University of Amsterdam
Faculty of Science

Master's thesis Artificial Intelligence,
track Intelligent Systems

3D Models Enhancement Using GIS Data

Scientific Advisors
Isaac Esteban
Frans Groen
Judith Dijk

Author
Costin Ionita

2011

Abstract

Given a sequence of images depicting the same scene, but from different viewpoints, we can compute the 3D depth of that scene using the displacements of corresponding pixels from one image to another. The result is a sparse 3D pointcloud, which often lacks important details due to missing data, occlusions or noise. In this thesis, we focus exclusively on building 3D models of large urban areas, and we propose several techniques that can be used to complete the sparse 3D pointclouds and obtain a more appealing and realistic look. The images used for reconstruction are taken from the ground level, so they contain little or no information about the ground surface or the buildings' roofs. Therefore, our first aim is to produce a more accurate representation of these two elements, using the position and orientation of the vertical walls present in the scene. Furthermore, the images provided are sometimes not enough to fully reconstruct a certain area and, as a consequence, many buildings have missing walls. Our second aim is to alleviate this problem by integrating the existing 3D pointclouds with ground maps extracted from Geographic Information Systems (GIS). The complete GIS integration is fast and it can scale up to very large urban areas. In addition, experimental results have shown that our ground estimation technique is highly effective both for flat and uneven surfaces.

Contents

Chapter 1

Introduction

Building 3D models of large urban areas is currently a hot topic of research both at industrial and at scientific level. Recent applications such as Google Earth [11] and Microsoft Virtual Earth [12] are already very popular due to their ability to deliver multi-layered visualizations based on large scale 3D models [26]. These systems can be used to support people in taking optimal decisions in a wide range of activities such as urban planning, urban change monitoring, emergency response services and environmental protection services.

The industry approach to 3D reconstruction is based on aerial and satellite imagery annotated with measurements from Global Positioning Systems (GPS). Although this is an important step forward, its main drawback is that it provides little additional information compared to 2D maps. In order to obtain more realistic 3D environments, it is important to capture data from the ground. Google and Microsoft have started recording large amounts of data at street level, which is now already available for multiple cities in the form of large panoramic mosaics [28]. This however limits the user's ability to freely navigate the environment, so there is growing demand for better solutions.

Given a sequence of images depicting the same scene, but from different viewpoints, we can compute the 3D depth of that scene using the displacements of corresponding pixels from one image to another. This is known as motion estimation, and it typically produces a sparse 3D pointcloud. In this thesis, we propose several techniques that can be used to enhance the sparse 3D models and obtain a more realistic look. We do not draw too much attention on the actual estimation of the 3D structure, and instead we concentrate on the post-processing of the resulting pointcloud. We assume that the input images are taken from the ground level and that the pointcloud has already been generated using one of the stereo vision techniques described in the computer vision literature [13].

There are several design principles that need to be considered when building large scale 3D models. First of all, the algorithms have to be reasonably fast, so that they can deal with the sheer size of the data sets. Many modern algorithms try to produce highly detailed 3D reconstructions, which can lead to computational bottlenecks. Keeping in mind the potential applications of the 3D models, we realize that a high level of detail is not always necessary, so we can trade-off some representation quality for an increased processing speed.

Another important observation is that we should keep to a minimum the number of assumptions regarding the scene structure. The images used as input are not necessarily taken in a confined setting (i.e. fixed camera, constant orientation etc), so variations in camera pose are frequent. However, some prior knowledge about the orientation of the camera is actually required, otherwise we cannot say anything about the orientation of the reconstructed scene.

For this reason, it is reasonable to assume that the y axis of each image is roughly parallel to the gravity direction, which corresponds to the natural position in which pictures are usually taken. Without this prior, it is not possible to tell apart the ground surface from other surfaces contained in the pointcloud, due to its intrinsic ambiguity.

Images taken from the ground level are usually focused on one or more target buildings. Unlike aerial images, they provide very little information about the ground surface or the buildings' roofs, since these are barely visible in the images. Therefore, our first objective is to design a method that can provide us with a more accurate representation of the ground surface, starting from the minimal information provided by the pointcloud. Next, we can use this to estimate the height of each building and consequently, the relative position of each roof. We approximate roofs with planar structures lying at a certain height above the ground surface.

Another major challenge is that there is no guarantee whether each building is fully reconstructed. Some walls might not be captured in any image, while others are subject to occlusions, so in general, some of the buildings might have missing walls. Thus, our second objective is to address this problem by integrating information extracted from 2D ground plans. As discussed in [17], ground plans of buildings have already been acquired and are represented either in analog form, by maps and plans, or digitally, in 2D Geographic Information Systems (GIS). Their advantage is that they contain aggregated information, which has been made explicit by human interpretation.

Next, we review some of the existing work and we try to put it in context. Then, at the end of this chapter, we provide a thesis outline, where we describe the general structure of the thesis and we highlight our main contributions.

1.1 Related work

In its early days, 3D urban reconstruction was performed using airborne laser scanning [17][32]. Aerial LiDAR (Light Detection and Ranging) offers short data acquisition and processing times, it is highly accurate and it generates very dense 3D pointclouds. With modern systems, an hour of data collection can result in over 10 million points with an accuracy in the range of tens of centimeters to one meter [22]. Moreover, city scale acquisitions can be performed in only a couple of hours, making laser scanning a popular technology for acquiring terrain elevation data.

Instead of actively measuring the surface height using laser scanners, one can use aerial images to estimate the height using photometric stereo techniques [3][2][6][16]. Nonetheless, the resulting height maps are usually sparser and less accurate than the LiDAR generated pointclouds. While improvements are still possible in both cases, there are some inherent limitations when using aerial data. In most of the cases, the vertical walls are poorly reconstructed and they lack texture. For this reasons, recent approaches have shifted from an aerial-based to a ground-based reconstruction.

Frueh et al [15] have developed a system that is capable of recording data using both active and passive sensors. A laser scanner and a digital camera are mounted on a vehicle that is used to drive around public roads. This enables them to fuse the 2D surface scans with the texture information contained in the images. While the results obtained with such a system are impressive, laser-equipped vehicles are not yet a very common technology, as they are still very expensive.

On the other hand, surveying vehicles capable of recording video-streams are relatively cheap, and therefore a large amount of data is already available. Video data is often annotated

with GPS and INS[1] data, which allows pose estimation and correction of different measurement errors. Video-based reconstruction of urban environments is described in [23][9]. High accuracy and faster than real-time performance can be obtained by implementing this system on a GPU.

Moving on to the more specific problem of modeling the ground surface, many of the approaches so far are rather naive and they make multiple simplifying assumptions. For instance, in [27], Spinello et al assume that a 3D point belongs to the ground plane if its corresponding normal deviates only slightly (less than 25°) from the upright vector $[0, 0, 1]^T$, and it is not further away from its closest neighbor than a given threshold (see appendix A.1 for the vectorial notation). In [31], von Hansen assumes that the ground surface is large compared to the other surfaces, and that the ground plane is approximately parallel to the horizontal plane. Using a similar set of assumptions, Mordohai et al [23] determine the ground plane by sweeping horizontal planes at varying heights. Moreover, the underlying assumption of all these approaches is that the ground surface can be approximated by a plane. However, this is not always the case in real life applications, so the problem with these assumptions is that they are very restrictive and they limit the range of applicability of such methods.

In what concerns the integration of existing 3D models with GIS data, the idea is inspired by the work of Vosselman et al [32] and Haala et al [17], which have successfully used ground maps from GIS data to refine the 3D models obtained from airborne laser scans. The novelty of our approach is that instead of dense and accurate pointclouds, we use sparse and noisy pointclouds generated from ground images. In addition, no geo-location information about the pointcloud is known a priori, so user input is required for the actual GIS registration.

1.2 Thesis outline

In chapter 2, we overview the main steps of 3D Reconstruction from multiple images. Note however that the focus is on two-view geometry, which means that the initial sequence of images is split in pairs of consecutive frames, so that only two images are actually processed at a time. The first part of this chapter is more generic, and it describes two estimation techniques, which will be used in multiple sections of this thesis. Then, the focus shifts on the practical problems that need to be solved in order to produce the final pointcloud: camera estimation, motion estimation, bundle adjustment etc.

Once a pointcloud is obtained, we need to determine the orientation and the position of the ground surface. In chapter 3, we propose an iterative technique for gravity estimation. Based on the gravity direction, we compute the reference plane, which provides us with a rough approximation of the ground surface. Our primary assumption is that the gravity direction is orthogonal to all vertical wall normals present in the scene. Typically, the number of vertical walls found in the scene is quite large, so the resulting system is overconstrained. Therefore, we formulate this as a least squares optimization problem.

Having a planar approximation of the ground surface is often enough for relatively flat landscapes. However, if the landscape is highly uneven, then the errors are too large. Alternatively, we can represent the ground surface as a mesh. In chapter 4, we propose several techniques for initializing and then refining this mesh, such that we obtain a more accurate representation of the ground surface. Note that there is a tradeoff between the quality and the speed of the reconstruction. High quality reconstructions require dense meshes, which are usually more expensive to compute.

Our next step is to integrate our 3D models with ground maps extracted from GIS data. We

[1]Inertial Navigation Systems use accelerometers and gyroscopes to continuously calculate the position, orientation, and velocity of a moving object without the need for external references

import the GIS data from OpenStreetMap [25], which is an open-source project that provides free access to worldwide geographic data. The problem is that this data does not provide a directly accessible representation of buildings, so it cannot be integrated as it is. In addition, there is also a significant amount of noise that can negatively influence the quality of the reconstruction. In chapter 5, we propose an algorithm that can clean up the data and then transform it into a more convenient format.

In chapter 6, we perform the actual GIS registration of the 3D models. This step is not 100% automated due to the inherent reconstruction ambiguities. The user is required to select points correspondences using a simple Graphical User Interface (GUI), so that the necessary 2D transformations can be computed. Finally, on top of the new 3D models, we overlay a Google map, which contributes to a more visually appealing result.

We have decided to include a separate evaluation section for each of chapters 3, 4, 5, 6. This way, the reader can have a better understanding about the individual role of each of our contributions. Finally, in chapter 7, we draw the conclusions and we outline some interesting directions for future research.

Chapter 2

Preliminaries on 3D Reconstruction

In this chapter, we review the main stages of 3D reconstruction from a sequence of images. Let $(\mathrm{x}_i, \mathrm{x}'_i)$ be a set of point correspondences, and let P_i and P'_i be two camera matrices that project a set of 3D points X_i into x_i and x'_i, respectively. We can write this as $PX_i = \mathrm{x}_i$ and $P'X_i = \mathrm{x}'_i$. Based on the input pairs $(\mathrm{x}_i, \mathrm{x}'_i)$, our aim is to determine the camera models P_i and P'_i, and recover the 3D structure of the scene, i.e. compute the 3D points X_i.

Before going into more detail, it is important to identify some of the inherent ambiguities occurring in 3D reconstruction. First of all, regardless of how many images are used for reconstruction, it is impossible to establish the absolute position and orientation of a scene. Without additional information regarding the latitude and longitude of each point, the scene can only be determined up to an Euclidean transformation (see section A.2) with respect to the world frame. In chapter 6, we will see that this is one of the reasons why the scene cannot be automatically registered with additional geographic data.

A more subtle issue is that in the absence of any prior knowledge, it is impossible to determine the exact scale of the reconstructed scene. As shown in Fig. 2.1, the images themselves do not provide any information about the absolute size of an object. Our ability to tell apart a toy car from a real car is based on our previous experience, and not on image measurements. Without this experience, the scene can only be determined up to a similarity transformation (see section A.2). Consider H_s a similarity transformation. If we replace X_i by $H_s X_i$ and the camera by PH_s^{-1}, then each image projection remains unchanged, since $x_i = PX_i = (PH_s^{-1})(H_s X_i)$.

Finally, if nothing is known about the calibration of the two cameras, then the scene can only be reconstructed up to an arbitrary projective transformation (see section A.2). The proof follows directly from the previous one. Instead of H_s, we now have an arbitrary projective transformation H, such that $x_i = PX_i = (PH^{-1})(HX_i)$ (Fig. 2.2). In general, a projective reconstruction is not very useful from an applications perspective. The real objects can get distorted under a projective transformation, and they might appear strange to human eye. In order to eliminate this ambiguity, we need to determine the cameras calibration, so that we obtain an isometric (Euclidean) reconstruction.

The input for most algorithms described in this chapter is a set of point correspondences between two images. For this reason, we assume that a set of matchings between salient point in the two images have already been identified using one of the standard techniques in computer vision [20][5]. This corresponds to the first two steps of the 3D reconstruction pipeline summarised in Fig. 2.3.

Figure 2.1: The similarity ambiguity: toy car vs real-world car

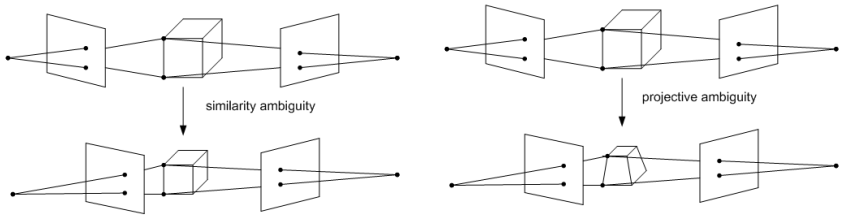


Figure 2.2: On the left, the similarity ambiguity. The two objects differ in scale, but under the appropriate camera transformation the image projections remain the same. On the right, the projective transformation. Again, under the appropriate camera transformation, the image projections remain the same.

Given these feature matches, we can then use the 8-point algorithm described by Zisserman [18] to compute the frame-to-frame motion. The motion estimation stage produces a fundamental matrix F that can be used to compute the camera matrices. In addition, the fundamental matrix F establishes an important relation between x_i and x'_i, known as the epipolar constraint: $\mathrm{x}_i'^T F \mathrm{x}_i = 0$.

If two points satisfy the epipolar constraint, it means that the rays back-projected through each image point intersect in a 3D point X_i. This process of estimating the 3D depth of the scene from image measurements is known as linear triangulation. Finally, given a sequence of images, our aim is to find the camera matrices P_i and the 3D points X_j such that $P^i X_j = x^i_j$. This step typically involves an iterative optimization procedure known as bundle adjustment.

Before we analyze in more detail each of these steps, we abstractly describe two important estimation techniques: the Direct Linear Transformation (DLT) and the robust estimation. Later on, we will see how these algorithms can be adapted for solving a wide range of more specific problems. An implementation of the 3D reconstruction pipeline is available in the FIT3D toolbox [30].

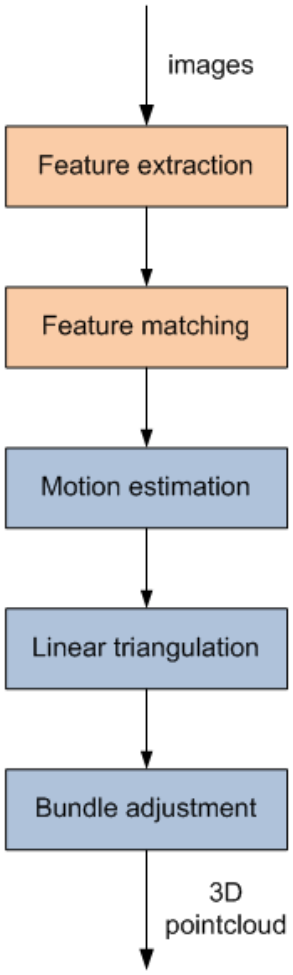


Figure 2.3: The 3D reconstruction pipeline

2.1 Estimation techniques

2.1.1 The Direct Linear Transformation algorithm

The Direct Linear Transformation (DLT) algorithm is used to compute a transformation H such that $\mathrm{x}'_i = H\mathrm{x}_i$, where $(\mathrm{x}_i, \mathrm{x}'_i)$ are pairs of point correspondences. Note that x_i and x'_i are homogeneous vectors (see section A.1), so x'_i and $H\mathrm{x}_i$ must have the same direction, but their magnitude might differ by a non-zero scaling factor. Therefore, we rewrite the equation using the vector cross product $\mathrm{x}'_i \times H\mathrm{x}_i = 0$.

Consider

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix} = \begin{pmatrix} h^{1T} \\ h^{2T} \\ h^{3T} \end{pmatrix} \tag{2.1}$$

$H\mathrm{x}_i$ can now be rewritten as:

$$H\mathrm{x}_i = \begin{pmatrix} h^{1T}\mathrm{x}_i \\ h^{2T}\mathrm{x}_i \\ h^{3T}\mathrm{x}_i \end{pmatrix} \tag{2.2}$$

Let $\mathrm{x}'_i = (x'_i, y'_i, w'_i)$. Then, we can expand the cross product as:

$$\mathrm{x}'_i \times H\mathrm{x}_i = \begin{pmatrix} y'_i h^{3T}\mathrm{x}_i - w'_i h^{2T}\mathrm{x}_i \\ w'_i h^{1T}\mathrm{x}_i - x'_i h^{3T}\mathrm{x}_i \\ x'_i h^{2T}\mathrm{x}_i - y'_i h^{1T}\mathrm{x}_i \end{pmatrix} \tag{2.3}$$

According to the duality principle, we have $h^{jT}\mathrm{x}_i = \mathrm{x}_i^T h^j$, for all j. We can now rewrite the equations as:

$$\begin{pmatrix} 0^T & -w'_i\mathrm{x}_i^T & y'_i\mathrm{x}_i^T \\ w'_i\mathrm{x}_i^T & 0^T & -x'_i\mathrm{x}_i^T \\ -y'_i\mathrm{x}_i^T & x'_i\mathrm{x}_i^T & 0^T \end{pmatrix} \begin{pmatrix} h^1 \\ h^2 \\ h^3 \end{pmatrix} = 0 \tag{2.4}$$

We have thus obtained a homogeneous system of the form $A_i h = 0$, where A_i is a 3×9 matrix, and h is a 9-vector, which contains the elements of H in row-wise order. Note however that the third equation is a linear combination of the first two, so we can omit it. In the end, a set of two independent equations is obtained for each point correspondence.

In order to find h, at least four pairs of points are required. If exactly four pairs are provided, then we can build a homogeneous system $Ah = 0$, where A is an 8×9 matrix. Matrix A has rank 8, so h can only be determined up to scale (we ignore trivial solution $h = 0$). There is usually no preference over the exact scale, but in practice we often choose $||h|| = 1$.

