Rafael WEILHARTER, BSc.

Globally Consistent Dense Real-Time 3D Reconstruction from RGBD Data

MASTER'S THESIS

to achieve the university degree of
Diplom-Ingenieur

Master's degree programme
Telematics

submitted to

Graz University of Technology

Supervisor
Ass.Prof. Dipl.-Ing. Dr.techn. Friedrich Fraundorfer
Institute for Computer Graphics and Vision

Advisor
Dipl.-Ing. Fabian Schenk

Graz, Austria, May 2018

Abstract

Creating dense 3D reconstructions from real-world scenes has been an active research topic in recent years, especially with the emergence of cheap RGBD sensors like the Microsoft Kinect. Accurate 3D models from scans can be employed in a variety of fields including robotics (navigation), augmented/virtual reality and gaming. The challenges of such a system are manifold, from the accuracy of the trajectory estimation via the efficient memory usage through to the ability to react to model updates in real-time.

In this thesis, we present a dense 3D reconstruction framework for RGBD data that can handle loop closure, i.e. a closure in the trajectory through recognition of an already visited scene, and other pose updates online. Handling updates online is essential to get instant feedback about the coverage and quality of the scan and generate a globally consistent 3D reconstruction in real-time. Hence, we introduce fused depth maps for each keyframe that contain the fused depths of all associated frames to greatly increase the speed for model updates. Furthermore, we show how we can use integration and de-integration in a volumetric fusion system to adjust our model to online updated camera poses. In addition, we propose a plane estimation algorithm to detect and complete large planes within our model. Based on the Manhattan environment assumption, i.e. man-made environments are build on a Cartesian grid which leads to regularities, this idea can further improve the model.

We build our system on top of the InfiniTAM framework to generate a dense 3D model from the sparse, keyframe-based ORB-SLAM2 reconstruction. We extensively evaluate our system on real world and synthetic generated RGBD data regarding tracking accuracy and surface reconstruction.

Keywords. dense 3D reconstruction, RGBD data, volumetric fusion, SLAM, plane estimation

Kurzfassung

Die Erstellung dichter 3D-Rekonstruktionen aus realen Szenen war in den letzten Jahren ein vieldiskutiertes Forschungsthema, insbesondere wegen des Aufkommens leistbarer RGBD-Sensoren, wie der Microsoft Kinect. Präzise 3D-Modelle durch Scans können in einer Vielzahl von Bereichen wie Robotik (Navigation), Augmented / Virtual Reality und Gaming eingesetzt werden. Die Anforderungen an ein solches Systems sind vielschichtig, von der Genauigkeit der Trajektorienabschätzung über die effiziente Speichernutzung bis hin zur Fähigkeit, auf Modellaktualisierungen in Echtzeit zu reagieren.

In dieser Arbeit präsentieren wir ein dichtes 3D-Rekonstruktions-Framework für RGBD-Daten, das Loop-Closure, i.e. Schließung der Trajektorie durch Wiedererkennung bereits besuchter Szeneteile, und andere Posenveränderungen online durchführen kann. Online-Updates sind wichtig, um sofort Rückmeldung über die Vollständigkeit und Qualität des Scans zu erhalten und eine global konsistente 3D-Rekonstruktion in Echtzeit zu erstellen. Für diesen Zweck führen wir fusionierte Tiefenkarten für jedes Schlüsselbild ein, die die Information der Tiefen aller zugehörigen Bildern enthalten, um die Geschwindigkeit für die Modellaktualisierung erheblich zu erhöhen. Des Weiteren zeigen wir, wie wir Integration und De-Integration in einem volumetrischen Fusionssystem nutzen können, um unser Modell an in Echtzeit aktualisierte Kameraposen anzupassen. Darüber hinaus stellen wir einen Algorithmus zur Schätzung von Ebenen vor, um große Ebenen innerhalb unseres Modells zu erkennen und zu vervollständigen. Diese Idee basiert auf der Annahme der Manhattan-Umgebung, i.e. von Menschen erbaute Strukturen weisen geometrische Regelmäßigkeiten auf (Wände, Böden), und kann das 3D Modell weiter verbessern.

Wir bauen unser System auf dem InfiniTAM-Framework auf, und generieren aus der spärlichen, Schlüsselbild-basierten ORB-SLAM2 Rekonstruktion ein dichtes 3D-Modell. Wir evaluieren unser System auf realen und synthetisch generierten RGBD-Daten bezüglich Trajektorienabschätzung und Oberflächenrekonstruktion.

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used.

The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Date	Signature

Acknowledgments

First of all, I would like to express my deep gratitude to my advisor, Dipl.-Ing. Fabian Schenk, for his outstanding support during this thesis. His ideas, hints and fruitful discussions paved the way to the completion of this work.

I would also like to thank my supervisor, Ass.Prof. Dipl.-Ing. Dr.techn. Friedrich Fraundorfer, for enabling me to conduct my Master's Thesis at the Institute for Computer Graphics and Vision.

Last but not least, I want to thank my friends and family, especially my parents, for providing me with unfailing support and continuous encouragement throughout my years of study.

This work was financially supported by the Ludwig Boltzmann Institute for Clinical Forensic Imaging (LBI CFI).

Contents

List of Figures

List of Tables

1 Introduction

Contents

1.1 Motivation

In this thesis, we address the problem of creating accurate, dense 3D models in real-time. Building dense 3D models has potential in a variety of fields including cultural heritage or famous touristic sites scans, reconstruction of houses, buildings, rooms, objects etc. In recent years, the ubiquity of inexpensive RGBD cameras, pioneered by the Microsoft Kinect, has led to a series of applications for indoor 3D scene reconstruction in research areas such as augmented/virtual reality, robotics and gaming. The generated 3D models can also be of advantage to the technical inexperienced consumer: For example, an accurate, photo-realistic model of a room, can enable scenarios like virtual remodeling or online furniture shopping. Furthermore, such a model can help in court proceedings by providing a precise snapshot of the crime scene for later investigations. In order to be usable in these scenarios, the system needs to run on a portable device, ideally a laptop or tablet with standard hardware. Moreover, the challenges lie in an automated model generation and adaption in real-time, allowing the user to get instant feedback about the coverage and quality of the scan. This does not only require a robust camera pose estimation algorithm but also on-the-fly model updates that incorporate loop closures and pose refinements. The robust camera motion estimation process is the main difference between current systems and roughly divides them into three categories: (i) Iterative closest point (ICP) methods aim to align 3D points but require sufficient 3D structure and a correspondence matching step [29, 43]. Instead of point clouds, (ii) direct methods esti-

mate motion by processing image information directly. Dense [31], semi-dense [8, 15] and sparse [13] variants based on the photo-consistency assumption exist. This makes these approaches especially susceptible to illumination changes and direct methods are typically restricted to small inter-frame motion. (iii) Feature-based methods extract features, match correspondences and estimate motion by minimizing the reprojection error [12, 42]. The extracted features are more robust to illumination changes than the direct methods based on the photo-consistency assumption and are also suitable for larger inter-frame motions.

Regardless of the applied method, many systems rely solely on frame-to-model tracking to estimate the camera movement in real-time and sequentially build a dense 3D model [29, 43]. While such systems are real-time capable they accumulate significant drift over time and usually cannot correct this drift by revisiting the same place (see Fig. 1.1). To tackle this problem, more advanced Simultaneous Localization and Mapping (*SLAM*) systems perform loop closure and pose graph optimization to reduce the drift. However, such an approach is computationally very expensive and often only acquires a semi-dense [15] or sparse map [13, 42]. If in addition a dense 3D model is desired, the SLAM system usually relies on expensive hardware, e.g. the Bundle Fusion system [8] only runs on a combination of an NVIDIA GeForce GTX Titan X and a GTX Titan Black.

1.2 Contribution

In this thesis, we propose a real-time dense 3D reconstruction method that successfully combines the state-of-the-art ORB-SLAM2 system [42] with the dense volumetric fusion framework InfiniTAM [29] and works on standard consumer hardware. We extend InfiniTAM to support the necessary operations that are required for online dense reconstruction such as updating and re-adjusting the 3D model to find and estimate loop closures and pose refinements in real-time. Furthermore, we propose a method to estimate large planes in our 3D model to complete geometric structures (see Fig. 1.2). The idea is based on the Manhattan environment assumption, i.e. the assumption that in man-made environments plane-like structures dominate (e.g. walls, floors, etc.) and can contribute to a more complete model. This can be especially important in applications where boundaries need to be checked, e.g. in an augmented reality application a character might be prevented from falling through a "hole" in an incompletely scanned wall.

To validate our method, we compare the trajectory estimations and surface reconstruction accuracy of several methods on the standard TUM RGBD benchmark dataset, the BundleFusion dataset and the synthetic ICL NUIM dataset. The key contributions presented in this work can be summarized as:

- Implementation of a de-integration method which allows to refine and alter the 3D model online when large changes in the estimated trajectory occur, e.g. in the case of a loop closure detection.
- A global model update which can delete and merge keyframes in retrospect.

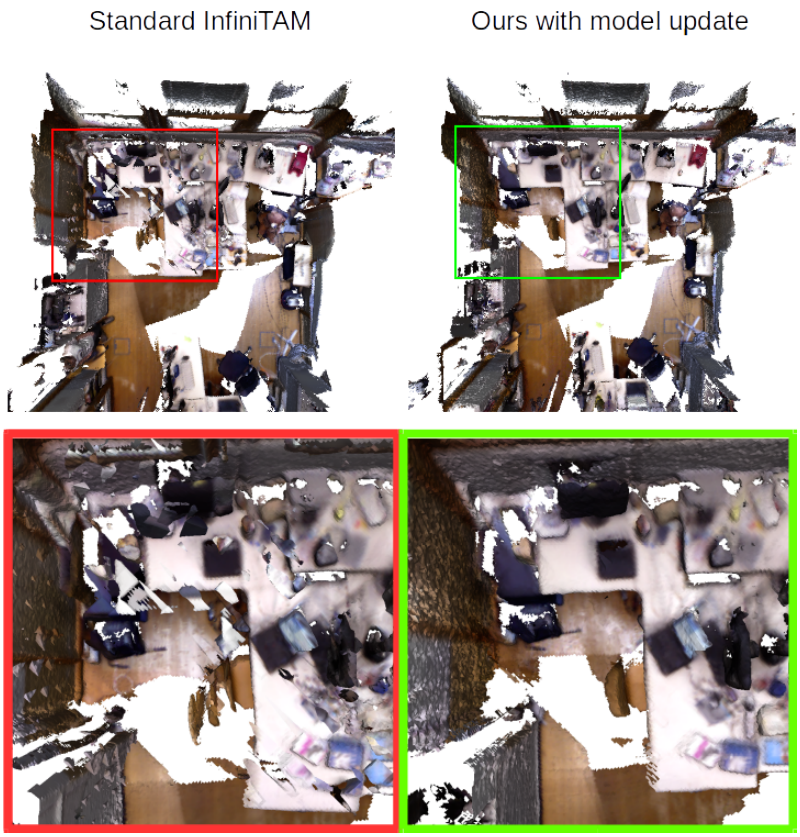


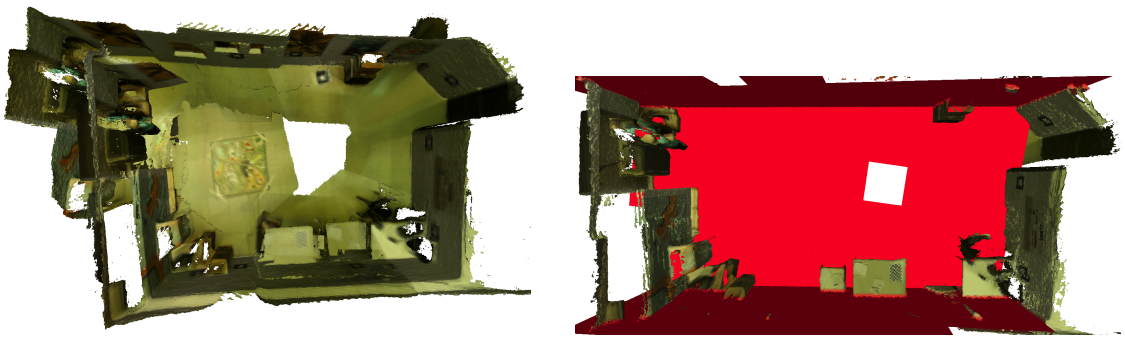
Figure 1.1: Reconstructed dense 3D Models: Standard InfiniTAM vs our system with keyframe-based depth map fusion and global model update. We can see the effects especially in the upper left corners where a loop closure occurs.

- A keyframe-based depth fusion, where we fuse information of frames into their respective keyframes instead of integrating them directly into the model. This significantly speeds up the re-integration process required for a global model update.

- A method to estimate planes in a dense 3D model to further improve the model by eradicating noise and filling holes in scans of planar geometry like walls and floors

- An extensive evaluation of both, the trajectory error and the surface reconstruction error on several benchmark data sets

Parts of this thesis were submitted as a paper [66] to the Austrian Association for Pattern Recognition (OAGM/AAPR) Workshop 2018 and accepted. The covered topics include de-integration, depth fusion and global model update.



(a) Original InfiniTAM **(b)** Our approach

Figure 1.2: Original InfiniTAM vs our approach with model update and plane estimation.

1.3 Outline

The remainder of this thesis is structured as follows: In Chapter 2, we introduce the reader to the theoretical concepts which form the basis of this thesis. We also discuss related work and state-of-the-art methods.

Chapter 3 explains our approach in detail. We show the work flow of our system and how we apply the mathematical theory.

We describe our experiments and results in Chapter 4. We provide a comprehensive evaluation on several real-world and synthetic datasets. Furthermore, we present the globally consistent 3D models of our own RGBD sensor recordings.

We conclude the thesis in Chapter 5, where we summarize our method and discuss potential future directions.

2 Background and Related Work

This chapter describes the sensor and the theoretical concepts of this thesis. We explain the majority of appearing terms and the underlying mathematical theory. At first we introduce the notations and conventions we follow throughout this thesis. Then we give an overview of how to describe a motion in an image series. Next, we discuss different methods of how to obtain a movement trajectory and estimate a 3D model from visual input only. Furthermore, we discuss related work in the area of VO/SLAM and dense reconstruction. Finally, we talk about how plane estimation can help to improve the 3D model estimation.

Contents

Entity	Notation
Scalar	a, b_i
Vector 3D	$\mathbf{X} = (X, Y, Z)^\top$
Homogeneous Vector 3D	$\tilde{\mathbf{X}} = \left(\mathbf{X}^\top, 1\right)^\top$
Vector	$\mathbf{x}$
Homogeneous Vector	$\tilde{\mathbf{x}} = \left(\mathbf{x}^\top, 1\right)^\top$
Matrix	$\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
Transformation Matrix ($a \to b$)	$\mathbf{T_{b,a}} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix}$
Vector Space	$\mathbb{R}^3$
Mappings	$\pi : \mathbb{R}^3 \to \mathbb{R}^2$
Functions	$\mathcal{P}$

Table 2.1: Mathematical notations used in this thesis.

2.1 Notation and Conventions

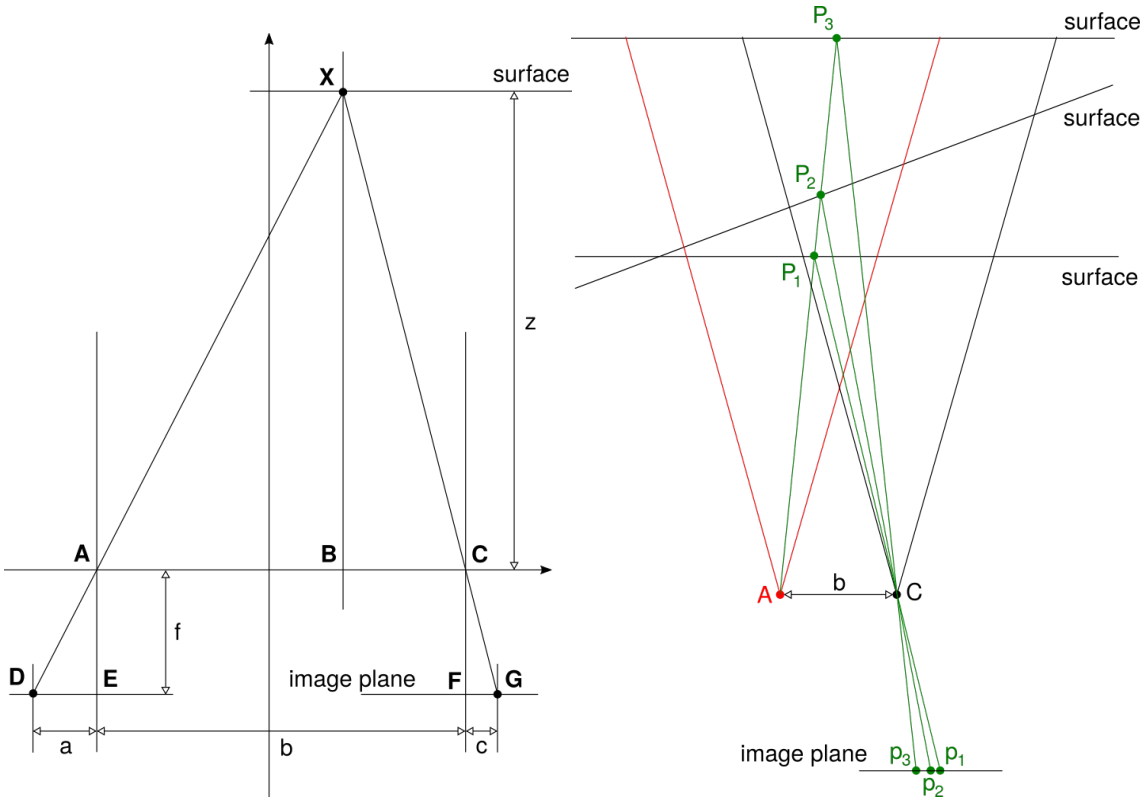
Throughout this thesis, we adopt the following conventions:

- Scalar values are denoted by italic fonts, e.g. a or b_i,
- Matrices and 3D (column-)vectors are represented by bold, upper-case letters, e.g. $\mathbf{M}$ or $\mathbf{X}$, and all other (column-)vectors by bold, lower-case letters, e.g. $\mathbf{v}$,
- Vector spaces are given by double-lined upper case letters, e.g. $\mathbb{R}^3$ or $\mathbb{Z}^3$,
- Homogeneous representations of points/vectors $\mathbf{x} \in \mathbb{R}^2$ or $\mathbf{X} \in \mathbb{R}^3$ are characterized by a tilde, e.g. $\tilde{\mathbf{x}} = \left(\mathbf{x}^\top, 1\right)^\top \in \mathbb{R}^3$ or $\tilde{\mathbf{X}} = \left(\mathbf{X}^\top, 1\right)^\top \in \mathbb{R}^4$,
- Mappings between different spaces are denoted in lower case Greek letters, e.g. $\pi : \mathbb{R}^3 \to \mathbb{R}^2$,
- Functions are denoted in upper case calligraphic letters, e.g. $\mathcal{G}$ or $\mathcal{T}$,
- Transformation matrices, which transform a homogeneous 3D point $\tilde{\mathbf{X}}$ from frame a to frame b are given by $\mathbf{T_{b,a}} \in \mathbb{R}^{4\times4}$.

Table Table 2.1 shows an overview of the notations we follow in this thesis.

2.2 Sensors

One of the crucial parts of any image-requiring computer application is the selection of an adequate image sensor, i.e. a camera. A first choice would be a monocular camera, which



(a) Passive Stereo **(b)** Active Stereo

Figure 2.1: Canonical stereo configuration: (a) Passive Stereo: A 3D point **X** is projected through the two camera centers **A** and **C**. The relationship between a + c, b, f and z is determined by the similar triangle pairs ($\overline{\text{XBA}}$, $\overline{\text{AED}}$) and ($\overline{\text{XCB}}$, $\overline{\text{CGF}}$). (b) Active Stereo: The baseline is now between the projector center **A** and the camera center **C**. A change in depth ($\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$) results in a horizontal shift in the image plane ($\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_3}$).

passively captures a single image/image stream. However, as this thesis focuses on 3D scene reconstruction and a monocular setup is unable to provide a scale factor we dismiss this option. 3D lasers are a costly alternative and not all of them do provide the often needed color information (for visual features).

Therefore, we consider a stereo setup: In order to recover the depth z of point **X** we need to apply stereo geometry to Figure 2.1. Since the triangles $\overline{\text{XBA}}$ and $\overline{\text{AED}}$ as well as $\overline{\text{XCB}}$ and $\overline{\text{CGF}}$ are similar we get $z : b = f : d$, with $d = a + c$ and can therefore calculate:

$$z = \frac{bf}{d} \quad , \tag{2.1}$$

where d is the disparity, b the baseline and f the focal length. This is known as the canonical stereo configuration.

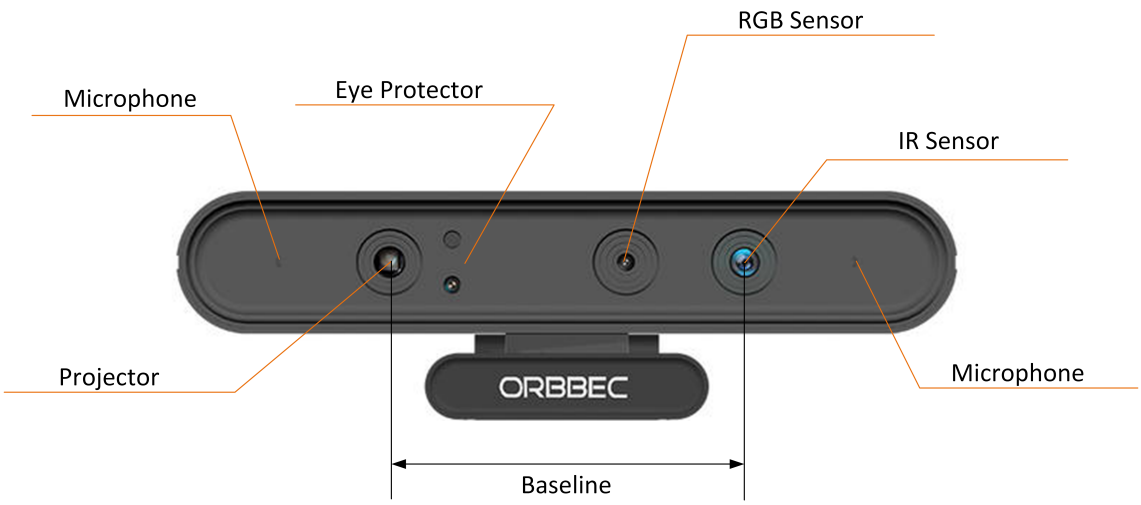
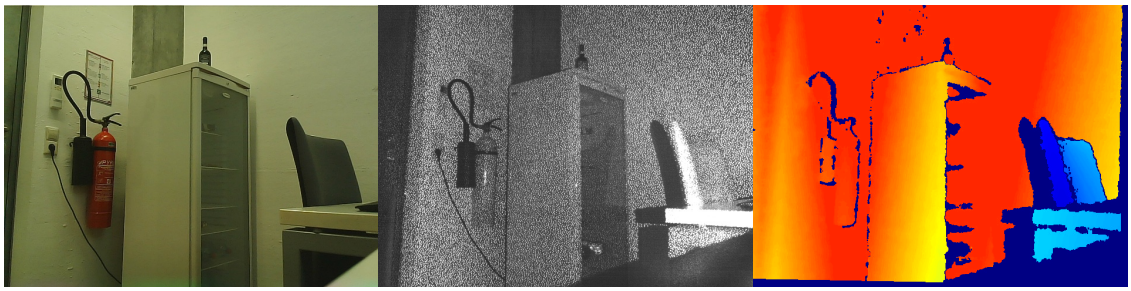


Figure 2.2: Orbbec Astra Pro sensor [46] with its main components: IR Projector, IR Sensor and RGB Sensor. It furthermore includes 2 microphones and an Advanced Eye Protector which shuts down the projector if an obstacle is within 40cm of the sensor. The baseline in the active stereo case is between the projector center and the camera center.



(a) RGB image **(b)** IR image **(c)** Depth image

Figure 2.3: Orbbec Astra Pro images: (a) acquired from the RGB sensor, (b) aquired from the IR sensor and (c) calculated from the known pattern in the IR image (blue near, red far).

