RWTH AACHEN UNIVERSITY

Diese Arbeit wurde vorgelegt am
Lehr- und Forschungsgebiet Informatik 8 (Computer Vision)
Fakultät für Mathematik, Informatik und Naturwissenschaften
Prof. Dr. Bastian Leibe

Bachelor Thesis

Generation and Semi-Automatic Annotation of Visual SLAM Reconstructions for 3D Semantic Segmentation

vorgelegt von

Marvin Pförtner
Matrikelnummer: 355431
2018-09-28

Erstgutachter: Prof. Dr. Bastian Leibe
Zweitgutachter: Prof. Dr. Leif Kobbelt

Eidesstattliche Versicherung

Marvin Pförtner	355431
Name	Matrikelnummer

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel

Generation and Semi-Automatic Annotation of Visual SLAM Reconstructions for 3D Semantic Segmentation

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 2018-09-28	
Ort, Datum	Unterschrift

Belehrung:

§ **156 StGB: Falsche Versicherung an Eides Statt**
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ **161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt**

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten dementsprechend.

Die vorstehende Belehrung habe ich zur Kentnis genommen:

Aachen, 2018-09-28	
Ort, Datum	Unterschrift

Contents

Acknowledgements

I would like to thank my supervisor Francis Engelmann for giving me the opportunity to work autonomously on an interesting and highly relevant project which sparked my interest in computer vision. Special thanks go to Alexander Hermans for providing helpful discussions, honest feedback, and advice in desperate times. Also, the advice and guidance given by Prof. Dr. Bastian Leibe was invaluable in the completion of this thesis. I appreciated the help of István Sárándi and Jonathon Luiten who tested my annotation tool and made valuable suggestions for improvements. Moreover, I wish to thank Jonas Schult for his relentless encouragement and optimism. Special thanks also go to Inge Sörensen and Philine Witzig for proofreading my thesis thoroughly. Further thanks to all members of the RWTH computer vision group for the open and friendly work environment. Finally, I would like to say thank you to my family and friends who have constantly supported me over the past 3 years.

Thank you all!

1 Introduction

Automatically gaining knowledge about the semantics of a real-world scene depicted by visual data has long been one of the core goals in computer vision. In autonomous driving and driver assistance, arguably the most prominent practical applications of scene understanding, a vehicle must be able to perceive and understand its environment robustly in a multitude of scenarios in order to navigate the world in a secure fashion. For example, it is crucial that the vehicle safely identifies the boundaries of the road so that it does not depart its lane unexpectedly. Additionally, possible obstacles in the planned trajectory of the car such as people, other vehicles, buildings and many more must be recognized and naturally be avoided. While this can technically be achieved in a trivial fashion by means of proximity sensors such as laser range scanners or radar, these devices might also react to other objects such as leaves or pieces of plastic waste that pass the trajectory. In these cases it is vital to actively ignore the obstacles as an emergency stop would be both dangerous and largely unnecessary. However, differentiating between the two cases, again requires robust ways to distinguish the obstacles taking into account their semantic context in the scene. Due to their high practical relevance, autonomous driving and driver assistance will be used as somewhat canonical examples of practical scenarios throughout this thesis. Nevertheless, there are other applications of scene understanding such as robotics (which can be seen as a strict generalization of autonomous driving) and medical imaging. Hence, we conclude that scene understanding is of tremendous practical importance.

Nowadays the problem of scene understanding is typically phrased in terms of obtaining a semantic segmentation of an image or a point cloud and the most promising methods tackling this task usually apply deep learning techniques. Unfortunately, deep learning methods are inevitably data-hungry, requiring vast amounts of training data to perform well. While a large variety of training data is available for 2D semantic segmentation, virtually all corpora for 3D semantic segmentation are recorded in indoor scenarios. Yet, autonomous driving and driver assistance are inherently bound to outdoor scenarios and it is obvious that

we need outdoor training data to apply the deep learning methods in this case. This is why this thesis investigates into a largely unexplored strategy to generate training data for 3D semantic segmentation in outdoor scenarios.

1.1 2D Semantic Segmentation

It has already been mentioned that a common way to phrase the relatively vague task of automatic scene understanding is by means of semantic segmentation. Even though this thesis focuses on generating training data for 3D semantic segmentation, we chose to review important aspects of 2D semantic segmentation here. Historically, 2D semantic segmentation preceded 3D semantic segmentation. Additionally, we apply a 2D semantic segmentation method in our pipeline.

In 2D *semantic segmentation*, a class[1] from a predefined set $\mathbb{K}$ of K semantic classes has to be assigned to each individual pixel of an image. Hence, semantic segmentation can be interpreted as per-pixel image classification. Common examples of such semantic classes include road, sidewalk, tree, building, sky, human, dog, chair and table.

1.1.1 Methods

Nearly all semantic segmentation methods rely on some kind of classifier which outputs confidence scores over the possible class predictions for each individual pixel. The input to these classifiers typically consists of features extracted from a region around the pixel to be classified.

In early semantic segmentation, the classifiers were typically chosen to be relatively simple to prevent a high computational complexity. Ranging from class posteriors of *Gaussian mixture model (GMM)* classifiers (e.g. for foreground-background segmentation in [RKB04], or in [SWRC09]) over classification confidence scores of *boosted decision stubs* (e.g. in the *TextonBoost* pipeline [SWRC09]), *randomized decision forests* (e.g. *Texton Forests* [SJC08]) and *support vector machines (SVMs)*, a relatively wide variety of classifiers has been put to use.

In contrast to the relatively local approach of sliding window classification, there are also *holistic* segmentation approaches, i.e. methods that infer the predicted labels by taking into account information from the whole image. To this end, *Markov random fields (MRFs)* or *conditional random fields (CRFs)* (both described in [Mur12, chapter 19]) are usually employed. Both are undirected graphical models, the former modeling the joint distribution $p(\boldsymbol{y}, \boldsymbol{x})$ and the latter describing the conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$ where $\boldsymbol{x}$ are the image features and $\boldsymbol{y}$ the predicted labels, both interpreted as random vectors. The graph structure of the random field explicitly introduces conditional dependences between the predicted labels $\boldsymbol{y}$ of the image (as shown in Figure 1.1) that make label

[1] The terms label and class are used interchangeably in this thesis.

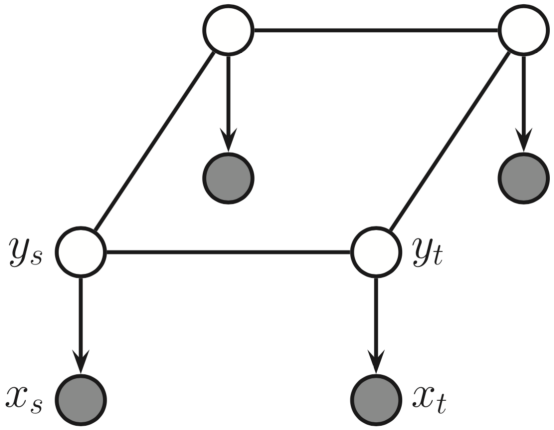


Figure 1.1: The underlying graph structure of an MRF with 4-neighborhood connectivity

changes between neighboring pixels with similar appearance unlikely. Hence, a maximum a posteriori estimate of the labels given the image features will have globally optimal properties. MRFs are used by [RKB04] while [SWRC09, KK11] employ CRFs.

The advent of ever larger scale annotated datasets (see section 1.1.3) and an increasing insight into the mechanics of convolutional neural networks (CNNs) paved the way for deep learning techniques to be applied in semantic segmentation. However, originally, CNN architectures, especially those used in image classification, were engineered towards a fixed input image size. This is due to the fully connected layers at the end of the network which require a fixed size, spatially flattened input vector.

In their seminal work [LSD15] the authors popularized end-to-end training of *fully convolutional networks (FCNs)* for semantic segmentation. The authors enable the use of arbitrary image sizes by replacing fully connected layers with 1x1 convolutions applied to the entire image. The final per-pixel classification used to generate the target segmentation labels can then be obtained as softmax activations over the depth dimension of the final feature volume. Note that the stacked convolutional layers realize the aforementioned sliding-window classifier scheme in a very efficient manner because pre-computed features are shared between different locations of the sliding-window. It is common to convert existing CNN architectures (such as *AlexNet* [KSH12] and *VGG-16* [SZ15] in [LSD15]) into an FCN and use it as a backbone in a semantic segmentation architecture. However, these architectures downsample the input in the forward pass so that a semantic label cannot be directly assigned to each input pixel. Downsampling of the image is, among other reasons, used to increase the receptive field of filters while keeping the memory requirements of deep networks at bay and is thus not to be discarded. There exist several strategies to assign the semantic labels at input scale. In [LSD15], *transpose* or *fractionally strided* convolutions, combined

with skip connections from higher resolution stages of preceding layers are used to perform learnable upsampling.

Extending the work presented in [LSD15], approaches like *SegNet* [BKC17] and *U-Net* [RFB15] (for biomedical data) comprise fully convolutional networks in an *encoder-decoder* fashion. The encoder acts as a feature extractor transforming the image into a downsampled feature volume, while the decoder performs multiple stages of (usually learnable) upsampling of the feature volume into the target size in order to predict the labels. U-Net adopts the upsampling strategy by [LSD15] (transpose convolutions and skip connections). SegNet on the other hand memorizes the maximum indices during max-pooling in the encoder and uses these positions to undo the max-pooling operations later while upsampling. In both cases, the process of down- and upsampling is fully symmetric (which is in fact required by the max-unpooling operation in SegNet), and several convolutional layers are applied after the upsampling steps to make the predictions denser.

As opposed to the upsampling approach presented above, *atrous* or *dilated convolutions* allow to perform dense feature extraction without downsampling. In principle, atrous convolutions are regular convolutions with nearest neighbor upsampled filters where the missing values are filled with zeros [CPK+18]. Thus, with the same number of parameters and practically the same computational cost, the receptive field of the network increases substantially. The *DeepLab* system in its various iterations (v1 [CPK+15], v2 [CPK+18], v3 [CPSA17] and v3+ [CZP+18]) makes heavy use of atrous convolutions for various tasks (e.g. to integrate multi-scale information in the *atrous spatial pyramid pooling (ASPP)* module from version 2 onwards). The versions 1 and 2 of DeepLab also apply a fully connected CRF [KK11] to perform holistic image segmentation which was omitted in later versions. DeepLab v3+ will be covered in more detail in section 4.3.

1.1.2 Evaluation Metrics[2]

The performance of semantic segmentation approaches is typically assessed by measuring the deviation between the algorithm's prediction and a ground truth segmentation. Let $\mathbb{Y}_k$ be the set of pixels with predicted label k and $\mathbb{T}_k$ the set of pixels with target label k. A simple measure to quantify the deviation is given by the *(mean) accuracy*

$$\text{mAcc} = \frac{1}{K} \sum_{k \in \mathbb{K}} \frac{|\mathbb{T}_k \cap \mathbb{Y}_k|}{|\mathbb{Y}_k|} \in [0, 1] \tag{1.1}$$

[2] see e.g. [EEVG+15, COR+16, LSD15]

where the ratio in the sum is often referred to as the *per-class accuracy* which measures the fraction of true positives among all predicted labels per class. Decomposition of the denominator yields

$$|\mathbb{Y}_k| = \underbrace{|\mathbb{T}_k \cap \mathbb{Y}_k|}_{\text{true positive}} + \underbrace{|\mathbb{Y}_k \setminus \mathbb{T}_k|}_{\text{false positive}}$$

which shows the implicit penalty of false positives.

The most widely used performance measure employed in semantic segmentation is the *(mean) intersection-over-union* (or *mean IoU*) which is sometimes also referred to as the *Jaccard index* [COR+16]

$$\text{mIoU} = \frac{1}{K} \sum_{k \in \mathbb{K}} \frac{|\mathbb{T}_k \cap \mathbb{Y}_k|}{|\mathbb{T}_k \cup \mathbb{Y}_k|} \in [0, 1]. \tag{1.2}$$

As before, the ratio in the sum is often referred to as the *per-class intersection-over-union* which measures the similarity of the two sets $\mathbb{T}_k$ and $\mathbb{Y}_k$. Again, the size of the union in the denominator can be decomposed as follows

$$|\mathbb{T}_k \cup \mathbb{Y}_k| = \underbrace{|\mathbb{T}_k \cap \mathbb{Y}_k|}_{\text{true positive}} + \underbrace{|\mathbb{T}_k \setminus \mathbb{Y}_k|}_{\text{false negative}} + \underbrace{|\mathbb{Y}_k \setminus \mathbb{T}_k|}_{\text{false positive}}.$$

Hence, in contrast to per-class accuracy, per-class IoU also penalizes false negatives.

1.1.3 Datasets

Virtually every semantic segmentation approach relies heavily on machine learning techniques and thus requires training data to be applied. Over the years, several large-scale datasets with annotated training and test images have been published. Some of the most influential ones among those will be presented in this section.

PASCAL VOC The *PASCAL Visual Object Classes (VOC)* [EEVG+15] challenge includes a dataset of annotated images for the computer vision tasks *classification*, *detection* and *semantic segmentation* (and also *action classification* and *person layout*) with labels from a set of 20 diverse classes from the categories vehicles, household, animals and other. The semantic segmentation dataset contains an additional background class. The publicly available dataset for training and validation comprises 2913 annotated images (in the final iteration of the dataset for PASCAL VOC 2012). The evaluation measure for semantic segmentation is per-class IoU.

Cityscapes *Cityscapes* [COR+16] aims at providing a diverse dataset of street scenes and was recorded by cameras mounted onto a moving car while driving through 50 cities in central Europe (mostly Germany). Complementing the images, the corresponding right stereo images, GPS positions, odometry data and outdoor temperatures are available. Annotations are provided for the tasks of *semantic* and *instance segmentation*. Among the images of the dataset, 5,000 have fine and 20,000 have coarse annotations. Cityscapes defines a set of 30 semantic classes in 8 categories (flat, human, vehicle, construction, object, nature, sky and void) of which only 19 are used in evaluation. The methods' performance is assessed by mIoU on both class and category level and additionally by a modified IoU metric that ought to correct for the bias of the IoU measure towards semantic classes that typically occupy many pixels.

KITTI The *KITTI Vision Benchmark Suite* [GLU12] is mainly targeted at automotive scenarios, providing data for many common tasks associated with autonomous driving (such as visual odometry, stereo matching, and object detection). Over the years, there have been several extensions of the benchmarks. The original dataset did not contain semantic segmentation dataset which is why people published unofficial annotations for parts of the dataset. Recently, 400 images from the stereo and flow benchmark 2015 have been annotated and added as an official semantic segmentation benchmark. The dataset uses the same classes and evaluation metrics as Cityscapes does.

Mapillary Vistas *Mapillary Vistas* [NOBK17] is by far one of the largest datasets with semantic segmentation (and instance segmentation) annotations. It is to be considered extremely diverse, covering 6 continents, all sorts of weather conditions, seasons, and times of day. Additionally many different high-resolution cameras have captured the images from varying viewpoints. The dataset comprises 25,000 annotated images with labels from 66 classes. mIoU is employed as an evaluation metric.

1.2 3D Semantic Segmentation

Being able to identify semantic regions in images is already an important step towards automatic scene understanding. However, robots such as autonomous cars are agents in a three dimensional world whose structure cannot be fully captured in a 2D image. In light of this, it is only natural to extend the task of semantic segmentation to 3D data.

In a single image, depth is inherently absent and hence, different representations of a 3D scene must be chosen, among which the *point cloud* (sometimes also *point set* [QSMG17]) is popular in the vision community. 3D data that appear in robotics are often relatively sparse and thus point clouds offer several advantages

over triangle meshes or voxel-based representations among which their compact representation and robustness to geometric noise are the most important ones. Additionally, depth images are also commonly used to represent to represent 3D data.

The task definition from Section 1.1 generalizes trivially in that now every point of a point cloud has to be assigned a semantic label. Most 2D semantic segmentation methods, especially those based on feature extraction by means of convolutions, presented in Section 1.1.1 strongly rely on the spatial regularity of the input data (e.g. an image grid structure) while point clouds are inherently unstructured. This suggests that the extension of 2D semantic segmentation approaches is highly nontrivial.

1.2.1 Methods

A variety of wildly different approaches used to deal with 3D data in semantic segmentation has been proposed. Broadly speaking, there are those that integrate 3D information into 2D semantic segmentation and those that directly aim to solve the task of 3D semantic segmentation. Additionally, there are approaches that directly learn to operate on unstructured 3D data while others explicitly build a structured representation of a point cloud. While there are many traditional methods applied to the field, recent developments have strongly popularized the use of deep learning methods which is why we will only focus on those in this overview.

[GGAM14] use depth images from RGB-D sensors and compute a so called *HHA* encoding which unifies the height above the ground plane with the horizontal disparity and the angle to the gravity vector in a three channel image. According to their experiments this feature representation is superior to using the depth images when applying a CNN.

Several methods such as [HY16] apply networks using 3D convolutions on voxelized versions of a point cloud.

In order to tackle characteristic issues with voxelized representations, the seminal work of *PointNet* [QSMG17] and its extensions (for instance [QYSG17, EKHL17]) use a network that can directly cope with unstructured 3D data by learning a function that is invariant to permutations of the input points.

Approaches like [QLJ+17,WSL+18] explicitly introduce structure to a 3D point cloud using a neighborhood graph (mostly k-NN graphs). [QLJ+17] use the depth image from RGB-D data to construct a k-NN graph with subsets of the 3D points as vertices. By means of an RNN, they then perform several steps of message passing similar to that employed in loopy belief propagation on the nodes of the graph. The message passing is used to compute an embedding per point which can then be used to augment the 2D segmentation of the image. [WSL+18] on the other hand use k-NN graphs on the points in different layers of the network to compute *edge features* using the adjacent point features. Using

the edge features, they define the *EdgeConv* operator which accumulates the neighborhood information per point and feature dimension e.g. by summing or maximizing over all the incident edge features.

There are many other interesting approaches to 3D semantic segmentation. However, for the sake of brevity, we restrict our survey to the ones mentioned above.

1.2.2 Evaluation Metrics

The evaluation metrics widely used in 3D semantic segmentation coincide with those used in 2D semantic segmentation. [QSMG17, QYSG17, EKHL17, QLJ+17, WSL+18] all report mIoU, mAcc, or both, among other metrics. Also, [HSL+17] evaluate benchmark submissions using mIoU and per-class IoU (and overall accuracy).

1.2.3 Datasets

NYUDv2 The *NYU Depth V2* [SHKF12] dataset consists of 1449 RGB-D images recorded in 464 indoor scenes with dense semantic and instance labels. The images were acquired using a *Microsoft Kinect v1* active stereo camera.

SUN3D *SUN3D* [XOT13] is also an indoor dataset of video streams recorded by an RGB-D camera (the *Asus Xtion PRO LIVE* in this case) in 254 different areas of 41 buildings. However, instead of just providing a few annotated images as is in NYUDv2, SUN3D comprises annotations and camera poses for all frames within the video sequence. Hence, a multi-view reconstruction of the scene rather than just one view with depth information can be fed to the segmentation algorithm. At the time of writing, 8 of the 415 sequences are fully annotated.

SUN RGB-D Building upon the work of [SHKF12, XOT13] and several other RGB-D datasets, *SUN RGB-D* [SLX15] provides a large-scale indoor dataset of 10,335 annotated RGB-D images. It combines images from several existing datasets with new ones collected exclusively for the dataset. Different RGB-D cameras (namely *Intel Realsense*, *Asus Xtion*, *Microsoft Kinect v1/v2*) have been used to collect the data.

S3DIS The *Stanford Large-Scale 3D Indoor Spaces (S3DIS)* [ASZ+16] dataset contains annotated point cloud reconstructions of six indoor areas from three different buildings which were captured using a Matterport camera. Labels from one of the 13 semantic classes are provided on point and instance level for over 600,000,000 points. The dataset was extracted from the *2D-3D-S* [ASZS17] dataset which provides corresponding RGB and depth images (over 70,000) and camera poses.

Matterport 3D Further pushing the limits of indoor 3D semantic segmentation datasets, *Matterport 3D* [CDF+17] provides 194,400 RGB-D images capturing several panoramic views of 90 different buildings with camera poses relating the different views within a building. Semantic and instance labels are provided from 40 semantic classes. The 3D data is acquired using a rotating rig that comprises 3 RGB and 3 depth cameras with different orientations.

ScanNet Another popular indoor RGB-D dataset is *ScanNet* [DCS+17]. It provides 2,492,518 RGB-D frames recorded in 1513 indoor locations spanning a total area of 34,453 square meters. These RGB-D frames are densely annotated on the instance level using one of 20 semantic labels. The annotation process was crowdsourced on Amazon Mechanical Turk. Additionally, camera poses, surface reconstructions and CAD models aligned with the furniture and other objects in the room are provided. In contrast to the other indoor datasets, ScanNet does not rely on specialized RGB-D hardware but uses an off-the-shelf iPad Air 2 with an additional depth sensor attached. This hardware choice in conjunction with a simple recording interface opens the recording process to novice users.

Virtual KITTI The *Virtual KITTI* [GWCV16] dataset comprises 50 photo-realistic synthetic videos of urban street scenes with depth images, semantic and instance annotations from 13 semantic classes, and camera poses. 5 of those sequences are virtual reconstructions of the original KITTI (see section 1.1.3) sequences. [EKHL17] provide 5 of the sequences converted to 3D point clouds with several projective errors fixed.

Semantic3d.net The *Large-Scale Point Cloud Classification Benchmark* or *semantic3d.net* dataset [HSL+17] differs to the aformentioned ones insofar as it provides scans of 30 large outdoor scenes from a static laser scanner. In total the dataset comprises over 4 billion colored 3D points with semantic ground-truth labels from 8 classes.

1.3 Problem Statement

Sections 1.1.1 and 1.2.1 gave a brief representative survey of the methods applied in both 2D and 3D semantic segmentation.

Delving into the results obtained from 2D semantic segmentation methods on the available benchmarks, the algorithms achieve impressive accuracy and even decent generalization to other data which, among other things, can be attributed to the availability of diverse large-scale datasets.

However, as mentioned before, 3D semantic segmentation, specifically for urban or more general outdoor scenarios, is subject to a severe lack of diverse large-scale datasets for evaluation and testing [EKHL17]. This is especially detrimental to

the deep learning models which require vast amounts of training data [HSL+17, COR+16, GBC16]. Furthermore, most available 3D datasets are recorded with RGB-D cameras which can be interpreted as a strong bias in the training set sample drawn from the data-generating distribution.

Taking all the above into account, it becomes evident that there is a strong need for annotated training data for semantic segmentation collected in outdoor scenarios (e.g. street scenes, as these are of great interest in autonomous driving) preferably recorded with a method other than RGB-D cameras. Hence, this thesis aims at developing a system to acquire and annotate new training data for 3D semantic segmentation, preferably, but not exclusively, recorded in outdoor scenarios.

1.4 Structure

Our solution to the problem addressed in this thesis is threefold. In chapter 2 our recording rig and software to generate and augment the 3D data will be described in depth. Additionally, our choice of data source will be justified. Chapter 3 presents our custom annotation tool which leverages the fact that the 3D data generated by our recording rig originates from sequences of 2D images. However, it has to be noted that this tool not only supports the data generated by our recording rig but it generalizes to arbitrary temporal sequences of point cloud data. Labeling 3D data is generally believed to be a tedious and time-consuming task which is why chapter 4 presents our strategy to automatically initialize the semantic labels by means of an existing method for 2D semantic segmentation.

