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

Master Thesis

3D City Reconstruction by Parsing Street View Images and Map Data

vorgelegt von

Oleg Chernikov
Matrikelnummer: 351016
February 27, 2017

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

Eidesstattliche Versicherung

Oleg Chernikov	351016
Name	Matrikelnummer

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

3D City Reconstruction by Parsing Street View Images and Map Data

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, February 27, 2017	
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, February 27, 2017	
Ort, Datum	Unterschrift

Contents

1 Introduction

In the last decade the interest to the robotics field was constantly growing. Although a lot algorithms are developed for the robots to perform different tasks, the navigation problem still remains the main challenge. The focus of this Thesis lies in the enhancement of the outdoor navigation via refinement of the open-source maps. For this purpose we use OpenStreetMap [osma] as those are one of the best open-source maps in terms of quality and amount of area covered. Essentially we do 3D reconstruction, although our purpose lies outside mainstream reconstruction methods. We focus on precise modeling of building's boundaries, while keeping the relief of their walls and roofs flat and simple, as the location and depth of doors, windows and other relief objects are irrelevant to navigation. First, we extract and parse the OSM map of the region of interest, which we want to enhance. Then we collect street-view stereo images made along the streets of this terrain, creating disparity maps and integrating depth in the scene. Based on integrated depth we move building models from the map to their observed positions. At the end the buildings with common walls have their facades separated. Our approach was evaluated on sequences of KITTI dataset [GLSU13] [GLU12], the ones, that have fulfilled our data requirements, that is having buildings and GPS information for matching the images with the map.
The novelty of our approach lies in observation-based localization of buildings on the map, with adjustment of their orientation. Most approaches [MMT+16], [CWUF16], that require both, maps and images count the maps as ground truth, while we only use map data as a prior for facade separation and initialization for localization. The facade separation idea is not new itself, but previously only the buildings, that face the camera, were considered, while for our method the building orientation on the image is irrelevant.

2 Related work

Our work is similar in spirit to [CWUF16] and [MMT+16]. Both of them present reconstruction approach, using the map and images to create an appropriate texture and relief for the model buildings. The first one uses rental shields to model house's surface, which limits the applicability of the method. The second one takes street-view photos and generally does it in large scale, for the whole city of Aachen, creating building models that closely resemble the real ones in the shape, but not in position or orientation. We, on the other hand, are more concerned with building's precise position and orientation.
The method we developed depends heavily on depth integration techniques or more specifically octrees. The original idea was taken from [Gar82], but extended with integration techniques from [UB15], so the octree stores not just the fact of presence of objects at a location, but the distance to the surface at different points of space and its reliability. For the integration we use disparity maps and semantic data.

2.1 Disparity methods

The octree requires multiple depth maps as input, but it is difficult to get depth directly from a stereo camera setup, so we rely on disparity computation methods and then, using calibration data, we extract depth maps for future integration. Different methods yield different results, so we tried to feed different disparity maps as input.

2.1.1 ELAS

Geiger et. al. developed a disparity estimation method in 2011 called Efficient Large-Scale Stereo Matching or ELAS [GRU10]. Even today this method is able to yield competitive performance in terms of speed and precision. The main idea of this method to first extract support points, that yield reliable matching

in the second image and then optimize the disparity around those points using maximum a-posteriori estimation, based on observations along the epipolar lines. They use the L_1 distance between concatenated Sobel filter responses, to determine the support points, as those prove to be both effective and efficient. Then Delaney triangulation is applied to the set of points, so there are some means for linear interpolation, which is later applied for mean computation for Gaussian probability. Then the energy minimization procedure takes place using maximum a-posteriori estimation for disparity computation, which enforces smooth disparities. Originally this method was tested on Middlebury dataset, which contains images of indoor scenes with limited depth, but in outdoor scenes we observed, that estimations include much more noise and are less reliable, especially on high ranges. Hence, we tried to find alternatives, that will perform better in the environments of the outdoor KITTI dataset.

2.1.2 SPS-Stereo

SPS-Stereo algorithm was introduced by Yamaguchi et. al. in [YMU14]. It is based on assumption, that the scene is piece-wise planar and mostly static, while the motion is rigid. The key difference from usual stereo methods is that it uses three images: a stereo pair and an image from one of the cameras at a later point of time. From the stereo pair a semi-dense disparity map is computed, from motion pair - semi-dense epipolar flow. Both are used to establish correspondence between flow and disparity via RANSAC. With this information Semi-global matching (SGM) can be applied with an energy function including a cost term and a smoothness term. The result is a semi-dense disparity map, which only exploits very local neighborhood relations. For further improvement the disparity map is split into superpixels and a slanted plane smoothing algorithm is applied, with a new energy formulation, that includes location, appearance, disparity, complexity, boundary-plane agreement and boundary length terms. An overview of the algorithm can be found on Fig. 2.1. Although the code was published only for the stereo part of the method, thus we did not use flow, when computing disparity maps.

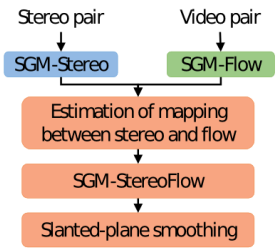
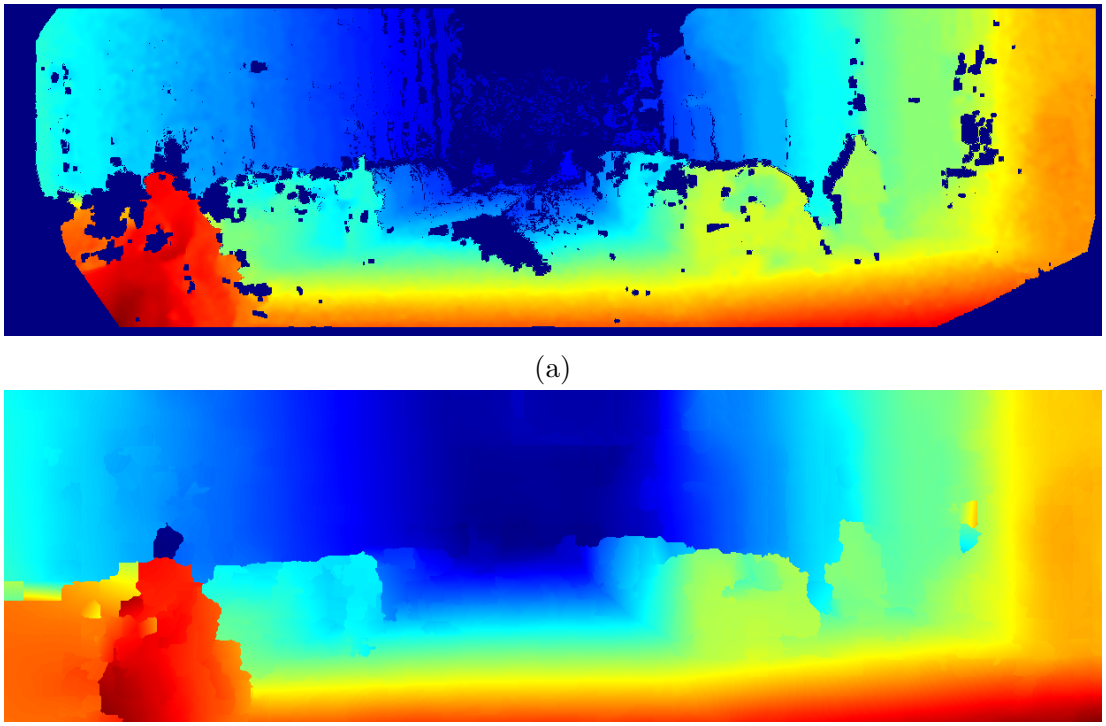


Figure 2.1: Pipeline of [YMU14]. Image courtesy of Yamaguchi et. al.



(a)

(b)

Figure 2.2: Heat maps based on the output of different disparity methods, on (a) the output of ELAS is shown and on (b) the result of running SPS-Stereo.

2.2 Semantic segmentation

In this thesis multiple goals are concerned with building localization or separation. In order to achieve that goal, we first have to be able to identify the buildings on the images. We found the results of semantic segmentation, described in [OHE+16] to be useful. They strive to segment out known background categories, such as roads, buildings, trees etc., which are of interest for our work. First, the point clouds are generated via ELAS method. Then VCCS algorithm [PASW13] is used to partition the point clouds into segments, which refers to over-segmentation procedure. For each segment a set of features is computer, each of the features belongs to one of following classes: appearance, density, geometry or location. The total length of resulting feature vector is 150. Then the Randomized Decision Forest (RDF) [Bre01] is trained for these features on the images, where the ground truth is available. Employing trained RDF results in semantic labels for each segment. As a final step, the labels are smoothed via fully connected Conditional Random Fields (CRF) [Kol11] defined over the segments' centers.

3 3D City Reconstruction

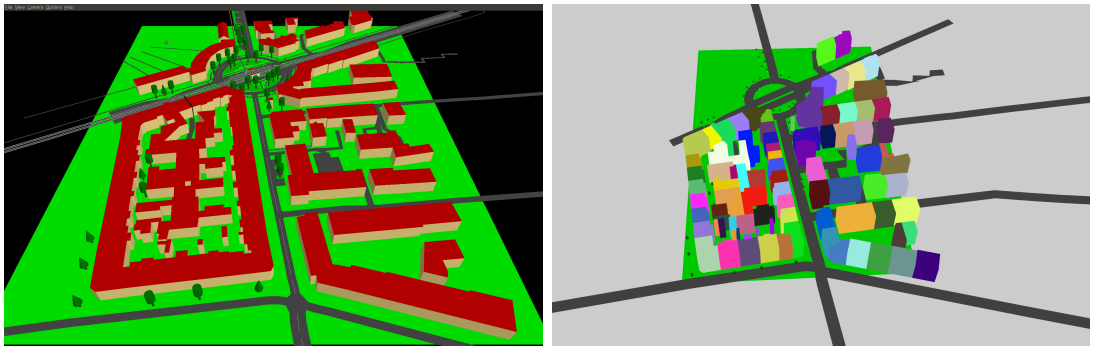
In this chapter we will describe the pipeline of our approach. We first parse an OSM map to extract prior locations of buildings as well as their prior 3D models. Then we use disparity maps of the scene and build an octree with integrated depth and surface information. Finally we run an energy minimization on building points and minimize the error of their alignment to the appropriate 3D model.

3.1 Building a 3D Model from OSM

Nowadays 3D Reconstruction is a field, that is popular within, Computer Graphics and Computer Vision communities. There are approaches that focus on quality reconstruction of smaller objects, coarse recreation of larger scenes or even both at the same time [UB15]. While we do not pursue high quality of reconstruction, we still try to keep some reconstruction errors as small as possible. In short, we omit relief of the buildings and assume all walls to be flat. Under this assumption error of the alignment of the buildings model to the observations is minimized.

3.1.1 Parsing OSM data

OpenStreetMap files have a great potential, when it comes to the description of buildings. Different qualities of buildings, such as the roof or the stairs shape, wheelchair ramp's availability, height etc., are described with **tags**. It does not end there: the tags also include semantic information, what kind of object it is, the address, whether it is a shop or cafe: the possibilities are limitless, as it is one of the OpenStreetMap's policies to use a free tagging system, meaning that we can introduce our own tags. Although the list of accepted tags can be found at http://wiki.openstreetmap.org/wiki/Map_Features. Thus a good parser, that will process all the tags correctly might require years of work. There are solutions though, like osm2world [osmb], that deliver a 3D model, that complies with map



(a) (b)

Figure 3.1: The comparison of reconstruction tools. On (a) the output of osm2world is shown. On (b) is our reconstructed model from parsing the map file

information (see Fig. 3.1a). Unfortunately, in the 3D model all the semantic and position information is omitted and matching between street-view images and the map could not be made, which eliminated the possibility to use this tool.

3.1.2 Parsing trees

The aforementioned reason made write a new parser. Since the focus of our work lies in buildings, this is the first type of the objects we process. We include roads, because we want to find the facades later and trees, because they are the main static source of occlusion. Still we cannot fully model the trees, as their shape is very complex, may involve transparency, changes with seasons and might grow in size. They are also the easiest to extract from the map. A typical tree is described with an XML entry like this

```
1 <node id="1328472000" lat="49.0095565" lon="8.4222346"
      version="3" timestamp="2015-07-23T17:04:33Z"
      changeset="32831581" uid="528930" user="Benjafrie">
2   <tag k="denotation" v="urban"/>
3   <tag k="natural" v="tree"/>
4 </node>
```

The header contains a lot of different information, but we only use latitude **lat** and longitude **lon** to retrieve the correspondence between the map and the poses of the image sequence. The tree is unique in a sense, that it only uses one node to describe its location, only a tag can give a hint, what kind of object it is. The rest of map objects, we consider, is represented by **ways**, structures, that are similar to polylines and also have possibility of looping, to be able to show polygons. If the tag **tree** is omitted, the structure is considered to be a usual

node. Then the field **id** is added to the information, we want to extract, since we use that id to reference the nodes from more complex structures, as they do not store their position information.

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

Figure 3.2: An example of tree changing over the seasons. In this case most obvious changes are transparency and color. Image taken from http://www.studiomacbeth.com/images/four-seasons-trees.html

3.1.3 Building parsing

Since tree reconstruction is a challenging task, the main focus in modeling is applied to buildings and roads. To save the workload we do not consider other objects, because it will increase the amount of work dramatically, while improvement is doubtful. The buildings have more priority, as one of the goals of this work is to optimize their location. An XML node of the building is shown below

```
1  <way id="90426027" version="3"
       timestamp="2015-05-26T15:38:30Z"
       changeset="31476734" uid="165061" user="mapper999">
2    <nd ref="1049023258"/>
3    <nd ref="1049023264"/>
4    <nd ref="1049023298"/>
5    <nd ref="1049023220"/>
6    <nd ref="1049023258"/>
7    <tag k="building" v="yes"/>
8    <tag k="shop" v="kiosk"/>
9    <tag k="source:geometry" v="Maps4BW, LGL,
         www.lgl-bw.de"/>
10   <tag k="wheelchair" v="yes"/>
11 </way>
```

The only use of the header this time is, that it has **way** in it and sometimes id to show uniqueness, the rest of information is irrelevant. **nd** child nodes are

just references to the nodes, like in case of a tree, but without such a tag. Those nodes have the location information, which we use to build a model. But first we check into tags and search for a line

```
<tag k="building" v="yes"/>
```

we can also extract some semantic information from the node, like the fact, that it is a shop or it has a wheelchair ramp. For the height estimation the most useful tags would be **height** or **min_height**. Even though those are rare, they are extremely helpful, as hypothesis for neighboring buildings. Parsing buildings is also difficult, because there exists tag **building:part**

```
<way id="404026395" visible="true" version="1"
    changeset="37866310" timestamp="2016-03-16T09:29:58Z"
    user="Matthias Frank" uid="287306">
  <nd ref="4063060551"/>
  <nd ref="4063060550"/>
  <nd ref="4063060547"/>
  <nd ref="4063060549"/>
  <nd ref="271787501"/>
  <nd ref="2938202852"/>
  <nd ref="2938202856"/>
  <nd ref="271787502"/>
  <nd ref="4063060551"/>
  <tag k="building:part" v="yes"/>
  <tag k="height" v="40"/>
</way>
```

This is not an actual building, but a part of an other one. That is the reason, we have to fuse it with its parent and consider it as one during the pose optimization. There is no direct link made between a part and its parent, therefore we simply pick out a pair, which has the most nodes in common.

3.1.4 Road parsing

The final type of objects is **Road**. The parsing was also not trivial due to different tag values.

```
<way id="134655454" version="3"
    timestamp="2015-12-21T22:15:58Z"
    changeset="36096014" uid="2600695" user="jlcod">
<nd ref="21498581"/>
<nd ref="21498576"/>
<nd ref="1480192283"/>
<nd ref="1255392691"/>
<nd ref="3903695906"/>
<tag k="highway" v="residential"/>
<tag k="maxspeed" v="30"/>
<tag k="name" v="Gerwigstrasse"/>
</way>
```

The roads are marked with a tag **highway**, but the value of that tag may vary. So far we consider these values: **motorway**, **trunk**, **primary**, **secondary**, **tertiary**, **residential**, **service**, **unclassified**. These roads may be for pedestrians or cars, it is not always understandable from tags.
We use Boost's XML parser to extract the information from OSM file, the full algorithm is described in Alg. 1.

3.1.5 Matching with street-view observations

What is not described here, but nevertheless is of interest for us, is *TransformToWorldCoordinates* function. That includes three stages: geo-projection, normalization and transformation. Geo-projection transforms latitudes *lat* and longitudes *lon* into a coordinate system, where meters are used. Noticeable, KITTI dataset has its own geo-projection with a set of simple equations. First using *lat* scale s is defined as

$$s = cos(lat). \tag{3.1}$$

Geiger et al. uses $R_e = 6378137$ as the earth's radius. The coordinates can then be computed as

$$\begin{aligned} x &= s \cdot lon \cdot R_e \\ y &= s \cdot R_e \cdot log(tan(\frac{90 + lat}{2})) \end{aligned} \tag{3.2}$$

While it works nice for trajectories, such as car positions of a car in KITTI's sequence, we faced some problems by trying to apply it to the buildings. The main problem was, that the buildings were skewed after applying that projection. While we do not know an exact cause of this, we suppose, that the procedure described by Geiger et. al. is just an approximation and works in a limited set of cases. Therefore we decided to use a different projection, using code snippets by Eugene Reimer, Peter Dana and Chuck Gantz [ER]. This projection is highly complex and involves different zones, datums and ellipsoids, therefore the exact algorithm will be omitted, as it is not the focus of this thesis.

