Master Thesis

Coupling ICP and Whole Image Alignment for Real-time Camera Tracking

Nikolaus Mayer

January 31, 2014

University of Freiburg
Faculty of Engineering
Department of Computer Science

Processing period

August 01, 2013 – January 31, 2014

Examiners

Prof. Dr. Thomas Brox

Prof. Dr. Matthias Teschner

Advisor

Benjamin Ummenhofer

Erklärung / Declaration

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Abstract

Tracking a camera's movement through space while compiling a map of the observed environment, commonly called Simultaneous Localization And Mapping (SLAM), is a central task in robotics and computer vision. Many variants of this task exist, depending on the available sensor data, whether it has to run at interactive rates or with offline data, and the desired properties of the resulting scene map.

A particularly challenging form of SLAM, in which the only available data source is a video stream from a single-lens RGB camera that has to be tracked live, is called monocular SLAM or realtime Structure from Motion. It combines the difficulties of unknown camera pose, unknown structure of the observed scene, and tight bounds on computation time. While being hard, monocular SLAM is also especially interesting, as RGB cameras are ubiquitously available and allow for virtually limitless applications.

This work combines two approaches for accurate tracking in a monocular SLAM setting: First, the physical camera is tracked in realtime using a semi-dense frame-to-keyframe scheme. By using keyframe depth maps which are dense only where the corresponding input color image exhibits significant visual structure, this scheme minimizes the time spent on data that does not contribute useful tracking information. Areas displaying little or no structure are ignored. Second, newly constructed depth maps are aligned to the existing collection of depth maps using a variant of ICP that is robust with respect to incomplete overlap between data sets, which helps reduce camera drift left by frame-to-keyframe tracking.

The tracked input frames are used in realtime to incrementally estimate dense depth maps for new keyframes, thus extending the model and allowing the camera to explore more parts of the scene.

Deutsche Zusammenfassung

Die als SLAM (*Simultaneous Localization And Mapping*) bekannte Problemstellung, die Bewegung einer Kamera im dreidimensionalen Raum zu verfolgen und gleichzeitig ein Modell der beobachteten Umgebung zu erstellen, ist eine der zentralen Aufgaben in den Bereichen des maschinellen Sehens und der Robotik. Der Schwierigkeitsgrad dieses Problems ist unter anderem abhängig von den verfügbaren Sensordaten, den gewünschten Eigenschaften des erstellten Umgebungsmodells, sowie davon, ob die Aufgabe in Echtzeit gelöst werden muss.
Im speziellen Fall des sogenannten *monocular SLAM* oder *realtime Structure from Motion* ist eine herkömmliche Videokamera die einzige Datenquelle, und ihre Position und Lage müssen in Echtzeit bestimmt werden. Diese Einschränkung ist eine besondere Herausforderung, da sowohl Position und Lage der Kamera als auch die Geometrie der Umgebung unsicher sind, jedoch beides zugleich optimiert werden soll und zudem durch die Echtzeitanforderung die Berechnungen innerhalb kürzester Zeit ausgeführt werden müssen. Diese Form von SLAM hat in den letzten Jahren an Bedeutung gewonnen, da sie durch die zunehmende Verbreitung von RGB-Kamerasystemen insbesondere in Smartphones und Tablets großes Anwendungspotenzial bietet.
In der vorliegenden Arbeit werden zwei komplementäre Ansätze kombiniert, um die Kameralokalisierung im *monocular SLAM* zu verbessern: Zum einen wird die Kamerapose in Echtzeit mithilfe eines semi-dichten Frame-zu-Keyframe-Verfahrens bestimmt. Die benutzten Tiefenkarten enthalten dichte Information, jedoch nur für Pixelregionen, die im entsprechenden Farbbild Gradienten aufweisen. Bildsegmente ohne visuelle Struktur werden ignoriert, wodurch weniger Rechenzeit auf Daten entfällt, die wenige oder keine für das Tracking interessanten Daten enthalten. Zum anderen werden neue Tiefenkarten mithilfe einer Variante von ICP (*Iterative Closest Point*), die robust gegenüber nur teilweiser Überlappung der einzelnen Datensätze ist, an einem globalen Modell aller bereits bestehenden Tiefenkarten ausgerichtet.
Parallel zum Tracking werden in Echtzeit weitere dichte Tiefenkarten inkrementell aus den Eingabebildern und ihren berechneten Kameraposen erstellt. Sie bilden neue Keyframes, welche unmittelbar das Umgebungsmodell erweitern und der Kamera die weitere Exploration der Szene ermöglichen.

Acknowledgments

I thank Benjamin Ummenhofer for his continuing and unfailing support throughout the work on this thesis. His always insightful comments and suggestions highly influenced both design and performance of the presented system.

I also want to thank the Testo AG[1)] for supporting my work on this thesis.

[1)] `http://www.testo.de/`

Contents

1 Introduction

Simultaneous Localization and Mapping (SLAM) is one of the most iconic applications in the field of robotics and computer vision: A sensor-equipped device moves through space, and the task to be solved is to accurately track the device's position and attitude—its *pose*—relative to some frame of reference (Localization) while at the same time constructing a model of the observed scene (Mapping).

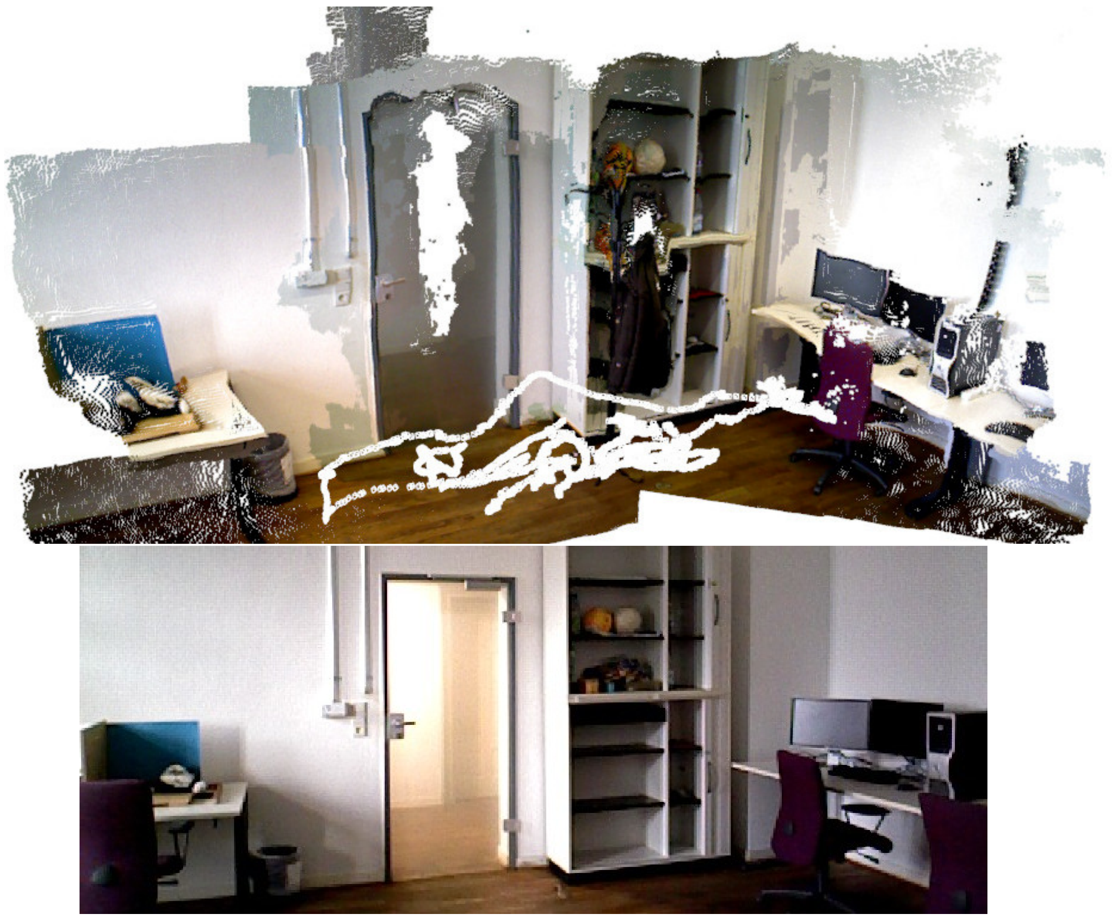


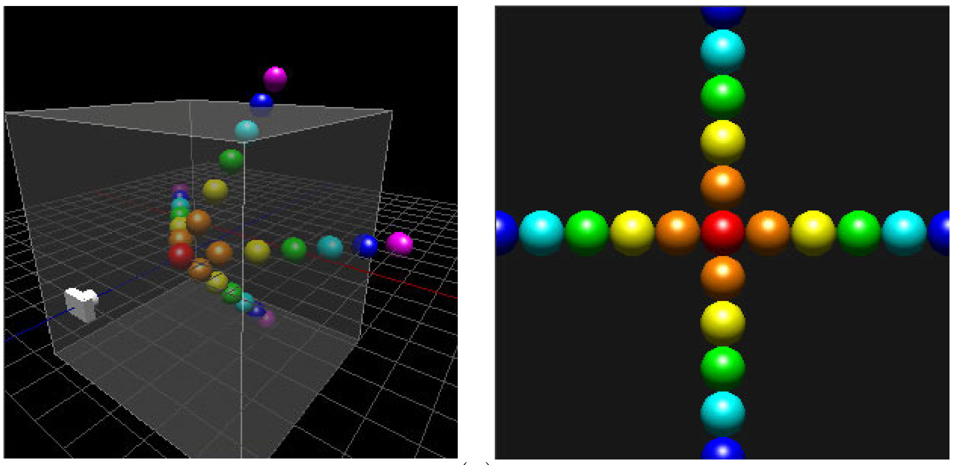
Figure 1.1: *Top:* Scene view reconstructed in our program. Multiple dense textured depth maps from different view points form a consistent scene model. The white curve along the bottom displays the live camera trajectory. *Bottom:* The same scene in the real world (not recorded at the same time, hence the differences in lighting and the missing hat stand in the bottom image).

SLAM can be widely applied and incorporates two distinct aspects: **Determining the sensor's position and attitude in space over time**—also known as odometry, (3D) tracking or ego-motion of the sensor—is crucial to tasks like robot navigation and, more recently, augmented reality programs. Retrieving this information from visual cues alone, i.e. from a live RGB video feed, is particularly interesting as the continued development of better and cheaper RGB cameras for the use in consumer electronics such as smart phones and tablet devices is making high-quality sensors ubiquitous. Using commodity hardware available practically everywhere opens up a vast range of possible applications for SLAM. Not requiring dedicated or even stationary external sensors also is an advantage for robotic platforms, as it directly translates to less weight and volume as well as increased flexibility. The specific variant of SLAM using only visual input data is called **monocular SLAM** (it uses one single-lens camera, as opposed to binocular or multiview systems) and was first described by Davison in 2003 under the name MonoSLAM [8, 3].

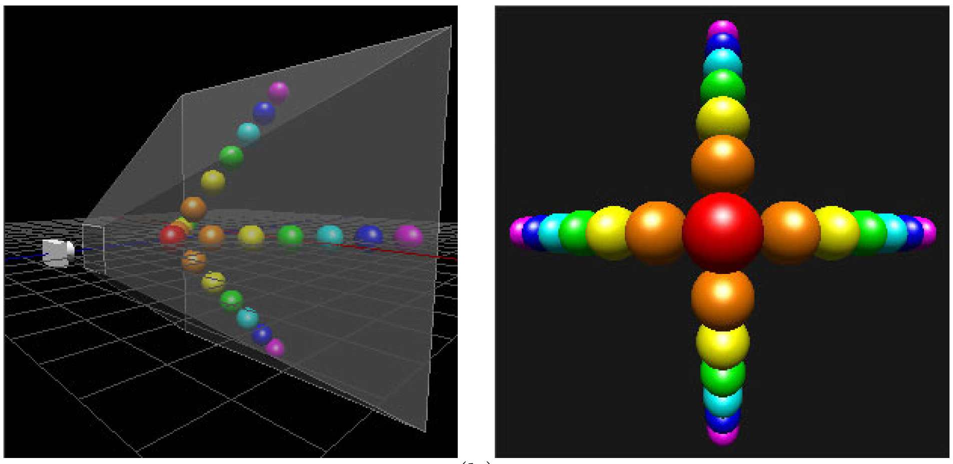
The second aspect of SLAM is **reconstructing the observed scene's three-dimensional structure**. Dense or even textured 3D models are needed in many fields, among them autonomous robotic mapping of new environments, medical imaging, model acquisition for animation and rendering in computer graphics, or 3D printing. They are also required for augmented reality if virtual objects should interact with the real world (as seen by the user) in a plausible way, for example balls bouncing off walls or being occluded by (real) furniture. Extracting geometry from only visual information is known as Structure from Motion. It relies on the movement (or illusion thereof) which an object displays on an imaging sensor when seen from different viewpoints or panning the camera, called stereopsis or binocular disparity, and on the apparent change in the object's size depending on its distance to the imaging sensor, caused by non-parallel projection (Figure 1.2). The same principles allow humans and animals to see with depth perception. Having two or more eyes enables stereoscopic vision, and since the eyes usually do not have infinite focal length, they automatically project with perspective distortions. In this aspect, a monocular SLAM system can replace complex 3D scanning rigs. Requiring neither multiple sensors with precisely known relative pose and large baseline nor depth sensors such as time-of-flight (TOF) scanners or structured light projectors, a 3D reconstruction system based on monocular SLAM can be contained in a much smaller and much cheaper package. Additionally, such a system does not have to modify the scene, as do approaches based on structured light, and can deal with a greater variety of scenes compared to fixed scanning rigs which are usually limited to single objects of a certain size[1)].

Structure from Motion is not intrinsically an online task and can essentially be

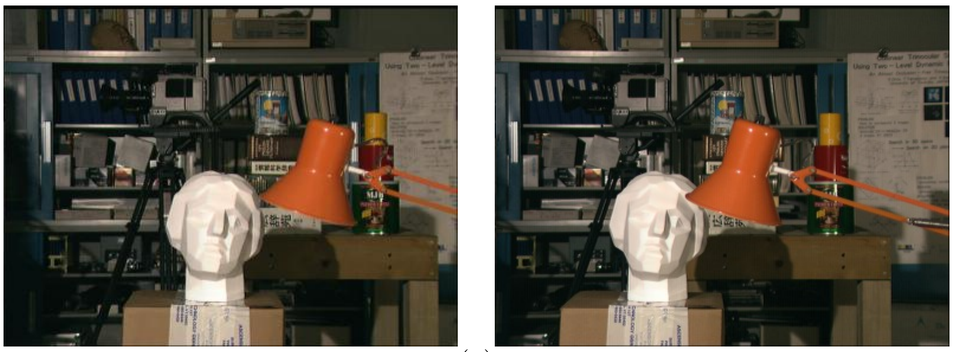
[1)]Each approach has limitations that do not hinder other methods; for example, laser scanners are only usable up to a certain distance, infrared light projectors (such as used by the Kinect system) are blinded in unfiltered sunlight, and RGB cameras of course do not work in the dark. All of these in turn rely on good visibility which is not generally necessary for radar-based range sensing, and so on.



(a)



(b)



(c)

Figure 1.2: Inferring geometry from visual clues: ((a),(b) *Left:* Scene with view frustum, *right:* Scene as seen by the camera.) (a) Parallel projection (infinite focal length): All spheres' projections are the same size. (b) Perspective projection (finite focal length): Objects closer to the camera appear larger. (c) When panning the camera, nearer objects display larger apparent motion than farther objects. These phenomena are used to draw conclusions about the scene's geometry. Images taken from [12] ((a), (b); original data from [14]) and courtesy of the University of Tsukuba ((c), see [46]).

performed on any collection of images [38, 40, 41]. However, it can be used in combination with a tracker to build a SLAM system. In this context, monocular SLAM is also known as **realtime Structure from Motion (SfM)**. The SLAM concept encompasses a broad range of specific problem settings with varying degrees of difficulty, depending on the available data and the desired scene model.

A great difficulty in SLAM is that a mutual dependency between pose tracking and scene mapping arises: To determine the sensor's pose, the geometric structure of the visible scene has to be known with a certain accuracy (in the respective representation, for example keypoints, a triangle mesh or a dense point cloud). Otherwise, it is unknown how sensor motion relates to changes in the appearance of the scene, which is a necessity for tracking. Conversely, combining individual measurements to a consistent model is not feasible until it is known which measurement was taken from which sensor pose.
The interdependency between tracking and mapping also makes some sort of initialization procedure necessary. This is decidedly easier when using e.g. a laser scanner or a depth camera, which can immediately provide reliable information about the scene geometry, than in a purely visual setting where even the first scene model has to be actively constructed.
In monocular SLAM both parts (tracking or mapping) can serve as starting points, but either knowledge about the scene geometry or a set of input data with known poses have to be given[2)]. From this, the SLAM system can then bootstrap itself. After the initialization, it is self-supporting: The tracker enables the computation of new geometry information via SfM, which in turn allows the camera to be tracked through scene views for which no structure was previously known. Over time, this allows the camera (or robot) to explore more and more of its environment.

In this work we implement and test a monocular SLAM system. We combine two strategies to ensure both local and global tracking accuracy: First, to track the camera on its path through space, the system utilizes semi-dense whole-image alignment. Compared to feature-based approaches, this avoids having to rely on the scene exhibiting a sufficient number of feature points (conforming to the specific definition of what even can be a feature), and feature extraction and matching performing adequately. On the other hand, fully dense approaches waste computational resources processing featureless image regions which do not contribute to better camera tracking. By walking a path between these two extremes and focusing on dense areas with sufficient visual structure, the system presented in this work maximizes the information gain per time unit, as less time is spent on pixels without useful data.
Second, the system maintains a global model of the static scene observed by the camera. It continuously generates partial dense 3D reconstructions, which are then aligned to and merged with the existing model. This corrects camera drift which might accumulate in the tracking steps, and aids loop closure.

This remainder of this work is structured as follows: After reviewing previous related

[2)]However, this initialization can be done visually as well, see section 7.1.

work in chapter 2, we introduce notations and concepts in chapter 3. In chapter 4 we give an overview of different approaches to represent knowledge about the environment, specifically the depth maps that we use to track the camera in chapter 5. The mapping process from which new depth maps are obtained is laid out in chapter 6. Finally, after the system's initialization is shortly described in chapter 7, we finish with some details of our implementation as well as our experiments in chapter 8 and a conclusion in chapter 9.

2 Related work

Dense depth reconstruction with only visual input was first done in realtime by Stühmer *et al.* [37] and, at the same time, by Newcombe and Davison [25], after much earlier non-dense work by Davison in [8, 3].

Our main paper for this work is the seminal DTAM introduction by Newcombe, Lovegrove and Davison [1]. It was the first to reach realtime performance in combining dense 3D geometry reconstruction and tracking using only an RGB camera. From their paper we have adopted the realtime tracking approach as well as the framework for depth map estimation by accumulation of photometric matching errors in a cost volume.

The DTAM system and [37] utilize (as do many more) the earlier PTAM work by Klein and Murray [6], who tackled SLAM by running their sparse feature-based tracking in parallel with the mapping task. PTAM focuses on small work spaces for augmented reality programs. Together with Castle, in [7] Klein and Murray extended their own approach to larger environments.

The concept of semi-dense visual tracking to maximize information gain was introduced in [17] by Engel, Sturm and Cremers, although they use it in a frame-to-frame setting that does not build a global model.

While the purely monocular setting is at the heart of this work, there are other approaches that utilize more available information: In [19], Pollefeys *et al.* incorporate GPS and inertia measurements for a more precise model in large-scale urban 3D reconstruction. This specific target allows them to use plane priors (as in [20]) for the reconstructed depths to better fit doors, building fronts and such. From their work we adopt the idea of confidence-score based filtering of our depth reconstructions.
The Kinect RGB-D camera system made popular by Microsoft is used by Henry *et al.* [23] in a sparse tracking scheme combined with ICP, and later by Newcombe *et al.* [2] to fuse hundreds of depth measurements into a volumetric SDF model. The latter render the model using raycasting and obtain dense novel point cloud views of the estimated surface, which in turn are used to track the camera by aligning live depth measurements to renderings via ICP.
In [4, 5], Whelan *et al.* expand the work of [2] to spatially extended environments such that the camera is no longer bound to a fixed model volume. RGB-D cameras are also used by Kerl *et al.* [18] for dense tracking.

The combination of individual range measurements in a volumetric scheme is also done by Steinbrücker *et al.* [39, 36] in a SLAM setting, and without focus on tracking by Curless and Levoy [32]. The fusion of multiview range data itself goes back much further and has for a long time been used to acquire 3D models, for example by Chen and Medioni [24] (introducing the concept of point-plane metric later used by [2]), Blais and Levine [27] or Trucco *et al.* [29] (with a robust form of ICP that we also use in this work), and later Rusinkiewicz *et al.* [22] (using a structured light projector).

Structure from Motion has also been done on very large scale and diverse data sets, such as in the reconstruction of popular landmark structures from internet image databases by Snavely *et al.* [38], Agarwal *et al.* [40] and Frahm *et al.* [41].

In [9], Ummenhofer and Brox couple optical-flow tracking from [10], a sparse reconstruction via bundle adjustment and finally a dense volumetric optimization to iteratively refine both camera poses and model surfaces. Using an RGB camera, they obtain dense surfaces without holes.

The field also expands by exploring more specialized aspects, such as the reconstruction of non-rigid surfaces by Garg *et al.* [47] or thin objects by Ummenhofer and Brox [31] from monocular video. SLAM systems usually assume static scenes and Lambertian surfaces (brightness constancy) which makes the easing of such restrictions interesting. Jachnik *et al.* aim in a similar direction with their work in [48], estimating light fields of planar specular (non-Lambertian) surfaces for more realistic and immersive rendering in augmented reality.

3 Camera and camera pose

In this chapter, we introduce the notion of *camera* as it is used throughout this work, namely a sensor which projects visible light onto a two-dimensional image plane. We describe how the position of objects and the camera influences their projection on the image plane. This is the single most essential concept for our work, as it lays the foundations for both main parts of our SLAM algorithm, camera tracking and scene mapping. We make a number of simplifying assumptions about the properties of the camera sensor to make computations feasible, and describe how these properties can then be summarized in few parameters. Related to this, we quickly introduce the concept of *homogeneous coordinates*, a way of expressing points in Euclidean (2D or 3D) space that helps avoid costly nonlinear computation steps.