1.5 Notation and Conventions

We denote scalars by light, italic letters (x), vectors by lowercase, bold letters ($\boldsymbol{x}$), and matrices by uppercase bold letters ($\boldsymbol{X}$).

Camera poses are expressed as elements of the Lie group of 3D rigid body transformations $\boldsymbol{P} \in \mathrm{SE}(3) \subseteq \mathbb{R}^{4\times4}$ transforming camera coordinates to world coordinates. The corresponding Lie algebra of twists is denoted by $\mathfrak{se}(3)$. The exponential map and its inverse are denoted by $\exp_{\mathfrak{se}(3)}\colon \mathfrak{se}(3) \to \mathrm{SE}(3)$ and $\log_{\mathrm{SE}(3)}\colon \mathrm{SE}(3) \to \mathfrak{se}(3)$, respectively. We refer to the inverse of a pose matrix (i.e. a matrix that transforms from world coordinates to camera coordinates) as a *view matrix* $\boldsymbol{T} = \boldsymbol{P}^{-1} \in \mathrm{SE}(3)$. The depth of a point in camera coordinates is denoted by $d \in \mathbb{R}_{>0}$ and the corresponding inverse depth is denoted by $\rho = d^{-1} \in \mathbb{R}_{>0}$. The projective mapping of a camera is denoted by the function $\Pi_{\boldsymbol{c}}\colon \mathbb{R}^3 \to \mathbb{R}^2$. With a slight abuse of notation, we define the unprojection mapping by $\Pi_{\boldsymbol{c}}^{-1}\colon \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^3$ where the second argument is the depth of the point in camera coordinates. In both cases, $\boldsymbol{c}$ contains the intrinsic calibration of the camera model. Unless otherwise stated, the camera model used in this thesis is

the *pinhole camera model* where $\boldsymbol{c} = \boldsymbol{K} \in \mathrm{GL}(3) \subseteq \mathbb{R}^{3\times 3}$ is the *camera calibration matrix*. By Π_0 we denote the standard projection, i.e. the function $\Pi_0 \colon \mathbb{R}^3 \to \mathbb{R}^2, \boldsymbol{x} \mapsto \frac{1}{x_3}\begin{pmatrix} x_1 & x_2 \end{pmatrix}^T$ performing a division by the z-component of the vector. Images are modeled by as functions $I \colon \Omega \subseteq \mathbb{R}^2, \boldsymbol{p} \mapsto \mathbb{R}^D$ where $D \in \{1, 3\}$ (greyscale or color images). Point clouds are represented as sets $\mathbb{P} \subseteq \mathbb{R}^3$.

The (*measurement*) *uncertainty* of a value x is expressed as a standard deviation σ_x (i.e. assuming a Gaussian error distibution).

2 3D Data Acquisition

Over the years, several methods of capturing a 3D representation of a real world scene have emerged. Among those, a calibrated stereo rig has several important practical advantages such as low cost, low energy consumption, and passive sensing. Unfortunately, these advantages come at the price of inferior accuracy compared to LiDAR scanners or RGB-D cameras and the inability to obtain depth in untextured areas. Recent advances in the design of visual SLAM systems gave rise to powerful algorithms that improve the visual quality of 3D reconstructions acquired by means of stereo cameras by a large margin. Hence, we propose visual SLAM reconstructions as a data source for 3D semantic segmentation. On top of that, we elaborate upon the design decisions made while engineering our recording rig and we present our software pipeline used to generate 3D point clouds from the stereo image sequences.

2.1 Possible Data Sources

When it comes to acquiring 3D data, the most popular methods nowadays are arguably *RGB-D cameras*, *LiDAR* and *stereo camera setups*. Broadly speaking, one usually distinguishes between *active* and *passive sensors*. The former emit some form of energy into the scene and rely on its reflection to sense depth while the latter solely make use of the energy that is already present in the scene [RN10]. In the following, we will give a brief survey on the three methods mentioned above including an evaluation of their strengths and weaknesses for our particular use case.

2.1.1 RGB-D Cameras[1]

The popularity of RGB-D cameras in indoor scenarios can be observed by the large fraction of 3D datasets presented in section 1.2.3 that use RGB-D sensors to

[1] this section is largely inspired by [SLK15]

record the data. Several examples of RGB-D cameras, namely *Microsoft Kinect* v1 and v2, *Asus Xtion*, *Intel RealSense* and the *Matterport camera* have already been mentioned. In general, RGB-D cameras are active sensors using one of two main depth sensing technologies.

The first one of those is often referred to as *structured-light* depth sensing. A projector projects a light pattern (usually infrared) with known structure onto the scene. A camera captures the reflection of this light pattern and uses the distortion of the known structure in order to triangulate a dense depth map of the scene. Examples of such cameras include Microsoft Kinect v1, Asus Xtion, Intel Realsense and the Matterport camera.

Secondly, a more recent depth sensing technology gave rise to so-called *time-of-flight (ToF)* cameras. Here, the core idea is to emit modulated light waves (also mostly infrared) into the scene which, after reflection, are again picked up by a sensor and correlated with the modulation pattern. The resulting phase shift combined with knowledge of the constant speed of light can then be used to obtain dense depth maps at each sensor pixel. The Kinect v2 is an example of such a sensor.

An advantage of RGB-D cameras is the density of their depth measurements, along with the possibility to run this approach at high frame rates. Moreover, in many scenarios the measurements are less noisy than e.g. stereo cameras.

The main downside of these approaches is that it is difficult (and mostly impossible for structured-light sensing) to apply them in outdoor scenarios. This is due to the fact that sunlight and other sources of light waves contain infrared light making it harder to detect the pattern or reflected modulated light against the background radiation.

Also, the sensing range of most RGB-D cameras currently available is relatively limited which, again, makes it difficult to apply them in scenarios such as autonomous driving.

In principle, the following two problems appear among most active sensors.

Multiple RGB-D cameras might interfere in each other's measurements as they all make use of the same wavelengths. This is especially critical in autonomous driving as one must assure that the perceptive systems function even if there are multiple cars in the same street.

Surfaces that don't reflect the emitted light sufficiently well can impose a problem onto the camera as the depth cannot be measured without reflections. Examples of such surfaces include (semi-)transparent and strongly scattering materials, and, highly specular surfaces that reflect the light into a different direction.

2.1.2 LiDAR

Especially in the robotics community, so called *Light Detection and Ranging (LiDAR)* sensors have reached great popularity. LiDAR is an active sensing technology with a working principle similar to that of ToF cameras. The sensor

emits beams of laser light which are reflected in the scene. The reflected light is then processed in a similar fashion as in the case of ToF cameras to obtain depth measurements. However, in contrast to ToF cameras, LiDAR scans are typically sparse, because each of the laser beams only generates one depth sample. To obtain scans with a large field of view, a rotating mirror is typically used to redirect the laser beams in different directions. Also, instead of emitting just one laser beam at a time, an array of multiple such beams can be used to increase the density of the scans. Prominent LiDAR sensors in robotics include the *Velodyne HDL-64E* used e.g. in [GLU12] and the *Sick LMS500*.

An important argument in favor of LiDAR scanners is their high accuracy which is higher than the accuracy of the other data sources presented here [HSL+17]. Additionally, LiDAR scanners usually have a very large sensing range and the accuracy of points measured at high distances does typically not differ much from that of closer points.

A major issue of LiDAR sensors is that they only provide depth information and no point colors. In principle, one could obtain color samples from calibrated LiDAR-Camera setups but calibration of such systems is difficult [GLU12] and the color samples might be wrong due to occlusion.

LiDAR sensors also suffer from most common active sensor problems (e.g. those mentioned in Section 2.1.1) Furthermore, at the time of writing, 360 degree LiDAR sensors are relatively expensive.

2.1.3 Stereo Cameras

One of the oldest approaches to obtain depth information of a scene is by means of a calibrated pair of *stereo cameras*, which is related to the way humans perceive depth. This approach attempts to detect corresponding points in two views of a scene (which are typically facing in the same direction with a small horizontal displacement, the *baseline*, between them). These corresponding points can then be used to triangulate the coordinates of the observed point in 3D.

Obviously, stereo cameras (all cameras in fact) are passive sensors. As such they don't inherit the common problems of active sensors such as those presented above.

In terms of hardware, this approach is arguably the simplest among those presented here, as it only requires two cameras in a fixed relative geometric configuration which capture images at the same points in time. Hence, depending on the type of camera used in the rig, it is also relatively cheap. Additionally, it is clear that the 3D information captured with a stereo camera rig will include colors.

Due to the fact that stereo setups inherently need correspondences to compute depth, untextured or largely uniformly textured areas complicate the reconstruction of 3D information substantially. In these situations, depending on the algo-

rithm used to find corresponding points, either wrong or no matches at all might render the 3D reconstruction of the scene practically useless.

Another severe problem of stereo camera systems is the accuracy of the depth measurements which declines quadratically with increasing depth of the observed 3D point. To illustrate this, we first rearrange the equation used to obtain the depth of a 3D point from a disparity map of a rectified image pair:

$$d = \frac{bf}{\delta} \quad \Leftrightarrow \quad \delta = \frac{bf}{d} \tag{2.1}$$

where b is the baseline between the cameras, f is the focal length, and δ is the disparity between the two pixels observing the point. By means of propagation of uncertainty, we estimate the measurement uncertainty of the depth, assuming that the errors of the calibration (f and b) are sufficiently small:

$$\sigma_d = \sqrt{\left(\frac{\partial d}{\partial \delta}\sigma_\delta\right)^2} = \frac{bf}{\delta^2}\sigma_\delta \overset{(2.1)}{=} \frac{d^2}{bf}\sigma_\delta. \tag{2.2}$$

Due to the fact that pixel coordinates are inherently discrete, the uncertainty in the disparity is directly dependent on the image resolution. It is also clearly visible that a larger baseline has a positive influence on the error of the measured depth. This however comes at a price, as a larger baseline entails (for fixed focal lengths) that the fields of view of the two cameras begin to overlap at greater depth which means that small depths might not be measurable anymore.

2.1.4 Discussion

Taking all the above into consideration, we can largely exclude RGB-D cameras as a source of data as they are mostly not suited for outdoor application and don't have a large-enough sensing range. Also, as mentioned in section 1.3, there are already many RGB-D datasets available which introduces significant bias towards a certain type of sensor in the available training data. This leaves LiDAR and stereo camera rigs. Among those we chose to use stereo cameras as a source of 3D data which can be justified by three arguments.

For one, building and calibrating a stereo camera rig is cheap and relatively easy once the details have been figured out. Hence, other people have the ability to recreate our hardware setup and use our software to contribute training data. Collaboration in creating training data is a promising way to exceptionally large-scale diverse datasets.

What is more, stereo cameras are likely to be an important part of many robotic systems in the future (in fact they already are) which means that being able to cope with stereo depth data is an important feature in 3D semantic segmentation.

Also, it is possible to correct the error in the depth estimated by stereo cameras by running of a temporal sequence of stereo images through a visual SLAM system

(see section 2.2). While the 3D reconstructions are far from being as accurate as a LiDAR scan, we suspect that their geometric noise might in fact be beneficial to deep learning algorithms as they generally learn to be robust to noise by introducing them to noisy data.

2.2 Visual SLAM

When recording a sequence of stereo camera images as a video, one has the option to integrate the depth information obtained from the individual stereo images into a larger joint point cloud. In order to perform such an integration, it is crucial to estimate the relative camera poses (relative rotation and translation). Additionally, by means of multiple views of the same scene, the 3D locations of the points can be refined, enhancing the overall quality of the reconstruction.

While offline *structure-from-motion* (SfM) (e.g. [SF16, SZPF16]) approaches solve this problem in a highly accurate fashion, they are unfit to be deployed on a mobile platform for real-time operation. This is due to the fact that they usually require vast amounts of computational resources and computation time, and, more importantly, cannot deal with the continuous stream of images in real-time scenarios but need all the images of the scene to be reconstructed beforehand. Hence, we use a *visual SLAM* system to cope with this problem in a real-time scenario.

As the name implies, the task of *simultaneous localization and mapping* (SLAM) requires an agent (e.g. a robot) to build a map of its environment using sensor data (such as LiDAR, radar, sonar, camera, RGB-D, etc.) and to localize itself within it at the same time [TBF05]. When using cameras as sensors, the task is often referred to as *visual SLAM*. While it is technically possible to apply a visual SLAM system without a known intrinsic camera calibration, there is a vast number of benefits arising from a known calibration (see [HZ03]) which is why we always assume our cameras to be at least intrinsically calibrated. Also, many visual SLAM systems are developed with monocular video sequences in mind. Nevertheless, these systems are often extended to leverage a calibrated stereo camera because the 3D geometry observed in monocular video sequences can only be reconstructed up to an arbitrary scale factor [HZ03].

The majority of SLAM systems assumes that the scene at hand is static (i.e. does not change over time) which is a severe limitation to our use-case. Yet, a large quantity of the 3D data in real-world scenarios will indeed depict static parts of the scene and thus training data of static scenes is still of use.

Over the years, a number of visual SLAM methods have been developed. Among those, two main approaches emerged: *filtering-based* methods and methods based on *bundle adjustment* (BA) [SMD12].

2.2.1 Filtering-Based Methods

In filtering-based SLAM, a mathematical filter such as the *Extended Kalman filter* (EKF) [TBF05] is applied to the estimated 3D geometry and the camera poses. The filter both propagates their measurement uncertainty to future measurements and fuses future observations into the system's state with these uncertainties in mind. Examples of such systems include [DRMS07] (monocular cameras) and [SMS07] (stereo cameras). [SMD12] argue that filtering-based visual SLAM is inferior to visual SLAM based on bundle adjustment as the latter requires less computation time to produce results of equal quality. They also mention that the time complexity of BA-SLAM is linear in the number of 3D points while filtering-based approaches have a cubic time complexity. This is especially important for our use-case, as denser reconstructions are more useful when being used in semantic segmentation. Hence, we do not consider filtering-based SLAM in this thesis but merely mention it for the sake of completeness.

2.2.2 Indirect Methods Based on Bundle Adjustment[2]

More recent visual SLAM systems mostly employ *bundle adjustment*. Given the image coordinates of N image points each observed by at least two of M cameras, bundle adjustment aims to jointly estimate the world coordinates $\boldsymbol{x}_n \in \mathbb{R}^3$ of the points and the camera poses $\boldsymbol{P}_m = \boldsymbol{T}_m^{-1} = \left(\begin{smallmatrix} \boldsymbol{R}_m & \boldsymbol{t}_m \\ \boldsymbol{0} & 1 \end{smallmatrix}\right)^{-1} \in \mathrm{SE}(3)$. Broadly speaking, we attempt to find the world points and camera poses in such a way that the projection of the world points onto the image planes of the cameras deviates as little from the given points as possible. This is traditionally achieved by minimizing the *(geometric) reprojection error*

$$E_{\mathrm{geo}}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \sum_{m \in \mathrm{obs}(n)} d(\boldsymbol{p}_{mn}, \Pi_{\boldsymbol{c}_m}(\boldsymbol{R}_m \boldsymbol{x}_n + \boldsymbol{t}_m)) \tag{2.3}$$

where $\boldsymbol{\theta} = (\boldsymbol{T}_1, \ldots, \boldsymbol{T}_M, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$, $\mathrm{obs}(n) \subseteq \{1, \ldots, M\}$ is the set of cameras that observe point n, $\boldsymbol{p}_{mn} \in \mathbb{R}^2$ are the observed image coordinates of the n-th point in the image of the m-th camera, and $d\colon \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ is a suitable distance metric. This formulation assumes calibrated cameras, i.e. that the intrinsic parameters $\boldsymbol{c}_m$ of the m-th camera are known. Moreover, the geometric reprojection error presented above is just one of many largely equivalent formulations of the same problem.

Note that E_{geo} cannot be directly minimized, as the view matrices $\boldsymbol{T}_m$ are constrained to be rigid-body transformations. A common way to cope with this problem is to optimize using the *twist* representation $\boldsymbol{\xi}_m \in \mathfrak{se}(3)$ of the rigid-body

[2] This section is largely inspired by [Cre16]

motion from the corresponding tangent space (which is a Lie algebra in this case). Substituting the view matrices with the exponential map

$$\boldsymbol{T}_m = \exp_{\mathfrak{se}(3)}(\boldsymbol{\xi}_m)$$

renders the constrained minimization problem unconstrained, as a twist is directly constructed of 6 unconstrained parameters. The problem can then be solved by an optimization algorithm such as the *Gauss-Newton* or *Levenberg-Marquardt* algorithm [ESC14]. More information on Lie groups in multi-view geometry can be found in [MSKS12] and optimization of Lie manifolds is treated in [ESC14].

Depending on the choice of distance metric d, we can interpret the minimization of the geometric reprojection error as a maximum likelihood estimation of the unknown model parameters assuming the measurements are corrupted by a noise distribution (which depends on the choice of d). For example, if we choose $d(\boldsymbol{a}, \boldsymbol{b}) = \|\boldsymbol{b} - \boldsymbol{a}\|_2$, the Euclidean norm, the noise distribution is assumed to be Gaussian [HZ03].

The geometric reprojection error is a highly non-convex function and thus a good initialization of the parameters is required for optimization methods to converge to a decent local minimum [HZ03]. Initializing the relative camera pose of a new image with respect to a previous image is often referred to as *tracking* (e.g. [ESC14, MAT17, EKC18]). One way to provide an appropriate initial estimate of these parameters is to apply the *eight-point algorithm* or its generalizations to multiple views to estimate the camera poses and subsequently compute initial estimates of the 3D points by triangulation [HZ03].

Typically, in a real-time scenario, it is too expensive to optimize over all the camera images and points that have been passed to the SLAM system. Instead, one usually only applies bundle adjustment to a relatively small subset of the most recent images, the so-called *keyframes*, to optimize the local geometric consistency. In general, the content of the subsequent images in a video with sufficient frame rate does not change substantially. Hence, the keyframes are selected sparsely once the image contents vary sufficiently which allows for the geometric consistency to be optimized on a larger scale [LLB+15]. To choose whether a frame should be used as a keyframe or not is generally chosen heuristically, e.g. by measuring the content variations of the image.

Above we simply assumed the exact correspondences between the image points that represent the same 3D point (but in different images) to be known. In practice, this is a challenging matching task which requires the system to identify the same point in the image of (ideally) all the cameras that observe it. One common strategy to find these correspondences is to find interest points in the images by means of local feature detectors, extract suitable feature descriptors at the locations of those interest points and match the descriptors between images. However, this approach is error prone and typically produces many wrong matches. Methods like *RANSAC* (Random Sample Consensus) [HZ03] are commonly used to make the initialization robust to outliers. The use of local features

to find corresponding points conveniently introduces a certain invariance to illumination changes, different types of sensor noise, and other problems that arise when using the image intensities [EKC18]. Another strategy uses the optical flow between the video frames to warp the image points of one frame to its corresponding points in subsequent frames [MSKS12]. However, methods using local features seem to prevail.

Due to the fact that the whole optimization process is decoupled from the image data once the feature extraction was performed, one also refers to methods that follow this approach as *indirect* [EKC18].

Popular indirect methods based on bundle adjustment include [KM07,LLB+15, MAT17] (all local feature based).

2.2.3 Direct Methods Based on Bundle Adjustment[3]

Once an indirect method established all the point correspondences, the actual image information is discarded and the parameter estimation is carried out solely using the image coordinates. While using local features is highly robust to several photometric nuisances (as described above), the interest points must generally be selected on distinctive points such as corners. However, edges of all sorts and smoother intensity gradients usually prevail in the images which means that a large part of the image information cannot be exploited by most indirect methods. Also, detecting, extracting, and matching local features introduces computational overhead which can be detrimental in a real-time scenario.

Taking all the above into account, a number of recent approaches to visual SLAM discard the feature extraction step and instead operate on the image intensities directly. To this end, the geometric reprojection error used above is replaced by a *photometric reprojection error*

$$E_{\text{photo}}(\boldsymbol{\theta}) = \sum_{n=1}^{N} \sum_{m \in \text{obs}(n)} d(I_{\boldsymbol{x}_n}, I_m(\Pi_{\boldsymbol{c}_m}(\boldsymbol{R}_m \boldsymbol{x}_n + \boldsymbol{t}_m))) \tag{2.4}$$

where $\boldsymbol{\theta} = (\boldsymbol{T}_1, \ldots, \boldsymbol{T}_M, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N)$ and $I_{\boldsymbol{x}_n}$ is the image intensity of the first observation of point $\boldsymbol{x}_n$. It is also common to replace the intensities of the query points used above by a whole patch of intensities localized around the query point. Again, this is just one popular of many possible formulations of the same problem. Equation 2.4 illustrates that the need for point correspondence is alleviated by means of the image intensities. The point correspondences are now implicitly estimated during optimization along with the other parameters.

Obviously, the photometric error presented above assumes the 2D appearance of the 3D points to be equal in all camera images that observe the point (*brightness constancy assumption*). Various issues such as changing exposure times for auto-

[3]This section is mainly adapted from [NLD11], [ESC14], and [EKC18]

exposure cameras challenge this assumption in practice. Several methods have been proposed to cope with this issue (see Section 2.2.4 for an example).

Also, a good initialization of the camera poses and the 3D structure is presumably even more important than before. Yet, most methods applied in the context of indirect methods (see Section 2.2.2) do not apply to direct methods as, generally, no point correspondences are given. A popular strategy to perform tracking in the direct case is by means of two frame *direct image alignment* (e.g. in [ESC14, EKC18]). Given several 3D points and their projections to one image, direct image alignment optimizes a photometric reprojection error between two frames where the camera pose of the second image is the only free parameter. Again, a good initial estimate for the camera pose is needed. One popular initialization choice is to assume a constant motion model (use the previous camera motion for the next frame). Integrating data from an *inertial measurement unit* (IMU) can provide more accurate initial pose estimates.

When using direct image alignment, the first frame needs to provide several depth estimates in order for the algorithm to work. When working with stereo cameras, it is possible to perform static stereo triangulation to obtain accurate initial depth estimates easily (as in [ESC15]). In the monocular case this process is more challenging and different strategies have been proposed (e.g. in [ESC14]).

Note that it is in fact possible to carry out the optimization of the photometric reprojection error over all pixels of the keyframe images. These methods are usually referred to as *dense* methods. [NLD11] is a dense real-time capable method that uses variational methods in its optimization backend.

In contrast, *sparse* methods select a subset of points from the keyframes to estimate the 3D geometry. Common examples of such methods include [EKC18] and [ESC14] (which is in fact referred to as a *semi-dense* method).

2.2.4 Direct Sparse Odometry

Due to the outstanding visual quality of its 3D reconstructions, the impressively accurate visual odometry and its real-time capabilites, we consider the DSO system developed by the TU Munich computer vision group to be well-suited to generate the point clouds used in this thesis. *Direct Sparse Odometry* or *DSO* [EKC18] is a sparse and direct monocular visual SLAM system using bundle adjustment that is capable to operate in real-time on a CPU. A modified version for stereo cameras is also available [WSC17]. In the following we will give a brief overview over the system.