Algorithm 1 Parse the map

```
1: procedure ParseOSM
2:     for all s ∈ osm.children do
3:         if isNode(s) then
4:             if isTree(s) then
5:                 T ← T + {s}
6:             else
7:                 N ← N + {s}
8:             end if
9:         else
10:            if isWay(s) then
11:                if isBuilding(s) then
12:                    B ← B + {s}
13:                else
14:                    if isBuildingPart(s) then
15:                        Bp ← Bp + {s}
16:                    else
17:                        if isRoad(s) then
18:                            R ← R + {s}
19:                        end if
20:                    end if
21:                end if
22:            end if
23:        end if
24:    end for
25:    gatherNodes(B, N)
26:    gatherNodes(Bp, N)
27:    gatherNodes(R, N)
28:    Fuse(B, Bp)
29:    TransformToWorldCoordinates(B)
30:    TransformToWorldCoordinates(R)
31:    TransformToWorldCoordinates(T)
32:    return B, R, T
33: end procedure
```

After we get the x and y coordinates, we have to center it around the position $p = (p_x, p_y)$ of the car in a first frame

$$\begin{aligned} x' &= x - p_x \\ y' &= y - p_y \\ z' &= z = 0 \end{aligned} \tag{3.3}$$

After getting the coordinates, small adjustments are required. The KITTI dataset and most parts of vision and graphics community use a coordinate format, where x represents width, y - height and z - depth. After map parsing we have different convention used, thus the points have to be cast to a different format

$$\begin{bmatrix} x'' \\ y'' \\ z'' \\ 1 \end{bmatrix} = \begin{bmatrix} -x' \\ z' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \tag{3.4}$$

Now, the locations are aligned and position of the car in a map is known. But the camera orientation is still unknown as the rotation component is missing. The solution of this task could be very challenging, if we would have only a single frame. Fortunately, we have a sequence with a lot of additional position information. There are two ways to extract the necessary rotation, to match the observation with a map. The first is to consult the data of KITTI dataset itself. Their machinery is quite advanced, so there is information from the IMU device available. That means, we have values such as roll, pitch and heading known in each frame. These three parameters show the difference in angles between watching direction and the axes defined by the result of geo-projection. The most interesting for us is a pitch: rotation angle r_y around axis Y. The rotation matrix can then be defined as

$$R_y = \begin{bmatrix} cos(r_y) & 0 & sin(r_y) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(r_y) & 0 & cos(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

Usually the pitch differs the most from the original axes. In the general case though, it is not the parameter that is different from 0. We have to rotate around X with angle r_x and Z with angle r_z given by roll and heading respectively. The matrices are built in the same way as for r_y, but for different axes. For r_x

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(r_x) & -sin(r_x) & 0 \\ 0 & sin(r_x) & cos(r_x) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

and for r_z

$$R_z = \begin{bmatrix} cos(r_z) & -sin(r_z) & 0 & 0 \\ sin(r_z) & cos(r_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.7}$$

And the full rotation matrix R is

$$R = R_x \cdot R_y \cdot R_z \tag{3.8}$$

There is one part left though, after rotation is ready: the camera is on a ground level, with height equal to 0 that is, therefore we have to adjust the height with parameter y_c. According to KITTI's documentation, the camera is 1.73 m above the ground, so $y_c = 1.73$. In fact y_c might be considered redundant, since we later adjust the lower ends of the buildings by fitting them to the ground plane, we get from [CKZ+15]. Nevertheless, the final transformation matrix is

$$T = \left(\begin{array}{ccc|c} & R & & \begin{matrix}0\\ y_c\\ 0\end{matrix} \\ \hline 0 & 0 & 0 & 1 \end{array}\right) \tag{3.9}$$

The problem of this approach is that an IMU device is required. If the data is used, where no roll, pitch and heading information is provided, we would still want to be able to run our method with only GPS coordinates. We found out, that a good approximation would be to take these angles from comparison of two neighboring positions. Supposing that we have two such positions $p^i = (p_x^i, p^i - y, p_z^i)$ and $p^{i+1} = (p_x^{i+1}, p^{i+1} - y, p_z^{i+1})$. We first get the differences

$$\begin{aligned} \triangle x &= p_x^{i+1} - p_x^i \\ \triangle y &= p_y^{i+1} - p_y^i \\ \triangle z &= p_z^{i+1} - p_z^i \end{aligned} \tag{3.10}$$

Then the angles rx, ry and rz we can get from

$$\begin{aligned} rx &= atan\frac{\triangle y}{\triangle z} \\ ry &= atan\frac{\triangle x}{\triangle z} \\ rz &= atan\frac{\triangle x}{\triangle y} \end{aligned} \tag{3.11}$$

and then we can proceed as before. The weakness of this method though is, that the recording vehicle should be adjusted to its trajectory, which is not a problem in case of a car, but will be problematic if someone decided to record a dataset using stereo setup placed on a helmet.

3.2 Octree

For the high quality reconstruction of the scene of this scale we need an efficient representation, which should also support interpolation. We chose to use an octree [Gar82], [UB15] for the integration of the surface information inside the scene and a notion of truncated signed distance to encode the distance to the surface at any point of the scene. As input we used disparity images generated by SPS-Stereo [YMU14] and ELAS [GRU10].

3.2.1 Definition

An octree is an efficient structure in terms of memory consumption and access speed. However it may take a while to construct an octree, since it heavily depends on the input information. We do not aim for the efficiency, but for the precision of the end result, hence we use a rather simple hashmap implementation instead of a pointer-based one. The idea behind the octree approach is quite

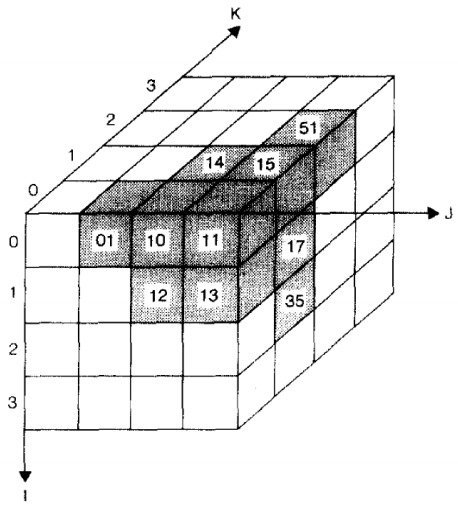


Figure 3.3: An example of an octree representation. Image courtesy of Irene Gargantini [Gar82].

simple: every non-leaf node has 8 children, thus with just one byte we can encode which child are we using. In our implementation we denote the dimension as I, J and K, where I represents height, J - width and K - depth. We assign a number from 0 to 7 based on the location of a child as shown on Fig. 3.3. In other words, we encode:

- octant NWF with 0
- octant NEF with 1
- octant SWF with 2

- octant SEF with 3
- octant NWB with 4
- octant NEB with 5
- octant SWB with 6
- octant SEB with 7

where I changes from North (N) to South (S), J from West (W) to East (E) and K from Forward (F) to Backward (B). When assigning an id to voxel, we start with an id of an upper level child. For instance, on Fig. 3.3 id "1" denotes a voxel consisting out of basic voxels (of size one), where $i \in \{0; 1\}$, $j \in \{2; 3\}$, $k \in \{0; 1\}$. An NWF voxel inside voxel "1" would have id "10". Generally, the longer the id of a voxel is, the deeper it lies in hierarchy and the smaller its size is.
The intuition behind the size is, that the voxels of the bigger size represent regions of the scene, where the information is coarse, while smaller voxels mean, that the data about this region can be trusted. There is variety of the scene information, we can fit into the octree: surface distances, normals, semantic labels etc. We follow [UB15] approach and use normals and depth measurements to construct the octree.

3.2.2 Truncated signed distance

While usual distance functions show, how far the object is, the signed distance function (SDF) can additionally determine on which side the object lies. We use this notion to detect, whether the point of space lies in front or behind the surface. Furthermore, we employ the *truncation* for SDF (TSDF): if the distance exceeds some threshold, we truncate it to this threshold. Thus we limit the influence of the outliers.

3.2.3 Construction

The construction of the octree is a two step process which involves structure generation and data aggregation. During the structure generation step we fill the space with nodes of different scale, but without any data inside. In the second step we fill them with aggregated normal, TSDF and scale information.

Structure generation

As input we use disparity maps, generated from KITTI sequences [GLSU13], [GLU12] by SPS-Stereo method [YMU14]. In fact, any reliable 3D data would be

sufficient, although sparsity and outliers both have relatively strong impact on the performance. Nevertheless it can be reduced by integration of several consequent frames, if the errors are not repeated.
From the disparity maps we generate a 3D point cloud with origin at the camera position in the first frame. As stereo disparity methods tend to have outliers, we limit our scene size with some bounding box to cut out the unreliable points. As soon as the point cloud's dimension sizes are known we derive a transformation to the octree space, which is essentially just scaling and translation. This transformation is not a necessity, we can actually do the computations in world coordinates, but it won't be as convenient, since an integer voxel grid allows to avoid a lot of numerical difficulties and potential bugs. There is one more thing to note by the transformation: scene specificity. A lot of KITTI sequences are made by following just one direction. That makes the scene longer in that direction, while the sizes of the other dimensions are minimal. Thus if we only build the octree from the point cloud, those dimensions are stretched out to match the length of the longest dimension. Therefore we pick a possible minimal cube, containing the cloud and build the octree from that cube. We do that by adding empty space to the both sides of the intervals of smaller dimensions equally, so that they match the longest one in length. Since it does not matter for the octree if we include an empty space, the performance remains unchanged. We also need normals for each point to be able to extract the surface information. We use integral image normal estimation [HRD+12] implemented in PCL [RC11] as it shows performance superior to the other normal estimation methods.

Voxel placement When the point cloud and the transformation are ready we start to create voxels in the octree. Here we mostly follow [UB15], except for little details. We define voxel grid's edge length as L, individual voxel's edge length as l, depth level as d and a point scale as σ. σ is a result of multiplication of distance between camera and the point and a depth constant d_c. In our experiments we used $d_c = 0.1$. Once we have point's scale, we can directly compute the voxels, that will be affected by this point. First, we calculate a depth level, at which voxels will be created. Intuitively, the dependency of voxel size from the depth can be described as $l = L/2^d$. So, for the voxel of the size l we will take the points of scale up to $l/2$, i.e. where $2\sigma \leq L/2^d$. Obviously there might be several d values, which satisfy this condition, therefore we only take maximal satisfying d, to limit the influence to a single depth level. Now, we only want to use voxels close to the point. Thus for each voxel we define a search window R_h, which has a center at the voxel's center and a radius $h = 3l$ and point window W_{σ_i} with center at the point position p_i and a radius σ_i as shown at Fig. 3.4. If the point falls inside R_h, then the point affects this voxel. If the point affects the voxel we create this voxel, if it does not exist and we later use this point to update this voxel, if it is still there or update its children. The difficulty here is that the

algorithm supposes that we iterate through the voxels and for each of those we gather close points. In reality this approach proves to be very inefficient, since there are $\sum_{d=0..log_2 L}(L/(L/2^d))^3 = \sum_{d=0..log_2 L} 2^{3d}$ voxels and only a small part of them contains at least some information. Instead we can use scale property of the point. Considering, that the scale of the point is constant, the depth level d will be the same for all the affected voxels. That means, that their support window will have the same radius and we use reverse search: find any voxels, the center of which fall in a support window R_h based on point location. We add the voxels that yet do not exist to the octree, regardless of whether there is a parent-child relation between old and new voxels. For convenience we store a list of the contributing points in each voxel. As soon as every voxel is in the octree, we launch the splitting procedure. That is, we split each parent into 8 children, until there is no parents left. We copy the list of the contributing points as well as the scale of the parent voxel. Note, that voxel's scale and size are two separate things, which may be equal only if the voxel was added directly and not through splitting procedure. Instant splitting is possible too, but from our experience, post-splitting works faster and can be ran in parallel.

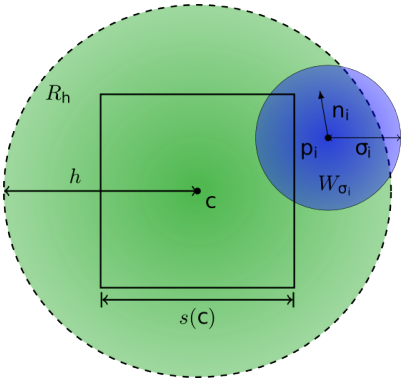


Figure 3.4: An intersection of a point and the support window R_h. h is a radius of this window, c is the center of both the voxel and the window, $s(C) = h$ is a length of the edge of a voxel. n_i denotes normal of the point i, p_i denotes its position and σ_i its support. Image courtesy of B. Ummenhofer and T. Brox [UB15]

Data aggregation

We move on to this step, when every voxel has been added and there is no parent-child relation between the nodes of the octree. Here we go over all existing voxels and aggregate their contributing points. First, for each point we check, that its center p_i lies inside of the support window R_h, i.e. $||c - p_i|| \leq h$, because during

splitting procedure, this condition could have been violated. For each point, that satisfies we want to compute signed distance $f_i(c)$, which is equal to

$$f_i(c) = \frac{1}{w_i} \int R_h(x - c)\ W_{\sigma_i}(x - p_i)\ \langle n_i, c - x \rangle\ \mathrm{dx} \tag{3.12}$$

The corresponding weight for this distance is:

$$w_i = \int R_h(x - c)\ W_{\sigma_i}(x - p_i)\ \mathrm{dx} \tag{3.13}$$

Unfortunately, the evaluation of the integrals is a computationally expensive problem, therefore we bound to use the solution from Ummenhofer et. al. We also approximate support window function as in [UB15]

$$R_h(r) = \begin{cases} \frac{315}{64\pi h^9}(h^2 - ||r^2||^2)^3, & \text{if } ||r|| \leq h \\ 0, & \text{otherwise} \end{cases} \tag{3.14}$$

where r denotes the vector between the point p_i and the voxel center c. Now we can compute approximate discrete weights w_i and signed distances f_i

$$w_i(c) = \frac{4}{3}\pi\sigma_i^3 R_h(p_i - c) \tag{3.15}$$

$$f_i(c) = \frac{1}{w_i}\frac{4}{3}\pi\sigma_i^3 R_h(p_i - c)\ \langle n_i, c - p_i \rangle = \langle n_i, c - p_i \rangle \tag{3.16}$$

Given that every voxel can contain measurements from many points and a single point can contribute to many voxels, we cannot store every measurement. Instead we aggregate them into a histogram, for each voxel separately. The histogram h has 8 bins, each index b of which represents particular signed distance range, based on the scale of the voxel. Since each measurement is inside the support window R_h, $-h \leq f_i(c) \leq h$ holds. Then the bin index can be defined as

$$b = \frac{f_i(c) + h}{2h} \cdot 8 \tag{3.17}$$

However, if we just increment a single bin for each point, this will introduce quantization effects. To reduce those, we apply a technique, called soft binning, that is along b-th bin we increment $b-1$-th and $b+1$-th by the half of the value, if those bins are present. Thus, the full formula for the bin value looks like this

$$h_b = \sum_{i, b = \frac{f_i(c)+h}{2h}\cdot 8} w_i + \sum_{i, b = \frac{f_i(c)+h}{2h}\cdot 8 - 1} \frac{w_i}{2} + \sum_{i, b = \frac{f_i(c)+h}{2h}\cdot 8 + 1} \frac{w_i}{2} \tag{3.18}$$

In our approach we want to avoid ambiguity, therefore as soon as all the values are aggregated, we replace histograms by a single value, which represents observed distance to the surface for the voxel. We apply Gaussian fitting to the histogram

and take the value of its peak. There is one problem though, with the voxels that lie close to the intersection of the surfaces, like building corners. In this case two distributions arise in the histogram and fitting a single Gaussian will yield unexpected results. This problem has rare occurrence though, it still can be solved by running Gaussian clustering algorithm with different number of clusters and taking the result, which has the least error. The simpler alternative to fitting a Gaussian is to just take the maximal peak value of the histogram, which works a bit worse, if window size parameter h is not tuned, but is able to yield better results in comparison to a Gaussian if h is a bit off. For our purpose fitting Gaussian along with tuning parameter h was sufficient for satisfying results.

3.2.4 Octree interpolation

The main difference between the octree and a regular voxel grid is that the octree allows to have cells of different sizes and scales in the same structure (Fig. 3.5). It brings up both, advantages and disadvantages. The regular grid is more convenient to use, things like interpolation or ray tracing are very simple to implement for this structure at the cost of memory and additional workarounds for low resolution regions, where we do not have a lot of surface and normal information, but still have to maintain a small cell size of a grid. The octree overcomes these shortcomings, while sacrificing convenience of the use and possibly efficiency.

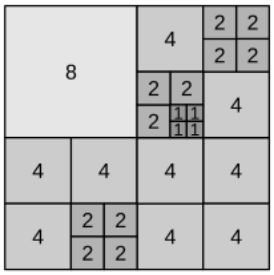
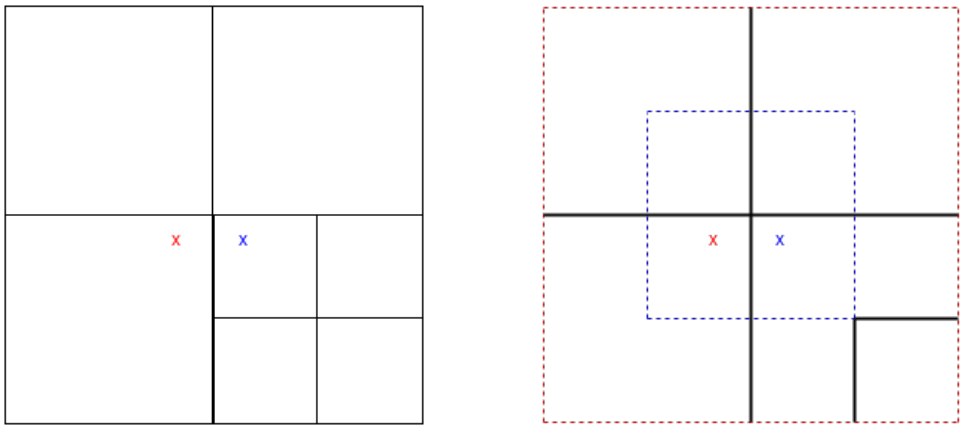


Figure 3.5: An example of an multiscale quadtree, with scale values given inside the nodes. Image courtesy of Ummenhofer et al. [UB15]

While it is possible to easily employ linear interpolation for a regular grid, it may be difficult to do the same in the octree. If we have to interpolate between nodes of the same scale, then it is the same as for the regular grid. There are cases though, when linear interpolation won't do. The simplest non-standard case is when the point to interpolate lies between nodes of different scale. We have to differentiate between nodes of bigger and smaller scale. To the best of our knowledge there is no well known method to apply linear interpolation to multiple scales case. Therefore we try boil down the problem to a regular grid case.



(a) (b)

Figure 3.6: On (a) an example of interpolation case of multiscale quadtree is shown. The red "x" shows the case of downscaling, when the point is inside a node of larger scale and nodes of smaller scale have to be interpolated to create a temporal node of the same size, as the bigger ones. The blue "x" shows the opposite case, where we have to create smaller nodes from bigger ones. On (b) one can see final regions of interest for points.

