions, metrics and datasets.

Method	Res.	F2 Desk With Person	F2 Long Office Household
KinFu	256	0.060193 m	0.061833 m
KinFu	512	**0.056934 m**	0.060245 m
Point-To-Plane	256	0.079959 m	0.054150 m
Point-To-Plane	512	0.088280 m	0.055974 m
Point-To-Point	256	0.069734 m	**0.036943 m**
Point-To-Point	512	0.073863 m	0.039027 m

Table 5.2: The root-mean square absolute trajectory error for KinFu and our method for different resolutions, metrics and datasets.

than for 512^3 vertices. Since a higher resolution yields a more accurate signed distance function, one could expect that the tracking would work better. A possible explanation to why a lower resolution works better is that it is less sensitive to local minima. When the resolution increases one can also expect that there are more finer details which can result in local minima for the error function.

The results also shows that the point-to-point metric is more stable and in general better than the point-to-plane metric.

5.3 L_1-Norm

When the L_1-norm was evaluated it turned out to be much harder to find the trajectory than for L_2. It is necessary to assign τ a value in the size of 10^{-6}, otherwise the primal-dual algorithm will not converge. Since τ is that small, $\boldsymbol{\xi}^k$ will not change much in each iteration, meaning that many iterations are needed to find the next camera.

Therefore it takes much more time to optimise in the L_1-norm. Another problem is that the tracking seems to work fine in the beginning but eventually, it diverges as can be seen in Figure 5.8 where the primal-dual algorithm was tested against the F1 Desk dataset.

The same problem occurs when it is tested against the dataset Long Office Household as

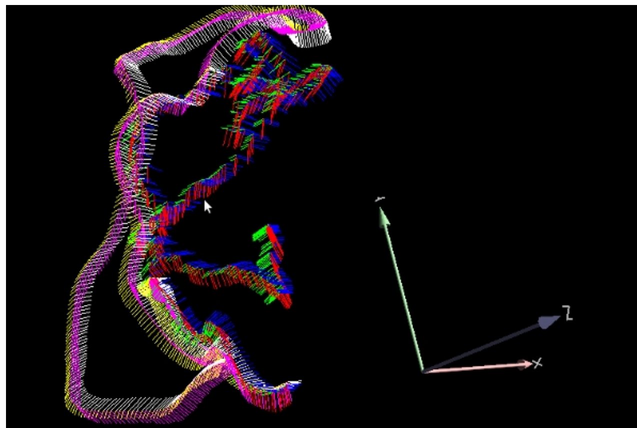


Figure 5.8: The tracking using the L_1 drifts away.

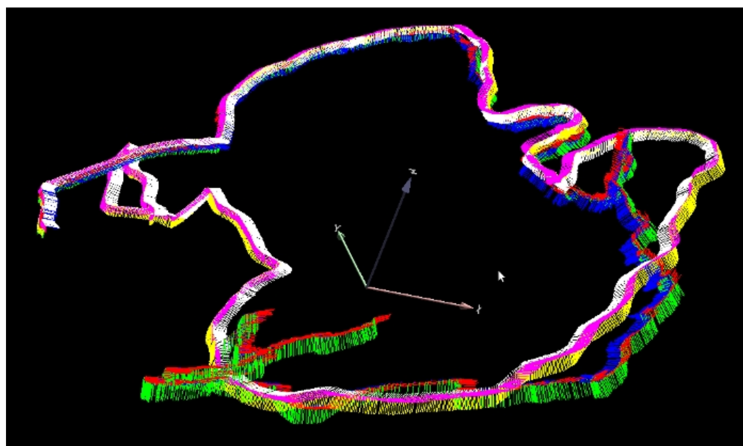


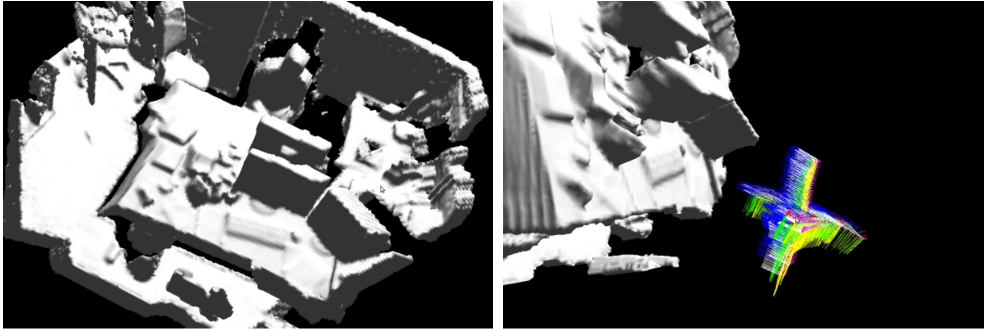
Figure 5.9: The estimated trajectory is shown in the red, blue and green marks and the ground truth is the white, yellow and pink, it is clear that the tracking diverges but works ok in the beginning.