The image points used to estimate the 3D geometry are selected so that they are evenly distributed in the images and points with higher image gradients are preferred. To obtain depth estimates, the system searches the best match along the epipolar line in the subsequent image frame and performs a triangulation. However, to keep the computational requirements of bundle adjustment at bay, only a subset of active points is used in optimization.

Once a new image frame is fed to the system, tracking is performed by means of direct image alignment applied in a coarse-to-fine fashion on multiple levels of an image pyramid. The new camera pose is initialized by means of a constant motion model (i.e. the camera motion between the two most recently tracked frames is used to initialize the new camera pose). If direct image alignment fails, a heuristic to recover the camera pose is applied. A new frame is made a keyframe if the field of view changes (here, this is measured by the mean square optical flow of the points from the last keyframe) or if the exposure time changes significantly. Again, DSO only uses a subset of active keyframes during BA optimization. Once the number of active keyframes exceeds a certain threshold or if the overlap of the field of view of the active keyframes shrinks excessively, old keyframes are excluded from the set of active keyframes.

Once the camera poses and point depths are initialized, the system minimizes a version of the photometric reprojection error using the Gauss-Newton algorithm on a Lie manifold. A simplified version of the error minimized by DSO is given by

$$E_{\text{DSO}}(\boldsymbol{\theta}) = \sum_{m\in\mathbb{F}} \sum_{\boldsymbol{p}\in\mathbb{P}_m} \sum_{m'\in\text{obs}(\boldsymbol{p})} \sum_{\boldsymbol{p}\in\mathbb{N}_{\boldsymbol{p}}} \left\| (I_{m'}(\boldsymbol{p'}) - b_{m'}) - \frac{t_{m'} e^{a_{m'}}}{t_m e^{a_m}} (I_m(\boldsymbol{p}) - b_m) \right\|_\gamma .$$

Here, $\mathbb{F}$ is the set of active keyframes, $\mathbb{P}_m$ the set of all active points in keyframe m, $\mathbb{N}_{\boldsymbol{p}}$ a set of points in the vicinity of point $\boldsymbol{p}$, t_m the exposure time of frame m, $\boldsymbol{p'} = \Pi_{c_{m'}}(\boldsymbol{R}\Pi^{-1}_{c_m}(\boldsymbol{p}, \rho_{\boldsymbol{p}}^{-1}) + \boldsymbol{t})$ with $\left(\begin{smallmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{smallmatrix}\right) = \boldsymbol{T}_{m'}\boldsymbol{T}_m^{-1}$ the projection of a point $\boldsymbol{p}$ in frame m into frame m', $\|\cdot\|_\gamma$ the Huber norm

$$\|x\|_\gamma = \begin{cases} \frac{1}{2\gamma}x^2 & \text{if } |x| < \gamma \\ |x| - \frac{\gamma}{2} & \text{otherwise} \end{cases}$$

and the parameters $\boldsymbol{\theta} = (\{\boldsymbol{T}_m, \boldsymbol{c}_m, a_m, b_m\}_{m\in\mathbb{F}}, \{\rho_{\boldsymbol{p}}\}_{m\in\mathbb{F},\boldsymbol{p}\in\mathbb{P}_m})$. Note that DSO also accounts for errors in the intrinsic calibration by optimizing the camera parameters jointly with the camera pose

Differences in the brightness due to a change in the exposure times are corrected by means of the factor $\frac{t_{m'}}{t_m}$. The parameters a_m and b_m realize an affine brightness transfer function between the frames which can be used if the exposure times are not available. Otherwise, a Lagrange multiplier constrains these parameters to be zero-valued.

The removal of a keyframe from the active set of keyframes entails marginalizing the random variables that represent the corresponding parameters (camera poses, camera intrinsics, affine lighting transfer parameters and point depths) from the stochastic model. This is realized by means of the Schur complement.

If a photometric calibration is available (inverse response function and lens attenuation), DSO photometrically undistorts the images before working on them. The authors show that photometrically undistorted images boost the performance of the system.

(a) The rig mounted to a baby stroller (b) The aluminum profile with a camera and the monitor mounting plate

Figure 2.1: Images of our stereo recording rig

2.3 Recording Rig

In order to capture new stereo image sequences that can be used to generate 3D reconstructions of a scene using a direct visual SLAM method like DSO, we designed a custom recording rig tailored to the needs of those algorithms. As mentioned in section 1.3, we mainly target data recorded in outdoor scenarios. That being said, the recording rig is built with flexibility in mind, so that it can easily be reconfigured to be used in indoor scenarios.

2.3.1 Hardware

Cameras We use two *Allied Vision Technologies Prosilica GT 1930C*[4] cameras with *Kowa LM6HC*[5] lenses in a standard horizontal stereo pair configuration to acquire the images.

[4] https://www.alliedvision.com/fileadmin/content/documents/products/cameras/Prosilica_GT/techman/Prosilica_GT_TechMan.pdf, accessed: 19.10.18

[5] https://lenses.kowa-usa.com/hc-series/417-lm6hc.html, accessed: 19.10.18

The cameras use a global shutter *Sony IMX174* 1/1.2 CMOS sensor with a resolution of 1936×1216 pixels (≈ 2.4 MP), 12 bit analog to digital converters and color sensing capabilities using a Bayer pattern. To transport the images to a recording device (such as a computer or a frame grabber) the cameras rely on a Gigabit Ethernet connection. The global shutter avoids the *rolling-shutter effect* present in cheaper cameras with rolling shutter sensors which is especially important as the rig must be in constant motion to generate depth. Due to the authors, DSO can not cope with rolling shutter cameras[6] out of the box which made the choice of a global shutter camera necessary. Also, as explained in section 2.1.3, the relatively high image resolution of the cameras diminishes the errors in the estimated depths.

The lenses have a fixed focal length of 6 mm, resulting (for the given sensor) in a field of view of approximately $87° \times 61°$, allow for continuous iris adjustment within the range F1.8-16 and cause low image distortion (-0.2 % TV distortion). In our the horizontal stereo configuration, the vertical FoV angle of 87° is a good tradeoff between strong image distortion (smaller focal length/larger FoV) and too little overlap in the cameras' common field of view (larger focal length/smaller FoV).

Trigger In stereo camera systems, it is imperative to assure exact synchronization of the camera trigger signals. This is due to the fact that the extrinsic calibration will generally not hold if the cameras are moving and the frames are captured at different instances in time. Furthermore, the scene might be dynamic and thus consist of different geometry in images taken at different times. Hence, stereo matching performed in those images will lead to erroneous depth or might even fail completely in the extreme case.

We therefore devised an accurate trigger synchronization mechanism (depicted in figure 2.2) which is based on a periodic pulse emitted by an *Xsens MTi-G* AHRS/IMU. The pulse (SyncOut in the circuit diagram) is amplified using a UCC27524P gate driver with a fast signal propagation characteristic and then fed to the opto-isolated input pin (In 2) of both cameras. This was necessary because the IMU's output current seems to be limited.

Operating on a variable base frequency f_{IMU}, the IMU has the capability to emit the SyncIn pulse at every n-th measurement, resulting in a trigger rate of $f_{\text{cam}} = f_{\text{IMU}} \,/\, n$. For the experiments in this thesis, we chose $f_{\text{IMU}} = 200$ Hz, $n = 10$, and thus $f_{\text{cam}} = 20$ fps.

As the implementation of DSO which is used in this thesis does not make use of inertial measurements, we do not record the data produced by the IMU. Note that inertial data is extremely valuable to a SLAM system as it provides a pose initialization for untracked camera frames. Therefore, we included the IMU into the system to be able to leverage this initialization with other SLAM systems that support inertial data.

[6] see https://github.com/JakobEngel/dso, accessed: 19.10.18

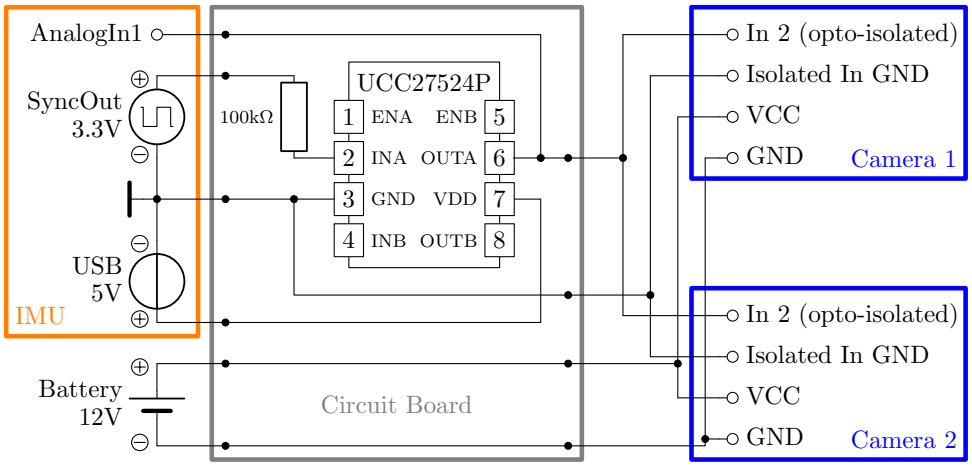


Figure 2.2: Circuit diagram of the IMU-based trigger synchronization mechanism

Mounting Once calibrated, the relative position and rotation of the stereo cameras must not change anymore owing to the fact that the external calibration is only valid for the geometric configuration during calibration. To assure this, we use a sturdy 40×80 mm aluminum profile (the *Bosch Rexroth 40x80L*) to mount the cameras in their fixed horizontal stereo configuration.

In order to keep the setup flexible and modular we designed custom 3D printed mounting plates for the components of the setup (specifically for the cameras, the IMU and a small HDMI field monitor). These are mounted to the Rexroth profile using T-head bolts which means that they can be slid along the profile once the nuts are loosened. Thus, we are able to change the geometry of the setup quickly, even in the field, which is useful to adapt parameters such as the baseline to the scene at hand.

In its current configuration, the recording rig is mounted to the handle of a baby stroller (see figure 2.1a) on an aluminum profile that allows for a maximum baseline of approximately 59 cm. In an outdoor scenario, this baseline enables the reconstruction of close-by objects from an acceptable minimal depth on while keeping the depth errors of points with larger depth at bay. Indoor scenarios tend to require a much smaller baseline as objects are usually much closer. Also, one might want to collect data from a car or from a bicycle in which case the current size factor of the setup is too small (e.g. if one would like to mount the cameras on top of the car) or too large, respectively. Reconfiguring the setup to these use-cases is easily done by exchanging the aluminum profile by a wider or narrower one.

Compute Hardware The images captured by the cameras are recorded on an *Intel NUC 7i7BNHX1* with fast M.2 SSD storage. Its small form factor and fairly large computational power combined with a relatively low energy consumption are advantageous for our use-case. As the NUC easily fits into a mid-sized backpack,

the stereo rig can be deployed in highly mobile scenarios (e.g. on a bicycle's handlebar or on a bicycle helmet). However, in this case the thermal dissipation of the computer has to be taken into account.

Clearly, a high frame rate reduces the risk of DSO's frame tracking algorithm failing. It is thus vital to achieve a decent frame rate for the stereo sequences. As the cameras are connected to the NUC via Gigabit-Ethernet, its data rate is the main bottleneck to the cameras' frame rate. Hence, to maximize the frame rate, we dedicate the NUC's on-board NIC to one and an external USB 3 NIC to the other camera. Following the recommendation in Allied Vision's documentation, the network is configured to use an MTU of 8228 bytes.

Power Supply In order to provide the rig with enough power for recording sessions of an appropriate length, we use two batteries to power the cameras and the Intel NUC separately. With a full charge of our 67 Wh battery, the NUC records data for approximately 2.5 hours while the cameras drain their 200 Wh battery in about 6.5 hours.

2.3.2 Recording Software

We use Allied Vision's *VimbaCPP* library to interact with the cameras. Initially, we experimented with both an available and a custom ROS node which obtain the camera images from Vimba and pass them on to ROS which records the data to a *rosbag* file. However, a significant number of frame drops occurred at higher frame rates in both cases which alludes to the fact that the overhead introduced by the ROS middleware and the rosbag format are detrimental to the recording performance.

This is why we created a light-weight custom recording software which exploits the multi-threaded multi-core architecture of modern CPUs to cope with the load introduced by the cameras' data rate. For each camera, we introduce a thread pool which writes the frames of the camera to disk. We found that debayering and compressing the images on the fly significantly reduces the maximum possible frame rate and hence we defer those tasks to our pre-processing pipeline. As a front end to the recording tool, a Qt-based GUI (see figure 2.3) was developed allowing for a fast and easy interaction with the cameras. For instance, the user set selector in the lower left corner of the GUI provides access to different presets for the camera settings which can be used to load different settings for calibration and data acquisition. We also provide relevant information on the recording process such as the number of dropped frames per camera or the available storage space on the drive that is used to record the data. As described in section 2.3.1 both cameras are triggered at the same time but in the presence of frame drops the corresponding frames have to be matched. This is achieved by means of the frames' sequence numbers generated by the cameras.

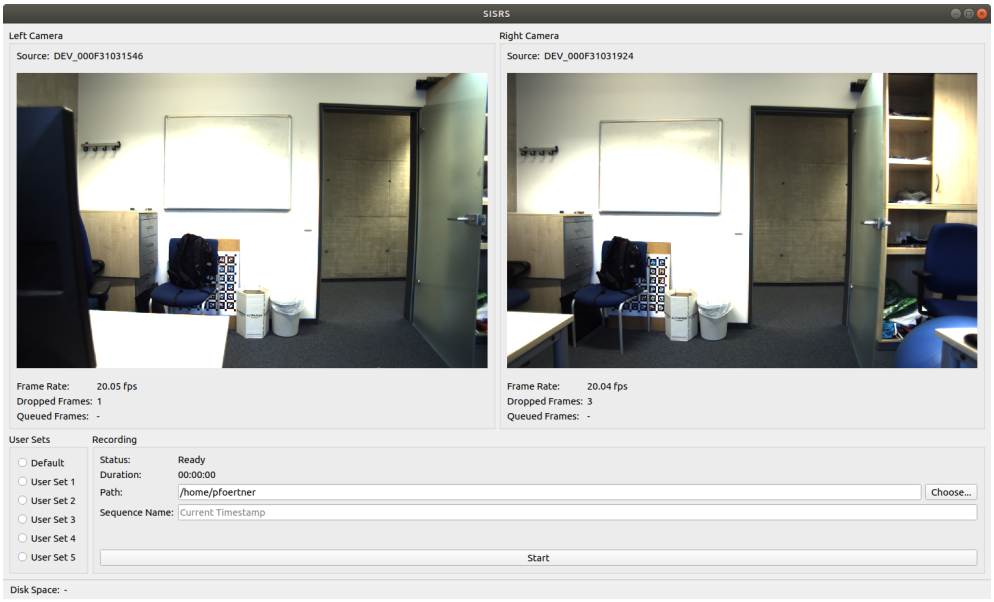


Figure 2.3: GUI of our *Stereo Image Stream Recording Software* (SISRS)

2.3.3 Discussion

In our experiments, it was easily possible to acquire images at a frame rate of 40 Hz (using 8 bit raw Bayer data) without excessively many frame drops. As argued before, this is especially beneficial for visual SLAM systems such as DSO.

Compared to the recording rig used to capture the Kitti vision benchmark [GLU12] which records images at a frame rate of 10 Hz and a resolution of 1392 $\times$ 512 pixels, our system generates videos with a much higher resolution and a higher frame rate at the same time. The horizontal opening angle of 90 degrees is comparable to ours. However, if we crop the images acquired by our camera to an aspect ratio similar to the one used in Kitti even higher frame rates would be possible.

Cityscapes [COR+16] comprises sequences of stereo images with a resolution of 2048 $\times$ 1024 recorded at a frame rate of 18 Hz. While the resolution of the images is comparable to ours, our maximal frame rate is much higher than the one applied in Cityscapes. The AR0331 CMOS sensor used to record Cityscapes which provides 16 bit images. Our cameras only support up to 12 bit color depth at a maximum frame rate of approximately 30 Hz.

Hence, we conclude that our setup can at least keep up with the recording rigs employed to record well-known datasets. However, we think that the flexibility and the small form factor of our rig stand out as it can easily be reconfigured for use in a variety of scenarios.

Moreover, the rig is engineered with reproducibility in mind. As it is relatively cheap and easy to assemble, identical copies of it could be built by different research groups to acquire a large variety of data in many different locations.

2.4 Pre-Processing

Before we can apply DSO to generate 3D point clouds from the image sequences captured with our recording rig, we need to apply several common pre-processing steps.

Bayer Demosaicing As a first step in our pre-processing pipeline the camera frames which are usually recorded in raw uninterpolated Bayer format to minimize the amount of data that has to be transferred over the network (8 bit/pixel uninterpolated Bayer vs. 3 x 8 bit/pixel interpolated Bayer as RGB) have to be debayered and compressed into a common image format such as PNG. To this end we developed a Python C extension interfacing with Allied Vision's VimbaImageTransform library which includes their proprietary implementations of four different debayering algorithms. Also, if either the left or the right frame was dropped during recording, the corresponding stereo frame is discarded in this step.

Calibration In order to calibrate the stereo rig intrinsically and extrinsically we employ the *Kalibr* [MFS13,FRS13] calibration toolbox[7]. As recommended by the developers, we record the calibration sequences at a frame rate of 4 Hz moving an Aprilgrid [Ols11] calibration target in front of the stereo rig whose position is kept static. The intrinsic parameters of the cameras are modeled using the pinhole projection model and the radial-tangential distortion model. Conveniently, Kalibr comprises a tool to calibrate camera-IMU systems which comes in handy if we choose to include IMU data in the future.

Rectification As the final step in our pre-processing pipeline we apply OpenCV's [Bra00] stereo rectification functions to jointly remove lens distortion and generate rectified images from the debayered stereo frames using the intrinsic and extrinsic calibration estimated by Kalibr.

2.5 Point Cloud Generation

As the code for the official implementation of Stereo DSO [WSC17] is not publicly available, we use a third party implementation[8]. However, this system does not implement the same algorithm as in [WSC17] but rather uses the official monocular DSO implementation on the left images (temporal stereo) while leveraging the corresponding stereo frames to initialize the depth estimates in the keyframes

[7] for references and further information see https://github.com/ethz-asl/kalibr/wiki/ and subpages (accessed 20.09.2018)

[8] https://github.com/JiatianWu/stereo-dso, accessed 30.09.2018

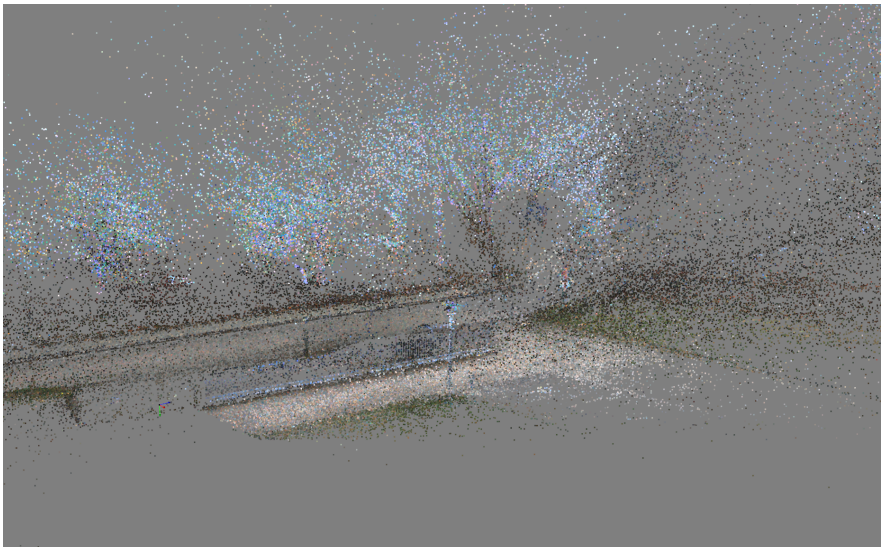


Figure 2.4: Unfiltered point cloud as obtained from DSO

(static stereo). This strategy is analogous to the extension of LSD-SLAM [ESC14] to Stereo LSD-SLAM [ESC15].

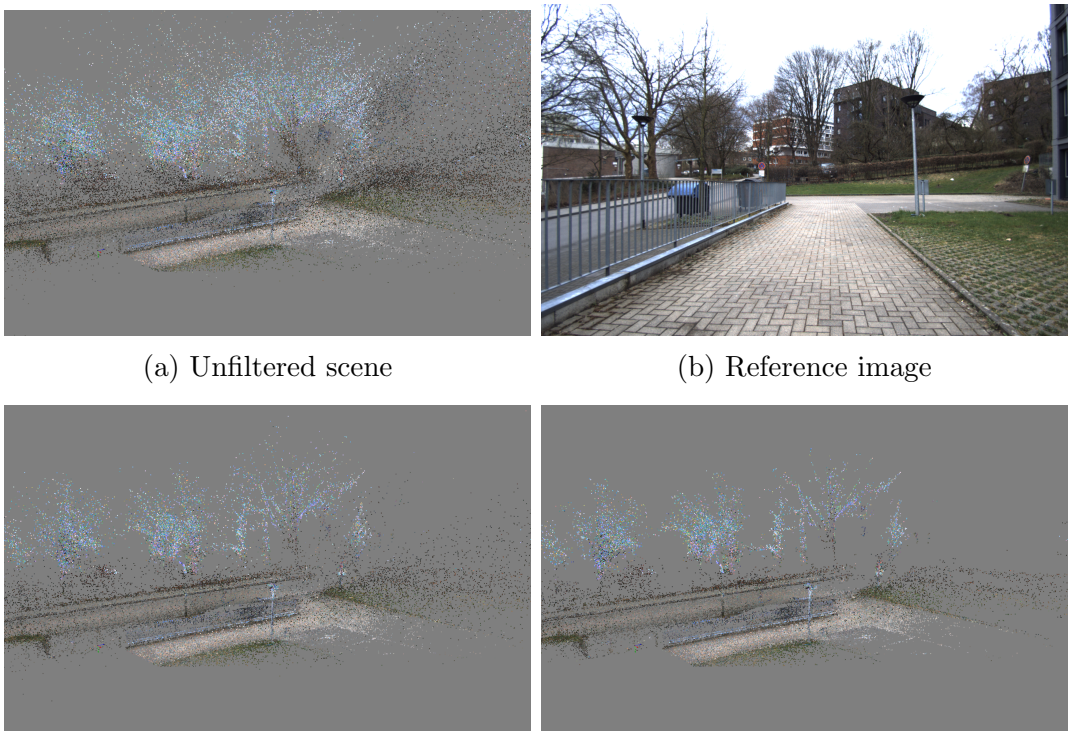
To obtain high-quality reconstructions with a high point density, we usually run DSO with all relevant settings on the maximum values which are supported by the GUI (`activePoints`: 5000, `pointCandidates`: 5000, `maxFrames`: 10, `kfFrequency`: 3) and we leave the gradient threshold at its default value (`minGradAdd`: 7).

