Large-Scale Textured 3D Scene Reconstruction

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN (Dr.-Ing.)

von der KIT-Fakultät für Maschinenbau des
Karlsruher Instituts für Technologie (KIT)

angenommene

DISSERTATION

von

M.Sc. Tilman Kühner

Tag der mündlichen Prüfung: 15.10.2020

Hauptreferent: Prof. Dr.-Ing. Christoph Stiller
Korreferent: Prof. Dr.-Ing. Stefan Hinz

Abstract

The creation of three-dimensional models of the environment is a fundamental task in computer vision. They help to solve a variety of tasks ranging from mapping for surveying over documentation of cultural heritage to the creation of virtual worlds for entertainment. In the field of automated driving, they promise to help solving multiple challenges, such as localization, labeling of large datasets or the fully automatic creation of simulation environments.

The challenge of 3D reconstruction is the joint estimation of sensor poses and environment model. Redundant and potentially erroneous data of multiple sensors has to be integrated into a common representation of the world to create a metrically accurate and photometrically correct model. At the same time, the method has to make full use of limited hardware resources and has to have run times which allow for practical use.

In this thesis, we present a reconstruction framework which is capable of creating textured photo-realistic 3D reconstructions of large environments spanning multiple kilometers in length. Range measurements from laser scanners and stereo camera systems are jointly fused using a volumetric reconstruction approach. Loop closures are detected and added as additional constraints to obtain a globally consistent map. The resulting mesh is textured from multiple camera images and using a weighting scheme which weights observations according to their quality. For seamless appearance, the unknown exposure times and parameters of the optical system are estimated to correct each image accordingly.

We evaluate our method on synthetic data as well as on real sensor data from our own experimental vehicle and from publicly available datasets. Qualitative results of large inner city areas are shown and quantitative evaluations of the vehicle trajectory and mesh quality are presented.
Lastly, we show multiple applications and prove the applicability of our framework for automated driving.

Zusammenfassung

Die Erstellung dreidimensionaler Umgebungsmodelle ist eine fundamentale Aufgabe im Bereich des maschinellen Sehens. Rekonstruktionen sind für eine Reihe von Anwendungen von Nutzen, wie bei der Vermessung, dem Erhalt von Kulturgütern oder der Erstellung virtueller Welten in der Unterhaltungsindustrie. Im Bereich des automatischen Fahrens helfen sie bei der Bewältigung einer Vielzahl an Herausforderungen. Dazu gehören Lokalisierung, das Annotieren großer Datensätze oder die vollautomatische Erstellung von Simulationsszenarien.

Die Herausforderung bei der 3D Rekonstruktion ist die gemeinsame Schätzung von Sensorposen und einem Umgebunsmodell. Redundante und potenziell fehlerbehaftete Messungen verschiedener Sensoren müssen in eine gemeinsame Repräsentation der Welt integriert werden, um ein metrisch und photometrisch korrektes Modell zu erhalten. Gleichzeitig muss die Methode effizient Ressourcen nutzen, um Laufzeiten zu erreichen, welche die praktische Nutzung ermöglichen.

In dieser Arbeit stellen wir ein Verfahren zur Rekonstruktion vor, das fähig ist, photorealistische 3D Rekonstruktionen großer Areale zu erstellen, die sich über mehrere Kilometer erstrecken. Entfernungsmessungen aus Laserscannern und Stereokamerasystemen werden zusammen mit Hilfe eines volumetrischen Rekonstruktionsverfahrens fusioniert. Ringschlüsse werden erkannt und als zusätzliche Bedingungen eingebracht, um eine global konsistente Karte zu erhalten. Das resultierende Gitternetz wird aus Kamerabildern texturiert, wobei die einzelnen Beobachtungen mit ihrer Güte gewichtet werden. Für eine nahtlose Erscheinung werden die unbekannten Belichtungszeiten und Parameter des optischen Systems mitgeschätzt und die Bilder entsprechend korrigiert.

Wir evaluieren unsere Methode auf synthetischen Daten, realen Sensordaten unseres Versuchsfahrzeugs und öffentlich verfügbaren Datensätzen. Wir zeigen qualitative Ergebnisse großer innerstädtischer Bereiche, sowie quanti-

tative Auswertungen der Fahrzeugtrajektorie und der Rekonstruktionsqualität. Zuletzt präsentieren wir mehrere Anwendungen und zeigen somit den Nutzen unserer Methode für Anwendungen im Bereich des automatischen Fahrens.

Acknowledgment

This dissertation was written during my work as a research assistant at the Institute for Measurement and Control Systems (MRT) at the Karlsruhe Institute of Technology (KIT) and at the FZI Research Center for Information Technology. I would like to thank my supervisor Prof. Dr.-Ing. Christoph Stiller for giving me the opportunity to be part of his group. Without the freedom and time I could dedicate to this work it would not have been possible. Also I would like to thank my co-supervisor Prof. Dr.-Ing. Stefan Hinz for the supervision of my thesis and the interest in my work. I want to thank my group leader Dr. Martin Lauer for all the help he provided. I want to thank Johannes Gräter for all the help when I was new at the institute and for motivating me to stick to the topic of this work. Thanks to Julius Kümmerle for proof reading this dissertation and for all the collaboration in recent years, especially during the last four months of writing. I would like to thank our department heads at FZI, Sahin Tas and Niels Ole Salscheider, for the many hours they spent on taking care of our department. Finally, I want to thank all my colleagues for the wonderful time we had together.

Karlsruhe, in March 2020 *Tilman Kühner*

Contents

List of Symbols and Abbreviations

Acronyms

BRDF	Bidirectional reflectance distribution function
CAD	Computer-aided design
CGI	Computer created imagery
CPU	Central processing unit
CRF	Camera response function
DoF	Degrees of freedom
GPS	Global positioning system
GPU	Graphics processing unit
HD	High definition
HDR	High dynamic range
ICP	Iterative closest point
IMU	Inertial measurement unit
IR	Infrared
IRLS	Iterative reweighted least squares
KD	K-dimensional
LiDAR	Light detection and ranging

MRF	Markov random field
MRI	Magnetic resonance imaging
PCA	Principal component analysis
RGB-D	Red-green-blue-depth
SDF	Signed distance function
SfM	Structure from motion
SIFT	Scale invariant feature transform
SLAM	Simultaneous localization and mapping
SLERP	Spherical linear interpolation
TSDF	Truncated signed distance function
VR	Virtual reality

Symbols

a, A	scalar, scalar function
$\mathbf{a}$	vector, vector-valued function
$\hat{\mathbf{a}}$	normalized vector
$\tilde{\mathbf{a}}$	homogeneous coordinates
$\mathbf{a}^\top$	transposed vector
$\mathbf{A}$	matrix
$\mathcal{A}$	set

1 Introduction

1.1 Motivation

Autonomous driving gained momentum in recent years with millions of kilometers having been driven without human intervention. This is possible due to specialized sensors, high computational power and new algorithms. For an autonomous system to work, several software modules have to cooperate. Perception provides a high level interpretation of raw sensor data about obstacles and traffic participants. Planning and behavior generation take these as input in order to determine how the vehicle should move in order to reach its goal destination while being compliant to traffic rules and guaranteeing a safe and comfortable ride for the passenger. Lastly, localization determines where the car is located in order to be able to determine where to go next and to use information from maps if provided. Maps are still widely used for driving, as they facilitate numerous tasks like predicting other road users or inferring traffic rules that otherwise would have to be reasoned about online.

3D reconstruction touches most of the previously mentioned topics, as a reconstruction is basically a map that is build from LiDAR and camera data which makes it a perception task. To create the reconstruction, localization falls off as a byproduct.

Numerous applications for reconstructions exist from the field of autonomous driving, as well as other areas. These will be introduced briefly in the following sections.

1.1.1 Localization

In general, localization describes a method which determines a sensor's pose in a previously created map from sensor observations, or, in the case of si-

multaneous localization and mapping (SLAM), localization is performed at the same time as mapping. Since GPS is too unreliable in inner city areas due to buildings shadowing satellites and other effects, autonomous cars rely on cameras and LiDARs to localize themselves. It is a fundamental task in autonomous driving for the creation and usage of HD maps. In order to make use of the information they provide, one has to know the precise position of the vehicle within the map. This allows to drive within a lane without having to detect road markings or curbs. Also, the state of a traffic light can be observed by looking at its mapped location which makes its detection unnecessary.

1.1.2 Planning

Planning and obstacle avoidance is a crucial part of any autonomous system, and it is tightly coupled to localization. Given the own position within a map, the task is to reach a destination under certain constraints which can be shortest time and collision avoidance. This is most challenging in uncontrolled environments encountered by drones or other robots which navigate indoors and where traversable area is not as obvious as a road. Further, a model of the world can provide information about visibility of the scene which is of great importance for autonomous cars.

1.1.3 Object Detection and Tracking

LiDAR is the primary sensor for detection and tracking of objects on most self driving platforms. Traditional methods first perform segmentation of the point cloud to obtain objects. However, this is quite challenging since it is hard to say which point belongs to an object and which point lies on the road, especially when the measurements are at far distances. A 3D model of the world makes this task very easy if precise localization is given as well. Each point can then be compared to the 3D model. If it is close to the model, its likely part of the static scene. Otherwise, the point should be considered to be dynamic.

1.1.4 Simulation

To test and verify new algorithms, researchers often use simulations. Testing on a real vehicle is in most cases much more time-consuming than running a simulation. A test vehicle is a valuable resource which most of the time is not readily available. Further, simulation allows to create scenarios that would not predictably appear in real traffic, like specific weather conditions or road participants acting in a certain way. Lastly, simulation allows to test algorithms on thousands of kilometers per second, as they can be run in parallel and are not constraint to real time.

Various simulation environments exist and all of them use textured 3D meshes to represent the environment. Creating a digital twin of a real world route requires designers to manually replicate the real scene which is an expensive and time consuming task. Further, the manual process is prone to leaving out small but significant detail. In simulation, geometry is often composed of basic geometric primitives. The walls of a building are modeled as ideal planes, and therefore lack fine details such as the structure of the individual bricks. The same holds for texture. The standard approach is to create a texture patch of e.g. asphalt, and then tile all roads in simulation using the identical patch over and over again. This is often not noticeable to the human observer, but it can lead to significant overfitting when the simulation is used for the training of algorithms.

1.1.5 Engineering and Infrastructure

In engineering, multiple tasks demand for reconstructions. Amongst them is the creation of digital models to create CAD models of objects or plans of buildings which is commonly referred to as reverse engineering. Other tasks are surveillance and the detection of defects of manufactured parts or plants.

In order to plan infrastructure, communes digitize entire cities. Measurements can then be derived from the model after leaving the site, and the effects of structural interventions on the environment can be simulated.

1.1.6 Cultural Heritage

A lot of effort was spent on capturing cultural heritage. This can have the goal of having exact virtual copies at hand when restoration is needed or to make them accessible remotely for educational purposes. One of the most prominent projects being the Digital Michelangelo Project [LPC+00] which had the purpose of digitizing Michelangelo's David statue using laser scanning technology.

1.1.7 Virtual Reality and Entertainment

The emergence of cheap head mounted displays created a lot of possible applications for VR. Digitized worlds can be visited which allows for digital tourism and exhibitions which purely exist on computers. The entertainment industry uses reconstruction techniques to integrate real world buildings into video games or CGI where they can, for instance, be destroyed.

1.2 Problem Formulation

3D reconstruction deals with the creation of digital copies of objects or scenes from sensor data. This can be range data from LiDAR or any other depth sensor, such as RGB-D cameras like the Kinect, structured light projections or stereo cameras. The challenges are the following:

- Unknown sensor poses:
  Only raw sensor measurements serve as input to the reconstruction pipeline. To fuse all measurements in a common coordinate frame, the individual sensor poses need to be determined at the same time as the reconstruction is performed. The process of jointly inferring shape and poses is commonly known as SLAM.
- Sensor characteristics:
  The raw sensor measurements we use are the product of a long processing chain (see subsection 3.3.1 as an example). As all stages introduce small errors by either simplified assumptions or limited measurement accuracy,

the raw data is always noisy. Other effects, such as rolling shutter for rotating LiDARs or the image formation process of cameras, have to be considered. We use the term rolling shutter to refer to the effects which are caused by a sensor not recording all data points of a sensor frame at a single point in time. Lastly, all sensors have to be intrinsically and extrinsically calibrated in order to turn a raw measurement like, for example, a range into a 3D point.

- Sampling, redundancy and missing data:
  Sensor resolution and measurement quantization are often the limiting factors for perceiving the world. The challenge is to combine many low resolution or incomplete sensor frames to build a single representation of the scene with a higher level of detail and completeness than the individual frames. This process is referred to as fusion and it often incorporates prior knowledge about the world.

- Texturing:
  For a visually appealing result, the camera images have to be projected onto the 3D structure in a way that creates a sharp and consistent scene from any viewing angle. Camera images and range measurements might not come from the same point of view or the same point in time. Further, occlusions have to be considered when selecting the correct view for a specific part of the scene.

- Dynamic objects:
  We use the assumption that most parts of a scene are static. Dynamic objects like cars and pedestrians are unavoidable during recording, and therefore have to be treated as outliers in our pipeline. This requires the reconstruction as well as the texturing methods, to be robust against erroneous input data to a certain degree.

- Large scenes:
  Most reconstruction approaches only deal with a single object which is recorded in a controlled environment, such as a laboratory. For spacious outdoor scenes, like inner city areas spanning multiple kilometers, aspects such as loop closures and efficiently managing data and computational resources become important.

1.3 Outline and Contribution

In the following chapter, the current state of the art concerning volumetric reconstruction, LiDAR SLAM and texturing is presented.

After that, we provide fundamentals about coordinate systems and transformations, parallel algorithms, spatial data structures, sensor measurement principles, point cloud registration and nonlinear optimization of which we will make use of in later chapters.

Then, we describe the volumetric 3D reconstruction method which is used in this work for the geometric reconstruction part. This includes the sequential processing of range measurements which covers preprocessing, sensor pose estimation, fusion into a world model, surface extraction and finally post-processing of the meshed model. Since the proposed methods are computationally expensive, they are implemented on graphics hardware which requires specific adjustments in order to leverage the parallelism they provide. These adjustments will be described in detail.

The volumetric reconstruction approach is then incorporated into a framework for large-scale reconstruction of areas of arbitrary size. This part deals with how a consistent map can be created if parts of the scene are revisited multiple times which requires loop closure detection.

The following chapter describes the appearance reconstruction. It deals with how camera images can be used to texture the mesh from the previous steps by mapping information from 2D to 3D. Camera poses have to be optimized for photo-consistency and the best views for all triangles of the mesh have to be determined considering occlusions. Finally, all color information has to be fused and photometrically corrected to create a consistently textured model that looks photo-realistic to a human observer.

Next, the reconstruction results are evaluated on real data as well as on synthetic data from simulation for precise ground truth and to investigate the influence of individual assumptions or errors. Both, estimated sensor poses and accuracy of the reconstruction are evaluated, as they are the most relevant quality criteria for any application using reconstructions. We also show the influence of certain parameters on the reconstruction result and compare different approaches from literature to find the best performing one.

In the next chapter, we motivate and implement three applications from the field of autonomous driving which make use of our reconstructions and show the results that we achieve.

Lastly, conclusions are drawn and an outlook is provided which addresses promising directions for future work and how the results can be improved in the future.

The contributions of this work are as follows:

- The method of volumetric fusion is applied to the field of autonomous driving by efficiently modeling rotating LiDAR sensors using a cylinder projection model. Point splatting is applied to obtain dense projections with high resolution.
- A framework is presented which allows to automatically create textured meshes of large outdoor scenes. Multiple range sensors of different types can be fused. The framework is capable of creating consistent maps in the case of loop closures by alternating between pose and geometry estimation. Due to our texturing approach, it is further capable of creating reconstructions with photo-realistic appearance.
- We provide an in-depth evaluation of our framework. Relevant effects that occur with rotating LiDAR sensors are discussed and evaluated. We prove the framework's capabilities by creating large high quality reconstructions using publicly available datasets.
- Multiple applications which make use of our framework are presented. We use our method to solve challenging tasks from autonomous driving, such as localization, simulation and image annotation of large datasets.

2 State of the Art

This work deals with the reconstruction of the geometry and appearance of large scenes. We therefore split this chapter into multiple sections. First, we provide an overview over the volumetric reconstruction method that we use later. Then, we show how loop closures are handled by different reconstruction approaches, how they deal with uncertainty and different scales, and what other extensions to the original method were proposed. We show an overview over different texturing methods. Next, multiple existing reconstruction frameworks are listed together with their capabilities. Then, we discuss state of the art LiDAR SLAM, since LiDAR is the main sensor used for reconstruction in this work. Finally, we provide an overview over other reconstruction approaches along with their strengths and weaknesses.

2.1 Volumetric Reconstruction

In [CL96], the authors propose a method for fusing multiple depth images by updating an implicit signed distance function in a voxel grid along the line of sight of each depth measurement. Each voxel holds a running average of the distance to its nearest surface and a weight. New measurements can be integrated sequentially by a recursive update scheme for each voxel. The surface is implicitly represented as the zero iso level of the distance field and can be extracted using the marching cubes algorithm [LC87]. Further, the authors prove that the reconstructed surface is optimal in a least squares sense under certain assumptions.

However, it wasn't until the emergence of commodity RGB-D cameras, such as the Kinect sensor and powerful graphics cards that the method found widespread use in the scientific community. [NIH+11, IKH+11] fuse the depth stream of a Kinect into a fixed size voxel grid in real time while performing an ICP for each new frame to the world model in order to get the precise sensor

pose. In contrast to frame-to-frame ICP, this method vastly reduces sensor drift. Also, the method allows the user to interactively build 3D models in real time while moving the sensor around.

This publication lead to significant improvements in the following years. One major drawback of [NIH+11] is the large memory consumption which is a result of the fixed size grid and which limits the method to rather small scenes, such as a tabletop. An approach to reconstruct larger scenes is to shift the entire voxel grid along with the scanner. This method is employed in [WMK+12] and allows the authors to reconstruct multiple connected rooms as well as street scenes. Another approach is shown in [HFBM13] where multiple dense voxel grids, which they call patch volumes, are utilized to only cover occupied area. This approach also allows for loop closure by optimizing a pose graph over all patches.

Since most voxels of the grid do not hold any values, one does not want to store them in memory. Therefore, sparse representations are more efficient. In [ZZZL13] and [SKSC13], octrees are used and [CBI13] uses a similar hierarchical data structure to only keep voxels in memory that are close to a surface and therefore hold information. This vastly reduces memory consumption, however, it adds complexity and runtime when querying a specific voxel. Another approach which exploits the sparsity of the voxel grid but has no need for hirarchical data structures is called voxel hashing and was first introduced in [NZIS13]. To access a voxel, its coordinates are used to compute a hash value that points to the respective location in memory.

2.2 Loop Closures

Since graphics cards usually have much less memory than the host device, some methods make use of streaming data. This means that voxels are continuously moved between GPU and CPU such that the GPU memory only holds the data of the currently relevant parts of the scene. [WMK+12] stores entire layers of voxels, which leave the dense grid, on the host. [NZIS13] defines a spherical area which approximates the sensor's viewing frustum to detect voxels which leave the visible area and transfers them to a spacial data structure in device memory. Streaming data in and out only works in cases where the sensor moves back to a previously mapped area on the same path it left the area. This is not

the case for loop closures where a location is visited coming from different directions. Due to the accumulation of sensor drift, the current observations cannot be integrated into the world model without creating inconsistencies. To avoid this, multiple approaches exist.

[SKSC13] estimates sensor poses in a keyframe-to-keyframe manner and handles loop closures by means of a pose graph in which each edge represents registration costs of a pair of keyframes. A similar approach is used in [DNZ+17] where keyframes together with SIFT features are used to build a globally consistent map. Short sequences, called chunks, are processed with the first frame of a chunk defining the keyframe. Once a pose is updated due to a detected loop closure, its corresponding depth map is reintegrated by first fusing the depth map into the voxel grid with a negative weight, thus removing it from the grid, and then integrating it using the new pose. This is similar to [FTF+15] where multiple local grids are used to obtain small parts of the map with low drift. To obtain the global map, they optimize the submap poses using ICP and then blend the submaps in areas in which they overlap. Like [DNZ+17], the authors of [WWL16] use visual features to detect loop closures. A pose graph is optimized with keyframes as its nodes and costs from ICP as its edges. After optimization, the world model is rebuild using the optimized poses. Surfel-based methods, which we will introduce in section 2.8, such as [WLSM+15], have an advantage when it comes to closing loops, as the world model can easily be deformed to make the map consistent. They make use of a deformation graph. Each surfel is transformed by a weighted affine transformation stored within each node of the graph. The weights correspond to the distance of the node to the surfel.

2.3 Scale and Uncertainty

In cases where sensor frames with vastly different distances to a surface are integrated, it can happen that fine details observed from close distance are smoothed out by observations from far distances. This is due to the fact that a pixel's footprint depends on the measured distance. In reconstruction, this phenomenon is also referred to as scale. Several publications deal with this topic. [FG11] has multiple voxel grids for different scales. [MKG11] builds a confidence map in which each sample point adds the same amount of

confidence to a region of the grid proportional to the scale. The final surface is the one with highest confidence and can be extracted via graph cut. [FG14] considers scale by a weighting function which is defined in a local coordinate system around the measurement point and surface normal.

The weighting function in volumetric reconstruction allows for a natural way to incorporate measurement uncertainties in the fusion process. [NIH+11] uses a weighting scheme which is proportional to the cosine of the angle of incidence and inversely proportional to the measured distance. This assigns lower weights to far measurements and those under grazing angles.
In [BSK+13], multiple weighting functions are evaluated. Under the assumption that individual range measurements are Gaussian distributed with zero mean around the actual range, the weighting function becomes also Gaussian. This led to the best tracking result in their evaluation, whereas linear weights led to the most accurate reconstruction. However, differences were quite small. Exponential weighting is also used in other publications like [Dry16].
In many publications, the weight accounts for the fact that the area in between sensor and measurement can be observed to be free space, whereas the area behind the measurement could be free space or inside an object. This can be expressed by an asymmetric weighting function which assigns higher weights to voxels in front of the surface and falls off quicker behind the surface. This approach was implemented in [FG14] and [BSK+13].
A common method is to update all voxels in observable free space even those voxels which lie far away from any surface. This removes erroneous surfaces from the voxel grid caused, for example, by dynamic objects.

2.4 Implementations and Frameworks

Most methods presented here leverage the parallel computation power of graphics hardware. Some exceptions exist, mostly implementations on mobile devices, such as smartphones or drones [OKI15, KDSX15, SSHP15, MSC+16, YGS17]. As most phones do not have depth sensors, they employ motion stereo or SfM techniques to generate depth maps and use the build in IMU to improve tracking.

A couple of open source frameworks for volumetric fusion exist. Open3D[1] provides depth fusion [ZPK18] and texturing, as described in [ZK14]. It is targeted towards indoor reconstructions and does not use graphics hardware. InfiniTAM[2] is optimized for real time applications and also runs on mobile devices [KPM16, KPR+15]. It allows for large-scale reconstructions [BLPG18] but does not provide surface texturing. PCL[3] provides a basic fusion framework, called KinFu, which implements the work of [NIH+11] using a fixed size voxel grid, and therefore comes with the limitations previously described.

2.5 Texturing

Since most of the methods introduced above use RGB-D sensors, color information for each depth sample is available. It can be easily integrated into the voxel grid by extending each voxel by additional running averages for the red, green and blue color channel [SSC14, BSK+13, Dry16, FG14]. When extracting the final mesh from the voxel grid, the color of each vertex can be obtained by interpolating the color values at the respective vertex position. During registration of new sensor frames, color information can be used as an additional matching criterion by minimizing not only geometric distances, but also distances in color space. In [WLSM+15], this method could improve tracking results in areas with less geometric features. While easy to implement, the method creates a washed out appearance of the reconstruction due to simple color averaging. There are other, more advanced methods for texturing which will be described in the following.

Most methods for automatic object texturing use a similar approach. First, for each triangle the best camera image is chosen under the additional constraint to texture neighboring triangles from the same image if possible. This minimizes the number of seams which can appear on the textured reconstruction. The best view can be determined by the projected triangle size in the image thus preferring close and orthogonal views on a surface. Some methods incorporate gradient magnitudes to reject views suffering from being out of focus or having

[1] http://www.open3d.org
[2] https://github.com/victorprad/InfiniTAM
[3] http://pointclouds.org

motion blur.
Camera selection can be formulated as a discrete labeling process using an MRF. One of many methods using this approach is [LI07]. The authors perform seam leveling by an additive term which is optimized to enforce color-consistency of neighboring texture patch borders. [WMG14] builds upon this method and introduces some improvements, like a mean-shift approach to find agreeing views for each triangle which helps to filter out views which are corrupted by dynamic objects. They model the color distribution of a world point, seen in multiple images, as a multi-variate Gaussian and iteratively determine inlier samples by their Mahalanobis distance to the mean. This is a similar approach to [GKK+04] where views are rejected if they are too far away from the mean color. Further, they utilize Poisson blending [PGB03] within a margin around patch borders to achieve smooth transitions. In [XLL+10], the colors of adjacent texture fragments are adjusted in two steps. First, a scalar factor is applied to each patch, and then gradient-domain image blending removes remaining artifacts.

A common problem when texturing a mesh from reconstruction is the fact that the mesh deviates from the actual geometry or might be incomplete. Therefore, a simple projection of images onto the mesh causes inconsistencies. Other effects like erroneous camera poses, erroneous calibration parameters or the mesh being too coarse can have the same effect. To diminish these effects, the texture can be warped to increase photometric consistency.
[EDDM+08] determines optical flow between input images and a rendered image from a different viewpoint. The individual flow fields are then combined by weighted averaging, and applied to the image for texturing. [AMK10] uses sparse feature points to determine a displacement map for each image. They use thin-plate splines for interpolation of the map in between feature points. In [ZK14], the authors first optimize camera poses for photo-consistency. Then, they optimize the position of the nodes of a transformation lattice. In between the nodes, they linearly interpolate the transformation which is applied to the corrected texture. [GWO+10] compensates for mesh irregularities by allowing each triangle on the texture map to be moved by some small discrete value in each of the two directions of an image. This, however, is not a differentiable problem, and therefore is costly to optimize.
While many approaches texture a triangle from a single view only, there are other methods which combine images from different views. In [JJKL16], multiple views are selected creating sub-textures which are then blended. To

reduce ghosting, texture coordinates within each sub-texture are optimized. [GKK+04] blends textures per pixel by weighting each contribution with the footprint size of the pixel in the respective camera image. This, as a result, favors color information from close distance views.

One major difference in texturing is the way the texture is stored. Many approaches, such as [ZK14], use per vertex colors which limits the texture resolution to the mesh resolution. For visualization, the pixel on a triangle is determined by interpolating its three vertex colors.
A more efficient method is a texture atlas. Each vertex of the mesh is mapped to a 2D location on the atlas. This is the representation used in [JJKL16,WMG14]. It allows the meshes to have far less triangles while the reconstructions still look detailed due to their high texture resolution.

Most methods assume the textured object to be Lambertian which means that only one color value has to be determined for each point on the surface of the object. However, there are methods which allow to model more complex material properties.
[MKC+17] and [GXY+17] use spherical harmonics which are parametrized with the surface normal to estimate the scene lighting. Others add a reflective component to the diffuse component which can be modeled using BRDFs. In [WZ15], the authors use a reflective sphere to create an illumination map of the scene first. They then estimate a BRDF using the active ranging sensor of a Kinect to determine the ratio of reflected light and emitted light of its IR channel. Others jointly estimate lighting and BRDF, such as [WWZ16].
All methods determine their parameters by optimizing for photometric consistency which can be formulated as the sum of squared per pixel intensity differences between the camera image and a rendered view.

2.6 Extensions

The basic method of volumetric fusion has been extended in multiple ways which will briefly be presented.
An important component of volumetric fusion is robust tracking of the sensor's pose. If it fails once, it is hard to recover and the reconstruction is corrupted. [ZK15] improves tracking robustness by enforcing the contour of an object to match the contour of its reconstruction during ICP.

All methods so far only use depth or range information for geometry reconstruction. [MKC+17] jointly optimizes shading and geometry. The signed distance field is optimized directly using photo-consistency as optimization costs. They show that they significantly improve the level of detail of the reconstruction at the cost of a higher computational complexity due to the large number of optimization parameters.

Optimizing the SDF makes it possible to incorporate priors in the problem. They help to reduce noise in the reconstruction, as they often enforce smoothness. In [RFBH16], the authors use total variation denoising. In [DPRR13], learned shape priors are used to optimize the SDF. [DSM+17] first fits planes to cubic subvolumes of the voxel grid's TSDF. Then, the TSDF values are adjusted with the distances of the voxels to the extracted planes. Their approach also allows to close large holes in the reconstruction in unobserved areas.

So far, all methods assumed a static scene. However, volumetric reconstruction can also be used to reconstruct deformable objects, as numerous publications have shown. The basic idea was proposed in [NFS15]. The first frame defines a canonical model. For each new sensor frame, a warp field is optimized to align the current sensor frame with the canonical frame. The warp field consists of discrete nodes which hold a transformation with six degrees of freedom, each. Then, the current frame is fused into the canonical model. To reduce the number of parameters, the warp field is approximated using a Gaussian mixture with a small number of components. This approach was used in succeeding work like [GXY+17], and a similar approach using a deformation lattice in [IZN+16]. The described methods cannot handle topological changes, and are limited to a predefined number of objects in the scene due to the formulation of the warp field. These limitations were overcome with the work of [SBCI17] by optimizing a Killing vector field which approximates an isometric motion within each voxel of the voxel grid.
A common method for regularization is to keep the deformation as rigid as possible to prevent overfitting to the data.

A more in-depth comparison of state of the art methods in the field of volumetric fusion can be found in [ZSG+18].

2.7 LiDAR SLAM

3D reconstruction can be seen as a special case of the SLAM problem. While most SLAM algorithms use a simplified representation of the world to reduce memory and computational costs as much as possible, 3D reconstruction tries to capture the whole appearance without loosing too much detail.

In general, there are three classes of SLAM which differ in the kind of features they use. There are methods which use low level features, such as the individual point measurements. They are less descriptive than higher level features but this is usually compensated by their large amount. The other classes of algorithms use higher level features or an intermediate representation which describes more complex geometry. These methods have the advantage that corresponding features are easier to determine which makes the methods more robust.

The first class uses simple frame-to-frame or frame-to-model (the map) registration, such as ICP [BM92, CM91]. Since point-to-point correspondences are hard to determine, the authors of [VLGP16] compute a feature vector for each point which encodes local surface properties, such as curvature. Correspondence search is then performed in feature space instead of assigning nearest neighbors in Euclidean space.

The second class of algorithms uses a higher degree of abstraction to represent a LiDAR scan or the map. In [BS03] the authors propose a method they call normal distribution transform. The world is divided by a regular grid. Within each grid cell a Gaussian distribution is computed describing the LiDAR point distribution within it. This can be seen as a measurement probability. New scans can then be matched such that the probability is maximized. The method has the advantage that it is piecewise differentiable due to the fact that no explicit point or feature correspondences are used.
A problem that often occurs with ICP is the fact that registration errors and noisy scans cause a surface in the world model to have some extend orthogonal to the surface. A scan point which is to be matched against this model will be matched to the nearest point of the surface. However, one would like to match it against the point which represents the mean location of all points of the surface, and therefore represents the expected surface location. This problem was tackled in [Des18] by fitting a least squares surface approximation through the world model. New scans are then matched against this surface.

The last class of algorithms uses geometric primitives as features. [6] uses planes, such as facades of buildings and the poles of traffic lights and traffic signs which drastically reduces map sizes compared to other SLAM approaches. In [ZS14], the authors use plane and corner features achieving one of the highest scores on the SLAM benchmarks they evaluate on. There is also a variety of 3D point features which imitate their 2D counterparts from the image domain by encoding a local neighborhood in a feature vector. One of the most prominent feature descriptors is the fast point feature histogram (FPFH) ([RBB09]) which can be used for bundle adjustment like in image-based SLAM. However, these features are commonly less descriptive than image-based features, and therefore are rarely used for localization.