Over-determined systems

If more than four point correspondences are provided, then the resulting system $Ah = 0$ is over-determined. If the measurements of x_i and x'_i are precise, then the system will have an exact non-zero solution. However, this is rarely the case in real applications. Usually the measurements are perturbed by noise, so the system will not have an exact solution. In this case, it is often better to search for an approximate solution $\tilde{h}$ that minimizes $A\tilde{h} = 0$. In order to avoid the trivial solution, we need to constrain $\tilde{h}$ such that $||\tilde{h}|| = 1$. In section B.2, we show that this is equivalent with finding the unit singular vector corresponding to the smallest singular value of A. The steps taken to find h (and consequently H) form the Direct Linear Transformation (DLT) algorithm and is summarized below:

Listing 2.1: The DLT algorithm

```
for each correspondence (x_i, x'_i)
        compute matrix A_i using equation 2.4
assemble the n matrices A_i into a single 2n × 9 matrix A
obtain the SVD of A, i.e. A = UDV^T
h is given by the last column of V
restore matrix H from h
```

Inhomogeneous solution

Another approach for solving h is to transform the original set of equations (2.4) into an inhomogeneous system by imposing $h_9 = 1$. As we discussed in the previous section, the solution is determined only up to scale, so we can choose this scale such that $h_9 = 1$. The two independent equations from 2.4 can now be written as:

$$\begin{pmatrix} 0 & 0 & 0 & -x_i w'_i & -y_i w'_i & -w_i w'_i & x_i y'_i & y_i y'_i \\ x_i w'_i & y_i w'_i & w_i w'_i & 0 & 0 & 0 & -x_i y'_i & -y_i y'_i \end{pmatrix} \tilde{h} = \begin{pmatrix} -w_i y'_i \\ -w_i x'_i \end{pmatrix} \tag{2.5}$$

where $\tilde{h}$ is an 8-vector consisting of the first 8 elements of h.

Adding together the equations for all pairs of points we obtain an equivalent inhomogeneous system of the form $M\tilde{h} = b$. If only four pairs are provided, the system has an exact solution that can be found using Gaussian elimination. If more points are given, then the system is over-determined and an approximate solution can be computed using least-squares techniques (see section B.1).

Normalization

We have already seen that if the measurements are noisy then the over-determined system $Ah = 0$ does not have an exact solution. Instead, we search for an approximate solution $\tilde{h}$, which minimizes $||A\tilde{h}||$ subject to $||\tilde{h}|| = 1$. This is equivalent with searching for the rank 8 matrix $\tilde{A}$ closest to A, and then determine the exact h from $\tilde{A}h = 0$.

Result 1 *Let $A = UDV^T$ be the Singular Value Decomposition of A. The rank of A equals the number of non-zero singular values, i.e. the number of non-zero diagonal elements in D.*

We can now formulate the following minimization problem using the Frobenius norm [1]:

$$||A - \tilde{A}||_F = ||UDV^T - U\tilde{D}V^T||_F = ||D - \tilde{D}||_F \tag{2.6}$$

This means that $\tilde{A}$ can be found by setting the smallest singular value of $\tilde{D}$ to 0. $\tilde{D}$ will now have only 8 non-zero diagonal elements, so $rank(A) = 8$. In addition, the influence of the smallest singular value has the least effect, so by turning it to zero, we obtain the closest approximation of A.

Compared to A, some values of the resulting $\tilde{A}$ are either increased or decreased, such that overall, the expression $||A-\tilde{A}||_F$ is minimal. This means that the result of the minimization, and consequently the result of equation $\tilde{A}h = 0$, depends on the particular choice of the coordinate frame.

Consider a pair of points (x, y, w) and (x', y', w'), such that:

$$x >> y >> w$$
$$x' >> y' >> w'$$

Then, some entries in matrix A will be much larger than others, for instance $xx' >> ww'$. Therefore, increasing or decreasing xx' and ww' by a constant k will not have the same effect

[1] the Frobenius norm is defined as $||A||_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij}}$

on the two terms, due to their different magnitudes. Although $\tilde{A}$ is the closest approximation in terms of Frobenius norm, this does not mean that the solution of the original system $Ah = 0$ is preserved during this transformation.

In order to make the DLT algorithm independent of the coordinate frame, we must first normalize our data as follows:

- translate all points so that their centroid is at the origin.
- scale all points so that the average distance from the origin is equal to $\sqrt{2}$ (i.e. the average point is $(1, 1)^T$)
- apply the transformations independently on the two set of points

2.1.2 Robust estimation

In the previous section we have seen that the measurement of a point's position can be perturbed by noise, and as a consequence we often search for the 'best' solution and not for an exact solution. However, this is not the only source of error that can affect the estimation algorithm. The input point correspondences $(\mathrm{x}_i, \mathrm{x}'_i)$ are found by matching salient features from the two images, based on a similarity measure (e.g. Euclidean distance between descriptors), and therefore, some of the input pairs might be mismatched (outliers). In the presence of outliers, a least squares solution does not provide reliable results, so a better estimation technique is required.

Robust estimation attempts to identify a set of correct point correspondences (known as inliers), and use them to compute a transformation, or more generally, to fit a model. The general idea of the algorithm is to find a set of minimal points in the data, sufficient to determine the model. Then, the support for this model is measured using a pre-defined fitness function. In the end, the model with the maximum support is returned as a solution. This procedure is known as the RANdom SAmple Consensus (RANSAC [14]).

Coming back to our initial problem, if the model is a planar homography, and the data is a set of 2D point correspondences, then the minimal subset consists of four correspondences. RANSAC attempts to find those four point correspondences for which there is maximum support across the data. Typically, the fitness function is given by the number of inliers for each model.

Listing 2.2: The RANSAC algorithm

```
repeat
        draw a sample of s points uniformly and at random
        fit model using this subset
        for each data point outside sample
        compute distance to model
                if distance > t
                        reject as outlier
                else
                        count as inlier
                if best fit so far
                        save current model
until k iterations
```

The number of iterations required

A natural question that arises when we look at the RANSAC algorithm is how many iterations are needed to find a good solution. According to [14], the process is terminated when the likelihood of finding a better model becomes low, i.e. the probability of missing a set of inliers of size I within k samples falls under a predefined threshold:

$$\eta = (1 - P_I)^k \tag{2.7}$$

P_I is the probability that an uncontaminated sample of size m is randomly selected from all N data points and it can be computed as follows:

$$P_I = \frac{\binom{I}{m}}{\binom{N}{m}} = \frac{I!}{(I-m)!} \cdot \frac{(N-m)!}{N!} = \frac{I \cdot (I-1) \cdot ... \cdot (I-m+1)}{N \cdot (N-1) \cdot ... \cdot (N-m+1)} \tag{2.8}$$

$$\text{since } m << I < N \text{ then} \tag{2.9}$$

$$P_I = (\frac{I}{N})^m = w^m \tag{2.10}$$

We can thus approximate P_I by the fraction of inliers from the total number of data points. Finally, the number of iterations required can be expressed as:

$$k = \frac{log(\eta)}{log(1 - w^m)} \tag{2.11}$$

The problem is that in many applications we have to deal with data where w is unknown. However, each fitting attempt contains information about w. By observing a long sequence of fitting attempts, we can estimate w from this sequence. This suggests that we start with a relatively low value of w, and then we iteratively increase it as we improve our model. When the number of fitting attempts becomes higher than the current k, the process stops.

2.1.3 Error analysis

In this section, we assume that the outliers have already been removed, so we deal exclusively with measurement errors. Typically, we assert that the measurement errors follow a zero-mean isotropic Gaussian distribution:

$$p(x) = \frac{1}{2\pi\sigma^2} e^{\frac{-d(x,\bar{x})^2}{2\sigma^2}} \tag{2.12}$$

This means that the measured image point x is obtained as:

$$\mathrm{x} = \bar{\mathrm{x}} + \Delta\mathrm{x} \tag{2.13}$$

where $\bar{\mathrm{x}}$ is the true point and $\Delta\mathrm{x}$ is a zero-mean Gaussian distribution with variance σ^2. Taking the log of 2.12, we obtain:

$$\begin{aligned}\ln p(x) &= -\frac{1}{2\sigma^2} d(x,\bar{x})^2 - \ln(2\pi\sigma^2) \\ &= -\frac{1}{2\sigma^2} d(x,\bar{x})^2 + constant\end{aligned}$$

which shows that the Maximum Likelihood Estimate(MLE) is equivalent with minimizing the geometric error $d(\mathrm{x},\bar{\mathrm{x}})^2$.

We now return to the problem of estimating a 2D homography and we consider the case when measurements from both images have been perturbed with Gaussian noise. Let $(\bar{\mathrm{x}}_i, \bar{\mathrm{x}}'_i)$ be the noise-free point correspondences between the two images, and let $(\mathrm{x}_i, \mathrm{x}'_i)$ be their noisy measurements. In general, an estimated homography $\hat{H}$ will not map x_i to x'_i, nor $\bar{\mathrm{x}}_i$ to $\bar{\mathrm{x}}'_i$.

We define the RMS[1] residual error as:

$$\epsilon_{res} = \frac{1}{\sqrt{4n}} \left(\sum_{i=1}^{n} d(\mathrm{x}_i, \hat{\mathrm{x}}_i)^2 + d(\mathrm{x}_i, \hat{\mathrm{x}}_i)^2 \right)^{\frac{1}{2}} \tag{2.14}$$

which measures the difference between the estimated points ($\hat{\mathrm{x}}'_i = H\hat{\mathrm{x}}_i$) and the noisy input points. Note however that the RMS error is not an absolute measure of quality. At the limit, if only four point correspondences are provided the residual error is zero, since the model fits the data perfectly. Below, we give an important result, as described in [18]:

Result 2 *Consider an estimation problem where N measurements are to be modeled by a function depending on a set of d essential parameters. Suppose the measurements are subject to independent Gaussian noise with standard deviation σ in each measurement variable.*

- *The RMS residual error (distance between the measured and estimated value) for the maximum likelihood estimator:*

$$\epsilon_{res} = E[||\hat{X} - X||^2/N]^{\frac{1}{2}} = \sigma\,(1 - d/N)^{\frac{1}{2}} \tag{2.15}$$

- *The RMS estimation error (distance between the estimated and the true value) for the maximum likelihood estimator:*

$$\epsilon_{est} = E[||\hat{X} - \bar{X}||^2/N]^{\frac{1}{2}} = \sigma\,(d/N)^{\frac{1}{2}} \tag{2.16}$$

For the 2D homography estimation, we have $N = 4n$ measurements (namely two coordinates for each point), and $d = 2n+8$ essential parameters (namely $2n$ coordinates for $\hat{\mathrm{x}}_i$, and 8 degrees of freedom for H). This results in the following expected errors:

$$\epsilon_{res} = \sigma \left(\frac{n-4}{2n} \right)^{\frac{1}{2}} \tag{2.17}$$

$$\epsilon_{est} = \sigma \left(\frac{n+4}{2n} \right)^{\frac{1}{2}} \tag{2.18}$$

As the number of matchings increases then the residual error increases, while the estimation error decreases. At the limit, they both converge to $\sigma(0.5)^{\frac{1}{2}}$. The intuition is that the more measurements we use, the more the estimated model should agree with the noise-free true values.

[1] root-mean-squared

2.2 Camera models

In this section, we begin with the basic pinhole camera, and then we progress towards the most general model, represented by the projective camera. The pinhole camera is the simplest camera model and is depicted in Fig. 2.4. In this case we assume that the center of projection (O), also known as the *camera center*, lies at the origin of an Euclidean coordinate system. There are several important entities that we need to define before going into more detail. Let f be the *focal distance* of the camera. Then, the plane with equation $z = f$ forms the *image plane*, also known as the focal plane. A line perpendicular to the image plane, which passes through the center of projection is known as the *principal axis*, while its intersection with the image plane (o') is called the *principal point*.

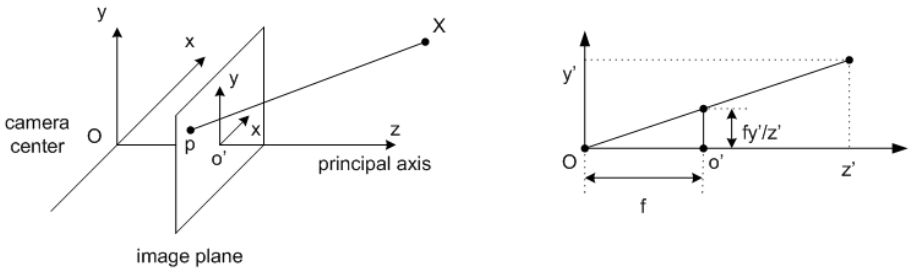


Figure 2.4: The pinhole camera geometry

We can now define the pinhole camera as a function that maps a point in the 3D space to a point in the image plane (central projection mapping). Let $X = (x', y', z')^T$ be a point in the 3D space. Then, its corresponding projection is $(\frac{fx'}{z'}, \frac{fy'}{z'})^T$, or in homogeneous coordinates $(fx', fy', z')^T$. Using homogeneous vectors, we can write the central projection mapping as:

$$\begin{pmatrix} fx' \\ fy' \\ z' \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \tag{2.19}$$

Compactly, this has the form x = PX, where P is the camera projection matrix. So far, we have assumed that the principal point and the origin of coordinates (for the image plane) are one and the same. In practice, however, this might not hold. For this reason, we also need to account for an offset (p_x, p_y) between the principal point and the origin of coordinates. System 2.19 becomes:

$$\begin{pmatrix} fx' + z'p_x \\ fx' + z'p_y \\ z' \end{pmatrix} = \begin{pmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} \tag{2.20}$$

Consider K, a 3×3 matrix, such that:

$$K = \begin{pmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{pmatrix} \tag{2.21}$$

Now we rewrite 2.20 as $\mathrm{x} = K[I|0]X_{cam}$, where K is known as the calibration matrix. We use the notation X_{cam} to show that the coordinates of the 3D point are expressed with respect to the camera coordinate frame.

Note however that the camera coordinate frame and the world coordinate frame might be different. Nonetheless, they are related by an Euclidean transformation, i.e. a rotation and a translation. Let $\tilde{C}$ be the coordinates of the camera with respect to the world coordinates frame, and let R be a rotation transformation such that $\tilde{X}_{cam} = R(\tilde{X} - \tilde{C})$. We use the tilde notation to illustrate that we deal with inhomogeneous vectors. Transforming this to homogeneous coordinates we obtain:

$$X_{cam} = \begin{pmatrix} R & -R\tilde{C} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{2.22}$$

We can now express the projection from the 3D space to the 2D image plane as:

$$\mathrm{x} = KR[I| - \tilde{C}]X \tag{2.23}$$

The three parameters contained in K are known as the internal camera parameters, while the six parameters contained in $R[I| - \tilde{C}]$ (three for R, and three for $\tilde{C}$) form the external camera parameters.

So far we have concentrated on the pinhole camera, which is an ideal model. In practice however, we use CCD cameras which are subject to manufacturing errors, such as non-square pixels. Therefore, the initial assumption of equal scales in all directions might not hold. For this reason, we introduce a more general calibration matrix containing uneven scale parameters, and also an additional skew factor s (2.24). In practice, the skew factor is almost always ignored.

$$K = \begin{pmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.24}$$

2.2.1 Computing the camera matrix

The algorithm used to determine the camera matrix is inspired by the DLT algorithm presented in section 2.1.1. Instead of image point correspondences, we now use correspondences between 3D points and their image projections. Consider a set of pairs (X_i, x_i). Our aim is to find P such that $\mathrm{x}_i = PX_i$, for all i.

Recall from equation 2.4 that each point correspondence gives rise to three equations in the entries of P. However, only two of them are independent:

$$\begin{pmatrix} 0^T & -w_i' X_i^T & y_i' X_i^T \\ w_i' X_i^T & 0^T & -x_i' X_i^T \end{pmatrix} \begin{pmatrix} p^1 \\ p^2 \\ p^3 \end{pmatrix} = 0 \tag{2.25}$$

Given n point correspondences, we obtain an homogeneous system $Ap = 0$, where A is a $2n \times 12$ matrix. A solution is then found using the same techniques described for the DLT algorithm. A summary of the camera estimation algorithm is given below:

Listing 2.3: The camera estimation algorithm

compute a similarity transformation T to normalize image points
compute a similarity transformation U to normalize space points
$\tilde{\mathrm{x}}_i = T\mathrm{x}_i$
$\tilde{X}_i = UX_i$
assemble the system $Ap = 0$
obtain the SVD of A, i.e. $A = UDV^T$
p is given by the last column of V
restore matrix $\tilde{P}$ from p
$P = T^{-1}\tilde{P}U$

2.2.2 Radial distortion

Under a pinhole camera model, the point in the 3D space, the image point, and the optical center are collinear. Real lenses however introduce an error known as the radial distortion. The effect of radial distortion is that straight lines appear as curved, particularly on image margins (Fig. 2.5), and particularly for short focal lengths.

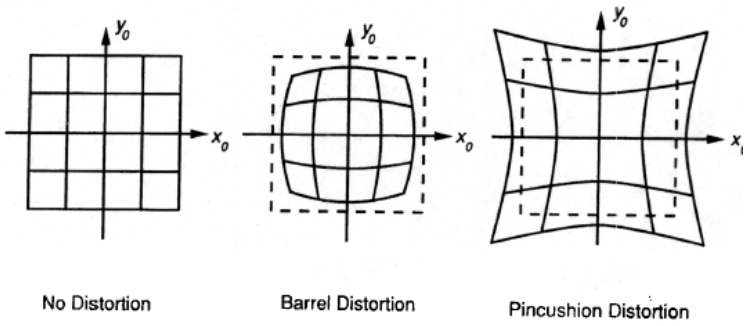


Figure 2.5: Examples of radial distortion

The correction for radial distortion is given by:

$$\hat{x} = x_c + L(r)(x - x_c) \tag{2.26}$$
$$\hat{y} = y_c + L(r)(y - y_c) \tag{2.27}$$

where x and y are the measured coordinates, $\hat{x}$ and $\hat{y}$ are the corrected coordinates, and (x_c, y_c) is the center of radial distortion (typically the image center), with $r = (x-x_c)^2+(y-y_c)^2$. According to [33], the distortion function $L(r)$ can be written as:

$$L(r) = 1 + k_1 r + k_2 r^2 + k_3 r^3 + ...k_n r^n \tag{2.28}$$

Typically, only the first term of this series is considered, so $L(r) = 1 + k_1 r$. In order to find the distortion function, a special calibration object is used, known as the Tsai grid (Fig. 2.6). We know that some of the corners found in the Tsai grid belong to straight lines. Therefore, the radial distortion can be corrected by minimizing the distance between the ideal and the distorted points:

$$\min \sum_i (x_i - x_c)^2 + (y_i - y_c)^2 \tag{2.29}$$

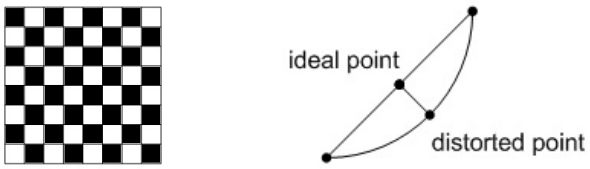


Figure 2.6: On the left, the Tsai grid used for calibration. On the right, the distortion error

The corner points can be obtained using a three-step procedure:

- perform Canny edge detection
- fit lines using the detected edges
- intersect lines to obtain corners

2.3 Motion estimation

In this section, we describe the epipolar geometry and then, based on that, we derive an equation for the fundamental matrix. This matrix establishes a very important relation between pairs of point correspondences, known as the epipolar constraint. Finally, we introduce the essential matrix, which is a specialization of the fundamental matrix for the case when the camera calibration is known.