In the passive case, (at least) two conventional cameras with known baseline, known focal length and overlapping fields of view are used. To obtain the disparity, image correspondences need to be found, i.e. finding the same scene point in the two images (see Sec. 2.6). Nevertheless, such a system suffers from a lack or a repeating of texture and when no correspondences can be found, e.g. walls, floors, too much rotation between images, etc.

To tackle these problems, we can take advantage of active stereo, i.e a system that actively emits light. With the release of the Microsoft Kinect, these RGBD-Sensors have become available to a broad audience. RGB refers to the 3 colors red, green and blue, which are added in various ways in the RGB color model to represent a broad array of

colors. In addition to the 3 RGB channels the "D" refers to a fourth "depth" or "distance" channel. RGBD-Sensors consist of 3 main components: (i) an InfraRed (*IR*) projector, (ii) an IR sensor and (iii) an RGB camera. The depth data of the fourth channel can be acquired by different principles such as structured light or Time of Flight (ToF). In the first case, a known infrared light pattern is projected onto the scene. The illumination appears to be distorted from the infrared sensor view point, which is different from that of the projector. This distortion allows to geometrically reconstruct the surface and extract a depth measurement with Equation 2.1, where the baseline b is now the distance between IR projector and IR sensor. On the other hand, the TOF approach uses the known speed of light to determine the distance. A light pulse is emitted from a projector to illuminate the scene for a short time. Afterwards, the reflected light is collected by the camera lens and the distance can be calculated from the time delay.

A natural application for the data provided by such sensors is 3D reconstruction and SLAM, which will be discussed in the later sections. The downsides of both principles include their ineptitude for outdoor reconstructions due to sunlight, which can render the IR sensor unable to detect the projected IR pattern, their faultiness on reflective surfaces like glass and their susceptibility to noise. Despite these disadvantages we choose the structured light approach due to its ubiquitous availability, real time capacity and our focus on indoor environments.

The RGBD-Sensor providing the image stream in this thesis is the Orbbec Astra Pro sensor, which can be seen in Figure 2.2. This version has an enhanced 720p @ 30 frames-per-second RGB camera and a range of 0.6m - 8m. Table 2.2 shows the further specifications for this sensor. It captures RGB and IR images and provides a depth image (see Fig. 2.3).

We choose this sensor due to its high resolution, high range and its relatively small size (compared for example to the Microsoft Kinect). Further selection criteria included that the Asus Xtion is no longer produced and that the Intel RealSense ZR300, which is comparable in size, suffers from heavy noise above 2 - 2.5m distance.

2.3 Camera Model

The camera model describes the relationship of points in the real 3D world to those in the 2D image plane. We utilize the common pinhole camera model (see Fig. 2.4).

At each timestep t, we receive an RGBD frame that consists of an RGB image I_t and a depth map D_t from our sensor. We expect I_t and D_t to be aligned and synchronized, such that at a certain pixel position $\mathbf{x} = (x, y)^\top$ the RGB values are given as $I_t(\mathbf{x})$ and the corresponding depth as $D_t(\mathbf{x})$. Let $\mathbf{X} = (X, Y, Z)^\top$ be a 3D point in the camera reference frame with depth measurement $Z = D_t(\mathbf{x})$ and $\mathbf{x} = (x, y)^\top$ its projected image coordinates. The mapping $\pi(\mathbf{X}) : \mathbb{R}^3 \to \mathbb{R}^2$ is then given by the perspective projection

Entity	Value
Size/Dimensions	160 × 30 × 40 mm
Weight	0.3 kg
Range	0.6 - 8m (Optimal 0.6 - 5m)
Depth Image Size	640 × 480 (VGA) @ 30FPS
RGB Image Size	1280 × 720 @ 30FPS (UVC Support)
Field of View	60° horiz × 49.5° vert. (73°diag)
Data Interface	USB 2.0
Microphones	2
Operating Systems	Windows, Linux, Android
Power	USB 2.0
Software	libUVC + OpenNI

Table 2.2: Specifics of the Orbbec Astra Pro sensor [46].

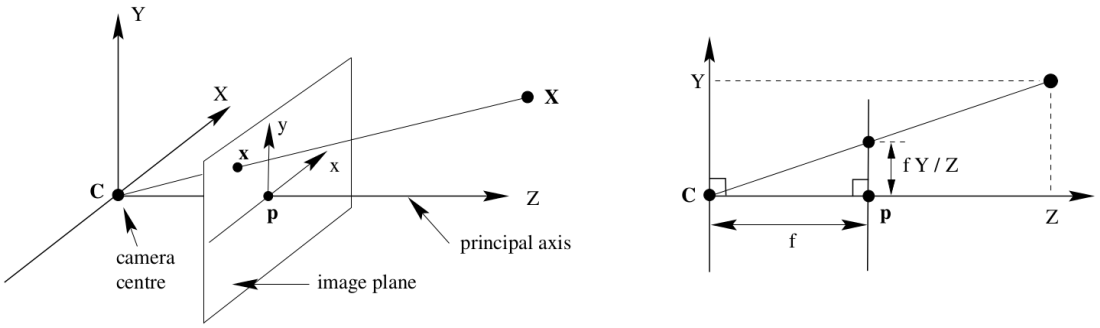


Figure 2.4: Pinhole camera model [24]. The focal length f scales the image coordinates x and y.

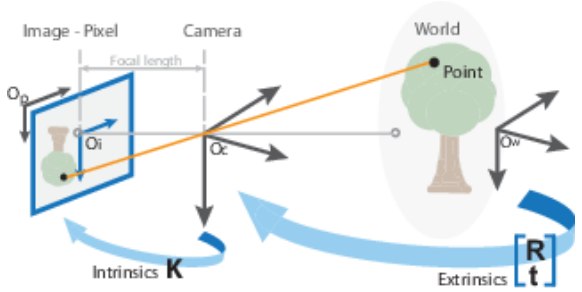


Figure 2.5: Depiction of intrinsic and extrinsic camera parameters [38].

equation with scaling factor a:

$$a \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \mathbf{K}\mathbf{X} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad , \tag{2.2}$$

where $\mathbf{K}$ is the intrinsic camera matrix, f_x and f_y the focal lengths and c_x, c_y the offset of the optical center $\mathbf{C}$. The two different focal lengths are obtained through multiplying the actual physical focal length, as seen in 2.4, with the scaling factors s_x and s_y in the respective direction, thus $f_x = f \cdot s_x$ and $f_y = f \cdot s_y$. They possess different values, if the pixels in the Charge-Coupled Device (CCD) array are not square. All these intrinsic parameters are measured in pixels. Note that in this model, the sensor skew parameter gets neglected due to the assumption that the image coordinate axes are orthogonal to each other and therefore the parameter equals zero. The mapping $\pi(\mathbf{X})$ can be written in compact notation:

$$\begin{aligned} \pi(\mathbf{X}) &= \mathbf{x} \\ \pi\left(\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}\right) &= \begin{bmatrix} \frac{X \cdot f_x}{Z} + c_x \\ \frac{Y \cdot f_y}{Z} + c_y \end{bmatrix} \quad . \end{aligned} \tag{2.3}$$

If the depth measurement $Z = D_t(\mathbf{x})$ is known, the 3D scene point can be recovered from the image coordinates by using the inverse mapping $\pi^{-1}(\mathbf{x})$:

$$\begin{aligned} \pi^{-1}(\mathbf{x}, D_t(\mathbf{x})) &= \mathbf{X} \\ \pi^{-1}\left(\begin{bmatrix} x \\ y \end{bmatrix}, D_t(\mathbf{x})\right) &= \begin{bmatrix} \frac{x - c_x}{f_x} D_t(\mathbf{x}) \\ \frac{y - c_y}{f_y} D_t(\mathbf{x}) \\ D_t(\mathbf{x}) \end{bmatrix} \quad . \end{aligned} \tag{2.4}$$

So far, the camera is always located in the 3D world center (0, 0, 0). If the 3D position of the camera needs to be considered, the extrinsic parameters $\mathbf{R}$, $\mathbf{t}$ have to be taken into account:

$$a\tilde{\mathbf{x}} = \mathbf{K}[\mathbf{R}\ \mathbf{t}]\tilde{\mathbf{X}} = \mathbf{P}\tilde{\mathbf{X}} \tag{2.5}$$

where $\mathbf{P}$ is the so called camera matrix, $\mathbf{R}$ the rotation matrix and $\mathbf{t}$ the translation vector. The properties of $\mathbf{R}$ and $\mathbf{t}$ will be discussed in detail in Section 2.5. As depicted in Figure 2.5, the extrinsic parameters determine the rigid body transformation $\mathbb{R}^3 \rightarrow \mathbb{R}^3$ from 3D world coordinates to the 3D camera's coordinate system. How to obtain the intrinsic (f_x, f_y, c_x, c_y) and extrinsic ($\mathbf{R}$, $\mathbf{t}$) parameters is described in the next section.

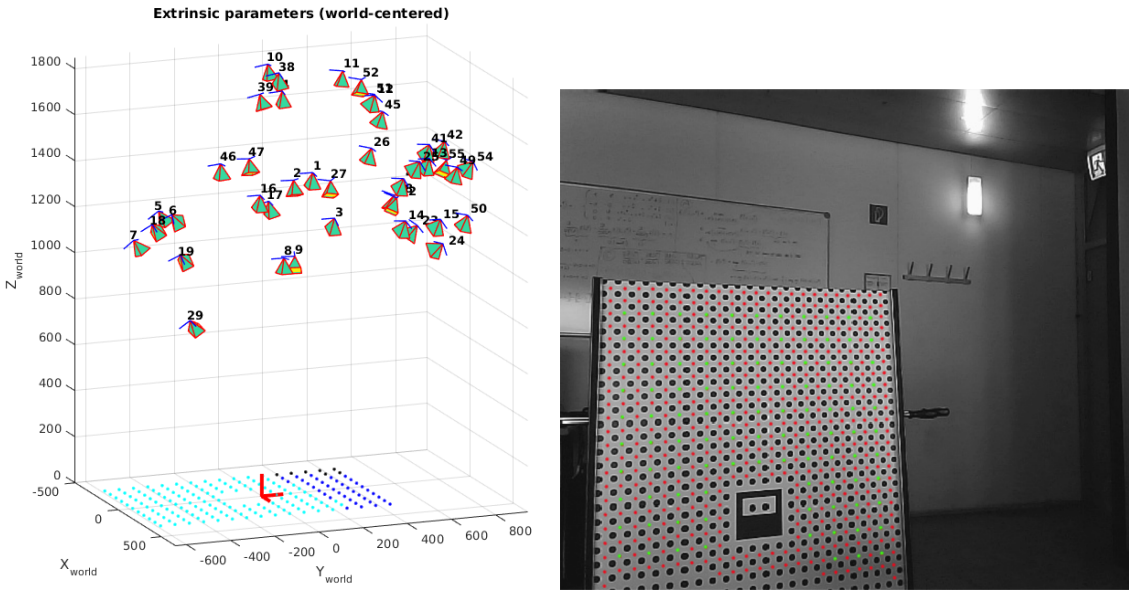
Figure 2.6: Camera calibration target with circular pattern from different positions and angles as captured by the RGB sensor (top) and the IR sensor (bottom).

2.4 Camera Calibration

Camera calibration is the process of estimating the intrinsic, extrinsic and distortion camera parameters. For this task we need 3D scene points and their corresponding 2D image points. We acquire these by taking multiple images from different angles and positions of a calibration target with known dimensions, which contains a specific pattern (i.e. squares (checkerboard), circles, QR-codes and so on). From these correspondences we then calculate the camera parameters.

For calibration, we use the target depicted in Figure 2.6. It consists of a central marker with other circular patterns around it. This calibration target has several advantages [17]: Firstly, the target does not have to be visible as a whole as opposed to standard checkerboard targets. Secondly, groups of circular patterns are more robust to distortion. Thirdly, the feature detection is more accurate for low-resolution cameras.

We perform the calibration in this thesis with the matlab calibration toolbox [3] with the ICG addon "automatic feature extraction for camera calibration" [17] which utilizes the pinhole camera model that was described in Section 2.3. Note that, although this model does not account for lens distortion due to the fact that an ideal pinhole does simply not have a lens, the radial and tangential lens distortion is still calculated by the algorithm. However, as shown in [58], the lens distortion effects can be neglected, if the distortion parameters are low enough and the accuracy requirement can still be met. We record 55 IR and RGB images as input for the algorithm whose camera positions and angles can be seen in Figure 2.7. Table 2.3 shows the calibration results of our sensor and also lists the skew parameter α and distortion coefficients kc. We usually operate our sensor within a working distance of 1 - 5m.



(a) All camera positions **(b)** Feature extraction from target

Figure 2.7: Camera positions and target detection: (a) Estimated camera positions and angles of the recorded data. The target is fixed and can be seen at the bottom. (b) Extracted features from a sample image of the target.

Parameter	Result
f_x	594.83209
f_y	596.78291
c_x	320.97477
c_y	238.99285
α	0.00000
kc	$[0.11909; -0.16726; 0.00156; 0.00081; 0.00000]$

Table 2.3: Camera calibration results. Focal lenghts f_x, f_y: The focal lengths in pixels. Principal point offsets c_x, c_y: Offset coordinates in pixels. Skew parameter α: The skew coefficient defining the angle between the x and y pixel axes. Distortion coefficient kc: The image distortion coefficients (radial and tangential distortions), stored in a 5×1 vector.

2.5 Rigid Body Motion

A rigid body is an object that does not have any internal degrees of freedom, i.e. the distances between any point pair p, q is fixed:

$$|p - q| = constant \quad . \tag{2.6}$$

Hence, a rigid body motion is a mapping $\mathbb{R}^3 \to \mathbb{R}^3$ which preserves the orientation and distance between any such pair, also known as a special Euclidean transformation. The set of all these direct displacements in the 3D Euclidean space forms the special Euclidean group SE(3).

A rigid body transformation is always relative to some reference coordinate frame and has 6 degrees of freedom: three for translation and three for rotation. The translation part is able to change the location of the object in space, whereas the rotational part can change its orientation. In order to represent the rotation, various alternatives like quaternions, Euler angles, rotation vectors or rotation matrices exist [9]. Due to the neat mathematical attributes a matrix offers, the latter is widely used. A proper rotation matrix is a 3×3 orthogonal Matrix $\mathbf{R}$ and belongs to the special orthogonal group SO(3). This is expressed by the properties

$$\mathbf{R}^\top \mathbf{R} = \mathbf{R}\mathbf{R}^\top = \mathbf{I} \quad , \quad \mathbf{R}^\top = \mathbf{R}^{-1} \quad , \tag{2.7}$$

where $\mathbf{R}^\top$ denotes the transpose of $\mathbf{R}$, $\mathbf{R}^{-1}$ denotes the inverse of $\mathbf{R}$ and $\mathbf{I}$ is the 3×3 identity matrix, and

$$det\, \mathbf{R} = 1 \quad , \tag{2.8}$$

Note, that if the determinant is -1, the rotation matrix is called improper (produces a reflection) and is not a rigid body transformation. Although $\mathbf{R}$ has 9 components, it only has 3 degrees of freedom (corresponding to the possible rotation angles around the x, y and z axis respectively), being restricted by its orthogonality requirement and a norm of 1 for all its rows and columns to preserve the length of any vector it is applied to. The translation of the rigid body motion is expressed as a translation vector $\mathbf{t} = (t_x, t_y, t_z)^\top$. It also represents 3 degrees of freedom for the 3 possible displacements t_x, t_y and t_z in the x, y and z directions respectively. By combining those two components, a rigid body motion $\mathcal{G}$ for any 3D point $\mathbf{X} = (X, Y, Z)^\top$ is defined as

$$\mathcal{G}(\mathbf{X}) = \mathbf{R} \cdot \mathbf{X} + \mathbf{t} \quad . \tag{2.9}$$

This can also be expressed with the help of a transformation matrix $\mathbf{T}$ and the extension of $\mathbf{X}$ to its homogeneous coordinates $\tilde{\mathbf{X}} = (X, Y, Z, 1)^\top$:

$$\mathcal{G}(\tilde{\mathbf{X}}) = \mathcal{G}(\mathbf{T}, \tilde{\mathbf{X}}) = \mathbf{T} \cdot \tilde{\mathbf{X}} \quad , \tag{2.10}$$

with

$$\mathbf{T} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad , \quad \mathbf{T^{-1}} = \begin{bmatrix} \mathbf{R}^\top & -\mathbf{R}^\top \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix} \quad . \tag{2.11}$$

From Equation 2.10, we can calculate the coordinates of the transformed point $\tilde{\mathbf{X}}' =$

$(X', Y', Z', 1)^\top$:

$$\begin{aligned} X' &= r_{11}X + r_{12}Y + r_{13}Z + t_x \quad , \\ Y' &= r_{21}X + r_{22}Y + r_{23}Z + t_y \quad , \\ Z' &= r_{31}X + r_{32}Y + r_{33}Z + t_z \quad . \end{aligned} \tag{2.12}$$

The matrix representation is particularly advantageous when multiple transformations need to be chained (left multiplication of transformation matrices):

$$\mathbf{T_j}(\mathbf{T_i} \cdot \tilde{\mathbf{p}}) = (\mathbf{T_j}\mathbf{T_i}) \cdot \tilde{\mathbf{X}} = \mathbf{T} \cdot \tilde{\mathbf{X}} \quad , \tag{2.13}$$

or $\tilde{\mathbf{X}}$ needs to be recovered from $\tilde{\mathbf{X}}'$ (using the inverse):

$$\tilde{\mathbf{X}} = \mathbf{T^{-1}} \cdot \tilde{\mathbf{X}}' \quad . \tag{2.14}$$

A rigid body motion can also be regarded as a change of reference frame, i.e. a change of observer. For example if we have two images taken from different angles and/or positions and know the transformation matrix between the two cameras, we can project the information from one image into the frame of the other.

The transformation matrices for such an operation are denoted as follows:

- Converting from frame i to j: $\mathbf{T_{j,i}}$
- Converting from frame i to j: $\mathbf{T_{j,i}}^{-1} = \mathbf{T_{i,j}}$

With the knowledge of Equation 2.13 and 2.14, transformations e.g. from frame j into frame k, while only knowing the transformations $\mathbf{T_{j,i}}$ and $\mathbf{T_{k,i}}$, can now be conveniently written:

$$\mathbf{T_{k,i}}\mathbf{T_{j,i}}^{-1} \cdot \tilde{\mathbf{X}} = \mathbf{T_{k,i}}\mathbf{T_{i,j}} \cdot \tilde{\mathbf{X}} = \mathbf{T_{k,j}} \cdot \tilde{\mathbf{X}} \quad . \tag{2.15}$$

We choose this definition because it can easily be seen which frame (rightmost) is transformed into which (leftmost) and in which order the matrices must be chained (numbers of transformation matrices must be aligned, highlighted in red in Eq. 2.15). If we want to estimate $\mathbf{T_{j,i}}$, point correspondences between two different images have to be found. This can be done algorithmically and is described in the next section.

Since the rigid motion $\mathcal{G}$ only has 6 degrees of freedom, $\mathbf{T}$ with its 12 parameters is over-parametrized. We use a minimal representation as twist coordinates ξ defined by the Lie algebra $se(3)$ associated with the group $SE(3)$. From the 6-vector ξ the transformation matrix $\mathbf{T}$ can be recovered by the matrix exponential $\mathbf{T} = exp(\xi)$.

We define the full warping function τ that re-projects $\mathbf{x}$ from frame j with depth $D_j(\mathbf{x})$ to frame i under the transformation matrix T_{ij} as:

$$\mathbf{x}' = \tau(\xi_{\mathbf{i,j}}, \mathbf{x}, D_j(\mathbf{x})) = \pi(\mathbf{T}_{i,j}\pi^{-1}(\mathbf{x}, D_j(\mathbf{x}))) \quad . \tag{2.16}$$

2.6 Features

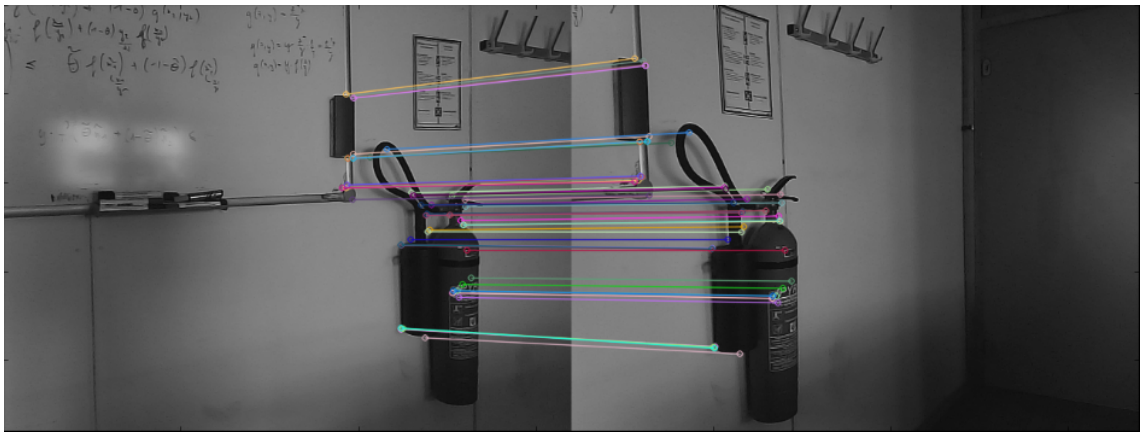


Figure 2.8: Real world data of ORB features and matches. Only the 50 best matches are drawn.

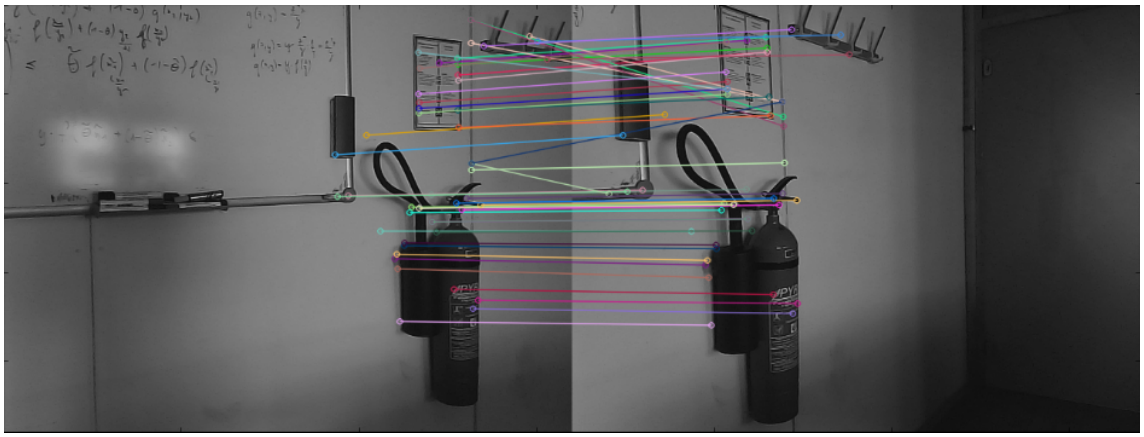


Figure 2.9: Real world data of SIFT features and matches. Only the 50 best matches are drawn.

Features describe salient parts of an image. There are several different types of image features: (i) Edges are points, which appear as boundaries between two image regions. Edge points usually possess a strong gradient magnitude. (ii) Corners are edges with a rapid change in direction. They are also known as interest points. (iii) Blobs are regions that differ in certain properties (e.g. brightness, color) from their neighboring regions. Inside the blob, the properties are approximately constant and similar. Further types of features may include lines (see Line Segment Detector (LSD) [63]), especially seen in elongated objects, or represent the underlying surface model properties like Fast Point Feature Histograms (FPFH) in [54].