2.8 Other Reconstruction Methods

Reconstruction is a widely used name for different kinds of representations of the world. The simplest representation is an accumulated point cloud, as it directly gathers raw sensor measurements without any fusion. Any method from section 2.7 can be used for localization, then the points are transformed to the world coordinate system and appended to a single point cloud representing the world. While being simple, the method has numerous disadvantages. Firstly, the model contains a lot of redundant data. In case the sensor platform stands still for a while, the same point measurements are added to the model without providing any new information. This leads to huge map sizes, as modern LiDARs scan millions of points per second. Secondly, the reconstruction contains no topological information, such as whether two points lie on the same surface. This can only be inferred in costly post-processing steps. Thirdly, the point density in the world model varies drastically which makes it hard to work with such models. This also makes the reconstruction highly sensor specific, as a high resolution LiDAR will create a denser model than a low resolution one.
A mesh representation eliminates all of these drawbacks. Numerous algorithms exist for meshing point clouds. An overview is given in [BTS+17] with Poisson reconstruction ([KBH06]) being one of the most prominent methods. An indicator function, which is an implicit function just as used in section 2.1, is determined which is zero outside and one inside the reconstructed object. This is done by making its gradient match the vector field of the surface nor-

mals of the input point cloud in a least squares sense. The problem can be reformulated as a Poisson equation where the name of the method comes from. Other methods, like [CBC+01], use radial basis functions to interpolate distances at any given point in space to the surface and then fuse them in a signed distance field. However, all these methods do not consider sensor characteristics, such as sensor noise or measurement footprints.

Surfels are another method to fuse point measurements. They represent a disc-shaped surface patch and are therefore parameterized by position, normal direction and radius of the disc. To find the surfel into which a new sensor measurement is fused, all surfels are projected into the sensor model keeping only the closest surfel for each pixel. This method was used in various SLAM and reconstruction frameworks, such as [BS18, KLL+13, WLSM+15]. The reconstruction is more dense than pure point clouds due to the modeling of the surface patches. Also, the map size is reduced since multiple points are fused into a single surfel.

There are other volumetric approaches than the one introduces in section 2.1, like occupancy grids, as proposed in [HWB+13]. Just like in section 2.1, the world is discretized by voxels. All voxels along the line of sight of a LiDAR measurement are updated with occupancy probabilities. The resulting reconstruction, however, looks quite coarse since the method lacks the accuracy which is achieved by interpolation used by the methods in section 2.1. Also, their reconstructions are often incomplete further away from the sensor because only voxels along the LiDAR beams are updated.

3 Fundamentals

3.1 Coordinate Systems and Transformations

Each sensor measures relative to its local coordinate system. In order to fuse measurements from multiple sensors, the individual measurements have to be expressed in a common coordinate system. This can be achieved when the relative poses of the individual sensors to each other are known. The process of obtaining these poses is called extrinsic sensor calibration. Further, for 3D reconstruction, all sensor measurements have to be fused into a world model which is defined within its own world coordinate system. This chapter deals with how measurements can be transformed from one coordinate system to another one.

3.1.1 Rotation

There exist multiple possibilities to represent 3D rotations. We discuss four different representations in the following and show individual advantages and disadvantages.

Rotation Matrix

A rotation matrix $\mathbf{R}$ is an orthonormal matrix which has the properties $\mathbf{R}^\top = \mathbf{R}^{-1}$ and $|\mathbf{R}| = 1$. 3×3 matrices with these properties belong to the special orthogonal group $SO(3)$ and describe rotations in three dimensions. Rotation matrices can be thought of in two different ways. They can be thought of as a matrix which rotates a vector around the origin by matrix multiplication, or they can be thought of as a transformation of a vector to a different coordinate system which is rotated in the opposite direction (see subsection 3.1.2). Since

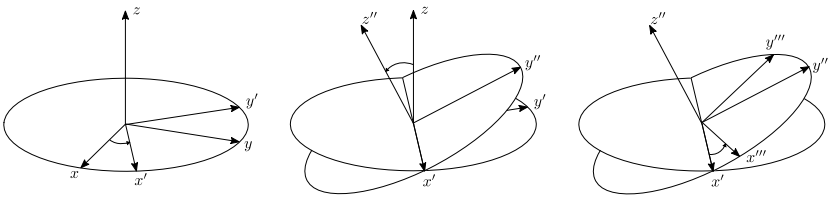

Figure 3.1: Euler angles with rotation order z-x-z applied to to the coordinate axis in sequence from left to right.

rotation matrices have nine parameters, but a rotation in 3D is fully defined by only three parameters, they are not suitable for parameterization of rotations when a minimal representation is needed.

Euler Angles

A rotation can be parameterized by three consecutive rotations, such as $\mathbf{R}(\alpha, \beta, \gamma) = \mathbf{R}_z(\gamma)\mathbf{R}_y(\beta)\mathbf{R}_x(\alpha)$ with rotation matrices

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{3.1}$$

$$\mathbf{R}_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \tag{3.2}$$

$$\mathbf{R}_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.3}$$

The three angles are known as Euler angles. They are particularly easy to understand, as they describe three consecutive rotations around body fixed axis, as shown in Figure 3.1. As the rotation matrices are multiplied from left to a vector, the rightmost rotation matrix is applied first. In Figure 3.1, the rotation order is z-x-z. This can also be thought of as transforming a point to a new coordinate system with new coordinate axes $\mathbf{x}'$ and $\mathbf{y}'$ which are rotated

around $\mathbf{z}$ in the opposite direction. The next rotation then rotates the new point around the new x-Axis $\mathbf{x}'$ and so on.
As matrix multiplication is not commutative, the order of the rotations is of concern and one has to agree on conventions, such as x-y-z. In total, twelve rotation orders and therefore twelve sets of Euler angles exist for the same 3D rotation. Linearization of the matrices yields the following rotation matrix for small angles α, β, γ

$$\tilde{\mathbf{R}}(\alpha, \beta, \gamma) \approx \begin{bmatrix} 1 & -\gamma & \beta \\ \gamma & 1 & -\alpha \\ -\beta & \alpha & 1 \end{bmatrix} . \tag{3.4}$$

One reason why other representations are often favored over Euler angles is the fact that certain sequences of rotations cause gimbal lock. This happens when the axes of the first and the third rotation align. In this case, one degree of freedom is lost and it is not possible to describe the desired rotation by Euler angles.

Rotation Vector

A rotation vector is also called angle-axis representation, and consists of a vector $\alpha\hat{\mathbf{a}}$ whose norm corresponds to the rotation angle α around the rotation axis $\hat{\mathbf{a}}$ with unit length. Due to its minimal representation, it is suitable for rotation parameterizations in optimization problems. From a rotation vector, a rotation matrix can be computed using Rodrigues' formula ([Sze11])

$$\mathbf{R}(\alpha, \hat{\mathbf{a}}) = \mathbf{I}_3 + \sin\alpha \, [\hat{\mathbf{a}}]_\times + (1 - \cos\alpha) \, [\hat{\mathbf{a}}]_\times^2 , \tag{3.5}$$

with $[\cdot]_\times$ denoting the skew-symmetric matrix that multiplied with a vector corresponds to the cross product of its argument and the vector.

Unit Quaternion

Unit quaternions are advantageous when rotations need to be interpolated, as none of the aforementioned representations is capable of doing this. Further, rotating a vector using unit quaternions needs less arithmetic operations than

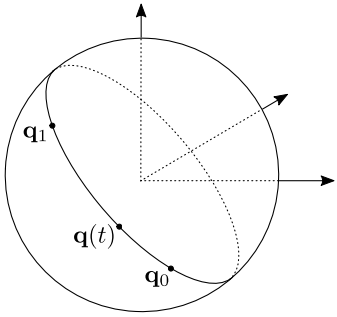

Figure 3.2: Visualization of what SLERP would look like in 3D for $\mathbf{q}(t) = \text{slerp}(t, \mathbf{q}_0, \mathbf{q}_1)$. Actual SLERP interpolates on a 4D sphere.

multiplying a matrix with a vector.
Quaternions are an extension of complex numbers with one real and three imaginary parts i, j and k, and therefore can be written in vector form $\mathbf{q} = [x, y, z, w]^\top$. Notations can differ in the order of the components in the vector. Here the notation of [Sze11] is used with the real part w being the last element. Unit quaternions are quaternions of unit length $\|\mathbf{q}\| = 1$. They reside on the surface of a four-dimensional hypersphere with radius one. Only unit quaternions represent rotations. They are related to rotation vectors by the following equation

$$\mathbf{q} = \left[\sin\frac{\alpha}{2}\hat{\mathbf{a}}^\top, \cos\frac{\alpha}{2}\right]^\top, \tag{3.6}$$

and to rotation matrices via

$$\mathbf{R}(\mathbf{q}) = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy-zw) & 2(xz+yw) \\ 2(xy+zw) & 1-2(x^2+z^2) & 2(yz-xw) \\ 2(xz-yw) & 2(yz+xw) & 1-2(x^2+y^2) \end{bmatrix}. \tag{3.7}$$

Two orientations $\mathbf{q}_0$ and $\mathbf{q}_1$ can be interpolated using spherical linear interpolation (SLERP). While linear interpolation moves along the straight line between both vectors, SLERP moves on the surface of the hypersphere, as shown in

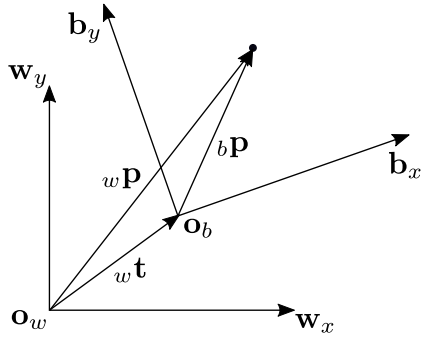

Figure 3.3: Visualization of point coordinates of the same point $\mathbf{p}$ in two coordinate systems B and W.

Figure 3.2. The interpolated orientation for any given control variable $t \in [0, 1]$ can be computed using

$$\text{slerp}(t, \mathbf{q}_0, \mathbf{q}_1) = \frac{\sin((1-t)\theta)}{\sin\theta}\mathbf{q}_0 + \frac{\sin(t\theta)}{\sin\theta}\mathbf{q}_1\,, \tag{3.8}$$

with $\theta = \arccos(\mathbf{q}_0^\top \mathbf{q}_1)$, treating $\mathbf{q}$ like a vector.

3.1.2 Coordinate Systems

A Cartesian coordinate system in $\mathbb{R}^n$ is defined by its origin $\mathbf{o}$ and n pairwise orthogonal basis vectors $\mathbf{e}_1, ..., \mathbf{e}_n$. Distances within each dimension are measured with the same unit of length. We use left subscripts to denote the coordinate system in which a vector is represented. In the following, we will look at transformations between a world coordinate system $W = (\mathbf{o}_w, \mathbf{w}_x, \mathbf{w}_y, \mathbf{w}_z)$ and a body fixed coordinate system $B = (\mathbf{o}_b, \mathbf{b}_x, \mathbf{b}_y, \mathbf{b}_z)$ which, for instance, could be the local sensor coordinate system of a sensor. As shown in Figure 3.3, the same point can be described by ${}_w\mathbf{p}$ and ${}_b\mathbf{p}$. The origin of B is offset from

the origin of W by a translation ${}_w\mathbf{t}$. By projecting ${}_w\mathbf{p}$ onto the basis vectors of B, the point coordinates with respect to B can be determined

$$ {}_b\mathbf{p} = \begin{bmatrix} ({}_w\mathbf{p} - {}_w\mathbf{t})^\top {}_w\mathbf{b}_x \\ ({}_w\mathbf{p} - {}_w\mathbf{t})^\top {}_w\mathbf{b}_y \\ ({}_w\mathbf{p} - {}_w\mathbf{t})^\top {}_w\mathbf{b}_z \end{bmatrix} . \tag{3.9} $$

This can be written more compactly using a matrix vector multiplication

$$ {}_b\mathbf{p} = \underbrace{\begin{bmatrix} {}_w\mathbf{b}_x^\top \\ {}_w\mathbf{b}_y^\top \\ {}_w\mathbf{b}_z^\top \end{bmatrix}}_{{}_b\mathbf{R}_w} ({}_w\mathbf{p} - {}_w\mathbf{t}) . \tag{3.10} $$

This is an Euclidean transformation with rotation matrix ${}_b\mathbf{R}_w$ which stacks the basis vectors of B in world coordinates. Multiplying ${}_b\mathbf{R}_w^{-1}$ from left yields

$$ {}_w\mathbf{p} = {}_b\mathbf{R}_w^{-1} {}_b\mathbf{p} + {}_w\mathbf{t} \tag{3.11} $$

$$ = {}_w\mathbf{R}_b {}_b\mathbf{p} + {}_w\mathbf{t}, \tag{3.12} $$

which is the back transformation from body to world coordinates. Therefore, the rows of ${}_w\mathbf{R}_b$ are the basis vectors of W in body fixed coordinates. A more general derivation of coordinate transformations can be found in [AHK+15] where both coordinate systems are expressed with respect to a third coordinate system.

3.1.3 Transformation Matrix

In many cases, it is convenient to use a 4×4 transformation matrix $\mathbf{T}$ to transform a point. Equation 3.11 can be written as

$$ \begin{bmatrix} {}_w\mathbf{p} \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} {}_w\mathbf{R}_b & {}_w\mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix}}_{{}_w\mathbf{T}_b} \begin{bmatrix} {}_b\mathbf{p} \\ 1 \end{bmatrix} . \tag{3.13} $$

using homogeneous coordinates of the point $\mathbf{p}$ by appending a fourth element to the vector which is one. The transformation matrix which maps a point from body-fixed coordinates to world coordinates will be referred to as the pose of the body. Mapping from world to body coordinates yields

$$\begin{bmatrix} {}_b\mathbf{p} \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} {}_b\mathbf{R}_w & -{}_b\mathbf{R}_w\, {}_w\mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix}}_{{}_b\mathbf{T}_w} \begin{bmatrix} {}_w\mathbf{p} \\ 1 \end{bmatrix} . \tag{3.14}$$

This notation is particularly practical when multiple transformations need to be concatenated. If, for instance, the pose of a vehicle ${}_w\mathbf{T}_v$ and the extrinsic calibration of a sensor ${}_v\mathbf{T}_s$ are known, a sensor measurement ${}_s\mathbf{p}$ can be transformed to world coordinates using

$${}_w\tilde{\mathbf{p}} = {}_w\mathbf{T}_v\, {}_v\mathbf{T}_s\, {}_s\tilde{\mathbf{p}}, \tag{3.15}$$

with the tilde denoting homogeneous coordinates. The same holds when a pose of a vehicle needs to be updated by a delta pose, which can be the result of odometry

$${}_w\mathbf{T}_v^{i+1} = {}_w\mathbf{T}_v^i\, {}_v\Delta\mathbf{T}^i, \tag{3.16}$$

with ${}_v\Delta\mathbf{T}^i$ being the transformation that maps from the vehicle frame at time $i+1$ to the vehicle frame at i, or, in other words, it contains the transformation the vehicle undergoes during the movement seen from the vehicle frame at time i.

In the following sections, we will only make use of subscripts in some cases to avoid confusion and state used coordinate systems in the text otherwise.

3.2 Image Formation

Image formation describes the processes which determine how a two-dimensional image is formed from a three-dimensional object in the world. It includes the physical principles of how light interacts with an object and how it influences the measured light intensity at a specific pixel location. This is also known as radiometry, the science of measuring light.

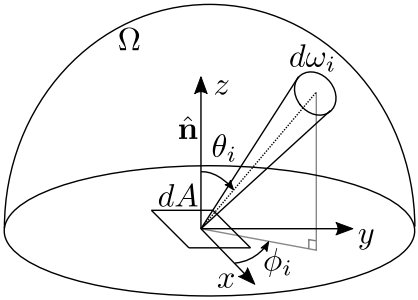

Figure 3.4: Illumination of a surface patch from a single direction.

While computer graphics deals with replicating this process on a computer to generate a realistic look by modeling light and surface properties, computer vision tries to invert the image formation process in order to derive geometric or radiometric properties of a scene from images. The latter is an inherently ill-posed problem due to the many effects which are involved, and the large number of variables which have to be determined. The following radiometric principles are described in [Hor86].

In order to be visible in a scene, an object has to be lit by a light source. The radiometric quantity which describes the energy transported by a stream of photons is the radiant flux Φ which is measured in Watts. The irradiance E is the flux density of incident light on a surface

$$E = \frac{d\Phi}{dA}, \tag{3.17}$$

therefore, it is measured in Wm^{-2}.

As light can come and leave from different directions over the upper hemisphere of a surface patch, a directional measure is convenient. The radiance L is the flux per unit foreshortened area per unit solid angle

$$L = \frac{d^2\Phi}{dA\, cos(\theta)\, d\omega}. \tag{3.18}$$

Foreshortening is the effect of light spreading over a larger area when the angle of incidence θ, which is measured in between the direction of the light and

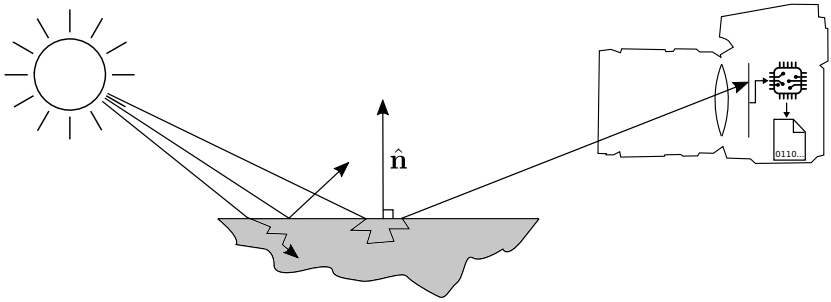

Figure 3.5: Image formation process from light source to digital image.

the surface normal, increases. Therefore, only the flux perpendicular to the surface contributes to radiance. The solid angle ω is a planar angle defined by its area on the unit sphere, just as angles are defined by the arc length on the unit circle. The whole sphere has a solid angle of 4π sr. Radiance is measured in $\mathrm{Wm}^{-2}\mathrm{sr}^{-1}$. The irradiance $dE(\theta_i, \phi_i)$ from a single direction can therefore be computed as

$$dE(\theta_i, \phi_i) = L(\theta_i, \phi_i) \cos(\theta_i)\, d\omega\,. \tag{3.19}$$

ϕ is the azimuth angle measured in the tangent plane of the surface patch relative to an arbitrary reference direction, as shown in Figure 3.4.

When light hits an object, it interacts with its material in different ways (see Figure 3.5). Depending on the atomic structure of the material and surface properties, such as its roughness, different effects occur. Light can be absorbed and converted to heat. It can be transmitted through the object and bent which is referred to as refraction. Light can be refracted multiple times by inhomogeneities of the material in the surface layer of an object until it leafs the object again. This is what causes a matte look. Further, light can be reflected which happens on glossy surfaces which act like a mirror.

Reflection properties of surfaces can vary with lighting direction and point of view. It can be mathematically described by the bidirectional reflectance distribution function (BRDF) which is defined as

$$f_r(\theta_i, \phi_i, \theta_r, \phi_r) = \frac{dL(\theta_r, \phi_r)}{dE(\theta_i, \phi_i)}\,, \tag{3.20}$$

with indices i denoting the incidence direction and r denoting the reflected direction. For any BRDF, these directions are interchangeable due to the fact that light behaves the same in both directions which is known as Helmholtz reciprocity. Sometimes, more complex BRDFs also depend on the wavelength λ of the light.
BRDFs can be very complex and spatially varying which makes recovering BRDFs a hard task. For that reason, simple models are usually used. Most surfaces are matte and the diffuse component of the BRDF is therefore the dominating part which defines its appearance. This means that the surface looks equally bright from all directions. It is what one commonly perceives as the shading or body color of an object. In this case, the reflected radiance L is constant over all directions, and therefore the BRDF is also a constant. This is referred to as Lambertian reflection.

The previous paragraphs described the path of the light up to the camera. So, the next question is how it affects the intensity of a pixel in a digital image.
The light is bundled by an optical system which is composed of several lenses. Their primary purpose is to focus incoming light on the sensor plane to resemble the working principle of a pinhole camera. The brightness of a pixel is a function of its irradiance, also called image irradiance, and exposure time which is affected by the shutter speed. The amount of incoming light is converted to a current by the photo-sensitive elements on the camera chip which make up the pixels.
Next, we want to know what determines image irradiance. Considering a camera with a focal length f and a lens diameter d observing a scene point under an angle α from its principal axis, [Hor86] shows that the image irradiance is

$$E = L\frac{\pi}{4}\left(\frac{d}{f}\right)^2 \cos^4\alpha\,. \tag{3.21}$$

It can be seen that the image irradiance is proportional to scene radiance. Actually, the definition of radiance was deliberately chosen to establish this relation. Also, image irradiance falls off by a factor of $\cos^4\alpha$ from the image center. This effect is called vignetting.
The relation between light falling on a pixel and the final pixel value are quite complex, as there are numerous stages involved in converting one quantity to the other. Relevant effects are sensor characteristics, noise, analog gain, analog to digital conversion and many more. An overview is given in [Sze11].

All these effects can be gathered in the camera response function (CRF) which maps image irradiance to pixel brightness. This function, usually, is nonlinear and can be determined by photometric calibration.

3.3 Range Sensing

Various methods for measuring ranges from mobile platforms exist. In the following, we explain the two most commonly used methods for autonomous driving.

3.3.1 Light Detection and Ranging

Light detection and ranging (LiDAR) is a method to measure ranges through the time of flight of an emitted laser pulse from sensor to an obstacle and back to the sensor ([WHLS16]). The measured distance for a measured time of flight t can be determined as

$$d = \frac{c_0 t}{2}, \tag{3.22}$$

with the speed of light $c_0 \approx 3 \cdot 10^8\,\mathrm{m/s}$. Although solid state LiDAR sensors are already available, the most common type of sensor has a rotating sensor head which means that the whole sensor is rotated or the sensor is fixed looking onto a rotating mirror. The emitted wavelength is usually in the infrared (IR) spectrum with $\lambda = 905\,\mathrm{nm}$. The amount of emitted energy of each pulse is limited by eye safety and cooling. This is the main limitation for maximum range, accuracy and frequency, as all three benefit from high peak power.

In order to reliably filter out only the reflection of the emitted light, multiple filters are employed, as explained in [BZ16]. First, a spectral filter only lets the emitted wavelength pass. As the sun partly emits in the same spectrum, a spatial filter (field diaphragm) filters for the area in the focal plane where the reflection is expected to appear. Since emitter and receiver are not aligned, this can be achieved by a plate with a rectangular hole in front of emitter and receiver, as shown in Figure 3.6. Returned photons are turned into electronic impulses by avalanche diodes which are able to detect single photons. The impulses are counted in small time intervals which creates a discrete histogram

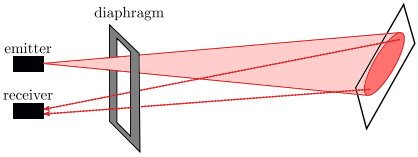

Figure 3.6: Diaphragm filtering out ambient light. The returned light comes from the footprint of the laser beam on a surface, and therefore can have different run times, as shown by the two dashed lines.

of photons over time bins. Sub-centimeter accuracy for the range measurement can still be achieved by fitting a function through the binned data and determining its maximum.
The returned signal is less distinct than the emitted signal due to beam widening, atmospheric losses and reflective properties of the scanned object. Beam widening means that the emitted light beam is rather a cone than a beam. When it hits a surface, the light is reflected from a larger area depending on the distance. If the surface is inclined or has discontinuities, this can cause light from different distances being reflected and the returned signal will spread out. This effect is shown in Figure 3.6. Also transparent objects can cause multiple return signals which some LiDARs are able to detect.

In a rotating LiDAR, a rotary encoder provides the azimuth angle of the sensor head ϕ. Even though, due to the offset of emitter and receiver, the measurement is not along a straight path it is approximately modeled as such. Due to minor manufacturing tolerances and imprecisions the measured raw distance is corrected by a scale and offset. These parameters have to be determined with a suitable intrinsic calibration procedure. For a LiDAR rotating around its z-axis, the Cartesian coordinates of a range measurement d of a single diode can be computed as

$$\mathbf{p} = \mathbf{R}_z(\phi)(\mathbf{o} + (sd + d_0)\hat{\mathbf{v}}), \tag{3.23}$$

with beam origin $\mathbf{o}$, beam direction $\hat{\mathbf{v}}$, range scale s and range offset d_0. To determine these parameters, one has to keep in mind that an offset in depth has the same effect as moving the origin of the beam along the line of sight. Therefore, it is more practical to assume that all diodes lie in the y-z-plane for $\phi = 0\,^\circ$. The determined set of parameters is then no longer the real position of the diodes but one that explains the measurements best.

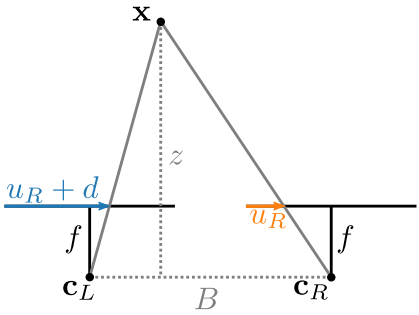

Figure 3.7: Depth computation from disparity d.

3.3.2 Stereo Depth Estimation

Given two camera images and known extrinsic and intrinsic parameters of the sensor setup, the depth of each pixel can be estimated using stereo correspondences. This means that for each pixel in the left camera image the corresponding pixel in the right camera image is determined. Each pixel within an image defines a ray which projects onto an epipolar line in the other image ([Sze11]) if we can assume a pinhole camera model. Therefore, correspondences only have to be searched along epipolar lines. Usually, images are rectified first. During this process they are warped in such a way that pixels from the same row in both images lie on the same epipolar line. For corresponding pixels from left and right camera images, the following holds

$$u_L = u_R + d \tag{3.24}$$

$$v_L = v_R\,, \tag{3.25}$$

with d being the disparity. The z-coordinate in camera coordinates can be computed as

$$z = \frac{fB}{d}\,, \tag{3.26}$$

with the baseline B which is the distance between the projection centers, and focal length f in pixels, as shown in Figure 3.7.
The main challenge of any stereo algorithm is to determine disparities. A common approach is to find the pixel with highest similarity which is often done by comparing a small window around each pixel using sums of squared

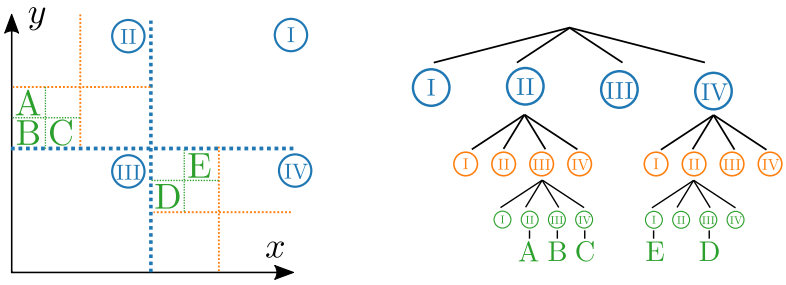

Figure 3.8: Left: Spatial partitioning of a quadtree. Right: Corresponding tree.

differences or normalized cross correlation. Prior information can be incorporated to enforce smoothness in the disparity map. This can be achieved by optimizing the disparity values such that neighboring pixels have similar disparities.
Applying error propagation to Equation 3.26 yields the following uncertainties for depth estimates at depth z which have a disparity standard deviation of σ_d

$$\sigma_z = \frac{z^2}{fB}\sigma_d \,. \tag{3.27}$$

This means that errors grow quadratically with depth.

3.4 Spatial Data Structures

Structuring spatial data in a way which is memory efficient and computationally cheap when querying data is a fundamental task in computer science. In the following, three concepts will be introduced.

3.4.1 Octrees

Octrees are tree-like spatial data structures which hierarchically partition 3D space into eight equally large cubic subvolumes. In the tree, each subvolume is a node and each node has eight child nodes which again partition the subvolume into eight subvolumes. The structure of a quadtree, which essentially has the same behavior in two dimensions, is shown in Figure 3.8.
Octrees have the advantage that empty space is gathered in large empty cells which do not have to be partitioned further. A regular grid, in contrast, holds all cells regardless of being empty or occupied by data. Octrees are widely used in computer graphics to filter only geometry which lies in the visible area. Other applications are physics simulations which use octrees to check for collisions only between objects which lie within a certain range to each other. Multiple variants of octrees exist. The variant discussed here is also called MX quadtree with MX being an abbreviation for matrix since the underlying data can be seen as entries of a sparse matrix ([Sam06]).

3.4.2 KD-Trees

KD-trees are a special case of binary search trees for points in K-dimensional space. This makes them highly relevant for point cloud processing but also for other tasks like feature matching. Binary tree means that each node contains two child nodes. They were first introduced in [Ben75]. There are two common tasks one wants to perform with a KD-tree. The first one is to find the nearest neighbor of a point and the second task is to find all neighbors within a certain radius of a query point. The naïve approach in both cases would be to compute the distances from the query point to all other points, which for N points, is of complexity $O(N)$. KD-trees can speed this up significantly achieving $O(N^{1-\frac{1}{K}} + F)$ with F being the number of points withing the specified search region.

The idea of KD-trees is to consecutively split space along one dimension into two half planes. The dimension which is split is also called discriminator and the value at which the dimension is split is called discriminator value. Many variants of KD-trees exists which mainly differ in the way the discriminator is chosen and how the discriminator value is determined. An overview can be found in [Sam06]. One of the most common ways to build up the tree is

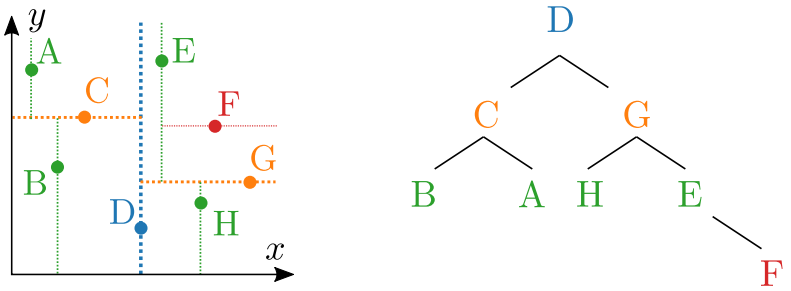

Figure 3.9: Left: Partitioning hyperplanes. The line thickness represents their depth in the tree (the thinner, the deeper). Right: Corresponding tree structure.

to periodically cycle through the dimensions and split at the median of the point coordinates along the chosen dimension, thus cutting the point set in half each time. This results in a balanced tree which means that the depths of two subtrees of a node differ by no more than one. The convention is that a left child of a node contains all points which have a value smaller than the discriminator value along the dimension of the discriminator. Right children have a larger value.
So, for the two-dimensional example depicted in Figure 3.9, the first discriminator is the x-dimension. The space is split at the median x-coordinate of all points which is at point D. Point D becomes the root node with all points having an x-coordinate smaller than D being attached to its left branch and points with a larger x-coordinate attached to its right branch. The process repeats for each of the two subspaces. This time the discriminator is the y-coordinate.

Finding the nearest neighbor of a query point Q is done in the following way: Starting from the root node, the tree is traversed to the left or right, depending on the half plane Q resides in, until a leaf node is reached. The distance from Q to the leaf node is d and the leaf node becomes the current nearest neighbor. In the example shown in Figure 3.10, this is node F with the blue circle visualizing the currently closest distance. The tree is then traversed back up and at each node we check whether the circle intersects with the separating half plane which is visualized as a dashed line. If this happens, as in the case of node G, we have to check if there is any closer point in the other half plane.

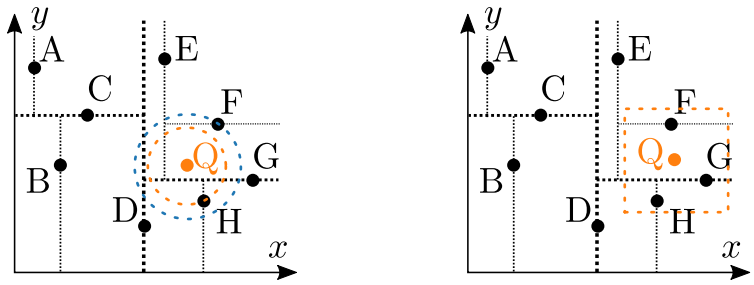

Figure 3.10: Left: Nearest neighbor search for a query point Q. Right: Region search within a rectangular box around Q.

In the example, we find node H as the new nearest neighbor. Now, moving up to node D, we can prune the complete left subtree of D because the orange circle does not intersect with the dashed line D lies on. The nearest neighbor is therefore point H.

A similar approach can be used for a region search, as shown in the right image of Figure 3.10. When searching for the n nearest neighbors within distance d to a query point Q with coordinates (q_x, q_y), a hypercube of length $2d$ can be used as an approximation. In the 2D case, the left bottom point has the coordinates $(q_x - d, q_y - d)$ and the top right point has the coordinates $(q_x + d, q_y + d)$. When the bottom left corner of the query region is a right child of a node, we can prune the left subtree of the node and we only have the points in the right subtree left. This is the case for node D in our example, so we do not have to check if nodes A,B and C are within the query region. The same is true for the top right corner being a left child. In this case, we do not have to check the right subtree.