2.3.1 The epipolar geometry

Let X be a point in the three-dimensional space, which is observed by two cameras with optical centers C and C'. Each camera has an associated image plane, marked with Π and Π', respectively. The two camera centers form a line, which we will further refer to as the *baseline*. The baseline intersects the image planes in e and e', respectively, which are known

as the *epipoles*. Equivalently, an epipole is the image in one view of the camera center of the other view.

The lines connecting X with each camera center intersect the image planes in x and x$'$, respectively. The two camera centers, the 3D point and its corresponding image points define a plane, known as the *epipolar plane*. Note that for each pair of cameras we have a family of epipolar planes obtained by rotating an epipolar plane around the baseline. Each epipolar plane intersects the two image planes in l and l', which are known as the *epipolar lines*. The main entities described by the epipolar geometry are schematically represented in Fig. 2.7.

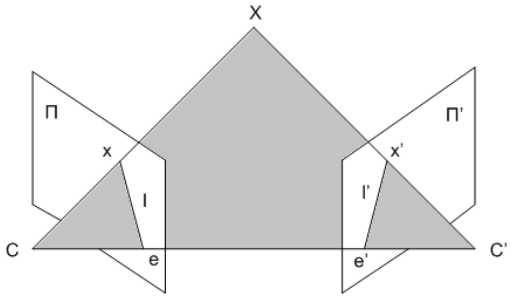


Figure 2.7: The epipolar geometry

An important rule derived from the above geometry is that if x and x' are images of the same point, then x' must lie on the epipolar line associated with x, i.e. l'. Of course, in order to establish the exact position of x' we need to know the position X(Fig. 2.8). This constraint is fundamental in computer vision, particularly in 3D reconstruction, and is known as the *epipolar constraint*.

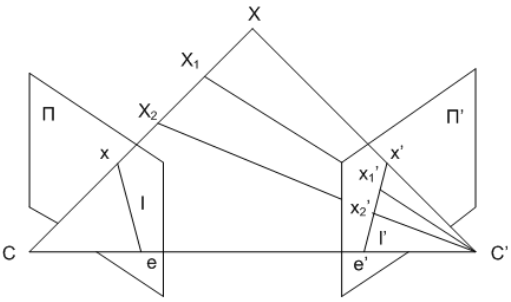


Figure 2.8: The epipolar constraint

2.3.2 Computing the fundamental matrix

Algebraically, the epipolar constraint can be written as:

$$x'Fx = 0 \tag{2.30}$$

where F is known as the fundamental matrix. Let $\mathrm{x} = (x, y, 1)^T$ and $\mathrm{x}' = (x', y', 1)^T$. Then we can rewrite equation (2.30) as:

$$x'xf_{11} + x'yf_{12} + x'f_{13} + y'xf_{21} + y'yf_{22} + y'f_{23} + xf_{31} + yf_{32} + f_{33} = 0 \tag{2.31}$$

Call f the vector containing the elements of F in row-major order, and then we can rewrite 2.31 as a vector inner product:

$$(x'x, x'y, x', y'x, y'y, y', x, y, 1)f = 0 \tag{2.32}$$

Stacking the equations obtained from a set of n point matches, we obtain:

$$Af = \begin{pmatrix} x'_1x_1 & x'_1y_1 & x'_1 & y'_1x_1 & y'_1y_1 & y'_1 & x_1 & y_1 & 1 \\ \dots & & & & & & & & \\ x'_nx_n & x'_ny_n & x'_n & y'_nx_n & y'_ny_n & y'_n & x_n & y_n & 1 \end{pmatrix} f = 0 \tag{2.33}$$

Since this is a homogeneous set of equations, f can only be determined up to scale. A solution to this system can be found using one of the techniques described in section B.2 of the appendix.

Before we move forward, it is important to mention some of the important properties of the matrix F, as well as its relation to the main geometric entities described in the previous subsection.

- F is a singular matrix ($det(F) = 0$), or equivalently it has rank 2
- F has seven degrees of freedom
- the epipolar lines are given by $l' = F\mathrm{x}$ and $l = F^T\mathrm{x}'$
- the epipoles are the null spaces of F and F^T: $Fe = 0$ and $F^Te' = 0$

The singularity constraint

Most of the applications of the fundamental matrix rely on the fact that it is singular. For instance, if this assumption does not hold, then the epipolar lines would not be coincident. In general, solving the linear system described by 2.33 would not result in a matrix F of rank 2, due to noisy measurements. For this reason, we need to take special steps to enforce this constraint. Again, we can formulate this as a minimization problem: find $\min_{F'}||F - F'||_{Frobenius}$ such that $\det(F') = 0$. This can be solved using SVD:

$$||F - F'||_{Frobenius} = ||UDV^T - UD'V^T||_{Frobenius} = ||D - D'||_{Frobenius} \tag{2.34}$$

D' is then found by setting the smallest singular value of D to 0.

The 8-point algorithm

The 8-point algorithm computes the fundamental matrix in two stages: first it determines a linear solution to $Af = 0$ subject to $||f|| = 1$. Then, the resulting matrix $\hat{F}$ is further refined to $\hat{F}'$ such that $\det(\hat{F}') = 0$. Recall from the DLT algorithm that in order to obtain reliable results, a proper normalization step needs to be included at the beginning of this procedure. This means that both x_i and x'_i must be first translated such that both centroids lie at the origin of the coordinates system. This is then followed by a scaling operation such that for each set of points, the average distance to the centroid is $\sqrt{2}$. The 8-point algorithm is summarized below:

Listing 2.4: The 8-point algorithm

```
normalize x̃_i = T x_i
normalize x̃'_i = U x'_i
assemble the system Af = 0
obtain the SVD of A, i.e. A = UDV^T
f is given by the last column of V
restore matrix F̂ from f
replace F̂ by F̂' such that det(F̂') = 0
F = U^T F̂' T
```

2.3.3 The essential matrix

The essential matrix is a specialization of the fundamental matrix when the camera calibration is known. Recall from section 2.2 that a camera can be decomposed as $P = K[R|t]$, where K is the calibration matrix and $t = -R\tilde{C}$. We now define the normalized coordinates as:

$$\hat{\mathrm{x}} = K^{-1}\mathrm{x} = [R|t]X \tag{2.35}$$

The equation of the essential matrix is:

$$\hat{\mathrm{x}}'^{T} E \hat{\mathrm{x}} = 0 \tag{2.36}$$

Substituting the normalized coordinates we obtain:

$$\mathrm{x}'^{T} K'^{-T} E K^{-1} \mathrm{x} = 0 \tag{2.37}$$

Thus, the relation between the fundamental and essential matrix is given by:

$$E = K'^{T} F K \tag{2.38}$$

Result 3 *Given the SVD decomposition of* $E = Udiag(1, 1, 0)V^T$ *and the first camera matrix* $P = [I|0]$*, there are four possible choices for* P'*:*

$$P' = [UWV^T|+u_3] \quad or\ [UWV^T|-u_3] \quad or\ [UW^TV^T|+u_3] \quad or\ [UW^TV^T|-u_3]$$

where

$$W = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } u_3 = U(0,0,1)^T \tag{2.39}$$

In order to select the right solution we test whether a point lies in front of both cameras. Summing up, there is a duality between the camera matrices and the essential matrix. In practice, once the calibration is determined, we compute the essential matrix either from the normalized point correspondences, or directly from the fundamental matrix. The advantage over the fundamental matrix F is that the projective ambiguity is removed, so the cameras are retrieved up to scale.

2.4 Pointcloud generation and bundle adjustment

2.4.1 Computing the 3D structure

Once the camera models and the fundamental matrices have been estimated, we can now proceed with the actual 3D reconstruction. Let X be a point in the 3D space, and let $\mathrm{x} = PX$ and $\mathrm{x}' = P'X$ be its corresponding image projections. Given the coordinates of x and x', we can determine the position of X by intersecting the two back-projecting rays from the measured image points, process known as triangulation.

Result 4 *The two rays connecting each camera center with x and x', respectively, will intersect if and only if the points satisfy the epipolar constraint.*

According to this result, if the image projections are perturbed by noise, then the back-projecting rays will never intersect, since x and x' will not satisfy the epipolar constraint 2.9. Under these circumstances, it is not possible to find an exact solution. Instead, we attempt to find the best solution by minimizing a suitable cost function.

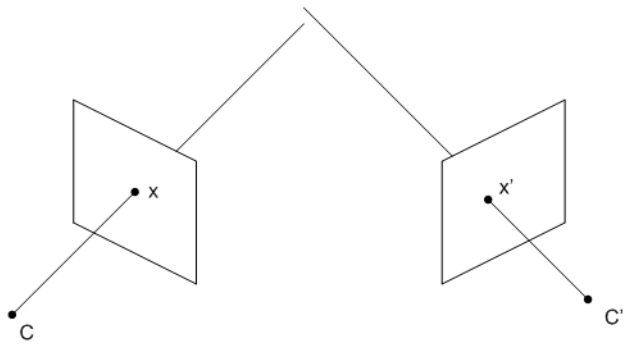


Figure 2.9: Non-intersecting rays due to noisy measurements

In this section, we assume that the cameras are known only up to a projective transformation, so we focus exclusively on projective reconstructions. An important design principle is that the method of triangulation should be invariant to this class of transformations:

$$\tau(x, x', P, P') = H^{-1}\tau(x, x', PH^{-1}, P'H^{-1}) \tag{2.40}$$

where τ is the triangulation method and H is a projective transformation. For instance, one might avoid the non-intersecting rays problem by taking the midpoint of their common perpendicular to be the original 3D point. However, this method is not projective invariant, because distance and perpendicularity are not preserved under this type of transformations.

Alternatively, instead of minimizing distance in the projective 3D space, we can attempt to minimize the reprojection error, i.e. the distance between the projections of X and the actual measured points 2.10. As we have seen in the previous sections, this should give us the maximum likelihood estimate under a Gaussian noise assumption. Note that if a different projection is used, X will project to the same image points, so this triangulation method is projective-invariant.

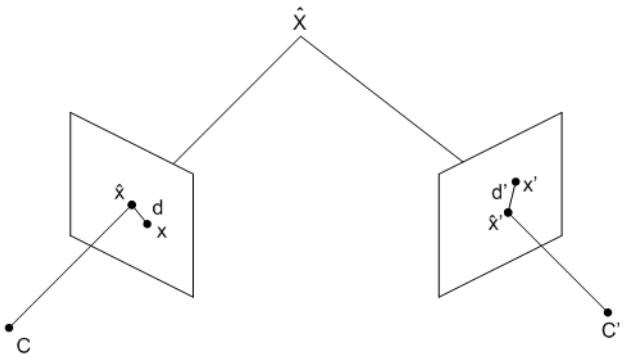


Figure 2.10: Minimizing the reprojection error

Linear triangulation

The linear triangulation is analogous to the DLT algorithm presented at the beginning of this chapter. Given two matching points $\mathrm{x} = PX$ and $\mathrm{x}' = PX'$, we can write:

$$\mathrm{x} \times PX = 0 \tag{2.41}$$
$$\mathrm{x}' \times P'X = 0 \tag{2.42}$$

Let $\mathrm{x} = (x, y, 1)$ and $\mathrm{x}' = (x', y', 1)$. Each cross product gives rise to a set of three equations, out of which only the first two are linearly independent. Thus, for each matching pair we obtain a homogeneous system of the form $AX = 0$:

$$\begin{pmatrix} xP^{3T} - P^{1T} \\ yP^{3T} - P^{2T} \\ x'P'^{3T} - P'^{1T} \\ y'P'^{3T} - P'^{2T} \end{pmatrix} X = 0 \tag{2.43}$$

A solution can be found either by solving directly the homogenous system, or by solving the equivalent inhomogeneous system (see appendix B). The problem is that none of these approaches is invariant to projective transformations. Under a transformation H, the original cameras P and P' are replaced by PH^{-1} and $P'H^{-1}$. Consequently, A is replaced by AH^{-1}, so the solution to the new system is HX, since $(AH^{-1})(HX) = 0$. Recall that the inhomogeneous system was solved by setting $X = (X, Y, Z, 1)^T$. However, this condition is not preserved by the transformation H. Analogously, the homogeneous method set the condition $||X|| = 1$, which is not invariant either.

As we discussed in the beginning of this section, a better approach is to minimize the reprojection error and then search for the exact solution of the resulting system. In other words, we search for $\hat{\mathrm{x}}$ and $\hat{\mathrm{x}}'$, which minimize:

$$C(\mathrm{x}, \mathrm{x}') = d(x, \hat{\mathrm{x}})^2 + d(x', \hat{\mathrm{x}}')^2, \text{ subject to } \hat{\mathrm{x}}'^T F \hat{\mathrm{x}} = 0 \tag{2.44}$$

where x and x′ are the measured points, while $\hat{\mathrm{x}}$ and $\hat{\mathrm{x}}'$ are the maximum likelihood estimates for the true image point correspondences. Note that these points satisfy the epipolar constraint, so the backprojecting rays will certainly intersect in the 3D space. Therefore, any triangulation method can now be used to determine the position of the point in space.

2.4.2 Bundle adjustment

Given a sequence of images our aim is to find the camera matrices P^i and the 3D points X_j such that $P^i X_j = x^i_j$. The problem is that if the measurements are noisy, then this system of equations will not have an exact solution. Under a Gaussian noise assumption we can find the Maximum Likelihood Estimate solution by solving a geometric distance minimization problem:

$$\min_{\hat{P}^i, \hat{X}_j} \sum_{i,j} d(\hat{P}^i \hat{X}_j, \mathrm{x}^i_j)^2 \tag{2.45}$$

where $\hat{P}^i$ are the estimated projection matrices and $\hat{X}_j$ are the estimated 3D points satisfying $\hat{x}^i_j = \hat{P}^i \hat{X}_j$, while x^i_j are the measured image points. This minimization problem is known as bundle adjustment and is usually included as a final step of any reconstruction algorithm.

Bundle adjustment suffers from two majors drawbacks: first, it can easily become a very large minimization problem and second, it is sensitive to local optima, and thus it requires a good initialization.

Assume that there are m cameras involved in the reconstruction (each with 11 degrees of freedom), and n 3D points (each with 3 degrees of freedom). This leads to a minimization problem with $3n + 11m$ parameters. Factoring or inverting matrices of size $(3n + 11m) \times (3n + 11m)$ can easily become infeasible as n and m increase. In order to address this issue, we can choose to include only a subset of all points, and fill the rest later by triangulation. Alternatively, we can partition the views in several sets (for instance groups of 3) and then bundle adjust separately each of these sets.

Recall that any projective transformation is only defined up to scale. Therefore, in order to uniquely identify a transformation we need to define the constant scale factors explicitly. Let λ_j^i be the corresponding scale factors for each projective camera, such that:

$$\lambda_j^i \mathrm{x}_j^i = P^i X_j \tag{2.46}$$

Stacking together the equations for each camera and for each 3D point, we obtain:

$$\begin{pmatrix} \lambda_1^1\mathrm{x}_1^1 & \lambda_2^1\mathrm{x}_2^1 & ... & \lambda_n^1\mathrm{x}_n^1 \\ \lambda_1^2\mathrm{x}_1^2 & \lambda_2^2\mathrm{x}_2^2 & ... & \lambda_n^2\mathrm{x}_n^2 \\ ... & & & \\ \lambda_1^m\mathrm{x}_1^m & \lambda_2^m\mathrm{x}_2^m & ... & \lambda_n^m\mathrm{x}_n^m \end{pmatrix} = \begin{pmatrix} P^1 \\ P^2 \\ ... \\ P^n \end{pmatrix} \begin{pmatrix} X_1 & X_2 & ... & X_n \end{pmatrix} \tag{2.47}$$

An important observation is that the right side of the equation contains two matrices of size $m \times 4$ and $4 \times n$, respectively. As a consequence, the matrix on the left (denote it by W) should also have rank 4. In practice, this will not hold because the measurements are perturbed by noise. Instead, we can compute matrix $\hat{W}$, such that $||W - \hat{W}||_F$ is minimal, subject to $\mathrm{rank}(\hat{W}) = 4$. Using SVD, we can write:

$$||W - \hat{W}||_F = ||UDV^T - U\hat{D}V^T||_F = ||D - \hat{D}||_F \tag{2.48}$$

Thus, $\hat{W}$ can be obtained by setting all but the first four diagonal entries of D to 0. Using the decomposition of $\hat{W}$, we then find the camera matrices and the 3D points:

$$(P^{1T}, P^{2T}, ..., P^{mT}) = U\hat{D} \tag{2.49}$$

$$(X_1, X_2, ..., X_n) = V^T \tag{2.50}$$

Note that we can always interpose a transformation H in 2.46, such that:

$$\lambda_j^i \mathrm{x}_j^i = (P^i H^{-1})(H X_j) \tag{2.51}$$

which shows that the reconstruction has a projective ambiguity. The bundle adjustment is typically done in several iteration steps, where at each step we determine the camera matrices and the 3D points, and we update the scale factors for each projective camera, as outlined below:

Listing 2.5: The projective factorization algorithm

```
normalize image points
start with λ_j^i = 1
repeat
        normalize λ_j^i = 1
        form the 3m × n measurement matrix W
        find Ŵ such that ||W − Ŵ|| is minimal and rank(Ŵ) = 4
        decompose Ŵ = UD̂V^T
        normalize x̃'_i = Ux'_i
        find camera matrices and the 3D points
        reproject points into each image to obtain new estimates of λ_j^i
```

Chapter 3

Gravity estimation

In this chapter, we compute the gravity direction based on the orientation of the vertical walls present in the reconstructed scene. Our first observation is that the notions of gravity and verticality are closely intertwined. A wall is vertical if and only if its normal is approximately orthogonal to the gravity direction. Although exceptions can be found, particularly in modern architectural styles, most urban environments obey this very basic design rule.