In order to be able to work with features, they have to be detected, extracted and

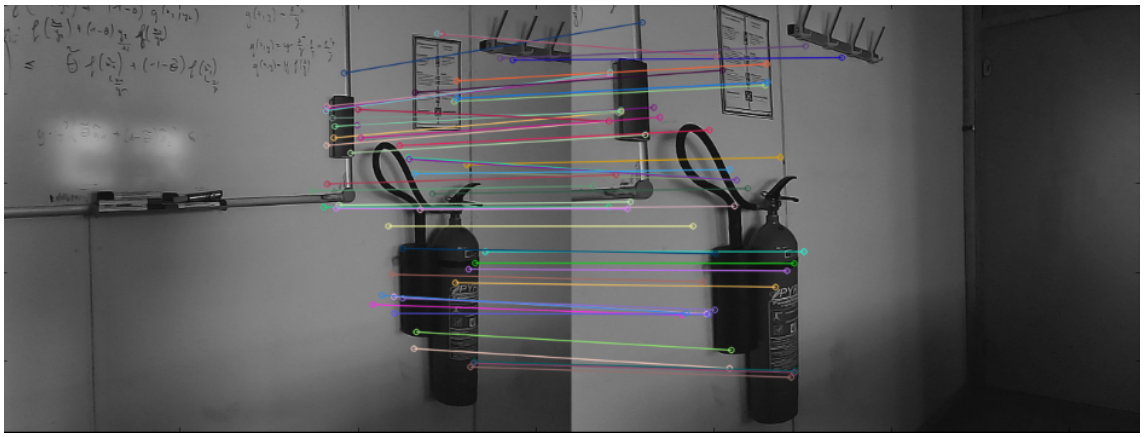
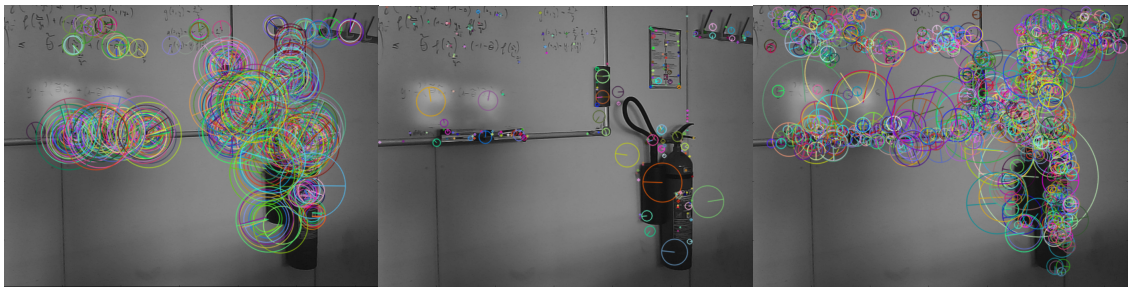


Figure 2.10: Real world data of SURF features and matches. Only the 50 best matches are drawn.



(a) ORB **(b)** SIFT **(c)** SURF

Figure 2.11: All extracted features for ORB, SIFT and SURF. The circles represent the chosen neighborhood area with the dominant orientation.

described algorithmically at first. This series of actions is often the initial step in many computer vision applications and thus a large number of feature detectors and descriptors exists. The task of the feature detector is to identify significant points/areas (i.e. edges, corners, blobs, ridges) which are then represented by the descriptor, usually in a multi-dimensional vector. Being part of the initial step, subsequent algorithms depend heavily on their feature selection. Therefore, it is desirable for the features to possess repeatability and robustness. A few of the more prominent representatives of feature detectors and descriptors are Scale-Invariant Feature Transform (*SIFT*) [36], Speeded Up Robust Features (*SURF*) [1], and Features from Accelerated Segment Test (*FAST*) [51] (the latter beeing only a detector). SIFT is a texture-based algorithm that models the image rotation, affine transformations, intensity and viewpoint changes in feature matchings (see Fig. 2.9). SURF is also texture-based, uses wavelet responses in horizontal and vertical direction for feature detection and improves its matching speed by only comparing

features with the same type of contrast (based on sign of the Laplacian) (see Fig. 2.10). FAST, in contrast to the former two, is a very fast feature-based corner detector which relies on a corner response function. A detailed comparison of the mentioned algorithms can be found in [21], which concludes that FAST has the best overall performance.

Another example is Oriented FAST and Rotated BRIEF (*ORB*) [52]. ORB is a combination of the FAST keypoint detector and the Binary Robust Independent Elementary Features (*BRIEF*) descriptor [4]) and speed wise it outperforms SIFT and SURF [30]. Due to the fact that BRIEF is not rotation-aware, ORB introduces the rBRIEF descriptor. Like BRIEF, at first it performs binary tests between pixels on a smoothed image patch, but tries to find less correlated tests. Then, the descriptor is steered according to the orientation of keypoints (called steered BRIEF). Since BRIEF is only a descriptor, FAST is needed as the feature detector and extended by an orientation component (hence oFAST). The orientation in turn is calculated by a measure of corner orientation, the so called intensity centroid [50]. An example of extracted orb features and their corresponding matches in two images is depicted in Figure 2.8 and Figure 2.11 shows a comparison of the different feature extractions. In this thesis, we choose ORB over the others because of the following advantages: ORB is at two orders of magnitude faster than SIFT and an order of magnitude faster than SURF. It is free of any licensing restrictions that apply to SIFT and SURF. Furthermore it is relatively immune to Gaussian image noise.

Independent of the method, the goal is to find matching features in different images of the same scene. The matching performance under synthetic rotations can be found in Figure 2.12, where ORB was tested against SIFT, SURF and BRIEF. Note that the standard BRIEF descriptor falls off drastically after only a few degrees. With enough of these point correspondences, we can estimate a transformation matrix $\mathbf{T}$ (see Sec. 2.5) in a least squares manner and therefore estimate the motion between two images or fuse information from one image into the other. A problem when finding correspondences can be outliers, which lead to a wrong estimate. Random Sample Consesus (*RANSAC*) is widely applied to combat this issue. RANSAC basically consists of 3 steps: (i) randomly sample a minimal subset of data points required to fit the model. (ii) solve for the model parameters using this subset. (iii) check the inliers of the whole set with the calculated parameters (consensus set). The parameters may be improved by re-estimating them with all inliers. Repeat (i) - (iii) a fixed number of times and keep the parameters of the transformation matrix $\mathbf{T}$ with the largest consensus set.

2.7 Visual Odometry (VO)

Visual Odometry (*VO*) [45] estimates the local egomotion of an agent, e.g. a robot or a vehicle, exclusively from visual input of one or more cameras. It has been an active research field for many decades and was already used as a real time navigation system on planetary rovers in the 1980s [39]. In its essence, VO detects similarities between frames in an image stream, tracks them over time and recovers a complete trajectory from the

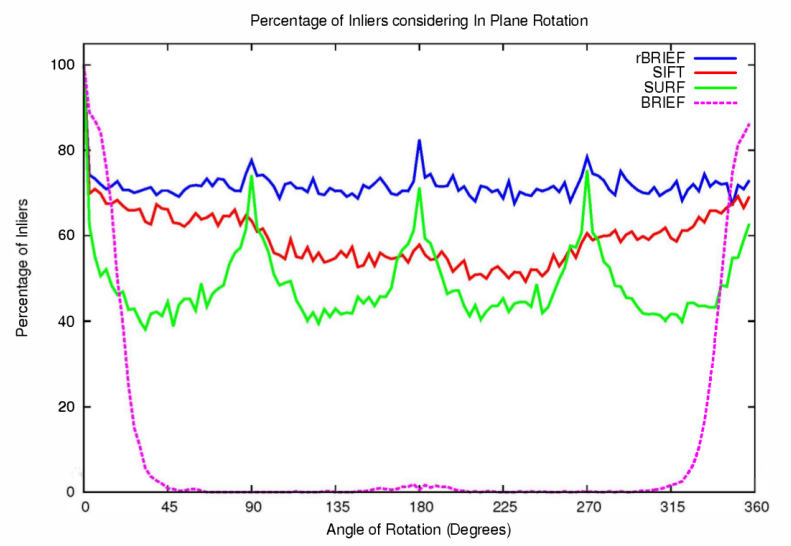


Figure 2.12: Matching performance of SIFT, SURF, BRIEF with FAST, and ORB (oFAST with rBRIEF) under synthetic rotations with Gaussian noise of 10 [52]. The standard BRIEF operator falls off severely after a few degrees.

relative transformation matrices $\mathbf{T_{i,j}}$ (see Sec. 2.5).

In appropriate conditions, VO is superior to wheel odometry. It is of uttermost importance in Global Positioning System (GPS) denied environments, like indoors or underwater. However, there are several constraints regarding the acquisition of good results: To begin with, there needs to be sufficient illumination. Moreover, the scene should be static (i.e. no moving objects) and provide enough texture. Last but not least, consecutive frames must contain enough scene overlap (motion constraint). Furthermore, even if there are nearly perfect conditions, the estimate will still contain some error. Due to the fact that VO recovers the path incrementally, this error accumulates over time and leads to drift, a difference between real and estimated trajectory. A countermeasure to reduce this drift is to locally optimize over the last n poses as was suggested in [19].

For more details on VO, the reader is referred to [55]. To further improve accuracy, a method that considers a global model, like SLAM, is needed.

2.8 Simultaneous Localization and Mapping (SLAM)

The task of SLAM is to map an unknown environment while simultaneously keeping track of the position of an agent/sensor within it [64, 65]. It is one of the fundamental problems in robotics and has a strong focus on real-time operation. SLAM can be performed with a variety of sensors, but since the increasing ubiquity and affordability of cameras (e.g. in mobile devices) there is a raised focus on Visual SLAM (*VSLAM*) [13, 14, 41]. VSLAM takes advantage of the rich information provided by one or multiple cameras. In contrast

to VO, VSLAM needs to keep a global map of the environment, regardless if the map is required per se, for place recognition at all times. If a place is revisited, i.e. the robot has recognized that this place has been seen before, a "loop closure" procedure is executed, which reduces drift in both the map and camera path. The detection of a loop closure and the integration of this new information are two crucial parts of VSLAM. Since VSLAM enforces additional constraints on the path, it is potentially more precise than a VO technique. However, it suffers heavily from wrong matches against the global map. Especially wrong loop closure can destroy the whole map and trajectory. Furthermore, due to the necessary estimation of a map, VSLAM is computationally far more complex and expensive.

Although one monocular camera is already sufficient to perform VSLAM, there are several problems: Firstly, because of its inability to provide depth information, the scale of the map and the agent's trajectory is unknown. Secondly, pure rotation can lead to failure of the algorithm, because the recorded frames do not provide enough information to triangulate new map points. Thirdly, like pure rotation, pure forward/backward translation of the camera may lead to failure due to only providing a very small baseline. Still, the interest for monocular VSLAM is ongoing due to the wide availability and because the stereo case can degenerate to this case if the observed distance in the scene is much larger than the stereo baseline

To tackle these challenges and since we primarily focus on indoor scenes, we work with an RGBD camera in this thesis.

2.9 Direct Methods

Whether it is a VO or VSLAM system, the robust camera pose estimation process can be performed in a variety of ways:

Firstly, one can directly align 3D point clouds geometrically via an Iterative Closest Point (*ICP*) algorithm [2], whereof several different variants exist [53]. Essentially, ICP minimizes the (squared) difference between a source point cloud set S and a destination point cloud set D by finding the closest point $d_i \in D$ for each given point $s_i \in S$. From these correspondences, a transformation matrix $\mathbf{T_{d,s}}$ can be calculated and applied to S. This process is repeated until some criteria has been met (e.g. max iteration steps reached or the error becomes smaller than a threshold). If the starting positions of the points are close enough, the algorithm converges. Instead of calculating the point-to-point error metric, one can also minimize the distance between a point and the tangent plane of its corresponding point (point-to-plane). Figure 2.13 illustrates the difference between the point-to-point error e_{pt2pt} and the point-to-plane error $e_{pt2plane}$. Let $\mathbf{s_i} = (s_{ix}, s_{iy}, s_{iz}, 1)^\top$ be a source point, $\mathbf{d_i} = (d_{ix}, d_{iy}, d_{iz}, 1)^\top$ its corresponding destination point and $\mathbf{n_i} = (n_{ix}, n_{iy}, n_{iz}, 0)^\top$ the unit normal vector at $\mathbf{d_i}$. Each ICP iteration tries to

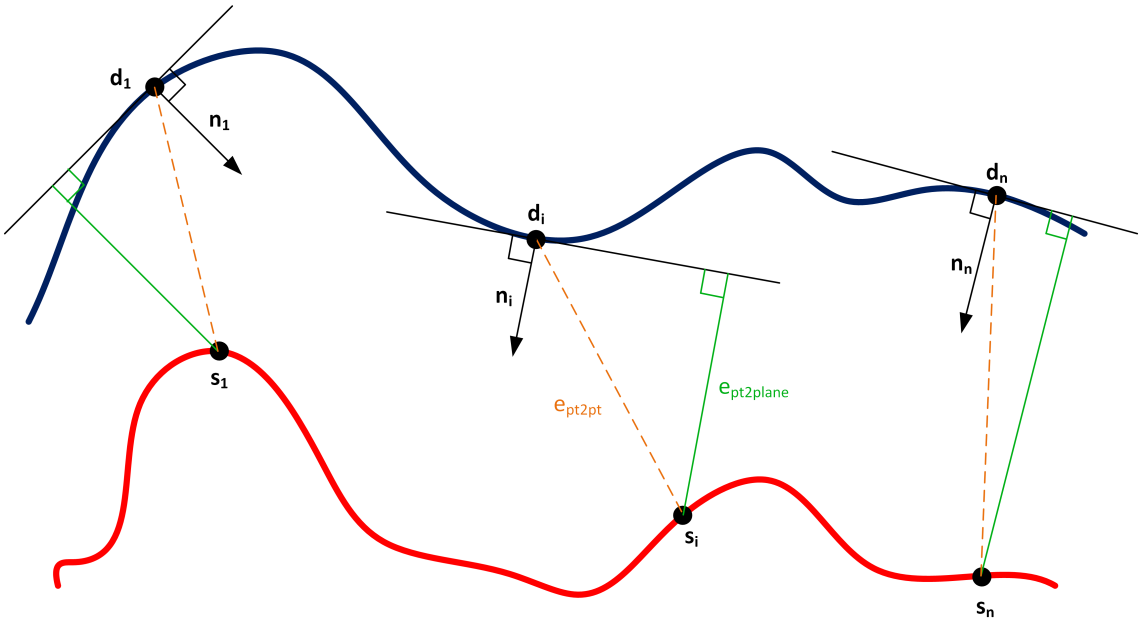


Figure 2.13: Point-to-point vs point-to-plane error: While the point-to-point error e_{pt2pt} is measured directly between the source point $\mathbf{s_i}$ and its corresponding destination point $\mathbf{d_i}$, the point-to-plane error $e_{pt2plane}$ is calculated from source point to the tangent plane of the destination point.

find the optimal rigid body transformation $\mathbf{T_{d,s,opt}}$ such that:

$$\mathbf{T}^*_{\mathbf{d,s}} = \arg\min_{\mathbf{T_{d,s}}} \sum_i ((\mathbf{T_{d,s}} \cdot \mathbf{s_i} - \mathbf{d_i})^\top \mathbf{n_i})^2 \quad , \quad e_{pt2pt} = \mathbf{T_{d,s}} \cdot \mathbf{s_i} - \mathbf{d_i} \quad . \tag{2.17}$$

The problem has no closed-form solution and needs a nonlinear least squares method, such as the Levenberg-Marquard Algorithm (*LMA*) (see Sec. 2.14). Despite the slower execution of each iteration step, Rusinkiewicz and Levoy [53] observed a significantly better convergence rate for the point-to-plane approach.

Other direct methods process the complete image information and minimize the photometric error. The photometric error is usually formulated as a pairwise alignment of 2 images: Let $\mathbf{I}_i$ be the reference image and $\mathbf{I}_j$ the other input image. Then we can warp a subset of pixel coordinates $\mathbf{x} \in \Omega_i$ into image $\mathbf{I}_j$ under the warping function $\tau(\xi_{\mathbf{j,i}}, \mathbf{x}, D_i(\mathbf{x}))$ and estimate the parameters of $\xi_{\mathbf{j,i}}$ such that the squared intensity error is minimized:

$$\xi_{\mathbf{j,i}}{}^* = \arg\min_{\xi_{\mathbf{j,i}}} \sum_{\mathbf{x}\in\Omega_0} \frac{1}{2} \left|\left|\mathbf{I}_i(\mathbf{x}) - \mathbf{I}_j(\tau(\xi_{\mathbf{j,i}}, \mathbf{x}, D_i(\mathbf{x})))\right|\right|^2 \quad . \tag{2.18}$$

This minimization of the photometric error dates back to the works of Lucas and Kanade [37] and Horn and Schunk [27], but has recently resurfaced due to availability

of high computational power [15, 32, 59]. Kerl at al. [31] proposed a combination of photometric and geometric error minimization in their visual SLAM system. To reduce the acquired drift, they adapted the idea of keyframes [25]: Every new frame is at first matched to the latest keyframe and as long as there is not too much difference, i.e. the camera has not moved to far, no drift is accumulated. Furthermore, when revisiting a previously seen region old keyframes can enforce additional constraints on the pose graph, also known as loop closures. Although this system is capable of real-time performance on a Central Processing Unit (*CPU*), it can not do so at a full resolution of 640×480.

With a similar approach, Concha et al. [6] implement a real-time capable system at full resolution by additionally applying multi-view constraints and only using a semi-dense photometric error (see Sec. 2.11 for more information on semi-dense reconstruction). Their system even outperforms Elastic Fusion [70], which in turn claims to outperform all previous baselines.

However, for direct methods to work accurately, two key assumptions must be met:

1. Brightness constancy:

$$I(x, y, t) = I(x + u, y + v, t + 1) \quad , \tag{2.19}$$

   where, x, y are image coordinates, t is the time, and (u, v) is some spacial displacement.

2. Small inter-frame motion:

$$I(x + u, y + v) \approx I(x, y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v \quad , \tag{2.20}$$

   which can be obtained through Taylor series expansion.

Together these two assumptions form the brightness constancy constraint equation:

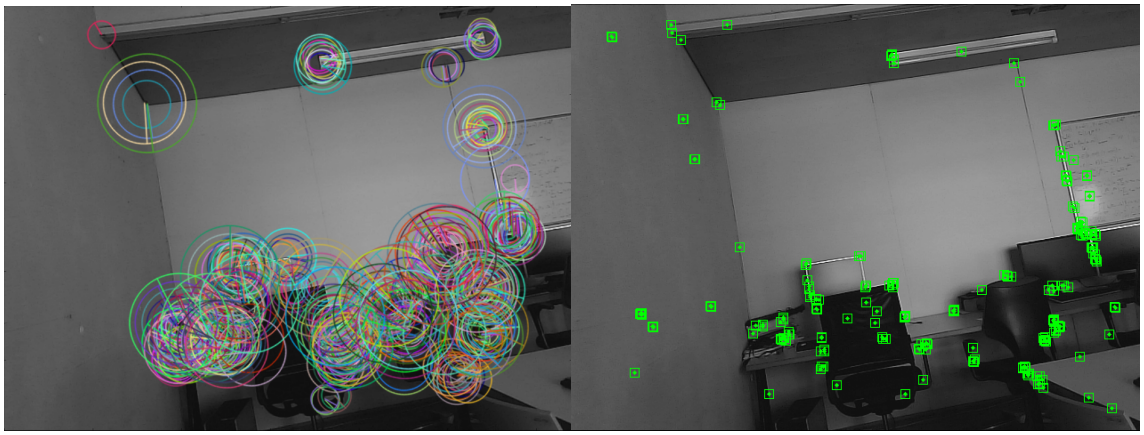
$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \quad . \tag{2.21}$$

Furthermore, direct methods need a good initialization and may get stuck in local minima. Another issue is that they cannot handle outliers very well. Their whole image alignment approach will always try to fit all of the available information.

2.10 Feature Based Methods

In contrast to direct methods, the error for feature-based methods is based on distances between the corresponding features (reprojection error).

$$e(\mathbf{x_i}, \mathbf{x_j}) = ||\mathbf{x_i} - \tau(\xi_{\mathbf{i,j}}, \mathbf{x_j}, D_j(\mathbf{x_j}))|| \quad , \tag{2.22}$$



(a) Standard ORB extraction **(b)** ORB-SLAM2 ORB extraction

Figure 2.14: Comparison of ORB and ORB-SLAM2 feature extraction: We can see that the cell division approach of ORB-SLAM2 also yields feature extractions in regions where texture is scarce, e.g. on the left wall

where $\mathbf{x_i}$, $\mathbf{x_j}$ are two corresponding feature points in frame i and j respectively and $\tau(\xi_{\mathbf{i,j}}, \mathbf{x_j}, D_j(\mathbf{x_j}))$ warps point $\mathbf{x_j}$ into frame i.

This approach can discard most of the image information, i.e. everything that is not a feature point. A major advantage of feature-based methods is their ability to overcome outliers with RANSAC-based [18] schemes. Examples for systems with this type of approach include PTAM [33], DT-SLAM [26] and more recently RGBD-SLAM [12] and ORB-SLAM2 [42]. While the former two rely solely on FAST features, RGBD-SLAM and ORB-SLAM2 extract ORB features (which, however, are also based on FAST; see Sec. 2.6). Note that RGBD-SLAM also works with SIFT or SURF features but both are outperformed by ORB in the latest version [11].

In this thesis, we substitute the trackers provided by InfiniTAM with ORB-SLAM2 which already showed very promising results regarding runtime and tracking accuracy in several comparisons [6, 42]. ORB-SLAM2 is an open-source real-time SLAM system for monocular, stereo and RGBD cameras which implements loop closure, map reuse and relocalization. It is composed of three main parallel threads: (i) Tracking: finds ORB feature matches in every camera frame and matches them with the local map. (ii) Local Mapping: optimizes the local map by performing local Bundle Adjustment (*BA*). (iii) Loop Closing: If a loop is detected, this thread corrects the accumulated drift. After the adjustment by pose-graph optimization, a fourth thread is launched to perform a global BA. The loop detection and place recognition is based on DBoW2 [20].

ORB SLAM2 extracts ORB features at 8 scale levels with a standard scale factor of 1.2. To ensure a homogeneous distribution, each scale level is divided into a grid and a minimum number of features has to be found for each grid. Figure 2.14 shows a comparison

of the normal ORB feature extraction and the ORB-SLAM2 approach. Keypoints in the tracking stage are classified as close or far, depending on their associated depth and only close keypoints are triangulated from one frame while far keypoints are only processed if they are supported by multiple views.

ORB-SLAM2 inserts keyframes very frequently and culls redundant ones afterwards. A new keyframe is inserted if all of the following conditions are met:

1. More than 20 frames have passed since the last global relocalization.
2. More than 20 frames have passed since the last keyframe insertion or local mapping is idle.
3. At least 50 keypoints are tracked in the current frame.
4. More than 10% of keypoints in the current frame are not seen by its reference keyframe, i.e. the keyframe with the most points in common.

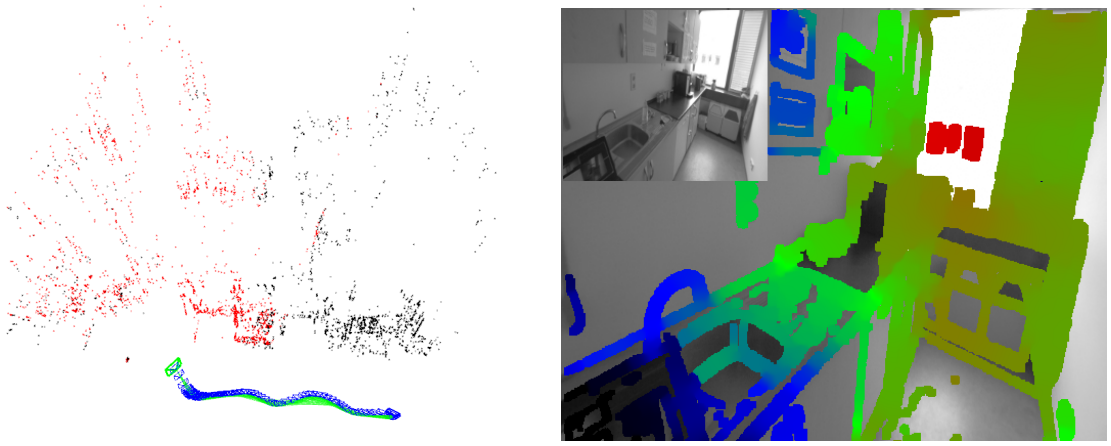
Additionally (for stereo and RGBD image data), a keyframe is added whenever the number of close keypoints drops below a certain threshold $\tau_t = 100$ and the frame could at least create $\tau_c = 70$ new close keypoints. A keyframe is deleted if 90% of its points are seen in at least 3 other keyframes in the same or finer scale.

An evaluation of the performance of ORB-SLAM2 can be found in Section 4.