Two such cases are pictured on Fig. 3.6a. First we consider the "red" case, when we have to create a node of larger scale from the smaller nodes. We apply linear interpolation of a TSDF value inside those nodes, to get an appropriate value for a hypothetical bigger node. Since we have the information of higher resolution inside the smaller nodes, the downscaling works in similar way as blurring filters, so we do not lose a lot of information and can safely do so. The "blue" case is more complicated, although we do almost the same thing, except we now do it in reverse: we change the resolution from coarser to finer. For this problem there is no satisfying solution, that will work in all of cases, because we basically fill in information that is not there. Anyway, we still do the most straightforward thing and split the node, copying the information from a parent node and adapting it to a new scale.
The described methods may not work in general, because of aforementioned reason, that is making up information, which is missing. But if considered in more detail, that might be not that much of a problem. As the smaller nodes lie near the bigger ones, we may assume that this is a border case, where the nodes transfer from one scale to an other, which in turn shows, that in reality the nodes are of similar scale and the difference was caused by discretization artifacts. That proves, that the information inside the small nodes is almost identical and we can make up a direct parent for them. It works in the opposite direction as well, due to the same reasoning. The only problematic case is when one of the nodes was generated by outlier points, since the TSDF values may differ dramatically in this

case. Albeit this scenario is also rather unrealistic. The key thought is that the octree is created via integration over multiple consequent frames, meaning that every node grabs the points not only from the same frame, but also from the other ones. So if outliers occur, they should be repeatable to create real disturbance inside the octree and this is a rather rare occurrence.
Another problem is missing data. If there is a parent or children of the voxel, we can inter- or extrapolate the missing TSDF value, but if there are neither, we can only try to lay our best guess or miss it completely. We decided to adhere to the latter strategy. Our motivation for this particular case was, that we already integrated all the points, we want to interpolate for, hence the described case may only happen, when points are leaving the scene due to transformations, described in 3.3, which we consider improbable. After all, points are expected to shift in direction of estimated surface, where the octree should have more density.

3.2.5 Algorithms

There are multiple useful algorithms on the octree, described in initial Gargantini's paper [Gar82], that were also used in our work. In particular those were encoding, decoding and neighbor search algorithms. We also describe our way of mapping the data to octree coordinate system.

Encoding

As described in Sec. 3.2.1, each node may be described as set of three indices: J, I and K. While for some applications these coordinates would be sufficient, some more efficient algorithms require an octal code of a node to work with. An octal code is sequence of digits which shows the location of the voxel and its depth, which is also advantageous, as voxels of different scales may have the same coordinates, since in our implementation we use the coordinates of the voxel's corner, that is the closest to the origin to determine voxel's location. The problem with ambiguity of voxel's coordinates can be avoided though, if voxel's centers are used instead. Still this will lead to either usage of floating point coordinates or extending grid in size to make the minimal node's size equal to two, both of which are rather unwanted.
So, we use octal codes mainly, because it allows us to query the nodes efficiently, as we use a hashmap implementation, and to quickly find the neighboring nodes, which are of particular interest for the interpolation procedure. As input we get a point in octree coordinates J, I and K which represent width, height and depth

respectively. The side of a grid $L = 2^n$, thus J, I, $K \in \{0, 1, ..., 2^{n-1}\}$. We first convert them to a binary representation

$$\begin{aligned} J &= c_{n-1} \cdot 2^{n-1} + ... + c_t \cdot 2^t + ... + c_0 \cdot 2^0 \\ I &= d_{n-1} \cdot 2^{n-1} + ... + d_t \cdot 2^t + ... + d_0 \cdot 2^0 \\ K &= e_{n-1} \cdot 2^{n-1} + ... + e_t \cdot 2^t + ... + e_0 \cdot 2^0 \end{aligned} \tag{3.19}$$

and the octal representation is

$$Q = q_{n-1} \cdot 8^{n-1} + ... + q_t \cdot 8^t + ... + q_0 \cdot 8^0 \tag{3.20}$$

The task now is basically to find the dependency between q_t and c_t, d_t, e_t, which we define as

$$q_l = e_t \cdot 2^2 + dt \cdot 2^1 + c_t \cdot 2^0 \tag{3.21}$$

So, if we have for example a voxel with coordinates $J = 1$, $I = 2$, $K = 3$ in an octree with $n = 2$ and $L = 2^n = 2^2 = 4$, then

$$\begin{aligned} J = 01_2 &= c_1 \cdot 2^1 + c_0 \cdot 2^0 = 0 \cdot 2^1 + 1 \cdot 2^0 \\ I = 10_2 &= d_1 \cdot 2^1 + d_0 \cdot 2^0 = 1 \cdot 2^1 + 0 \cdot 2^0 \\ K = 11_2 &= e_1 \cdot 2^1 + e_0 \cdot 2^0 = 1 \cdot 2^1 + 1 \cdot 2^0 \end{aligned}$$

And coefficients of Q are

$$\begin{aligned} q_0 &= e_0 \cdot 2^2 + d_0 \cdot 2^1 + c_0 \cdot 2^0 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5 \\ q_1 &= e_1 \cdot 2^2 + d_1 \cdot 2^1 + c_1 \cdot 2^0 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6 \end{aligned}$$

Thus the final octal code of a node is

$$Q = q_1 \cdot 8^1 + q_0 \cdot 8^0 = 6 \cdot 8^1 + 5 \cdot 8^0 = 65_8$$

Decoding

Since each digit in the octal code represents the child's relative position inside the parent, we need to iterate through digits of the code, starting from the highest order digit to determine the voxel's location step by step. Intuitively when iterating we just reduce possible intervals $I_t = [i_{a,t}, i_{b,t}]$, $J_t = [j_{a,t}, j_{b,t}]$ and $K_t = [k_{a,t}, k_{b,t}]$ of the voxel. We are starting with $I_0 = J_0 = K_0 = [0, 2^n - 1]$ and halving each of them in each iteration. Since there are only eight cases, such decomposition for a single digit may be described with a Tab. 3.1

Table 3.1: Decoding table of the octal code

Input	**Output**		
q_{n-t}	Action for I_t	Action for J_t	Action for K_t
0	$i_{b,t} = i_{b,t-1} - 2^{n-t}$	$j_{b,t} = j_{b,t-1} - 2^{n-t}$	$k_{b,t} = k_{b,t-1} - 2^{n-t}$
1	$i_{b,t} = i_{b,t-1} - 2^{n-t}$	$j_{a,t} = j_{a,t-1} + 2^{n-t}$	$k_{b,t} = k_{b,t-1} - 2^{n-t}$
2	$i_{a,t} = i_{a,t-1} + 2^{n-t}$	$j_{b,t} = j_{b,t-1} - 2^{n-t}$	$k_{b,t} = k_{b,t-1} - 2^{n-t}$
3	$i_{a,t} = i_{a,t-1} + 2^{n-t}$	$j_{a,t} = j_{a,t-1} + 2^{n-t}$	$k_{b,t} = k_{b,t-1} - 2^{n-t}$
4	$i_{b,t} = i_{b,t-1} - 2^{n-t}$	$j_{b,t} = j_{b,t-1} - 2^{n-t}$	$k_{a,t} = k_{a,t-1} + 2^{n-t}$
5	$i_{b,t} = i_{b,t-1} - 2^{n-t}$	$j_{a,t} = j_{a,t-1} + 2^{n-t}$	$k_{a,t} = k_{a,t-1} + 2^{n-t}$
6	$i_{a,t} = i_{a,t-1} + 2^{n-t}$	$j_{b,t} = j_{b,t-1} - 2^{n-t}$	$k_{a,t} = k_{a,t-1} + 2^{n-t}$
7	$i_{a,t} = i_{a,t-1} + 2^{n-t}$	$j_{a,t} = j_{a,t-1} + 2^{n-t}$	$k_{a,t} = k_{a,t-1} + 2^{n-t}$

We start with $t = 1$, as we have already values for $t = 0$. This process may seem complicated, but it also may be thought of as a reverse for encoding process: take an octal code, get three binary codes from it and convert binary codes to the decimals.

Neighbor search

The real power of the octree lies in the neighborhood search [Gar82]. Since the octal codes are hierarchical in a sense, that every next digit narrows the set of possible voxel's locations, we can directly compute the codes of the neighbors and check their existence. We will consider only half of the algorithms: for finding southern, eastern and back neighbors, since the other half just mirrors these three. On Fig. 3.7 an example quadtree is shown. The red node is shown

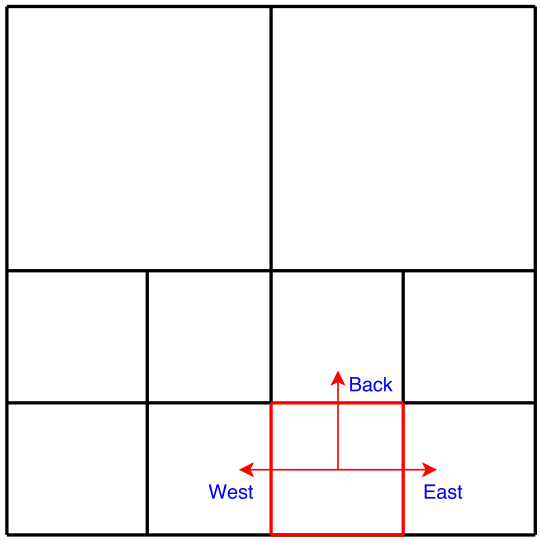


Figure 3.7: An example of quadtree adjacencies, without the height dimension

along its adjacencies, where 3 cases are considered. East and Back adjacencies correspond to the first case, when the neighbors lie in the direct common parent. West neighbor is in the higher level common parent. And a forward node does not exist, which leads to the third case. Since the position of the node is derivable from the octal code, we can start from the end of the code, that is from the digits, that represent the location on a deepest level. The basic idea of the algorithm is, that we traverse the octree from the node through its parents, until searched neighbor lies in the same parent.

Algorithm 2 Search for eastern Neighbor

```
1: procedure FindEasternNeighbor
Input: n; q_0, q_1, ..., q_{n-1}
Output: E(q_0), E_(q_1), ..., E(q_{n-1})
2:     if isEven(q_0) then
3:         E(q_0) = q_0 + 1
4:     else
5:         E(q_0) = |q_0 + 7|_8
6:     end if
7:     i = j = 1
8:     while i ≠ n and j ≠ n − 1 do
9:         if isEven(q_{i−1}) then
10:            E(q_j) = q_j, j = i, i + 1, ..., n − 1
11:        else
12:            if isEven(q_i) then
13:                E(q_i) = q_i + 1
14:            else
15:                E(q_i) = |q_i + 7|_8
16:            end if
17:            i = i + 1
18:        end if
19:    end while
20:    return E(q_0), E_(q_1), ..., E(q_{n-1})
21: end procedure
```

The result $E(q_i)$ returned in this algorithm shows the octal code of the neighboring node, even if the algorithm fails to find one. In that case we have to track the outcome of the "if" operator. The condition inside the clause should be true at least once for any valid node, otherwise we deal with a border voxel, i.e. the one that stays exactly at the border of the cube. We have to be careful when the clause is true as well, because the existence of the valid octal id does not guarantee the existence of a node itself.
Coming next is an algorithm for a southern neighbor. The pipeline is the same, except for the if clause. Instead we check, if the node stays in the upper part of

direct parent (0, 1, 4, 5 are the nodes, that are in the upper part, check Fig. 3.3 for reference).

Algorithm 3 Search for southern Neighbor

1: **procedure** FINDSOUTHERNNEIGHBOR
Input: $n; q_0, q_1, ..., q_{n-1}$
Output: $S(q_0), S_(q_1), ..., S(q_{n-1})$
2: **if** $q_0 \in \{0, 1, 4, 5\}$ **then**
3: $S(q_0) = |q_0 + 2|_8$
4: **else**
5: $S(q_0) = |q_0 + 6|_8$
6: **end if**
7: $i = j = 1$
8: **while** $i \neq n$ and $j \neq n - 1$ **do**
9: **if** $q_{i-1} \in \{0, 1, 4, 5\}$ **then**
10: $S(q_j) = q_j, j = i, i + 1, ..., n - 1$
11: **else**
12: **if** $q_i \in \{0, 1, 4, 5\}$ **then**
13: $S(q_i) = q_i + 2$
14: **else**
15: $S(q_i) = |q_i + 6|_8$
16: **end if**
17: $i = i + 1$
18: **end if**
19: **end while**
20: **return** $S(q_0), S_(q_1), ..., S(q_{n-1})$
21: **end procedure**

The case with a back neighbor is a bit more special, since their numbering goes consequently. Therefore we already know, that the digit will differ by $|q_i + 4|_8$, thus we do not need to add an additional clause, which makes this algorithm slightly more efficient.

These are the three basic algorithms. Another three are easily achievable from those by slightly modifying the if clause.

Mapping from world to octree

As one could already notice, an octree requires a cube with a side of the size 2^n to work. The real scene won't usually contain those proportions. Hence the bounding box of the scene has to be mapped to the octree coordinates properly.

Algorithm 4 Search for back Neighbor

```
1: procedure FindBackNeighbor
Input: n; q_0, q_1, ..., q_{n-1}
Output: B(q_0), B_(q_1), ..., B(q_{n-1})
2:     B(q_0) = |q_0 + 4|_8
3:     i = j = 1
4:     while i ≠ n and j ≠ n − 1 do
5:         if q_{i-1} ∈ {0, 1, 2, 3} then
6:             B(q_j) = q_j, j = i, i + 1, ..., n − 1
7:         else
8:             B(q_i) = |q_i + 4|_8
9:             i = i + 1
10:        end if
11:    end while
12:    return B(q_0), B_(q_1), ..., B(q_{n-1})
13: end procedure
```

The first idea that comes to mind is to scale and translate the box. Suppose we have a box of sizes

$$\begin{aligned} X &= [x_{min}, x_{max}] \\ Y &= [y_{min}, y_{max}] \\ Z &= [z_{min}, z_{max}] \end{aligned} \tag{3.22}$$

Then the corresponding transformation for the point $p = (x, y, z)$ is just minmax normalization and scaling

$$\begin{aligned} j &= \frac{x - x_{min}}{x_{max} - x_{min}} \cdot 2^n \\ i &= \frac{y - y_{min}}{y_{max} - y_{min}} \cdot 2^n \\ k &= \frac{z - z_{min}}{z_{max} - z_{min}} \cdot 2^n \end{aligned} \tag{3.23}$$

This will actually work fine under condition that $x_{max} - x_{min} = y_{max} - y_{min} = z_{max} - z_{min}$. In real world scenarios height will have much smaller interval, than width or depth. This results in a scene becoming stretched, so the correct proportions of the objects are lost and our judgements are estimations are not valid anymore. We developed two solutions for this problem, both using interval arithmetic and stretching the bounding box. In fact the two ways only differ in how

the box is stretched. At first we just searched the dimension of the box that had the most length len and scaled other dimension to match

$$\begin{aligned} X &= [x_{min}, x_{max}] \cdot \frac{len}{x_{max} - x_{min}} \\ Y &= [y_{min}, y_{max}] \cdot \frac{len}{y_{max} - y_{min}} \\ Z &= [z_{min}, z_{max}] \cdot \frac{len}{z_{max} - z_{min}} \end{aligned} \tag{3.24}$$

Although the proportions are preserved in this case, the location of 3D point cloud inside the octree might end up in unpredictable places, as the left and right endpoints of intervals are scaled unevenly. That, actually did not cause a lot of problems, but for the sake of standardization we applied a slightly different method: addition and subtraction of equal values from each side of the intervals

$$\begin{aligned} X &= [x_{min} - (len - (x_{max} - x_{min}))/2, x_{max} + (len - (x_{max} - x_{min}))/2] \\ Y &= [y_{min} - (len - (y_{max} - y_{min}))/2, y_{max} + (len - (y_{max} - y_{min}))/2] \\ Z &= [z_{min} - (len - (z_{max} - z_{min}))/2, z_{max} + (len - (z_{max} - z_{min}))/2] \end{aligned} \tag{3.25}$$

This way the longest dimension remains the same, while the other two are placed in the middle of the octree.

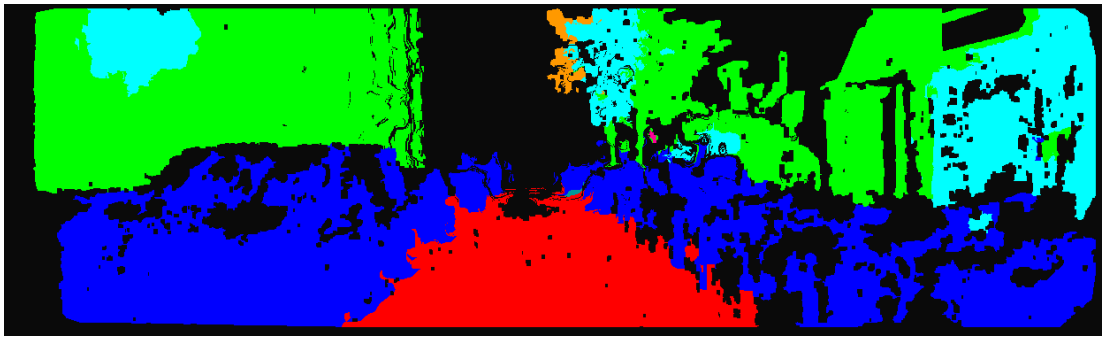
3.2.6 Parameters

There are parameters that could influence the final result, e.g. the quality of integration or the resolution of the octree. First of all we choose length of the side of the cube L. Having it fixed will result in highly different resolutions for different scenes and bounding boxes. We try to set L so that the side of the smallest possible voxel will be 15-25 cm. In practice $L = 2048$ is often enough for a good quality reconstruction, while spending not so much memory. After choosing L we have to deal with window size H. We use $H = 3 * l$, as it allows to capture bigger structures, like buildings we are interested in. For smaller objects with dense point clouds lesser H is recommended, because it reduces the number of outliers for a single object and for a single depth level. The sparser and larger the objects of interest are, the larger H is needed. H should be set carefully, because it influences an other important parameter, the histogram's bin size. The points are picked, with distance to the surface between $-H$ and H, so the interval $[-H, H]$ is mapped to $[0, 8]$. The discretization error becomes higher for larger H values. Also since H depends on cell size, the previously described problem happens for voxels of larger scale in the same octree. We usually put less trust in such voxels.