For each keyframe, DSO provides the corresponding pose $\boldsymbol{P} = \left(\begin{smallmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{smallmatrix}\right)^{-1} \in \mathrm{SE}(3)$, (optimized) camera matrix $\boldsymbol{K} \in \mathbb{R}^{3\times3}$, and a list of all points anchored to the keyframe which in turn contains the image coordinates $\boldsymbol{p} \in \mathbb{R}^2$ of the point along with its estimated inverse depth ρ. To retrieve the full 3D point cloud, we unproject the image points as follows

$$\boldsymbol{R}\Pi_{\boldsymbol{K}}^{-1}(\boldsymbol{p}, \rho^{-1}) + \boldsymbol{t} = \boldsymbol{R}\rho^{-1}\boldsymbol{K}^{-1}\begin{pmatrix}\boldsymbol{p}\\1\end{pmatrix} + \boldsymbol{t}.$$

2.6 Post-Processing

Before the point clouds we obtain from DSO can be labelled, we need to post-process them to get rid of errors both in terms of the geometry and the colors sampled from the images. Figure 2.4 shows an example point cloud of 610620 points which illustrates the two main problems addressed in this chapter. The reconstructed geometry is relatively noisy overall which, even for humans, makes it difficult to distinguish the semantics of the points. More prominently, the trees in the upper left part of Figure 2.4 are reconstructed with strong color noise. In the following we will analyze the problems at hand and present our attempt to cope with them.



(a) Unfiltered scene

(b) Reference image

(c) After uncertainty-based filtering

(d) After uncertainty- and geometry-based filtering

Figure 2.5: Visualizations of the geometric filtering approaches applied to the point clouds obtained from DSO. The reference image was captured at the position of the coordinate system at the lower left

When evaluating our algorithms, we found out that the first point cloud we generated is scaled by a factor of approximately 2. We found no adequate explanation for this observation, but the problem seems to have disappeared after recalibrating the setup which strongly suggests that the initial calibration was erroneous. Despite the fact that the original point cloud has a wrong scale, it illustrates common issues of the DSO reconstruction well which is why we chose to use it as an example in this section. Hence, one ought to keep in mind that parameters like world space radii used in the following subsections will also be scaled by a factor of two.

2.6.1 Geometric Filtering

The geometric noise in the reconstructions is presumably caused by incorrect depth estimates which in turn arise from a bad depth initialization. Our attempt to filter out those noise points comprises two different approaches applied on top of one another.

Uncertainty-based filtering First we apply a group of filters that discard points with high measurement uncertainties in the depth values. These can also be found in the code[9] of the original DSO system. The authors apply three thresholds `minRelativeBS` (θ_b), `absVarTH` (θ_ρ), and `relVarTH` (θ_d), to discard a point $\boldsymbol{p}$ if one of the following conditions holds.

$$b_{\boldsymbol{p}} < \theta_b \tag{2.5}$$
$$H^{-1}_{\rho_{\boldsymbol{p}}} > \theta_\rho \tag{2.6}$$
$$d^4_{\boldsymbol{p}} H^{-1}_{\rho_{\boldsymbol{p}}} > \theta_d \tag{2.7}$$

holds. Here, $H_{\rho_{\boldsymbol{p}}}$ denotes the entry of the Hessian approximation corresponding to the inverse depth $\rho_{\boldsymbol{p}}$ of point $\boldsymbol{p}$ and $b_{\boldsymbol{p}}$ denotes the maximum baseline with which point $\boldsymbol{p}$ was observed by the system. We could not find any mention of these thresholds in the papers and hence provide our own explanation as to whether they are indeed useful.

The condition 2.5 enforces that a point's depth estimate has to be backed by an observation at a certain minimal baseline. This can be motivated by equation 2.2 which illustrates that the uncertainty of a point's estimated depth decreases with an increasing baseline.

As explained before, the photometric bundle adjustment approach used in DSO can be interpreted as a maximum likelihood estimation of the parameters (camera poses, camera calibration, inverse depths, etc.). Along those lines, the Hessian approximation $\boldsymbol{H}$ used in the Gauss-Newton minimization is an approximation to the inverse covariance matrix of the parameter estimates [ESC14] (neglecting correlations between the estimates). As $\boldsymbol{H}$ is a diagonal matrix, the variance σ_x^2 of any parameter estimate x can be approximated by

$$H_x^{-1} = \sigma_x^2. \tag{2.8}$$

Furthermore, propagation of uncertainty yields

$$\sigma_d = \sqrt{\left(\frac{\partial d}{\partial \rho}\sigma_\rho\right)^2} = \frac{1}{\rho^2}\sigma_\rho = d^2\sigma_\rho \tag{2.9}$$

for the measurement uncertainty σ_d of the depth, given the uncertainty σ_ρ of the inverse depth.

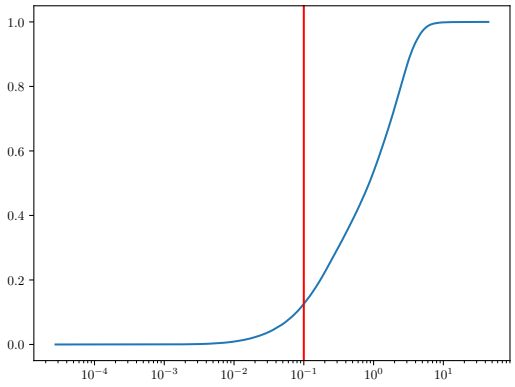
With these insights in mind, conditions 2.6 simplifies to

$$H^{-1}_{\rho_{\boldsymbol{p}}} = \sigma^2_{\rho_{\boldsymbol{p}}} > \theta_\rho, \tag{2.10}$$

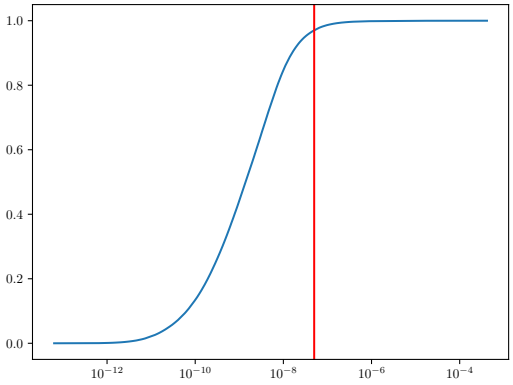
while condition 2.7 turns into

$$d^4_{\boldsymbol{p}} H^{-1}_{\rho_{\boldsymbol{p}}} = \left(d^2_{\boldsymbol{p}}\sigma_{\rho_{\boldsymbol{p}}}\right)^2 = \sigma^2_{d_{\boldsymbol{p}}} > \theta_d. \tag{2.11}$$

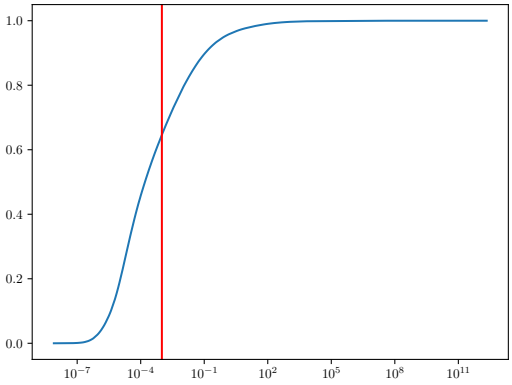
[9] https://github.com/JakobEngel/dso/blob/master/src/IOWrapper/Pangolin/KeyFrameDisplay.cpp lines 221-232, accessed 22.09.2018



(a) $b_{\boldsymbol{p}}$, $\theta_b = 0.1$



(b) $\sigma^2_{\rho_{\boldsymbol{p}}}$, $\theta_\rho = 5 \cdot 10^{-7}$



(c) $\sigma^2_{d_{\boldsymbol{p}}}$, $\theta_d = 10^{-3}$

Figure 2.6: Cumulative distributions of the thresholded values (blue) along with the chosen threshold values (red). All abscissas use a logarithmic scale.

Hence, the thresholds θ_ρ and θ_d are upper bounds on the variances of the inverse depth and the depth, respectively. The variance of these parameters gives a measure of locality or certainty for the estimates of the (inverse) depth which means that depths with high variances are more likely to contain errors than those that have a low variance.

As there is no meaningful way to find appropriate values for the thresholds automatically, we evaluated several combinations manually and compared the quality of the reconstruction manually. The values of the thresholds along with their effect on the cumulative point distribution over the thresholded values is visualized in Figure 2.6. After applying all three thresholds, the number of points drops to 391286.

In Figure 2.5c, the effect of the thresholds on the geometry is visualized. The structure of the scene, especially around the trees in the upper left corner, is

drastically refined. Also the noise points in the upper right corner of Figure 2.5a almost vanished completely.

Neighborhood-based filtering On top of the uncertainty-based filtering approach, we discard additional points by means of the spatial structure of a point's neighborhood. For $\boldsymbol{p} \in \mathbb{P}$ and $r \in \mathbb{R}_{\geq 0}$ let

$$\mathbb{B}_{\boldsymbol{p}}(r) = \{\boldsymbol{p}' \in \mathbb{P} \mid \|\boldsymbol{p}' - \boldsymbol{p}\|_2 \leq r\} \tag{2.12}$$

be the set of all points within a neighborhood of radius r (w.r.t. the Euclidean metric). The filter discards a point $\boldsymbol{p}$ if the number of points $|\mathbb{B}_{\boldsymbol{p}}(r)|$ within a radius r does not exceed a specified threshold $\theta_r \in \mathbb{N}$. Naturally, the threshold should be chosen such that $\theta_r \geq 2$ as it is obvious that $\boldsymbol{p} \in \mathbb{B}_{\boldsymbol{p}}(r)$ for all $r \in \mathbb{R}_{\geq 0}$ and hence $\theta_r < 2$ would not have any effect. For efficiency reasons, we only apply this procedure once instead of filtering until a stable configuration is reached. A ball tree [Omo89] (as implemented in scikit-learn [PVG+11]) is used to perform efficient radius queries in sublinear time.

The working principle of the filter can be motivated by the following observation: Assuming that the world geometry estimated by DSO is locally smooth, there should be a small region around each point which contains several neighboring points. If this it not the case, there is a good chance that the point is an outlier. Naturally this approximation is not always true, as DSO solely estimates sparse depth maps. However, the point clouds produced by DSO are sufficiently dense, so it works well enough in practice.

Again, we tuned the algorithm's hyperparameters $r = 1.0$ and $\theta_r = 4$ manually while inspecting the quality of the reconstruction. This final filtering step gradually reduces the number of points to 372805. We visualize its effect in Figure 2.5d. We observe that the structure of the tree branches is further refined and several of the noise points above the road in the foreground are removed.

2.6.2 Color Correction

After refining the geometry, the errors in the colors of the trees stand out possibly even stronger than before (see figure 2.5d). When investigating into the problem at hand, we learned that DSO samples the points that make up the trees in the point cloud mostly along the occlusion boundary with the sky in the keyframe images (see Figure 2.7). This is due to the fact that these boundaries exhibit strong image gradients while the sky and the trees appear mostly untextured. However, we found strong color artifacts along the occlusion boundaries which are most likely Bayer artifacts and/or chromatic aberrations. An example of these artifacts can be found in Figure 2.7. Even though the color artifacts at times reach a strength that corrupts the color of the whole branch (especially for thinner branches, see Figure 2.8), we attempted to get rid of them automatically.

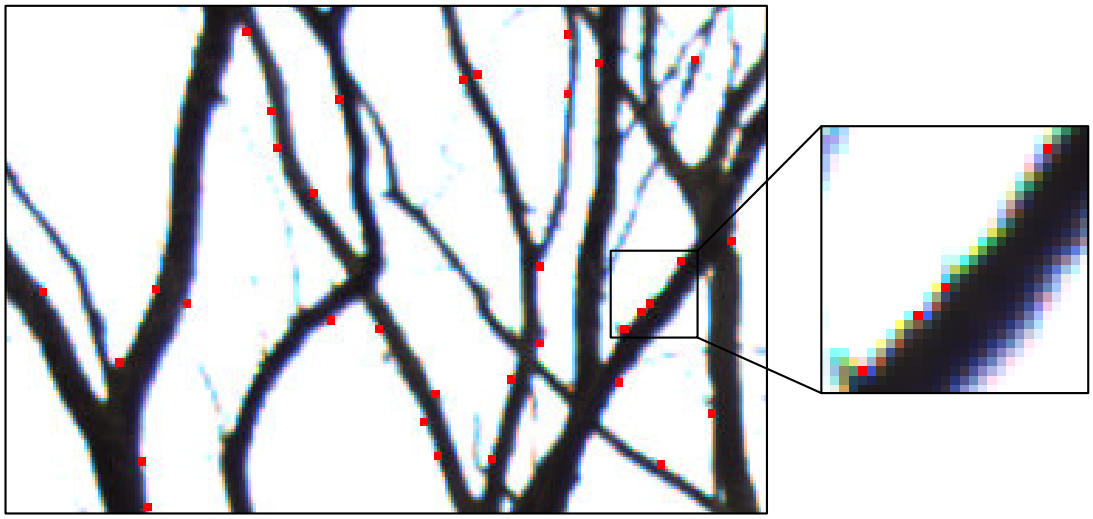


Figure 2.7: Crop of a keyframe image with several of its points (red) superimposed. The color artifacts along the occlusion boundary are clearly visible in the enlarged part on the right.

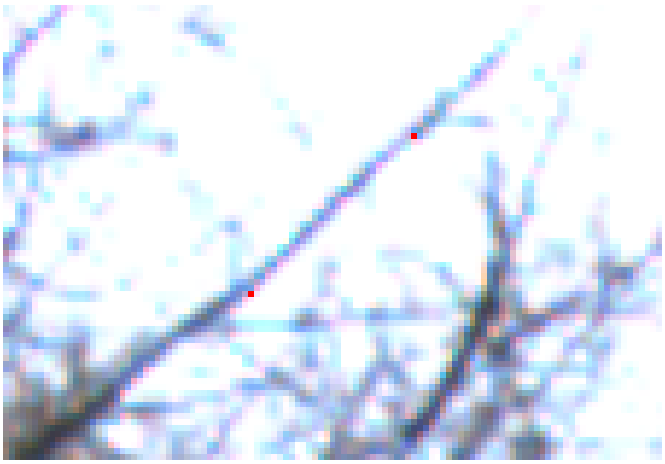
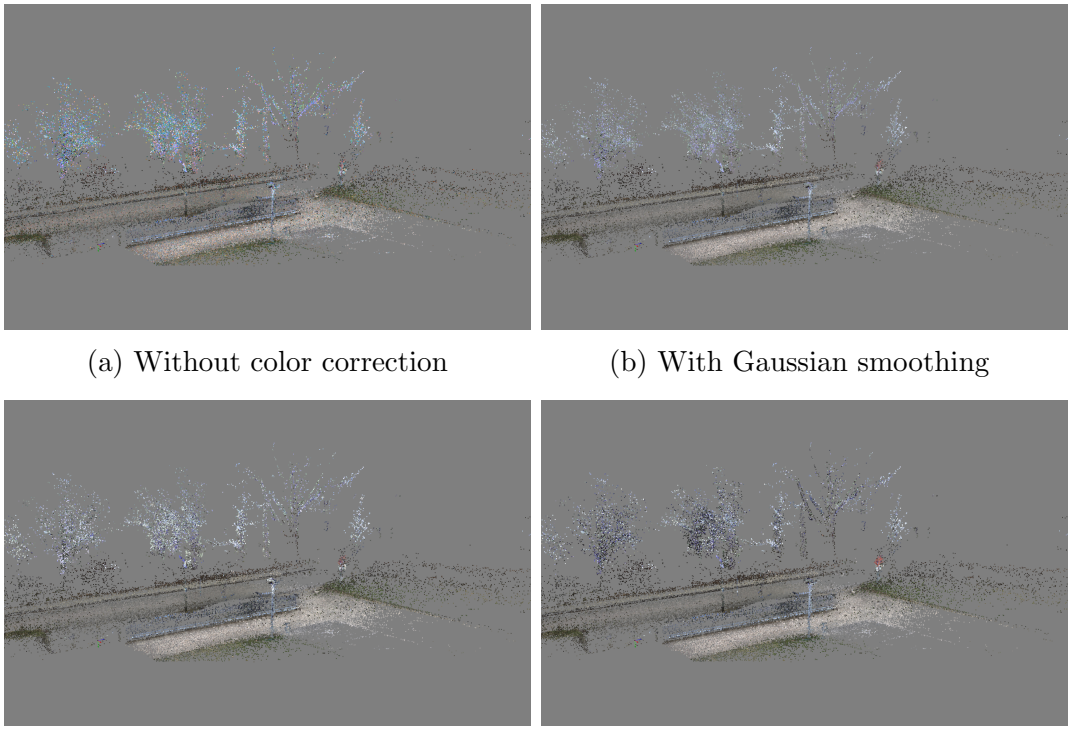


Figure 2.8: Strong color noise corrupting a thin tree branch

In a first step, we apply Gaussian smoothing to the keyframe images to get rid of the high frequency color variations in the vicinity of the occlusion boundary. The Gaussian noise also reduces other instances of sensor noise. While this comes at the cost of blurred edges in the keyframes, it visually enhances the quality of the point cloud color samples without introducing visible errors. We found that a Gaussian kernel with a standard deviation of 2 pixels and a kernel size of 13x13 yield a good result for our particular point cloud. The result can be seen in Figure 2.9b.

The observation that the artifacts mostly occur along occlusion boundaries suggests that a slight shift of the points into the direction of the occluder (on the image plane) before querying the image color should correct the artifacts.



(a) Without color correction (b) With Gaussian smoothing

(c) With Gaussian smoothing and normal-based shift (d) With Gaussian smoothing and disparity-based shift

Figure 2.9: Visualizations of the color correction approaches applied to the point clouds after geometric filtering

Hence, we experimented with the following approaches to find the corresponding direction of the occluder for every point.

Normal-Based Shift A shift along the negative of the surface normal vector would move a point inside the object in 3D. Hence, when projected onto the image plane, the negative normal vector points in the direction of the occluder along an occlusion boundary.

To estimate the normal vectors of the points, we apply a covariance based method as described in [BC94]. These methods natively incorporate a certain robustness to geometric noise which is important in our case. First, given all points in the vicinity (defined by a radius r and the Euclidean norm) of a given point p, we compute the *surface covariance matrix*

$$\boldsymbol{\Sigma}_{\boldsymbol{p}}(r) = \sum_{\boldsymbol{p}' \in \mathbb{B}_{\boldsymbol{p}}(r)} (\boldsymbol{p}' - \boldsymbol{p})(\boldsymbol{p}' - \boldsymbol{p})^T \in \mathbb{R}^{3\times 3}. \tag{2.13}$$

As it is symmetric and positive semi-definite, there is an orthonormal basis of $\mathbb{R}^3$ which consists of eigenvectors of $\boldsymbol{\Sigma}_{\boldsymbol{p}}(r)$. If there is only one minimal eigenvalue, its corresponding eigenvector is the estimated surface normal. This is due to

the fact that the eigenvectors corresponding to the two largest eigenvalues span a plane which is the best estimate of the local tangent plane in a least squares sense (see [BC94] for details) and as the eigenvectors are chosen orthogonal to one another, the third one has to be orthogonal to this tangent plane estimate. However, this only determines the normal vector up to its sign. We therefore choose the sign such that the normal vector points in the direction of the camera which captured the point's keyframe. This is motivated by the fact that we can only observe points whose normal vectors are directed towards the camera.

In our implementation we started off with a slightly simplified version of the algorithm to test the performance of the approach. First, we check whether $\mathbb{B}_{\boldsymbol{p}}(r)$ at least $\theta_{\boldsymbol{\Sigma}}$ points ($\theta_{\boldsymbol{\Sigma}} = 4$ in our experiments) and only proceed to estimate a normal vector if this is the case. We then compute $\boldsymbol{\Sigma}_{\boldsymbol{p}}(r)$ and its symmetric eigendecomposition $\boldsymbol{\Sigma}_{\boldsymbol{p}}(r) = \boldsymbol{U}\boldsymbol{D}\boldsymbol{U}^T$ where the columns of $\boldsymbol{U}$ (the orthonormal eigenvectors) are ordered according to the size of the corresponding eigenvectors in descending order. The last column of $\boldsymbol{U}$ is then assumed to be the point's normal vector $\boldsymbol{n}_{\boldsymbol{p}}$. We project $\boldsymbol{n}_{\boldsymbol{p}}$ onto the corresponding keyframe and normalize the projection which is then negated and scaled by a factor d_n to realize the shift.

To be able to estimate a normal vector for sufficiently many points, we had to choose $r = 0.5$.

Figure 2.10a shows that the normal vector shifts are estimated correctly on the ground plane which we take to mean that the implementation functions correctly. Nevertheless, we observed that the normal estimates on the tree branches are often incorrect, if present at all (an example of the resulting shifts can be found in Figure 2.10b). This is likely caused by one of two reasons. For one, the point density around the tree branches is low. Thus, the estimate of the normal vector is not backed by many points which means that outliers have an immense impact on the orientation. Secondly, to obtain sufficiently many normal vector estimates, we chose a relatively high radius r which means that points of other branches nearby corrupt the surface covariance matrix. Applying the normal shift to all point sample positions rather makes the color artifacts worse, instead of improving them (see Figure 2.9c).

Evidently, covariance based normal estimation methods work best if the local geometry is roughly planar. As the points extracted from the tree branches are very non-planar, the method largely fails. Hence, the working principle of the normal estimation seems to be inapplicable in our case.

Mode-Based Shift To circumvent the issue of locally non-planar geometry around the tree branches, we attempted to use an adapted version of the *mean shift* algorithm for robust mode search [CM02] directly on the point cloud. Due to the fact that, in general, tree branches are locally convex, the mean over all points in the neighborhood of a given point (i.e. the local center of mass or mode) should lie inside the branch. Computing the aforementioned local mean amounts to performing one iteration of mean shift with a uniform kernel.

(a) Normals shifts on the ground plane

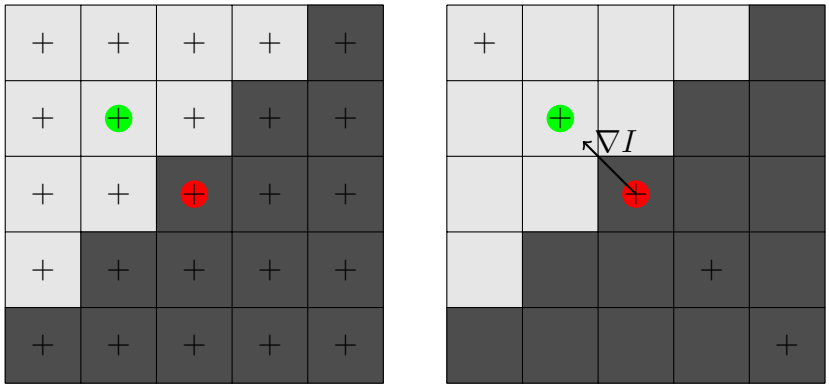
(b) Normal shifts on a tree

Figure 2.10: Visualizations of the 2D shifts computed using normal estimation. The red points are the original point locations and the green points are the shifted point locations.

In our experiments, the algorithm performed even worse than the normal-based shift which is why we do not present the results here. Again, the bad performance mainly seems to be caused by the low point density around the tree branches.

Disparity-Based Shift As mentioned before, the poor performance of the two algorithms above can be attributed to the fact that the estimated geometry is too sparse in the regions of interest. This is why we also attempted to leverage additional data to tackle the problem.