2.11 SLAM and VO Systems with Dense 3D Reconstruction

If we now know the motion of a camera through the environment, we can reconstruct a 3D model of the world by using the motion and image information. This 3D model can be represented either in sparse, semi-dense or dense form. Since feature-based methods only keep information from feature points, their model usually only contains a sparse representation of the scene (see Fig. 2.15(a)). Semi-dense models, like produced by LSD-SLAM [14] (see Fig. 2.15(b)), can propagate a depth map from frame to frame which is dense in all image regions that carry information. In this manner memory can be saved, but both approaches only provide an incomplete world reconstruction.

With the increased memory capacity of computers in recent years, volumetric representations have become a powerful tool to generate dense 3D models (see Fig. 2.16) from image data [8, 28, 60, 69, 70]. They can be used in a variety of applications such as augmented reality, virtual reality, robotics and gaming. Furthermore, inexpensive RGBD cameras, foremost the Microsoft Kinect, have also enforced the positive development and led to the Kinect Fusion system [28, 43]. The Kinect Fusion system estimates the current sensor pose with an ICP algorithm (see Sec. 2.9) and integrates the aquired data via Truncated Signed Distance Function (*TSDF*) (see Sec. 2.12). In order to achieve real-time performance, the algorithms for both tracking and mapping are fully parallelized and



(a) ORB-SLAM2 **(b)** LSD-SLAM

Figure 2.15: Sparse and semi-dense models: (a) The sparse model of ORB-SLAM2. The points correspond to extracted feature points. (b) A semi-dense map of LSD-SLAM [14]. The colored points correspond to map points.

Figure 2.16: A dense 3D model created by InfiniTAM.

exploit the massively parallel processors on the Graphics Processing Unit (*GPU*). However, the Kinect Fusion system lacks the scalability for larger scenes due to memory issues (addressing and/or lack thereof). As a further work of the former, Kintinuous addresses this issue, but does not implement a full SLAM approach which leads to the accumulation of drift in the estimated trajectory.

In order to tackle the problem of a large memory footprint, research on sparse volumetric representations [5, 44, 60, 71] has sprouted. These works successfully use either octrees or hashtables to efficiently refer to allocated memory blocks.

Dai et al. proposed BundleFusion [8] that models the world with a voxel block hashing framework. In their work, they combine sparse features (SIFT) and dense geometric and photometric correspondences for trajectory estimation (explained in Sec. 2.9 and 2.10). Then they globally optimize the poses and dynamically integrate and de-integrate image data into their 3D model. Nevertheless, Bundle Fusion needs a combination of an NVIDIA GeForce GTX Titan X and a GTX Titan Black to run. Since we wanted our method to work on a portable system with a more standard hardware, their approach was disregarded. Another system which relies on strong hardware is Elastic Fusion [70], which reconstructs the 3D world as a number of circular surfels that correspond to the surfaces in the real world but it is not capable of large-scale reconstructions on standard hardware.

Therefore, we looked into alternatives and build this thesis on the framework of InfiniTAM v2 [29]. InfiniTAM implements a volumetric 3D reconstruction from depth images based on voxel block hashing and offers a very high rate for frame integration. The 3D world is also modeled using a TSDF, which we will discuss in more detail in the next section. In order to reduce memory usage, only the scene parts inside the truncation band, i.e. the voxels close to a surface, are represented densely in a usually $8 \times 8 \times 8$ block. A hash table manages these voxel block to guarantee a constant lookup time. This is called voxel block hashing and works as follows (see Fig. 2.17): From every 3D location in world coordinates a hash value can be calculated. The hash value is then looked up in the hash table, which consists of ordered entries and an excess list (also called unordered entries). Each ordered entry has exactly 1 bucket where a pointer to the voxel block array, i.e. a large array where the TSDF data of all $8 \times 8 \times 8$ blocks, is stored. Additionally, each bucket stores the 3D block position and the index to the next entry in the excess list. If multiple different 3D voxel block locations are mapped to the same bucket, the data is stored in the excess list. By only using 1 bucket for each hash value, but a larger number of overall buckets, the overall system performance improves. The reason behind this is, that by investing additional storage space into more buckets instead of a larger bucket size, the number of hash collisions is reduced [29].

InfiniTAM performs camera data integration in the same way as the original Kinect Fusion [43] system (see Sec. 2.12). The extraction of information from the implicit representation as a TSDF is done in the rendering stage by raycasting and needed for tracking and visualisation in the user interface.

InfiniTAM implements two different trackers: An ICP tracker as in [43] and a direct image alignment tracker based on color (see Sec. 2.9). Furthermore, since InfiniTAM aims to run on tablet computers, an (optional) implementation to utilize Intertial Measurement Unit (*IMU*) data exists to provide a much more accurate result.

InfiniTAM also offers a swapping algorithm to swap out memory from the GPU. However, after a lot of testing we came to the realization that there appeared inconsistencies in the 3D model when applying de-integration and swapping together. Therefore, we do not activate swapping in this thesis. To sum up, InfiniTAM provides an excellent runtime and a small memory footprint, but suffers from inaccurate tracking (compared to state of the

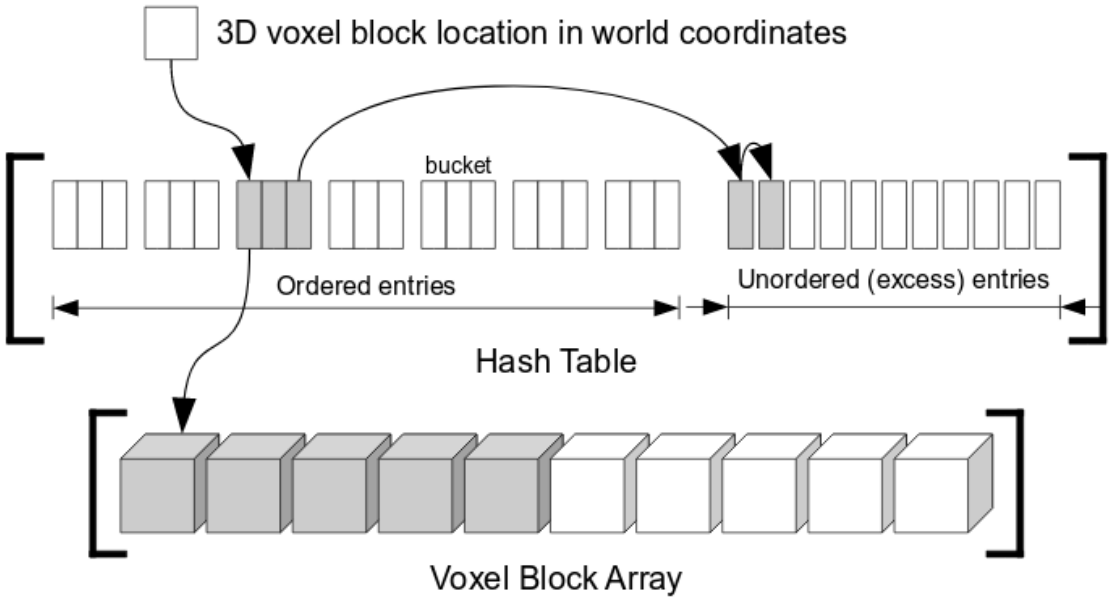


Figure 2.17: The logical hash table structure of InfiniTAM [29].

art algorithms), lack of loop-closure detection and no means of pose-graph optimization or relocalization. These are the issues we address in this thesis.

Note that during the work on this thesis, InfiniTAM v3 [49] was released, which implements a more robust camera tracking module, a camera relocaliser and relative pose optimization next to a surfel-based reconstruction approach. However, the tracking approach is still the same and loop closure is only applied to submaps, which in itself may accumulate error.

2.12 Volumetric Fusion

We model the 3D world in this thesis with the help of the TSDF. The TSDF determines the distance of each 3D voxel to its nearest surface. It can be a great tool to get a volumetric representation for many computer applications [8, 29, 43]. In order to achieve that, the entire space is divided into grids of equally sized voxels. For each voxel, the value of the Signed Distance Function (*SDF*) $\mathcal{S}(\mathbf{V})$ is calculated:

$$\mathcal{S}(\mathbf{V}) = D_t(\pi(\mathbf{V})) - Z \quad , \tag{2.23}$$

where $\mathbf{V} = (X, Y, Z)^\top$ is a voxel given by its center coordinates in its respective camera frame, $\pi(\mathbf{V})$ computes the projection of the voxel onto the depth image (see Sec. 2.3), while $D_t(\pi(\mathbf{V}))$ is the measured depth at the calculated image location at time t. Since $\pi(\mathbf{V})$ maps the voxel into the camera frame, Z is the distance between the camera and

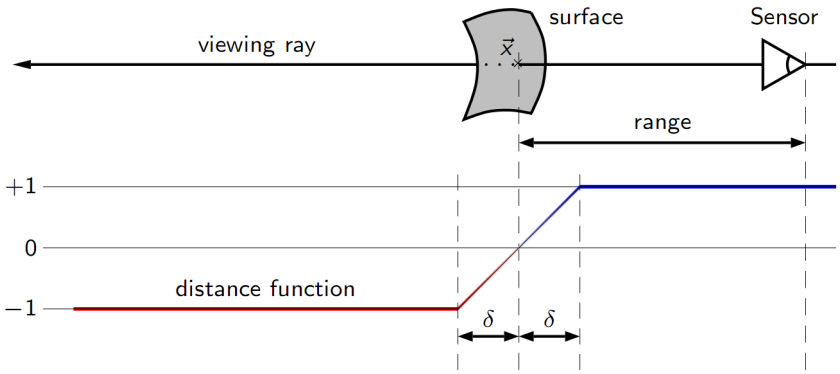


Figure 2.18: Truncated Signed Distance Function [22]: Voxels between surface and sensor are assigned positive values. Voxels behind the surface contain negative values.

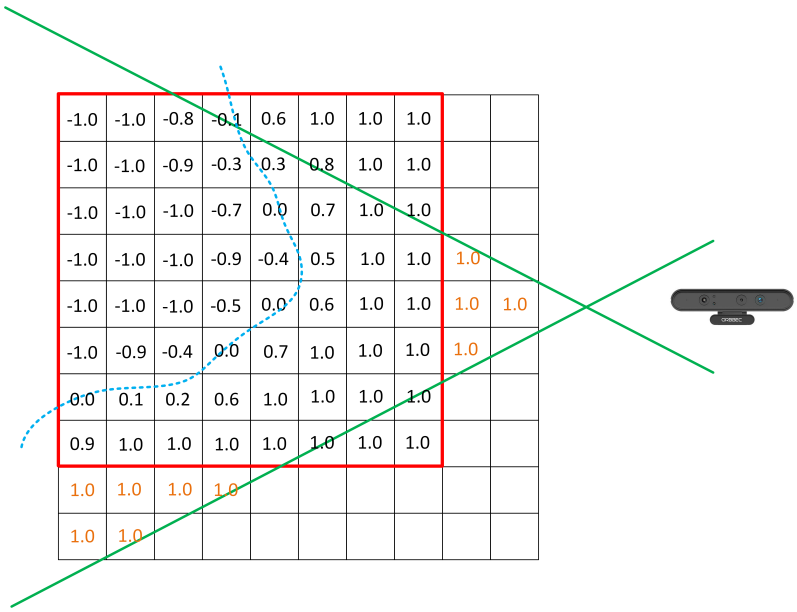


Figure 2.19: TSDF representation in InfiniTAM: The $8 \times 8 \times 8$ voxel block is depicted in red, the view frustum of the sensor in green and the surface in blue. Values above 1.0 which lie outside the voxel block of the surface do not need to be allocated (orange values). Note that only voxels with values between $-1.0 < \mathcal{T}(\mathbf{V}) < 1.0$ are assigned a weight.

voxel along the optical axis. Consequently $\mathcal{S}(\mathbf{V})$ assigns a distance value relative to the camera. As a result, positive values are assigned to voxels that reside in free space: The closer the voxel is to the surface, the smaller its value. If the voxel lies directly on the surface, its value is set to zero and behind the surface, increasing negative values are assigned. Extending this idea with an additional parameter δ, the TSDF is denoted as:

$$\mathcal{T}(\mathbf{V}) = max\left(-1, min\left(1, \frac{\mathcal{S}(\mathbf{V})}{\delta}\right)\right) \quad , \tag{2.24}$$

which allows only values between -1 and 1 to be assigned, corresponding to the distances $-\delta$ and δ respectively as depicted in Figure 2.18.

This kind of volumetric representation has several benefits: In order to save memory, any point outside the truncation band $[-\delta...\delta]$ can be set to an out-of-range value and be disregarded due to the fact that large distances are not relevant for surface reconstruction (see Fig. 2.19). Other advantages include time as well as space efficiency and its feasibility for parallel processing on GPUs [7, 67]. This makes a TSDF representation a prime candidate for real time applications. Furthermore, multiple observations of 3D points can be fused into the same model and thus decrease uncertainty or add missing information.

Combining information from different viewpoints, also known as integration, is done by weighted summation in iterative steps. At first, every voxel in the grid is initialized with some initial value for $\mathcal{D}_0(\mathbf{V})$ and $W_0(\mathbf{V}) = 0$. For every new observation i we calculate the updated TSDF:

$$\mathcal{D}_i(\mathbf{V}) = \frac{W_{i-1}(\mathbf{V})\mathcal{D}_{i-1}(\mathbf{V}) + w_i(\mathbf{V})\mathcal{T}(\mathbf{V})}{W_{i-1}(\mathbf{V}) + w_i(\mathbf{V})} \quad , \quad W_i(\mathbf{V}) = W_{i-1}(\mathbf{V}) + w_i(\mathbf{V}) \quad . \tag{2.25}$$

Note that at the first observation $\mathcal{D}_1(\mathbf{V})$ is set to the value of $\mathcal{T}(\mathbf{V})$ regardless of its initial value $\mathcal{D}_0(\mathbf{V})$ (since $W_0(\mathbf{V}) = 0$). The uncertainty weight $w_i(\mathbf{V})$ is usually set to 1, which results in an averaging of the measured TSDF observations.

Similarly, we can de-integrate an observation by reversing this operation as shown in [8]. The update step is then denoted as:

$$\mathcal{D}_i(\mathbf{V}) = \frac{W_{i-1}(\mathbf{V})\mathcal{D}_{i-1}(\mathbf{V}) - w_i(\mathbf{V})\mathcal{T}(\mathbf{V})}{W_{i-1}(\mathbf{V}) - w_i(\mathbf{V})} \quad , \quad W_i(\mathbf{V}) = W_{i-1}(\mathbf{V}) - w_i(\mathbf{V}) \quad . \tag{2.26}$$

The operations of integrating and de-integrating are symmetric, i.e. one inverts the other. Thus, an observation, if it becomes invalid or updated, can be deleted by de-integrating it from its original pose and re-integrating it with a new pose if necessary.

2.13 Plane Estimation

Since estimated 3D models from RGBD data are usually noisy and incomplete, we incorporate geometric priors to further improve the 3D model. Plane estimation is especially

applicable in man-made environments such as cities or rooms where plane-like structures dominate, e.g. walls, floors, ceilings, etc. In this so-called "Manhattan environment", plane estimation can help to eliminate noise and fill unobserved regions. Common methods for this task are RANSAC-based [56, 57]. However, RANSAC often requires a lot of computational time to estimate planes in 3D point clouds. In order to tackle this problem, region-growing algorithms have emerged. Poppinga et al. [47] proposed a fast surface extraction from 3D point clouds by exploiting the sequential data acquisition of range images on robots. Their algorithm segments 3D point cloud into regions with a common plane and extracts proper surface models. Feng et al. [16] developed an efficient plane extraction algorithm based on agglomerative hierarchical clustering for organized point clouds. Zhang et al. [72] extend the Kinect Fusion [43] framework by an online structure analysis which allows primitive shapes and heuristics to be incorporated into the 3D model. This can be used to fill holes, reduce drift (through plane structure analysis) and segment objects.

However, none of the above mentioned works are developed for implicit surface representations such as TSDFs. This was addressed in [10], where Dzitsiuk et al. build a plane estimation algorithm on top of CHISEL [34]. They efficiently compute local plane candidates in a least squares approach directly on the TSDF grid. Planes are then globally clustered by a 1-point RANSAC. Their approach proved to be successful in significantly reducing noise and recover missing information by extending planar regions.

In this thesis we utilize the Levenberg-Marquardt Algorithm (LMA) for plane estimation (see Sec 2.14).

2.14 Levenberg-Marquardt Algorithm (LMA)

The LMA [40] is an iterative technique to solve a non-linear least squares curve fitting problem. Its goal is to find the n parameters of $\boldsymbol{\beta} \in \mathbb{R}^n$ by solving:

$$\boldsymbol{\beta}^* = \arg\min_{\boldsymbol{\beta}} \mathcal{S}(\boldsymbol{\beta}) = \arg\min_{\boldsymbol{\beta}} \sum_{i=1}^{m} (y_i - \mathcal{F}(x_i, \boldsymbol{\beta}))^2 \quad , \tag{2.27}$$

where m is the set of empirical data pairs (x_i, y_i) and $\mathcal{F}(x_i, \boldsymbol{\beta})$ the function of the curve. The term $(y_i - \mathcal{F}(x_i, \boldsymbol{\beta})$ is called residual.

As with all non-linear optimization methods, the solution is found through iteration [35]. In each iteration step, we substitute $\boldsymbol{\beta}$ with its new estimate $\boldsymbol{\beta} + \boldsymbol{\delta}$:

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i + \boldsymbol{\delta} \quad . \tag{2.28}$$

Combining the measured data x_i into a vector $\mathbf{x}$ and assuming that $\boldsymbol{\delta}$ is small, the Taylor series expansion leads to the linear approximation:

$$\mathcal{F}(\mathbf{x}, \boldsymbol{\beta} + \boldsymbol{\delta}) = \mathcal{F}(\boldsymbol{\beta} + \boldsymbol{\delta}) \approx \mathcal{F}(\boldsymbol{\beta}) + \mathbf{J}\boldsymbol{\delta} \quad , \tag{2.29}$$

where

$$\mathbf{J} = \frac{\partial \mathcal{F}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} \tag{2.30}$$

is the Jacobian matrix.

Due to the knowledge that the sum of square deviations $\mathcal{S}(\boldsymbol{\beta})$ has at its minimum a zero gradient, we calculate:

$$\begin{aligned}
\mathcal{S}(\boldsymbol{\beta}+\boldsymbol{\delta}) &\approx ||\mathbf{y} - \mathcal{F}(\boldsymbol{\beta}) - \mathbf{J}\boldsymbol{\delta}||^2 \\
&= [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta}) - \mathbf{J}\boldsymbol{\delta}]^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta}) - \mathbf{J}\boldsymbol{\delta}] \\
&= [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})] - [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]^\top \mathbf{J}\boldsymbol{\delta} - (\mathbf{J}\boldsymbol{\delta})^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})] + \boldsymbol{\delta}^\top \mathbf{J}^\top \mathbf{J}\boldsymbol{\delta} \\
&= [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})] - 2\,[\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]^\top \mathbf{J}\boldsymbol{\delta} + \boldsymbol{\delta}^\top \mathbf{J}^\top \mathbf{J}\boldsymbol{\delta} \quad .
\end{aligned} \tag{2.31}$$

Taking the derivative of Equation 2.31 with respect to $\boldsymbol{\delta}$ and setting its result to zero yields:

$$\begin{aligned}
-2\,[\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]^\top \mathbf{J} + 2\boldsymbol{\delta}^\top (\mathbf{J}^\top \mathbf{J}) &= 0 \\
2\boldsymbol{\delta}^\top (\mathbf{J}^\top \mathbf{J}) &= 2\,[\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]^\top \mathbf{J} \\
(\mathbf{J}^\top \mathbf{J})\boldsymbol{\delta} &= \mathbf{J}^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})] \quad ,
\end{aligned} \tag{2.32}$$

where $\mathbf{J}^\top\mathbf{J}$ is the approximate Hessian matrix, i.e. the matrix of second order derivatives. The acquired set of linear equations can now be solved for $\boldsymbol{\delta}$ in a least squares manner of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{A} = (\mathbf{J}^\top\mathbf{J})$, $x = \boldsymbol{\delta}$ and $\mathbf{b} = \mathbf{J}^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]$.

Levenberg introduces a slight variation to Equation 2.32 by adding a dampening parameter λ which is adapted in every iteration:

$$(\mathbf{J}^\top \mathbf{J} + \lambda \mathbf{I})\boldsymbol{\delta} = \mathbf{J}^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})] \quad . \tag{2.33}$$

If the current solution is close to the final solution, i.e. the reduction of $\mathcal{S}$ is rapid, λ can be assigned a smaller value and therefore bring the algorithm closer to the Gauss-Newton method. On the other hand, if the residual reduction is insufficient, λ can be increased to bring it closer to the gradient-descent method. The algorithm terminates if either the step size or the residual reduction becomes to small. Marquardt further improved the algorithm by scaling each component of the gradient according to the curvature, which resulted in the following equation for the LMA:

$$(\mathbf{J}^\top \mathbf{J} + \lambda \cdot diag(\mathbf{J}^\top \mathbf{J}))\boldsymbol{\delta} = \mathbf{J}^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})] \quad . \tag{2.34}$$

At the start of the LMA, we have to provide a guess for the parameter vector $\boldsymbol{\beta}$. Provided that there is only one minimum, an arbitrary initialization will suffice. In cases of several minima, the global minimum can only be found if the initial guess is already close to the solution.

3 Method

Contents

3.1 Method Overview

In this thesis we present an approach for real-time dense 3D model reconstruction from RGBD data by extending the volumetric fusion framework of InfiniTAM v2 [29] and combining it with the accurate SLAM trajectory estimation of ORB-SLAM2 [42]. We add novel techniques to InfiniTAM in order to support global model updates that arise from e.g. loop closure. Our main contributions include:

- Implementation of a de-integration method, which allows to refine and alter the 3D model. This is especially important when large changes in the estimated trajectory occur, e.g. in the case of a loop closure detection.
- Keyframe-based depth fusion: Instead of integrating every frame, we fuse information of similar frames into one keyframe (selected by the ORB-SLAM2 algorithm) to decrease computational time. Although this might not provide a huge improvement in the integration step because we still have to transform all points into the keyframe, it significantly reduces the runtime for de-integration. Furthermore, we can stop integrating additional frames when a certain threshold of observations has been reached for a given keyframe.

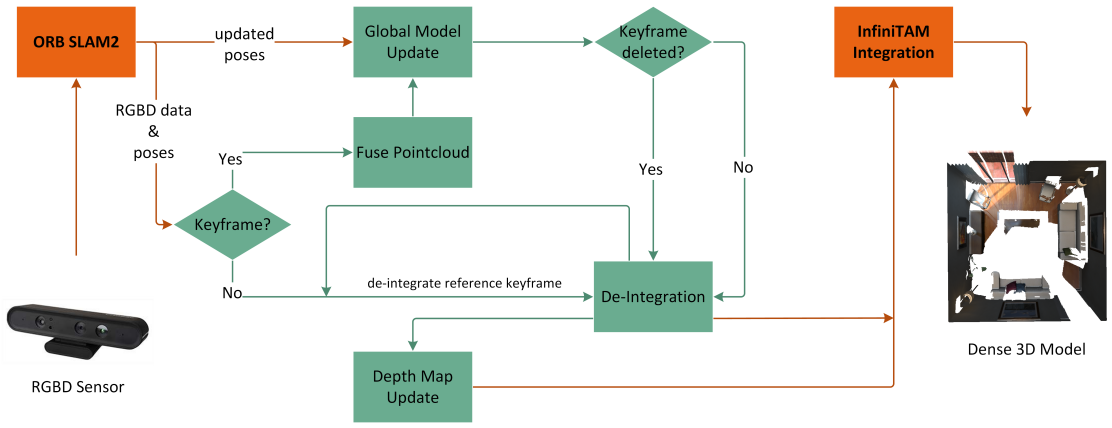


Figure 3.1: Globally consistent real-time dense 3D model update: Our system takes as input the estimated poses from ORB-SLAM2 and the RGBD data from the sensor. If the current frame is not a keyframe, we update the corresponding depth map. Otherwise we fuse its depth map with the pointcloud and update the global model. In case that a keyframe has been deleted, we de-integrate it and fuse its information into the next best (closest) keyframe. If not, we de-integrate the frame with its old pose and re-integrate it with its new pose.

- A global model update which can delete and merge keyframes in retrospect. The ORB-SLAM2 system continuously refines the estimated poses and whenever a new keyframe is selected, we verify the integrated poses from the model with the updated poses. If a significant change occurs, we update our 3D model in real-time.