This principle suggests that in order to determine the gravity direction we must first find all vertical walls in the reconstructed scene. The underlying assumption is that the scene contains at least two non-parallel walls, otherwise the gravity direction is not uniquely determined.

Furthermore, it is important to make a distinction between a planar structure and a vertical wall. A planar structure is defined as a set of points that lie on the same plane, and it does not necessarily correspond to a physical structure, fact which is known as the virtual plane problem. Therefore, our aim is to design an algorithm that can identify exclusively physical vertical walls.

An interesting question that rises here is how can we make a distinction between a vertical wall and other structures present in the scene. Our secondary assumption is that a large proportion of the points in the 3D pointcloud belong to vertical walls, and any other structures can be eliminated through a filtering stage.

3.1 Vertical walls detection

A common approach for planar surface detection is the RANdom SAmple Consensus (RANSAC) algorithm (see section 2.1.2), which attempts to find model parameters in an iterative fashion. RANSAC's main assets are its simplicity and its relatively good performance in practice. In principle, no assumptions are made about the data and no unrealistic conditions have to be satisfied for RANSAC to succeed [8].

In a classical formulation of RANSAC, the problem is to find all inliers in a set of data points. Inliers are data points consistent with the target model, and typically, the number of inliers I is not known a priori.

The first step of the algorithm is to draw a sample of p points from the data uniformly and at random. The value of p is given by the minimum number of points required to fit the required abstraction. For instance, in our particular case, fitting a plane requires at least 3 points.

During the next stage, for each data point outside the sample we compute the distance from that point to the fitted plane. If the distance is lower than a threshold t, then we mark it as an

inlier. The procedure is then repeated for k iterations. At each iteration, the algorithm stores the model with the highest number of inliers, i.e. the best fit.

The virtual plane problem

A potential drawback of the RANSAC algorithm is known as the virtual plane problem [34]. As the name suggests, this is an artifact resulting from attempts to find structure in a particular area of the point cloud where there is no structure at all. RANSAC methods are prone to this form of over-fitting because they search exclusively for planes maximizing the number of inliers. In other cases, virtual structures are generated by merging multiple physical structures. For instance, if two parallel walls lie in the same plane, they can be interpreted as a single structure by the RANSAC algorithm (Fig. 3.1).

3.2 Clustering and noise filtering

In order to alleviate the virtual plane problem, we propose a two-step procedure. First, we cluster the 3D points using the K-means algorithm [21], and then, instead of running RANSAC on the entire pointcloud, we run it on individual clusters. The intuition behind this approach is that points belonging to the same physical object are usually close together, forming relatively dense structures. Using this rough segmentation of the pointcloud, we significantly reduce the chance of merging unrelated physical structures into a single structure (Fig. 3.2).

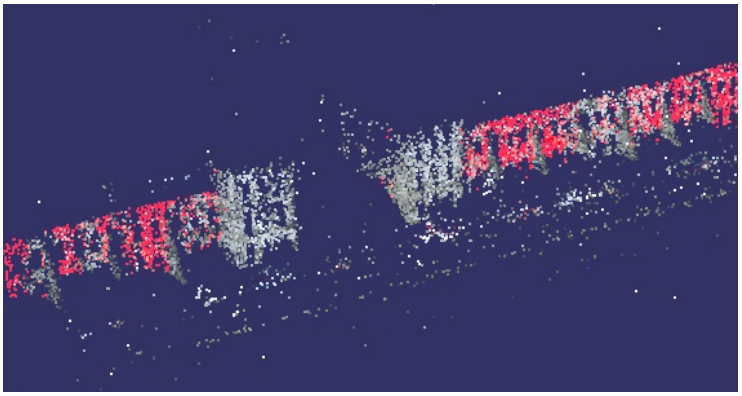
Note however that virtual planes can still be generated due to the noisy feature points present within each cluster. Nonetheless, a distinguishing characteristic of these points is that they tend to be very sparse, and therefore, they can be easily identified by estimating local density of the cluster, i.e. the number of 'close' neighbors of each point. We use a proximity threshold to define the maximum distance up to which two points are considered 'close', and a density threshold to define the minimum number of 'close' neighbors a point needs to have to be considered valid. Any point that does not satisfy this density constraint is removed from the pointcloud (Fig. 3.2).

Computing the interpoint distance inside each cluster requires a significant amount of memory, since it needs to store a square matrix of size given by the number of elements in the cluster. Thus, if the cluster is too large we can easily run into computational bottlenecks. On the other hand, splitting the pointcloud in too many clusters can lead to loss of semantic information, as some monolithic structures may be accidentally split between multiple clusters. Determining the optimal number of clusters is therefore not trivial as it requires a balance between the computational cost and keeping a meaningful separation of the points.

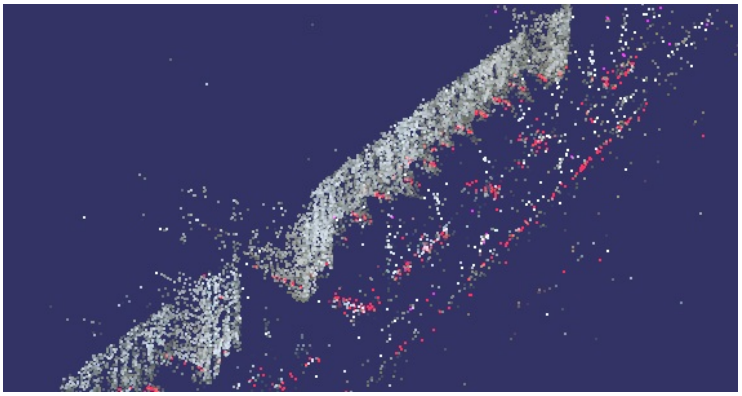
Another problem is that searching for this optimal value is also computationally expensive since at each iteration the clusters have to be recomputed. In addition, the cost of computing the clusters increases linearly with the number of clusters.

In order to design a search strategy, we must first define a stopping criterion. Since the size of the cluster poses significant computational limitations, we believe that a good idea would be to stop when the size of the largest cluster has fallen below a certain threshold.

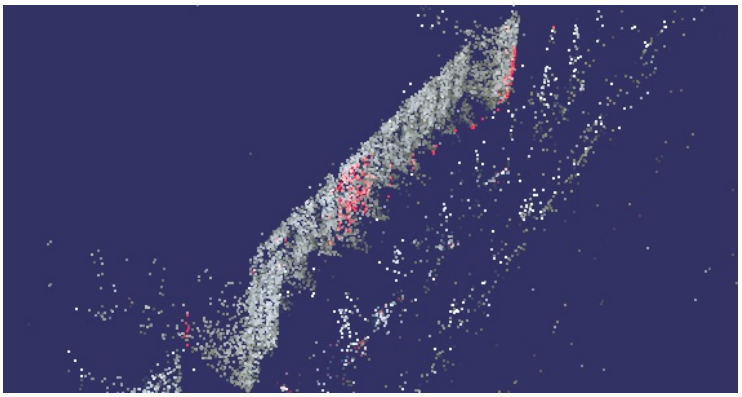
Our initial search strategy was inspired by the exponential backoff scheme [1] used in computer networks. At each iteration we doubled the number of clusters and checked if the stopping criterion is satisfied. The problem with this approach is that it is largely suboptimal if the number of clusters is relatively small, and it cannot be written as a power of 2. For instance, if the optimal value is 18, our algorithm will repeatedly cluster in 2, 4, 8, 16, and 32 clusters. In this case, computing the 32 clusters takes more time than if we actually searched linearly from 2 to 18.



(a)



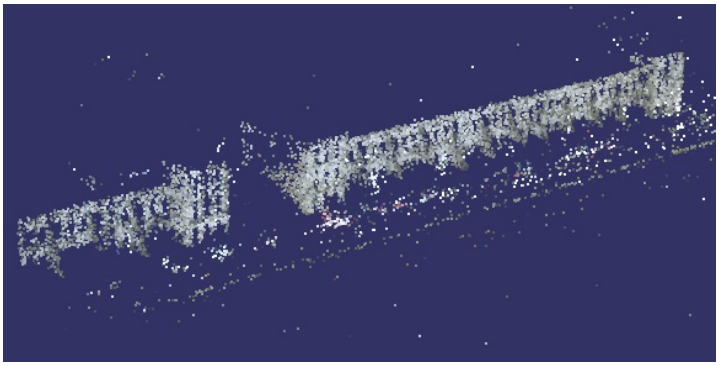
(b)



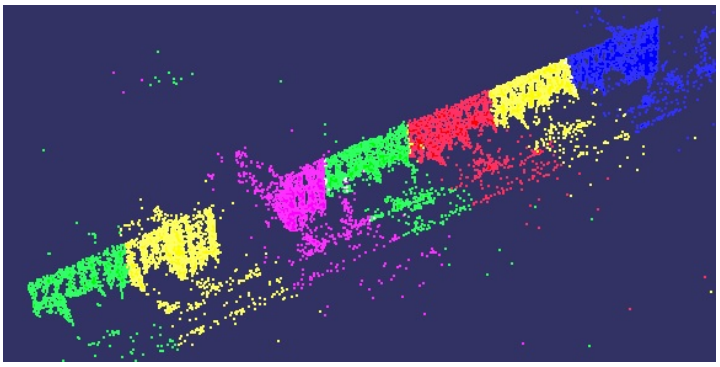
(c)

Figure 3.1: Different types of virtual planes (marked with red). (a) two parallel walls merged together (b) virtual plane made up of noisy feature points (c) virtual plane containing both real walls and noisy feature points

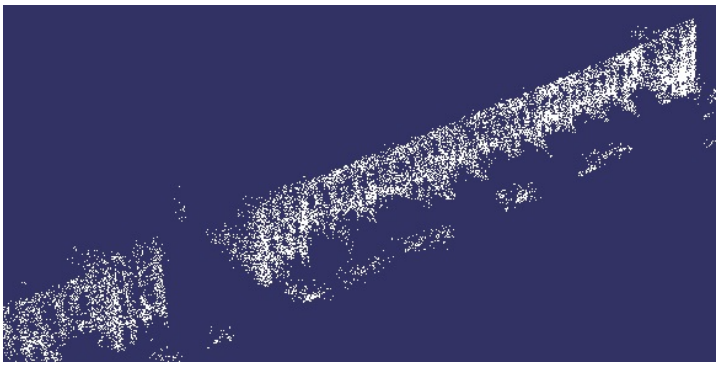
An important observation is that a large number of clusters is required only when the size of the pointcloud is very large, i.e. hundreds of thousands of elements. However, even in this case, we can downsample it, so that only a small number of clusters are actually generated. For this reasons, we decided that the best course of action is to use a simple linear search strategy.



(a) Original pointcloud



(b) Clusters



(c) Noise filtering

Figure 3.2: The effect of clustering and noise filtering

3.3 Gravity estimation

Once a set of vertical walls is identified in each cluster, we are left with the problem of computing the actual gravity direction. Given two non-parallel normals, the gravity direction is given by the cross product of the two vectors. Note that in our case, the system is over determined, since we often get more than two vertical walls. Therefore, the gravity estimation can be modeled as a nonlinear least-squares curve fitting problem of the form:

$$min_x||(f(x))||^2 = min_x(f_1^2(x) + f_2^2(x) + ...f_n^2(x)) \tag{3.1}$$

where $f(x)$ is an n-dimensional error function and $f_i(x)$ are its components for each dimension. We define the error function components as follows:

$$f_i(x) = \left| \frac{\pi}{2} - acos\left(\frac{v_i \cdot x}{||v_i|| \cdot ||x||} \right) \right| \cdot w_i; \tag{3.2}$$

where x is the current estimation of the gravity direction, v_i is a normal to a building wall, and w_i is the weight of this normal, i.e. the number of inliers of the corresponding plane. The first term of the error function suggests that we want x to be 'as orthogonal as possible' to v_i. Note that this term is adjusted such that the angle is always kept in the $[0..\frac{\pi}{2}]$ interval. The intuition behind the second term is that the planes with more inliers should play a bigger role in estimating the gravity direction. More weight is given to walls containing a large number of inliers because they are more likely to provide us with a good indication of the gravity direction.

The non-linear least square optimization starts with an initial guess of the gravity direction x_0, and then, it repeatedly tries to optimize the value of this vector such that the overall error function $f(x)$ is minimal.

In practice, the *acos* function becomes numerically instable for angles that are near 0 or π. Alternatively, we can minimize directly:

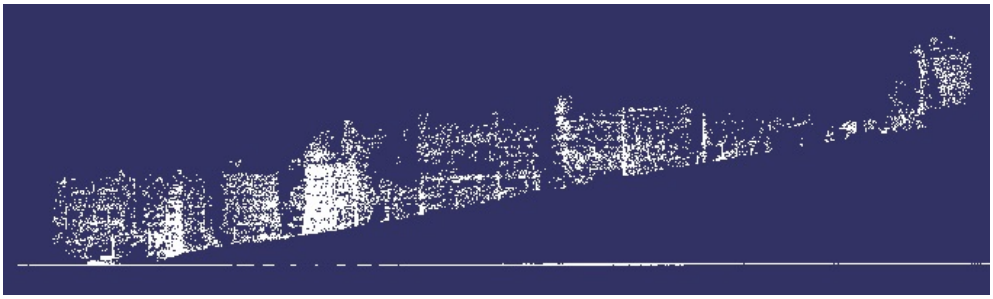
$$f_i(x) = \left| \left(\frac{v_i \cdot x}{||v_i|| \cdot ||x||} \right) \right| \cdot w_i; \tag{3.3}$$

The reference plane

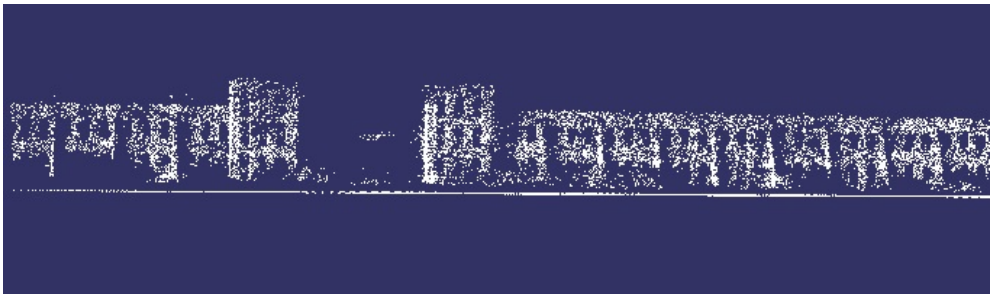
We define the reference plane as a plane with normal parallel to the gravity direction. Initially, we set its height at $-\infty$ and we evaluate the height of each point as the distance to the reference plane. Then, we update the height of the reference plane such that it passes through the lowest point of the pointcloud. As we will see in the next chapter, the reference plane serves as an initial estimate of the ground surface. Fig. 3.3 shows the position of the reference plane for our two data sets.

3.4 Evaluation

Fig. 3.2 and 3.4 show the results of clustering and noise filtering procedures on two separate data sets. The first pointcloud is large - it contains approximately 155K points, while the second pointcloud is significantly smaller, with around 16K points. In order to avoid computational bottlenecks, the first pointcloud has been downsampled to approximately 17K points. In both cases we can see that the resulting clusters are only a rough segmentation of the pointcloud.

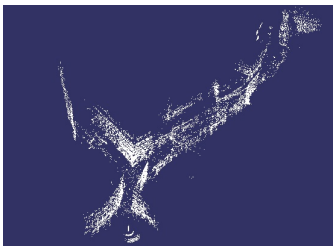


(a) Uneven landscape

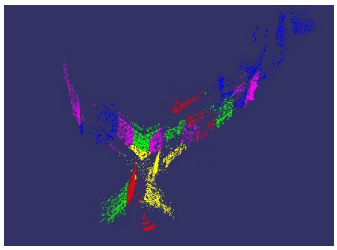


(b) Flat landscape

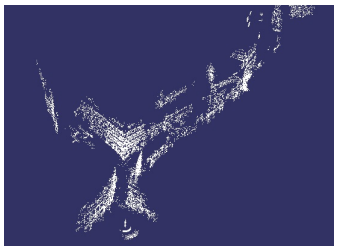
Figure 3.3: Cross section of two 3D models. The reference plane is represented by the white line lying at the bottom of the pointclouds. For a relatively flat landscape, the reference plane is a good approximation of the ground surface. For uneven landscapes, the errors are too large.



(a) Original pointcloud



(b) Clusters



(c) Noise filtering

Figure 3.4: Clustering and noise filtering on a different dataset

A building wall is often shared between multiple clusters, and each cluster can contain pieces of multiple vertical walls (Fig. 3.5). However, this is enough to ensure that two separate walls are not interpreted as a single structure by the RANSAC algorithm.

For a more quantitative analysis, one needs to know the true gravity direction. Since this information is not available, we establish the true gravity direction by manually tweaking the coordinates of the gravity vector. Although this is not perfectly accurate, it still provides us with a good indication of the true gravity direction, so we can use it as a ground truth.