For high-dimensional spaces, which commonly occur when dealing with descriptors, KD-trees loose their advantage over brute force neighbor search. In these cases, one is usually satisfied finding an approximate nearest neighbor which lies at most a small predefined distance further away than the actual nearest neighbor. A comparison of these techniques can be found in [ML14].

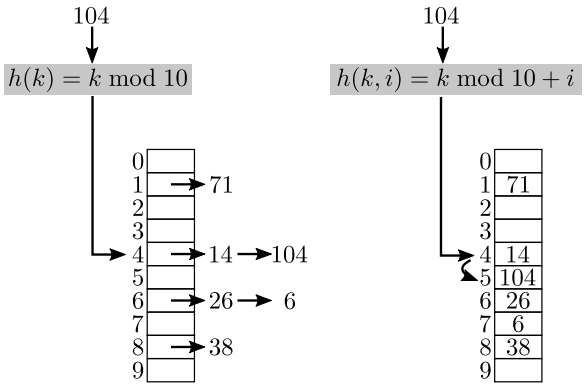

Figure 3.11: Two strategies to resolve hash collisions. Left: Hash table using linked lists. Entries which hash to an occupied entry are appended to the list. Right: Hash table using probing. When a collision occurs, the hash value is recomputed increasing i by one until a free entry is found.

3.4.3 Hash Tables

The previously presented methods use tree-like structures. To find an entry, the tree needs to be traversed starting from its root up to the leaf. Hash tables take a different approach by using a hash function $h(k)$ which directly maps a key k to the location where the associated value resides. The output of $h(k)$ is called the hash value. The hash value can then be used as the index of the corresponding data in a list.

The expected complexity of a lookup according to [CLRS01] is $O(1)$, even though the worst case can be $O(n)$ for n entries. This is because the simple concept has the downside that multiple keys can hash to the same hash value. This is called a collision and needs to be resolved. To keep the number of collisions small, the hash function should distribute the keys evenly but in a deterministic way over the range of available hash values. Still, collisions can not be avoided, especially when the hash table is to be used in a memory efficient way thus having approximately the size of expected elements to store. It is also required to store key and value in the hash table since multiple keys can map to the same entry and therefore the key of the entry has to be compared to the query key to check identity. A common way of handling collisions is to keep a linked list for each entry of the hash table. If a key collides, its

key and value pair is simply appended to the list. This, however, requires to dynamically manage memory which is not ideal for parallelization on a GPU, as allocating memory can become the bottleneck of the algorithm. Also, inserting and removing entries from the list comes with an overhead.
A much simpler approach is open addressing. The data is directly stored in the hash table, and therefore the number of elements which can be stored is fixed. When a collision occurs, the table is probed in a predefined manner until a free entry is found for insertion or the correct element is found for data retrieval. The simplest form of probing is linear probing for which the hash value is increased by one until the correct entry is found. An example of a simple hash function is shown in Figure 3.11. A problem which comes up with linear probing is that an empty entry proceeded by many occupied entries will more likely be picked while probing which causes clustering. Large amount of data will be concentrated at certain locations in the table which slows down the method. Better methods are quadratic probing using

$$h(k,i) = (h_1(k) + c_1 i + c_2 i^2) \bmod m, \tag{3.28}$$

with probe number i for a hash table of length m, or double hashing by combining two hash functions

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m. \tag{3.29}$$

The probe number starts from zero and is incremented each time a collision occurs.

3.5 Parallel Algorithms

Compared to CPUs, modern graphics cards offer orders of magnitude more threads which can run in parallel, and while clock speeds of CPUs have come to a stop the number of cores on GPUs is ever increasing. In order to leverage this computational power, standard algorithms, such as sorting a list of values, have to be adapted to achieve any significant speed up compared to sequential processing.

In the following, the boxes in the diagrams show elements of an array which is holding n values $x_0, \ldots, x_{n-1}$. The output is the last row with elements

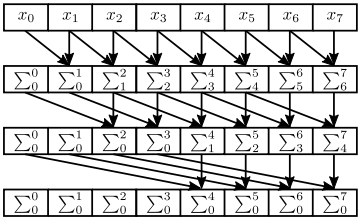

Figure 3.12: Computation steps for an inclusive scan for an array with eight elements. After three steps the array holds the cumulative sum of its input data.

$y_0, \ldots, y_{n-1}$.
Each row represents one step of the algorithm which means that all operations within one row have to finish before the next row can be executed. Arrows show the data flow from one step to the next.

When comparing parallel algorithms, two measures are important to assess their performance. The first is step complexity which describes how the number of computational steps increases with the complexity or size of the problem. This assumes that we have an infinite number of threads available.
The second measure is work complexity. It describes the number of computational operations needed in total. In many cases, parallel algorithms accept a higher work complexity to reduce step complexity compared to a serial algorithm due to the large number of available threads.

3.5.1 Prefix-Sum

Parallel prefix-sum, which is also called scan, is an important primitive in parallel computing which is used for a variety of tasks. The method was first introduced in [HS86]. A comprehensive implementation can be found in [Ngu07].
If the input contains an array of values $x_0, \ldots, x_{n-1}$ then the result of an inclusive scan for element i of the output array is the partial sum $y_i = \sum_{j=0}^{i} x_j$, whereas an exclusive scan returns $y_0 = 0$ and continues with $y_i = \sum_{j=0}^{i-1} x_j$. Figure 3.12 shows how the algorithm works. In the first step, each thread adds the two entries from its own position in the array and one to the left. Then two to the

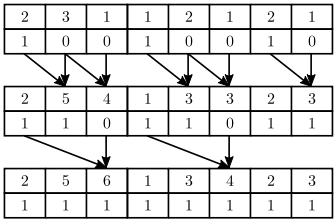

Figure 3.13: Segmented scan applied to an array with three segments. In each step the top row holds values and the bottom row holds the segment head flags.

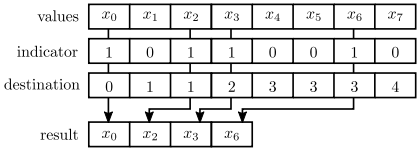

Figure 3.14: Compaction operation on an array. In the final array only the elements with an indicator of one are present.

left in the second and four to the left in the third step.
This algorithm has a step complexity of $O(\log_2 n)$ and a work complexity of $O(n \log_2 n)$ which is higher than the work complexity of a sequential scan which is $O(n)$. Therefore, the algorithm is not work-efficient. For cases in which a more work-efficient algorithm is required, the alternative of Blelloch [Ble89] can be used which has twice the step complexity $O(2 \log_2 n)$ but a work complexity of $O(n)$.

In some situations one has to perform multiple scans at once. In this case, we can add a second array holding flags indicating segment heads by a value of one and zero everywhere else. This is shown in Figure 3.13. The same algorithm is run as before, but this time, we only add elements if the destination flag is zero. The flags are combined by the logical or operator meaning that the result is one if either of the two inputs is one. In the example, it can be seen that after the final step, each segment holds the partial sums of only its own elements.

3.5.2 Stream Compaction

If we want to copy certain elements of an array that fulfill a binary predicate $p(x)$, to a second contiguous array this is called stream compaction. First, the result of the predicate, which is either zero or one, is stored in a second array which we call indicator. A predicate can be a simple comparison like, for example, $p(x) = x > 3$ if we want to keep all elements which are greater than three. An exclusive scan, as described in subsection 3.5.1, is then run on the indicator array generating the destination address as an output, as shown in Figure 3.14. The length of the compacted array can be computed by summing up the elements of the indicator array. In a last pass, each thread copies the element to its destination address if the predicate is one.

A similar approach can be used for allocation of memory in cases where each thread needs to write multiple values to an output array. Each thread simply writes the number of elements it wants to write into the indicator array.

3.5.3 Sort

Sort algorithms sort an arbitrary sequence of values such that the values in the output sequence monotonically increase. One of the best performing parallel algorithms is quite simple in the way it works. It is called radix sort and the method sorts values using the lowest to highest significant digit in each step to determine the new position of a value.
As an example, we sort values from zero to seven, as shown in Figure 3.15. The binary representation of the values is used for sorting. We start by splitting the sequence into two partial sequences depending on the value of the lowest significant bit. All values ending with a zero go into the first half, all values ending with a one go into the second half of the new sequence. The order within each partial sequence remains unchanged. In each iteration, we repeat the process using the next higher significant digit of the binary number. Commonly used radix sort algorithms use hexadecimal number representations and split up a sequence into 16 partial sequences in each step. The process of selecting and moving values in each step can be performed by compacting the sequence, as shown in subsection 3.5.2.

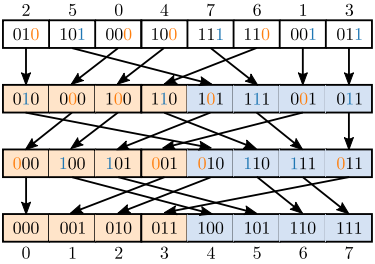

Figure 3.15: Radix sort using binary representation of the values zero to seven. In each iteration (row) the value of the highlighted digit determines the partial sequence it is moved to (light shaded boxes).

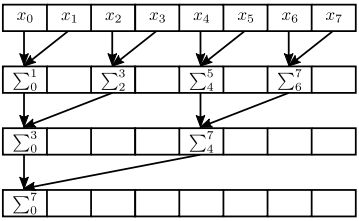

Figure 3.16: Summing up all elements of an array using reduction.

3.5.4 Reduction

Reduction algorithms reduce a collection of values to a single value by applying a binary operator multiple times. A binary operator has two inputs and one output. It is required to be associative for a parallel implementation due to a potentially unknown order in which the operator is applied to the values. An example for adding up values is shown in Figure 3.16. In each step, a thread applies the operator to the output of two threads from the previous step. It can be seen that the step complexity is $O(log_2 n)$ and the work complexity is $O(n)$.

3.6 Method of Least Squares

A least squares problem is of the following form

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x}), \tag{3.30}$$

with an objective function

$$f(\mathbf{x}) = \frac{1}{2}\sum_{j=1}^{m} r_j(\mathbf{x})^2 \tag{3.31}$$

$$= \frac{1}{2}\|\mathbf{r}(\mathbf{x})\|^2 \tag{3.32}$$

$$= \frac{1}{2}\mathbf{r}(\mathbf{x})^\top \mathbf{r}(\mathbf{x}). \tag{3.33}$$

Thus, we try to find the parameter vector $\mathbf{x} \in \mathbb{R}^n$ which minimizes the squared Euclidean norm of a residual vector $\mathbf{r}(\mathbf{x}) = [r_1(\mathbf{x}), ..., r_m(\mathbf{x})]^\top \in \mathbb{R}^m$. In the following, we will drop the argument $\mathbf{x}$ of functions in most cases for the sake of readability. We assume to have more residuals than parameters $m > n$.
Problems of this form occur in a wide range of applications like fitting a model which best describes a series of measurements. It turns out that in this case the least squares solution is the optimal solution if the data is measured with zero mean Gaussian noise, as shown in [NW06]. Many optimization problems are purposefully formulated as least squares problems due to the efficient solvers that exist.
The second order Taylor expansion of the objective function around a point $\mathbf{x}_k$ is

$$f(\mathbf{x}_k + \mathbf{s}) \approx f(\mathbf{x}_k) + \mathbf{s}^\top \nabla f(\mathbf{x}_k) + \frac{1}{2}\mathbf{s}^\top \nabla^2 f(\mathbf{x}_k)\mathbf{s}. \tag{3.34}$$

with Hessian $\nabla^2 f(\mathbf{x}_k)$. A local minimum $\mathbf{x}^*$ of $f(\mathbf{x})$ must fulfill the necessary condition

$$\nabla f(\mathbf{x}^*) = \mathbf{0}. \tag{3.35}$$

The sufficient condition additionally demands for

$$\nabla^2 f(\mathbf{x}^*) > 0, \tag{3.36}$$

stating that the Hessian is positive semidefinite. Using the Jacobian of the residual vector

$$\mathbf{J} = \frac{\partial \mathbf{r}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial r_1}{\partial x_1} & \cdots & \frac{\partial r_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial r_m}{\partial x_1} & \cdots & \frac{\partial r_m}{\partial x_n} \end{bmatrix}, \tag{3.37}$$

the gradient $\mathbf{g} = \nabla f$ and the Hessian $\mathbf{H} = \nabla^2 f$ of the objective function can be written in the following way

$$\mathbf{g} = \sum_{j=1}^{m} r_j \nabla r_j = \mathbf{J}^\top \mathbf{r} \tag{3.38}$$

$$\mathbf{H} = \sum_{j=1}^{m} \nabla r_j \nabla r_j{}^\top + \sum_{j=1}^{m} r_j \nabla^2 r_j = \mathbf{J}^\top \mathbf{J} + \sum_{j=1}^{m} r_j \nabla^2 r_j \,. \tag{3.39}$$

3.6.1 Linear Least Squares

A closed form solution exists for cases in which the residual vector is linear in $\mathbf{x}$ which have the form

$$\mathbf{r} = \mathbf{J}\mathbf{x} + \mathbf{r}_0 \,. \tag{3.40}$$

Computing Equation 3.38 and Equation 3.39 yields

$$\mathbf{g} = \mathbf{J}^\top (\mathbf{J}\mathbf{x} + \mathbf{r}_0) \tag{3.41}$$

$$\mathbf{H} = \mathbf{J}^\top \mathbf{J}, \tag{3.42}$$

with vanishing second order terms of the residual in the Hessian. Applying Equation 3.35 to Equation 3.41 results in the optimal solution

$$\mathbf{x}^* = -\left(\mathbf{J}^\top \mathbf{J}\right)^{-1} \mathbf{J}^\top \mathbf{r}_0 \,. \tag{3.43}$$

3.6.2 Nonlinear Least Squares

Often, we deal with optimization problems in which the residuals are nonlinear. In these cases, we try to iteratively approach a local minimum by starting

from an initial solution $\mathbf{x}_0$ and then take steps $\mathbf{s}$ to improve the solution in each iteration $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k$. In general, there are two approaches which are line search and trust-region methods.
Line search methods first determine the direction $\hat{\mathbf{s}}$ of the step and then minimize the objective function along this direction by finding the best step size α. Trust-region methods approximate the objective function locally and find the minimum of the approximation within the trust-region. The trust-region is the region around the current solution in which we assume our approximation to be close enough to the actual objective function. If the new point does not result in the expected decrease of the objective function, the trust-region is shrunk.

Gauss-Newton Method

The Gauss-Newton method is a modified Newton's method. Newton's method determines $\mathbf{s}$ in a way that minimizes Equation 3.34 by setting its derivative with respect to $\mathbf{s}$ to zero which yields the search direction

$$\mathbf{s}_k = -\mathbf{H}_k^{-1}\mathbf{g}_k \,. \tag{3.44}$$

In each iteration of the algorithm, Hessian and gradient are evaluated at the current solution $\mathbf{x}_k$ which we indicate by the subscript k. Computing the Hessian is quite costly due to the last term in Equation 3.39. Therefore, the Gauss-Newton method approximates the Hessian by neglecting the last term, just using the Jacobians. Practically, this shows just as good convergence as Newton's method since the last term in Equation 3.39 is very small close to the solution. The advantage of Gauss-Newton is that only the Jacobian has to be determined for both, gradient and Hessian, which makes the method computationally efficient. The search direction is given by

$$\mathbf{s}_k = -\left(\mathbf{J}_k^\top \mathbf{J}_k\right)^{-1} \mathbf{J}_k^\top \mathbf{r}_k \,. \tag{3.45}$$

Comparing this to Equation 3.43 shows that Gauss-Newton effectively minimizes $\|\mathbf{J}_k\mathbf{s}_k + \mathbf{r}_k\|^2$ in each iteration by linearizing $\mathbf{r}$ around $\mathbf{x}_k$ ([Bjö96]).
Since a large number of residuals leads to large matrices, it is often more efficient to incrementally compute the approximation of the Hessian and the

gradient which can be done independently for each residual using the sum notation from Equation 3.38 and Equation 3.39

$$\mathbf{J}^\top \mathbf{r} = \sum_{j=1}^{m} r_j \nabla r_j \tag{3.46}$$

$$\mathbf{J}^\top \mathbf{J} = \sum_{j=1}^{m} \nabla r_j \nabla r_j^\top . \tag{3.47}$$

Gauss-Newton is a line search method but in contrast to other methods like gradient descent it has a natural step size of $\alpha = 1$.

Levenberg-Marquardt Method

Gauss-Newton encounters problems when the Jacobian is rank deficient and $\mathbf{s}$ becomes not well defined. For this reason, the Levenberg-Marquardt method is preferred which shows superior convergence in most cases. It belongs to the family of trust-region methods. It solves the same minimization problem as Gauss-Newton but additionally imposes a constraint on the step size which is

$$\|\mathbf{D}_k \mathbf{s}_k\| \leq \Delta_k \,, \tag{3.48}$$

with a diagonal scaling matrix $\mathbf{D}$ to weight components of $\mathbf{s}$ against each other to obtain approximately equal scales and the radius of the trust-region Δ. Equation 3.48 can be added as a quadratic constraint with a Lagrange multiplier λ. The solution of the constraint problem is

$$\mathbf{s}_k = -\left(\mathbf{J}_k^\top \mathbf{J}_k + \lambda_k \mathbf{D}_k^\top \mathbf{D}_k\right)^{-1} \mathbf{J}_k^\top \mathbf{r}_k \,. \tag{3.49}$$

$\mathbf{D}^\top \mathbf{D}$ is typically chosen as the diagonal of the approximation of the Hessian. As shown in [NW06], the new problem is equivalent to the original optimization problem by extending the Jacobian matrix and the residual vector in the following way

$$\mathbf{s}_k^* = \arg\min_{\mathbf{s}} \frac{1}{2} \left\| \begin{bmatrix} \mathbf{J}_k \\ \sqrt{\lambda_k} \mathbf{D}_k \end{bmatrix} \mathbf{s}_k + \begin{bmatrix} \mathbf{r}_k \\ \mathbf{0} \end{bmatrix} \right\|^2 . \tag{3.50}$$

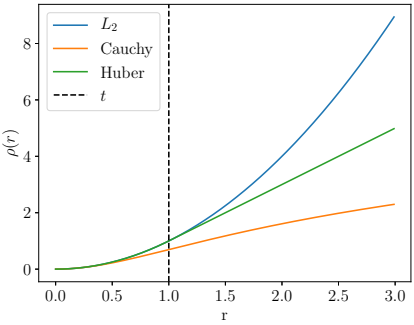

Figure 3.17: Overview of different loss functions. The threshold for all methods is set to $t = 1$.

For $\lambda \to \infty$, Levenberg-Marquardt becomes the gradient descent method, whereas for $\lambda \to 0$, it becomes the Gauss-Newton method. This is used to adapt the solver in each step by checking the ratio of actual decrease of the objective function and expected decrease

$$\rho = \frac{\|\mathbf{r}(\mathbf{x}_k + \mathbf{s}_k)\|^2 - \|\mathbf{r}(\mathbf{x}_k)\|^2}{\|\mathbf{J}_k\mathbf{s}_k + \mathbf{r}(\mathbf{x}_k)\|^2 - \|\mathbf{r}(\mathbf{x}_k)\|^2} . \tag{3.51}$$

If ρ is close to one, the local approximation proved to be valid, the step is accepted and λ is decreased which increases the trust-region. If ρ is too small, the trust-region is shrunk by increasing λ and the step is rejected.

3.6.3 Robustification

Least squares problems are inherently sensitive to outliers which often appear when data is wrongly associated or when measurements are corrupted. In these cases, a least squares solution will be extremely drawn towards a solution which explains the erroneous data due to its large contribution to the objective function. The effect is commonly alleviated by using robust loss functions $\rho(r)$. The method comes from robust statistics where estimators using these losses

are known under the name M-estimators. Using robust losses, the objective function becomes

$$f(\mathbf{x}) = \frac{1}{2} \sum_{j=1}^{m} \rho(r_j(\mathbf{x})) \,. \tag{3.52}$$

Ordinary least squares can be seen as a method using a quadratic loss $\rho(r) = r^2$, also called L_2-loss.
Different loss functions exist. Some of the best know ones are Huber's loss and Cauchy loss

$$\rho_H(r) = \begin{cases} r^2 & \text{for } |r| \leq t \\ 2t|r| - t^2 & \text{for } |r| > t \end{cases} \tag{3.53}$$

$$\rho_C(r) = \log\left(1 + \frac{r^2}{t^2}\right) . \tag{3.54}$$

They resemble the quadratic loss function close to zero but grow more slowly above some threshold t which downweights larger residuals. Figure 3.17 shows the effect of different losses on the residual.
Since Equation 3.52 is no longer a least squares problem, it has to be reformulated in order to solve it with the methods we introduced so far. This can be done by the method called iteratively reweighted least squares (IRLS). A weighted least squares problem minimizes the objective function

$$f(\mathbf{x}) = \frac{1}{2} \sum_{j=1}^{m} w_j r_j^2 \tag{3.55}$$

$$= \frac{1}{2} \|\mathbf{W}^{\frac{1}{2}} \mathbf{r}\|^2 \tag{3.56}$$

$$= \frac{1}{2} \mathbf{r}^\top \mathbf{W} \mathbf{r}, \tag{3.57}$$

with a diagonal weight matrix $\mathbf{W} = \text{diag}(w_1, \ldots w_m)$. The Gauss-Newton step of this problem is

$$\mathbf{s}_k = -\left(\mathbf{J}_k^\top \mathbf{W}_k \mathbf{J}_k\right)^{-1} \mathbf{J}_k^\top \mathbf{W}_k \mathbf{r}_k \,. \tag{3.58}$$

We look at the necessary condition Equation 3.35 of Equation 3.52 for an optimum which yields

$$\sum_{j=1}^{m} \frac{\partial \rho_j}{\partial r_j} \frac{\partial r_j}{\partial x_i} = 0 \qquad \text{for } i = 1, ..., n\,. \tag{3.59}$$

Using the influence function $\psi = \rho'$ and weights ([Gro03])

$$w(r) = \begin{cases} \frac{\psi(r)}{r} & \text{for } r \neq 0 \\ 1 & \text{for } r = 0 \end{cases}, \tag{3.60}$$

these conditions can be rewritten as

$$\sum_{j=1}^{m} w(r_j) r_j \frac{\partial r_j}{\partial x_i} = 0 \qquad \text{for } i = 1, ..., n \tag{3.61}$$

which is the necessary condition of the weighted least squares problem Equation 3.55 if we assume w to be constant. Using IRLS, $w(r)$ is recomputed in each iteration using the current residual and then treated as a constant.

3.7 Iterative Closest Point

Iterative closest point (ICP) is one of the fundamental algorithms in point cloud processing. It is used for registration of two point clouds. Many variants emerged over the years. An in-depth overview of existing methods is provided in [PCS15], however there are two variants which are most commonly used and which we will briefly describe in the following sections.
The transformation $\mathbf{T}$ is to be determined such that the source points $\mathbf{s} \in \mathcal{P}_s$ transformed by $\mathbf{T}$ match the destination points $\mathbf{d} \in \mathcal{P}_d$ in some optimal way. Costs are defined between corresponding point pairs $C = \{(\mathbf{s}, \mathbf{d})_k\}$. However, it is not guaranteed that corresponding points actually describe the same point on an object. So, after transforming $\mathcal{P}_s$ with $\mathbf{T}$, new correspondences can be found which are more accurate, as the two point clouds match better. The process is repeated multiple times until the algorithm converges. This is shown in Figure 3.18 over two iterations using nearest point correspondences and point-to-point distance costs. Many different approaches for correspondence search

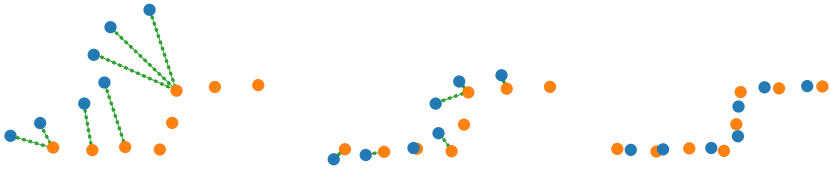

Figure 3.18: A source point cloud (blue) is aligned to a destination point cloud (orange) using nearest points as correspondences (dashed green lines). After two iterations (from left to right) both surfaces match. In each iteration, the distances between corresponding points are minimized.

exist. Most use search trees like KD-trees (see subsection 3.4.2) to find pairs of closest points between both point clouds. Correspondences are accepted or rejected depending on various criteria, such as distance or similarity of local surface properties, like normal orientation or curvature, to filter out wrong matches. Due to simple correspondence search and little distinctiveness of individual points, ICP is prone to getting stuck in local minima. It is therefore crucial to start from an initial solution which is close to the actual solution.

3.7.1 Point-to-Point ICP

The oldest and most basic variant of ICP is point-to-point ICP which determines $\mathbf{T}$ such that the sum of squared distances between corresponding points is minimal, as described in [BM92]. The problem can be formulated as

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}} \sum_{(\mathbf{s},\mathbf{d})\in C} \|\mathbf{R}\mathbf{s} + \mathbf{t} - \mathbf{d}\|^2, \tag{3.62}$$

with parameter vector

$$\boldsymbol{\xi} = \left[\boldsymbol{\alpha}^\top, \mathbf{t}^\top\right]^\top = \left[\alpha, \beta, \gamma, t_x, t_y, t_z\right]^\top \tag{3.63}$$

concatenating the three Euler angles of the rotation matrix $\mathbf{R}$ and the components of the translation vector $\mathbf{t}$. Since Equation 3.62 is a nonlinear problem, we have to solve it by means of nonlinear optimization. How to linearize point-to-plane ICP will be shown in the following.

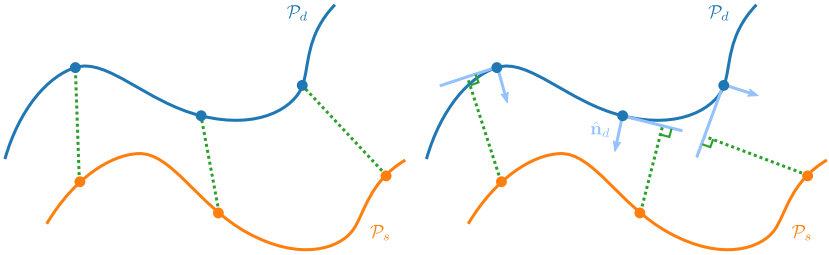

Figure 3.19: Costs of point-to-point (left) and point-to-plane ICP (right). The sum of squared lengths of the dashed green lines is minimized.

3.7.2 Point-to-Plane ICP

Another variant of ICP is point-to-plane ICP ([CM91]) which minimizes point distances to the tangent plane of the corresponding point, as shown in Figure 3.19. It shows better convergence than point-to-point ICP and is therefore the most used method. Imagine two point clouds sampled from a plane. There is an infinite number of solutions for registrating both point clouds however point-to-point ICP will get stuck in a local minimum whereas point-to-plane ICP allows to slide both point clouds along the surface without a change in costs, as one would expect. The problem formulation becomes

$$\boldsymbol{\xi}^* = \arg\min_{\boldsymbol{\xi}} \sum_{(\mathbf{s},\mathbf{d})\in C} (\underbrace{(\mathbf{R}\mathbf{s}+\mathbf{t}-\mathbf{d})^\top \hat{\mathbf{n}}_d}_{r})^2, \tag{3.64}$$

where $\hat{\mathbf{n}}_d$ is the surface normal of the destination point $\mathbf{d}$. By linearization of the rotation matrix, as described in [Low04a], the problem can be turned into a linear least squares problem. In order to do so, we use Equation 3.4 as an approximation for the rotation matrix. The residual becomes

$$r \approx (\mathbf{s} + \boldsymbol{\alpha} \times \mathbf{s} + \mathbf{t} - \mathbf{d})^\top \hat{\mathbf{n}}_d, \tag{3.65}$$

which can be further simplified using the properties of the triple product $(\mathbf{a}\times\mathbf{b})^\top\mathbf{c} = (\mathbf{b}\times\mathbf{c})^\top\mathbf{a}$ ([AHK+15])

$$r \approx (\mathbf{s}\times\hat{\mathbf{n}}_d)^\top \boldsymbol{\alpha} + \hat{\mathbf{n}}_d^\top\mathbf{t} + (\mathbf{s}-\mathbf{d})^\top \hat{\mathbf{n}}_d, \tag{3.66}$$

which can also be written in the following form using parameter vector $\boldsymbol{\xi}$

$$r \approx \underbrace{\begin{bmatrix} (\mathbf{s} \times \hat{\mathbf{n}}_d)^\top & \hat{\mathbf{n}}_d^\top \end{bmatrix} \begin{bmatrix} \boldsymbol{\alpha} \\ \mathbf{t} \end{bmatrix}}_{\boldsymbol{\xi}} + (\mathbf{s} - \mathbf{d})^\top \hat{\mathbf{n}}_d \,. \tag{3.67}$$

Again, the optimization can be performed using the methods described in section 3.6. The transformation $\mathbf{T}$ can be obtained by converting $\boldsymbol{\alpha}$ from $\boldsymbol{\xi}$ to a rotation matrix, as shown in section 3.1.1. This is advisable because the linearized rotation matrix $\tilde{\mathbf{R}}$ is not orthonormal and updating a pose with it will lead to an erroneous result over time.
In many cases, ICP is used for sensor tracking. The destination cloud is then derived from the world model, given in world coordinates, and the source points are the current scan. Matching source to destination, as described above, the resulting transformation is the movement in world coordinates. The new sensor pose ${}_w\mathbf{T}_v^{i+1}$ can be obtained by updating the pose from the previous time step, as described in subsection 3.1.2

$${}_w\mathbf{T}_v^{i+1} = \mathbf{T}\, {}_w\mathbf{T}_v^{i} \,. \tag{3.68}$$

4 Volumetric Reconstruction

We described different reconstruction methods in chapter 2, however, the volumetric approach from [CL96] became the preferred method for many applications for multiple reasons. The core idea is simple and easy to parallelize. The reconstructions are dense due to the way the sensor measurements are applied to all visible voxels. Further, the method allows to consider measurement uncertainties by applying different weighting schemes, and finally sub-voxel accuracy can be achieved by interpolation.

Figure 4.1 shows the modules of the reconstruction pipeline and how the modules are distributed over CPU and GPU. The following sections deal with each of the modules in the sequence they are executed when new measurements are processed.
The working principle in a nutshell is as follows: Range data in the form of a point cloud is provided by either LiDAR, stereo cameras or any other range sensing device. After raw data preprocessing, the sensor pose is determined using a variant of ICP. As we are using voxel hashing, entries for all new voxels have to be allocated in the hash table. This goes hand in hand with voxel streaming which keeps the currently visible area within the GPU's memory and constantly exchanges voxels between GPU and CPU, where a complete world model is kept in an octree. The integration step fuses the new sensor

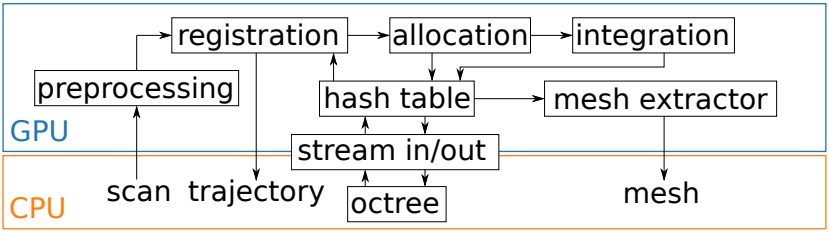

Figure 4.1: Overview of the reconstruction pipeline.

measurements into the world model using the determined sensor pose. Since we want to obtain an explicit representation of the reconstruction, a triangle mesh is extracted from the TSDF which is then simplified in a final post-processing step.
The underlying theory and the individual steps from above are described in detail in the following section.