seen in Figure 5.9. Testing the L_1-norm against the benchmark F1 XYZ, the trajectory was successfully estimated as can be seen in Figure 5.10a and Figure 5.10b.

The results for the L_1-norm was not as good as expected. The fact that one had to take very small steps in the direction of the gradient made the runtime very long. L_1 also seems to drift away easier than L_2.

5.4 Truncated L_2

To investigate if the results could be improved by truncation of the L_2-norm, we tested the truncated L_2 against some different datasets, both for point-to-point metric and point-to-plane metric and the results are in Table 5.3 and Table 5.4.



(a) The dataset successfully reconstructed by the primal-dual algorithm.

(b) The estimated trajectory in blue follows the trajectory in white.

Dataset	Res.	Trunc.	RMSE	Std Dev	Max Error	Min Error
Teddy	512	0.1 m	1.01298 m	0.41117 m	1.80178 m	0.28277 m
Teddy	512	0.2 m	0.14382 m	0.07860 m	0.44254 m	0.01318 m
Teddy	512	None	**0.12448 m**	**0.05913 m**	**0.34131 m**	**0.00678 m**
F1 Desk 2	512	0.1 m	0.18981 m	0.14859 m	0.84877 m	0.00784 m
F1 Desk 2	512	0.2 m	0.10414 m	0.04533 m	0.23536 m	0.01334 m
F1 Desk 2	512	None	**0.06584 m**	**0.03430 m**	**0.20156 m**	**0.00241 m**
Room	512	0.1 m	0.26996 m	0.12080 m	0.50018 m	0.05997 m
Room	512	0.2 m	0.17158 m	0.07662 m	0.31386 m	**0.00643 m**
Room	512	None	**0.07133 m**	**0.02769 m**	**0.18187 m**	0.00873 m

Table 5.3: This table shows the result for truncated L_2 using point-to-point metric.

Dataset	Res.	Trunc.	RMSE	Std Dev	Max Error	Min Error
Teddy	512	0.1 m	0.44314 m	0.22955 m	1.45429 m	0.07075 m
Teddy	512	0.2 m	**0.12755 m**	**0.08003 m**	**0.42613 m**	**0.01120 m**
Teddy	512	None	0.14260 m	0.08811 m	0.52492 m	0.01221 m
F1 Desk 2	512	0.1 m	0.37171 m	0.15267 m	0.86153 m	0.04778 m
F1 Desk 2	512	0.2 m	0.47499 m	0.23962 m	0.97972 m	0.05007 m
F1 Desk 2	512	None	**0.25733 m**	**0.11550 m**	**0.52659 m**	**0.03097 m**
Room	512	0.1 m	0.28102 m	0.13475 m	0.49279 m	0.04168 m
Room	512	0.2 m	**0.10246 m**	0.03965 m	0.21474 m	0.00843 m
Room	512	None	0.10575 m	**0.03850 m**	**0.20966 m**	**0.00708 m**

Table 5.4: This table shows the result for truncated L_2 using point-to-plane metric.

The results for the truncated L_2 indicates that the truncations do not give better results. The trend is clearly that the closer the truncation gets to the original threshold of 0.3 m, the better the result gets. For the point-to-plane metric one can see a small improvement

for a truncation of 0.2 m for some of the datasets, but in general the results shows that a truncated norm does not improve the results. An explanation might be that when the tracking drifts away a bit, the 3D points for the next frame will be reconstructed further away from the surface. If we then truncate, there might be to few 3D points inside the truncated zone to find the correct configuration, leading to that also that camera gets a bad configuration, eventually leading to a failure of the tracking.