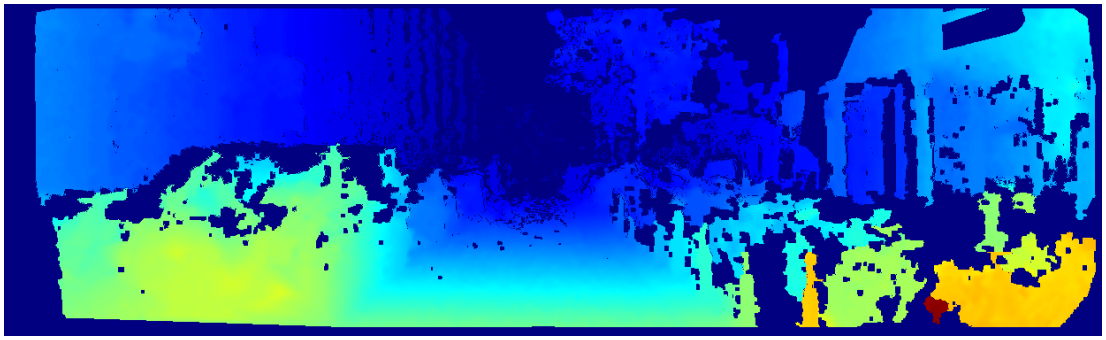
3.2.7 Input

The other side that influences the final octree is the input. Initially an octree needs the sets of 3D vertices, normals and semantic segmentation from both real world and backprojected model to run. We replace 3D vertices and normals with disparity maps, as PCL's normal computation method from [HRD+12] is capable of returning decent normal estimation for the scene. We tested the octree against two disparity methods: ELAS [GRU10] and SPS-Stereo [YMU14]. The SPS-Stereo method proved to be more precise as ELAS in a single frame, but after integration of high enough number of frames, the difference in octrees is very small and can be considered as irrelevant. We believe that the reason for such behavior is high number of outliers with high errors in ELAS. Although the influence of outliers is smoothed out in the octree, the faulty points still contribute and create voxels, where there should not be ones. The results of running ELAS and SPS-Stereo on an image from sequence are shown on Fig. 3.8c and Fig. 3.8d respectively. The problem was not only in the methods, but in the scene itself: lighting and shading conditions were a bit extreme, the left side is dark and the right side is so bright, it almost does not contain textures, as seen on Fig. 3.8a. That is also the reason, RGBD semantic segmentation algorithm (see Fig. 3.8b) fails for some regions, some pixels of a building (green) are recognized as of a tree (cyan). Supposedly we could reduce this error by training decision forests on scenes with variety of lighting conditions.

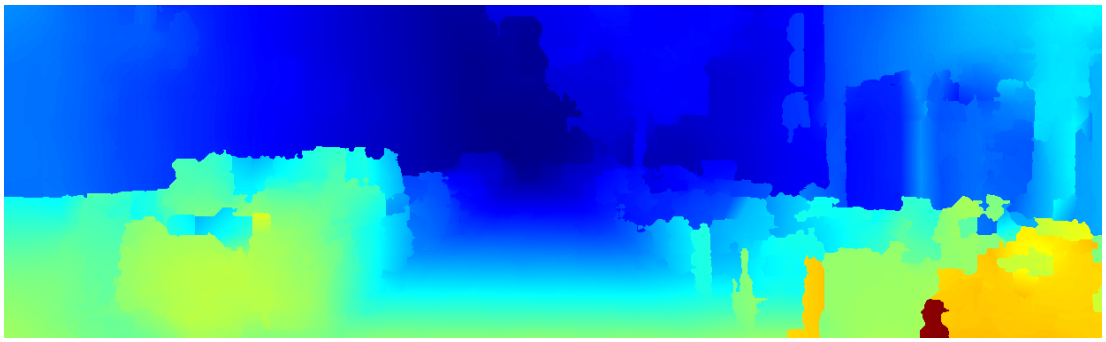
(a) Image from sequence 2011_09_26_drive_106_sync of raw KITTI dataset [GLSU13].



(b) Semantic segmentation by [OHE+16], with buildings green and road red.



(c) Output of ELAS.



(d) Output of SPS-Stereo.

Figure 3.8: The results of the preparation stage

3.3 Inference

In this step we use the information we gathered so far: the integrated depth from the octree, the building prior poses parsed from the OpenStreetMap and their correspondence to our observations. Our task is to optimize the pose of each building, but we approach this problem differently: we optimize points position, while leaving building static for implementation reasons. We solve this problem via minimization of the quadratic error between observed 3D points and surface of the generated OpenStreetMap model. The tool used for that purpose is Ceres Solver [AMO], which is widely used for non-linear optimization problems.

3.3.1 Problem formulation

Before the problem can be defined it is necessary to review different parts of the model. First, we have a model, built from the OpenStreetMap exported XML file, which is a good prior for building's pose. Second, we have 3D points of each frame, used for integration of the octree. As we do not use ground truth depth, the amount of error in depth estimation for these points is entirely up to stereo method used, in our case that was mostly SPS Stereo, because it yields more reliable disparity values. Finally, we have two semantic segmentations [OHE+16] and the one we generated from backprojecting the OSM model to the camera space. The first one is used to identify buildings on the image and the second one tells us, where the building's borders are, relative to each other.
The position optimization refers to finding an appropriate transformation, which can be defined as a matrix. Since we are working in 3D space and want to preserve translation, the matrix' size has to be 4×4, with 6 degrees of freedom (3 rotation and 3 translation components). But since the map only provides us with latitude and longitude, the position along the height axis is unknown, as well as rotation angles around two other axes. That leaves us with only three parameters, the set of which for building b_i we denote as π_i. The set of pi_i is denoted as π. It is important not to confuse it with the pose, as we only use π_i to get the pose

$$\pi_i = \{\theta^i, t_k^i, t_j^i\}$$

where θ^i denotes rotation of the building, while t_k^i and t_j^i show shifts in directions z and x respectively, the system is shown on Fig. 3.9. I, J, K notation is used, since the most of the computations are performed in octree coordinates. We can apply transformation π_i only to the building b_i or to the point of that belongs to the building b_i. So for the convenience we will use both $\pi(b_i)$ and $pi_i(b_i)$, although they refer to the same process. In following sections we will describe the potentials in more detail.

With that we now able to define our energy function, using unary ϕ and pairwise ψ potentials:

$$E(\pi) = \sum_i \phi(\pi(b_i)) + \sum_{i,j} \psi(\pi(b_i), \pi(b_j)) \tag{3.26}$$

3.3.2 Unary potentials

The initial idea is to optimize buildings to their true locations. For that we minimize the error between 3D points' positions and OSM model's surface iteratively via linear optimization. Hence the unary potential can be defined as

$$\pi_i = \arg_\pi \min \sum_j d(\pi_i(p_j^i))$$

Here p_j^i is a is a point that was initially identified as the one of building i. d stands for TSDF and denotes the distance to the surface at certain point in 3D space. The transformation T_i itself is then computed as follows

$$T_i = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_i & -\sin\theta_i & t_j^i \\ 0 & \sin\theta_i & \cos\theta_i & t_k^i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.27}$$

We assume the height to be constant for each building, therefore the corresponding term is set to zero, but we still use the rotation around Y- or I-coordinate, if in the octree system.

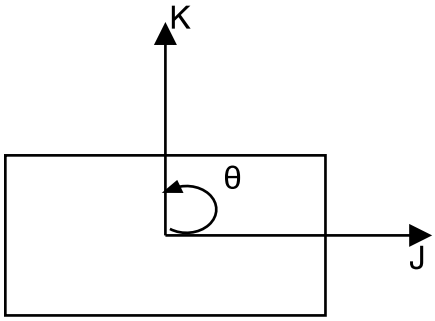


Figure 3.9: The parameters of the building transformation.

Thus, each 3D point is first converted into octree coordinates, then transformed via T_i. The problem here is that the transformed point has different TSDF value, so it has to be recomputed. For that octree interpolation, described in Sec. 3.2.4, is employed. As a little implementation enhancement, we also compute the derivatives, during the interpolation, so we do not have to do it twice. We need those for gradient estimation and for choosing the the direction in which we want to move the points, that belong to a building. The interpolated point is shown on

Fig. 3.10. It is straightforward for a quadtree case, and we can easily transform an octree cell of eight nodes to the one of the quadtree with only four nodes by applying linear interpolation along height dimension I. We have to first compute the TSDF values of intermediate nodes r and l

$$\begin{aligned} d(l) &= (1 - \triangle k)d(a) + \triangle k d(c) \\ d(r) &= (1 - \triangle k)d(b) + \triangle k d(d) \end{aligned} \tag{3.28}$$

Having those, $d(p)$ is very easy to find

$$d(p) = (1 - \triangle j)d(l) + \triangle d(r) \tag{3.29}$$

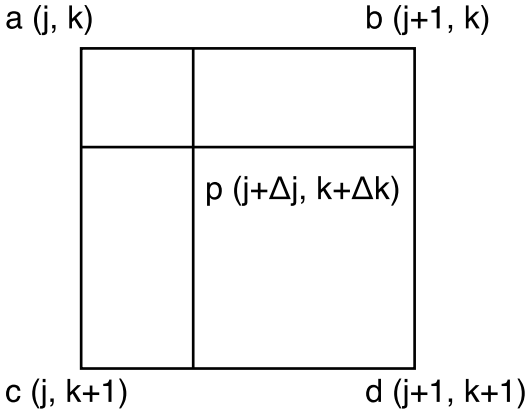


Figure 3.10: Interpolation, shown on the example of the quadtree. The derivatives are computed by subtracting TSDF values of left or top node from right or bottom node respectively.

The residuals are assigned interpolated TSDF values, because we want to minimize them, since the lesser the distance to the surface is, the closer the points are to the building. Still we have to consider the cases, when interpolation could not be done properly. As described in 3.2.4 this problem may be encountered on the borders of the scene or the objects, where the depth information was missing in the first place and the node was not created. In that case we count the point as not reliable and assign the penalty to the residual

$$r_i^j = \begin{cases} d(p_i^j), & \text{if } d(p_i^j) \text{ is valid} \\ 5 \cdot 2^{vd_{max} - vd_i^j}, & \text{otherwise} \end{cases}$$

We use voxel depth vd_i^j here to determine the penalty, that is scale-based penalty for each non-reliable point, because the nodes of bigger scale also yield higher TSDF values.
The whole unary potential computation pipeline is described in Alg. 5

Algorithm 5 Unary potentials minimization

```
1: procedure MinimizeUnaryPotentials
2:     //Initialization
3:     P := {P_1, P_2,.., P_n}, P_i ← ∅
4:     π := {π_1, π_2,.., π_n}, π_i ← {0, 0, 0}
5:     for all p_j ∈ points do
6:         for all b^i ∈ buildings do
7:             if p_j ∈ b^i then
8:                 p_j^i ← p_j
9:                 P^i := P^i + {p_j^i}
10:            end if
11:        end for
12:    end for
13:    while notConverged(r, ∇r) do
14:        for all P^i ∈ P do
15:            for all p_j^i ∈ P^i do
16:                T_i ← ComposeTransformationMatrix(π_i)
17:                p ← T_i * p_j^i
18:                TSDF, ∇TSDF ← d(p)
19:                if isValid(TSDF) then
20:                    updateResidual(TSDF)
21:                    updateGradient(∇TSDF)
22:                else
23:                    updateResidual(5 · 2^(vd_max − vd_j))
24:                end if
25:            end for
26:        end for
27:    end while
28:    return π
29: end procedure
```

3.3.3 Pairwise potentials

At first it may seem, that optimization of only unary potentials is sufficient to solve the task, which may still be true in some cases, for instance when there is only on building present. Although in the general case a new problem will arise: as buildings' positions are optimized independently, at some point they might collide, which does not seem like a real case scenario. To address this problem we employ pairwise potentials, which should detect and penalize collisions between buildings.

Collision detection of 3D objects may be tricky and computationally expensive, if done incorrectly. To reduce the complexity we limit the number of buildings, collisions of which we should track. We call those buildings *neighbors*. For each

building b_i we find a set of closest buildings n_i and each pair of b_i and $n_i^j \in n_i$ are neighbors. Note that neighborhood relation is not symmetrical in the general case, as we only use top k closest buildings for set n_i. Still, if there are symmetrical pairs in $\bigcup_i (b_i, n_i)$, they are deleted, because a single pair of neighbors is enough for the method to work and two symmetrical pairs will introduce unwanted bias.
After the number of buildings to check collisions on is reduced by neighbor pairs, collision detection itself is needed to be done. First for each pair of neighbors the faces $f_i^j \in faces(b_i)$ and $f_j^i \in faces(n_i^j)$, that represent the minimal distance between these two buildings, are found, along with their normals no_i^j and no_j^i respectively. As per previous assumption, that the walls are perpendicular to the ground, the normal's height component is equal to zero and the 3D collision problem may be reduced to a 2D case. Here it is sufficient to only check the distance between line segments and the normals direction. A useful note here is, that the collision problem has its most relevance in cases where the neighbors are very close or even sharing the common wall, see Fig. 3.11. Hence we replaced distance between line segments with two distances: a center c_i^j of the first face to the second line and a center c_j^i of a second face to the first.

Figure 3.11: A case of buildings sharing a common wall. Image from http://harlembespoke.blogspot.ru/2015/01/dwell-139-west-136th-street-townhouse.html

The residual is assigned zero, if collision is not happened (see eq. 3.30). The motivation is that we do not want to promote increase of the distance between buildings, instead we just want to keep them at the same distance, if possible. Therefore we penalize the collision by assigning the distance between faces to the residual, weighted by the number of points, that belong to this pair of buildings and participate in the optimization process

$$r_j^i = \begin{cases} dist \cdot |P_i| \cdot |P_j|, & \text{if } min(dot(c_i^j - c_j^i, no_j^i), dot(c_j^i - c_i^j, no_i^j)) > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.30}$$

The residual is defined, though to solve it as a linear optimization problem the derivatives need to be defined, which in this case is not trivial, especially since there are two sets of parameters present

3.3.4 Derivatives

The part worth mentioning is derivatives computation. We compute them for both potentials and follow [ZGWG15] on that. The parameter we optimize is $\pi = \{\theta, t_k, t_j\}$, where θ is a rotation angle and t_k and t_j are translations along axes K and J respectively. Ceres solver requires a Jacobian matrix, thus we have to find derivatives for all the parameters of $\frac{\delta d}{\delta \pi}$. This derivative is not directly computable, therefore have to apply chain rule

$$\frac{\delta d}{\delta \pi} = \frac{\delta d}{\delta \pi(p)} \cdot \frac{\delta \pi(p)}{\delta \pi} \tag{3.31}$$

Now it is fairly easy to extract derivatives for t_i and t_j as shown in Fig. 3.10. Given nodes a, b, c, d and the offsets $\triangle j$, $\triangle k$ we can compute TSDF values for vu, vd, l and r

$$\begin{aligned} d(vu) &= (1 - \triangle j)d(a) + \triangle j d(b) \\ d(vd) &= (1 - \triangle j)d(c) + \triangle j d(d) \\ d(l) &= (1 - \triangle k)d(a) + \triangle k d(c) \\ d(r) &= (1 - \triangle k)d(b) + \triangle k d(d) \end{aligned} \tag{3.32}$$

Now the derivatives for this particular node are

$$\begin{aligned} \delta j &= d(r) - d(l) \\ \delta k &= d(vd) - d(vu) \end{aligned} \tag{3.33}$$

The difficult part is $\delta\theta$. In order to figure out the derivative for θ we have to recap, how the transformation matrix T looks like:

$$T = \begin{pmatrix} & & & t_x \\ & R & & t_y \\ & & & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.34}$$

Algorithm 6 Binary potentials

procedure AddBinaryPotentials
  //Initialization
  $P := \{P_1, P_2, .., P_n\}$, $P_i \leftarrow \emptyset$
  $\pi := \{\pi_1, \pi_2, .., \pi_n\}$, $\pi_i \leftarrow \{0, 0, 0\}$
  for all $p_j \in$ **points do**
    for all $b^i \in$ **buildings do**
      if $p_j \in b^i$ **then**
        $p_j^i \leftarrow p_j$
        $P^i := P^i + \{p_j^i\}$
      end if
    end for
  end for
  while $notConverged(r, \nabla r)$ **do**
    for all $b_i, b_j \in neighbors$ **do**
      $T_i \leftarrow ComposeTransformationMatrix(\pi_i)$
      $T_j \leftarrow ComposeTransformationMatrix(\pi_j)$
      $f_i \leftarrow T_i * f_j^i$
      $f_j \leftarrow T_j * f_i^j$
      $dist \leftarrow min(dot(c_i^j - c_j^i, no_j^i), dot(c_j^i - c_i^j, no_i^j))$
      if $dist > 0$ **then**
        $updateResidual(dist \cdot |P_i| \cdot |P_j|)$
        $updateGradient()$
      else
        $updateResidual(0)$
      end if
    end for
  end while
  return π
end procedure

Since we already laid down an assumption, that buildings do not move along axis Y, we can set t_y to zero. t_x and t_z correspond to δj and δk respectively. Finally θ represents the rotation angle around axis Y. We also swap X and Y dimensions. Hence

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & t_j \\ 0 & sin\theta & cos\theta & t_k \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Given point $p = (i, j, k, 1)$ the transformed point p' is

$$\pi(p) = p' = Tp = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & t_j \\ 0 & sin\theta & cos\theta & t_k \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i \\ j \\ k \\ 1 \end{bmatrix} = \begin{bmatrix} i \\ j\cos\theta - k\sin\theta + t_j \\ j\sin\theta + k\cos\theta + t_k \\ 1 \end{bmatrix} \tag{3.35}$$

We truncate the homogeneous dimension for simplicity reasons, so $\pi(p) = p' = \{i, j\cos\theta - k\sin\theta + t_j, j\sin\theta + k\cos\theta + t_k\}$. We denote

$$\begin{aligned} f_1 &= p'_i = i \\ f_2 &= p'_j = j\cos\theta - k\sin\theta + t_j \\ f_3 &= p'_k = j\sin\theta + k\cos\theta + t_k \end{aligned} \tag{3.36}$$

Now the matrix, that results from $\frac{\delta\pi(p)}{\delta\pi}$ can be written down as

$$\frac{\delta\pi(p)}{\delta\pi} = \left(\begin{array}{c|ccc} & \delta\theta & \delta t_j & \delta t_k \\ \hline f_1 & 0 & 0 & 0 \\ f_2 & -j\sin\theta - k\cos\theta & 1 & 0 \\ f_3 & j\cos\theta - k\sin\theta & 0 & 1 \end{array} \right) \tag{3.37}$$