Most of the material is found in the lecture notes of [11, 12, 13]. It defines notation and concepts needed for the rest of this thesis, mainly procedures for tracking (chapter 5) and mapping (chapter 6).

3.1 Theoretical model

A camera is an imaging device. It contains a planar, rectangular photosensitive sensor divided into square pixels, usually subpixel assemblies for primary colors from which an RGB color is interpolated. Light reflected off objects traverses optic lenses mounted in front of the sensor and hits the sensor, where it produces an image. The perspective that this image depicts depends on thickness and shape of the lenses and the camera's aperture, which is the size of a circular hole in an opaque plane parallel to the imaging sensor, mounted in front of the sensor, and determines the cone angle of the light that traverses towards the sensor. Lenses additionally distort images, depending on their material and grade, and produce chromatic aberrations (different refraction depending on wavelength). Except for wide-angle lenses, these effects are usually minor. For this reason, they are often ignored, resulting in what is called a *thin lens model* with no aberrations or distortions.

A further simplification assumes that the aperture is infinitely small, which corresponds to the *pinhole camera model.* The smaller the aperture, the smaller is the influence of the projected object's distance to the camera on its sharpness/focus. With a hypothetical point-shaped aperture everything is in focus, i.e. the sharpness of an object no longer depends on its distance. This significantly reduces the complexity of the model.

In the following, the camera center defines the origin of $\mathbb{R}^3$. The image plane is parallel to the X-Y plane, and the camera “points” in positive Z-direction. The pinhole camera model originally projects onto a plane behind the aperture, where the resulting image is inverted (see Figure 3.1). However, in the simplified camera model the image plane can equivalently be placed *in front* of the aperture, and the image is then upright.

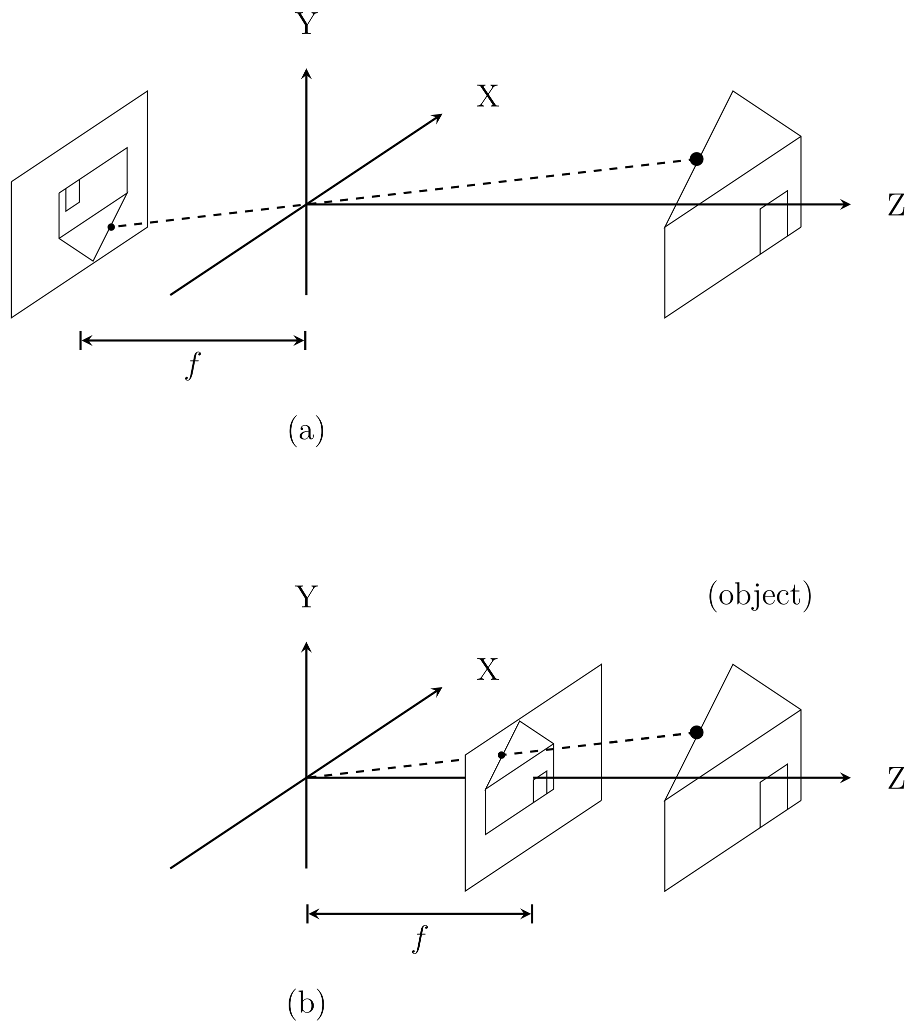


Figure 3.1: (a) The original pinhole camera model, where the projection is inverted. (b) The equivalent model with the image plane in viewing direction. The projected image is upright.

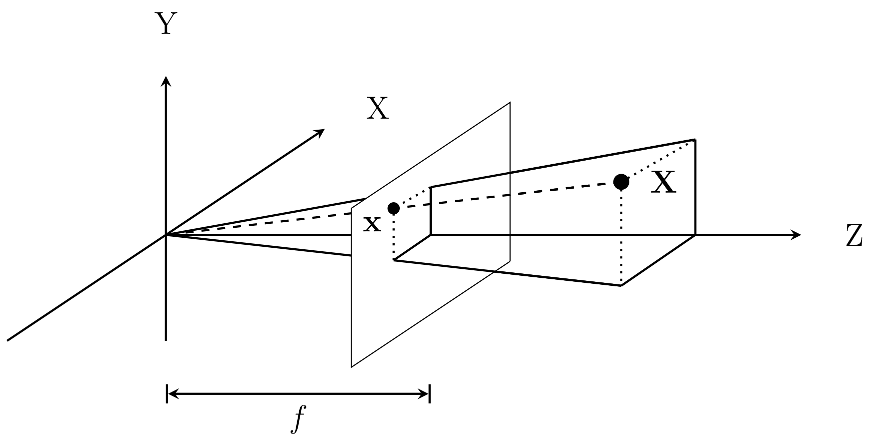


Figure 3.2: Visualization of Equation 3.1

Projection to the image plane

In the pinhole camera model, a 3D point in Euclidean coordinates $\mathbf{X} = (X, Y, Z)^\top \in \mathbb{R}^3$ is projected onto the image plane and mapped to the 2D pixel $\mathbf{x} = (x, y)^\top \in \mathbb{R}^2$. The pixel coordinates $\mathbf{x}$ relate to $\mathbf{X}$ depending on the camera's focal length f, visualized in Figure 3.2:

$$x = f\frac{X}{Z},\ y = f\frac{Y}{Z} \tag{3.1}$$

Note that an object's projected image on the image plane becomes larger as the object moves towards the camera (compare Figure 1.2), as is characteristic of a *perspective projection*[1)].

3.2 Homogeneous coordinates

The perspective projection in Equation 3.1 is a nonlinear operation due to the division by Z. This nonlinearity is undesirable when considering computational costs,

[1)] The counterpart is *parallel* projection, where the focal length f is infinite. This results in the object's projection always having the same size, regardless of the object's distance. Parallel projection is useful for many applications, for example architectural plans or maps, as it preserves relative sizes and the appearance of an object is invariant to camera translation. Here however, we explicitly *use* the fact that non-parallel projection does not preserve size.

but can be avoided by switching from Euclidean to *homogeneous coordinates* which simply adds a 1 to the end of a coordinate vector:

$$\text{2D pixel: } (x, y)^\top \in \mathbb{R}^2 \longrightarrow (x, y, 1)^\top \in \mathbb{P}^2. \tag{3.2}$$

$$\text{3D point: } (X, Y, Z)^\top \in \mathbb{R}^3 \longrightarrow (X, Y, Z, 1)^\top \in \mathbb{P}^3. \tag{3.3}$$

Contrary to Euclidean space, homogeneous points are not unambiguous. For example, $(2, 2, 1, 2)^\top$ and $(\frac{1}{2}, \frac{1}{2}, \frac{1}{4}, \frac{1}{2})^\top$ represent the same Euclidean point $(2, 2, 1)^\top \in \mathbb{R}^3$. In fact, in the projective space $\mathbb{P}^3$ all homogeneous points $(\lambda X, \lambda Y, \lambda Z, \lambda)^\top \in \mathbb{P}^3$ with $\lambda \in \mathbb{R} \setminus \{0\}$ belong to the same equivalence class[2)]. The obvious representative for the class is $(X, Y, Z, 1)^\top \in \mathbb{P}^3$ which directly contains the unique Euclidean point $(X, Y, Z)^\top \in \mathbb{R}^3$:

$$(x', y', \lambda)^\top \in \mathbb{P}^2, \lambda \neq 0 \longrightarrow \left(\frac{x'}{\lambda}, \frac{y'}{\lambda}\right)^\top \in \mathbb{R}^2 \tag{3.4}$$

$$(X', Y', Z', \lambda)^\top \in \mathbb{P}^3, \lambda \neq 0 \longrightarrow \left(\frac{X'}{\lambda}, \frac{Y'}{\lambda}, \frac{Z'}{\lambda}\right)^\top \in \mathbb{R}^3 \tag{3.5}$$

With homogeneous coordinates, the projection equation from Equation 3.1 becomes a linear operation:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}, \tag{3.6}$$

and the final pixel coordinates are

$$x = \frac{x'}{z'}, y = \frac{y'}{z'} \tag{3.7}$$

This nonlinear step to retrieve the Euclidean coordinates can often be postponed until the end. Preceding computations can be done within homogeneous space.

[2)] $\lambda = 0$ is a special case where the point is at infinity. It can be seen as a direction in $\mathbb{R}^3$ and is then called a *homogeneous vector*

3.3 Internal camera parameters

The *internal* or *intrinsic parameters* describe how the spatial positions of 3D points within the camera's coordinate system relate to their pixel coordinates when projected onto the image plane.

The pinhole camera model is an idealized assumption which can not expected to hold in physical reality. Where the model defines the principal point to be completely on the viewing axis, i.e. dead center on the image plane, this is not the case in real devices. The focal length f becomes two parameters $f_x = f \cdot m_x$ and $f_y = f \cdot m_y$, where m_x and m_y are the pixel widths in x-/y-direction. Additionally, up to now the pixel origin coincided with the principal point (the image plane origin), whereas it is common to set it into a corner which is done by shifting by the principal point's pixel coordinates (c_x, c_y). The parameter s represents the skew coefficient between the image's x- and y-axes.
These parameters fully specify the *camera calibration matrix* $\mathbf{K}$:

$$\mathbf{K} = \begin{pmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

However, even accounting for real-world devices, usually not all parameters are needed. Pixels are generally square; in this case $f_x = f_y$, and the skew parameter s is 0.

The physical camera is assumed to be *calibrated*, i.e. the internal parameters are known. If this is not the case, they have to be obtained first by performing a calibration procedure [49]. The internal parameters also do not change while the program is running, i.e. varying the camera's zoom factor is not allowed.

Projection and inverse projection

With $\mathbf{K}$, the projection $\mathbf{x} \in \mathbb{R}^2$ of a homogeneous point $\mathbf{X}' \in \mathbb{P}^3$ into the image becomes

$$\mathbf{x}' = \pi(\mathbf{X}') = \mathbf{K}'\mathbf{X}', \text{ with } \mathbf{K}' = (\mathbf{K} \,|\, \mathbf{0}) \in \mathbb{R}^{3,4} \tag{3.9}$$

The projection is denoted as function $\pi(\cdot)$. Likewise, we can define the *inverse* projection function $\pi^{-1}(\cdot,\cdot)$, which takes a homogeneous 2D point $\mathbf{x}' \in \mathbb{P}^2$ and the desired *depth* $d \in \mathbb{R}$, i.e. the Z-coordinate of the resulting Euclidean 3D point $\mathbf{X} \in \mathbb{R}^3$. The depth is necessary since all points on a line through the origin map to the same pixel, and it is thus impossible to determine the correct line position from pixel coordinates alone.

$$\mathbf{X} = \pi^{-1}(\mathbf{x}', d) = d \cdot \mathbf{K}^{-1}\mathbf{x}' \tag{3.10}$$

3.4 External camera parameters

The *external* or *extrinsic parameters* denote the coordinate transform that brings a 3D point from scene-space coordinates (relative to some arbitrary origin) into a camera's coordinates (relative to the camera's center) and/or back.

A camera's pose $\mathbf{T} \in \mathbb{R}^{4,4}$ is given by the rigid body transform specified by a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$

$$\mathbf{T} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{pmatrix}, \text{ with } \mathbf{R} \in \mathbb{SO}_3, \mathbf{t} \in \mathbb{R}^3, \tag{3.11}$$

where the camera's actual position in space $\mathbf{C} \in \mathbb{R}^3$ is computed as

$$\mathbf{C} = -\mathbf{R}^{-1}\mathbf{t} = -\mathbf{R}^\top\mathbf{t} \tag{3.12}$$

The second equivalence holds due to $\mathbf{R}$ being a rotation matrix, and thus orthogonal.

$\mathbf{T}$ transforms a point $\mathbf{p}$ from (homogeneous) camera coordinates into scene coordinates

$$\mathbf{p}_{\text{scene}} = \mathbf{R}\mathbf{p}_{\text{cam}} + \mathbf{t} = \mathbf{T}\mathbf{p}_{\text{cam}} \tag{3.13}$$

and backwards via

$$\mathbf{p}_{\text{cam}} = \mathbf{R}^\top(\mathbf{p}_{\text{scene}} - \mathbf{t}) = \mathbf{T}^{-1}\mathbf{p}_{\text{scene}}. \tag{3.14}$$

We can now define the complete projection matrix $\mathbf{P} \in \mathbb{R}^{3,4}$, which takes a homogeneous 3D point in scene coordinates and projects it into a homogeneous 2D pixel:

$$\mathbf{P} = \mathbf{K}\mathbf{M}, \text{ where } \mathbf{M} = (\mathbf{R} \,|\, \mathbf{t}) \in \mathbb{R}^{3,4} \tag{3.15}$$

3.5 Minimal parametrization of a 3D rigid body motion

Camera tracking will work by taking an initial camera pose guess and then iteratively applying incremental relative changes. The observed scene is static and the camera's internal parameters do not change, so the pose updates correspond to small 3D rigid body motions. Poses and pose updates are fundamentally the same thing: Both describe a coordinate transform relative to some origin. The difference is that poses relate to the origin of *scene* space while pose updates describe a transform relative to a *camera's* space.
Pose updates thus also consist of a rotation matrix $\mathbf{R} \in \mathbb{R}^{3,3}$ and a translation vector $\mathbf{t} \in \mathbb{R}^3$. However, this description contains *twelve* free parameters, while a 3D rigid body motion only has *six* degrees of freedom (DOF). We seek a minimal representation that only requires the estimation of exactly six parameters since such a formulation is more apt to build our system of equations for camera tracking (see section 5.2).

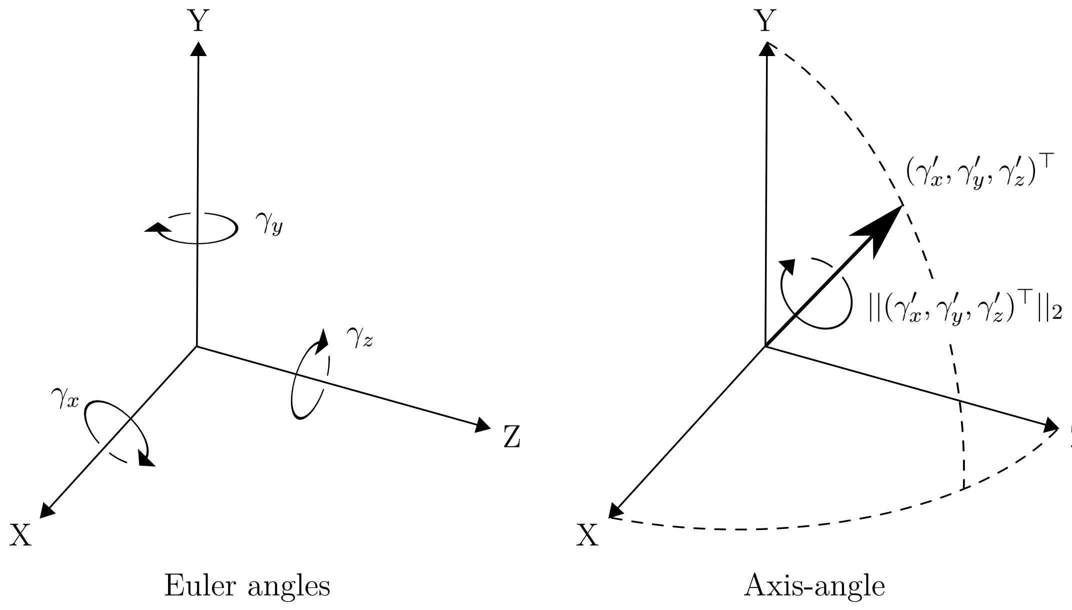

Y
γ_y
γ_z
γ_x
Z
X
Euler angles
Y
$(\gamma'_x, \gamma'_y, \gamma'_z)^\top$
$||(\gamma'_x, \gamma'_y, \gamma'_z)^\top||_2$
Z
X
Axis-angle

Figure 3.3: Euler angles and axis-angle parametrizations for the rotational part of a 3D rigid body motion.

Euler angles

We use a common parametrization called Euler angles. It composes the rotational part of the pose by concatenating three rotations, one around each of the x-, y- and z-axis. Each of these rotations needs only a single value, namely the angle. The remaining three parameters of the pose describe a translation from the origin as an offset vector in $\mathbb{R}^3$. Together, we get the six-parametrization ψ

$$\psi = (\gamma_z, \gamma_y, \gamma_x, t_x, t_y, t_z)^\top \in \mathbb{R}^6 \tag{3.16}$$

where each γ parameter controls the rotation around the axis in its subscript index. From ψ, the matrix representation $\mathbf{T}(\psi)$ can be retrieved as follows [13]:

$$\mathbf{T}(\psi) = \begin{pmatrix} \mathbf{R}(\psi) & \mathbf{t}(\psi) \\ \mathbf{0}^\top & 1 \end{pmatrix}, \text{ with} \tag{3.17}$$

$$\begin{aligned} \mathbf{R}(\psi) = & \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma_x) & -\sin(\gamma_x) \\ 0 & \sin(\gamma_x) & \cos(\gamma_x) \end{pmatrix} \cdot \\ & \begin{pmatrix} \cos(\gamma_y) & 0 & -\sin(\gamma_y) \\ 0 & 1 & 0 \\ \sin(\gamma_y) & 0 & \cos(\gamma_y) \end{pmatrix} \cdot \\ & \begin{pmatrix} \cos(\gamma_z) & -\sin(\gamma_z) & 0 \\ \sin(\gamma_z) & \cos(\gamma_z) & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and} \end{aligned} \tag{3.18}$$

$$\mathbf{t}(\psi) = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \tag{3.19}$$

Axis-angle representation

An alternative parametrization uses the *axis-angle* representation. The translational part is the same as before, but instead of encoding rotations around the three main axes, the parameters $(\gamma_x, \gamma_y, \gamma_z)^\top$ here are seen as a vector encoding the *axis* of rotation. The norm of that vector $||(\gamma_x, \gamma_y, \gamma_z)^\top||_2$ describes the magnitude of the rotation (see Figure 3.3). The axis-angle representation is used to avoid the *gimbal lock* problem associated with specific configurations in Euler angles.

In this work we store camera orientations using rotation matrices ($\mathbf{R} \in SO(3)$) since they are convenient to use in transforms (see section 3.4). Euler angles are used in pose optimization during tracking, where only small relative pose changes are involved.

4 Model representation

There are multiple ways to represent knowledge about the observed scene's geometry. Depending on the available sensor information and the used approaches to tracking and mapping, different models offer advantages and disadvantages. This chapter gives a short overview of possible scene representations including the depth maps used in this work.

Point clouds

Perhaps the simplest model is a *point cloud*, a loose collection of 3D points (usually) representing object surfaces in the scene. If range data is directly available, tracking can be done by aligning live range measurements to the model point cloud, for example using the iterative closest point (ICP) algorithm, as is done in [33] ([2] use a similar approach, but their model point cloud is created on demand (see below)). Point clouds can be colorless (a laser range scanner does not provide color information) or textured, in which case the points are annotated with color values.

Probabilistic occupancy grids

Occupancy grids [35] take a different approach to geometry representation. Instead of storing discrete point samples, an occupancy grid divides the scene space into cells, each of which knows whether the space it encloses is "empty" (a robot can traverse the cell) or "occupied" by some object, and how certain that information is. The grid is filled using range measurements, for example from a laser scanner, and multiple measurements can be fused to reduce uncertainties in the cells.

Signed distance functions

A related representation is a volumetric *signed distance function* (SDF). The model is enclosed within a 3D voxel volume (cuboid) where each voxel contains an SDF value describing the voxel's distance from the nearest object surface, and whether it is on the outside (visible) or the occluded (behind the surface as seen by the sensor). The surfaces of objects are implicitly represented by the SDF's zero crossing, which has to be interpolated from the voxel grid. This concept can be seen as an extension of an occupancy grid: Where the latter only knows about the binary states “empty” and “occupied”, a volumetric SDF stores more precise floating-point knowledge about the distance to the nearest “occupied” cell. The SDF representation enables dense visual tracking: The volume cells can hold color values in addition to distance information, which allows the volume to be rendered, using a marching-cubes algorithm or raycasting [2].

Volumetric (truncated) SDF representations have been introduced in [32], and recently used in [2] and its extensions [4, 5]. In [2] the model is built by combining depth maps retrieved from an RGB-D Kinect sensor. Prior to fusion, the individual measurements are aligned using ICP [2], sparse visual feature correspondences [4], or a combination of both [5].