- A method to estimate planes in a 3D scene by using the Levenberg-Marquardt Algorithm. This optional step enables us to significantly reduce noise in plane reconstructions and even fill holes in the 3D model where no observations have been made.

Figure 3.1 depicts the work flow of our system. Note that our contributions are depicted in green. We feed the RGBD data (either acquired live via sensor, or from a dataset) into the ORB-SLAM2 system and receive the estimated poses for every frame. Then the provided pose estimate is used in the fusion step to integrate new data into the existing 3D world, which we model as described in Section 2.12. Afterwards, we update our depth maps and adjust the global model if necessary, before we integrate the new information into the volumetric representation of InfiniTAM. (see Sec. 3.5). Finally we add a function to estimate planes, which can optionally be called from the user interface. Further features include reading and writing of pose-files which support keyframe enumerations, several evaluation options like reading from ground-truth and alteration of almost any parameter through a configuration file.

We explain our depth map updates, global model updates and plane estimation in detail in the following sections.

3.2 Combining ORB-SLAM2 and InfiniTAM

The system is implemented in C++ utilizing CUDA. We extend the existing InfiniTAM framework [48] in the following manner: A new class `ITMORBTracker` which derives from the standard `ITMTracker` is introduced as the interface to the ORB-SLAM2 algorithm. We add several functionalities to the `ITMMainEngine` class which are all called from within the `ProcessFrame()` method, like `deleteFrame()`, `insertFrame()` or `GlobalModelUpdate()`.

For a fast interaction, we integrate the whole ORB-SLAM2 system into the InfiniTAM code. As stated above, the `ITMORBTracker` class provides the two necessary functions to get all the essential information from ORB-SLAM2, namely `TrackCamera()` (adopted from the original InfiniTAM) and `GetUpdatedKFPoses()`. The former takes an `ITMTrackingState` object as one of two arguments, which we extend by the following variables:

- `bool trackLost`: indicates if track was lost (no pose → no model update)
- `bool isKeyFrame`: indicates if the current frame was selected as keyframe
- `int refKF`: refers to the keyframe the frame is associated with (refers to itself if it is a keyframe)
- `std::vector<int> connectedKFs`: list of keyframes which are connected to the current keyframe (used for point cloud fusion and culling; optional)
- `ITMPose pose_d_relative`: relative position of the frame to its keyframe

All these informations are extracted via the ORB-SLAM2 function `TrackRGBD` which is only slightly altered to pipe all the necessary data through. The `GetUpdatedKFPoses()` function iterates through all pose estimations the ORB-SLAM2 system currently holds, identifies deleted entries and returns the updated poses.

3.3 Integration and De-integration

We describe the mathematical concepts for integration and de-integration in a volumetric fusion method using a TSDF in Section 2.11. As part of the fusion step, integration is already implemented in the InfiniTAM framework (see Sec. 2.12). In order for de-integration to work, a new method `DeleteFrame()` is implemented in the `ITMDenseMapper` class: At first it calls the `sceneRecoEngine` function `AllocateSceneFromDepth()` with the parameter `onlyUpdateVisibleList` set to true, which means that no new voxelblocks need to be allocated in the hashtable. We substitute the original `setToType3()` function with our new `setToType0()` function that sets all previously seen voxel blocks to "not visible". If this is not done, some voxels might still have values left after de-integrating. This is because the original method would also check for visible blocks in the last seen frame, but as we de-integrate images, the order of seen frames changes.

Then, we call `DeintegrateFromScene()`, another `SceneRecoEngine` function. The steps are identical to `IntegrateIntoScene()` up to the point where `deintegrate()` in DeviceAgnostic/ITMSceneReconstructionEngine.h is executed. Here the integration step gets reversed by using subtraction (see Eq. 3.3). Additionally, a few checks have to be made to make sure that the right values are written back to the voxel (i.e. color vector does not become negative and sdf value is between -1.0 and 1.0). We propagate these checks also into the integration step.

Re-Integration can be performed by using `ProcessFrame()` as usual. Note that `AllocateSceneFromDepth()` now uses the same `setToType0()` function.

3.4 Depth Map Fusion in Keyframes

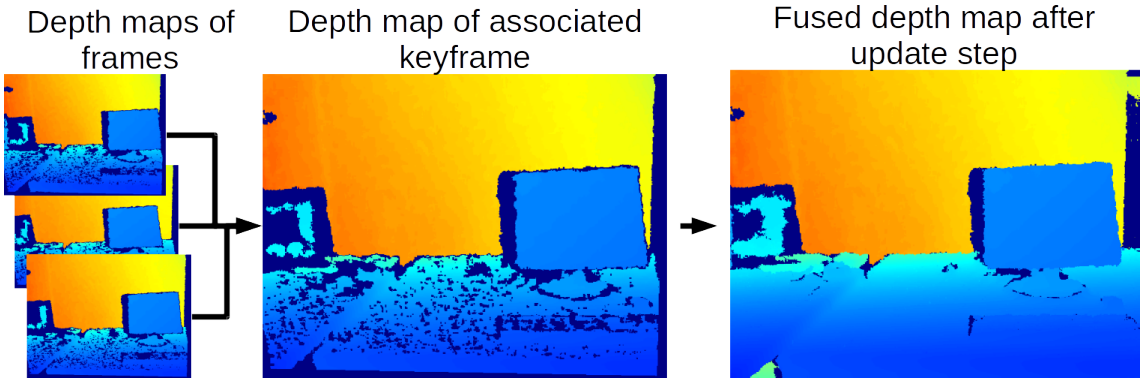


Figure 3.2: Our depth map update complements and smooths the depth map of the keyframe.

The idea behind fusing the depth maps of frames into their reference keyframe is to create a system that is able to adapt to global changes within real-time. Without the depth map fusion each frame would have to be re-integrated separately when a model update is induced, while our technique re-integrates only the fused depth maps of keyframes. Since on average only every 10th frame is selected as keyframe, this reduces the amount of operations by a factor of 10. As the ORB-SLAM2 algorithm already identifies keyframes and associates them with other frames we only have to extend the fusion step.

In order to fuse information from a frame into its reference keyframe we introduce the `DepthMap` and `DepthMapEngine` classes. The `DepthMap` class represents a keyframe and does not provide any significant functions but rather contains all relevant information: the (fused) depth map, the rgb image, the absolute camera position of the keyframe in the 3D world (`pose_d`), the current keyframe ID, a vector of keyframe IDs that are connected to this current keyframe and a possible pointcloud.

Fig. 3.1 depicts our process flow: At the beginning, when we register the first frame as keyframe, all information, except the pointcloud, is provided by the ORB-SLAM2 tracker and the depth map is the recorded depth measurement of the frame. The keyframe is

then integrated into the 3D world model. Subsequent frames, which are not chosen as keyframes, will update the depth map data of their associated keyframe (see Fig. 3.2). This is done via the `DepthMapEngine` class, a CUDA - CPU hybrid implementation to combine information from several images into a single depth map. The update step is closely related to the volumetric fusion integration step presented in (2.25) :

$$\begin{aligned} D_{KF,i}(\mathbf{x}') &= \frac{W_{i-1}(\mathbf{x}')D_{KF,i-1}(\mathbf{x}') + w_i(\mathbf{x})Z'}{W_{i-1}(\mathbf{x}') + w_i(\mathbf{x})} \quad , \\ W_i(\mathbf{x}') &= W_{i-1}(\mathbf{x}') + w_i(\mathbf{x}) \quad , \end{aligned} \tag{3.1}$$

where $\mathbf{x}' = \tau(\xi_{KF,c}, \mathbf{x}, D_c(\mathbf{x}))$ is the reprojected pixel position, $Z' = [\mathbf{T}_{KF,c}\pi^{-1}(\mathbf{x}, D_c(\mathbf{x}))]_z$ is the z-coordinate of the transformed point, D_c the depth map of the current frame, $\mathbf{T}_{KF,c}$ the transformation from current frame to keyframe and the weight $w_i(\mathbf{x})$ is set equal to 1, which leads to an averaging of the depth values. Please note that we truncate $\mathbf{x}'$ to always work on integer pixel positions. The difference to the volumetric fusion (2.25) step is that we update the depth map of the keyframe $D_{KF,i}$ instead of the TSDF values in the model.

In order to not lose any information, we store unfused points in a pointcloud. The pointcloud is represented as a vector, where each entry corresponds to a 3D point, which is transformed into the keyframe coordinates but could not be added to the depth map. Points are not fused into the depth map and added to the pointcloud when either of the following conditions arise: (i) points transform to out of boundaries, i.e. the x and/or y coordinate are negative or larger than the image size, (ii) the depth difference is too large, (iii) a transformed point maps to an invalid depth measurement. The provided depth map may contain invalid values ($D_{KF}(\mathbf{x}) = 0$) due to reflective, absorptive or transparent surfaces or occlusion of the IR pattern. In this case it can either be added to the pointcloud or overwrite the invalid measurement (controlled via the `use_invalid_depths` parameter in the configuration file). We kept `use_invalid_depths` set to true in order to gather more information in the keyframes and close invalid pixel holes. However, this might lead to depth inconsistencies as it cannot check the depth difference. This depth difference of (ii) can be described as:

$$\left| \frac{1}{D_{KF}(\mathbf{x}')} - \frac{1}{Z'} \right| < \Theta_\tau \quad , \tag{3.2}$$

where $D_{KF}(\mathbf{x})$ is the entry in the keyframe depth map at position $\mathbf{x} = (x, y)^\top$, Z' is the z-coordinate of the transformed point and Θ_τ is some threshold value. This is especially needed on edges in the scene (e.g. table edge vs floor), where it might occur that a point far behind the edge in the new frame would transform onto the edge in the keyframe, e.g. due to rounding. Algorithm 1 depicts the whole depth map update process.

After this procedure, all relevant information resides within the `DepthMap` object and the processed frame can be disregarded. We choose to not update the RGB data which might yield better coloring results but would also increase runtime. Furthermore, unlike

Algorithm 1: Depth Map Update

$w(\mathbf{x}) \leftarrow w(\mathbf{x})_{init} = 1$
forall *pixels* $\mathbf{x} \in D_c$ **do**
  $\mathbf{x}' \leftarrow \tau(\xi_{\mathbf{i,j}}, \mathbf{x}, D_c(\mathbf{x}))$
  $\mathbf{X}' \leftarrow \mathbf{T}_{KF,c}\pi^{-1}(\mathbf{x}, D_c(\mathbf{x}))$
  if $\mathbf{x}' \notin D_{KF}$ **then**
    `// point is outside of reference keyframe`
    `AddToPointCloud(`$\mathbf{X}'$`)`
  else if $D_{KF}(\mathbf{x}') = 0$ **then**
    `// invalid depth measurement`
    if *use_invalid_depths* **then**
      `// overwrite invalid depth measurement`
      $D_{KF}(\mathbf{x}') \leftarrow Z'$
    else
      `AddToPointCloud(`$\mathbf{X}'$`)`
  else if $\left|\frac{1}{D_{KF}(\mathbf{x}')} - \frac{1}{Z'}\right| < \Theta_\tau$ **then**
    `// update reference depth map`
    $D_{KF}(\mathbf{x}') \leftarrow \frac{W(\mathbf{x}')D_{KF}(\mathbf{x}')+w(\mathbf{x})Z'}{W(\mathbf{x}')+w(\mathbf{x})}$
    $W(\mathbf{x}') \leftarrow W(\mathbf{x}') + w(\mathbf{x})$
  else
    `// depth difference too large`
    `AddToPointCloud(`$\mathbf{X}'$`)`

depth information, where invalid measurements can occur, color information is available for every pixel and it is therefore sufficient to color the whole 3D model by just using the RGB image of the keyframe.

There is one catch if the current frame is not a keyframe: Since after every new frame the 3D world model is updated, we need to de-integrated the depth map of the reference keyframe ID first, then update it with the new depth map and finally re-integrate it. We speed up this process by setting the `fast_mode` parameter in the configuration file: If `fast_mode` is activated, frames will only be integrated into the model if a new keyframe is processed, i.e. new frames will only be integrated in the `DepthMap` object and not the volumetric model. A downside to this is that less visual feedback is provided.

On the other side, if the current frame is a subsequent keyframe (so the very first keyframe is already registered in the 3D world model), we try to fuse the pointclouds into the depth map of the new keyframe KF_{new}. For this case, the `DepthMapEngine` class provides the `fusePCS()` function. It requires as input the depth map of the new keyframe, its `DepthMap` object and a list of all already processed `DepthMap` objects. The `frames_to_check` parameter regulates how many previous pointclouds should be checked

(reminder: every `DepthMap` has its own pointcloud). Every homogeneous 3D point $\tilde{\mathbf{X}}$ of the pointcloud is then transformed from its keyframe KF into the new keyframe KF_{new} by the transformation matrix $\mathbf{T}_{KF_{new},KF}$:

$$\tilde{\mathbf{X}}' = \mathbf{T}_{KF_{new},KF} \cdot \tilde{\mathbf{X}} \quad , \quad \mathbf{T}_{KF_{new},KF} = \mathbf{T}_{KF_{new},w} \cdot \mathbf{T}_{w,KF} \quad , \tag{3.3}$$

where $\mathbf{T}_{KF_{new},w}$ is the transformation matrix from world coordinates into the current keyframe, and $\mathbf{T}_{w,KF} = \mathbf{T}_{KF,w}^{-1}$ the inverse of the transformation matrix from world coordinates into the keyframe KF. We now apply the mapping $\pi(\mathbf{X}')$ (2.3) to get the 2D image coordinates of the current keyframe the 3D point maps to. Finally, we can again calculate the update step (3.1) if the mapped point satisfies our previously mentioned conditions. Every point we are able to map in this manner is deleted from its pointcloud and if the number of points within a pointcloud falls below a certain threshold Θ_{pc}, we discard the whole pointcloud. After this process, we use the depth map as a complemented and smoothed depth image (see Fig. 3.2), which we integrate into the InfiniTAM model.

Both `update()` and `fusePCS()` can be parallelized on the GPU via CUDA specific code. However, since saving all `DepthMaps` with their respective point clouds on the GPU is infeasible, we transfer only the pointcloud data to the GPU for the `fusePCS()` function. As a result, the increased processing speed of the operation is overshadowed by the time it takes for the memory transfer. Nevertheless, we successfully implemented a CUDA version of the `update()` function since here we only need the current keyframe and the frame that is currently processed in the GPU memory. Note that in a few cases we run into the problem of collision when two or more points in the frame correspond to the same coordinates in the keyframe. In this case, only 1 point will be integrated and the other points are lost. However, this loss of information can be tolerated for the sake of speed because no atomic operations are required.

3.5 Global Model Update

As stated above, every time a new keyframe is detected, all currently integrated poses are re-evaluated with the ORB-SLAM2 system, which continuously refines estimated poses and if a significant change occurred, i.e. the translation and/or the rotation are above a threshold, the model will be updated. This is done by the `updateGlobalModel()` or the `updateGlobalModelCulling()` function, which is called after a new keyframe has been integrated. In the first function, the `GetUpdatedKFPoses()` of the `ITMORBTracker` is called and returns all the (updated) poses for each keyframe. If a change is big enough, the old keyframe will be de-integrated at its old pose and re-integrated at the updated pose.

ORB-SLAM2 follows the policy to insert many keyframes and cull them later. This requires the `updateGlobalModelCulling()` function which can delete and re-integrate keyframes in retrospect into the 3D model. For this case, the `ITMORBTracker` can provide an additional list with the IDs of every deleted keyframe since the last check. The infor-

mation of the deleted keyframe KF_{delete} is also deleted from the 3D model, then fused into its closest keyframe $KF_{closest}$, which is de-integrated from the 3D model beforehand, via the `DepthMapEngine` function `fuseKeyFrame()` and finally re-integrated. This process is identical to the update step of the `DepthMap` in Section 3.4 with the exception that the weight $w_i(\mathbf{x}')$ is now set to the number of points which have been fused.

3.6 Plane Estimation

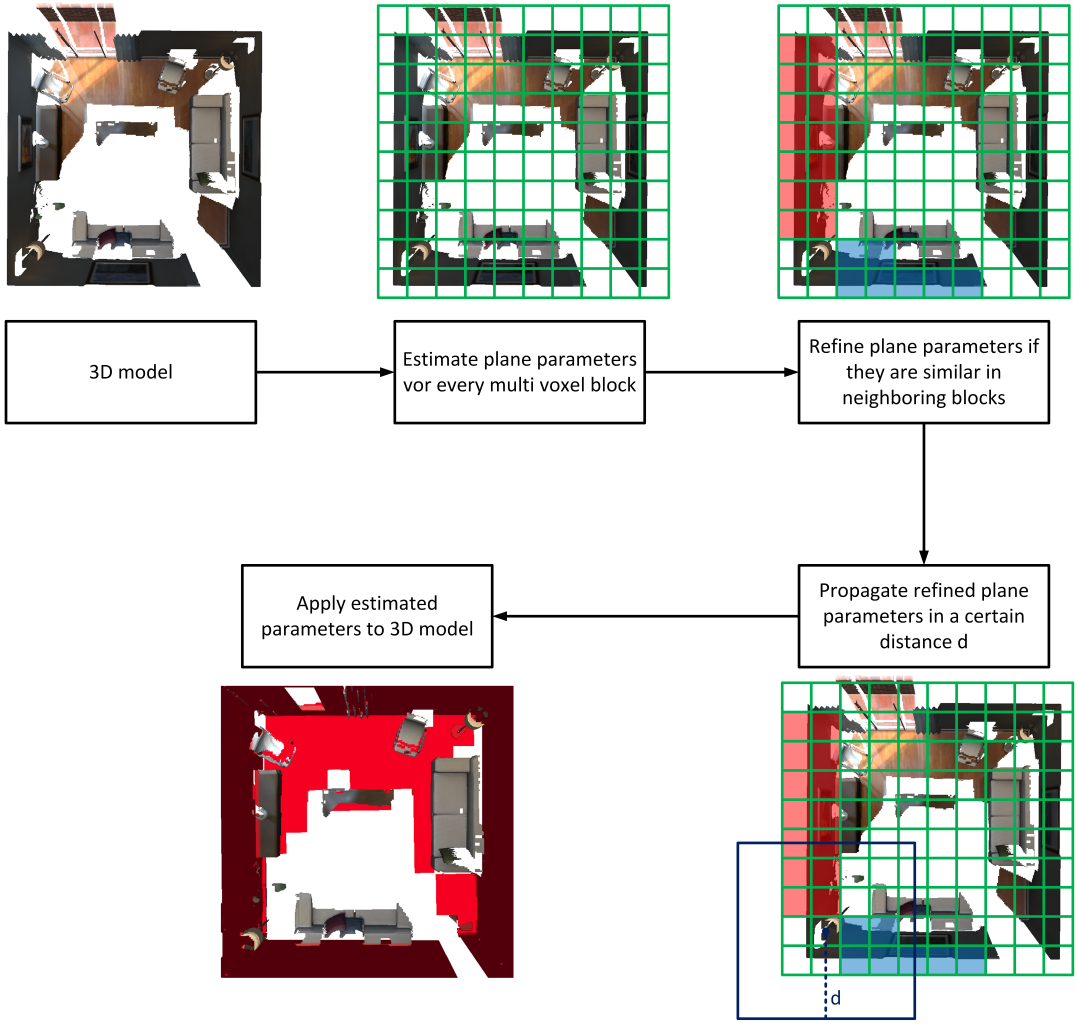


Figure 3.3: Plane estimation flow chart: At first we estimate plane parameters for every multi voxel block in our 3D model. Then we merge similar neighboring planes to refine them (depicted in red and blue). After the refinement, we propagate the plane parameters within a certain distance. Finally, we apply the estimated parameters to the model (estimated planes are depicted in red).

After we have acquired a 3D model by utilizing the previously explained methods,

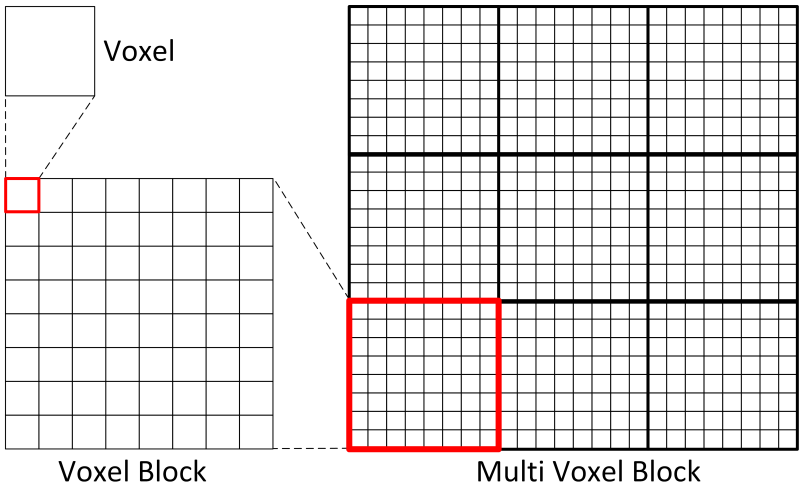


Figure 3.4: A voxel is the smallest possible representation in our method. A voxel block consists of $8 \times 8 \times 8 = 256$ voxels and is always allocated as a whole. We introduce the multi voxel block as a multiple of voxel blocks, e.g. $3 \times 3 \times 3 = 27$, to estimate our plane parameters in a wider volume.

the `EstimatePlanes()` function can be called manually from within the InfiniTAM User Interface (UI). This method is inspired by [10] and discussed in Section 2.13.

Essentially there are 3 steps in our plane estimation algorithm (see Fig. 3.3): At first, we calculate the plane parameters for every multi voxel block. The size of a multi voxel block within the scope of our algorithm is a multiple of the `SDF_BLOCK_SIZE` defined in InfiniTAM (see Fig. 3.4). Then, these parameters are refined and propagated through their block's neighboring blocks. Finally, we apply the best fitting parameters to the 3D model.

3.6.1 Estimating Plane Parameters

In the first step we estimate the best fitting plane for each multi voxel block in parallel on the GPU. We do so, by using the Levenberg-Marquardt Algorithm (see Sec. 2.14) and solving:

$$(\mathbf{J}^\top \mathbf{J} + \lambda \cdot diag(\mathbf{J}^\top \mathbf{J}))\boldsymbol{\delta} = \mathbf{J}^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})] \quad , \quad \mathbf{y} = (\mathcal{D}(\mathbf{X}_1)\delta, \mathcal{D}(\mathbf{X}_2)\delta, \ldots, \mathcal{D}(\mathbf{X}_n)\delta)^\top \quad , \tag{3.4}$$

where $\mathbf{J}$ is the Jacobian matrix and $\mathcal{F}(\boldsymbol{\beta})$ is the vector of SDFs from point to plane. The data vector $\mathbf{y}$ consists of the updated TSDF values $\mathcal{D}(\mathbf{X}_k)$ (see Sec. 2.24) for every voxel coordinate $\mathbf{X}_k = (X, Y, Z)^\top$ multiplied by the truncation band value δ. In this manner, every entry of $\mathbf{y}$ with a TSDF value below 1 can be compared to $\mathcal{F}(\boldsymbol{\beta})$. The parameter vector $\boldsymbol{\beta}$ we need to find, is the one of the 3D plane in Hessian normal form $\boldsymbol{\beta} = (n_1, n_2, n_3, d)^\top$, where $\mathbf{n} = (n_1, n_2, n_3)^\top$ is the plane normal of unit length, i.e. $|\mathbf{n}| = 1$ and d is the distance to the coordinate center.

As depicted in Algorithm 2, we start with the arbitrary guess $\boldsymbol{\beta} = (0.0, 0.0, -1.0, d)^\top$ and set d to the length of the first voxel vector in the block, i.e. to the length of its world coordinates. The initial value for λ is 0.01 and the maximum number of iteration steps $steps_{max}$ is set to 50. Then we calculate the error for this first guess before we start with the LMA iteration. The `CalculateError()` function solves the error equation e_k for each voxel k and returns the average value:

$$e_k = r_k^2 \cdot w(r_k) \quad , \tag{3.5}$$

with

$$r_k = \boldsymbol{\beta}^\top \tilde{\mathbf{X}}_k - \mathcal{D}(\mathbf{X}_k)\delta \quad , \tag{3.6}$$

$$w(r_k) = \begin{cases} \frac{1}{2} r_k^2, & \text{for } |r_k| \leq \Theta_H \\ \Theta_H(|r_k| - \frac{1}{2}\Theta_H), & \text{otherwise} \end{cases} \quad , \tag{3.7}$$

where $\tilde{\mathbf{X}}_k = (X_k, Y_k, Z_k, 1)^\top$ are the homogeneous coordinates of voxel k and $\mathcal{D}(\mathbf{X}_k)$ is the updated TSDF value of the same voxel. The weighting function $w(r_k)$ is based on the robust Huber loss function (with threshold parameter Θ_H) and gives less weight to outliers. We exclude voxels with a TSDF value which exceeds the `sdfThreshold` parameter (set in the configuration file) and voxels with no observation value, i.e. with weight zero. If not enough voxels with these criteria can be found, we stop the algorithm here and do not return an estimated plane. In our experiments, we found that a threshold of $\theta_V = 128$) works well.