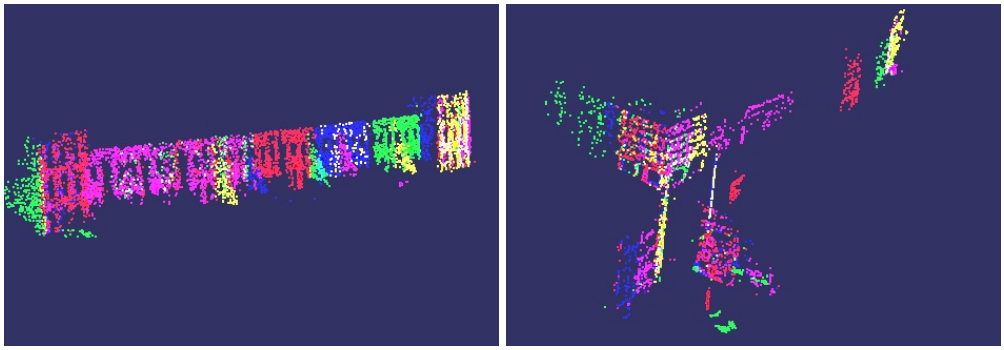


Figure 3.5: The identified vertical planes. For a better visualization, we randomly assign colors to the identified planes. Note however that if two planes share the same color, it does not mean that they have been identified ensemble

In our first experiment, we try to identify the importance of the weights on the error estimation. For this reason, we compare the performance of our system for a weighted and a non-weighted error function, averaged over 20 trials. The difference between the two error functions is almost undistinguishable. The mean error after 20 trials is 0.2329 for the weighted function, and 0.2356 for the non-weighted function (Table 3.1). This can be explained by the fact that there is a large consensus among the identified walls, and no virtual planes with arbitrary orientations have been included.

Next, we set up an ablation experiment, where we remove the clustering and noise filtering components. The resulting system is never able to compute the correct gravity direction. The mean error is 84.1208, fact which indicates that the identified gravity direction is almost orthogonal to the true gravity direction. This shows that some of structures returned by the RANSAC algorithm are now virtual planes parallel with the reference plane. The cross product of these vectors result in a direction that is almost orthogonal to the true gravity direction.

	mean error (in degrees)
weighted error function	0.2329
non-weighted error function	0.2356
ablation experiment	84.1208

Table 3.1: The mean error for different experimental setups

From a computational perspective, the additional cost incurred by clustering and noise filtering steps is relatively low (Table 3.2). As expected, the most expensive operation in the gravity estimation pipeline is represented by the vertical planes estimation.

sampling	0.005s
clustering	1.082s
noise filtering	1.955s
vertical planes estimation	6.527s
gravity estimation	1.009s

Table 3.2: The computational cost of the gravity estimation pipeline

Chapter 4

Ground surface estimation and refinement

The ground surface estimation is of great importance for many computer vision applications such as autonomous navigation systems, accurate scene modeling for serious gaming, surveying applications, topographical mapping and so on. In some applications, having a detailed representation of the ground surface is not a strict requirement and a simple model such as a single planar structure is often a good approximation. However, for 3D urban reconstruction, it is critical to catch all the intricacies of the local landscape, otherwise the quality of the resulting 3D models will be very poor. Therefore, our aim is to build a method that can provide us with an accurate estimation of the ground surface, which we can then use for GIS and Google maps registration, and for estimating building heights.

Looking at a 3D scene, humans rarely have trouble inferring the position of the ground surface. They fuse information from multiple sources including spatial and temporal information, but they also incorporate prior knowledge [7]. From a computer vision perspective, this task is less trivial, as we are often faced with a disambiguation problem. Relying strictly on the information provided by the 3D pointcloud it is impossible to distinguish the ground surface from any other surrounding surface, for instance the surface defined by the building roofs. For this reason, our primary assumption in estimating the ground surface is that the normals to this surface point roughly towards zenith direction.

4.1 Ground surface initialization

We represent the ground surface as a mesh of $N \times N$ cells, which allows us to approximate surfaces of arbitrary shape. We further refer to this structure as the *heightmap*. Each cell in the mesh contains height information about a certain region in the 3D space. Consequently, the higher the number of cells, the higher the quality of the approximation. Of course, the number of cells cannot be increased indefinitely, because the height information contained in the pointcloud is finite. Therefore, the maximum quality that can be obtained is bounded by the pointcloud density. Note however that high quality approximations can lead to computational bottlenecks, since it becomes more difficult to manipulate the resulting mesh.

Our first step is to find a correspondence between the original 3D space that we need to represent, and the heightmap. We define u : X -> [0..1] such that:

$$u(x) = \frac{x - min(x)}{max(x) - min(x)}; \tag{4.1}$$

Given a 3D point p with coordinates (x, y, z), we use function b to normalize its x and y coordinates such that their values lie in the [0..1] interval. By multiplying the normalized coordinates with N, we obtain the (i, j) coordinates of the cell that contains point p. From this perspective, we can view the heightmap as a 2D histogram that assigns for each point in the 3D space a corresponding bin.

The mesh is initialized with height information given by the reference plane $P = (a, b, c, d)$:

$$heightmap(u(x), u(y)) = -\frac{ax + by + d}{c} \tag{4.2}$$

As we discussed in the previous chapter, the reference plane is only a rough estimation of the ground surface. In order to get a more accurate estimation, we also need to take into account the height information provided by the pointcloud. For each cell, we select the 3D points contained within that cell and we compute the minimum distance to the reference plane. This distance provides an approximate measure of the error in estimating the height in that particular region of space (4.3). As a consequence, we increase the height of that cell with an amount equal to this distance, such that the error is approximately 0. The resulting 3D surface is depicted in Fig. 4.1.

$$err(i, j) = \min_{k \in cell_{ij}} \frac{|ax_k + by_k + cz_k + d|}{\sqrt{a^2 + b^2 + c^2}} \tag{4.3}$$

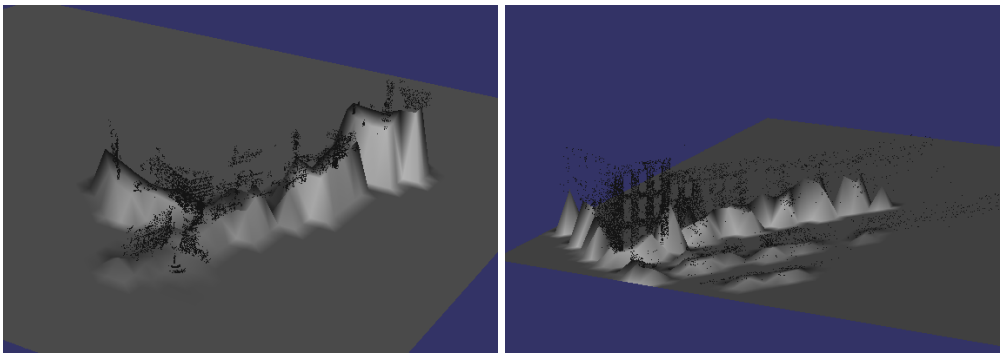
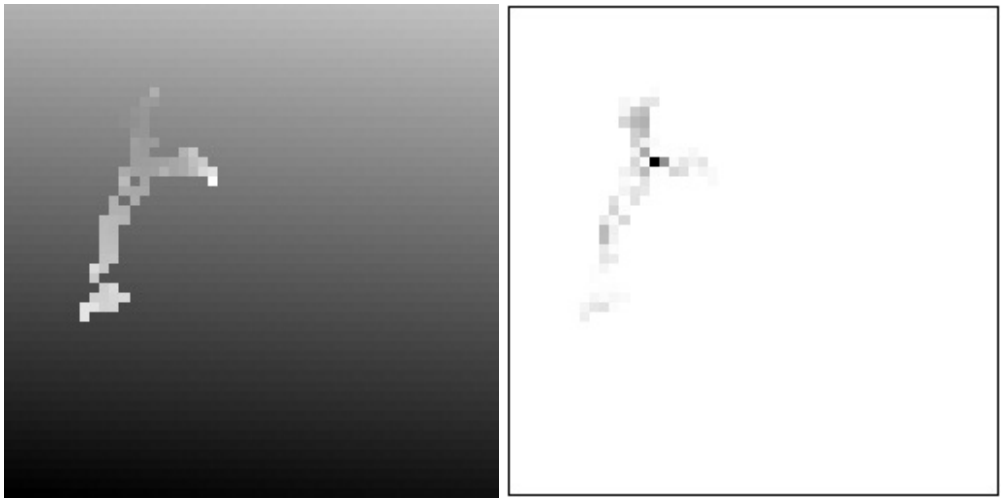


Figure 4.1: Initialization of the ground surface

In parallel with the heightmap, we build an additional structure that stores weight information about each cell. This is further referred to as the *weightmap* (Fig. 4.2). The intuition behind this structure is that we want cells with a high support (i.e. containing a lot of 3D points) to play a bigger role in computing the final shape of the ground surface. The weight of each cell is given by

$$w(i, j) = \frac{support(i, j)}{\text{size of the pointcloud}} \tag{4.4}$$

In addition, we define a minimum support threshold of 0.001. If a cell has a weight smaller than this threshold, then the error estimation is not reliable enough, so we discard it. As a consequence, we round its weight at 0, while the corresponding cell in the heightmap it is left unchanged.



(a) Heightmap (b) Weightmap

Figure 4.2: Visualization of heightmap and weightmap. On the left, dark regions correspond to low heights, while on the right dark colors suggest large weights

4.2 Refinement techniques

After the initialization stage, a large proportion of the heightmap contains abrupt transitions due to missing or unreliable data. Our aim is to use the cells with accurate height information as a basis for updating the entire map and creating a much smoother surface. For this reason, we have implemented a set of refinement techniques that lead to a more accurate and more visually pleasing 3D model.

The first refinement step is to smooth out declinations that are too steep using an iterative procedure. In order to make this step independent of the magnitude and sign of the heights, we first normalize the heights such that they all lie in the [0..1] interval. Then, for each cell in the mesh, we compute the maximum declination with respect to its neighbors, using a search window of size w. The maximum declination is defined as the absolute difference between the height of the current cell and the minimum height of its neighbors:

$$maxdeclination(i,j) = abs(H(i,j) - \min_{k,l} H(k,l)), \quad \begin{cases} i-w \leq k \leq i+w \\ j-w \leq l \leq j+w \end{cases} \tag{4.5}$$

If maximum declination is higher than a predefined threshold, then the current height is reduced to the minimum height of its neighbors.

Some structures cannot be considered noise, but they cannot be used for height estimation either. Think for example of a partially reconstructed wall, lying at an arbitrary height, with no

connection with the surrounding structures. If this height is poorly estimated, then it cannot be adjusted, as there is no support data in its proximity. This can have a major negative impact on the entire surface estimation. For this reason, it is often better to remove such isolated structures. During the next refinement step, we view the weightmap as a grayscale image, and we remove any isolated structures that have less than a predefined number of pixels. In order to ensure consistency between the weightmap and the heightmap, we also restore the value of the corresponding cells in the heightmap to their initialization values, i.e. the values given by the reference plane at that location.

Using the limited height information provided by the pointcloud, we need to make an estimation of the ground surface for cells where no height data is available. Although these cells have already been initialized with height information given by the reference plane, this is not a reliable estimation particularly for uneven landscapes. Our idea is to use an iterative procedure, which propagates the heights from the cells where we do have evidence towards the cells with missing data.

At each iteration step, a cell with missing data computes the average height of the cells found within its lookout window. Note that any cells with unreliable height information, i.e. zero-weighted cells, are ignored. Current cell is then assigned a value equal to this average height. Since no data was available at this location, a very low value is given for the weight of this cell (e.g. 0.001). In order to get a better understanding of this process, one can view it as if each cell with a known height would propagate this value on the entire unknown space. The effect of all propagations is then averaged resulting in the final configuration of the map.

Finally, the last refinement step smoothens out the resulting surface. Unlike the propagate heights step, which computed the averages of cells with non-zero weights, this step takes into account all neighbors found within the search window. However, the cells which were updated during the propagate heights step carry a lesser weight, since their estimation is less reliable.

4.3 Evaluation

Fig. 4.3 shows the effect of the declination refinement on two separate datasets. The first pointcloud depicts a set of buildings situated on an uneven surface. The second pointcloud is a close-up on a single building situated in relatively plane area. In both cases, we use a number of 3 iterations and a search window of 3. The maximum declination threshold is 0.2 in the first case, and 0.1 in the second.

The effect of declination refinement is most visible for the second data set. We can see that a large portion of a wall is missing, so there is a large error in estimating the initial heights. However, based on the information provided by the surrounding cells, the algorithm is able to correct this error and obtain a more accurate representation.

Fig. 4.4 shows the resulting heightmap, after any structures made up of less than 2 pixels have been removed from the corresponding weightmap. Apparently, only one pixel (corresponding to one cell in weightmap) has been changed after this step. Although quantitatively this might not seem much, from a qualitative perspective, the effect can be significant. This location is left unchanged by the declination refinement stage, as there is no support data around it. Therefore, propagating a potentially unreliable height at a later stage might have disastrous effects.

Next, we propagate heights, using a number of 33 iterations and a lookout window of 1. The maximum number of iterations is bounded by the size of the heightmap (which is 51 in our case). However, the actual number of iterations is usually much lower, depending on the exact position and the coverage of the initial estimation. Fig. 4.5 shows the surface estimation after 5, 10, 20 and 33 iterations.

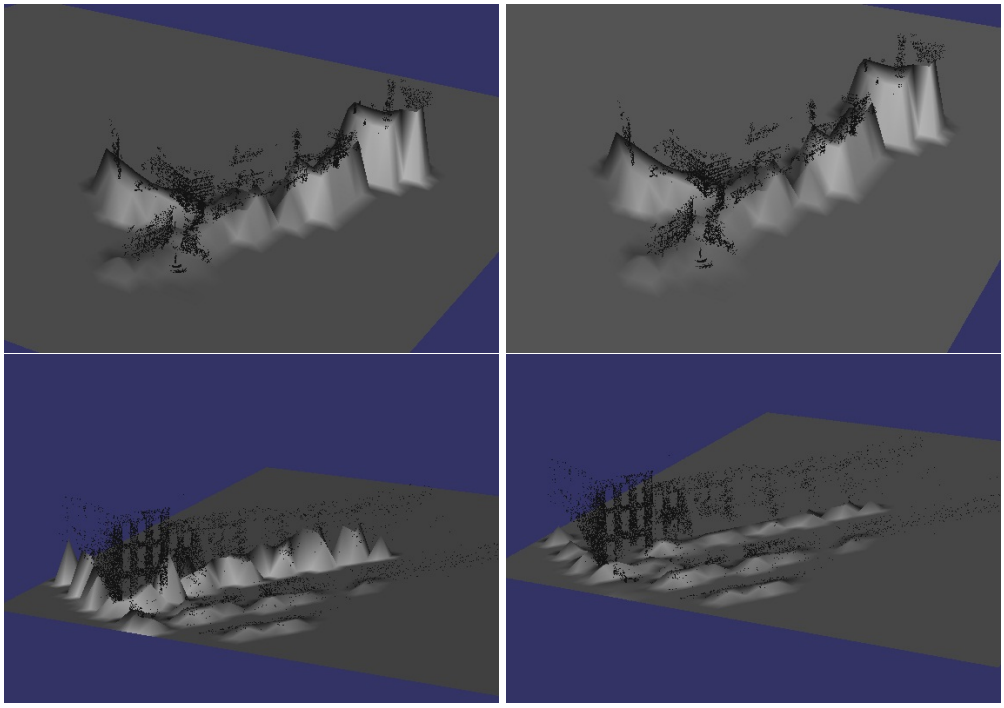


Figure 4.3: Visualization of declination refinement. On top, a pointcloud and a ground surface depicting a larger area; at the bottom, close-up on a single building reconstruction. On the left, we see the initial height estimation, while on the right we see the corrected heights after 3 iterations

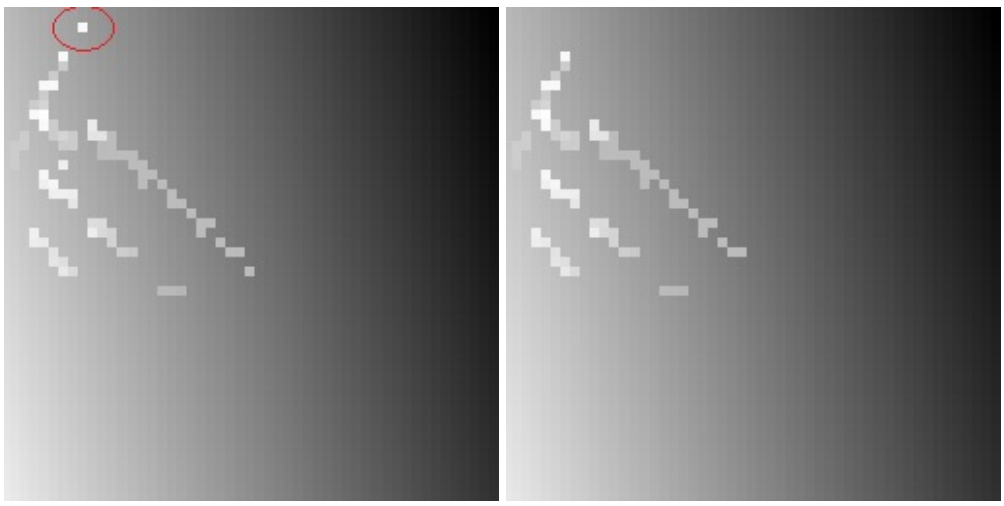
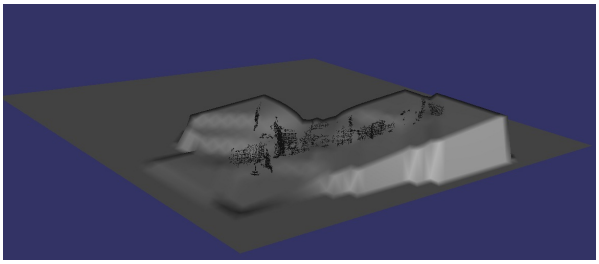
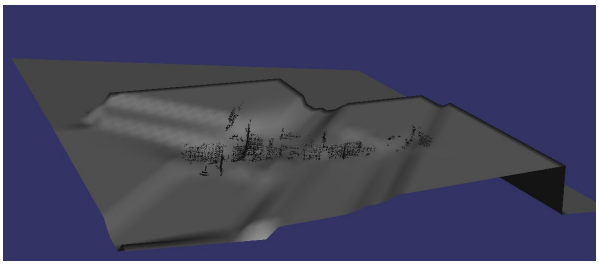


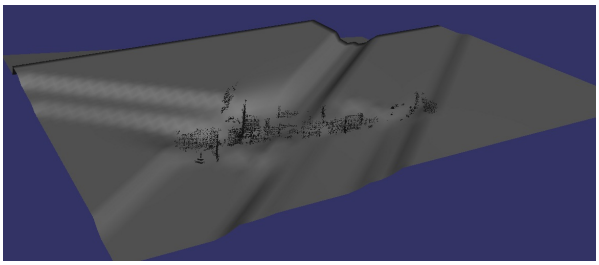
Figure 4.4: Visualization of isolated patch removal