As we have stereo images at our disposal, we can generate dense disparity maps to estimate the occlusion boundaries directly. According to equation 2.1, we can detect occluders by finding local maxima of the disparity values. Hence, to shift an image point which lies next to or on an occlusion boundary onto the occluder, we need to find the direction in which the disparity values are the largest. To find a robust estimate of this direction, our approach computes a disparity-weighted mean over the pixel coordinates in the neighborhood of each image point. Hence, this weighted spatial mean is drawn towards areas with high disparity values while being relatively robust to noise and outliers in the disparity maps. Conceptually, this is equivalent to a mode search in the disparity values around each query point. An advantage of this approach is that points will only be shifted if there is a strong change in the neighboring disparities, indicating the presence of an occlusion boundary. We define two different kinds of neighborhood used in the



(a) Full neighborhood (b) Line neighborhood

Figure 2.11: The two different kinds of neighborhoods used to find the direction towards the occluder on an occlusion boundary. Brighter pixels have a higher disparity. The + signs indicate which disparity values are used in the computation. The red point is the original query location while the green point is the corrected query location.

computation of the weighted mean. Let D be the left disparity map, $\boldsymbol{p}$ be the original query point and $\boldsymbol{p}'$ be the shifted query point.

For one, we use a full square neighborhood window of width $2W+1$ (see Figure 2.11a). In this case, the weighted mean is computed as

$$\boldsymbol{p}' = \sum_{\Delta v=-W}^{W} \sum_{\Delta u=-W}^{W} \frac{D(p_u+\Delta u, p_v+\Delta v)}{\sum_{\Delta v'=-W}^{W} \sum_{\Delta u'=-W}^{W} D(p_u+\Delta u', p_v+\Delta v')} \begin{pmatrix} p_u+\Delta u \\ p_v+\Delta v \end{pmatrix}.$$

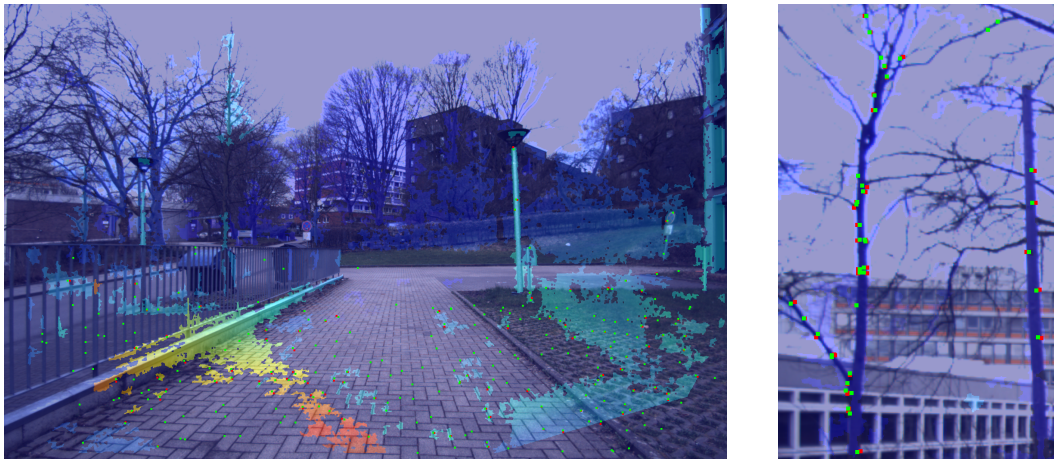
Usually, occlusion boundaries go hand in hand with strong texture boundaries which also entails that the occlusion boundary's normal vector is given by the image gradient. Hence, we can technically restrict the mean to the neighborhood pixels on the line

$$l_{\boldsymbol{p}} \colon \{-W, \dots, W\} \to \mathbb{R}^2, w \mapsto \boldsymbol{p} + w \cdot \frac{\nabla I(\boldsymbol{p})}{\|\nabla I(\boldsymbol{p})\|_\infty}$$

defined by the image gradient (see Figure 2.11b) and clipped to a window of $W \times W$. We normalize the image gradient using the supremum norm $\|\cdot\|_\infty$ so that the neighborhood pixels are densely queried on the pixel raster along one direction. For efficiency reasons the pixel coordinates are rounded to the nearest integer instead of performing bilinear interpolation. Consequently, we compute the weighted mean as

$$\boldsymbol{p}' = \sum_{w=-W}^{W} \frac{D\left(l_{\boldsymbol{p}}(w)\right)}{\sum_{w'=-W}^{W} D\left(l_{\boldsymbol{p}}(w')\right)} \cdot l_{\boldsymbol{p}}(w).$$

In both cases, $W = 10$ proved to be an appropriate window size.



(a) A poor result for the disparity map causes unwanted displacements, even for planar geometry

(b) Shifts along occlusion boundaries

Figure 2.12: Left disparity maps superimposed onto the corresponding left image. The warmer the color, the higher the disparity value.

In our implementation, we compute the disparity maps using the stereo matching library *ELAS* [GRU10]. Image derivatives are computed using the *Sobel operator* [SF68] as implemented in OpenCV [Bra00].

Even after extensive tuning of ELAS's hyperparameters (manual tuning, as there is no ground truth available) the quality of the generated disparity maps is generally poor for unidentified reasons (see Figure 2.12a). Nevertheless, given reasonable disparity maps, our algorithm is robust enough to produce the expected shifts (see Figure 2.12b). Figure 2.9d shows that the algorithm improves the appearance of the point cloud which manifests itself along the tree trunks and the lamp pole in the foreground. We found that the choice of neighborhood used in the computation of the weighted mean has no visible influence on the quality of the point cloud. This is why we only present examples generated using the line neighborhood.

The remaining errors are mostly due to the fact that fine structures are not recovered well by ELAS (e.g. the thin branches of the trees in the background of Figure 2.12a). Our experimental results suggest that our algorithm will perform much better if proper disparity maps can be generated. One has to take into account that (window-based) dense stereo matching is inherently difficult in the presence of thin structures and around occlusion boundaries. However, we suspect that the algorithm is at least robust enough to handle bleed on the occlusion boundaries.

Discussion With the disparity-based shift algorithm we devised a method which improves and which, given better disparity maps, can in principle solve the color

artifacts of the point clouds. Be that as it may, neither of the algorithms presented above can operate at frame rate which means that they cannot be applied in a production environment. In lack of better and faster algorithms to get rid of the artifacts, it is important that the semantic segmentation methods can handle point clouds whose colors are corrupted by similar artifacts. As algorithms that rely on machine learning techniques need a representative sample from the data generating distribution, it can thus be considered beneficial to keep the color artifacts in the training data in our case. Taking all the above into account, we choose not to apply a color correction algorithm in the post-processing step.

2.7 Future Work

Exposure Times Currently, we don't retrieve the exposure times of the image frames recorded by the camera. DSO's performance is significantly increased when providing exposure times with the images [EKC18] and thus we plan on adding this capability to our recording software.

Photometric Calibration Once we have access to the exposure times for each individual video frame, we can obtain a photometric calibration for our cameras. Again, this increases the performance of DSO as shown in [EKC18]. To carry out the calibration, the implementation of the algorithm in [EUC16] seems promising as it produces a photometric calibration which is well-integrated with the official implementation of DSO.

IMU Integration As we already integrated an IMU into our recording rig, it is relatively simple to acquire inertial measurements into our recording rig. As mentioned before visual SLAM systems can leverage inertial data to initialize the new camera poses during tracking. It has been shown that a tight integration of inertial measurements improves the accuracy of the visual odometry [LLB+15]. There even exists a version of DSO dubbed VI-DSO [vSUC18] which adds tight integration of inertial data to the system. Unfortunately, at the time of writing, there is no publicly available implementation of VI-DSO.

3 Semantic Annotation of 3D Point Clouds

Creating large-scale annotated datasets for semantic segmentation is usually a tedious task consuming vast amounts of man-hours to provide the manually assigned labels for the pixels/points. Intuitive, user-friendly tools supporting the human annotator are key to maximizing the amount of data that, given a fixed set of resources, can be labelled.

Due to the dense, regular 2D structure of images, the interaction workflow of most annotation tools for 2D semantic segmentation is conceptually simple while still being highly effective. However, pixel-accurate annotations along semantic boundaries are still difficult to obtain and sometimes ambiguous.

Unfortunately, annotation workflows for 3D semantic segmentation are arguably challenged by the fact that we visualize the 3D data on a 2D screen. The most common types of interaction hardware (computer mouses, touchpads/trackpads, and touch screens) are inherently bound to 2D interaction which makes it difficult to deal with the 3D structure of the data intuitively. Furthermore, while being beneficial in terms of memory consumption and robustness to noise, the sparse, unstructured nature of point clouds introduces additional challenges which will be described in more detail in Section 3.2.

In this chapter we present the central challenges of labeling 3D point clouds. Moreover, we present our custom annotation tool which aims to overcome several of these challenges, especially by leveraging 2D images to simplify the 3D annotation process (if 2D images are available).

3.1 Related Work

Arguably, the most popular annotation interfaces for 2D semantic segmentation use arbitrary polygons to select the pixels. Among those, the well-known web-based *LabelMe* tool [RTMF08] inspired many similar tools such as those used to label Cityscapes [COR+16] and and COCO [LMB+14]. The *Labelbox* [Lab] library

offers a LabelMe-style polygon-based interface along with pixel-wise interfaces based on a brush and superpixel selection.

For 3D semantic segmentation, many different annotation schemes have been proposed.

Analogously to the 2D LabelMe interface, many tools use 3D geometric primitives as bounding volumes to annotate all points inside with the same label. Cuboids [Aut, Sup, XKSG16] and ellipsoids [XKSG16] are common choices for these primitives.

[HSL+17] and [AL] render a fixed view of the point cloud to use a 2D selection tools on the rendering canvas. The selection is carried out via LabelMe-style polygon selection [HSL+17, AL] or a lasso tool [AL] and points can be added to or removed from the set of selected points. As this strategy suffers from occlusion-related artifacts (see Section 3.2.1), the selection mask must be refined from multiple different view points.

Similar to the superpixel labeling interface in [Lab], approaches like [Yan] or [MAZV17] group close-by points of the point cloud which are then labeled jointly. Further details concerning these approaches can be found in Section 4.1.

The annotation tool used for the SUN3D dataset [XOT13] explicitly leverages the corresponding 2D images of the RGB-D videos to annotate data in 3D. They use a LabelMe-style 2D interface to provide a semantic labeling for the image points. Additionally, the manually provided polygonal labelings are propagated to different video frames which is possible because the point clouds of the individual frames have been aligned by means of SfM.

To our knowledge there are currently no annotation tools that are tailored to point clouds obtained from visual SLAM systems.

3.2 Challenges

As mentioned before, point clouds offer a distilled and hence compact representation of 3D geometry that handles geometric errors (e.g. due to sensor noise) with grace. Be that as it may, when annotating point clouds, there are several important issues that mainly arise from their sparse nature.

3.2.1 Occlusions

It is obvious that, in a 2D rendering of a 3D scene, points which are occluded by other geometry should generally not be visible. Given a dense geometry representation such as polygonal meshes, this is easily achieved by means of z-buffering. However, due to the fact that point clouds are inherently sparse and unstructured, it is immensely difficult to attain the expected occlusions when rendering point clouds naively. Particularly, algorithms like z-buffering which rely on overdrawing a fragment once another fragment with a smaller depth value has been found will fail since the sparse geometry representation will naturally not overdraw all points

Figure 3.1: The sparse geometry representation of the car is not able to overdraw the occluded sidewalk and vegetation behind it

of the occluded geometry. This problem is visualized in Figure 3.1. Evidently, we would need to extract information about the dense geometry in order to fully sort out all occlusion problems. However, this is an expensive and difficult step which might yield poor results if the geometry is too sparse. From Section 2.6.2 we already know that the SLAM point clouds suffer from excessive sparsity in some cases. Consequently, we ignore the occlusion problems when rendering the point clouds. We observe that this is largely unproblematic in practice because the point clouds are mostly dense enough for the human eye to perceive the 3D geometry correctly, at times with a little aid of camera motion.

While the occlusion problems play a minor role in rendering, they give rise to severe issues in interaction. The most convenient way to label the point clouds is by means of cursor interaction on the 2D rendering canvas. Unfortunately, any naive 2D brush or 2D polygon based tool for annotating the point clouds will suffer from massive problems related to the lack of dense occlusion information. For example, a tree positioned in front of a wall cannot be properly labeled with these tools since any brush stroke or polygon selection will also select many points of the wall owing to the fact that there is no way to distinguish the respective point occlusions reliably.

3.2.2 Sparse Geometry

Section 2.6.2 has already shown that parts of the SLAM point clouds are likely too sparse to infer properties of the dense geometry algorithmically. However, even for humans, it is sometimes difficult to resolve all ambiguities when assigning a label to every point in the point cloud. This problem mainly manifests itself in two ways.

For one, in a number of cases, the local density of the 3D geometry drops to a point where one can virtually not recognize what is depicted by that part of the

(a) 3D view (b) 2D view

Figure 3.2: The excessively low point density makes it almost impossible to identify the objects depicted by the part of the point cloud shown here. A 2D view reveals that the points belong to a building and a hand rail.

(a) 2D view (b) 3D view

Figure 3.3: Different views of a street-sidewalk boundary

point cloud (see Figure 3.2). Note that this problem is less prominent with point clouds from LiDAR or RGB-D as they sample the geometry evenly (in terms of angular resolution).

The dense depiction of a scene in a 2D image contains many visual cues that help assign a semantic label to each pixel, especially along semantic borders (e.g. object borders). Examples of such cues include edges (i.e. strong image gradients), smooth intensity/color variations, and equally textured areas. By and large, the sparsity and the irregularity of the SLAM point clouds make it impossible to leverage the same kinds of visual cues in the rendered 3D scenes. This is problem particularly prominent in planar structures such as along street-sidewalk and street-curb boundaries (see Figures 3.3 and 3.4). As mentioned

(a) 2D view (b) 3D view

Figure 3.4: Different views of a curb

before, LiDAR and RGB-D reduce the irregularities in the point cloud by their regular sampling process. However, especially when zooming into the geometry, LiDAR and RGB-D point clouds suffer from similar density issues and can thus not leverage the visual cues mentioned above either.

3.2.3 Geometric Noise

Even though the algorithms presented in Section 2.6.1 already filtered out a significant amount of geometric noise, there are still some noise and outlier points left. This is visualized in Figure 3.5. Arguably, the most elegant way to handle the outlier points during annotation is to introduce an additional outlier label. However, it is a tedious task to label all outlier points as they are either isolated and spatially distributed over the whole point cloud or difficult to distinguish from proper points due to spatial proximity. Also it is at times difficult to identify outlier points without camera motion which, in a way, renders naive point-and-click annotation useless.

Most noise points in our experiments are scattered in a relatively compact region around the expected geometry. This is why we chose to label geometric noise as if it were the actual geometry. Note that, for most points, there is no adequate way of distinguishing noise and the actual geometry because the noise distribution is smooth.

3.2.4 Alignment Drift

An aligned temporal sequence of 3D data (e.g. visual SLAM reconstructions, RGB-D SLAM reconstructions, ICP-aligned LiDAR point clouds, etc.) tends to accumulate errors over the spatial extent of the point cloud. This is due to the fact that these alignments usually only take the local geometry into account. Unless a procedure like loop closure is applied, these drifts can cause severe alignment issues if temporally distant parts of the reconstruction overlap (see Figure 3.6).

Figure 3.5: Outlier and noise points above and below the ground planes of the reconstructed scenes

Figure 3.6: Temporal drift in the reconstructed camera poses leads to alignment issues. There are two ground planes with different heights.

3.2.5 Performance and Responsiveness

As argued above, labeling large amounts 3D data is in itself a tedious task. Hence, the usability of an annotation tool used to tackle the task needs to maximized in order for human labelers to be as productive as possible. Nuisances like large input lag, low rendering frame rates or long waiting times for UI actions to complete are thus unacceptable as they introduce additional complications to the labeling process. Ideally, all operations should run a frame rate of about 30-60 fps.

Unfortunately, 3D data sources tend to generate large numbers of points. For instance, our recording pipeline is capable of generating well over 600000 points from a 1:55 min video recorded at 20 fps (2299 frames) and LiDAR or RGB-D devices typically produce even more points.

Altogether, we impose tight real-time constraints onto the annotation tool while presenting it with large amounts of data to process. This means that all UI related algorithms need to be highly efficient to allow for responsiveness, ideally even on consumer hardware.

3.3 Annotation Tool

Taking all the problems presented in Section 3.2 into consideration, we engineered an annotation tool which is especially suited for the data generated with our recording pipeline. However, we also support point clouds from most other common data sources.

We chose to develop the tool using Python and its official Qt bindings (PySide2) mainly because this GUI stack allows for rapid development while providing decent performance. Also, many packages in Python's scientific stack (such as NumPy and scikit-learn) provide essential functionalities used for numerous tasks within our system. Finally, this ecosystem easily enables cross-platform support.

The GUI of the annotation tool is showcased in Figure 3.7. In the following we will pinpoint several key characteristics of the tool.

3.3.1 Data Model

Subjecting existing 3D datasets to close scrutiny, one finds that temporal sequences of point clouds captured by a moving sensor recur throughout most of them. Multiple LiDAR scans are usually recorded at a certain frame rate (e.g. [GLU12]), 3D geometry in many visual SLAM systems originates from image pixels in the keyframes of a video sequence (our recording pipeline) and RGB-D video (e.g. [XOT13,SLX15,ASZS17,CDF+17]) produces one point cloud from the depth map of each video frame. In our annotation tool, we leverage the temporal sequence structure to account for several of the problems introduced above.

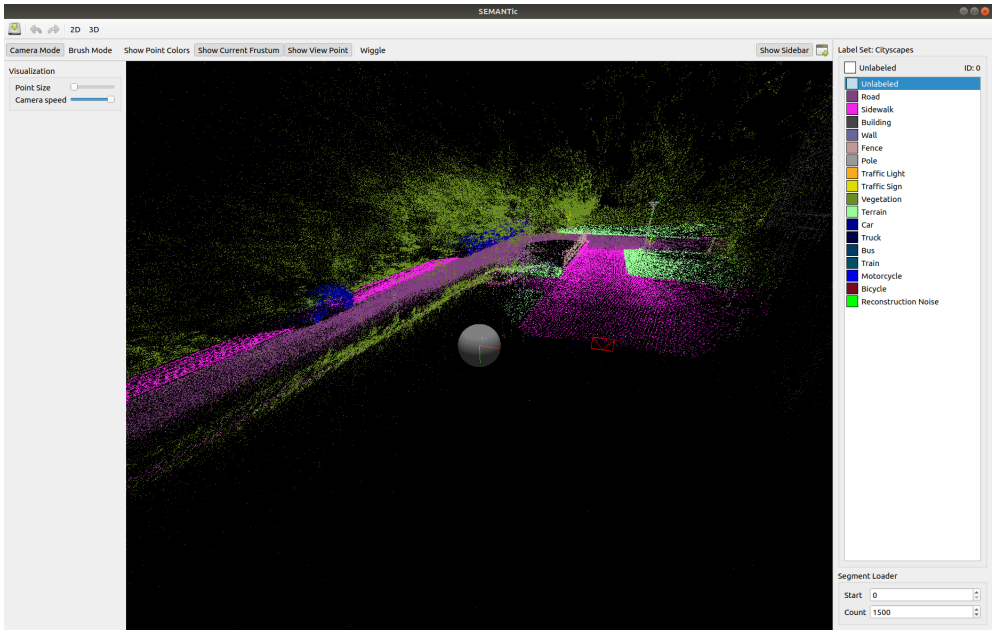


Figure 3.7: The annotation tool showing its 3D annotation pane with camera mode activated and several point cloud segments loaded

In the following, we dub one element of a point cloud sequence (the 3D points from one LiDAR scan/keyframe/RGB-D frame) a (point cloud) *segment*. The segments in a sequence are assigned consecutive sequence numbers. We denote the m-th segment of a point cloud sequence of length M as $\mathbb{P}^{(m)}$.

We assume that the transformations between the local coordinate systems of the segments (i.e. the camera or LiDAR poses) are known. Hence, all points $p \in \mathbb{P}^{(m)}$ are assumed to be given in a joint world coordinate system (e.g. the coordinate system of the first segment). This allows for an easy integration of all point cloud segments into one joint point cloud which increases the local point cloud density. Additionally, many of the segments are largely redundant, depicting the same scene from a slightly different view point. Consequently, the labeling process is drastically sped up by labeling the joint point cloud. Unfortunately, the integration of all individual point cloud segments into a joint point cloud comes at the cost of a large increase in the number of points that needs to be handled at once. We cope with this issue by loading only a selected range of point clouds segments at once. To this end, the user needs to specify an offset sequence number and a number of segments to load from there on using the segment loader UI module in the bottom right corner of the tool's main window (see Figure 3.7). Another benefit of loading only a specified range of the data is that strong alignment issues won't show up in the labeling process. This is due to the fact that the local geometry will mostly only contain unobservable drift while global geometry cannot show up as the temporal range is limited.

As is common in semantic segmentation, a fixed set of labels is used to annotate the 3D points. Each label of such a label set has a unique name, a unique ID number, a predefined color and an optional description string. To offer more flexibility to the user, the unique ID numbers can be arbitrary integers and need not be consecutive. Internally, we use different unique IDs which are in fact consecutive unsigned 8-bit integers. While limiting the number of possible classes to 256, this helps keep the memory requirements of class color LUTs, index arrays, etc. at bay. Each label set needs to comprise one label which is used to initialize unlabeled points (referred to as the *default label*).

If available, we also leverage corresponding 2D image data of the scene (see Section 3.3.6 for details). The camera images are required to be undistorted and to include intrinsic calibration as well as a camera pose relative to the joint world frame of the point cloud. In our current data model, we assume that there is exactly one image per point cloud segment or no images at all. Obviously, these requirements hold for many aligned RGB-D sequences and for visual SLAM reconstructions but also datasets like Kitti include one image per LiDAR frame that shares a partial overlap with the LiDAR scan [GLU12].

We assume that all data but the point labels are immutable.

3.3.2 Persistence

Naturally, the point cloud data, the (optional) camera images, and the labels assigned to the points need to persist the lifetime of our annotation tool. To allow for short loading times of the requested segment range, we devised a file-based persistence model which is optimized for quick access to whole segments of the sequence.

Each point cloud sequence is stored in its own directory. The label set is specified in a JSON file `label_set.json` on the top level of this directory. An example of such a file showcasing the expected format can be found in Figure 3.8. A subdirectory `points` contains the points of each segment $\mathbb{P}^{(m)}$ in world coordinates in $|\mathbb{P}^{(m)}| \times 3$, 32-bit floating point NumPy arrays saved as `npy` files with their sequence numbers as a file names. Similarly, color and label for each individual point are stored in $|\mathbb{P}^{(m)}| \times 3$, 8-bit unsigned integer and $|\mathbb{P}^{(m)}| \times 1$, 8-bit unsigned integer NumPy arrays in the subdirectories `colors` and `labels`, respectively. If the subdirectory `cameras` is present, the tool expects to find subdirectories `cameras/`m`/` for each segment m containing the undistorted image file `image.png`, the camera pose matrix relative to the world coordinate system as a 4×4, 32-bit floating point NumPy array `pose.npy` and the projection matrix which is assumed to have the form $\begin{pmatrix} \boldsymbol{K} & \mathbf{0} \end{pmatrix} \in \mathbb{R}^{3\times4}$ as a 3×4, 32-bit floating point NumPy array.

```
{
  "name": "Cityscapes",
  "labels": [
    {
      "id": 0,
      "name": "Unlabeled",
      "color": [255, 255, 255]
    },
    {
      "id": 7,
      "name": "Road",
      "color": [128, 64, 128],
      "description": "Part of ground on which ..."
    },
    ...
  ],
  "default_label": 0
}
```

Figure 3.8: An excerpt from a `label_set.json` file

3.3.3 Rendering

In initial experiments with the VTK [SLM04] library for 3D visualization (version 8.1.1), we found the library to be too inefficient to render the points even for relatively small segment ranges. This manifests itself in notable drops of the rendering frame rate and strong input lag.