Chapter 6

Conclusion And Future Work

We have in this work showed that our method gives clearly better results than the KinectFusion approach and it is also more robust, especially for reconstruction of larger scenes. We perform on about the same level as RGB-D SLAM when it comes to tracking. Furthermore we have shown a simple method to involve colours on the surface which gives good results. This is without doing coarse-to-fine manner or using real bilateral filters. Therefore future work would be to implement both coarse-to-fine manner and to use a real bilateral filter.

Since we have introduced colours, another interesting direction would be to include colours in the tracking to make it more stable against planar structures and more accurate since that would include more information.

It would also be interesting to represent the SDF using octrees instead of a voxel grid, theoretically it would be possible to represent free space with a lower resolution and the space close to the surface with a higher resolution. This will be more memory efficient than using a voxel grid. That would make it possible to do a more accurate reconstruction of larger environments.

Bibliography

[1] l_1-magic library. http://users.ece.gatech.edu/~justin/l1magic/. Accessed: 07/11/2012.

[2] Open source kinfu implementation. http://svn.pointclouds.org/pcl/trunk/. Accessed: 07/11/2012.

[3] J. A. Bærentzen. On the implementation of fast marching methods for 3D lattices, 2001.

[4] Paul J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239–256, February 1992.

[5] Gérard Blais and Martin D. Levine. Registering multiview range data to create 3d computer objects. *IEEE Transactions on Pattern Analysis And Machine Intelligence*, 17:820–824, 1993.

[6] José-Luis Blanco. A tutorial on se(3) transformation parameterizations and on-manifold optimization. Technical report, University of Malaga, September 2010.

[7] Paul Bourke. Polygonising a scalar field. http://paulbourke.net/geometry/polygonise/, 1994. Accessed: 07/11/2012.

[8] Antonin Chambolle and Thomas Pock. A first-order primal-dual algorithm for convex problems with applications to imaging, 2010.

[9] Yang Chen and Gérard Medioni. Object modelling by registration of multiple range images. *Image Vision Comput.*, 10(3):145–155, April 1992.

[10] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 303–312, New York, NY, USA, 1996. ACM.

[11] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard. An evaluation of the RGB-D SLAM system. In *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, St. Paul, MA, USA, May 2012.

[12] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 559–568, New York, NY, USA, 2011. ACM.

[13] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454, 2012.

[14] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[15] Y. Ma, S. Soatto, J. Kosecka, and S. Sastry. *An Invitation to 3D Vision: From Images to Geometric Models.* Springer Verlag, 2003.

[16] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew W. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *ISMAR*, pages 127–136, 2011.

[17] T. Pock, D. Cremers, H. Bischof, and A. Chambolle. An algorithm for minimizing the piecewise smooth mumford-shah functional. In *IEEE International Conference on Computer Vision (ICCV)*, Kyoto, Japan, 2009.

[18] Daniel Ricao Canelhas. Scene representation, registration and objectdetection in a truncated signed distance functionrepresentation of 3d space. Master's thesis, Örebro University, School of Science and Technology, Örebro University, Sweden, 2012.

[19] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the ICP algorithm. In *Third International Conference on 3D Digital Imaging and Modeling (3DIM)*, June 2001.

[20] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

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

[22] T. Whelan, H. Johannsson, M. Kaess, J.J. Leonard, and J.B. McDonald. Robust tracking for real-time dense RGB-D mapping with Kintinuous. Technical Report MIT-CSAIL-TR-2012-031, Computer Science and Artificial Intelligence Laboratory, MIT, Sep 2012.

[23] Kai M. Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems. In *In Proc. of the ICRA 2010 workshop*, 2010.

Chapter 7

Appendix

In this appendix we present a more detailed evaluation of our method. We compare our method with both KinFu and RGB-D SLAM. The results show that we outperform KinFu and we perform almost on the same level as RGB-D SLAM.

Teddy	Res.	RMSE	Std Dev	Max Error	Min Error
Kinfu	256	0.155356 m	0.064906 m	0.666333 m	0.014715 m
Kinfu	512	0.314937 m	0.157922 m	0.814632 m	0.006704 m
Point-To-Plane	256	**0.076112 m**	**0.026049 m**	**0.188383 m**	**0.004002 m**
Point-To-Plane	512	0.142598 m	0.088109 m	0.524915 m	0.012211 m
Point-To-Point	256	0.089038 m	0.031064 m	0.215403 m	0.012309 m
Point-To-Point	512	0.124477 m	0.059128 m	0.341314 m	0.006778 m

Table 7.1: The results for the tracking on the dataset Teddy. Clearly, our approach here outperforms KinFu.