4.1 Signed Distance Functions

Explicit surface representations, such as polygon meshes, are hard to manipulate when topological changes occur, like splitting or merging surfaces. In both cases, one has to make sure that the result is still a valid representation of the surface. Self intersections or non manifold vertices and non manifold edges can occur which are unwanted in most cases.
A representation that naturally handles these challenges is a signed distance function $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}$ for the three-dimensional case which holds the signed Euclidean distance to the nearest surface for each point $\mathbf{x}$. The sign of ϕ determines on which side of a surface $\mathbf{x}$ is located. In this work, a negative sign indicates points on the back side of a surface and a positive sign indicates points in front of a surface. A property of an SDF is that its gradient has a norm of one

$$\|\nabla\phi(\mathbf{x})\| = 1\,. \tag{4.1}$$

The surface $\mathcal{S}$ is implicitly encoded in ϕ as its zero iso level

$$\mathcal{S} = \{\mathbf{x} \mid \phi(\mathbf{x}) = 0\}\,. \tag{4.2}$$

Its normals are the gradient of the SDF

$$\hat{\mathbf{n}}(\mathbf{x}) = \nabla\phi(\mathbf{x})\,. \tag{4.3}$$

Since for most practical applications no analytical representation of ϕ can be found, an approximation is used by discretizing space with a regular grid of cubic volumes with edge length l_v, in which a linear basis is used to approximate ϕ. Practically, that means that ϕ is determined for the center location $\mathbf{b}$ of each voxel and a linear basis is used to approximate ϕ everywhere else. We use $F(\mathbf{x})$ to denote this approximation. Trilinear interpolation of the SDF provides

signed distances at all locations. For practical reasons, the signed distances are only stored within a narrow band around the surface within the truncation distance d_t. In this case, we refer to the SDF as a truncated signed distance function (TSDF).
The two parameters l_v and d_t should be as small as possible to preserve as much details as possible during reconstruction. However, this is not possible due to computational limitations in the case of l_v, as a small voxel size results in a large amount of voxels which need to be processed and stored. The same holds for d_t which is important for the robustness of our algorithm. Choosing d_t too small will create holes and other unwanted artifacts from noisy data in the reconstruction while choosing it too large will smooth away too much detail. A good tradeoff has to be found for a specific sensor setup and environment.

In our implementation, we store $F(\mathbf{b})$ in a hash table. The hash (see subsection 3.4.3) is computed using a spatial hash function as used in [NZIS13] which takes integer voxel coordinates

$$\mathbf{b}_{\text{int}} = \left\lfloor \frac{\mathbf{b}}{l_v} \right\rfloor \tag{4.4}$$

as input. $\lfloor \cdot \rfloor$ denotes the operator that rounds a value down. We use quadratic probing to resolve hash collisions and set a maximum number of tries to prevent successive collisions from blocking our algorithm.

4.2 Sensor Models

The sensor model maps an object point in sensor coordinates $\mathbf{x}$ to a 2D point on the image plane $\mathbf{u} = [u, v]^\top$ via a projection function $\boldsymbol{\pi} : \mathbb{R}^3 \to \mathbb{R}^2$. In the following, it is mainly needed for fast assignment of sensor measurements to an object point. Therefore, we project all points of $\mathcal{P}$, which can be an unordered point cloud, into the respective sensor model to obtain a point map $\mathbf{P} : \mathbb{R}^2 \to \mathbb{R}^3$ which takes a pixel location $\mathbf{u}$ as its argument and returns points $\mathbf{p} \in \mathcal{P}$ for which $\lfloor \boldsymbol{\pi}(\mathbf{p}) \rfloor = \lfloor \mathbf{u} \rfloor$ holds. In our implementation, we keep at most a single point for each pixel.
For sensors like RGB-D cameras or stereo setups, $\mathbf{P}$ is already the output of the sensor. For LiDAR, we create $\mathbf{P}$ by projecting all points onto the image plane using the sensor model.

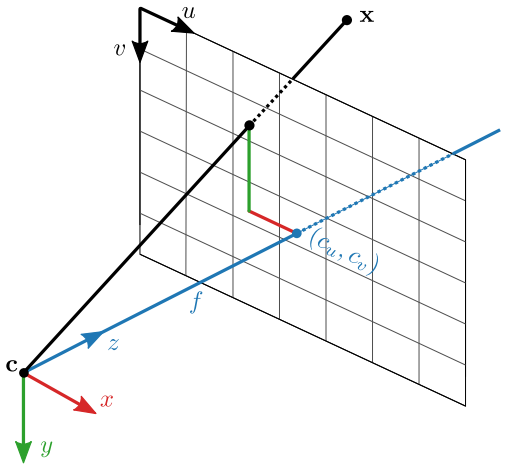

Figure 4.2: Pinhole model used for depth from stereo.

Two sensor models that cover most range sensors are described in the next sections.

4.2.1 Pinhole Model

The pinhole camera model is the simplest camera model. It assumes that incoming light passes an infinitely small opening (the aperture) before it hits the image plane ([Sze11]). It is shown in Figure 4.2. For clarity, the image plane is drawn in front of the focal point **c**. In a real camera, it is behind the focal point. The image needs then to be flipped since it perceives an upside down version of the observed scene.
The projection function can be derived from the theorem of intersecting lines with the help of Figure 4.2

$$\boldsymbol{\pi}(\mathbf{x}) = \begin{bmatrix} f\frac{x}{z} + c_u \\ f\frac{y}{z} + c_v \end{bmatrix} . \tag{4.5}$$

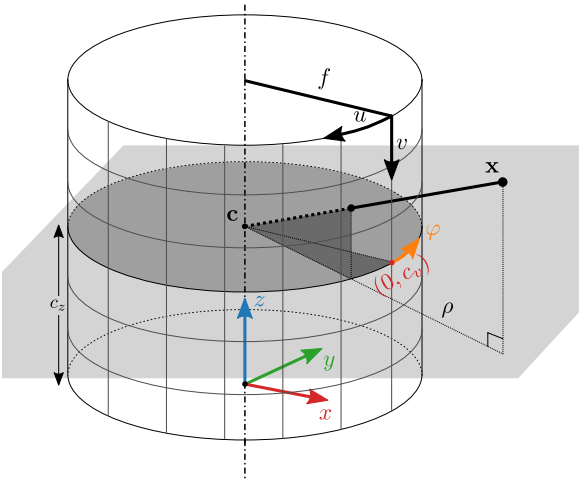

Figure 4.3: Cylindrical projection model used for a rotating LiDAR.

The focal length f is the distance between focal point and image plane. For Equation 4.5 to hold, f and (c_u, c_v) have to be expressed in the same unit of measure which usually is in pixels.

4.2.2 Cylinder Model

For sensors such as rotating LiDARs, a cylinder projection model can be assumed. The center of projection $\mathbf{c}$ lies on the rotation axis and is offset by a distance c_z from the LiDAR coordinate system in which the points are measured. A cylindrical image plane with n_u horizontal pixels wraps around

the rotation axis with a radius of f. The projection function can therefore be derived from Figure 4.3 to be

$$\boldsymbol{\pi}(\mathbf{x}) = \begin{bmatrix} n_u(1 - \frac{\varphi}{2\pi}) \\ -f\frac{(z-c_z)}{\rho} + c_v \end{bmatrix} \tag{4.6}$$

$$\rho = \sqrt{x^2 + y^2} \tag{4.7}$$

$$\varphi = \operatorname{arctan2}(y, x), \quad \varphi \in [0, 2\pi), \tag{4.8}$$

with ρ being the distance of $\mathbf{c}$ to $\mathbf{x}$ projected onto the xy-plane and the azimuth angle φ.

4.3 Preprocessing

For later steps, the raw sensor measurements have to be preprocessed. This step contains compensating for the sensor movement which is necessary for rotating LiDARs, as they are rolling shutter sensors. The other processing steps are bilateral filtering of depth data and normal estimation. Ranges from LiDAR have a low level of noise compared to depth from RGB-D cameras or stereo. Therefore, we initially smooth their depth information by means of an edge preserving filter to preserve depth discontinuities. Since normals are very sensitive to noise of the underlying points, normal estimation benefits from this smoothing as well.

4.3.1 Motion Compensation

For rotating LiDAR sensors, a scan covering 360 ° contains measurements from different points in time. All point measurements are taken with respect to the sensor's local coordinate system at the corresponding time. As briefly mentioned in section 1.2, we use the term rolling shutter which originally comes from cameras, to refer to this phenomenon. By performing motion compensation, we transform all points to one common point in time, as if all measurements were recorded at the same time. This however, is only valid for static points, as we would have to also compensate the motion of points

on dynamic objects which is only possible if the movement of the object is known. We assume the scene to be mainly static in the following.
Further, we assume that during a full revolution of the sensor head, its motion can be approximated by a linear motion and a rotation around a constant axis with constant angular velocity. If the sensor pose at the start of the revolution is ${}_w\mathbf{T}_{s(t_0)}$ and at the end of the revolution it is ${}_w\mathbf{T}_{s(t_0+\Delta T)}$ then the delta pose as seen from ${}_w\mathbf{T}_{s(t_0)}$ is

$$\Delta\mathbf{T} = {}_w\mathbf{T}^{-1}_{s(t_0)}\,{}_w\mathbf{T}_{s(t_0+\Delta T)} \tag{4.9}$$

according to Equation 3.16. With control variable $\lambda = \frac{t-t_0}{\Delta T}$, translational and rotational part of ${}_{s(t_0)}\mathbf{T}(t)_{s(t)}$ can be computed using linear interpolation and SLERP from section 3.1.1

$$\mathbf{t}(t) = \lambda\Delta\mathbf{t} \tag{4.10}$$

$$\mathbf{R}(t) = \text{slerp}(\lambda, \mathbf{I}, \Delta\mathbf{R})\,. \tag{4.11}$$

Point measurements ${}_{s(t)}\mathbf{p}$ in sensor coordinates at t can be transformed to the sensor frame at t_0 using

$${}_{s(t_0)}\mathbf{p} = {}_{s(t_0)}\mathbf{T}(t)_{s(t)}\;{}_{s(t)}\mathbf{p}\,. \tag{4.12}$$

When we run the reconstruction in odometry mode, the movement of the sensor is not known at the time we want to perform motion compensation. Therefore, the last delta pose is used as an approximation of the current delta pose. For the final mapping described in chapter 5, delta poses can be directly computed from the trajectory of a preceding iteration.

4.3.2 Bilateral Filtering

Bilateral filtering, as described in [Sze11], is a method to smooth data while preserving discontinuities. $\mathcal{N}(\mathbf{u}) = \{\mathbf{v} \mid \|\mathbf{u} - \mathbf{v}\| < l\}$ denotes the pixels in a local neighborhood of pixel $\mathbf{u}$, defined by a maximum distance l in the image plane. A pixel $\mathbf{v}$ of the neighborhood contributes to the smoothed result in two ways: It is weighted by its distance to $\mathbf{u}$ in the image plane and by the similarity

of the underlying data. Here, we look at measured ranges $d_p(\mathbf{u}) = \|\mathbf{P}(\mathbf{u})\|$. This can be achieved by employing two Gaussian kernels

$$d_p(\mathbf{u}) = \frac{1}{W} \sum_{\mathbf{v} \in \mathcal{N}(\mathbf{u})} d_p(\mathbf{v}) \underbrace{\exp\left(-\frac{\|\mathbf{u} - \mathbf{v}\|^2}{2\sigma_l^2} - \frac{(d_p(\mathbf{u}) - d_p(\mathbf{v}))^2}{2\sigma_d^2} \right)}_{w(\mathbf{u},\mathbf{v})}, \tag{4.13}$$

and $W = \sum_{\mathbf{v} \in \mathcal{N}(\mathbf{u})} w(\mathbf{u}, \mathbf{v})$.

4.3.3 Normal Estimation

In this step, normal vectors for each point in $\mathcal{P}$ are computed. Each thread determines the normal of a single point by computing sample mean and covariance of the point distribution in a local neighborhood in the image plane

$$\bar{\mathbf{p}} = \frac{1}{|\mathcal{N}(\mathbf{u})|} \sum_{\mathbf{v} \in \mathcal{N}(\mathbf{u})} \mathbf{P}(\mathbf{v}) \tag{4.14}$$

$$\mathbf{C}_p = \frac{1}{|\mathcal{N}(\mathbf{u})| - 1} \sum_{\mathbf{v} \in \mathcal{N}(\mathbf{u})} (\mathbf{P}(\mathbf{v}) - \bar{\mathbf{p}})(\mathbf{P}(\mathbf{v}) - \bar{\mathbf{p}})^\top . \tag{4.15}$$

The neighborhood is adjusted depending on $d_p(\mathbf{u}) = \|\mathbf{P}(\mathbf{u})\|$ in a way that the points cover approximately the same area in Euclidean space. We perform principal component analysis (PCA), as shown in [Bis06], by computing the eigenvalues of $\mathbf{C}_p$. In the case of 3×3 covariance matrices, this can be efficiently done with the method from [Smi61]. For points describing a local surface patch, the covariance matrix will have one eigenvalue which is significantly smaller than the other two eigenvalues. Its corresponding eigenvector points in the direction of smallest variance which is in direction of the surface normal

$$\hat{\mathbf{n}}_p = \frac{\mathbf{e}_1}{\|\mathbf{e}_1\|}, \tag{4.16}$$

given the eigenvalues $\lambda_1 < \lambda_2 < \lambda_3$ and corresponding eigenvectors $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$.

4.4 Pose Estimation

In order to fuse range measurements into the voxel grid, the precise pose of the range sensor has to be determined first. We use variants of point-to-plane ICP, as discussed in subsection 3.7.2, to align the current scan with the world model. Instead of aligning two point clouds, we directly align the point cloud to the TSDF. However, if the sensor moves further than the truncation distance in between consecutive scans, points will initially fall into voxels containing no TSDF, thus ICP will not be possible. Therefore, a two stage approach is used which first derives a point cloud from the TSDF by raycasting. This method is known as projective correspondence search and it is well suited for larger sensor movement. After this initial alignment, a second registration is carried out, directly minimizing TSDF values for each scan point by using the TSDF's gradient information.

4.4.1 Projective ICP

The points from the current point cloud will be denoted source points, as they will be aligned to the target which is the map. For all source points from $\mathbf{P}$, a corresponding destination point is derived from the voxel grid by raycasting, as described in [NIH+11]. A ray is shot from the projection center through the center of each pixel. The ray is parameterized by

$$\mathbf{x}(t) = \mathbf{c} + t\hat{\mathbf{v}}, \tag{4.17}$$

with control variable t and ray direction $\hat{\mathbf{v}} = \boldsymbol{\pi}^{-1}(\mathbf{u})/\|\boldsymbol{\pi}^{-1}(\mathbf{u})\|$ of a pixel $\mathbf{u}$. Then, TSDF values are checked along the ray by increasing t by steps of $\Delta t = d_t$. This is sufficient to guarantee that consecutive sample points are within the truncation distance when intersecting the surface. When the TSDF switches sign from positive to negative, the surface has been intersected in between both sample points $\mathbf{x}(t^*)$ and $\mathbf{x}(t^* + \Delta t)$. The zero level can be determined by linear interpolation, giving us the destination point

$$\mathbf{d} = \mathbf{c} + \left(t^* + \Delta t \frac{F(\mathbf{x}(t^*))}{F(\mathbf{x}(t^*)) - F(\mathbf{x}(t^* + \Delta t))} \right) \hat{\mathbf{v}} \,. \tag{4.18}$$

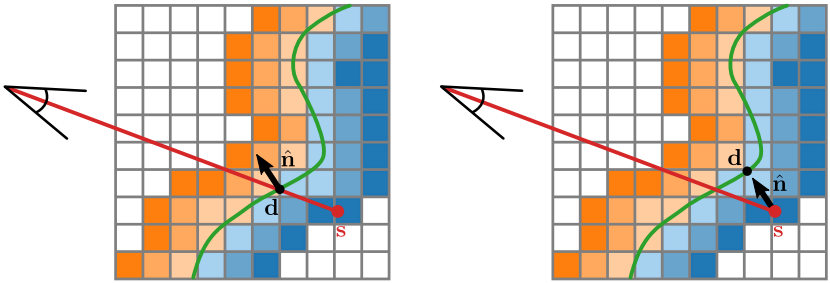

Figure 4.4: Two possible ways to determine the destination point **d** for a given source point **s**. Orange represents positive and blue represents negative values of the TSDF. On the left, **d** is determined by finding the zero crossing along the red line. On the right, **d** is determined by moving along the normal in **s** by a distance equal to the signed distance value in **s**.

It is shown on the left of Figure 4.4. In case the sign switches from negative to positive, a surface was intersected from its back side which is not possible, and therefore no destination point is generated.
Surface normals for each destination point are computed from the gradient of the TSDF using central finite differences

$$\nabla F(\mathbf{x}) \approx \frac{1}{2l_v} \begin{bmatrix} F(x + l_v) - F(x - l_v) \\ F(y + l_v) - F(y - l_v) \\ F(z + l_v) - F(z - l_v) \end{bmatrix} . \tag{4.19}$$

Having determined source points and corresponding destination points with normals, we can now solve Equation 3.64. In parallel, each thread on the GPU computes one summand of Equation 3.46 and Equation 3.47. To obtain the full gradient and approximate Hessian, we use the reduction technique from subsection 3.5.4 for each element of the vector and matrix. For robustification against outlying data, we employ Cauchy loss, as proposed by [BE14], and use IRLS, as described in subsection 3.6.3, to solve for the pose parameters.

4.4.2 Point-to-TSDF ICP

Now, we directly align the point cloud with the zero level of the TSDF. The destination point is determined by moving in the opposite direction of the gradient by a distance equal to the TSDF value $F(\mathbf{s})$. This, in theory, results in a point lying on the zero level by definition of an SDF. Since F does not hold proper signed distances due to its approximate nature, it will lead to convergence problems during ICP. To correct F, we divide it by the magnitude of its gradient which results in much better convergence. The destination point is therefore determined using

$$\mathbf{d} = \mathbf{s} - \frac{F(\mathbf{s})}{\|\nabla F(\mathbf{s})\|} \frac{\nabla F(\mathbf{s})}{\|\nabla F(\mathbf{s})\|}, \tag{4.20}$$

with the first fraction being the corrected signed distance and the second fraction being the normal direction. The destination point is shown on the right of Figure 4.4.
We use the same optimization and robustification techniques, as described in subsection 4.4.1, to solve the ICP.

4.5 Allocation

Since we are using voxel hashing, as introduced in [NZIS13], we first have to allocate entries in the hash table for all voxels we want to use. In order to do so, all relevant voxels for the current sensor measurement are determined. These are all voxels that are visible, i.e. they project into the sensor model, and lie within the truncation distance to the sensor measurement.
Rays are shot through the center of all pixels and we traverse them from a distance of one truncation distance in front of the corresponding measurement point to one truncation distance behind it. An equidistant sampling of the rays, as used in subsection 4.4.1, is straightforward but inefficient and might lead to missed voxels when the ray cuts a voxel close to one of its corners. We use the method from [AW87] which moves along a line and updates the running variable such that we move from voxel border to voxel border in each iteration. This is simply done by checking for the nearest intersection in x,y and z-direction with the voxel grid. Then, we move up to this point and repeat

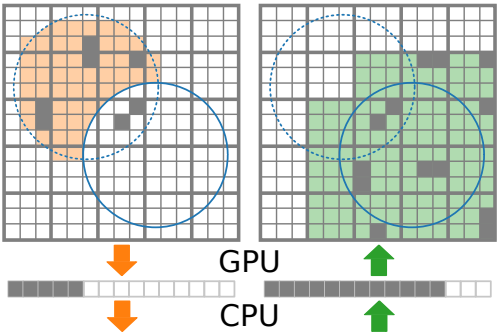

Figure 4.5: Visualization of the streaming process. The visible area of a sensor moves from dashed to the solid circle. Gray boxes represent allocated voxels. The bold lines represent voxel blocks. All voxels within the orange area are streamed out while all voxels within the green area are streamed in. Streaming is done by first moving data to a buffer and then from the buffer to its destination.

the process.
When the resolution of the sensor model is low, voxels are small or we want to allocate at far distances, it might still occur that voxels are missed because they are not intersected by any of the rays. This can be handled by temporarily increasing the resolution of the sensor model for allocation thus shooting more rays. An alternative which guarantees that all voxels are allocated, is to project all voxels within sensor range into the sensor model and check their distance to the measurement. This, however, comes at an increased runtime.
Since allocation also runs on the GPU in parallel, we have to prevent race conditions in which two threads try to allocate the same entry in the hash table. To prevent this from happening, each thread tries to lock a mutex for the corresponding hash entry first and unlocks it when it finishes allocation. Locking and unlocking are done by atomic operations which are guaranteed to be executed by only one thread at a time.

4.6 Streaming

Like in the work of Nießner et al. ([NZIS13]), we stream voxels from host to device and the other way around, right after the current sensor pose has been

determined and voxels have been allocated. This is a two stage process which consists of streaming voxels out and streaming voxels in.
First, voxels are streamed out which means that we move them from the GPU's memory to the host memory. For each entry in the hash table, we check whether the entry holds data. If the entry holds a voxel, we check whether its center is outside the sensor range which we assume to be a sphere around the sensor with a certain radius. For voxels outside the sphere, their integer position $\mathbf{b}_{\text{int}}$ is written to an array which is then compacted (see subsection 3.5.2) to assign each of these voxels a location in a buffer. The voxel position is a unique identifier for each voxel. In a succeeding pass, the voxel data is transfered to the buffer and the buffer is downloaded onto the host.
The storage method on the host is different then on the device. An octree (see subsection 3.4.1) holds voxel blocks of $5\times5\times5$ voxels. Since the block position is implicitly contained in the octree leaf, it has not to be stored compared to the hash table.
The second stage streams voxels in from host to device. For all voxel blocks within the sphere, we apply the same procedure. A voxel block is within the sphere if one of its corners has a distance to the sensor smaller than the sphere's radius. The voxel data, containing only weight and TSDF value, is written to the buffer and the voxel position is added by computing it from the leaf position in the octree. The buffer is uploaded to the host where each entry is processed by one thread. A hash entry has to be allocated, as described in section 4.5, then the data is written into the hash table.

4.7 Integration

The integration step integrates the sensor measurement into the world model by fusing it with all previous measurements. This is achieved by updating the TSDF values of all voxels using a weighted average. Due to some characteristics of LiDAR, we have to perform point splatting before integrating the measurements.

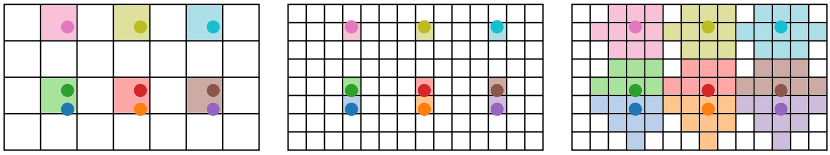

Figure 4.6: Point measurements (colored circles) are assigned to pixels of three sensor models. The last sensor model uses point splatting which assigns a single measurement to multiple pixels (boxes with light shading).

4.7.1 Point Splatting

The cylinder model from subsection 4.2.1 is a simple approximation for a real rotating LiDAR. In the general case, their laser beams have no common center of intersection and their orientations do not exactly obey the model either. As a results, when generating $\mathbf{P}$ from section 4.2, the projection of multiple points onto the sensor plane looks like one of the first two cases in Figure 4.6. In the first case, the pixel size is chosen too coarse which results in multiple points falling into the same pixel. Since we only keep one of them, this results in loss of information.
In the second case, the pixel size is decreased such that each point falls into a single pixel. This, however, will lead to holes in the reconstruction, as we only update voxels which project into pixels holding a measurement.
For this reason, we use point splatting which means that we choose a small pixel size and a single point measurement contributes to several pixels around its projection within the splat radius (Figure 4.6 right image). Practically, this is achieved by launching one thread for each pixel in the image plane. Each empty pixel will then look for the closest pixel within the splat radius which holds a measurement and copies the measurement to its own pixel.
Point splatting is only used for the integration step of the reconstruction algorithm. For registration, only the original points are used.

4.7.2 Fusion

New range measurements can be fused into the TSDF by a recursive update scheme. First, all voxels from the hash table have to be associated to measurements by projecting their center positions ${}_w\tilde{\mathbf{b}}$ in homogeneous world

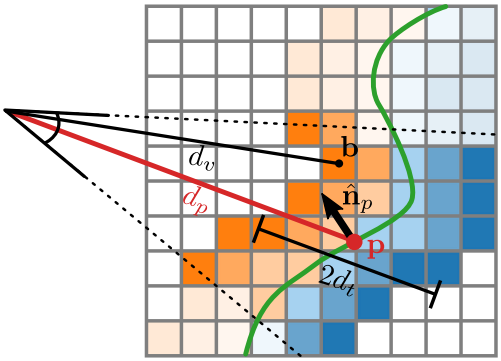

Figure 4.7: Updating the signed distance field for a single pixel of a sensor. All voxels are projected into the sensor model. Those voxels that fall into a pixel and lie within the truncation distance (highlighted cells) are updated with the corresponding measurement **p**.

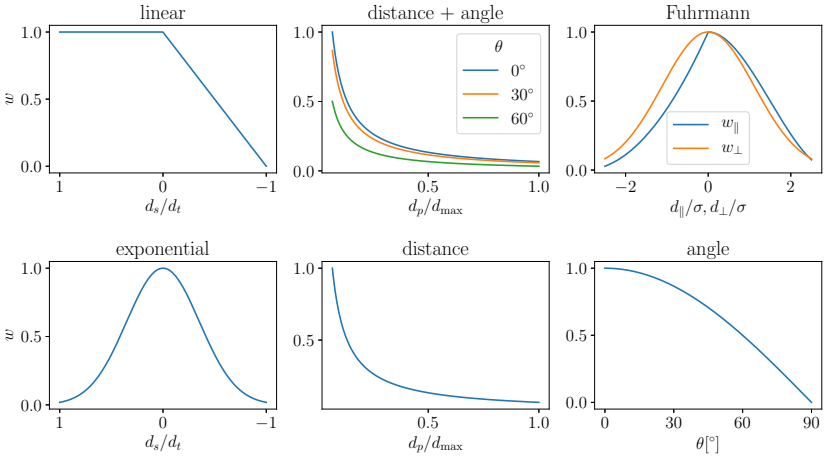

Figure 4.8: Overview of different weighting schemes used in literature. The weights in the left column depend on the signed distance estimate d_s. The weights in the center column depend on the distance of the measurement. Fuhrmann weights depend on the parallel and orthogonal distance with respect to the surface normal. θ denotes the angle between line of sight and surface normal.

coordinates into the image plane by transforming them with the inverse sensor pose $\mathbf{T}$ to the sensor frame and then using the sensor models from section 4.2 for projection. The approximate signed distance d_s of a voxel to the measurement can then be computed as

$$ {}_s\tilde{\mathbf{b}} = \mathbf{T}^{-1}{}_w\tilde{\mathbf{b}} \tag{4.21} $$

$$ d_s = d_p - d_v = \|\mathbf{P}(\boldsymbol{\pi}({}_s\mathbf{b}))\| - \|{}_s\mathbf{b}\| \,. \tag{4.22} $$

Each voxel holds a TSDF value F and a weight W. If a voxel can be associated to a range measurement, its running average is updated using

$$ F(\mathbf{b})_i = \frac{W(\mathbf{b})_{i-1}F(\mathbf{b})_{i-1} + w(\mathbf{b})d_s(\mathbf{b})}{W(\mathbf{b})_{i-1} + w(\mathbf{b})} \tag{4.23} $$

$$ W(\mathbf{b})_i = W(\mathbf{b})_{i-1} + w(\mathbf{b}) \,. \tag{4.24} $$

with weight $w(\mathbf{b})$. Here, for brevity, we only list $\mathbf{b}$ as an argument and to make clear that w is a function. As we will show in the following, there are different ways to compute w.

Different weighting schemes have been used in literature. Some depend on the signed distance d_s, others depend on the distance of the sensor measurement d_p, and a third group defines signed distances relative to a local coordinate system.
In [KDSX15], the authors use constant weights, e.g. $w = 1$. Curless and Levoy ([CL96]) use constant weights which fall off with a constant slope behind the surface to account for higher uncertainty in areas which cannot be observed

$$ w(d_s) = \begin{cases} 1 & \text{for } d_s \geq 0 \\ 1 + \frac{d_s}{d_t} & \text{for } d_s < 0 \end{cases} . \tag{4.25} $$

Newcombe ([NIH+11]) weights down far observations, as well as observations at grazing angles

$$ w(\mathbf{p}, \hat{\mathbf{n}}_p) = -\frac{\hat{\mathbf{p}}^\top \hat{\mathbf{n}}_p}{d_p} \,. \tag{4.26} $$

The sign comes from our definition of the normal pointing towards the observer. We further introduce two similar weighting schemes. The first one only weights distances by setting the numerator of Equation 4.26 to one, and the second

one only weights angles by setting the denominator of Equation 4.26 to one. Fuhrmann ([FG14]) computes the signed distance of a voxel relative to a local coordinate system which has its origin in the measured point and which is aligned with the surface normal. Two distances are computed

$$d_{\parallel} = (\mathbf{b} - \mathbf{p})^{\top} \hat{\mathbf{n}}_p \tag{4.27}$$

$$d_{\perp} = \|(\mathbf{b} - \mathbf{p}) - d_{\parallel} \hat{\mathbf{n}}_p\|, \tag{4.28}$$

with $d_{\parallel}$ being the projection of the signed distance onto the normal and $d_{\perp}$ being the distance orthogonal to the normal. The final weight is the product of the two weights $w(d_{\parallel}, d_{\perp}) = w_{\parallel}(d_{\parallel}) w_{\perp}(d_{\perp})$ with

$$w_{\parallel}(d_{\parallel}) = \begin{cases} \frac{1}{9}\frac{d_{\parallel}^2}{\sigma^2} + \frac{2}{3}\frac{d_{\parallel}}{\sigma} + 1 & \text{for } d_{\parallel} \geq 0 \\ \frac{2}{27}\frac{d_{\parallel}^3}{\sigma^3} - \frac{1}{3}\frac{d_{\parallel}^2}{\sigma^2} + 1 & \text{for } d_{\parallel} < 0 \end{cases} \tag{4.29}$$

$$w_{\perp}(d_{\perp}) = \frac{2}{27}\frac{d_{\parallel}^3}{\sigma^3} - \frac{1}{3}\frac{d_{\parallel}^2}{\sigma^2} + 1, \tag{4.30}$$

and σ controlling the width of the curve. The shape of the weighting function as well as all the other weights are depicted in Figure 4.8. The weighting scheme from Fuhrmann is specifically designed to account for different footprint sizes of a sensor measurement. The exponential-like falloff for voxels far from the measurement ensures that measurements with a large footprint do not smooth out details from measurements with a smaller footprint.
As a final weighting scheme, we use an exponential weight in the form of

$$w(d_s) = \exp\left(-\frac{d_s^2}{2\sigma^2}\right). \tag{4.31}$$

A common practice is to update all voxels from sensor up to a distance of d_t behind the measured point $\mathbf{p}$. This removes erroneous surfaces from observable free space which can be caused by dynamic objects or corrupted measurements.

weighting scheme	accuracy (pose given) [m]	accuracy (pose estimated) [m]	translation error [%]	rotation error 10^{-3} [°/m]
linear	**0.09**	0.16	0.19	7.79
distance + angle	0.12	0.14	0.19	8.71
Fuhrmann	0.24	0.24	0.16	**6.88**
exponential	0.11	**0.12**	**0.13**	8.65
distance	0.11	0.17	0.33	8.94
angle	0.17	0.20	0.35	9.45

Table 4.1: Reconstruction quality and pose estimation errors using different weighting schemes. The best performing weighting scheme is bold for each metric.

4.7.3 Experiment

In order to evaluate the weighting schemes, we conduct the following experiment: We use synthetic data of a rotating LiDAR with resolution 64×2000 which moves along a smooth trajectory through a model of a city at a speed of 30 km/h. The trajectory is approximately 160 m long. Sensor range noise with $\sigma_r = 0.015$ m is used and rolling shutter is disabled.
We run the reconstruction twice for each weighting scheme. One time using ground truth sensor poses and a second time during which the sensor pose is estimated using ICP. The reconstruction is compared to the ground truth mesh and the accuracy is computed. Further, for the second reconstruction, we compute translation and rotation errors of subtrajectories with a length of 10 m. The error metrics for both, meshes and trajectories, will be introduced in section 7.2 and section 7.1 in detail.
The results are shown in Table 4.1. Linear weights create the most accurate reconstruction if the poses are given while exponential weights have the overall best performance with the best reconstruction if poses are determined by ICP. Further, the estimated trajectory is the most accurate one with the lowest translation error. The weights by Fuhrmann result in the lowest rotation error. However, differences are only marginal.
It can also be seen that all weighting schemes which rely on normal information perform worse than the others. This might be due to unreliable normal

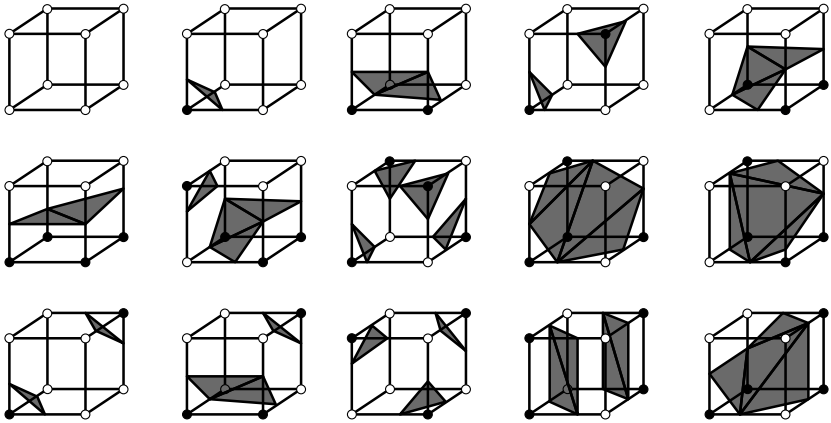

Figure 4.9: All possible constellations that can appear. Filled and empty circles in the corners of the cubes indicate opposite signs of signed distance values.

estimation from sparse LiDAR data.
As expected, errors from pose estimation also affect the reconstruction quality, as all accuracies are better using ground truth poses.