Consequently, we engineered a custom visualization module based on OpenGL 3.3 (the PyOpenGL Python wrapper, to be precise) which is able to handle large point clouds ($\geq$ 10M points) at decent frame rates. Virtual 3D objects to be rendered (such as labeled point clouds, camera frusta, coordinate axes, etc.) are represented by subclasses of the `Renderer3D` class with the `OpenGLRenderer` mixin. These subclasses manage all their OpenGL resources themselves (e.g. GPU buffers, VAOs, textures, etc.). They also define their OpenGL draw calls and the GLSL program to be activated during rendering. The `Visualization3D` class (which is a subclass of `QOpenGLWidget`) manages the drawing surface and the OpenGL context, compiles the GLSL programs once they are needed and triggers resource allocations and deallocations of the attached `Renderer3D` classes. To minimize the number of GLSL program switches on the GPU, we group all attached renderers by the GLSL program they use for rendering, activate the program on the GPU once, and then issue all drawing commands in that group.

For rendering 2D graphics (such as projections of the 3D points onto the 2D images), we initially used Qt's GraphicsView API but soon ran into similar per-

formance issues as in the 3D case. Hence, we extended our visualization API to 2D rendering by introducing Z indices and an orthographic camera model. The architecture is analogous to the 3D case (classes `Visualization2D`, `Renderer2D`, etc.).

A simple abstraction of the virtual camera model integrates the different conventions in most computer vision camera models (view direction: z-axis, down direction: y-axis) and OpenGL's camera model (view direction: negative z-axis, up direction: y-axis). Especially when working with images, the former model can be considered more intuitive which is why it is used throughout the system while the OpenGL convention used in rendering is hidden by the classes `PerspectiveCamera` and `OrthographicCamera`.

3.3.4 User Interface

The main window of the annotation tool comprises a main sidebar on the right, a main tool bar along the top of the window and a main content area which can show one of two *annotation panes* (see Figures 3.7 and 3.10).

The main sidebar contains the aforementioned segment loader and the *label picker* which shows all available labels in a list. Hovering the mouse cursor over a list entry reveals additional information about the corresponding label (such as the label description). We refer to the label in the selected list entry, which is additionally displayed above the list, as the *active label*. The active label is generally used to annotate points.

The tool supports the common key commands ctrl + S to save the current point labels and ctrl + Z / ctrl + Y to undo/redo label changes. A button for each of these key commands is available in the main toolbar.

The two annotation panes offer 3D (see Section 3.3.5) and 2D (see Section 3.3.6) annotation capabilities. Naturally, the 2D annotation pane is only available if suitable 2D image data is provided (as described in Section 3.3.1). It is possible to switch between the annotation panes by means of the keys 3 and 2 or the corresponding buttons in the main toolbar.

Each annotation pane has an own sidebar (on the left), toolbar (on top), and a central visualization canvas. The toolbars can be toggled by pressing the T key to maximize the screen space available for the visualization canvas. Furthermore, to leverage workstations with multiple monitors, it is possible to detach one of the annotation panes into its own window by pressing the D key. This conveniently allows for interleaved 2D and 3D annotation. The sidebar toggle and pane detach are also visualized by buttons in the corresponding annotation pane's toolbar.

3.3.5 3D Annotation

In the context of 3D annotation, there are two crucial capabilities which dominate the quality of a tool implemented for that purpose: interaction with the virtual

UI Action/Key Combination	Description
ctrl + S	Save the point labels of the currently loaded segments to disk
ctrl + Z and ctrl + Y	Undo/redo label changes
3 and 2	Open the 3D/2D annotation pane
T	Show/hide the sidebar of the current annotation pane
D	Detach the current annotation pane (i.e. show it in a separate window)

Table 3.1: General key combinations in the annotation tool

UI Action/Key Combination	Description
C	Switch to camera mode
B	Switch to brush mode
P	Show point/label colors
F	Show/hide current camera frustum

Table 3.2: Key combinations in the 3D annotation pane

camera and the tool used to select points to annotate. The following section describes how the respective components of our annotation tool are motivated and how they can be used in an annotation workflow.

The 3D annotation pane (see Figure 3.7) shows the currently loaded point cloud segments with two different modes of interaction. *Camera mode* is activated by pressing the C key and allows the user to adjust the camera position and orientation and by activating *brush mode* with the B key, the user can use a 3D brush to annotate points with the active label. Details about the different interaction modes can be found below.

The P key switches between the label colors and the point colors in the visualization and the F key toggles the visualization of the current 2D camera's viewing frustum (see Section 3.3.6).

Virtual Camera Interaction When labeling 3D data it is vital that the user is provided with an intuitive and fast way of moving the virtual camera through the scene. We find that camera interaction as implemented in visualization frameworks such as VTK or Pangolin is largely unfit, unintuitive, and tedious in our use case. This is why we devised custom strategies of interacting with the camera.

Primarily, the user must be able to orient and position the camera freely in the scene without much effort. To achieve this, dragging the mouse on the viewport with the left mouse button clicked rotates the camera about the optical center. One of the main nuisances in VTK's and Pangolin's camera interaction is that the camera often performs roll movements (i.e. rotations around the axis defined by the view direction) while the user alters the pitch and yaw angles. When

visualizing a real world scene, this roll movement is mostly unwanted, as there mostly is only one canonical down/gravity vector. Hence, while rotating, we fix the roll angle and only allow changes in the yaw and pitch angles. Altering the roll angle explicitly is possible by pressing the ⇧ key and dragging the mouse with the left button down. To change the position of the camera, we implement two complementary approaches. For one, pointing to a certain location of the viewport and scrolling with the mouse wheel moves the camera's optical center along the sight ray defined by that location. The "Camera Speed" slider in the toolbar on the left of the viewport can be used to adjust the speed at which the camera moves. This allows for easy scene traversal as the user solely needs to point to the desired destination. Additionally, the user can move the camera in the plane defined by the view direction by dragging the mouse over the viewport with the left mouse button down.

When annotating an object in a 3D scene, it is common to inspect it from multiple different view points to check whether all points of the object are indeed labeled correctly. To facilitate this, we added a *view point mode* to our camera interaction features. The view point is visualized by a semi-transparent sphere with a coordinate frame which visualizes the orientation of the world frame inside. Generally, the view point is always positioned at a certain distance from the camera center in the view direction. However, once view point mode is activated by holding down the ctrl key, the view point position is fixed and the camera will move around the view point. In this case, dragging the mouse with the left mouse button pressed will not rotate the camera about its own optical center but it will rotate the camera position around the view point while keeping the view point at a fixed distance. Again, we fix the roll angle of the camera. Scrolling the mouse wheel in view point mode alters the distance between the camera and the view point.

Occlusions are arguably among the most effective visual cues of depth perception. Unfortunately, as described in section 3.2.1, these can largely not be leveraged due to our dense geometry representation. Nevertheless, in many cases, the parallax effect caused by camera displacements can help overcome these issues (see Figure 3.1 for an example). This is why we introduced the shortcut W which triggers a small temporary horizontal displacement in the camera position (referred to as a *wiggle*).

3D Brush Tool As we have seen in Section 3.1 there are many different interaction strategies for 3D point cloud annotation. Due to its simplicity and accuracy, we chose to implement a brush-like tool in 3D. The brush itself is visualized as a semi-transparent sphere in the color of the active label which snaps to the point whose projection onto the viewport is closest to the current position of the mouse cursor (see Figure 3.9). Triggering the left mouse button changes the label of all points within the sphere to the active label while triggering the right mouse button changes the active label to the label of the point to which the brush tool

UI Action/Key Combination	Description
Left Mouse Button + Drag	Change the camera's view direction
⇧ + Left Mouse Button + Drag	Rotate the camera around its view direction
Mouse Wheel	Move the camera in the direction of the mouse cursor
Right Mouse Button + Drag	Shift the camera sideways, up, or down
ctrl + Left Mouse Button + Drag	Move the camera around the fixed view point
ctrl + Mouse Wheel	Move the camera towards/away from the fixed view point
V	Hide/show the view point visualization
W	"Wiggle" the camera (short horizontal displacement)

Table 3.3: UI actions in camera mode of the 3D annotation pane

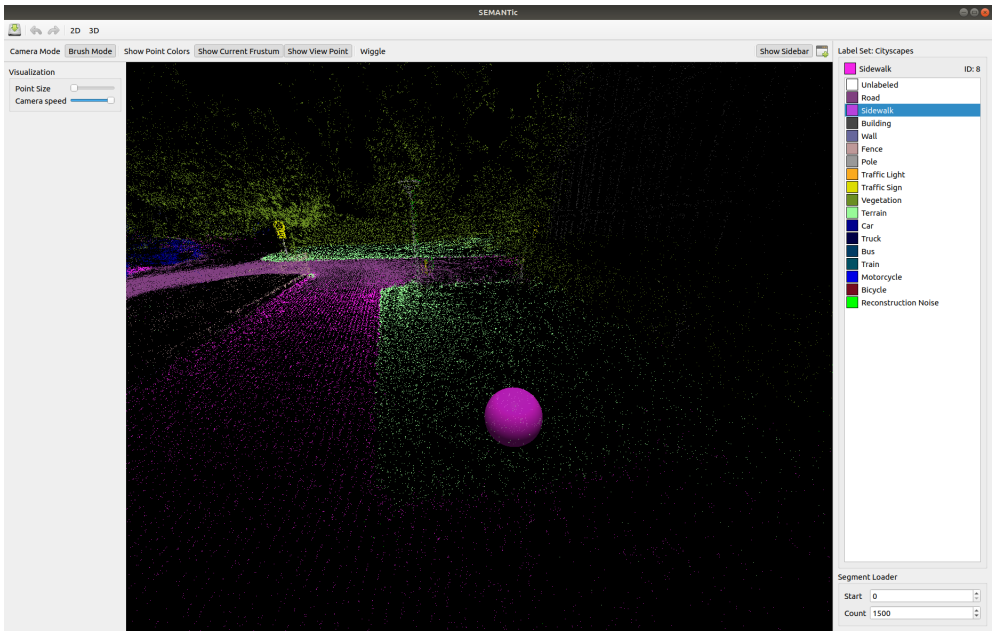


Figure 3.9: The annotation tool's 3D annotation pane with brush mode activated

snapped. Furthermore, scrolling the mouse wheel while holding down the ctrl key changes the radius of the brush tool.

To cope with the occlusion problems mentioned in Section 3.2.1, we define a search radius on the viewport, extract all points that project within that radius around the mouse cursor. Among those points we choose the one that is closest to the camera and let the brush tool snap to its 3D location. Currently, we switch the viewport radius to 5 px. This strategy makes our brush tool relatively robust to occlusion problems. However, if the geometry is excessively sparse, the brush

UI Action/Key Combination	Description
Left Mouse Button	Change label of points covered by the brush
Right Mouse Button	Change currently selected label to label of snapped point
ctrl + Mouse Wheel	Change the brush radius

Table 3.4: UI actions in brush mode of the 3D annotation pane

performs sudden unexpected movements which is why we disabled the possibility to annotate by continuously dragging the brush over the viewport (i.e. multiple clicks are required).

The vast number of points usually loaded in the annotation tool impose a severe computational burden onto the algorithms used to perform the brush annotation. This is due to the fact that it is difficult to run the algorithms on the GPU. To cope with the large amount of data involved, we carried out several important performance optimizations. Once brush mode is activated, we fix the position and orientation of the camera so that the points do not need to be reprojected after every camera move. We project all points onto the image plane of the viewport and build a ball tree with the 2D location to allow for fast radius queries. However, the fully projected 2D point positions depend on the size of the viewport which implies that a resize of the annotation tool's window would require the points to be reprojected and the ball tree to be rebuilt. Simply disabling the capability of the window to resize would be extremely inconvenient. Hence, we leverage a property of the pinhole camera model to allow for viewport size changes without the need to reproject. In the pinhole camera model, the projection $\boldsymbol{p} \in \mathbb{R}^2$ of a world point $\boldsymbol{x} \in \mathbb{R}^3$ onto the image plane is given by

$$\boldsymbol{p} = \boldsymbol{K}\Pi_0(\boldsymbol{R}\boldsymbol{x} + \boldsymbol{t}).$$

where $\left(\begin{smallmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0} & 1 \end{smallmatrix}\right)$ is the camera view matrix. For the brush tool, we are interested in all $\boldsymbol{x}$ which fulfill

$$\|\boldsymbol{K}\Pi_0(\boldsymbol{R}\boldsymbol{x} + \boldsymbol{t}) - \boldsymbol{p}_{\text{cursor}}\| \leq r.$$

Under the assumption that the matrix $\boldsymbol{K}$ scales both axes of the viewport isotropically without any shearing (which is almost always true in computer graphics) we can formulate the equivalent condition

$$\|\underbrace{\Pi_0(\boldsymbol{R}\boldsymbol{x} + \boldsymbol{t})}_{\boldsymbol{p}'} - \underbrace{\boldsymbol{K}^{-1}\boldsymbol{p}_{\text{cursor}}}_{\boldsymbol{p}'_{\text{cursor}}}\|_2 \leq \underbrace{\frac{r}{K_{11}}}_{r'}. \tag{3.1}$$

The $\boldsymbol{p}'$ do not depend on the viewport size (which plays a role in the calibration matrix $\boldsymbol{K}$). This means that we can build the ball tree using the image plane coordinates $\boldsymbol{p}'$ and query it with the modified cursor position $\boldsymbol{p}'_{\text{cursor}}$ and modified radius r' to obtain the same results without the need to reproject the points after

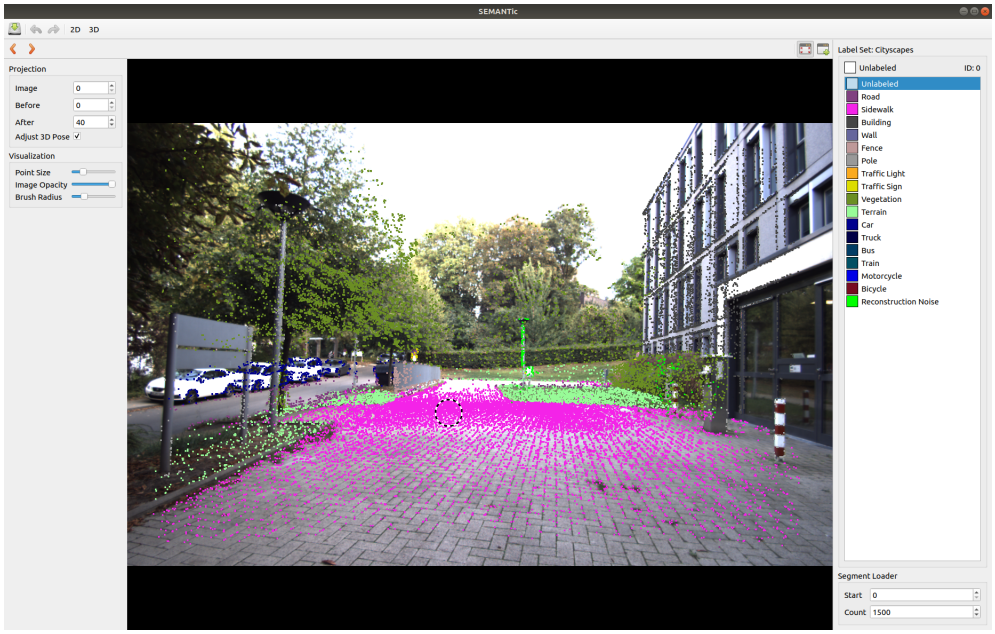


Figure 3.10: The annotation tool's 2D annotation pane with the 2D brush tool (in the center of the image)

a resize of the viewport. Once we obtained all points $\boldsymbol{x}$ which fulfill the condition in Equation 3.1, we let the brush tool snap to the one which has the smallest distance to the virtual camera. To label all points within the radius of the brush tool once the left mouse button is clicked, we use another ball tree on the 3D points which is constructed once a new range of point cloud segments is loaded.

3.3.6 2D Annotation

In section 3.2.2 we have seen that excessive sparsity in parts of the point cloud makes it difficult to determine the correct labels of the points. 2D images on the other hand can be considered dense representations of the scene which makes finding the correct labels easier in many cases. Hence, if 2D images and camera parameters are available, we leverage them in order to simplify the annotation process in many cases.

To this end, the tool offers the 2D annotation pane (see Figure 3.10) which can be used to step through the 2D images corresponding to the currently loaded point cloud segments (see Section 3.3.1) with the ← and → keys. Scrolling the mouse wheel zooms into the image and dragging the mouse with the right mouse button held down drags the image around.

Once a 2D image is loaded, we project the 3D points from a specified subset of the currently loaded point cloud segments onto it. The "Projection" group in the sidebar contains spin-boxes labelled "Image", "Before", and "After" which specify

UI Action/Key Combination	Description
Right Mouse Button + Drag	Change the visible image area
Mouse Wheel	Zoom into the image
Left Mouse Button + Drag	Label points with the active label in a brush style
ctrl + Mouse Wheel	Change the brush radius
⇧ + Mouse Wheel	Change the opacity of the image
← and →	Step through the available image sequence
P	Toggle the "Adjust 3D Pose" checkbox

Table 3.5: UI actions of the 2D annotation pane

the maximum number of segments before and after the segment corresponding to the 2D image that should be projected.

To label the projected points in the images, we can again resort to a large set of existing strategies for 2D annotation (some of which were presented in Section 3.1). However, most of these strategies (mainly polygonal selections) are designed to efficiently label images at the pixel-level (i.e. densely). We find that a simple circular brush tool is well suited for the sparse projections as it allows to select image regions coarsely without much effort. The tool is visualized by a circle that replaces the mouse cursor on the visualization canvas. Dragging the mouse with the left mouse key pressed changes the labels of all points within the circle to the active label. Again, scrolling the mouse wheel with the ctrl key held down changes the radius of the brush tool.

Changes to the point labels made in the 2D annotation pane are immediately displayed in the 3D annotation pane (and vice versa). This can be used to verify the 2D annotations in the 3D point cloud. The 3D annotation pane also visualizes the viewing frustum of the camera that captured the image visualized in the 2D annotation pane. Additionally, if the checkbox "Adjust 3D Pose" in the sidebar of the 2D annotation pane is checked, the pose of the virtual camera in the 3D annotation pane is changed to the pose of the camera that captured the image visualized in the 2D annotation pane. The checkbox can be toggled by means of the P key.

At times it is somewhat challenging to distinguish the points projected on the image from the image colors (e.g. the points on the trees in Figure 3.10). This is why we explicitly let the user control the opacity of the image in the background by means of the corresponding slider in the sidebar or by scrolling while holding down the ⇧ key. Moreover, increasing the point size using the appropriate slider in the sidebar might also help enhance visibility in some cases.

3.3.7 Install and Run

We recommend installing the annotation tool to a Python virtual environment. This is easily possible by navigating to the root of the tool's source code and executing pip install . within the virtual environment. As we include a suitable `setup.py` file, pip conveniently resolves all dependencies automatically. After installing the tool it can be started via semantic.py path/to/point/cloud/sequence.

3.4 Evaluation

3.4.1 SLAM point clouds

While developing the features of our annotation tool, we continuously tested the tool on several SLAM point clouds, including the one presented in Figure 2.5d. Most of the observations in Section 3.2 were made during these test runs.

Once all features presented in Section 3.3 were implemented, we chose to evaluate the tool in a final test run. In this test run, we also used SLAM point clouds from two different data sources to test how well our tool generalizes to point clouds that differ from those generated by our setup.

Test Data One of the point clouds was generated from the first 2000 rectified stereo color images in Kitti's odometry 0 sequence (i.e. a 2:20 min video because Kitti is recorded at 10 Hz). After running DSO with the input data, we obtained depth estimates from 1945 keyframes. We applied our geometric filtering pipeline without the neighborhood-based filtering algorithm (see Section 2.6.1). Examples from the resulting point cloud can be found in Figures 3.2, 3.3, 3.4, 3.5 (upper image), and 3.6. For the test run, we restricted ourselves to annotating the first 50 keyframes. These arise from the first $\approx$ 46 meters of the car's trajectory and they contain 50126 points.

The other point cloud which we dub the *RWTH point cloud* originates from a custom sequence recorded with our stereo rig in Aachen. It comprises 4878 stereo frames recorded at 20 Hz (i.e. a 3:04 min video) which resulted in 1702 keyframes once forwarded through DSO. Here, we applied the full geometric filtering pipeline to get rid of geometric noise. Figures 3.5 (lower image), 3.7, 3.9, and 3.10 show parts of the point cloud. The first 170 keyframes were used to test the annotation tool. These comprise 132553 points and they span a trajectory of $\approx$ 35 meters length. Note that we did not annotate this point cloud from scratch but we used an automatic initialization of the point labels (see Section 4.5).

Label Set We chose to use a label set which is essentially a reduced version of the Cityscapes labels with an additional noise label. It consists of the: Unlabeled, Road, Sidewalk, Building, Wall, Fence, Pole, Traffic Light, Traffic Sign, Vegeta-

tion, Terrain, Car, Truck, Bus, Train, Motorcycle, Bicycle, and Reconstruction Noise.

2D Image Annotation We emphasize that the inclusion of 2D image data is of immense value for the annotation workflow. As shown in Figures 3.2, 3.3, and 3.4 it is largely impossible to determine the correct semantic label for many of the points in the sparse point clouds. Hence, we believe that our 2D annotation feature is crucial for SLAM point clouds.

Density It should be noted that the Kitti point cloud is much sparser than the RWTH point cloud. It contains $\approx 63\%$ less points while spanning a trajectory that is $\approx 31\%$ longer than the other point cloud. This can be attributed to the higher frame rate and the lower speed at which the cameras moved.

We observed that the time required to label a point cloud is mostly increased by an increasing geometric complexity and an increasing spatial extent of the scene. In other words, a denser point cloud representation of the same scene will likely not take more time during annotation. On the contrary, in accordance with Section 3.2.2, it is often easier to annotate a denser point cloud resulting in faster annotation times. Hence, we would argue that one should strive for high recording frame rates to obtain denser point clouds.

Annotation Time Annotating the Kitti point cloud from scratch took 4:05:06 hours. While this seems to be extremely slow, we suspect that the long annotation time is due to the fact that the way in which we annotated the point cloud did not play to the strengths of the tool. Approximately 29% of the 66841 label changes were performed in the 3D annotation pane. This percentage is arguably too high, as there are many nuisances to direct 3D annotation. In later attempts to annotate the same point cloud this percentage dropped considerably (e.g. $\approx$ 13%, see 4.5) which (together with automatic label initialization) caused massive speed-ups in annotation time. Moreover, as this was the first large sequence we annotated with the full feature set of our tool, we think that the learning curve also played an undeniable role in increasing the annotation time.