F1 Desk	Res.	RMSE	Std Dev	Max Error	Min Error
KinFu	256	0.058550 m	0.033446 m	0.213222 m	0.008404 m
KinFu	512	0.067729 m	0.041765 m	0.239925 m	0.013357 m
Point-To-Plane	256	0.094144 m	0.063969 m	0.376336 m	0.004752 m
Point-To-Plane	512	0.143094 m	0.100204 m	0.467883 m	0.004953 m
Point-To-Point	256	**0.036481 m**	**0.016181 m**	**0.111886 m**	0.005357 m
Point-To-Point	512	0.037537 m	0.018140 m	0.176459 m	**0.002704 m**

Table 7.2: The results for the tracking on the dataset Desk. Also here our method works better than the KinFu when using point-to-point metric.

F1 Desk2	Res.	RMSE	Std Dev	Max Error	Min Error
KinFu	256	0.425655 m	0.344616 m	1.598357 m	0.063352 m
KinFu	512	1.069398 m	0.563465 m	2.381280 m	0.137786 m
Point-To-Plane	256	0.226816 m	0.081262 m	0.391790 m	0.013364 m
Point-To-Plane	512	0.257331 m	0.115499 m	0.526585 m	0.030968 m
Point-To-Point	256	**0.058547 m**	0.034641 m	**0.189066 m**	0.002457 m
Point-To-Point	512	0.065838 m	**0.034304 m**	0.201560 m	**0.002405 m**

Table 7.3: The results for the tracking on the dataset Desk2 is clearly better for our method. In particular for the point-to-point metric.

Room	Res.	RMSE	Std Dev	Max Error	Min Error
Kinfu	256	Failed	Failed	Failed	Failed
Kinfu	512	Failed	Failed	Failed	Failed
Point-To-Plane	256	0.266557 m	0.128407 m	0.488320 m	0.032583 m
Point-To-Plane	512	0.105750 m	0.038497 m	0.209663 m	**0.007082 m**
Point-To-Point	256	0.186188 m	0.084997 m	0.357137 m	0.022757 m
Point-To-Point	512	**0.071330 m**	**0.027691 m**	**0.181868 m**	0.008730 m

Table 7.4: The results for the tracking on the dataset Room, KinFu loses track completely. Interesting here is that point-to-point and point-to-plane works better for a higher resolution. In other tests, a resolution of 256 has given better results.

F2 Desk With Person	Res.	RMSE	Std Dev	Max Error	Min Error
KinFu	256	0.060193 m	0.032140 m	0.346967 m	0.008885 m
KinFu	512	**0.056934 m**	**0.026122 m**	0.351091 m	0.006457 m
Point-To-Plane	256	0.079959 m	0.039748 m	0.257505 m	0.007386 m
Point-To-Plane	512	0.088280 m	0.043563 m	0.270965 m	**0.002652 m**
Point-To-Point	256	0.069734 m	0.032211 m	**0.201468 m**	0.006958 m
Point-To-Point	512	0.073863 m	0.028920 m	0.265489 m	0.009842 m

Table 7.5: Tracking against the scene Desk With Person with a moving person in it. All methods gives good results, even though the assumption is a static environment.

F3 Long Office Household	Res.	RMSE	Std Dev	Max Error	Min Error
KinFu	256	0.061833 m	0.036903 m	0.176655 m	0.006411 m
KinFu	512	0.060245 m	0.034988 m	0.156623 m	0.005090 m
Point-To-Plane	256	0.054150 m	0.030793 m	0.143606 m	0.003244 m
Point-To-Plane	512	0.055974 m	0.031606 m	0.150762 m	**0.003207 m**
Point-To-Point	256	**0.036943 m**	**0.019271 m**	**0.114775 m**	0.003208 m
Point-To-Point	512	0.039027 m	0.020543 m	0.124296 m	0.003849 m

Table 7.6: Tracking against the dataset Long Office Household gives good results for all methods. Again, our method with point-to-point gives clearly better result than KinFu.