Using a volumetric SDF has a key advantage: Multiple measurements from different viewing angles automatically combine and extend each other, filling holes and smoothing out depth noise [2].
A disadvantage of this approach is that the model has a fixed size ([4] and [5] use SDFs for spatially extended mapping, but the SDF volume itself stays the same size and travels along with the camera). This also has the effect that a lot of memory and computational resources are used for processing empty voxel regions: The important parts are the areas around the SDF’s zero crossings, but the actual object surfaces will likely occupy only a very small part of the volume. If the individual partial reconstructions contributing to the volume are not sufficiently aligned or imprecise, the SDF averaging can also blur the model.

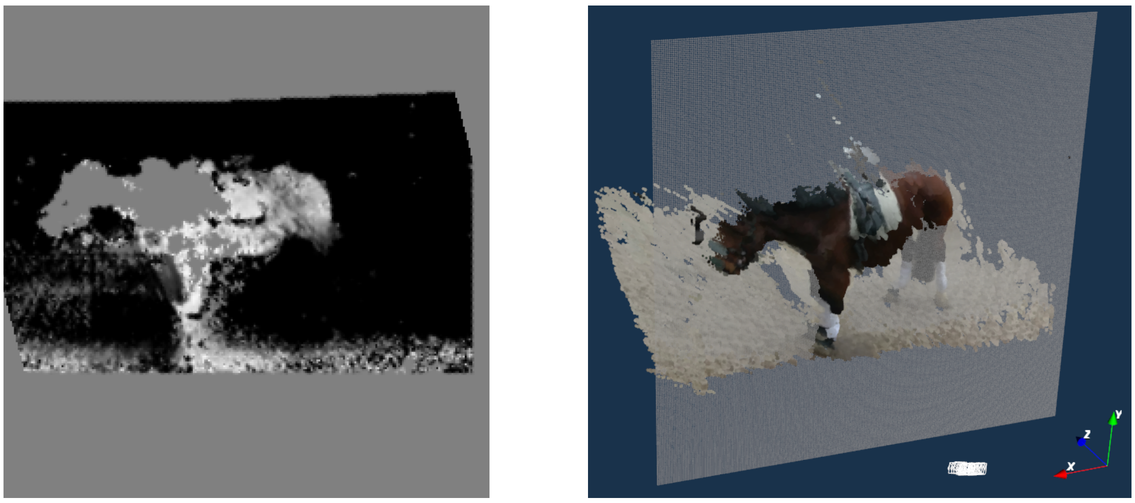


Figure 4.1: Example view using a volumetric TSDF. *Left:* A slice of the TSDF volume. Black areas are truncated “outside” values. Truncated “inside” (rather: occluded) values appear gray. The cloudy-white regions are within the truncation zone in which the zero-crossing happens. (Gray pixels surrounding the black area are voxels for which no measurement exists yet.) *Right:* The same slice visualized as intersecting the model’s surface reconstruction, in an oblique view (compare [2], Figure 4).

Octrees

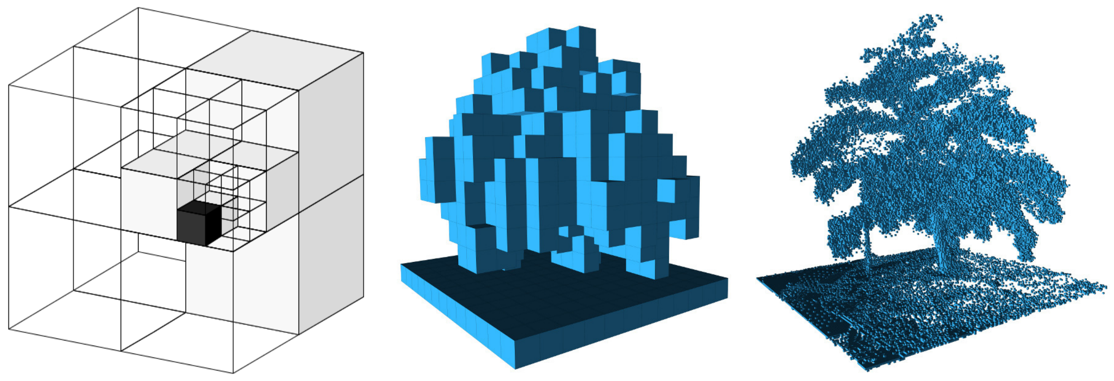


Figure 4.2: Octree / Octomap. Images from [34].

Hierarchical structures such as *octrees* (in the 3D case) can store grid cell information in a way that avoids many shortcomings of fixed grids or volumes. By locally adapting the tree depth, an octree can densely populate space where it is needed (along object surfaces). Empty regions do not use up memory, a distinct advantage compared to fixed cell arrays. Octrees are also not limited to a fixed scene size, as the tree can keep expanding wherever new information needs to be added, and they can easily provide coarser scene descriptions by cutting off finer node levels, which can be used to improve computational efficiency when only high-level information is needed.
Octrees have successfully been used for robot navigation, for example in [34].

Depth maps

We represent our model mainly by a collection of *keyframes*. A keyframe is an image taken from the stream of input color frames, and each keyframe stores its absolute camera pose and a (semi-)dense depth map.

The depth map ξ for image I has the same dimensions as I. With ξ and I combined, we can construct a 3D point cloud and render the keyframe for a different camera pose (see Figure 4.4). From 2D pixel coordinates alone, it is not possible to reconstruct a unique 3D point: All points on a line through the origin and the pixel project into this same pixel. To get a unique solution, we require a third parameter, the *depth*. Assuming the camera center $\mathbf{C}$ to define the origin, the depth value $\xi(\mathbf{x}) \in \mathbb{R}$ of a pixel $\mathbf{x} \in \Omega_I$ denotes the Z-coordinate of the 3D point $\mathbf{X}$ that is reconstructed from $\mathbf{x}$.

While the keyframes are essential for frame-to-keyframe tracking (see chapter 5), we additionally maintain a point cloud model of the scene. This model is built from all keyframes by creating textured point clouds using the semi-dense keyframe depth

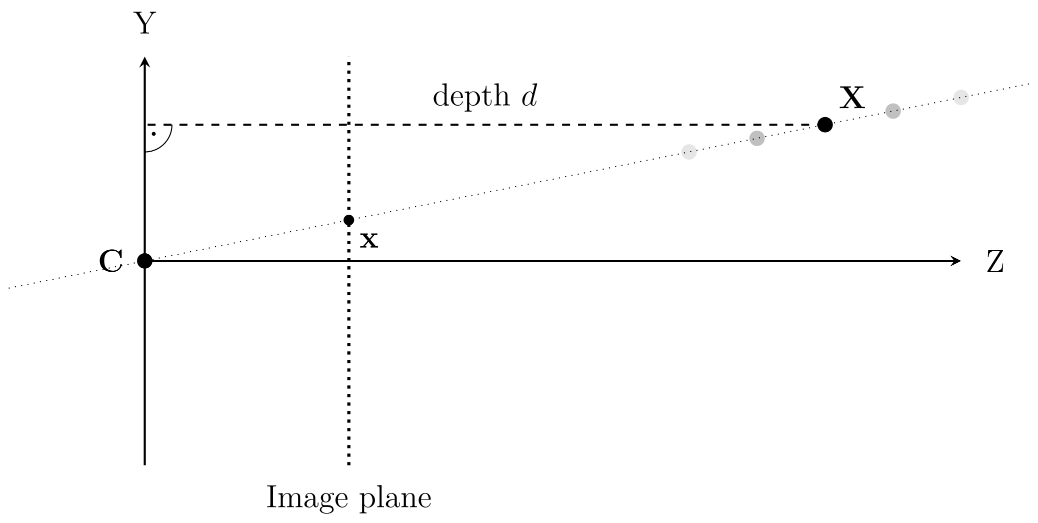
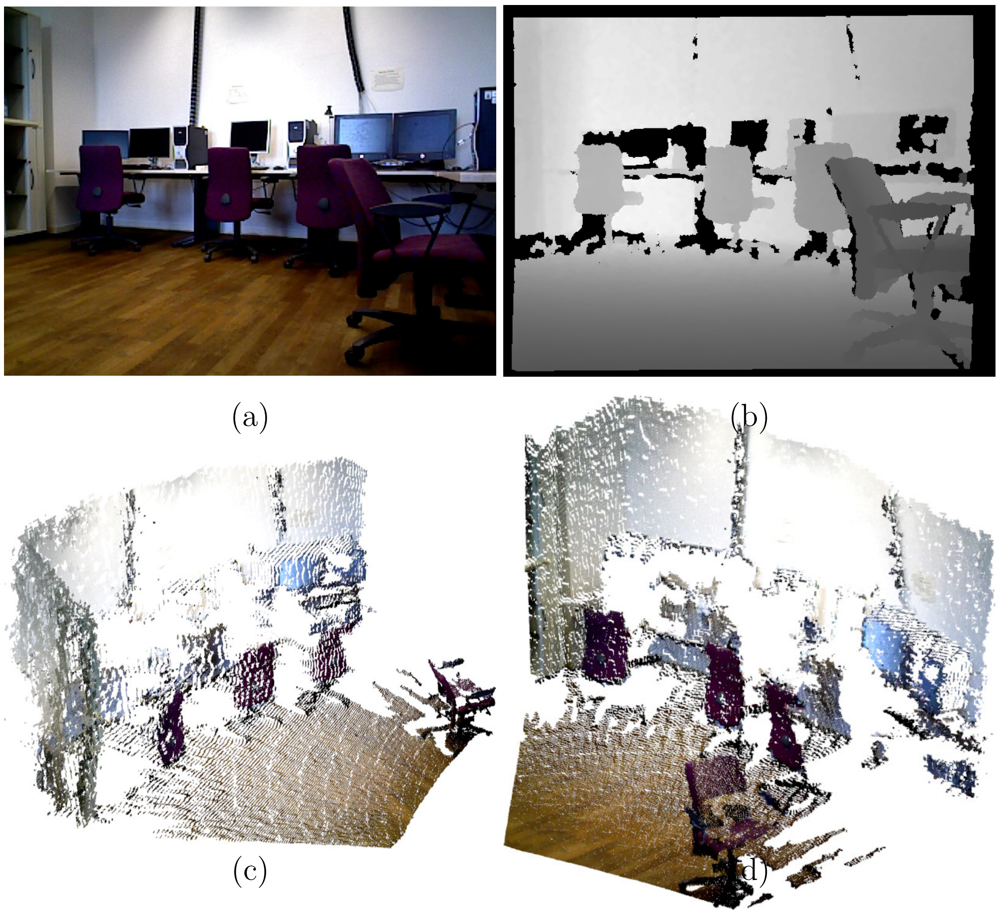


Figure 4.3: The depth d of pixel $\mathbf{x}$ is the distance of the 3D point $\mathbf{X}$ to the plane through the camera center $\mathbf{C}$ and normal to the viewing direction Z. Varying d changes the position of $\mathbf{X}$, but not its projected pixel position $\mathbf{x}$.

maps, then aligning them using ICP, and finally fusing them into a single point cloud, discarding near-duplicates of points to keep the model lean (see section 6.2).

(Triangular) Meshes

Finally, it is also possible to represent the model as a polygon mesh. The authors of DTAM [1] construct depth maps to represent the scene model, but compute triangular meshes from these for tracking. Consumer graphics cards excel in the task of rendering triangles, so they can conveniently be used for dense visual tracking purposes.



(a) (b) (c) (d)

Figure 4.4: (a) Video frame, (b) The corresponding depth map (taken from an RGB-D camera), (c)(d) Two 3D views of the reconstructed textured point cloud.

5 Camera tracking

This chapter is about camera tracking. First, we give a short overview of visual tracking and the differences between the sparse feature-based and the dense approach. We then describe our method for frame-to-keyframe tracking and motivate the use of semi-dense depth maps. Our twofold strategy for loop closure concludes this chapter.

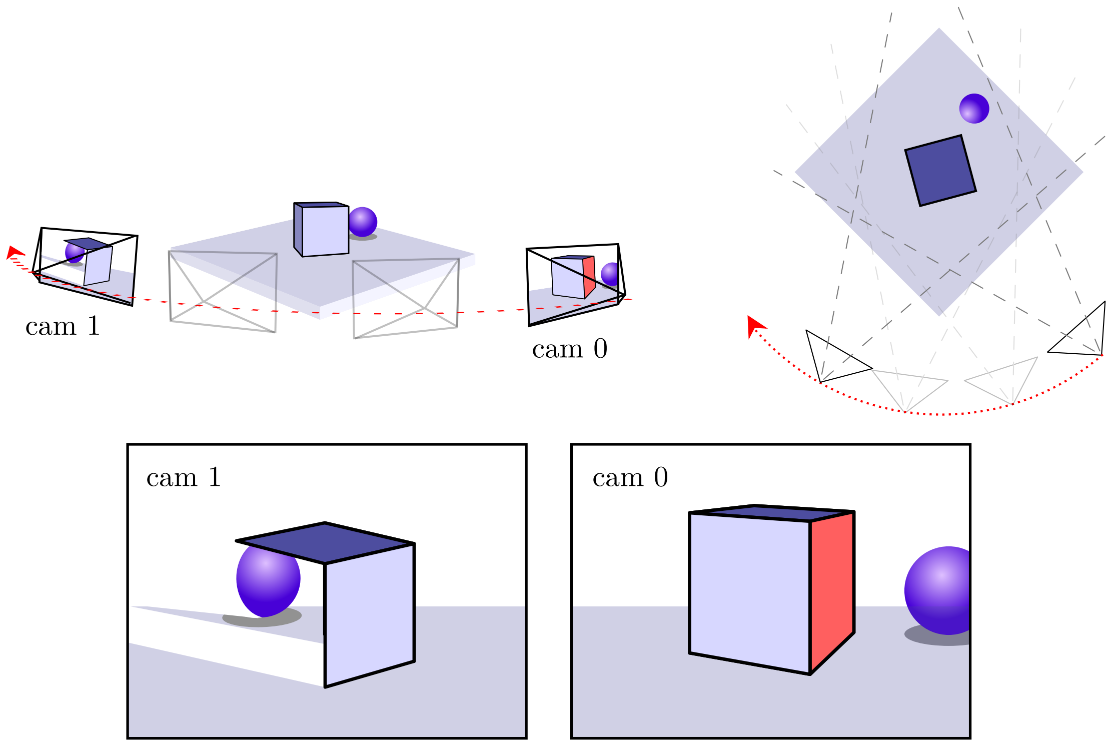


Figure 5.1: Limitations when tracking with keyframe depth maps: *Cam 0* is a keyframe, thus it gets a fully dense depth map of its view of the scene. This keyframe can be problematic when used for tracking *cam 1*, which has a very different viewpoint. The second camera observes structures that the keyframe could not see (back of the cube/sphere, base "behind" objects), and the depth map thus does not contain.

5.1 Sparse and dense tracking

Tracking, the process of continuously locating the camera as it moves around, is one of the two problems to be solved in a SLAM task. When it comes to visual SLAM, there are two main approaches to camera tracking:

The first idea models knowledge about scene geometry as a collection of *keypoints*, descriptors for specific interest points in the scene for which the 3D position within scene space is known. These keypoints are constructed where the input images exhibit enough visual structure. What kind of structure is required depends on the specific formulation of the keypoints that the algorithm uses (such as the popular SIFT or SURF descriptors), and possible options include corners, blobs or lines. Once keypoints have been created, they must be found ("matched") in other images, which yields so-called 2D-3D point correspondences[1)]. Using routines such as the 8-point algorithm (see section 7.1), it is then possible to estimate the camera pose for a new input frame. If camera poses for the input frames are already known, new keypoints can be created by triangulating 2D features from multiple frames. As the number of keypoints (area used for tracking) is usually small compared to the total number of image pixels (total image area), algorithms using this idea are said to perform **sparse tracking**.

The second approach tries to match the appearance of entire frames to a reference model on a per-pixel basis. This immediately distinguishes it from the idea of sparse tracking, which is why it is called **dense tracking**. For this, it is necessary that the system already has partial knowledge about the scene's textured geometry. Specifically, the available information must be dense enough to allow for large areas of pixels in new input frames to be compared to the model. This marks another difference from sparse tracking where only single key features in the scene have to be known.

In a feature-based approach, keypoint matches are relatively unambiguous (which is exactly what they are designed to do). Contrary to this, dense tracking works without known correspondences. Instead, it attempts to find a camera pose such that a rendered view of the current model matches the input frame in its entirety. Since single pixels can not usually be matched uniquely (or even correctly, as pixels only capture discretized fragments of an object's appearance and the object can not be expected to move in exact pixel-sized steps), dense tracking heavily relies on the concept of compromise, i.e. it seeks the camera pose that minimizes the pixelwise error in a least-squares sense.

We apply dense tracking via whole-image alignment, i.e. by directly comparing the current live camera image to a reference image for which the camera pose is known. Compared to feature-based methods which only look for matches of certain keypoints, dense tracking has several advantages: First, it is not necessary to

[1)] The feature itself is in 3D scene space, and is linked to the 2D pixel position where it has been matched in the input frames.

create feature points. This might fail if the image does not contain enough areas that conform to the specific type of “featureness” required by the feature model, for example strong corners or blobs. Second, a dense method also does not have to *find* these feature points in the frame it is tracking. This step is often expensive as good features have to be invariant against a number of transformations (noise, additive or multiplicative brightness changes, rotation, affine or even perspective warping or partial occlusion), and extracting more robust descriptors is generally more costly. A dense tracker can use the entire image for pose estimation, thus utilizing all available visual information. Pixels are compared directly by color. This eliminates the need for prior feature extraction or invariance computation, and the lack of a dedicated matching step guarantees that for every pixel a “correspondence” is found (although it should not be thought of as a correspondence, but more like the pixel wanting to move in a certain direction).

5.2 Dense frame-to-keyframe tracking

In tracking, we attempt to estimate the live camera pose $\mathbf{T}_l$, such that the keyframe rendered into $\mathbf{T}_l$ (the virtual image I_v) best matches the current live input frame I_l. We assume that a keyframe is always available, either via the system’s initialization (see chapter 7) or by using already reconstructed depth maps (see section 6.1).

Photometric cost function

Tracking closely follows the scheme laid out in [1]. A virtual camera v (with the same intrinsic parameters as the physical camera) is set up with an initial pose guess $\mathbf{T}_v$ which is required to be fairly close to $\mathbf{T}_l$. In a realtime setting, the previously tracked frame’s pose yields a sufficiently close estimate.

The keyframe’s textured depth map is projected into the virtual camera which produces the virtual image I_v and its depth map ξ_v. Then, a cost function $f_{\mathbf{u}}(\psi)$ is set up for every valid pixel $\mathbf{u} \in \Omega_{I_v}$ in the virtual image. This function depends on the six components of $\psi \in \mathbb{R}^6$, a minimal parametrization (see section 3.5) of a relative pose change $\mathbf{T}_{\text{rel}}(\psi)$ applied to $\mathbf{T}_v$, and computes the photometric difference between the pixel $\mathbf{u}$ in the virtual image I_v and its corresponding pixel in the live image I_l (assuming that $\mathbf{T}_l = \mathbf{T}_{\text{rel}}(\psi)\mathbf{T}_v$ holds).

$$f_{\mathbf{u}}(\psi) := I_l\Big(\pi\left(\mathbf{K}\mathbf{T}_{\text{rel}}(\psi)\pi^{-1}\left(\mathbf{u}, \xi_v(\mathbf{u})\right)\right)\Big) - I_v(\mathbf{u}) \tag{5.1}$$

The sum of squared pixelwise differences over all valid pixels $\mathbf{u}$ in I_v constitutes the total error function $F(\psi)$

$$F(\psi) := \frac{1}{2} \sum_{\mathbf{u} \in \Omega_{I_v}} \Big(f_{\mathbf{u}}(\psi) \Big)^2 \tag{5.2}$$

$$= \frac{1}{2} \|f(\psi)\|_2^2, \text{ with } f(\psi) = \begin{pmatrix} f_{\mathbf{u}_1}(\psi) \\ f_{\mathbf{u}_2}(\psi) \\ \vdots \\ f_{\mathbf{u}_n}(\psi) \end{pmatrix}, \mathbf{u}_i \in \Omega_{I_v} \tag{5.3}$$

Energy minimization

Now the goal is to find the pose change ψ_0 minimizing the error

$$\psi_0 = \arg\min_{\psi} F(\psi) \tag{5.4}$$

$$= \arg\min_{\psi} \frac{1}{2} \sum_{\mathbf{u} \in \Omega_{I_v}} \left(I_l\Big(\pi\left(\mathbf{K}\mathbf{T}_{\text{rel}}(\psi)\pi^{-1}\left(\mathbf{u}, \xi_v(\mathbf{u})\right)\right)\Big) - I_v(\mathbf{u}) \right)^2 \tag{5.5}$$

in which case $\nabla F(\psi_0) = 0$. Unfortunately, the color of a pixel does not generally relate to the pixel's coordinates[2)], and neither does the photometric error, which makes $F(\psi)$ highly nonlinear. To work around this problem, in the style of Lucas-Kanade [30] we iteratively compute least-squares solutions to a linearized version of $F(\psi)$. We approximate $F(\psi)$ by

$$\hat{F}(\psi) = \frac{1}{2} \hat{f}(\psi)^\top \hat{f}(\psi) \tag{5.6}$$

where $\hat{f}(\psi)$ is the first-order Taylor expansion of $f(\psi)$ about $\psi = \mathbf{0}$. This works because we currently assume that $\mathbf{T}_v = \mathbf{T}_l$, so there is no relative pose change yet, and thus $\mathbf{T}_{\text{rel}}$ is the identity transform.

$$\begin{aligned} \hat{f}_{\mathbf{u}}(\psi) &= f_{\mathbf{u}}(\psi) + f'_{\mathbf{u}}(\psi) \\ &= I_l\Big(\pi\left(\mathbf{K}\mathbf{T}_{\text{rel}}(\psi)\pi^{-1}\left(\mathbf{u}, \xi_v(\mathbf{u})\right)\right)\Big) - I_v(\mathbf{u}) \\ &\quad + I_l\Big(\pi\left(\mathbf{K}\mathbf{T}_{\text{rel}}(\psi)\pi^{-1}\left(\mathbf{u}, \xi_v(\mathbf{u})\right)\right)\Big)' - 0 \\ &= I_l\Big(\pi\left(\mathbf{K}\mathbf{T}_{\text{rel}}(\psi)\pi^{-1}\left(\mathbf{u}, \xi_v(\mathbf{u})\right)\right)\Big) - I_v(\mathbf{u}) + \nabla I_l \frac{\partial \mathbf{T}_{\text{rel}}}{\partial \psi} \hat{\psi} \end{aligned} \tag{5.7}$$

[2)] Although it can, in constant or constant gradient images, but these do not make for very interesting SLAM applications.