In every iteration cycle (Alg. 2, l. 3) we calculate the updated values for $\mathbf{H}$ and $\mathbf{b}$: The matrix $\mathbf{H}$ here is the approximated Hessian matrix $\mathbf{J}^\top\mathbf{J}$ scaled with the weight $w(r_k)$ from (3.7) and $\mathbf{J}$ is the Jacobian matrix of $\mathcal{F}$ with respect to $\boldsymbol{\beta}$:

$$\mathbf{J} = \frac{\partial \mathcal{F}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} \quad , \quad \mathcal{F}_k(\boldsymbol{\beta}) = \boldsymbol{\beta}^\top \tilde{\mathbf{X}}_k = n_1 X_k + n_2 Y_k + n_3 Z_k + d \quad , \tag{3.8}$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathcal{F}_1(\boldsymbol{\beta})}{\partial n_1} & \frac{\partial \mathcal{F}_1(\boldsymbol{\beta})}{\partial n_2} & \frac{\partial \mathcal{F}_1(\boldsymbol{\beta})}{\partial n_3} & \frac{\partial \mathcal{F}_1(\boldsymbol{\beta})}{\partial d} \\ \frac{\partial \mathcal{F}_2(\boldsymbol{\beta})}{\partial n_1} & \frac{\partial \mathcal{F}_2(\boldsymbol{\beta})}{\partial n_2} & \frac{\partial \mathcal{F}_2(\boldsymbol{\beta})}{\partial n_3} & \frac{\partial \mathcal{F}_2(\boldsymbol{\beta})}{\partial d} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \mathcal{F}_n(\boldsymbol{\beta})}{\partial n_1} & \frac{\partial \mathcal{F}_n(\boldsymbol{\beta})}{\partial n_2} & \frac{\partial \mathcal{F}_n(\boldsymbol{\beta})}{\partial n_3} & \frac{\partial \mathcal{F}_n(\boldsymbol{\beta})}{\partial d} \end{bmatrix} = \begin{bmatrix} X_1 & Y_1 & Z_1 & 1 \\ X_2 & Y_2 & Z_2 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ X_n & Y_n & Z_n & 1 \end{bmatrix} \quad . \tag{3.9}$$

We can see that the row entries of the Jacobian equal the homogeneous coordinates of the voxel. To obtain $\mathbf{H}$ we compute $(\mathbf{J}_k^\top \mathbf{J}_k) w(r_k)$ for every voxel k individually and add the results:

$$\mathbf{H} = \sum_{k=1}^{n} (\mathbf{J}_k^\top \mathbf{J}_k) w(r_k) \quad , \tag{3.10}$$

$$(\mathbf{J}_k^\top \mathbf{J}_k) w(r_k) = (\tilde{\mathbf{X}}_k \tilde{\mathbf{X}}_k{}^\top) w(r_k) = \begin{bmatrix} X_k^2 & X_k Y_k & X_k Z_k & X_k \\ X_k Y_k & Y_k^2 & Y_k Z_k & Y_k \\ X_k Z_k & Y_k Z_k & Z_k^2 & Z_k \\ X_k & Y_k & Z_k & 1 \end{bmatrix} w(r_k) \quad . \tag{3.11}$$

Similarly, we compute $\mathbf{b} = \mathbf{J}^\top [\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]$ and scale it with $w(r_k)$:

$$\mathbf{b} = -\sum_{k=1}^{n} \mathbf{J}_k^\top r_k w(r_k) \quad , \tag{3.12}$$

Since the residual is in the form of $r_k = \mathcal{F}(\boldsymbol{\beta}) - \mathbf{y} = (-1)[\mathbf{y} - \mathcal{F}(\boldsymbol{\beta})]$ we need to multiply $\mathbf{b}$ by -1 resulting in a subtraction.

If we substitute the parameters in (3.4) by $\mathbf{H}$ and $\mathbf{b}$, we get the equation system $(\mathbf{H} + \lambda \cdot diag(\mathbf{H}))\boldsymbol{\delta} = \mathbf{b}$. With $\mathbf{A} = (\mathbf{H} + \lambda \cdot diag(\mathbf{H}))$ we can calculate $\boldsymbol{\delta} = \mathbf{A}^{-1}\mathbf{b}$. In a loop (Alg. 2, l. 6) we try to find a value for λ so that the error gets smaller. If this is the case, the parameter $\boldsymbol{\beta}$ and the *error* value get updated and the next iteration of the LMA commences. We alter λ in the following manner: If the update step led to an error decrease, λ is either set to 0 if it was below 0.2 or halved otherwise. If the error increased, λ is either set to 0.2 if it was close to 0 or multiplied by 2^{incTry} otherwise. The LMA terminates in the following cases: (i) if $\mathbf{A}$ is not invertible, e.g. due to rounding issues. (ii) if the residual reduction is too small (line 17). (iii) if the stepsize gets too small or the maximum number of tries ($incTry_{max} = 50$) has been reached (line 27). Lastly, we check if the final *error* value is below the empirically found threshold of 0.02. In this case, we return the estimated plane parameters. Otherwise a zero vector is returned, which indicates that no fitting plane could be found.

3.6.2 Plane Refinement and Propagation

This step consists of two parts: Firstly, we traverse the multi voxel blocks and compare their plane parameters. If they are similar, i.e. their angular (`diff_degree`) and Euclidean (`diff_euclidean`) differences are below a threshold, we merge them to get a new, refined plane estimation. Secondly, we create a proximity map, where we propagate the plane parameters of every block to its neighbors (up to a distance defined in `plane_propagation_dist`). Since the plane parameters can be altered from different multi voxel blocks, this step is not processed in parallel on the GPU.

For the first part we apply a traversal queue: We start with the plane parameters of one multi voxel block and compare them to all its (max $3 \times 3 \times 3 - 1 = 26$) neighbors. To determine if two planes are similar, we check the following two conditions:

1. The angular difference:

$$\mathbf{n}_i^\top \mathbf{n}_j > cos(\Theta_\alpha) \quad , \tag{3.13}$$

where $\mathbf{n}_i$, $\mathbf{n}_j$ are the normal vectors of plane i and j and Θ_α is the threshold angle.

Algorithm 2: Estimate Plane Parameters

$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta_{init}}$, $\lambda \leftarrow \lambda_{init}$
$error_{old} \leftarrow$ `CalculateError(...)`
for $steps \leftarrow 0$ **to** $steps_{max}$ **do**
    $incTry, error, \mathbf{H}, \mathbf{b} \leftarrow 0$ `// initialize everything with zeros`
    $\mathbf{H}, \mathbf{b} \leftarrow$ `CalculateUpdate(...)`
    while *true* **do**
        $\mathbf{A} \leftarrow \mathbf{H} + \lambda \cdot diag(\mathbf{H})$
        if $\mathbf{A}$ *invertible* **then**
            $\boldsymbol{\delta} \leftarrow \mathbb{A}^{-1} \cdot \mathbf{b}$
            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta_{est}} + \boldsymbol{\delta}$
        else
            $steps = steps_{max}$
            break
        $error \leftarrow$ `CalculateError(...)`
        $incTry \leftarrow incTry + 1$
        if $error < error_{old}$ **then**
            if $error/error_{old} > 0.9999$ **then**
                `// residual reduction too small`
                $steps \leftarrow steps_{max}$
            $\boldsymbol{\beta_{est}} \leftarrow \boldsymbol{\beta}$
            $error_{old} \leftarrow error$
            if $\lambda < 0.2$ **then**
                $\lambda \leftarrow 0$
            else
                $\lambda \leftarrow \lambda \cdot 0.5$
            break
        else
            if $(|\boldsymbol{\delta}| < 1e-10) || (incTry > incTry_{max})$ **then**
                `// stepsize too small or too many tries for lambda`
                $steps \leftarrow steps_{max}$
                break
            if $\lambda < 1e-10$ **then**
                $\lambda \leftarrow 0.2$
            else
                $\lambda \leftarrow \lambda \cdot 2^{incTry}$
if $error < 0.02$ **then**
    `// found a plane with low enough error`
    return $\boldsymbol{\beta_{est}}$
else
    return $\mathbf{0}$

2. The Euclidean difference:

$$|\boldsymbol{\beta}_i^\top \tilde{\mathbf{X}_{ij}}| < \Theta_\epsilon \quad , \tag{3.14}$$

with

$$\mathbf{X}_{ij} = \mathbf{V}_j - (\boldsymbol{\beta}_i^\top \tilde{\mathbf{V}_j})\mathbf{n}_i \quad , \tag{3.15}$$

where $\boldsymbol{\beta}_i$ are the plane parameters of plane i, $\tilde{\mathbf{X}_{ij}}$ is the projection of volume center $\mathbf{V}_j$ onto plane i and Θ_ϵ is the threshold for the Euclidean distance.

If the two criteria are met, we add the block to the traversal queue and assign it a region ID, e.g. the first checked block gets region ID 1 and all similar blocks are also marked with 1. After all comparisons we delete the current entry from the queue and process the next queue entry. If a match is found, we add the plane parameters to a vector $\boldsymbol{\beta}_{refined}$. Due to the fact that every visited block is marked, we avoid loops and multiple comparisons. If the traversal queue is empty, i.e. there are no more similar parameters in the neighborhood, we divide $\boldsymbol{\beta}_{refined}$ by the number of matches (resulting in an averaging) and update the plane parameters of all multi voxel blocks with the current region ID. Thus, all blocks with the same region ID also have the same plane parameters. However, a downside to this approach is that every block will not match with its best fit but with its first.

Afterwards, we take each multi voxel block, iterate through all its neighbors within a certain distance `plane_propagation_dist` and add all found region IDs to a `proximity_map`. This plane propagation approach also counters the aforementioned downside of first fit versus best fit, because in the end a multi voxel block will always adapt to its best plane candidate. After this process every multi voxel block has its own (merged) plane parameters and a set of plane parameters from the multi voxel blocks around it. We call the combination of all these parameters plane candidates.

3.6.3 Applying Plane Parameters to 3D Model

In the last step, we update the TSDF values $\mathcal{D}(\mathbf{X}_k)$ for every multi voxel block of the 3D world model. Within every multi voxel block with at least 1 plane candidate, we update $\mathcal{D}'(\mathbf{X}_k)$ for every voxel k in parallel on the GPU in the following manner:

$$\mathcal{D}'(\mathbf{X}_k) = \begin{cases} \frac{D_{min}(\mathbf{X}_k)}{\delta}, & \text{if } \exists \boldsymbol{\beta}_i, \boldsymbol{\beta}_j : \ -\delta < \boldsymbol{\beta}_i^\top \tilde{\mathbf{X}_k} \cdot \boldsymbol{\beta}_j^\top \tilde{\mathbf{X}_k} < 0 \\ \frac{D_{closest}(\mathbf{X}_k)}{\delta}, & \text{if } |D_{closest}(\mathbf{X}_k)| < \delta \text{ and } |\frac{D_{closest}(\mathbf{X}_k)}{\delta} - \mathcal{D}(\mathbf{X}_k)| < 1 \\ \mathcal{D}(\mathbf{X}_k), & \text{otherwise} \end{cases} \quad , \tag{3.16}$$

Here, the first case addresses voxels near plane intersections (needs more than 1 plane candidate) where the distances ($\boldsymbol{\beta}_i^\top \tilde{\mathbf{X}_k}$ for plane i and $\boldsymbol{\beta}_j^\top \tilde{\mathbf{X}_k}$ for plane j) are not of the same sign, indicating that the voxel is behind a surface. We therefore assign the smallest signed distance value $D_{min}(\mathbf{X}_k)$, which in this case is always negative. Note that we have to divide this value by our truncation band value δ to fit into our volumetric representation

(see Sec. 2.12). In the second case, we assign the smallest absolute value $D_{closest}(\mathbf{X}_k)$ to any voxel within the truncation band and not too far from its original value $\mathcal{D}(\mathbf{X}_k)$. This overwrites the distance values of all voxels close to an estimated plane. If none of the above cases applies, indicating that the voxel is far from any estimated plane, we leave the TSDF value unchanged.

Note that we apply this update to our whole multi voxel block (which consists of multiple allocated and unallocated blocks of the size `SDF_BLOCK_SIZE` as stated before) and therefore might get values for voxels were no observations have been made. We then allocate new memory if necessary to complete the geometry. This leads to a hole filling in regions where planes have been estimated.

4 Evaluation and Results

To demonstrate the capabilities of our system, we evaluate several real-world image sequences from well known datasets [8, 61] and the synthetic ICL-NUIM dataset [23]. Furthermore, we test our approach on our own Orbbec Astra Pro recordings. At first we compare the trajectory error of various systems to our ORB-SLAM2-based approach. Then we qualitatively analyze the surface reconstruction accuracy based on already refined trajectories and live tracking. Finally, we show how we can further improve the 3D model with our plane estimation.

Contents

4.1 Trajectory Error

In this section, we examine the trajectory error of several existing VO and VSLAM systems. As VO systems we choose the ICP-based tracking of InfiniTAM (ITM) [29] and ICP CUDA [68]. Our tested VSLAM systems include ORB SLAM 2 [42], DVO-SLAM [31] and RGBD-SLAM [12]. We evaluate all systems on multiple image sequences from the TUM RGBD dataset [61], the synthetic ICL NUIM dataset [23] and the Bundle Fusion dataset [8]. We run all systems in their standard settings using the code available online at maximum resolution of 640×480. For RGBD-SLAM, we set the feature detector and descriptor type to ORB and extract a maximum of 600 keypoints per frame. In ORB-SLAM2, we extract 1000 features per frame with a minimum of 7 per cell and 8 scale pyramid levels. Finally, we run DVO-SLAM with its standard 3 scale pyramid levels. We test all the systems on an Intel Core 2 Quad CPU Q9550 desktop computer with 8GB RAM and an NVIDIA GeForce GTX 480.

We use the evaluation tools provided by [62] for the TUM benchmark to calculate the Absolute Trajectory Error (*ATE*) and the Relative Pose Error (*RPE*). The ATE directly compares the absolute distances of the trajectory in the ground truth file and the output trajectory of the various systems. This is a good measurement for global consistency in VSLAM systems. Let $\mathbf{P}_{1:n}$ be the estimated trajectory and $\mathbf{Q}_{1:n}$ the ground truth trajectory. Then we can find a least-squares solution for the rigid-body transformation $\mathbf{S}$ which maps $\mathbf{P}_{1:n}$ onto $\mathbf{Q}_{1:n}$ and compute the absolute trajectory error at time step i:

$$\mathbf{F}_i := \mathbf{Q}_i^{-1}\mathbf{S}\mathbf{P}_i \quad . \tag{4.1}$$

Table 4.1 shows the results for the ATE Root-Mean-Square Error (*RMSE*) on several selected TUM RGBD sequences. On the ICL-NUIM datasets (see Tbl. 4.2), DVO-SLAM outshines ORB-SLAM2. This is due to the synthetic nature of the datasets, where perfect depth values allow a very accurate tracking for DVO-SLAM, whilst ORB-SLAM2 still needs to rely on the extracted ORB features. The high error value for the lr/kt1 sequence with ORB-SLAM2 is a result of not revisiting any structure and therefore being unable to perform a loop closure. The "tl" (track lost) entries on the Bundle Fusion datasets (see Tbl. 4.3) indicate that the system failed and therefore no meaningful comparison could be made. Furthermore, DVO-SLAM and RGBD-SLAM could not process the large datasets (over 8000 frames) on our hardware.

The RPE computes the relative difference of the trajectory over a fixed time interval Δ. In VO systems it can evaluate the drift and in VSLAM systems it can measure the accuracy at loop closures. The RPE at time step i is defined as:

$$\mathbf{E}_i := (\mathbf{Q}_i^{-1}\mathbf{Q}_{i+\Delta})^{-1}(\mathbf{P}_i^{-1}\mathbf{P}_{i+\Delta}) \quad . \tag{4.2}$$

We choose to evaluate the RPE for every frame (see Tbl. 4.4, Tbl. 4.5 and Tbl. 4.6) and over the time interval of 1 second (see Tbl. 4.7, Tbl. 4.8 and Tbl. 4.9). Here again, the synthetic ICL-NUIM datasets show slightly better results for the other systems compared to ORB-SLAM2.

In order to be able to use the TUM tools [62], we converted all datasets into the TUM format, i.e. we changed the image and ground truth formats and added the associate files which can also be generated with the provided tools. In cases where the algorithm non deterministic, i.e. the estimated trajectories differ for every run, we execute the algorithm 10 times and take the mean value. Algorithms belonging to this category are ORB SLAM 2 and RGBD-SLAM.

4.2 Surface Reconstruction Accuracy

For qualitative evaluation we compare our system on several well known datasets (see Fig. 4.2) and also on datasets recorded with our own Orbbec Astra Pro. For all models we set

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
fr1/desk	0.291	0.144	0.169	0.027	**0.022**
fr1/desk2	0.483	0.273	0.148	0.041	**0.023**
fr1/room	0.523	0.484	0.219	0.104	**0.069**
fr1/xyz	0.032	0.042	0.031	0.017	**0.010**
fr2/desk	0.114	1.575	0.125	0.092	**0.079**
fr2/xyz	0.042	0.223	0.021	0.016	**0.013**
fr3/office	1.258	1.161	0.120	0.034	**0.011**
fr3/nstn	1.979	1.666	0.039	0.051	**0.018**

Table 4.1: Translational ATE RMSE on the TUM RGBD dataset *[m]*

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
lr/kt0	0.045	0.697	**0.006**	0.011	0.008
lr/kt1	0.009	0.045	**0.005**	0.013	0.162
of/kt0	0.054	0.205	**0.007**	0.029	0.027
of/kt1	0.025	0.275	**0.004**	0.724	0.051

Table 4.2: Translational ATE RMSE on the synthetic ICL-NUIM dataset *[m]*

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
apt0	tl	1.242	-	-	0.096
apt1	0.818	1.300	-	-	0.086
apt2	tl	1.546	-	-	0.135
copyroom	1.100	0.571	-	-	0.055
office0	1.543	0.656	-	-	0.093

Table 4.3: Translational ATE RMSE on the Bundle Fusion dataset *[m]*

the voxel size to 2 cm, the truncation band μ to 8 cm and limit the depth measurements from 0.2 m to 5.0 m. We empirically found the parameters $\Theta_\tau = 0.005$ and $\Theta_{pc} = 1000$.

To measure the surface reconstruction accuracy, we calculate the one-sided Hausdorff distance from the groundtruth 3D model to the reconstructed 3D model:

$$d_H(X, Y) = \sup_{x \in X} \inf_{y \in Y} \mathrm{d}(x, y) \quad , \tag{4.3}$$

where X is the set of groundtruth vertices, Y the set of the reconstructed vertices and

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
fr1/desk	0.018	**0.010**	0.012	0.011	**0.010**
fr1/desk2	0.030	0.012	0.011	0.011	**0.010**
fr1/room	0.021	**0.008**	**0.008**	**0.008**	**0.008**
fr1/xyz	**0.004**	**0.004**	0.007	0.006	0.006
fr2/desk	0.024	0.108	0.016	0.004	**0.003**
fr2/xyz	0.007	0.027	0.006	0.003	**0.002**
fr3/office	0.052	0.131	0.017	0.007	**0.005**
fr3/nstn	0.242	0.263	0.017	0.012	**0.010**

Table 4.4: Translational RPE RMSE on the TUM RGBD dataset with $\Delta = 1 frame$ *[m/f]*

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
lr kt0	0.005	0.005	0.001	0.004	0.004
lr kt1	0.001	0.001	0.001	0.003	0.016
of kt0	0.003	0.002	0.001	0.009	0.005
of kt1	0.002	0.010	0.001	0.033	0.012

Table 4.5: Translational RPE RMSE on the synthetic ICL-NUIM dataset $\Delta = 1 frame$ *[m/f]*

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
apt0	tl	0.011	-	-	0.008
apt1	0.011	0.011	-	-	0.009
apt2	tl	0.035	-	-	0.032
copyroom	0.021	0.006	-	-	0.006
office0	0.021	0.011	-	-	0.016

Table 4.6: Translational RPE RMSE on the Bundle Fusion dataset $\Delta = 1 frame$ *[m/f]*

$\mathrm{d}(x, y)$ is the Euclidian distance between the two vertices x and y. We sample each vertex in X, find the distance to the closest point in Y and take the average. Table 4.10 lists the result of this process for different datasets and methods. ORB-SLAM2 outperforms all other systems on the evaluated TUM datasets (fr1) when integrating the model frame by frame without using de-integration (all frames). Note that we used already optimized trajectories for this test and thus no pose updates had to be incorporated. When we only integrate keyframes into the model, i.e. all non keyframes will not be processed by the system, the reconstruction error increases slightly. We show, that we can counter this

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
fr1/desk	0.207	0.100	0.052	0.036	**0.026**
fr1/desk2	0.327	0.164	0.061	0.045	**0.033**
fr1/room	0.259	0.129	0.056	0.053	**0.048**
fr1/xyz	0.047	0.031	0.024	0.027	**0.016**
fr2/desk	0.024	0.109	0.016	0.018	**0.012**
fr2/xyz	0.007	0.027	0.005	0.006	**0.004**
fr3/office	0.052	0.131	0.017	0.016	**0.009**
fr3/nstn	0.242	0.263	0.017	0.019	**0.015**

Table 4.7: Translational RPE RMSE on the TUM RGBD dataset with $\Delta = 1s$ *[m/s]*

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
lr/kt0	0.005	0.140	**0.002**	0.003	0.008
lr/kt1	**0.001**	0.017	0.002	0.002	0.074
of/kt0	**0.003**	0.061	**0.003**	0.005	0.016
of/kt1	**0.002**	0.152	**0.002**	0.007	0.034

Table 4.8: Translational RPE RMSE on the synthetic ICL-NUIM dataset with $\Delta = 1s$ *[m/s]*

	ITM	**ICP CUDA**	**DVO SLAM**	**RGBD SLAM**	**ORB SLAM2**
apt0	tl	0.011	-	-	0.008
apt1	0.010	0.011	-	-	0.008
apt2	tl	0.035	-	-	0.026
copyroom	0.021	0.006	-	-	0.006
office0	0.021	0.011	-	-	0.015

Table 4.9: Translational RPE RMSE on the Bundle Fusion dataset $\Delta = 1s$ *[m/s]*

effect by using our fused depth maps. On the ICL-NUIM datasets, InfiniTAM and DVO-SLAM outshine ORB-SLAM2. This is due to the previously mentioned synthetic nature of the dataset, where the trajectory estimation of the former two systems is better. RGBD-SLAM works well on the icl/lr0 and icl/lr1 datasets, but fails to accurately reconstruct icl/of1, resulting in a large error. Furthermore, we can see in Fig. 4.1 that no loop closure could be performed in the icl/lr1 dataset (due to not revisiting any structure), which leads to a larger error. Note that in the icl/of1 dataset our method shows some areas with an increased error. The reason for this is that no keyframe was detected there and

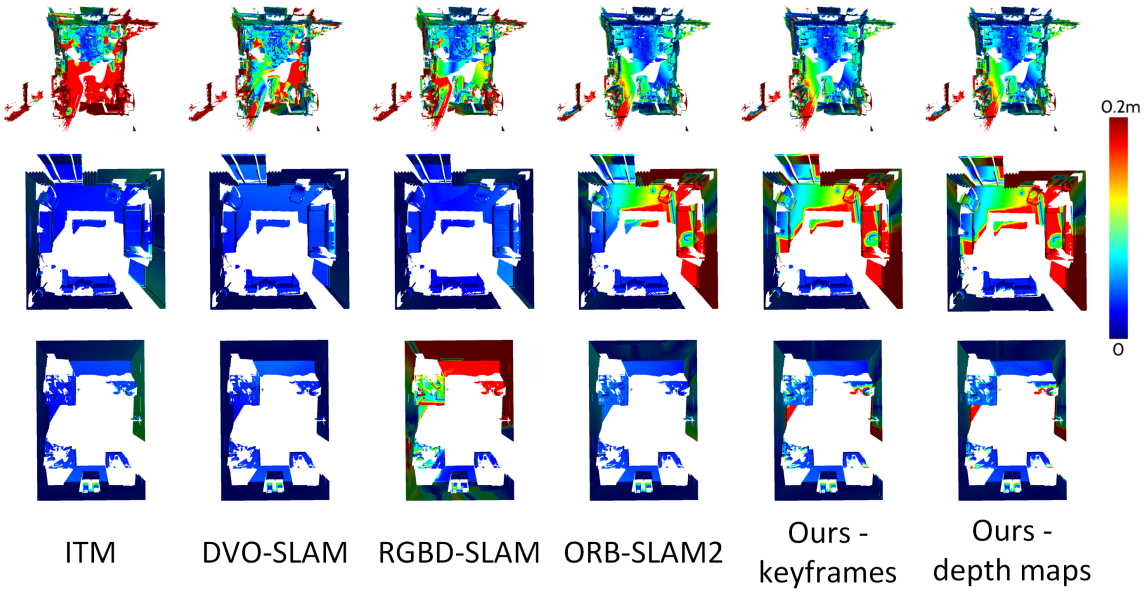


Figure 4.1: Surface reconstruction: Heat maps depicting the error from the ground truth model to the estimated model. Datasets from top to bottom: fr1/room, icl/lr1, icl/of1.