Teddy	Res.	RMSE	Std Dev	Max Error	Min Error
RGB-D SLAM		0.110657	0.049863 m	0.336505 m	0.010821 m
Kinfu	256	0.155356 m	0.064906 m	0.666333 m	0.014715 m
Kinfu	512	0.314937 m	0.157922 m	0.814632 m	0.006704 m
Point-To-Plane	256	**0.076112 m**	**0.026049 m**	**0.188383 m**	**0.004002 m**
Point-To-Plane	512	0.142598 m	0.088109 m	0.524915 m	0.012211 m
Point-To-Point	256	0.089038 m	0.031064 m	0.215403 m	0.012309 m
Point-To-Point	512	0.124477 m	0.059128 m	0.341314 m	0.006778 m

Table 7.7: Against the dataset Teddy, our method performs better than the other two.

F1 Desk	Res.	RMSE	Std Dev	Max Error	Min Error
RGB-D SLAM		**0.025831 m**	**0.011497 m**	**0.079256 m**	0.004203 m
KinFu	256	0.058550 m	0.033446 m	0.213222 m	0.008404 m
KinFu	512	0.067729 m	0.041765 m	0.239925 m	0.013357 m
Point-To-Plane	256	0.094144 m	0.063969 m	0.376336 m	0.004752 m
Point-To-Plane	512	0.143094 m	0.100204 m	0.467883 m	0.004953 m
Point-To-Point	256	0.036481 m	0.016181 m	0.111886 m	0.005357 m
Point-To-Point	512	0.037537 m	0.018140 m	0.176459 m	**0.002704 m**

Table 7.8: Against the dataset Desk, RGB-D SLAM performs better than the other methods, followed by our method with point-to-point metric.

F1 Desk2	Res.	RMSE	Std Dev	Max Error	Min Error
RGB-D SLAM		**0.042558 m**	**0.023261 m**	**0.183123 m**	**0.000991 m**
KinFu	256	0.425655 m	0.344616 m	1.598357 m	0.063352 m
KinFu	512	1.069398 m	0.563465 m	2.381280 m	0.137786 m
Point-To-Plane	256	0.226816 m	0.081262 m	0.391790 m	0.013364 m
Point-To-Plane	512	0.257331 m	0.115499 m	0.526585 m	0.030968 m
Point-To-Point	256	0.058547 m	0.034641 m	0.189066 m	0.002457 m
Point-To-Point	512	0.065838 m	0.034304 m	0.201560 m	0.002405 m

Table 7.9: Against the dataset Desk2, RGB-D SLAM performs better than the other methods, followed by our method with point-to-point metric.

F2 Desk With Person	Res.	RMSE	Std Dev	Max Error	Min Error
RGB-D SLAM		0.069705 m	**0.016015 m**	**0.109066 m**	0.021789 m
KinFu	256	0.060193 m	0.032140 m	0.346967 m	0.008885 m
KinFu	512	**0.056934 m**	0.026122 m	0.351091 m	0.006457 m
Point-To-Plane	256	0.079959 m	0.039748 m	0.257505 m	0.007386 m
Point-To-Plane	512	0.088280 m	0.043563 m	0.270965 m	**0.002652 m**
Point-To-Point	256	0.069734 m	0.032211 m	0.201468 m	0.006958 m
Point-To-Point	512	0.073863 m	0.028920 m	0.265489 m	0.009842 m

Table 7.11: Against the dataset Desk With Person, with a moving person in the scene, all methods performs well. RGB-D SLAM outperforms the other methods with respect to maximal error.

Room	Res.	RMSE	Std Dev	Max Error	Min Error
RGB-D SLAM		0.101165 m	0.070696 m	0.436558 m	**0.005000 m**
Kinfu	256	Failed	Failed	Failed	Failed
Kinfu	512	Failed	Failed	Failed	Failed
Point-To-Plane	256	0.266557 m	0.128407 m	0.488320 m	0.032583 m
Point-To-Plane	512	0.105750 m	0.038497 m	0.209663 m	0.007082 m
Point-To-Point	256	0.186188 m	0.084997 m	0.357137 m	0.022757 m
Point-To-Point	512	**0.071330 m**	**0.027691 m**	**0.181868 m**	0.008730 m

Table 7.10: Against the dataset Room, our method with point-to-point metric performs the best.

Master's Theses in Mathematical Sciences 2012:E41
ISSN 1404-6342

LUTFMA-3235-2012

Mathematics
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/