We also know, that

$$\frac{\delta d}{\delta\pi(p)} = \begin{bmatrix} 0 \\ \delta j \\ \delta k \end{bmatrix} \tag{3.38}$$

Finally from (3.37) and (3.38) we can compute

$$\frac{\delta d}{\delta\pi} = \frac{\delta d}{\delta\pi(p)} \cdot \frac{\delta\pi(p)}{\delta\pi} = \begin{bmatrix} \delta j(-j\sin\theta - k\cos\theta) + \delta k(j\cos\theta - k\sin\theta) \\ \delta j \\ \delta k \end{bmatrix} \tag{3.39}$$

So, the derivative of θ is defined through the other derivatives, which are directly inferable from interpolation of the octree.
The derivatives for pairwise potentials are more complicated case. We have to optimize two transformations simultaneously. We define a predicate $\pi^p(p_1, p_2)$, which arguments now are 2 points of two different buildings. As in (3.31) we have to apply chain rule again $\frac{\delta d}{\delta\pi_p} = \frac{\delta d}{\delta\pi_p(p_1,p_2)} \cdot \frac{\delta\pi_p(p_1,p_2)}{\delta\pi_p}$ Then, the derivative vector $\frac{\delta d}{\delta\pi_p(p_1,p_2)}$ could be written as

$$\frac{\delta d}{\delta\pi_p(p_1, p_2)} = \{\frac{\delta d}{\delta\theta_1}, \frac{\delta d}{\delta t_{j,1}}, \frac{\delta d}{\delta t_{k,1}}, \frac{\delta d}{\delta\theta_2}, \frac{\delta d}{\delta t_{j,2}}, \frac{\delta d}{\delta t_{k,2}}\}^T \tag{3.40}$$

The matrix $\frac{\delta\pi_p(p_1,p_2)}{\delta\pi_p}$ is similar to the one for unary potential, but with two sets of parameters, the size of matrix is also doubled along both dimension, with irrelevant values set to zero

$$\frac{\delta\pi(p)}{\delta\pi} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ -j_1 \sin\theta_1 - k_1 \cos\theta_1 & 1 & 0 & 0 & 0 & 0 \\ j_1 \cos\theta_1 - k_1 \sin\theta_1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -j_2 \sin\theta_2 - k_2 \cos\theta_2 & 1 & 0 \\ 0 & 0 & 0 & j_2 \cos\theta_2 - k_2 \sin\theta_2 & 0 & 1 \end{pmatrix} \tag{3.41}$$

And what is left now is to multiply the vector with a matrix, to get a a vector, that will be put into Jacobian matrix

$$\frac{\delta d}{\delta\pi_p} = \frac{\delta d}{\delta\pi_p(p_1,p_2)} \cdot \frac{\delta\pi_p(p_1,p_2)}{\delta\pi_p} = \begin{bmatrix} \frac{\delta d}{\delta t_{j,1}} \cdot \frac{\delta f_2}{\delta\theta_1} + \frac{\delta f_3}{\delta\theta_1} \cdot \frac{\delta d}{\delta t_{k,1}} \\ \frac{\delta d}{\delta t_{j,1}} \\ \frac{\delta d}{\delta t_{k,1}} \\ \frac{\delta d}{\delta t_{j,2}} \cdot \frac{\delta f_5}{\delta\theta_2} + \frac{\delta f_6}{\delta\theta_2} \cdot \frac{\delta d}{\delta t_{k,2}} \\ \frac{\delta d}{\delta t_{j,2}} \\ \frac{\delta d}{\delta t_{k,2}} \end{bmatrix} \tag{3.42}$$

The outcome of the inference process strongly depends on the Jacobians we just got. Changing weights for different dimension might enhance or deteriorate the results. In case of the raw sequence we tested our method on, promoting dimension t_j while others were set to zero, was successful. The reason is that the car trajectory and therefore road and buildings are aligned to axis K with small fluctuations. In general case this should not work, but with small workarounds we could cast almost any scene to such representation and boil it down to previously described problem.

3.3.5 Parameters and limitations

The aforementioned process already shows improvements in buildings' pose relative to the prior. Although it is possible to tune the parameters of the inference process to increase the quality of the pose estimation. For each scene these parameters may be very specific and very different. The first type of these parameters is weight of the derivatives we assign to Jacobian matrix. Our idea was to set those weights proportionally to the size and the other properties of the scene. The dimensions of the scene are very straightforward to compute, while knowledge about in which direction to move the houses is not clear. We made an assumption, that the houses are more likely to move along the direction perpendicular to the road, which is true in most of the cases. Thus the biggest weight is assigned

to this direction, in some experiments that was the only non-zero weight and still it has improved the final pose.
The second parameter was Huber loss [Hub77] threshold. This threshold sets the value after exceeding of which squared loss switches to linear. This part controls the interaction between unary and pairwise potentials. The number of pairwise potentials is much less, than that of unary, but their weight is much greater. If the threshold value is too high, the pairwise potential start to prevail, if too low, then the unary. This leads to another problem: when unary potentials start to dominate, the penalty for the collision drops and the houses end up one inside the other. In case pairwise potentials overwhelm the unary, the penalty of collision becomes too high to move houses towards each other. This is rather the limitation of our energy function, if at least one house moves in unwanted direction and generates a penalty for the cost, the whole move is rejected. And if the houses are optimized separately, the collision will be out of control. In our experiments we used threshold 100 in uniform weights case and 2 in case of one dominant weight. In future work we plan to make the energy function more flexible, so the described situation won't occur again.
There are other parameters as well, but they do not cause such an impact on the result. We experimented with changing line search methods, solver type and number of iterations, but as soon as they are more or less reasonable they cause neither performance drop nor gain.

3.4 Facade Separation

In Sec. 3.3 we encountered a problem: if buildings share a physical common wall, it becomes difficult to move them around without violation of constraints. One solution was to introduce pairwise potentials, to regulate the intersection relation between buildings. A second proposed solution was to assume such building as one in context of this thesis and then move the common walls inside that building. This step is also useful as a postprocessing step for the inference with both potentials, as due to our visual observations, such walls are misplaced very often. The overview of the whole facade separation algorithm can be found on Fig. 3.15

We discussed octree generation and inference so far. This section is also major part of the work, but somehow more independent, since it does not need the results of the previous two, although we still indirectly need the OSM model. Instead we require edges and 2D normals information. There are two possibilities to solve this challenge: use a homography to transfer the task into a different domain of camera facing buildings or separate the facades directly in the input images. The motivation for the first way is, that this task was solved multiple times for the facades, that face camera [RWL11]. In our case we did not have camera faced buildings (see Fig. 3.12), so we had to use homography transformation, to be able to apply those methods. Since the same building appears in several images we also had to find out which frame to use for the homography. So for each building we picked a frame, where the fraction of pixels semantically labeled as building $rgbd_b$ in the set of pixels p, that belong to this building osm_j, according to OSM model is maximal

$$index_j = argmax_i \frac{|p \in rgbd_b \bigcap p \in osm_j|}{|p \in osm_j|} \tag{3.43}$$

Figure 3.12: An image 132 from sequence 2011_09_26_drive_106_sync of raw KITTI dataset [GLSU13]

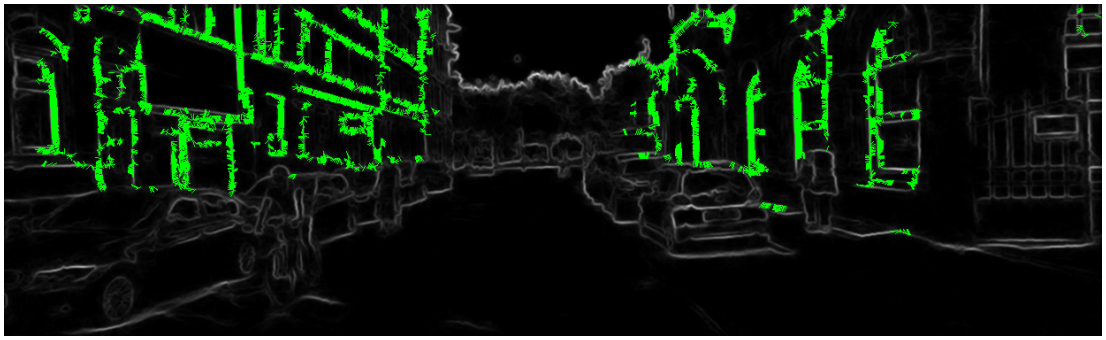
We also considered different criteria for picking out the best frame: number of pixels, that belong to building j, number of pixels, that both, belong to a building according to RGBD segmentation and to osm_j as well, etc. In general they gave

worse performance, due to multiple reasons, the most important one being, that the measure has to include as OSM as real semantics terms. If we do not use the OSM model, we cannot guarantee, that the necessary building is shown. If the real segmentation is not used, the frame with the biggest number of pixels labeled as building j might have an occlusion, a car for instance, which will greatly reduce our ability to analyze the edges of the building. The divisor is added due to regularization reasons, as we prefer smaller, not occluded regions to larger ones, but with obstacles. We also use the same frame for ground estimation of the building, although we perform averaging with neighboring frames, to reduce outliers effect.

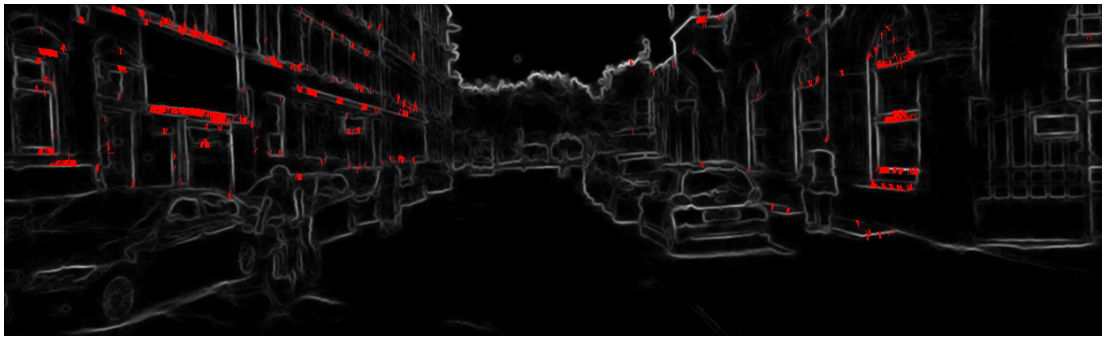
Our task is to find separation lines between houses. We acquire an edge image

Figure 3.13: Result of running edge detection algorithm described in [DZ14] [DZ13] [Dol] on image 132 from sequence 2011_09_26_drive_106_sync of raw KITTI dataset [GLSU13]

from [DZ14], but it is still not possible to decide on edges just from that information. On this we follow the idea of [Mat13]: the separation must be vertical and it should not be intersected by horizontal lines. The second condition is unrealistic though, therefore we soften it to the state of penalty for each horizontal line intersecting a vertical one. But before we can formulate the separation condition, we need the means to detect vertical and horizontal edges. We can find out that for a single edge by calculating the angle between ground plane and the edge itself. The ground plane might be found in different ways. The simplest one is to use pixels, classified as 'Road' by semantic segmentation, for constructing a 3D point cloud and then apply PCA to find the road plane. Although we achieve better results by applying method from [CKZ+15]. The edge has more options to choose from. It is possible to build a line from a set of pixels by applying RANSAC [FB81], but we have to add new constraints to avoid including unwanted points and run it multiple times, as we have many edges to find, which is quite expensive. We apply a simpler solution by using short sequence of filters: Gaussian and Sobel. The method from [DZ14] leaves quite some noise, thus we first smooth it out with Gaussian filter and then apply Sobel filter to get 2D derivatives (see Fig. 3.14). We then compute the 2D normal orientation angle α from that information for each pixel $p = (p_x, p_y)$, marked as edge. Using disparity and vertices map we find



(a) Vertical normals



(b) Horizontal normals

Figure 3.14: Normals computed via cascade of Gaussian and Sobel filters on image 130 from sequence 2011_09_26_drive_106_sync of raw KITTI dataset [GLSU13]

the corresponding 3D points for the 2D pixels with horizontal or vertical normals. Knowing α we can find a pixel p', which is as close to the extracted by RANSAC line model as possible, that is find such k, that following holds

$$\begin{aligned} p'_x &= p_x - k \cdot cos(\alpha) \\ p'_y &= p_y - k \cdot sin(\alpha) \\ k &= argmin_k(|p'_x - round(p'_x)| + |p'_y - round(p'_y)|) \end{aligned} \tag{3.44}$$

Using p and p' we can extract vertices v and v', we got from the disparity map earlier. We construct a 3D normal now

$$\overline{n} = \frac{v' - v}{||v' - v||} \tag{3.45}$$

and then calculate a dot product between $\overline{n}$ and ground normal $\overline{gn}$

$$\beta = acos(\overline{n} \cdot \overline{gn}) \tag{3.46}$$

The angle β is an angle between normals of the ground plane and edge normal, that means it is also an angle between edge and the ground plane themselves

as well. The pixels, that have 3D normals close to parallel to the ground are labeled as vertical, close to perpendicular - as horizontal. Each pixel marked that way casts a vote into a a 2D voting grid, that is extended over the whole scene. In each cell of the grid vertical pixels add the probability of a separation, while horizontal ones decrease it. In fact, we decided to use two separate grid for this purpose, to preserve more information up to decision step. We omit the height component in the grid, due to small variability of this parameter. Given the point $p = \{p_x, p_y, p_z\}$, the projection to the grid is therefore defined as

$$g_x = \frac{p_x - x_{min}}{s_x} \tag{3.47}$$

$$g_z = \frac{p_z - z_{min}}{s_z} \tag{3.48}$$

x_{min} and z_{min} are the lower bounds of the scene's bounding box, described in Sec. 3.2.5. s_x and s_z represent a step, a parameter, that defines grid's resolution. This parameter has to be balanced and may have critical impact on performance. Too small steps will result into a grid with finer resolution, so the votes will be cast into different cells and maxima might end up in a wrong place. The computation complexity will also increase. Larger steps and coarser resolution will have blurring effect on a grid. The maxima cell will probably be correct, but due to a big cell size, the separation might shift from the correct location. For our experiments we keep both s_x and s_z around 15 cm. and achieve decent results.

There is a consequence of excluding height though: because of multiple ground planes in different frames, the variability of the height might change, which will lead to creation of regions, where the points will be distributed more sparsely or densely, than they should be. An alternative would be to project directly on the ground plane, which will exclude the projection error for one particular frame, but with adding more frames and more ground planes, the previously described problem will take place again and impact might be even worse.

We also perform edge thinning via non-maxima suppression procedure, modified from Canny Edge Detection algorithm. Given edge image and normal map, we acquired from applying Gaussian and Sobel filter chain we use the same procedure as we used in Eq. (3.44), but this time with constraint, that p' falls into 8-neighborhood of p. We also find a pixel p'' in reverse direction of the normal, which is easily derivable from Eq. (3.44)

$$p''_x = p_x + k \cdot cos(\alpha) \tag{3.49}$$

$$p''_y = p_y + k \cdot sin(\alpha) \tag{3.50}$$

So we set p to zero, if either p' or p'' has higher intensity. An important note here is, that we apply the changes to the copy of the image and not in-place, since it would lead to unpredictable behavior of the algorithm. The thinned image is used thereafter as an indicator: if the intensity value of the pixel is higher than

user-defined value, the pixel's vote is taken into account.
After all the frames are integrated, we start to process the voting grid. First we include the priors: lines, separations that are extracted from the 3D OSM model by finding the sides of the buildings, that face road and project them onto the grid. Around each prior inside some window we find k maximal peaks (in our experiments we used $k = 3$). The final decision step is to find visual cues for the separation. We project the peaks back in an image, the frame that we use for building j is the best frame $index_j$, that we already picked out before. For each line we get, the pixels from both sides are gathered. We use color histograms of each side for that goal

$$\begin{aligned} r_r &= \sum_{\substack{l_u<u<l_u+w_u \\ l_v^b \le v \le l_v^t}} \frac{p_r(u,v)}{n_r} \cdot D(u,v) \\ r_l &= \sum_{\substack{l_u-w_u<u<l_u \\ l_v^b \le v \le l_v^t}} \frac{p_r(u,v)}{n_l} \cdot D(u,v) \end{aligned} \tag{3.51}$$

where l_u, l_v^b, l_v^t are a horizontal location and a lower and upper ends of the line, $p_r(u,v)$ is a red component of the pixel at location (y,v) and w_u is an user-defined horizontal window size to take samples from. n_r and n_l are the pixel numbers on different sides, they both roughly equal to $w_u \cdot (l_v^t - l_v^b)$, but since it only makes sense to use pixels, that are labeled as buildings, this number might drop. $D(u,v)$ is a flag function, that tells whether pixel is labeled as building or not

$$D(u,v) = \begin{cases} 1, & \text{if label(p(u, v) = building} \\ 0, & \text{otherwise} \end{cases} \tag{3.52}$$

r_r and r_l are the red components of the histogram on the right and left side respectively. g_r, g_l, b_r and b_l are defined by the same means. We normalize the results, as the histogram intersection may not work correctly otherwise. We then compare them with histogram intersection and take the line with the smallest value

$$k_{opt} = argmin_{i,0\le i<k}(min(r_l^i, r_r^i) + min(g_l^i, g_r^i) + min(b_l^i, b_r^i)) \tag{3.53}$$

Although color histograms are not a reliable measure on its own, due to previous filtering steps it shows advantageous results. Still the full method might benefit from more complex texture analysis. We also experimented with different histogram distances, like simple Euclidean or Earth Mover Distance [RTG98], but it did not show any significant performance gain or loss. More details as well as full algorithm could be found in Alg. 7 After getting line location inside 2D grid we want to get its coordinates in 3D world space. Although we omitted the height

Algorithm 7 Search for back Neighbor

```
1: procedure SeparateFacades
2:     //preparation step
3:     for all f ∈ Frames do
4:         gn ← GetGroundNormal(f)
5:         Edges ← GetEdges(f)
6:         Dx ← SobelDx(Gaussian(Edges, Size = 11x11))
7:         Dy ← SobelDy(Gaussian(Edges, Size = 11x11))
8:         // we only get orientation angle of the 2D normal here
9:         Normals2D ← atan(Dy/Dx) − π
10:        for all (p ∈ Pixels do
11:            v ← Vertices(p)
12:            a ← Normals2D(p)
13:            pn ← GetPixelInDirection(a)
14:            vn ← Vertices(pn)
15:            n3d ← GetNormal(v, vn)
16:            diff ← GetAngleBetween(n3d, gn)
17:            if |diff|%π < π/6 then
18:                CastHorizontalVote(grid, v)
19:            end if
20:            if |diff − π|%π < π/6 then
21:                CastVerticalVote(grid, v)
22:            end if
23:        end for
24:    end for
25:    for all b ∈ buildings do
26:        f ← PickBestFrame(b))
27:        N ← GetNeigbors(b))
28:        for all n ∈ N do
29:            bfacade ← GetSideFacingRoad(b)
30:            nfacade ← GetSideFacingRoad(n)
31:            prior ← GetCommonEdge(bfacade, nfacade)
32:            Peaks ← FindPeaksAround(prior, grid)
33:            for all peak ∈ Peaks do
34:                line2D ← GetLineInImageSpace(peak)
35:                lhist ← GetLeftColorHist(line2D)
36:                rhist ← GetRightColorHist(line2D)
37:                D ← D + {GetDistance(lhist, rhist)}
38:            end for
39:            maxdist ← Max(D)
40:            separation ← GetPeak(Peaks, maxdist)
41:            S ← S + separation
42:        end for
43:    end for
44:    return S
45: end procedure
```

parameter previously, we can still get a decent guess using the ground plane of the best frame for the separation. We first recover p_x and p_z

$$\begin{aligned} p_x &= s_x \cdot g_x + x_{min} \\ p_x &= z_{max} - s_z \cdot g_z \end{aligned} \tag{3.54}$$

Given ground plane representation

$$Ax + By + Cz + D = 0 \tag{3.55}$$

the y is directly computable

$$y = \frac{Ap_x + Cp_z + D}{-B} \tag{3.56}$$

The height determined in such way is almost always correct. Natural exceptions are incorrect input, such as wrong ground plane or the cases, when the building's begin does not coincide with the intersection with the ground plane. We minimize the effects of the first by averaging the ground plane over multiple neighboring frames. We do not consider the second, since it is an extremely rare occasion and was not observed in the testing sequences. In the future work though we may solve this problem by applying similar texture comparison, as we did for final decision about building separation line.