4.8 Mesh Extraction

As an explicit representation of the reconstructed surface $\mathcal{S}$, we use triangle meshes. A triangle mesh is composed of planar triangular faces, sometimes also referred to as facets, which are defined by their corner points, also called vertices $\mathbf{v}$. Triangle meshes can also be seen as graphs $\mathcal{M} = \{\mathcal{V}, \mathcal{E}\}$ with the set of vertices $\mathcal{V}$ and the set of edges $\mathcal{E}$ connecting the vertices. In the following, we will make use of face indices f and vertex indices v to clearly distinguish between per-face and per-vertex quantities.

The implicit representation using an SDF shows great advantages for registration and for fusing measurements, however we want an explicit representation for visualization and texturing. Marching cubes ([LC87]) is an algorithm for iso surface extraction from a scalar field which was first invented for medical purposes, such as for visualization of MRI scans. It splits up the task into

smaller subtasks which can be solved independently. This makes it particularly suitable for parallel implementations on GPUs. The idea is to look at individual blocks of $2 \times 2 \times 2$ voxels and extract the surface within this group. We visualize such blocks in Figure 4.9 by cubes with each corner point representing the center point of one of the voxels. Since we want to extract the zero level surface from the SDF, we have to look at edges which connect corners with different signs which are visualized by white and black dots. The exact position $\mathbf{x}$ of a certain iso value t along an edge, connecting two corner points $\mathbf{b}_0$ and $\mathbf{b}_1$, can be determined by linear interpolation using

$$\mathbf{x}(t) = \mathbf{b}_0 + (t - F(\mathbf{b}_0))\frac{\mathbf{b}_1 - \mathbf{b}_0}{F(\mathbf{b}_1) - F(\mathbf{b}_0)} \,. \tag{4.32}$$

In the case of surface extraction one sets $t = 0$.
There are eight corners which each can either be positive or negative, so there are in total $2^8 = 256$ possible constellations. However, each of them can be seen as one of 15 basic constellations shown in Figure 4.9 due to cube symmetry. So, the first step of the algorithm determines which of the 256 cases is present. This can be achieved by turning the sign of each corner into one bit of an eight bit cube index. This cube index is used to retrieve a twelve bit number from a precomputed lookup table, called edge table. Each bit of this twelve bit number indicates whether one of the twelve edges of the cube intersects the iso surface. For each intersected edge, the vertex position of the generated facet is computed according to Equation 4.32. Then, in a last step, facets have to be formed from the vertices. Another table, called triangle table, contains as many triplets as triangles have to be formed. Each triplet contains the three edges the facet intersects. So, putting the three corresponding vertices from interpolation in order results in the correct facet.
Since each thread processes one cube, the number of triangles each thread generates can be different. For this reason, each thread checks how many vertices it will generate and allocates the needed amount of memory using the allocation method from subsection 3.5.2.

4.9 Post-Processing

Since section 4.8 generates triangles independent from each other, each triangle contains three unique vertices even when they share the same edge. To remove the redundant vertices, we employ nearest neighbor search using a KD-tree, as described in subsection 3.4.2, to merge vertices which are closer than some predefined small distance ϵ.

A second side effect from marching cubes is that the meshing is not ideal regarding the number and shape of the faces. Very small or thin triangles increase the mesh size in memory and slow down processing times, but contribute little to the overall geometry. These kinds of triangles can be seen in the top image of Figure 4.10. There are even faces that are completely unnecessary, such as multiple faces lying on the same plane. To get rid of these faces, we use a mesh simplification algorithm which tries to reduce the number of faces while it preserves the geometry as best as possible. The method that we use is called QSlim ([GH97]). It is an iterative edge contraction technique which in each iteration selects an edge and contracts both its vertices $\mathbf{v}_a$ and $\mathbf{v}_b$ to a new vertex $\mathbf{v}$. The order in which edges are contracted is by increasing contraction costs. For QSlim, costs are defined as the sum of squared distances of $\mathbf{v}$ to each surface $\mathbf{v}_a$ and $\mathbf{v}_b$ are connected to. Each plane through a face can be described using the Hesse normal form with plane parameters $\mathbf{\Pi}_f = [\hat{\mathbf{n}}_f^\top, d_f]^\top$ for facet f subject to $\mathbf{\Pi}_f^\top \tilde{\mathbf{x}} = 0$ for a point $\mathbf{x}$ on the plane and using homogeneous coordinates. Costs can now be computed as

$$E(\mathbf{v}) = \sum_{f \in \mathcal{N}_f(\mathbf{v})} (\mathbf{\Pi}_f^\top \tilde{\mathbf{v}})^2 \tag{4.33}$$

$$= \tilde{\mathbf{v}}^\top \underbrace{\sum_{f \in \mathcal{N}_f(\mathbf{v})} (\mathbf{\Pi}_f \mathbf{\Pi}_f^\top)}_{\mathbf{Q}} \tilde{\mathbf{v}} \,. \tag{4.34}$$

We use $\mathcal{N}_f(\mathbf{v})$ to denote the set of neighboring faces of vertex $\mathbf{v}$. Using this cost term, we want to find the position of $\mathbf{v}$ that minimizes $E(\mathbf{v})$

$$\mathbf{v}^* = \arg\min_{\mathbf{v}} E(\mathbf{v}) \,. \tag{4.35}$$

We assume $\mathbf{Q}$ to be constant during the optimization even though the plane parameters change as we move the vertex. Setting the partial derivatives of Equation 4.33 with respect to $\mathbf{v}$ to zero yields the optimal position

$$\tilde{\mathbf{v}}^* = \mathbf{Q}^{-1} \begin{bmatrix} \mathbf{0}_{3\times 1} \\ 1 \end{bmatrix} . \tag{4.36}$$

Figure 4.10 shows the algorithm at work. The top image shows a model consisting of a large number of triangles. After mesh simplification, there are no degenerate triangles left. The number of triangles is reduced to only 30 % of the original number while the geometry did not noticeably suffer.

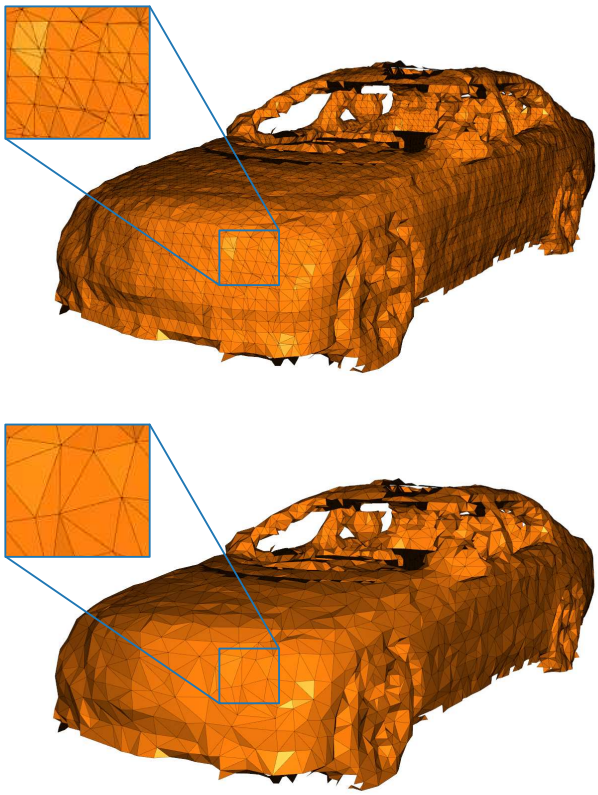

Figure 4.10: Mesh simplification of a reconstructed car. Original mesh (top) and simplified mesh with the number of faces reduced to 30 % (bottom).

5 Large-Scale Mapping

In this chapter, we deal with the methods necessary to map large environments. An overview over the reconstruction pipeline is shown in Figure 5.1.
The proposed mapping pipeline starts off by computing an initial trajectory from consecutive sensor frames which we therefore refer to as odometry. Simultaneously, small portions of the world are reconstructed which we call

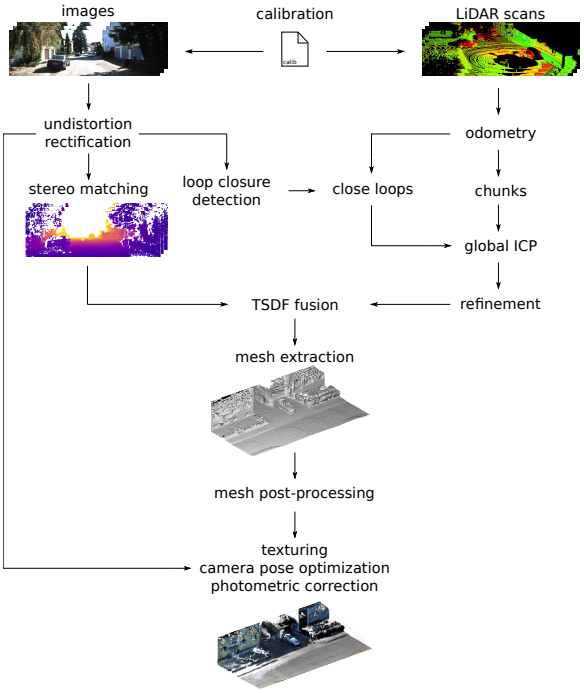

Figure 5.1: Overview of the individual modules of our framework.

chunks. The next step is to detect loop closures using visual point features. This information is used to optimize the trajectory from odometry, such that loops are closed. The resulting trajectory is then used to initialize position and orientation of the chunks which are then aligned using a global ICP making the trajectory globally consistent. The trajectory is used as an initialization for the volumetric reconstruction. Geometry and sensor poses are iteratively optimized by alternating between integration and pose estimation of all sensor frames. As a result, we obtain a globally consistent and accurate geometric reconstruction. Depth from stereo is fused into the TSDF using camera poses interpolated from the LiDAR's trajectory. Finally, meshes are extracted and stitched together. The final stage is texturing and will be discussed in the next chapter.

The methods used in this chapter are linked to our publication [2].

5.1 Odometry

To obtain an initial trajectory $\mathcal{T}_{\text{odom}}$, we run volumetric reconstruction in odometry mode which means that sensor frames are processed in consecutive order. To prevent pose estimation from failing when loops are closed, we only stream out voxels and never stream voxels in. At all times, only a local map within the sensor range is used which is then lost when we move on. The first sensor pose is set to identity $\mathbf{T}_0 = \mathbf{I}$ and defines our world coordinate system.
After obtaining $\mathcal{T}_{\text{odom}}$, we create chunks containing a certain number of sensor frames each. A chunk is a point cloud containing only the vertices of the meshed reconstruction of such a sequence. Consecutive chunks have an overlap of several frames to later increase the number of point correspondences between them. The normal vector of each point can be computed from the normals of adjacent faces by weighting them by the area of the corresponding face. Each face normal is the cross product

$$\mathbf{n}_f = (\mathbf{v}_{f,1} - \mathbf{v}_{f,0}) \times (\mathbf{v}_{f,2} - \mathbf{v}_{f,0}) \,. \tag{5.1}$$

Since the norm of the cross product of two vectors is the area of the parallelogram spanned by the two vectors, $\mathbf{n}_f$ is already scaled by the area of its face

and the vertex normal can be directly computed by summing all face normals followed by normalization

$$\mathbf{n}_v = \sum_{f \in \mathcal{N}_f(v)} \mathbf{n}_f \tag{5.2}$$

$$\hat{\mathbf{n}}_v = \frac{\mathbf{n}_v}{\|\mathbf{n}_v\|} . \tag{5.3}$$

5.2 Loop Closure

During mapping, it can occur that the vehicle revisits an area. If it moves along an already mapped route, the new measurements can seamlessly be integrated into the map. However, if the vehicle comes from a new direction, the map can become inconsistent due to pose drift. The loop can therefore not be closed. Detecting and handling this situation is known as loop closure and it is part of many SLAM framework. In the following two sections, we will explain how we detect loops and how we fix the map such that multiple passes make the map even more accurate.

5.2.1 Loop Closure Detection

Loop closure detection is basically place recognition. This means that we want to know if two sensor frames show approximately the same scene. This can be seen as a classification task.
Numerous techniques have been developed, mainly for camera images. Some approaches try to detect loop closures in LiDAR scans which, in general, tends to be harder due to the lower sensor resolution and the lack of discriminative texture.
In [SGB10], the authors use range images and compute descriptors for a set of interest points. Nearest neighbor search is used to find scans with similar features. A single matched 3D feature point provides the full 6 DoF pose for registration. Different transformation hypotheses are then verified by projecting validation points into the scan to see how well they match.
In [DDS+17], the authors match segmented objects from the current scan to segments from previous scans. Segments are obtained by Euclidean clustering

of the point cloud after removing the ground plane. A descriptor is computed for each segment and then segments are matched using a random forest which provides segment similarity as an output.
One approach that was employed in [Lat13] is a holistic image descriptor which is the concatenation of multiple visual point descriptors that are computed for locations on a regular grid over the image. Computing a similarity measure to all other images of a recorded sequence yields a similarity matrix in which streaks of high similarity indicate loop closures.
A very popular approach is the Fast Appearance-based Mapping algorithm (FAB-MAP) by Cummins et al. ([CN08]) which uses a bag of words representation to describe each image. A visual feature vocabulary is created from a training dataset with each word in the vocabulary being a cluster of visual feature descriptors. FAB-MAP achieves high recall with zero false positives over long routes in the original publication. This is of high importance for every SLAM framework, as false positives result in a wrong topology of the pose graph and can cause SLAM to fail. FAB-MAP was also used in [RIG15] together with FPFH descriptors ([RBB09]) from point clouds. The authors, however, cannot achieve the same recall and precision as the authors of FAB-MAP could achieve with visual features.

In this work, we use SIFT features ([Low04b]) and OpenFABMAP [GMW+12] which is an open source implementation of FAB-MAP.
The method works as follows: First, all extracted features of a training dataset are clustered using mean shift to get a set of common features in the world. This feature set is also called the codebook. The features of an image k are turned into a bag of words representation which is a binary vector $\mathbf{z}_k = [z_1, ..., z_n]_k^\top$ with z_i indicating whether feature i from the codebook is present in the image. Instead of representing a location l_k by the occurrence of z_i directly, FAB-MAP introduces hidden variables e_i which represent the occurrence of physical objects that lead to the observations z_i. Each location l_i is then modeled as the probabilities of each object being in the scene $[p(e_1 = 1|l_i), ..., p(e_n = 1|l_i)]^\top$. Given all observations $\mathcal{Z}_k$ up to the current frame k, being at location l_i can be calculated using Bayes' theorem

$$p(l_i|\mathcal{Z}_k) = \frac{p(\mathbf{z}_k|l_i, \mathcal{Z}_{k-1})\, p(l_i|\mathcal{Z}_{k-1})}{p(\mathbf{z}_k|\mathcal{Z}_{k-1})}\,. \tag{5.4}$$

The prior $p(l_i|\mathcal{Z}_{k-1})$ is computed using a simple motion model which assumes to be close to a previously detected location and therefore assigns neighboring locations a higher probability. The observation likelihood can be expressed using the naïve Bayes assumption

$$p(\mathbf{z}_k|l_i) \approx \prod_{j=1}^{n} p(z_j|l_i), \tag{5.5}$$

and each element of the product can further be expanded to

$$p(z_j|l_i) = \sum_{s\in\{0,1\}} p(z_j|e_j = s)\, p(e_j = s|l_i), \tag{5.6}$$

assuming that the first term of the sum, which is the detection probability, is independent of the location. The detection probability is a parameter provided by the user.
The authors of FAB-MAP show that considering the co-occurrence of features in a scene instead of the simplified assumption of Equation 5.5 yields far better results.

5.2.2 Loop Closing

The output of subsection 5.2.1 are probabilities for each pair of images of showing the same scene. We will refer to images, showing the same scene, as corresponding images. A threshold of 0.99 is applied to filter out unlikely correspondences, and we additionally require corresponding images to be further than 20 seconds apart to avoid matches from the same pass. This provides us a set of correspondences $C = \{(s, d)_k\}$ between source and destination poses $\mathbf{T}_s$ and $\mathbf{T}_d$. Using these correspondences, we optimize the trajectory $\mathcal{T}_{\text{odom}}$, which we obtained from odometry, such that corresponding poses coincide while changing the delta poses of succeeding frames as little as possible. Therefore, we solve a least squares optimization problem for all poses of the

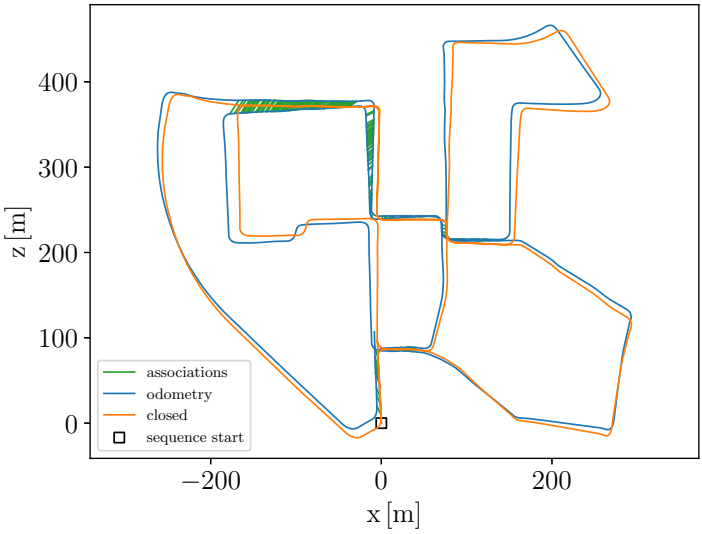

Figure 5.2: Trajectory before (blue) and after loop closure (orange). The green lines connect corresponding poses from the loop closure detection step.

trajectory $\mathcal{T} = \{\mathbf{T}_0, \ldots, \mathbf{T}_{n-1}\}$ using Levenberg-Marquardt optimization (see section 3.6.2)

$$\mathcal{T}_{\text{loop}} = \arg\min_{\mathcal{T}} \sum_{i=0}^{n-2} \|\Delta\mathbf{t}_i - \Delta\mathbf{t}_{i\,\text{odom}}\|^2 + \alpha \measuredangle^2(\Delta\mathbf{R}_i{}^{\top}_{\text{odom}}\Delta\mathbf{R}_i) + \beta \sum_{(s,d)\in C} \|\mathbf{t}_s - \mathbf{t}_d\|^2 + \alpha \measuredangle^2(\mathbf{R}_d{}^{\top}\mathbf{R}_s), \quad (5.7)$$

with

$$\Delta\mathbf{R_i} = \mathbf{R_i}^{\top}\mathbf{R_{i+1}}, \quad \Delta\mathbf{t}_i = \mathbf{t}_{i+1} - \mathbf{t}_i, \quad (5.8)$$

and $\mathbf{R}, \mathbf{t}$ being the rotational and translational part of $\mathbf{T}$. α is a weighting factor to weight the influence of translational and rotational differences and β weights the importance of closing loops against preserving the delta poses from odometry. $\measuredangle(\cdot)$ denotes the rotation angle of the rotation matrix of its argument.

Figure 5.2 shows the result of the pose optimization. The blue trajectory is from odometry and the green lines between poses indicate correspondences from loop closure detection. After optimizing all poses, the orange trajectory is obtained which resembles the original trajectory while corresponding poses coincide.

5.3 Global ICP

Now, that we have a good initialization, we use global ICP to align all chunks simultaneously. The method is called LUM after its inventors Lu and Milios [LM97]. It's an extension to subsection 3.7.2 which, instead of aligning a source to a target point cloud, aligns multiple point clouds at once. Residuals for corresponding points $\mathcal{C}_{ij} = \{(\mathbf{p}, \mathbf{q})_k\}$ with $\mathbf{p} \in \mathcal{P}_i$ and $\mathbf{q} \in \mathcal{P}_j$ are given by

$$r_{ijk} = \left((\mathbf{R}_i\mathbf{p} + \mathbf{t}_i) - (\mathbf{R}_j\mathbf{q} + \mathbf{t}_j)\right)^\top \mathbf{R}_j\hat{\mathbf{n}}_q \,, \tag{5.9}$$

with $\mathbf{R}_i, \mathbf{t}_i$ and $\mathbf{R}_j, \mathbf{t}_j$ being the rotation matrices and translation vectors which transform the two points. Using

$$\mathbf{a} = \begin{bmatrix} \mathbf{p} \times \hat{\mathbf{n}}_q \\ \hat{\mathbf{n}}_q \end{bmatrix} , \tag{5.10}$$

the linearized equation can be written as

$$r_{ijk} \approx \underbrace{\begin{bmatrix} \mathbf{a}^\top & -\mathbf{a}^\top \end{bmatrix}}_{\nabla r_{ijk}^\top} \begin{bmatrix} \boldsymbol{\xi}_i \\ \boldsymbol{\xi}_j \end{bmatrix} + \hat{\mathbf{n}}_q^\top (\mathbf{p} - \mathbf{q}) \,. \tag{5.11}$$

Hessian and gradient of two point clouds i and j are

$$\mathbf{H}_{ij} = \sum_k \begin{bmatrix} \mathbf{a}\mathbf{a}^\top & -\mathbf{a}\mathbf{a}^\top \\ -\mathbf{a}\mathbf{a}^\top & \mathbf{a}\mathbf{a}^\top \end{bmatrix} , \tag{5.12}$$

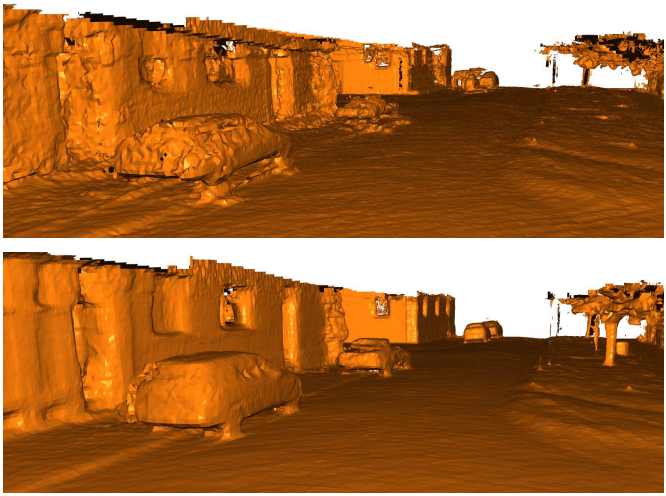

Figure 5.3: Area which was passed three times during recording. Reconstruction from misaligned chunk poses (top) and after three iterations of pose refinement (bottom).

and

$$\mathbf{g}_{ij} = \sum_k r_{ijk} \begin{bmatrix} \mathbf{a} \\ -\mathbf{a} \end{bmatrix}, \tag{5.13}$$

respectively. The full Hessian and gradient can then be composed of the individual 6×6 and 6×1 blocks. We solve the problem using the Gauss-Newton method (section 3.6.2). The resulting trajectory is $\mathcal{T}_{\text{ICP}}$.

5.4 Iterative Pose and Geometry Estimation

Since we rigidly transform each chunk in section 5.3, small discontinuities will be introduced in between chunks. This is because pose drift can still occur within each chunk. As a result, the reconstruction using $\mathcal{T}_{\text{ICP}}$ is of poor quality which can be seen in the top image of Figure 5.3.
For this reason, geometry and sensor poses of the whole sequence have to be jointly optimized. Since there can be thousands of poses and millions of

voxels, we alternate between pose estimation and geometry estimation and hold the other part fixed. Starting with the prior trajectory $\mathcal{T}_{\text{ICP}}$, we integrate all sensor frames into the voxel grid. Then, similar to [ZK14], each sensor pose is aligned to the TSDF, resulting in a new estimate for $\mathcal{T}$. All sensor frames are then integrated into a new voxel grid using the poses from $\mathcal{T}$ and the poses are determined again. The process repeats multiple times until it converges and poses no longer change from one iteration to the next. Figure 5.3 shows the reconstruction result after three iterations.

5.5 Multi-Sensor Fusion

Our framework allows for the integration of measurements from an arbitrary number of sensors, as long as they obey one of the sensor models introduces in section 4.2, the data is stamped and the extrinsic calibration parameters are known. We additionally use depth estimates from stereo cameras. For stereo computation (see subsection 3.3.2), we use a variant of semiglobal matching, as described by Hirschmüller in [Hir08]. We use the implementation from OpenCV[1] which uses larger windows for similarity computation in contrast to the original work which uses per pixel similarities. We noticed that stereo depth, alone, is not sufficient to run our reconstruction pipeline. Therefore, we first run the pipeline using LiDAR only and then fuse stereo depths into the voxel grid using sensor poses obtained from section 5.4. Linear interpolation is used to determine the poses for the given timestamps of the stereo data. Due to the higher level of noise, we fuse stereo depth with only one third the weight of LiDAR which we found empirically to work well. Also the truncation distance has to be increased to twice the distance used for LiDAR to obtain a reconstruction without holes. Using weights inversely proportional to the squared depth, as the considerations from stereo errors in subsection 3.3.2 would suggest, did not improve our results.

[1] https://opencv.org

6 Texturing

In this chapter, we deal with the problem of how to texture a mesh from multiple camera images. Since only LiDAR poses were estimated during reconstruction, we obtain the camera poses by first interpolating the LiDAR pose for the time stamp of the image using the same method as in subsection 4.3.1. Then, we apply the known extrinsic calibration parameters between LiDAR and camera to determine the camera pose in world coordinates.
In the first step of the texturing pipeline, each triangle of the mesh has to be projected into the camera images in which it is visible. To check visibility, we use a depth buffer which is a standard approach from computer graphics. Since a triangle can be visible in multiple images, we have to determine which images to use to achieve the best result and how to combine the information from multiple images to create the final texture.
The textured mesh will show intensity discontinuities due to varying exposures and vignetting of the used images. Further, small errors in the camera poses can cause inconsistent textures. All these effects can be compensated by optimizing the photo-consistency of the texture which leads to a seamless and visually appealing result.

6.1 Visibility Check

For large scenes, there can be millions of faces which have to be checked for visibility in each camera image so we employ parallel computing for each triangle on the GPU to speed up the process. Each thread checks one triangle. First, we project the triangle onto the image plane. The winding order of the triangle is computed to determine whether its front or back side is visible. This can be achieved by checking its normal orientation. Using the definition of Equation 5.1, a visible triangle must fulfill $\hat{\mathbf{n}}_z < 0$ in camera coordinates, i.e. the z component of the normal has to point towards the camera.

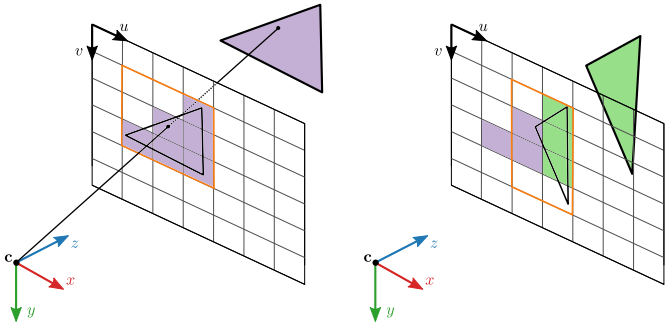

Figure 6.1: Rasterizing two triangles. A ray is shot through each pixel within the orange rectangle which is the enclosing bounding box of the triangle's projection. The green triangle is closer than the purple triangle. Therefore, the depth buffer and index buffer is overwritten with the new values.

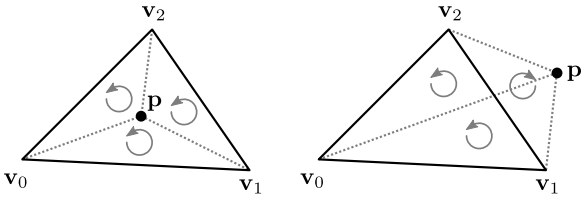

Figure 6.2: Winding of triangles when one vertex is replaced by point $\mathbf{p}$. On the right hand side it can be seen that the triangle $\mathbf{p}, \mathbf{v}_1, \mathbf{v}_2$ has a clockwise winding, as opposed to the other two triangles.

Next, a camera coordinate axis aligned bounding box is determined and we check for each pixel inside the box if its center lies within the triangle (see Figure 6.1). A point lies within the triangle if all triangles that are created when one vertex is replaced by the point, have the same winding direction, as illustrated in Figure 6.2. If the point is within the triangle, the distance from camera origin to the point of intersection between triangle and viewing ray is computed using

$$t^* = \frac{d_f}{-\hat{\mathbf{n}}_f^\top \hat{\mathbf{v}}}, \tag{6.1}$$

assuming that all coordinates are relative to the camera frame, the viewing ray is along $\mathbf{x}(t) = t\hat{\mathbf{v}}$ with ray direction $\hat{\mathbf{v}}$, and the plane the triangle spans is given

in Hesse form $\mathbf{\Pi}_f = [\hat{\mathbf{n}}_f^\top, d_f]^\top$ like in section 4.9. The negative sign comes from the fact that the normal of a visible face is pointing towards the camera. The depth buffer holds the currently smallest t^* for each pixel and the index buffer holds the corresponding face index f. If a single pixel of a triangle is not visible, we mark the whole triangle as not visible, as we only consider fully visible triangles in the proceeding steps.

6.2 View Selection

Each triangle is likely to be visible in multiple images. However, it is recommended to only use information from images which show the triangle from close distance and, ideally, orthogonal to its surface. Problems can occur when triangles are close to occluding edges in the image plane. Small errors in the camera pose or in the mesh geometry can cause a wrong texture result, as can be seen in Figure 6.4 bottom row.

For this reason, we compute a visibility score for each triangle projection which considers the risk of being corrupted by occluding edges. The score $s_{f,c}$ for face f projected into image c is

$$s_{f,c} = \sum_{\mathbf{u} \in \mathcal{U}_{f,c}} 1 + \alpha \min(d(\mathbf{u}), d_{\max}), \tag{6.2}$$

with $\mathcal{U}_{f,c}$ being the set of pixels belonging to a triangle in an image, weighting factor α and $d(\mathbf{u})$ being the distance of a pixel from the nearest occluding edge in the image. The maximum distance is set to $d_{\max}$.

The first component of the score is the number of pixels visible in the image. It increases, the closer the triangle is to the camera and the more orthogonal the camera looks at its surface.

The second component is the sum over the pixel-wise distances from an occluding edge. Occluding edges are obtained by thresholding the gradient magnitude of the depth buffer. A distance transformation is applied to get the distance of each pixel to the nearest occluding edge. Therefore, the further away a pixel is from an occluding edge, the higher the score becomes.

Both scores are combined, using the weighting factor α, in a pixel-wise score map. The score map is shown in Figure 6.3 with underlying triangle projections. The score of a triangle is obtained by summing up all its pixel scores.