The Jacobian of the transformation matrix $\frac{\partial \mathbf{T}_{\text{rel}}}{\partial \psi}$ describes how the matrix parameters react to changes in the components of ψ. Via the product rule for gradients, it holds that

$$\begin{aligned} \nabla \hat{F}(\psi) &= \frac{1}{2}\hat{f}(\psi)^\top \nabla \hat{f}(\psi) + \frac{1}{2}\nabla \hat{f}(\psi)^\top \hat{f}(\psi) \\ &= \nabla \hat{f}(\psi)^\top \hat{f}(\psi). \end{aligned} \tag{5.8}$$

We can get $\hat{\psi} \approx \psi_0$ by solving

$$\begin{aligned} 0 &= \nabla \hat{f}(\hat{\psi})^\top \hat{f}(\hat{\psi}) \\ &= \left(\nabla I_l \frac{\partial \mathbf{T}_{\text{rel}}}{\partial \psi}\right)^\top \left(I_l\Big(\pi\left(\mathbf{K}\mathbf{T}_{\text{rel}}(\hat{\psi})\pi^{-1}\left(\mathbf{u}, \xi_v(\mathbf{u})\right)\right)\Big) - I_v(\mathbf{u}) + \nabla I_l \frac{\partial \mathbf{T}_{\text{rel}}}{\partial \psi}\hat{\psi}\right). \end{aligned} \tag{5.9}$$

Equivalently, we can solve

$$\nabla f(\mathbf{0})\hat{\psi} = -\, f(\mathbf{0}) \tag{5.10}$$

or its normal equations

$$\nabla f(\mathbf{0})^\top \nabla f(\mathbf{0})\hat{\psi} = -\, \nabla f(\mathbf{0})^\top f(\mathbf{0}) \tag{5.11}$$

which only leave an equation system of size 6×6.

Finally, we use $\hat{\psi}$ to update the virtual pose estimate $\mathbf{T}_v$:

$$\mathbf{T}_v(\psi) := \mathbf{T}_v(\psi)\mathbf{T}(\hat{\psi}) \tag{5.12}$$

These steps are repeated until convergence, i.e. $\hat{\psi} \approx \mathbf{0}$ and thus hopefully $\mathbf{T}_v \approx \mathbf{T}_l$. Note that this is not guaranteed since we only approximately solve an approximation of the original function. However, the input frames are coming in from a live camera, so we can reasonably assume the baseline and pose change between frames to not be too large.

As in [1], pixels with an error exceeding a user-specified threshold are ignored and not included into the solving of $\hat{\psi}$. While occluded pixels are not explicitly handled by the tracking procedure, thresholding and the small baseline between subsequent frames avoid the inclusion of many wrong pixel correspondences.

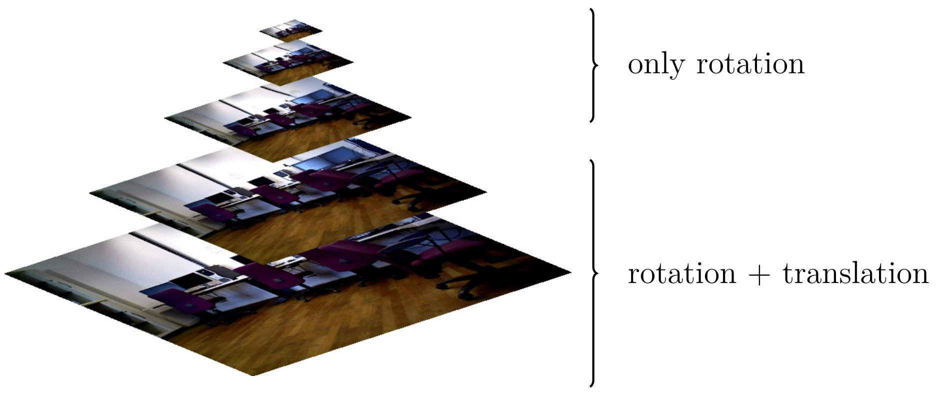


Figure 5.2: Image pyramid: On the upper/coarser/smaller levels, tracking only considers rotation. On the lower/finer/larger levels, all six degrees of freedom are computed (image not representative of actual number of levels / switching level).

For improved speed and range of convergence, tracking works on a coarse-to-fine image pyramid [1, 17]. The ratio of image sizes between pyramid levels is user-adjustable; in our experiments, a setting of 0.7 achieved the best compromise between performance and stability.

Additionally, on the higher (coarse, smaller) pyramid levels the translational elements of the estimated pose change are discarded, constraining the pose update to a pure rotation [1]. When using a hand-held camera, rotation due to (accidental) wrist motion quickly introduces large pixel movement in the input images (much more so than the actual camera translation), which is more easily caught on coarser levels.

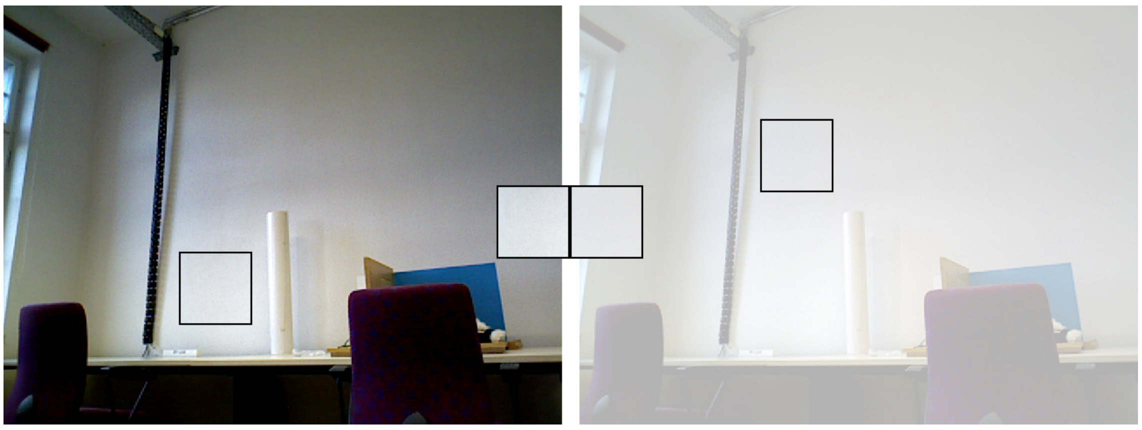


Figure 5.3: Ambiguities in intensity-based dense tracking: The cropped (featureless) section in the left image "moves" (is best matched elsewhere) if the brightness changes, which happens in the case of changes in exposure or white balance. This is much less of a problem with (uniquely) structured regions.

5.3 Semi-dense tracking

While using all available data is arguably better founded than tracking only a small number of individual features, it is unlikely that all pixels of a frame actually contribute meaningful information. For example, a blue sky does not exhibit any features and is mostly useless for tracking. In an indoor scene the walls, ceiling, floor or desks are often monochrome. They can also appear featureless with a single color gradient, for example due to lighting effects, which can give false motion hints and can be a problem under variable lighting conditions or changes in the camera's exposure (see Figure 5.3). Ignoring information-poor image regions is thus in the interest not only of performance (less considered pixels means less time spent on computation), but robustness as well.
To achieve this, Engel *et al.* [17] suggest ignoring image areas where the gradient is weak, indicating that these pixels do not carry information useful to the tracker. The authors call such thinned-out images and depth maps "**semi-dense**", as they ignore uninteresting areas yet provide dense coverage where it is judged useful. We use the same concept in this work to reduce the amount of computation time spent on bland image regions.

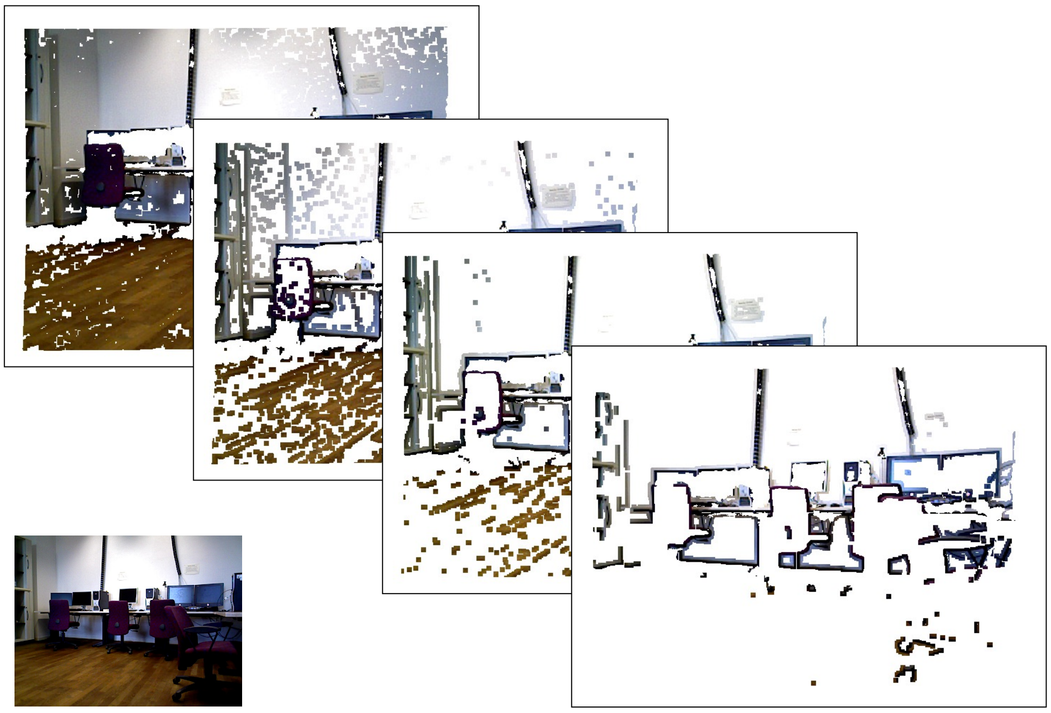


Figure 5.4: Different thresholds on gradient magnitude yield different degrees of "semi-density". Less surviving pixels means less computational effort, but if the image is reduced too far there may not be enough data left to be used in tracking.

The use of depth maps keyframes in tracking leads to certain limitations. Depth maps naturally only contain scene information for one specific viewpoint, so if the camera travels around an object and away from the keyframe pose, it can happen that objects are incorrectly visible in the rendered depth maps although in reality they are occluded. The keyframe depth map only knows about objects' "front" sides and does not model depth extents of the objects. The issue is visualized in Figure 5.1. For this reason, it is a good idea to often spawn new keyframes when the camera is circling around something. This is helpful for another reason: Small errors in the depth maps do not matter much if the depth map is rendered for a viewing angle similar to that of the keyframe to which it belongs, because if the depth of a point changes, the point moves almost directly towards or away from the camera. However, as the angle between the keyframe view and the render view approaches 90°, depth errors lead to large positional errors in the rendering, which is harmful to tracking (see also Figure 6.6).

5.4 Loop closure

The system performs loop closure in two different ways:

Explicitly, in regular intervals (every X frames) all keyframes (including the one currently used for tracking) are rendered for the latest tracked frame's camera view. The number of valid pixels these renderings produce yields a coverage ratio for each keyframe, indicating how well that keyframe's view coincides with what the camera is currently observing. Tracking then switches to using the keyframe with the best (highest) coverage ratio. Effectively, this constantly seeks loop closure to all known keyframes, allowing the camera to travel around the already mapped part of the scene without getting lost. Reusing older keyframes for tracking prevents the buildup of pose drift. In our experiments this loop closure works well; we assume that tracking does not induce large camera drift. This is consistent with our mapping strategy which requires the individual keyframes to overlap with previously mapped parts of the scene, so the camera can never travel very far without tracking switching to another keyframe.

The system also realigns each newly constructed depth map against the existing model (the union of all previous keyframes' point clouds). This constitutes a more implicit loop closure step and is described in section 6.2.

6 Dense scene mapping

The tracking procedure described in chapter 5 requires dense depth maps. In this chapter we introduce the framework we use to interactively obtain new depth maps for individual viewpoints of the scene, namely the keyframes that serve as reference points for camera tracking. For a new keyframe we build a cost volume from the input frames. The volume relates discrete depth hypotheses for each pixel in the keyframe to the photometric error a pixel would generate in all included frames given a candidate depth. Each new frame increments the cost volume which allows for live previews of the current depth map estimate.
We also explain our approach of using point cloud-based ICP to align newly constructed depth maps to the existing model, in order to keep our model consistent and to reduce camera drift.

6.1 Depth map reconstruction

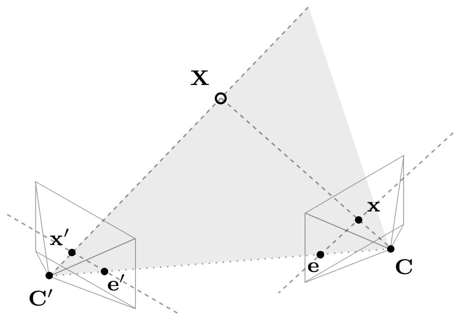
The general idea behind depth estimation is matching point correspondences between images. With known correspondences, known camera poses and known camera intrinsics, 3D points can be triangulated directly[1]. We assume a pre-calibrated camera, and our tracking procedure yields the necessary pose information, but using a featureless tracking approach means that there is no data available on 2D-2D or 2D-3D correspondences.

From only the appearance of an object with unknown structure in one image, it is not possible to determine the depth (distance) of a point on that object's surface (at least not in our setting; there are other approaches such as *shape from defocus* which can give single-image visual depth cues [43]). Consequently, without further information it is also not possible to determine where this point appears in a second image taken from a different viewpoint[2]. Since every single pixel in the entire second image is a potential match for the point—and considering a featureless setting as is used in this work—it is hard or even impossible to identify the correct match by visual comparison alone (if that match even exists). It is very likely that many locations match equally well, or that an incorrect location matches best because the

[1] Although if correspondences are available, poses and intrinsics do not have to be known (see also section 7.1).

[2] Leaving out feature matching or object recognition, but these are only applicable to relatively rare feature points.

correct correspondence is occluded, distorted by noise, or looks different because it does not strictly adhere to the brightness constancy assumption.
However, if the camera pose for the second frame is known, the pixel position of the point in the second image is constrained to a known 1D line called its *epipolar line*, and only depends on the depth associated with the pixel (see Figure 6.1). In the case of two cameras (binocular vision) this relationship—the epipolar geometry—can be expressed by the *fundamental matrix* [11], also called bifocal tensor. Extensions exist for three and even four included views (tri-/quadrifocal tensor), but the concise representation stops there.



(a)

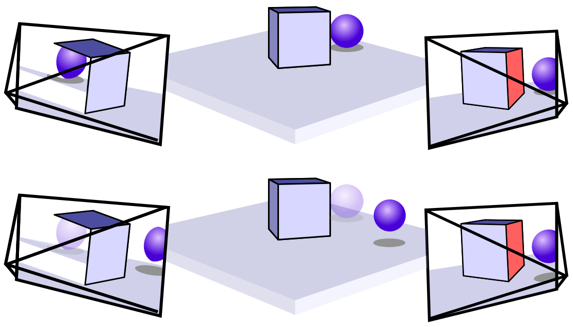

(b)

Figure 6.1: (a) Visualization of the epipolar geometry connecting two views: The camera centers $\mathbf{C}$ and $\mathbf{C}'$, their projections into the respective other camera (the *epipoles*) $\mathbf{e}$ and $\mathbf{e}'$, the 3D point $\mathbf{X}$ and its projections into both views $\mathbf{x}$ and $\mathbf{x}'$ are all coplanar. (b) Moving an object (the blue sphere) towards/away from the right camera in line of sight does not change the appearance of its center in that view[3)]. In the left view, the sphere moves on its epipolar line.

We want to utilize as much visual information as possible for scene mapping. Beside color noise in the input images, the realtime setting includes further difficulties such as noisy camera poses, complex camera motion, motion blur, changes in lighting, camera exposure and white balance, objects passing through (against the assumption that the scene is static), and possibly rolling shutter artifacts, all of which pose a danger to stereo reconstruction. We aim to counteract these effects by using a large number or small-baseline frames for the estimation of each new keyframe's depth map. This averages out color noise, and mitigates the impact of single frames with imprecise pose estimates .

Cost volume

We use the input frames and their camera poses to fill a projective 3D *cost volume* $\mathbf{V}$. Following [1], this is an extension of the *disparity space image* from [42, 44], adapted

[3)]This visualization ignores the fact that the blue sphere would appear *larger* if moved towards the camera (hence the caption only mentioning its center's appearance).

to a multi-view application. The cost volume contains a row $\mathbf{V}(\mathbf{u})$ of entries for every pixel $\mathbf{u} \in \Omega_{I_r}$ in the reference frame I_r (the soon-to-be new keyframe). Every volume cell $\mathbf{V}(\mathbf{u}, d)$ in $\mathbf{u}$'s row corresponds to a certain possible depth d for that pixel. The value in that cell is the average of a series of photometric errors, one for each frame $I_i \in \mathcal{N}_r$ within the set of frames $\mathcal{N}_r$ that are used for depth reconstruction (except for the keyframe itself[4]).
Reconstructing a 3D point for $\mathbf{u}$ with depth d and then reprojecting that 3D point into I_i (compare Equation 5.1) yields a 2D-2D correspondence between I_r and I_i from which the desired error can then be computed.

$$\mathbf{V}(\mathbf{u}, d) = \frac{1}{|\mathcal{N}_r|} \sum_{i \in \mathcal{N}_r} \rho(I_r, I_i, \mathbf{u}, d) \tag{6.1}$$

where ρ is the used cost function. The depth range and discrete depth steps of the cost volume have to be define beforehand. If the reprojection of $\mathbf{u}$ with d falls outside the image boundaries of I_i, the cell $\mathbf{V}(\mathbf{u}, d)$ is not updated.
In order to make depth map reconstruction interactive and to be able to show the user of our system a live preview of a depth map in construction, we need to reformulate the cost volume: Figure 6.2 describes a simple adjustment of to allow for incremental updates using newly arriving frames.

The important decisions to make are the choice of the photometric error function, and the algorithm used to recover the best depth for every pixel.

Cost functions

We have implemented and evaluated two different cost functions for this work: First, a simple pixelwise **sum of absolute differences** (SAD):

$$\rho_{\text{SAD}}(I_r, I_i, \mathbf{u}, d) = \left| I_i\Big(\pi\left(\mathbf{K}\mathbf{T}_i \pi^{-1}(\mathbf{u}, d)\right)\Big) - I_r(\mathbf{u}) \right|, \mathbf{u} \in \Omega_{I_r} \tag{6.2}$$

This function simply constructs a 3D point from the pixel $\mathbf{u}$ and projects that 3D point into I_i. The error is the absolute intensity difference between $\mathbf{u}$ and the (linearly interpolated) value at the resulting position in I_i.
The SAD error function is computationally cheap, but heavily depends on the brightness constancy assumption and is thus more susceptible to e.g. brightness changes and image noise.

[4] Actually it does not matter whether the keyframe is included in the photometric error sum or not. A pixel reprojected into its own image will always have *zero* error, so it is entirely up to the designer's preferences whether to explicitly exclude the keyframe or not.

Sidebar: Incremental cost volume update

For live depth map previews, we need to be able to incrementally update the cost volume. We store two values for each volume cell $\mathbf{V}(\mathbf{u}, d)$, the total sum of photometric errors and the number of contributing frames.

$$\begin{aligned}\mathbf{V}(\mathbf{u}, d, 0) &= \sum_{i \in \mathcal{N}_r} \rho(I_r, I_i, \mathbf{u}, d) \\ \mathbf{V}(\mathbf{u}, d, 1) &= |\mathcal{N}_r|\end{aligned}$$

This allows including the error terms for a new frame I_j by simply updating each affected cell (where the reprojection of $\mathbf{u}$ with d falls within the boundaries of I_j):

$$\begin{aligned}\mathbf{V}'(\mathbf{u}, d, 0) &:= \mathbf{V}(\mathbf{u}, d, 0) + \rho(I_r, I_j, \mathbf{u}, d) \\ \mathbf{V}'(\mathbf{u}, d, 1) &:= \mathbf{V}(\mathbf{u}, d, 1) + 1\end{aligned}$$

The averaged error can at any time be trivially retrieved as $\frac{\mathbf{V}(\mathbf{u},d,0)}{\mathbf{V}(\mathbf{u},d,1)}$.