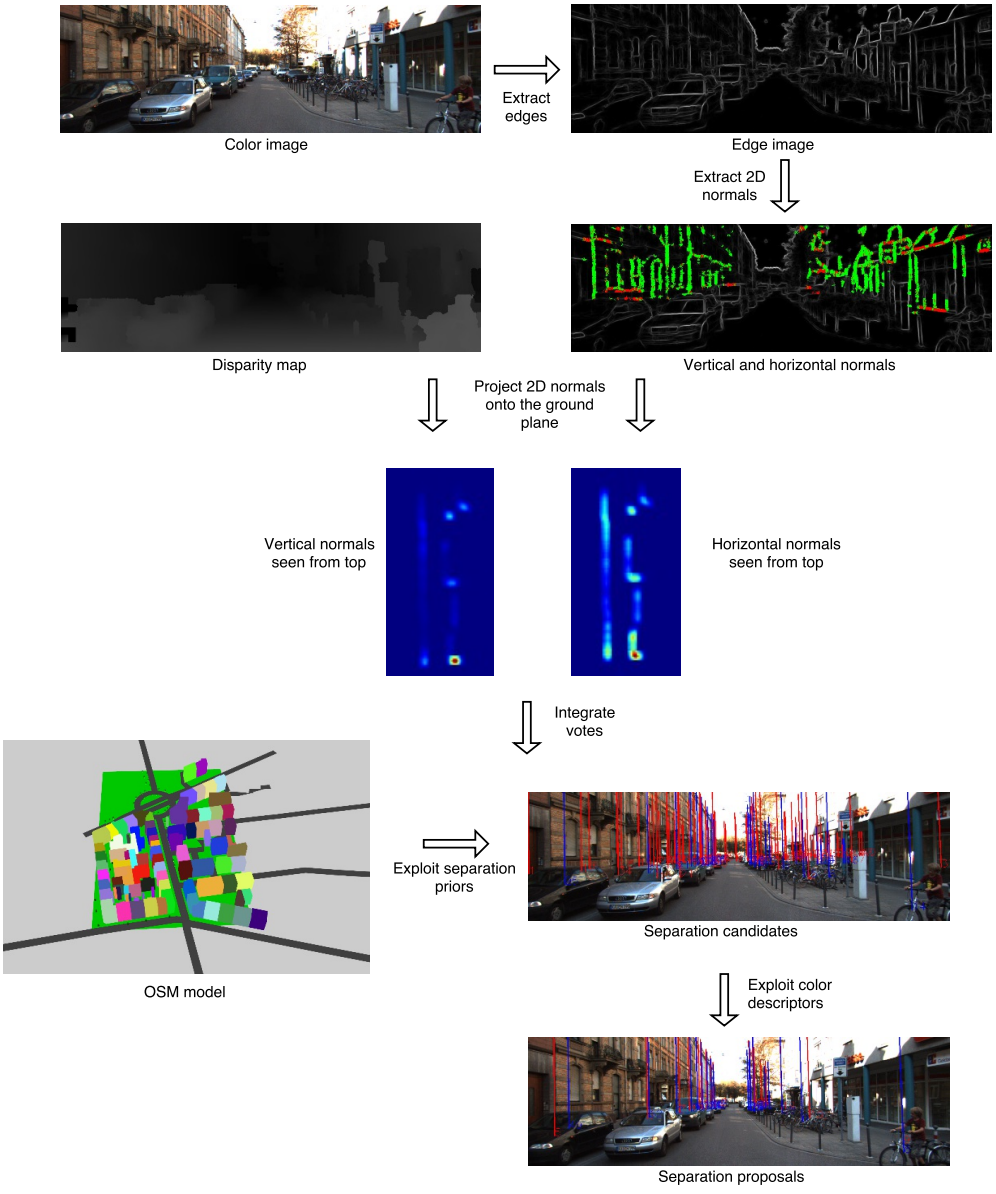


Figure 3.15: Overview of the facade separation pipeline

4 Evaluation

For each stage two kinds of evaluation were performed: one on a toy example and a second one on a real data with the ground truth. Each of the three stages is discussed below.

4.1 Datasets

This method was calibrated on the sequence 2011_09_26_drive_0106_sync of the KITTI raw dataset, but also tested on sequences 13 and 19 of the KITTI tracking dataset as well as on Oxford RobotCar Dataset [MPLN17]. The reason for the low number of testing material is that a lot of potentially interesting sequences failed to fulfill at least one of the two conditions:

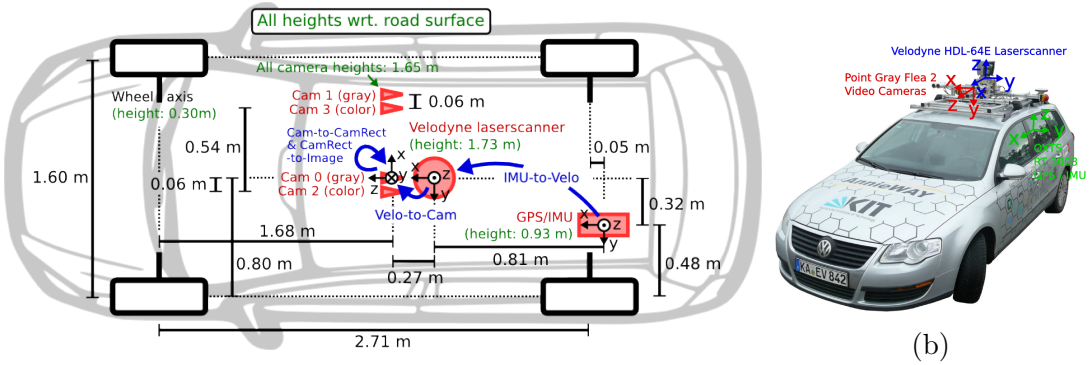
1. Having embedded GPS coordinates of a car
2. Having buildings

The first one is needed, as we need to align the map and the images, because the localization problem is not a part of this thesis. The second one's necessity is obvious, since the goal of this work is to optimize buildings' locations. For the facade separation part we need 2 more conditions:

1. There must be buildings with a common wall
2. The buildings with a common wall should be visually separable

Without buildings with a common wall, the step does not make any sense, the second part of the condition exists for a simple reason, that we have to generate the ground truth by labeling different buildings. This can be omitted, if ground truth is provided, nevertheless the method may fail, since it is texture-based and might not find the separation (see Fig. 4.1).

Figure 4.1: An example of visually inseparable buildings (shown with a red looped line). The image 88 from sequence 2011_09_26_drive_0106_sync of the KITTI raw dataset



(a) (b)

Figure 4.2: On (a) the scheme of the experimental setup in KITTI is shown. On (b) one can see a fully equipped vehicle

4.2 Experimental setup

4.2.1 KITTI

The KITTI dataset is recorded with always the same setup, which is shown on Fig. 4.2a. The calibration parameters might slightly differ from sequence to sequence, but overall the setup stays static. The following elements are used in this setup

- 1 Inertial Navigation System (GPS/IMU): OXTS RT 3003
- 1 Laserscanner: Velodyne HDL-64E
- 2 Grayscale cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3M-C)
- 2 Color cameras, 1.4 Megapixels: Point Grey Flea 2 (FL2-14S3C-C)
- 4 Varifocal lenses, 4-8 mm: Edmund Optics NT59-917

According to the documentation, the spin rate of the laser is 10 frames per second. The laser captures approximately 100000 points per spin with the vertical resolution of 64. The cameras, that are close to parallel with the ground plane, have recording rate of 10 frames per second and are synchronized with the laser cycles. The size of the images after cropping via libdc's format 7 mode is 1382 × 512, but it is decreased by rectification. The car, equipped with this setup is shown on Fig. 4.2b

4.2.2 Oxford RobotCar

The Oxford RobotCar dataset contains recordings from the streets of Oxford from May 2014 to December 2015. The specificity of this dataset is, that there is only a single route, which is repeated over 100 times, but under different weather and lighting conditions. The setup can be seen on Fig. 4.3. It contains:
Cameras:

- 1 × Point Grey Bumblebee XB3 (BBX3-13S2C-38) trinocular stereo camera, 1280×960×3, 16Hz, 1/3" Sony ICX445 CCD, global shutter, 3.8mm lens, 66° HFoV, 12/24cm baseline
- 3 × Point Grey Grasshopper2 (GS2-FW-14S5C-C) monocular camera, 1024×1024, 11.1Hz, 2/3" Sony ICX285 CCD, global shutter, 2.67mm fisheye lens (Sunex DSL315B-650-F2.3), 180° HFoV

LIDAR:

- 2 ×SICK LMS-151 2D LIDAR, 270° FoV, 50Hz, 50m range, 0.5° resolution
- 1 × SICK LD-MRS 3D LIDAR, 85° HFoV, 3.2° VFoV, 4 planes, 12.5Hz, 50m range, 0.125° resolution

GPS/INS:

- 1 × NovAtel SPAN-CPT ALIGN inertial and GPS navigation system, 6 axis, 50Hz, GPS/GLONASS, dual antenna

Since it was published very recently, we did not have the chance to try our method on the whole dataset, but only on a small part of it. We used a subsequence of frames from 4381 to 4874 of the dataset 2014-07-14-14-49-50, as it contained a lot of buildings.

4.3 Results

4.3.1 Integrated Depth Evaluation

We first show with a toy example, that the depth integration works and gives reasonable results, then the improvement is shown on a real case image sequences.

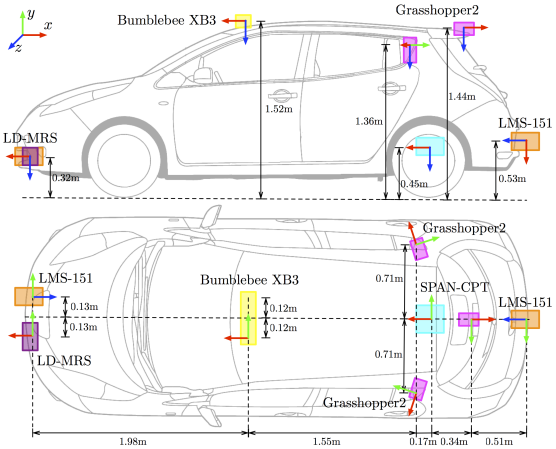
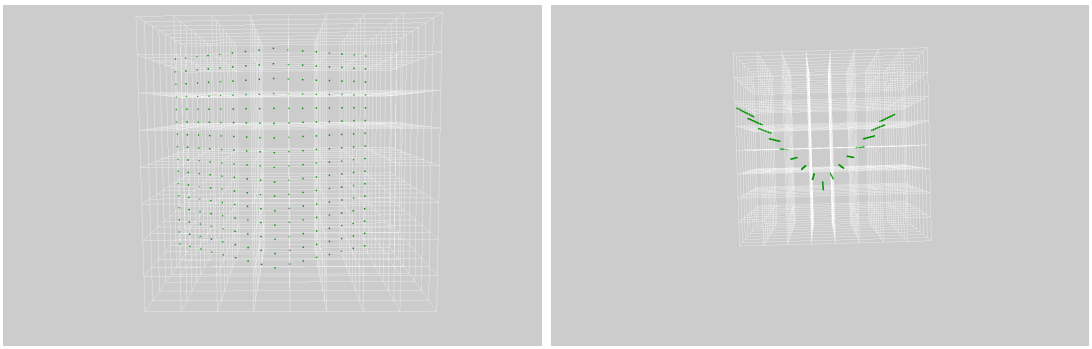


Figure 4.3: The setup of Oxford RobotCar dataset [MPLN17]

Toy example

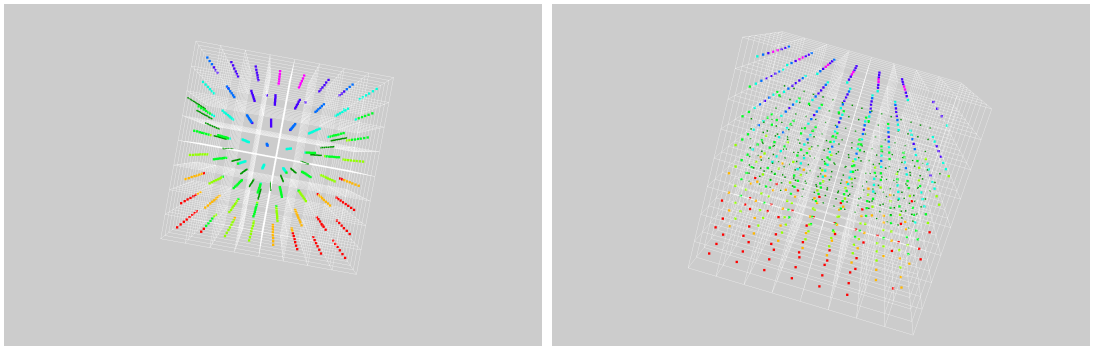
We created a controlled environment to get a proof, that the depth integration with an octree works. As a 3D figure we chose an intersection of two planes in a form of a corner (see Fig. 4.4. This kind of shape has several motivations. A simple plane will discover only a small amount of the irregularities in the algorithm, while complicated surfaces have rare occurrence in a real world scenario. Moreover they might mislead a human inspector, since it will be difficult to see which points contribute to which node of the octree. One more advantage is that a corner-like surface is a case which happens very often in a real environment. It also shows, that the algorithm, which heavily depends on surface normals, can deal with a lot of different orientations and scales and returns a plausible result.



(a) Front view (b) Top view

Figure 4.4: A toy example with a corner-like shape

The results although depend on voxel integration window size h. For a scene of size $16 \times 16 \times 16$ an optimal value was 1.5. The results are shown on Fig. 4.5. As expected, the nodes, that are far from the camera, which is in the middle, are given larger scale and are not split.



(a) Top view

(b) View from right. In top right corner the nodes of higher scale are seen, centered at a different position as the rest

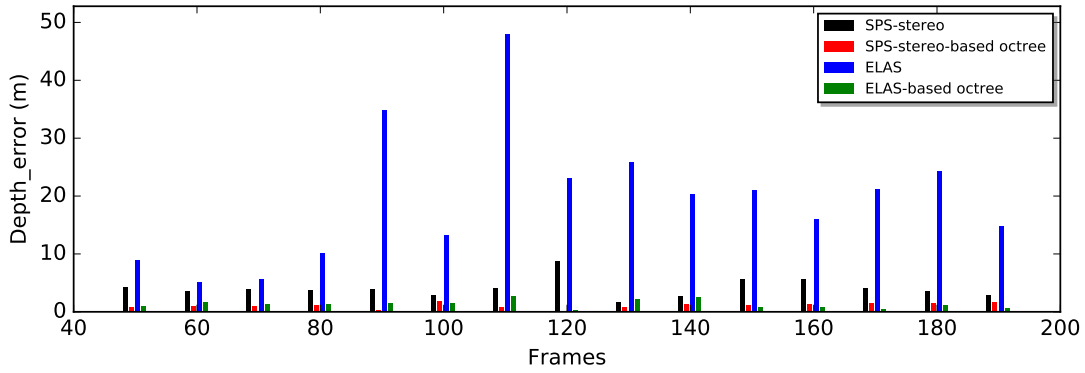
Figure 4.5: An toy example with centers of the octree nodes with distance-based color coding. From highest distance to lowest: red, orange, light green, cyan, blue, pink; between light green and cyan lies zero intersection level, thus red and blue have different sign and represent different sides of the surface

Real case testing

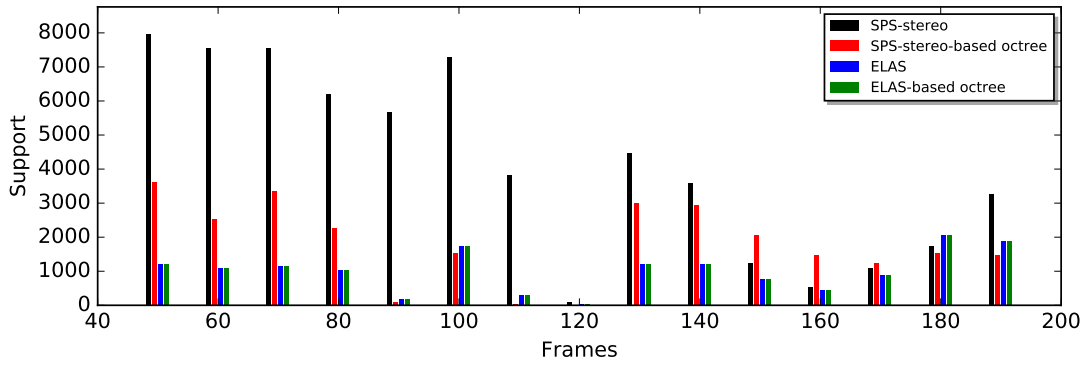
For a real case we used sequence 2011_09_26_drive_106_sync of raw KITTI dataset. We took every 10th image, from frame 50 till 200. As a ground truth LIDAR's point clouds are used, projected into the image space. We take depth value generated by ELAS, SPS-Stereo and Fusion method at the pixels of the image, which are semantically labeled as buildings and where ground truth exists. And as we know the precise 3D positions of those points, we can interpolate the octree to get the distance to the surface, which in this case shows an amount of error, while we have to take absolute difference in case of non-integrative methods. The average errors along with their support (number of points, where both, estimation and ground truth exist) are shown on Fig. 4.6. It shows, that in observed sequence SPS-Stereo is able to outperform ELAS algorithm. The reason might be, that ELAS' estimation includes a lot of outliers, while SPS-Stereo's planar assumption reduces or completely nullifies their effect. However, both these methods are not able to compete with depth integration approach. While there is a large difference between SPS-Stereo and ELAS, the difference between octrees, based on those methods is minor, since the outlier influence is minimized via integration and voting.