(a) lr/kt0 **(b)** fr1/xyz

Figure 4.2: Sample reconstruction models of our approach.

consequently no values exist.

In Figure 4.3 we compare the original InfiniTAM tracker to our approach (top). We can see that InfiniTAM reconstructs 2 walls in the upper right corner due to an absence of loop closure detection. On the bottom we compare our approach to tracking and integrating every frame with ORB-SLAM2 but not using any de-integration.

Figure 4.4 depicts the effects of loop closure. Our method keeps the global model consistent when revisiting places.

	ITM	DVO SLAM	RGBD SLAM	ORB-SLAM2 all frames	ORB-SLAM2 keyframes	ORB-SLAM2 depth maps
fr1/desk	0.067	0.071	0.037	**0.033**	0.037	0.034
fr1/desk2	0.091	0.088	0.078	**0.043**	0.051	0.048
fr1/room	0.228	0.152	0.164	**0.084**	0.091	0.087
fr1/xyz	0.033	0.046	0.019	**0.012**	0.017	0.015
icl/lr0	**0.004**	0.005	0.006	0.008	0.016	0.016
icl/lr1	0.015	**0.007**	0.008	0.097	0.114	0.113
icl/of1	0.014	**0.006**	0.095	0.017	0.027	0.025

Table 4.10: Evaluation of the surface reconstruction accuracy: Mean Hausdorff distances from the ground truth surface to the reconstructed surfaces *(m)*.

4.3 Runtime

Regarding the runtime, InfiniTAM with its own tracker, running completely on the GPU, needs an average of $6ms$ per frame in our experiments. ORB-SLAM2, running on the CPU, combined with the InfiniTAM reconstruction but without depth map update and global model update needs an average of $56ms$. When adding the depth map update, the time increases to roughly $67ms$. This is due to the requirement of the creation of `DepthMap` objects on the CPU and the need for de-integration when a new frame is assigned to an older keyframe. However, we are now able to perform an online adaption of the 3D model with our global model update at an average runtime of $73ms$ even when large loop closures occur like in Figure 4.4. By using only every other frame for a depth map update, this number could be reduced to $68ms$.

There still exist possibilities to optimize the code: (i) optimize swapping between GPU and CPU memory: Right now, all `DepthMap` objects of the keyframes reside on the CPU and need to be transfered to the GPU whenever a new reference keyframe is needed. We could keep the latest few keyframes on the GPU, because they are most likely to be referenced again, to avoid constant memory allocation, (ii) let ORB-SLAM2 and InfiniTAM run in separate threads: By decoupling tracking and model generation we could avoid leaving either system idle, (iii) substitute ORB-SLAM2 for another tracker. ORB-SLAM2's principle of first adding a lot of keyframes and later cull them is not ideal here, because this needs additional de-integration and re-integration steps.

4.4 Plane Estimation

We estimate planes in our 3D model in order to remove noise and fill holes in large planes like walls, floors or ceilings. Due to the fact that fine structures are smoothed

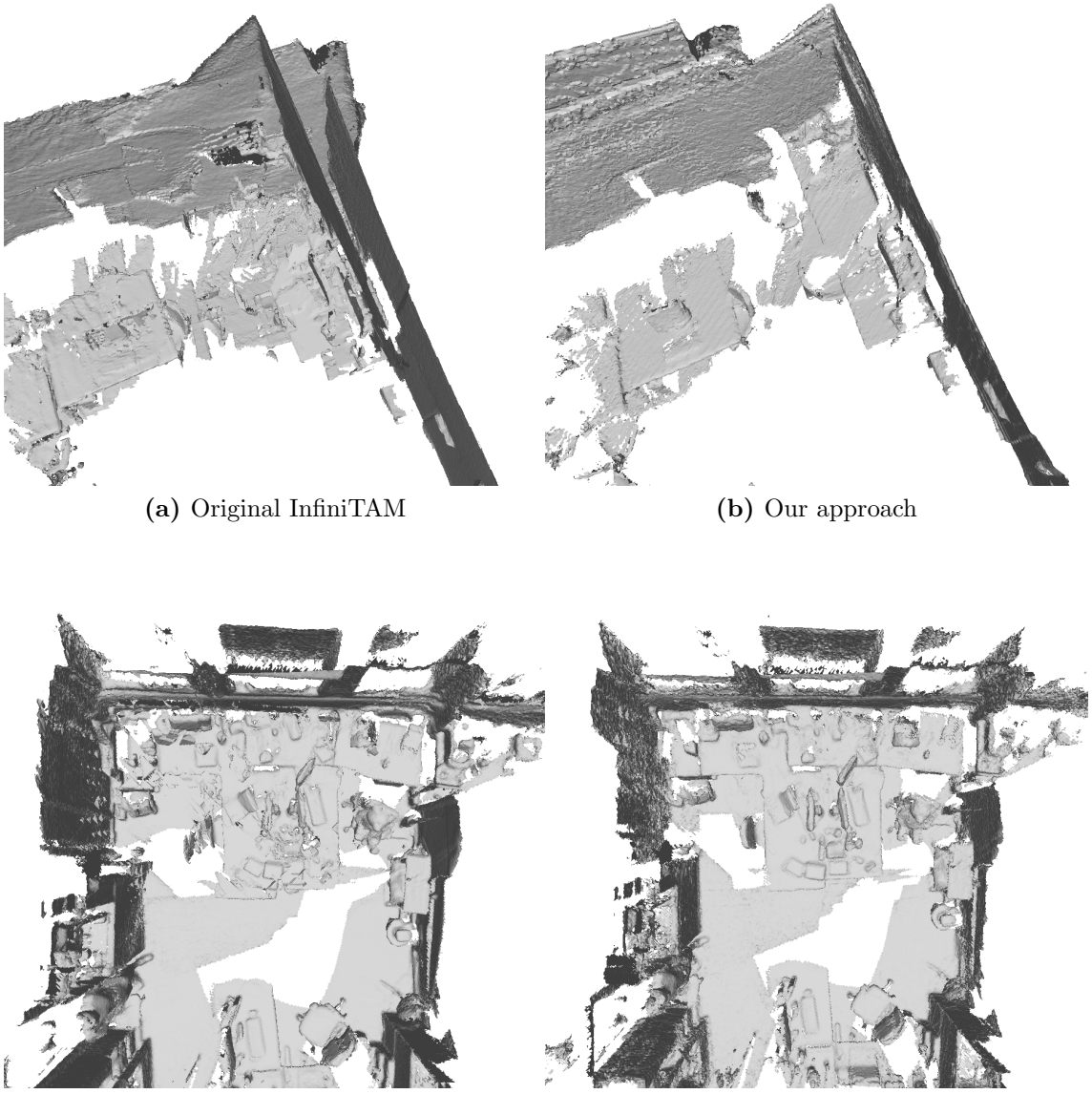
and our groundtruth cannot fill holes, a meaningful comparison based on the Hausdorff distance could not be performed. For the model reconstruction we choose a voxel size of 3 cm, a truncation band of $\mu = 10$ cm and limit the depth measurements to a range of 0.2 m to 5.0 m. The depth map update parameters are again $\Theta_\tau = 0.005$ and $\Theta_{pc} = 1000$. Our plane estimation parameters are set in the following manner: multi voxel block is of size 2, `sdfThreshold` is set to 0.8, $\Theta_\alpha = 5°$, $\Theta_\epsilon = 0.1$ m, $\Theta_H = 0.05$ and `plane_propagation_dist` is 10, which means that we consider 10 multi voxel blocks in every direction from the current position.

In Figure 4.5, 4.6 and 4.7 we can see that our algorithm correctly estimates the planes for walls and floors. Figure 4.8 shows the effects in more detail: Irregularities in the wall surface get smoothed and holes due to occlusions from the monitors get closed. However, since fine structures, e.g. pictures on the wall, only differ minimally from the estimated wall plane, they disappear too. Depending on the application, this is a trade-off one has to take into account: If we want to maintain a detailed model of the room, plane estimation might not be desired. However, if we want to check if a structure is closed, e.g. for an augmented reality application, plane estimation can be of huge importance.

The whole extend of our method is illustrated in Figure 4.9: The original InfiniTAM is unable to adapt the model to global updates and therefore structures can appear at the wrong places, e.g. the reconstruction of 2 walls on the left and the tables at the bottom. With our approach we obtain a globally consistent model. Plane estimation helps us to smooth and complete planes. Note that even after plane estimation, a hole in the middle of the floor remains. This is due to the size of the un-scanned area. If no voxels are allocated within the multi voxel block from scanning, no plane parameters are propagated to avoid the closing of actual openings, e.g. windows or open doors. More results of our plane estimation are depicted in Figure 4.10 and Figure 4.11.

An issue with the current implementation is, that at intersection of planes, i.e. corners, the planes expand beyond the border in some cases. The reason behind this behavior is probably, that whenever we detect a plane within a multi voxel block, we allocate all voxels within the truncation band of the surface.

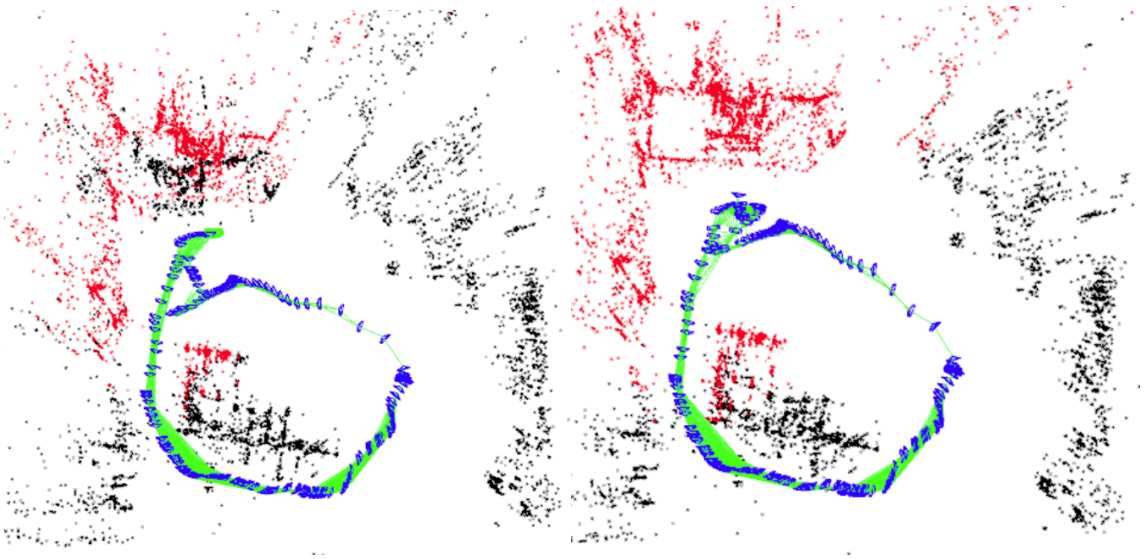
Regarding the runtime of this plane estimation, it is very parameter dependent. A choice of a large multi voxel block size with a rather small propagation distance can be handled within a few hundred milliseconds. Smaller blocks and a larger distance would take up to 20 seconds in our experiments.



(a) Original InfiniTAM **(b)** Our approach

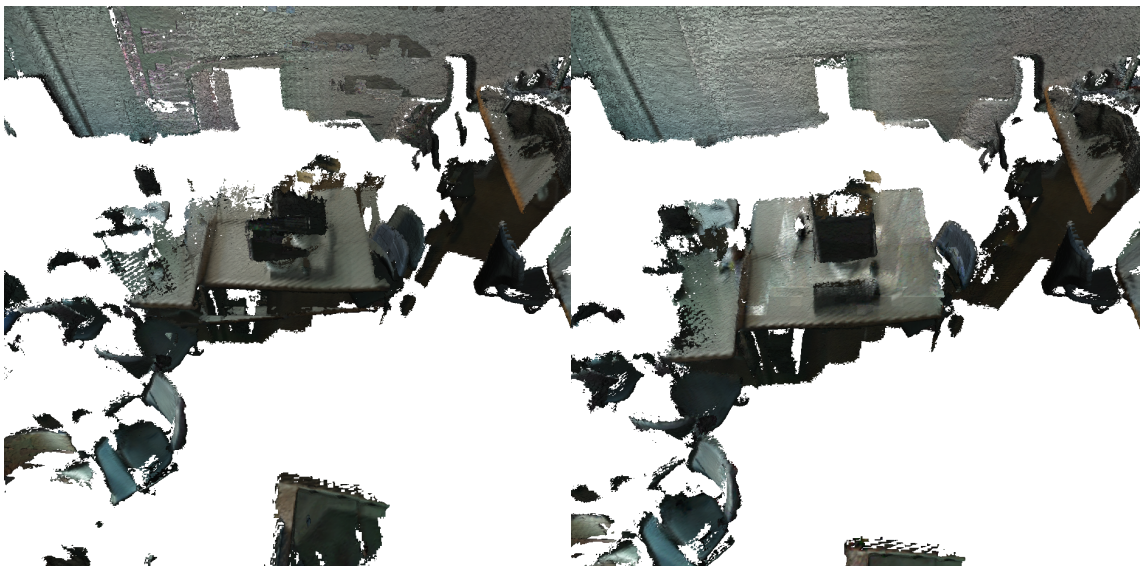
(c) ORB-SLAM2 without model update **(d)** Our approach

Figure 4.3: Surface reconstruction of office scenes with loop closure: (a) Tracking and integration of every frame with the original InfiniTAM ICP tracker. (b) Our approach with ORB-SLAM2 pose estimation, depth map and global model update. (c) Tracking and integration of every frame with ORB-SLAM2 without model update. (d) Our approach.



(a) ORB-SLAM2 before loop closure

(b) ORB-SLAM2 after loop closure



(c) Our method before loop closure

(d) Our method after loop closure

Figure 4.4: Effects of loop closure: (a) ORB-SLAM2 before detecting a loop closure. (b) ORB-SLAM2 after a loop closure has been recognized and the estimated poses have been adjusted. (c) Without a loop closure, the InfiniTAM model is inconsistent. (d) With our method we can adapt the 3D model on the fly.

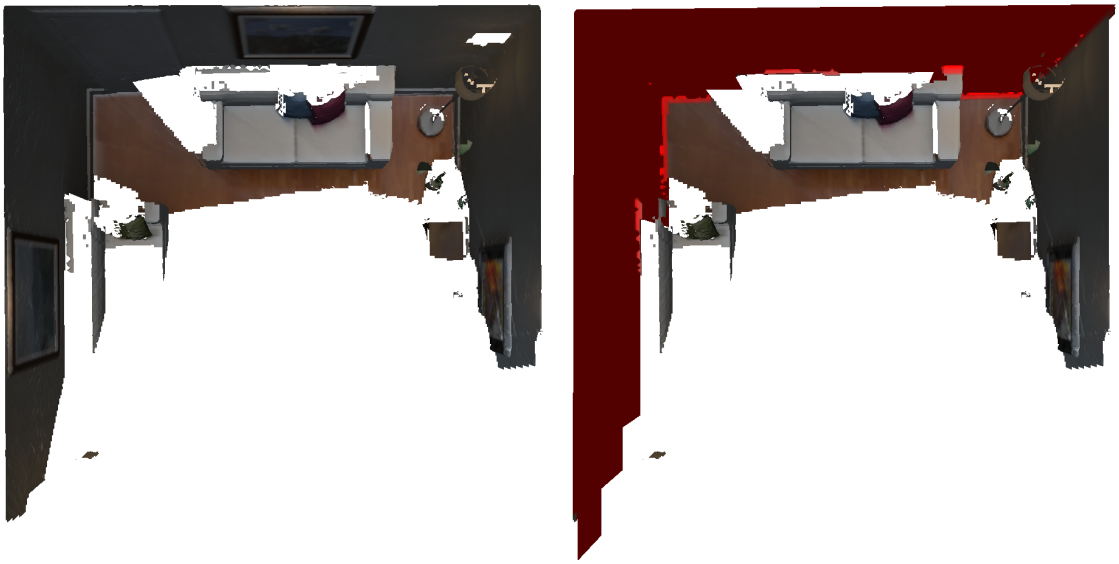


Figure 4.5: Plane estimation (depicted in red) in the icl/lr0 dataset with ORB-SLAM2 trajectory.

Figure 4.6: Plane estimation (depicted in red) in the icl/lr1 dataset with groundtruth trajectory.

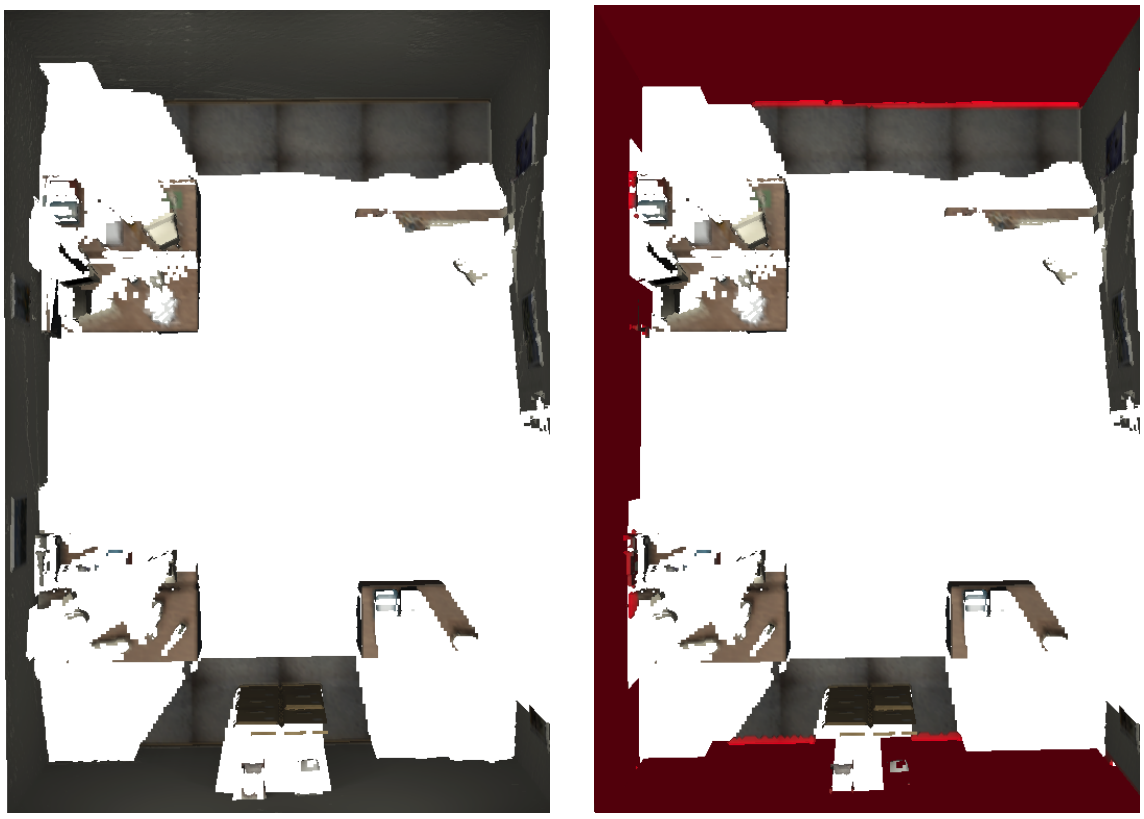
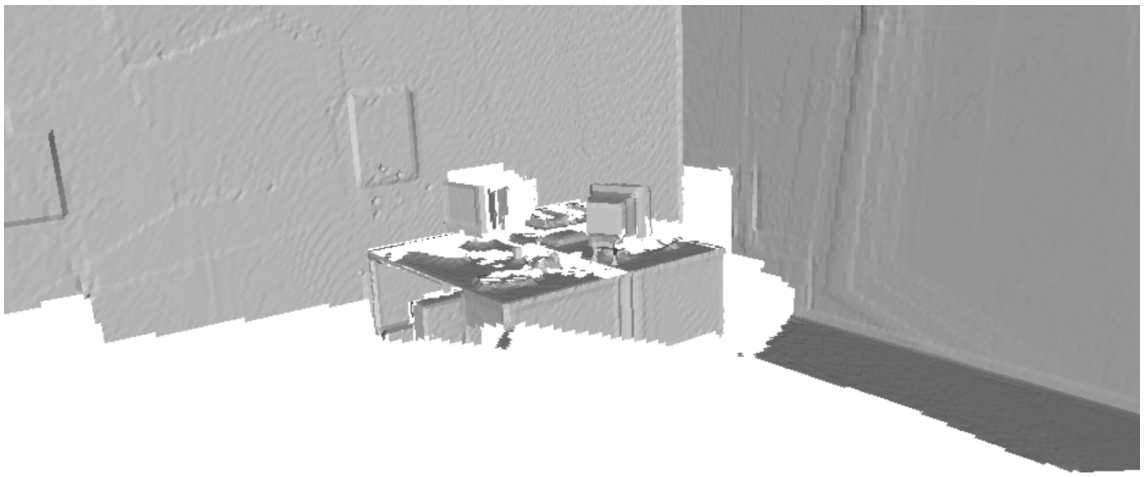
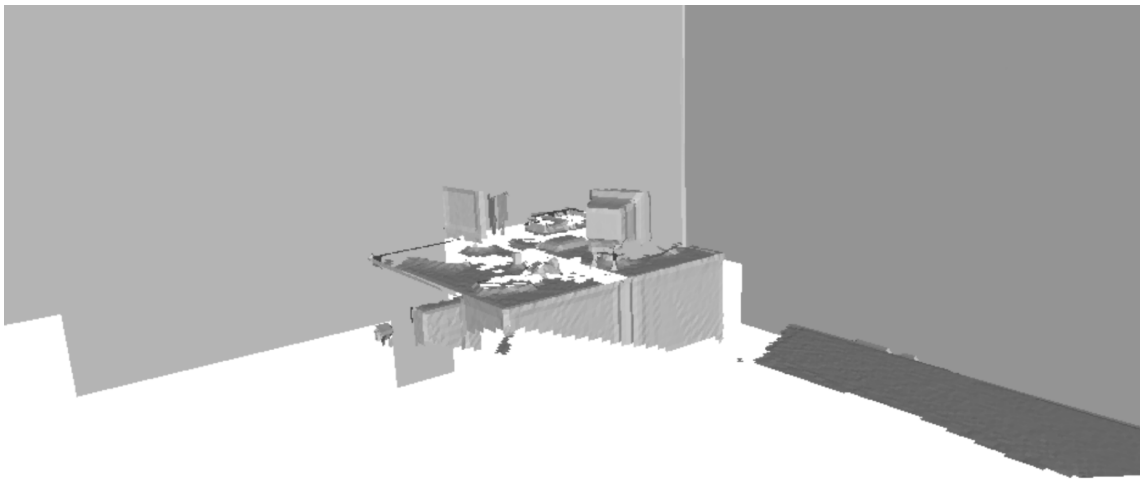


Figure 4.7: Plane estimation (depicted in red) in the icl/of1 dataset with ORB-SLAM2 trajectory.