Figure 6.2: Incremental cost volume update

The second tested function is fixed-window **normalized cross-correlation** (NCC). Let

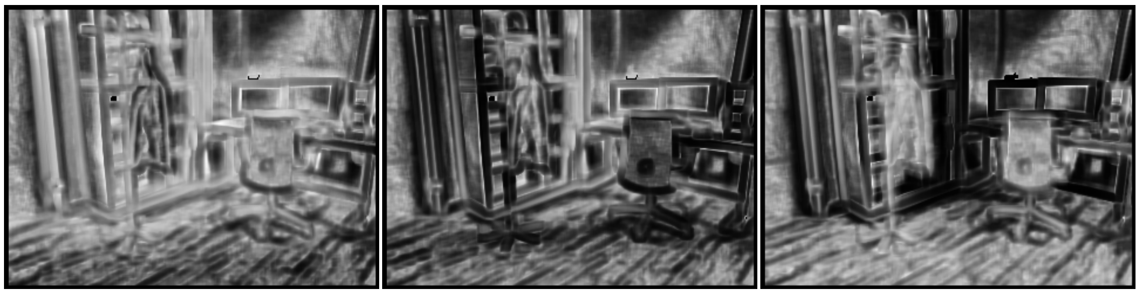
$$\mu(\mathcal{N}_\mathbf{u}, I) = \frac{1}{|\mathcal{N}_\mathbf{u}|} \sum_{\mathbf{v} \in \mathcal{N}_\mathbf{u}} I(\mathbf{v}) \tag{6.3}$$

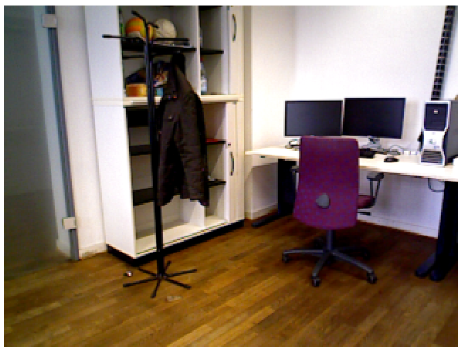
be the average intensity over the pixel neighborhood $\mathcal{N}_\mathbf{u}$ in image I. Then

$$\rho_{\text{NCC}}(I_r, I_i, \mathbf{u}, d) = \frac{\sum\limits_{\mathbf{v} \in \mathcal{N}_\mathbf{u}, \mathbf{v}' \in \mathcal{N}_{\mathbf{u}'}} \Big(\mu(\mathcal{N}_\mathbf{u}, I_r) - I_r(\mathbf{v})\Big) \cdot \Big(\mu(\mathcal{N}_{\mathbf{u}'}, I_i) - I_i(\mathbf{v}')\Big)}{\sqrt{\sum\limits_{\mathbf{v} \in \mathcal{N}_\mathbf{u}} \Big(\mu(\mathcal{N}_\mathbf{u}, I_r) - I_r(\mathbf{v})\Big)^2} \cdot \sqrt{\sum\limits_{\mathbf{v}' \in \mathcal{N}_{\mathbf{u}'}} \Big(\mu(\mathcal{N}_{\mathbf{u}'}, I_i) - I_i(\mathbf{v}')\Big)^2}} \tag{6.4}$$

$$\text{where } \mathbf{u}' = \pi\left(\mathbf{K}\mathbf{T}_i \pi^{-1}(\mathbf{u}, d)\right) \tag{6.5}$$

NCC costs are much more expensive to compute, but also more robust. In our experiments, the NCC window has a size of 7×7 pixels.


(a)


(b)

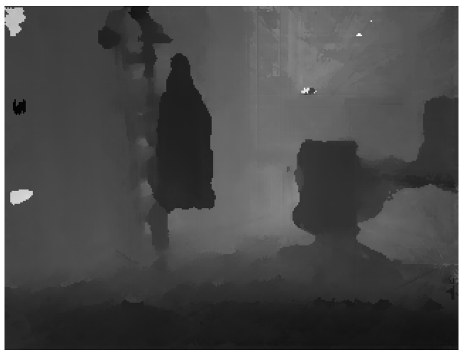

(c)

Figure 6.3: (a) Three depth slices (*from left to right:* Near, middle, far) from a cost volume built with normalized cross-correlation costs. The brighter the image, the larger the error that a pixel would generate if it had the slice's depth. In the left image, the floor appears dark, indicating that this is the correct depth for this part of the scene. Further away from the camera, in the center image a different part of the floor is dark, as is the chair. Distant structures have their lowest cost in the right image, where the floor is bright (large cost). (b) Corresponding color input image for the cost volume. (c) The depth map that SGM extracts from the cost volume. Note that the depths found for the white wall pixels are not reliable as there is virtually no visible structure.

Depth computation

It is trivial to find the "best" depth map regarding the cost function by simply traversing all depth levels and choosing, for each pixel separately, the depth with the lowest cost. However, the naive approach is very susceptible to noise, and by solving each pixel individually it completely ignores smoothness. Ideally, each pixel should take into consideration all other pixels, and the chosen depths should be those that not only have low cost, but also yield a smooth depth map. Unfortunately, it is not feasible to tackle the problem in this form—especially in a setting where speed is a critical factor—as the global optimization including smoothness is NP-complete for many energy formulations that preserve discontinuities [21].

One method to include smoothness into the solution is *message passing*, which tries to find a smooth solution within a tree graph[5)] [11]. If the used images are rectified (epipolar lines are horizontal), message passing can treat each horizontal scanline independently and in parallel as a dynamic programming instance, which makes for good performance.
A problem with scanline message passing is that even though a globally optimal solution is found, that solution is often not globally smooth or particularly good because the energy only constrains smoothness along a line (see Figure 6.4).

Semi-global matching

Our program solves the cost volume using *semi-global matching* (SGM) [21] which approximates global 2D smoothness of the depth map by aggregating matching costs from multiple 1D lines ("paths") at the same time. It thus achieves a compromise between speed and global optimality (see Figure 6.4). The energy $E(L, \mathbf{u}, d)$ for a pixel $\mathbf{u}$ at depth d along a path L is here recursively defined as

$$\begin{aligned} E(L, \mathbf{u}, d) = E_{\text{data}}(\mathbf{u}, d) + \min \Big(& E(L, \mathbf{u} + \mathbf{r}, d), \\ & E(L, \mathbf{u} + \mathbf{r}, d - 1) + P_1, \\ & E(L, \mathbf{u} + \mathbf{r}, d + 1) + P_1, \\ & \min_i E(L, \mathbf{u} + \mathbf{r}, i) + P_2 \Big) \end{aligned} \tag{6.6}$$

where $E_{\text{data}}(\mathbf{u}, d)$ is a unary data term which measures the error generated in the input images by the depth map (using one of the cost functions introduced earlier in this chapter). The recursion continues with the next path pixel, by walking a step in the path's direction $\mathbf{r}$ (away from $\mathbf{u}$). The total energy for $\mathbf{u}$ at depth d is simply the sum of $E(L, \mathbf{u}, d)$ over all paths L ending in $\mathbf{u}$. The second (binary)

[5)] This cannot be applied to the entire image at once; the resulting graph connects each pixel (at least) to its direct neighbors, and thus is not a tree.

component of the energy works as follows: If the neighboring pixel $\mathbf{u} + \mathbf{r}$ has the same depth label as $\mathbf{u}$, there is no additional cost (this favors smoothness). If the next depth label is only a small change (e.g., one discrete step) from $\mathbf{u}$'s label, there is a small constant penalty P_1. This allows the depth map to include slanted or curved surfaces at reasonable cost. All larger depth changes are punished with a larger constant penalty P_2. While this discourages sudden changes, the magnitude of the jump is irrelevant, so discontinuities along object borders can be sharp and are not forced to drop off over a longer distance to save cost [21].

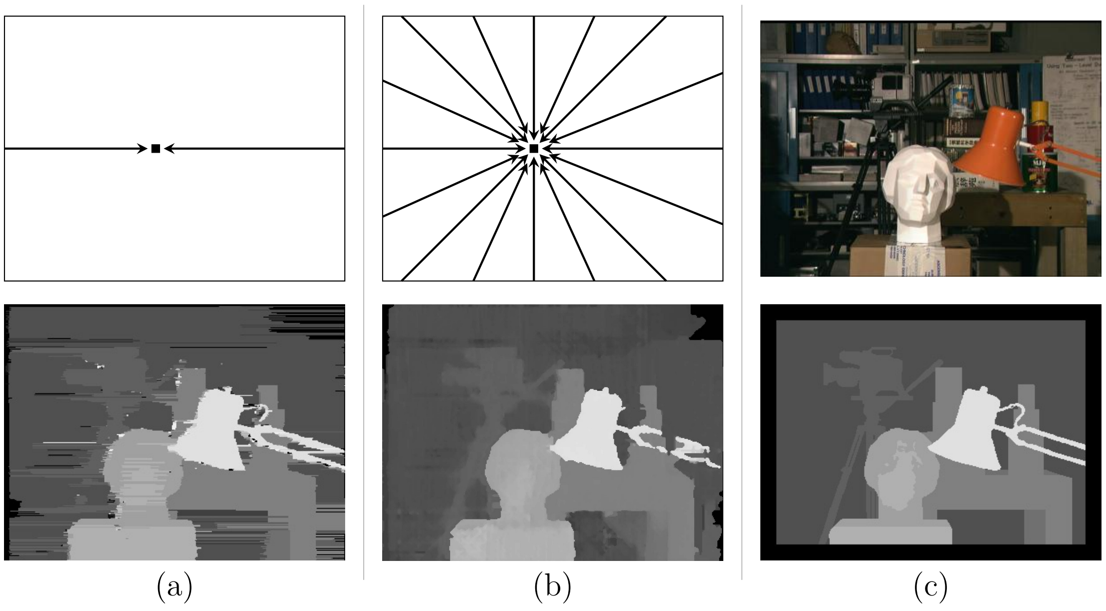


Figure 6.4: Smoothness in stereo estimation: (a) Scanline message passing seeks smoothness along a horizontal line, but ignores vertical constraints. The solution is likely to contain line artifacts. (b) Semi-global matching (SGM) aggregates cost terms from many paths (here: 16) in different directions and considers all to get a more globally smooth solution. (c) A frame from the dataset and its ground truth (Images from [45] (bottom left), [21] (bottom center) and [46] (right column)).

While the resulting depth map attempts to be smooth as far as the depth labels are concerned, it still consists of discrete steps. For a smoother result, we append an interpolation step as described in [21]: Given a pixel with its label as assigned by SGM, we fit a quadratic curve through the costs of the next smaller and next greater depth labels for that pixel and solve for the minimum of the curve.

Depth confidence filtering

Discarding depth map regions belonging to poorly textured image regions not only saves computational effort in tracking. It also acts against the most obvious source of incorrect depths: Pixels within unstructured blobs are likely to have unclear depth

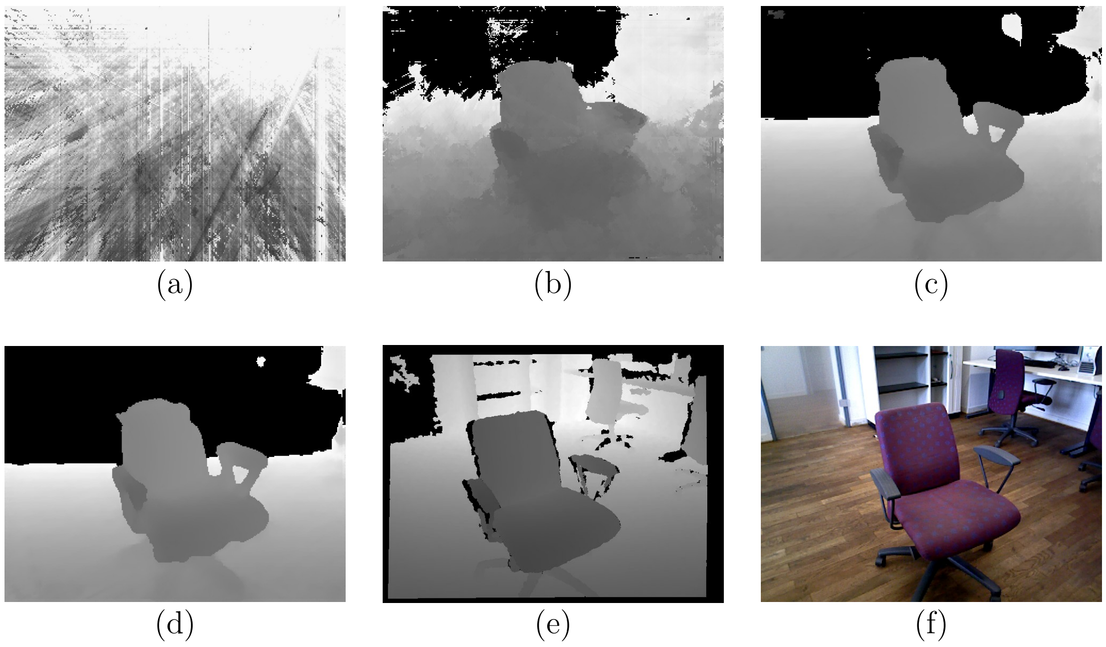


Figure 6.5: Constructing a depth map: (a) Depth map solved from very early NCC cost volume. Only SGM artifacts are visible. (b) Depth map after ca. 20 included frames. The chair's rough outlines are slowly emerging. (c) After ca. 30 included frames, even the chair's legs are recognizable. (d) Final refined depth map using ca. 50 frames. (e) Reference depth map (not exactly the same viewpoint) obtained from an RGB-D camera. (e) The RGB image which, together with the solved depth map, forms a new keyframe. (a)-(e) Black pixels are invalid depths, in (e) mostly due to automatic occlusion detection (and there is a reflecting glass door in the upper left corner), in (a)-(d) because the estimated depths hit the limits of the cost volume.

optima, and they run a greater risk of being assigned wrong labels in the stereo solving process. While varying their depth will change their 3D position, they are free to move as long as their reprojections still fall into pixels of the same color (see Figure 6.6).

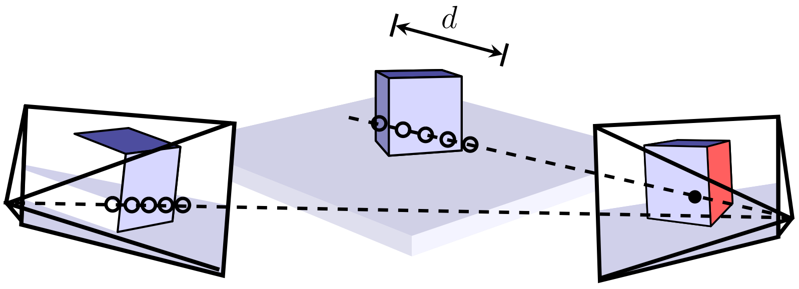


Figure 6.6: Low depth confidence in unstructured regions: Varying the depth value d for a pixel in the right camera view changes the position of the reconstructed 3D point. However, as long as the projection of that 3D point into the left camera still hits a pixel of the same color, there is no matching cost penalty. Consequently, a large range of depths for the original pixel will have minimal cost, and the pixel will get a low depth confidence score.

Additionally to keeping the depth maps semi-dense via the visual criterion of gradient magnitude, we also follow [19] in their approach of invalidating depth map pixels which do not possess a sufficiently unique best depth.
The *confidence score* at depth level d for pixel $\mathbf{u}$ is defined as

$$c(\mathbf{u}, d) = \frac{-(E_{\text{data}}(\mathbf{u}, d) - E_{\text{data}}(\mathbf{u}, d^*))^2}{\sigma^2} \tag{6.7}$$

for some predefined variance σ^2, where d^* is the depth chosen by SGM. With this, the *depth confidence* $C(\mathbf{u})$ of $\mathbf{u}$ is the sum over all possible depth levels (excluding d^*)[6]:

$$C(\mathbf{u}) = \sum_{d \neq d^*} e^{-c(\mathbf{u}, d)} \tag{6.8}$$

This confidence is high if there is a single distinct minimum in the depth costs, and low if the minimum region is wide or if there are multiple similarly good minima.

[6] It does not matter much whether d^* is included in the sum or not. The confidence score for d^* is $c(\mathbf{u}, d^*) = -(E_{\text{data}}(\mathbf{u}, d^*) - E_{\text{data}}(\mathbf{u}, d^*))^2/\sigma^2 = 0^2/\sigma^2 = 0$, so including d^* just adds a constant offset of 1.0 to all pixel's confidences. While the original fixed-threshold approach of [19] might need to be adjusted, our histogram-based thresholding automatically adapts.

Going beyond [19], we compute a histogram of confidences on the entire depth map, and invalidate a fixed ratio of all pixels with the lowest scores. Depth confidences change depending on the structure and texture attributes of the observed scene and lighting effects. Contrary to the original idea of using a fixed confidence threshold, our method can adjust to unfavorable conditions where it is still better to use low-confidence depths than to simply discard all because they scored below the threshold.

Remark

Note that while semi-dense depth maps are used for tracking, the reconstruction process actually produces fully dense depth information for every keyframe. The resulting reconstruction thus is dense as well and not limited to gradient-exhibiting structures. However, the reconstructed depths for such plain areas are generally less reliable, for the same reason that they are ignored in tracking: It is not possible to precisely estimate the position of a single point if the point cannot be distinguished from its neighborhood. Strategies to get better depth maps include applying a regularization term during reconstruction or post-processing the results (for example using a bilateral filter). It is also possible to assume a prior on the distribution of depths. For example, Gallup *et al.* [20] and Pollefeys *et al.* [19] propose using a plane prior. However, they explicitly apply their algorithms to urban scenes where buildings and streets offer mainly planar surfaces, an assumption that is not generally met by natural structures.

6.2 Depth map alignment

A well known problem of frame-to-frame tracking is error accumulation: The estimated poses are unlikely to be absolutely correct, and if the tracking process determines each new frame's pose relative to the preceding frame, these inaccuracies quickly sum up.
We use frame-to-*keyframe* tracking, i.e. a new frame's pose is computed relative to the current keyframe. Since the keyframe changes much less frequently than the input framerate, we do not amass as much error. Nevertheless, the buildup of drift is an issue since each new keyframe's pose is still only relative to the last keyframe.

To avoid drift, we keep a global model point cloud. This model consists of the union of all keyframe's point clouds constructed from their semi-dense depth maps. When a new keyframe has been finalized, it is aligned to the global model using an *iterative closest point* (ICP) procedure. Once aligned, the keyframe's semi-dense point cloud is fused into the model. To keep the point cloud lean, new points are only accepted into the model if there is not already a point very close to the target position.

Prerequisite: Umeyama method

The Umeyama method [15] is a way to solve the Procrustes problem of least-squares distances between **known pairs** of (in our case) 3D points [13]. Given two point sets of identical size $\mathbf{P} = \{\mathbf{p}_0, \mathbf{p}_1, \dots, \mathbf{p}_n\}$ and $\mathbf{Q} = \{\mathbf{q}_0, \mathbf{q}_1, \dots, \mathbf{q}_n\}$ and w.l.o.g. the known correspondence pairs $(\mathbf{p}_i, \mathbf{q}_i), 0 \leq i \leq n$, we seek a rigid-body transform consisting of a rotation $\mathbf{R}' \in SO(3)$ and translation $\mathbf{t}' \in \mathbb{R}^3$ such that the sum of squared Euclidean distances is minimized[7]:

$$\mathbf{R}' = \arg\min_{\mathbf{R}'} \frac{1}{n} \sum_{i=0}^{n} ||\mathbf{R}\mathbf{p}_i + \mathbf{t}' - \mathbf{q}_i||_2^2 \tag{6.9}$$

The translational component $\mathbf{t}'$ is trivially the translation between the two point sets' means[8].

$$\begin{aligned} \mu_{\mathbf{P}} &= \frac{1}{n} \sum_{i=0}^{n} \mathbf{p}_i, \mu_{\mathbf{Q}} = \frac{1}{n} \sum_{i=0}^{n} \mathbf{q}_i \\ \mathbf{t}' &= \mu_{\mathbf{Q}} - \mu_{\mathbf{P}} \end{aligned} \tag{6.10}$$

We then demean $\mathbf{P}$ and $\mathbf{Q}$ and compute the covariance matrix $\Sigma_{\mathbf{P}',\mathbf{Q}'}$ between the results:

$$\mathbf{P}' = \{\mathbf{p}'_0, \mathbf{p}'_1, \dots, \mathbf{p}'_n\} = \{\mathbf{p}_0 - \mu_{\mathbf{P}}, \mathbf{p}_1 - \mu_{\mathbf{P}}, \dots, \mathbf{p}_n - \mu_{\mathbf{P}}\} \tag{6.11}$$

$$\mathbf{Q}' = \{\mathbf{q}'_0, \mathbf{q}'_1, \dots, \mathbf{q}'_n\} = \{\mathbf{q}_0 - \mu_{\mathbf{Q}}, \mathbf{q}_1 - \mu_{\mathbf{Q}}, \dots, \mathbf{q}_n - \mu_{\mathbf{Q}}\} \tag{6.12}$$

$$\Sigma_{\mathbf{P}',\mathbf{Q}'} = \sum_{i=0}^{n} \mathbf{q}'_i \mathbf{p}'^{\top}_i \tag{6.13}$$

$\Sigma_{\mathbf{P}',\mathbf{Q}'}$ now contains the correct rotation, but is skewed because the elements in the two point sets do not conform to strict unit-length and orthogonality requirements [13]. To extract the rotational part we perform a *singular value decomposition* (SVD) of $\Sigma_{\mathbf{P}',\mathbf{Q}'}$

$$\Sigma_{\mathbf{P}',\mathbf{Q}'} = \mathbf{U} \cdot \mathbf{D} \cdot \mathbf{V}^{\top} \tag{6.14}$$

$$\text{and we finally obtain } \mathbf{R}' = \mathbf{U} \cdot \mathbf{V}^{\top}. \tag{6.15}$$

We cannot use this method directly since there are no known point correspondences between our point clouds. However, we can apply it iteratively within the ICP framework, where in every step there is a set of “known” pairs.

[7]The original Umeyama method also incorporates (and optimizes) a scaling factor, but we assume that our partial reconstructions do not vary in scaling.

[8]A proof of this is given in [15].

Trimmed iterative closest point

```
Sidebar: Iterative closest point algorithm

Input:           Fixed point set $\mathbf{P} = \{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n\}$,
                 Moving point set $\mathbf{Q} = \{\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_m\}$
Output:          Least-squares rigid body transform $\mathbf{R} \in SO(3)$, $\mathbf{t} \in \mathbb{R}^3$
Initialization:  $\mathbf{R}$ := Identity matrix; $\mathbf{t} := (0, 0, 0)^\top$
Repeat:
    1. Compute a set of closest point pairs ($\mathbf{P}'$, $\mathbf{Q}'$) from $\mathbf{P}$ and $\mathbf{Q}$
    2. Apply Umeyama method and retrieve rigid body transform $(\mathbf{R}', \mathbf{t}')$
    3. Transform $\mathbf{Q}$ using $(\mathbf{R}', \mathbf{t}')$; update $(\mathbf{R}, \mathbf{t})$
    4. STOP if stopping conditions are fulfilled.
```

Figure 6.7: Iterative closest point algorithm [13]

The standard ICP procedure introduced in [26] aligns two point clouds **P** and **Q** by iteratively minimizing point pair distances (see Figure 6.7). It is, however, best suited for point clouds that cover the same space or objects: Since every point pair contributes to the computed motion with equal weight, points that cannot find good matches (because the corresponding structure is not present in the other point set) can lead to transformations that might be optimal in the least-square sense, but wrong in what the procedure is actually supposed to, i.e. align the point sets such that those parts coincide that represent the same *object*. Additionally, noise and outliers in the point clouds distort the estimated motion, again due to the algorithm treating all point pairs equally.