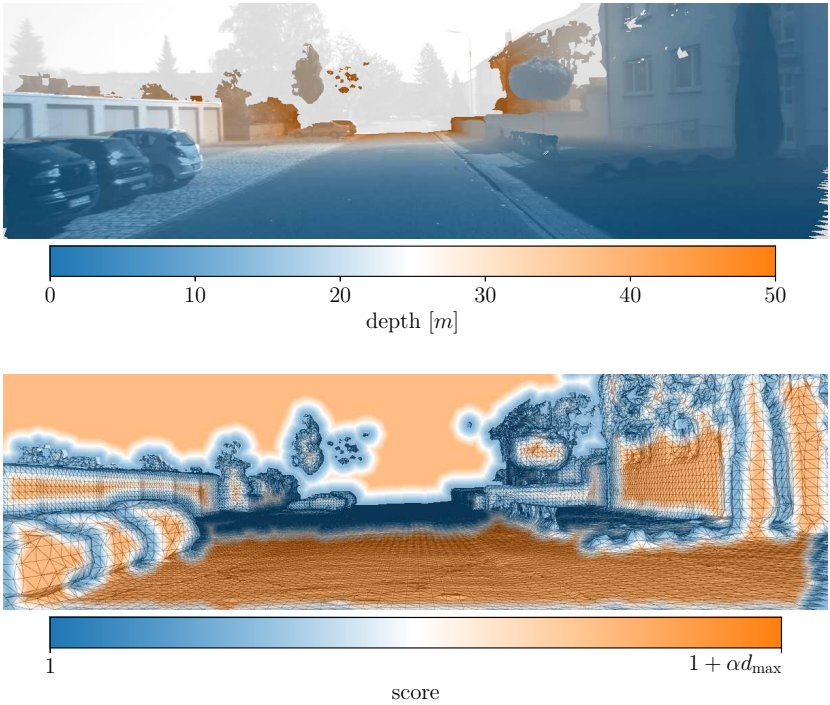

Figure 6.3: Camera image with overlayed depth map (top) and reconstructed mesh with overlayed scores for each pixel (bottom). Areas without depth discontinuities, like road and walls, have a high score whereas areas close to depth discontinuities, like around the parked cars on the left hand side or the tree on the right, have a low score which makes it unlikely that image information from these pixels will be used for texturing.

Figure 6.4: Texturing results without occlusion score (left) and with occlusion score (right). The orange arrows highlight where the changes are best visible.

This can be solved on GPU as following: First, key and value pairs corresponding to face index and score are created for each pixel. Then, using the methods from section 3.5, we sort the pairs by keys. Next, we use segmented scan to add up all scores of a face. Since the pairs are sorted, a segment head can be determined by detecting a changing face index from one element in the list to the next.
The result of this stage of the texture pipeline is a list containing the n best images for each triangle.

Figure 6.4 shows some results without and with considering occlusions by weighting them in the score. In the left column it can be seen that parking cars leave streaks on the ground and on the hedge if we only score triangles by their number of pixels (i.e. setting $\alpha = 0$). Additionally scoring their distance to occluding edges yields the results on the right. The streaking effects almost vanish completely in some areas. Some areas still suffer from streaking because no better camera view is available for texturing. For these cases, one can add a criterion to only texture triangles with a minimum score and leave them untextured otherwise.

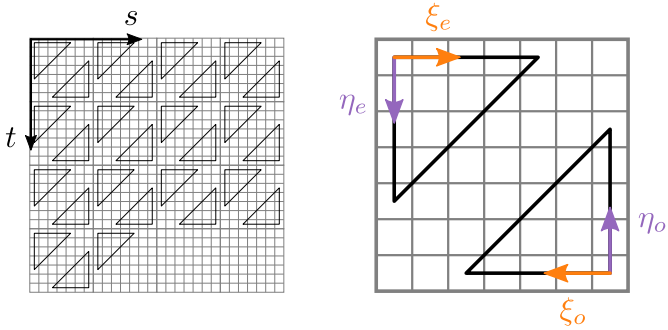

Figure 6.5: Grid structure of the texture map (left) and local coordinate systems of one of its tiles (right).

6.3 Texture Mapping

In computer graphics, textures are usually stored as an image file. A pixel from the texture is therefore also called texel. Each vertex of the mesh does not only hold the vertex position $\mathbf{v}$ but also a pair of texture coordinates s and t which describe the vertex position in the texture. Therefore, the three-dimensional mesh has to be mapped onto a plane just like the surface of the globe is mapped onto the page of an atlas. Since this is not possible for closed meshes, one has to cut the mesh into smaller pieces. This is usually done manually to get as little distortion as possible when the pieces are flattened.

For our goal of creating large textured meshes, we use an approach which is more memory consumptive but which is fully automatic and which does not have to deal with complicated mesh topologies. We texture each triangle separately and independent of each other. The corresponding texture has a fixed position in the texture map which only depends on the face index, so neighboring faces in the texture map are not necessarily neighbored in 3D space. Figure 6.5 shows the texture map on the left. It is a square image which is partitioned into square tiles which hold the texture of two triangles each. A tile has two local coordinate systems with axes ξ and η, which are located in the top left corner for triangles with an even index and in the bottom right corner for triangles with an odd index.

In between the two triangles of a tile, there are three diagonal stripes of texels. This is necessary because most graphics applications interpolate the texture

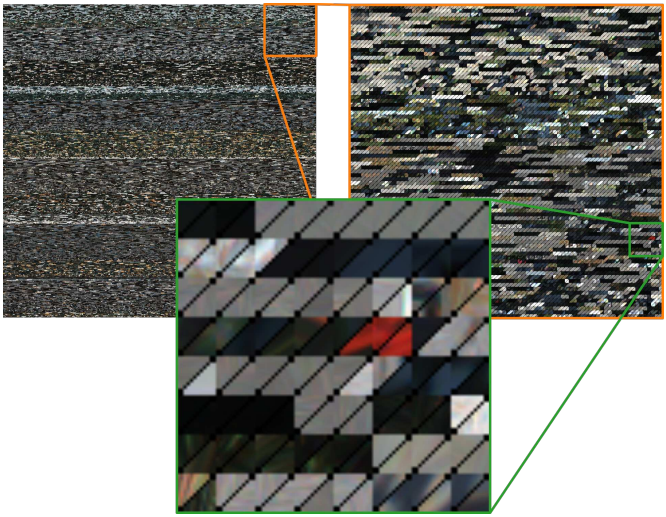

Figure 6.6: Real texture map example of size 5772 × 5772 px (top left) and successively enlarged areas (orange, green rectangles) showing the map's layout up to the individual texels.

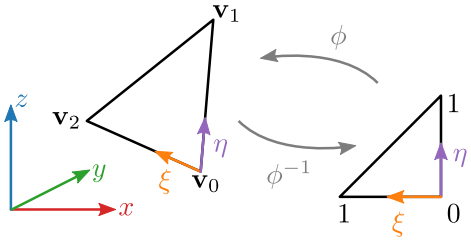

Figure 6.7: Mapping from an arbitrary triangle to the unit triangle and back.

from neighboring texel values. If both triangles of a tile touched each other, the texel values along the shared edge would be interpolated using texels from two different triangles. The two diagonals of texels, touching the triangles, are filled with mean values of neighboring texels. Figure 6.6 shows an example of a texture generated for a real world example. To determine the texel value, we find its location in the camera image and use bilinear interpolation for sub-pixel precision. To map a texel to the camera image, we use the unit triangle

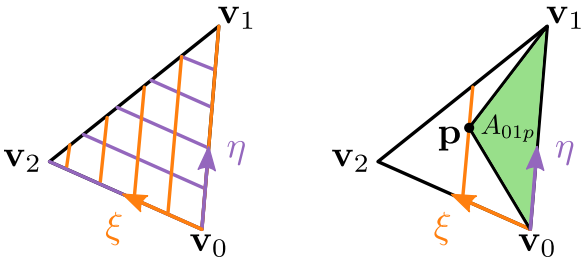

Figure 6.8: Iso lines of barycentric coordinates (left) and computation of barycentric coordinates of a point $\mathbf{p}$ in the triangle plane (right).

mapping ϕ which maps a pair of barycentric texture coordinates $\boldsymbol{\xi} = [\xi, \eta]^\top$ to world coordinates $\mathbf{x}$. Barycentric coordinates are commonly used in computer graphics since they facilitate many tasks [MS16]. As shown in Figure 6.7, each triangle in 3D space defines a local nonorthogonal coordinate system with $\mathbf{v}_0$ as its origin and $\mathbf{v}_2 - \mathbf{v}_0$ and $\mathbf{v}_1 - \mathbf{v}_0$ as its basis. We can therefore map from barycentric to world coordinates using

$$\mathbf{x} = \phi(\boldsymbol{\xi}) = \begin{bmatrix} \mathbf{v}_2 - \mathbf{v}_0 & \mathbf{v}_1 - \mathbf{v}_0 \end{bmatrix} \boldsymbol{\xi} + \mathbf{v}_0 \,. \tag{6.3}$$

The inverse mapping of a point on the surface of a triangle to barycentric coordinates can be obtained by inverting Equation 6.3. However, this leads to an overconstrained problem and requires costly matrix inversions. A much simpler approach can be derived by looking at Figure 6.8 where the iso lines for constant ξ and η coordinates are depicted as orange and purple lines. In the right image, we try to determine the ξ coordinate of a point $\mathbf{p}$ lying in the plane of the triangle. The green area is zero for $\mathbf{p}$ lying on the η-axis and A if it lies on $\mathbf{v}_2$, with A being the area of the whole triangle. Further, the green area does not change when $\mathbf{p}$ moves along the orange iso line. Barycentric coordinates can therefore be computed by the following equations

$$\xi = \frac{A_{p01}}{A} \tag{6.4}$$

$$\eta = \frac{A_{20p}}{A} \,. \tag{6.5}$$

Using $\mathbf{a}^\top\mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos \measuredangle(\mathbf{a}, \mathbf{b})$ and $A = \frac{\|\mathbf{n}\|}{2}$ (see Equation 5.1 for definition of face normal, subscript f is dropped for brevity) this can be rewritten as

$$\xi = \frac{\mathbf{n}^\top \mathbf{n}_{p01}}{\|\mathbf{n}\|^2} \tag{6.6}$$

$$\eta = \frac{\mathbf{n}^\top \mathbf{n}_{20p}}{\|\mathbf{n}\|^2} \,. \tag{6.7}$$

A full derivation can be found in [MS16].

6.4 Texture Blending

An object point might be visible in multiple camera images. Therefore, we want to make full use of all observations by blending the individual observations to obtain the final texel. We implement four blending strategies which are *best view* (no blending), *mean*, *median* and a robustly *weighted score*. For *best view*, the camera with the highest score from section 6.2 is used to texture the triangle. This leads to the sharpest textures but noticeable seams will appear when the camera poses are erroneous. For *mean* and *median*, the final texel value is computed as the mean and median value of the n views with highest scores. *Median* has the advantage that it can filter out outliers which can be the result of dynamic objects that cover a triangle temporarily in a single view. The last method, *weighted score*, blends the pixel values of the n best views by weighting them with the respective triangle scores. To make the method robust against outliers, we use the method from Waechter et al. ([WMG14]) by computing the mean intensity $\bar{\mathbf{I}}$ and covariance matrix $\mathbf{C}_I$ using all n samples to model the color distribution as a multivariate Gaussian distribution. A single pixel is a vector containing intensities of its red, green and blue color channel $\mathbf{I} = [I_r, I_g, I_b]^\top$. For texel $\mathbf{v}$, corresponding sample pixels $\mathbf{u}$ from camera c are only considered in the weighting scheme if their Mahalanobis distance is smaller than a threshold Δ_I

$$\sqrt{(\mathbf{I}_c(\mathbf{u}) - \bar{\mathbf{I}})^\top \mathbf{C}_\mathbf{I}^{-1} (\mathbf{I}_c(\mathbf{u}) - \bar{\mathbf{I}})} < \Delta_I \,. \tag{6.8}$$

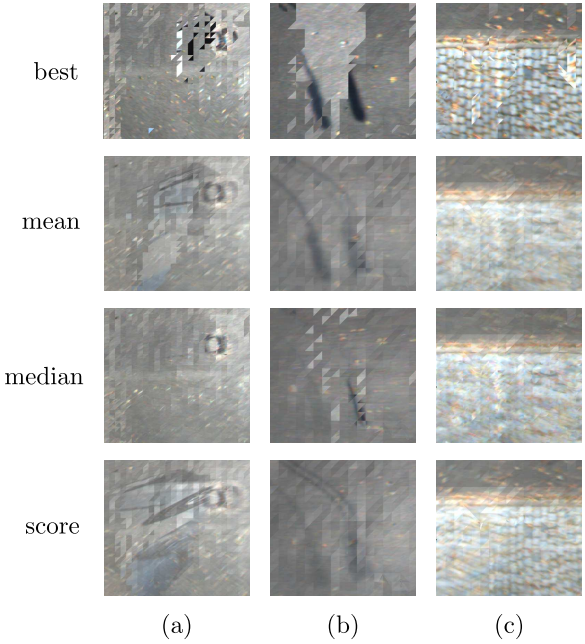

Figure 6.9: Three texture examples (a,b,c) using the four proposed methods for texture blending (rows). (a) shows a piece of road with a gully cover which was passed by a car while recording. (b) shows the shadow of a light pole which appears twice due to erroneous camera poses. (c) shows some leafs next to a curb and some cobblestone below.

All samples $(c, \mathbf{u})$ which fulfill this criterion are gathered in $\mathcal{U}_\mathbf{v}$. The final texel can then be computed using

$$\mathbf{I}_{\text{tex}}(\mathbf{v}) = \frac{\sum_{(c,\mathbf{u})\in\mathcal{U}_\mathbf{v}} s_{f,c}\mathbf{I}_c(\mathbf{u})}{\sum_{(c,\mathbf{u})\in\mathcal{U}_\mathbf{v}} s_{f,c}}, \tag{6.9}$$

with $s_{f,c}$ being the score of the facet which projects to $\mathbf{u}$ in camera c.

Figure 6.9 shows three challenging scenes for texturing and the results from using the different blending schemes discussed above, setting $n = 5$. The challenge in scene (a) is a dynamic object which passed in front of the camera while the scene was recorded. The dynamic object is not present in the reconstruction due to its short presence within the sensor range. This can lead

to triangles on the road being textured from images containing the object. This is the case for *best view* where some white and black triangles are scattered on the road. *Mean* and *weighted score* blend the individual images which results in some ghosting. *Median* robustly removes all traces of the car.
Next, scene (b) shows the effect of erroneous camera poses. The scene was recorded two times, passing the scene in opposite directions. The camera poses have some error, and therefore the images do not properly align. As a result, the shadow of the lamp post appears twice on the street. Again, *median* almost completely removes all shadows while *mean* and *weighted score* show lighter shadows. *Best view* creates the most noticeable seams.
Lastly, scene (c) shows some high frequency structure in the image. There are leafs lying next to a curbstone and there is cobblestone in the lower part. *Best view* preserves most details with the individual stones being clearly visible. *Mean* and *median* smooth out most of the details with *median* preserving the structure a little better than *mean*. *Weighted score* achieves a better result than *mean* and *median*, showing a lot of detail but not as sharp as *best view*.
In conclusion, we can say that using *best view* is a bad strategy if the camera is on auto exposure, as it leaves the texture result fractionated which is highly noticeable to an observer. Using *mean* shows more visually appealing results while smoothing over too many details. This can be improved by weighting individual observations using their scores. The method is able to remove outliers due to its robust weighting scheme. In the scenes shown in the example, the outlying color values were too close to the mean color to be robustly removed. Lowering the maximum Mahalanobis distance below a value of two lead to unpleasant results which come close to the *best view* blending scheme. Using *median* provided a good texture result in the shown example and we therefore use it as our preferred blending scheme.

6.5 Photometric Correction

Any texturing method will lead to noticeable seams which are mainly the result of two effects which are vignetting and varying exposures. In the following, we discuss where these effects come from and how it is possible to correct camera images for these effects even in the case when only image collections are available and no photometric calibration is known, as in the case of the datasets we use.

As discussed in section 3.2, the pixel intensity in an image is proportional to image irradiance E and exposure time t. E is proportional to scene radiance L and $cos^4(\alpha)$ which considers the effects of an aperture, modeled as a hole of certain diameter. With increasing angle α between viewing ray and principal axis, the aperture's area seen along the ray becomes smaller which is the reason for a radial brightness falloff in the image. This effect is called vignetting. The $cos^4(\alpha)$ falloff is a simplified model which does not apply to real cameras that use a system of lenses to bundle incoming light. Real lenses introduce multiple sources of vignetting which come mainly from light paths being blocked by parts of the lens. Lens manufacturers try to compensate for vignetting in their optical systems, however some vignetting still remains even for high quality lenses.

Dynamic range is the range of brightness which a camera can perceive. The scene brightness in the real world can vary significantly. A white surface under direct sunlight can be orders of magnitude brighter than a black surface at night. However, the dynamic range of an image is quite limited and the human eye can only distinguish a relatively small number of shades of gray. In a typical eight bit image, there are 256 pixel intensities possible for each pixel. Therefore, when set to auto exposure, a camera adjusts the exposure time t and gain a for each image, such that the scene brightness is mapped over the whole range of available gray values to preserve as much detail as possible. This causes problems when multiple images with different exposures are stitched together, as pixel intensities of the same object point can be different in each image.
For other applications, like high dynamic range photography, this is a wanted effect. Many images with different exposures are taken of the same scene ([Sze11]). Then, they are combined to create a single image with a higher dynamic range than the individual images. Details in areas which were saturated in over or underexposed images become visible.

In contrast to other work, like [Pit14] or [WMG14], we do not use image blending to get rid of these effects. Instead, we present a method to compensate for varying and unknown exposures by estimating exposures and the camera response curve simultaneously. The method is based on the work of Goldman ([Gol10]).

We assume that the pixel intensity I is the output of the unknown camera response function g which takes exposure time t and image irradiance $E = Lf(d)$ as its input

$$I = g(Ltf(d)). \tag{6.10}$$

E is the product of scene radiance and the vignetting function $f(d)$. d denotes the distance of a pixel to the principal point in the image plane. We model the camera response using an exponential function scaled by a gain a. This is a reasonable assumption, as results from photometric calibration like [GN03] have shown, thus

$$I = a(Ltf(d))^\gamma = (Lkf(d))^\gamma . \tag{6.11}$$

It can be seen that there is a scale ambiguity since the product Lt is the argument of g. Further, we combine a and t to a single variable k. As shown in [Gol10], there is also a gamma ambiguity and an additional ambiguity from the unknown vignetting function f. This makes it impossible to recover the actual parameters. Practically, this is not a problem since any solution is sufficient for photometric correction. The first image is chosen to be the reference image for which we assume $k = 1$ to make the problem solvable.
Further, we assume the vignetting function $f(d)$ to be a polynomial of the form

$$f(d) = 1 + \beta d^2 . \tag{6.12}$$

We assume the world to be Lambertian, therefore the scene radiance L of an object point is constant. We can use this condition to create constraints between observations of the same point in different camera images. Since γ is constant for all images, we can solve Equation 6.11 for L^γ

$$L^\gamma = \frac{I}{(kf(d))^\gamma} . \tag{6.13}$$

For two cameras s and d which see the same object point, the following condition holds

$$L_s^\gamma = L_d^\gamma \tag{6.14}$$

$$\frac{I_s}{(k_s f(d_s))^\gamma} = \frac{I_d}{(k_d f(d_d))^\gamma} . \tag{6.15}$$

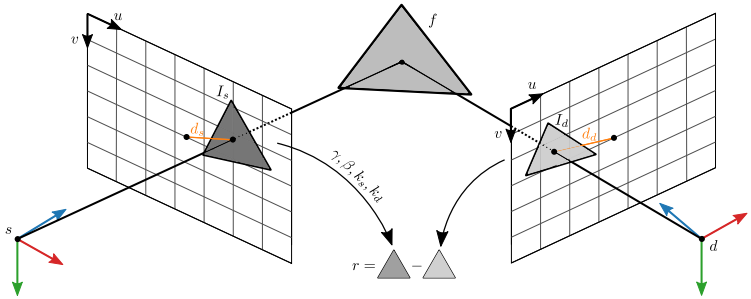

Figure 6.10: Visualization of residual computation between two cameras which see the same triangle. The brightness of the triangle in the source camera is mapped to the destination camera using the estimates of γ, β, k_s and k_d. The remaining brightness difference is the residual.

The intensity of the source image can therefore be mapped to the destination image

$$I_d = \left(\frac{k_d f(d_d)}{k_s f(d_s)} \right)^{\gamma} I_s \,. \tag{6.16}$$

We now minimize the intensity difference of destination intensity and mapped intensity. The computation of a single residual is depicted in Figure 6.10. The residual therefore becomes

$$r = \left(\frac{k_d(1 + \beta d_d^2)}{k_s(1 + \beta d_s^2)} \right)^{\gamma} I_s - I_d \,. \tag{6.17}$$

I_s and I_d are constant values from the respective camera images whereas k of each image and γ, β are the unknowns which can be determined by solving a nonlinear least squares problem. We use Levenberg-Marquardt (section 3.6.2) with a robust loss function to be robust against outliers of I due to erroneous camera poses or temporary occlusions by dynamic objects.
To reduce the complexity of the problem, we do not add per pixel residuals but one residual per triangle. The intensities I_s and I_d are determined by averaging over all pixels of a triangle in an image. Since intensities of 0 and 255 can be saturated, we do not consider these pixels and do not include triangles with too many of these pixels in our problem. Overall, we create one residual for each triangle and each pair of cameras, in which the triangle is visible.

6.6 Camera Pose Optimization

For camera pose optimization, we employ the method proposed in [ZK14]. When camera poses change, the texture result will also change. Therefore, in each iteration of the optimization, the texture map I_{tex} is determined first and held constant while all poses are optimized individually. For optimization, we minimize the photo-consistency residual r which is given by

$$r = I_{\text{tex}}(\mathbf{p}) - I(\boldsymbol{\pi}(\mathbf{g}(\boldsymbol{\xi}, \mathbf{p}))) \tag{6.18}$$

for an object point $\mathbf{p}$ which is transformed to camera coordinates by the linearized transformation $\mathbf{T}$, parameterize by $\boldsymbol{\xi}$ like in subsection 3.7.1. $\mathbf{g}$ is the transformed point and $\boldsymbol{\pi}$ is its projection into the camera image according to Equation 4.5. This also means that $\mathbf{T}$ is the inverse camera pose. We incorporate Equation 6.18 into a least squares problem. Therefore, we need to determine the gradient of r with respect to the parameter vector $\boldsymbol{\xi}$ which yields

$$\nabla r = -\nabla I \mathbf{J}_{\mathbf{g}}(\boldsymbol{\pi}) \mathbf{J}_{\boldsymbol{\xi}}(\mathbf{g}) \tag{6.19}$$

by applying the chain rule. The gradient describes how the brightness of a pixel changes when a small transformation is applied to the camera pose.
The Jacobian of $\boldsymbol{\pi}(\mathbf{x})$ is

$$\mathbf{J}_{\mathbf{g}}(\boldsymbol{\pi}) = \begin{bmatrix} \frac{f}{g_z} & 0 & -\frac{g_x f}{g_z^2} \\ 0 & \frac{f}{g_z} & -\frac{g_y f}{g_z^2} \end{bmatrix}, \tag{6.20}$$

and the Jacobian of the linearized coordinate transformation

$$\mathbf{g}(\boldsymbol{\xi}, \mathbf{p}) = \mathbf{p} + \boldsymbol{\alpha} \times \mathbf{p} + \mathbf{t} \tag{6.21}$$

is given by

$$\mathbf{J}_{\boldsymbol{\xi}}(\mathbf{g}) = \begin{bmatrix} 0 & p_z & -p_y & 1 & 0 & 0 \\ -p_z & 0 & p_x & 0 & 1 & 0 \\ p_y & -p_x & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.22}$$

$\nabla I = [I_u, I_v]^\top$ can be computed using the Sobel operator or the Scharr operator, as done in [ZK14].

There are two possible approaches to choose points **p** for which to set up a residual. The first is to evenly sample each visible triangle in 3D. This method assigns the same importance to all triangles regardless of the size they appear in the camera image. The second method is to compute **p** as the intersection of the viewing ray of each pixel with the triangles. This assigns more weight to close triangles. Also, this method has constant runtime since the number of residuals is bound by the number of pixels. We use Gauss-Newton (see section 3.6.2) for optimization.

6.7 Experiment

To validate the algorithm, we create a simple test scenario in simulation in which ten textured and arbitrarily tilted squares are seen by four cameras. All four cameras take the picture from the exact same position and have the same orientation. The distance to the squares is between 10 m to 13 m. Then, we perturb the camera poses by applying a random offset of up to 10 cm and a rotation of up to 1 °. The perturbed sensor frames are shown in Figure 6.11 alongside the ground truth frames. Using the perturbed sensor poses for texturing, we obtain a blurred result since we use mean blending from section 6.4. After optimization, the textures are sharp and the camera frames are aligned. However, they are off from the ground truth by an average distance of 0.48 m and the mean error in orientation is 2.21 °. The distances between pairs of camera centers are below 0.03 m and relative orientation differences are below 0.17 °. Multiple runs with different initializations show that the method creates a photo-consistent texture but is prone to local minima. For applications in which the precise camera poses are of secondary nature and one only wants to achieve good texture results, this is acceptable.

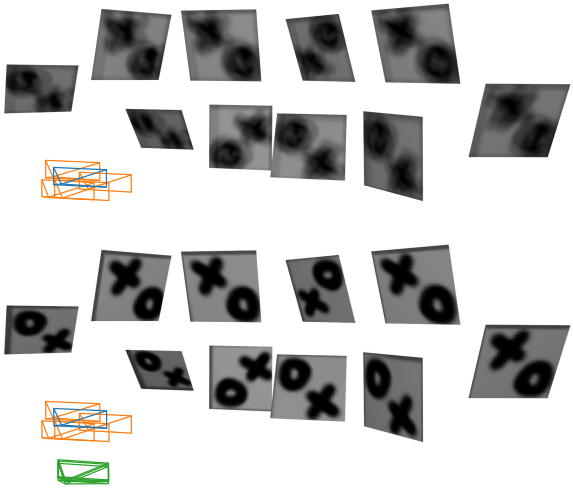

Figure 6.11: Initial camera poses with texturing result (top) and after camera pose optimization (bottom). The orange sensor frames are the initial frames, the green ones are after optimization and the blue ones are the ground truth. The gray borders along the top and left edges of the squares come from texturing with the background color which is a dark gray.

7 Evaluation

In this chapter, we will show results of the proposed reconstruction pipeline and evaluate it quantitatively. The chapter is split into three parts. In the first part, we evaluate the mapping part on the well known KITTI odometry benchmark ([GLU12]) by comparing the trajectories to ground truth trajectories.
In the second part, we look at the accuracy of our reconstructions by using synthetic data which allows to compare against a ground truth which otherwise would be impossible to generate. Also, we investigate the effects of sensor characteristics and assumptions we made on the reconstruction quality.
In the third and last part, we look at results from the texturing stage and show how photometric correction improves the results.
All evaluations and mapping results were generated on a consumer grade laptop, equipped with 16 Gb of RAM, an i7-6700HQ processor with four cores and a GeForce GTX 960M graphics card with 2 Gb of memory.

7.1 Mapping

The sensor data KITTI provides, consists of point clouds recorded with a Velodyne HDL-64E, rotating at 10 Hz, which were compensated for rolling shutter. We use the cylindrical sensor model from subsection 4.2.2 for fusing it into the TSDF. Further, a voxel size of $l_v = 10\,\text{cm}$, a truncation distance of $d_t = 5l_v$ and a maximum sensor range of 50 m is used. Larger sensor ranges, in general, improve pose estimation, as far measurements constrain the pose stronger than close measurements. Also, this increases the chance of having vertical structures within range which is necessary for a good odometry estimate. From the results shown in Figure 7.1, it can be seen that our trajectories match the ground truth trajectories quite accurately, especially in cases with many loop closures like sequences 00 and 05. The KITTI odometry error metrics are shown in Figure 7.2. They consist of translation and rotation error over path length and

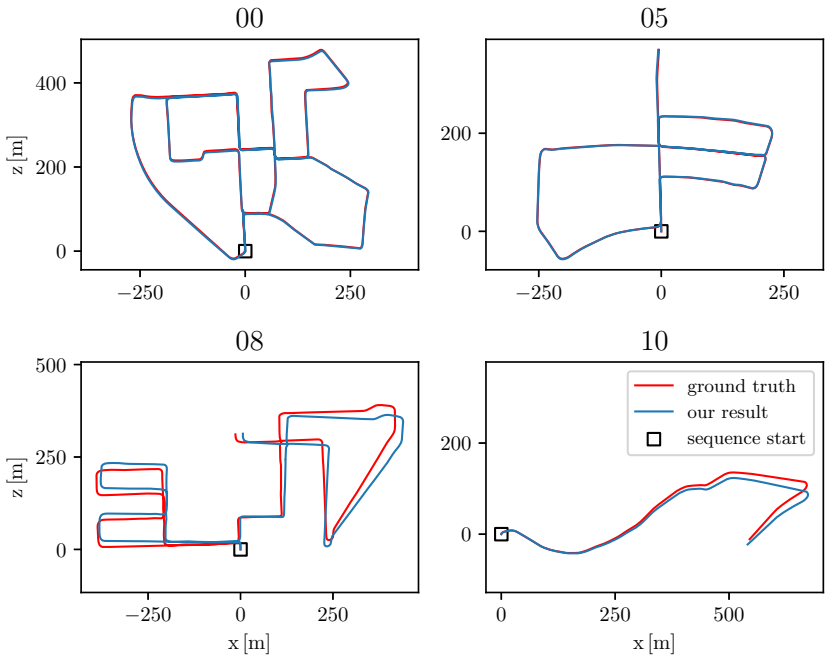

Figure 7.1: Trajectories of KITTI odometry benchmark sequences 00, 05, 08 and 10 using our method (blue) and ground truth (red).

speed. In a sliding window fashion subtrajectories with path lengths ranging from 100 to 800 m are extracted from the result. Then, the delta poses from first to last pose are computed and compared with the ground truth.
While the rotation errors decrease over longer paths, translation errors don't change much. As expected, errors increase for higher speeds, as the initial solution of the ICP is further away from the result each time a pose is estimated. The sudden drop of the translation error for high speeds can be explained by the low number of samples. The mean translation error is 2.4 % and the mean rotation error is 0.011 °/m over all path lengths.
While showing good results in urban scenarios, our reconstruction pipeline fails on sequences 17 and 21 which were recorded on highways. Due to the lack of vertical structure within the range of the LiDAR, ICP cannot converge to a good solution and hence no odometry can be computed.

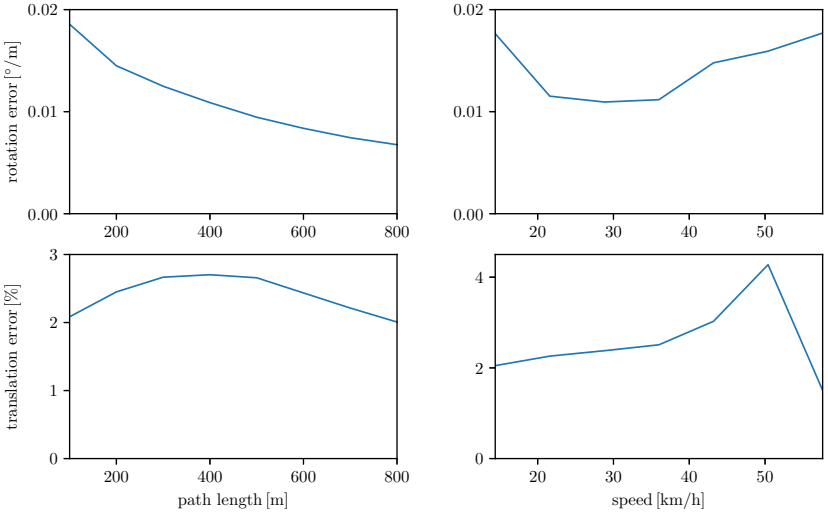

Figure 7.2: Errors averaged over sequences 00, 05, 08 and 10.

Lastly, we show qualitative results of sequence 00 in Figure 7.3. For this example, a voxel size of $l_v = 5\,\text{cm}$ was chosen. The image shows the whole map from a top down perspective and several detailed views alongside the corresponding camera image for comparison. Only LiDAR data was used which explains the limited vertical extent of the reconstruction.

7.2 Reconstruction

In this section, we reconstruct a car using synthetic data from simulation. This has multiple advantages over real data. Firstly, we have full control over real world effects, such as sensor noise and rolling shutter which allows us to compare how these effects influence the reconstruction result. Further, by simulating an ideal sensor, we can determine the theoretical limit of our method. The second advantage of simulation is that the exact ground truth is known so we can directly compare our result to the ground truth mesh.