Geometric Noise Even though many noise and outlier points are already discarded by our geometric filtering pipeline, there are some that remain in the point cloud. As described in Section 3.2.3, the most elegant way to handle these points is to introduce a special noise label. However, as mentioned in Section 3.2.3, we find that it is extremely tedious to identify and label the outlier points. Our experiments show that it is virtually impossible to identify noise or outlier points in the 2D annotation pane because the geometric error naturally only manifest themselves in the points' depth values which is discarded in a 2D view. Even in the 3D pane it is difficult to identify and pick outlier points in 3D which is largely due to their spatial proximity to correct points. This nuisance is amplified by

the fact that our 3D brush is largely unsuitable to annotate these points. After 39:02 min we aborted the annotation of the RWTH point cloud (with automatically initialized labels) because many noise and outlier points hindered a fluid annotation process.

Usage Recommendation While testing the annotation tool, we developed a workflow which plays to the strengths of our annotation tool when annotating SLAM point clouds. We found that the fastest way to annotate a point cloud sequence is to perform as many annotations as possible in the 2D annotation pane. Hence, it is advisable to step through the 2D frames of the point cloud following the temporal order, projecting as many points as possible onto the image and annotating all points that do not suffer from occlusion problems (i.e. the ground plane, sidewalks, walls of houses along the street). Occluders can also be annotated in 2D by first annotating the occluder and then correcting the (now wrongly labeled) occluded points from a different view point. Finally, the 3D annotation pane can be used to correct remaining errors and to label noise/outlier points.

3.4.2 Aligned LiDAR point clouds

We also observe that the annotation tool generalizes well to LiDAR point clouds, as a coworker in another project of our research group uses it to label ICP-aligned LiDAR scans from the Kitti odometry dataset. In the current approach, the point labels are partially initialized from 2D ground truth annotation in images, then coarsely corrected using the annotation tool and finally refined using superpoint clustering.

The aligned LiDAR sequences are much denser than SLAM reconstructions. While this helps during annotation, we observe that activating the brush tool in the 3D annotation pane takes several seconds which hinders the annotation process as it is the only appropriate tool to label these point clouds. However, once the brush tool is loaded it functions smoothly.

3.5 Future Work

During the development and evaluation process of the annotation tool, we found and devised several promising features and potential fixes for open problems which were not implemented due to the limited implementation time available for this thesis. Nevertheless, we sketch the main points of these ideas here as we believe that they are promising subjects of future work.

Improved 3D Brush A major downside of the 3D brush tool developed in this thesis is its limited efficiency which arises from the CPU-based implementation.

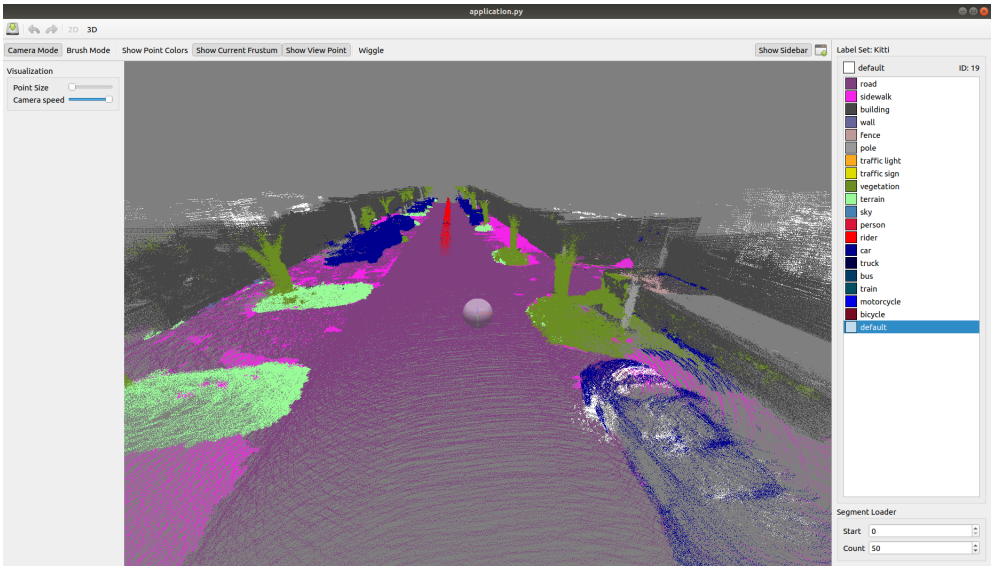


Figure 3.11: A partially labeled, ICP-aligned sequence of LiDAR point clouds in the annotation tool (courtesy of Kushal Sharma)

The need to lock the camera position and the long time required to activate the brush for large point clouds disturb a fluid annotation workflow.

We believe that a GPU-based implementation of the 3D brush algorithm would likely result in a giant speedup. There are two simple ways to realize such an implementation.

One could create an OpenGL frame buffer with an additional color attachment which is not used to store actual color values but to store the index of the point visible in the corresponding pixels. The (depth-wise closest) point under the mouse cursor can then be found by searching through the relatively small number of point indices stored in the vicinity of the mouse cursor in the additional color attachment. Filling the additional color attachment would not introduce much computational overhead as this can be realized by one additional assignment in the fragment shader of the corresponding GLSL program.

Another way of searching the points that are projected in the vicinity of the mouse cursor is to perform a parallelized linear search over all points by means of an OpenGL compute shader or an OpenCL kernel (e.g. via *PyOpenCL*).

Interaction Clipping A promising alternative approach to 3D annotation is to use 2D selection tools (such as polygons or lasso tools) on the viewport. This strategy is realized in the SSE Labeling Tool by Hitachi's Automotive And Industry Lab mentioned in Section 3.1. However, (the current version of) their tool ignores occlusion-related problems.

To cope with the occlusion problems, one could provide an adjustable *interaction clipping plane*, i.e. a maximal depth value which clips away points that should not be affected by selections on the viewport. In order to provide an in-

tuitive workflow, the interaction clipping plane should be visualized, for example by a semi-transparent quadrilateral parallel to the image plane that spans the entire viewing frustum.

Masking Currently, selecting a point with the brush tools in the 2D and 3D annotation panes directly changes its label. We believe that decoupling the point selection and labeling process might speed up the annotation process. The SSE Labeling Tool by Hitachi's Automotive And Industry Lab and the labeling tool for the semantic3d.net dataset [HSL+17] realize this strategy.

Points can be selected using any of the selection tools presented in this chapter (e.g. 2D polygon/lasso, 2D brush, 3D brush, etc.). Afterwards, points can be added to and removed from this selection by means of the same tools. Subtractive coarse-to-fine selection can be used to conveniently solve the occlusion problem while relieving the computational burden of the selection algorithms as the set of points that is relevant for the algorithms becomes successively smaller.

Bounding Box Annotation Several of the annotation tools presented in Section 3.1 use a cuboid which is rotatable, resizable and movable to select 3D points for annotation. While we believe that this strategy is not flexible enough to be used for general annotation, it seems to be well-fit to select noise and outlier points, especially those above and below the ground plane of the scene. Automatically fitting the bottom of the cuboid to the ground plane at a specified position would further simplify the process.

Web-Based Implementation The fact that the current annotation tool must be installed on the user's computer makes it difficult to crowdsource annotation jobs, e.g. via platforms similar to Amazon Mechanical Turk. This is why a web-based reimplementation (i.e. using HTML, JavaScript, WebGL, etc.) of our tool might prove to be useful in the future.

4 Automatic Label Initialization

Annotating a large point cloud manually and from scratch is a time consuming and yet, in many cases, relatively simple task. Hence, one might consider manual annotation for 3D semantic segmentation as a waste of human intelligence. Fortunately, there are many algorithms that can be applied to assist and thus speed up manual annotation in a semi-automatic fashion. A promising approach is to use an existing semantic segmentation approach to initialize the labels of all points and to correct the errors in the automatic initialization. Modern semantic segmentation approaches (especially in 2D) achieve decent performance once a sufficient amount of training data is available. The errors made by those models can be considered difficult parts of the segmentation problem (e.g. finding accurate semantic boundaries) which require human supervision to be performed accurately. Hence, this strategy leverages human intelligence much more efficiently.

4.1 Related Work

Due to the vast time consumption of manually annotating 2D or 3D data for semantic segmentation, numerous approaches to semi-automatic labelling have been proposed. For the sake of brevity, we only present a few representative examples.

Technically, the grouping-based annotation interfaces mentioned in Section 3.1 are semi-automatic methods. [Lab] and [CUF18] use superpixels for grouping in 2D. Conveniently, superpixels also simplify the difficult exact boundary selection as they snap to visual boundaries (e.g. edges) in the image. [PCN17] also use superpixels to annotate images for 2D semantic segmentation. However, in their method, the user only needs to annotate one superpixel because afterwards an algorithm based on decision trees boosted with AdaBoost successively selects neighboring superpixels that are added to the selection. To this end, they extract

features from the 2D images and from corresponding stereo or LiDAR point clouds.

[Yan] applies the Euclidean clustering algorithm proposed in [Rus09] on LiDAR point clouds. [MAZV17] let the user select so-called *control points* in the point cloud. Given a set of control points, their algorithm computes a shortest-path tree from the k-NN graph of the point cloud. Every control point is connected to the root of the shortest path tree and hence is the root of a subtree. The subtrees are used as annotatable groups of points in the point cloud.

Similarly, the label propagation used in the SUN3D annotation tool can be considered a semi-automatic.

Human-in-the-loop and active learning approaches, i.e. approaches where a human annotator successively provides corrections of erroneous predictions of an ML model which can then used to retrain the model, are another important example of semi-automatic labeling strategies. [Aut] uses a so-called *active learning loop* for 2D bounding box annotation. Here, a detector is trained while manually annotating ground truth boxes. Corrections of the detector's output can be used to further train the algorithm. [HSL+17] apply a human-in-the-loop strategy for 3D semantic segmentation. In their method, a user first picks several points of an object to be annotated and "a simple model" is fitted to these points. Afterwards, the errors of this model are removed from the selection and the model fitting is repeated. This process is iterated until the object is correctly selected.

[CFYU14] is a largely automated method for 2D segmentation of cars in street scenes based on weak 3D supervision. Given 3D ground truth bounding boxes, the method performs a foreground/background segmentation in the image patch defined by the projected bounding box via an MRF. Stereo or LiDAR depth is used to identify which 2D pixels in the patch lie inside or outside of the bounding box and GMM color models are learned for the foreground and background classes using this information. Additionally, another unary potential in the form of GMM depth models for foreground and background pixels is introduced. 3D CAD models of cars are aligned with the bounding boxes and their 2D contours are transformed into a signed distance field. The values of the signed distance functions at the pixel locations are employed as further unary potentials An Ising prior and a contrast sensitive Potts model on the image data are chosen as pairwise potentials.

The annotation interface proposed in [AUF18] can be used to label images with instance-level semantic annotation. To this end, an instance segmentation method is applied to generate segment proposals and to automatically initialize their semantic labels. Afterwards, correction tools to add/remove segments, to change labels of segments, and to change the depth ordering of overlapping segments are provided.

4.2 2D-3D Label Transfer

Applying a 3D semantic segmentation approach to initialize the labels is currently not an option. This is largely due to the fact that virtually no adequate training data is available at the moment. In general, there are too few large-scale outdoor datasets for 3D semantic segmentation available at the moment (as mentioned in Section 1.3). Specifically, point clouds obtained from visual SLAM systems wildly differ from LiDAR, RGB-D, or virtual datasets in terms of visual appearance, point density and point distribution. To our knowledge, no datasets comprising annotated visual SLAM reconstructions are currently available. Both of these points strongly hinder generalization to our data.

Image data on the other hand does usually not exhibit such strong appearance variations. Moreover, there many large-scale 3D datasets for 2D semantic segmentation available (see Section 1.1.1). These prerequisites promise decent generalization of 2D semantic segmentation to data from other sources.

Consequently, we leverage the fact that visual SLAM point clouds originate from image data by applying a 2D semantic segmentation method to initialize the labels of the 3D points via simple 2D-3D label transfer. To be precise, we know that each keyframe of a point cloud obtained from DSO gave rise to a known subset of the 3D points. Hence, we obtain a pixel-level semantic labeling of the rectified keyframe images by means of an arbitrary semantic segmentation approach. The 3D point labels are then initialized by querying the 2D semantic labeling at the locations to which the points project in the keyframe. They use 3D ground truth bounding boxes around the car to extract the part of the

4.3 DeepLab

We chose to test our label initialization approach with DeepLab v3+ as a 2D semantic segmentation method. DeepLab v3+ [CZP+18] is a powerful, well-known FCN architecture for semantic segmentation with strong performance on the Cityscapes [COR+16] benchmark[1] (overall 7th best, 3rd best among the methods with code available). The project is well-maintained and frozen inference graphs trained on the Cityscapes dataset are available. In the following we will pinpoint several important characteristics of DeepLab v3+.

Architecture In principle, DeepLab v3+ is an encoder-decoder network. The full architecture is depicted in Figure 4.1.

The encoder is essentially the DeepLab v3 [CPSA17] network. A modification of a common backbone architecture such as Xception [Cho17], ResNet [HZRS16], or MobileNet-v2 [SHZ+18] is applied as a feature extractor ("DCNN" in Figure

[1] https://www.cityscapes-dataset.com/benchmarks/#pixel-level-results, accessed 18.10.18

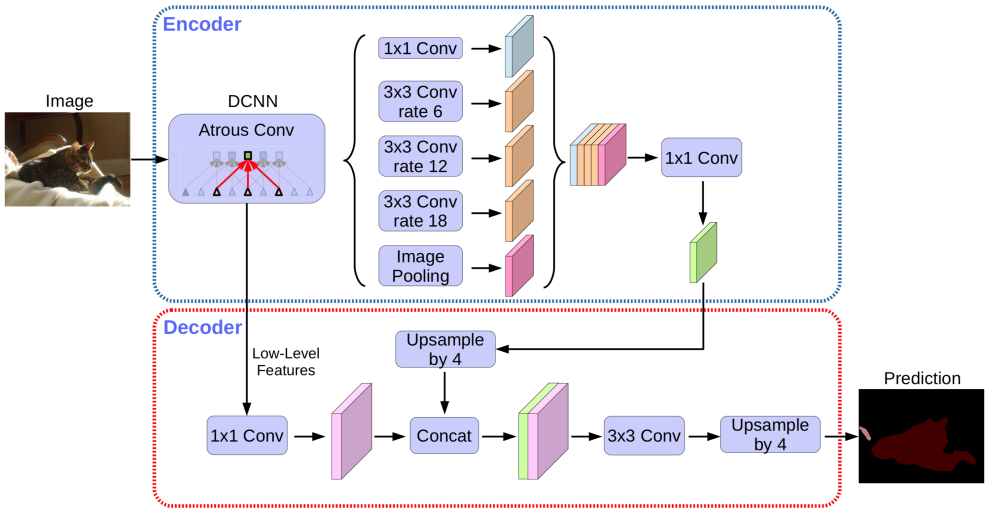


Figure 4.1: The DeepLab v3+ architecture (source [CZP+18])

4.1). Afterwards, the features are fed to the Atrous Spatial Pyramid Pooling Module (explained below). The factor with which an image is downsampled in the encoder is referred to as the *output stride* (i.e. if the input image is downsampled by a factor of 16, the output stride is 16).

To enhance the segmentation masks, especially around the borders, a decoder is applied. Low level image features from an early layer in the backbone network are concatenated with the bilinearly upsampled encoder features and run through several layers of 3x3 convolutions before a final bilinear upsampling operation.

Atrous Convolutions Typically, CNNs downsample the input image by applying pooling layers or strided convolution to reduce memory requirements and computation time and to widen the receptive field of convolutions without increasing the number of learnable parameters. A downside of this strategy is that fine-grained details are largely discarded which are, however, of great importance to obtain sharp semantic boundaries. This is why DeepLab replaces several downsampling operations with so-called *atrous convolutions*. An atrous convolution is essentially a regular convolution with a dilated filter kernel (hence they are also referred to as *dilated convolutions*). The dilation is performed by introducing $r - 1$ zeros between the original kernel values along each dimension (see Figure 4.2). Atrous convolutions perform dense feature extraction with a large receptive field while keeping the number of parameters and the computation time at bay. However, the memory requirements do not decrease which means that atrous convolutions can not completely replace strided convolutions/pooling.

Depthwise Separable Convolutions To reduce the computational cost and number of parameters of a convolution operation, DeepLab v3+ applies *depthwise separable convolutions*. Assume a standard convolution operation applies

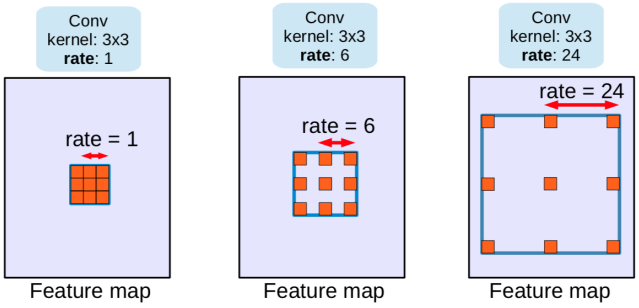


Figure 4.2: Atrous/dilated convolutions applied at different rates (source [CPSA17])

D' filters with the dimensions $W \times H \times D$ to each spatial location of a feature volume with D depth channels. An equivalent depthwise separable convolution first applies one spatial convolutions with kernel size $W \times H$ to each of the input channels individually followed by D' 1×1 convolutions. Hence, a regular convolution has $\mathcal{O}(W \cdot H \cdot D \cdot D')$ learnable parameters and needs as many arithmetic operations per input location, which reduces to $\mathcal{O}(W \cdot H \cdot D + D \cdot D')$ in the case of depthwise separable convolutions. As D and D' are usually large, this is an immense improvement over regular convolutions. In DeepLab v3+ some of the spatial convolutions in depthwise separable convolutions are realized as atrous convolutions.

Atrous Spatial Pyramid Pooling In order to efficiently process information acquired at multiple different scales, DeepLab v3+ comprises the *atrous spatial pyramid pooling layer* (see right part of the encoder in Figure 4.1). Here, a 1x1 convolution and three 3x3 atrous convolutions at different rates are applied to a feature volume to extract features at different scales. Additionally, the global context is incorporated by means of global average pooling followed by a 1x1 convolution and upsampling to the input feature volume. All these feature maps are concatenated into one output volume whose depth dimension is then reduced by means of a 1x1 convolution.

4.4 Implementation

In our implementation of the label initialization strategy proposed in Section 4.2, we use a TensorFlow [AAB+15] frozen inference graph of DeepLab v3+ with an Xception 65 backbone. The network backbone was pretrained on ImageNet [DDS+09] and the whole instance of DeepLab v3+ was further trained on Cityscapes [COR+16]. The encoder has an output stride of 8 and the decoder

has an output stride of 4. Further information and a download link for the frozen inference graph are available in the "TensorFlow DeepLab Model Zoo"[2].

By trying several different input image resolutions, we found that resizing the images to a width of 1026 pixels while keeping the aspect ratio yielded good segmentation results for both the Kitti images and the images recorded with our recording setup. The output segmentation labels are then rescaled to the original image size by means of nearest neighbor interpolation.

When initializing the points, all labels that do not belong to the reduced Cityscapes label set presented in Section 3.4 are mapped to the default label (i.e. they are considered unlabeled).

4.5 Evaluation

Quantitative Evaluation In order to evaluate the performance of our simple 2D-3D label transfer initialization strategy quantitatively, we used the manually annotated keyframes of the Kitti point cloud from Section 3.4 as the ground truth. In our experiment we initialized the labels automatically and used our annotation tool to correct the errors as good as possible. After 1:23 hours of manual corrections, we did not notice any further errors in the annotation. Table 4.1 reports the standard performance metrics applied to analyze the correctness of a semantic labeling. These were computed using the fully manual annotation as the ground truth. The remaining errors visible in mean accuracy and mean IoU as lie in an acceptable margin of human error. The human error is largely due to ambiguities in the choice of the labels (e.g. due to the sparsity of the point clouds as mentioned in 3.2.2). One can observe that classes with smaller relative frequencies (Wall, Fence, Pole, Terrain, Noise) tend have lower per-class accuracy/IoU scores which seems to support this thesis. Ambiguities, especially along the semantic boundaries, make it difficult to reproduce a manual segmentation perfectly.

The bad scores of the noise labels prove our point from 3.4 that labeling noise and outlier points is difficult and error prone.

[CFYU14] report that the human labeling accuracy of capable Amazon Mechanical Turk annotators while segmenting cars in images is approximately 86 % IoU. With the arguments from Section 3.2 in mind, one might conclude that our comparable accuracy in the much harder task of full-scene 3D semantic segmentation is relatively low.

A quantitative argument about the usefulness of our automatic label initialization can be made by means of the overall accuracy and frequency weighted IoU metrics [LSD15]. While mIoU and mAcc are well suited to evaluate the performance of semantic segmentation methods, they over-penalize errors in less

[2] https://github.com/tensorflow/models/blob/master/research/deeplab/g3doc/model_zoo.md, accessed 30.08.18

		Unlabeled	Road	Sidewalk	Building	Wall	Fence	Pole	Traffic Light	Traffic Sign	Vegetation
GT	RF	0.1	23.5	6.9	28.4	0.1	0.5	0.8	0.0	0.1	21.6
init.	RF	0.5	25.6	5.1	28.1	0.03	0.1	0.4	0.01	0.02	24.0
	Acc	0.0	99.8	54.0	89.5	29.0	6.0	37.6	-	0.0	93.6
	IoU	0.0	91.2	44.6	81.7	23.7	5.4	34.5	0.0	0.0	79.5
corr.	RF	0.01	23.6	7.2	29.3	0.1	0.6	0.8	0.0	0.2	20.9
	Acc	0.0	99.8	93.5	98.5	87.1	81.7	86.9	-	100.0	94.4
	IoU	0.0	99.0	83.4	93.9	77.1	57.7	83.1	-	73.7	92.1

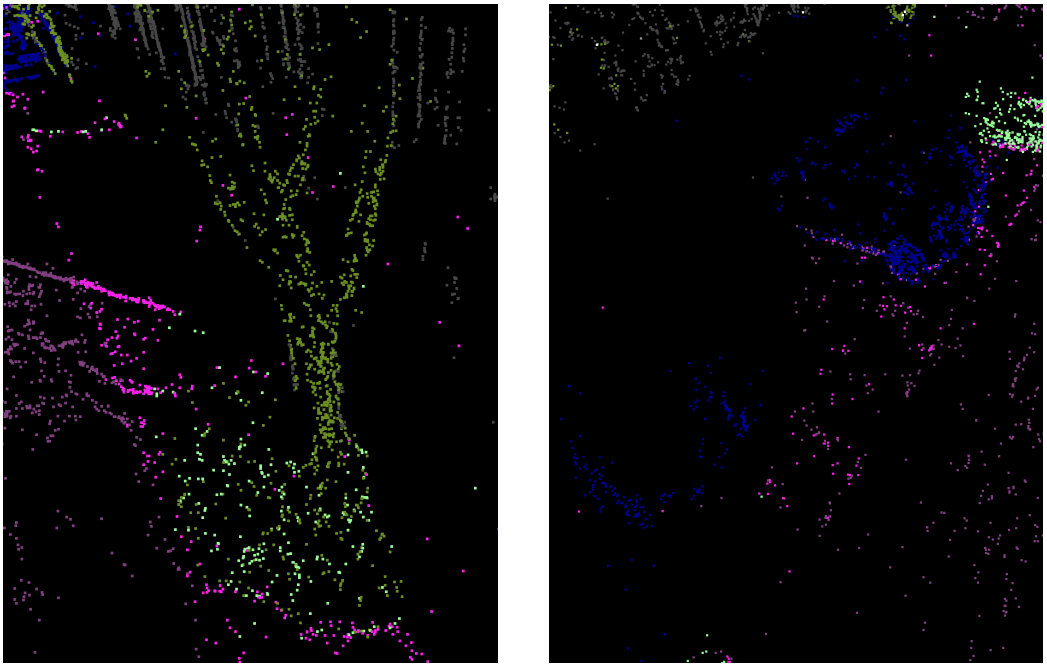
		Terrain	Car	Truck	Bus	Train	Motorcycle	Bicycle	Noise	Mean
GT	RF	6.7	11.2	0.0	0.0	0.0	0.0	0.0	0.6	-
init.	RF	4.0	12.1	0.0	0.0	0.0	0.003	0.003	0.0	-
	Acc	58.4	95.8	-	-	-	-	-	0.0	56.4
	IoU	54.6	85.6	-	-	-	0.0	0.0	0.0	38.5
corr.	RF	5.6	11.3	0.0	0.0	0.0	0.0	0.0	0.4	-
	Acc	86.2	97.5	-	-	-	-	-	58.2	92.6
	IoU	82.8	94.7	-	-	-	-	-	55.1	83.7

Table 4.1: Per-class accuracies (Acc), IoUs and relative frequencies (RF) as well as mAcc and mIoU for the first 50 keyframes of the Kitti point cloud (see Section 3.4). "GT" is the manual annotation from Section 3.4, "init." denotes the automatic label initialization and "corr." is the semi-automatic annotation. All values are given as percentages. "-" among the per-class accuracies means that the value could not be computed because a division by zero occurred.