Our tracking process always requires the current reference keyframe to be partially visible by the camera, and consequently some overlap between a new keyframe's point cloud and the existing model is guaranteed. However, when exploring a scene the camera can progress faster if it moves further between keyframes, which generally means that *less* overlap is desirable. Considering poorly structured scenes or lighting changes, we also have to be able to cope with noisy depths and depth outliers.

The variant of the ICP algorithm that we use, called *Trimmed ICP* [28, 29], is specifically designed to handle point clouds with noise and limited overlap. Conceptually, it attempts alignment using only regions present in both point clouds. Trimmed ICP attributes a weight to each point correspondence, thus dampening the effects of outliers, and then (optionally) ignores some of the point pairs with smallest weights. If there are structures represented in one point set, but missing from the other, these points can be expected to generate pairs with significantly larger distances than the point pairs [9], which in turn means that they will be the first to be cut off in Trimmed ICP. Figure 6.2 shows the Trimmed ICP algorithm and the differences compared to normal ICP.

Note that while ICP originally assumes that **P** and **Q** are of equal size and thus all points can find distinct partners, we allow the assignment of one model point to multiple points in the depth map's point set. Even where both sets represent the same structure, depth noise and the difference in viewpoint still make it unlikely that a single point in the scene is accurately represented by any point in either of **P** and **Q**, and its closest match in one set may well fall right between several points in the other set.

Stopping conditions

We use the stopping conditions as described in [28], i.e. the alignment loop terminates if one of the following holds:

- The mean square error
$$\frac{1}{|(\mathbf{P}', \mathbf{Q}')|} \sum_{(\mathbf{p}_i, \mathbf{q}_i) \in (\mathbf{P}', \mathbf{Q}')} ||\mathbf{p}_i - \mathbf{q}_i||_2^2 \tag{6.16}$$
is sufficiently small,
- the relative change in mean square error since the last iteration is sufficiently small,
- or the maximum number of iterations has been reached.

[9] All this assumes that the point clouds are roughly pre-registered prior to (Trimmed)ICP. Since a new keyframe's pose is always linked to a previous keyframe (which has already been aligned to the global model), we can expect the new keyframe to be reasonably well aligned as well.

Sidebar: Trimmed iterative closest point algorithm

Input:	Fixed point set $\mathbf{P} = \{\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n\}$,
	Moving point set $\mathbf{Q} = \{\mathbf{q}_0, \mathbf{q}_1, \ldots, \mathbf{q}_m\}$
Output:	Least-squares rigid body transform $\mathbf{R} \in SO(3)$, $\mathbf{t} \in \mathbb{R}^3$
Initialization:	$\mathbf{R}$:= Identity matrix; $\mathbf{t} := (0, 0, 0)^\top$

Repeat:

1. Compute a set of closest point pairs ($\mathbf{P}'$, $\mathbf{Q}'$) from $\mathbf{P}$ and $\mathbf{Q}$
2. Weight point pairs, giving less importance to pairs with greater distance
3. Reject some point pairs, e.g. those with distances greater (or weights smaller) than a given threshold, or a fixed percentage of pairs with greatest distances (or smallest weights)
4. Apply Umeyama method to weighted and trimmed ($\mathbf{P}'$, $\mathbf{Q}'$) and retrieve rigid body transform $(\mathbf{R}', \mathbf{t}')$
5. Transform $\mathbf{Q}$ using $(\mathbf{R}', \mathbf{t}')$; update $(\mathbf{R}, \mathbf{t})$
6. STOP if **stopping conditions** are fulfilled.

Figure 6.8: Trimmed iterative closest point algorithm [28]

7 System initialization

This chapter describes an initialization concept of our implementation. Due to time constraints, we opted for simply using a single depth map, provided by a RGB-D camera, to jump start the system. However, since it is our ultimate goal to use a standard RGB camera (and using a range sensor somewhat contradicts the point of doing monocular SLAM), we here describe a procedure to get an initialization using only visual information.

The algorithm follows a number of steps and suggestions laid out in [11].

7.1 RGB-only initialization

The first step is to track a number of points in the scene across several subsequent frames. This can be done by explicitly tracking feature points, or via optical flow [10]. Once a sufficient baseline has been acquired, we estimate the *fundamental matrix* **F** between the first and last initialization frames. The fundamental matrix models the relation between 2D points in one frame and epipolar lines in the other, and can be estimated from 2D point correspondences alone using the 8-point algorithm[1]. There is a scale ambiguity in **F**. To still get a consistent estimate, we have to assume that the distance between the first and last initialization frames' camera poses is known.

From **F** we can then retrieve the pose of the last initialization frame (the first frame's pose is trivially known as it defines the scene space origin). **F** relates to the pose via the *essential matrix*

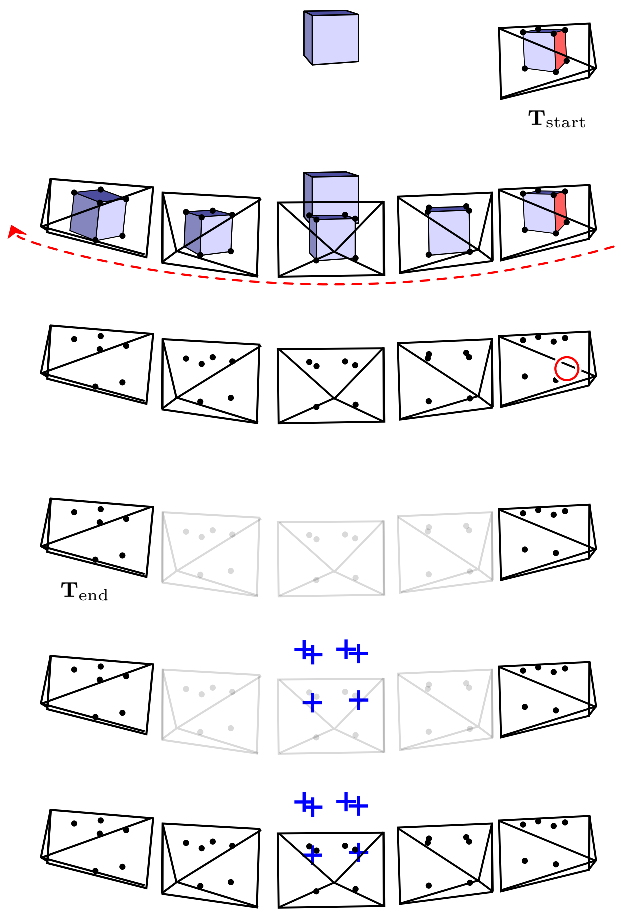
$$\mathbf{E} = \mathbf{K}^\top \cdot \mathbf{F} \cdot \mathbf{K}. \tag{7.1}$$

With the poses of two cameras (and their intrinsics) known, we can now triangulate the 3D points for the tracked points. This in turn allows the estimation of the camera poses for the rest of the initialization frames using the *Orthogonal Iteration* algorithm [16].

At this point, we can use the initialization frames to build up a cost volume and estimate a depth map as described in section 6.1. If there are too few frames for a

[1]In fact, it is better to combine the 8-point algorithm with a RANSAC approach to find a good subset of point correspondences. This helps avoid the inclusion of incorrectly matched correspondences (outliers).

good solution, a sparse tracker can be used to bridge the gap until a good enough depth map is available for dense tracking. The initialization process is visualized in Figure 7.1.



1. Recording begins, points are set up. The first frame's camera pose $\mathbf{T}_{\text{start}}$ defines the origin.

2. Camera moves, point trajectories are tracked.

3. Picking persistent trajectories: One point was lost due to occlusion and is discarded.

4. 2D-2D correspondences between the first and last frame yield the fundamental matrix $\mathbf{F}$ and the last frame's pose $\mathbf{T}_{\text{end}}$.

5. 3D points can be triangulated from the known 2D-2D correspondences and poses.

6. Poses of intermediate frames are computed using their (now) known 2D-3D correspondences.

Figure 7.1: RGB-only initialization procedure (note that while here the cube's edges serve as tracked features, any point tracker will work as long as it can reliably track some points through the entire sequence of frames used for the initialization).

8 Implementation and experiments

This chapter names some details of our implementation, such as the interface presented to the user and how interaction with the program works. We also describe a number of steps we have taken in order to improve performance.
The last part of the chapter contains the evaluation of our experiments.

8.1 Interface and user interaction

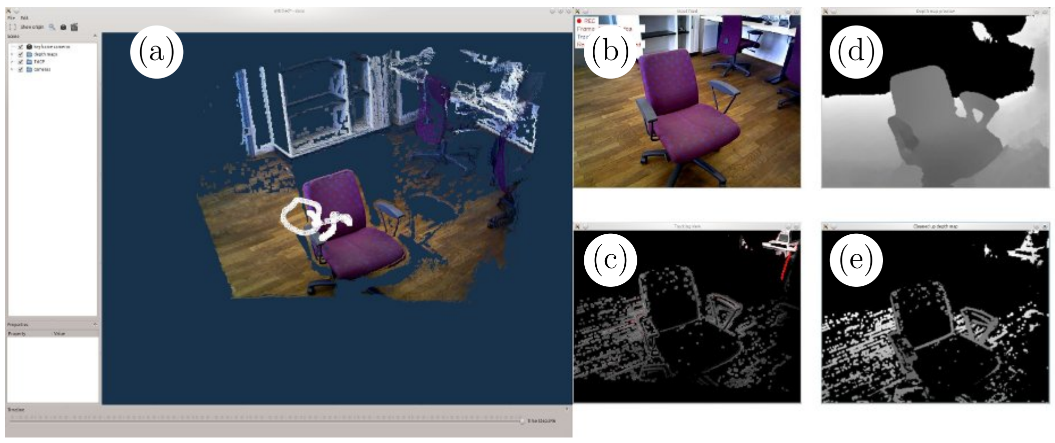


Figure 8.1: User interface: (a) 3D viewer for point clouds and camera poses; (b) live camera feed; (c) live virtual view of the keyframe currently used for tracking; (d) live preview of the estimated depth map during reconstruction; (e) live semi-dense preview of (d).

Our program's user interface is shown in Figure 8.1, and Figure 8.2 shows the program states and interaction possibilities. On startup, the application retrieves a depth map from the RGB-D camera (or a prerecorded dataset) to get an initialization (see chapter 7). It then immediately starts tracking the camera, but waits for the user's decision when to begin constructing a depth map. The user positions the camera such that the next-to-be-mapped scene part is in view, and activates the depth map reconstruction process (the camera pose at that moment will be used for the new keyframe). During reconstruction, the program incrementally fills the cost volume for SGM (see section 6.1) and provides a live preview of the current reconstruction progress. When content with the new depth map, the user triggers the finalization of the new keyframe which is then included into the global model and immediately made available for tracking. Afterwards, the system goes back to tracking mode and waits for the next reconstruction command.

Should tracking failure occur, one of two cases applies:

- Complete tracking loss: The currently used keyframe is out of view because the estimated camera pose has deviated too far from the keyframe's pose. This makes tracking impossible as the tracker relies on at least partial overlap between the input frame and a rendering of the keyframe for the currently assumed live pose. The system recognizes this event, discards the cost volume and switches to manual relocalization.
- Partial tracking loss: Tracking is misaligned (it has run into a local optimum). This can happen e.g. when viewing objects with repeating texture. The user has to identify this and can react by manually switching to relocalization.

During relocalization, the system presents a rendering of the latest keyframe as seen from that keyframe's camera pose. The user repositions the camera until its live view is in alignment with the virtual view, and reinitiates camera tracking. In practice the program is often able to regain tracking lock even if the two views match only roughly.

All tracked camera poses are sent to a 3D viewer application, along with (both dense and semi-dense) textured point clouds for all constructed depth maps.

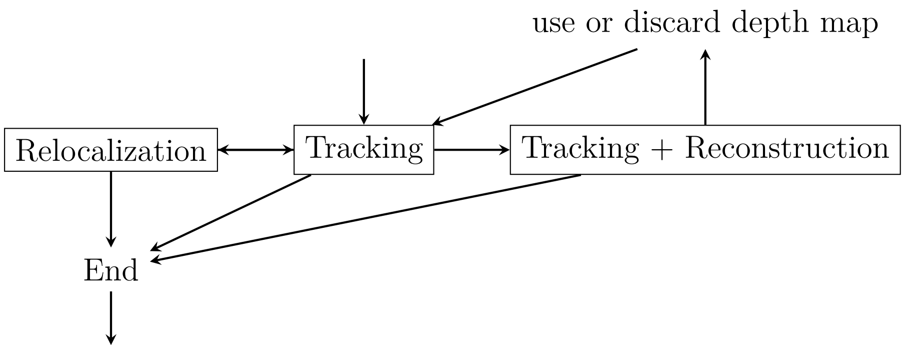


Figure 8.2: User interaction: The user controls all transitions between the program's states. The transition [Relocalization]←[Tracking] is an exception: The tracker can recognize complete tracking failure and automatically switch to (manual) relocalization.

8.2 Performance and architecture

Our program is implemented in C++. All experiments were run on a machine with one Intel Core 2 Duo processor clocked at 3.00GHz and 8GB of RAM. Camera tracking and depth map reconstruction run as parallel CPU threads, and large parts of both are accelerated with Nvidia's CUDA framework which utilizes the computer's dedicated GPU (one Nvidia GTX 680, with 1536 CUDA cores and 2GB RAM)[1]. Additionally, we use extra threads for the retrieval of input data (from a live camera or from a prerecorded dataset) and to write file outputs to disk. All necessary and optional program parameters can be set in verbose configuration files to facilitate the handling of multiple sessions and settings.

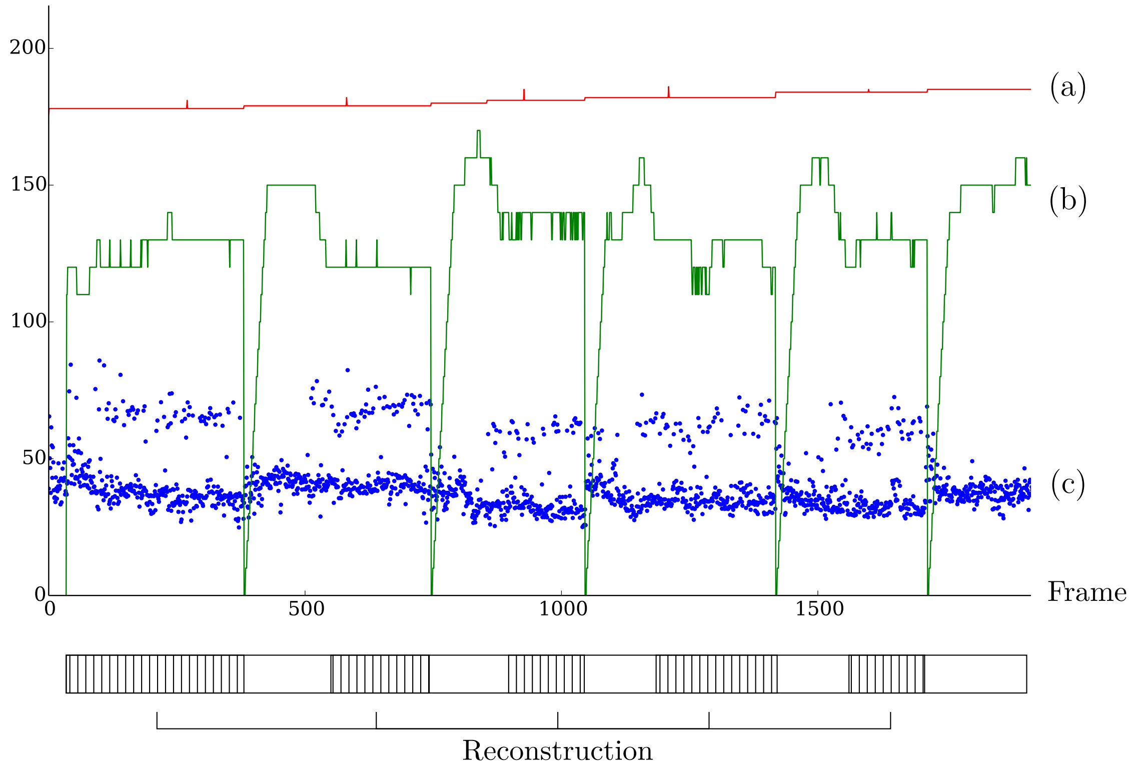


Figure 8.3: Analyzing a run on the dataset for Figure 8.14: (a) GPU memory usage (in Megabyte) slowly increases over time as keyframes are added; (b) Framerate (in 0.1/second) drops slightly when a reconstruction process starts, then shortly dips to 0 while the system waits for ICP alignment, but most of the time stays above 10fps; (c) Time used to track a frame (in milliseconds).

One goal of this work was to achieve realtime performance, which allows the user to directly control the construction of a depth map, especially by providing additional

[1]The computer housed an additional Nvidia Quattro FX 1700 graphics card which was only used to connect the monitors.

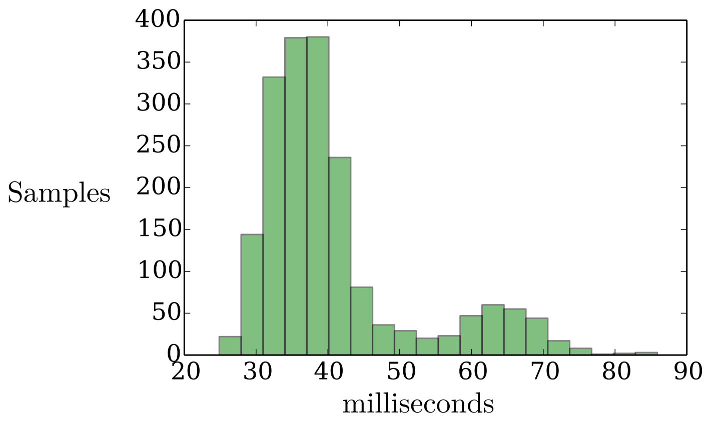


Figure 8.4: Distribution of tracking times in Figure 8.3. The smaller peak around 60–70ms is due to tracking and depth map reconstruction both trying to run on the GPU at the same time.

input data to improve underdeveloped areas, and by recognizing and actively discarding depth maps with reconstruction errors.
This goal has been mostly fulfilled: While the system is not quite able to keep up with the camera's full 30fps, for the majority of an interactive session using VGA (640×480) input images 10-15fps are maintained, placing the user experience well within the region of realtime interactivity. There is only one notable disruption: The ICP alignment step (following the construction of a new keyframe) pauses the system for a few seconds and is due to the fact that ICP does at this time not run in a separate thread[2)].

The tracking procedure runs on grayscale images which reduces the size of the equation system. For the coarse-to-fine pyramid (see Figure 5.2 in chapter 5) a size factor of 0.7 between levels performed best in our tests.
An annotated performance log for a typical interactive session is shown in Figure 8.3 and Figure 8.4. Tracking usually takes less than 50ms per frame, but during reconstruction, tracking and the costly live depth map preview compete for GPU time and some frames take longer, slightly reducing the overall framerate. We chose to reduce the photometric cost volume's resolution to half that of the input images (320×240) but kept a fine depth resolution of 128 levels. This allows us to use normalized cross-correlation as our photometric cost function (section 6.1) while limiting its impact on tracking speed.

[2)]Speed is also impacted by our use of the OpenCV library. Omitting all OpenCV calls for drawing and event queue processing improves performance considerably, but unfortunately means losing all graphical output without which our program is difficult to use.

8.3 Experiments

Throughout the work on this thesis, an Asus Xtion Pro Live[3)] RGB-D camera was used, housing a sensor made by PrimeSense[4)] and identically constructed to Microsoft's widely popular Kinect device. The camera provides an RGB video stream with VGA resolution (640×480) and for each frame a depth map in which each pixel maps to the same pixel coordinates in the color image. Both outputs are available at 30 frames per second. Unfortunately, the camera offers only limited image quality and no external control of exposure or color balance.

Here we present a summary of our experiments and give explanations for our observations. Some example reconstructions of single depth maps are shown in Figure 8.5, compared to depth maps retrieved from our RGB-D camera.

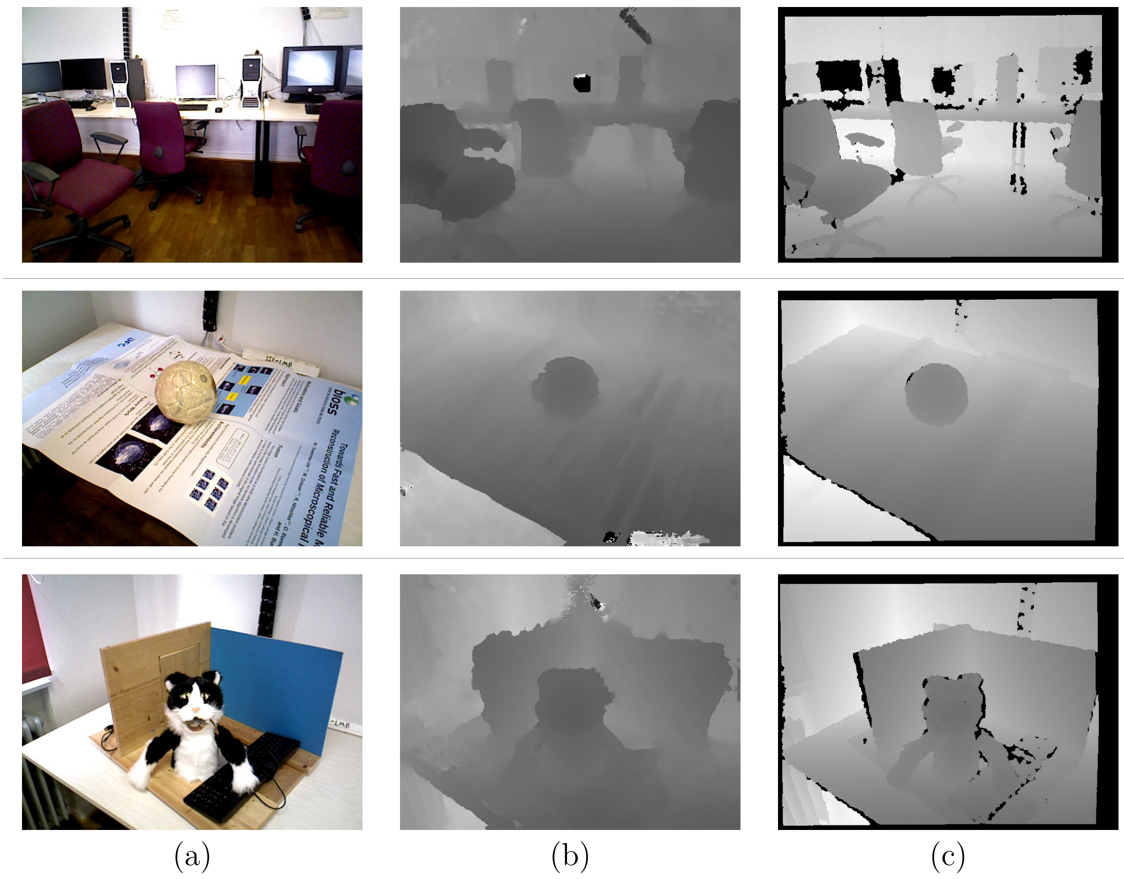


Figure 8.5: Three reconstructed scene views: (a) RGB image for the view; (b) Our reconstructed depth map; (c) RGB-D camera depth map for the same view.