(a) No plane estimation



(b) Plane estimation

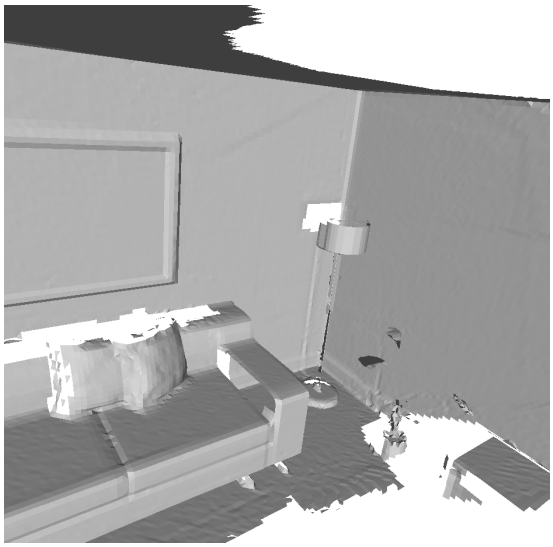
Figure 4.8: Closeup of the plane estimation effects in the icl/of1 dataset: The ripples in the wall due to slightly wrong pose estimations are eradicated. Furthermore, holes in the wall get filled. However, also fine structures like pictures dissappear.

(a) Original InfiniTAM

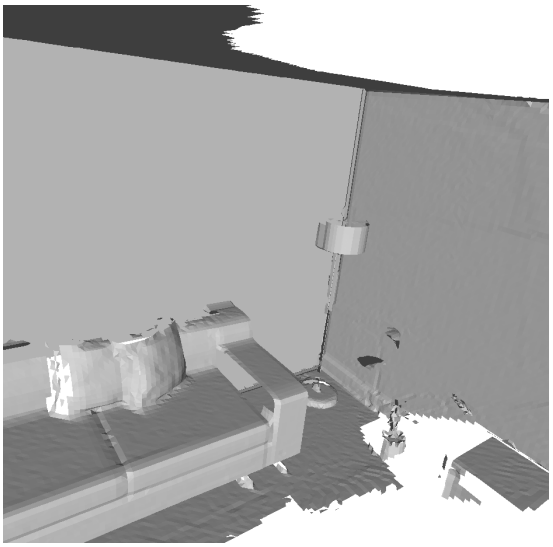
(b) Our Approach

(c) Our Approach with Plane Estimation

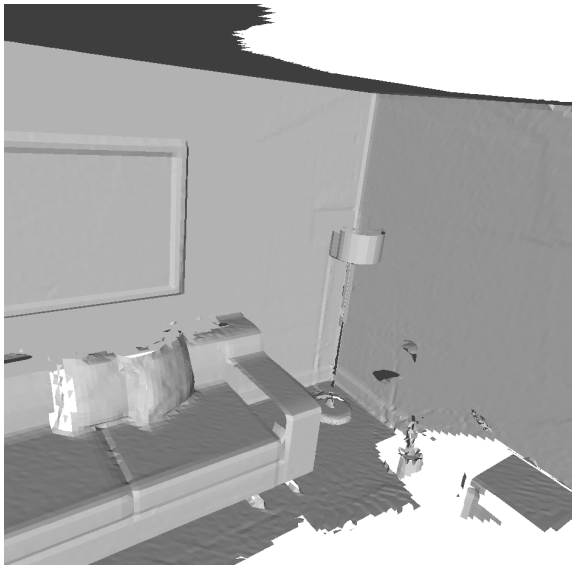
Figure 4.9: Sample reconstruction of a room recorded and reconstructed in real-time with our Orbbec Astra Pro. (a) shows the original InfiniTAM reconstruction, which is unable to adapt the model to loop closure (see top left corner). (b) depicts our approach with a globally consistent model. (c) demonstrates the effects of our plane estimation: large planes get smoothed and the un-scanned area in the middle of the scene shrinks.



(a) No plane estimation



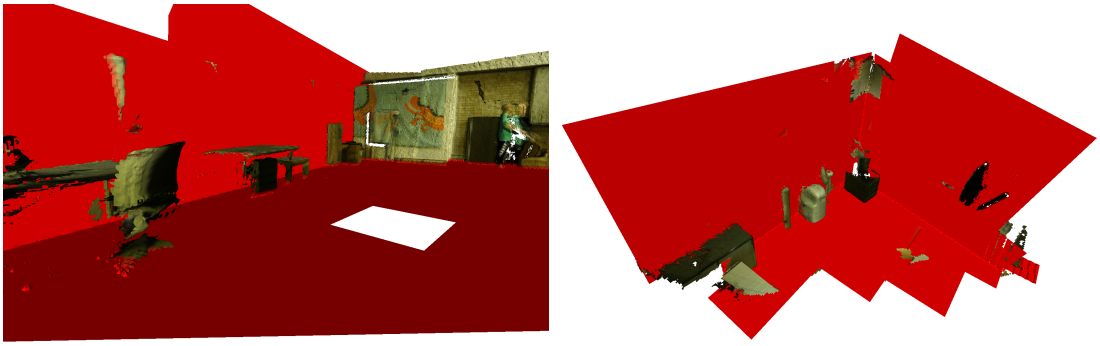
(b) Plane estimation



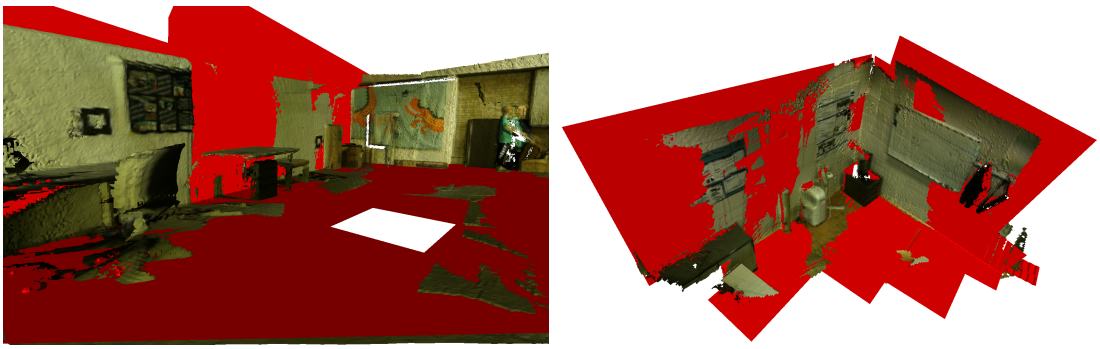
(c) Combined

Figure 4.10: Closeup of the plane estimation effects in the icl/lr0 dataset: By using a combination of original reconstruction and plane estimation, holes can be filled while fine structures can still be maintained.

(a) No plane estimation



(b) Plane estimation



(c) Combined

Figure 4.11: Closeups of reconstructions from our own recordings. Our plane estimation approach is able to smooth the geometry and close holes in the wall and floor.

5 Conclusion

In this thesis we presented a real-time capable method to create globally consistent dense 3D models. To solve this challenging problem, we combined the tracking accuracy of a state-of-the-art SLAM system [42] with the dense model generation of a volumetric fusion system [29]. We utilize the depth information of all frames but fuse all points into the depth map of their corresponding keyframes. Any points that we cannot incorporate into the depth map directly, e.g. because they transform to out-of-boundary values, we store in a pointcloud. We try to fuse the information of the pointcloud whenever we add a new keyframe. The fused depth map is then integrated into the 3D model instead of every single frame. This allows us to dynamically de-integrate and re-integrate keyframes, which hold the information of several frames, resulting in a speed up of about a factor of 10. In this manner our system is able to adapt the model online, when updated poses are available, e.g. after loop closure or bundle adjustment. Furthermore, we proposed a plane estimation algorithm to improve the quality of the model by significantly reducing noise and filling holes in large planes. For that purpose, we at first calculate the plane parameters within a larger voxel neighborhood. We then refine these parameters and propagate them through the neighboring blocks. Finally, we adapt the 3D model according to its best fitting plane if appropriate.

For real world data we have shown that our method yields excelling results, especially when compared to the original InfiniTAM ICP approach. Both, the trajectory and surface reconstruction error, improve significantly. We have also shown comparable results for synthetic datasets, despite not utilizing an ideal tracking method for this kind of data. A downside to our approach is the increased runtime, which is a result of the processing time of ORB-SLAM2 and the need for constant re-integration. Another issue is that, despite trying to recycle all of the information, our depth map fusion is a little less dense than a frame-by-frame integration. Our plane estimation algorithm correctly detected planes and was able to automatically smooth and complete geometry. However, the parameter choice can be crucial and fine structures (e.g. pictures on the wall) will disappear.

In the course of this thesis we submitted a paper [66] to the Austrian Association for

Pattern Recognition (OAGM/AAPR) Workshop 2018 and were accepted. The issues of the paper covered our de-integration process, depth map fusion and global model update.

In future work, it would be desirable to incorporate the swapping feature of InfiniTAM, which we did not utilize in this thesis. This would allow us to reconstruct larger scenes even with only limited GPU memory available. Another issue that has not been addressed in this thesis is texturing. Right now, we only use the RGB values of the keyframes and average them, which leads to a motion blur. Through a more sophisticated approach the appearance of the 3D model could probably be further enhanced.

A next step to increase speed is to decouple model creation and tracker, as mentioned in Section 4.3. Note that our system is not limited to ORB-SLAM2, but can in theory work with any keyframe-based tracking method. Therefore, a decoupling with a clean interface would enable means for a dense 3D reconstruction for many different SLAM and VO systems often lacking this feature.

A

List of Acronyms

ATE	Absolute Trajectory Error
BA	Bundle Adjustment
BRIEF	Binary Robust Independent Elementary Features
CPU	Central Processing Unit
FAST	Features from Accelerated Segment Test
GPU	Graphics Processing Unit
ICP	Iterative Closest Point
IMU	Intertial Measurement Unit
IR	InfraRed
LMA	Levenberg-Marquard Algorithm
ORB	Oriented FAST and Rotated BRIEF
RANSAC	Random Sample Consesus
RMSE	Root-Mean-Square Error
RPE	Relative Pose Error
SDF	Signed Distance Function
SIFT	Scale-Invariant Feature Transform
SLAM	Simultaneous Localization and Mapping
SURF	Speeded Up Robust Features
ToF	Time of Flight
TSDF	Truncated Signed Distance Function
UI	User Interface
VO	Visual Odometry
VSLAM	Visual SLAM

Bibliography

[1] Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf: Speeded up robust features. *European Conference on Computer Vision (ECCV)*, pages 404–417. (page 17)

[2] Besl, P. J. and McKay, N. D. (1992). Method for registration of 3-d shapes. In *Sensor Fusion IV: Control Paradigms and Data Structures*, volume 1611, pages 586–607. International Society for Optics and Photonics. (page 20)

[3] Bouguet, J.-Y. (2015). Camera calibration toolbox for matlab. `http://www.vision.caltech.edu/bouguetj/calib_doc/`. [Accessed 02-January-2018]. (page 12)

[4] Calonder, M., Lepetit, V., Strecha, C., and Fua, P. (2010). Brief: Binary robust independent elementary features. *European Conference on Computer Vision (ECCV)*, pages 778–792. (page 18)

[5] Chen, J., Bautembach, D., and Izadi, S. (2013). Scalable real-time volumetric surface reconstruction. *ACM Transactions on Graphics (ToG)*, 32(4):113. (page 25)

[6] Concha, A. and Civera, J. (2017). Rgbdtam: A cost-effective and accurate rgb-d tracking and mapping system. *International Conference on Intelligent Robots and Systems (IROS)*. (page 22, 23)

[7] Curless, B. and Levoy, M. (1996). A volumetric method for building complex models from range images. In *Computer graphics and interactive techniques*, pages 303–312. ACM. (page 29)

[8] Dai, A., Nießner, M., Zollhöfer, M., Izadi, S., and Theobalt, C. (2017). Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface re-integration. *ACM Transactions on Graphics 2017 (ToG)*. (page 2, 24, 26, 27, 29, 47)

[9] Diebel, J. (2006). Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16):1–35. (page 14)

[10] Dzitsiuk, M., Sturm, J., Maier, R., Ma, L., and Cremers, D. (2016). De-noising, stabilizing and completing 3d reconstructions on-the-go using plane priors. *International Conference on Robotics and Automation (ICRA)*. (page 30, 41)

[11] Endres, F. (2018). Rgbdslamv2. `https://github.com/felixendres/rgbdslam_v2`. [Accessed 20-March-2018]. (page 23)

[12] Endres, F., Hess, J., Sturm, J., Cremers, D., and Burgard, W. (2014). 3-d mapping with an rgb-d camera. *Transactions on Robotics (T-RO)*, 30(1):177–187. (page 2, 23, 47)

[13] Engel, J., Koltun, V., and Cremers, D. (2018). Direct sparse odometry. *Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 40(3):611–625. (page 2, 19)

[14] Engel, J., Schöps, T., and Cremers, D. (2014). Lsd-slam: Large-scale direct monocular slam. In *European Conference on Computer Vision (ECCV)*, pages 834–849. Springer. (page 19, 24, 25)

[15] Engel, J., Stückler, J., and Cremers, D. (2015). Large-scale direct slam with stereo cameras. In *Intelligent Robots and Systems (IROS)*, pages 1935–1942. IEEE. (page 2, 22)

[16] Feng, C., Taguchi, Y., and Kamat, V. R. (2014). Fast plane extraction in organized point clouds using agglomerative hierarchical clustering. In *International Conference on Robotics and Automation (ICRA)*, pages 6218–6225. IEEE. (page 30)

[17] Ferstl, D., Reinbacher, C., Riegler, G., Rüther, M., and Bischof, H. (2015). Learning depth calibration of time-of-flight cameras. In *British Machine Vision Conference (BMVC)*, pages 102–114. (page 12)

[18] Fischler, M. A. and Bolles, R. C. (1987). Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. In *Readings in Computer Vision*, pages 726–740. Elsevier. (page 23)

[19] Fraundorfer, F., Scaramuzza, D., and Pollefeys, M. (2010). A constricted bundle adjustment parameterization for relative scale estimation in visual odometry. In *International Conference on Robotics and Automation (ICRA)*, pages 1899–1904. IEEE. (page 19)

[20] Gálvez-López, D. and Tardos, J. D. (2012). Bags of binary words for fast place recognition in image sequences. *Transactions on Robotics (T-RO)*, 28(5):1188–1197. (page 23)

[21] Guerrero, M. (2011). A comparative study of three image matcing algorithms: Sift, surf, and fast. (page 18)

[22] Haala, N. and Wenzel, K. (2016). Volumetric range image integration. `http://www.ifp.uni-stuttgart.de/lehre/diplomarbeiten/korcz/index.html`. [Accessed 25-January-2018]. (page 28)

[23] Handa, A., Whelan, T., McDonald, J., and Davison, A. (2014). A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM. In *International Conference on Robotics and Automation (ICRA)*, Hong Kong, China. IEEE. (page 47)

[24] Hartley, R. and Zisserman, A. (2003). *Multiple View Geometry*. Cambridge University Press. (page 10)

[25] Henry, P., Krainin, M., Herbst, E., Ren, X., and Fox, D. (2010). Rgb-d mapping: Using depth cameras for dense 3d modeling of indoor environments. In *International Symposium on Experimental Robotics (ISER)*. Citeseer. (page 22)

[26] Herrera, C. D., Kim, K., Kannala, J., Pulli, K., and Heikkilä, J. (2014). Dt-slam: deferred triangulation for robust slam. In *3D Vision (3DV)*, volume 1, pages 609–616. IEEE. (page 23)

[27] Horn, B. K. and Schunck, B. G. (1981). Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203. (page 21)

[28] Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., et al. (2011). Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Symposium on User Interface Software and Technology (UIST)*, pages 559–568. ACM. (page 24)

[29] Kahler, O., Adrian Prisacariu, V., Yuheng Ren, C., Sun, X., Torr, P., and Murray, D. (2015). Very high frame rate volumetric integration of depth images on mobile devices. *Transactions on Visualization and Computer Graphics (TVCG)*, 21(11):1241–1250. (page 1, 2, 26, 27, 33, 47, 63)

[30] Karami, E., Prasad, S., and Shehata, M. (2015). Image matching using sift, surf, brief and orb: Performance comparison for distorted images. *Newfoundland Electrical and Computer Engineering Conference (NECEC)*. (page 18)

[31] Kerl, C., Sturm, J., and Cremers, D. (2013a). Dense visual slam for rgb-d cameras. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 2100–2106. IEEE. (page 2, 22, 47)

[32] Kerl, C., Sturm, J., and Cremers, D. (2013b). Robust odometry estimation for rgb-d cameras. In *International Conference on Robotics and Automation (ICRA)*, pages 3748–3754. IEEE. (page 22)

[33] Klein, G. and Murray, D. (2007). Parallel tracking and mapping for small ar workspaces. In *International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 225–234. IEEE. (page 23)

[34] Klingensmith, M., Dryanovski, I., Srinivasa, S., and Xiao, J. (2015). Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields. In *Robotics: Science and Systems*, volume 4. (page 30)

[35] Lourakis, M. I. (2005). A brief description of the levenberg-marquardt algorithm implemented by levmar. *Foundation of Research and Technology*, 4(1). (page 30)

[36] Lowe, D. G. (1999). Object recognition from local scale-invariant features. In *Computer Vision, 1999*, volume 2, pages 1150–1157. IEEE. (page 17)

[37] Lucas, B. D., Kanade, T., et al. (1981). An iterative image registration technique with an application to stereo vision. (page 21)

[38] Mathworks (2018). What is camera calibration? `https://de.mathworks.com/help/vision/ug/camera-calibration.html`. [Accessed 02-January-2018]. (page 10)

[39] Moravec, H. P. (1980). Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical report, Stanford U. (page 18)

[40] Moré, J. J. (1978). The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer. (page 30)

[41] Mur-Artal, R., Montiel, J. M. M., and Tardos, J. D. (2015). Orb-slam: a versatile and accurate monocular slam system. *Transactions on Robotics (T-RO)*, 31(5):1147–1163. (page 19)

[42] Mur-Artal, R. and Tardós, J. D. (2017). ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras. *Transactions on Robotics (T-RO)*, 33(5):1255–1262. (page 2, 23, 33, 47, 63)

[43] Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohi, P., Shotton, J., Hodges, S., and Fitzgibbon, A. (2011). Kinectfusion: Real-time dense surface mapping and tracking. In *International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 127–136. IEEE. (page 1, 2, 24, 26, 27, 30)

[44] Nießner, M., Zollhöfer, M., Izadi, S., and Stamminger, M. (2013). Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, 32(6):169. (page 25)

[45] Nistér, D., Naroditsky, O., and Bergen, J. (2004). Visual odometry. In *Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages I–I. Ieee. (page 18)

[46] Orbbec3d (2018). Orbbec astra pro. `https://orbbec3d.com/product-astra-pro/`. [Accessed 02-January-2018]. (page 8, 10)

[47] Poppinga, J., Vaskevicius, N., Birk, A., and Pathak, K. (2008). Fast plane detection and polygonalization in noisy 3d range images. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 3378–3383. IEEE. (page 30)

[48] Prisacariu, V. A., Kähler, O., Cheng, M. M., Ren, C. Y., Valentin, J., Torr, P. H., Reid, I. D., and Murray, D. W. (2014). A framework for the volumetric integration of depth images. (page 35)

[49] Prisacariu, V. A., Kähler, O., Golodetz, S., Sapienza, M., Cavallari, T., Torr, P. H., and Murray, D. W. (2017). Infinitam v3: A framework for large-scale 3d reconstruction with loop closure. *arXiv preprint arXiv:1708.00783*. (page 27)

[50] Rosin, P. L. (1999). Measuring corner properties. *Computer Vision and Image Understanding*, 73(2):291–307. (page 18)

[51] Rosten, E. and Drummond, T. (2006). Machine learning for high-speed corner detection. *European Conference on Computer Vision (ECCV)*, pages 430–443. (page 17)

[52] Rublee, E., Rabaud, V., Konolige, K., and Bradski, G. (2011). Orb: An efficient alternative to sift or surf. In *International Conference on Computer Vision (ICCV)*, pages 2564–2571. IEEE. (page 18, 19)

[53] Rusinkiewicz, S. and Levoy, M. (2001). Efficient variants of the icp algorithm. In *3-D Digital Imaging and Modeling (3DIM)*, pages 145–152. IEEE. (page 20, 21)

[54] Rusu, R. B., Blodow, N., and Beetz, M. (2009). Fast point feature histograms (fpfh) for 3d registration. In *International Conference on Robotics and Automation (ICRA)*, pages 3212–3217. IEEE. (page 16)

[55] Scaramuzza, D. and Fraundorfer, F. (2011). Visual odometry [tutorial]. *Robotics & automation magazine*, 18(4):80–92. (page 19)

[56] Schnabel, R., Wahl, R., and Klein, R. (2007). Efficient ransac for point-cloud shape detection. In *Computer Graphics Forum*, volume 26, pages 214–226. Wiley Online Library. (page 30)

[57] Se, S. and Brady, M. (2002). Ground plane estimation, error analysis and applications. *Robotics and Autonomous Systems*, 39(2):59–71. (page 30)

[58] Shih, S.-W., Hung, Y.-P., and Lin, W.-S. (1995). When should we consider lens distortion in camera calibration. *Pattern recognition*, 28(3):447–461. (page 12)

[59] Steinbrücker, F., Sturm, J., and Cremers, D. (2011). Real-time visual odometry from dense rgb-d images. In *Computer Vision Workshops (ICCV Workshops)*, pages 719–722. IEEE. (page 22)

[60] Steinbrücker, F., Sturm, J., and Cremers, D. (2014). Volumetric 3d mapping in real-time on a cpu. In *International Conference on Robotics and Automation (ICRA)*, pages 2021–2028. IEEE. (page 24, 25)

[61] Sturm, J., Engelhard, N., Endres, F., Burgard, W., and Cremers, D. (2012a). A benchmark for the evaluation of rgb-d slam systems. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 573–580. IEEE. (page 47)

[62] Sturm, J., Engelhard, N., Endres, F., Burgard, W., and Cremers, D. (2012b). A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*. (page 48)

[63] Von Gioi, R. G., Jakubowicz, J., Morel, J.-M., and Randall, G. (2012). Lsd: a line segment detector. *Image Processing On Line*, 2:35–55. (page 16)

[64] Wang, C.-C., Thorpe, C., and Thrun, S. (2003). Online simultaneous localization and mapping with detection and tracking of moving objects: Theory and results from a ground vehicle in crowded urban areas. In *International Conference on Robotics and Automation (ICRA)*, volume 1, pages 842–849. IEEE. (page 19)

[65] Wang, C.-C., Thorpe, C., Thrun, S., Hebert, M., and Durrant-Whyte, H. (2007). Simultaneous localization, mapping and moving object tracking. *The International Journal of Robotics Research*, 26(9):889–916. (page 19)

[66] Weilharter, R., Schenk, F., and Fraundorfer, F. (2018). Globally consistent dense real-time 3d reconstruction from rgbd data. *Workshop of the Austrian Association for Pattern Recognition (OAGM)*. (page 3, 63)

[67] Werner, D., Al-Hamadi, A., and Werner, P. (2014). Truncated signed distance function: experiments on voxel size. In *International Conference Image Analysis and Recognition (ICIAR)*, pages 357–364. Springer. (page 29)

[68] Whelan, T. (2018). Icpcuda. `https://github.com/mp3guy/ICPCUDA`. [Accessed 20-March-2018]. (page 47)

[69] Whelan, T., Kaess, M., Fallon, M., Johannsson, H., Leonard, J., and McDonald, J. (2012). Kintinuous: Spatially extended kinectfusion. (page 24)

[70] Whelan, T., Salas-Moreno, R. F., Glocker, B., Davison, A. J., and Leutenegger, S. (2016). Elasticfusion: Real-time dense slam and light source estimation. *International Journal of Robotics Research (IJRR)*, 35(14):1697–1716. (page 22, 24, 26)

[71] Zeng, M., Zhao, F., Zheng, J., and Liu, X. (2013). Octree-based fusion for realtime 3d reconstruction. *Graphical Models*, 75(3):126–136. (page 25)

[72] Zhang, Y., Xu, W., Tong, Y., and Zhou, K. (2015). Online structure analysis for real-time indoor scene reconstruction. *ACM Transactions on Graphics (ToG)*, 34(5):159. (page 30)