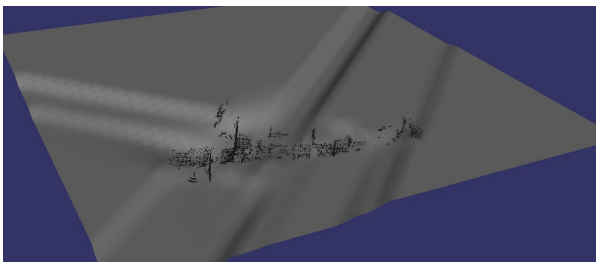
(a) after 5 iterations



(b) after 10 iterations



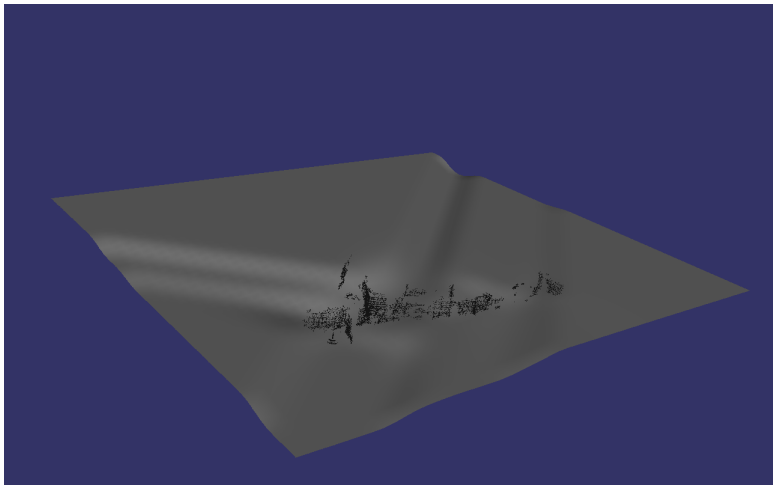
(c) after 20 iterations



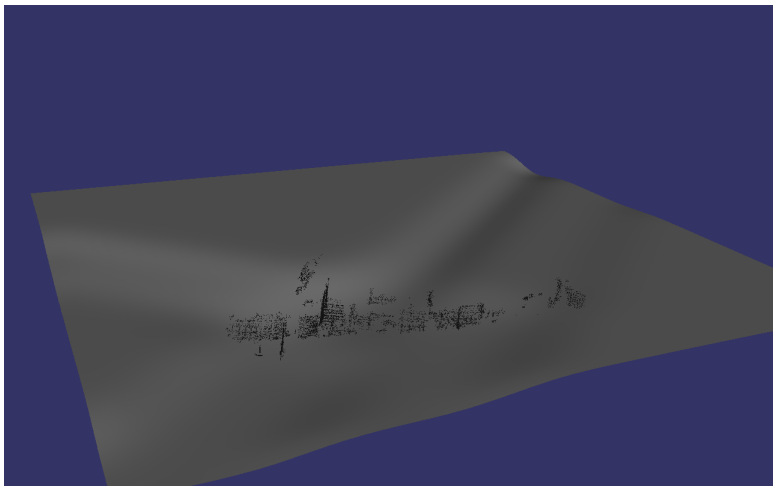
(d) after 33 iterations

Figure 4.5: Visualization of height propagation

For the final smoothing operation, we run 5 iterations with a lookout window set to 1. Naturally, the higher the number of iterations, the smoother the surface. At the limit, if the number of iterations is high enough the surface becomes perfectly flat. Therefore, we need to find a balance between too abrupt transitions and a too flat surface. Experiments have shown that a number of 5 iterations provides good quality results (Fig. 4.6).



(a) after 1 iteration



(b) after 5 iterations

Figure 4.6: Visualization of the final smoothing step

Summing up, the height propagation and the final smoothing step can be viewed as applying a convolution filter of the form:

$$\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix} \tag{4.6}$$

where the weights w_{ij} are given by the local weightmap. The main distinction between the two is that the height propagation is done only for cells with missing data, while the smoothing operation is run over the entire mesh.

Chapter 5

Extracting GIS data

3D models obtained using stereo vision techniques often lack a significant amount of detail due to missing data, occlusions or noise. In 3D urban reconstruction, there is growing interest to use measurements from Geographical Information Systems (GIS) and Global Positioning Systems (GPS) for further refinement of building models. Integrating the existing 3D models with this additional information can lead to more appealing and realistic reconstructions.

We extract GIS data from OpenStreetMap (OSM), which is an open-source project that provides free access to worldwide geographic data. This is a community effort, where each member records the precise locations of roads, buildings, amenities etc of a particular area, and then uploads it to the OSM database. Its main advantage is that any errors produced by one user, are quickly detected and corrected by the other members of the community. On the other hand, the downside of this aspect is that each user can choose its own way to structure the data and, as a consequence, the information is not always represented consistently.

The greatest challenge with the data exported from OSM is that it does not provide a directly accessible representation of buildings. Each map is stored as a set of polygons and there is no guarantee whether the polygons are closed or open, or whether two polygons belong to the same structure or not. Therefore, before registering our 3D models with this data, there are several problems that we need to solve.

First of all, we must identify and eliminate any structure that does not refer to a building. Elements such as parks or underground parking cannot be reconstructed as they are either flat, or not visible. Secondly, we need to group together all polygons that belong to the same building, so that we can represent each building uniformly. Thirdly, in order to be able to render the resulting 3D models, it is critical to compute the contour of each building, or equivalently, a closed polygon that marks the building perimeter.

In this chapter, we propose several techniques that can be used to partially or totally alleviate the above mentioned problems. The algorithm for the actual GIS registration is presented in the next chapter.

5.1 Parsing the data

The OpenStreetMap project allows us to export data in multiple formats. However, we decided to use the *.osm* format, which is XML-based, and therefore directly accessible for automated processing. The most important entities found within the *.osm* file are the *nodes* and the *ways*. Each *node* is described using a set of 9 attributes, out of which the most important are the latitude and the longitude coordinates. No information concerning the height is provided. The

ways are abstract entities modeling not only 1D structures such as roads or lanes, but also 2D structures corresponding to buildings, parks etc. Each *way* is uniquely defined by a set of nodes.

We manually compiled a list of words (e.g. 'railway', 'barrier', 'parking' etc), which provide us with a good indication of irrelevant structures. If one of the cue words is found within the annotated data, then the corresponding structure is discarded. In the end, the parsing stage produces a set of *ways* and *nodes*, which are represented as a weighted unoriented graph.

The weight of each edge is given by the Euclidean distance between the two vertices that mark the edge margins. Note however that in order to compute this distance, we must transform the ellipsoidal coordinates to Universal Transverse Mercator (UTM) coordinates using the equations below:

WGS84 ellipsoid constants:

$$a = 6378137$$

$$e = 8.1819190842622E^{-2}$$

Prime vertical radius of curvature:

$$N = a/sqrt(1 - e^2 \cdot sin(lat)^2)$$

Cartesian coordinates:

$$x = N \cdot cos(lat) \cdot cos(lon)$$

$$y = N \cdot cos(lat) \cdot sin(lon)$$

$$z = ((1 - e^2) \cdot N) \cdot sin(lat)$$

When computing the transformation, we make the assumption that a - the major semi-axis of reference ellipsoid, and e - the squared eccentricity of the reference ellipsoid, correspond to the values defined by the World Geodetic System standard (WSG84 [24]).

5.2 Cleaning the data

Fig. 5.1 shows the GIS map after the parsing stage. On top of this map we have marked two major issues: the non-closing polygons and the bridges, which block us from representing each building as a closed polygon.

We define terminal nodes as nodes in the graph which are linked to a single other node. If a polygon contains one or more terminal nodes, then the polygon is non-closing. Non-closing polygons occur mostly on the map edges due to the fact that the buildings lying on the map margins are truncated, and only the visible segments are actually included in the *.osm* file.

A bridge is defined as an edge whose deletion increases the number of connected components in the graph. Equivalently, an edge is a bridge if and only if it is not contained in any cycle.

As we discussed at the beginning of this section, our aim is to represent each building as a closed polygon. Therefore, given the above definitions, it becomes evident that in order to satisfy our goal, we need to remove from the graph any non-closing polygons or bridges.

Removing non-closing polygons

Searching for non-closing polygons is equivalent with searching for nodes with a single adjacent element. Note however that once a node is removed from the graph, a previous non-terminal

Figure 5.1: Visualization of GIS map. The black text boxes mark some of the non-closing polygons and bridges occuring in this map

node can now become terminal. This indicates using an iterative procedure for eliminating all terminal nodes.

First, we traverse the initial graph and we add al terminal nodes to a queue. Then, at each iteration step, we remove the first node from the queue and we check if any of its neighbors becomes a terminal node. All new terminal nodes are then added to the queue. The process stops when there are no nodes left in the queue. The complexity of this algorithm is given by $O(V + E)$, where V is the number of vertices and E is the number of edges.

Listing 5.1: Removing non-closing polygons

```
Q = [];
for each node n in the graph
        if n is terminal
                add n to Q

while Q is not empty
        remove e - first element in the queue
        for each neighbor of e
                if neighbor is isolated
                        add neighbor to queue
```

Removing bridges

We can find a simple algorithm to detect bridges starting right from their definition. First, we count the number of connected components in the initial configuration of the graph. Then, we pick an edge from the graph and we remove it. If the number of connected components

increases, it means we have found a bridge and we store it in a separate list. We add back the edge in the graph and we repeat the process for a different edge. The process stops when all edges have been exhausted. Finally, we traverse once more the stored bridges, and we remove them from the graph. The number of connected components can be determined in $O(V + E)$ operations, so the overall complexity of this step is $O(E(V + E))$.

Listing 5.2: Removing bridges

```
bridges = [];
initialize n1 with the number of connected components
for each edge e in the graph
        remove e
        let n2 be the new number of connected components
        if n2 > n1
                add e to bridges
        put back e in the graph

for all e in bridges
        remove e from the graph
```

An important step in the above procedure is to compute the number of connected components. This is typically done using the Breadth First Search (BFS) algorithm [19]. We first choose a random node and we explore the graph starting from that node. All the nodes that can be reached from that node form a connected component. At the next step, we choose another node that has not been visited so far and we repeat the procedure. The algorithm stops when all nodes have been visited.

Listing 5.3: Detecting connected components

```
procedure getConnectedComponents
        let NV be the nodes that have not been visited
        while NV is not empty
                choose node n from NV
                cc = BFS(n)
                add cc to connected_components

        return connected_components

procedure BFS(n)
        let V be the nodes that have been visited
        Q = []
        component = []
        add n to Q
        while Q is not empty
                add n to V
                extract e from Q
                add e to component
                for each neighbor nb of e
                        add nb to Q

        return component
```

5.3 Extracting building contours

We defined a building contour as a closed polygon that marks the building perimeter. One approach for finding such a polygon would be to search for the Hamiltonian cycle that encompasses the largest area. However, as shown in DeLeon[10], determining if a graph is Hamiltonian, i.e. contains a Hamiltonian cycle, is an NP-complete problem. This suggests that in order to find a solution to our problem, a better approach is required.

Our idea is inspired by the wall following algorithm [29], which is the best-known rule for traversing mazes. By keeping his left or his right hand in contact with one wall, a player is guaranteed to find an exit, if there is one. Otherwise, he will return to the entrance. Note however that on its way back to the entrance, the player will follow the maze contour, which is exactly what we are looking for.

Initially, we make a rough estimation of the building contours by computing the convex hull of each connected component. The convex hull of a set of points is the smallest convex set that contains the points. Note that the points in the convex set are also ordered, so this already brings us closer to our final objective. We perform this step using the MATLAB implementation of the Qhull algorithm [4].

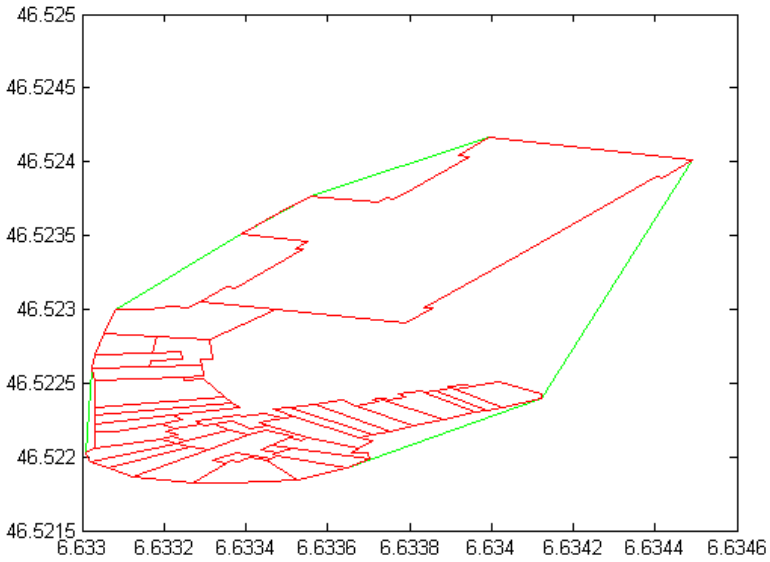


Figure 5.2: The convex hull of a building ground plan. Real edges are marked with red. False edges induced by the convex hull are marked with green

As can be seen in Fig. 5.2, there is no guarantee whether the edges of the convex hull are also edges in the graph. In many cases, the area defined by the convex hull is much larger than the area contained within the building perimeter. In order to address this issue, we first identify false edges (i.e. edges induced by the convex hull), and then we replace them with an alternative path computed by the wall follower algorithm. Each alternative path is determined iteratively, by choosing the left-most edge at each iteration (Fig. 5.3).

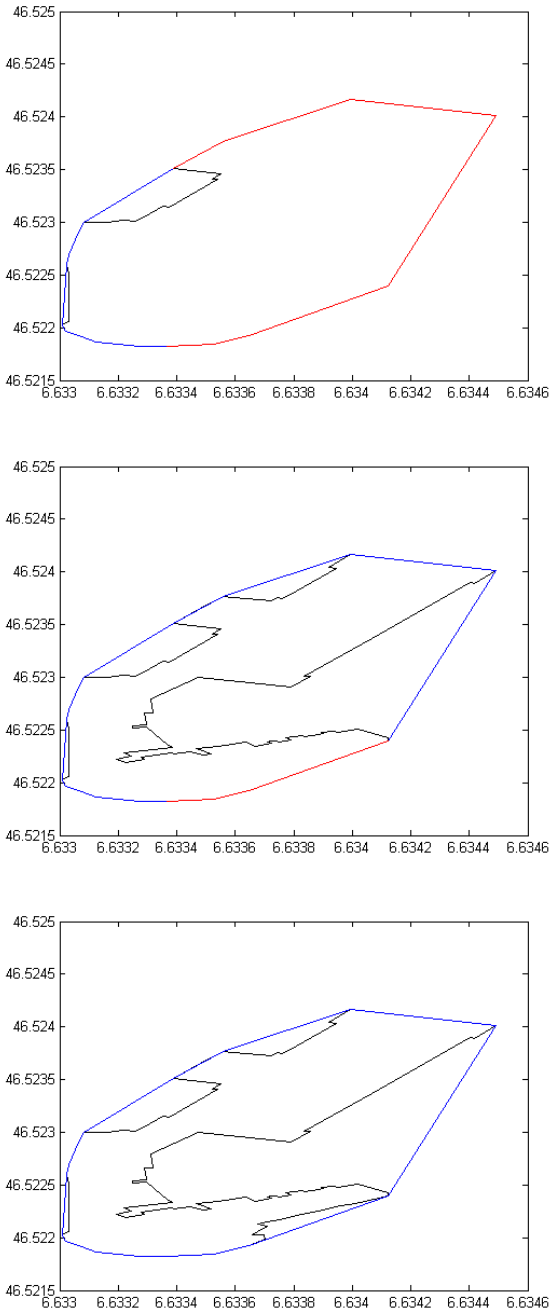


Figure 5.3: Replacing the false edges induced by the convex hull. Hull edges that have already been considered are marked with blue. Black edges represent the alternative paths computed by wall follower algorithm. Red edges represent hull edges that have not been considered yet

5.4 Evaluation

The resulting building contours overlaid on the initial map are depicted in Fig. 5.4. In order to evaluate the performance of our algorithm, we artificially generate a set of grid graphs (Fig. 5.5) with sizes varying from 25 to 100 nodes. This type of graphs is characterized by a relatively high number of edges, so they are usually expensive to explore. In what concerns the real data, the largest connected component extracted from the initial GIS data has approximately 150 nodes.

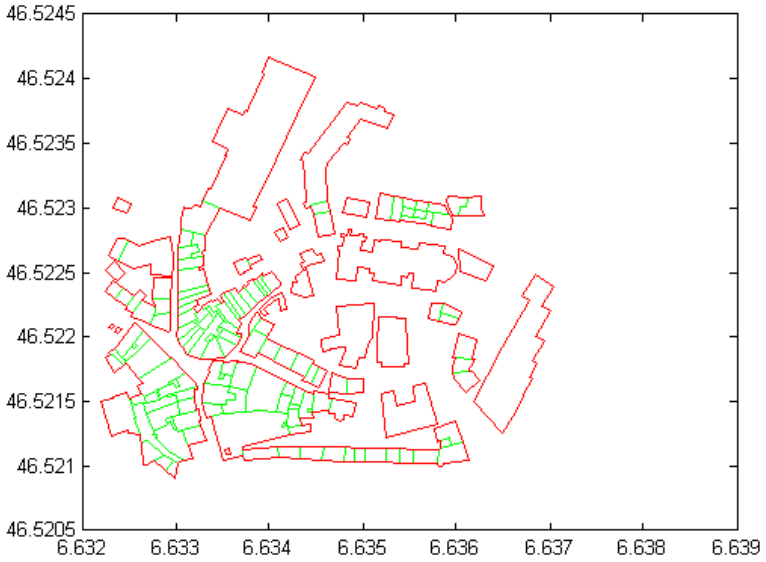


Figure 5.4: The resulting building contours, marked with red. The edges belonging to the original graph are marked with green

As can be seen from Fig. 5.5, the runtime grows only linearly and it takes less than a second to deal with the largest graph. This is explained by the fact that the complexity of our algorithm is not dependent on the entire graph size, but rather on the size of the perimeter. While the total number of edges increases exponentially with the number of nodes, the perimeter increases only linearly, and therefore our system is highly scalable. The Qhull algorithm runs in $O(V^{\frac{d}{2}})$, where V is the number of vertices in the original graph and the d is the number of dimensions. Therefore, the overall time complexity of our method is $O(V + E_P)$, where E_P is the number of edges forming the perimeter.