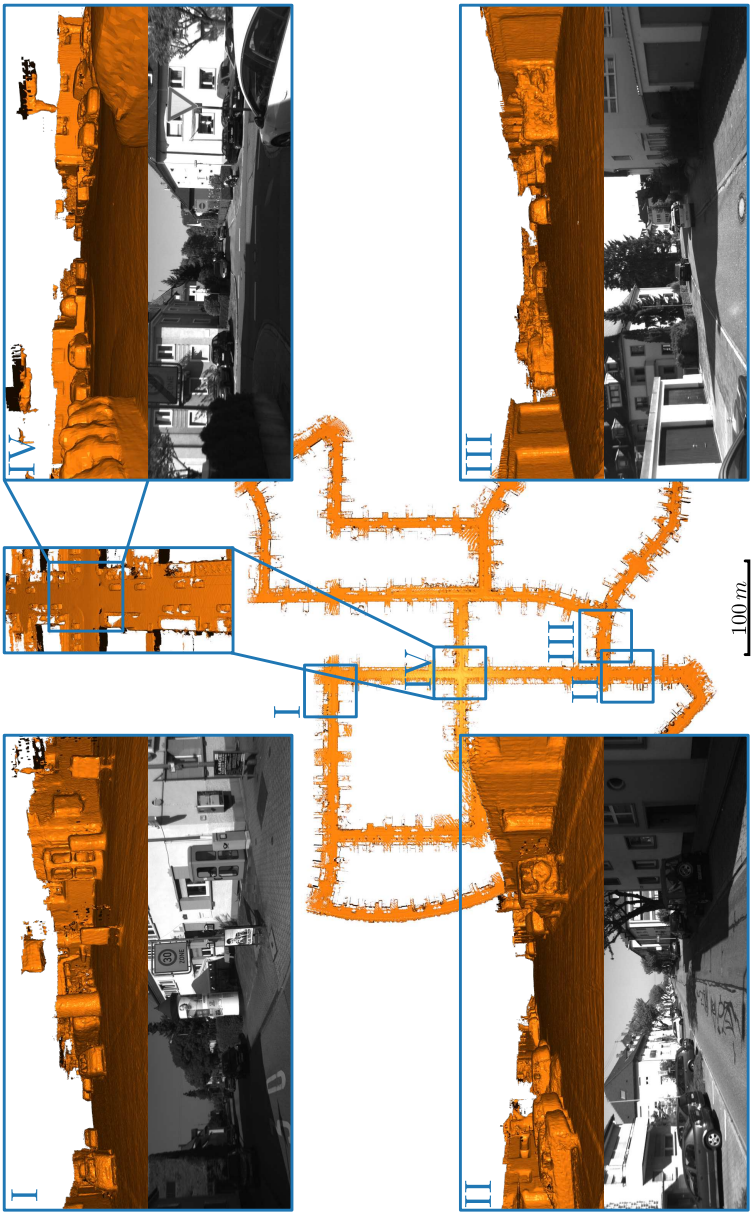

Figure 7.3: Reconstruction of KITTI odometry sequence 00.

7.2.1 Simulation

We use our custom simulation environment which is based on the free modeling software Blender[1]. The reason for this is that most other simulation environments do not consider all effects of rotating LiDARs, such as non single viewpoint of the individual laser beams and rolling shutter. Therefore, we simulate a virtual LiDAR which follows a smooth trajectory which is modeled using a spline. The LiDAR's pose along the trajectory is updated for each point in time a laser beam is fired. Beam directions can be read directly from the LiDAR's calibration file. Measurements are derived by raytracing the meshed ground truth model. Additional Gaussian noise is added to the range measurements to simulate sensor noise.

For the following experiment, we want to investigate how effects from a real LiDAR sensor influence the reconstruction result. The effects we consider are the following:
Non single viewpoint: The cylindrical projection model we use, assumes a common point from which all laser beams originate. This is necessary to allow for fast projection of world points into the sensor model but it is a simplification of a real LiDAR.
Rolling shutter: During a full scan, which we assume to be the measurements of a full sensor head revolution, the LiDAR continuously moves. Therefore, the spatial relation of measured points from different points in time within a scan are not correct. With known sensor movement, this effect can be compensated for static objects according to subsection 4.3.1.
Sensor noise: While we assume the direction of a point measurement to be exact, its range measurement is noisy. This can have various reasons like reflective surface properties, atmospheric effects like fog and dust, or interfering signals from the background (see subsection 3.3.1). In most cases, additive zero mean Gaussian noise is a good approximation for these effects.

We simulate 63 LiDAR scans recorded from a distance of 10 m from a car. The simulated sensor scans from a height of 1.9 m above the ground plane and moves with a speed of 10 m/s on a circular trajectory around the car. The rotation frequency is 10 Hz. Two datasets are recorded. The first one assumes an ideal LiDAR sensor which obeys the cylinder projection model and measures

[1] https://www.blender.org

distances without noise. Further, all points of a scan are recorded from a single position along the trajectory, i.e. rolling shutter is disabled.
The second dataset takes all effects into account which were discussed above. The individual laser beams are modeled using the calibration file of a Velodyne HDL-64. Zero mean Gaussian noise with $\sigma_r = 1.5\,\text{cm}$ is added to each range measurement and individual measurements are taken continuously while the sensor moves along the trajectory.
The ideal dataset allows us to create a reconstruction which we can assume to be the best reconstruction possible using our method while the second dataset will provide a reconstruction which we can assume to achieve in a real world scenario. By comparing both reconstructions, we can further see how real world effects deteriorate the results.

7.2.2 Error Metrics

Since the reconstructions are triangle meshes, we can use the error metrics proposed by Seitz et al. ([SCD+06]). They describe two quantities, accuracy and completeness which are visualized in Figure 7.4.
Accuracy is how close the reconstruction is to the ground truth. As an approximation, it is computed by determining the distances of each vertex from the reconstruction to the nearest ground truth face. Accuracy is the distance up to which 90 % of points reside. Consequently a lower accuracy means a better reconstruction result.
The second quantity is completeness which tells us how much of the ground truth is covered by the reconstruction. It is measured the other way around by measuring the distance of each vertex of the ground truth to the closest reconstructed face. If the nearest face is no more than a predefined distance d away from the vertex, it counts as an inlier. Completeness is the ratio of inlier vertices to the total number of vertices, so a higher ratio means better completeness.

7.2.3 Experiment

In this section, we extend the experiment from [2]. We use the ground truth sensor poses and a voxel size of $l_v = 5\,\text{cm}$ for reconstruction. In total, four

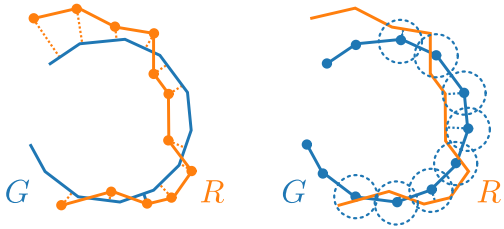

Figure 7.4: Error metrics from [SCD+06] to compare a reconstructed mesh (R) with a ground truth mesh (G). Definition of accuracy on the left side and completeness on the right. For the completeness metric, the inlier distance d is shown as dashed circles.

reconstructions are created using the realistic dataset. First, we reconstruct without compensating rolling shutter. Then, we compensate each scan with the known trajectory and the method from subsection 4.3.1. Next, we integrate each column of a scan separately from the exact location where the sensor was located during recording. This completely removes rolling shutter effects at the cost of a longer run time. We call this method continuous integration. Lastly, we use compensated scans like in the first reconstruction and use Poisson reconstruction ([KBH06]) as a baseline.

The result of each method, together with the ground truth model, are shown in Figure 7.5. Accuracy and completeness, using $d = 5\,\text{cm}$, are listed in Table 7.1. As can be seen, not compensating rolling shutter leads to drastically worse results with an accuracy more than twice as big compared to all other methods. Using simple motion compensation already leads to an accuracy of around 4.5 cm which is below the voxel size. As expected, the reconstruction from continuously integrated measurements is better, but only by 0.255 cm in accuracy and 5.4 % completeness which is quite a small improvement for the extra computation time which is increased by a factor of 2000 for the integration step. Since Poisson reconstruction creates water tight meshes, it creates a bulged out underbody, as there are no measurements which see the car from below. For fair evaluation, we removed these parts manually, as they created large errors which made it the worst reconstruction method in our comparison. Still, it leads to the worst results using compensated measurements despite producing the smoothest surface of all methods. This is because of the areas with large errors inside the car which, again, are a result of the Poisson reconstruction hallucinating surfaces in unobserved areas.

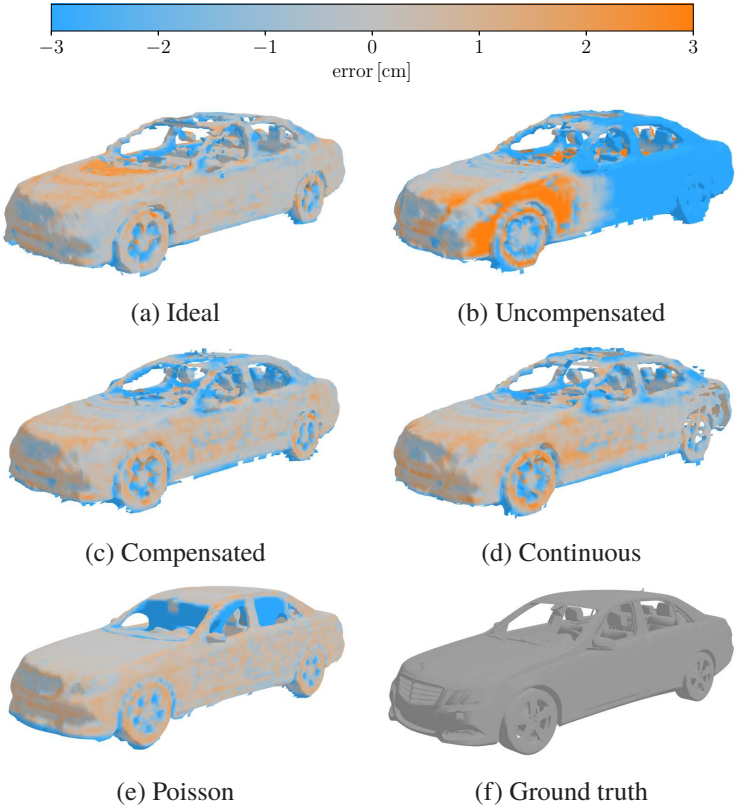

Figure 7.5: Signed distance of the reconstructed surface to the ground truth. Errors are cut off at 3 cm for better visualization. Negative values indicate that the reconstruction is too small, positive values indicate that it is too large.

Scenario	Accuracy [cm]	Completeness [%]
Ideal	3.557	77.04
Uncompensated	11.184	47.85
Compensated	4.553	72.33
Continuous	4.298	77.73
Poisson	7.076	67.97

Table 7.1: Error metrics for the reconstructed car model.

We conclude that simple motion compensation is sufficient for achieving good results. Further, the results from realistic data are quite close to the results from ideal data which leads to the conclusion that the limiting factors of our reconstruction are sensor resolution (number of point samples) and voxel size. Lastly, we can say that our method is capable of creating results with state of the art accuracy. Compared to Poisson reconstruction it does not hallucinate surfaces in unobserved areas.

7.3 Texture

7.3.1 Photometric Correction

Since the true exposures of the KITTI dataset are not known, we show qualitative results from KITTI odometry sequence 00. We texture a loop in the center of the map (see highlighted area in the small map of Figure 7.6). The reason for choosing this section is the many overlapping image sequences, as can be seen by the red camera frames drawn into the images. We use every fourth image of the sequence and use three samples to compute median pixel intensities in the final texture map. After compensating for exposures and vignetting, we obtain $\gamma = 1.167$ and $\beta = -3.752 \times 10^{-7}$. The negative sign of β indicates that the result shows the expected behavior of creating a brightness falloff towards the borders of an image. Figure 7.8 shows the number of residuals that were created for each pair of camera frames. We show results of this scene from a different perspective than the cameras which recorded the scene to give a

true impression of the results. Using the exact same view point for texturing a scene and for rendering the reconstruction will always show perfect alignment of texture and mesh.
A texturing result using raw images and photometrically corrected images can be seen in Figure 7.7. The image on the left shows patches of asphalt with different brightness in areas with homogeneous lighting. These are clearly artifacts from varying camera exposure. After correcting the images, we texture again and obtain the result on the right hand side. The seams mostly vanish. Since the whole dynamic range of the camera is now encoded in an eight bit image, some contrast is lost and the scene looks washed out. However, the brightness of the texture is now a true representation of the brightness of the scene.

7.3.2 Color Integration

Lastly, we want to show colored reconstructions of the KITTI dataset. In Figure 7.9, four scenes are shown which demonstrate the high quality of our textured reconstructions.

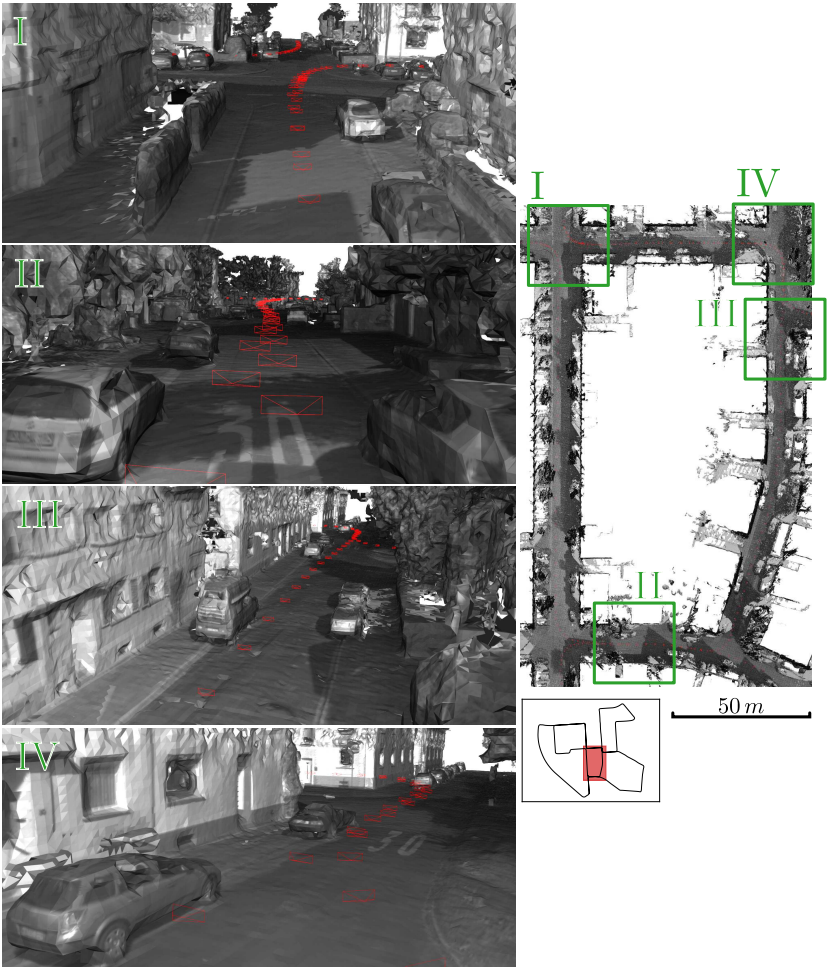

Figure 7.6: Inner part of KITTI odometry sequence 00 after photometric correction. No noticeable seams remain.

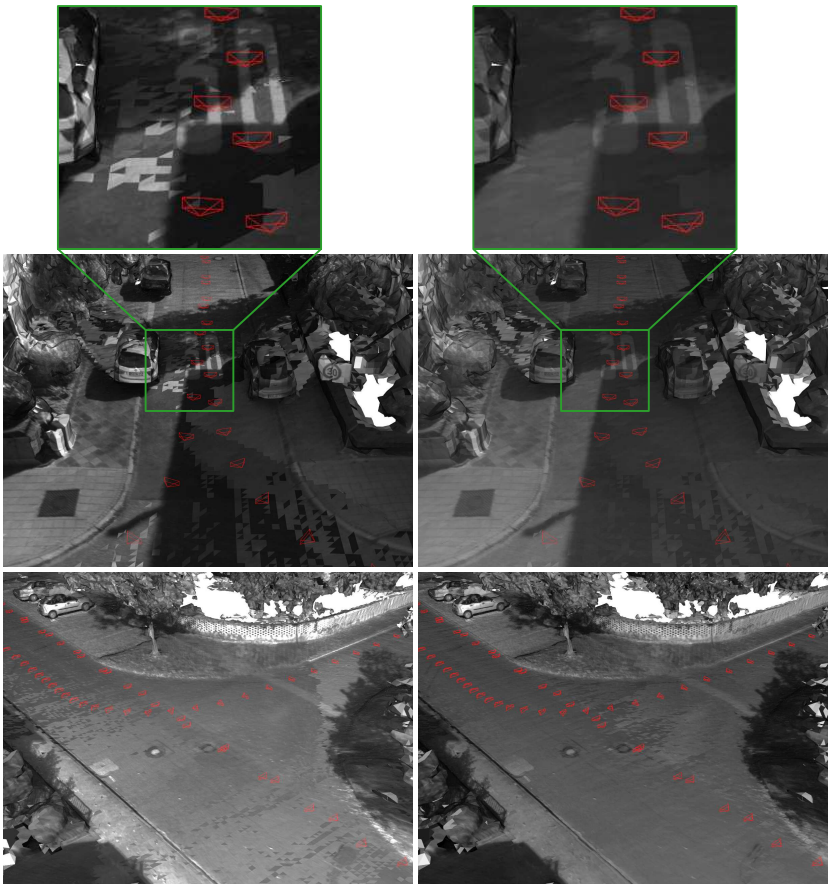

Figure 7.7: Comparison of textured reconstruction with raw images (left) and with images corrected for exposure and vignetting (right). The horizontal change of brightness in the top images is caused by a house casting a shadow onto the street. In the example at the bottom, it can be seen how isolated triangles vanish, as does the sudden change of brightness on the street to the right.

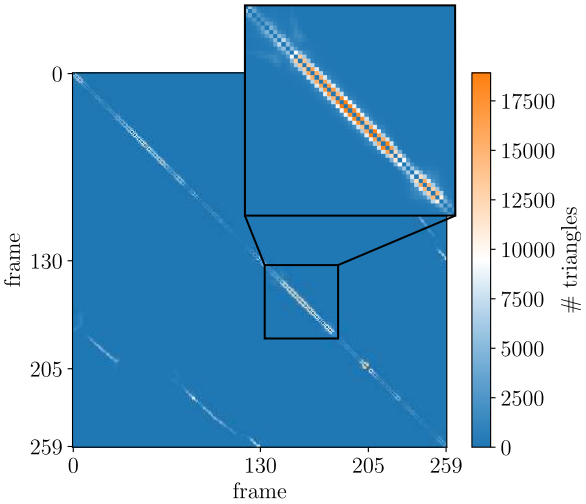

Figure 7.8: Number of observed triangles by pairs of cameras. Frames 0-129, 130-204 and 205-259 are sequences, each. Entries off the main diagonal show the coupling between the sequences. The enlarged area shows an area which is covered by one of the sequences only. Therefore, most residuals are between triangles of consecutive frames.

Figure 7.9: Colored KITTI scenes.

8 Applications

The creation of large-scale textured environment models allows us to solve a variety of different tasks. In the following, we present three applications in which we solve challenging problems that occur in the field of autonomous driving.

8.1 Localization

Autonomous driving today still heavily relies on data from annotated maps with centimeter accuracy, such as [PPJ+18]. With accurate localization in the map, information from the map can be extracted, such as traffic rules or areas of interest like traffic lights or zebra crossings.
For most parts, camera-based localization like [ZLS+14] has been used to achieve this task. These methods, however, heavily rely on visual feature descriptors which robustly encode the appearance of an image patch around an interest point like a corner. During mapping, similar features are found and landmarks describing the same 3D location are triangulated. For localization, the camera pose is optimized to minimize reprojection errors between landmarks in the map and corresponding features in the camera image. However, image features are only robust to changing point of view and illumination changes to some extend. Localization only works close to the mapped route and matching features from day time to the same feature from night time might not be possible.
Localization using LiDAR is a promising approach and the top ranked algorithms in the KITTI odometry benchmark, at the date of this work, are LiDAR-based.
Figure 8.1 shows our self-driving car Bertha. On the corners of its rooftop there are four Velodyne VLP-16 LiDAR scanners. Each scanner has 16 scan lines sweeping the surrounding at 10 Hz. Since their rotation axes are tilted

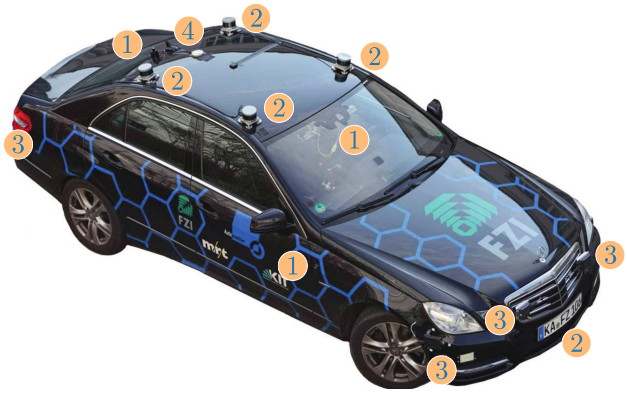

Figure 8.1: Image of our experimental vehicle Bertha. It is equipped with numerous different sensors, such as color and monochrome cameras behind the windshield, the rear window and below the side mirrors (1), four LiDAR sensors with 16 scan lines each on the top of the car and one four line scanner in the front bumper (2), full coverage of the surrounding by radars (3), and GPS (4). Not visible are the wheel encoders which provide wheel odometry and an IMU for acceleration and turn rates.

relative to the vertical, consecutive scans of one LiDAR have little overlap and hence make it difficult for ICP to work. For this reasons, we accumulate all point clouds within a temporal window of 0.1 s and stitch them using the known extrinsic calibration of the four scanners. For extrinsic sensor calibration relative to the rear axle of our vehicle, we employed the methods from [1, 4, 5].
For mapping, a cylindrical sensor model is used which has its origin in the center of mass of the four scanners. After mapping the route for the first time, we use the resulting trajectory to reintegrate each individual scan into a new voxel grid. This time, we use one cylinder model per scanner.
For localization, we, again, use an accumulated point cloud from all four scanners. The pose estimations from our localization are fused with wheel odometry and steering angles using an unscented Kalman filter with a kinematic single-track vehicle model. Its outputs are filtered x and y coordinates of the center of the rear axle and heading angle ψ. We run an independent filter for the z position and feed back the prediction to the localization to initialize ICP. We record the route two times. The first drive is used to compute the map for our LiDAR-based approach. The second drive is used for evaluation so that an independent dataset is used. Additionally, the second drive is in

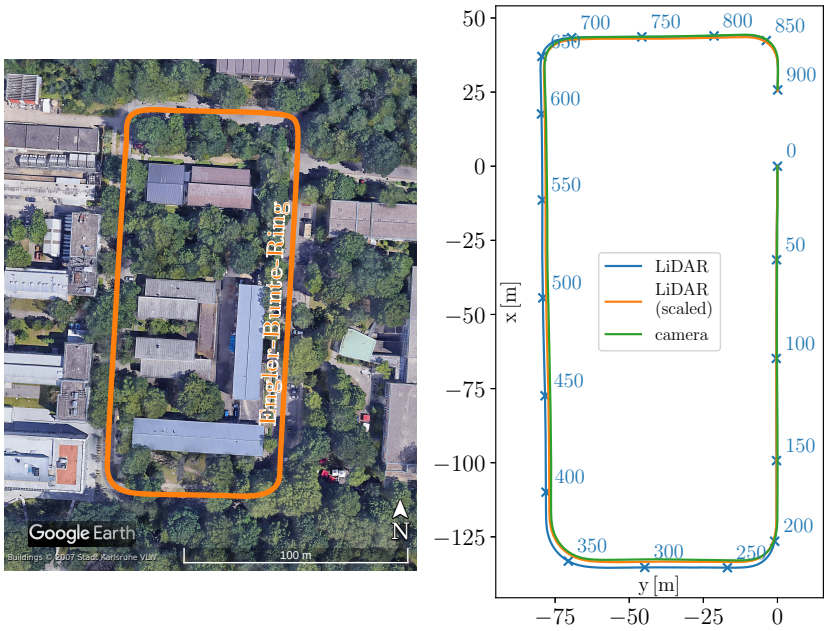

Figure 8.2: Test route on Engler-Bunte-Ring in Karlsruhe, Germany (left) and trajectories from localization (right). Scale and orientation of both images are intentionally not the same. Frame numbers are plotted every 50 frames. The frame rate is 10 Hz.

the opposite direction. We compare our localization to a visual localization from [SLKS17, SS18], which uses point features. For both trajectories, we use the center of the rear axle as the reference frame. Then, we align the first pose of both trajectories. Corresponding poses are determined by linear interpolation.

The resulting trajectories are shown in Figure 8.2. Both trajectories, from LiDAR and from camera, align very well. We notice a scale difference between both trajectories which could be the result of small intrinsic calibration errors, since the focal length is known for causing scale errors in visual SLAM. Therefore, we scale the LiDAR-based trajectory by a factor of 0.986 to make both trajectories match and show it for comparison. We compare poses using the same method as in section 7.1. Figure 8.3 shows the rotation differences

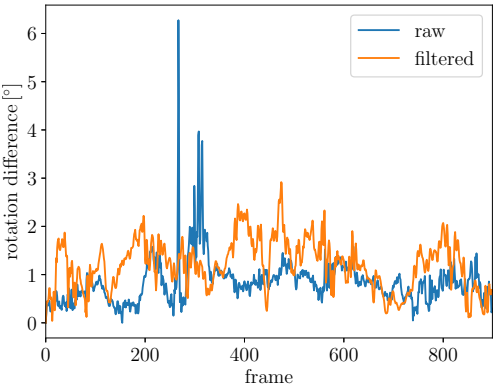

Figure 8.3: Rotation differences between visual localization and LiDAR localization.

of LiDAR and camera localization. We compare unfiltered and filtered poses of both methods. The median difference of the unfiltered poses is 0.8 °.
The average time for pose estimation is 10 ms on an Nvidia Titan X graphics card. The algorithm is therefore real time capable and could even deal with denser point clouds or higher frequencies than 10 Hz. The file size of the map for visual localization is 330 Mb while the file size of the signed distance field used for our LiDAR-based method is only 56 Mb which is only 17 % of the size of the visual map. Our map still contained weights for all occupied voxels. Since the weights were not used for localization, the file size could be further reduced by removing them.
As a final note, we want to point out that LiDAR is not the only sensor which can be used for localization with our reconstructions. FARLAP ([PMSN15]) uses a monocular camera for localization in a textured mesh. The method is similar to section 6.6. However, the authors use the normalized information distance as a similarity metric for the current camera image and a rendered image of the scene.

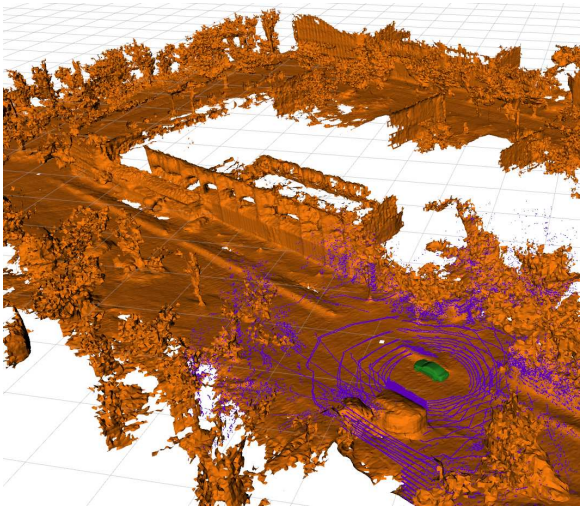

Figure 8.4: Live visualization of the localization map (orange) and current estimate of Bertha's pose (green vehicle model). The current point cloud from LiDARs is shown in purple.

8.2 Simulation

As stated in subsection 1.1.4, simulation is a valuable tool for researchers in autonomous driving and robotics. There are numerous simulation environments available. Some of them are open source projects like Carla [DRC+17] for autonomous driving or AirSim [SDLK18] which is designed for drone simulation. Also, professional tools like IPG CarMaker[1], NVIDIA DRIVE Sim[2] and VIRES Virtual Test Drive[3] exist. Creating detailed models for these applications is labor-intensive, as it requires artists to recreate buildings and vegetation from imagery and city maps. With our proposed reconstruction pipeline, we can generate 3D models fully automatic and can export them to all common 3D file formats, such as .dae, .3ds and .fbx.
We run the following simulations in our own simulation environment, which is

[1] https://ipg-automotive.com
[2] https://www.nvidia.com
[3] https://vires.com

Figure 8.5: A virtual model of Bertha follows a trajectory in simulation. Actual vehicle dynamics are simulated using a game engine.

based on the Blender modeling software to derive virtual sensor measurements like camera images and point clouds from it. The model can also be loaded into the Blender game engine which allows for vehicle physics simulations. In Figure 8.5, we show how a model of Bertha follows a trajectory through a round course that we reconstructed. Alternatively, the user can manually steer the vehicle with the keyboard.

8.3 Labeling

Today's perception heavily relies on machine learning, such as deep neural networks for pixel-wise labeling and instance segmentation of camera images. In order to train and validate such algorithms, a large amount of ground truth data has to be available in the form of image annotations. Annotating is usually done manually by drawing polygons in images. While early datasets, such as KITTI [GLU12], only contain a few hundred images, later datasets like Cityscapes [COR+16] and Mapillary Vistas [NOBK17] already provide tens of thousands of images. Latest released datasets go even further in terms of quantity. ApolloScape [HCG+18] and BDD100K [YXC+18] have over one hundred thousand images each.

The annotation density and annotation quality also increased up to complete and pixel-accurate labels, as claimed by the creators of Cityscapes. On average, it took 90 minutes to label a single image with the highest quality level and seven minutes for coarse annotations.

Other approaches tried to automate labeling to some extent by incorporating 3D information. [SGST13, GFS13, XKSG16, 3] all use LiDAR or stereo reconstructions and project labels from 3D to 2D. Some approaches utilize conditional random fields to get more consistent labels.

Current datasets are not purely labeled manually. The authors of [HCG+18] explain in their work how they use a geo mapping system, using LiDAR, to create a point cloud representation of the environment which is then labeled in 3D. Point labels are then transfered to camera images by projection and point splatting. BDD100K was partly labeled with the aid of neural networks which provided proposals that human annotators had to extend and correct if they were missing or wrong.

An alternative method to generate large amounts of labeled data, is the creation of synthetic images by rendering virtual scenes. Although the data does not provide as much information to an algorithm as real images, it helped to improve results by some extent. Some work that was done in this field is virtual KITTI [GWCV16] and SYNTHIA [RSM+16]. Due to the simplicity of the models, their synthetic nature can easily be noticed. More photo-realistic results were generated by the authors of [RVRK16] who used images from video games.

We created a 3D label tool which is a Blender plugin. The mesh is loaded into the tool and boxes can be placed on objects, such as parking cars, trees and facades. Figure 8.6 shows the graphical user interface. Other parts of the scene, like the road surface, can be easily annotated by drawing a polygon in a top down view. The polygon is then extruded to a volume and a label is assigned via a button in the graphical user interface. When labeling is finished, the program checks for each vertex within which volume it resides and assigns the corresponding label. Then, for each camera image of the sequence, all visible triangles are determined and their label is determined by majority voting of its three vertex labels. The triangle is then labeled in the camera image with the respective label. Due to having per triangle labels, the label contours in the images can be quite jagged. Therefore, we smooth them by applying morphological filtering to each label mask separately, which leads to a smooth appearance.
Using this method, a sequence of 150 images could be labeled in less than 10 minutes. The used reconstruction was created from LiDAR only and with a voxel size of $l_v = 10\,\text{cm}$. Therefore, the vertical extend of the labels is limited. Labels were created for triangles up to a distance of 50 m from the camera. Five pixel-wise labeled samples of the sequence can be seen in Figure 8.7. Instance labels can be created as well. The label boundaries are accurate up to the used voxel grid resolution and for some areas they are even more accurate like next to curbs. The label quality does not achieve the accuracy of the fine Cityscapes annotations, however it is much better than the coarse annotations which make up a large portion of the dataset. Regarding the fast labeling time and the possibility to obtain per frame labels of entire video sequences, our method is a promising tool for future datasets.