By ignoring pixels with excessive photometric error, our tracker is robust against

3) `http://www.asus.com/Multimedia/Xtion_PRO_LIVE/`
4) `http://www.primesense.com/`

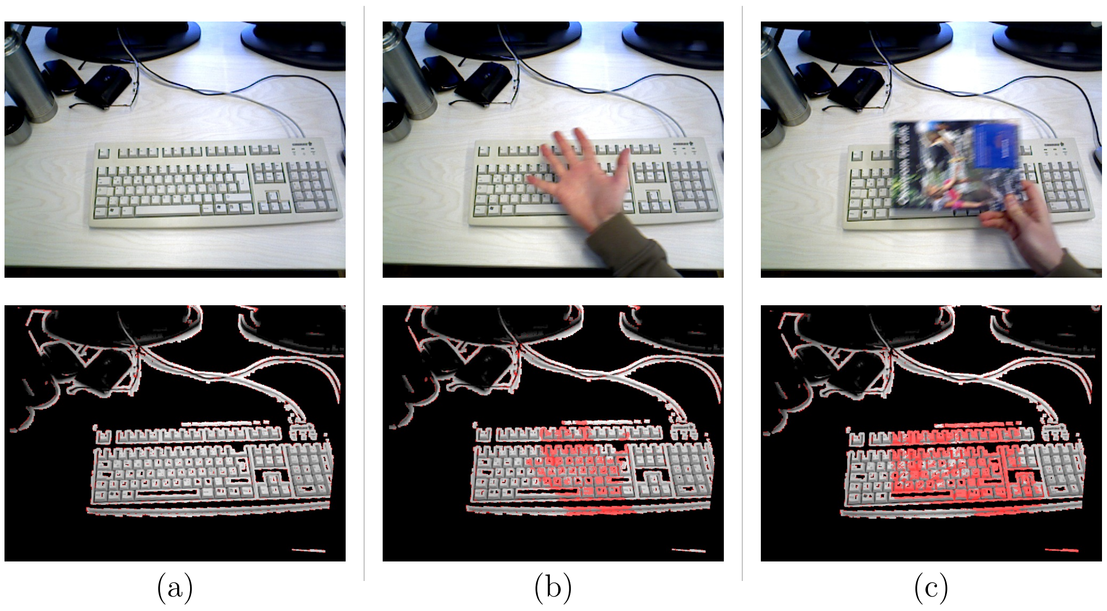


Figure 8.6: Robustifying tracking against small disturbances: *Top row:* Camera image; *Bottom row:* Semi-dense tracking view. (a) Unchanged scene. Even with quickly moving objects that change a significant part of the view in (b) and (c), the tracker does not lose the camera. Red color in the lower row marks where pixels were ignored because their photometric error exceeded the threshold.

small interference. As shown in Figure 8.6, waving objects in front of the camera even at high speed does not lead to failure because the affected areas are excluded from contributing to pose estimation, as long as there is still enough unoccluded keyframe structure visible. The keyboard on the desk is a good source of visual structure for tracking, but its almost periodic structure also bears the risk of local minima (if the view shifts by the size of one key, the images still match well). Note that these disturbances would also not impact the reconstruction of a depth map for this view as long as enough clean frames are fed into the cost volume to average out the erroneous data.

Given good lighting conditions, our system smoothly tracks the camera (Figure 8.7) and yields high quality dense 3D reconstructions, demonstrated in Figure 8.8. However, it often fails to find accurate depth discontinuities along object boundaries which leads to artifacts when rendering the textured depth map for a different viewpoint.

Figure 8.9 shows the reconstruction of a large indoor office scene. The geometry of the room is captured well, although it is apparent that not all keyframes were created with very accurate depth maps. Unfortunately, we were unable to complete the loop because by the time the camera was facing the windows (towards the left in the bottom image in Figure 8.9), lighting conditions had become so unfavorable that tracking was affected and the system could no longer generate good depth maps

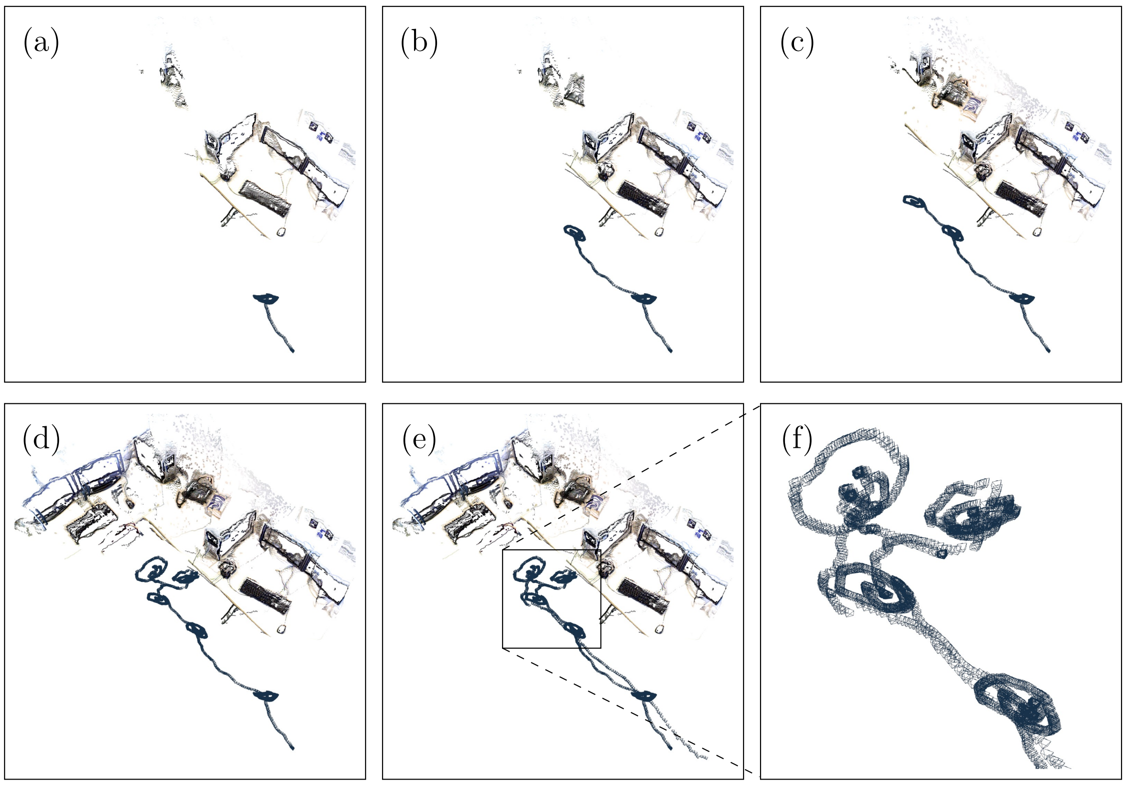
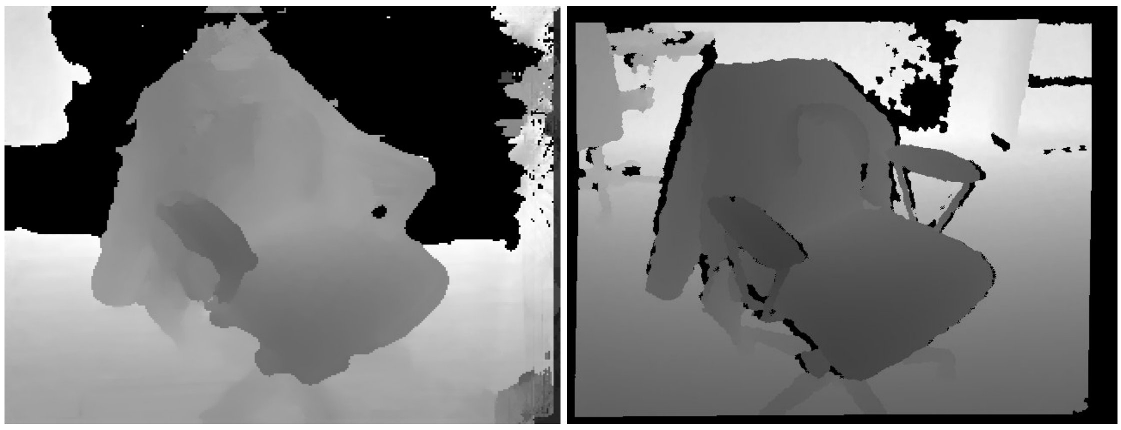


Figure 8.7: Tracking progress for the scene in Figure 8.14: (a)-(d) The camera travels to the next view and collects frames for a new keyframe; (e) Afterwards the camera moves back to the starting point; (f) Closeup of the camera trajectory.



(a) Our reconstruction (b) Asus Xtion Pro Live / Kinect

Figure 8.8: Our reconstruction of a chair with a jacket put over the back, compared to a depth map given by an RGB-D camera. The geometry matches well, except for the fact that our result often fails to extract precise contours, e.g. around the struts holding the arm rests. Black pixels in (b) are due to occlusion which the RGB-D camera cannot fill due to its fixed view. In contrast to that, black pixels in our result (a) mark areas for which the best estimated depths hit the limits of the cost volume. These pixels were subsequently invalidated since it is very unlikely that this exact value is their optimum depth, and we want to avoid wrong depth values. The slight difference in absolute brightness values is due to different minimum depth limits in the RGB-D camera and our program's configuration.

for additional keyframes.

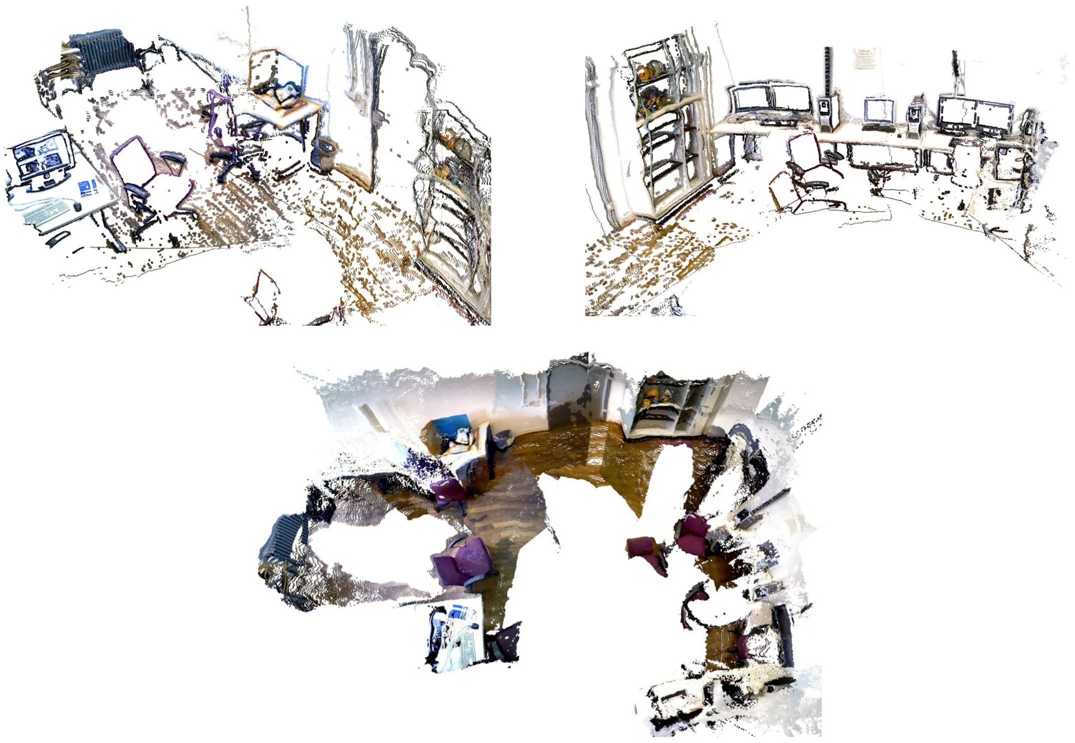


Figure 8.9: A large scene reconstruction (bottom) and two views of the semi-dense keyframes used for tracking. 8 keyframes were created to map the room.

In Figure 8.10 we compare the quality of depth maps extracted from cost volumes with different depth resolutions. With 128 levels (our highest setting and used in all other experiments) the chair is cleanly separated from the floor. At half that resolution, parts of the back “cling” to the background, and the depth map does not capture the true discontinuity along the contour. This becomes even more apparent when reducing the number of available depth labels for SGM to 32, at which point the entire outline of the chair smoothly (and incorrectly) blends into the floor. Note that the effect shown in Figure 8.16 gets worse at lower depth resolutions, too.

The example view in Figure 8.11 shows the difference in quality of a reconstructed depth map between our two tested cost functions (see section 6.1) and justifies our decision to favor the slower normalized cross-correlation (NCC): While the sum of absolute differences manages to capture the rough scene geometry, it yields rough and noisy results. The corresponding view using NCC costs exhibits a much smoother structure and captures details such as the contour of the chair far more accurately (the same data—images and camera poses—was used for both reconstructions).
It is also obvious in Figure 8.12 that the interpolation step we perform on the raw depth maps computed by SGM is absolutely necessary.

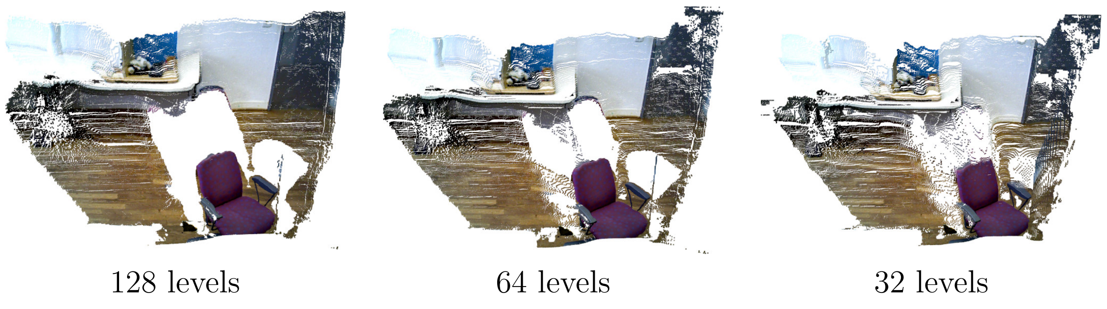


Figure 8.10: Comparison of depth maps computed using different depth resolutions: As the number of discrete depth hypotheses declines (from left to right), some objects are no longer correctly separated from the background (same input for all three volumes).

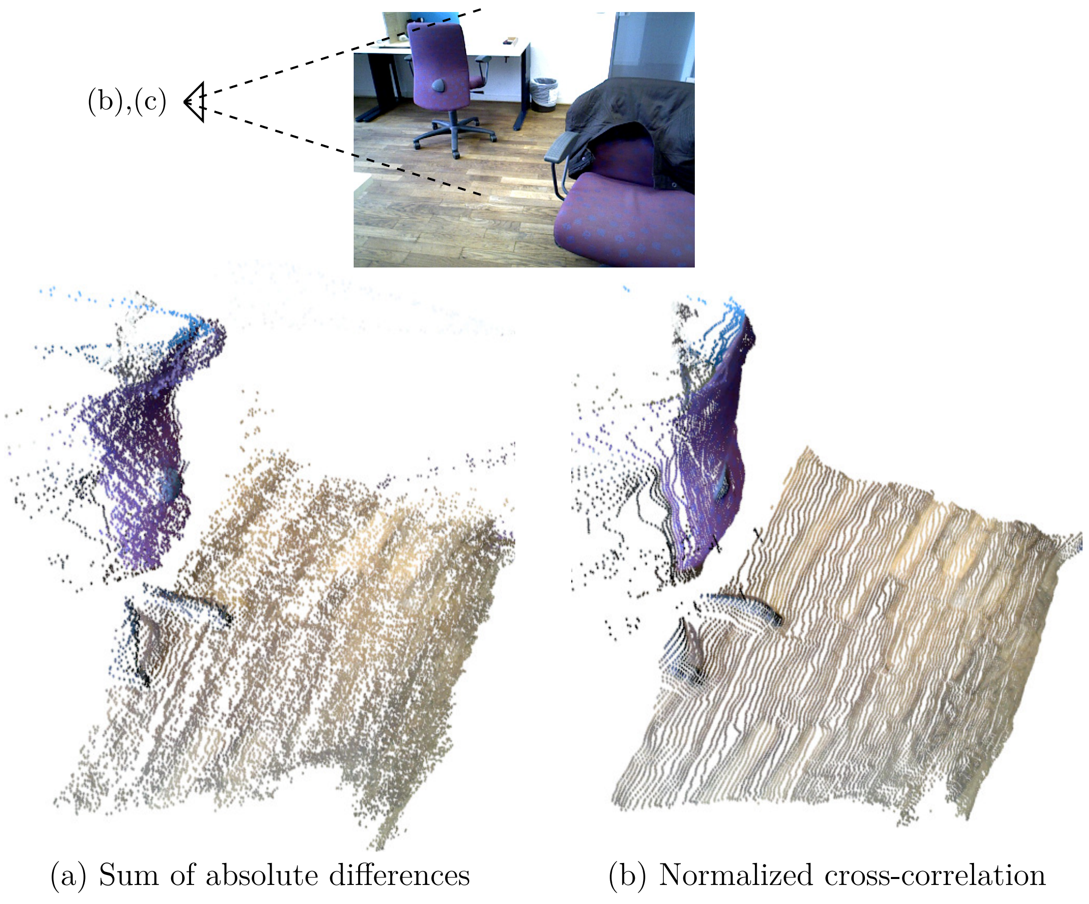
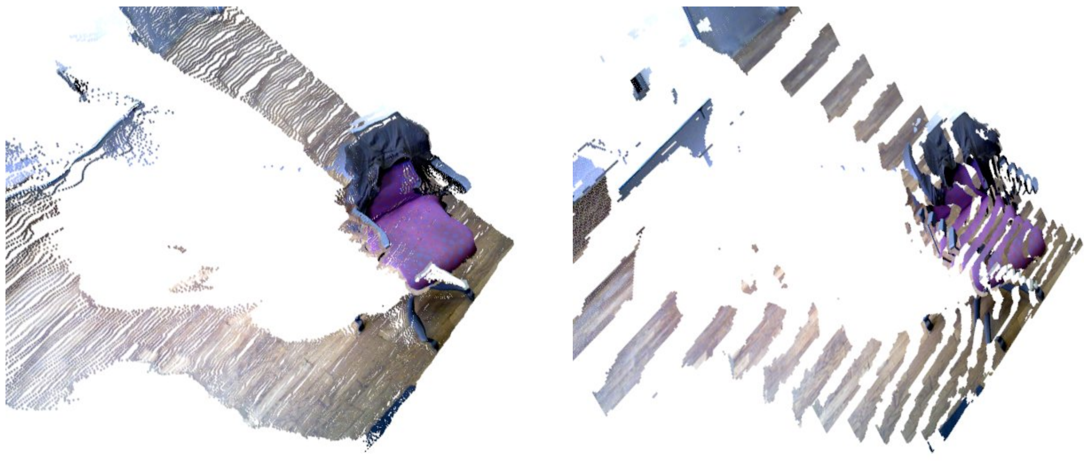

(b),(c)
(a) Sum of absolute differences
(b) Normalized cross-correlation

Figure 8.11: Comparison of photometric cost functions: (a) and (b) look at the back of the chair in the background. The sum of absolute pixelwise distances in (a) is faster, but locally normalized cross-correlation (b) yields better and smoother results. Note that NCC ran within a 7×7 pixel window while the SAD cost function did not use windows.



(a) With interpolation (b) Without interpolation

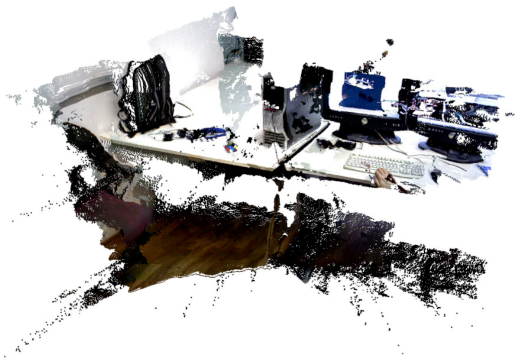
Figure 8.12: Without our quadratic interpolation step (see section 6.1), a reconstructed depth map is just a solution to the cost volume's discrete labeling problem and not a good approximation of the scene's true geometry (see also [1]).

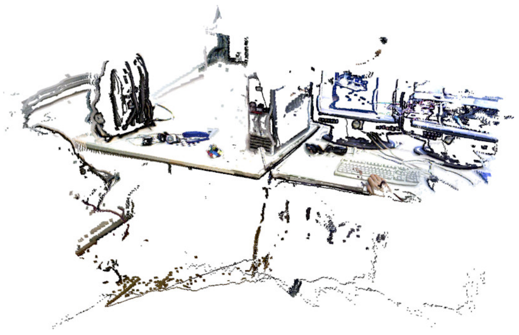
In Figure 8.13 we compare our system with a sparse reconstruction approach. In the latter, initial point correspondences were obtained from optical flow between input frames, and the result was optimized using bundle adjustment on the points and input camera poses. The two reconstructions are comparable in quality: While our program captures structures such as the desk outlines better, we also get duplicate contours where our keyframes' depth maps are inaccurate and do not align well. Note that our result was computed live in less than a minute but required user interaction, while the sparse method worked autonomously yet took over one hour to finish.