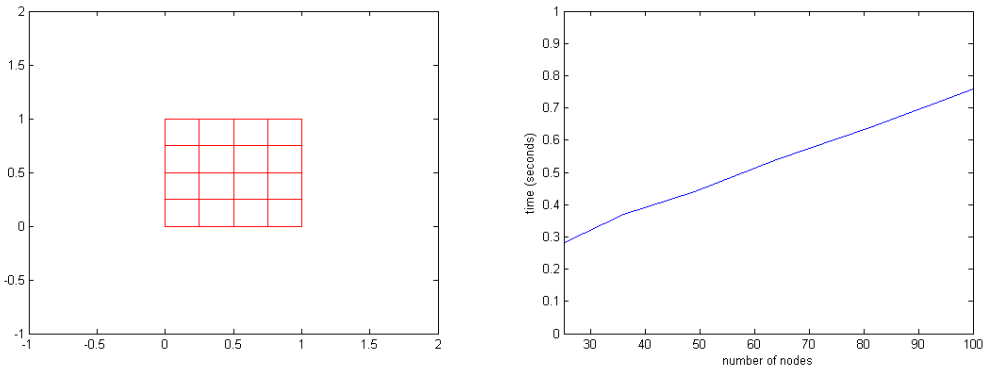


Figure 5.5: On the left, an example of a grid graph with 25 nodes. On the right, the runtime of the wall following algorithm for grid graphs of different sizes

Chapter 6

GIS and Google maps registration

After extracting the GIS data and putting it into a more convenient format, we now need to register this data with our existing 3D models. This however requires prior knowledge about the exact position of the reconstructed scene. Typically, this information is unavailable, except for more advanced systems that annotate the images used for reconstruction with GPS data. In this chapter, we propose a semi-automated method for 3D model geo-location.

We define the *pointcloud footprint* as the projection of a 3D pointcloud onto the reference plane. Our aim is to find a method to overlay the building contours computed in the previous chapter onto the pointcloud footprint. This will allow us to establish the exact coordinates of the pointcloud and to fill in possible missing regions of the 3D model. The *pointcloud footprint* and the building contours are related to each other by a set of 2D transformations: a rotation, a translation, and a scaling factor. Therefore, in order to compute the overall transformation we need at least two points correspondences between the two.

These point correspondences cannot be detected in a fully automatic manner due to the limited amount of information. On one hand, the pointcloud footprint might be incomplete (i.e. some walls might be missing), and therefore, matching it against some possible candidates would not produce reliable results. On the other hand, even if we assume that the pointcloud footprint is complete, we are still left with a disambiguation problem. For example, in dense residential areas, which are planned systematically, it is a common feature to have multiple areas bearing the same structure. Our solution to this problem is to create a simple Graphical User Interface (GUI), where the user can manually select the point correspondences, based on his prior knowledge on where the pictures were taken.

6.1 GIS registration

In order to match the 3D models against the building contours, we must first transform the 3D pointcloud into a 2D structure. Therefore, our first step is to project each data point onto the reference plane. Once the 3D pointcloud is projected on the reference plane, we can then match it against the building contours. At this step, the user is prompted to select two pairs of point correspondences using the simple GUI depicted in Fig. 6.1.

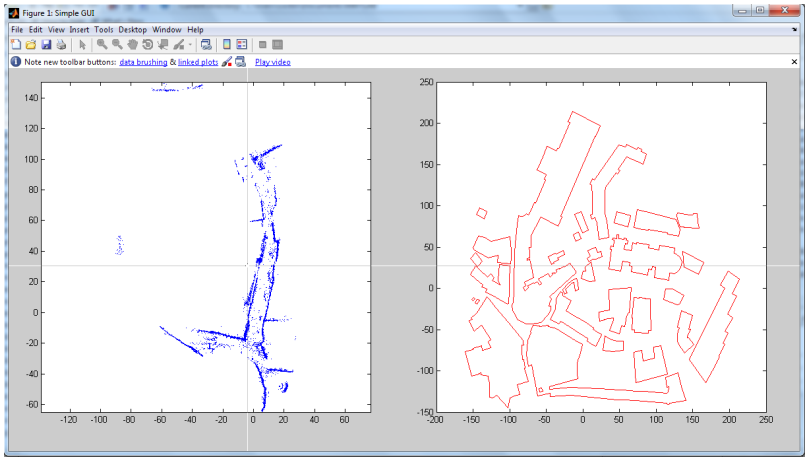


Figure 6.1: The GUI for selecting point correspondences. On the left, the 2D projection of the point cloud. On the right, the building contours extracted from the GIS data

Let x_i and x'_i be the selected data points from the building contours and the pointcloud footprint, respectively. Our aim is to find a matrix H, such that $\mathrm{x}'_i = H\mathrm{x}_i$. Note however that H is not a general homography as in section 2.1.1, but rather an Euclidean transformation. Therefore, there are only 4 degrees of freedom (2 for translation, 1 for rotation, and 1 for scaling), so two point correspondences are enough to compute this transformation.

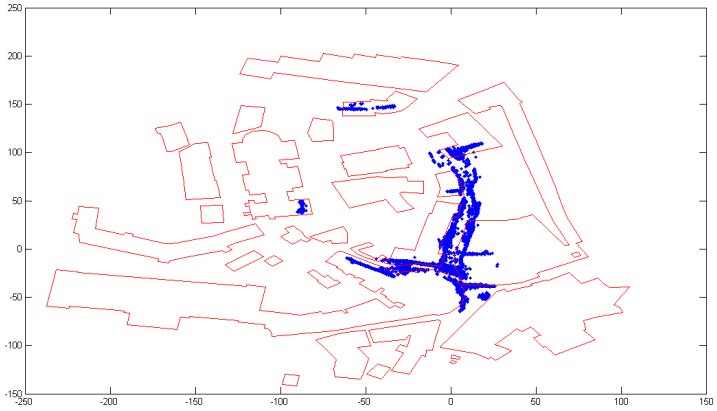


Figure 6.2: The GIS registration. Blue dots represent the projection of the original 3D point-cloud. Red edges represent the overlaid ground plans

As can be seen in Fig. 6.2, the building contours do not match exactly with the pointcloud footprint. This is caused by the fact that there are several sources of error that can affect the registration. For instance, the building ground plans might not be perfectly accurate. The data uploaded to OpenStreetMap is not recorded using a controlled setup, so it might contain

measurement errors, which are then further propagated to each component of our system. There are also depth estimation errors within the pointcloud itself, so the pointcloud footprint is also an approximation. Finally, the system is very sensitive to the point correspondences selected by the user. Any errors introduced during the selection phase can produce a skew in the GIS registration.

In section 2.1.3, we have seen that the RMS estimation error can be reduced by using more measurements. Analogously, we can reduce the user-induced errors, by allowing the user to select more point correspondences. However, for the purpose of this thesis, we considered that two point correspondences are enough, as the effect of this error is relatively small compared to the effect of the other sources of error described above.

6.2 Google map registration

Following the same workflow as for the GIS registration, we now register the building contours with a Google map. This allows us to add texture on the ground surface and on building roofs, which creates the impression of a more realistic reconstruction. Fig. 6.3 shows the result of the Google map registration. Again, the matching is not perfect, but it provides us with a good indication of the ground and roofs texture.

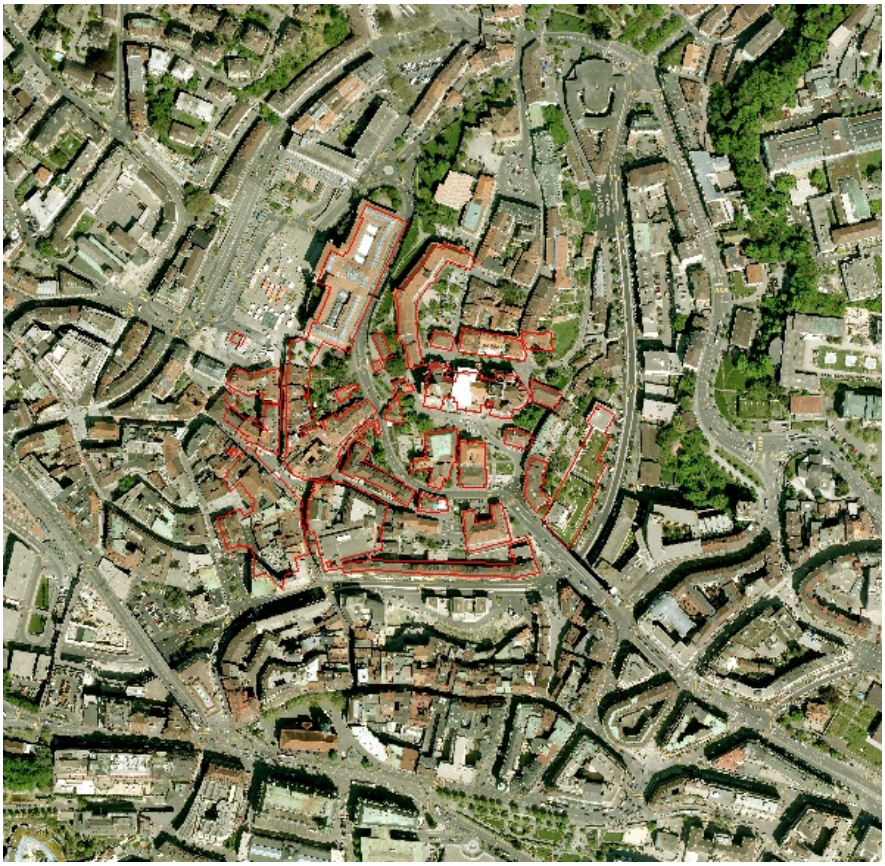


Figure 6.3: The Google map registration. Red edges represent the overlayed ground plans

6.3 Evaluation

In this section, we present a qualitative analysis of our system. Fig. 6.4 shows a typical 3D pointcloud, which we use as an input. We can see that in the initial 3D model most of the buildings are only partially reconstructed: many vertical walls are incomplete, while the ground surface or the buildings' roofs are completely missing.

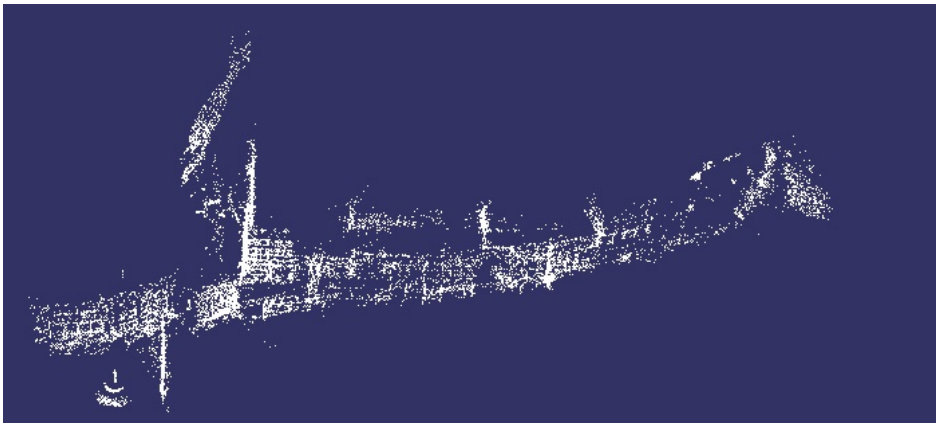


Figure 6.4: The initial 3D pointcloud

In Fig. 6.5, we can see the mesh that models the ground surface. Although the landscape is highly uneven, the mesh is able to capture all the intricacies and produce a very good estimation of the ground surface. All areas, including those where no height information is available, are seamlessly fused together creating a more realistic impression of the 3D scene.

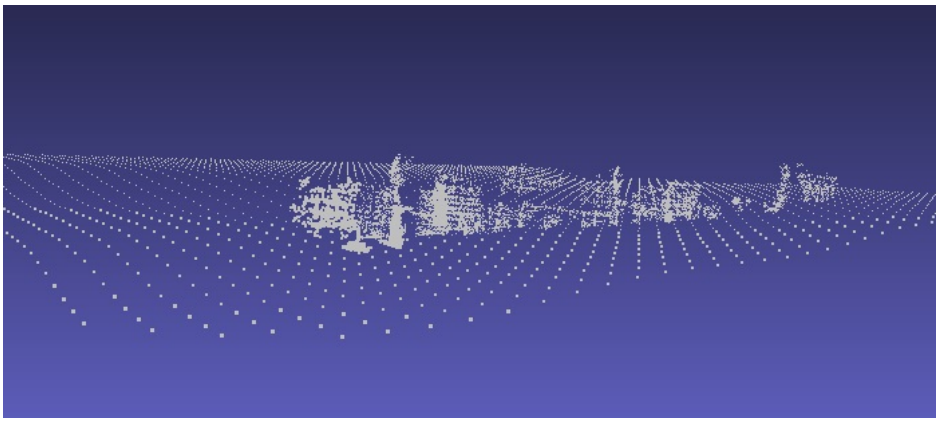


Figure 6.5: The mesh approximating the ground surface

After the GIS registration, the building contours are grown with their corresponding height. For each building we estimate the height by taking the average height of the pointcloud in that particular region of space. This intermediate step can be visualized in Fig. 6.6.

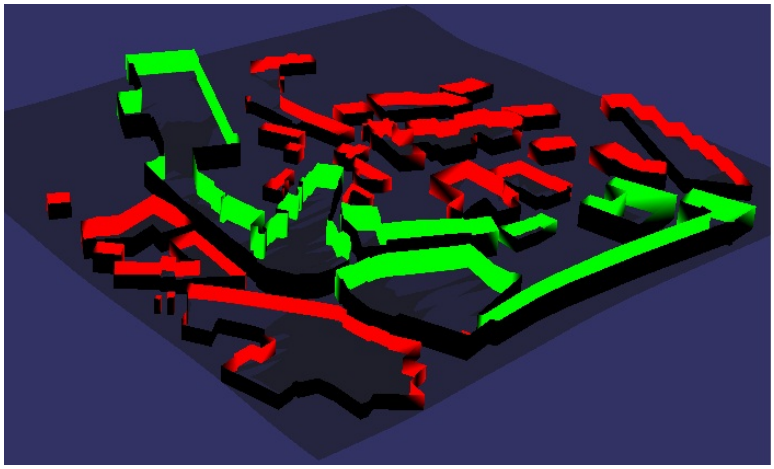


Figure 6.6: Growing the building contours. Buildings marked with red are assigned a random height, as there is no data to estimate the height

Finally, the last step is to texture the ground surface and the roofs with aerial images extracted from Google maps. As can be seen in Fig. 6.7, the overlay is not perfect. On one hand, there are small inaccuracies in the OSM database, while on the other, the aerial images are not taken orthogonally to the ground surface (building walls are visible from certain angles). For this reason, it is not possible to match exactly the building contours and the aerial images. However, despite these problems, the resulting 3D model is much more realistic and more visually appealing than the initial pointcloud.

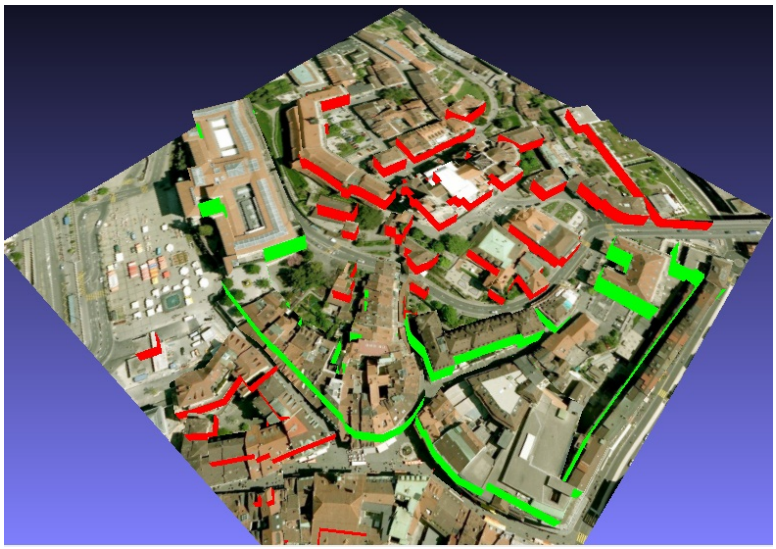


Figure 6.7: The final 3D model

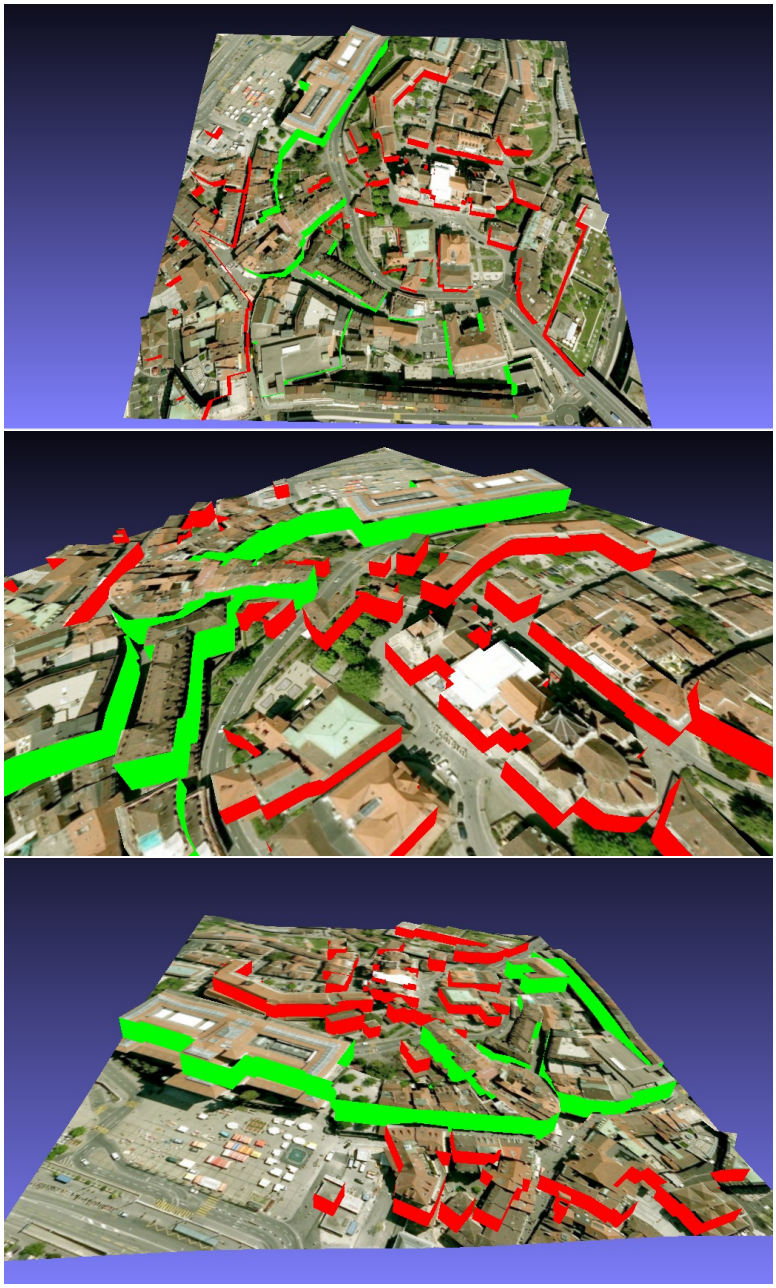


Figure 6.8: The final 3D model - different perspectives

Chapter 7

Conclusions

In this thesis, we have proposed several techniques that can be used to enhance existing 3D models and obtain a more appealing and realistic look. We assumed that a pointcloud has already been generated using the well-established stereo vision algorithms, and we concentrated on the post-processing of the resulting pointcloud.