4.3.2 Facade Separation Evaluation

Facade separation on the other hand is much easier to evaluate visually, so a sanity check by a human inspector replaced the creation of a controlled environment. But the problem is with a real case evaluation, as the ground truth of buildings



(a) Depth error on a sequence 2011_09_26_drive_106_sync of raw KITTI dataset [GLSU13].



(b) Support of depth error computation on a sequence 2011_09_26_drive_106_sync of raw KITTI dataset

Figure 4.6: Comparison of different depth/disparity estimation methods

separations is not provided with KITTI dataset. Thus we created a ground truth ourselves, by labeling different buildings in some images of the sequences, where they are visually separable (see Fig. 4.7). Since the focus was lying in the separations, we did not care about fine details doing the labeling, but only drew a quality separation.

Figure 4.7: An example of a labeled image

The separation procedure is performed on a 2D grid, representing the ground plane, not in image space. Therefore a projection must be defined, that will match the separation lines to the image space lines. In Sec. 3.4 we have shown, how to do the transformation from image space to 3D. Now we show how to do the inverse.
Given the line $l = (l_x, l_y, l_z)$ and the ground plane $G_i = (A, B, C, D)$ for frame i, the lower endpoint of the line lies at

$$\begin{aligned} l_x^{low} &= l_x \\ l_y^{low} &= \frac{Al_x + Cl_z + D}{-B} \\ l_z^{low} &= l_z \end{aligned} \tag{4.1}$$

The upper point can then be located by following the direction of the ground plane's normal

$$\begin{aligned} l_x^{up} &= l_x^{low} + A \cdot h \\ l_y^{up} &= l_y^{low} + B \cdot h \\ l_z^{up} &= l_z^{low} + C \cdot h \end{aligned} \tag{4.2}$$

where h is a height parameter. Ideally it should be equal to the height of the current building, but it is usually enough to set it to some realistic constant. In our experiments we used $h = 20m$.
The next is transformation to the image space. We must first transform the points according to the position of the car, when the frame i was taken using pose matrix P_i provided by KITTI. As the matrix P_i originally transforms points from coordinate space of frame i to the one of frame 0, we have to use the inverse of P_i. The last step should be projection into the image space with a projection matrix Pr. Both matrices also require homogeneous points to work with, therefore we extend l^{up} and l^{low} with fourth coordinate. Thus the full transformation is

$$\begin{aligned} p^{low} &= Pr \cdot P_i \cdot l^{low} \\ p^{up} &= Pr \cdot P_i \cdot l^{up} \end{aligned} \tag{4.3}$$

The final operation should be division of $p^{low} = (p_x^{low}, p_y^{low}, p_d^{low})$ and p^{up} by the third coordinate

$$\begin{aligned} p_x^{low} &= \frac{p_x^{low}}{p_d^{low}} \\ p_y^{low} &= \frac{p_y^{low}}{p_d^{low}} \\ p_x^{up} &= \frac{p_x^{up}}{p_d^{up}} \\ p_y^{up} &= \frac{p_y^{up}}{p_d^{up}} \end{aligned} \tag{4.4}$$

Now, x represents a width coordinate inside the image, while y shows vertical position. We calculate the average along the width axis

$$\overline{x} = \frac{p_x^{low} + p_x^{up}}{2} \tag{4.5}$$

We do the same for a ground truth model. As you may recall from Sec. 3.4 we do not extract points, but line models in form $y = mx + b$. We only have to pick vertical coordinates y^t and y^b that denote endpoints of the separation. We used $y^t = 374$ and $y^b = 0$ as they create a maximal possible vertical margin. Hence the average along horizontal axis for the ground truth is

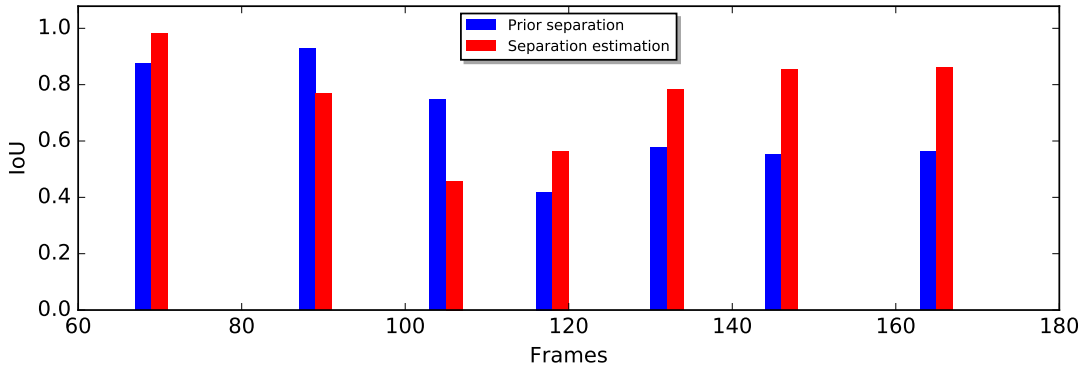
$$\overline{x_{gt}} = \frac{(y^t - b) + (y^b - b)}{2m} = \frac{374 - 2b}{2m} \tag{4.6}$$

Each ground truth separation was matched with a prior separation extracted from the OSM directly and with our separation estimation. Each building has two separations a priori. We employ intersection over union for priors and estimations to be compared with ground truth. The results can be seen at Fig. 4.9 and Fig. 4.8. Our method is able to outperform the prior separation on most cases, while maintaining the prevailing support, which shows, that it hits true interval more often than a prior does.

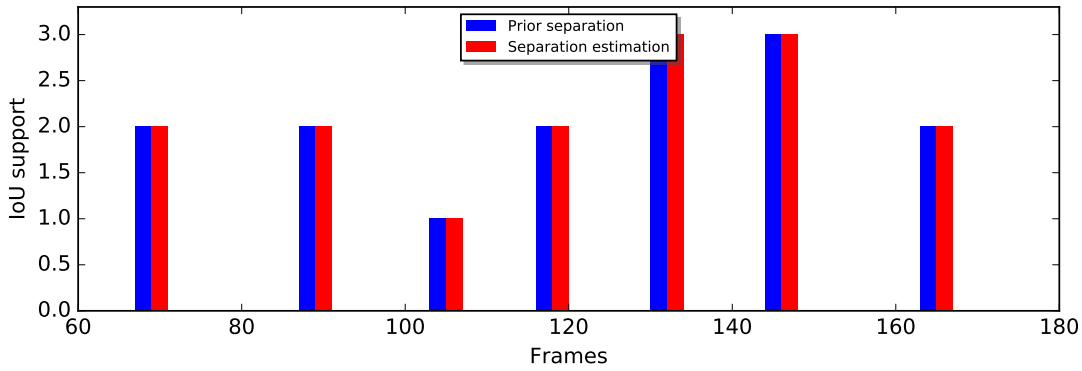
4.3.3 Building Pose Evaluation

For this part we used the same setup as described in Sec. 4.3.1. To test the inference process and non-linear least squares initial corner-like point cloud was shifted around inside the scene and then was optimized by our algorithm. The pose estimation and enhancement procedure was developed iteratively in the first place, even for a toy example. We started with the changes along single dimension, then two dimensions simultaneously and at the end we added rotational component. The results are shown in Fig. 4.11.

Though for the toy example the results are promising, there are certain difficulties that we face, when starting to evaluate in a real case scenario. That is, there is no ground truth poses provided in KITTI or any known dataset. There were several possibilities to handle this problem. First was to create the ground truth manually, with moving the buildings around inside OSM model, until they coincide perfectly with the LIDAR depth estimations. But even for one sequence this might be very difficult to do, since visual inspection of humans is not ideal and moved objects may also be faulty in the end. Secondly an idea of taking cadastre maps emerged, although it was soon rejected due to troubles of getting those from local government and legal issues. Finally we decided to perform the evaluation in 2D image space, by backprojecting the optimized model to an image and using the depth laser data on pixels, that are manually labeled as buildings. By a first glance it seems off, that we perform the optimization in 3D and then



(a) Average intersection over union values on buildings from a sequence 2011_09_26_drive_106_sync of raw KITTI dataset [GLSU13].

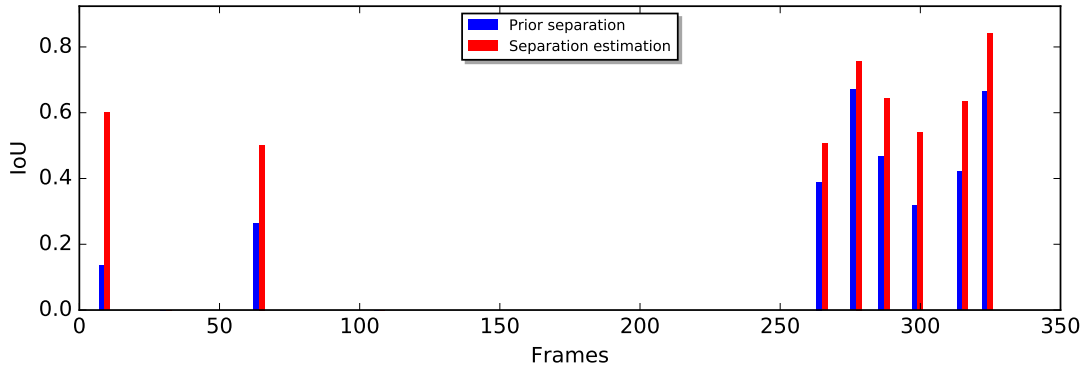


(b) The number of buildings participated in computation on a sequence 2011_09_26_drive_106_sync of raw KITTI dataset

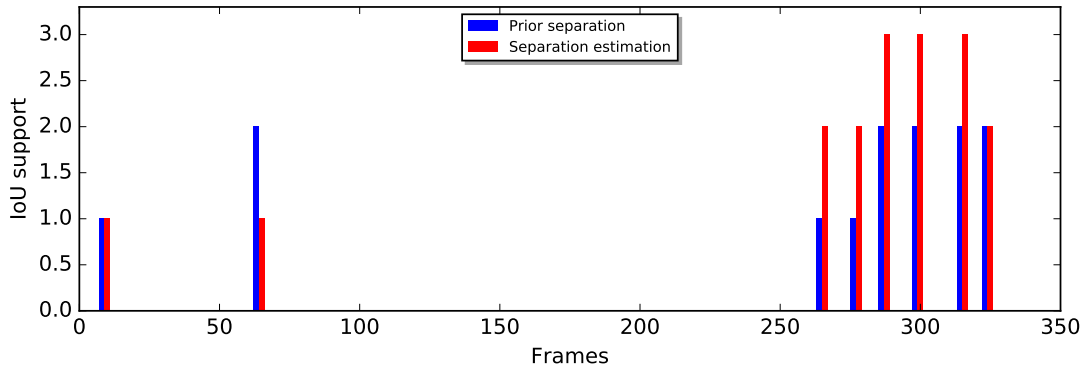
Figure 4.8: Facade separation estimation vs prior methods

evaluate the results in 2D. The motivation for that is that when moving through the image sequence, we only see one side of the buildings. The points we sample are also lying on this side, so the whole method is based on facades. We still have the 3D information from OSM model, like the position or shape of the building, but we only assume, that those are reasonable, as we cannot prove it through street-view observations. Hence projecting in 2D domain does not lose a lot of information for us. Also it is probably the only way to measure the changes, that were introduced by optimization. The resulting comparison is provided in Fig. 4.12

We project both, prior and optimized models into an image space via VTK. As those are just models, we can directly read the depth buffer to read the depth values at each pixel. Those depth values are then compared to laser measurements, which are taken as ground truth. We only use those pixels, that are labeled as buildings in ground truth labeling and have laser and model depth measurements. To construct a laser depth map, we need to iterate over the laser point cloud and project each point into an image. In KITTI setup laser points are in different



(a) Average intersection over union values on buildings from a sequence 0013 of tracking KITTI dataset [GLU12].



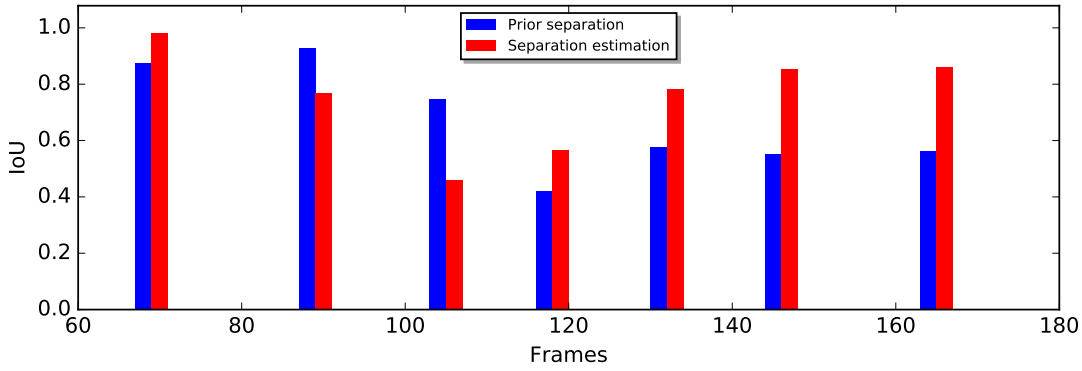
(b) The number of buildings participated in computation on a sequence 0013 of tracking KITTI dataset

Figure 4.9: Facade separation estimation vs prior methods

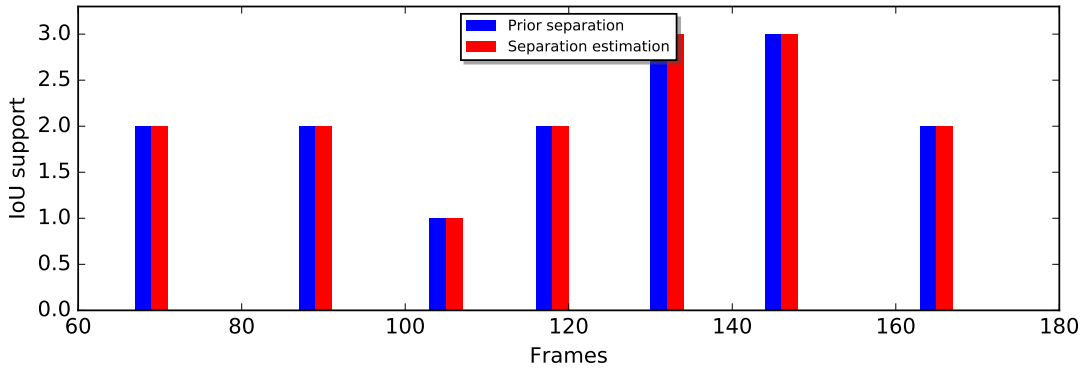
coordinate system, as the camera points, so the first thing to do is to cast it to that system with matrix Tr, provided by KITTI

$$\begin{bmatrix} 7.533745e-03 & -9.999714e-01 & -6.166020e-04 & -4.069766e-03 \\ 1.480249e-02 & 7.280733e-04 & -9.998902e-01 & -7.631618e-02 \\ 9.998621e-01 & 7.523790e-03 & 1.480755e-02 & -2.717806e-01 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

After that we apply the same sequence of matrix multiplications as in Eq. (4.3) with division of resulting vector by third component. The last thing we check is that the image space coordinates of the point are within boundaries of the image, e.g. $0 \leq x < width$ and $0 \leq y < height$. After all the iterations are done we have a laser depth map ready.

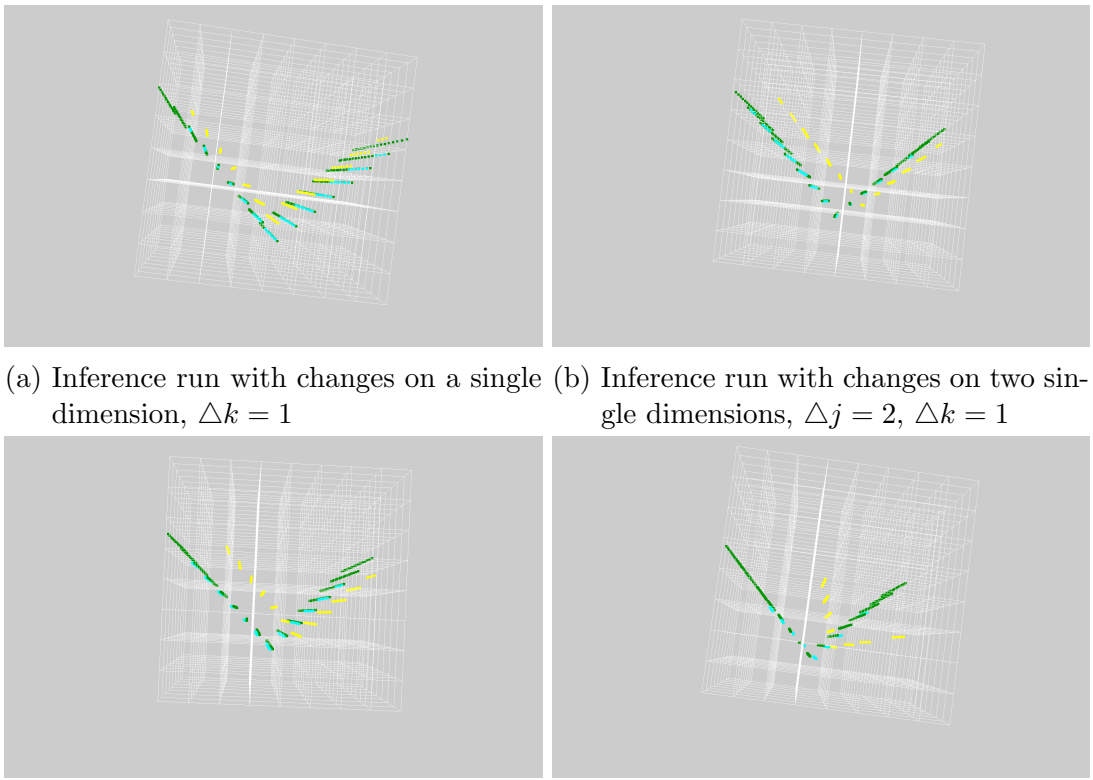


(a) Average intersection over union values on buildings from a subsequence of RobotCar 2014-07-14-14-49-50 dataset [MPLN17].