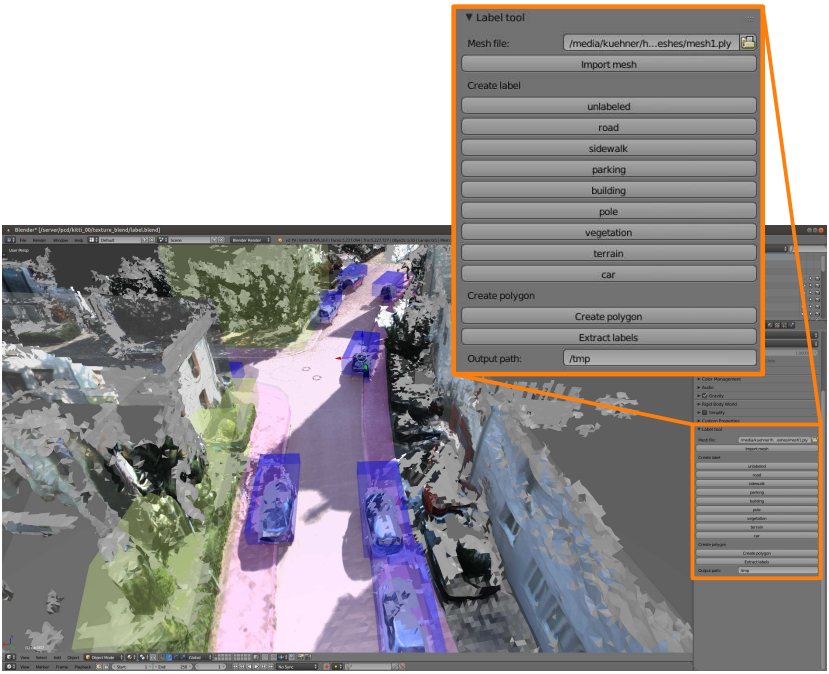

Figure 8.6: Label tool with 3D scene on the left and label options in the right panel. Blue boxes are placed on cars, green boxes on vegetation and so on. The vertex labels can then be exported to a binary file for later processing.

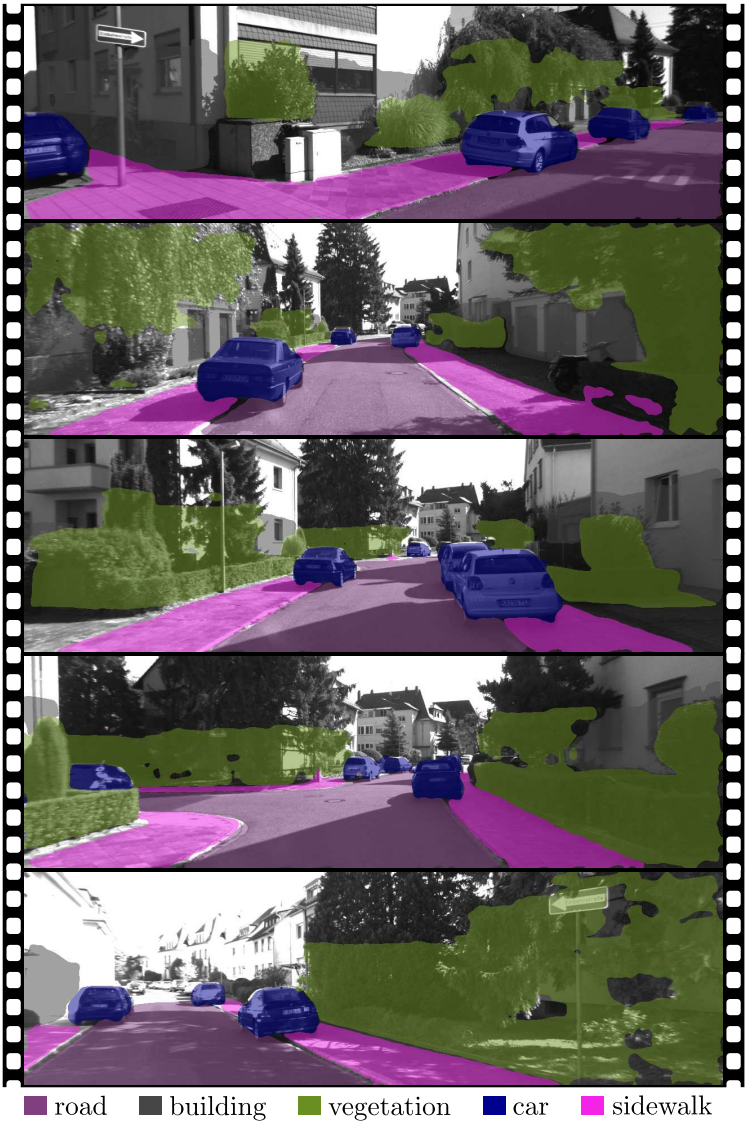

Figure 8.7: Sequence of automatically labeled images in order of their recording from top to bottom.

9 Conclusion and Outlook

In this work, we presented a method for automatic creation of large-scale textured 3D reconstructions. Range measurements from LiDAR and stereo cameras were fused using volumetric reconstruction with truncated signed distance functions. We extended the method by a cylindrical projection model to utilize data from LiDAR. The usage of spatial hashing to store voxel information and streaming data between GPU and CPU, as well as efficient data structures on CPU allowed us to reconstruct areas of arbitrary size.
We presented a mapping pipeline which is capable of detecting loop closures and which used these additional constraints to initialize a global ICP that aligns small fractions of the map. An additional refinement step optimizes sensor poses and geometry in an alternating manner to create a globally consistent reconstruction.
A triangle mesh was extracted and textured from camera images. A weighting scheme was proposed that scores each triangle according to its visibility in an image, considering occlusions. In the following steps, photometric correction was applied to compensate for effects like varying exposure times and vignetting. An additional optimization of all camera poses, maximizing photoconsistency, further improves the visual appearance of the reconstruction.

We ran our algorithms on different sensor setups and reconstructed sequences over multiple kilometers in length using a consumer laptop for computation only. Different scenes from the well known KITTI benchmark were reconstructed and mapping accuracy was evaluated using the KITTI odometry benchmark. While achieving very accurate results in urban areas, our method fails on highways with less vertical structure.
Further, we evaluated the reconstruction quality with synthetic data that allowed for comparison with exact ground truth data. Effects of different parameters and assumptions, such as the sensor model and sources of errors, were investigated and evaluated quantitatively. We found volumetric reconstruction to be suitable for the reconstruction of urban scenarios but the method has some

limitations, as it is not possible to capture thin structures smaller than the voxel grid resolution, such as poles and traffic signs.

Multiple real world applications showed the use and applicability of our method. We used it on our experimental vehicle on which it provided accurate and robust localization. A tool for fast semantic image annotation was developed that reduced labeling time by orders of magnitude compared to conventional approaches, by projecting labels from 3D to 2D. Lastly, we used our textured reconstructions to create photo-realistic simulations for autonomous driving and let a virtual car drive withing a reconstructed world. The simulation could further be used to derive scans and images from new viewpoints of a real scene.

We see numerous promising directions for future work that could significantly improve the performance and applicability of our reconstruction framework.
Firstly, motion constraints and additional information from IMU and GPS should be incorporated into the mapping stage to get smoother trajectories in areas where ICP has problems and to obtain georeferenced maps.
Secondly, the reconstruction of dynamic objects allows for 4D scene reconstruction which not only captures the spatial properties but also the temporal course of a scene. This is particularly of interest for tasks like ground truth generation.
Lastly, information from camera and range measurements from LiDAR should be jointly used for pose estimation and reconstruction instead of fusing them in separate stages.

Bibliography

[AHK+15] T. Arens, F. Hettlich, C. Karpfinger, U. Kockelkorn, K. Lichtenegger, and H. Stachel. *Mathematik*. Springer, Berlin/Heidelberg, Germany, 3rd edition, 2015.

[AMK10] Ehsan Aganj, Pascal Monasse, and Renaud Keriven. Multi-view texturing of imprecise mesh. In *Asian Conference on Computer Vision (ACCV), Xi'an, China*, pages 468–476, 2010.

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics, Amsterdam, Netherlands*, pages 3–10, 1987.

[BE14] Per Bergström and Ove Edlund. Robust registration of point sets using iteratively reweighted least squares. *Computational Optimization and Applications*, 58(3):543–561, 2014.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, USA, 1st edition, 2006.

[Bjö96] Åke Björck. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, USA, 1st edition, 1996.

[Ble89] Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[BLPG18] Ioan Andrei Bârsan, Peidong Liu, Marc Pollefeys, and Andreas Geiger. Robust dense mapping for large-scale dynamic environments. In *IEEE International Conference on Robotics and Automation (ICRA), Brisbane, Australia*, pages 7510–7517, 2018.

[BM92] Paul J. Besl and Neil D. McKay. A method for registration of 3-d shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 14(2):239–256, 1992.

[BS03] Peter Biber and Wolfgang Straßer. The normal distributions transform: a new approach to laser scan matching. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Las Vegas, USA*, pages 2743–2748, 2003.

[BS18] Jens Behley and Cyrill Stachniss. Efficient surfel-based slam using 3d laser range data in urban environments. In *Robotics: Science and Systems (RSS), Pittsburgh, USA*, 2018.

[BSK+13] Erik Bylow, Jürgen Sturm, Christian Kerl, Fredrik Kahl, and Daniel Cremers. Real-time camera tracking and 3d reconstruction using signed distance functions. In *Robotics: Science and Systems (RSS), Berlin, Germany*, 2013.

[BTS+17] Matthew Berger, Andrea Tagliasacchi, Lee M. Seversky, Pierre Alliez, Gaël Guennebaud, Joshua A. Levine, Andrei Sharf, and Claudio T. Silva. A survey of surface reconstruction from point clouds. *Eurographics Computer Graphics Forum*, 36(1):301–329, 2017.

[BZ16] Nicolas Baghdadi and Mehrez Zribi. *Optical Remote Sensing of Land Surfaces: Techniques and Methods*. ISTE Press/Elsevier, London/Oxford, UK, 1st edition, 2016.

[CBC+01] Jonathan C. Carr, Richard K. Beatson, Jon B. Cherrie, Tim J. Mitchell, W. Richard Fright, Bruce C. McCallum, and Tim R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, Los Angeles, USA*, pages 67–76, 2001.

[CBI13] Jiawen Chen, Dennis Bautembach, and Shahram Izadi. Scalable real-time volumetric surface reconstruction. *ACM Transactions on Graphics (TOG)*, 32(4):art. no. 113, 2013.

[CL96] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, New Orleans, USA*, pages 303–312, 1996.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge/London, USA/UK, 2nd edition, 2001.

[CM91] Yang Chen and Gérard G. Medioni. Object modeling by registration of multiple range images. In *IEEE International Conference on Robotics and Automation (ICRA), Sacramento, USA*, pages 2724–2729, 1991.

[CN08] Mark Cummins and Paul Newman. Fab-map: Probabilistic localization and mapping in the space of appearance. *International Journal of Robotics Research (IJRR)*, 27(6):647–665, 2008.

[COR+16] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, USA*, pages 3213–3223, 2016.

[DDS+17] Renaud Dubé, Daniel Dugas, Elena Stumm, Juan Nieto, Roland Siegwart, and Cesar Cadena. Segmatch: Segment based place recognition in 3d point clouds. In *IEEE International Conference on Robotics and Automation (ICRA), Singapore*, pages 5266–5272, 2017.

[Des18] Jean-Emmanuel Deschaud. Imls-slam: Scan-to-model matching based on 3d data. In *IEEE International Conference on Robotics and Automation (ICRA), Brisbane, Australia*, pages 2480–2485, 2018.

[DNZ+17] Angela Dai, Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Christian Theobalt. Bundlefusion: Real-time globally consistent 3d reconstruction using on-the-fly surface reintegration. *ACM Transactions on Graphics (TOG)*, 36(3):art. no. 24, 2017.

[DPRR13] Amaury Dame, Victor A. Prisacariu, Carl Y. Ren, and Ian Reid. Dense reconstruction using 3d object shape priors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Portland, USA*, pages 1288–1295, 2013.

[DRC+17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Annual Conference on Robot Learning (CoRL), Mountain View, USA*, pages 1–16, 2017.

[Dry16] Ivan Dryanovski. 3d reconstruction with tango. In *IEEE Hot Chips Symposium (HCS), Cupertino, USA*, pages 1–24, 2016.

[DSM+17] Maksym Dzitsiuk, Jürgen Sturm, Robert Maier, Lingni Ma, and Daniel Cremers. De-noising, stabilizing and completing 3d reconstructions on-the-go using plane priors. In *IEEE International Conference on Robotics and Automation (ICRA), Singapore*, pages 3976–3983, 2017.

[EDDM+08] Martin Eisemann, Bert De Decker, Marcus Magnor, Philippe Bekaert, Edilson de Aguiar, Naveed Ahmed, Christian Theobalt, and Anita Sellent. Floating textures. *Eurographics Computer Graphics Forum*, 27(2):409–418, 2008.

[FG11] Simon Fuhrmann and Michael Goesele. Fusion of depth maps with multiple scales. *ACM Transactions on Graphics (TOG)*, 30(6), 2011.

[FG14] Simon Fuhrmann and Michael Goesele. Floating scale surface reconstruction. *ACM Transactions on Graphics (TOG)*, 33(4):art. no. 46, 2014.

[FTF+15] Nicola Fioraio, Jonathan Taylor, Andrew Fitzgibbon, Luigi Di Stefano, and Shahram Izadi. Large-scale and drift-free surface reconstruction using online subvolume registration. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, USA*, pages 4475–4483, 2015.

[GFS13] Valeria Garro, Andrea Fusiello, and Silvio Savarese. Label transfer exploiting three-dimensional structure for semantic segmentation. In *International Conference on Computer Vision/Computer Graphics Collaboration Techniques and Applications, Berlin, Germany*, pages 1–7, 2013.

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, Los Angeles, USA*, pages 209–216, 1997.

[GKK+04] Lazaros Grammatikopoulos, Ilias Kalisperakis, George Karras, T. Kokkinos, and E. Petsa. Automatic multi-image photo-texturing of 3d surface models obtained with laser scanning. In *CIPA International workshop on Vision Techniques Applied to the Rehabilitation of City Centres, Lisbon, Portugal*, 2004.

[GLU12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Providence, USA*, pages 3354–3361, 2012.

[GMW+12] Arren Glover, William Maddern, Michael Warren, Stephanie Reid, Michael Milford, and Gordon Wyeth. Openfabmap: An open source toolbox for appearance-based loop closure detection. In *IEEE International Conference on Robotics and Automation (ICRA), Saint Paul, USA*, pages 4730–4735, 2012.

[GN03] Michael D. Grossberg and Shree K. Nayar. What is the space of camera response functions? In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Madison, USA*, pages 595–602, 2003.

[Gol10] Dan B. Goldman. Vignette and exposure calibration and compensation. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 32(12):2276–2288, 2010.

[Gro03] Jürgen Groß. *Linear Regression*. Springer, Berlin/Heidelberg, Germany, 1st edition, 2003.

[GWCV16] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtualworlds as proxy for multi-object tracking analysis. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, USA*, pages 4340–4349, 2016.

[GWO+10] Ran Gal, Yonatan Wexler, Eyal Ofek, Hugues Hoppe, and Daniel Cohen-Or. Seamless montage for texturing models. *Eurographics Computer Graphics Forum*, 29(2):479–486, 2010.

[GXY+17] Kaiwen Guo, Feng Xu, Tao Yu, Xiaoyang Liu, Qionghai Dai, and Yebin Liu. Real-time geometry, albedo, and motion reconstruction using a single rgb-d camera. *ACM Transactions on Graphics (TOG)*, 36(3):art. no. 32, 2017.

[HCG+18] Xinyu Huang, Xinjing Cheng, Qichuan Geng, Binbin Cao, Dingfu Zhou, Peng Wang, Yuanqing Lin, and Ruigang Yang. The apolloscape dataset for autonomous driving. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Workshop, Salt Lake City, USA*, pages 1067–1073, 2018.

[HFBM13] Peter Henry, Dieter Fox, Achintya Bhowmik, and Rajiv Mongia. Patch volumes: Segmentation-based consistent mapping with

rgb-d cameras. In *International Conference on 3D Vision (3DV), Seattle, USA*, pages 398–405, 2013.

[Hir08] Heiko Hirschmüller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 30(2):328–341, 2008.

[Hor86] Berthold K. P. Horn. *Robot Vision*. The MIT Press, Cambridge/London, USA/UK, 1st edition, 1986.

[HS86] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

[HWB+13] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. Octomap: an efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[IKH+11] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and et al. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera. In *ACM Symposium on User Interface Software and Technology (UIST), Santa Barbara, USA*, pages 559–568, 2011.

[IZN+16] Matthias Innmann, Michael Zollhöfer, Matthias Nießner, Christian Theobald, and Marc Stamminger. Volumedeform: Real-time volumetric non-rigid reconstruction. In *European Conference on Computer Vision (ECCV), Amsterdam, Netherlands*, pages 362–379, 2016.

[JJKL16] Junho Jeon, Yeongyu Jung, Haejoon Kim, and Seungyong Lee. Texture map generation for 3d reconstructed scenes. *The Visual Computer*, 32(6):955–965, 2016.

[KBH06] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Eurographics Symposium on Geometry Processing (SGP), Cagliari, Italy*, pages 61–70, 2006.

[KDSX15] Matthew Klingensmith, Ivan Dryanovski, Siddhartha Srinivasa, and Jizhong Xiao. Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields. In *Robotics: Science and Systems (RSS), Rome, Italy*, 2015.

[KLL+13] Maik Keller, Damien Lefloch, Martin Lambers, Shahram Izadi, Tim Weyrich, and Andreas Kolb. Real-time 3d reconstruction in dynamic scenes using point-based fusion. In *International Conference on 3D Vision (3DV), Seattle, USA*, pages 1–8, 2013.

[KPM16] Olaf Kähler, Victor A. Prisacariu, and David W. Murray. Real-time large-scale dense 3d reconstruction with loop closure. In *European Conference on Computer Vision (ECCV), Amsterdam, Netherlands*, pages 500–516, 2016.

[KPR+15] Olaf Kähler, Victor A. Prisacariu, Carl Y. Ren, Xin Sun, Philip Torr, and David Murray. Very high frame rate volumetric integration of depth images on mobile devices. *IEEE Transactions on Visualization and Computer Graphics*, 21(11):1241–1250, 2015.

[Lat13] Henning Lategahn. *Mapping and Localization in Urban Environments Using Cameras*. KIT Scientific Publishing, Karlsruhe, Germany, 1st edition, 2013.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, Anaheim, USA*, pages 163–169, 1987.

[LI07] Victor Lempitsky and Denis Ivanov. Seamless mosaicing of image-based texture maps. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Minneapolis, USA*, pages 1–6, 2007.

[LM97] Feng Lu and Evangelos Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4(4):333–349, 1997.

[Low04a] Kok-Lim Low. *Linear Least-Squares Optimization for Point-to-Plane ICP Surface Registration*. Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, USA, 2004.

[Low04b] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision (IJCV)*, 60(2):91–110, 2004.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson,

James Davis, Jeremy Ginsberg, and et al. The digital michelangelo project: 3d scanning of large statues. In *ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, New Orleans, USA*, pages 131–144, 2000.

[MKC+17] Robert Maier, Kihwan Kim, Daniel Cremers, Jan Kautz, and Matthias Nießner. Intrinsic3d: High-quality 3d reconstruction by joint appearance and geometry optimization with spatially-varying lighting. In *IEEE International Conference on Computer Vision (ICCV), Venice, Italy*, pages 3133–3141, 2017.

[MKG11] Patrick Mücke, Ronny Klowsky, and Michael Goesele. Surface reconstruction from multi-resolution sample points. In *Eurographics Vision Modeling and Visualization (VMV), Berlin, Germany*, pages 105–112, 2011.

[ML14] Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 36(11):2227–2240, 2014.

[MS16] Steve Marschner and Peter Shirley. *Fundamentals of Computer Graphics*. Taylor & Francis, Boca Raton, USA, 4th edition, 2016.

[MSC+16] Oleg Muratov, Yury Slynko, Vitaly Chernov, Maria Lyubimtseva, Artem Shamsuarov, and Victor Bucha. 3dcapture: 3d reconstruction for a smartphone. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Workshop, Las Vegas, USA*, pages 893–900, 2016.

[NFS15] Richard A. Newcombe, Dieter Fox, and Steven M. Seitz. Dynamicfusion: Reconstruction and tracking of non-rigid scenes in real-time. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, USA*, pages 343–352, 2015.

[Ngu07] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley, Boston, USA, 1st edition, 2007.

[NIH+11] Richard A. Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J. Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *IEEE*

International Symposium on Mixed and Augmented Reality (ISMAR), Basel, Switzerland, pages 127–136, 2011.

[NOBK17] Gerhard Neuhold, Tobias Ollmann, Samuel R. Bulò, and Peter Kontschieder. The mapillary vistas dataset for semantic understanding of street scenes. In *IEEE International Conference on Computer Vision (ICCV), Venice, Italy*, pages 5000–5009, 2017.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Science+Business Media, New York, USA, 2nd edition, 2006.

[NZIS13] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (TOG)*, 32(6):art. no. 169, 2013.

[OKI15] Peter Ondrúška, Pushmeet Kohli, and Shahram Izadi. Mobilefusion: Real-time volumetric surface reconstruction and dense tracking on mobile phones. *IEEE Transactions on Visualization and Computer Graphics*, 21(11):1251–1258, 2015.

[PCS15] François Pomerleau, Francis Colas, and Roland Siegwart. A review of point cloud registration algorithms for mobile robotics. *Foundations and Trends® in Robotics*, 4(1):1–104, 2015.

[PGB03] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson image editing. In *ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, San Diego, USA*, pages 313–318, 2003.

[Pit14] Benjamin Pitzer. *Automatic Reconstruction of Textured 3D Models*. KIT Scientific Publishing, Karlsruhe, Germany, 1st edition, 2014.

[PMSN15] Geoffrey Pascoe, Will Maddern, Alexander D. Stewart, and Paul Newman. Farlap: Fast robust localisation using appearance priors. In *IEEE International Conference on Robotics and Automation (ICRA), Seattle, USA*, pages 6366–6373, 2015.

[PPJ+18] Fabian Poggenhans, Jan-Hendrik Pauls, Johannes Janosovits, Stefan Orf, Maximilian Naumann, Florian Kuhnt, and Matthias Mayr. Lanelet2: A high-definition map framework for the future of automated driving. In *IEEE International Conference on*

Intelligent Transportation Systems (ITSC), Maui, USA, pages 1672–1679, 2018.

[RBB09] Radu B. Rusu, Nico Blodow, and Michael Beetz. Fast point feature histograms (fpfh) for 3d registration. In *IEEE International Conference on Robotics and Automation (ICRA), Kobe, Japan*, pages 3212–3217, 2009.

[RFBH16] M. A. A. Rajput, Eugen Funk, Anko Börner, and Olaf Hellwich. Recursive total variation filtering based 3d fusion. In *International Converence on Signal Processing and Multimedia Applications (SIGMAP), Lisbon, Portugal*, pages 72–80, 2016.

[RIG15] John G. Rogers III and Jason M. Gregory. *Have I Been Here Before? A Method for Detecting Loop Closure With LiDAR*. Technical Report, U.S. Army Research Laboratory, Computational and Information Sciences Directorate, Adelphi, USA, 2015.

[RSM+16] German Ros, Laura Sellart, Joanna Materzynska, David Vazquez, and Antonio M. Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, USA*, pages 3234–3243, 2016.

[RVRK16] Stephan R. Richter, Vibhav Vineet, Stefan Roth, and Vladlen Koltun. Playing for data: Ground truth from computer games. In *European Conference on Computer Vision (ECCV), Amsterdam, Netherlands*, pages 102–118, 2016.

[Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers, San Francisco, USA, 1st edition, 2006.

[SBCI17] Miroslava Slavcheva, Maximilian Baust, Daniel Cremers, and Slobodan Ilic. Killingfusion: Non-rigid 3d reconstruction without correspondences. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, USA*, pages 5474–5483, 2017.

[SCD+06] Steven M. Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), New York, USA*, pages 519–528, 2006.

[SDLK18] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics, Zurich, Switzerland*, pages 621–635, 2018.

[SGB10] Bastian Steder, Giorgio Grisetti, and Wolfram Burgard. Robust place recognition for 3d range data based on point features. In *IEEE International Conference on Robotics and Automation (ICRA), Anchorage, USA*, pages 1400–1405, 2010.

[SGST13] Sunando Sengupta, Eric Greveson, Ali Shahrokni, and Philip H. S. Torr. Urban 3d semantic modelling using stereo vision. In *IEEE International Conference on Robotics and Automation (ICRA), Karlsruhe, Germany*, pages 580–585, 2013.

[SKSC13] Frank Steinbrücker, Christian Kerl, Jürgen Sturm, and Daniel Cremers. Large-scale multi-resolution surface reconstruction from rgb-d sequences. In *IEEE International Conference on Computer Vision (ICCV), Sydney, Australia*, pages 3264–3271, 2013.

[SLKS17] Marc Sons, Martin Lauer, Christoph G. Keller, and Christoph Stiller. Mapping and localization using surround view. In *IEEE Intelligent Vehicles Symposium (IV), Los Angeles, USA*, pages 1158–1163, 2017.

[Smi61] Oliver K. Smith. Eigenvalues of a symmetric 3×3 matrix. *Communications of the ACM*, 4(4):168, 1961.

[SS18] Marc Sons and Christoph Stiller. Efficient multi-drive map optimization towards life-long localization using surround view. In *IEEE International Conference on Intelligent Transportation Systems (ITSC), Maui, USA*, pages 2671–2677, 2018.

[SSC14] Frank Steinbrücker, Jürgen Sturm, and Daniel Cremers. Volumetric 3d mapping in real-time on a cpu. In *IEEE International Conference on Robotics and Automation (ICRA), Hong Kong, China*, pages 2021–2028, 2014.

[SSHP15] Thomas Schöps, Torsten Sattler, Christian Häne, and Marc Pollefeys. 3d modeling on the go: Interactive 3d reconstruction of large-scale scenes on mobile devices. In *International Conference on 3D Vision (3DV), Lyon, France*, pages 291–299, 2015.

[Sze11] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer, London, UK, 1st edition, 2011.

[VLGP16] Michiel Vlaminck, Hiep Luong, Werner Goeman, and Wilfried Philips. 3d scene reconstruction using omnidirectional vision and lidar: A hybrid approach. *IEEE Sensors*, 16(11):art. no. 1923, 2016.

[WHLS16] Hermann Winner, Stephan Hakuli, Felix Lotz, and Christina Singer. *Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort*. Springer, Cham, Switzerland, 1st edition, 2016.

[WLSM+15] Thomas Whelan, Stefan Leutenegger, Renato F. Salas-Moreno, Ben Glocker, and Andrew J. Davison. Elasticfusion: Dense slam without a pose graph. In *Robotics: Science and Systems (RSS), Rome, Italy*, 2015.

[WMG14] Michael Waechter, Nils Moehrle, and Michael Goesele. Let there be color! large-scale texturing of 3d reconstructions. In *European Conference on Computer Vision (ECCV), Zurich, Switzerland*, pages 836–850, 2014.

[WMK+12] Thomas Whelan, John McDonald, Michael Kaess, Maurice Fallon, Hordur Johannsson, and John J. Leonard. Kintinuous: Spatially extended kinectfusion. In *Robotics: Science and Systems (RSS), Workshop on RGB-D: Advanced Reasoning with Depth Cameras, Sydney, Australia*, 2012.

[WWL16] Hao Wang, Jun Wang, and Wang Liang. Online reconstruction of indoor scenes from rgb-d streams. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, USA*, pages 3271–3279, 2016.

[WWZ16] Hongzhi Wu, Zhaotian Wang, and Kun Zhou. Simultaneous localization and appearance estimation with a consumer rgb-d camera. *IEEE Transactions on Visualization and Computer Graphics*, 22(8):2012–2023, 2016.

[WZ15] Hongzhi Wu and Kun Zhou. Appfusion: Interactive appearance acquisition using a kinect sensor. *Eurographics Computer Graphics Forum*, 34(6):289–298, 2015.

[XKSG16] Jun Xie, Martin Kiefel, Ming-Ting Sun, and Andreas Geiger. Semantic instance annotation of street scenes by 3d to 2d label

transfer. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, USA*, pages 3688–3697, 2016.

[XLL+10] Lin Xu, Eric Li, Jianguo Li, Yurong Chen, and Yimin Zhang. A general texture mapping framework for image-based 3d modeling. In *IEEE International Conference on Image Processing, Hong Kong, China*, pages 2713–2716, 2010.

[YGS17] Zhenfei Yang, Fei Gao, and Shaojie Shen. Real-time monocular dense mapping on aerial robots using visual-inertial fusion. In *IEEE International Conference on Robotics and Automation (ICRA), Singapore*, pages 4552–4559, 2017.

[YXC+18] Fisher Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, Vashisht Madhavan, and Trevor Darrell. Bdd100k: A diverse driving video database with scalable annotation tooling. https://arxiv.org/pdf/1805.04687v2.pdf, last retrieved 2020-04-24, 2018.

[ZK14] Qian-Yi Zhou and Vladlen Koltun. Color map optimization for 3d reconstruction with consumer depth cameras. *ACM Transactions on Graphics (TOG)*, 33(4):art. no. 155, 2014.

[ZK15] Qian-Yi Zhou and Vladlen Koltun. Depth camera tracking with contour cues. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, USA*, pages 632–638, 2015.

[ZLS+14] Julius Ziegler, Henning Lategahn, Markus Schreiber, Christoph G. Keller, Carsten Knöppel, Jochen Hipp, Martin Haueis, and Christoph Stiller. Video based localization for bertha. In *IEEE Intelligent Vehicles Symposium (IV), Dearborn, USA*, pages 1231–1238, 2014.

[ZPK18] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3d: A modern library for 3d data processing. https://arxiv.org/pdf/1801.09847v1.pdf, last retrieved 2020-04-24, 2018.

[ZS14] Ji Zhang and Sanjiv Singh. Loam: Lidar odometry and mapping in real-time. In *Robotics: Science and Systems (RSS), Berkeley, USA*, 2014.

[ZSG+18] Michael Zollhöfer, Patrick Stotko, Andreas Görlitz, Christian Theobalt, Matthias Nießner, Reinhard Klein, and Andreas Kolb.

State of the art on 3d reconstruction with rgb-d cameras. *Eurographics Computer Graphics Forum*, 37(2):625–652, 2018.

[ZZZL13] Ming Zeng, Fukai Zhao, Jiaxiang Zheng, and Xinguo Liu. Octree-based fusion for realtime 3d reconstruction. *Graphical Models*, 75(3):126–136, 2013.

Publications by the Author

[1] Tilman Kühner and Julius Kümmerle. Extrinsic multi sensor calibration under uncertainties. In *IEEE International Conference on Intelligent Transportation Systems (ITSC), Auckland, New Zealand*, pages 3921–3927, 2019.

[2] Tilman Kühner and Julius Kümmerle. Large-scale volumetric scene reconstruction using lidar. In *IEEE International Conference on Robotics and Automation (ICRA), Paris, France*, pages 6261–6267, 2020.

[3] Tilman Kühner, Sascha Wirges, and Martin Lauer. Automatic generation of training data for image classification of road scenes. In *IEEE International Conference on Intelligent Transportation Systems (ITSC), Auckland, New Zealand*, pages 1097–1103, 2019.

[4] Julius Kümmerle and Tilman Kühner. Fast and precise visual rear axle calibration. In *IEEE International Conference on Intelligent Transportation Systems (ITSC), Auckland, New Zealand*, pages 3942–3947, 2019.

[5] Julius Kümmerle, Tilman Kühner, and Martin Lauer. Automatic calibration of multiple cameras and depth sensors with a spherical target. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Madrid, Spain*, pages 1–8, 2018.

[6] Julius Kümmerle, Marc Sons, Fabian Poggenhans, Tilman Kühner, Martin Lauer, and Christoph Stiller. Accurate and efficient self-localization on roads using basic geometric primitives. In *IEEE International Conference on Robotics and Automation (ICRA), Montreal, Canada*, pages 5965–5971, 2019.

[7] Sascha Wirges, Björn Roxin, Eike Rehder, Tilman Kühner, and Martin Lauer. Guided depth upsampling for precise mapping of urban environments. In *IEEE Intelligent Vehicles Symposium (IV), Los Angeles, USA*, pages 1140–1145, 2017.

Supervised Theses

Wei Ma, 3D Object Reconstruction using Level Sets, Master's Thesis, Institute of Measurement and Control Systems, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2020-02.

Weiyun Chen, 3D Feature Descriptors using Signed Distance Functions, Master's Thesis, Institute of Measurement and Control Systems, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2020-02.

Yawei Jueluo, Mesh Texturing from Camera Images, Master's Thesis, Institute of Measurement and Control Systems, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2020-04.

Zhaoran Wu, Monocular 3D Scene Reconstruction for Localization, Master's Thesis, Institute of Measurement and Control Systems, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2020-10.