In chapter 3, we have shown that it is possible to compute the gravity direction based on the orientation of the vertical walls. Robust estimation techniques are prone to a form of over-fitting, known as the virtual plane problem. We have successfully solved this issue using clustering and noise filtering. Ablation experiments have shown that without these two steps the performance of the system degrades significantly. In fact, the system is no longer able to detect the true gravity direction. Based on the gravity direction, we made an initial estimation of the ground surface, which we called the reference plane. For relatively flat landscapes, this is actually a very good approximation of the ground surface.

For uneven landscapes, a more advanced model is required in order to represent the intricacies of the ground surface. In chapter 4, we replaced the reference plane with a mesh and we used several iterative steps to build a more accurate representation of the ground model. The main challenge is that the relative height of the landscape is known only for very few positions. Our algorithm propagates these heights in their neighboring areas, and then composes the effects using a weighted average. The intuition here is that dense regions should have a stronger effect, since the measurements are more reliable than for sparse regions.

Finally, our last contribution was to integrate the existing 3D models with GIS data. In chapter 5, we proposed a fully automatic technique that can extract ground plans from GIS data. We used the wall following algorithm to locally refine the initial contours produced by the Qhull algorithm. Experimental results have shown that this method is highly scalable as it can deal with relatively large graphs in less than a second.

The GIS data is registered with the 3D pointcloud semi-automatically, due to the inherent ambiguities of the reconstruction. Recall from chapter 2, than a scene can only be reconstructed up to an Euclidean transformation with respect to the world frame. Therefore, user input is required in order to match the ground plans with the pointcloud footprint. In chapter 6, we used the registered 3D models to fit more complex primitives (blocks), and to project the texture provided by a Google map.

A potential extension to this work would be to obtain a more fine-grained representation of the ground plans. So far, we only managed to obtain the building contours, while the walls contained within this contours are completely ignored. A more advanced algorithm should try to represent each building as a set of closed non-self-intersecting polygons. In addition, all

edges belonging to a connected component should be included in at least one polygon. Dynamic programming combined with some form of heuristic should be a good course of action. However, our preliminary results have not been successful at achieving this task.

Another future extension would be to allow optimal texturing of vertical walls. Some portions of vertical walls are visible in multiple images, so an interesting problem is choosing the 'best' image for texturing. For this, we first need to define an optimality criterion, i.e. define what does 'best' mean for this specific problem. For instance, choosing the closest image, or the 'most parallel' image are some potential criteria that might ensure a good performance.

Appendix A

2D Transformations

A.1 Homogeneous representation of primitives

A line in the plane is represented by an equation of the form $ax + by + c = 0$. Alternatively, we can describe the same line using a vectorial notation $(a, b, c)^T$. Two vectors $(a, b, c)^T$ and $k(a, b, c)^T$ represent the same line, so they are considered equivalent. Therefore, there is not a one-to-one correspondence between vectors and lines. The family of vectors related by a non zero-scaling factors defines an equivalence class known as a *homogeneous vector*.

A point $\mathbf{x} = (x, y)^T$ lies on the line $\mathbf{l} = (a, b, c)^T$ if and only if it satisfies $ax + by + c = 0$. Equivalently, we can rewrite this equation as $(x, y, 1)\mathbf{l} = 0$. Note that for any $k \neq 0$, if $(x, y, 1)\mathbf{l} = 0$ then the equation $(kx, ky, k)\mathbf{l} = 0$ is also. Therefore, it is natural to consider the set of vectors $(kx, ky, k)^T$ to be a homogeneous representation of the point $(x, y)^T$.

The equation of a plane is given by $ax + by + cz + d = 0$. Following the same reasoning as above, we can consider the vector $(ka, kb, kc, kd)^T$ to be the homogeneous representation of the plane. Below we enumerate some important results derived from the homogeneous representation of primitives.

Result 1 *A point* $\boldsymbol{x}$ *lies on the line* $\boldsymbol{l}$ *if and only if* $\boldsymbol{x}^T\boldsymbol{l} = 0$.

Result 2 *The intersection of two lines* $\boldsymbol{l}$ *and* $\boldsymbol{l'}$ *is the point* $\boldsymbol{x} = \boldsymbol{l} \times \boldsymbol{l'}$.

Result 3 *The line passing through two points* $\boldsymbol{x}$ *and* $\boldsymbol{x'}$ *is* $\boldsymbol{l} = \boldsymbol{x} \times \boldsymbol{x'}$.

The operator $\times$ is the cross product of two vectors and is defined as follows:

$$\mathbf{a} \times \mathbf{b} = \begin{pmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix} = (a_2b_3 - a_3b_2)\mathbf{i} + (a_3b_1 - a_1b_3)\mathbf{j} + (a_1b_2 - a_2b_1)\mathbf{k} \tag{A.1}$$

A.2 A hierarchy of transformations

In this section, we start with the most specialized 2D transformations - the isometries, and then we move forward towards the most general - the projective transformations.

Isometries

Isometries or Euclidean transformations model the motion of a rigid object and are usually represented as:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \epsilon cos(\theta) & -sin(\theta) & t_x \\ \epsilon sin(\theta) & cos(\theta) & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} R & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{A.2}$$

Depending on whether ϵ is $+1$ or -1, the isometry preserves, or reverses the orientation. In the compact representation, R is known as the rotation matrix , $\mathbf{t}$ is the translation vector and $\mathbf{0}$ is a null vector. Special cases of an isometry are a pure rotation ($\mathbf{t} = 0$) and a pure translation ($R = I$). An important observation is that any isometry preserves distances, angles and areas. This transformation has three degrees of freedom (1 for rotation and 2 for translation), so it can be computed from only two point correspondences.

Similarity transformations

A similarity transformation is an isometry composed with an isotropic scaling [1]:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s \cdot cos(\theta) & -s \cdot sin(\theta) & t_x \\ s \cdot sin(\theta) & s \cdot cos(\theta) & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sR & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = H_s \mathbf{x} \tag{A.3}$$

Compared to an isometry, this transformation has an additional degree of freedom represented by the scaling factor. In order to compute a similarity transformation, a minimum of two point correspondences is required.

An important property is that the angles between lines are not affected, so shape is always preserved under a similarity transform. Ratios of lengths and areas are also invariant, because the scaling factor cancels out. However, the distance between two points is not similarity-invariant.

Affine transformations

An affine transformation is a non-singular [2] linear transformation followed by a translation:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} A & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = H_a \mathbf{x} \tag{A.4}$$

The general matrix representation contains six unknowns corresponding to six degrees of freedom. As a consequence, at least three point correspondences are required to compute this transformation.

The geometric effect of the affine transformation can be viewed as a composition of two fundamental transformations, namely rotations and non-isotropic scalings:

$$A = R(\theta)R(-\phi)DR(\phi) \tag{A.5}$$

[1] the scaling factor is equal for all directions

[2] this means that matrix A is invertible

This follows directly from the Singular Value Decomposition (SVD) if we write:

$$A = UDV^T = (UV^T)(VDV^T) = R(\theta)R(-\phi)DR(\phi)$$
where
$$D = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

The novelty with respect to a similarity transformation is represented by the non-isotropic scaling oriented at a particular angle. Due to this aspect, length ratios and angles between lines are not invariant under an affine transformation. Nonetheless, this transformation preserves parallel lines and length ratio of parallel lines, as well as the ratio of areas.

Projective transformations

The projective transformation is the most general of all 2D transformations. We represent this transformation using a non-singular 3×3 matrix:

$$\begin{pmatrix} x'_1 \\ x'_2 \\ x'_3 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} A & \mathbf{t} \\ \mathbf{v}^T & v \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = H_p\mathbf{x} \tag{A.6}$$

Multiplying the transformation matrix H by any non-zero factor does not change the projective transformation. Since only the ratio between elements is significant, we can say that H is a homogeneous matrix, by analogy with the homogeneous vector representation. Consequently, matrix H has 8 degrees of freedom, so at least four point correspondences are required to compute this transformation.

The generality and invariance properties of a transformation are usually complementary concepts. High generality means poor invariance properties and vice-versa. Thus, only a few properties such as concurrecy, collinearity or order of contact are preserved under a projective transformation.

Using the hierarchy of transformations discussed so far, any projective transformation can be decomposed as:

$$H = H_s H_a H_p = \begin{pmatrix} sR & \mathbf{t} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} K & \mathbf{0} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} I & \mathbf{0} \\ \mathbf{v}^T & v \end{pmatrix} = \begin{pmatrix} A & \mathbf{t} \\ \mathbf{v}^T & v \end{pmatrix} \tag{A.7}$$

Appendix B

Least-squares Minimization

B.1 Inhomogeneous equations

Consider a system $Ax = b$, with m linearly independent equations and n unknowns. In general, we can have the following cases:

- if $m < n$ there are more unknowns than equations, so the solution is an $(n - m)$-dimensional subspace of $\mathbb{R}^n$
- if $m = n$ there will be a unique solution (if A is invertible)
- if $m > n$ then there are more equations than unknowns and in general, the system will not have a solution.

The first two cases are trivial, and are usually solved using Gaussian elimination. However, many computer vision algorithms produce systems of equations that belong to the third case. Therefore, although an exact solution it is not possible, it is important to search for at least a good approximation. Formally, we aim to find $\tilde{x}$ such that $||Ax - b||$ is minimized, which is known as the least-squares solution.

Let $A = UDV^T$ be the Singular Value Decomposition of A. Then, the original system can be re-written as $||A\tilde{x} - b|| = ||UDV^T\tilde{x} - b||$. Recall that U and V are orthogonal matrices, and any multiplication with an orthogonal matrix preserves the norm. As a consequence, we can instead minimize $||UDV^T\tilde{x} - b|| = ||DV^T\tilde{x} - U^Tb||$. Call $y = V^T\tilde{x}$ and $b' = U^Tb$. Thus, we obtain $||Dy - b'||$, where D is an $m \times n$ matrix with diagonal elements sorted in descending order.

Clearly, the closest approximation of b' is the vector $(b'_1, b'_2, ...b'_n, 0, ..., 0)^T$. $\tilde{y}$ can then be found by setting $\tilde{y}_i = b'_i/d_i$, for i = $\overline{1,n}$, while $\tilde{x} = V\tilde{y}$.

B.2 Homogeneous equations

If b is a null vector, then the system of the form $Ax = 0$ is called homogeneous. Note that it is always possible to convert an inhomogeneous system into a homogeneous one and vice-versa. An interesting observation is that the scale of x is not important, because if x is a solution to this set of equations, then so is kx for any scalar k. This includes the trivial solution $x = 0$ which is of little interest to us. Therefore, in practice we constrain x such that $||x|| = 1$. As in the previous section, if the system is over-determined and the equations are linearly independent,

then it is not possible to find an exact solution. Alternatively, we search for $\tilde{x}$ that minimizes $||A\tilde{x}||$ subject to $||\tilde{x}|| = 1$.

The solution is again found using SVD. If $A = UDV^T$, then $\min||A\tilde{x}|| = \min||UDV^T\tilde{x}||$. U and U are orthogonal matrices, so $||UDV^T\tilde{x}|| = ||DV^T\tilde{x}||$ and $||\tilde{x}|| = ||V^T\tilde{x}||$. Let $y = V^T\tilde{x}$ and the problem is now to minimize $||Dy||$ subject to $||y|| = 1$. Recall that D is a diagonal matrix with its diagonal entries in descending order. Therefore, in order to minimize $||Dy||$, we choose $y = (0, 0, ..., 0, 1)^T$. Consequently, $\tilde{x}$ can be obtained as the last column of V, since $\tilde{x} = Vy$.

Bibliography

[1] IEEE Standard 802.3-2008.

[2] C. Baillard, 0. Dissard, and H. Maitre. Segmentation of urban scenes from aerial stereo imagery. In *In Proc. ICPR*, pages 1405–1407, 1998.

[3] C. Baillard and A. Zisserman. Automatic reconstruction of piecewise planar models from multiple views. In *In Proc. IEEE Conference on Computer Vision and Pattern Recognition*, 1999.

[4] C. Barber, D. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. In *ACM Transactions on Mathematical Software vol 22*, pages 469–483, 1996.

[5] H. Bay, A. Ess, T. Tuytelaars, and L. van Gool. Surf: Speeded up robust features. In *Computer Vision and Image Understanding*, pages 346–359, 2008.

[6] M. Berthod, L. Gabet, G. Giraudon, and J. L. Lotti. Highresolution stereo for the detection of buildings. In *In Automatic Extraction of Man-Made Objects from Aerial and Space Images*, pages 135–144, 1995.

[7] A. Cherian, V. Morellas, and N. Papanikolopoulos. Accurate 3d ground plane estimation from a single image. In *Proceedings of the 2009 IEEE international conference on Robotics and Automation*, pages 519–525, Piscataway, NJ, USA, 2009. IEEE Press.

[8] O. Chum, J. Matas, and J. Kittler. Locally optimized ransac. In *DAGM-Symposium'03*, pages 236–243, 2003.

[9] N. Cornelis, B. Leibe, and L. Gool. 3d urban scene modeling integrating recognition and reconstruction. In *International Journal of Computer Vision*, pages 121–141, 2008.

[10] M. DeLeon. A study of sufficient conditions for hamiltonian cycles. In *Undergraduate Math Journal*, 2000.

[11] Google Earth. http://www.google.com/earth/index.html.

[12] Microsoft Virtual Earth. http://www.bing.com/maps/.

[13] I. Esteban, J. Dijk, and F. Groen. Automatic 3d modelling of the urban landscape. In *International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, 2010.

[14] M. Fischler and R. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. In *Communications of the ACM*, pages 381–395, 1981.

[15] C. Frueh, S. Jain, and A. Zakhor. Data processing algorithms for generating textured 3d building facade meshes from laser scans and camera images. In *International Journal of Computer Vision*, pages 159–184, 2005.

[16] S. Girard, P. GuCrin, H. Maitre, and M. Roux. Building detection from high resolution colour images. In *In Int. symp. on Remote Sensing*, 1998.

[17] N. Haala and C. brenner. Fast production of virtual reality city models. In *International Archives of Photogrammetry and Remote Sensing*, pages 77–84, 1998.

[18] R. Hartley and A. Zisserman. Multiple view geometry in computer vision. page 136, 2006.

[19] D. Knuth. The art of computer programming vol 1, 3rd ed. 1997.

[20] David Lowe. Distinctive image features from scale-invariant keypoints. In *International Journal of Computer Vision*, pages 91–110, 2004.

[21] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *In Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, 1967.

[22] Professional Surveyor Magazine. http://www.profsurv.com/magazine/article.aspx?i=2110.

[23] P. Mordohai, J. Frahm, A. Akbarzadeh, B. Clipp, C. Engels, D. Gallup, P. Merrell, C. Salmi, S. Sinha, B. Talton, L. Wang, Q. Yang, H. Stewenius, H. Towles, G. Welch, R. Yang, M. Pollefeys, and D. Nister. Real-time video-based reconstruction of urban environments. In *In ISPRS*, 2007.

[24] US Department of Defense World Geodetic System standard.

[25] OpenStreetMap. http://www.openstreetmap.org/.

[26] M. Pollefeys, D. Nister, J. Frahm, A. Akbarzadeh, P. Mordohai, B. Clipp, C. Engels, D. Gallup, S Kim, P. Merrell, C. Salmi, S. Sinha, B. Talton, L. Wang, Q. Yang, H. Stewenius, R. Yang, G. Welch, and H. Towles. Detailed real-time urban 3d reconstruction from video. In *International Journal of Computer Vision*, 2008.

[27] L. Spinello, A. Macho, R. Triebel, and R. Siegwart. Detecting pedestrians at very small scales. In *IROS'09: Proceedings of the 2009 IEEE/RSJ international conference on Intelligent robots and systems*, pages 4313–4318, Piscataway, NJ, USA, 2009. IEEE Press.

[28] Google street view. http://maps.google.com.au/help/maps/streetview/.

[29] I. Sutherland. A method for solving arbitrary-wall mazes by computer. In *IEEE Transactions on Computers*, 1969.

[30] FIT3D toolbox. http://www.fit3d.info/.

[31] W. von Hansen. Automatic detection of zenith direction in 3d point clouds of built-up areas. In *PIA07*, page 93, 2007.

[32] G. Vosselman and S. Dijkman. 3d building model reconstruction from point clouds and ground plans. In *Proceedings of the ISPRS Workshop*, 2001.

[33] Z. Zhang. On the epipolar geometry between two images with lens distortion. In *Proceedings of ICPR*, 1996.

[34] J. Zhou and B. Li. Robust ground plane detection with normalized homography in monocular sequences from a robot platform. In *IEEE International Conference on Image Processing*, pages 3017 – 3020, 2006.