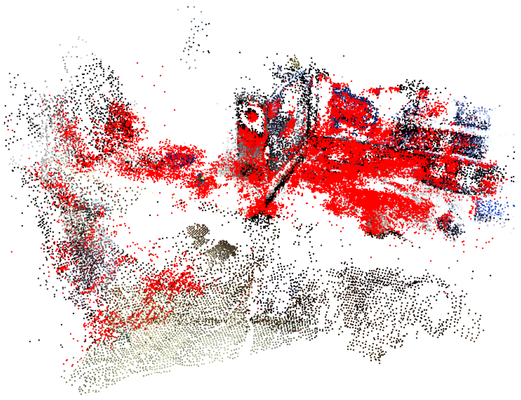
We found that our ICP alignment step following the construction of a new keyframe does not effect large changes (Figure 8.14). On the one hand, this could be attributed to our tracker already yielding very accurate poses for the keyframes so there is nothing upon which ICP could improve, but on the other hand it could also mean that ICP quickly ran into a local optimum because the involved point clouds were not precise enough for a clear best solution.

Failure modes

Fast camera translation or rotation is the most obvious source of tracking failure. Our tracking procedure renders the currently used keyframe for the estimated camera pose, compares this rendering to the live frame, and attempts to adjust the pose estimate such that the two images align. Obviously this requires (a) the camera to keep in view a part of the scene modeled in the keyframe and (b) the live pose


(a)


(b)


(c)

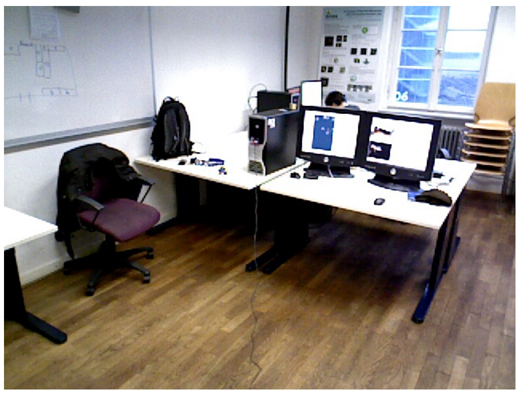

(d)

Figure 8.13: Our program vs. bundle adjustment: (a) Dense geometry obtained from our program in realtime. (b) Semi-dense mapped structures used for tracking. (c) Sparse reconstruction of the same dataset using bundle adjustment on point correspondences computed using optical flow (runtime >1h; red points were marked by the used program and do not represent actual red structures within the scene). (d) The scene.

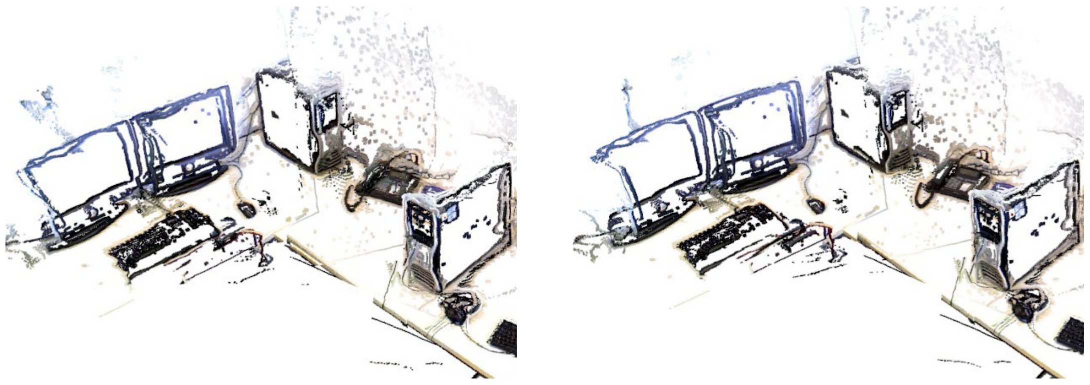


Figure 8.14: ICP alignment of newly constructed keyframes: The results reveal only little difference between using and omitting the additional ICP step.

estimate to be reasonably close to the true live pose. In the case of extremely fast camera motion, either of these conditions can be violated. This leads to immediate complete tracking failure from which the system cannot recover on its own but requires manual relocalization.
Additionally, pose estimation solves only a linearized version of the true alignment problem. This generally means that if the camera moves quickly—but not so fast as to completely lose sight of the keyframe—the large difference in alignment between the live pose and the pose estimate cannot be overcome due to local minima, although a human might find it easy to estimate the required pose change. Our iterative coarse-to-fine scheme alleviates this problem, but it is still required to move the camera with care.

Image quality is another (though related) issue, especially with the camera we used. Figure 8.15 shows the problem with the greatest impact: Even under normal indoor lighting conditions with no direct sunlight, the camera generally yields low quality pictures. Unfortunately, we could not simply switch to a different device since our initialization method (see chapter 7) currently requires the use of an RGB-D camera.

The third problem when it comes to tracking failure stems from our method of reconstructing depth maps, namely using a cost volume of discrete depth hypotheses (see section 6.1). Solving the cost volume as a discrete labeling problem means that each pixel in the depth map has to choose a specific depth from a limited pool, and even if the chosen label is the closest to the correct depth value, the result is at best imprecise. We use a quadratic interpolation step to better approximate the true depths, and although the results are far better than without interpolation, the quadratic model is not generally a good fit in reality.
Figure 8.16 shows "waves" in the estimated depths of the floor. As the camera's perspective moves away from that of the keyframe (for example when circling around an object such as the chair in the figure), the depth map less and less represents the true geometry of the floor, until at some point the difference is so great that

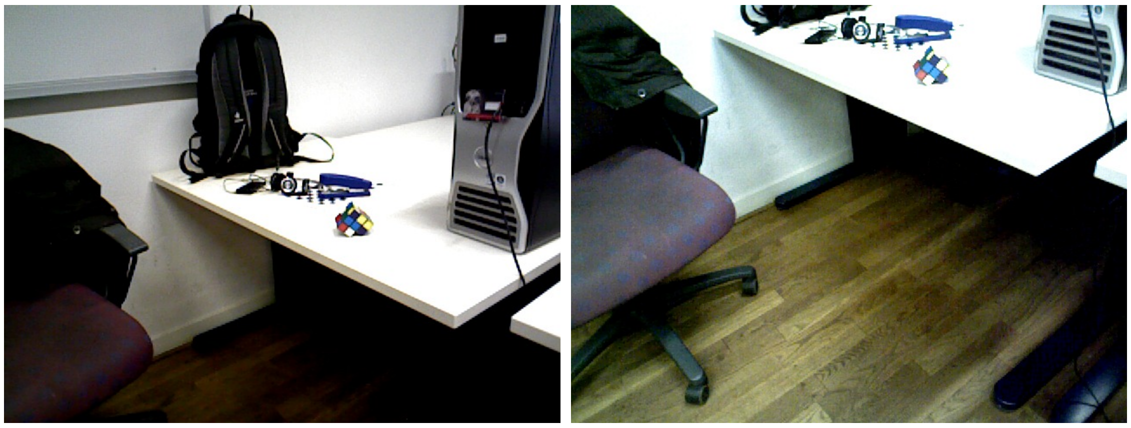


Figure 8.15: Susceptibility to brightness changes: Although lighting conditions were nowhere as unfavorable as the images suggest, the camera simultaneously over- and underexposed. Few areas such as the table's front were usable for tracking, and as the camera tilted down and adjusted brightness, even those were lost. At the time of recording the image on the right side, tracking failed.

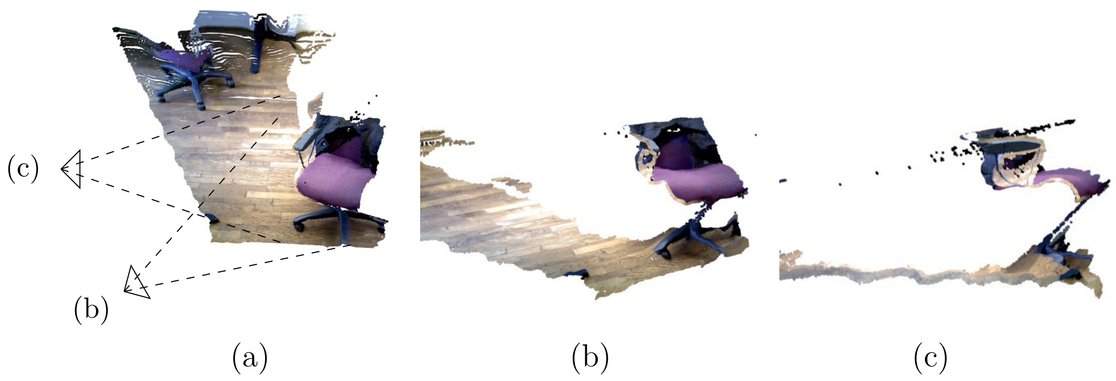


Figure 8.16: The use of a cost volume with discrete steps for a pixel's depth hypotheses leads to artifacts in the resulting depth map in (a). Although we use an interpolation step after extracting a depth map from the cost volume (see section 6.1 or [21] respectively), we are unable to correctly reconstruct the planar floor (yet without this step, the floor is not even a continuous surface). When seen from an oblique perspective such as in (c), the depth map no longer represents the scene with sufficient accuracy and tracking is likely to fail.

the tracking procedure can no longer use it. While this problem could be solved by increasing the number of depth levels available in the cost volume (thus allowing a more finegrained solution to the labeling problem), doing so immediately increases the system's memory footprint and slows down the depth map reconstruction process.

One solution could be to adapt the approach of [17] and to continuously propagate earlier depth maps to the live camera, instead of computing separate depths for individual keyframes. Our discrete-depths problem might thus be solved, since as the camera turns around an object, the depth map is adjusted for each new frame and there is never a situation where the camera sees a depth map from the side as in Figure 8.16.

9 Conclusion and future work

We have presented a realtime-capable SLAM system for dense 3D reconstruction. While our implementation currently needs an RGB-D camera for initialization, the extension to an entirely monocular (RGB-only) setting is straightforward. By tracking the live camera using a semi-dense featureless approach, we focus on input data that provides visual gradient clues for tracking, and ignore pixel regions with poor structure since they contain, at best, ambiguous information.

Our experiments show that the system works at high framerates and within the realm of realtime, however there are currently a number of issues that affect usability: On the software side, a still higher framerate (fully processing all 30–60fps of a video stream) would have two distinct benefits: It would decrease the camera pose difference between two consecutive frames (meaning that pose optimization would converge faster and more reliably), and the (realworld) time needed for constructing a new keyframe would be shorter, which is desirable from the user's point of view. It also appears that our combination of normalized cross-correlation costs and SGM could be problematic since the reconstruction process is often unable to extract precise contours around objects.
Apart from this, the hardware side is also an issue, namely the image quality of our RGB-D camera. Implementing an RGB-only solution will allow us to use better cameras, from which we expect immediate improvements to both tracking and reconstruction.

The fact that both tracking and mapping run in parallel on the graphics processor meant we had to limit the resolution of our cost volume for depth map reconstruction. One possible solution to this problem is the use of a second dedicated GPU purely for mapping, which would improve the framerate and allow for a more detailed reconstruction. However, a more elegant approach would be to run the tracker on the system's CPU so that the GPU could be used exclusively for mapping. That this is possible has been demonstrated e.g. in [17].

Our tracking procedure currently works by selecting and aligning to one specific keyframe at a time, but it is not trivial to decide which keyframe to choose (based on expected coverage, distance to the keyframe's camera pose etc.). An arguably better approach would be to construct a monolithic global model such as a triangular mesh in the DTAM system [1].

A further idea for optimization is the inclusion of a separate thread that continuously performs bundle adjustment on all collected data. Our program should be able to provide a good enough initialization to ensure convergence.

In its current state, our system requires frequent user interaction: The user decides when to start creating new keyframes and when to accept a reconstructed depth map as good enough. This could be improved by automatically initiating new keyframes. For example, the DTAM system [1] keeps track of how much of the live view is covered by the model and starts a new mapping step if the coverage drops too far. The same applies to tracking failure which currently requires the user to manually relocalize the camera. One idea to solve this is to seed sparse feature points throughout the mapped scene. In the case of tracking failure, feature matching could then be used to obtain a rough pose estimate which would enable the dense tracker to regain its lock on the camera. The same approach could assist ICP when aligning a new keyframe to the global model, and when attempting loop closure.

Bibliography

[1] R. A. Newcombe, S. Lovegrove, and A. J. Davison, “Dtam: Dense tracking and mapping in real-time,” in *ICCV*, 2011, pp. 2320–2327.

[2] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. W. Fitzgibbon, “Kinectfusion: Real-time dense surface mapping and tracking,” in *ISMAR*, 2011, pp. 127–136.

[3] A. J. Davison, I. D. Reid, N. Molton, and O. Stasse, “Monoslam: Real-time single camera slam,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1052–1067, 2007.

[4] T. Whelan, M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald, “Kintinuous: Spatially extended KinectFusion,” in *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, Sydney, Australia, Jul 2012.

[5] T. Whelan, H. Johannsson, M. Kaess, J. J. Leonard, and J. McDonald, “Robust real-time visual odometry for dense rgb-d mapping,” in *ICRA*, 2013, pp. 5724–5731.

[6] G. Klein and D. Murray, “Parallel tracking and mapping for small ar workspaces,” in *Mixed and Augmented Reality, 2007. ISMAR 2007. 6th IEEE and ACM International Symposium on*. IEEE, 2007, pp. 225–234.

[7] R. O. Castle, G. Klein, and D. W. Murray, “Video-rate localization in multiple maps for wearable augmented reality,” in *Proc 12th IEEE Int Symp on Wearable Computers, Pittsburgh PA, Sept 28 - Oct 1, 2008*, 2008, pp. 15–22.

[8] A. Davison, “Real-time simultaneous localisation and mapping with a single camera,” in *Proc. International Conference on Computer Vision, Nice*, Oct. 2003.

[9] B. Ummenhofer and T. Brox, “Dense 3d reconstruction with a hand-held camera,” in *Pattern Recognition (Proc. DAGM)*, ser. LNCS. Springer, 2012. [Online]. Available: http://lmb.informatik.uni-freiburg.de//Publications/2012/UB12

[10] N. Sundaram, T. Brox, and K. Keutzer, “Dense point trajectories by gpu-accelerated large displacement optical flow,” in *European Conference on Computer Vision (ECCV)*, ser. Lecture Notes in Computer Science. Springer, Sept. 2010. [Online]. Available: http://lmb.informatik.uni-freiburg.de//Publications/2010/Bro10e

[11] T. Brox, “Computer Vision II (lecture notes),” 2011. [Online]. Available: http://lmb.informatik.uni-freiburg.de/lectures/computer_vision_II/

[12] M. Teschner, “Image Processing and Computer Graphics (lecture notes),” 2011. [Online]. Available: http://cg.informatik.uni-freiburg.de/teaching.htm

[13] O. Ronneberger, “3D Image Analysis (lecture notes),” 2013. [Online]. Available: http://lmb.informatik.uni-freiburg.de/lectures/3D_image_analysis/index.en.html

[14] S. H. Ahn, “OpenGL.” [Online]. Available: http://www.songho.ca/

[15] S. Umeyama, “Least-squares estimation of transformation parameters between two point patterns,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, no. 4, pp. 376–380, 1991.

[16] C.-P. Lu, G. D. Hager, and E. Mjolsness, “Fast and globally convergent pose estimation from video images,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 6, pp. 610–622, 2000.

[17] J. Engel, J. Sturm, and D. Cremers, “Semi-dense visual odometry for a monocular camera,” in *IEEE International Conference on Computer Vision (ICCV)*, Sydney, Australia, December 2013.

[18] C. Kerl, J. Sturm, and D. Cremers, “Robust odometry estimation for rgb-d cameras,” in *ICRA*, 2013, pp. 3748–3754.

[19] M. Pollefeys, D. Nistér, J.-M. Frahm, A. Akbarzadeh, P. Mordohai, B. Clipp, C. Engels, D. Gallup, S.-J. Kim, P. Merrell, C. Salmi, S. Sinha, B. Talton, L. Wang, Q. Yang, H. Stewénius, R. Yang, G. Welch, and H. Towles, “Detailed real-time urban 3d reconstruction from video,” *International Journal of Computer Vision*, vol. 78, no. 2-3, pp. 143–167, 2008. [Online]. Available: http://dx.doi.org/10.1007/s11263-007-0086-4

[20] D. Gallup, J.-M. Frahm, P. Mordohai, Q. Yang, and M. Pollefeys, “Real-time plane-sweeping stereo with multiple sweeping directions,” in *Computer Vision and Pattern Recognition, 2007. CVPR ’07. IEEE Conference on*, 2007, pp. 1–8.

[21] H. Hirschmuller, “Stereo processing by semiglobal matching and mutual information,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 2, pp. 328–341, 2008.

[22] S. Rusinkiewicz, O. Hall-Holt, and M. Levoy, “Real-time 3d model acquisition,” *ACM Trans. Graph.*, vol. 21, no. 3, pp. 438–446, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/566654.566600

[23] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox, “RGB-D mapping: Using depth cameras for dense 3D modeling of indoor environments,” in *Proc. of the International Symposium on Experimental Robotics (ISER)*, 2010.

[24] Y. Chen and G. Medioni, “Object modeling by registration of multiple range images,” in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, 1991, pp. 2724–2729 vol.3.

[25] R. A. Newcombe and A. Davison, “Live dense reconstruction with a single moving camera,” in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, 2010, pp. 1498–1505.

[26] P. J. Besl and N. D. McKay, “A method for registration of 3-d shapes,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 2, pp. 239–256, 1992.

[27] G. Blais and M. D. Levine, “Registering multiview range data to create 3d computer objects,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 17, no. 8, pp. 820–824, 1995.

[28] D. Chetverikov, D. Svirko, D. Stepanov, and P. Krsek, “The trimmed iterative closest point algorithm,” in *ICPR (3)*, 2002, pp. 545–548.

[29] E. Trucco, A. Fusiello, and V. Roberto, “Robust motion and correspondence of noisy 3-d point sets with missing data,” *Pattern Recognition Letters*, vol. 20, no. 9, pp. 889–898, 1999.

[30] S. Baker and I. Matthews, “Lucas-kanade 20 years on: A unifying framework,” *International Journal of Computer Vision*, vol. 56, no. 3, pp. 221–255, 2004. [Online]. Available: http://dx.doi.org/10.1023/B%3AVISI.0000011205.11775.fd

[31] B. Ummenhofer and T. Brox, “Point-based 3d reconstruction of thin objects,” in *IEEE International Conference on Computer Vision (ICCV)*, 2013. [Online]. Available: http://lmb.informatik.uni-freiburg.de//Publications/2013/UB13

[32] B. Curless and M. Levoy, “A volumetric method for building complex models from range images,” in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques.* ACM, 1996, pp. 303–312.

[33] D. M. Cole and P. M. Newman, “Using laser range data for 3d slam in outdoor environments,” in *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on.* IEEE, 2006, pp. 1556–1563.

[34] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss, and W. Burgard, “Octomap: A probabilistic, flexible, and compact 3d map representation for robotic systems,” in *Proc. of the ICRA 2010 workshop on best practice in 3D perception and modeling for mobile manipulation*, vol. 2, 2010.

[35] A. Elfes, “Occupancy grids: A stochastic spatial representation for active robot perception,” *CoRR*, vol. abs/1304.1098, 2013.

[36] F. Steinbrücker, J. Sturm, and D. Cremers, “Real-time visual odometry from dense rgb-d images,” in *ICCV Workshops*, 2011, pp. 719–722.

[37] J. Stühmer, S. Gumhold, and D. Cremers, “Real-time dense geometry from a handheld camera,” in *DAGM-Symposium*, 2010, pp. 11–20.

[38] N. Snavely, S. M. Seitz, and R. Szeliski, “Photo tourism: Exploring photo collections in 3d,” in *ACM TRANSACTIONS ON GRAPHICS.* Press, 2006, pp. 835–846.

[39] F. Steinbruecker, C. Kerl, J. Sturm, and D. Cremers, “Large-scale multi-resolution surface reconstruction from rgb-d sequences,” in *IEEE International Conference on Computer Vision (ICCV)*, Sydney, Australia, 2013.

[40] S. Agarwal, N. Snavely, I. Simon, S. M. Seitz, and R. Szeliski, “Building rome in a day,” in *Computer Vision, 2009 IEEE 12th International Conference on.* IEEE, 2009, pp. 72–79.

[41] J.-M. Frahm, P. Fite-Georgel, D. Gallup, T. Johnson, R. Raguram, C. Wu, Y.-H. Jen, E. Dunn, B. Clipp, S. Lazebnik *et al.*, “Building rome on a cloudless day,” in *Computer Vision–ECCV 2010.* Springer, 2010, pp. 368–381.

[42] A. F. Bobick and S. S. Intille, “Large occlusion stereo,” *International Journal of Computer Vision*, vol. 33, no. 3, pp. 181–200, 1999.

[43] P. Favaro and S. Soatto, “Learning shape from defocus,” in *ECCV (2)*, 2002, pp. 735–745.

[44] R. Szeliski and D. Scharstein, “Sampling the disparity space image,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 26, no. 3, pp. 419–425, 2003.

[45] D. Scharstein and R. Szeliski, “A taxonomy and evaluation of dense two-frame stereo correspondence algorithms,” *International Journal of Computer Vision*, vol. 47, no. 1-3, pp. 7–42, 2002.

[46] Y. Nakamura, T. Matsuura, K. Satoh, and Y. Ohta, “Occlusion detectable stereo-occlusion patterns in camera matrix,” in *Computer Vision and Pattern Recognition, 1996. Proceedings CVPR ’96, 1996 IEEE Computer Society Conference on*, 1996, pp. 371–378.

[47] R. Garg, A. Roussos, and L. Agapito, “Dense variational reconstruction of non-rigid surfaces from monocular video,” June 2013.

[48] J. Jachnik, R. A. Newcombe, and A. J. Davison, “Real-time surface light-field capture for augmentation of planar specular surfaces,” in *ISMAR*, 2012, pp. 91–97.

[49] Z. Zhang, “Flexible camera calibration by viewing a plane from unknown orientations,” in *ICCV*, 1999, pp. 666–673.