(b) The number of buildings participated in computation on a subsequence of RobotCar 2014-07-14-14-49-50 dataset

Figure 4.10: Facade separation estimation vs prior methods



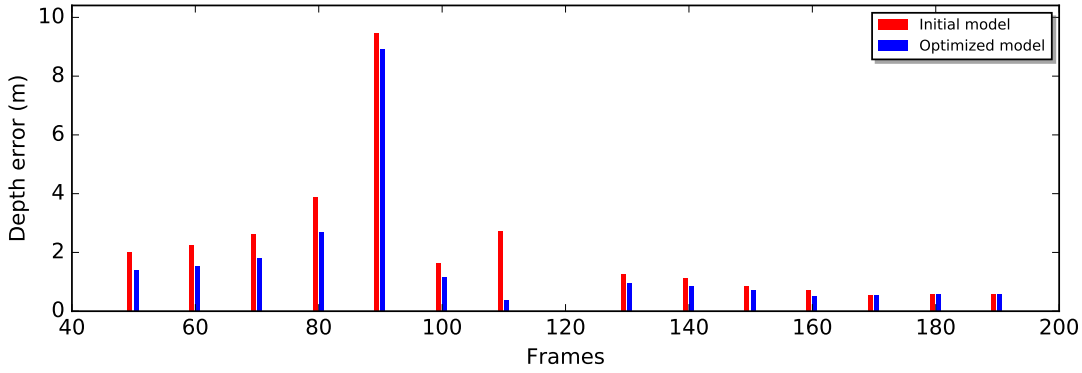
(a) Inference run with changes on a single dimension, $\triangle k = 1$

(b) Inference run with changes on two single dimensions, $\triangle j = 2$, $\triangle k = 1$

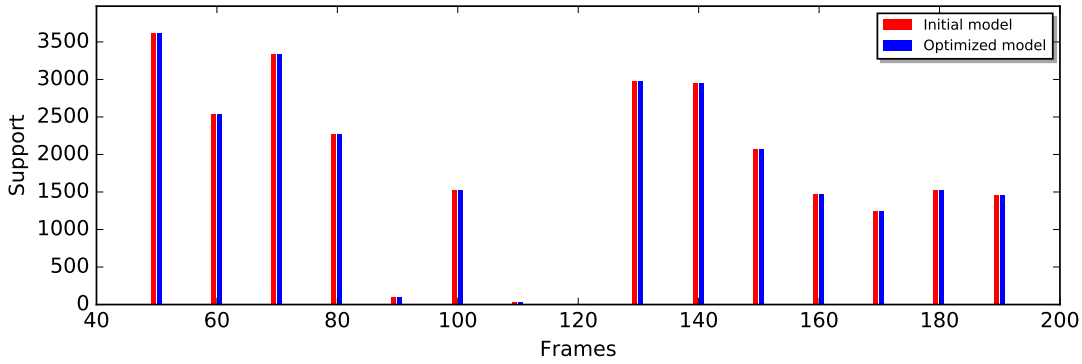
(c) Inference run with changes on all three dimensions, $\triangle j = 2$, $\triangle k = 1$, $\triangle \theta = 0.05$

(d) Inference run with changes on all three dimensions, $\triangle j = 2$, $\triangle k = 1$, $\triangle \theta = 0.5$

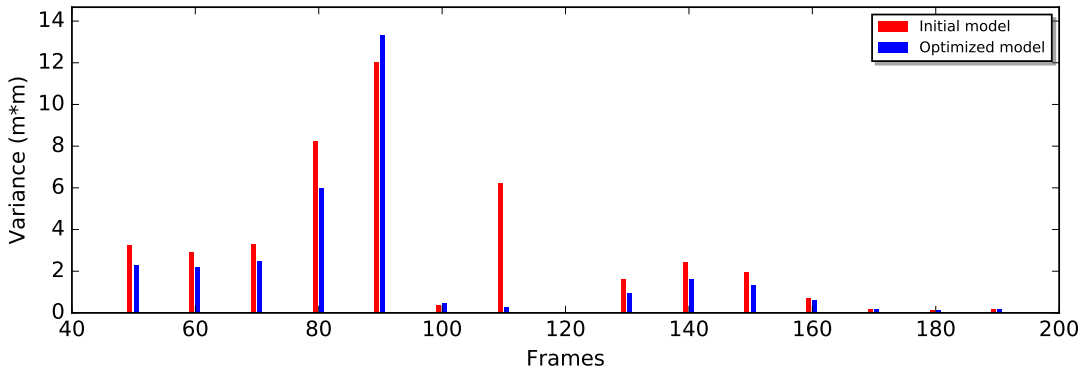
Figure 4.11: Results of running inference on a toy example, optimizing different amount of dimensions. Yellow points show initialization, green - ground truth surface, cyan - the result. The changes along dimension are given under figures



(a) Depth error on a raw sequence of KITTI dataset.

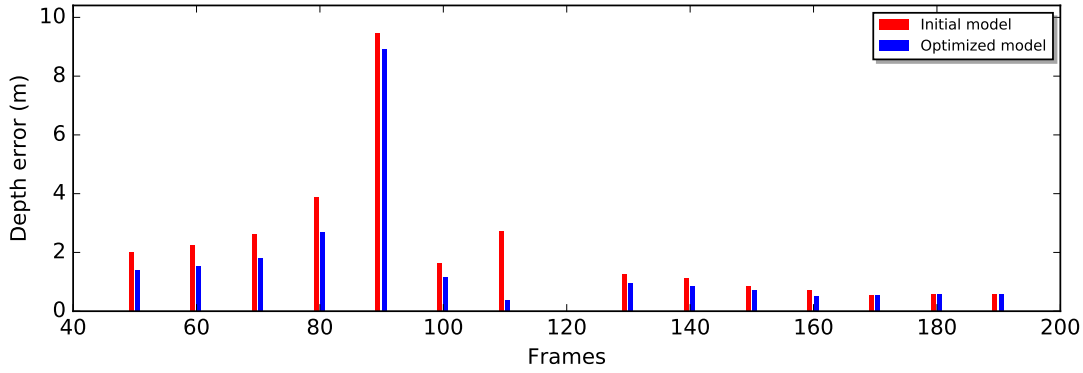


(b) Support in the number of points used for measurement of depth error

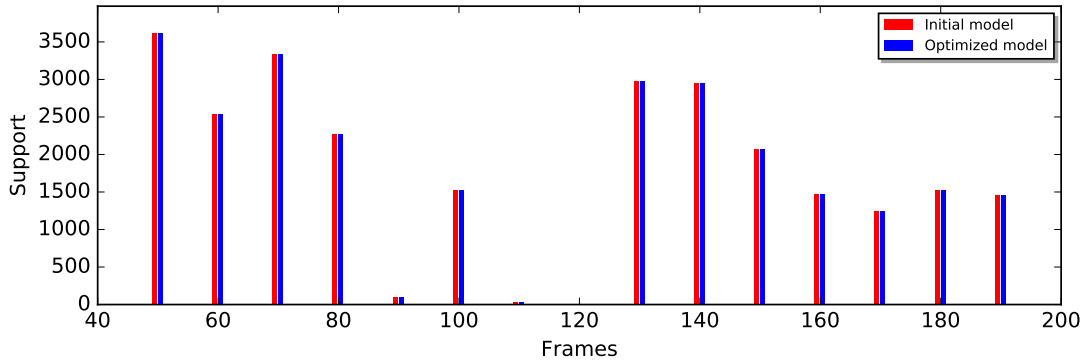


(c) Variance of the depth error

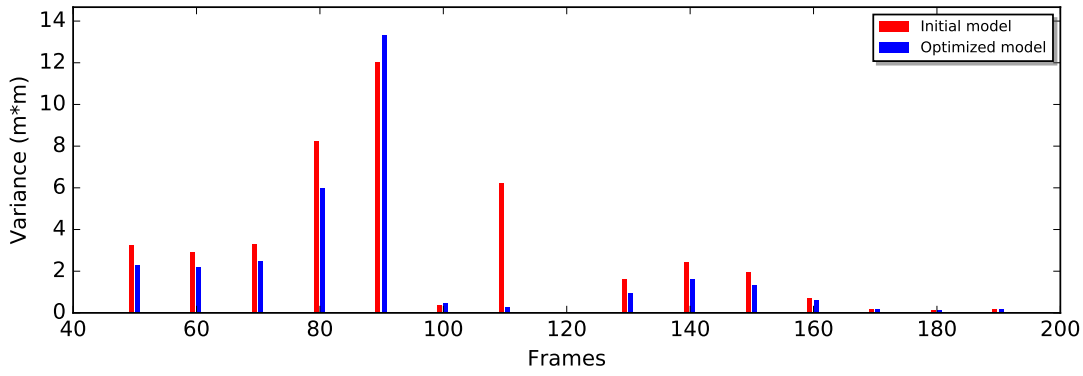
Figure 4.12: Results of running inference on a sequence 2011_09_26_drive_106_sync of raw KITTI dataset



(a) Depth error on a subsequence of RobotCar 2014-07-14-14-49-50 dataset.



(b) Support in the number of points used for measurement of depth error



(c) Variance of the depth error

Figure 4.13: Results of running inference on a subsequence of RobotCar 2014-07-14-14-49-50 dataset

4.4 Discussion

As was pointed before, our method has three major parts: depth integration, pose optimization and facade separation. Each of them includes many parameters and algorithms, that were established empirically, but could be selected differently. The depth integration part is very similar to [UB15], but we were changing the integration distance h, as well as the depth constant d_c. For scenes with smaller and finer objects h was set to 1 or 2, while the depth constant d_c varied from 0.01 to 0.1. In fact we can use smaller value for h, but that brings up an other problem: the input. We work with stereo disparities, which usually yield decent results, but still have error margin, unacceptable for real world application, like autonomous driving. The errors in disparity lead to errors in normal estimation, which affects the construction of the octree, especially, when errors are systematic. The problem of Elas was a high number of outliers in each frame for pixels, that lie on borders of the objects. SPS-Stereo handles the borders rather well, but the planar assumption creates artifacts in disparity sometimes. Laser fusion from [HK+11] returns decent results for pixels, close to laser detections, but is helpless on depth discontinuities. Also because of basing on laser input the disparity is limited to the maximal laser elevation or the lower part of the image. This problem makes disparities useless for our algorithm, if the building was occluded, since the upper half of the image is not present and we lack building points. Hence given better disparity maps we would be able construct better octrees, thus obtaining an improved alignment of the model to the observations. The second problem was the lack of ground truth, especially for the pose estimation evaluation. Not having a correct map made us search for alternative ways to measure the pose correctness. The backprojection of the optimized model into 2D image space only evaluates the pose indirectly. We were though able to create ground truth for facade separation through manual labeling of the buildings in different frames. Another difficulty was a lack of appropriate sequences, that contain buildings. In whole KITTI dataset we only could find three sequences, that fit our purpose.

The last considerable problem was the energy formulation for the inference process. We could not express our energy formulation so that it would both, fulfill our constraints and move the buildings. For the raw sequence we could just disable two dimensions, depth and rotation, and only move the buildings along the width dimension, but it is not possible for general case, since the scenes are not always Z-axis oriented. There are two solutions we found to address that problem. First is to change the ways of implementation, via ceres or some other tool, to be able to use constraints properly. The second was to exploit facade information, which is extracted for facade separation step, to compute its normal and then encourage movements along that normals, while penalizing deviations. This way the shift parameters can be bounded which allows a better use of optimization tools.

4.5 Conclusion

We have shown several algorithms, with main focus on building's pose optimization and facade separation from street-view images. The algorithms work for standard cases, where building's shape is not unique and it is visually separable from the others. We first extract disparity maps and then integrate them into octree. Then we use non-linear optimization on the points, sampled from the model, to compute building's transformation. Finally edge detection and integration are applied, for voting on the building separation. The experiments show, that improvement exists in most cases, although sometimes errors are introduced, due to irregularities of the data.
In future work we plan to enhance the method in many directions. From software point of view, we want to increase the speed of our implementation, while reducing the memory cost. As for algorithmic side, the method should be more independent from the user, regarding various parameters of depth integration, inference and facade separation. An interesting course would be to omit KITTI's position data and perform self-localization, knowing only approximate GPS coordinates of the car. Also, the optimized buildings model can be used for reconstruction of a big part of the scene, so we plan to work on reconstruction of the smaller objects, like cars or poles, which will brings us closer to quality digital recreation of the real scenes.

Bibliography

[AMO] Sameer Agarwal, Keir Mierle, and Others. Ceres solver. http://ceres-solver.org.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[CKZ+15] Xiaozhi Chen, Kaustav Kundu, Yukun Zhu, Andrew G Berneshawi, Huimin Ma, Sanja Fidler, and Raquel Urtasun. 3d object proposals for accurate object class detection. In *Advances in Neural Information Processing Systems*, pages 424–432, 2015.

[CWUF16] Hang Chu, Shenlong Wang, Raquel Urtasun, and Sanja Fidler. Housecraft: Building houses from rental ads and street views. In *European Conference on Computer Vision*, pages 500–516. Springer, 2016.

[Dol] Piotr Dollár. Piotr's Computer Vision Matlab Toolbox (PMT). https://github.com/pdollar/toolbox.

[DZ13] Piotr Dollár and C. Lawrence Zitnick. Structured forests for fast edge detection. In *ICCV*, 2013.

[DZ14] Piotr Dollár and C. Lawrence Zitnick. Fast edge detection using structured forests. *ArXiv*, 2014.

[ER] Peter H Dana Eugene Reimer, Chuck Gantz. Conversion tool: Latlong to utm and utm to latlong. http://ereimer.net/programs/LatLong-UTM.cpp.

[FB81] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[Gar82] Irene Gargantini. Linear octtrees for fast processing of three-dimensional objects. *Computer graphics and Image processing*, 20(4):365–374, 1982.

[GLSU13] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[GLU12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[GRU10] Andreas Geiger, Martin Roser, and Raquel Urtasun. Efficient large-scale stereo matching. In *Asian conference on computer vision*, pages 25–38. Springer, 2010.

[HK+11] Daniel Huber, Takeo Kanade, et al. Integrating lidar into stereo for fast and improved disparity computation. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2011 International Conference on*, pages 405–412. IEEE, 2011.

[HRD+12] Stefan Holzer, Radu Bogdan Rusu, M Dixon, Suat Gedikli, and Nassir Navab. Adaptive neighborhood selection for real-time surface normal estimation from organized point cloud data using integral images. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2684–2689. IEEE, 2012.

[Hub77] Peter J Huber. Robust methods of estimation of regression coefficients 1. *Statistics: A Journal of Theoretical and Applied Statistics*, 8(1):41–53, 1977.

[Kol11] Vladlen Koltun. Efficient inference in fully connected crfs with gaussian edge potentials. *Adv. Neural Inf. Process. Syst*, 2(3):4, 2011.

[Mat13] Markus Mathias. Object detection for urban modeling (object-detectie voor stadsmodellering). 2013.

[MMT+16] J. Robert Menzel, Sven Middelberg, Philip Trettner, Bastian Jonas, and Leif Kobbelt. City Reconstruction and Visualization from Public Data Sources. In Vincent Tourre and Filip Biljecki, editors, *Eurographics Workshop on Urban Data Modelling and Visualisation*. The Eurographics Association, 2016.

[MPLN17] Will Maddern, Geoff Pascoe, Chris Linegar, and Paul Newman. 1 Year, 1000km: The Oxford RobotCar Dataset. *The International Journal of Robotics Research (IJRR)*, 36(1):3–15, 2017.

[OHE+16] Aljoša Ošep, Alexander Hermans, Francis Engelmann, Dirk Klostermann, , Markus Mathias, and Bastian Leibe. Multi-scale object candidates for generic object tracking in street scenes. In *ICRA*, 2016.

[osma] Openstreetmap. http://www.openstreetmap.org/copyright/.

[osmb] Osm2world: Create 3d models from openstreetmap. http://osm2world.org/.

[PASW13] Jeremie Papon, Alexey Abramov, Markus Schoeler, and Florentin Worgotter. Voxel cloud connectivity segmentation-supervoxels for point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027–2034, 2013.

[RC11] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May 9-13 2011.

[RTG98] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. A metric for distributions with applications to image databases. In *Computer Vision, 1998. Sixth International Conference on*, pages 59–66. IEEE, 1998.

[RWL11] Michal Recky, Andreas Wendel, and Franz Leberl. Façade segmentation in a multi-view scenario. In *3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT), 2011 International Conference on*, pages 358–365. IEEE, 2011.

[UB15] B. Ummenhofer and T. Brox. Global, dense multiscale reconstruction for a billion points. In *IEEE International Conference on Computer Vision (ICCV)*, Dec 2015.

[YMU14] Koichiro Yamaguchi, David McAllester, and Raquel Urtasun. Efficient joint segmentation, occlusion labeling, stereo and flow estimation. In *European Conference on Computer Vision*, pages 756–771. Springer, 2014.

[ZGWG15] Chen Zhou, Fatma Güney, Yizhou Wang, and Andreas Geiger. Exploiting object similarity in 3d reconstruction. In *International Conference on Computer Vision (ICCV)*, December 2015.