	Overall Accuracy	Frequency Weighted IoU
init.	87.5	78.8
corr.	96.1	93.1

Table 4.2: Overall accuracy and frequency weighted IoU computed from the values in Table 4.1. All values are given as percentages.

frequent labels. In theory, correcting errors for classes with fewer points takes less time than correcting a similar error for much larger classes. This is why overall accuracy or frequency weighted IoU are more adequate measures when quantifying the increase in labeling efficiency due to the automatic initialization. The values for our particular experiment are reported in Table 4.1. Especially



(a) The "vegetation" labels (dark green) of the tree bleed into the "terrain" labels (light green) around the trunk

(b) "Road" labels (purple) corrupt the sidewalk labels (pink) at the side of the road

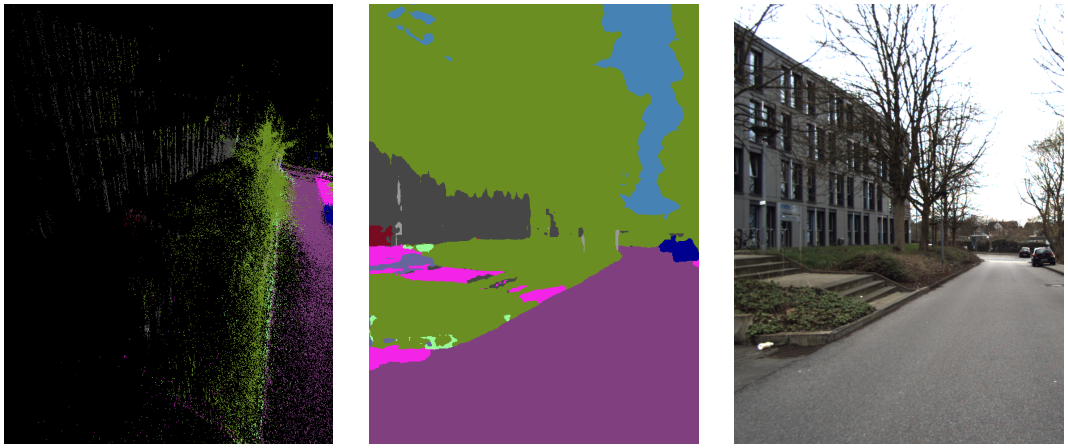
Figure 4.3: The missing temporal consistency in the 2D segmentation labels of the individual keyframes causes label noise

the gentle increase in overall accuracy (less than 10 %) can be seen as proof that many of the automatically initialized points are indeed correct.

Moreover, only 6034 label changes (5242 in the 2D pane and 792 in the 3D pane) of the 50125 point labels were necessary to obtain the final labeling, which is a decrease of approximately 91 % compared to the fully manual annotation process. Also, the 66 % decrease in annotation time can mainly be attributed to the automatic initialization. Nevertheless, advanced experience in using the annotation tool and a slightly optimized annotation strategy also contributed to this result.

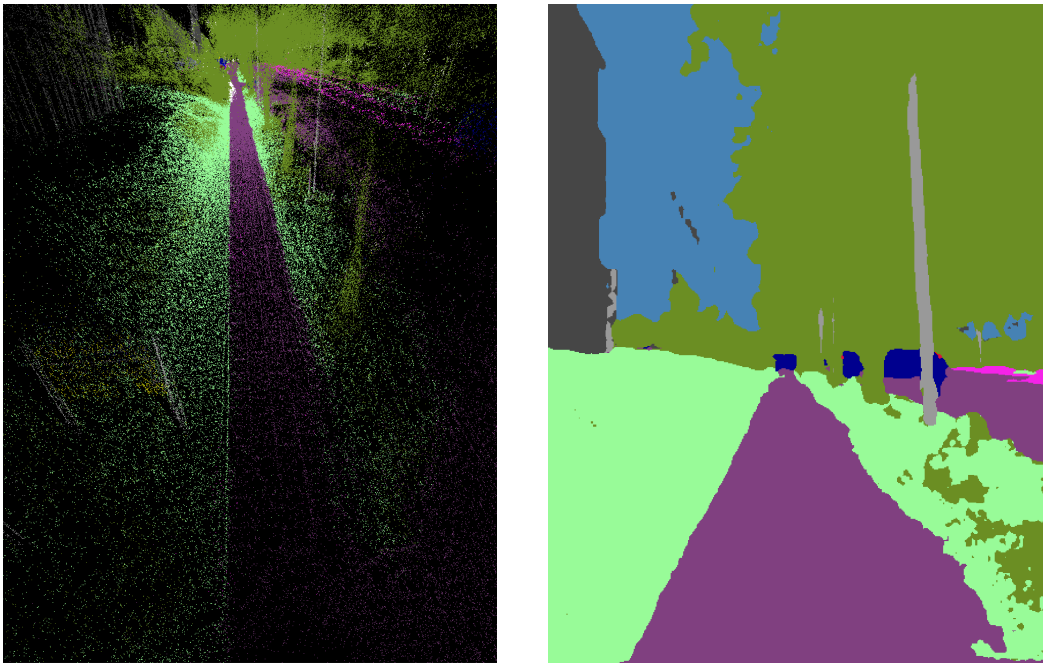
Our results indicate that our automatic label initialization method is a simple yet effective means to minimizing the annotation time substantially.

Qualitative Evaluation As conjectured above, the manual corrections were largely performed along semantic boundaries which can be seen as the most difficult cases of semantic labeling. Additionally, the fact that the keyframe images are input to the semantic segmentation method individually (i.e. without enforcing any temporal consistency) causes noise in some difficult cases (see Figure 4.3). Sometimes, DeepLab's segmentation masks exceed the object boundaries



(a) (b) (c)

Figure 4.4: Inaccurate segmentation masks cause projective artifacts (i.e. "vegetation" labels on the wall of the building)



(a) (b)

Figure 4.5: The sidewalk is confused with a road (labels should be pink)

which may lead to projective artifacts along occlusion boundaries (see Figure 4.4). Moreover, the part of the ground plane below the camera is nearly always labeled "road", even if the child stroller was positioned on the sidewalk in parts of the RWTH sequence (see Figure 4.5). This is most likely due to the priors introduced by training DeepLab on Cityscapes (the sequences are acquired from a car driving on different roads).

4.6 Future Work

Again, some of the ideas we developed to cope with the problems described above could not be implemented due to the limited development time and thus they remain as future work. To improve the quality of the automatic initialization, we propose two simple extensions.

Fully Connected CRFs In order to improve the spatial (and temporal) consistency of the label initializations, one could use the softmax activations of DeepLab as the input to a fully connected CRF with Gaussian edge potentials [KK11] and choose the edge potentials based on 2D/3D geometric information, color similarities, etc..

Label Substitution Errors like similar to the one presented in Figure 4.5 can be efficiently corrected by means of a label substitution tool. As the boundary of the sidewalk is sharp and correct, one simply needs to select the sidewalk coarsely (e.g. with a polygonal or lasso tool) and then substitute all road labels in the selection with sidewalk labels.

Clustering and Control Points It may be beneficial to implement further semi-automatic labeling methods on top of the automatic label initialization. Using clustering approaches to group points which are then annotated with a single click is a potential speed-up for our labeling tool. Similarly, adapting the sparse control point algorithm from [MAZV17] seems like a promising addition to our annotation tool.

Pretraining If only a limited amount of data is available, it is a common strategy to pretrain (2D) CNNs (or just their backbone networks) on ImageNet [DDS+09]. Unfortunately, a database at the scale of ImageNet is not available for 3D data.

In Section 4.5 we have seen that a significant fraction (87.5 %) of the points is already labeled correctly by our automatic initialization strategy. Hence, we can generate vast amounts of 3D data labeled with a relatively low quality. Despite the low quality, these data already contain valuable information about scene semantics and 3D geometry. Hence, we think that it is worth investigating whether our automatically initialized data can be used to pretrain deep learning methods for 3D semantic segmentation.

5 Conclusion

In this thesis, we studied the applicability of direct visual SLAM systems to training data generation for 3D semantic segmentation. To our knowledge this is a largely unexplored area of research with large potential, especially in the context of autonomous driving and driver assistance. The main practical contribution of the thesis is the prototype of a system to generate annotated 3D point clouds for 3D semantic segmentation. The main components of our system include

- versatile recording hardware based on calibrated stereo cameras with optional IMU integration which can record high-resolution stereo image sequences at a high frame rate,
- a software pipeline leveraging a visual SLAM system to generate large 3D point clouds from calibrated stereo image sequences with a sufficiently high frame rate,
- an annotation tool for 3D semantic segmentation which is tailored to temporal sequences of aligned point clouds, and
- a simple yet effective algorithm to initialize the semantic labels of a 3D point cloud obtained from a visual SLAM system automatically which aims at reducing the annotation time.

As our approach tackles training data generation for 3D semantic segmentation from end to end, we were able to gain valuable insights at all stages of the process. The central challenges and problems encountered during the development process at each of these stages were thoroughly scrutinized and elaborated upon in the respective chapters of the thesis. Solutions to some of the problems have already been woven into our pipeline.

Conceptually, there is one important downside to our approach. The visual SLAM system as well as various other components in our pipeline assume the scene recorded by the cameras to be static. While we already argued that training

data for static parts of the scene is still of value, we still need to obtain dynamic 3D training data which hints at the fact that our approach only solves a part of the overall problem.

As the scope of this thesis is limited, we could only scratch the surface of the topic at hand. However, our work hints at the fact that visual SLAM reconstructions are a valuable addition to LiDAR or RGB-D data in semantic segmentation, especially with the desperate need for annotated 3D outdoor datasets in mind. Moreover, while being far from production-ready, we think that our system is a good base for future research.

5.1 Future Work

Sections 2.7, 3.5 and 4.6 already gave pointers to future work concerning the respective parts of our system. However, we think that the most important direction of future work is to run experiments that investigate whether the data generated by our system is indeed useful for deep learning methods in 3D semantic segmentation.

Particularly, we conjecture that the characteristic noise in the visual SLAM point clouds might also increase the robustness to noise while decreasing the tendency to overfit the data if visual SLAM data are used jointly with LiDAR data.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[AL] Hitachi Automotive and Industry Lab. Semantic Segmentation Editor: SSE. https://github.com/Hitachi-Automotive-And-Industry-Lab/semantic-segmentation-editor. Accessed: 19.10.2018.

[ASZ+16] Iro Armeni, Ozan Sener, Amir R. Zamir, Helen Jiang, Ioannis Brilakis, Martin Fischer, and Silvio Savarese. 3D Semantic Parsing of Large-Scale Indoor Spaces. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[ASZS17] Iro Armeni, Sasha Sax, Amir R. Zamir, and Silvio Savarese. Joint 2D-3D-Semantic Data for Indoor Scene Understanding. *arXiv preprint arXiv:1702.01105*, 2017.

[AUF18] Mykhaylo Andriluka, Jasper RR Uijlings, and Vittorio Ferrari. Fluid Annotation: A Human-Machine Collaboration Interface for Full Image Annotation. *arXiv preprint arXiv:1806.07527*, 2018.

[Aut] CMORE Automotive. C.LABEL: A smart and efficient way to label your data. https://www.cmore-automotive.com/en/products/software-tools/clabel/. Accessed: 19.10.2018.

[BC94] Jens Berkmann and Terry Caelli. Computation of Surface Geometry and Segmentation Using Covariance Techniques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(11):1114–1116, 1994.

[BKC17] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[Bra00] Gary Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[CDF+17] Angel Chang, Angela Dai, Thomas Funkhouser, Maciej Halber, Matthias Niessner, Manolis Savva, Shuran Song, Andy Zeng, and Yinda Zhang. Matterport3D: Learning from RGB-D Data in Indoor Environments. *International Conference on 3D Vision*, 2017.

[CFYU14] Liang-Chieh Chen, Sanja Fidler, Alan L Yuille, and Raquel Urtasun. Beat the MTurkers: Automatic Image Labeling from Weak 3D Supervision. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2014.

[Cho17] Francois Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[CM02] Dorin Comaniciu and Peter Meer. Mean Shift: A Robust Approach toward Feature Space Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, 2002.

[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*, 2016.

[CPK+15] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Semantic Image Segmentation with Deep Convolutional Nets and Fully Connected CRFs. In *International Conference on Learning Representations*, 2015.

[CPK+18] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848, 2018.

[CPSA17] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking Atrous Convolution for Semantic Image Segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[Cre16] Daniel Cremers. Multiple View Geometry, 2016. Lecture at Technische Universität München.

[CUF18] Holger Caesar, Jasper Uijlings, and Vittorio Ferrari. COCO-Stuff: Thing and Stuff Classes in Context. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[CZP+18] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation. 2018.

[DCS+17] Angela Dai, Angel X Chang, Manolis Savva, Maciej Halber, Thomas A Funkhouser, and Matthias Nießner. ScanNet: Richly-Annotated 3D Reconstructions of Indoor Scenes. In *CVPR*, 2017.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[DRMS07] Andrew J Davison, Ian D Reid, Nicholas D Molton, and Olivier Stasse. MonoSLAM: Real-Time Single Camera SLAM. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1052–1067, 2007.

[EEVG+15] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision*, 111(1):98–136, 2015.

[EKC18] Jakob Engel, Vladlen Koltun, and Daniel Cremers. Direct Sparse Odometry. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(3):611–625, 2018.

[EKHL17] Francis Engelmann, Theodora Kontogianni, Alexander Hermans, and Bastian Leibe. Exploring Spatial Context for 3D Semantic Segmentation of Point Clouds. In *IEEE International Conference on Computer Vision, 3DRMS Workshop*, 2017.

[ESC14] Jakob Engel, Thomas Schöps, and Daniel Cremers. LSD-SLAM: Large-Scale Direct Monocular SLAM. In *European Conference on Computer Vision*, 2014.

[ESC15] Jakob Engel, Jörg Stückler, and Daniel Cremers. Large-Scale Direct SLAM with Stereo Cameras. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2015.

[EUC16] Jakob Engel, Vladyslav Usenko, and Daniel Cremers. A photometrically calibrated benchmark for monocular visual odometry. *arXiv preprint arXiv:1607.02555*, 2016.

[FRS13] Paul Furgale, Joern Rehder, and Roland Siegwart. Unified Temporal and Spatial Calibration for Multi-Sensor Systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2013.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[GGAM14] Saurabh Gupta, Ross Girshick, Pablo Arbeláez, and Jitendra Malik. Learning Rich Features from RGB-D Images for Object Detection and Segmentation. In *European Conference on Computer Vision*, 2014.

[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*, 2012.

[GRU10] Andreas Geiger, Martin Roser, and Raquel Urtasun. Efficient Large-Scale Stereo Matching. In *Asian Conference on Computer Vision*, 2010.

[GWCV16] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtual Worlds as Proxy for Multi-Object Tracking Analysis. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[HSL+17] T Hackel, N Savinov, L Ladicky, JD Wegner, K Schindler, and M Pollefeys. SEMANTIC3D.NET: A New Large-Scale Point Cloud Classification Benchmark. *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 2017.

[HY16] Jing Huang and Suya You. Point Cloud Labeling using 3D Convolutional Neural Network. In *International Conference on Pattern Recogntion*, 2016.

[HZ03] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[KK11] Philipp Krähenbühl and Vladlen Koltun. Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials. In *Neural Information Processing Systems*, 2011.

[KM07] Georg Klein and David Murray. Parallel Tracking and Mapping for Small AR Workspaces. In *IEEE/ACM International Symposium on Mixed and Augmented Reality*, 2007.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Neural Information Processing Systems*, 2012.

[Lab] Labelbox. Labelbox Image Labeling. https://support.labelbox.com/docs/image-labeling-1. Accessed: 19.10.2018.

[LLB+15] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. Keyframe-Based Visual-Inertial Odometry Using Nonlinear Optimization. *International Journal of Robotics Research*, 34(3):314–334, 2015.

[LMB+14] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft COCO: Common Objects in Context. In *European Conference on Computer Vision*, 2014.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully Convolutional Networks for Semantic Segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[MAT17] Raul Mur-Artal and Juan D Tardós. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[MAZV17] Riccardo Monica, Jacopo Aleotti, Michael Zillich, and Markus Vincze. Multi-label Point Cloud Annotation by Selection of Sparse Control Points. In *International Conference on 3D Vision*, 2017.

[MFS13] Jérôme Maye, Paul Furgale, and Roland Siegwart. Self-supervised Calibration for Robotic Systems. In *IEEE Intelligent Vehicles Symposium*, 2013.

[MSKS12] Yi Ma, Stefano Soatto, Jana Kosecka, and S. Shankar Sastry. *An Invitation to 3-D Vision: From Images to Geometric Models*. Springer Science & Business Media, 2012.

[Mur12] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[NLD11] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. DTAM: Dense Tracking and Mapping in Real-Time. In *International Conference on Computer Vision*, 2011.

[NOBK17] Gerhard Neuhold, Tobias Ollmann, Samuel Rota Bulò, and Peter Kontschieder. The Mapillary Vistas Dataset for Semantic Understanding of Street Scenes. In *International Conference on Computer Vision*, 2017.

[Ols11] Edwin Olson. AprilTag: A robust and flexible visual fiducial system. In *IEEE International Conference on Robotics and Automation*, 2011.

[Omo89] Stephen M. Omohundro. Five Balltree Construction Algorithms, 1989.

[PCN17] Andra Petrovai, Arthur D Costea, and Sergiu Nedevschi. Semi-Automatic Image Annotation of Street Scenes. In *IEEE Intelligent Vehicles Symposium*, 2017.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[QLJ+17] Xiaojuan Qi, Renjie Liao, Jiaya Jia, Sanja Fidler, and Raquel Urtasun. 3D Graph Neural Networks for RGBD Semantic Segmentation. In *International Conference on Computer Vision*, 2017.

[QSMG17] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. 2017.

[QYSG17] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. In *Neural Information Processing Systems*, 2017.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, 2015.

[RKB04] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. In *ACM Transactions on Graphics*, 2004.

[RN10] Stuart J. Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010.

[RTMF08] Bryan C Russell, Antonio Torralba, Kevin P Murphy, and William T Freeman. LabelMe: A Database and Web-Based Tool for Image Annotation. *International Journal of Computer Vision*, 77(1-3):157–173, 2008.

[Rus09] Radu Bogdan Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Technische Universität München, 2009.

[SF68] I. Sobel and G. Feldman. A 3x3 Isotropic Gradient Operator for Image Processing, 1968. Talk at the Stanford Artificial Intelligence Project (SAIL).

[SF16] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-Motion Revisited. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[SHKF12] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor Segmentation and Support Inference from RGBD Images. In *European Conference on Computer Vision*, 2012.

[SHZ+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[SJC08] Jamie Shotton, Matthew Johnson, and Roberto Cipolla. Semantic Texton Forests for Image Categorization and Segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2008.

[SLK15] Hamed Sarbolandi, Damien Lefloch, and Andreas Kolb. Kinect Range Sensing: Structured-Light versus Time-of-Flight Kinect. *Computer Vision and Image Understanding (CVIU)*, 139:1–20, 2015.

[SLM04] Will J Schroeder, Bill Lorensen, and Ken Martin. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*. Kitware, 2004.

[SLX15] Shuran Song, Samuel P Lichtenberg, and Jianxiong Xiao. Sun RGB-D: A RGB-D Scene Understanding Benchmark Suite. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[SMD12] Hauke Strasdat, José MM Montiel, and Andrew J Davison. Visual SLAM: Why Filter? *Image and Vision Computing*, 30(2):65–77, 2012.

[SMS07] Gabe Sibley, Larry Matthies, and Gaurav Sukhatme. Bias Reduction and Filter Convergence for Long Range Stereo. In *International Symposium of Robotics Research*, 2007.

[Sup] Supervisely. 3D Cloud Labeling. https://supervise.ly/lidar-3d-cloud/. Accessed: 19.10.2018.

[SWRC09] Jamie Shotton, John Winn, Carsten Rother, and Antonio Criminisi. TextonBoost for Image Understanding: Multi-Class Object Recognition and Segmentation by Jointly Modeling Texture, Layout, and Context. *International Journal of Computer Vision*, 81(1):2–23, 2009.

[SZ15] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015.

[SZPF16] Johannes Lutz Schönberger, Enliang Zheng, Marc Pollefeys, and Jan-Michael Frahm. Pixelwise View Selection for Unstructured Multi-View Stereo. In *European Conference on Computer Vision*, 2016.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. MIT Press, 2005.

[vSUC18] L. von Stumberg, V. Usenko, and D. Cremers. Direct Sparse Visual-Inertial Odometry using Dynamic Marginalization. In *IEEE International Conference on Robotics and Automation*, 2018.

[WSC17] Rui Wang, Martin Schwörer, and Daniel Cremers. Stereo DSO: Large-Scale Direct Sparse Visual Odometry with Stereo Cameras. In *International Conference on Computer Vision*, 2017.

[WSL+18] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. Dynamic Graph CNN for Learning on Point Clouds. *arXiv preprint arXiv:1801.07829*, 2018.

[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*, 2016.

[XOT13] Jianxiong Xiao, Andrew Owens, and Antonio Torralba. SUN3D: A Database of Big Spaces Reconstructed using SfM and Object Labels. In *International Conference on Computer Vision*, 2013.

[Yan] Zhi Yan. L-CAS 3D Point Cloud Annotation Tool. https://github.com/yzrobot/cloud_annotation_tool. Accessed: 19.10.2018.